Cluster Computing (2025)28:557
https://doi.org/10.1007/s10586-025-05216-0

Oltunes: Online learning-based auto-tuning system for DL inference in heterogeneous GPU cluster

Seoyoung Kim[1] · Jiwon Ha[2] · Yoonhee Kim[1]

Received: 27 October 2024 / Revised: 10 January 2025 / Accepted: 14 February 2025

Abstract
With rapid advancements in AI, GPU accelerator technology is evolving, leading to an increase in heterogeneous computing nodes within data centers. This necessitates schedulers that can identify and efficiently manage diverse resources to dynamically meet application demands. For latency-sensitive tasks such as deep learning inference, imprecise GPU scheduling can cause resource interference, degrading both application performance and overall GPU utilization. The rise of NLP and large language models (LLMs) has heightened the focus on balancing throughput and latency. However, dynamic loads on specific resources can lead to performance degradation due to head-of-line blocking. Consequently, proactive resource management is essential to reduce costs while ensuring quality of service (QoS) and maintaining energy efficiency. This paper introduces OLTunes, a cluster-level scheduling system for deep learning inference models that integrates streaming and batch methods to efficiently manage both online and offline models. By leveraging FM-FTML, an online learning technique, OLTunes optimizes runtime environments and resource allocation to meet user SLAs through prediction and optimization. It groups tasks based on their characteristics and model variants to minimize interference, ensuring complementary affinities. It also automatically adjusts resources and configurations to improve performance and reduce resource fragmentation. Performance experiments on a heterogeneous GPU cluster demonstrated a 58% average improvement in GPU utilization, up to 49% reduction in p99 tail latency, and a 61% increase in throughput. It also achieved approximately 84.6% energy savings with a maximum accuracy loss of 4% and reduced latency-sensitive SLO violations by up to 92% compared to other baselines, ensuring end-to-end QoS.

Keywords Heterogeneous GPU Cluster · Online-learning · Machine Learning · Deep Learning Inference · Resource Scheduling · Affinity-aware

1 Introduction

Artificial intelligence (AI) has recently gained significant attention as a result of demonstrating its potential in a variety of applications and the rapid advancement of AI technology. AI is driving innovative changes in traditional problem-solving methodologies across a wide range of industries, including healthcare, finance, manufacturing, and autonomous driving. Moreover, the demand for deep learning (DL) applications is rapidly increasing, driving significant growth in research, development, and various applications across multiple domains As a result, the demand for technologies to develop and optimize deep learning (DL) models is at an all-time high. To address this demand, most companies now commonly establish GPU data centers equipped with heterogeneous computing

✉ Yoonhee Kim
yulan@sookmyung.ac.kr

Seoyoung Kim
seoyoung.mennelet@sookmyung.ac.kr

Jiwon Ha
jwh0245@snu.ac.kr

[1] Department of Computer Science, Sookmyung Women's University, 100 Cheongpa-ro 47-gil, Seoul 04310, South Korea

[2] Department of Computer Science, Seoul National University, 1 Gwanak-ro, Gwanak-gu, Seoul 08826, Seoul, South Korea

Published online: 30 August 2025

resources [1]. In these heterogeneous computing environments, an efficient scheduler is essential for managing computing resources and deep learning tasks, thereby enhancing computational efficiency and hardware utilization.

Deep learning applications generally consist of two stages: training and inference. In the inference stage, deep learning applications are often provided as online real-time services that need to respond quickly to user requests. Online inference tasks predominantly exhibit latency-critical (LC) characteristics, with service users expecting high performance in terms of both response time and inference accuracy. Consequently, if tasks fail to meet the service level agreement (SLA) or demonstrate lower-than-expected accuracy, this situation can significantly undermine service reliability and degrade the user experience. Hence, the inference process must carefully balance latency, accuracy, and cost.

Meanwhile, since the emergence of ChatGPT [39], large language models (LLMs) have gained radical popularity, leading to a surge in demand for LLM inference services. However, these LLM-based inference services are not provided as traditional online inference services sensitive to latency, but rather as offline inference services. Such tasks have the nature of throughput-oriented (TO) work, where optimization focused on throughput is prioritized. These two methods can be selectively used depending on the characteristics and needs of the system, but recently, various services utilizing AI have been appropriately mixing the two inference methods. For example, Netflix uses an online inference method that corresponds to LC tasks for real-time video recommendations, while it employs an offline inference method that corresponds to TO tasks for improving recommendation algorithms or movie classifications. In other words, scheduling and resource management techniques must support both forms of inference, optimizing resources based on their characteristics. To provide optimized inference service performance, it is necessary to carefully adjust H/W(*e.g.,* GPU type, GPU memory, etc.) and S/W parameters (*e.g.,* variant, batch size, etc.). This greatly improves the efficiency of applications and helps reduce data center operating costs. However, having service users set or change runtime parameters themselves is a complex and time-consuming task. Additionally, providing the optimal combination of H/W and S/W parameters requires a significant amount of computation, considering the allocation constraints on a heterogeneous GPU cluster and unexpected events. In particular, it becomes a more challenging task when it comes to providing real-time services.

From the perspective of the data center, it is challenging to unconditionally provide the best resources for all user demands. Thus, in addition to addressing client-side SLA requirements, efficient resource management and load balancing across computing resources play a critical role in ensuring service stability and sustainability. As AI workloads grow in scale and complexity, the demand for better resource utilization intensifies, placing data centers under increasing pressure to optimize operations and reduce costs. Furthermore, to mitigate the growing environmental impact, data centers have been actively pursuing carbon emission reductions [29, 36]. The rapid adoption of AI technologies has led to over a 200% increase in data center workloads in recent years [1, 7], and this trend shows no signs of slowing down. Since carbon emissions are closely tied to computation time and resource usage, balancing the load among computing resources is essential to minimize energy consumption and enhance overall efficiency.

Although various prior studies have been conducted to provide an inference service that guarantees optimal performance for similar objectives [8, 23, 42, 45, 51], most studies have focused on online inference services, emphasizing optimization mainly from the user's perspective. Moreover, few studies have considered optimization from the system or data center perspective, highlighting the need for a more holistic approach that balances user demands with efficient resource management and scalability. In response to the increasing demand for offline inference services, there is a need for a scheduling design that simultaneously analyzes the characteristics of both online and offline inference, while also considering optimizations from the data center perspective, thereby providing a "hybrid" service system. In a data center composed of a heterogeneous cluster, orchestrators such as Kubernetes [4, 50] and Mesos [2] are used, offering a best-effort scheduling method that provides generalized and limited features, but does not offer fine-grained optimization capabilities.

Therefore, this paper proposes an online learning-based scheduling framework for heterogeneous GPU clusters designed for deep learning inference services. The framework offers automated tuning for both client-side and system-wide resource optimization. On the client side, the objective is to deliver an optimized container environment that ensures compliance with the user's Service-Level Agreement (SLA), providing high-performance and cost-efficient services. From the system operator's perspective, the framework aims to minimize interference and fragmentation by incorporating affinity-aware co-scheduling strategies, ultimately reducing operational costs and improving overall resource efficiency.

The main contributions of this paper can be summarized as follows:

- Analysis of the characteristics of two inference tasks: online and offline.

- Provision of auto-tuning for the container runtime environment to meet SLO (Service Level Objective) requirements.
- Design of a scheduling system based on online learning-based prediction for a hybrid (online + offline) inference service.

The primary objective of this paper is to propose a system that enables data center operators to optimize resource utilization while delivering diverse high-performance inference services with assured Quality of Service (QoS).

This paper is structured as follows. Section 2 explains the background knowledge and motivation, presenting various preliminary experiments and their results. Section 3 discusses in detail the structure of the proposed scheduling system, the underlying prediction and tuning model, and the scheduling method. Section 4 discusses the results of a performance evaluation. Section 5 reviews related works. Finally, Sect. 6 discusses the key findings of this work and Sect. 7 presents the conclusions.

2 Background and motivations

To provide scheduling for inference services, it is important to understand the characteristics of inference tasks. First, we will examine the characteristics of the inference model and address related issues.

2.1 Analyzing DL Inference and resource usage

Inference is the process by which a trained machine learning model derives conclusions or predictions from new data. It is primarily applied to real-time data and provided as an online service. Therefore, inference tasks often need to meet strict latency requirements (i.e., SLA) to respond to various queries in real time. Each inference request can be executed individually, balancing the user-defined performance preferences (latency, accuracy) with resource utilization, or processed in batches to handle multiple requests simultaneously. The common characteristics of these inference tasks and the considerations for scheduling them are summarized below.

Constraints of Applications Online inference services generally require strict latency guarantees to preserve user experience and must deliver rapid responses (*e.g.*, generating recommendations within 100 ms to prevent user drop-off) [8]. These tasks primarily involve latency-critical operations, and maintaining high accuracy is equally crucial, and such tasks are referred to as *latency-critical*(LC) tasks. Accuracy is particularly important in sectors such as healthcare and finance, where errors can lead to significant consequences. Thus, minimizing accuracy degradation while achieving rapid response times is essential. Conversely, large language models (LLMs) performing offline inference are typically referred to as batch inference tasks. These tasks must adhere to SLOs within specified timeframes (ranging from several minutes to hours), making throughput the primary concern, and such tasks are described as *throughput-oriented*(TO) tasks.

Resource Utilization Patterns Unlike training tasks, inference tasks generally require sub-second response times and consume fewer resources. Most inference tasks exhibit consistent execution flows and predictable execution times for fixed query inputs. When the input size remains constant, resource usage and execution duration can be anticipated, enabling fine-grained scheduling optimization [55]. However, average GPU resource consumption per inference task is relatively low, contributing to underutilization of GPU resources [45]. In contrast, large language models (LLMs), which have recently garnered significant attention in AI, present different requirements than conventional inference tasks. LLMs exhibit higher computational complexity and consume substantially more memory than other models. As a result, unlike standard inference workloads delivered as real-time online services, LLMs are primarily deployed for offline inference. Offline inference is conducted in batches, differing from online inference that immediately responds to individual requests in real time. Additionally, as real-time responsiveness is less critical, offline inference allows for greater latency flexibility.

Considerations for Resource Sharing Traditional inference services (e.g., image classification and object detection) primarily involve the forward propagation stage and typically consume fewer GPU resources, resulting in low GPU utilization and reduced cost efficiency for inference tasks [22, 57]. The challenge of GPU under-utilization has become increasingly pronounced with rapid H/W advancements [32, 45]. Simultaneously executing *multiple* inference tasks offers significant advantages by minimizing request latency and maximizing resource utilization. However, despite the benefits of increased resource utilization and system throughput, GPU-sharing technologies introduce potential difficulties in meeting latency requirements due to interference and the added complexity of concurrent task execution. Thus, when adopting GPU-sharing techniques, it is essential to account for the affinity and interference among co-executing applications. Several studies have explored improving resource utilization in modern architectures by leveraging temporal resource sharing (*e.g.*, NVIDIA Multi-Process Service (MPS) [34]) and spatial resource sharing (e.g., NVIDIA Multi-Instance GPU (MIG) [33]) technologies [11, 17, 44, 49]. Nevertheless, issues related to resource utilization persist.

Resource Utilization Challenges in LLMs Despite the high GPU and memory demands of large language models (LLMs), low resource utilization remains a persistent issue [16]. Furthermore, sharing tasks that require significant resources is challenging, indicating that resource sharing alone is insufficient to resolve these inefficiencies. As a result, several critical factors must be considered: **(1)** Users or developers conducting inference tasks may misinterpret the application's resource requirements, potentially leading to over-provisioning to meet performance objectives (e.g., latency or throughput). **(2)** In LLMs, input sequence lengths are dynamic and difficult to predict, resulting in inefficient resource allocation and underutilization of reserved resources. Moreover, the memory requirements of batch inference tasks vary significantly with request length, contributing to frequent Out-Of-Memory (OOM) errors. In such cases, task scheduling order becomes crucial, directly impacting overall system throughput [59]. When numerous long requests are prioritized, head-of-line blocking may occur, causing a spike in memory usage. Consequently, other requests may experience delays or remain unprocessed due to insufficient memory. To mitigate this, accurately predicting performance based on request length and assigning appropriate priorities is essential to ensure fair job allocation.

As mentioned earlier, efficient scheduling is required to enable the concurrent execution of multiple tasks on a single GPU. However, distributed deployment across multiple GPUs is also necessary to maximize throughput while adhering to latency constraints. This underscores the need for diverse scheduling strategies tailored to application characteristics and balanced resource provisioning.

Based on the aforementioned characteristics, this paper highlights the following scheduling objectives:

- Balanced task tuning that addresses the trade-offs between latency, throughput, and cost
- Task allocation that ensures SLO compliance (LC vs. TO) while maintaining system utilization balance
- Interference-aware scheduling that considers job characteristics (compute- or memory-intensive)

2.2 Importance and analysis of affinity

As kernels from a single DNN task run sequentially due to data dependencies between them, when one kernel fully utilizes GPU computation or memory bandwidth, it often results in other GPU resources being temporarily underutilized. Therefore, as mentioned in the previous section, we need to increase resource utilization by co-locating kernels. In this case, co-locating the kernels that have complementary resource demands or scale can reduce performance interference between concurrent applications. Although data dependencies constrain overlapping kernel execution within a single DNN task, we can co-locate kernels from different tasks or containers in a single GPU. Thus, when we refer to complementary characteristics as affinity, we consider the affinity between co-running tasks to be an essential element in scheduling. It is common in various studies [3, 20] to meet affinity and reduce interference through complementary allocation based on the resource demands of applications. After considering the robustness, it is common to find the optimal application pairs that minimize interference, as seen in previous studies, by examining the ratio of solo-execution time to colocation-execution time. However, in the inference service, considering the latency-sensitive characteristics, it is inefficient to spend additional time on calculations to provide a slightly more optimal configuration for each task. Therefore, in this study, only complementarity is considered to minimize interference in inference tasks, and the following characteristics are taken into account for this purpose. The affinity between applications can be evaluated from the following two perspectives.

2.2.1 Workload characteristics

In this section, we classify the kernels of the inference model discussed in this paper into compute-intensive(CI), memory-intensive(MI) types based on their characteristics, and present a performance comparison according to these combinations. For example, Convolution 2D, which is primarily used in image models, is the compute-intensive kernel, and Token Generation, which is mainly used in LLMs, is the memory-intensive kernel. Table 1 compares the performance of tasks based on kernel pairs, comparing the sequential execution time, concurrent execution time, and the resulting performance improvement for each kernel pair. In the case of compute-compute kernel pairs, there is almost no performance improvement when executed concurrently compared to sequential execution (0.98 times), which is due to both kernels heavily utilizing computing resources, leading to resource contention. On the other hand, in the case of memory-memory kernel pairs, a performance improvement of 1.10 times is observed during

Table 1 Experiment results of collocating Compute-intensive with Memory-intensive kernels

Kernel Pairs	Sequential	Collocated	SpeedUp
Conv2d-Conv2d	2.3*ms*	2.35*ms*	0.98x
TokenGen-TokenGen	1.6*ms*	1.45*ms*	1.10x
Conv2d-TokenGen	1.9*ms*	1.3*ms*	1.46x

simultaneous execution. The compute-memory kernel pairs demonstrate a 1.46x performance enhancement during concurrent execution, primarily because the two kernels utilize different resources, leading to less resource competition and resulting in a performance advantage.

