# Efficient GPU Resource Pooling for Machine Learning Workloads

Adeesha Jayasinghe, Kavindu Janith, Ravindu Wickramage, Dr. Adeesha Wijayasiri
University of Moratuwa
Moratuwa, Sri Lanka
{adeeshaj.20, kavinduj.20, ravinduw.20, adeeshaw}@cse.mrt.ac.lk

***Abstract*—General Purpose Computing on GPU (GPGPU) has become popular in the recent decade as it enabled significant faster computation speeds compared to traditional CPU based computing. Tasks like training machine learning models can be accelerated significantly by using GPUs. Current commercial cloud computing resources with GPUs incur considerably high cost for users and there is no efficient mechanism for personal GPU holders to rent their GPUs for GPGPU tasks. This paper presents a GPU resource pooling system, designed to address the inefficiencies in current GPU allocation mechanism which leads to resource underutilization and increased costs, by enabling dynamic sharing of idle GPUs for executing one-to-one and distributed machine learning tasks. The system operates on a pay-as-you-go cost model, optimizing GPU usage and ensuring cost efficiency. Through a dynamic scheduling algorithm, the system intelligently allocates GPU resources, enhancing workload distribution and reducing execution times. Built with scalability in mind, the system's architecture can be extended to support various general-purpose computing tasks in the future. Our evaluations demonstrate significant improvements in resource utilization, making this solution a practical and cost-effective choice for decentralized GPU computing.**

***Index Terms*—GPGPU, GPU Resource pooling, Distributed Machine Learning, Dynamic GPU Scheduling**

## I. Introduction

Graphics Processing Units (GPUs) have become crucial in computer-intensive and data parallelism tasks due to their high parallel processing capabilities. The increasing demand for GPU computing has led to challenges in resource management, especially in shared environments such as cloud environments. There is a challenge in finding reliable, high-performance GPU machines at a fair cost. Also, there are many underutilized GPUs of personal computers all over the world. Conventional GPU allocation methods often lead to inefficient resource utilization due to static partitioning. This paper introduces a GPU resource pooling system that facilitates a pool to rent GPUs of GPU owners, and as a system where users can use to do their tasks by distributing the workload. This system is mainly focused on training machine learning models.

Several previous studies have explored remote GPU utilization, focusing on decoupling GPUs from host machines and enabling efficient resource sharing. Early research, such as Kotani et al. [1], explored grid environments to allocate idle GPUs for computational tasks. However, it highlighted CPU bottlenecks in single-core systems and was limited to specific Operating Systems. Ino [2] introduced a GPU multitasking technique based on host system utilization, but it limits dynamic adaptability as its execution mode relies on predefined thresholds. Shitara et al. [3] addressed the challenge of direct coupling between guest machines and GPU nodes by using Message Passing Interface (MPI) for inter-node communication. However, their approach required OpenCL-based execution, limiting broader applicability. Recently, commercial solutions like vast.ai [4] and run.ai [5] have emerged with the rise of Artificial Intelligence (AI). They have introduced proprietary platforms that integrate GPU pooling within Kubernetes clusters, primarily focused on AI workflows. These existing enterprise solutions focus on closed-source, business-driven implementations and introduce architectural constraints that hinder fully dynamic, decentralized GPU sharing for a broader range of applications.

To overcome these limitations, we propose a novel GPU resource pooling system that enables efficient pooling of GPU resources while ensuring the security aspects of both the GPU owner and the GPU user. Unlike conventional allocation methods, our system implements a flexible pooling architecture such that the users can create custom GPU clusters to handle their training tasks. Additionally, our system eliminates the need of port forwarding or modifying OS kernels, ensuring a smooth onboarding process for GPU owners.Also our approach assumes that GPU owners have a stable internet connection and use compatible hardware and software configurations to integrate seamlessly with our system.

**Contribution:** This paper presents the design and implementation details of a GPU resource pooling system that optimizes resource utilization and effective task scheduling to handle dynamic workloads. The key contributions of our work include:

- **An effective GPU pooling architecture** that enables efficient allocation of GPU resources across multiple workloads
- **A dynamic scheduling algorithm** that optimizes the performance of the task based on the number of GPUs
- **Performance evaluation and benchmarking** demonstrating significant improvements in GPU utilization and reduced task execution times

**Organization:** The remainder of this paper is organized as follows. Section 2 discusses related work, Section 3 describes the system architecture of the GPU pooling system, Section 4

presents the implementation of the system, Section 5 presents the results of an evaluation of the system, Section 6 discusses some future works, and Section 7 concludes the paper.

## II. Related Work

The advent of General-Purpose Graphic Processing Units (GPGPUs), notably propelled by NVIDIA CUDA, marked a transformative milestone in computational acceleration. As the physical infrastructure constraints, programmers increasingly turned to remote GPU platforms to harness the computational power necessary for these GPGPU tasks. This evolution led the research landscape into distinct pre-CUDA and post-CUDA eras. pre-CUDA is characterized by foundational studies in remote GPU utilization while post-CUDA has further expanded the scope and scalability of remote GPU platforms with the advancement in GPU architectures.

In the pre-CUDA era, Kotani et al. [1] introduced a resource selection system aimed at leveraging GPUs as general-purpose computational resources within grid environments. They mainly focused on computing LU decomposition and the conjugate gradient method on network-connected PCs equipped with NVIDIA GeForce GPUs. The system is designed to utilize these GPUs only when they are in idle state. Their study proposed a method for identifying idle GPUs using a screensaver-based approach that monitors video memory usage to detect idle periods of the host machine, thereby minimizing interference with host operations when executing a guest program. This system employs a matchmaking service to align user requirements with benchmark results from currently available grid resources.However, they observed a significant increase in CPU utilization, particularly with single-core host machines, due to CPU intervention during GPU execution. They also noted that their method for obtaining VRAM usage relied on the GetCaps function, which is a Direct Draw feature available only on Windows computers. In INO et al. [6] the system was applied to accelerate biological sequence alignment in a laboratory environment.

