Exploiting ML Task Correlation in the Minimization of Capital Expense for GPU Data Centers

Srinivasan Subramaniyan and Xiaorui Wang
Department of Electrical and Computer Engineering
The Ohio State University, Columbus, Ohio, USA
{subramaniyan.4, wang.3596}@osu.edu

***Abstract*—Efficiently scheduling Machine Learning (ML) training tasks in a GPU data center presents a significant research challenge. Existing solutions commonly schedule such tasks based on their demanded GPU utilization, but simply assume that the GPU utilization of each task can be approximated as a constant number (e.g., by using the peak value), even though the ML training tasks commonly have their GPU utilization varying significantly over time. Using a constant number to schedule tasks can result in an overestimation of the needed GPU count and, therefore, a high capital expense for GPU purchases.**

To address this, we design CorrGPU, a correlation-aware GPU scheduling algorithm that considers the utilization correlation among different tasks to minimize the number of needed GPUs in a data center. CorrGPU is designed based on a key observation from the analysis of real ML traces that different tasks do not have their GPU utilization peak at exactly the same time. As a result, if the correlations among tasks are considered in scheduling, more tasks can be scheduled onto the same GPUs, without extending the training duration beyond the desired due time. For a GPU data center to be constructed based on an estimated ML workload, CorrGPU can help the operators purchase a smaller number of GPUs, thus minimizing their capital expense. Our hardware testbed results demonstrate CorrGPU's potential to reduce the number of GPUs needed. Our simulation results on real-world ML traces also show that CorrGPU outperforms several state-of-the-art solutions by reducing capital expense by 20.88%.

***Index Terms*—GPU, ML, scheduling, data center**

I. Introduction

Machine Learning as a Service (MLaaS) is becoming increasingly popular in recent years, because it offers an attractive alternative for users who either cannot afford their own high-end GPUs or have little experience maintaining those GPUs [1]. MLaaS can encompass a variety of frameworks, including the widely used TensorFlow [2], PyTorch [3], and Caffe [4]. These frameworks leverage ML modules to support service providers like Google, Amazon, and Alibaba in training ML models submitted by users for various AI applications, such as image recognition [5], natural language processing [6], and transfer learning.

The rapidly growing demand for such MLaaS services has been driving the constructions of new GPU data centers. However, an important question for many data center operators is: How many GPUs should be purchased for such a data center, so that 1) all the ML training tasks directed to this data center can be serviced in a timely manner, and 2) the capital expense (CapEx) of the data center can be minimized by purchasing only the smallest number of GPUs needed for those tasks. Any over-provisioning of GPUs can lead to wasted computing resources and an unnecessarily higher CapEx, while any under-provisioning can increase the contention of GPU resources among tasks during peak times, thereby delaying their completion.

A common practice adopted by many data center operators is to analyze the real-world traces of a similarly-sized data center (e.g., Alibaba's PAI trace [7] or Microsoft's Philly trace [8]). Then, the number of GPUs required to run all those tasks in a timely manner can be estimated in a fairly accurate way. This approach enables data center operators to make informed decisions about GPU provisioning and optimize resource allocation for efficient data center operations. Specifically, we aim to minimize the number of GPUs needed for processing all the training tasks in given ML traces, without violating the desired task *due times*, which refer to user-defined deadlines for those training tasks to be completed.

Many scheduling solutions have been previously proposed for ML training tasks [9]–[13]. While those solutions rely on various metrics for scheduling, GPU utilization (i.e., the percentage of time a GPU is actively involved in computation) is commonly used to schedule multiple tasks on the same GPU if their aggregated GPU utilization is lower than the GPU capacity (e.g., 100%), subject to other constraints like memory [7], [11], [14]. However, those utilization-based solutions typically assume that the GPU utilization of each task can be approximated as a constant number (e.g., peak or average utilization) and then use those constant numbers in their scheduling. This oversimplified approach may not accurately reflect the fluctuating nature of GPU utilization in ML training workloads, as observed in empirical traces (e.g., [7] [15]), where the GPU utilizations of many tasks vary significantly throughout their task duration. Hence, such approximation may not result in the best scheduling results. For example, if peak utilization is always used, operators may end up purchasing an unnecessarily large number of GPUs, which leads to a higher CapEx. If average is always used, when multiple tasks are scheduled on the same GPU but peak at the same time, intensive GPU contention may cause those tasks to violate their desired due times.

To minimize the number of GPUs needed in a data center,

This work was supported, in part, by the U.S. National Science Foundation under Grant CNS-2336886.

we propose CorrGPU, a correlation-aware GPU scheduling algorithm that considers the utilization correlation among different ML training tasks to reduce needed GPUs. CorrGPU is designed based on a key observation from the analysis of real ML traces that different tasks do not have their GPU utilization peak at exactly the same time. As a result, if the correlations among different tasks are analyzed and considered in scheduling, more tasks can be scheduled onto the same GPUs, without extending the training duration beyond the desired due time. For example, if two tasks have negative utilization correlation, they can be co-scheduled on the same GPU because they do not simultaneously demand for a large amount of GPU time. Because data centers often provision training and inference GPUs as separate clusters to ensure that inference is processed promptly without being affected by training [11], [16], CorrGPU focuses on scheduling training tasks, since inference tasks typically have very short durations, usually in milliseconds [11], [14], [17]–[19]. Therefore, we focus on training tasks with longer durations and provide multiple data points for correlation analysis. For a GPU data center to be constructed, CorrGPU can help the operators purchase a smaller number of GPUs, thus minimizing the data center's CapEx. Even for an existing data center with pre-purchased GPUs, CorrGPU can help operators avoid purchasing more GPUs when their workloads increase due to the rapidly booming ML business, which also reduces CapEx.

