HyperSack: Distributed Hyperparameter Optimization for Deep Learning using Resource-Aware Scheduling on Heterogeneous GPU Systems

Nawras Alnaasan, Bharath Ramesh, Jinghan Yao, Aamir Shafi, Hari Subramoni, and Dhabaleswar K Panda
Department of Computer Science and Engineering,
The Ohio State University, Columbus, Ohio, USA,
{alnaasan.1, ramesh.113, yao.877, shafi.16, subramoni.1}@osu.edu, panda@cse.ohio-state.edu

***Abstract*—Hyperparameter Optimization (HPO) can unlock the full potential of Deep Learning (DL) models; however, it is considered one of the most compute-intensive tasks in the DL domain due to multi-dimensional search spaces and complex neural network architectures. A common method for accelerating HPO workloads is parallelizing training jobs on multiple computing devices, such as modern GPUs in High-Performance Computing (HPC) environments. Nonetheless, existing HPO parallelization strategies underutilize powerful GPU devices, like the NVIDIA A100 and H100, especially for training lightweight Deep Neural Networks. Resource-sharing mechanisms can improve GPU utilization; nonetheless, naïve adaptations for HPO workloads lead to poor performance. Therefore, we propose HyperSack—a distributed HPO framework for dynamic and resource-aware scheduling on heterogeneous GPU-based HPC systems with resource elasticity and fault tolerance. HyperSack reduces the execution time of HPO workloads by orchestrating the placement of DL training jobs on GPU devices with different computational capabilities. It supports different hardware architectures, HPO workloads, and scheduling policies. Our evaluations on vision and language models HPO workloads show up to 2.8x performance improvement in execution time on A100 GPUs, 4.0x on H100 GPUs, and 3.9x on a combination of 12 A100 and 4 H100 GPUs using HyperSack over standard HPO parallelization methods.**

***Index Terms*—Hyperparameter Optimization, Deep Neural Networks, Graphics Processing Units**

I. Introduction

The quality of solutions to which Machine Learning (ML) and Deep Learning (DL) models converge depends on the initial configuration of the training hyperparameters [1]. Manual tuning is a widely used strategy to optimize models where the hyperparameter values are selected based on a trial-and-error approach by an ML/DL expert. On the other hand, automated Hyperparameter Optimization (HPO) strategies systematically navigate search spaces to tune the training configurations. HPO reduces the reliance on expert intuition and leads to the training of more accurate DL models [2]. However, HPO comes at a significant cost—it is one of the most compute-intensive tasks in the ML/DL domains.

Today, Deep Neural Networks (DNNs) are considered state-of-the-art for solving complex problems in many domains, such as Computer Vision (CV) and Natural Language Processing (NLP). While there is a trend toward training larger and more powerful DNNs with billions of parameters, lightweight DNNs remain crucial [3], especially for performing low-latency and power-efficient inference in resource-constrained environments. Nonetheless, performing HPO on such models still requires significant computational resources to run in a reasonable time as target accuracies may only be reached after training several hundreds of models [4].

Graphics Processing Units (GPUs) have undergone remarkable advancements in recent years to address the ever-growing computational demands of DL applications. DL parallelization strategies have been proposed to leverage distributed computing environments such as High-Performance Computing (HPC) and Cloud systems. However, current HPO parallelization techniques underutilize powerful GPUs such as the NVIDIA A100s and H100s, especially for optimizing lightweight DNNs, leading to idle resources. While hardware-specific features such as NVIDIA Multi-Process Service (MPS) [5] can be used to share GPU resources, the main challenge lies in deciding the placement of HPO jobs on GPUs. **We refer to this challenge as the GPU Assignment Problem (*GAP*)**.

Multiple factors must be considered to efficiently solve *GAP*, including the memory and compute requirements for DNN training, GPU architecture, number of GPUs, available CPU cores, GPU memory, and GPU compute. Naïve job scheduling policies may result in poor performance due to oversubscription or underutilization of resources. Therefore, we introduce *HyperSack*—a distributed HPO framework designed for dynamic and resource-aware scheduling on heterogeneous GPU-based HPC systems. *HyperSack* addresses *GAP* by mapping it to the Multiple Knapsack problem *MKP* and proposing efficent scheduling policies. Furthermore, *HyperSack* supports multiple features such as preliminary profiling of GPU utilization leveraging job history, resource elasticity allowing the addition or removal of GPUs, and fault tolerance to monitor resources and reschedule jobs.

*This research is supported in part by NSF grants #1818253, #1854828, #2007991, #2018627, #2112606, #2311830, #2312927, #2415201, and XRAC grant #NCR-130002.

2640-0316/24/$31.00 ©2024 IEEE
DOI 10.1109/HiPC62374.2024.00019

A. Motivation

We conduct preliminary evaluations on Convolutional Neural Networks (CNNs): MoibleNet [6], ShuffleNet [7], ResNet-18 [8], and ResNet-34; Vision Transformer models: ViT-Tiny [9] and DeiT-Tiny; Transformer encoder-based language model BERT-Tiny [10]; and Transformer decoder-based language models Pythia-14M [11], and Pythia-31M. This evaluation measures the GPU utilization during training. Model size ranges between 2.3M and 31M parameters. We use the NVIDIA NVML tool [12] to sample the average GPU utilization throughout multiple training iterations.

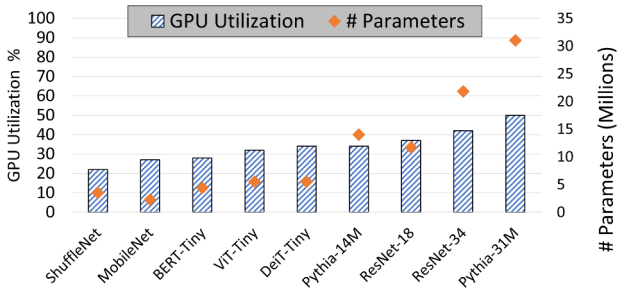

Fig. 1. GPU utilization of the NVIDIA A100 training on different vision and language models.

Figure 1 shows the GPU utilization and number of parameters for the different vision models trained on CIFAR-10 [13] and language models trained on Stanford's IMDB dataset [14]. These experiments are each conducted on a single A100 GPU. We observe that utilization is low for all of these models, ranging between 22% for ShuffleNet and 50% for Pythia-31m. The GPU utilization depends on several factors, including the model, batch, and sample sizes.

Furthermore, many lightweight DL models are bottlenecked by dataloading. This leads to GPU idling bubbles in the training pipeline until the next batch of data is available with I/O prefetching. Figure 2 is a Nsight Systems [15] profile that shows the GPU utilization (in blue) and CPU utilization (in black) over several training iterations of the ResNet18 model on CIFAR-10 exhibiting frequent GPU bubbles due to the dataloading bottleneck.

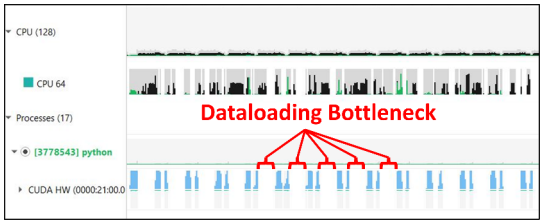

Fig. 2. Nsight Systems profile of ResNet18 training showing a dataloading bottleneck.