Ino. [2] has suggested a GPU multitasking technique designed to execute guest applications on host resources by monitoring the CPU and video memory usage to minimise the disruption to the host machine. To determine whether the host resources should operate in a periodic or continuous execution mode, the system initially runs a null kernel program before executing the guest task and measures its execution time, denoted as k. This measured time k then compared to a pre-measured time $\alpha$, obtained from dedicated execution on the same resource. The resource is considered partially idle if $k \geq \alpha$ and fully idle if $k < \alpha$ occurs successively three times. When the resource is partially idle, the guest program runs periodically by sharing the host CPU cycles with guest and local programs. And, when the resource is fully idle, the guest program runs continuously.

He et al. [7] proposed a method to address host perturbation through the abstraction of a transport layer using Non-volatile Memory Express over Fabrics (NVMe-oF). This approach enables memory operations across various interconnects, such as Fibre Channel, RDMA, or TCP, allowing storage resources to be maintained and upgraded independently. To facilitate remote access from a host server to a GPU, they implemented two proxies: one at the host end and one at the GPU end. These proxies handle the conversion between PCIe

Transaction Layer Packets (TLP) and network packets, effectively functioning as a PCIe virtual switch. Additionally, the proxies provide configuration interfaces that support the dynamic distribution of GPUs within a cloud orchestration system. However, a notable limitation of this approach is that each GPU is dedicated to a single server during use, and the binding relationship between GPUs and the host server remains fixed unless manually released by the users. Shitara et al. [3] have effectively addressed the existing limitation of direct coupling between guest machines and GPUs in multi-node computational environments, where each node is managed by the inter-node communication library MPI. When a user application requests a task from a remote server, it invokes an OpenCL function on the OpenCL devices (worker nodes) and returns the result. Within this OpenCL function call, the remote server requests the OpenCL device currently executing the guest program to read data from its GPU memory. This data is then sent to another OpenCL device, as per the remote server's request, using the MPI library.

In the present-CUDA era Duato et al [8] suggested a programming framework that enables remote execution of CUDA programs with small overhead called rCUDA. A runtime system and a CUDA-to-rCUDA transformation framework are provided to intercept CUDA function calls and redirect these calls to remote GPUs. rCUDA primarily focuses on dedicated high-performance clusters rather than shared grids. In addition, rCUDA does not allow a user program to be distributed over multiple GPUs for parallel computing. Liang and Chang [9] proposed a similar virtualization technology called GridCuda, implemented as a grid-enabled programming toolkit.

CUDA does not provide an interface to distribute compute kernels to remote nodes. To address this limitation Ji et al. [10] introduced a Remote Kernel Launch (RKL) approach which is designed to distribute CUDA kernels to remote GPUs for execution. The lexical analyser written in Fast Lexical Analyzer Generator (Flex) identifies and extracts the kernel part from a given CUDA application. Then a dynamic mapping schema will distribute kernel parts to remote GPUs ensuring balanced workload among nodes. The system consists of three core components: client machine, directory server and the cluster of GPU servers. After the client gives the CUDA code, the directory service will search for an appropriate GPU on the server by analysing the complexity of the CUDA kernel. Complexity will be calculated by the number of threads and blocks in the kernel code. If the selected GPU can handle the workload alone, RKL will not be called, and otherwise RKL will distribute the task to remote GPU servers. If the task is assigned to the GPU cluster, lexical analyser will separate the kernel and the C programming portions and then RKL function stores the kernel on the GPU server side using the directory service. Prior to executing the task on the GPU cluster, it

will send an acknowledgement to get the additional data as required. Finally the result will be sent back to the client side. The mapping scheme is used to map the kernel to the accurate GPU with sufficient threads. Kernels are divided into classes based on the thread requirements and mapping schema will ensure the correct mapping between the GPU class and the Kernel class. Experimental results demonstrated that RKL has minimal overhead, with execution starting within nearly one second for high-complexity GPU kernels making RKL efficient for distributed GPU computing.

A recent study identified several other business-oriented proprietary solutions in the GPU pooling domain. vast.ai [4] is a platform that facilitates distributed GPU pooling capabilities for machine learning tasks. They have reduced the cost of using distributed training of machine learning models using HPC (High performance Computing) clusters made by personal GPUs of users worldwide. vast.ai provides a Jupyter notebook interface with interactive shells for writing scripts. However vast.ai has added some restrictions on the GPU owner side to have a high available network connectivity to prevent any bandwidth issues and the availability of the GPUs for 24/7. Also GPU owners should prevent utilizing his/her GPU after renting to the system.

run.ai [5] is also an enterprise GPU resource management platform specialized in ML tasks. System facilitates an orchestration service for existing datacenter GPU resources using Kubernetes. However, this doesn't facilitate renting dometic personal GPUs into their GPU pool. GPUDeploy [11] is another GPU renting platform specifically designed for ML tasks. They use SCP (Secure Copy Protocol) to transfer files from local machines to their servers and RSYNC [12] for file synchronization. The system provides a fault tolerance mechanism using TMUX [13] to keep applications running even after any disconnections from the client side. They have provided some instructions that need to be followed in order to rent GPUs to their pool, such as Linux distribution with root privileged users, NVIDIA GPUs with the latest drivers pre-installed, and expose ports 5000-5999. GPU owners who satisfy these requirements can pool their resources by establishing a SSH connection with the main server.

## III. System Architecture

### A. Centralized Management Setup

This is the backbone of this system, and almost all the operations are executed and controlled by this system. In this system, there are multiple standalone operations going on, such as authentication, job scheduling, monitoring, file uploading, HTTP request routing, etc. These are implemented as microservices. The system is fully structured with a microservice architecture.

*1) Authentication Service:* This service is responsible for user authentication.