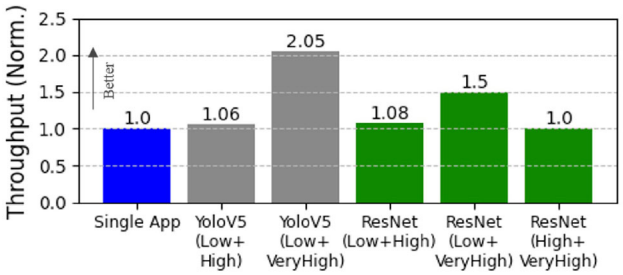
The characteristics of the model are determined through the profiling module discussed in Sect. 3. By utilizing the Nvidia Night Compute profiling tool [35], we calculate the ratio of kernels in the target model where `Compute Throughput` and `Memory Throughput` (or memory bandwidth utilization) each exceed 50%, in order to assess whether the model is Compute-intensive or Memory-intensive. The exact equation is as follows:

$$\frac{\text{kernel\# with Compute throughput } \geq 50}{\text{kernel \# with Memory throughput } \geq 50} \quad (1)$$

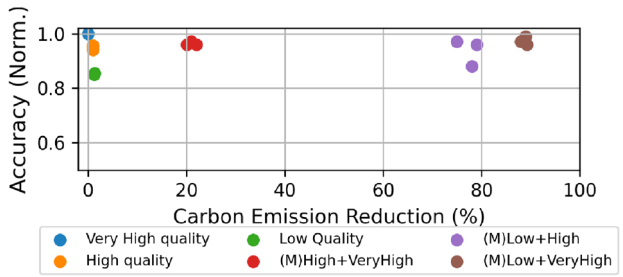
Equation 1 indicates the relative number of compute-intensive kernels and memory-intensive kernels within the application. If the result of Eq. 1 for the application is 1.2 or higher, it is classified as compute-intensive; otherwise, it is classified as memory-intensive. However, if either the numerator or the denominator is 0, only the number of non-zero kernels should be considered.

2.2.2 Model variant

The basic structure of a deep learning model contains several variants, adjusted to fit specific goals or datasets. Although they fundamentally share the same basic structure, they often vary in the number of layers or the number of parameters [20, 21]. There are variants ranging from lightweight ones for real-time inference, such as those used in mobile applications, to large-scale variants. The smaller the variant, the more it can provide real-time services, utilizing fewer parameters and computational resources, which results in lower memory consumption and computing resource usage. However, this comes at the cost of comparatively diminished quality (accuracy) relative to other variants. In the case of large variants, they provide relatively high-quality, that is, highly accurate results, but they require significantly more parameters and computational resources. Such variants already exist for each model (for instance, in the case of ResNet [14], there are lightweight versions like ResNet18 and ResNet34, alongside larger variants such as ResNet50, 101, and 152) and can be developed through hyperparameter tuning. The Clover system proposed by Li et al. [23] demonstrated that mixing low and high-quality variants among the various models maintained high accuracy while also contributing to a reduction in carbon emissions. Based on this idea, this work conducted experiments with variants of opposing qualities on the same inference model to compare their throughput. Figure 1 shows a comparison of performance



(a) Comparison of Normalized Throughput



(b) Accuracy on Carbon Emission Reduction across Quality Combination

Fig. 1 Comparison of Performance and Accuracy, Carbon Emission Reduction between Single Quality and Mixed-qualities

results by generating variants through various combinations of YOLOv5 [18] and ResNet. As a result, the greater the difference in quality between the variants combined, the higher the performance improvement observed (Fig. 1a).

Figure 1b calculates the amount of carbon emissions by referencing the method from the work [23] where indirect estimation based on computing workload is adopted. The carbon emissions were calculated based on the formula: *Carbon Emissions = Energy Consumption * Carbon Emission Factor*, where *Energy Consumption* is defined as the product of GPU utilization, GPU power consumption, and the duration of usage. Hence, the reduction in *carbon emissions* was estimated based on the decrease in resource utilization and execution time. This demonstrates that running the mixed variant has almost no impact on accuracy compared to performing the single variant, while achieving high accuracy and significantly reducing carbon emissions.

Based on these results, this study considers the combination of variants under affinity conditions and applies the combination of tasks with significant quality differences to co-running scheduling for resource sharing, *e.g.,* low and high-quality combination. The above experiment(Fig. 1b) has shown that the greater the difference in quality between the combined variants, the more it helps reduce carbon emissions. However, in this paper, the focus is on applying

the method of mixing variants to task allocation to contribute to such results, rather than addressing the specific degree of carbon emission reduction in detail. Unfortunately, the exact degree of carbon emissions is not covered in this paper.

2.3 H/W and S/W parameter tuning for performance optimization

This section explains the necessity of tuning parameters, such as resource and runtime settings in inference services, and how they affect their performance. It also empirically demonstrates the configurations of related tuning parameters through experiments.

2.3.1 The necessity of tuning

It is common for inference services to run in container form. Therefore, optimizing container-level configuration alone can significantly improve inference performance and also reduce resource provisioning costs. However, most studies focus on tuning the model itself (adjusting parameters, switching variants, or changing frameworks) rather than tuning the container environment [42, 52], providing improvements in inference service performance. Tuning the model itself offers users optimized performance results, but there is a possibility of variants that may not yield the desired outcomes. Moreover, the developer must provide the model variant in the form of a container, which takes additional time to configure the optimal container. Providing an optimized container environment through optimized resource allocation can improve performance by over 10x without significantly affecting the results [51].

It is crucial to consider which parameters are necessary to provide an optimized container environment. Container-level tuning is possible from both H/W and S/W perspectives, and the factors that can directly impact performance are as follows. From a H/W perspective, we consider GPU memory, GPU type, and the usage of Streaming Multiprocessor (SM) including the sharing ratio when sharing is used. From a S/W perspective, we consider the batch size. The number of GPUs can be a consideration, but in the case of inference applications, primarily using one or fewer GPUs is common, except for certain specific domains. Therefore, the number of SMs is taken into account.

2.3.2 H/W tuning

GPU Type With the rapid advancement of technology, various types of GPU accelerators are being developed at a fast pace, each exhibiting strengths and weaknesses in terms of their characteristics. The goals differ depending on the domain of the DL model, so the suitable GPU type varies for each domain. For example, as shown in Fig. 2, the A100 demonstrates the best performance in terms of throughput and latency across all models, with superior memory and frequency. However, as indicated in Table 2, the cost per hour is very high (2x compared to the A30 and 4x compared to the RTX 4090), and the energy efficiency is low (1/3x compared to the maximum value). On the contrary, the RTX 4090 offers a low cost per hour and high energy efficiency, providing reasonable performance, especially for computer vision models. In the case of the A30, it offers lower performance (latency) compared to the two GPUs in terms of speed, but its energy efficiency is good, and it provides MIG, which can increase throughput. Therefore, it is suitable for models like Bert-base [10] when dealing with small-scale datasets, or computer vision models when there is sufficient deadline flexibility. In this way, selecting the optimal GPU type considering the trade-off of *Performance-Cost-Energy efficiency* is essential for users and the underlying resources. Through this heuristic analysis, selecting the optimal GPU type becomes easy. However, in real-time service, the demand for resources, the environment, and the resulting performance continuously change, so predetermined heuristic rules cannot guarantee optimal choices in all situations. Therefore, an online-based analysis and optimization task is needed to select a balanced optimal GPU type for the inference service.

GPU Memory For inference services, GPU memory occupancy methods can be categorized into two types based on the model's characteristics: static and dynamic memory usage. Typically, Vision models, for example, exhibit consistent GPU memory utilization patterns during execution, provided that sufficient memory is allocated to load the model and input images. However, some language models in NLP are likely to require additional memory allocation beyond the initial memory requirements, depending on the parameter size or sequence size during the intermediate processes. Meeting this requirement is essential to achieve optimal throughput results, as shown in Fig. 4a. The amount of GPU memory used can vary depending on the target resource and available GPU memory size. The relationship between GPU memory size and performance improvement is not always linear; instead, performance gains plateau beyond a certain point (knee point), and this threshold varies across different models. Figure 4b shows the performance changes according to the increase in memory by application. As seen in this result, it was found that certain applications do not show performance improvement even when additional memory is provided after a certain level of memory increase. If we provide optimal resources that match the model and its parameters based on the performance

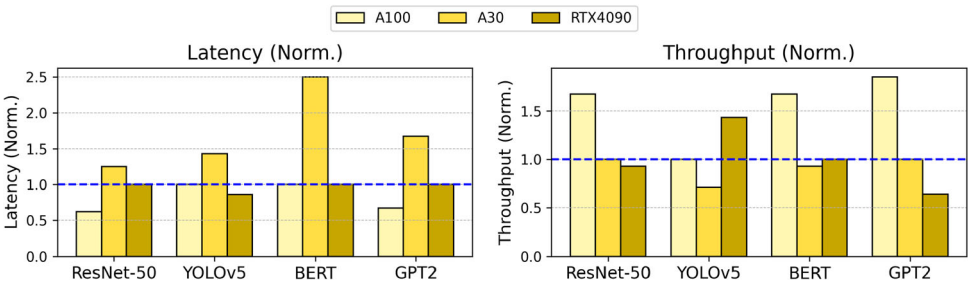
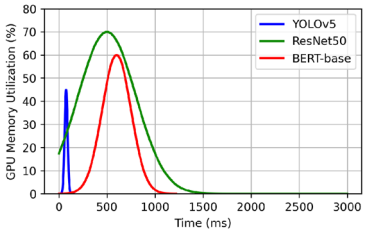


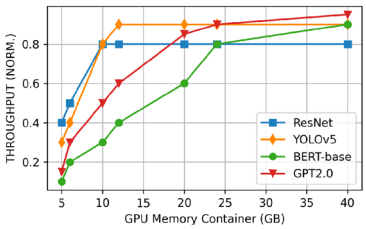
Fig. 2 Performance Comparison by GPU Type for different models: latency(left), throughput (right) (median normalized)

Table 2 Cost/time & Power consumption by GPU type (based on FP32) [37, 38]

GPU Types	$/hour	Energy Efficiency
A100	1.2	19.5TFLOPS/250W = 0.0624
A30	0.6	10.3TFLOPS*/165W =0.04876
RTX4090	0.394	82.6TFLOPS*/450W =0.1836



(a) GPU memory utilization pattern by model



(b) Performance according to GPU Memory size

Fig. 3 Model Performance Comparison by Memory

bottleneck threshold for each application, we can maximize efficiency in terms of both performance and resource utilization.

SM Usage Ratio As discussed in Sect. 2.1, it is common to share GPUs for DL workloads using either spatial or temporal methods. When using temporal sharing technologies (*e.g.*, MPS), a fully isolated environment is not provided, which can lead to certain applications preempting specific resources more than others, depending on the characteristics of the concurrently running applications. For this reason, as discussed in Sect. 2.2, scheduling that considers affinity is performed. However, contention for compute resources cannot be completely avoided even among tasks with high affinity. By determining the ratio of streaming multiprocessors (SMs) allocated for sharing based on the computational load of each task, resources can be utilized more efficiently between tasks compared to when the ratio is not specified [6]. Figure 4 compares the performance of two application tasks with complementary characteristics(high affinity) simultaneously on a single GPU, while varying the MPS resource allocation ratio.

In the case of compute-intensive models like BERT and GPT-2, setting the ratio based on predictions showed a greater performance improvement than the default ratio setting, with a performance difference of up to 50%.

However, in the case of some LLM models, according to [16], extreme SM is used in certain intervals. If the SM usage rate is fixed in this case, it can actually lead to decreased resource utilization efficiency. Therefore, it is necessary to analyze and find the relationship between the performance and the usage rate of SM according to the application, and to apply tuning accordingly.

2.3.3 S/W tuning

Batch Size Batching jobs are an effective method to increase throughput because they allow better utilization of the GPU cores in parallel. However, increasing the batch size is not always advantageous for all models [42]. Figure 5 depicts performance degradation with a large batch size due to their input size exceeding the GPU memory capacity.

Increasing the batch size can improve throughput; however, it also raises the computing load, which may significantly extend batch processing time and substantially increase latency. In Fig. 5b, it can be observed that latency itself increases in proportion to the batch size. Therefore, latency-critical tasks require careful tuning of the batch size within the limits to avoid violating the SLO. Similar to GPU memory, it is crucial to identify an optimal tuning point for the batch size considering the specific characteristics and objectives of the target application, rather than simply increasing the batch size indiscriminately.