The underutilization shown in figures 1 and 2 suggest that multiple compute cycles and CUDA cores may be available on the GPU but not being fully utilized. Therefore, *the primary motivation of this paper is to design advanced parallelization strategies to increase the occupancy of GPUs and thus reduce the makespan*[1] *of HPO workloads.*

B. Problem Overview

To provide an efficient solution for the GPU Assignment Problem (*GAP*), we analyze the CPU and GPU requirements for training different DNNs in an HPO workload. The problem setting for *GAP* is defined by the number of jobs in the search space, compute/memory requirement for each job, number of available nodes, CPU cores per node, GPUs per node, and GPU architectures. Figure 3 shows an example of running an HPO workload consisting of 16 DNN training jobs on two nodes with two GPUs per node. Each job is associated with a job index and weight representing the percentage of GPU utilization for the respective GPU architecture. The figure shows a scenario where nine jobs are naïvely assigned to the four GPUs, resulting in an average of 72.5% GPU occupancy. Pending jobs are then assigned as previous jobs finish execution and resources are available. However, the solution presented in the figure is non-optimal, as the utilization can be further improved by fitting more jobs in a different arrangement. We address this problem by proposing resource-aware scheduling policies that dynamically maximize GPU occupancy for different HPO workloads and hardware architectures.

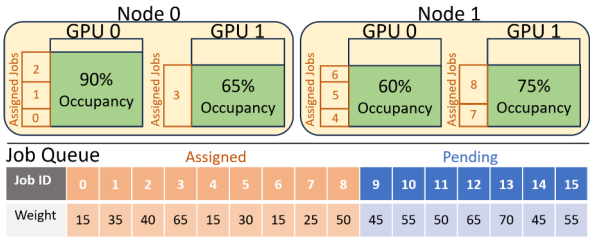

Fig. 3. An example of the GPU Assignment Problem (*GAP*) with two nodes, two GPUs per node, and an HPO workload of 16 DNN training jobs.

C. Problem Statements

The broad problems that we address in this paper are as follows:

1) How can we schedule training jobs of HPO workloads on modern **heterogeneous GPU clusters** to **improve the GPU utilization** and **reduce the HPO makespan**?
2) What are the **hardware constraints** for scheduling HPO workloads and how can we design **scheduling policies that are aware of such constraints** for job assignment?
3) How can we design an **HPO workflow/framework** that is **adaptable** to handle different scheduling policies, search spaces, DNN architectures including both **vision and language models**, and hardware architectures?
4) How can we analyze job requirements, **monitor job status and hardware state**, support efficient **GPU space sharing**, and provide **resource elasticity and fault tolerance**?

[1]In the context of this paper, makespan refers to the time taken to execute all HPO jobs in the search space until completion.

D. Contributions

Table I compares the key features of related studies on GPU scheduling for DNN training or inference workloads with the proposed design. Our extensive literature survey shows that distributing HPO workloads, especially for lightweight DNNs, is a major challenge in the DL domain that's yet to be addressed. We expand on this comparison later in the related work section VI.

The key contributions of this paper are as follows:

1) Design and implement *HyperSack*—a distributed HPO framework designed for dynamic and **resource-aware scheduling on heterogeneous GPU-based HPC systems with resource elasticity and fault tolerance.**
2) Formally define the problem constraints for the GPU Assignment Problem (*GAP*) in terms of **GPU compute, GPU memory, and CPU compute.**
3) Propose **scheduling policies that maximize the utilization of GPUs** for HPO workloads by mapping (*GAP*) to the Multiple Knapsack problem (*MKP*).
4) Devise a profiling scheme to **sample utilization metrics and leverage prior job history** for job scheduling.
5) Conduct comprehensive evaluations on different **vision and language models** HPO workloads, showing up to **2.8x performance improvement** in execution time on **4 A100 GPUs**, **4x on 4 H100 GPUs**, and **3.9x on a combination of 12 A100 and 4 H100 GPUs** using *HyperSack* compared to traditional HPO parallelization methods.

The rest of the paper is organized as follows: Section II covers the background on NVIDIA MPS. Section III provides a formal definition of the GPU Assignment Problem (*GAP*). Section IV maps *GAP* to the Multiple Knapsack Problem (*MKP*), proposes four resource-aware scheduling policies, and describes the design and implementation of *HyperSack*. Section V includes a comprehensive evaluation and analysis of the proposed designs. Section VI reviews and compares related works. Finally, we conclude the paper in Section VII.

II. Background

A. Hyperparameter Optimization (HPO)

1) Exhaustive Search Methods: Exhaustive search HPO explores search spaces by systematically evaluating hyperparameter configurations to optimize a black-box function. Examples of exhaustive search methods include the Grid Search and Random Search algorithms. These algorithms often reach a better optimum compared to iterative search [21]; however, they require running more trails. Therefore, exhaustive search methods are more suitable for cheap black-box functions such as lightweight DNNs.

2) Iterative Search Methods: Iterative search HPO algorithms perform consecutive experiments with different hyperparameter settings to optimize an objective function. Examples of iterative search methods include the Bayesian Optimization and Aging Evolution algorithms. Iterative search may require fewer trials than exhaustive search, making it more suitable for optimizing expensive objective functions [21] such as large DNNs.

B. Lightweight Deep Neural Networks

Lightweight Deep Neural Networks (DNNs) have become increasingly essential due to their ability to perform inferencing efficiently in resource-constrained environments unlike larger and deeper models [3]. Nonetheless, HPO for lightweight models remains a resource-demanding task due to the training of many models before reaching a convergence criterion. In this paper, we focus on improving GPU utilization specifically for lightweight DNNs. We explore a wide array of state-of-the-art models including 1) Decoder-based language models such as the GPT-like Pythia [11] models, 2) Encoder-based language models such as BERT, 3) Vision Transformers such as ViT [9] and DeiT, and 4) Convolutional Neural Networks (CNNs) such as ResNet [8].

C. NVIDIA Multi-Process Service (MPS)

NVIDIA Multi-Process Service (MPS) [5] is a feature provided by the CUDA API designed to improve the performance of multi-process GPU applications by enabling concurrent operations via a client-server paradigm. The MPS infrastructure assigns each CUDA process to an individual client context, each operating within a dedicated and secure GPU address space. This feature is particularly effective when dealing with many small-scale tasks that can be executed simultaneously using space-sharing to partition resources logically. A key attribute of MPS is the dynamic allocation of GPU resources among concurrent client processes. Figure 4 shows a scenario where four clients (processes) are launched via the MPS server. Clients may utilize resources differently; therefore, available CUDA cores and memory are allocated on a need basis. Furthermore, users can set an upper limit on the resource allocation per client by configuring the MPS server.

In addition to MPS, there are other techniques that facilitate spatial sharing of multiple processes on GPUs, such as Multi-Instance GPU (MIG) [22], CUDA Streams [23], and Hyper-Q [24]. In the context of this paper, we choose to focus on MPS, given its dynamic resource allocation feature.