*2) Job Scheduling Service :* This service is responsible for the creation of Kubernetes Jobs and Stateful sets based on the dynamic tasks which are submitted by the customers. Moreover, it manages the labeling and training of worker nodes and adds affinity and tolerations for pods ensuring accurate job creation.

*3) API Gateway:* It serves as the interface that receives HTTP requests from the frontend and routes them to respective microservices by creating [14] messages using google proto3 serialization method and pushing them to related RabbitMQ message queues using AMQP.

*4) Monitoring Service:* This service is responsible for monitoring the completion of the pod job and cleaning up the resources created to perform the job. When a job has been created, the job scheduling service triggers the monitoring process using a protobuf message.

*5) Utility Service:* It manages the interaction of the system with GCP bucket storage. The training scripts that have been submitted by the users will be saved in the GCP bucket store.

The Kubernetes control plane, which handles the multi-node distributed system, is also on the same server machine. The reason for having the Kubernetes control plane and the microservices on the same server is to maintain better communication with the system and reduce the time taken to apply a job to the cluster. Fig.1 showcases the high-level architecture design of the microservice based central management setup.

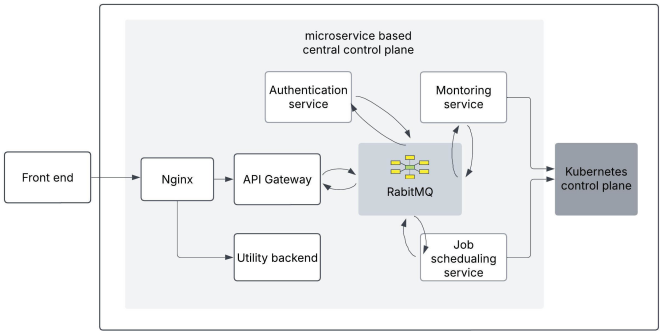


Fig. 1. centralised microservices-based system

### B. Load Balancing

In front of the API Gateway, an NGINX reverse proxy has been deployed to manage traffic, redirect requests from the frontend to the API Gateway and utility service, and enforce CORS policies. The utility service was kept independent of the API Gateway while optimizing its performance to minimize latency.

### C. Communication Protocols

Our system uses both synchronous and asynchronous communication. We mainly use HTTP for synchronous communication between the frontend and the API gateway and

asynchronous communication for inter-service communication using a message broker, which is RabbitMQ. Fig.2 illustrates the Remote Procedure Call (RPC) pattern used for handling authentication requests within the system in an asynchronous manner.

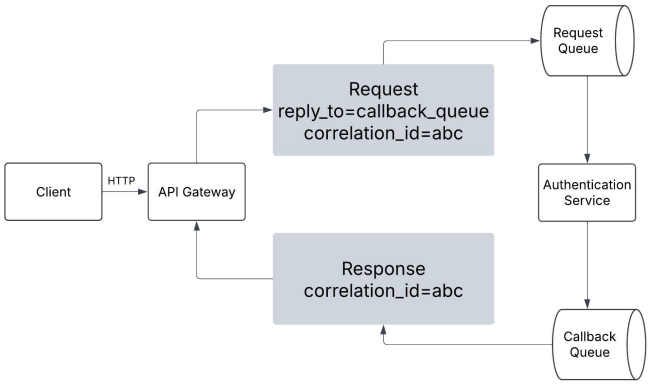


Fig. 2. RabbitMQ RPC Pattern

When a client sends an HTTP request to the API Gateway, it creates an exclusive callback queue to receive responses as the initial step. Then the request reroutes to a queue that the authentication service listens to. That request includes $reply\_to$, which is set to the callback queue, and the unique $correlation\_id$. Once the request has been received, the authentication service processes it and sends the results back to the specified callback queue. The API gateway, waiting on the callback queue, verifies the $correlation\_id$ and forwards the authentication response to the client.

### D. VPN Setup

This system establishes a distributed architecture by leveraging Kubernetes, which can be deployed as either a single-node or multi-node configuration. In this implementation, a multi-node Kubernetes cluster is employed to orchestrate a distributed system of GPUs located worldwide. However, this approach presents certain challenges. To enable seamless operation, all nodes must reside within the same network. VPN offers an effective solution by providing secure communication channels over the Internet. Accordingly, an OpenVPN server, configured with a certificate authority, has been implemented [15]. Upon onboarding, new customers are required to download the **.ovpn** file as their initial step and establish a connection to the VPN. Figure 3 illustrates how the VPN server ensures secure communication among nodes within the Kubernetes Pod Network. This configuration facilitates private, efficient, and isolated inter-pod and inter-node communication across the cluster.

### E. Distributed system using Kubernetes

Since Kubernetes is a container orchestration tool, it provides a fault-tolerant mechanism to maintain a reliable and scalable resource cluster. When running a script on a computer,

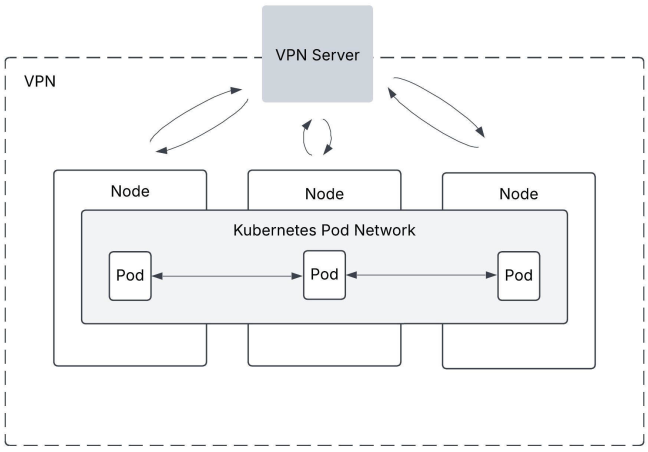


Fig. 3. Kubernetes on VPN

the isolation needs to be implemented in advance. For that Docker is used to run the tasks on the GPU owner side. When using such a container runtime, having an orchestration tool is crucial for reliability. Kubernetes serves that purpose reliably.

