

GPU resource allocation in Clouds

Hoda Sedighi

A Thesis
in
Concordia Institute
for
Information System Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy (Information and System Engineering) at
Concordia University
Montréal, Québec, Canada

April 2026

© Hoda Sedighi, 2026

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By: **Hoda Sedighi**

Entitled: **GPU resource allocation in Clouds**

and submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Information and System Engineering)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

Dr. Abdelhak Bentaleb Chair

Dr. Andreas Kassler External Examiner

Dr. Dhrubajyoti Goswami External to Program

Dr. Chadi Assi Examiner

Dr. Jamal Bentahar Examiner

Dr. Roch Glitho Supervisor

Approved by

Dr. Andrea Schiffauerova, Graduate Program Director

Date of Defence: April 28, 2026

Dr. Mourad Debbabi, Dean
Gina Cody School of Engineering and Computer Science

Abstract

GPU resource allocation in Clouds

Hoda Sedighi, Ph.D.

Concordia University, 2026

Cloud computing enables on-demand access to a shared pool of configurable computing resources, including Graphics Processing Units (GPUs), which are essential for accelerating compute-intensive workloads such as artificial intelligence (AI), machine learning (ML), and microservice-based applications. As GPU adoption grows in modern cloud environments, the diversity of workloads, heterogeneous resource requirements, and strict isolation demands make efficient GPU resource allocation a critical challenge. Inefficient scheduling and static allocation policies often result in GPU underutilization, performance interference, prolonged task completion times, and degraded Quality of Service (QoS).

Modern cloud workloads require dynamic and fine-grained GPU resource management to satisfy fairness, performance isolation, and latency constraints. In a multi-tenant cloud environment, GPU allocation mechanisms must enforce fairness and strong isolation to prevent interference across workloads while maintaining high utilization. Moreover, cloud-native applications, such as microservices, consist of loosely coupled, interdependent components that exhibit diverse GPU demands and dynamic execution behaviours. Inefficient resource sharing in such applications can lead to performance bottlenecks and increased communication and data-transfer overhead. Furthermore, real-time cloud services, particularly latency-sensitive AI inference workloads, require priority-based GPU allocation and scheduling to meet deadline requirements. The requirements together outline GPU resource allocation as a multi-dimensional challenge. At the cluster level, the problem concerns ensuring fairness and isolation among tenants. At the application level, the focus is on maximizing efficiency for cloud-native applications. Lastly, at the runtime level, the challenge lies in executing tasks with priority awareness under real-time latency constraints.

This thesis addresses these challenges with three key contributions for multi-tenant cloud environments, where efficient, fair, and latency-aware GPU allocation is critical. First, we propose a fairness-driven GPU allocation mechanism that enforces strong isolation among tenants while maximizing GPU utilization in shared cloud infrastructures. Second, we introduce a dynamic GPU resource allocation framework designed for microservice-based applications. This framework adapts to workload variations and inter-component dependencies to improve throughput and reduce end-to-end latency. Third, we present a priority-based GPU scheduling strategy that supports task preemption and resumption, enabling the timely execution of real-time workloads while preserving fairness.

Acknowledgments

First and foremost, my deepest gratitude goes to my supervisor, Prof. Roch Glitho. His guidance and constructive feedback were instrumental in strengthening the quality of this research. I appreciate his trust and the independence he granted me during this work.

I would like to thank my thesis committee members, Dr. Dhrubajyoti Goswami, Dr. Chadi Assi, and Dr. Jamal Bentahar, for their thoughtful suggestions and support throughout this process. I am also grateful to the external examiner, Dr. Andreas Kassler, for his time and for accepting to serve on my Ph.D. thesis committee. I am grateful to my colleagues and friends at Concordia University, especially in the TSE lab, for their support and collaboration.

I extend my sincere appreciation to Dániel Gehberger and Dr. Fetahi Wuhib at Ericsson for their collaboration and support. Working alongside them provided valuable industry perspectives and allowed me to bridge academic research with practical system challenges. Their technical discussions and constructive feedback significantly enriched this work. I greatly value the opportunity to have collaborated with them.

I owe my deepest gratitude to my mother. She has always been my role model, and her belief in education, hard work, and integrity shaped my path long before this Ph.D. began. Her strength, sacrifices, and unwavering encouragement have been the foundation of my resilience and determination. I am profoundly grateful for her constant support and the values she instilled in me. My father, although no longer with us, remains an enduring source of inspiration in my life. His love and his faith in my abilities gave me confidence from an early age. I carry his memory with me in every step forward. To my sister, you have been my rock and a guiding presence in every stage of my life. You have always stood beside me with sincere encouragement, offering advice and support without hesitation or expectation. Your guidance has influenced many of my decisions, and I truly believe that without your wisdom and encouragement, I would not be where I am today. I remain deeply thankful for your presence in my life.

My heartfelt gratitude goes to my husband, whose love and support sustained me throughout this journey. Your encouragement during difficult moments and your steady confidence in me meant more than I can fully express. You stood beside me through every challenge with belief, especially when I needed it most. Your presence made this demanding path easier to walk, and I am truly grateful to share this achievement with you.

Contents

List of Figures	viii
List of Tables	x
List of Abbreviations	xi
1 Introduction	1
1.1 Overview	1
1.2 Challenges	3
1.3 Thesis Contributions	5
1.3.1 Efficient GPU Resource Allocation in Clouds [1][2][3]	5
1.3.2 Efficient GPU Resource Allocation for Microservice Applications [4][5][6][7]	5
1.3.3 Priority-based GPU Resource Allocation in Clouds [8]	6
1.4 Background Information	6
1.4.1 Cloud Computing	7
1.4.2 GPUs in Cloud Computing	7
1.4.3 GPU Architecture and Execution Flow	7
1.4.4 GPU Sharing in Cloud Environments	8
1.4.5 GPU Multitasking Techniques	10
1.4.6 Microservice-Based Applications and Resource Management	10
1.4.7 Priority-Based Scheduling and Real-Time Resource Allocation	11
1.5 Thesis Outline	12
2 Requirements and Related Work	13
2.1 Illustrative Use Case	13
2.2 Requirements	15
2.2.1 General Requirements for GPU Resource Allocation in Clouds	15
2.2.2 Requirements for Efficient GPU Resource Allocation in Clouds	15
2.2.3 Requirements for Efficient GPU Resource Allocation for Microservice Applications	16

2.2.4	Requirements for Priority-based GPU Resource Allocation in Clouds	17
2.3	Related Work	17
2.3.1	Efficient GPU Resource Allocation in Clouds	18
2.3.2	Efficient GPU Resource Allocation for Microservice Applications	20
2.3.3	Priority-based GPU Resource Allocation in Clouds	22
2.4	Conclusion	26
3	Efficient GPU Resource Allocation in Clouds	27
3.1	Introduction	27
3.2	Illustrative Use Case and Motivating Scenario	28
3.3	System Model	30
3.4	Problem Formulation	32
3.5	Proposed Solution	36
3.5.1	Task Grouping	36
3.5.2	GPU partitioning	40
3.5.3	Task Placement	41
3.6	Performance Evaluation	43
3.6.1	Experiment Settings	43
3.6.2	Evaluation Results	45
3.7	Conclusion	49
4	Efficient GPU Resource Allocation for Microservice Applications	51
4.1	Introduction	51
4.2	Illustrative Use Case and Motivating Scenario	53
4.3	System Model	55
4.4	Problem Formulation	56
4.5	Proposed Solution	58
4.5.1	GPU Memory Manager	58
4.5.2	Monitoring System	60
4.5.3	Online Performance Optimizer	61
4.5.4	GPU Resource Manager	68
4.6	Performance Evaluation	68
4.6.1	Prototype	68
4.6.2	Benchmark	69
4.6.3	Experiment Settings	69
4.6.4	Evaluation Results	72
4.7	Conclusion	79

5	Priority-based GPU Resource Allocation in Clouds	80
5.1	Introduction	80
5.2	Illustrative Use Case and Motivating Scenario	81
5.3	System model	85
5.4	Problem Formulation	86
5.5	Proposed Solution	90
5.5.1	Priority-Aware Task Scoring	90
5.5.2	Task Grouping and GPU Partition Selection	93
5.5.3	Priority-Aware Scheduling Algorithm	96
5.5.4	GPU Repartitioning and Scheduling Interaction	98
5.6	Prototype Description	99
5.6.1	Architectural Components	99
5.6.2	Execution Flow	101
5.7	Performance Evaluation	103
5.7.1	Kubernetes Prototype Evaluation	103
5.7.2	Trace-Driven Scalable Evaluation with Alibaba Workloads	107
5.8	Conclusions	113
6	Conclusions and Future Work	114
6.1	Conclusions	114
6.2	Future Work	115
6.2.1	Energy- and Cost-Aware GPU Resource Management	115
6.2.2	Predictive and Learning-Based GPU Scheduling	115
6.2.3	Resilient and Fault-Tolerant GPU Scheduling	116
6.2.4	Multi-GPU Execution for Microservice-Based Applications	117
	Bibliography	118

List of Figures

Figure 1.1 Typical NVIDIA GPU architecture [9]	8
Figure 3.1 Illustration of a GPU resource allocation in multi-tenant cloud environment	29
Figure 3.2 System Model	31
Figure 3.3 Overview of the proposed method	36
Figure 3.4 Distribution of GPU Resource Requests Over Time in Alibaba	44
Figure 3.5 A100 MIG Instance configuration policy	44
Figure 3.6 Distribution of Requests Over Time by User	46
Figure 3.7 Resources Underutilization During Task Execution for different number of users in the cluster	47
Figure 3.8 Max Tasks Normalization Time During Task Execution for different number of users in the cluster	48
Figure 3.9 Resources Underutilization During Task Execution for different number of GPUs in the cluster	49
Figure 3.10 Max Tasks Normalization Time During Task Execution for different number of GPUs in the cluster	50
Figure 4.1 Illustrative use case of for GPU resource allocation in for Microservice Applications	53
Figure 4.2 System Model	55
Figure 4.3 Overview of the proposed method.	59
Figure 4.4 Overview of the GPU memory manager.	59
Figure 4.5 Overview of the Online Performance Optimizer.	62
Figure 4.6 Characteristics of benchmark applications in terms of memory data transfer and kernel execution time	70
Figure 4.7 Normalized input process time comparison of various applications under consideration.	71
Figure 4.8 Normalized throughput comparison of various applications under consideration.	73
Figure 4.9 Normalized input drop rate comparison of various applications under consideration.	74

Figure 4.10 Normalized application throughput comparison of various applications under consideration.	75
Figure 4.11 Benchmark applications characteristics in term of CPU and GPU execution time.	76
Figure 4.12 Throughput of benchmark applications on a logarithmic scale. . . .	77
Figure 4.13 Normalized DAG application throughput comparison of various applications under consideration.	78
Figure 4.14 Overhead of the proposed method in benchmark applications.	79
Figure 5.1 System model	83
Figure 5.2 Time-based view of GPU utilization across two priority classes, showing interaction between guaranteed and best-effort workloads	84
Figure 5.3 Time-based view of GPU utilization within one class showing unfair slowdowns	85
Figure 5.4 Overview of the proposed priority-aware GPU sharing framework implemented within the scheduler component of the orchestrator control plane shown in Figure 5.1	91
Figure 5.5 Prototype Architecture.	100
Figure 5.6 Mean admission delay by scenario and class.	106
Figure 5.7 Distributions of the execution time for two users	108
Figure 5.8 Scalability experiment results with 8 GPUs and mixed workload: (a) GPU utilization, (b) average NCT, (c) maximum NCT.	109
Figure 5.9 Scalability experiment results with GPU scaling 50/50 mix workload: (a) GPU utilization, (b) average NCT, (c) maximum NCT.	110
Figure 5.10 Max-NCT during Task execution for different numbers of GPUs in the cluster	112

List of Tables

Table 2.1	Comparison of the related work on GPU Allocation Methods for GPU resource allocation.	20
Table 2.2	Comparison of the related work on GPU Allocation Methods for Microservices.	23
Table 2.3	Comparison of the related work on priority-based GPU scheduling.	26
Table 3.1	Notations used in the problem statement.	34
Table 4.1	Hardware/software setup and benchmark specifications.	70
Table 4.2	Component categories based on kernel execution time percentage.	72
Table 4.3	Specifications of the microservice-based applications.	72
Table 4.4	Components categories based on SM activity percentage.	75
Table 5.1	Workload Classes, Types, and Runtime Characteristics	82
Table 5.2	Summary of main notations	87
Table 5.3	End-to-end execution flow corresponding to the numbered steps in Figure 5.5.	102
Table 5.4	Baseline prototype results (no contention).	105
Table 5.5	Contention scenario with preemption.	105
Table 5.6	Mixed workload scenario with 10 tasks over 30 seconds.	106
Table 5.7	Priority-order preservation and preemption metrics.	106

List of Abbreviations

ACL	Access Control Lists
AI	Artificial Intelligence
API	Application Programming Interface
CCV	Consensus Characteristic Values
CP	Critical Path
CRD	Custom Resource Definition
CUDA	Compute Unified Device Architecture
DAG	Directed Acyclic Graph
DFS	Dynamic Fairness Scheduling
DGRA	Dynamic GPU Resource Allocation
DNN	Deep Neural Network
DPP	Data Processing Pipelines
EDF	Earliest Deadline First
FIFO	First-In-First-Out
FIRM	Fine-grained Resource Management
GPGPU	General-Purpose Graphics Processing Unit
GPU	Graphics Processing Unit
HPP	High-Performance Applications
IAS	Image Analysis Systems
IPC	Instructions Per Cycle

LLM	Large Language Model
LP	Linear Programming
MIG	Multi-Instance GPU
ML	Machine Learning
MPS	Multi-Process Service
NCT	Normalized Completion Time
NLP	Natural Language Processing
OpenCL	Open Computing Language
PC	Partition Combination
PL	Priority Level
QoS	Quality of Service
RAN	Radio Access Network
RApp	RAN Application
RMS	Rate Monotonic Scheduling
SIMT	Single Instruction Multiple Thread
SJF	Shortest-Job-First
SLA	Service Level Agreement
SM	Streaming Multiprocessor
VDU	Virtual Distributed Unit
VNF	Virtual Network Function

Chapter 1

Introduction

1.1 Overview

Cloud computing has become the foundation of modern infrastructure, providing flexible, on-demand, and scalable access to computing resources. Among these resources, Graphics Processing Units (GPUs) play a central role in accelerating compute-intensive workloads, such as AI, ML, and microservice-based applications. Due to their high parallel processing capabilities, GPUs are increasingly used in cloud environments where large-scale, heterogeneous workloads require substantial computational power. However, the high cost and limited availability of GPUs in cloud environments necessitate efficient resource allocation strategies that optimize utilization without sacrificing performance or security.