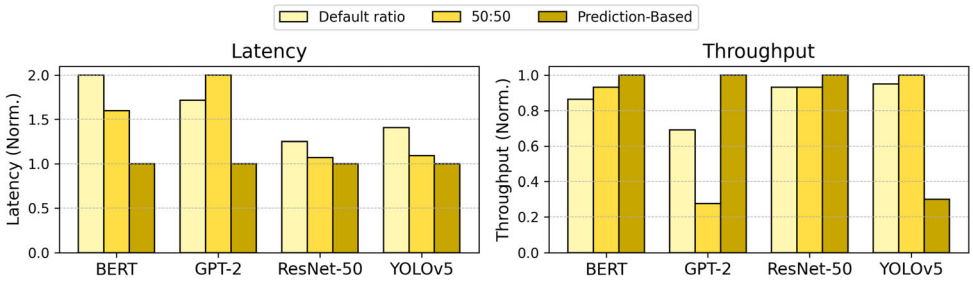
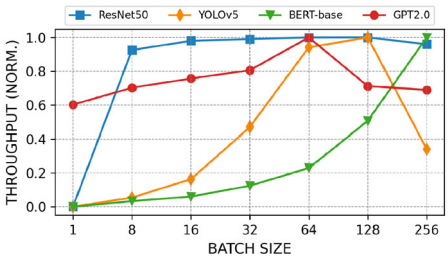
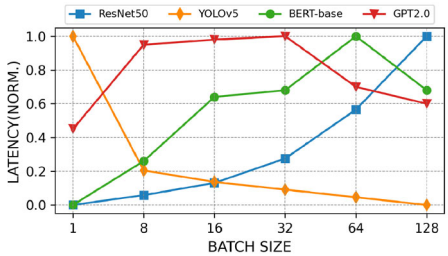


Fig. 4 Performance Comparison between models by MPS ratio: latency (left) and throughput (right) (median normalized)

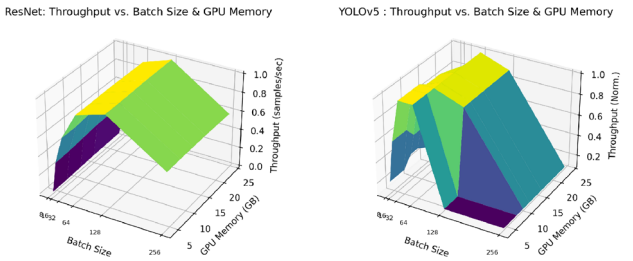


(a) Relationship between Batch size and Throughput



(b) Relationship between Batch size and Latency

Fig. 5 Performance Comparisons by Batch size



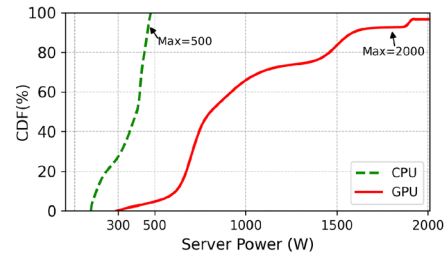
(a) ResNet (b) YOLOv5

Fig. 6 The inter-relationship between the Batch size, GPU memory, and Latency(brighter colors indicate higher throughput)

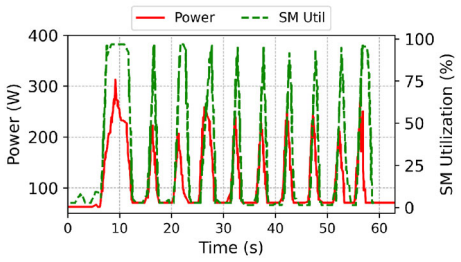
Inter-relationship As demonstrated by the relationship between application performance and the tuning factors mentioned earlier, it was found that simply increasing or parallelizing the available resources does not necessarily guarantee performance improvement. It can rather cause a decrease in performance, and it can also be understood that the knee point differs for each application. Thresholds exist that can cause performance bottlenecks depending on each element, but the thresholds for combined features vary between applications. Figure 6 shows how the performance of domain-specific representative applications ResNet, YOLOv5 varies according to the combinations of H/W and S/W properties mentioned earlier. It shows that not only does performance vary according to each parameter, but the interrelationships between parameters also differ depending on the application. In Fig. 6a, which shows the results executed on ResNet, the batch size steadily increases up to 128, but when it exceeds 128, the throughput decreases. Furthermore, when the batch size is 16 or less, throughput increases with the GPU memory size. However, when the batch size exceeds 32, there is no discernable performance difference attributable to the amount of memory. It is evident that simply increasing the batch size is sufficient to maintain optimal performance. It can be observed that the other two models also show different performance improvement trends. This relationship not only aids in performance prediction but also serves as an important indicator that reveals the optimal tuning point, which contributes to performance improvement under the best conditions. The FM (Factorization Machines) [41] based on this paper can quickly capture the relationships between features and predict performance based on these interrelationships, making the tuning process relatively easy.

2.4 Ecological resource consumption

Recently, due to rapid climate change, ecological computing has emerged as a major issue. Accordingly, data centers are making various efforts to improve energy efficiency and reduce carbon emissions from an environmental perspective [36]. Figure 7a shows the total power consumption of the CPU and GPU [16] across five servers. According to the results, the GPU Node consumes up to four times more power than the CPU. Therefore, it is increasingly important to identify the factors contributing to increased power consumption in GPU servers and to reduce them. Additionally, it can be observed that even in a



(a) Comparison of power consumption between GPU and CPU



(b) Relationship between SM util. and power consumption[30]

Fig. 7 Performance Comparisons by Batch size

completely idle state with a workload of 0, each server consumes 60W, totaling 300W of power. Therefore, it is preferable to distribute tasks overall to minimize idle resources rather than leaving resources unused.

When the amount of computation is appropriate, Dynamic Voltage and Frequency Scaling (DVFS) can achieve optimal efficiency, but maintaining computation for excessively long periods can lead to a decrease in efficiency. If many calculations are performed at once and this process is repeated frequently, it can often lead to overload in the system, resulting in a significant amount of heat generation. As a result, the cooling system also consumes a significant amount of additional power. The relationship between utilization and power consumption for the entire SM can be confirmed through the single node in Fig. 7b. More than 90% of the time, intermittent and a few peak points have little impact on power consumption, but when SM utilization is maintained above 90% for more than 7-10 s, the power consumption curve shows a sharp increase [30]. Since carbon emissions are directly related to power consumption, balanced load balancing among computing resources is essential to reduce them. In other words, it is important to distribute the load evenly so that the computing node does not fall into an idle or overload state. This helps to avoid frequent peak loads and maintain a balanced resource utilization. This can help reduce overall carbon emissions.

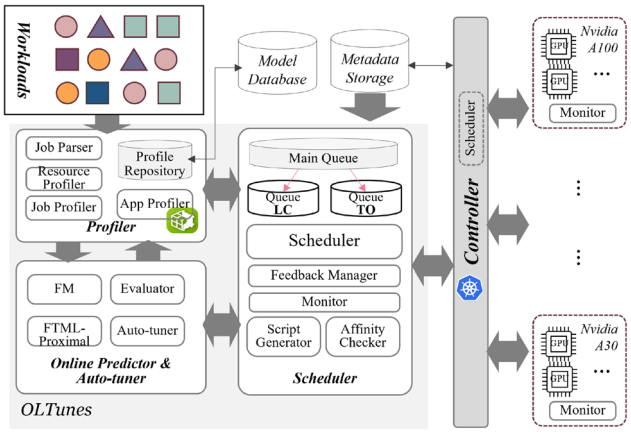


Fig. 8 System Design of *OLTunes*

In the following Sect. 3, we will discuss *OLTunes*, which was designed based on various experiments and findings conducted earlier, in detail.

3 System design

This paper proposes *OLTunes*, an auto-tuning system for machine learning inference services on heterogeneous GPU clusters. This section introduces the main system design of *OLTunes* and provides a detailed explanation of its key components. The online-based prediction method used for the proposed scheduling is also discussed in detail and how it was applied in actual modeling is explained.

3.1 Architecture

Figure 8 shows the overall system design. This system targets heterogeneous GPU clusters orchestrated through Kubernetes [4, 50]. OLTunes is largely composed of three main parts based on functionality: *Profiler*, *Online Predictor & Auto-tuner*, and *Scheduler*. The basic service execution process can be summarized as follows. When the user requests an inference task, the *Scheduler* will receive real-time performance prediction information and optimal container information from the *Online Predictor & Auto-tuner* based on the parsed information and the profiled data. The *Scheduler* makes the final scheduling decisions based on predicted information and the monitoring data from each node, optimizing container settings through Kubernetes' Scheduler Module.

3.2 Profiler

Profiler is composed of *Job Parser*, *Job Profiler*, *Resource Profiler*, and *App Profiler*.

Job Parser parses the requested user tasks and helps quickly utilize the necessary information in other modules. *Job Profiler* analyzes the execution history of previously performed tasks. It collects minimal data on newly registered models by evaluating them on available GPUs and storing the results. This process is conducted offline. *Resource Profiler* manages detailed information about the target GPU or newly registered resources, *e.g.,* GPU type, memory capacity, SM count, Thermal Design Power(TDP), memory bandwidth, L1/L2 cache speed, and operates offline. This information is used for prediction, optimization, and tuning. *App Profiler* performs pre-profiling for the target or newly registered application and analyzes its characteristics (compute/memory intensity). This profiling is conducted offline using the Nvidia Computing Profiler [35]. The characteristics are determined by the formula in Eq. 1, based on the profiling results.

3.3 Online predictor and auto-tuner

The **Online Predictor and Auto-tuner** provide accurate and rapid performance prediction information about available resources for inference tasks. It also provides tuning information required for an optimal runtime environment that meets the performance requirements of the user SLO. It is composed of four modules. First, it transforms the profiling information into feature vectors using the *FM*(Factorization Machine) module, based on the pre-execution information provided by the *Profiler*. This feature vector is combined with the minimal information of incoming new requests. Performance predictions regarding available resources are made by the **FTRL-Proximal** Module, which recommends the optimal runtime configuration through the optimization process. All of these processes will be carried out in real time. The **Auto-tuner** retrieves optimal performance and resource information for turning based on task characteristics and user requirements, ultimately selecting the final node and runtime details. Based on that information, tuning is performed for the task during scheduling, and the appropriate runtime environment information is updated. The **Evaluator** assesses the error rate by comparing the actual performance with the predicted performance after the execution of the task. If the error rate exceeds θ, it quickly performs re-training based on the latest data to update the parameters.

To provide an optimal runtime environment and tuning information for inference tasks, fast and accurate real-time predictions for incoming tasks are required. However, most studies predict performance through offline analysis, which requires large-scale operational data. As a result, if the available data is insufficient, it can significantly impact initial modeling and accuracy. This paper addresses these limitations by introducing online scheduling, which delivers accurate and fast predictions for incoming inference tasks with minimal information.

In the next section, the models utilized for online prediction and tuning are examined in detail.

3.3.1 FM-FTML-based prediction model

The FM-FTML [48] algorithm was used for the online prediction and optimization model. This method combines Factorization Machine (FM) and Follow-The-Regularized-Leader (FTRL)-Proximal. In this paper, FM was used for modeling, and the FTRL-Proximal algorithm was used to learn the modeled data in an online manner.

Factorization Machines (FM) were first introduced by Steffen Rendle [41]. This FM model can estimate all feature interactions, even in cases of extreme data sparsity. FMs possess a versatile nature, allowing them to replicate various factorization models simply through feature engineering with low computing complexity. This method (different from Linear Regression) ensures that all interactions between pairs of features are modeled using factorized interaction parameters.

In particular, the FM method has strengths in modeling high-dimensional interactions. Even if the number of features increases, the computational complexity increases linearly ($O(kn)$, where n is the number of features) because it uses a method of one-dimensional representation. In previous studies, matrix-based feature vectors were primarily used, or tree-based methods such as Random Forest [47] and Gradient Boosting [31] were mainly employed. This makes it inappropriate to provide as an online service, as the computational complexity increases exponentially with the number of features, requiring longer training time. Moreover, FM has the significant advantage of being able to quickly capture complex relationships with minimal data. Using this method, each set of regression tasks is defined to have feature vectors, $X = \{x_1, x_2, \ldots, x_m\}$, and each x_i, for example, can be composed of $x_i = [a_i, b_i, I_{i,\ldots}, g_i, c_i]$ based on the variables in Table 3. The goal is to estimate a function which, when provided with as the input, can correctly predict the corresponding target. For the input vector x_i, FM predicts the result using the following equation 2.

$$\hat{y}(x_i) = w_0 + \sum_{j=1}^{n} w_j x_{i,j} + \sum_{j=1}^{n} \sum_{k=j+1}^{n} x_{i,j} x_{i,k} \sum_{f=1}^{m} v_{j,f} v_{k,f} \qquad (2)$$

where w_0 is global bias, $w \in \mathbb{R}^n$ are the weights for feature vector $(x_i \forall i)$, $\mathbf{V} \in \mathbb{R}^{n \times m}$ is the weight matrix for feature vector combination. m refers the number of dimensions of the factorization (number of data). Therefore, its complexity of computation is $O(nm)$.

Table 3 Example input data

	Candidate Values	Data Type
Application (a)	Models in Table 6	One-Hot Encoding (Categorical Data)
Batch size(b)	1, 4, 8, 16, 32, 64, 128, 256	One-Hot Encoding (Categorical Data)
Input(Seq) size (I)	$0 < i <= 1280$	Continuous Data
Parameter size (p)	$0 < p <= 15B$	Continuous Data
GPU Memory_used ($m1$)	$m = 2k$, $k \in \mathbb{N}$, $0 < m <= 40$	One-Hot Encoding (Categorical Data)
GPU Memory_total ($m2$)	$m = 2k$, $k \in \mathbb{N}$, $0 < m <= 40$	One-Hot Encoding (Categorical Data)
GPU Type (g)	A100, A30, GTX4090	One-Hot Encoding (Categorical Data)
SM (c)	$c = 5k$, $k \in \mathbb{N}$, $1 < c <= 100$	One-Hot Encoding (Categorical Data)

Whether the parameter of the FM model is optimal(accuracy of prediction) is evaluated using a loss function (l). Each parameter is trained by applying the values in the set of observed values, S, to the loss function and minimizing their sum.

For regression, the least-square loss is used, which is expressed as follows:

$$l(\hat{y}(x_i), y_i) = (\hat{y}(x_i) - y_i)^2 \tag{3}$$

This paper performed modeling using the Bi-FM method, referencing the improved approach [56], as prediction and optimization are required for two objectives. Based on the model defined above, it is common to train the data and derive results using methods such as stochastic gradient descent (SGD), alternating least squares (ALS), and Markov Chain Monte Carlo (MCMC). However, these methods require a vast amount of data to be used for training. In general, inference tasks tend to be requested in a streaming manner, requiring immediate job scheduling. In traditional offline learning, FM models need to be retrained every time a new request arrives, resulting in significant computational overhead. This paper utilizes the Follow-The-Regularized-Leader (FTRL) [26] to tackle this challenge, as it is a validated and effective method for online learning in production settings. FTRL employs a distinct learning rate for each feature, utilizing the adaptive learning rate approach that modifies the learning rate according to the magnitude of the preceding gradient. Consequently, training is optimized by minimizing updates for frequently occurring features while enhancing updates for infrequently occurring features. This paper trains the parameters modeled with FM using the FTRL-Proximal method in an online manner. The parameter learning method using FTRL is described below. The overall process is illustrated in Algorithm 1.

It largely consists of three steps: **1)** Gradient accumulation, **2)** Adaptive learning rate, and **3)** Updating parameters (using Proximal term).

In step 1, gradient accumulation, all gradients(vectors representing the slope and direction of the function) up to the current point are accumulated and stored(line 4, z_j is the accumulated gradient, and $g_j(t)$ is the gradient at time t). In step 2, the sum of the squares of the gradients is used to adjust the learning rate for each parameter (line 6, n_j is the squared value of the accumulated gradient for parameter j) When updating the model's weights, the size of each step is determined by the learning rate. If the learning rate is too large, it overshoots the optimal value; if it is too small, the learning process slows down, which also affects prediction speed. In step 3, the Proximal term is used to update each parameter. The update follows the formula in line 8-11. Here, λ_1 and λ_2 are the L1 and L2 regularization

Algorithm 1 FTRL for Factorization Machine (FM)

Input: *Hyperparameters* $\eta, \zeta, \lambda_1, \lambda_2 > 0$
Data: Initialize $\mathbf{z}_j = 0, \mathbf{n}_j = 0, \mathbf{w} = 0$ for all j
1 **for** $t = 1$ **to** T **do**
2 Receive gradient $g_j(t)$ at time t;
3 /* *Gradient Accumulation:* Update accumulated gradient */
4 $z_j \leftarrow z_j + g_j(t)$;
5 /* *Adaptive Learning Rate:* Update adaptive learning rate */
6 $n_j \leftarrow n_j + g_j(t)^2$;
7 /* *Proximal Term:* Update parameter $\mathbf{w}_j$ */
8 **if** $|z_j| \leq \lambda_1$ **then**
9 $w_j \leftarrow 0$;
10 **else**
11 $w_j \leftarrow -\frac{z_j - \mathrm{sgn}(z_j)\lambda_1}{(\zeta + \sqrt{n_j})/\eta + \lambda_2}$;
12 Make prediction with updated $\mathbf{w}_j$;

hyperparameters, respectively. In line 11, each hyperparameter is used for L1 and L2 regularization, η is a parameter for learning rate adjustment, and ζ is a parameter for stability. $sgn(x)$ denotes the *signum* function, which returns -1 if $x<0$, 0 if $x=0$, and 1 if $x>0$.