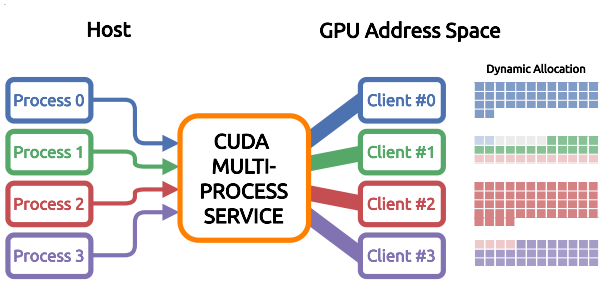

Fig. 4. Workflow of NVIDIA Multi-Process Service (MPS).

TABLE I
FEATURE COMPARISON BETWEEN HYPERSACK AND EXISTING DL JOBS SCHEDULING FRAMEWORKS

Existing and Proposed Studies on DL Jobs Scheduling	GPU Spatial Sharing	Extensible Scheduling Policies	Hardware Heterogeneity Awareness	Profiling/ estimating DL workloads	Utilization of Job History	Support for DNN Training	Support for Hyperparameter Optimization	Resource Elasticity and Fault Tolerance
Salus [16]	×	×	×	×	×	✔	✔	×
Gpulet [17]	✔	✔	×	×	×	×	×	×
GSLICE [18]	✔	×	×	×	×	×	×	×
Gandiva [19]	✔	✔	×	✔	×	✔	×	×
Gavel [20]	✔	✔	✔	✔	×	✔	×	×
HyperSack (Proposed Design)	✔	✔	✔	✔	✔	✔	✔	✔

III. CONSTRAINTS AND FORMAL DEFINITION OF THE GPU ASSIGNMENT PROBLEM (GAP)

A. Problem Constraints

An HPO workload consists of many DNN training jobs. For efficient job assignment, a scheduling policy should consider the number of compute nodes, the number of CPU cores available per node, the number of GPUs per node, the available compute capacity on a GPU device, the number of jobs, the GPU memory and computing requirements needed for each DNN training job, which can be estimated using profiling tools.

Figure 1 shows the underutilization of CUDA cores on the GPU at a given time instance (spatial underutilization), and figure 2 indicates that resources may become completely idle over different training phases (temporal underutilization). Therefore, we consider both spatial and temporal GPU sharing to improve the utilization. Spatial sharing can be facilitated via NVIDIA MPS to increase the number of used CUDA cores at any given time instance. On the other hand, temporal sharing can be achieved by oversubscribing the GPU resources and context-switching to minimize idle resources over time. Throughout the paper, we use the term "oversubscription ratio" to refer to a tunable value per GPU architecture used to scale the available GPU capacity and perform efficient context-switching. We analyze the impact of the oversubscription ratio later in section V-D.

B. Formal Problem Definition

In this subsection we formally formulate the GPU Assignment Problem (*GAP*) for HPO workloads. We first define the following parameters:

- Number of compute nodes available denoted by N and node index by $n \in \{0, 1, \cdots, N\}$
- Number of GPUs per node denoted by M and GPU index by $m \in \{0, 1, \cdots, M\}$
- Number of jobs to be scheduled, denoted by P and job index by $p \in \{0, 1, \cdots, P\}$
- Compute capacity for node n and GPU m, denoted by C_{nm}
- Oversubscription ratio for GPU m on node n denoted by r_{nm} and used for scaling the compute capacity C_{nm}
- Compute requirement needed by job p on the target GPU architecture, denoted by c_p
- GPU memory for node n and GPU m, denoted by D_{nm}
- Memory required for job p denoted by d_p
- #CPU cores available for node n, denoted by Q_n
- #CPU cores required for job p denoted by q_p
- Let $X(n, m, p)$ be a binary variable that takes the value 1 if job p is assigned to node n and GPU m, and 0 otherwise.

Given the previous parameters, the problem constraints can be defined as follows:

$$\sum_{n}^{N}\sum_{m}^{M} X(n,m,p) \leq 1, \forall p \tag{1}$$

$$\sum_{p}^{P} c_p \cdot X(n,m,p) \leq r_{nm} \cdot C_{nm}, \forall n, m \tag{2}$$

$$\sum_{p}^{P} d_p \cdot X(n,m,p) \leq D_{nm}, \forall n, m \tag{3}$$

$$\sum_{p}^{P} q_p \cdot X(n,m,p) \leq Q_n, \forall n, m \tag{4}$$

Equation 1 denotes that each job can be assigned to at most one GPU. Equation 2 denotes that the summation of required capacities for all assigned jobs to a GPU should not exceed its compute capacity scaled by the oversubscription ratio. Similarly, equation 3 states that the summation of memory requirements for all assigned jobs to a GPU should not exceed its available memory. Finally, equation 4 states that the sum over all jobs for the number of CPU cores should not exceed the total core count on a compute node.

In the context of the problem parameters and constraints, the objective of *GAP* can be defined as follows:

$$\max_{X}\{\sum_{p}^{P} c_p \cdot X(n,m,p)\}, \forall n, m \tag{5}$$

Equation 5 shows the objective of *GAP*, which aims to maximize the overall assigned compute to all GPUs and, in turn, minimize the makespan for finishing the HPO workloads. This objective is considered a single scheduling decision, as jobs may not be all initially scheduled due to resource constraints. Several scheduling decisions are iteratively made throughout the makespan of the HPO workload as jobs finish execution and computing resources become available.

IV. Proposed Design and Implementation

In this section, we first map the GPU Assignment Problem (*GAP*) to the Multiple Knapsack problem (*MKP*). We then present the proposed algorithms/scheduling policies to address *GAP*. Finally, we introduce the architecture and workflow of the *HyperSack* framework and its different components.

A. Mapping GAP to the Multiple Knapsack Problem MKP

The Multiple Knapsack problem (*MKP*) is a generalization of the Knapsack Problem (*KP*). The objective of *KP* is to maximize the total value of items packed in a knapsack, subject to its weight capacity. For *MKP*, the problem is extended by using N knapsacks, where the objective is to maximize the value of items packed in all the knapsacks given their weight capacities. The GPU Assignment Problem (*GAP*) for HPO workloads can be directly mapped to the 0-1 *MKP* as follows:

- Available GPUs $\{0, 1, \cdots, M\}$ for each node n are mapped to the set of knapsacks.
- Compute capacity C_{nm} for each GPU m on node n is mapped to the weight capacities of the knapsacks.
- The set of HPO jobs $\{0, 1, \cdots, P\}$ to execute on the GPUs is mapped to the set of items that need to be packed in the knapsacks.
- Compute requirement c_p for each job p is mapped to the weight of each item.
- The value of running a job is decided by its percentage of utilization on the GPU.
- Additionally, all constraints listed in section III must be satisfied for solving *GAP*.

Different solutions to *GAP* can be considered as scheduling policies for the HPO workloads on the GPUs. The HPO workload, however, may consist of many jobs that cannot all be initially assigned to the available GPUs. As jobs finish execution and/or more computing resources become available, a new scheduling decision must be made. This requires solving *GAP* iteratively multiple times throughout the makespan of the HPO workload. However, it should be noted that *MKP* is an NP-Hard problem. As a result, searching for the optimal solution using complex methods is infeasible for *GAP* because it introduces a significant scheduling overhead. In the following subsection, we explore efficient approximation algorithms with minimal overhead, taking into consideration the aforementioned problem constraints.