Specifically, this paper makes the following contributions:

- By analyzing real-world ML traces, we observe that ML tasks usually have weak pairwise correlations and thus do not peak at the same time. Based on this observation, we propose to schedule ML tasks according to their correlations, so that the number of needed GPUs can be minimized for a lower CapEx.
- We formulate the minimization of needed GPUs in a data center as a constrained optimization problem. Due to its high complexity, we mathematically derive the optimal solution that is impractical to use in a real data center due to its computational overheads, but can serve as an upper bound for the design of heuristic algorithms.
- We propose CorrGPU, a correlation-aware GPU scheduling algorithm that considers the utilization correlation among different tasks to minimize the number of needed GPUs in a data center. With a much smaller overhead, the scheduling performance of CorrGPU is close to that of the optimal solution.
- We evaluate CorrGPU both on our hardware GPU testbed with Nvidia V100 GPUs and in trace-driven simulation. Our testbed results demonstrate CorrGPU's potential reduction of needed GPUs. Our large-scale simulation results also show that CorrGPU outperforms several state-of-the-art solutions by having 20.88% less CapEx.

II. Problem Formulation

In this section, we first introduce the notation and correlation analysis, and then present the constrained optimization problem. Finally, we outline the iterative approach used to determine the minimal number of GPUs required. We begin by providing a list of the notation used throughout this paper.

Let G denote the total number of GPUs under consideration, and let L represent the total number of tasks. Each task j is characterized by its memory requirement m_j, normalized utilization u_j, arrival time a_j, and dispatch time d_j. The memory capacity of each GPU is denoted by c. The correlation between any two tasks j and k is expressed as $\rho_{j,k}$. The number of GPUs allocated to a task is given by v, where $v \in \{1, \dots, G\}$; for example, $v = 1$ for single-GPU training and $v > 1$ for multi-GPU training.

We define a binary decision variable $x_{i,j}$, where $x_{i,j} = 1$ if task i is assigned to GPU j, and 0 otherwise. Similarly, y_j is a binary variable such that $y_j = 1$ if GPU j is used and 0 if it is not. A correlation threshold α is used to restrict co-location of tasks; specifically, if $\rho_{i,k} > \alpha$, then tasks i and k are not allowed to be assigned to the same GPU. Finally, P denotes the price per GPU, which is used to compute the capital expense (*CapEx*).

Correlation Analysis. Before discussing the optimization problem, we define the Pearson correlation coefficient between two data vectors x and y, denoted $\rho(x, y)$. The correlation coefficient for two samples, $x = \{x_1, \dots, x_n\}$ and $y = \{y_1, \dots, y_n\}$, is given by:

$$\rho(x,y) = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}, \tag{1}$$

where $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$ and $\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$ are the sample means of x and y, respectively. The correlation coefficient ρ_{xy} usually ranges from -1 to 1. In this paper, the term *correlation* refers explicitly to the statistical correlation between the GPU utilization time series of two tasks.

Optimization Problem. The optimization problem focuses on minimizing CapEx. The formula for computing CapEx is:

$$\text{CapEx} = \left(\sum_{j=1}^{G} y_j\right) \times P. \tag{2}$$

Here, minimizing the number of GPUs effectively reduces CapEx since fewer active GPUs ($y_j = 1$) means lower cost. Instead of directly assuming a very large G, we use an iterative approach to estimate the minimum number of needed GPUs as shown below:

1) Initialize $G = 1$.
2) For a given G, the solver attempts to find a feasible solution to the optimization problem defined in Equation (3) with the constraints in Equations (4–7).
3) If no feasible solution is found, increment G and repeat.
4) Once a feasible solution is found for some $G = G^*$, this G^* is the minimum number of GPUs required.

Objective Function per iteration. For a given G, the optimization objective is to minimize the total number of active GPUs following the approaches used in previous work

[20], [21], ensuring that all tasks are scheduled to the available GPUs (G), subject to the constraints listed below.

$$\min \sum_{j=1}^{G} y_j, \tag{3}$$

1) **Memory Capacity:** The total memory usage of tasks on each GPU must not exceed its capacity:

$$\sum_{i=1}^{L} m_i \cdot x_{i,j} \leq c \cdot y_j, \quad \forall j \in \{1, 2, \ldots, G\}. \tag{4}$$

2) **Correlation:** Tasks that are highly correlated (i.e., $\rho_{i,k} > \alpha$) cannot be placed on the same GPU to avoid simultaneous peaks:

$$x_{i,j} + x_{k,j} \leq 1, \quad \forall (i,k) : \rho_{i,k} > \alpha, \quad \forall j \in \{1, 2, \ldots, G\}. \tag{5}$$

Here α is a user-defined correlation threshold, typically in the range (-1,1).

3) **Task Assignment:** Each task must be assigned to exactly v GPUs:

$$\sum_{j=1}^{G} x_{i,j} = v, \quad \forall i \in \{1, 2, \ldots, L\}. \tag{6}$$

Here $v = 1$ for Single GPU Training and $v > 1$ for Distributed GPU Training.

4) **Dispatching:** All tasks should be dispatched as soon as they arrive.

$$d_i \geq a_i + \epsilon, \quad \forall i \in \{1, 2, \ldots, L\}. \tag{7}$$

Here ϵ is a small positive constant representing the minimal dispatching overhead.

If the solver is unable to find a feasible solution for this optimization problem with the current G, we increase G and repeat the process. The smallest G for which feasibility is achieved, denoted G^*, is the optimal solution.

Computational Complexity and Solver Limitations: We use PuLP [22] to obtain the optimal solution. The solver's tolerance is set to 1×10^{-6}. As the number of tasks (L) increases, the complexity of the problem increases exponentially. This results in significantly longer solving times, as shown in Table I.