3.3.2 Prediction modeling

Our solution devises an online approach that requires little training data, works across various scenarios, and is effective. The FTRL-Proximal method is particularly useful when dealing with sparse feature vectors. This is because the model updates the weights for some feature values while keeping others at 0 during the learning process. This is referred to as L1 regularization(λ_1 serves this purpose in Algorithm 1). Therefore, as indicated in the example(Table 3), some tasks do not require features related to a single parameter count, and certain data can focus on the important features of the application for prediction and optimization, even without batch size information. Furthermore, by restricting the magnitude of the weights through L2 regularization(line 11) it prevents the model from excessive complexity and sensitivity to data noise. In other words, it helps ensure that specific weights do not take on excessively large values and that the model generalizes well to new data.

By applying the previously explained FM-FTRL method, we can reveal how each inference task affects latency and throughput under different running conditions and parameters on various resources. This paper assumes that when a new resource or model is registered, at least a small amount of historical execution data must exist for the inference models executed on the target resource. In addition, using this, we train the model through the process described in Sect. 3.3.1. The targets for prediction are latency($\hat{y_i^1}$) and throughput($\hat{y_i^2}$). The feature vector(x_i) used for analysis consists of the characteristics of the application and the characteristics of the resources. Table 3 shows an example of the defined feature vector parameters. The example consists of a total of *8* parameters. First, we define each type of model as $\mathbf{A} = a_1, a_2, ..., a_m$ and consider the characteristics related to the application, including batch size(b), input size (or sequence size) of the task(i), and parameter size(p). In the case of parameter size, LLM inference models related to NLP should be considered additionally because they differ from existing inference models in the field of image recognition, as the input size (or sequence length) and parameter size significantly affect performance. This will be dynamically reflected according to the weight adjustment, which is automatically optimized during learning based on the ratio of tasks more closely associated with their characteristics (regularization) and the importance of those tasks. Characteristics related to resources include the GPU types described in Sect. 2 ($G = g_1, ..., g_z$), the utilization rate of the SM, and the GPU memory (usage and the memory capacity of the target GPU). The memory and SM ratio can be defined as continuous data, but they are defined as sparse categorical data, except for parameters with a very wide range. The reason is that FM-FTRL is more efficient and effective, particularly for sparse data. The range of each parameter is determined based on the MIN and MAX values of the characteristics of the target model or resource. Therefore, the predicted values (latency, throughput) for each task are defined as $\hat{y}^1, \hat{y}^2$. In addition, each value is normalized to be expressed between 0 and 1, which will later undergo a denormalization process.

The characteristics of a resource can include factors such as the number of CUDA or Tensor cores and clock speed, in addition to memory size. This allows for the inclusion of a wider variety of GPUs, thereby enhancing prediction accuracy.

3.3.3 Tuning modeling

Tuning modeling is the process of selecting the optimal performance and resource according to the characteristics and user requirements of the task. The result of this process is specifically used as a reference for selecting GPUs based on resource information and determines whether tuning should be performed on the selected GPU, considering performance and runtime information. When tuning is applied, the runtime environment is updated to reflect the tuning results. Therefore, the first step of this process is to select the best three candidates of $(g, b, c, m, \hat{y_k^1}, \hat{y_k^2})$ combinations(D_t, Eq. 6) such that their results of a single objective function $f(t, g, k)$(Eq. 4) are maximized. In this function, t is the ID of the submitted task being targeted, g is the GPU type that belongs to G, and k is the ID of the variable combination, where $0 \leq k < K$. The variable combinations consist of tunable variables, e.g., batch size(b), used GPU memory(m), SM%(c)), for turning, and the total number of combinations K is the product of the number of each variable, resulting in $K = |b| \times |m| \times |c|$ combinations.

$$f(t,g,k) = \lambda \cdot \frac{1}{\hat{y_k^1}} + (1-\lambda) \cdot \hat{y_k^2} + \alpha \cdot \max(0, T_t - \hat{y_k^1}) + \beta \cdot EPC(g) \quad (4)$$

subject to $\lambda \in [0, 1)$

$$G^{t}_{candidate} = \left\{ top2_{g \in G} f(t, g, k) \right\} \tag{5}$$

$$D_t = \{(g_i, b_k, c_k, m_k, \hat{y}^1_k, \hat{y}^2_k) \mid \tag{6}$$

The process of obtaining these tuning candidates consists of three steps as follows. First, $f(t, g, k)$ (Eq. 4) is calculated for each combination of tunable variables based on the prediction information of the target t. Second, the top 2 GPUs with the highest values are selected from the calculated results(Eq. 5). Finally, tuning variable combinations (D_t) for the selected top 2 GPUs are extracted and stored(Eq. 6).

In the objective function $f(t, g, k)$, the part '$\lambda * \frac{1}{\hat{y}^1_k} + (1-\lambda) * \hat{y}^2_k$' reflects the predicted latency and throughput according to the latency weight depending on the application(job) type. In this case, the value of this part increases as the latency decreases for latency-critical job, and as the throughput increases for throughput-oriented, since the larger λ is, the more it is classified as a latency-critical job (LC, defined in this paper as 0.5 or higher), while jobs below 0.5 are categorized as throughput-oriented. The second part '$\alpha \cdot \max(0, T_t - \hat{y}^1_k)$' aims to provide the importance of the SLO and the urgency of the approaching deadline. Here, T_t represents the user-specified deadline so the urgency is determined by the difference between this deadline and the predicted execution time of the task, and $\hat{y}^1$ denotes the denormalization value. Finally, '$\beta \cdot EPC(g)$' reflects the power consumption degree of the targeted GPU g, and β denotes the overall system load intensity, which is adjusted by the workload. As the workload increases, the value of β also increases. This is because higher workloads lead to greater utilization of system resources, increasing the likelihood of performance degradation or bottlenecks, which can impact overall performance. In this study, instead of using specialized equipment to accurately measure power consumption, the following formula is used to approximate Estimated Power Consumption(EPC).

$$\mathbf{EPC}(\mathbf{g}) \approx P_{base} + (\mathrm{TDP}(g) - P_{base}) \times \text{SM util.} \times \frac{\text{Mem Used}}{\text{Total Mem}(g)} \times \hat{y}^1 \tag{7}$$

In Eq. 7, B_{base} denotes the GPU's baseline power consumption (idle state power), and $TDP(g)$(Thermal Design Power) represents the maximum power consumption of the GPU. $\hat{y}^1_k$ is the predicted execution time of a specific task performed on the GPU. The equation follows the structure of *baseline power* plus *variable power*, forming a realistic power consumption model that estimates power usage based on limited information, such as GPU utilization (SM, memory), without relying on direct measurements [19].

Table 4 Notation

Notation	Description
Q_{lc}	Queue for LC tasks
Q_{to}	Queue for TO tasks
W_{time}	Batch window time
t_{last_batch}	The timestamp of the last batch submission
t_{remain}	The remaining time within the current W_{time}
$batch_{limit}$	The maximum memory capacity available on the selected GPU
$batch_{curr}$	The list of tasks in the current batch
$batch\_size_{max}$	The maximum allowable number of tasks for a single batch
mem_{curr}	The cumulative memory capacity of tasks in the current batch
λ	The ratio that decides whether it is LC($0.5 < \lambda \leq 1$) or TO($0 < \lambda \leq 0.5$)
ρ_{lc}	The upper limit of LC workload ratio in the system, default 0.7
W_{min}	The minimum available value of W_{time}, default $50ms$
W_{max}	The minimum available value of W_{time}, default $1000ms$
w	The adjustment value of W_{time}, default $10ms$

For example, assuming that the GPU_A has a TDP of 300W (16GB gpu memory) and the GPU_B has 200W (12GB), if the performance prediction for Task1 is as follows: (GPU_A, b=8, c=0.7, m1=8GB, m2=16GB, Latency=12 s) and (GPU_B, b=8, c=0.7, m1=6GB, m2=12GB, Latency=19 s), and also assuming that the base power consumption value for all GPUs is 60W, the approximate power consumption of each GPU for Task1 can be calculated as follows(refer to the notation from Table 4);

EPC(A) $\approx 60 + (300\text{-}60) \times 0.7 \times (8/16) \times 12 = 1068$, **EPC(B)** $\approx 60 + (200\text{-}60) \times 0.7 \times (6/12) \times 19 = 991$

Therefore, a higher score is assigned to GPU_B because it was determined that running it at GPU_B is more cost-effective.

In contrast, as the second example, if the performance of Task 2 is (GPU_A, b=8, c=0.2, m1=8GB, m2=16GB, Latency=2 s) and (GPU_B, b=8, c=0.2, m1=6GB, m2=12GB, Latency=3.5s), the approximate power consumption of each GPU for Task2 can be calculated as follows:

EPC(A) $\approx 60 + (300\text{-}60) \times 0.2 \times (8/16) \times 2 = 108$, **EPC(B)** $\approx 60 + (200\text{-}60) \times 0.2 \times (6/12) \times 3.8 = 113.2$

As a result, GPU_A is selected. Therefore, it is tuned to select a relatively cost-effective GPU through the objective function. This result varies depending on the workload intensity and the utilization of the GPU.

Algorithm 2 Schedule Tasks

Input: Q_{lc}, Q_{to}, W_{time}, gpu_{avail}, $batch\_size_{max}$
Output: W_{time}, Q_{lc}, Q_{to}
Function `Schedule()`:
  $t_{last\_batch} \leftarrow$ `current_time()`;
  $batch_{curr} \leftarrow [\,]$;
  $mem_{curr} \leftarrow 0$;
  while *true* **do**
    `heapify`(Q_{lc});
    `heapify`(Q_{to});
    `/* Step 1: Select the GPU with the lowest utilization */`
    $target\_gpu \leftarrow$ `getResources`(gpu_{avail});
    $batch_{limit} \leftarrow target\_gpu[\text{`}mem\_size\text{'}]$;
    `/* Step 2: Process both queues sequentially for batching, ensuring optimal GPU match */`
    for $Q \in [Q_{lc}, Q_{to}]$ **do**
      while Q *is not empty* ***and*** $mem_{curr} < batch_{limit}$ **do**
        (latency, task) $\leftarrow$ `getFirst`(Q);
        $t_{remain} \leftarrow current_{time}$`()` - t_{last_batch};
        if $batch_{limit} - mem_{curr} - task[\text{`}gpu_{mem}\text{'}] > min_{mem}$ ***and*** $t_{remain} < W_{time}$ ***and*** $len(batch_{curr}) < batch\_size_{max}$ **then**
          if *fitForBatch*$(task, target_{gpu}, batch_{curr})$ **then**
            Add $task$ to $batch_{curr}$;
            `pop`(Q);
            $mem_{curr} \leftarrow mem_{curr} + task\_mem$;
    `/* Step 3: Tuning, deploying and feed-backing the batch */`
    if $batch_{curr}$ *is not empty* **then**
      if $mem_{curr} < batch_{limit}$ **then**
        `autotune`($batch_{curr}$);
      $result \leftarrow$ `assignBatchToGPU`($batch_{curr}$, $target_{gpu}$) ;
      if *result* **then**
        `feedback`($batch_{curr}$);
      Reset $batch_{curr}$ and mem_{curr};
      $t_{last\_batch} \leftarrow$ `current_time()` ;
    $W_{time} \leftarrow$ `adjustTimeWindow`(Q_{lc}, Q_{to}, W_{time});
    `update`(Q_{lc}, Q_{to}, W_{time});

3.4 Scheduling

Scheduler module is composed of the *Main Scheduler*, *Monitor*, *Feedback Manager*, *Script Generator*, *Affinity Checker*, and *Task Queues* (Q_{LC},Q_{TO}, main queue).

Once the task receives prediction and tuning information through the two parts described earlier, it accumulates in the *main queue* and is dequeued when the *Scheduler* is called, then inserted into either Q_{LC} or Q_{TO}. For each task, priorities are set based on the prediction results, and communication with the *Monitor* module is established to receive available GPU information. The *Affinity Checker* examines the affinity between tasks within the batch and new tasks based on the characteristic information of the model determined by the *App Profiler* in the **Profiler** module. Ultimately, tasks are scheduled based on the pod generation script created by the *Script generator* for Kubernetes. Detailed process of scheduling will be covered in the upcoming algorithm section. The *Feedback Manager* assists in re-scheduling by adjusting priorities when a deployment of a batch composed of determined resources fails (due to reasons such as resource configuration changes or system overload), and it adjusts the batch allocation frequency based on the failure rate. The *Monitor* is responsible for two roles: resource and task monitoring.

Through Kubernetes, it receives real-time information about available resources and stores GPU utilization data provided by the monitoring daemon running on each node, while also conveying the requested information to the scheduler.

Algorithm 3 Additional Functions

```
1  Function getResources(gpu_avail):
2      /* Get the GPU from the node
          with the lowest utilization
          */
3      target_gpu ← sorted(gpu_avail, key =
         gpu.utilization)[0];
4      if target_gpu is None then
5          target_gpu ←
             getSoonestAvailableGPU();
6      return target_gpu
7  Function fitForBatch(task, target_gpu,
     batch_curr):
8      /* Check if the given task fits to
          the target batch                */
9      return target_gpu['id'] ∈ task['opt_gpus']
10             and isAffinityHigh(task,
           batch_curr)
11 Function
     adjustTimeWindow(Q_lc, Q_to, W_time):
12     /* Dynamically adjust the time
          window based on the load        */
13     lc_ratio ← len(Q_lc) / (len(Q_lc)+len(Q_to));
14     if lc_ratio > ρ_lc then
15         /* Decrease time window, minimum
              W_min ms                    */
16         return max(W_time − w, W_min)
17     else
18         /* Increase time window, maximum
              W_max ms                    */
19         return min(W_time + w, W_max)
20 Function feedback(batch_curr):
21     /* Updating priority of batch_curr
          higher & putting them into Q_lc
          */
22     /* Adjusting W_time lower,
          batch_size_max lower            */
```

3.4.1 Algorithms

Algorithm 2 and show the main scheduling process of OLTunes (refer to Table 4 for notation). This consists of *three* main stages: resource selection, batch creation, tuning/batch scheduling/feedback stage. The basic principle of this scheduling is to create batches based on batch time window(W_{time}) and batch size ($batch_{limit}$). The $batch_{limit}$ (memory capacity of *target_gpu*) is filled by the memory requirements of the tasks that make up this batch, or if the memory capacity is not filled, the configured batch is deployed once the batch time window is reached.