B. Proposed Scheduling Policies for GAP

To ensure dynamic placement of HPO jobs on the GPUs, scheduling decisions must be taken at several points during the makespan of the HPO workload. Since *GAP* is an NP-Hard problem, we look at fast approximation algorithms with minimal scheduling overhead. In this subsection, we propose using four different scheduling policies: 1) First-fit (FF), 2) First-fit-decreasing (FFD), 3) Worst-fit (WF), and 4) Worst-fit-decreasing (WFD). We note that *HyperSack* is not limited to these scheduling policies as it provides hooks for implementing more scheduling schemes. Later in section V-B, we evaluate the performance of the proposed policies and the quality of solutions they provide.

Algorithm 1: FF and FFD Scheduling Policies

```
Input: Cluster—Available nodes and GPUs
       Grid—Jobs in the search space
Output: JobQueue—The list of jobs to run and their
        assigned devices
if Decreasing then Grid.sort(ExpectedRunTime,
 reverse) ;
for Job p in Grid do
    for Node n in Cluster do
        for GPU m in n do
            constraint1 ← r_nm · C_nm ≥ c_p
            constraint2 ← D_nm ≥ d_p
            constraint3 ← Q_n ≥ q_p
            if constraint1 and constraint2 and
             constraint3 then assignedGPU ← (n,m)
            break ;
        end
        if assignedGPU then break ;
    end
    JobQueue.push(p,n,m)
end
```

The four proposed policies are explained in detail below:

1) First-fit (FF): The First-fit (FF) policy is the simplest approach to solve *GAP*. To schedule a job p that requires capacity c_p, we iterate over each GPU m on node n to check the remaining GPU compute capacity C_{nm}, free memory D_{nm}, and available CPU cores on the node Q_n. We schedule the job on the first GPU that satisfies the problem constraints. This process is repeated for all jobs $p \in \{0, 1, \cdots, P\}$ until resources are fully occupied.

Algorithm 1 shows the First-fit policy. The algorithm takes as input the $Cluster$ list, which contains the system information and the $Grid$ list, which contains information about the jobs. Next, we iterate over all nodes and GPUs for each jobs to find the first GPU that satisfies the problem constraints. Finally, the algorithm returns the job queue, which contains the jobs to run along with their assigned nodes/GPUs.

2) First-fit-decreasing (FFD): A drawback of the FF policy is that it makes scheduling decisions without considering the expected run time of jobs. In some scenarios, long-running jobs are scheduled near the end of the HPO workload. This wastes the opportunity to overlap long-running jobs with shorter jobs and results in idle resources.

To address this limitation, the First-fit-Decreasing (FFD) policy sorts all jobs in the grid in decreasing order with respect to the expected completion time of the individual jobs, which is estimated by the profiler. The profiling mechanism is explained in section IV-C1. Algorithm 1 includes a condition that checks if the decreasing policy is selected and sorts the grid with respect to the expected completion time in decreasing order. The rest of the algorithm is carried out similarly to

the FF policy. Later in section V, we show that the ordering property of FFD leads to better performance compared to FF.

3) Worst-fit (WF): A significant drawback of the FF and FFD policies is the workload imbalance. In some scenarios, the jobs may be assigned to the first couple of available GPUs, which underutilizes the rest of the GPUs on the system. To address this issue, the Worst-Fit (WF) policy gives assignment priority to the GPUs with the most compute capacity C_{nm} available. This ensures a fair distribution of the HPO workload across all GPUs.

Algorithm 2 shows the Worst-fit policy. The inputs and outputs are similar to the FF policy. In this algorithm, we iterate over all nodes and GPUs for each job, identifying the GPUs that satisfy the problem constraints. Then, we assign the job to the GPU with the most capacity available. We later show in section V that workload balancing using the WF policy leads to better performance compared to the FF policy.

4) Worst-fit-decreasing (WFD): Worst-Fit-Decreasing (WFD) is a policy that ensures 1) maximum overlap between long and short-running jobs and 2) workload balancing across all available GPUs. Algorithm 2 includes a condition that checks if the decreasing policy is selected. The grid list is sorted in decreasing order with respect to the expected completion time of jobs. The rest of the algorithm is carried out similarly to the WF policy. Later in section V, we show that combining both the decreasing order and workload balancing properties in WFD results in a better performance compared to the rest of the scheduling policies.

The worst-case complexity for all four policies is subject to the number of jobs and the number of GPUs on the system, which is expressed as $O(P \cdot N \cdot M)$.

Algorithm 2: WF and WFD Scheduling Policies

```
Input: Cluster—Available nodes and GPUs
       Grid—Jobs in the search space
Output: JobQueue—The list of jobs to run and their
        assigned devices
if Decreasing then Grid.sort(ExpectedRunTime,
 reverse) ;
for Job p in Grid do
    WorstFit← 0
    for Node n in Cluster do
        for GPU m in n do
            constraint1 ← r_nm · C_nm ≥ c_p
            constraint2 ← D_nm ≥ d_p
            constraint3 ← Q_n ≥ q_p
            constraint4 ← C_nm ≥ WorstFit
            if constraint1 and constraint2 and
             constraint3 and constraint4 then
             WorstFit ← C_nm
            assignedGPU ← (n,m) ;
        end
    end
    JobQueue.push(p,assignedGPU)
end
```

C. Architecture and Workflow of the HyperSack Framework

In this subsection, we discuss the design and implementation of the proposed *HyperSack* framework, which consists of multiple software components ensuring modularity and extensibility. Figure 5 shows the layered architectural overview of *HyperSack*. The top layer represents the HPO workload, which is defined by the search space. The next layer shows the *HyperSack* framework and its various components, including the Orchestrator, Policy, Grid, Sampler, and State components. *HyperSack* interacts with both the Deep Learning frameworks and the hardware toolkits/libraries directly to ensure accurate hardware sampling and efficient execution of multiple workloads on the GPU. The final layer shows the modern GPU-based HPC systems with multiple nodes, each containing many/multi-core CPUs and GPUs.

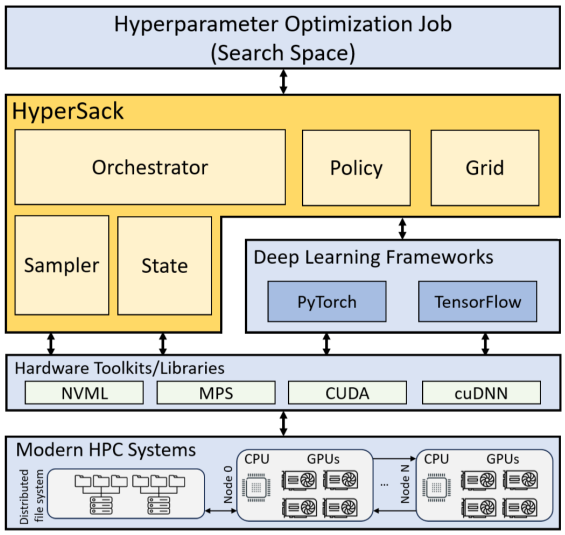

Fig. 5. Layered Architecture of the proposed *HyperSack* framework.