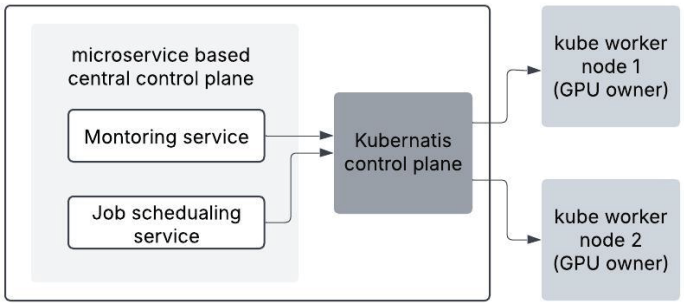


Fig. 4. Distributed system using Kubernetes

To allow Kubernetes to utilize the GPUs on each worker node, NVIDIA Container Toolkit [16] is automatically installed during the worker onboarding process. Additionally, NVIDIA Device Plugin Daemonset [17] is deployed in the Kubernetes cluster to detect and allocate GPU resources for training tasks. This setup ensures that training tasks can be effectively executed using the pooled GPUs without manual configuration on each node.

Containerd [18] is used for the container runtime and Calico [19] is integrated as the Container Network Interface (CNI) to enable pod networking and inter-node communication. A GCP key secret is used to grant read access of the GCP bucket to Kubernetes. This allows the training pods to automatically download user-uploaded scripts and the datasets at runtime.

Kubernetes mainly functions in the system from two aspects. First, it serves in creating the GPU pool with its multi-node functionality. The second is the fault tolerance mechanism for node failures. Every working GPU is considered as a Kubernetes node. Fig.4 illustrates the high-level architecture of

the distributed system of worker nodes, where the Kubernetes control plane manages multiple worker nodes.

### F. Service-Runner tool

This tool is designed and implemented to centrally start the services and manage all the environment variables and all other utilities of microservices. When there is a new update in one of the services, this program will automatically trigger through a pipeline and fetch the **new Docker image tag** from the Docker hub and spin up with new updates.

## IV. Implementation

### A. Worker Node Onboarding Process

For renting a GPU, a user has to be registered as a GPU owner in the system. Then, the user will be directed to a page that provides necessary instructions to set up the VPN connection and connect to the cluster. Additionally, the system is currently optimized for Ubuntu 20.04 LTS or later, making it incompatible with Windows-based machines. For efficient workload distribution, we assume homogeneous GPU clusters where all GPUs have similar specifications, specifically NVIDIA RTX 4000 series with uniform VRAM capacities. Once a worker node fulfills these basic requirements, it needs to be connected to the VPN server using the instructions provided by the system in order to rent their GPU. VPN allows communication within worker nodes, which are typically behind distinct NATs (Network Address Translations). Unique client certificates will be provided for workers, ensuring that mutual TLS authentication remains intact with the VPN server. A virtual IP address will be assigned to the worker node, upon successful registration with the VPN server.

Then the user can download the deb package called **worker-onboarding.deb** and run it in the user's local machine terminal. All necessary guidelines for running the deb package will be provided on the web page. Installation of all required tools has been automated by the deb package. It will check whether the user has a GPU, whether he is connected to the VPN, etc. If those verifications fail, the user will not be able to continue the installation process to connect with the system. Upon successful completion, the user will be automatically connected to the GPU pool and a local web application and a GPU monitoring process will be started to streamline the worker node processes. The specifications of the GPU will be sent to the main system to be saved in the database. Fig.5 illustrates the high-level flow of the worker onboarding process. It represents the sequence of steps that a user should follow, from setting up the VPN connection and downloading the onboarding package, to checking GPU configuration, and successfully connecting to the system.

### B. Job Scheduling Process

The job scheduling process is handled by the job scheduling service. The job scheduling process is dynamically handled based on the number of selected GPUs. Currently, our system does not support dynamic workload distribution among heterogeneous GPUs. Users must manually select GPUs of similar

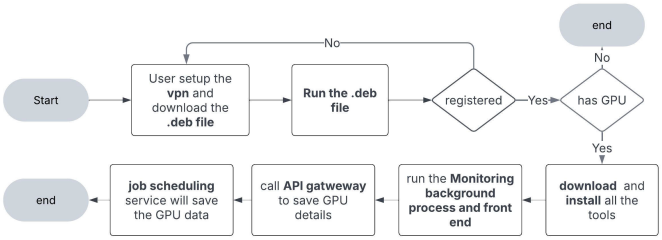


Fig. 5. Worker onboarding workflow

specifications for optimal performance. In the scheduling process we use **kubernetes/client-go** SDK in the job scheduling service to directly communicate with the Kubernetes control plane. This SDK provides almost all the functionalities to create yaml configuration files and apply them to the Kubernetes cluster without using any external libraries or kubectl commands.

*1) Saving GPU details and getting available nodes:* The existence of a GPU and its specifications are scraped by a background program in the onboarding process of a worker node. It includes GPU model, number of CUDA cores, VRAM, TFlops, PCIe lanes, etc. After a successful onboarding, the GPU specifications will be saved to the database by assigning a unique node name. Saved GPU data will be used when retrieving the available nodes in the cluster at any given time. Only the **Ready** state nodes will be displayed as available GPUs for the user. Also, nodes need to match with our name scheme for security purposes.

*2) Tainting a GPU node:* When users select one or more GPUs, a unique **taint** will be applied to the selected GPUs from the job scheduling service. Kubernetes taints work in such a way that any pod without a matching **toleration** will not be allowed to schedule in that node. This strategy isolates the rented GPUs from the other GPUs. Already tainted GPUs are not shown as available GPUs in the frontend, ensuring multiple taints are not applied to a node.