A user request corresponds to one task, which is defined as follows:

$$Task_i = [(\text{model, variant}), \lambda_i, deadline_i, D_i] \tag{8}$$

It includes model name(id) and its variant information(as specified in Table 6), and the values for λ and *deadline*. λ represents the degree of importance for latency and throughput and is decided by the profiler in the following range of the inference classification. For online inference(LC), λ is assumed to take values between 0.5 and 1 (excluding 1), while for offline inference(TO), λ takes values between 0 and 0.5. Depending on the λ, the tasks are classified into Latency-Critical(LC) and Throughput-Oriented(TO) and inserted into the respective job queues (Q_{LC},Q_{TO}). The deadline is the user's SLO, and the tasks must be executed with latency within it. D_i stores the information for tuning, obtained from the results of the *Tuning Modeling* in Sect. 3.3.3 after the prediction.

The tasks that come in real-time are parsed into a format that can be analyzed by the *Job Parser*. Based on that information, the task information($Task_i$) is updated by predicting performance data(D_i) for the existing computational resources, and then the task is inserted to *main queue*. After that, the *scheduler* dequeues the existing tasks from *main queue*, and decides task's priority depending on the urgency(*deadline/avg.latency*, the smaller the value, the higher the priority), then inserts them into task queues(Q_{LC},Q_{TO}). Both queues are then sorted in ascending order through a *PriorityQueue*. In case of tasks with the same priority value, the Shortest Job First principle is followed(line 6-7).

As the first step, the list of available GPU resources is obtained from the monitoring information. In the available GPU list(Algorithm 2, line 9; Algorithm 3, line 1) GPUs that are fully empty and are partitioned(for example, MIG-enbled GPU). In case all GPUs are occupied, a GPU with more than 80% available capacity or a *soon-to-be-ready* GPU, due to the early completion of a task or when the longest task in the batch is expected to be more than 80% complete, will be included even if tasks are currently running. Once the list of available GPUs is completed, the one with the lowest utilization in the list is selected as the *target_gpu*.

Secondly, the batch is formed according to its GPU memory capacity, starting with the LC task(line 12-20). As long as it does not exceed the batch capacity(target

memory limit, $batch_{limit}$) or the current time from the start of scheduling is within the W_{time}, or the number of current tasks in this batch won't exceed $batch\_size_{max}$, the task will be added to the batch.

After that, it checks whether the task is suitable to be included in the batch(line 17). Through the function `fitForBatch()` (Algorithm 3, line 7), it checks whether the GPU included in $target_{gpu}$ is part of the task's D_i (since D_i holds the top-2 GPUs info. with the best performance for the given task) and check affinity (function `isAffinityHigh()`, Algorithm 3 line 10) with the existing tasks if there are already tasks added within the batch. Affinity checking, as mentioned earlier in Sect. 2.2, is determined by considering both the workload characteristics and the variant of the model between tasks within a batch and the new task, and the detailed process of determining affinity is explained in Sect. 3.4.2. Therefore, the task will only be included in the batch if it meets those two conditions(-line 18). Once the selection process for the LC tasks is completed, the same process will be carried out for the TO tasks. After creating the batch, the *Step 3* proceeds, which includes tuning, deployment, and feedback.

The tuning (line 24) is performed when the batch is not completely filled - that is, when there is available space within the target GPU's memory capacity ($batch_{limit}$). In this case, the performance of the task (latency for LC, throughput for TO) is adjusted in the direction that maximizes it, using the setting combination information of D that each task possesses, within the capacity of the available resources. The tuning is carried out starting with tasks related to throughput increase(TO), as it adjusts the batch size. The task to which tuning is applied has its runtime information updated according to the tuning results, and this is reflected during container allocation. The detailed process of auto-tuning will be explained in Sect. 3.4.3. Once tuning is complete, secondly, the batch will be assigned to the $target_{gpu}$. If the $target_{gpu}$ is already allocated due to an unexpected situation or if the allocation fails due to unexpected events such as delays from previous tasks, the corresponding batch is re-inserted into Q_{lc} with a higher priority through the `feedback()` process (Algorithm 2, line 26; Algorithm 3, line 20). In this situation, the *feedback* enhances processing speed by slightly reducing the batch window.

Finally, the time slot is initialized. Based on the system load, the batch window is adjusted, mixing the remaining tasks with incoming tasks for re-scheduling.

3.4.2 Affinity modeling

The function `isAffinityHigh()`(Algorithm 3, line 10) evaluates the suitability of a new task to be included in the target batch by examining the affinity between the new task and the tasks already present in the batch. This determination is based on two aspects: workload characteristics and variant.

First, affinity checking using workload characteristics is performed by applying the following formula to all tasks in the target batch.

$$\boldsymbol{a}(Task_A, Task_B) = 100 \times \frac{1}{1 + e^{-|\Delta_{ratio}|}} \tag{9}$$

$where \Delta_{ratio} = ratio(A) - ratio(B)$

In the above equation 9, *ratio(X)* is the result of the equation 1 for the model used in $Task_X$. A higher *ratio* means the task is highly compute-intensive, while a lower *ratio* means the task is highly memory-intensive. The difference in ratios is calculated by computing the absolute difference between the *ratio*s (Eq 1) of two tasks' applications. A larger difference indicates that one task is more compute-intensive, while the other is more memory-intensive, suggesting complementary resource usage. The affinity score is calculated by applying the *sigmoid* function [53] to Δ_{ratio}, mapping the difference to a score between 0 and 100. The sigmoid function ensures that small differences result in lower scores, while larger differences yield higher scores. Therefore, the final affinity between $Task_{new}$ and the target batch is calculated as the average of the individual affinities with each task in the batch and is expressed as follows(N is the number of tasks in the batch before adding new task).

$$aff_{workload} = \frac{1}{N}\sum_{i=1}^{N} \boldsymbol{a}(Task_{new}, Task_i) \tag{10}$$

For affinity based on model variants, the degree is determined by the combination of the model variants of the tasks in the batch and the new task, following the conditions below.

$$v(Task_{new}, Task_i) = \begin{cases} 100, & high + low \\ 50, & low + low \\ 0, & high + high \end{cases} \tag{11}$$

Once the individual affinities between $Task_{new}$ and all tasks in the target batch are determined, the final $aff_{variant}$ value is derived by calculating the average overall affinity, similar to the workload characteristic-based method in Eq 10. The final affinity of the new task is determined by the average of the two affinity values. The task can be included in the batch only if this value meets or exceeds $threshold_{affinity}$. The function `isAffinityHigh()` returns true in this case. The default value of $threshold_{affinity}$ is 70 and can be adjusted depending on the system load level, but this is not covered in detail in this paper. For this affinity calculation, the workload characteristics of the model are pre-defined

offline, and the maximum allowable batch size ($batch\_size_{max}$) is set to 4. As a result, the computational overhead is negligible.

3.4.3 Auto-tuning

The function `autotune()`(Algorithm 3, line 24) is the step for updating and adjusting variable parameters to maximize throughput in Throughput-Oriented (TO) jobs or to reduce latency in Latency-Critical (LC) jobs. Referring to a set of tuning variable combinations D, as defined in Sect. 3.3.3, tuning is applied only if performance improves based on the targeted GPU's records. If performance decreases or remains unchanged, tuning is not applied. The tuning procedure is as follows.

Step 1: *Detect Performance Changes* - extract the GPU records from D for each task. If there is only one record, tuning is not performed. Extraction is carried out only if there are two or more records. For Throughput-Oriented (TO) jobs, tuning is performed only if the predicted throughput increases when the batch size increases. For Latency-Critical (LC) jobs, tuning is performed only if latency decreases when the batch size increases.

Step 2:*Perform Tuning* - the maximum possible batch size is derived through a prediction module. Tuning is applied only if the memory size of the best-performing record in D at the current batch size is within the given memory capacity (predicted memory + remaining memory).

When tuning is performed, the container environment for the corresponding task is reconfigured and applied. Once tuning is applied to one task, no further tuning is performed on other tasks within the same batch, and the batch is finalized for submission.

3.4.4 Scenario

Figure 9 shows an example scenario of the proposed scheduling. In this example, each task is represented by a circle or ellipse, where LC tasks are shown in white and TO tasks in gray. Compute-Intensive(**CI**) and Memory-Intensive(**MI**) indicate the model characteristics of the task, representing *compute-intensive* and *memory-intensive*, respectively. **V1** and **V2** signify high and low variants. Additionally, **b1** to **b11** represent the configured batches by *scheduler*, while "**Submit** $1 \sim 5$" marks the point at which the final batch, having either reached the batch limit ($batch_{limit}$) or W_{time} and completed tuning, is submitted to the GPU. W'_{time} indicates the adjusted and modified batch window time. The GPU lists on the right side of Fig. 9 represent the target GPUs (the further to the right, the more recent the list).

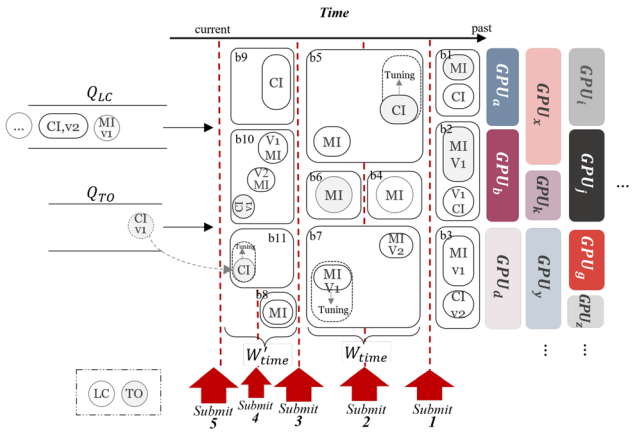


Fig. 9 Scheduling example scenario

There are two periodically inserted queues, Q_{LC} and Q_{TO}, in which each task is assigned a priority and is awaiting resource allocation.

In the case of **b1**-**b3**(*Submit 1*), the batches are configured considering affinity, and all three batches are completed before W_{time} and deployed to the corresponding GPUs. In this case, tuning is not applied due to either insufficient available GPU memory. There are cases, like **b4**, where a task that fits the GPU's capacity is allocated and deployed immediately without adding any further tasks(*Submit 2*).

On the other hand, in the cases of **b5** and **b7**, W_{time} is reached without fully utilizing the capacity of the respective GPUs (GPU_x, GPU_y). As a result, batch formation ends without adding further tasks, and tuning is performed within the available space(*Submit 3*) When TO and LC tasks are mixed, tuning is applied to TO tasks first (**b5**). If the batch consists only of LC tasks, tuning is applied to the tunable tasks (**b7**).

After *Submit 3*, the *scheduler* checks the system load to determine if the ratio of LC tasks exceeds a certain threshold, and W_{time} is adjusted (in this case, reduced to W'_{time}). This adjustment aims to increase the deployment speed of batches to process the tasks waiting in the queue more quickly. In the case of **b9**, due to the reduced window(W'_{time}), the capacity of the corresponding GPU (GPU_i) is not fully utilized. Additionally, the task in the batch cannot be tuned (increasing the model's parameter does not contribute to latency reduction), resulting in the deployment of the batch including a single task. For **b10**, the first two tasks, *task1* (MI+v1) and *task2* (MI+v2), have high affinity, so they are placed in the same batch. When *task3* is added, the overall affinity score exceeds $threshold_{affinity}$, allowing the batch to be formed. Similarly, in **b11**, the shortened W'_{time} results in only one task being included in the batch. Even if the task does not fully utilize the

maximum GPU capacity during tuning, the model's performance knee point is referenced, and parameter(s) is adjusted within the available range to maximize performance.

In this scenario, the *feedback* process is not reflected; however, if there is a failed batch among the deployed batches, it is reallocated with high priority, and W_{time} is readjusted.

4 Evaluation

Evaluation is performed to answer the following key questions:

- How much data was used for analysis compared to other commonly used models, how accurate was it, and how long did it take(overhead)?
- Does it provide reasonable queuing time, latency, and throughput in terms of service?
- Does it meet the user's SLO?
- From a resource perspective, is it considered resource efficiency, and is the load evenly distributed among the nodes?
- Is it energy efficient?

4.1 Experimental setup and testbed

In general, a data center is composed of multiple clusters. However, in this study, experiments were conducted first limited to a single cluster composed of heterogeneous GPU servers to validate the proposed method. The cluster is configured with the nodes specified in Table 5.

We tested the latest DRA (Dynamic Resource Allocation) feature and applied a scheduling technique using CRD (Custom Resource Definitions) with Kubernetes [50] v1.20.0. For analysis and profiling, a separate GPU A100-40 G is used.

The target models and their information are outlined in Table 6. For vision modeling, ResNet-18, 50, and 152 [15], as well as the YOLOv5 [18] are used, while in the NLP field, the LLM models BERT [10] and GPT2 [39] were adopted. The user SLO for the request was based on the latency specified in Table 6. The λ value that distinguishes LC/TO is set to $0.5 < \lambda$ for vision modeling tasks and $\lambda \leq 0.5$ for NLP (LLM) tasks, for 90% of the work. This is because inference tasks generally have latency constraints, but among them, NLP (LLM) often handles large batch datasets that generate multiple sentences from a large text dataset, rather than quickly responding to a single large input. Therefore, it can be considered more throughput-oriented. However, some tasks are sensitive to latency, so it was limited to 10%. The input data parameters and ranges used for performance prediction and analysis are as shown in the above Table 3.

4.1.1 Workloads

The workload was reconstructed in terms of parameters through a job simulator based on the existing service-based request arrival distribution for the targeted model. This performance evaluation has two main purposes for configuring the workload: ***i)*** performance comparison based on

Table 5 System Overview

	Controller		Computing nodes		
		GPU	RTX4090	A100-40 G	A30-24 G
Memory	32GB	**GPU#**	1	5	2
Kubernetes	v1.30.0	**Arch.**	Ada Lovelace	Ampere	Ampere
		VRAM	24GB	40GB	24GB
OS	Ubuntu 20.04.6 LTS	**Memory Type**	GDDR6X	HBM2e	HBM2e
		SM #	128	108	56
		OS	Ubuntu 20.04.6 LTS		
		CUDA Ver.	12.3	11.8	12.3
		TDP (Thermal Design Power)	450W	250 W	165 W
Analytics Node					
	GPU, GPU#	Ampere A100, 1			
	OS	Ubuntu 20.04.6 LTS			

Table 6 Information of the Target Inference Models (Used full-precision for all models)

Size	Type	Model	Variants	Dataset	Metric	Batch Size	Latency SLO	Params#
S	Image Classification	ResNet	18	ImageNet [9]	Accuracy	[1 - 256]	10ms	20 M
M	Object Detection	YOLOv5	5n, 5X	COCO [24]	85% mAP	[1 - 256]	25ms, 45ms	2.6M, 97 M
L	LLM	BERT	Base, medium	SQuADv2 [40]	F1 score	[1 - 128]	150ms	110 M
XL	Image Classification	ResNet	50,152	ImageNet	Accuracy	[1 - 128]	80ms, 170ms	25 M, 60.2M
XXL	LLM	GPT2.0	-	SQuADv2	0.88 F1 Score	[1 - 128]	200ms	1.5B

Table 7 RPS for each model

Model	Uniform	Poisson
ResNet(18)	70	60
YOLOv5	50	40
ResNet(50,152)	40	25
BERT-base	10	5
GPT 2.0	3	2

the model, and ***ii)*** stress testing (performance comparison based on workload patterns). To create a realistic workload for the experiment ***i)***, we selected the average request arrival rates shown in Table 7, which are based on the mean invocation request rates of the top 10 most frequently executed functions in the Microsoft Azure Functions trace [28, 43]. This was referenced from [45]. Additionally, models corresponding to LLM (NLP, TO tasks) are configured by mixing the two workloads using the request patterns and parameters from the 2023 Azure LLM inference trace [27] provided by Microsoft Azure Function, to measure the performance of each model.