1) Sampler: The Sampler component collects hardware and DNN training metrics. Three factors mainly determine the GPU utilization: model size, optimizer, and batch size. The metrics collected by the Sampler include average GPU utilization, peak GPU memory, training time per iteration, and pre-processing overhead. These metrics are collected for each GPU architecture to ensure heterogeneity awareness. Furthermore, the Sampler infers an epoch's expected training time and the job's overall training time from the training time per iteration. Hardware metrics are collected using the NVIDIA NVML library [12]. In our experience, running ten training iterations is sufficient to capture an accurate metrics. The Sampler excludes the first training iteration as it exhibit low GPU utilization due to setting up the training pipeline. Figure 6 shows the workflow of the Sampler component. It first identifies all unique models in the search space. Next, it searches the logs of previous HPO jobs to find if models were run previously. If one or more models are not found in the history, the Sampler sends the new models to the GPUs for profiling. Finally, the Sampler generates new logs for the current HPO workload. The overhead of the Sampler is

negligible as metrics are either found in history or the Sampler needs to run for ten iterations only. To put this into perspective, each full HPO job may run for tens of epochs, and each epoch consists of thousands to millions training iterations depending on the batch and dataset sizes. Thus, the overhead of running ten iterations can be ignored.

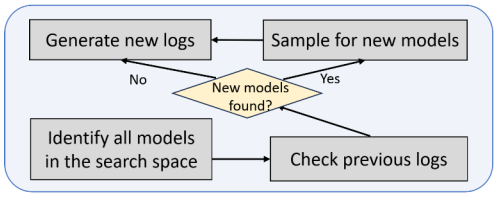

Fig. 6. Workflow of the Sampler component in the *HyperSack* framework.

2) Grid: The Grid component is a multi-dimensional matrix that represents the search space. The dimensionality of the Grid is determined by the number of hyperparameters to be optimized. This component maintains a record of all the hyperparameter configurations for the HPO jobs and their completion status. Figure 7 shows an example of a 2-dimensional Grid. Each axis corresponds to a different hyperparameter. The circles at the intersections represent the HPO jobs. The color of the circle indicates the current state of the job. 1) Pending (white): the job is not yet scheduled; 2) In progress (yellow): the job is currently running on a GPU; 3) Complete (green): the job finished execution earlier; 4) Failed (red): an error was detected and job rescheduling is needed.

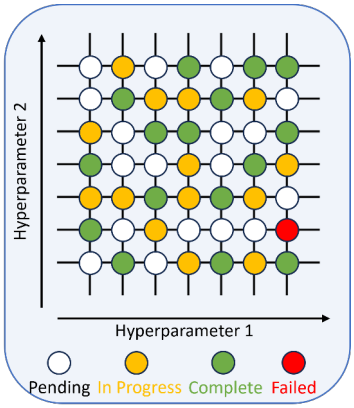

Fig. 7. Overview of a two-dimensional grid generated from the search space by the Grid component of the *HyperSack* framework.

3) State: The State component is responsible for monitoring the status of the hardware devices and ensuring resource elasticity and fault tolerance. Figure 8 shows a depiction of the State component. For each node allocated, it monitors the occupancy of the GPUs, detects the completion of jobs, and runs a dedicated thread in the background to detect the addition or failure of hardware devices. If a failure is detected, the job is flagged for rescheduling.

4) Policy: The Policy component is responsible for generating and updating the job queue, given the status of the Grid and State components. Several scheduling policies

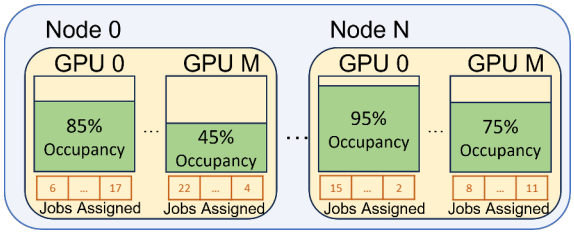

Fig. 8. Representation of hardware occupancy and job completion monitoring by the State component in the *HyperSack* framework.

are implemented in *HyperSack*: First-Fit (HS-FF), First-Fit-Decreasing (HS-FFD), Worst-Fit (HS-WF), and Worst-Fit-Decreasing (HS-WFD). Refer to subsection IV-B for a detailed description of these scheduling policies.

5) Orchestrator: The Orchestrator controls the main logic of the *HyperSack* framework and coordinates between all of its components. Figure 9 shows the sequence of steps taken when a new HPO job is launched:

1) Initialize MPS servers on every GPU on all nodes. The MPS servers are dynamically started using the `nvidia-cuda-mps -control` command, specifying the user ID, and setting the `CUDA_MPS_ACTIVE_THREAD_PERCENTAGE` variable to set an upper partitioning bound for each job sharing the same GPU resources.
2) Generate job configurations from the search space using the Grid component. The status of a job is either pending, in progress, complete, or failed.
3) Collect utilization metrics using the Sampler. Metrics are collected for each GPU architecture in the allocation.
4) Generate a new job queue using the Policy component.
5) Schedule jobs in the job queue on their assigned GPUs.
6) Check the state of the hardware devices using the State component and monitor any job completions, failures, or changes in resource availability.
7) Update the job queue using the scheduling policies based on the new hardware availability.
8) If pending jobs are still in the Grid object, repeat step 5 until all jobs in the Grid are fully completed.
9) Save the training metrics (loss, accuracy, training time, etc.) for the trained jobs and finalize the MPS servers.

V. Performance Evaluation and Analysis

Our evaluations and analyses consist of the following: V-B) Analysis of the computational time and quality of solutions of the proposed scheduling policies, V-C) Evaluation of the makespan improvement of different HPO workloads using the proposed scheduling policies, V-D) analysis of the impact of oversubscription ratios, V-E) evaluation of the scaling efficiency in homogeneous systems, and V-F) evaluation of the scaling efficiency in heterogeneous systems.

A. Experimental Setup

1) Hardware Setup: We perform our evaluations on two GPU clusters. 1) A100 Power Edge XE 8545 nodes equipped with 2 AMD EPYC 7643 (Milan) processors @2.3 GHz, each

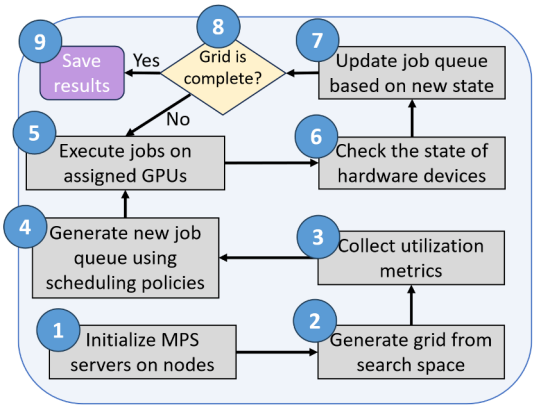

Fig. 9. Workflow of the Orchestrator component responsible for coordinating the functioning of all components of the *HyperSack* framework.

with 44 usable cores (88 in total), and 4 NVIDIA A100 GPUs per node with 80GB memory. 2) A100 and H100 partitions. A100 nodes are equipped with 3 A100 GPUs with 40GB memory and two AMD EPYC 7763 64-core processors (Milan). H100 nodes are equipped with two H100 GPUs with 80GB memory and two AMD EPYC 9454 48-core processors.