*3) Scheduling one-to-one jobs:* When the user selects only a single GPU, the jobs scheduling service creates a Kubernetes Job with all the environment variables and all the details about the uploaded training script by the user. First, all the metadata of the job will be saved in the **MongoDB database**, and the related **docId** is used to create the job name. Then a new yaml file will be created using the client-go SDK [20] in the format of (docid).yml. Finally, that job will be applied to the tainted node. By setting **.spec.ttlSecondsAfterFinished** value in the yaml file, the job will be automatically deleted with the respective pods after completion. Otherwise, if the user tries to execute the job again, that will not be scheduled by the Kubernetes.

*4) Scheduling distributed jobs:* If the user selects multiple GPUs, a Kubernetes **Statefulset** will be created by the job scheduling service along with a **headless service**. Node affinity is used to guarantee that the pods will be

only deployed in rented GPUs not the others. In distributed machine learning tasks, a master worker node needs to be automatically selected(will be discussed in a later section). For that, an indexing mechanism is needed. Statefulset facilitates the indexing mechanism. The zero indexed pod will be used as the master worker node and it is responsible to manage the whole process(will discuss later). Also, statefulset stores the state, and in case of a pod failure during execution, it spins up a new pod with the past state knowledge. This is helpful in distributed training. The headless service is an extension of the clusterIP service but doesn't have a static IP. It gives direct access to the worker pods to get updates from other worker nodes. After successfully applying the statefulset, the monitoring service will be triggered for monitoring the pods to signal the front end for job completion and clean out the resources. The system assumes that PyTorch correctly distributes data across all rented GPU instances, ensuring balanced training across multiple nodes.

### C. Distributed Model Training Process

The system leverages the PyTorch Distributed Data Parallel (DDP) library for distributed training among selected GPU nodes. As the initial step, the system dynamically chooses a master node from the selected nodes and broadcasts its IP address to the remaining nodes. To facilitate seamless communication, a Kubernetes headless service is used to resolve the IP address of the master pod, and a stateful set ensures the master pod's reference remains persistent throughout the training lifecycle. The master node serves as the rendezvous point for the initializing the distributed process group. It coordinates gradient synchronization, collective communication among all participating nodes and keeping checkpoints throughout the training process. The use of a headless service allows direct pod-to-pod communication, which is essential for low-latency, high-throughput operations. The stateful set guarantees a consistent network identity for the master pod, simplifying coordination during re-initialization and recovery from the last checkpoint in case of a node failure.

Once the process group is initialized, PyTorch DDP enables data parallelism by splitting the training dataset across all participating nodes. Each node independently processes its own shard of the dataset, performing a forward pass followed by a backward pass to compute gradients locally. After the backward pass, PyTorch DDP synchronizes gradients across all nodes using an all-reduce operation, which averages the gradients from each worker. This ensures that all nodes maintain consistent model updates. With the averaged gradients computed, each node simultaneously updates its model parameters, keeping the distributed models in sync. In our current system, gradient synchronization occurs at the end of each epoch, though we are actively researching more efficient synchronization strategies to minimize network latency and reduce overall training time.

### D. Pod Monitoring and Pod Cleanup

The **monitoring service** is responsible for this process. After successfully applying a statefulset, the monitoring service will be triggered by the job scheduling service with the name of the statefulset. Then, the monitoring service will get a pod list that matches the name starting from the statefulset name. Then it will scrape the restart count of each and every pod in the list. When the restart count becomes greater than zero, the stateful set will be deleted by the monitoring service. The reason for that is, statefulset always tries to maintain the same pod count. Hence, even in the success scenario, a new pod will be started again and again, increasing its restart count. Therefore, when the restart count becomes 1, the monitoring service automatically deletes the stateful set and sends a protobuf message via RabbitMQ to the API Gateway. Then that response will be used by the front end for displaying successful completion.

In a single-job scenario, a monitoring process will not be executed, But in case of failure, the failure state will be recorded. In success scenarios, the job will be automatically deleted, as mentioned earlier.

### E. Real-Time Monitoring and Logging

Logs provide valuable insights about the progress of a submitted job for the users who are using a GPU for some training task. Also, real-time GPU monitoring enables GPU owners to monitor the utilization of their rented GPUs.

*1) Logging:* A dedicated web-socket server and a nodeport service have been deployed on the Kubernetes master plane to collect and transmit real-time logs from the worker pods. A role has been created with permissions to access any pod in the cluster, and that role has been binded to the service account. Once a task is initiated, the websocket server gets the worker pod names from the job scheduling service and new web socket connections will be established for each worker node to the frontend. These websockets stream logs of the running pods in real-time. This allows the model trainer to track the progress of the task eliminating the need to manually check each worker pod. The websocket server has been programmed using Node.js and applied to the Kubernetes cluster using a deployment with 1 replica set. So it always keeps running inside the cluster.

*2) GPU Monitoring:* Here, Monitoring refers to monitoring of the GPU utilization. When a worker node connects to the Kubernetes cluster a background service is initiated in the GPU owner's machine. It collects the GPU utilization metrics in real-time and streams to the worker frontend via websockets. This information which includes GPU utilization, GPU temperature, etc. helps the GPU owner for visualization and analysis.

### F. API Request Handling

All the HTTP requests from the frontend will be routed to the API Gateway service. The API Gateway service runs an HTTP web server and continuously listens for the HTTP traffic from the frontend. It also continuously listens to the

RabbitMQ queues for the messages from the other services at the same time. When an HTTP request comes to the gateway it will call the required function. If the request involves another service, a new **protobuf** message will be created. Protobuf mechanism is used to serialize messages to communicate with the microservices through the RabbitMQ message broker. All the message types are defined in a .proto file and the necessary messages are created using that file.

To handle the asynchronous communications, there are predefined message queues from the API Gateway to the other services. To listen to the messages, a consumer is always running in the API gateway. But when it comes to the response side, some requests need to be handled synchronously. In such cases when sending the message a **replyTo** field is being used which is a feature of RabbitMQ. It will create a random reply queue to the api gateway as soon as the message is sent to the relevant service. With the delivery of a reply message, that queue will be deleted by RabbitMQ. This mechanism has been used for registering and signing in users in the API gateway.