Unlike CPUs, which are designed for general-purpose multitasking with mature operating-system-level schedulers, GPUs follow a fundamentally different execution and memory model. CPU schedulers rely on fine-grained context switching and lightweight preemption to enable efficient time-sharing among processes. In contrast, GPU kernels often execute for longer durations with limited preemption support, and GPU resources such as compute cores, memory bandwidth, and on-chip memory are tightly coupled. As a result, applying traditional CPU scheduling principles directly to GPUs can lead to inefficient resource fragmentation, unpredictable performance interference, and reduced utilization. These architectural differences require specialized GPU-aware allocation and partitioning strategies rather than conventional CPU-based methods. Moreover, existing methods for GPU scheduling and allocation face several critical challenges.

GPU underutilization is one of the most recognized challenges in cloud computing [10]. This issue arises when many workloads are allocated entire GPUs to meet QoS requirements despite requiring only a fraction of the GPU’s capacity [11]. Consequently, a significant portion of the GPU remains idle, resulting in suboptimal resource utilization. In addition, in multi-tenant cloud environments, it is crucial to enforce strong isolation to prevent performance interference and security vulnerabilities. While software-based multitasking methods allow for concurrent execution of tasks on the same GPU, they

often fall short of providing isolation guarantees. On the other hand, hardware-based partitioning offers stronger isolation but suffers from limited granularity and flexibility, making it challenging to allocate resources efficiently for smaller tasks. Fairness in resource allocation among competing users in a multi-tenant cloud environment is another long-standing challenge. Traditional scheduling methods such as First-In-First-Out (FIFO), Shortest-Job-First (SJF), and MISO [12] often overlook resource demand and completion time. As a result, long-running or high-demand tasks can monopolize access to GPUs, leading to increased latency and unfair treatment of lighter or shorter jobs.

The emergence of microservice-based cloud applications introduces additional complexity. These applications are composed of multiple interdependent components, often forming sequential or directed acyclic graph (DAG) structures. Each component may have different execution times and resource requirements, and their performance is highly tied to others. Assigning equal GPU resources to each component can lead to inefficiencies and bottlenecks, especially when specific components require significantly more processing power than others. Furthermore, in microservice or component-based applications, the overhead caused by frequent data transfers between the CPU, GPU, and the components themselves poses a critical challenge. These frequent transfers introduce latency, increase contention for memory bandwidth, and can significantly degrade overall application throughput, particularly when components are interdependent and exchange large volumes of intermediate data.

The problem of managing GPU resource allocation is further intensified when real-time or latency-sensitive workloads are involved. In these contexts, the cloud system must support priority-aware scheduling that ensures timely execution of high-importance tasks such as inference while still providing fair access to lower-priority jobs like training. Existing GPU schedulers often lack efficient mechanisms for prioritizing jobs based on urgency or workload context. As a result, this can lead to performance degradation or even starvation of critical services during high-demand periods. These challenges not only reduce GPU utilization efficiency but can also increase operational costs for cloud providers, degrade user experience, and lead to violations of Service Level Agreements (SLAs) in latency-critical applications. Although several GPU sharing and scheduling techniques have been proposed, many existing approaches struggle to simultaneously provide high utilization, strong isolation, fairness, and low-latency guarantees under dynamic and heterogeneous cloud workloads. Therefore, designing robust GPU allocation mechanisms remains an open research problem.

In general, GPU resource allocation in cloud environments involves a variety of sub-problems, each rooted in distinct requirements and technical constraints. These sub-problems can be viewed across three interconnected levels of abstraction. At the cluster level, the challenge concerns managing GPU resources in multi-tenant environments. In such scenarios, the key challenges include ensuring fair resource access and strong isolation

among competing users while maximizing overall utilization. At the application level, the focus shifts to the structure of cloud applications. Particularly, cloud-native microservice architectures, where interdependent components require workload-aware and pipeline-aware resource allocation strategies to prevent bottlenecks and throughput degradation. Finally, at the runtime level, the problem involves executing heterogeneous workloads with priority awareness under real-time latency constraints, ensuring that latency-sensitive applications, such as real-time inference, are prioritized over less time-sensitive workloads, such as batch training.

This PhD thesis focuses explicitly on three sub-problems that reflect these core challenges: (1) efficient GPU resource allocation in clouds, (2) efficient GPU resource allocation for microservice applications and (3) priority-based GPU resource allocation in clouds that ensures timely execution of real-time tasks while maintaining fairness. The following subsections discuss these challenges in detail, outline the thesis contributions, provide background on key concepts, and present the thesis structure.

1.2 Challenges

This section discusses the main challenges associated with GPU resource allocation in modern cloud environments, especially in multi-tenant, microservice-based, and latency-sensitive environment.

Efficient GPU Resource Allocation in Clouds: Efficient GPU resource allocation in multi-tenant cloud environments presents significant challenges due to workload heterogeneity, dynamic task arrivals, and resource granularity constraints. Cloud workloads vary widely in execution duration and resource demand, ranging from short-lived inference tasks to long-running deep learning training jobs. A primary challenge is GPU underutilization, which occurs when tasks reserve entire GPUs even though they require only a fraction of the GPU’s compute or memory capacity. At the same time, co-locating tasks from different tenants introduces performance interference and security concerns, requiring strong isolation mechanisms. Hardware-based partitioning techniques provide isolation but suffer from limited granularity, leading to fragmentation and inefficient allocation, while software-based sharing mechanisms enable concurrency but may compromise isolation guarantees. Furthermore, traditional scheduling policies such as FIFO or static partitioning fail to adapt to dynamic workloads, often resulting in long task completion times, unfair resource allocation, and suboptimal overall utilization.

Efficient GPU Resource Allocation for Microservice Applications: The adoption of microservice-based architectures introduces additional challenges for GPU resource management, particularly when applications are structured as synchronous pipelines or DAGs. In such systems, the performance of the entire application depends on the slowest microservice component, making balanced resource allocation across stages

critical. A key challenge lies in the heterogeneity of workloads across components, where different stages exhibit varying compute intensity, memory requirements, and sensitivity to latency. Static or equal-resource allocation strategies can create bottlenecks, leading to reduced end-to-end throughput. Moreover, microservice execution involves frequent CPU–GPU data transfers and inter-service communication, which introduce significant latency and PCIe bandwidth contention. These communication overheads, combined with fluctuating runtime behaviour and dynamic input characteristics, make it difficult to determine optimal GPU partitioning decisions. As a result, efficient resource allocation must account for both workloads and pipelines, dynamically adapting to component dependencies and processing rates to avoid imbalances and reduce throughput degradation.

Priority-based GPU Resource Allocation in Clouds: Priority-based GPU scheduling in multi-tenant cloud environments introduces significant challenges due to limited GPU availability, diverse workload classes, and strict service-level objectives. The first major challenge is ensuring tenant isolation while implementing priority policies across different workload classes, such as SLA-critical versus best-effort. This is essential for guaranteeing that high-priority workloads consistently achieve predictable performance, even when sharing GPUs with lower-priority tasks. Software-based GPU sharing mechanisms enable fine-grained concurrent execution, but they often struggle to provide strong isolation. An overloaded low-priority workload, for example, can consume excessive GPU compute resources or memory bandwidth, which can degrade the performance of SLA-critical workloads and violate priority guarantees. On the other hand, hardware-based partitioning enhances isolation by assigning dedicated GPU instances to specific workloads. However, the coarse granularity and static partitioning can be inadequate for addressing dynamic workload demands. This often leads to fragmentation and a lack of flexibility, making it difficult to respond quickly when urgent high-priority tasks require additional resources. The second major challenge is achieving fairness and efficient utilization under mixed workload durations and arrival patterns. Continuous SLA-critical services, such as online inference, must run continuously with stable performance, while other workloads, like batch analytics or ML training jobs, can be preempted. Under these heterogeneous conditions, existing scheduling policies, such as strict-priority queues, FIFO, or SJF, are often inadequate. These methods may over-serve short or high-priority workloads, leading to starvation for lower-priority tenants, and they often fail to ensure fairness among workloads within the same priority class. Therefore, effective GPU scheduling needs to support cross-class prioritization with strong isolation guarantees, as well as intra-class fairness across diverse workload types, and high GPU utilization in dynamic cloud environments.

1.3 Thesis Contributions

GPU resource allocation in cloud environments involves challenges at multiple levels. While efficient GPU allocation at the cluster level improves utilization and fairness among tenants, it does not fully address the requirements of modern cloud-native applications. In particular, microservice-based applications introduce additional complexities due to their pipeline structure and interdependent components. In such applications, performance depends on the coordination among multiple services rather than the efficiency of individual task execution. Consequently, allocation strategies designed for general workloads may lead to bottlenecks and throughput degradation in microservice pipelines. Furthermore, latency-sensitive workloads introduce additional requirements related to priority enforcement and deadline awareness. Therefore, this thesis addresses GPU resource allocation from three complementary perspectives: cluster-level allocation efficiency, application-level optimization for microservices, and priority-aware scheduling for latency-sensitive workloads.

This section presents three key contributions that address GPU resource allocation challenges across different system levels in cloud environments. The contributions progressively address efficient GPU allocation in multi-tenant cloud, application-aware GPU resource management for microservice-based workloads, and priority-aware scheduling for latency-sensitive tasks.

1.3.1 Efficient GPU Resource Allocation in Clouds [1][2][3]

The first contribution addresses the challenge of underutilized GPU resources and suboptimal task completion times in multi-tenant cloud environments. We propose an efficient GPU resource allocation technique integrated with an optimized task scheduling framework to maximize GPU utilization while minimizing task completion delays. Our approach leverages software- and hardware-based GPU partitioning, combined with an LP-solver-based optimization strategy, to dynamically allocate resources and reduce fragmentation. Experimental results demonstrate that the proposed method guarantees at least an 88% improvement in GPU utilization over state-of-the-art techniques, while also significantly reducing the maximum normalized task completion time. Furthermore, we evaluate the scalability of the proposed approach under two scenarios: scaling the number of users and scaling the number of GPUs. The results show that the method adapts effectively to increasing workload demand and maintains high utilization as GPU capacity grows.

1.3.2 Efficient GPU Resource Allocation for Microservice Applications [4][5][6][7]

The second contribution focuses on GPU resource allocation for microservice-based applications executed via spatial multitasking, where multiple concurrent components

compete for GPU compute resources and data transfer bandwidth. We propose an efficient GPU resource management technique that dynamically allocates GPU resources using a low-overhead online performance optimizer. In addition, we introduce a GPU memory-sharing mechanism based on shared memory to reduce data transfer overhead between microservice components and improve throughput. Extensive evaluation on applications with diverse resource requirements demonstrates that the proposed dynamic allocation method achieves performance improvements of up to 134.12% compared to Slate, a state-of-the-art spatial multitasking approach. Moreover, the shared-memory-based communication mechanism achieves up to a $6\times$ improvement in throughput over baseline UDP-based communication.

1.3.3 Priority-based GPU Resource Allocation in Clouds [8]

The third contribution addresses priority-based GPU scheduling in multi-tenant cloud clusters hosting mixed workload classes, including SLA-critical continuous services and time-limited best-effort workloads. We propose a priority-aware GPU sharing framework that satisfies two key requirements: enforcing tenant isolation while preserving configured class-level priorities, and achieving fairness within each class while maintaining high utilization. Our framework integrates three key components: (i) a priority-aware scoring mechanism that globally ranks tasks across tenants, (ii) a feasibility-based partitioning and grouping step that maps workloads onto a set of hardware partitions, and (iii) a scheduler that admits tasks and applies targeted preemption when necessary. We implement this framework in two complementary forms: a Kubernetes-integrated prototype and a large-scale simulator evaluated using the Alibaba GPU trace. Experimental results show that the proposed approach achieves approximately 97% GPU utilization and consistently outperforms baseline methods. On average, the proposed method improves normalized task completion time by approximately 76% across different GPU configurations and workloads.

1.4 Background Information

This section provides an overview of key concepts and technologies relevant to the scope of this work. It begins with a discussion of cloud computing and its role in enabling scalable, on-demand access to shared computing resources. Subsequently, the focus shifts to GPU sharing in cloud environments, highlighting the techniques for sharing GPUs among multiple applications and the challenges they introduce. The section then presents the fundamentals of GPU architecture and execution flow, followed by an overview of GPU multitasking techniques that enable concurrent workload execution through software- and hardware-based mechanisms. Finally, the section explores microservice-based applications and resource management, followed by priority-based scheduling and real-time resource allocation.

1.4.1 Cloud Computing

Cloud computing is a transformative paradigm that offers on-demand access to shared pools of configurable computing resources consisting of infrastructure (e.g., server, storage), platform and tools (e.g., databases, development frameworks), and software over the Internet[13]. Its defining characteristics include cost efficiency due to lower hardware access costs using a pay-per-use basis, the ability to take advantage of new hardware and software technologies, simplified software installation and control versioning, data sharing and fosters data accessibility and collaboration, high scalability and elasticity of resources to adapt quickly to changing demands and avoiding wastage of resources and over-provisioning, and finally the eliminating and reducing hardware and software maintenance [14]. The specific characteristics of cloud computing make it a highly flexible and scalable environment.

1.4.2 GPUs in Cloud Computing

GPUs are specialized accelerators designed to perform massively parallel computations. Unlike CPUs, which are optimized for sequential task execution and complex control logic, GPUs consist of thousands of lightweight cores that can execute the same instruction across multiple data elements simultaneously. This architecture makes GPUs particularly well-suited for compute-intensive workloads such as machine learning, scientific simulations, and large-scale data processing.

In cloud environments, GPUs have become an essential component for accelerating modern applications. Their ability to process large volumes of data in parallel enables significant performance improvements for workloads such as deep learning training, inference, and real-time analytics. However, GPUs differ fundamentally from CPUs in terms of architecture, memory hierarchy, and execution model. These differences introduce unique challenges for resource management and scheduling in shared cloud infrastructures.