TABLE I: Computation Time for Varying Numbers of Tasks

Tasks	2	4	8	16	32	64	128
Solve Time (s)	0.04	0.07	0.16	0.22	4.92	8.95	80.9

The iterative approach and constraints ensure that the smallest feasible number of GPUs, G^*, is identified, serving as an optimal solution for comparison against CorrGPU.

III. Design of CorrGPU

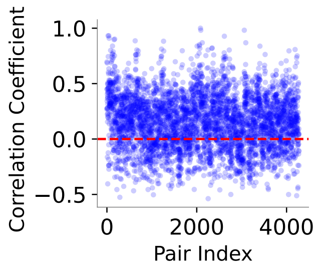
This section first performs the correlation analysis of machine learning workloads. Next, we provide an example showing how CorrGPU works. We then develop the CorrGPU framework, analyze its complexity and overhead, and conclude by discussing practical considerations.

A. Correlation Analysis of Machine Learning Workloads

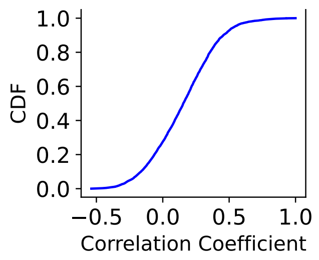
There are different granularities of utilization in GPUs. GPU utilization from the *nvidia-smi* interface measures the percentage of time over the past sample period during which one or more kernels are executing on the GPU [23]. At the streaming multiprocessor (SM) [24] level, utilization measures the distribution and execution efficiency of thread blocks across SMs. At the instruction level, utilization is measured as the ratio of successfully issued warp instructions to stalled cycles, indicating the effectiveness of the warp scheduler. Finally, tensor core utilization focuses on the performance of specialized cores for matrix operations, evaluating the dominance and throughput of tensor core instructions.

Although there are multiple methods to compute GPU utilization, coarser-granularity metrics, such as GPU utilization from the *nvidia-smi* interface, are preferred in data center scheduling because the overhead of obtaining this utilization is negligible. Finer-grained metrics, such as SM and tensor core utilization, are helpful for application tuning; however, they add significant overhead and complexity. This paper adopts GPU-level utilization, as we focus on cluster-level scheduling; however, it is important to note that CorrGPU can also utilize other metrics in its correlation analysis. In this work, we assume the availability of predicted task utilization patterns for all tasks through short-term profiling or predictive modeling.

We investigate both vision and transformer workloads to gain insight into correlation analysis for real-world Machine Learning tasks. The selected workloads include DenseNet [25], Inception [26], MobileNet [27], ResNet [5], GoogleNet [28], VGG [29], Swin Transformer [30], and GPT-2 [31]. These workloads are executed on our hardware testbed, where GPU utilization is measured using the '$nvml$' interface [23]. The measured GPU utilization indicates the percentage of time that GPU kernels were active during the previous sampling period. Each workload is executed by fixing the epoch count at 80 and varying the batch size (B) across $10, 20, 30, 40, 50$, and 60.



(a) Correlation Coefficient



(b) CDF

Fig. 1: (a) Scatter plot showing the distribution of pairwise correlation coefficients. (b) Cumulative Distribution Function (CDF) plot depicting the overall distribution pattern of the pairwise correlation coefficients.

We calculate the pairwise correlation coefficients using Equation (1) on GPU utilization obtained from the above workloads. Using correlation analysis is essential because

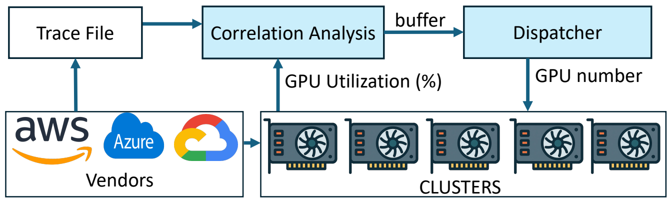


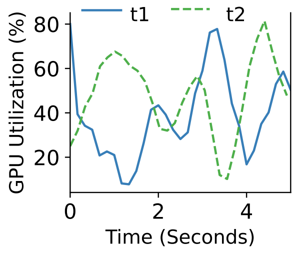
Fig. 2: The design of our proposed CorrGPU framework. The framework comprises a correlation analyzer and a dispatcher.

tasks that exhibit a negative correlation do not simultaneously peak in resource utilization. Consequently, these tasks do not compete for GPU resources, allowing for more efficient utilization of the GPUs for multiple tasks running concurrently. The scatter plot in Figure 1a illustrates the variation in the correlation coefficients for all pairs of workloads, while the cumulative distribution function (CDF) in Figure 1b shows the distribution of these coefficients. The scatter plot reveals that most workload pairs have weak or negative correlations, indicating that their GPU utilization does not peak at the same time. The CDF in Figure 1b shows that 75% of the workload pairs have correlation coefficients below 0.25, suggesting a limited overlap in utilization peaks.

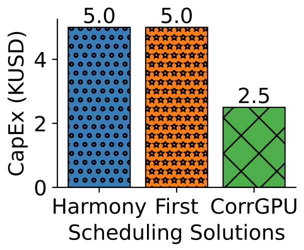
B. Overview and Example

The design of our proposed CorrGPU framework is illustrated in Figure 2. CorrGPU consists of component correlation analysis and a dispatcher. We perform correlation analysis and dispatch tasks to the GPU that has the lowest correlation coefficient.