### G. Data storages

MongoDB and the GCP storage buckets are the main data storage mediums used by the system. Since MongoDB is a scalable cloud-based platform, it is efficient to store and retrieve data from various services. In the services that use a database, a MongoDB client is created at the beginning of the service and added to the application context. That client will be used during a database operation.

MongoDB will be used to save all the metadata of the operations. The GCP storage buckets are used for storing the user training scripts. When the training starts, the training pod downloads the script from the bucket storage by matching the name provided in the YAML file and starts the training process.

### H. Internode Communication

At the beginning of each and every service, the RabbitMQ connection will be initialized. A RabbitMQ client, a channel for sending messages, and a channel for receiving messages are being created and added to the application context. Then a consumer will be started and services will continuously listen to the defined queue. When the service needs to send a message to a relevant queue, the producer channel and the RabbitMQ client will be taken by the context and send the message. To identify the different messages, a heading-type has been introduced. When a message comes to the service, it checks the **heading-type** and calls the necessary functions accordingly.

For synchronous communication, the reply-to queue is used. An empty string will be sent with such messages, and when replying, a unique queue will be created by RabbitMQ and send the reply on it.

### I. Service management (service-runner)

Since the control plane is operated by several microservices, managing them individually is challenging in both production and development environments. Therefore, to centrally manage them using a program, the service-runner service has been implemented. This service has two docker-compose files. One is **docker-compose.dev**, and the other is **docker-compose.prod**. These compose files are used to spin up the Docker containers for each and every microservice by pulling the newest Docker image from Docker Hub.

*1) Development Environment:* In the development environment, developers should be able to see real-time Docker logs for their changes. The challenge is to track the file changes and build the service again. For that, there is a live server tool called **Air** for Go. It will track the new changes and build the Go binary inside the Docker container. To do so in the Docker compose file, the file path to the service needs to be added as a volume instead of the Docker image.

In the development environment, to do testing, a local Kubernetes cluster needs to be created. For that **kind**, or **Minikube** can be used. When connecting to the Kubernetes, the .kube/config is needed by the monitoring service and the job scheduling service. In the kind cluster, the Kube API server is running on a localhost url. Hence, in the development environment, the network mode of all services needs to be changed to host. But Minikube uses a Docker container environment to make its single-node cluster. So, when using Minikube, the monitoring service and the job scheduling service need to be added to the Minikube Docker network, otherwise these services cannot access the Kube config file. Fig.6 illustrates the high-level architecture of the Service Runner in the development environment. As shown in the figure, the Service Runner interacts with the Docker Compose environment, where each service is configured to mount its respective file path and environment variables. devStart.sh shell script is used to run the Service Runner in development environment and it executes dockerfile.dev to spin up docker containers within the development environment.

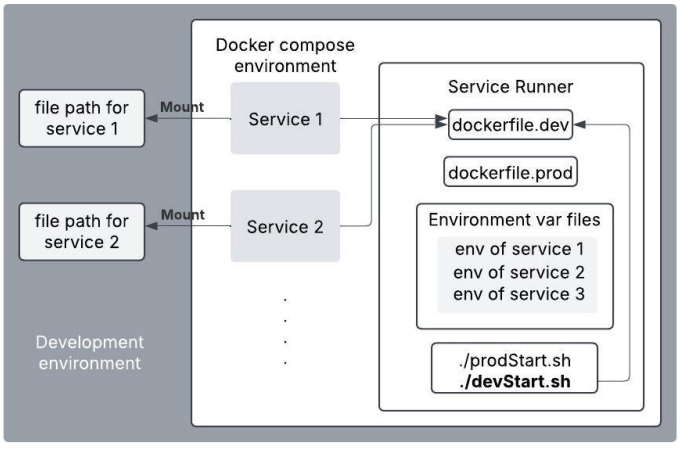


Fig. 6. Service Runner service in a development environment.

*2) Production environment:* In the production environment, built Docker images are used to spin up the services. On the devOps side, when a new change is merged to the main

branch of the github repository, a new image will be created by incrementing the existing tag by one. After pushing the new Docker image to the Docker hub, the Github action creates a ssh connection to the testing environment, and a new pipeline is created. After doing some testing, the changes will be added to the production VM. When adding them the service-runner program runs a script. It scrapes the newest tags of the services and if there is a new tag of one of the services, that image will be automatically pulled, and a Docker container will be automatically spinned up with the new changes. Since the binary files are used in the production environment container size has been reduced in a significant amount than the development environment. Fig.7 illustrates the high-level architecture of the Service Runner in the production environment. As shown in the figure, the Service Runner interacts with the Docker Compose environment, where images of each service is pulled from the Docker Hub and configured with their environment variables. prodStart.sh shell script is used to run the Service Runner in production environment and it executes dockerfile.prod to spin up docker containers within the production environment.

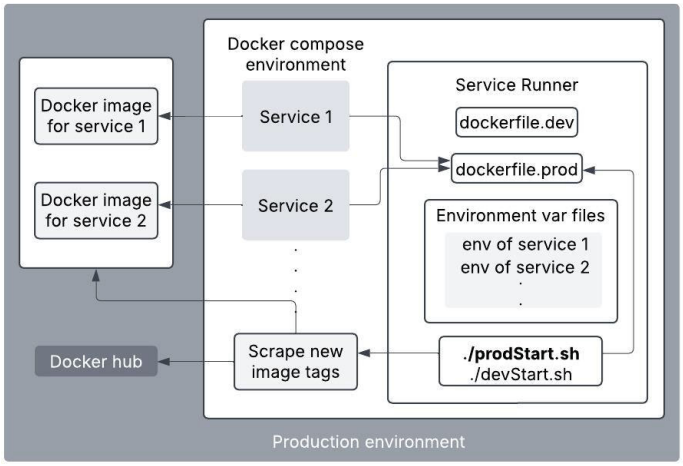


Fig. 7. Service Runner service in production environment.