1.4.3 GPU Architecture and Execution Flow

Modern GPUs follow the Single Instruction Multiple Thread (SIMT) architecture [9] that processes the same instruction simultaneously using tens of thousands of parallel processing threads. Threads on GPUs are organized into warps (e.g., 32 threads for NVIDIA GPUs and 64 threads for AMD GPUs), and warps are further partitioned into thread blocks [15]. NVIDIA GPU, as an example, is considered to illustrate the generic design of modern GPU architecture, as depicted in Figure 1.1. CPU (host) is connected to the GPU (device) via a PCI-Express bus and can allocate memory and transfer data by accessing device memories, including global, constant, and texture memories. NVIDIA GPU consists of several compute core clusters called streaming multiprocessors (SMs). Each thread block is assigned to one SM to operate a single instruction on different parts of data (e.g., processing a single operation on various matrix elements). The SIMT architecture enables applications written

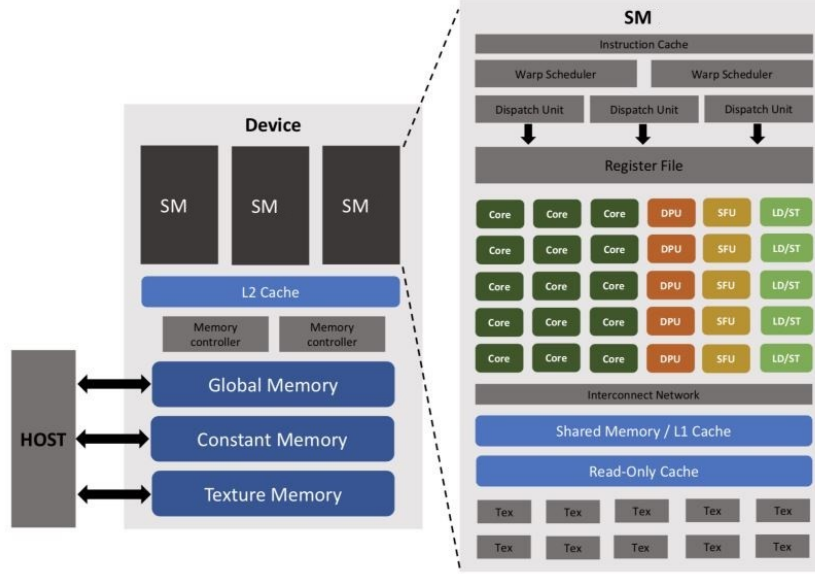


Figure 1.1: Typical NVIDIA GPU architecture [9]

in thread-level parallel code using General-Purpose Graphics Processing Unit (GPGPU) frameworks such as Compute Unified Device Architecture (CUDA) or Open Computing Language (OpenCL) to improve performance significantly [16]. These frameworks provide a broad set of libraries and Application Programming Interfaces (APIs) for managing GPUs, including managing computational resources and accessing GPU memory without dealing directly with lower-level controlling concepts.

GPU execution typically follows a multiple-step workflow known as offloading workload from CPU to GPU. First, memory is allocated on the GPU device using APIs provided by GPGPU frameworks such as CUDA or OpenCL. Input data is transferred from the CPU to the GPU over a PCIe bus. Kernels, code blocks designed for parallel execution to run on the GPU, are then launched and executed across GPU threads. Once execution is complete, the resulting data is transferred back to the CPU. This straightforward flow incurs overheads, especially when frequent data transfers are involved, contributing to inefficiencies in compute-intensive applications.

1.4.4 GPU Sharing in Cloud Environments

Sharing GPUs in cloud environments is defined as the ability to allocate and manage GPU resources among multiple concurrent workloads, enabling improved utilization and cost-efficiency without dedicating an entire GPU to a single task. Unlike CPUs, GPUs have historically been more difficult to share efficiently due to their monolithic design and tightly coupled memory systems. However, a comprehensive survey on GPU virtualization and scheduling [17] has classified techniques at the driver, library, and hardware levels,

highlighting approaches for both bare-metal and containerized deployments. These works also review scheduling methods, such as temporal, spatial, and hybrid scheduling designed to address fairness, performance isolation, and responsiveness in multi-tenant environments. Several techniques have emerged that allow multiple applications to share GPUs in bare-metal and containerized environments managed by orchestration platforms such as Kubernetes.

- **GPU Sharing in Bare-Metal Environments:** In bare-metal cloud deployments, GPU sharing can be achieved using two primary approaches: software-based and hardware-based mechanisms. Software-based approaches, such as NVIDIA’s Multi-Process Service (MPS), enable multiple applications to share GPU resources at runtime by scheduling them concurrently on the same GPU [18]. In contrast, hardware-based approaches, such as NVIDIA’s Multi-Instance GPU (MIG), provide physical partitioning of the GPU into isolated instances, each with dedicated computing and memory resources [19]. Both approaches allow concurrent execution of applications on a single GPU by assigning a fraction of the processing units to each task. This partitioning improves GPU utilization and application throughput, particularly when workloads cannot saturate the GPU on their own. However, these techniques involve trade-offs in terms of flexibility, configurability, and resource granularity. MPS may lead to contention under high loads, and MIG supports only a limited set of fixed partitioning configurations, restricting dynamic adaptability.
- **GPU Sharing in Containerized Environments:** In containerized environments, GPU access is typically managed through orchestration platforms such as Kubernetes. GPU orchestration is enabled through device plugins, custom scheduling policies, and resource management extensions that integrate GPU management into the container lifecycle. These tools allow fine-grained and dynamic allocation of GPU resources, supporting multi-tenant cloud use cases. In cloud environments, orchestration tools like Kubernetes enable GPU access and scheduling through device plugins [20]. These plugins provide information on available GPUs and support various sharing technologies, including software and hardware. The orchestrator selects a node based on resource availability and allocates a GPU for tasks using scheduling policies like first-fit or Round-Robin for even distribution. Together, these GPU sharing and orchestration techniques form the foundation for efficient, scalable, and cost-effective GPU usage in cloud environments, but they also introduce challenges related to fairness, performance isolation, and responsiveness, particularly when multiple tenants with diverse and dynamic workloads compete for limited GPU resources. [17].

1.4.5 GPU Multitasking Techniques

To maximize resource utilization and support concurrent application execution, modern GPUs implement multitasking mechanisms. These can be broadly categorized into temporal and spatial multitasking [21].

- **Temporal Multitasking:** Temporal multitasking involves time-sharing the GPU, allowing only one application to access the entire GPU at a time. While this model provides strong performance isolation and is ideal for latency-sensitive or high-priority tasks, it can lead to resource underutilization. In this technique, if an application requires GPU resources, it must wait to complete the running application. Preemption mechanisms allow access to GPU immediately [22], but it requires adding preemption points and context-switching [23] [24] and splitting kernels into a set of sub-kernels [25]. However, these mechanisms often introduce their own latency and complexity, limiting the responsiveness of GPU scheduling.
- **Spatial Multitasking:** Spatial multitasking [26] enables concurrent execution of applications by allocating portions of the GPU’s compute and memory resources to each task. Technologies such as MPS allow applications to share SMs, while MIG provides stronger isolation concurrency by partitioning GPU resources at the hardware level. However, these methods have limitations: MPS lacks fine-grained control over resource allocation [27] and does not support the reallocation of resources during the application execution [19]. MIG supports only static partitioning with predefined configurations, limiting flexibility in dynamic environments. Dynamic partition modification and reconfiguration are also not supported in this technology when the partition is not idle. While the spatial multitasking technique allows applications to run concurrently on the GPU, the policy of selecting the GPU resource portions for the concurrent applications has the potential for performance degradation. The allocated resource fraction for the application should be proportional to its requirements. The resource requirements of applications can be analyzed and detected using various methods, including offline profiling and monitoring [28], benchmarking [29], and kernel analysis. It is worth considering that dynamic and online partitioning is expected to yield better results mainly because it enables more adaptive resource utilization according to changing resource availability and workloads. Consequently, spatial multitasking requires intelligent scheduling policies to dynamically allocate resources and adjust the allocation based on current demands.

1.4.6 Microservice-Based Applications and Resource Management

In modern cloud-native environments, applications are increasingly designed using microservice architectures. Cloud-native applications typically decompose complex systems into a collection of loosely coupled services that can be independently deployed, scaled,

and managed through container orchestration platforms such as Kubernetes [30]. Each component runs in its own process and communicates through lightweight mechanisms such as RESTful APIs [31]. Recent systematic reviews indicate that while microservices architectures provide benefits like scalability, agility, and maintainability, transitioning from monolithic systems poses challenges. A key aspect of this transition is the effective identification of microservices. However, this process encounters issues such as the absence of standardized evaluation metrics, difficulties in defining context-sensitive service boundaries, and a lack of sufficient datasets for benchmarking [32]. Here, context-sensitive means that optimal service boundaries depend on the application’s specific characteristics, for example, in GPU-heavy workloads, boundaries might be drawn to minimize CPU–GPU data transfers.

In contrast, in financial systems, they might be optimized to reduce transaction latency. Poorly defined boundaries can increase communication overhead, degrade data locality, and exacerbate hardware resource contention, including GPU utilization inefficiencies in high-performance and data-intensive workloads. Microservice components, such as data processing services, authentication modules, or GPU-accelerated inference services, are often interdependent, meaning the overall application’s performance depends not only on individual component efficiency but also on their coordination and communication overhead. In cloud and data-intensive workloads, GPU resource management becomes particularly challenging, as components may have diverse computational demands. Performance bottlenecks can arise when critical components do not receive sufficient GPU resources or when allocation strategies are static rather than adaptive. Additionally, frequent data transfers between microservices, especially across the CPU–GPU interface, can lead to latency and bandwidth overhead. These inefficiencies are compounded if microservices are not optimally decomposed, highlighting the need for performance-sensitive identification techniques to support modern workloads such as machine learning inference or real-time analytics [32].

1.4.7 Priority-Based Scheduling and Real-Time Resource Allocation

Cloud systems increasingly support a wide range of workloads with different latency requirements and execution characteristics. While traditional cloud environments are designed to maximize throughput and resource utilization, many modern applications require real-time responsiveness, where tasks must complete within strict deadlines. Examples include autonomous systems, industrial automation, video analytics, and real-time inference services. In such scenarios, failure to meet timing constraints can lead to degraded service quality, violation of SLAs, and in critical domains, unsafe or incorrect system behavior. Consequently, cloud resource management must evolve beyond best-effort scheduling and incorporate priority-based and deadline-aware resource allocation mechanisms.

Priority-based scheduling refers to the process of assigning computational resources to tasks according to their importance, urgency, or service-level objectives. In cloud environments, priorities are often derived from workload types, user requirements, or application-level policies. For instance, real-time inference tasks are commonly assigned higher priority than batch-oriented training workloads because inference typically requires low latency and directly impacts end-user experience. Priority-aware scheduling mechanisms ensure that high-priority tasks receive preferential access to resources such as CPU, memory, storage, and accelerators (e.g., GPUs), particularly during periods of contention. Without such mechanisms, low-urgency tasks may consume a disproportionate share of resources, causing delays or starvation of latency-sensitive workloads.

Real-time resource allocation extends priority-based scheduling by incorporating timing constraints such as deadlines, worst-case execution times, and predictable response requirements. Classical real-time scheduling policies such as Rate Monotonic Scheduling (RMS) and Earliest Deadline First (EDF) have been widely studied in real-time operating systems. However, applying these techniques in cloud environments introduces additional challenges due to dynamic workloads, multi-tenancy, and unpredictable resource interference. Unlike dedicated real-time systems, cloud infrastructures must support heterogeneous applications that arrive dynamically, scale elastically, and share resources across multiple tenants. This complexity makes it difficult to guarantee deterministic performance, especially when resources are oversubscribed.

1.5 Thesis Outline

The remainder of this thesis is organized as follows: Chapter 2 outlines the requirements for GPU resource allocation in cloud environments and reviews related work. Chapter 3 presents an efficient GPU allocation and task scheduling framework for multi-tenant clouds. Chapter 4 introduces a workload-aware GPU resource management technique for microservice-based applications. Chapter 5 proposes a priority-based GPU scheduling framework to ensure isolation and fairness under heterogeneous workloads. Finally, Chapter 6 concludes the thesis and discusses future research directions.

Chapter 2

Requirements and Related Work

In this chapter, we first describe an illustrative use case that motivates the challenges of GPU resource allocation in multi-tenant cloud environments. Next, we use this use case to derive a set of general and solution-specific requirements for efficient, fair, and priority-aware GPU resource management, particularly in the presence of heterogeneous and microservice-based workloads. Finally, we review the related work in light of these requirements to position the contributions of this thesis.

2.1 Illustrative Use Case

In this section, we present an illustrative use case to derive the general requirements for GPU resource allocation in modern cloud environments. This illustrative use case is introduced to motivate the general challenges of GPU resource allocation in multi-tenant cloud environments and to derive the system requirements discussed in this chapter. Unlike the use cases presented in subsequent chapters, which focus on evaluating specific solutions proposed in this thesis, this scenario provides a unified view of the different workload characteristics and system constraints that motivate the research problems addressed throughout the thesis.

To better understand the challenges of GPU resource allocation in multi-tenant cloud systems, consider a cloud platform where multiple users submit GPU-accelerated workloads to a shared cluster of heterogeneous nodes. The data center consists of several compute nodes, many of which are equipped with one or more GPUs to support high-performance workloads such as deep learning training, inference, and data processing. Each tenant submits tasks independently, and workloads arrive dynamically over time with varying priorities and execution durations. For example, one tenant may submit a short-lived, latency-sensitive inference task that requires only a small fraction of GPU compute and memory but must complete within strict time constraints. In contrast, another tenant may execute a long-running training job that consumes a substantial portion of GPU capacity for extended periods. In such environments, users specify not only CPU and

memory requirements but also GPU resource demands and, in some cases, priority levels that define the urgency and importance of their tasks. As a result, the scheduler must simultaneously consider resource demand, execution duration, and workload criticality when making allocation decisions.