We illustrate a simple example to show how CorrGPU works. The primary goal of this example is to demonstrate how each baseline functions and to highlight the differences between the proposed solution and the baselines. The baseline used in this example is First-Fit (First). *First* uses bin packing to schedule tasks, relying on the utilization threshold (q) as a metric for their scheduling decisions. *First* computes the sum of the GPU utilization during the first second for the incoming task and adds it to the aggregated GPU utilization of the tasks already running on the GPU. If their total utilization is less than the threshold (q), it schedules the tasks to the same GPU.



(a) GPU Utilization for the motivation example.



(b) CapEx for different scheduling strategies.

Fig. 3: (a) GPU utilization and (b) CapEx comparison in the motivational example. CorrGPU yields the lowest CapEx among all baselines.

This example considers two training tasks (t_1) ResNet [32] with a batch size of 40 and (t_2) SqueezeNet [25] with a batch size of 30 as shown in Figure 3a. The average utilizations of t_1 and t_2 are 41.36% and 41.92%, respectively, and their peak utilizations are 80.80% and 61.80%, respectively. The first sample utilization of t_1 and t_2 are 80% and 25%, respectively. For DNN training, since the model architecture remains the same and only the input data changes, the GPU utilization can be periodic as we go through the same layers [33] [34]. *First* schedules tasks based on the initial utilization in the first second. For t_1 and t_2, the initial utilizations are 80% and 25%, respectively. Their sum ($80\% + 25\% = 105\%$) exceeds the threshold ($q = 100\%$). Thus, t_1 and t_2 are assigned to GPU 1 and GPU 2, respectively, as shown in Figure 4. *CorrGPU* schedules tasks with negative correlation coefficients on the same GPU to minimize resource contention. As soon as t_1 and t_2 arrive, we compute their correlation coefficient (ρ) using Equation (1). The correlation coefficient is -0.8, which is below the correlation threshold ($\alpha = 0$). The sum of their average utilization is $(41.36\% + 41.92\%) = 83.28\%$, which is less than the utilization threshold ($q = 100\%$). Since both criteria are satisfied, CorrGPU schedules t_1 and t_2 to GPU 0, as shown in Figure 4. It is important to note that even if the combined resource demand of two tasks temporarily exceeds the available capacity, it may only result in a minor delay in execution time without causing failure or system instability. Furthermore, even if the predicted utilization deviates from the desired utilization, it does not significantly impact the CTD, as discussed in Section V.

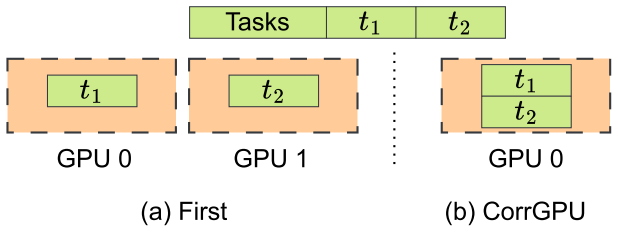


Fig. 4: CorrGPU uses only one GPU and decreases the *CapEx* for the motivational example shown in Section III-B.

We compare the *CapEx* of CorrGPU and First using Equation (2). We assume $P = 2.5$ K USD as the cost of a single 40 GB A-100 GPU [35]. Figure 3b shows the *CapEx* for all solutions. First requires two GPUs, leading to a *CapEx* of $2 \times 2.5 = 5$ K USD. CorrGPU requires only one GPU, leading to a *CapEx* of $1 \times 2.5 = 2.5$ K USD. Therefore, CorrGPU reduces *CapEx* by 50% compared to First.

C. CorrGPU Framework

Algorithm 1 details the design steps of CorrGPU. The algorithm activates whenever a new task arrives. As mentioned in Section III-B, CorrGPU performs correlation analysis and dispatching. The steps performed by CorrGPU are listed below.

1) When a new task $t_i \in L$ (where $i > 1$) is released from the task queue, the CorrGPU module accesses all

Algorithm 1 CorrGPU

```
Upon arrival of task $t_i$
Input: Memory requirement of task $m_i$, utilization of task $u_i$, correlation threshold $\alpha$, distribution factor $v$
Data: gputracker
1:  $buffer \leftarrow []$
2:  $task\_scheduled \leftarrow$ False
3:  for $g_j \in gputracker$ do
4:      $C_{avail} \leftarrow$ available DRAM capacity of $g_j$
5:      $g_j(u) \leftarrow$ current utilization of GPU $(g_j)$
6:      if $m_i \leq C_{avail}$ then
7:          $\delta \leftarrow \rho(g_j(u),\ t_i(u))$
8:          if $\delta < \alpha$ then
9:              $\delta_1 \leftarrow \mu(g_j(u),\ t_i(u))$
10:             $buffer \leftarrow$ push$((\delta,\ \delta_1,\ g_j))$
11:             $task\_scheduled \leftarrow$ True
12:         end if
13:     end if
14: end for
15: if not $task\_scheduled$ or $buffer = \emptyset$ then
16:     Allocate a new GPU $g_{new}$
17:     $gputracker \leftarrow gputracker \cup \{g_{\text{new}}\}$
18:     for $k = 1$ to $v$ do
19:         dispatch$(t_i) \rightarrow g_{\text{new}} + k$
20:     end for
21: else
22:     Sort $buffer$ by increasing $\delta$, then increasing $\delta_1$
23:     $g_{\text{best}} \leftarrow buffer[0].g_j$
24:     for $k = 1$ to $v$ do
25:         dispatch$(t_i) \rightarrow g_{\text{best}} + k$
26:     end for
27: end if
```

the GPUs listed in the *gputracker* (which contains GPUs that have tasks running on them) and evaluate if task (t_i) fits within the available DRAM capacity of GPU g_i (ie, $m_i < C_{\text{avail}}$) as shown in Lines $(3-4)$.