2) Software Setup: CUDA 11.7, cuDNN 8.9.2, Python 3.8.16, pyNVML 11.5, PyTorch 2.0.1, Transformers 4.30.2, and pandas 2.0.2.

3) Baseline: The baseline for all evaluations is the standard HPO scheduling scheme of one training job per GPU. In Table I, we have listed other GPU scheduling studies. Many of these packages do not support DNN training or HPO workloads. Still, we have explored the publicly available codebases of Gavel [20] and Salus [16]. Unfortunately, **these packages do not support new hardware architectures such as A100 and H100, as many parameters are hardcoded for older GPU generations such as P100 and K80.** As a result, we could not reproduce any results from other frameworks. Therefore, we use the standard HPO scheduling scheme of one training job per GPU as the baseline for all comparisons.

4) HPO Workloads: The HPO search spaces we use consist of the following vision and language models: 1) ShuffleNet [7], 2) MobileNet [6], 3), ViT-Tiny [9], 4) DeiT-Tiny, 5) ResNet-18 [8], 6) ResNet-34, 7) BERT-Tiny [10], 8) Pythia-14M [11], and 9) Pythia-34M. Vision models are trained on the CIFAR-10 [13] dataset and language models are trained on the IMDB [14] dataset. The different search spaces are defined in table II. We refer to the search spaces as SP1-4 throughout this section. SP1, SP2, and SP4 involve Neural Architecture Search (NAS), whereas SP3 optimizes for one neural architecture. We focus our analysis on the Grid Search algorithm as it represents the most general HPO scenario.

B. Efficiency of Proposed Scheduling Policies

We look at the computational time needed to find a solution (scheduling decision) and the quality of solutions of the proposed scheduling policies. We use the MIP solver from Google's OR-Tools [25] as a baseline, where the problem is formed as a linear programming problem with constraints. For this evaluation, we look at SP3 and SP4 shown in table II. We set the number of GPUs to 4 for SP3 and 16 for SP4. Table III shows the occupancy percentage reached by the solution and time needed to solve. As observed, the MIP solver takes around 3 minutes to find the optimal solution. On the other hand, the proposed scheduling policies reach a solution in less than 0.332ms. Comparing the policies, we observe that WFD achieves the highest occupancy for both search spaces.

TABLE II
DESCRIPTION OF THE SEARCH SPACES USED FOR EVALUATION

	SP1	SP2	SP3	SP4
Models	1,2,3,4,5,6	7,8,9	5	1,2,3,4,5,6
Dataset	CIFAR-10	IMDB	CIFAR-10	CIFAR-10
Batch Size	64,128, 256,512	8,16, 32,64	64,128, 256,512	64,128, 256,512
Learning Rate	1e-5 to 1e-1	5e-5 to 5e-1	1e-5 to 1e-1	1e-5 to 1e-1
Total number of jobs	96	96	96	384

TABLE III
COMPUTATIONAL TIME AND ACHIEVED GPU OCCUPANCY OF THE PROPOSED HYPERSACK SCHEDULING POLICIES

Policy	SP3 / 4 GPUs		SP4 / 16 GPUs	
	Occupancy achieved	Time to solve	Occupancy achieved	Time to solve
MIP Solver	99%	181.6 s	100%	212.8 s
HS-FF	88%	0.129 ms	97%	0.264 ms
HS-FFD	97%	0.163 ms	98%	0.298 ms
HS-WF	91%	0.135 ms	97%	0.319 ms
HS-WFD	98%	0.171 ms	100%	0.332 ms

C. Performance Improvement of HyperSack on HPO Workloads with Different Scheduling Policies

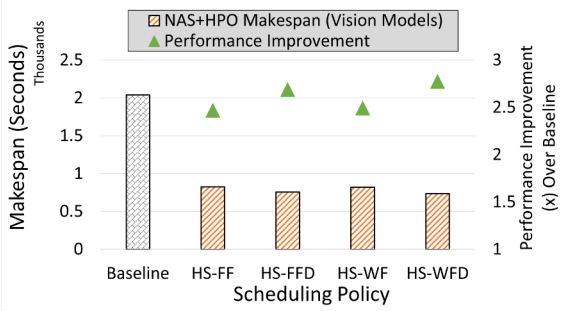

Fig. 10. Evaluation of the proposed *HyperSack* scheduling policies for HPO+NAS vision models workload on SP1.

1) HPO+NAS Vision Workload (SP1): Figure 10 shows the makespan and performance improvement for the proposed scheduling policies over the baseline on 4 A100s for SP1, which consists of 96 jobs spanning vision models 1 to 6. This workload involves Neural Architecture Search (NAS). We observe performance improvement ranging between 2.5x for HS-FF to 2.8x for HS-WFD.

2) HPO+NAS Language Workload (SP2): Figure 11 shows the makespan and performance improvement of using 6 A100s for SP2, which consists of 96 jobs spanning Transformer-based

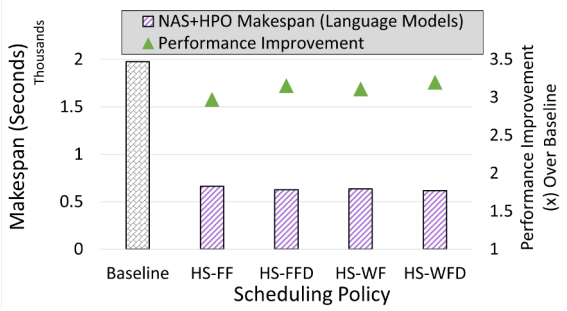

Fig. 11. Evaluation of the proposed *HyperSack* scheduling policies for HPO+NAS language models workload on SP2.

language models 7 to 9. We observe performance improvement ranging between 3.0x for HS-FF to 3.2x for HS-WFD.

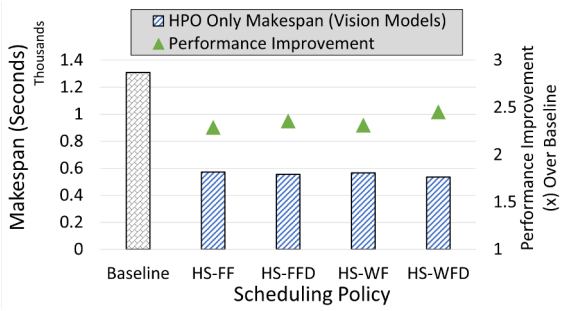

Fig. 12. Evaluation of the proposed *HyperSack* scheduling policies for HPO only vision models workload on SP3.