A centralized orchestrator (e.g., Kubernetes) is responsible for selecting suitable nodes and allocating GPUs to incoming workloads. However, naive scheduling strategies such as First Fit, Round-Robin, FIFO, or static partitioning often result in inefficient GPU utilization and unfair resource distribution. Allocating entire GPUs to small inference tasks leads to significant underutilization, while allowing long-running training jobs to monopolize resources may cause high-priority tasks to experience excessive queuing delays or even starvation. At the same time, co-locating tasks from different tenants on the same GPU introduces challenges related to performance interference, isolation, and data privacy. While software-based sharing mechanisms such as CUDA MPS enable concurrent execution, they may not provide sufficient isolation under contention. Alternatively, hardware-based partitioning approaches such as MIG offer stronger isolation guarantees but suffer from limited granularity, potentially leading to resource fragmentation. Therefore, an effective GPU resource management framework must balance utilization efficiency, strong isolation, fairness across tenants, and priority-aware scheduling to ensure that latency-sensitive workloads meet their performance objectives without degrading overall system throughput.

In addition to multi-tenant workloads, modern cloud applications increasingly follow microservice-based architectures, where user requests are processed through a set of interconnected components arranged in pipelines or DAGs. A representative example is a cloud-hosted object detection pipeline consisting of decoding, enhancement, and detection services that process streaming input frames. Each microservice performs a distinct stage of computation and may have different GPU requirements depending on the workload characteristics and input data complexity. Since the microservices are interdependent, the end-to-end performance of the application is constrained by the slowest stage in the pipeline. As a result, inefficient GPU allocation to a single component can introduce bottlenecks, propagate backpressure, and significantly reduce overall throughput.

This use case highlights the need for GPU resource allocation strategies that ensure strong isolation, efficient utilization, scalability, and fairness in multi-tenant cloud environments. Moreover, it emphasizes the importance of adaptive and fine-grained resource allocation for microservice-based applications, where inter-service dependencies, data transfer overhead, and latency constraints require coordinated scheduling decisions. In the following section, we derive a set of general requirements for GPU resource allocation in cloud systems based on the challenges identified in this illustrative scenario.

2.2 Requirements

In this section, we outline the general requirements for GPU resource allocation in cloud environments, considering the illustrative use case. First, we present general requirements that apply to GPU resource management in multi-tenant clouds. Next, we identify requirements for fair and adaptive scheduling in dynamic environments. We then describe the requirements for GPU resource allocation in microservice-based applications, where pipeline dependencies introduce additional challenges. Finally, we present the requirements for priority-based GPU scheduling to support latency-sensitive workloads. We then present solution-specific requirements addressing the challenges identified in Section 1.2.

2.2.1 General Requirements for GPU Resource Allocation in Clouds

The following requirements are essential for addressing GPU resource allocation issues in multi-tenant cloud environments:

- **Strong Isolation:** One of the main requirements of GPU resource allocation in multi-tenant cloud environments is strong isolation. The GPU resource management method must ensure strict execution isolation between tasks from different tenants to prevent performance interference and meet SLA commitments. Moreover, regarding Security and Data Privacy, the allocating decisions must preserve tenant data confidentiality, ensuring that memory isolation and access controls prevent data leakage or unauthorized access across co-located workloads.
- **Efficient Utilization:** GPU resource management must maximize GPU usage and avoid resource fragmentation and underutilization, especially when dealing with tasks with diverse and dynamic resource demands. This requires flexible and fine-grained resource allocation mechanisms that adapt to workload diversity.
- **Scalability:** The GPU resource management must efficiently handle a high volume of task submissions from many tenants, scale efficiently as tenant count and workload volume increase, and maintain responsiveness under load.

2.2.2 Requirements for Efficient GPU Resource Allocation in Clouds

While the requirements identified in Section 2.2.1 were general to GPU resource allocation in multi-tenant clouds, in this section, we present the requirements that are essential for achieving fair, adaptive, and efficient GPU allocation in dynamic multi-tenant cloud environments:

- **Adaptability and Dynamic Resource Management:** The resource allocator and scheduler should operate effectively in dynamic environments, where workloads arrive unpredictably at different times and tenant demands evolve over time. Static

assumptions or offline profiling are insufficient for such environments. The system must continuously adapt and adjust decisions to reflect changing tasks, system load, and fairness objectives.

- **Fairness:** The system should provide fair access to GPU resources across tenants and tasks, preventing long tasks from monopolizing resources or short tasks from starving.
- **Joint Scheduling and Allocation:** Given that the resource availability and scheduling order are tightly coupled, the system must perform integrated, co-optimized scheduling and allocation rather than handling them in isolation.
- **Hybrid Resource Partitioning:** The solution should support both software-based (e.g., MPS) and hardware-based (e.g., MIG) isolation, leveraging their combined strengths to balance flexibility, efficiency, and isolation.
- **Granularity-Aware Scheduling:** Allocation decisions must consider the granularity mismatch between task needs and available GPU partitions. The system should support dynamic resizing or reallocation to minimize resource waste and improve fit.

2.2.3 Requirements for Efficient GPU Resource Allocation for Microservice Applications

The following requirements are essential for addressing GPU resource allocation issues for Microservices in cloud environments:

- **Workload-Awareness:** The system must account for the varying computational needs of each microservice component and dynamically adapt GPU allocations based on real-time usage patterns and bottlenecks.
- **Pipeline-Aware and Throughput Optimization:** The GPU allocation framework should aim to maximize overall application throughput, particularly in synchronous microservice pipelines where one slow stage can throttle the entire workflow. In addition, the framework should be aware of inter-service dependencies and prioritize GPU allocations to avoid bottlenecks in tightly coupled pipelines.
- **Low Latency:** Real-time Microservice applications require rapid inference and minimal response times. These demands reducing GPU queuing delays, transfer overheads, and contention for compute resources.
- **Scalability:** As the number of microservices and incoming tasks scale, the system must maintain performance without significant degradation in scheduling decisions or resource utilization.

- **Dynamic Reallocation:** Unlike static partitioning or offline profiling, the system must support continuous adaptation of GPU resource allocations in response to fluctuating workloads or service-level demands.
- **Minimizing Data Transfer Overhead:** The GPU resource management system should reduce unnecessary CPU-GPU data transfers and intermediate data movement between components. Therefore, it can relieve PCIe bus saturation, preserve bandwidth for critical GPU workloads, and significantly reduce data transfer overhead.

2.2.4 Requirements for Priority-based GPU Resource Allocation in Clouds

The following requirements are essential for supporting priority-based GPU scheduling in cloud environments hosting heterogeneous workloads.

- **Priority Enforcement:** The scheduling framework must support explicit priority levels across workloads, ensuring that high-priority tasks (e.g., real-time inference) are served before lower-priority tasks (e.g., batch training). This enforcement is critical to guarantee responsiveness for latency-sensitive services under contention.
- **Latency Awareness:** The scheduler should consider latency constraints when allocating GPU resources. In real-time environments, scheduling decisions must minimize queuing delays and ensure that critical tasks complete within their required response-time bounds.
- **Preemption and Fast Resource Reallocation:** The system should support preemption or rapid reallocation mechanisms to reclaim GPU resources from low-priority workloads when urgent tasks arrive. Since GPUs are often shared among multiple tasks, the scheduler must ensure that priority adjustments can be applied.
- **Starvation Prevention:** While prioritizing urgent workloads, the system must also ensure progress for lower-priority jobs. Mechanisms such as aging or bounded waiting time should be incorporated to prevent starvation and guarantee long-term fairness.
- **Priority-Aware Utilization Optimization:** The scheduler should maximize GPU utilization while respecting priority constraints. Efficient resource packing and fine-grained allocation should ensure that prioritization does not lead to excessive fragmentation or idle GPU capacity.

2.3 Related Work

In this section, we present the current state of the art regarding the challenges identified in this thesis. We will review prior research on the three main contributions of this work.

First, we will examine the existing literature on GPU resource allocation in multi-tenant cloud environments. Next, we will analyze the related work on GPU resource allocation specifically for microservices. Finally, we will consider research on priority-based GPU scheduling in cloud environments hosting heterogeneous workloads.

2.3.1 Efficient GPU Resource Allocation in Clouds

In this section, we review the research work on GPU resource management in cloud. We categorize the existing GPU resource management into three categories: (i) GPU resource allocation, (ii) dynamic GPU task scheduling, and (iii) fair GPU task scheduling.

2.3.1.1 GPU resource allocation

The existing methods for allocating GPU resources are mainly categorized into temporal and spatial multitasking techniques [26].

1) Temporal multitasking: The concept of temporal multitasking refers to partitioning the time to allow one task to occupy and access the entire compute resources at a given time. Several research works have presented different methods for preemption to ensure that high-priority workloads can access the required resources immediately. GPUart [23], EffiSha [24], and nonPT [33] provide software-based mechanisms that enable kernel preemption by modifying the kernels of the applications and adding preemption points. To reduce the overhead of context switching and preemption, EffiSha and nonPT use persistent thread block mechanisms. FLEP [34] uses different priority queues to execute the kernels based on the shortest remaining time. This method transforms the kernels into the preemptable form by a compilation engine. Basaran et al. [35] and GPES [25] split kernels into a set of sub-kernels to allow kernel preemption between sub-kernel execution. On the other hand, Tanasic et al. [36] proposed a mechanism based on hardware extension, which uses the SM draining technique to wait for the completion of all the thread blocks in an SM. In temporal multitasking approaches, performance is adversely affected by context switching and preemption overhead. In addition, given that most tasks do not fully occupy the GPU resources, using temporal multitasking approaches leads to resource underutilization.

2) Spatial multitasking: spatial multitasking divides GPU resources among concurrent tasks, which boosts utilization and enhances overall system performance. Several works proposed policies for the spatial sharing of GPU resources and allocating proper SM fractions to concurrent kernels. [37], [38], [39] proposed dynamic spatial multitasking methods for concurrent independent applications, with a focus on guaranteeing QoS for individual kernels. In another study [40], the authors presented a spatial multitasking approach to minimize the cost of allocated GPU resources

while ensuring the latency SLA for deep neural network inference workloads co-located on the GPU. Services such as MPS [18] allow different tasks to share GPU resources simultaneously. However, while spatial multitasking improves efficiency, it may compromise task isolation. Recent advancements have introduced hardware-based sharing techniques, such as MIG [19] technology, which offers strong isolation by dividing a GPU into separate instances. While MIG provides robust isolation, it supports a limited number of partitions, which can lead to underutilization if the task requirement does not align with the available configurations. To address these challenges, some researchers have proposed methods that utilize hardware-based sharing techniques combined with learning-based performance predictors to estimate task performance on different GPU slices and determine optimal hardware partitions for co-located tasks [12]. However, relying solely on hardware-based technology may lead to GPU underutilization due to a limited number of feasible partitions supported by GPUs. ICEFROG [41] improves GPU utilization through adaptive resource control mechanisms by proposing a layer-elastic scheduling system. This method allows for overlapping computation and communication phases. However, ICEFROG relies on strong assumptions, such as a centralized infrastructure and static task structures, limiting its applicability in cloud-native environments.

2.3.1.2 Dynamic GPU Task Scheduling

In dynamic cloud environments with fluctuating task demands, schedulers must adapt using runtime techniques like auto-scaling, preemption, and workload-aware scheduling to optimize GPU utilization and meet resource needs. The proposed scheduling technique [42] sorts real-time tasks by their arrival time to maximize success rates while meeting deadlines. However, it can result in longer completion times and unfair scheduling due to delays in evaluating all options and neglecting task waiting times. MISO [12] employs FIFO scheduling to enhance overall system performance by minimizing task disruption through GPU hardware-based sharing. However, relying solely on this hardware limits partitioning, resulting in fewer concurrently executed tasks and increased completion times. The TGS scheduling technique in [43] maximizes task throughput by controlling the task dequeuing rate. However, this method results in longer waiting and completion times, leading to unfair scheduling. In dynamic task scheduling, notable progress has been made toward interference-aware execution models. PBS [44] introduces a scheduling framework for real-time GPU tasks. It utilizes a scheduling graph to identify feasible paths for parallel and sequential execution and ensures tasks meet their deadlines while maximizing parallelism. However, the PBS approach assumes complete knowledge of task arrival patterns and the absence of cross-tenant interference, conditions that are not realistic in cloud-based GPU environments.

Table 2.1: Comparison of the related work on GPU Allocation Methods for GPU resource allocation.

Work	Temporal Multitasking	Spatial Multitasking	Efficient Resource Utilization	Fairness	Isolation	Dynamic Resource Allocation
[23][24][25][35][33][34][36]	✓			✓	✓	
[37][38][39][40][41]		✓	✓		✓	
[12][44][42][43]		✓	✓			✓
[46][48][45][47][49]	✓	✓		✓		✓

2.3.1.3 Fair GPU Task Scheduling

Recent research on GPU task scheduling has shifted towards fair scheduling approaches, notably the min-max fairness policy, which maximizes the minimum resources allocated to each task [45], and tasks are weighted based on factors such as deadlines or priorities. However, many fairness scheduling methods rely on temporal and spatial multitasking techniques, which can lead to time-consuming preemption and inadequate task isolation [46],[47]. Tiresias [47] ensures fairness by minimizing average task completion time and using preemption to prevent short tasks from being delayed by larger ones. However, this approach incurs overhead from frequent data transfers to and from GPU memory and can lead to increased completion times due to continuous preemptions. Similarly, Themis [48] employs a bidding system for fairness scheduling to minimize maximum task completion time. Still, it is ineffective for short-running tasks, as they create significant allocation and de-allocation overhead. Moreover, Themis focuses solely on completion time, neglecting resource requirements. FGD [49] proposes a fair scheduling method that minimizes fragmentation and optimizes allocation rates by packing tasks. However, its reliance on software techniques restricts resource utilization and prevents task packing among different users. As a result, it fails to achieve maximum resource usage due to isolation policies in a cloud environment. Table 2.1 provides a summarized comparison of state-of-the-art approaches discussed in this section.

2.3.2 Efficient GPU Resource Allocation for Microservice Applications

The related work on GPU resource allocation for microservices, impacting overall application performance, can be broadly categorized into two key areas: (i) general microservice resource allocation, (ii) complex microservice resource allocation, and (iii) microservice data transfer overhead.