2) If the DRAM capacity condition holds, CorrGPU calculates the correlation coefficient δ between the utilization of task t_i and previously scheduled tasks on GPU g_i, as shown on Line 7. Since the correlation coefficient in Equation (1) requires vectors of equal length, we compute the minimum of the two task durations and truncate both traces to this common length before computing the coefficient. For upcoming tasks whose full utilization traces are not available, CorrGPU relies on *offline profiling*. Specifically, we pre-profile representative utilization traces for each combination of model architecture and batch size. When a new task t_i arrives, its expected utilization trace is retrieved from this offline database and compared with historical tasks on g_i. The correlation is evaluated over the overlapping window. For instance, if a previously executed task ran for 40 seconds and the profiled trace of t_i spans 30 seconds, the correlation is computed over the first 30 seconds of both traces. CorrGPU assumes that job metadata (e.g., architecture, batch size, and training configuration) is available in advance; this is standard in data center ML scheduling pipelines.
3) In Line 8, CorrGPU evaluates whether the correlation coefficient δ between task t_i and the tasks running on GPU g_i is below the specified threshold α. In Line 9, CorrGPU calculates δ_1, the average utilization between the new task t_i and the aggregate utilization of all tasks already running on GPU g_i. Importantly, this utilization value is treated as a soft constraint rather than a strict hardware limit. Since GPU utilization is dynamic, occasional overlaps that slightly exceed GPU capacity may introduce minor execution delays but typically do not result in failure or instability.
4) In line 8, once a GPU g_j satisfies the memory availability and correlation threshold criteria, the algorithm computes the mean utilization δ_1 between the new task t_i and the set of tasks currently running on g_j. It then stores a tuple $(\delta,\ \delta_1,\ g_j)$ in the *buffer*. This buffer acts as a temporary holding list of candidate GPUs that are eligible for task placement.
5) If the *buffer* is empty, it indicates that no existing GPUs that have tasks running on them can satisfy the scheduling constraints. This situation might arise from insufficient DRAM capacity or if the correlation coefficient exceeds the threshold. In this case, as illustrated in lines 15–19, a new GPU is dynamically allocated to t_i. The new GPU is added to the *gputracker*, and t_i is dispatched there. If $v = 1$, the task goes to a single GPU; if $v > 1$, it is distributed among multiple GPUs, beginning with g_{new} and increasing the indices as $g_{\text{new}+k}$.
6) If the *buffer* contains items, the scheduler sorts the *buffer* to choose the GPU with the lowest correlation coefficient (line 22) and mean utilization. The first GPU in the sorted list is selected for dispatch. Using the distribution factor (v), t_i is spread across GPUs starting from the selected GPU g_j, as shown in lines 25.

Complexity Analysis The time complexity of *Algorithm 1* is determined by three main operations: iterating over tasks, evaluating candidate GPUs, and sorting the candidate buffer. For n tasks, CorrGPU dynamically allocates GPUs as needed, with the number of active GPUs at any point denoted by g, where $g \leq n$. The outer loop iterates over all n tasks, contributing $O(n)$. For each task, the algorithm examines up to g GPUs stored in *gputracker*, resulting in an additional $O(g)$ per task. This leads to a task-GPU iteration cost of $O(n \times g)$. Additionally, for each task, CorrGPU sorts a buffer of up to g candidate GPUs based on correlation scores, contributing $O(g \log g)$ per task. Across all tasks, this results in a total cost of $O(n \times g \log g)$. Combining both contributions, the overall time complexity is $O(n \times g + n \times g \log g) = O(n \times g \log g)$. In the worst-case scenario where each task is assigned to a unique GPU (i.e., $g = n$), the complexity becomes $O(n^2 \log n)$.

D. Practical Consideration

CorrGPU assumes the availability of predicted task utilization patterns through short-term profiling or predictive modeling. CorrGPU tolerates small inaccuracies as long as the relative timing and magnitude of utilization fluctuations are preserved. Our evaluations (Sections V and VI) confirm that CorrGPU remains effective even under real-world traces. CorrGPU may be less beneficial when workloads are highly synchronized or when accurate utilization traces are unavailable. In such cases, prediction errors in GPU utilization may lead to a slight degradation in scheduling performance. While CorrGPU is designed to exploit weak or negative correlations under typical ML training traces, real-world data centers often encounter workload surges, shifting deadlines, and unpredictable arrival patterns. Evaluating CorrGPU under such bursty scenarios is left for future work.

Currently, CorrGPU conducts correlation analysis only for GPU utilization and considers other resource demands, such as memory capacity, as hard constraints. It is essential to note that CorrGPU can be extended to perform correlation analysis on memory bandwidth and interconnect traffic, thereby transforming the current one-dimensional scheduling model into a multidimensional scheduling model. However, such an extension may significantly increase the computation complexity and result in only a small improvement because the demanded memory capacities of ML workloads are generally more stable than their GPU utilization.

IV. Experimental Setup

Real-world ML Trace. We use the Alibaba PAI trace [7] for all the experiments. The PAI trace includes both training and inference jobs. Each job entry has a unique ID, start time, duration, requested GPU memory, utilization, and job status. We exclude jobs that fail or report zero GPU memory or utilization.

Hardware Testbed. Our testbed consists of six Nvidia V100 GPUs. We use Ubuntu 20.04 LTS as our server OS, along with CUDA Toolkit 11.6. We use PyTorch 1.12.1. as the ML framework. For hardware evaluation, we conduct scheduling online and dispatch offline. To enable spatial sharing, the CUDA Multiprocessing Service (MPS) is used. The MPS thread percentage is fixed at 100% to ensure full GPU provisioning. All workloads from Section III-A are incorporated into this evaluation. We adjust the batch sizes of the workloads to align with those specified in the PAI [7] trace.