3) HPO Only Vision Workload (SP3): Figure 12 shows the makespan and performance improvement for the proposed scheduling policies over the baseline on 4 A100s for SP3, which consists of 96 training jobs on the ResNet-18 model. We observe performance improvement ranging between 2.3x for HS-FF to 2.5x for HS-WFD.

The HS-WFD policy delivers the best performance across all workloads, which is consistent with our findings in table III. Therefore, we adopt the HS-WFD policy for the rest of our evaluations.

D. Analysis of GPU Oversubscription on a Single Node

We introduced the oversubscription ratio in section III-A, which is used to scale the compute capacity of the GPU. The oversubscription ratio is tuned by *HyperSack* for specific hardware architectures to achieve better overlap between the CPU and GPU computations.

Figure 13 shows the makespan of the SP1 workload on 4 A100 GPUs using the HS-WFD policy with different oversubscription ratios. We observe higher performance improvement as we increase the ratio gradually. After a ratio of 3, we start seeing a degradation in performance. Two factors cause this degradation: 1) Oversaturation of the available CUDA cores on the GPU, which are needed to perform the forward and backward propagations 2) Oversubscription of CPU cores, which are needed to perform the dataloading and augmentation. The Sampler component profiles metrics independently for each GPU architecture. In figure 14, we conduct similar evaluations

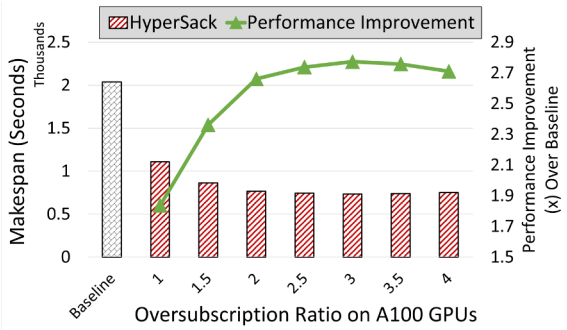

Fig. 13. Analysis of the impact of the GPU oversubscription ratio on the A100 GPU on SP1.

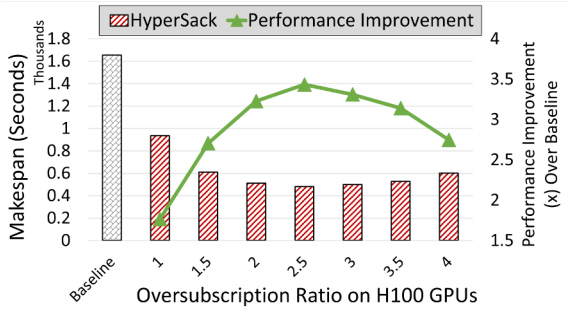

Fig. 14. Analysis of the impact of the GPU oversubscription ratio on the H100 GPU on SP1.

on the H100 architecture and observe similar trends and peak performance with an oversubscription ratio of 2.5.

E. Scaling Efficiency on Homogeneous Systems

To show scaling efficiency, we evaluate *HyperSack* on up to 16 A100 GPUs compared to the baseline on the SP4 workload, which consists of 384 training jobs. Figure 15 shows the makespan and performance improvement on 4, 8, and 16 GPUs. The makespan is reduced from 2.1 hours using the baseline method to 48.2 minutes using *HyperSack* on 4 GPUs, 1.1 hours to 24.5 minutes on 8 GPUs, and 35.1 minutes to 13.1 minutes on 16 GPUs. As we scale, we observe a consistent performance improvement over the baseline of around 2.7x.

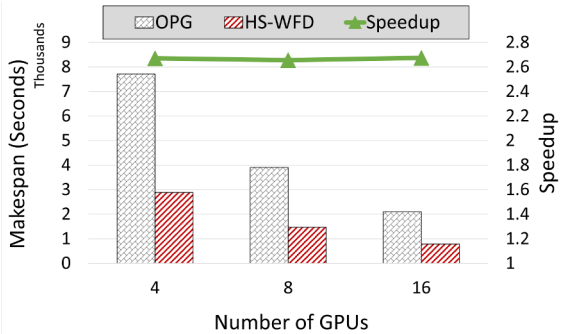

Fig. 15. Evaluation of the scaling efficiency of *HyperSack* on multiple nodes and GPUs on SP4.

F. Scaling Efficiency on Heterogeneous Systems

We evaluate *HyperSack* on up to 4 H100 + 12 A100 GPUs on the SP4 workload, which consists of 384 training jobs.

Figure 16 shows the makespan and performance improvement for three GPU combinations: 4 H100 GPUs, 4 H100 + 6 A100 GPUs, and 4 H100 + 12 A100 GPUs. The makespan is reduced from 1.7 hours using the baseline method to 25.8 minutes using *HyperSack* on 4 H100 GPUs, 49.5 minutes to 12.5 minutes on 4 H100 + 6 A100 GPUs, and 32.5 minutes to 8.4 minutes on 4 H100 + 12 A100 GPUs. We observe a consistent performance improvement over the baseline of around 3.9-4x across all GPU counts.

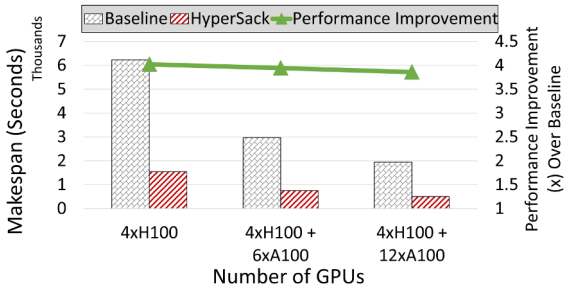

Fig. 16. Evaluation of *HyperSack* with heterogenous hardware resource of A100 and H100 GPUs on SP4.

VI. Related Work

Several works in the literature address GPU scheduling. Looking back at Table I, our main contribution is proposing a heterogeneous resource-aware scheduling framework with resource elasticity and fault tolerance for HPO workloads.

Gandiva [19] designs use time-slice-sharing for packing multiple DNN training jobs on the same GPU. Authors report degradation using MPS with the GPU architectures available in 2018. Gavel [20] builds on top of Gandiva and other scheduling works and uses a heterogeneity-aware round-based scheduling mechanism for DNN training. We have attempted to perform a direct performance comparison with Gavel; unfortunately, the public codebase [26] we found does not support new hardware architectures such as A100 and H100, as many parameters are hardcoded for older GPU generations such as V100 and P100.

Work stealing [27] is a popular scheduling technique used in parallel computing environments to balance the workload among the available processors. In HyperSack, we have one global job buffer that's accessible by all GPUs. GPUs are assigned new jobs based on their occupancy and job heuristics. Our approach avoids the communication/synchronization overhead required for reassigning jobs between buffers in distributed work stealing. Gpulet [17] and GSLICE propose scheduling techniques using spatiotemporal GPU sharing for DNN inference. Inference scheduling has a different set of constraints compared to HPO workloads.