2.3.2.1 General Microservice Resource Allocation

Temporal multitasking allows tasks to access GPU resources in specific time slices [21][22][50]. However, this method often leads to underutilization since only one task can use

the GPU at a time. Preemption techniques have been introduced to enhance responsiveness [23][34][36]; however, these can result in significant overhead due to the high context-switching costs associated with GPU architectures. In addition, this issue poses a significant limitation for microservice-based applications with a sequence of dependent components since the latency of preemption overhead propagates throughout the chain. Given that most kernels do not fully occupy the GPU resources, using temporal multitasking approaches for microservice-based applications becomes feasible only if each component utilizes the entire GPU or the resource requirements are equal. Spatial multitasking approaches [37],[38],[39],[40] allow the execution of multiple components simultaneously. However, in microservice applications, components collaborate together, and resource allocation must be considered based on the overall system rather than individual components. Several works have explored combining spatial and temporal multitasking to improve GPU sharing. Slat [51] and [52] select complementary kernels for spatial sharing but fall back to temporal multitasking when suitable kernel pairs or shared SM assignments are unavailable. In both proposed methods, the latency of suspending the kernel execution propagates throughout the chain and degrades the overall performance. Several works, such as Thomas et al. [53], [54], and [51], classify applications as compute- or memory-bound to guide resource allocation or pairing, but fixed allocation strategies and selective component execution are insufficient for real-time microservice-based applications that require all components to run concurrently. Isolation profiling and predicting the fairness allocation [28][29][55] are not proper approaches in real-time microservice-based applications. None of the existing works consider balance processing between components, which not only penalizes the overall performance but also wastes GPU resources.

2.3.2.2 Complex Microservice Resource Allocation

In [46], Astraea proposes a microservice contention-aware GPU resource allocation to optimize resource utilization and minimize latency by predicting component performance. The Astraea method can deal with both linear dependencies and DAGs. It dynamically identifies the DAG structure’s critical paths (CPs) using a weighted longest path approach. However, this approach introduces additional overhead due to online training and optimization. The authors of [56] proposed an intelligent fine-grained resource management (FIRM) framework designed to minimize resource contention for microservices with complex dependencies. The FIRM uses tracing data to identify CPs and detect microservice instances causing latency, thus preventing latency violations. However, since it does not consider GPU resource utilization, adding GPU-specific metrics may increase the overhead. In [57], a workflow-specific resource allocation policy was proposed that leverages the structural information of a given DAG to find CPs and predict resource requirements to minimize latency. Similarly, [58] determines CPs using a path clustering heuristic to

identify the minimum time needed to complete workflow jobs with complex dependencies. Tasks on the CP are scheduled using the same resource to reduce communication overhead. Both approaches [57] and [58] utilize either profiling data or statistical offline analysis. Additionally, they only consider the structural information of DAGs for detecting CPs without adaptively capturing the changes that may occur in the identified CPs.

2.3.2.3 Microservice Data Transfer Overhead

Data transfer overhead in microservice-based applications may lead to performance degradation and unnecessary loads on the PCIe bus. The GPU postpones the kernel execution until the required data is received, a delay known as memory transfer overhead. Kato et al. [59] considered the impact of data transfers on GPU computation. They provided a data communication scheme to allow direct access to device memory using PCIe address space and memory-mapped method. This method reduces the data transfer overhead by mapping the system I/O address space to the virtual address space of the GPU. However, the analyses show that this method’s performance depends on the data size, thus making it suitable only for small data sizes [60]. In current GPUs, kernel execution and data transfer can be overlapped to reduce the memory transfer costs and improve overall performance. For instance, in NVIDIA GPU, CUDA streams take advantage of having separate engines for kernel execution and CPU-GPU memory copy to allow the performance of multiple CUDA operations simultaneously [61]. Recent GPUs improve stream parallelism with multiple kernel engines and multiple copy engines capable of overlapping memory copy in both host-to-device and device-to-host directions and processing multiple kernels concurrently [62]. In [35], memory copies are split into a series of copies, and data transfers overlap with the executions. A similar method is proposed in [63], which divides the data transmission into several chunks. Werkhoven et al. [64] proposed an analytical performance model to select the best strategy between the several different methods for overlapping the computation and transferring data. Although the presented techniques reduce the data transfer overhead, they are primarily practical for executing one or independent applications on GPU. In the case of multiple components working with shared data, frequent data transfers to and from GPU are still required. Moreover, data dependency between different components and limited GPU memory resources restricts the communication and execution parallelism. Table 2.2 provides a summarized comparison of state-of-the-art approaches discussed in this section.

2.3.3 Priority-based GPU Resource Allocation in Clouds

In this section, we review the research work on priority-based GPU scheduling in cloud environments hosting heterogeneous workloads. We need to address two key requirements: First, enforce tenant isolation while maintaining configured priorities across classes, and

Table 2.2: Comparison of the related work on GPU Allocation Methods for Microservices.

Work	Temporal Multitasking	Spatial Multitasking	Efficient Resource Utilization	Dynamic Allocation	Supports All Type of Dependencies	Low Transfer Overhead	Efficient Overall Performance
[21] [22] [50]	✓						
[23] [34] [36]	✓			✓			
[37] [38] [39] [40]		✓	✓				
[51] [52]	✓	✓					
[53] [54] [51]		✓					
[28] [29] [55]		✓					
[46] [56] [57]		✓	✓		✓	✓	✓
[59] [62] [63] [64]	✓	✓				✓	

second, achieve fairness within each class under varying workloads while ensuring high utilization. We refer to these two challenges as C1 and C2, respectively. The efficient management of GPUs in multi-tenant cloud environments has been explored from several perspectives. To connect previous research to these two requirements, we review the literature in three categories and indicate whether each category addresses C1 and C2. First, we review (i) GPU resource allocation and sharing mechanisms, including temporal and spatial sharing and hardware partitioning, along with their tradeoffs between isolation and utilization (partial coverage of C1). Next, we discuss (ii) dynamic GPU task scheduling and fairness (coverage of C2). Finally, we cover (iii) priority-aware GPU scheduling (partial coverage of C1).

2.3.3.1 GPU Resource Allocation and Sharing

GPU allocation techniques can be categorized into two types: temporal multitasking and spatial multitasking. In temporal sharing, tasks are allocated to the entire GPU for designated time slices [21], which simplifies isolation. However, this approach often leads to underutilization of the GPU, as only one task can execute at a time. While the preemption mechanism enhances responsiveness [23][65][34], it also introduces significant context-switching overhead on GPU architectures. On the other hand, spatial sharing allows multiple tasks to utilize the GPU simultaneously [66][67][29]. For instance, NVIDIA’s MPS supports concurrent task execution within a shared context [18], thereby improving utilization. However, this method can compromise task isolation.

To enhance isolation, hardware-based partitioning divides a GPU into multiple instances, each with dedicated compute and memory resources [12], [44]. Recent studies on hardware partitioning for real-time and composable workloads offer finer control compared to standard MIG configurations, while maintaining low overhead [68]. The UGPU framework creates unbalanced partitions that better align with the varying demands of heterogeneous jobs, thereby reducing internal fragmentation [69]. At the software level, SGDRC implements dynamic, software-defined partitioning of compute units and memory bandwidth, allowing for the isolation of latency-sensitive and best-effort Deep Neural Network (DNN) inference workloads on a single GPU [70]. At the cluster level, frameworks such as MuxFlow [71], EasyScale [72], and DELTA [73] adapt GPU allocations or compress models to improve utilization and throughput in production and edge environments.

Additionally, AdaGap formulates placement as a Markov decision process and employs deep reinforcement learning to minimize CPU/GPU gaps in heterogeneous clusters [74]. Finally, OEF [75] introduces a fairness-aware allocation for DNN training.

The allocation and sharing mechanisms discussed partially address C1 by improving isolation through temporal sharing and hardware partitioning, or by enhancing utilization through spatial sharing. However, they typically do not address C1 because they fail to enforce a class-based priority ordering during the admission or placement processes in an orchestration setting. Additionally, these mechanisms rarely account for the impact of discrete partition constraints. Moreover, they generally do not tackle C2 because they typically operate within a limited workload domain, often focused on deep learning training or inference. As a result, continuous arrivals of mixed workload types are not effectively enforced at the orchestration level. Even when these mechanisms reduce interference or enhance packing efficiency, they typically do not establish fairness objectives. Our work differs by jointly considering priority classes, tenant isolation boundaries, and the discrete partitions that constrain feasible allocations under mixed workload types.

2.3.3.2 Dynamic GPU Task Scheduling and Fairness

Dynamic GPU schedulers adapt to varying workloads by using ordering rules, preemption, and workload-aware dispatching. A recent survey offers a comprehensive overview of scheduling strategies for deep learning and large language model workloads in GPU clusters [76]. While simple heuristics that prioritize tasks by arrival time or other attributes [42], [43] can enhance throughput, they may also lead to longer completion times or unfair treatment, primarily when task sizes differ. MISO [12] combines hardware-based sharing with FIFO scheduling to minimize interference; however, its dependence on fixed partition structures limits concurrency. In heterogeneous or edge environments, real-time schedulers utilize optimization or learning-based policies to balance fragmentation, GPU utilization, and deadline satisfaction [77], [78]. For distributed deep learning training, deadline-aware online schedulers and topology-aware reinforcement learning approaches collaboratively decide on GPU placement and communication to meet task objectives in heterogeneous clusters [79], [80].

Another area of research focuses specifically on fairness. Tiresias [47] uses preemption to prevent short tasks from being delayed by long ones, thereby reducing the average completion time. However, frequent context switches and data movement can negatively impact long-running or memory-intensive tasks. Themis [48] employs a bidding mechanism among users to allocate GPUs, prioritizing completion time at the expense of negotiation overhead. FGD [49] attempts to minimize fragmentation through packing; however, its software-only approach restricts resource sharing across users due to strict isolation policies. FFT [81] proposes a fairness-compensated scheduler for heterogeneous GPU clusters, primarily targeting deep learning workloads. Dike [78] further enhances both efficiency and

fairness in multi-tenant GPU clusters for deep learning by integrating queue management with interference-aware placement. Additionally, systems like Pollux [82] and DCUDA [83] co-optimize scheduling with workload adaptation or migration to improve throughput and completion time, again focusing mainly on deep learning environments.

This category primarily targets C2 by handling continuous workload arrivals and fairness objectives. However, it often fails to meet C2 because it does not account for mixed workload types (such as continuous services versus time-limited tasks) and lacks support for explicit preemptibility and per-class fairness. It also often falls short of C1 because it does not treat strict tenant isolation under discrete hardware partitions as a key constraint, and class-based priority is either missing or not coordinated with partition selection or reconfiguration. In addition, most fairness schedulers treat all tasks uniformly and do not encode a class-based priority model. In our earlier work [2], we introduced a hybrid fair GPU allocation framework that improves utilization under isolation constraints by combining hardware and software-level partitioning. More recently, [1] incorporated a dynamic scheduler and monitoring loop to support repartitioning and fair reallocation in response to time-varying demand. However, these systems still do not address C1 because they do not incorporate class-based priority, and they do not fully address C2 because continuous SLA-critical services are not explicitly differentiated from preemptible, time-limited tasks. The current work extends this by introducing explicit priority classes, within-class fairness, and modelling different workload types.

2.3.3.3 Priority-Based GPU Scheduling

A third line of research focuses explicitly on priority-aware GPU scheduling. Various systems have been designed with interference-aware schedulers that determine which kernels or deep-learning operators to co-schedule on a GPU. These schedulers consider resource contention and utilize priorities to protect latency-sensitive tasks [84]. Additionally, some studies examine priority-based arbitration for communication and PCIe traffic among GPU tasks, applicable in both multi-GPU servers and distributed training clusters [85]. This approach avoids communication bottlenecks and ensures that accelerators remain busy [86].

In embedded, real-time, and edge settings, priority-aware frameworks map OS or application priorities to GPU (or NPU) execution. This alignment ensures that higher-priority models receive better timeliness. Examples include spatial-multitasking schedulers that jointly manage energy and deadline-miss rate on GPUs [87]. There are also real-time or edge controllers that adapt kernel-level scheduling to meet specific latency requirements [88]. Additionally, mechanisms exist to proportionally allocate inference performance according to the priorities of multi-level DNNs [89]. At the cluster scale, recent work such as GFS introduces preemption-aware policies that protect high-priority tasks while managing preemptible, low-priority tasks in large GPU clusters [90]. A recent survey provides a comprehensive taxonomy of such GPU scheduling algorithms, highlighting the

Table 2.3: Comparison of the related work on priority-based GPU scheduling.

Category	Isolation	Efficient utilization	Priority	Fairness	Mixed workload	C1	C2
Temporal multitasking [23],[65],[34]	✓	–	–	–	–	partial	–
Spatial multitasking [66],[29],[71],[72],[73],[74]	–	✓	–	–	–	–	partial
Hardware partitioning [12],[44],[68]	✓	–	–	–	–	partial	–
Dynamic scheduling [42],[43]	–	–	✓	–	–	partial	partial
Fairness schedulers [47],[48],[81]	–	–	–	✓	–	–	partial
Fairness and utilization optimizers [2],[1],[78]	✓	✓	–	✓	–	partial	partial
Priority-aware GPU scheduling [85],[89],[84],[87],[88],[90]	–	–	✓	–	–	partial	–
This work	✓	✓	✓	✓	✓	✓	✓

trade-offs among priority, fairness, and utilization [91]. This category focuses on C1 and prioritizes tasks, often to protect those sensitive to latency. However, it typically does not satisfy C1 because it does not enforce both tenant isolation and class priority when selecting among discrete partition layouts and placements. In addition, these solutions often operate at the kernel, runtime, or single-node level, assume fixed device assignments, and disregard orchestration-level constraints, such as tenant-scoped isolation boundaries. Furthermore, these approaches do not adequately address C2 because within-class fairness under continuous arrivals and mixed task types is rarely part of the objective. Moreover, the fragmentation effects of discrete partitions are not optimized alongside priority considerations. In contrast, our framework integrates priority classes and workload types into the GPU-sharing model at the orchestration level. It employs a two-layer fairness policy that applies both across and within classes and coordinates the execution of both continuous and preemptible workloads in multi-tenant cloud environments.

Overall, existing work tends to cover either isolation partitioning (C1-partial), fairness in dynamic scheduling (C2-partial), or priority enforcement (C1-partial), but rarely all of these aspects together. Our framework aims to address the intersection of isolation and strict class priority (C1), ensuring within-class fairness across continuous and mixed workloads, while maintaining utilization under discrete partition configurations (C2).