Trace-driven Simulation. To address the limitations of our hardware testbed, we developed a Python simulator to evaluate CorrGPU on a large scale using real-world traces. The simulator is validated by ensuring that it produces the same results as our testbed in smaller-scale experiments, confirming its accuracy. The simulator is built from the open-source code of GPUColo [17], designed to emulate a heterogeneous data center.

Performance Metrics. We use two metrics to compare the performance of the proposed solution with the baselines. The first metric is (*CapEx*), defined in Equation (2). The second metric is Cumulative Task Duration (*CTD*) represented as CTD $= \sum_{j \in L} c_j$, where L is the set of all tasks, and c_j is the completion time of task j.

Baselines. *(1) Harmony* calculates the sum of the maximum utilization of the existing and incoming tasks on the GPU. If the combined peak utilization of the two is less than the utilization threshold (q), Harmony schedules them to the same GPU [13]. The prediction of utilization follows the approach suggested by Yeung et al. [36]. *(2) First* computes the sum of the GPU utilization during the first second for the incoming task and adds it to the aggregated GPU utilization of the tasks already running on the GPU. If their total utilization is less than the threshold (q), it schedules the tasks to the same GPU.

V. Hardware Evaluation

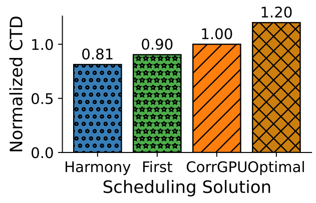
In this section, we evaluate the performance of *CorrGPU* against the baselines described in Section IV. Importantly, this experiment is designed to demonstrate *CorrGPU*'s robustness to model error, showing that our correlation-aware scheduler remains effective even when task utilization predictions are noisy or inaccurate. We generate synthetic GPU utilization traces by sampling, for each task, from a Gaussian distribution whose mean (μ) and variance (σ^2) match those of real ML training tasks in the PAI trace [7].

We select 14 ML tasks from the PAI trace based on their GPU utilization and memory requirements. Tasks arrive dynamically in two bursts. In stage 1, tasks t_1, t_2, t_3, t_4 arrive at 0, 1, 2, and 3 seconds, and tasks t_5, t_6, t_7, t_8 arrive at 20, 21, 22, and 23 seconds. In stage 2, tasks $t_9, t_{10}, t_{11}, t_{12}$ arrive at 400 seconds and tasks t_{13}, t_{14} at 420 and 422 seconds. Tasks t_9–t_{14} replicate the workloads of t_1–t_8 respectively (e.g. t_9 uses t_1's workload, t_{10} uses t_2's, etc.). We repeat each configuration five times and report mean results with one standard deviation.

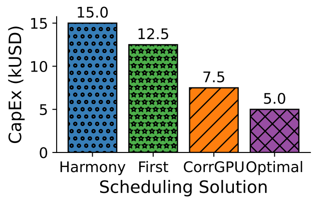
In the first step, we measure the number of GPUs used throughout the experiment. *CorrGPU* requires 3 GPUs, compared to 5 for *First*, 6 for *Harmony*, and 2 for *Optimal*.

Next, we compare the *CapEx* and normalized *CTD* for all baselines and *CorrGPU*. We normalize to *CorrGPU*. The normalized *CTD* for all solutions appears in Figure 5a. The normalized *CTD* of *Harmony* is 0.81, and for *First*, it is 0.90. Gao et al. [37] survey 103 researchers to understand their tolerance for deadline delays in deep learning training (DLT) tasks with SLA guarantees. The results show that approximately 3%, 12%, 21%, and 22% of participants accept deadline extensions of 0%, 5%, 10%, and 25%, respectively. Thus, we use 25% as the threshold for SLO violations (*due time*). *CorrGPU*'s *CTD* is 1.0, which stays within 1.25× *Harmony*'s *CTD*. This shows that CorrGPU meets its *due time*. *Optimal*'s *CTD* is slightly higher because it uses fewer GPUs than *CorrGPU*.

CorrGPU has the second smallest *CapEx* as shown in Figure 5b since it uses fewer GPUs. Compared to *Harmony*, *CorrGPU* saves 50% of *CapEx*. Compared to *First*, *CorrGPU* saves 40% of *CapEx*. *Optimal*'s *CapEx* is smaller since it uses fewer GPUs and the solver finds the most optimal solution. This is not feasible when the number of tasks increases, as stated in Section II.



(a) CTD



(b) CapEx

Fig. 5: Hardware evaluation using the PAI trace. (a) *CorrGPU*'s *CTD* is smaller than that of *Optimal* and remains within the due time. (b) *CorrGPU* uses fewer GPUs compared to *Harmony* and *First*, resulting in a lower *CapEx*.

VI. Simulation Evaluation

As introduced earlier, CorrGPU estimates the minimum number of GPUs required and does not significantly increase the CTD of the tasks, as shown in Section V. We perform large-scale simulations to determine the optimal number of GPUs required for CorrGPU. We compare CorrGPU against the baselines, First and Harmony. As mentioned earlier, we do not compare our results with the optimal solution for the simulation due to the complexity described in Section II. In this section, we perform a trace evaluation, assessing our solution using the PAI trace comprising of 5000 tasks.

In this experiment, we test the PAI trace after the pre-processing mentioned in Section IV. Figure 6 illustrates the variation in the average number of GPUs used for the experiment. CorrGPU utilizes fewer GPUs compared to the baselines, namely First and Harmony. On average, Harmony uses 23.2 GPUs, First uses 16.7 GPUs, while CorrGPU requires only 15.32 GPUs. This corresponds to a reduction in GPU usage of approximately 34% compared to Harmony and 8.3% compared to First.