In the context of HPO, authors in [16] propose Salus, a framework that provides fine-grained GPU primitives for deep learning applications. Salus exposes primitives for job switching between multiple tasks. Their approach outperforms traditional approaches by 2.38x for hyperparameter tuning. Liu et al. [28] propose DISC, a resource provisioning approach using time sharing for hyperparameter tuning on cloud services. They model the problem as an optimization problem and design early-release mechanisms for efficient memory sharing. Authors in [29] propose AccDP, an approach that uses MPS for data-parallel distributed deep learning training. Chen et al. [30] propose Euge, a framework that uses NVIDIA MPS and model sharing to accelerate DNN-based video analysis. In [31], Grey et al. use MPS and MIG technologies to maximize throughput in the GROMACS application.

VII. Conclusions and Future Work

Hyperparameter Optimization (HPO) unlocks the potential of DNNs by systematically navigating the hyperparameter space. However, HPO is considered one of the most compute-intensive tasks in DL due to large hyperparameter search spaces and complex neural architectures. Existing HPO parallelization strategies for tuning lightweight DL models underutilize powerful GPU devices, such as the NVIDIA A100 and H100. Assigning multiple jobs to a GPU can improve resource utilization; still, naïve scheduling of HPO jobs may lead to poor performance. We refer to this challenge as the GPU Assignment Problem *GAP*. To address *GAP*, we propose *HyperSack*—a distributed HPO framework designed for dynamic and resource-aware scheduling on heterogeneous GPU-based HPC with resource elasticity and fault tolerance. Our evaluations on language and vision HPO workloads consisting of Transformer language models, CNNs, and Vision Transformers show up to 2.8x performance improvement in execution time on 4 A100 GPUs, 4x on 4 H100 GPUs, and 3.9x on a combination of 12 A100 and 4 H100 GPUs using *HyperSack* compared to traditional HPO parallelization methods. For future work, we plan to extend *HyperSack* with more scheduling policies, HPO algorithms, and support for other GPU architectures from AMD and Intel.

References

[1] L. Yang and A. Shami, "On hyperparameter optimization of machine learning algorithms: Theory and practice," *Neurocomputing*, vol. 415, pp. 295–316, 2020.

[2] L. Liao, H. Li, W. Shang, and L. Ma, "An empirical study of the impact of hyperparameter tuning and model optimization on the performance properties of deep neural networks," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, apr 2022.

[3] A. Goel, C. Tung, Y.-H. Lu, and G. K. Thiruvathukal, "A survey of methods for low-power deep learning and computer vision," in *2020 IEEE 6th World Forum on Internet of Things (WF-IoT)*, pp. 1–6, 2020.

[4] H. J. P. Weerts, A. C. Mueller, and J. Vanschoren, "Importance of tuning hyperparameters of machine learning algorithms," 2020.

[5] "Multi-process service (mps)," 2024. https://docs.nvidia.com/deploy/mps/index.html.

[6] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," 2018.

[7] X. Zhang, X. Zhou, M. Lin, and J. Sun, "Shufflenet: An extremely efficient convolutional neural network for mobile devices," 2017.

[8] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015.

[9] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, "An image is worth 16x16 words: Transformers for image recognition at scale," *CoRR*, vol. abs/2010.11929, 2020.

[10] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," 2019.

[11] S. Biderman, H. Schoelkopf, Q. G. Anthony, H. Bradley, K. O'Brien, E. Hallahan, M. A. Khan, S. Purohit, U. S. Prashanth, E. Raff, *et al.*, "Pythia: A suite for analyzing large language models across training and scaling," in *International Conference on Machine Learning*, pp. 2397–2430, PMLR, 2023.

[12] NVIDIA, "Nvidia management library (nvml)." https://developer.nvidia.com/nvidia-management-library-nvml, 2024.

[13] A. Krizhevsky, V. Nair, and G. Hinton, "Cifar-10 - learning multiple layers of features from tiny images," *Canadian Institute for Advanced Research*, 2009.

[14] A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. Potts, "Learning word vectors for sentiment analysis," in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, (Portland, Oregon, USA), pp. 142–150, Association for Computational Linguistics, June 2011.

[15] NVIDIA, "NVIDIA Nsight Systems." https://developer.nvidia.com/nsight-systems, 2024.

[16] P. Yu and M. Chowdhury, "Fine-Grained GPU Sharing Primitives for Deep Learning Applications," in *Proceedings of Machine Learning and Systems* (I. Dhillon, D. Papailiopoulos, and V. Sze, eds.), vol. 2, pp. 98–111, 2020.

[17] S. Choi, S. Lee, Y. Kim, J. Park, Y. Kwon, and J. Huh, "Serving heterogeneous machine learning models on Multi-GPU servers with Spatio-Temporal sharing," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, (Carlsbad, CA), pp. 199–216, USENIX Association, July 2022.

[18] A. Dhakal, S. G. Kulkarni, and K. K. Ramakrishnan, "Gslice: Controlled spatial sharing of gpus for a scalable inference platform," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, (New York, NY, USA), p. 492–506, Association for Computing Machinery, 2020.

[19] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang, F. Yang, and L. Zhou, "Gandiva: Introspective cluster scheduling for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, (Carlsbad, CA), pp. 595–610, USENIX Association, Oct. 2018.

[20] 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)*, pp. 481–498, USENIX Association, Nov. 2020.

[21] J. Wu, X.-Y. Chen, H. Zhang, L.-D. Xiong, H. Lei, and S.-H. Deng, "Hyperparameter optimization for machine learning models based on bayesian optimizationb," *Journal of Electronic Science and Technology*, vol. 17, no. 1, pp. 26–40, 2019.

[22] NVIDIA, "Multi-instance gpu (mig)." https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html, 2024.

[23] NVIDIA, "Cuda streams." https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf, 2015.

[24] NVIDIA, "Nvidia hyper-q." https://developer.download.nvidia.com/compute/DevZone/C/html_x64/6_Advanced/simpleHyperQ/doc/HyperQ.pdf, 2013.

[25] L. Perron and V. Furnon, "Or-tools." https://developers.google.com/optimization/.

[26] Narayanan et al., "Gavel." https://github.com/stanford-futuredata/gavel.

[27] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, p. 720–748, sep 1999.

[28] L. Liu, J. Yu, and Z. Ding, "Adaptive and Efficient GPU Time Sharing for Hyperparameter Tuning in Cloud," in *Proceedings of the 51st International Conference on Parallel Processing*, ICPP '22, (New York, NY, USA), Association for Computing Machinery, 2023.

[29] N. Alnaasan, A. Jain, A. Shafi, H. Subramoni, and D. K. Panda, "AccDP: Accelerated Data-Parallel Distributed DNN Training for Modern GPU-Based HPC Clusters," in *2022 IEEE 29th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pp. 32–41, 2022.

[30] Q. Chen, G. Ding, C. Xu, W. Qian, and A. Zhou, "Euge: Effective utilization of gpu resources for serving dnn-based video analysis," in *Web and Big Data* (X. Wang, R. Zhang, Y.-K. Lee, L. Sun, and Y.-S. Moon, eds.), (Cham), pp. 523–528, Springer International Publishing, 2020.

[31] A. Gary and S. Páll, "Maximizing gromacs throughput with multiple simulations per gpu using mps and mig." https://developer.nvidia.com/blog/maximizing-gromacs-throughput-with-multiple-simulations-per-gpu-\\using-mps-and-mig, Nov 2021.