2.4 Conclusion

This chapter presented an illustrative use case and derived key requirements for GPU resource allocation in multi-tenant cloud environments. We then reviewed existing GPU scheduling, sharing, and resource management approaches and evaluated them against these requirements. Our analysis revealed that current solutions do not effectively achieve strong isolation, high utilization, and fairness under dynamic multi-tenant workloads. Additionally, they struggle to support workload-aware allocation for microservice pipelines and to offer priority-based guarantees for latency-sensitive tasks.

Chapter 3

Efficient GPU Resource Allocation in Clouds¹

3.1 Introduction

The growing demand for computational power in cloud computing has made GPUs essential for accelerating compute-intensive workloads. Given their high cost and limited availability, efficient GPU allocation is critical in multi-tenant cloud environments where multiple users or applications share GPU resources. While GPU sharing improves hardware utilization, it introduces new challenges, including resource contention, performance interference, and fairness in task scheduling. One of the primary challenges is ensuring efficient and isolation-preserving GPU allocation. In a multi-tenant cloud environment, tasks often vary significantly in resource demand. Without proper isolation, co-located tasks can interfere with each other's performance, leading to unpredictable execution and potential SLA violations [10], [11]. Existing software-based approaches (e.g., NVIDIA MPS) allow flexible sharing of GPU resources but may compromise task isolation under high contention [21]. In contrast, hardware-based solutions like NVIDIA MIG offer stronger isolation guarantees by partitioning GPUs [12]. However, these partitions have limited granularity and fixed resource boundaries, which can result in underutilization [44]. This becomes particularly problematic when tasks have varying resource requirements and fail to fully utilize the GPU's partition capacity. Thus, an effective allocation mechanism must combine strong isolation with flexible resource sharing to ensure efficient GPU usage.

The second challenge is designing fair and adaptive scheduling mechanisms that meet

¹This chapter is based on two published papers and one published Ericsson patent: [2] Hoda Sedighi, F. Wuhib, and R. Glitho, "Efficient task scheduling and allocation of gpu resources in clouds," in *IEEE International Conference on Network of the Future (NoF)*, 2024, pp. 115–123, [1] Hoda Sedighi, F. Wuhib and R. H. Glitho, "Dynamic Task Scheduling and Adaptive GPU Resource Allocation in the Cloud," in *IEEE Transactions on Network and Service Management*, vol. 23, pp. 1112–1127, 2026, and [3] Hoda Sedighi, F. Wuhib and R. H. Glitho, "A method for sharing gpu resources in multi-tenant cloud," WO2025219744A1, Patent published by *Telefonaktiebolaget LM Ericsson (Publ)*, 2025. Priority 2024-04-16, filed: 2024-04-16, published: 2025-10-23.

diverse task requirements, such as task deadlines or latency constraints and ensure fairness among tenants. Cloud workloads are highly heterogeneous, ranging from short-duration inference jobs to long-running training tasks. Traditional scheduling strategies such as FIFO and SJF, as well as several recent approaches [12], fail to achieve both fairness and responsiveness, especially in dynamic cloud environments where resource demands fluctuate continuously [46]. FIFO can cause short tasks to suffer long delays if long tasks dominate GPU resources. Conversely, approaches like SJF that prioritize shorter tasks can starve resource-intensive tasks and worsen inefficient resource allocation by not considering task resource requirements in their scheduling decision. The challenge becomes even more complex in dynamic cloud environments, where tasks with varying requirements are continuously added to the system over time. Static scheduling policies that assume all task demands are known upfront often fall short under such conditions [44], [46]. Therefore, dynamic scheduling approaches are needed to account for real-time changes in task demands and minimize waiting times. Existing solutions often address either efficient utilization or fairness scheduling separately but fail to integrate both aspects effectively, resulting in suboptimal performance. Given the NP-hard nature of this joint scheduling and allocation problem [92], there is a demand for a comprehensive GPU scheduling strategy that can jointly address fairness scheduling, preserve isolation in resource allocation and efficient resource utilization in cloud environments.

This chapter addresses these challenges by focusing on dynamic GPU resource allocation and requirement-aware task scheduling in multi-tenant cloud environments. We analyze how workload diversity, dynamic task arrivals, and resource granularity constraints impact allocation efficiency and fairness. We also discuss why existing approaches that treat allocation and scheduling separately fail to fully exploit GPU multitasking capabilities, particularly when fairness must be maintained over time. By addressing these issues, we highlight the need for adaptive frameworks that continuously reassess resource allocation decisions, support hybrid partitioning strategies, and minimize task completion delays while ensuring strong isolation.

The rest of this chapter is organized as follows. First, we present an illustrative use case that highlights the challenges of GPU underutilization, dynamic workload fluctuations, and fairness in multi-tenant clouds. Next, we describe the proposed dynamic task scheduling and adaptive GPU allocation algorithm, followed by its evaluation and comparison with state-of-the-art methods. Finally, we summarize the main findings and contributions of this chapter.

3.2 Illustrative Use Case and Motivating Scenario

As discussed in Chapter 2, GPU resource allocation in multi-tenant cloud environments must address challenges such as heterogeneous workloads, dynamic task arrivals, and

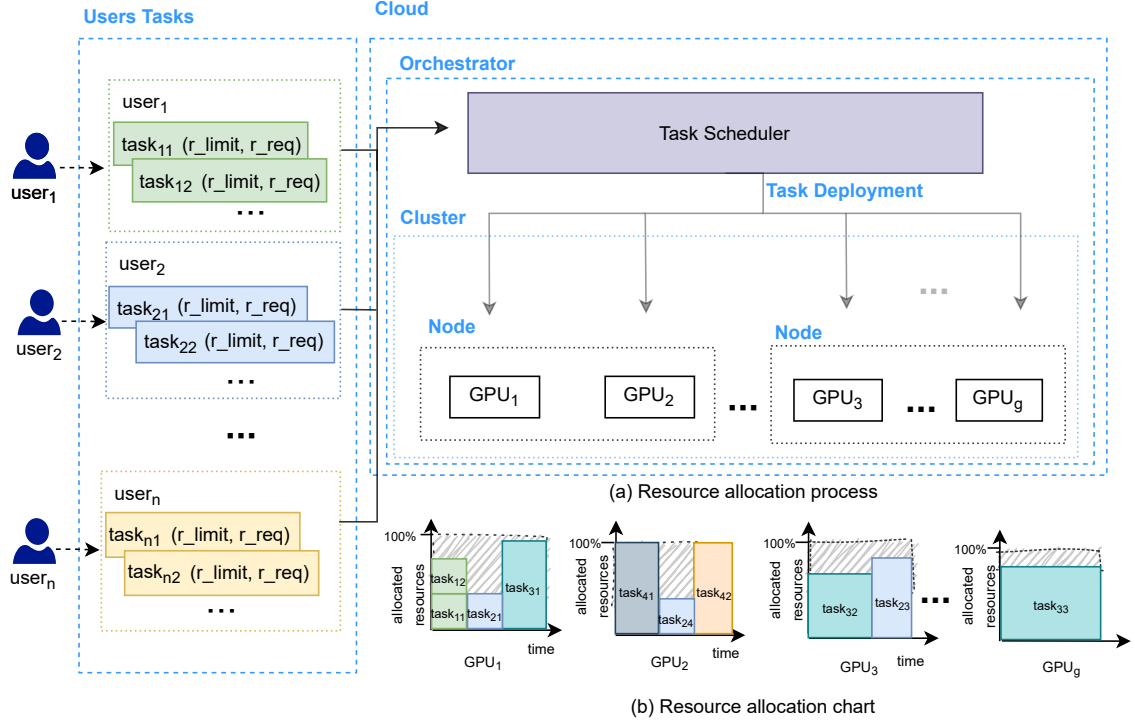


Figure 3.1: Illustration of a GPU resource allocation in multi-tenant cloud environment

efficient utilization of shared GPU resources. In this section, we present a use case that focuses on efficient GPU allocation among competing tenants in such environments. Figure 3.1 illustrates GPU resource allocation in a multi-tenant cloud environment, which serves as an example to motivate our research problem. Cloud users prefer to utilize cloud GPU resources, especially those with tasks requiring high computation demand, such as extensive data processing. Figure 3.1 depicts a simplified cloud system where users access resources in a data center with several nodes. Many nodes are equipped with GPUs to enhance processing capabilities. Typically, each node includes multiple GPUs, enabling high-performance computing across diverse workloads. This multi-tenant cloud platform hosts a diverse mix of GPU-accelerated workloads submitted by different users. Each user has a unique set of tasks that are colour-coded for differentiation, and these tasks arrive in the system at different times. For example, green represents tasks from user 1, blue indicates tasks from user 2, and so on. Each task has distinct resource requirements, reflecting the complexity of the cloud environment. For instance, a tenant may launch a short-lived inference task requiring minimal GPU memory and compute cycles. At the same time, another may run a long-duration deep learning training job consuming a substantial portion of GPU capacity. Accordingly, alongside specifying CPU characteristics (e.g., memory, cores), users can also define GPU requirements.

As shown in Figure 3.1 a, each task is linked to specific GPU resource requirements, with a maximum resource demand defined by cloud users [93]. Users cannot exceed the maximum

resources requested, and cloud resource providers must ensure they meet these limits. Moreover, tasks have varying completion times. Instead of relying on time-consuming runtime profiling [28][55], modern approaches employ proactive execution time prediction based on task features [94]. This enables immediate knowledge of execution time estimation upon task submission, facilitating prompt task scheduling without delays. As illustrated in the figure, task characteristics vary not only in GPU resource demand but also in execution duration. Long-term training tasks, such as those involving large AI models, can fully utilize GPUs by processing hundreds of samples in parallel [48]. In contrast, short-term tasks, such as medical image enhancement, may require only a fraction of GPU resources, often resulting in underutilization [46]. To manage such diversity effectively, a task scheduler is essential for assigning appropriate GPUs and optimizing resource allocation, thereby improving utilization while ensuring fairness. In cloud environments, orchestration tools such as Kubernetes facilitate GPU access and scheduling through device plugins [20]. These plugins report available GPUs and support both software- and hardware-based GPU-sharing technologies. The orchestrator selects a node based on resource availability and allocates GPUs to tasks using scheduling policies like First Fit or Round-Robin for load balancing. The resource allocation charts in Figure 3.1 b illustrate GPU sharing across nodes. Each chart shows the resource requirements of tasks, with gray areas representing unused GPU capacity over time. This highlights the need for a more adaptive and efficient allocation mechanism, as default scheduling policies may lead to increased resource waste. Naïve assignment strategies like First Fit or even resource assignments can result in large gray areas, which correlate with longer application completion times (x-axis) and increased resource underutilization (y-axis).

3.3 System Model

Figure 3.2 illustrates a multi-tenant cloud environment that provides high-performance computing through GPU-equipped data center nodes. This environment supports a variety of user workloads through an orchestration platform, such as Kubernetes, which manages both task scheduling and GPU resource allocation. Each node in the data center typically hosts multiple GPUs, enabling parallel execution of compute-intensive workloads. The orchestrator interacts with nodes via device plugins, which expose GPU availability and enable both hardware and software GPU sharing. Users submit tasks to the cloud platform with specified resource constraints, including CPU characteristics (e.g., number of cores and memory) and detailed GPU requirements (e.g., memory size and compute capacity). The system supports users with different computational needs, ranging from lightweight inference jobs to long-running deep-learning training tasks, creating a heterogeneous workload landscape.

Submitted tasks arrive at different times and are colour-coded to indicate their

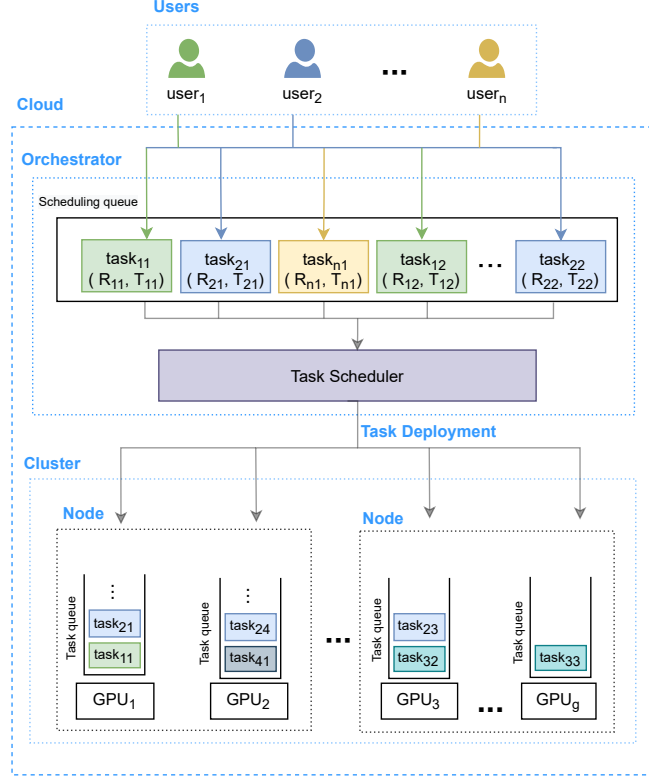


Figure 3.2: System Model

originating user. Each task specifies maximum GPU resource demands that cannot be exceeded during execution. These demands, declared at submission, reflect the task’s expected computational profile. Tasks vary in GPU utilization and execution time. For example, a medical image enhancement task requires minimal GPU resources and finishes quickly, whereas training a large-scale deep neural network may monopolize a GPU for an extended period. To avoid costly runtime profiling, modern approaches use execution time prediction based on task features, allowing the scheduler to make informed decisions immediately upon task arrival. The orchestration platform employs scheduling policies such as First Fit or Round-Robin to assign tasks to nodes based on current resource availability.

GPU resource allocation in the orchestration platform is managed through a combination of software-based and hardware-based mechanisms to enable efficient sharing of GPU resources. Software-based solutions, such as NVIDIA MPS, offer flexible and fine-grained GPU sharing by allowing simultaneous execution of tasks with varying sizes on a single GPU. However, this flexibility can compromise task isolation, leading to performance interference under high contention. Hardware-based solutions, such as NVIDIA MIG, provide stronger isolation by partitioning GPUs into multiple fixed instances with dedicated resources. Yet, MIG’s coarse partition granularity and fixed resource boundaries often cause resource underutilization when task demands don’t align well with partition sizes. Effective resource allocation thus requires combining the strengths of both approaches, ensuring

robust isolation while enabling flexible and efficient sharing to optimize GPU usage and fairness in multi-tenant clouds.