For the configuration of the workload for the stress test, Uniform and Poisson request arrival distributions were used. In the case of the uniform distribution, it is a request pattern found in fields such as autonomous driving (obstacle detection). The Poisson distribution is primarily a pattern observed in the application domain of real-time DNN applications (speech recognition). Therefore, we utilize the actual inference service trace from Baidu's Apollo Autonomous Driving System [13] (an open autonomous driving platform) and the provided trace based on the two distributions. Experiments will be conducted with three configurations: Uniform, Poisson, and Real-world trace.

4.1.2 Baseline

This paper compares performance considering the following baseline.

Prediction

- **Matrix Factorization(MF)** [54]: it is a collaborative filtering method that decomposes a large matrix, like user-item interaction data, into two smaller matrices to capture latent features and predict missing values. Commonly used in recommend systems, it helps predict user preferences for items based on interaction data, such as ratings or purchase history. Additionally, it is applied in areas like dimensionality reduction and topic modeling in text mining.
- **Random Forest Regression(RFR)** [47]: Random forest regression is an ensemble learning method that constructs multiple decision trees during training and combines their outputs to improve prediction accuracy, handling complex, non-linear relationships in data. It is usually used for predicting continuous values in applications like finance, healthcare, environmental science, and marketing due to its ability to handle complex, non-linear relationships.

Scheduling

- **Orion** [45]: Resource utilization is maximized by placing best-effort tasks and high-priority tasks together in spatial resource sharing. In job allocation, the affinity between tasks considers the characteristics of the tasks (compute-intensive, memory-intensive). However, since there is no condition regarding GPU selection in Orion's Scheduling Policy, we will select the target GPU using a best-effort approach for comparison with this study.
- **MLaaS** [52] **by Alibaba(Reserving-and-packing):** In the MLaaS system, the reserving-and-packing (R&P) policy is applied for scheduling on a heterogeneous GPU cluster. This method intentionally reserves high-end GPUs (primarily high-performance GPUs with NVLinks) for high-GPU tasks, while packing and allocating other workloads to GPUs that are less powerful or regular in performance. Therefore, tasks focused on performance are assigned to high-performance servers, while tasks that prioritize parallelism over performance are allocated to other servers.

4.1.3 Metrics

The performance of OLTunes is evaluated in three categories. The metrics for each category are as follows.

- **Prediction performance:** prediction latency, error rate (RMSE), accuracy (R-Square), sampling count
- **Service Performance:** p50 latency (*ms*)= avg JCT + avg. queuing time, p99 tail latency(*ms*), Throughput(inputs per sec), Accuracy Loss rate(%), SLO violation rate(%)
- **Cluster-level Performance:** resource utilization(%), power consumption (%)

4.2 Experimental results

4.2.1 Prediction performance

This indicates whether the FM-FTML model has made accurate and fast predictions compared to other models. To compare predictive performance, we examine the relationship between various features in a general manner and compare Matrix Factorization with the FM-FTML method used for predictions. In addition, this study demonstrates the superiority of the methods used by targeting Random Forest Regression, which is frequently employed for prediction in various research studies.

First, we compare three models using approximately 1,400 data to evaluate their error rates, accuracy, and performance. For comparison, the accuracy is evaluated using root mean squared error (RMSE) and R-square, while performance is compared through analysis speed and training time. RMSE is the value obtained by squaring the average difference between the model's predicted and the actual values, and then taking the square root of that result. This allows us to measure the magnitude of prediction errors as an indicator, which allows us to understand on average how much the predictions differ from the actual values, with lower values being preferable. R-squared is a statistical measure that indicates how well a model explains the data. Values closer to 1 indicate better explanatory power, suggesting that the model fits the data well. Table 8 shows the comparison results of the three models. When using the FM-FTRL method, the RMSE shows an improvement of approximately 57% and 40% in R-Square values compared to the Matrix Factorization (MF) and Random Forest (RF) models, respectively (x1.57, x1.41). In the case of R-Square, the FM-FTRL model shows an improvement of x6.68 compared to the MF model and x1.21 compared to the RF model. In terms of accuracy, it is quite similar to the RF model, but shows significantly better accuracy compared to the MF model. In terms of latency and training time, it showed performance of approximately x3.8 and x0.99 compared to MF, with training times being quite similar. Compared to RF, it demonstrated faster performances of x1.4 and x1.5, respectively.

Table 8 Comparison of error rate and performance by model

Model	RMSE	R-Square	Latency(sec)	Train time(sec)
FM-FTRL	0.5681	0.8772	0.0012	0.0634
MF	1.3358	−0.1873	0.0046	0.0628
RFR	0.9655	0.7275	0.0017	0.0952

Figure 10 compares the number of samples required to achieve optimal accuracy in terms of RMSE and R-Square. The reason for setting RMSE=1.3 and R-Squared=0.5 as the optimal accuracy criteria in the experiment is that these values are established based on the scale and range of similar problems and datasets, serving as an acceptable performance level. It is an appropriate benchmark indicating that an acceptable level of predictive accuracy was achieved. The results show that the FM-FTRL model can make predictions with the least amount of data, around 500 samples. This indicates that the system can efficiently adapt to performance prediction even when new models or GPU specifications are introduced. In particular, *the left* graph of Fig. 10 shows that as the data increases, the speed at which accuracy improves is faster compared to the other two models. Additionally, *the right* graph of Fig. 10 shows that the model used in this study provides the highest explanatory power for the variability in the data.

4.2.2 Service (Application) performance

This section compares the performance of the service, specifically from the application perspective, to the baseline research to see the extent of improvement. To answer the question, "*Does it provide reasonable queuing time, latency, and throughput in terms of service*?" as mentioned earlier, performance comparison is performed based on the model and workload.

Performance Comparison between Models: Fig. 11 shows the performance comparison results (p50 latency, p99 tail latency, throughput) based on the five models. In Fig. 11a, the faint bar represents the average queueing time, and the bold color part represents the job completing time (JCT). Overall, MLaaS showed poor performance for lightweight models S and M, while achieving comparatively better performance for the larger models L, XL, and XXL. For the two S and M models, the average latency took 112 times longer for queueing time and 2 times longer

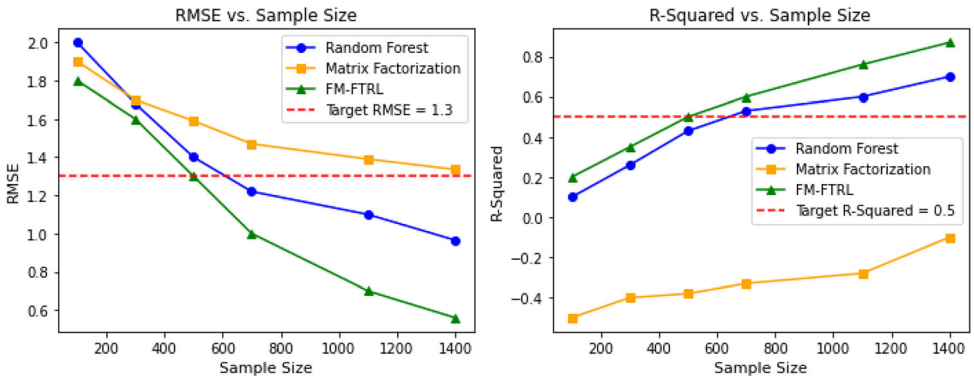
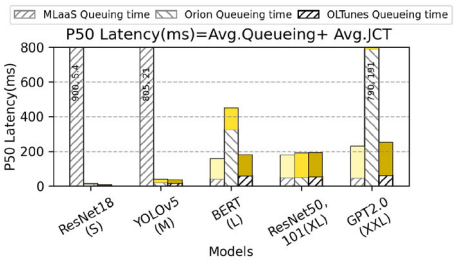
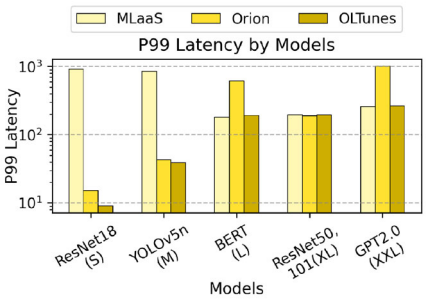


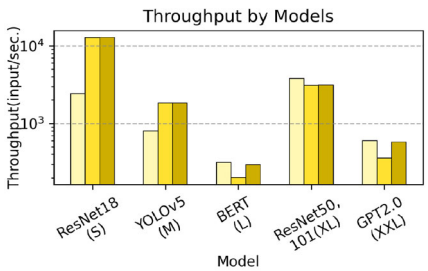
Fig. 10 Number of samples required to reach the optimal accuracy in terms of RMSE, R-Squared (left: RMSE, right: R-Squared)



(a) P50 Latency



(b) p99 tail Latency

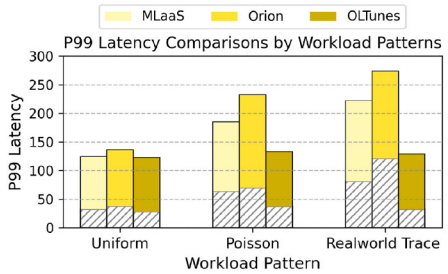


(c) Throughput

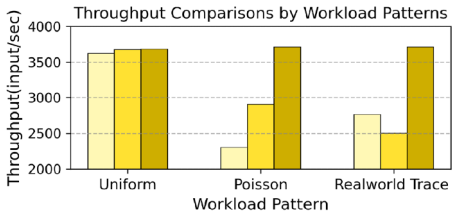
Fig. 11 Comparison of performances by models, (a) Patterned bar is Queuing Time, Colored bar means JCT(Job Completing time)

for JCT compared to OLTunes. The p99 tail latency took about 60 times longer on average. The small model experienced significant queuing delays, primarily due to the relatively low proportion of regular GPUs among the GPUs used in this experiment and the high proportion of small models in the workloads. On the other hand, for larger models (L, XL, XXL), the queueing time was 1.28 times slower compared to OLTunes and the JCT showed almost similar results. In the case of p99 tail latency, MLaaS was approximately 3.14 times faster. This can be attributed to OLTunes' scheduling method, which allocates a mix of relatively high and regular resources based on utilization, in contrast to MLaaS, which primarily allocates high-performance resources for models with high GPU demands. In the case of Orion, OLTunes showed good performance in terms of p50 tail latency for the overall model. In particular, the difference was more pronounced in BERT(L) and GPT2(XXL), which are mainly focused on TO tasks. The reason for the long queueing time is that Orion's scheduling method prioritizes high-priority tasks, primarily focusing on LC tasks, which results in relatively longer queueing times for TO tasks. As a result, the performance of the L and XXL models, which focus on TO tasks, was inferior to that of the S, M, and XL models (vision models) with the high-priority LC tasks. In terms of p99 tail latency, OLtunes was up to 7 times faster than Orion. In terms of throughput, OLTunes shows performance that is 7 times higher than that of MLaaS, which had poor performance with models of sizes S and M, and about 1.8 times lower compared to models of sizes L, XL, and XXL. Compared to Orion, OLTunes achieved an overall higher throughput of more than 1.01x. In particular, it is 1.54 times higher in the L and XXL models. When comparing the performance improvements in latency, it can be observed that there is a smaller performance difference in throughput. This suggests that the throughput of the L and XXL models tends to be lower on average compared to the overall model, which likely contributes to the relatively smaller performance gap.

Performance Comparison by Workload Pattern: the *second experiment* compared the effectiveness of each scheduling method in handling the workload by varying the pattern for the stress test, relative to OLTunes and other competing schemes. Figure 12 compares the performance



(a) p99 Latency Comparison



(b) Throughput Comparison

Fig. 12 Comparisons by Workload Patterns; Patterned bar is Queuing Time, Colored bar means JCT(Job Completing time)

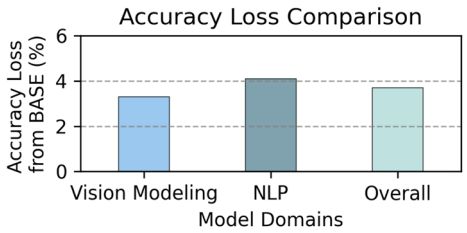


Fig. 13 Accuracy Loss Comparison

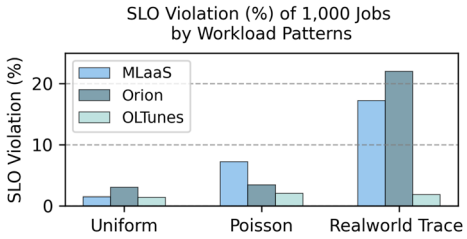


Fig. 14 SLO Violation

(p99 tail latency, throughput) according to three workload patterns(uniform, poisson, real-world trace).

p99 tail latency(Fig. 12a) shows that when the workload is **uniform**, all three methods exhibit similar performance with little difference (within 1.12x). However, in the case of Orion, the JCT was the highest among the three methods in the uniform pattern. This seems to be due to delays caused by performance bottlenecks, as this experiment was set to select GPUs based on best-effort, in contrast to MLaaS and OLTunes, which utilize specific criteria for GPU selection. The difference in throughput is also less than 0.04%. The **poisson pattern** has the characteristic that tasks arrive randomly. Additionally, even if the total number of tasks within a set period is the same, the arrival times can be unpredictable. The *R&P Scheduling* method of *MLaaS* showed relatively poor performance in terms of queueing time compared to JCT. This can be attributed to a surge in latency caused by a lack of resources, as low GPU tasks that were not reserved ended up clustering around limited resources due to irregular task arrivals. Relatively high-throughput low-GPU tasks were delayed, canceled, or re-executed due to deadline violations, resulting in the lowest throughput outcomes. In the case of *Orion*, its focus on high-priority tasks led to increased delays for TO tasks, resulting in the poorest performance. However, in terms of throughput, the average throughput was higher because the focus was on processing LC tasks, which have a relatively high range of throughput values. OLTunes demonstrated stable performance. This adjusts the processing speed of batches according to the type of tasks, which means it is less affected by sudden increases in tasks, thereby ensuring stable outcomes in terms of latency and throughput. In the case of real-world traces, there are sudden bursty and irregular patterns that mix the two previous patterns. The R&P of MLaaS showed a sudden surge in latency when high GPU tasks overwhelmed the reserved resources, leading to an unstable pattern. Throughput also increased when high GPU tasks exhibited a bursty pattern, resulting in a situation where all reserved resources were allocated and waiting times rose. As a result, the peak of the resource occurred frequently, and particularly for High GPU Tasks, there was a low throughput. Orion showed the poorest performance in the real-world pattern. This is because Orion's scheduling algorithm is designed to minimize latency, focusing primarily on high-priority tasks. As a result, when these tasks come in a bursty manner, there is often an indefinite wait for low-priority tasks, leading to improper distribution of tasks and a simultaneous increase in the waiting time (queueing time) for all tasks.