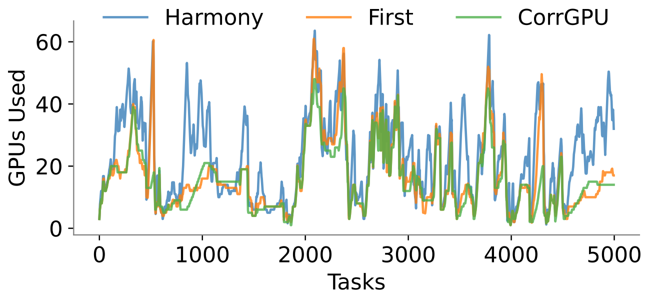
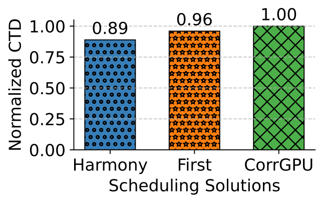
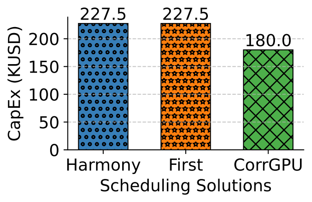


Fig. 6: CorrGPU uses fewer GPUs compared to the baselines for the large-scale simulation using the PAI trace.

In Figure 6, we observe a significant increase in GPU usage across all baselines during the first 200 tasks. This spike is attributed to the dispatch of a large volume of tasks that have GPU utilization values exceeding 80%, necessitating the assignment of separate GPUs for each task. Furthermore, during this period, the correlation values remain positive, further contributing to the increase in GPU usage in all baselines, including *CorrGPU* . Reducing the number of GPUs must not increase task durations beyond the *due times*.



(a) CTD



(b) CapEx

Fig. 7: (a) CorrGPU exhibits a slightly higher Cumulative Task Duration (CTD) than Harmony (0.89) and First (0.96), but remains within the acceptable due times. (b) CorrGPU achieves significantly lower *CapEx* than both baselines by reducing the total number of unique GPUs provisioned.

As shown in Figure 7a, *CorrGPU* demonstrates a slightly higher *CTD* compared to the baseline approaches, *Harmony* and *First*, which have *CTD* values of 0.89 and 0.96, respectively. In contrast, *CorrGPU's CTD* is 1.02, which remains within the due time threshold of $1.25 \times 0.89 = 1.11$, where 0.89 is *Harmony's* CTD. *CorrGPU* reduces resource contention by consolidating tasks with negative correlations and minimizing contention.

Next, we evaluate the (*CapEx*) for each scheduling solution using Equation (2). Following the methodology outlined in Algorithm 1, we count every GPU that is assigned at least one task during the entire trace, reflecting realistic provisioning requirements. As shown in Figure 7b, both Harmony and First require 91 unique GPUs, corresponding to a *CapEx* of \$227.5K. In contrast, *CorrGPU* uses only 72 GPUs, resulting in a significantly lower *CapEx* of \$180.0K. This represents a cost reduction of approximately 20.88% compared to both baselines. The savings arise from *CorrGPU's* ability to leverage task correlation and utilization patterns to co-locate tasks more effectively and minimize redundant GPU usage.

VII. Related Work

A large amount of research work has been carried out on studying scheduling training workloads in GPU data centers. These research works focus on efficient resource consumption, cost, and timing efficiency. The related work can be classified into two categories: (1) Popular solutions used in the industry for scheduling training tasks in data centers and (2) Training scheduling techniques used in the literature.

Industrial solution. Alibaba [38] conducted a comprehensive study on a GPU cluster comprising 6000 GPUs dedicated solely to machine learning workloads. The investigation delved into load imbalances and low GPU resource utilization. Additionally, the study critically analyzed the limitations of elastic scheduling within data centers. In response, Alibaba introduced an analytical framework [7] [39] designed to assess execution times and pinpoint bottlenecks specific to various machine learning workloads. The PAI trace from Alibaba encompasses training and inference tasks with their GPU utilization, memory consumption, and other task specifications. The trace indicated that the utilization of P100 GPUs deployed in the data center exhibited a broad spectrum, ranging from 40% to 100%. This illustrates the wide range of utilization levels observed across workloads.

Training scheduling techniques. Various scheduling techniques, such as gang and elastic scheduling, are used in data centers [9]. Gang scheduling requires all GPUs to be allocated simultaneously for a task [40], while elastic scheduling allows flexible GPU usage. Some schedulers employ bin packing or first-fit approaches, considering average memory usage and task utilization [41]. Gandiva [42] ensures resource fairness, while Gavel [43] optimizes for heterogeneity-aware scheduling.

To our best knowledge, CorrGPU is the first study that proposes a correlation analysis of GPU utilization to significantly reduce CapEx and ensure that task durations are within the due times.

VIII. Conclusion and Future Work

The rapidly growing demand for MLaaS services has been driving the construction of new GPU data centers. However, an important question for many data center operators is how to minimize their investment in CapEx, while managing to schedule all the expected ML workloads without causing undesired long delays. This paper presents CorrGPU, a correlation-aware GPU scheduling algorithm that considers the utilization correlation among different tasks to minimize the number of needed GPUs in a data center. CorrGPU is designed based on a key observation from the analysis of real ML traces that different tasks do not have their GPU utilization peak at exactly the same time. Our simulation results on real-world ML traces also show that CorrGPU outperforms several state-of-the-art solutions by reducing capital expense by 20.88%. As part of our future work, we plan to conduct a sensitivity study on the correlation threshold parameter to evaluate the trade-off between GPU savings and task latency. Furthermore, we intend to extend the evaluation of CorrGPU to large-scale LLM-based training tasks.