*3) Environment Variable Management:* Without adding the .env files in the production environment for the services itself all the environmental variables are included in the service runner. In the service runner, there is a folder which contains all the .env files specific to the services. Then in the docker compose file these .env files are loaded. With this setup, managing secret data for all the services can be handled in one place. Since all the secret values are included here, this service has been secured more than the other services.

### J. Security Measures

To ensure secure communication during distributed training, our system utilizes a VPN-based overlay network, which inherently provides encrypted pod-to-pod communication. On the execution side, the inherent isolation provided by containerized environments helps mitigate code injection risks to some extent. However, to further strengthen the system's security posture, we are planning to introduce sandboxing mechanisms for containers in future iterations. This will provide additional layers of isolation, preventing unauthorized code execution or lateral movement within the system.

To safeguard against resource-based attacks or unintentional overuse, such as memory leaks or CPU overflows, system-enforced resource limits and quotas in Kubernetes deployment configurations. These limits help ensure system stability and fair usage across distributed GPU nodes.

## V. Performance Evaluation

Evaluating the performance of our system is crucial to understanding its efficiency in handling distributed machine learning workloads. This section presents the hardware specifications, execution performance, and resource utilization metrics gathered during multi-node training experiments.

### A. Platform

Our system utilizes NVIDIA GeForce RTX 4070 Ti Super GPUs, each offering 2048 MB VRAM, 450 TFLOPS theoretical performance, and 4 PCIe lanes. These GPUs serve as the foundation for our distributed training environment, ensuring high computational efficiency and scalability.

### B. GPU Utilization Analysis

To evaluate distributed training performance, we analysed the GPU utilization, SM Clocks, temperature and power consumption for single and multi-node training tasks. We measured execution times for both single-node and two-node configurations to assess the scalability and efficiency of our approach. Table I summarizes the execution times.

TABLE I. Performance Metrics for Single and Multi-Node Training

| Mode | Utilization (%) | | Temperature (°C) | | Time (s) |
|---|---|---|---|---|---|
| | Mean | Max | Mean | Max | |
| Single GPU | 1.90 | 20 | 40.5 | 43 | 16.62 |
| Multi-GPU (two) | 23 | 83 | 43.95 | 52 | 29.28 |

**GPU Utilization and SM Clocks:** The utilization graphs shown in Fig.8 provide a detailed view of performance metrics during multi-GPU training. The upper graph displays the GPU SM(Streaming Multiprocessors) Clocks, which remained stable throughout the execution, indicating that the computational resources of the GPUs were fully utilized. The lower graph shows GPU Utilization, where the workload is evenly distributed across the two nodes, with both GPUs experiencing significant utilization peaks, reflecting the efficient handling of the training process.

**Temperature and Power Consumption:** Fig.9 displays the GPU temperature and power consumption metrics for a multi-GPU training process. The graphs show that the temperature was consistently maintained at safe levels (around 44°C), while power usage remained stable within expected limits,


Fig. 8. Grafana utilization for muti-GPU training

peaking at a maximum of 1.58 kW, reflecting the system's efficient performance during multi-GPU training.


Fig. 9. Grafana statistics for muti-GPU training

These results demonstrate that our system not only reduces training latency but also efficiently distributes workloads across multiple nodes while maintaining stable operational conditions. Future work will further optimize multi-node task scheduling and incorporate performance benchmarks to enhance the evaluation scope.

## VI. Future Work

While our GPU resource pooling system significantly handles resource utilization, there are several areas for future enhancement. There are many variations in GPUs and their performance also vary with the specifications. Future work will focus on scheduling work such that the workload given to a user created cluster to be divided among the work nodes considering the heterogeneity of GPUs. Currently the system is designed such that it has a centralized management setup with the Kubernetes control plane and the microservices deployed in the same virtual machine. This creates a single point of failure. So, we plan to migrate these microservices in a distributed manner that reduces dependency on a central controller.

In the process of distributed machine learning, the calculated gradients should be synchronised with the master node. This has to be optimized to reduce unnecessary communication between the worker nodes. Finding an optimal frequency for this gradient synchronisation is also planned as a future work in our system.

During the task execution there is a possibility for a worker node to be offline due to a connection issue or current failure. We plan to have a fault-tolerant mechanism to address these issues. This can be achieved by implementing checkpoints to save task execution progress in cloud storage, allowing recovery from the last checkpoint instead of restarting the process. Enabling dynamic node switching by migrating tasks to a new node within an optimal budget range specified by the user is also a mitigation method that we have planned. In that case, with enough power from the modern GPUs, several machine learning tasks can be run on the same GPU but with proper VRAM and storage management. Using that advantage we are planning to create an algorithm to calculate an average usage of currently busy GPUs and choose GPUs that have an average usage below 50 percent and further distribute the remaining task of the failed GPU among those GPUs.

Although this system is focused on machine learning training tasks, this system has the ability to be expanded for more general purpose applications. Therefore we plan to introduce a new extension language by providing specific functionalities to distribute programming tasks among multiple nodes using MPI like libraries. Moreover this system can be expanded to use cases like video editing, weather prediction, etc.

## VII. Conclusion

In this paper, the discussed GPU pooling system has the ability to efficiently manage GPU resources in a wide GPU pool, and provide custom clustering facility to users who want to have their machine learning training to be done in a cost effective and reliable manner. This system facilitates dynamic scheduling to ensure efficient usage of available GPU resources. Experimental results validate the effectiveness of the system.

Despite of these successful outcomes, there are several challenges that remain to be addressed. One significant concern is the fair workload distribution concerning the heterogeneity of GPUs. For efficient workload distribution more portion of data can be allocated for high-end GPUs of the cluster. There is also a concern with the concurrency handling when selecting GPUs by the users. Ensuring fairness and priority handling in such scenarios will be critical.