In the case of *OLTunes*, the instability of the task pattern and arrival pattern has led to frequent training on the task history, resulting in a slight increase in latency. However, it showed overall stable performance. OLTunes increases the priority and reallocates tasks that are delayed due to feedback control when the prediction accuracy decreases due to an unstable pattern, preventing immediate deployment of the batch. It appears to have produced stable results because it provides adaptive scheduling based on the workload.

Accuracy and SLO satisfaction: figure 13 shows how effective it is to consider Variants with Affinity when using OLTunes. The base used for comparison is the accuracy when running a single model on 1 GPU. It can be observed

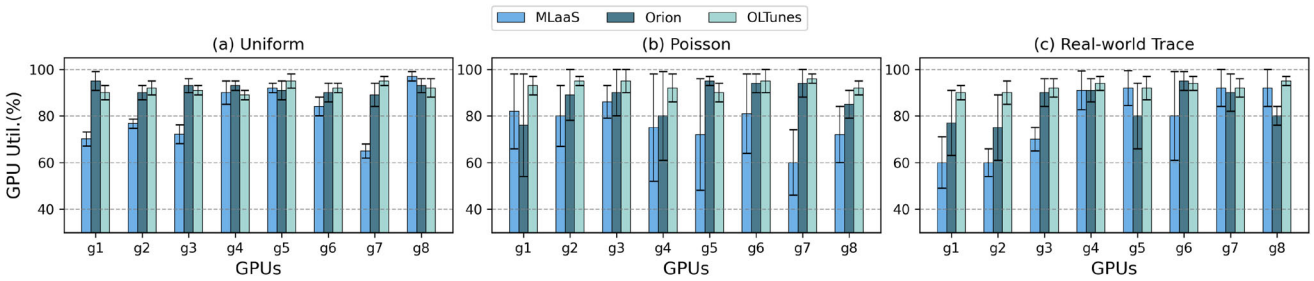


Fig. 15 Comparison of utilization by GPU according to workload pattern

that there is a minimal accuracy degradation of within 4%, indicating a minimal loss of accuracy.

Figure 14 shows the results of how well OLTunes meets user SLOs. When targeting 1k tasks uniformly, both MLaaS and OLTunes exhibit similarly low SLO violation rates. However, in the case of the Poisson Pattern, the scheduling method of MLaaS resulted in an increased violation rate due to increased latency for high-GPU tasks. In real-world trace, a similar result was observed where both tasks violated user SLO due to latency caused by load peaks. The Orion's scheduling method primarily focused on LC scheduling, and since the TO tasks, which had relatively flexible deadlines, were queued, the user SLO violation rate remain low. However, during the real-world trace, the Orion's scheduling method showed a significant increase in violations due to the indefinite waiting of low-priority tasks. In the case of OLTunes, it increased priority based on the proximity of the deadline and predicted latency, while also considering the scheduling speed and tuning degree according to the load, with deadline proximity taken into account during the tuning process. As a result, it mostly satisfied user SLOs across all three workload patterns. The cases where SLOs cannot be guaranteed occur during the initial scheduling phase, when new information is added to the analysis of work history, leading to an increase in prediction errors during the initial process. In this way, it can be seen that OLTunes effectively supports accuracy and deadline satisfaction, which are most closely related to the user's QoS.

4.2.3 Cluster-level Performance

Cluster-level performance analysis demonstrates how efficiently and balanced *OLTunes* consumed resources across heterogeneous GPU resources for various workloads.

GPU Utilization: Fig. 15 compares the utilization of heterogeneous GPUs based on workload patterns with OLTunes and competing methods. In the case of GPU memory, all three studies utilized it to its maximum capacity, making a comparison unfeasible. Therefore, the results presented here focus solely on a comparison of GPU (SM) utilization. G1 and G2-3 are regular GPUs (G1 and G2-3 belong to different nodes), while G4-8 are high-performance GPUs.

Under a uniform pattern load, as shown in Fig. 15(a), all scheduling methods exhibit stable average utilization results. However, MLaaS shows lower utilization on some GPUs compared to Orion and OLTunes. This is likely due to the lack of consideration for the affinity of tasks shared within a single GPU, even though the full capacity of GPU memory is utilized. In Fig. 15(b), under the Poisson pattern, dynamic utilization results are observed. MLaaS exhibits significant variation in GPU utilization overall. It frequently showed a change of up to 2x. Additionally, when a specific task increases, there are moments when the load concentrates on certain GPUs, causing peak loads and overloading the resources. In the case of Orion, GPU utilization was consistently high on average, but some GPUs experienced frequent utilization fluctuations. These fluctuations reached up to approximately 60%. In the case of OLTunes, there was some volatility when the load on the LC sharply increased or when models with insufficient data for training were introduced, requiring re-training. Otherwise, overall, stable results were observed. This is because runtime considerations, resource selection, batch size, and deployment frequency adaptively change according to the load. Lastly, in (c) Real-world Trace, due to the bursty pattern, when the load is not properly managed, a utilization imbalance-where a rich-get-richer and poor-get-poorer phenomenon-may occur. This phenomenon was observed in MLaaS, where compute-intensive tasks became bursty, leading to the monopolization of high-performance GPUs. As a result, these GPUs experience overload, while the remaining regular GPUs remain close to idle. In the case of Orion, since most of the work is LC-related, there were fewer tasks considered as BE, which led to a decrease in utilization. In particular, there was a significant fluctuation in utilization depending on the characteristics of the incoming tasks. In the case of OLTunes, since the tasks are focused on LC work scheduling during specific periods, it has slightly impacted latency. However, the load was evenly distributed across utilization, and the variation in average utilization remains under 10%.

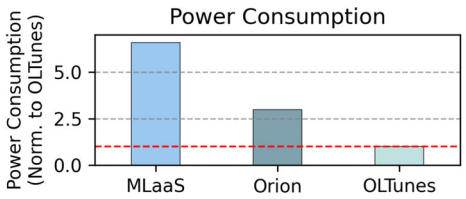


Fig. 16 Comparison of Power Consumption Estimate (Normalized to OLTunes)

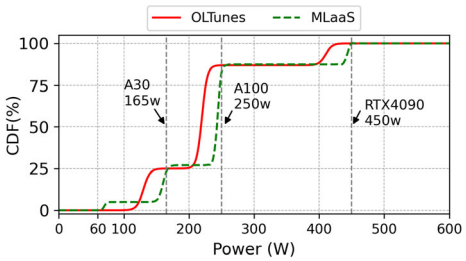


Fig. 17 Power Consumption CDF of OLTunes and MLaaS

Environmental Impacts: In the following, power consumption estimated based on GPU utilization is analyzed and compared. The power consumption in this experiment is an approximation, derived from the Estimated Power Consumption formula outlined in Sect. 3.3.3. This estimation is based on the number of calculations performed by tasks executed every 2 s on each GPU, as well as the corresponding GPU utilization. Figure 16 presents the results of the average comparison.

MLaaS showed about 6.5 times more power consumption compared to OLTunes, while Orion showed about 3 times more. It resulted in relatively high consumption because it primarily assigns long and computation-intensive tasks on high-performance resources. When high GPU tasks surged, peak loads frequently occurred on high-performance resources, and as discussed in Sect. 2, the increase in power consumption due to overload likely resulted in actual power usage being significantly higher. In the case of Orion, the estimated power consumption increased in several instances of overload. OLTunes generally maintains stable GPU utilization and selects resources considering relatively low power consumption among the target resources for task scheduling, which is why it showed somewhat lower results compared to the other two methods.

Figure 17 shows the CDF of the average power consumption calculated for each GPU. It is only compared with the MLaaS that had the highest power consumption results. The base power consumption(P_{base}) of the GPU is 60W, which indicates that the resource is in an idle state. In the case of MLaaS, it can be observed that about 3-5% of the GPUs were in an idle state. When using R&P scheduling, in unstructured workload patterns like real-world traces, situations can arise where tasks suddenly surge and then the arrival intervals widen sharply. In this case, during time slots without tasks, some GPUs may remain idle, or certain tasks (high or low GPU tasks) may surge, leaving GPUs that are not allocated to those tasks in an idle state. The second point to note is that each GPU type (A30, A100, RTX4090) shows a tendency to surge as they approach their respective TDPs of 165W, 250W, and 450W. This means that all heterogeneous GPUs are using their maximum power, which indicates that overloads were frequent. A load with an appropriate number of resources for subordinates and a consistent pattern may have shown more energy efficiency and better performance results.

In contrast, OLTunes tends to have no idle state GPUs and mostly stays in a lower power consumption range rather than at maximum power for extended periods. Therefore, it showed overall optimized and more energy-efficient results. This not only improved energy efficiency but also helped reduce resource management costs.

5 Related works

With the active provision of deep Learning-based applications as online services, there has been a variety of research on management and scheduling targeting large-scale inference workloads in GPU data centers. Resource sharing technologies have advanced not only for inference services but also to enhance GPU efficiency for deep learning applications themselves. Consequently, various studies on scheduling that consider affinity have been conducted to reduce interference among co-running tasks. These studies were compared with the present study in three aspects as follows.

Auto-tuning Inference system For systems that provide inference services, active research has been conducted to reduce the balance between latency and cost, aiming to offer users a cost-effective runtime that guarantees performance. INFaaS [42] generates variant candidates from existing models based on user requirements and other parameters (*e.g.,* batch size, H/W configuration, and H/W-specific parameters). After performing profiling for each variant, INFaaS selects the model variant with the least cost based on resource consumption using a heuristic approach. To meet guaranteed latency requirements and improve cost efficiency for the current load, two-level auto-scaling is implemented. In this study, the primary target was a relatively short execution time and a small-scale image classification model. Additionally, it focused solely on the user's perspective of cost-efficiency and scalability, without considering the aspects of data center resources that require load balancing and cannot provide infinite scaling, such as utilization and system throughput. Wang

et al., proposed auto-tuning inference system Morphling [51] that takes system aspects into account. It explores optimal settings to optimize resource provisioning configurations, ensuring cost-effectiveness while providing performance that guarantees user SLOs. Their study, similar to the present study, combines offline meta-modeling and online few-shot learning methods to provide rapid predictions and tunings. However, this work only considers the impact of GPU time-sharing and fails to consider the affinity or interference level of co-running tasks. Furthermore, despite conducting experiments on a cluster configured with Kubernetes, they could not determine the overall cluster resource utilization as the experiments primarily focused on the cost of exploration and the number of processing requests per second.

Scheduling system on Heterogeneous GPU cluster There are various researches to present a scheduling method in order to perform the DL application on heterogeneous GPU cluster. MLaaS [52] presents an in-depth examination of extensive workload traces from Alibaba and highlights the optimized scheduling method to cover both high-GPU and low-GPU tasks. It employs a simple reserving-and-packing(R&P) to schedule high and low-GPU tasks. It also focuses on the advantages of GPU sharing in real-world GPU data centers. S.J. Subramanya et al. present Sia [46] to schedule adaptive DL jobs on heterogeneous resources to get better cluster performance. In this work, spatial multiplexing is adopted to enhance GPU utilization while maximizing goodput. H. Zhang et al. [58] also present a model-serving system based on online algorithm. It introduces a model serving system with a two-level architecture to ensure reliable goodput even under unpredictable workloads. This is achieved by utilizing preemptions and the batching characteristics specific to each model. In addition, [5, 12, 25] also proposed research to optimize the cluster-level Scheduler for improving work performance and reducing latency on heterogeneous GPU clusters. Most of the research focuses on large-scale training tasks.

Affinity/sharing With the advancement of GPU sharing technology and the active support for its application within Kubernetes, efforts are underway to provide optimal performance in resource sharing at a rapid pace. Strati et al. conducted research to maximize resource utilization by co-locating best-effort tasks and high-priority tasks for spatial resource sharing through Orion [45]. This places the two tasks together in a way that minimizes interference, considering the characteristics of the tasks. However, when focusing solely on inference as in this study, no matter how throughput-oriented it may be, latency considerations cannot be ignored. Therefore, when applying the method proposed by Orion to these tasks, as shown in the previous experiments, there are limitations in improving utilization. Shen et al. [44] proposes a cluster-level optimization system for DNN-based video analysis in GPU dat acenters. Their study utilizes a heuristic approach to select requests that will co-run on the same GPU and identifies the optimal batch size that meets the SLO, while exploring a best-fit combination of tasks that satisfies latency requirements and maximizes utilization. As their study is focused on the model for video analysis, it does not consider the characteristics of inference models like the rapidly increasing LLMs or heterogeneous GPUs.

6 Discussion

Dynamic MIG Partitioning The scheduling of OLTunes adjusts the MIG partitions of the target GPU in advance through the MIG configuration manager when configuring batches with GPUs that support MIG, provided that there is available GPU memory remaining above a certain threshold even after auto-tuning. After changing the settings, include the available resource list for the next batch in the empty partition. The MIG settings can be dynamically changed as needed to further enhance GPU utilization. However, this paper did not address the dynamic configuration in detail. Future work will focus on the dynamic configuration of the optimized MIG.

Various features consideration for Auto-tuning This paper considers only the batch size for tuning S/W feature. By adjusting additional features such as precision or the number of layers, it is possible to provide tuning that aligns with the user's SLO. Future work will aim to offer more refined tuning by considering additional S/W features.

7 Conclusion

This paper proposed OLTunes, a resource scheduling framework for the efficient management of deep learning inference tasks in a heterogeneous GPU cluster environment. OLTunes has been designed with a hybrid scheduling method that combines the characteristics of both online and offline inference tasks, meeting the requirements necessary for each task. Through this, an optimal resource utilization plan that satisfies both the fast responsiveness of latency-critical tasks and the high throughput of throughput-oriented tasks has been presented. OLTunes minimized resource fragmentation and reduced interference between tasks through auto-tuning and dynamic scheduling, ensuring efficient resource allocation. In particular, by utilizing FM-FTML-based online learning to optimize predictive performance, this paper has derived the optimal application environment and resource selection that meets SLA.

Through performance evaluation, OLTunes achieved the following results compared to existing resource management systems under various realistic inference load conditions:

- It demonstrated the potential to enhance overall GPU utilization by up to 58% on average, contributing to the resource efficiency in heterogeneous GPUs.
- It improved p99 tail latency and average job completion time (JCT) by up to 49% and 7.17% respectively, enhancing the response performance of latency-sensitive tasks.
- On average, OLTunes achieved improvements of 29.5% in latency and 29% in throughput across various workloads, while also reducing operational costs and energy consumption through balanced GPU utilization.
- It ensured QoS by reducing SLO violations by up to 92% compared to recent studies.