The available scheduling and GPU resource allocation mechanisms are often treated as separate solutions, which can lead to significant inefficiencies, especially in heterogeneous cloud environments. For example, assigning a long-duration, low-resource task can block another short-duration task for a long time, resulting in resource underutilization, longer application completion times, and reduced system throughput. In practice, scheduling and resource allocation are tightly coupled and must be addressed jointly to maximize performance and resource efficiency. However, jointly optimizing task placement and GPU assignment under varying task durations, resource demands, fairness constraints, and isolation requirements constitutes a complex combinatorial problem known as NP-hard. The search space grows rapidly with the number of users, nodes, and resource configurations, making exact solutions infeasible for large-scale systems. Therefore, effective GPU management requires adaptive scheduling and allocation mechanisms that approximate optimal decisions in real-time while balancing system throughput, fairness, and GPU utilization.

3.4 Problem Formulation

This section formalizes the resource allocation model for a multi-tenant cloud environment that prioritizes isolation. We formulate our cloud environment as a cluster containing a set of G GPUs. Each specific GPU, denoted as g , can be partitioned into a maximum of K_g hardware partitions. These partitions can be allocated to process user tasks, enabling efficient GPU services. The cloud environment serves N users, each requiring access to the shared GPU resources. Every user has a specific set of tasks to execute on the cloud system. Let M_n represent the total number of tasks for user n . For each task m belonging to user n , R_{mn} indicates the resource requirements. Additionally, each task has a dynamic arrival time, A_{mn} , which refers to the moment user n submits task m . Finally, T_{mn} indicates the estimated completion time for task m belonging to user n , reflecting the anticipated duration needed to finish the task once it starts running with the allocated resources.

In this setting, various users aim to enhance their task performance by utilizing shared GPU resources in a data center. Our goal is to dynamically allocate these shared GPU resources through a combination of hardware and software partitioning techniques, optimizing GPU utilization while ensuring that isolation requirements are satisfied. We also take into account the dynamic nature of task demands, acknowledging that tasks can arrive at varying times with diverse resource requirements. This necessitates real-time adjustments in resource allocation to accommodate incoming tasks, release resources from completed ones, preempt tasks to free resources as needed and optimize overall GPU efficiency. Our goals also include fair task scheduling by minimizing the completion time while continuously

adapting to changes in task demands. In the following, we outline our problem formulation. A summary of these notations can be found in Table 3.1.

Input Parameters:

- G : The total number of GPUs that are available in the cluster.
- N : The total number of users utilizing shared cluster resources throughout the entire period.
- M_n : The number of tasks submitted by each user $n \in N$.
- R_{mn} : The resource needs for task $m \in M_n$ belonging to user $n \in N$.
- A_{mn} : The dynamic arrival time of a task refers to the time when the user $n \in N$ submits task $m \in M_n$.
- S_{mn} : The time when the execution of task $m \in M_n$ for user $n \in N$ starts after the necessary resources have been provided to it.
- F_{mn} : The time when the task $m \in M_n$ for user $n \in N$ is completed, after which it frees up the resources.
- T_{mn} : The estimated time of completion task $m \in M_n$ for user $n \in N$ reflects the anticipated time required to finish task m once it starts running with the resources that have been allocated. This time spans from S_{mn} to F_{mn} .
- $W_{mn}(t)$: A proactive prediction determines T_{mn} as the anticipated execution time of a task when it operates in an isolated environment with a dedicated GPU. Nevertheless, the actual completion time of the task can vary significantly when tasks share GPU resources. This variation is caused by delays associated with assigning shared resources and the placement of tasks in the scheduling process [95]. We define $W_{mn}(t)$ as the metric that indicates the delay in scheduling placement. This metric reflects the waiting time for task $m \in M_n$ of user $n \in N$ at a given time t . The total duration of W_{mn} includes the interval from when the task is submitted A_{mn} until it begins execution S_{mn} , while the task remains pending. It also encompasses the period during which the task is halted due to resource eviction until the resources can be reassigned and execution can resume.
- C_{mn} : The time taken to complete task $m \in M_n$ for user $n \in N$ is defined as the duration between A_{mn} and F_{mn} . This duration comprises the waiting time before the task is scheduled and the anticipated execution time. This metric reflects how an unfair scheduling policy can result in increased waiting times and, as a result, longer completion times, which is expressed by the formula $C_{mn} = W_{mn} + T_{mn}$.

Table 3.1: Notations used in the problem statement.

Notation	Description
G	The total number of GPUs that are available in the cluster
N	The total number of users utilizing shared cluster resources
M_n	The number of tasks submitted by each user n
R_{mn}	The resource needs for task $m \in M_n$ belonging to user $n \in N$
A_{mn}	The arrival time of a task when the user $n \in N$ submits task m
S_{mn}	The time when the execution of task $m \in M_n$ for user n starts
F_{mn}	The time when the task $m \in M_n$ for user $n \in N$ is completed
T_{mn}	The estimated time of completion task $m \in M_n$ for user $n \in N$
$W_{mn}(t)$	The waiting time for task m of user n at a given time t
C_{mn}	The time taken to complete task $m \in M_n$ for user $n \in N$
K_g	The number of hardware partitions in a specific GPU $g \in G$
$y_{n,m,k,g}(t)$	Binary decision variable for allocating partition k to task m

- K_g : The number of hardware partitions in a specific GPU $g \in G$.

Decision Variables:

- $y_{n,m,k,g}(t)$: Is a binary decision variable that takes the value of 1 if task $m \in M_n$ of user $n \in N$ is allocated a hardware partition k of GPU $g \in G$ at time t ; it is zero otherwise.

Objective Function: Our goal is to minimize the maximum normalized completion time across all tasks while considering their resource demands and estimated completion times. By focusing on reducing the maximum normalized completion time, we aim to prevent significant delays for any single task compared to others. This approach is crucial for ensuring fair resource allocation and avoiding situations in which some tasks finish far earlier than others. The normalized completion time is defined as the total completion time of a task, including both waiting and execution time, divided by its estimated execution time. A normalized completion time close to 1 indicates that the task was completed as expected, while higher values suggest delays caused by resource contention or scheduling inefficiencies. The normalized completion time for a given task N_{nm} is calculated as follows:

$$N_{nm} = (C_{mn}/T_{mn}), \quad (3.1)$$

We define the continuous version of the normalized completion time to evaluate task execution efficiency over time. Given that there are $\sum_{n=1}^N M_n$ active tasks at a specific time t within the cluster, we define the continuous version of the normalized completion time for any task $m \in M_n$ belonging to users $n \in N$ as follows:

$$N_{nm}(t) = (C_{mn}(t)/T_{mn}) \sum_{g=1}^G \sum_{k=1}^{K_g} y_{n,m,k,g}(t) \cdot I(t \geq A_{mn}), \quad (3.2)$$

where $y_{n,m,k,g}(t) \in \{0, 1\}$ is a binary decision variable indicating whether task m from user n is assigned to resource unit k on GPU g at time t , and $I(\cdot)$ is the indicator function defined as:

$$I(t \geq A_{mn}) = \begin{cases} 1, & \text{if } t \geq A_{mn} \\ 0, & \text{otherwise} \end{cases} \quad (3.3)$$

This definition is used to determine the maximum normalized completion time, which serves as an objective for optimization. The indicator functions $I(t \geq A_{mn})$ ensures that only tasks that have arrived by time t are considered for scheduling. The algorithm executes iteratively upon the arrival of new tasks. If the cluster operates for a time interval of Δ , we can determine the maximum normalized completion time, serving as our objective function, using the following formula:

$$\text{Minimize : } N_{\Delta} = \int_0^{\Delta} \max_{1 \leq n \leq N} \max_{1 \leq m \leq M_n} N_{nm}(t) dt, \quad (3.4)$$

Although the formulation in 3.4 is expressed in continuous time, our system operates in an event-driven manner in which scheduling decisions are updated only at discrete event times (e.g., task arrivals, completions, or performance degradation events). As a result, the objective can be equivalently represented in a discrete form over these event times.

Constraints:

We implement constraint (3.5) as a cloud isolation measure to guarantee that each task is allocated exclusively to a single hardware partition during the time interval Δ :

$$\int_{A_{mn}}^{\Delta} \sum_{g=1}^G \sum_{k=1}^{K_g} y_{n,m,k,g}(t) dt \leq 1, \quad \forall n \in N, m \in M_n \quad (3.5)$$

The integration begins with the arrival time A_{mn} to reflect the dynamic nature of task arrivals. Additionally, we define constraint (3.6) to ensure that the resource requirements of all tasks $m \in M_n$ for each user $n \in N$ are adequately met.

$$\sum_{n=1}^N \sum_{m=1}^{M_n} R_{nm} \leq \int_0^{\Delta} \sum_{n=1}^N \sum_{m=1}^{M_n} \sum_{g=1}^G \sum_{k=1}^{K_g} R_{nm} \cdot y_{n,m,k,g}(t) dt \quad (3.6)$$

In addition we presents (3.7) to model the waiting time accumulates for unscheduled tasks:

$$W_{nm}(t + \Delta t) = W_{nm}(t) + \Delta t \cdot (1 - \sum_{g=1}^G \sum_{k=1}^{K_g} y_{n,m,k,g}(t)), \quad (3.7)$$

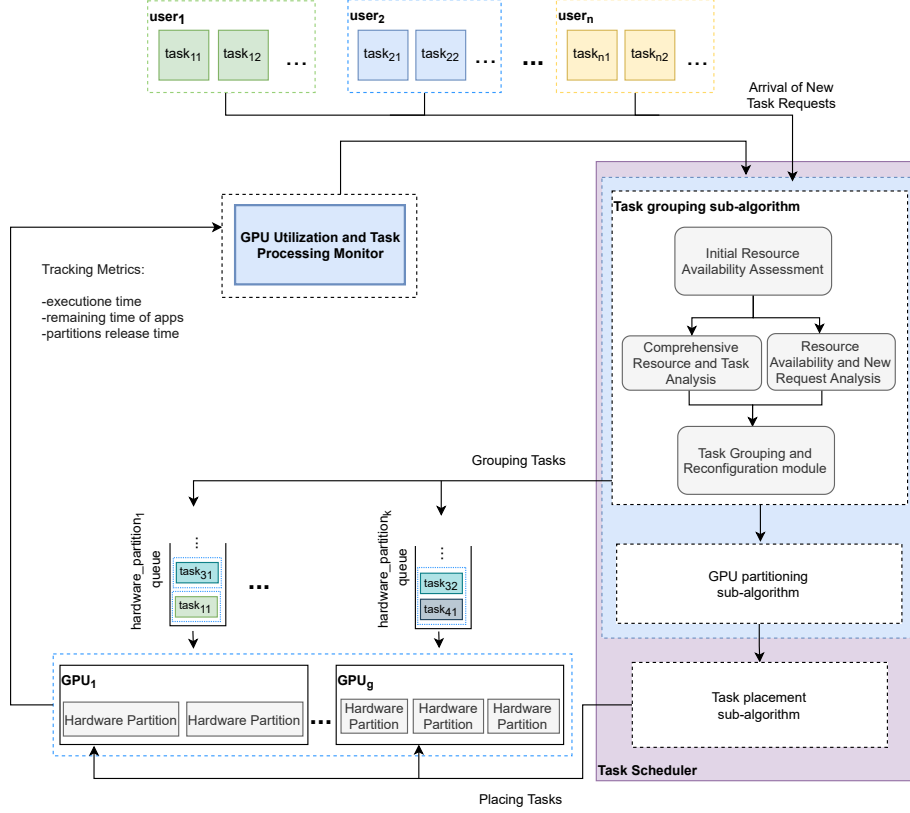


Figure 3.3: Overview of the proposed method

Finally, this constraint ensures tasks are not scheduled before their arrival time.

$$y_{n,m,k,g}(t) = 0, \quad t < A_{mn} \quad (3.8)$$

3.5 Proposed Solution

To tackle the problem described in Section 3.4, we proposed an approach encompasses three primary sub-algorithms: (i) Task grouping, (ii) GPU partitioning, and (iii) Task placement. An overview of this approach is illustrated in Figure 3.3. In the following, we provide a more detailed explanation of each phase.

3.5.1 Task Grouping

The Task Grouping Sub-Algorithm addresses the challenges of user isolation by calculating the necessary number and size of hardware partitions for each user using hardware partitioning. It also tackles inefficiencies in resource allocation by grouping and co-locating trusted tasks within a single partition through software-based techniques. This sub-algorithm receives the resource needs and completion time of each user's tasks as inputs

Algorithm 3.1: Task Grouping with Two-Routine Evaluation

- 1: **Inputs:** N List of users, M_n Number of tasks per user, R_{mn} Resource requirement for each task, T_{mn} Estimated execution time for each task, K Number of possible partitions in GPU, C_j Capacity of each hardware partition j , E_{mn} Remaining execution time of each task, W_{mn} Waiting time for each task, A_{mn} arrival time of each task
 - 2: **Output:** y_{mj} List of selected hardware partitions for each task and x_j the number of times hardware partitions j is selected
 - 3: **Step 1: Collect System State Information**
 - 4: **for** ($n = 1$ to N) **do**
 - 5: **for** ($m = 1$ to M_n) **do**
 - 6: Retrieve remaining execution time E_{mn} , waiting time W_{mn} , and assigned partitions.
 - 7: **end for**
 - 8: **end for**
 - 9: **Step 2: Execute Two Routines**
 - 10: **//Routine 1:** Call Direct Allocation (No Partition Modifications) with new set of submitted tasks and idle resources.
 - 11: min-norm-completion-time-1 = Task-grouping-sub-algorithm **Algorithm3.2**;
 - 12: **//Routine 2:** Call Reallocation and Repartitioning with all submitted tasks and all resources
 - 13: min-norm-completion-time-2 = Task-grouping-sub-algorithm **Algorithm3.2**;
 - 14: **Step 3: Compare and Apply Best Solution**
 - 15: Select the routine with the best fairness metric.
-

to identify the most suitable hardware partition and effectively group compatible tasks for co-location. The process of selecting a particular combination of tasks, which vary in resource requirements and execution times, is similar to bin packing[92], a well-known NP-hard problem. In this context, tasks correspond to items with sizes determined by their resource demands (e.g., GPU memory or compute units). At the same time, hardware partitions represent bins with fixed capacities imposed by GPU partitioning constraints. The objective is to assign tasks to partitions so that the total resource usage within each partition does not exceed its capacity, while minimizing the total number of partitions and reducing wasted capacity. This directly aligns with the classical bin-packing objective of minimizing the number of bins required to pack all items without exceeding bin capacity. Additionally, GPUs have constraints on partition sizes, and our proposed solution aims to optimize GPU resource consumption by minimizing the total number of partitions utilized. Alongside reducing resource consumption, we strongly emphasize minimizing the normalized task completion time and promoting fair resource distribution. To solve this optimization problem, we employ the Linear Programming (LP) approach.