Additionally, system robustness against hardware failures and network latency needs further attention to guarantee uninterrupted service and high availability. Fault tolerance should be addressed in various aspects such as considering about shifting fault processes to a user suggested backup GPU, or handling the GPU migration automatically by the system itself.

To improve the efficiency of distributed training, we are currently focusing on reducing pod-to-pod network latency

by transitioning from a centralized VPN topology to a decentralized mesh VPN. In the current system, the system utilizes OpenVPN, where all inter-pod communication is routed through a central VPN server. This setup introduces additional latency due to the server acting as a middleman for all traffic. To address this bottleneck, we are planning to migrate to the mesh VPN architecture, which enables peer-to-peer communication between pods without a central intermediary. It will help to reduce network latency and improve overall training performance.

During the evaluation phase of our system, we faced resource constraints that limited our ability to scale experiments beyond two GPUs. Currently, we are utilizing two GPUs provided by the university, which has allowed us to validate the core functionality and distributed training workflow. However, we acknowledge the importance of evaluating the system at a larger scale to demonstrate its full potential and robustness. To address this, we have applied for additional funding to support resource expansion by using GPU instances from Google Cloud Platform (GCP) to extend the number of participating nodes in our distributed setup.

By utilizing GPU pooling organizations and individuals can achieve more sustainable, cost effective computing infrastructure, opening new opportunities for innovation and advancement in productivity of high performance workload management.

## ACKNOWLEDGMENT

We would like to acknowledge the support received from the Senate Research Committee Grant, University of Moratuwa, Sri Lanka.

## REFERENCES

[1] Y. Kotani, F. Ino, and K. Hagihara, "A resource selection system for cycle stealing in gpu grids," *Journal of grid computing*, vol. 6, pp. 399–416, 2008.

[2] F. Ino, "The past, present, and future of gpu-accelerated grid computing," in *2013 First International Symposium on Computing and Networking*. IEEE, 2013, pp. 17–21.

[3] A. Shitara, T. Nakahama, M. Yamada, T. Kamata, Y. Nishikawa, M. Yoshimi, and H. Amano, "Vegeta: An implementation and evaluation of development-support middleware on multiple opencl platform," in *2011 Second International Conference on Networking and Computing*. IEEE, 2011, pp. 141–147.

[4] "Rent GPUs — Vast.ai — vast.ai," https://vast.ai/, [Accessed 01-03-2025].

[5] "Run:ai - AI Optimization and Orchestration — run.ai," https://www.run.ai/, [Accessed 01-03-2025].

[6] F. Ino, Y. Kotani, Y. Munekawa, and K. Hagihara, "Harnessing the power of idle gpus for acceleration of biological sequence alignment," *Parallel Processing Letters*, vol. 19, no. 04, pp. 513–533, 2009.

[7] B. He, X. Zheng, Y. Chen, W. Li, Y. Zhou, X. Long, P. Zhang, X. Lu, L. Jiang, Q. Liu *et al.*, "Dxpu: Large-scale disaggregated gpu pools in the datacenter," *ACM Transactions on Architecture and Code Optimization*, vol. 20, no. 4, pp. 1–23, 2023.

[8] J. Duato, A. J. Pena, F. Silla, R. Mayo, and E. S. Quintana-Ortí, "rcuda: Reducing the number of gpu-based accelerators in high performance clusters," in *2010 International Conference on High Performance Computing & Simulation*. IEEE, 2010, pp. 224–231.

[9] T.-Y. Liang and Y.-W. Chang, "Gridcuda: a grid-enabled cuda programming toolkit," in *2011 IEEE Workshops of International Conference on Advanced Information Networking and Applications*. IEEE, 2011, pp. 141–146.

[10] X. Ji, S. Davis, E. Hardesty, X. Liang, S. Saha, and H. Jiang, "Towards utilizing remote gpus for cuda program execution," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*. The Steering Committee of The World Congress in Computer Science, Computer ..., 2011, p. 1.

[11] D. S. Shenwai, "Meet GPUDeploy.com: An AI Startup that Provides a Marketplace for Renting GPUs - MarkTechPost —marktechpost.com," https://www.marktechpost.com/2024/06/19/meet-gpudeploy-com-an-ai-startup-that-provides-a-marketplace-for-renting-gpus/,[Accessed 01-03-2025].

[12] "rsync - Wikipedia — en.wikipedia.org," https://en.wikipedia.org/wiki/Rsync, [Accessed 01-03-2025].

[13] "tmux - Wikipedia — en.wikipedia.org," https://en.wikipedia.org/wiki/Tmux, [Accessed 01-03-2025].

[14] "RabbitMQ tutorial - Remote procedure call (RPC) — RabbitMQ — rabbitmq.com," https://www.rabbitmq.com/tutorials/tutorial-six-php, [Accessed 01-03-2025].

[15] "Business VPN For Secure Networking — OpenVPN — openvpn.net," https://openvpn.net/, [Accessed 01-03-2025].

[16] "GitHub - NVIDIA/nvidia-container-toolkit: Build and run containers leveraging NVIDIA GPUs — github.com," https://github.com/NVIDIA/nvidia-container-toolkit, [Accessed 03-03-2025].

[17] "GitHub - NVIDIA/k8s-device-plugin: NVIDIA device plugin for Kubernetes — github.com," https://github.com/NVIDIA/k8s-device-plugin, [Accessed 01-03-2025].

[18] "containerd — containerd.io," https://containerd.io/, [Accessed 01-03-2025].

[19] "Project Calico — Tigera - Creator of Calico — tigera.io," https://www.tigera.io/project-calico/, [Accessed 01-03-2025].

[20] "GitHub - kubernetes/client-go: Go client for Kubernetes. — github.com," https://github.com/kubernetes/client-go, [Accessed 01-03-2025].