This paper proposed a new approach for the efficient management of deep learning inference tasks in heterogeneous cluster environments, demonstrating the potential to maximize operational efficiency in data centers and provide high-performance inference services that meet the diverse needs of users. In the future, we will further expand the application scope of OLTunes based on various real-time and non-real-time task scenarios, and conduct performance evaluations targeting a wider range of the latest GPUs to validate the scheduling method proposed in this study. In this way, we would like to contribute to the sustainable development of AI services and the improvement of operational efficiency in data centers.

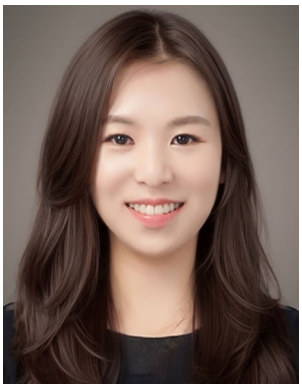
Acknowledgements This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No.2021R1A2C1003379). We appreciate the high-performance GPU computing support of HPC-AI Open Infrastructure via GIST SCENT.

References

1. *AI and the Data Center: Challenges and Investment Strategies*. [Accessed Sept 2024]. https://www.informationweek.com/it-infrastructure/ai-and-the-data-center-challenges-and-investment-strategies
2. *Apache Mesos*. [Accessed Sept 2024]. https://mesos.apache.org/
3. Arima, Eishi, et al.: "Optimizing Hardware Resource Partitioning and Job Allocations on Modern GPUs under Power Caps". In: *Workshop Proceedings of the 51st International Conference on Parallel Processing*. pp. 1–10 (2022). https://doi.org/10.1145/3545008.3545017
4. Burns, Brendan, et al.: "Borg, Omega, and Kubernetes". In: *ACM Symposium on Operating Systems Principles (SOSP)*. ACM, pp. 70–87 (2016). https://doi.org/10.1145/2986772.2986816
5. Chaudhary, Shubham, et al.: "Balancing Efficiency and Fairness in Heterogeneous GPU Clusters for Deep Learning". In: *Proceedings of the Fifteenth European Conference on Computer Systems*. pp. 1–16 (2020). https://doi.org/10.1145/3342195.3387521
6. Choi, Seungbeom, et al.: "Multi-model machine learning inference serving with gpu spatial partitioning". In: arXiv preprint arXiv:2109.01611 (2021)
7. Corcoran, Peter, Andrae, Anders. *Emerging Trends in Electricity Consumption for Consumer ICT*. Tech. rep. National University of Ireland, Galway (2013). https://researchrepository.universityofgalway.ie/handle/10379/3563
8. Crankshaw, Daniel, et al.: "Clipper: A Low-Latency Online Prediction Serving System". In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. pp. 613–627 (2017)
9. Deng, Jia, et al.: "ImageNet: A Large-Scale Hierarchical Image Database". In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE. 248–255 (2009). https://doi.org/10.1109/CVPR.2009.5206848
10. Devlin, Jacob: "BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding". In: arXiv preprint arXiv:1810.04805 (2018)
11. Dhakal, Aditya, Kulkarni, Sameer G., Ramakrishnan, K.K.: "Gslice: Controlled Spatial Sharing of GPUs for a Scalable Inference Platform". In: *Proceedings of the 11th ACM Symposium on Cloud Computing*. 492–506 (2020). https://doi.org/10.1145/3419111.3421284
12. Gu, Juncheng, et al.: "Tiresias: A GPU Cluster Manager for Distributed Deep Learning". In: *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* 485–500 (2019). https://doi.org/10.5555/3311880.3311921
13. Han, Mingcong, et al.: "Microsecond-Scale Preemption for Concurrent GPU-Accelerated DNN Inferences". In: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association, pp. 539–558 (2022). isbn: 978-1-939133-28-1. https://www.usenix.org/conference/osdi22/presentation/han
14. He, Kaiming, et al.: "Deep Residual Learning for Image Recognition". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. pp. 770–778 (2016). https://doi.org/10.1109/CVPR.2016.90
15. He, Kaiming, et al.: "Deep Residual Learning for Image Recognition". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 770–778 (2016). https://doi.org/10.1109/CVPR.2016.90
16. Hu, Qinghao, et al.: "Characterization of Large Language Model Development in the Datacenter". In: *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. pp. 709–729 (2024)
17. Jain, Paras, et al.: "Dynamic Space-Time Scheduling for GPU Inference". In: arXiv preprint arXiv:1901.00041, pp. 1–8 (2018)
18. Jocher, Glenn: *YOLOv5 by Ultralytics (Version 7.0)*. Computer software. (2020). https://doi.org/10.5281/zenodo.3908559
19. Kasichayanula, Kiran, Symposium on Application Accelerators in High Performance Computing: Symposium on Application Accelerators in High Performance Computing: "Power aware computing on GPUs ' '. et al. IEEE. **2012**, 64–73 (2012). https://doi.org/10.1109/SAAHPC.2012.16
20. Kim, Sejin, Kim, Yoonhee: "Co-scheML: Interference-Aware Container Co-Scheduling Scheme Using Machine Learning Application Profiles for GPU Clusters". In: *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE. pp. 104–108 (2020). https://doi.org/10.1109/CLUSTER49012.2020.00020
21. Lan, Z.: "Albert: A Lite BERT for Self-Supervised Learning of Language Representations". In: arXiv preprint arXiv:1909.11942 (2019)

22. LeMay, Matthew, Li, Shijian, Guo, Tian: "Perseus: Characterizing Performance and Cost of Multi-Tenant Serving for CNN Models". In: *2020 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE. 66–72 (2020). https://doi.org/10.1109/IC2E48712.2020.00014
23. Li, Baolin, et al.: "Clover: Toward Sustainable AI with Carbon-Aware Machine Learning Inference Service". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15 (2023). https://doi.org/10.1145/3581784.3607034
24. Lin, Tsung-Yi, et al.: "Microsoft COCO: Common objects in context". In: *European conference on computer vision*. Springer, 740–755 (2014). https://doi.org/10.1007/978-3-319-10602-1_48
25. Mahajan, Kshiteej, et al.: "Themis: Fair and Efficient GPU Cluster Scheduling". In: *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 289–304 (2020)
26. McMahan, Brendan: "Follow-the-Regularized-Leader and Mirror Descent: Equivalence Theorems and l1 Regularization". In: *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. JMLR Workshop and Conference Proceedings. 525–533 (2011). https://doi.org/10.48550/arXiv.1010.1163
27. Microsoft. *Azure LLM Inference Trace 2023: Dataset for Large Language Model Inference, 2023*. Accessed on [Sep 2024]. https://github.com/Azure/AzurePublicDataset/blob/master/AzureLLMInferenceDataset2023.md
28. Microsoft. *Azure Public Dataset: Open Datasets for Machine Learning*. Accessed on [Sep 2024]. 2020. https://github.com/Azure/AzurePublicDataset
29. Mittal, S., Vetter, J.S.: A Survey of Methods for Analyzing and Improving GPU Energy Efficiency. ACM Comput. Surv. **47.2**, 1–23 (2014). https://doi.org/10.1145/2636342
30. Nabavinejad, Seyed Morteza, Reda, Sherief, Ebrahimi, Masoumeh: "BatchSizer: Power-Performance Trade-Off for DNN Inference". In: *Proceedings of the 26th Asia and South Pacific Design Automation Conference*. 819–824 (2021). https://doi.org/10.1145/3394885.3431535
31. Natekin, Alexey, Knoll, Alois: Gradient Boosting Machines, a Tutorial. Front. Neurorobot. **7**, 21 (2013). https://doi.org/10.3389/fnbot.2013.00021
32. *NVIDIA Hopper, Ampere GPUs Sweep Benchmarks in AI Training (2022)*. [Accessed Sept 2024]. https://blogs.nvidia.com/blog/2022/11/09/mlperf-ai-traininghpc-hopper/
33. *Nvidia Multi-Instance GPU (MIG) User Guide, 2021*. [Accessed Sept 2024]. https://docs.nvidia.com/datacenter/tesla/mig-user-guide/
34. *Nvidia Multi-Process Service (MPS), 2021*. [Accessed Sept 2024]. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf
35. *NVIDIA Nsight Compute*. Online; accessed 22 August 2024. https://developer.nvidia.com/nsight-compute
36. The Washington Post. *World is on Brink of Catastrophic Warming, U.N. Climate Change Report Says, 2023*. [Accessed Sept 2024]. https://www.washingtonpost.com/climateenvironment/2023/03/20/climate-change-ipcc-report-15/
37. *Price per Time for GPU*. [Accessed Sept 2024]. https://salad.com/pricing
38. *Price per Time of Ampere GPU*. [Accessed Sept 2024]. https://massedcompute.com/faq-answers/?question=Can+you+provide+information+on+the+pricing+models+for+NVIDIA+RTX+A30+and+A100+GPUs+in+cloud+environments%2C+including+any+discounts+or+promotions+that+may+be+available%3F
39. Radford, Alec, et al.: Language Models are Unsupervised Multitask Learners. OpenAI Blog **1**(8), 9 (2019)
40. Rajpurkar, Pranav, Jia, Robin, Liang, Percy: "Know What You Don't Know: Unanswerable Questions for SQuAD". In: arXiv preprint arXiv:1806.03822 (2018)
41. Rendle, Steffen: "Factorization Machines". In: *2010 IEEE International Conference on Data Mining*. IEEE, pp. 995–1000 (2010). https://doi.org/10.1109/ICDM.2010.127
42. Romero, Francisco, et al.: "INFaaS: Automated Model-Less Inference Serving". In: *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pp. 397–411 (2021)
43. Shahrad, Mohammad, et al.: "Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider". In: *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pp. 205–218 (2020). https://doi.org/10.48550/arXiv.2003.03423
44. Shen, Haichen, et al.: "Nexus: A GPU Cluster Engine for Accelerating DNN-Based Video Analysis". In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 322–337 (2019). https://doi.org/10.1145/3341301.3359658
45. Strati, Foteini, Ma, Xianzhe, Klimovic, Ana: "Orion: Interference-Aware, Fine-Grained GPU Sharing for ML Applications". In: *Proceedings of the Nineteenth European Conference on Computer Systems*, 1075–1092 (2024). https://doi.org/10.1145/3627703.3629578
46. Subramanya, Suhas Jayaram, et al.: "Sia: Heterogeneity-Aware, Goodput-Optimized ML-Cluster Scheduling". In: *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 642–657 (2023)
47. Svetnik, Vladimir, et al.: Random Forest: A Classification and Regression Tool for Compound Classification and QSAR Modeling. J. Chem. Inf. Comput. Sci. **43**(6), 1947–1958 (2003). https://doi.org/10.1021/ci034160g
48. Anh-Phuong Ta. "Factorization Machines with Follow-the-Regularized-Leader for CTR Prediction in Display Advertising". In: *2015 IEEE International Conference on Big Data (Big Data)*. IEEE, 2889–2891 (2015). https://doi.org/10.1109/BigData.2015.7364082
49. Tan, Cheng, et al.: "Serving DNN Models with Multi-Instance GPUs: A Case of the Reconfigurable Machine Scheduling Problem". In: arXiv preprint arXiv:2109.11067 (2021)
50. The Kubernetes Authors. *Kubernetes: Production-Grade Container Orchestration*. Version 1.0, released in 2015. Ongoing development, latest version available online. (2014). https://kubernetes.io
51. Wang, Luping, et al.: "Morphling: Fast, Near-Optimal Auto-Configuration for Cloud-Native Model Serving". In: *Proceedings of the ACM Symposium on Cloud Computing*, 639–653 (2021). https://doi.org/10.1145/3472883.3486987
52. Weng, Qizhen, et al.: "MLaaS in the Wild: Workload Analysis and Scheduling in Large-Scale Heterogeneous GPU Clusters". In: *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 945–960 (2022). https://doi.org/10.5555/3530524.3530605
53. Wikipedia contributors. *Sigmoid function — Wikipedia, The Free Encyclopedia*. [Online; accessed 3-January-2025]. 2024. https://en.wikipedia.org/w/index.php?title=Sigmoid_function&oldid=1261826297
54. Xue, Hong-Jian, et al.: "Deep Matrix Factorization Models for Recommender Systems". In: *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence (IJCAI-17)*, 3203–3209 (2017). https://doi.org/10.24963/ijcai.2017/447
55. Ye, Zhisheng, et al.: Deep Learning Workload Scheduling in GPU Datacenters: A Survey. ACM Computing Surveys **56.6**, 1–38 . https://doi.org/10.1145/3638757
56. Yıldırım, Ezgi: Azad, Payam, ÖÄÿüdücü, Şule Gündüz: "biDeepFM: A Multi-Objective Deep Factorization Machine for Reciprocal Recommendation." Engineering Science and

Technology, an International Journal **24**(6): 1467–1477 (2021). https://doi.org/10.1016/j.jestch.2021.02.008
57. Zhang, Chengliang, et al.: Enabling Cost-Effective, SLO-Aware Machine Learning Inference Serving on Public Cloud. IEEE Transac. Cloud Comput. **10.3**, 1765–1779 (2020). https://doi.org/10.1109/TCC.2020.3008910
58. Zhang, Hong, et al.: "SHEPHERD: Serving DNNs in the Wild". In: *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pp. 787–808 (2023). https://doi.org/10.1145/3600006.3613175
59. Zhou, Zixuan, et al.: "A Survey on Efficient Inference for Large Language Models". In: arXiv preprint arXiv:2404.14294 (2024)



Seoyoung Kim is currently a researcher in the Department of Computer Science at Sookmyung Women's University, South Korea. She received her M.S. and B.S. degrees in Computer Science from Sookmyung Women's University in 2010 and 2012, respectively. She has extensive experience in developing high-performance computing platforms and cloud-based job scheduling systems, having worked at Huawei France and as a researcher at the Korea Institute of Science and Technology Information (KISTI). She is also involved in teaching courses on container technologies and Kubernetes for undergraduate and graduate students. Her research interests include resource scheduling and performance tuning for heterogeneous GPU sharing in deep learning applications.

Jiwon Ha is currently a Ph.D. student in the Department of Computer Engineering at Seoul National University, South Korea. She received her B.S. degree in Computer Science from Korea University in 2023. Her research interests include GPU scheduling, deep learning, and distributed systems.

Yoonhee Kim is currently a professor in the Department of Software at Sookmyung Women's University, South Korea. She received her B.S. degree in Computer Science from Sookmyung Women's University in 1991, and her M.S. and Ph.D. degrees in Computer Science from Syracuse University in 1996 and 2000, respectively. She worked as a researcher at the Electronics and Telecommunications Research Institute (ETRI) from 1991 to 1994. She then served as an assistant professor in the Department of Computer Engineering at the Rochester Institute of Technology from 2000 to 2001. From 2001 to 2016, she was a professor in the Department of Computer Science at Sookmyung Women's University, and since 2017, she has been a professor in the Department of Software at the same institution. Her research interests include cloud systems and workflow management.