References

[1] M. Ribeiro *et al.*, "Mlaas: Machine learning as a service," in *ICMLA*, 2015.
[2] M. Abadi *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.
[3] A. Paszke *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *NeurIPS*, 2019.
[4] Y. Jia *et al.*, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*, 2014.
[5] K. He *et al.*, "Deep residual learning for image recognition," in *CVPR*, 2016.
[6] I. Tenney *et al.*, "Bert rediscovers the classical nlp pipeline," *arXiv preprint arXiv:1905.05950*, 2019.
[7] M. Wang *et al.*, "Characterizing deep learning training workloads on alibaba-pai," in *IISWC*, 2019.
[8] M. Jeon *et al.*, "Analysis of large-scale multi-tenant gpu clusters for dnn training workloads," in *ATC*, 2019.
[9] W. Gao *et al.*, "Deep learning workload scheduling in gpu datacenters: Taxonomy, challenges and vision," *arXiv preprint arXiv:2205.11913*, 2022.
[10] J. Mohan *et al.*, "Synergy: Resource sensitive dnn scheduling in multi-tenant clusters," *arXiv preprint arXiv:2110.06073*, 2021.
[11] A. Dhakal *et al.*, "Gslice: controlled spatial sharing of gpus for a scalable inference platform," in *SoCC*, 2020.
[12] D. Yoon *et al.*, "Deft: Exploiting gradient norm difference between model layers for scalable gradient sparsification," in *ICPP*, 2023.
[13] Y. Bao *et al.*, "Deep learning-based job placement in distributed machine learning clusters," in *INFOCOM*, 2019.
[14] G. Chen *et al.*, "Performance optimization of machine learning inference under latency and server power constraints," in *ICDCS*, 2022.
[15] Q. Hu *et al.*, "Characterization and prediction of deep learning workloads in large-scale gpu datacenters," in *SC*, 2021.
[16] Z. Bai *et al.*, "Pipeswitch: Fast pipelined context switching for deep learning applications," in *OSDI*, 2020.
[17] G. Chen *et al.*, "Latency-guaranteed co-location of inference and training for reducing data center expenses," in *ICDCS*, 2024.
[18] Y. Ma *et al.*, "Power capping of gpu servers for machine learning inference optimization," in *ICPP*, 2025.
[19] K. Vanishree *et al.*, "Coin: Accelerated cnn co-inference through data partitioning on heterogeneous devices," in *ICACCS*, 2020.
[20] X. Wang *et al.*, "Correlation-aware traffic consolidation for power optimization of data center networks," *TPDS*, 2015.
[21] K. Zheng *et al.*, "Joint power optimization of data center network and servers with correlation analysis," in *INFOCOM*, 2014.
[22] S. Mitchell *et al.*, "Pulp: a linear programming toolkit for python," *The University of Auckland, Auckland, New Zealand*, vol. 65, 2011.
[23] N. Corporation, *NVIDIA System Management Interface*, https://developer.download.nvidia.com/compute/DCGM/docs/nvidia-smi-367.38.pdf, 2023.
[24] P. Delestrac *et al.*, "Multi-level analysis of gpu utilization in ml training workloads," in *DATE*, 2024.
[25] G. Huang *et al.*, "Densely connected convolutional networks," in *CVPR*, 2017.
[26] C. Szegedy *et al.*, "Inception-v4, inception-resnet and the impact of residual connections on learning," in *AAAI*, 2017.
[27] A. G. Howard *et al.*, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.
[28] C. Szegedy *et al.*, "Going deeper with convolutions," in *CVPR*, 2015.
[29] K. Simonyan *et al.*, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.
[30] Z. Liu *et al.*, "Swin transformer: Hierarchical vision transformer using shifted windows," in *CVF*, 2021.
[31] X. Zheng *et al.*, "Adapting gpt, gpt-2 and bert language models for speech recognition," in *ASRU*, 2021.
[32] A. Dosovitskiy, "An image is worth 16x16 words: Transformers for image recognition at scale," *arXiv preprint arXiv:2010.11929*, 2020.
[33] P. Patel *et al.*, "Characterizing power management opportunities for llms in the cloud," in *ASPLOS*, 2024.
[34] Q. Weng *et al.*, "Beware of fragmentation: Scheduling {GPU-Sharing} workloads with fragmentation gradient descent," in *(ATC)*, 2023.
[35] Amazon, "Amazon," 2024, accessed: 2024-12-18. [Online]. Available: https://www.amazon.com/
[36] G. Yeung *et al.*, "Towards {GPU} utilization prediction for cloud deep learning," in *HotCloud*, 2020.
[37] W. Gao *et al.*, "Chronus: A novel deadline-aware scheduler for deep learning training jobs," in *SoCC*, 2021.
[38] Q. Weng *et al.*, "Mlaas in the wild: Workload analysis and scheduling in large-scale heterogeneous gpu clusters," in *NSDI 2022*, 2022.
[39] J. Guo *et al.*, "Who limits the resource efficiency of my datacenter: An analysis of alibaba datacenter traces," in *IWQoS*, 2019.
[40] Feitelson *et al.*, "Packing schemes for gang scheduling," in *workshop on job scheduling strategies for parallel processing*, 1996.
[41] Z. Han *et al.*, "Scheduling placement-sensitive bsp jobs with inaccurate execution time estimation," in *INFOCOM*, 2020.
[42] W. Xiao *et al.*, "Gandiva: Introspective cluster scheduling for deep learning," in *OSDI*, 2018.
[43] D. Narayanan *et al.*, "Heterogeneity-aware cluster scheduling policies for deep learning workloads," in *OSDI*, 2020.