Algorithm 3.1 depicts the Task grouping sub-algorithm. The algorithm aims to find the optimal hardware partition and group tasks to optimize GPU resource consumption. Upon receiving a new task or a set of new tasks, along with the relevant resource and time information (r, t), these tasks are placed in a queue for subsequent processing. Initially, the system gathers data on currently running tasks, which includes the release times of the GPU partitions in use, available resources, execution times, and the remaining time for running tasks (lines 3-8). This data is essential for determining how to accommodate the new tasks efficiently. A monitoring module is integrated into the system to gather task progress data and GPU partition status. The system then employs two routines to assess the normalized completion time as the fairness metric for optimal task placement and resource allocation.

Algorithm 3.2: Task grouping sub-algorithm

```
1: Inputs:  $N$  List of users,  $M_n$  Number of tasks per user,  $R_{mn}$  Resource requirement for each task,  $T_{mn}$ 
   Estimated execution time for each task,  $K$  Number of possible partitions in GPU,  $C_k$  Capacity of each
   hardware partition  $k$ ,  $E_{mn}$  Remaining execution time of each task,  $W_{mn}$  Waiting time for each task,  $A_{mn}$ 
   arrival time of each task
2: Output:  $y_{nmk}$  list of selected hardware partitions for each task and  $x_k$  the number of times hardware
   partitions are selected
3: for ( $n = 1$  to  $N$ ) do
4:   for ( $m = 1$  to  $M_n$ ) do
5:     for ( $k = 1$  to  $K$ ) do
6:        $TRR_k \leftarrow$  calculate based on Equation.5.9
7:       if ( $TRR_k \leq Thr$ ) then
8:          $\alpha_k \leftarrow 1$ ;
9:       else
10:         $\alpha_k \leftarrow 0$ ;
11:       end if
12:        $Minimize : \sum (C_k - \sum R_{nm} \cdot y_{nmk}) \cdot x_k \cdot \alpha_k$ ;
13:        $Minimize : \left( \frac{(T_{mn} - E_{mn}(t)) + W_{mn}(t)}{T_{mn} - E_{mn}(t)} \right) \cdot y_{nmk} \cdot \alpha_k$ ;
14:       if option  $k$  fits to minimize the values then
15:          $y_{nmk} \leftarrow 1$ ;
16:          $x_k \leftarrow ++$ ;
17:       else
18:          $y_{nmk} \leftarrow 0$ ;
19:       end if
20:     end for
21:   end for
22: end for
```

The two routines can run concurrently, leading to faster execution.

In Routine 1, if there are enough idle resources, the NP problem is addressed by calculating the normalized completion time based on the new tasks and considering only the available idle resources (lines 10-11). Routine 2, on the other hand, takes into account all GPU resources, whether they are free or in use, leading to a reassessment of the normalized completion time for both existing and newly introduced tasks (lines 12-13). Both strategies consider different scenarios and employ the task grouping sub-algorithm 3.2 to solve the NP problem. In Routine 1, which focuses solely on idle resources, new tasks can be assigned without any adjustments. However, in Routine 2, where all resources are considered, including those that are currently occupied, it may be necessary to re-partition or evict some tasks. After comparing the outcome of evaluating the fairness metrics for both routines, the algorithm will select the optimal solution for grouping tasks (lines 14-15).

The Task grouping sub-algorithm, illustrated in algorithm 3.2, is designed to group tasks for each routine and find the optimal hardware partition for the grouped task. It classifies incoming tasks according to the user requesting them. Subsequently, the algorithm operates concurrently for all users to guarantee efficient and prompt handling of each request (lines 3-22). Throughout this process, the algorithm takes into account the hardware partitioning constraints regarding both the quantity and size of the partitions. Specifying the restricted partition size options that the particular GPU model supports is crucial. Consequently, we define K as the number of potential partitions available in the GPU, which can differ depending on the model. We represent C_k as the resource of a specific hardware partition k ,

where $k \in K$. The algorithm executes a nested iteration to allocate an appropriate partition for each task. For each task (line 4), the process examines all potential hardware partition options (line 5). We define the formula of selecting the most suitable partition for each task with the goal of minimizing resource wastage in each partition as follows:

$$\text{Minimize} : \sum_{k=1}^K (C_k - \sum_{m=1}^{M_n} R_{nm} \cdot y_{nmk}) \cdot x_k \cdot \alpha_k \quad (3.9)$$

In Equation 3.9, the variable x_k represents how often a certain hardware partition $k \in K$ is chosen. Additionally, a binary decision variable y_{mk} is created to identify the specific options that reduce resource waste. The algorithm assesses each option with this formula to determine if it effectively minimizes Equation 3.9 (line 12). Note that the decision variable $y_{n,m,k,g}(t)$ defined in the system model represents the time-dependent assignment of task m from user n to partition k on GPU g at time t . However, in the task grouping sub-algorithm, the GPU index g is not yet determined. Therefore, we use a reduced binary variable $y_{n,m,k}$ to indicate whether task m from user n is assigned to candidate hardware partition type k .

By minimizing resource waste as outlined in Equation 3.9, this method allows tasks without isolation requirements to be co-located within the same hardware partition. This strategy optimizes the use of available space in a partition, avoids the need for separate hardware partitions for each task, and prevents resource underutilization. Moreover, Equation 3.10 (line 13) represents a normalized measure of time utilization for a task, considering both execution and waiting times from the task's arrival time A_{mn} until the current time t . This value serves as an indicator of fairness in scheduling. Minimizing this value guarantees that tasks receive appropriate execution opportunities while considering their waiting periods, preventing unfair delays in multi-tasking environments. For each potential hardware partition option, the corresponding variable y_{mk} is set to 1 if task $m \in M_n$ for user $n \in N$ is allocated to partition k , which occurs when it minimizes both Equation 3.9 and Equation 3.10; otherwise, it is assigned 0 (lines 14-19).

$$\text{Minimize} : \left(\frac{(T_{mn} - E_{mn}(t)) + W_{mn}(t)}{T_{mn} - E_{mn}(t)} \right) \cdot y_{nmk} \cdot \alpha_k \quad (3.10)$$

When tasks are grouped together, hardware partition underutilization may occur if the completion times of the tasks within a partition vary. In this situation, even though some resources within the partition can be released earlier, other tasks from different users must wait until all the tasks in that partition are completed before they can access those resources. As a result, the algorithm should group each user's tasks with similar execution times to minimize resource waste for a given hardware partition size. To effectively organize tasks with close execution times within a single partition, we utilize the Time Range Ratio (TRR) metric. This metric calculates the ratio range of task execution times within each partition (line 6). The TRR_k value, which is defined for $k \in K$, lies within the range of 0 to 1 and is

calculated using the following formula:

$$TRR_k : \frac{(\max_{m=1}^{M_n} T_{nm} \cdot y_{nmk}) - (\min_{m=1}^{M_n} T_{nm} \cdot y_{nmk})}{(\max_{m=1}^{M_n} T_{nm} \cdot y_{nmk})} \quad (3.11)$$

TRR_k is designed to generate a binary variable α_k , which reflects the influence of task execution time (lines 7-11). The algorithm assesses the execution time of each task against a defined threshold (Thr). This process aids in ruling out partition choices that contain tasks with a significantly high ratio range.

$$\alpha_k = \begin{cases} 1 & \text{if } TRR_k < Thr \\ 0 & \text{if } TRR_k > Thr \end{cases} \quad (3.12)$$

The threshold mechanism is intended to balance optimal resource usage and fair distribution. By raising the threshold value (Thr) towards 1, resource utilization is enhanced by allowing more partitioning options. Conversely, decreasing the threshold value to zero limits the number of partition choices but aids in removing low ratio rate options, leading to better task completion times. Once this phase concludes, tasks from each user are grouped to fit within designated hardware partition sizes, forming k hardware partition queues for the grouped tasks.

3.5.2 GPU partitioning

The GPU partitioning sub-algorithm is designed to identify the most efficient way to partition available GPUs using selected hardware partitions, minimizing resource wastage. It takes into account restrictions related to partition fragmentation imposed by GPU policies, including limits on the number of and allowable hardware partition combinations. Algorithm 3.3 details this routine, emphasizing its goal of optimizing GPU resource utilization.

Determining the best configurations for hardware partitions is difficult because of the complex constraints involved in selecting valid combinations. This issue can be viewed as a variant of an NP-hard problem. The GPU partitioning algorithm employs an LP solver to find the optimal selection of hardware partitions. This algorithm takes as inputs the list of chosen hardware partitions denoted as Y , the frequency of each selected partition $k \in Y$ from the task grouping sub-algorithm, and the list of potential configurations for hardware partitions, CN (line 1). Each configuration of hardware partitions represented as $c \in CN$, is made up of multiple partitions denoted by D_c . The quantity and size of each partition i within c are specified by N_{ic} and R_{ic} . To identify the optimal solution, the algorithm loops through all selected partitions (line 3). It examines every possible CN option (line 4), attempting to allocate the required number of partitions to the most suitable CN configuration (lines 5-14). We formulate Equation 3.13 to select the most appropriate CN for each chosen hardware partition, with the goal of minimizing resource waste in each

Algorithm 3.3: GPU partitioning sub-algorithm

```
1: Inputs: List  $Y$  of selected hardware partition sizes, List  $x_k$  of the number of selection times of the selected hardware partition  $k$ , List  $CN$  of possible hardware partition configuration in GPU,  $N_{ic}$  number of partition  $i$  within hardware partition configuration  $c$ ,  $R_{ic}$  size of partition  $i$  within hardware partition configuration  $c$ ,  $C_g$  all available resources of GPU
2: Output:  $PC$  list of optimal partition configurations from the selected hardware partitions
3: for ( $k = 1$  to  $Y$ ) do
4:   for ( $c = 1$  to  $CN$ ) do
5:      $\backslash\backslash$   $x_k$  is the number of selection times of the selected partition  $k$ 
6:     while ( $x_k > 0$ ) do
7:       if partition  $k$  is in the hardware partition configuration  $c$  then
8:          $Minimize : \sum (C_g - \sum_{i=1}^{D_c} R_{ic} \cdot N_{ic} \cdot y_i) \cdot PC_c$ ;
9:         if option  $c$  fits to minimize the value then
10:            $y_i \leftarrow 1$ ;
11:            $PC_c ++$ ;
12:            $x_k \leftarrow x_k - N_{ic}$ ;
13:         else
14:            $y_i \leftarrow 0$ ;
15:         end if
16:       end if
17:     end while
18:   end for
19: end for
```

CN. In this formula, the variable PC_c represents how often a specific hardware partition configuration, denoted as $c \in CN$, is chosen. Additionally, C_g refers to the total available GPU resources.

$$Minimize : \sum_{c=1}^{CN} (C_g - \sum_{i=1}^{D_c} R_{ic} \cdot N_{ic} \cdot y_i) \cdot PC_c \quad (3.13)$$

The algorithm assesses each configuration option $c \in CN$ through this formula to evaluate its effectiveness in minimizing Equation 3.13 (as indicated in line 8). For each configuration option c , the variable y_i is set to 1 if choosing partition i within option c contributes to minimizing Equation 3.13, and 0 if it does not (lines 9-16). By the end of this process, the GPU Partition Manager collects a list of optimal partition configurations.

3.5.3 Task Placement

In the final stage, the time-based fairness allocation strategy is employed. The Task Placement sub-algorithm receives two key inputs: a list of grouped tasks assigned to k hardware partition queues by the Task Grouping sub-algorithm and a set of optimal partition configurations determined by the GPU Partitioning sub-algorithm. Its primary function is to allocate the earliest available GPU time slot for executing the grouped tasks. Additionally, the algorithm prioritizes hardware partition combinations for partitioning available GPUs sequentially, ensuring efficient allocation, especially when the number of selected partition combinations exceeds the available GPUs. We have designed a heuristic approach to manage the Task Placement sub-algorithm efficiently, as described in Algorithm 3.4. To promote fairness, the algorithm first determines the Maximum Task Execution

Algorithm 3.4: Task Placement sub-algorithm

```
1: Inputs:  $PC$  list of optimal partition configurations from the selected hardware partitions,  $K$  hardware
   partition queues for the grouped tasks, List  $GP$  of list of grouped tasks,  $G$  all available GPUs
2: for ( $g = 1$  to  $GP$ ) do
3:    $MTET_g \leftarrow \text{Maximize} : (T_{mn} \text{ of all the tasks in } g) ;$ 
4: end for
5: for ( $i = 1$  to  $K$ ) do
6:   Sort the list of  $GP_i$  in the queue  $i$  based on the  $MTET$ ;
7: end for
8: for ( $p = 1$  to  $PC$ ) do
9:    $PL_p \leftarrow$  calculate based on Equation.3.14
10: end for
11: for ( $p = 1$  to  $PC$ ) do
12:   Sort the list of  $PC$  based on the  $PL_p$ ;
13: end for
14: while ( $GP > 0$ ) do
15:   for ( $n = 1$  to  $G$ ) do
16:     if  $n$  GPU  $n$  has available resources then
17:       choose the PC with high-priority;
18:       for all partitions in selected  $PC$  do
19:         choose the GP with high-priority;
20:          $GP \leftarrow GP - 1$ ;
21:       end for
22:     end if
23:   end for
24: end while
```

Time (MTET) for each grouped task (lines 2–4), representing the longest duration a group occupies a partition. Next, it sorts the grouped tasks within each hardware partition queue based on their MTET values, giving priority to those with shorter MTETs for earlier scheduling (lines 5–7). This method helps to minimize task completion times by minimizing overall waiting periods for all tasks.

Once the priority of grouped tasks has been established, the next step involves partitioning the GPUs based on the selected partition combinations (PC) designated to host these tasks. To effectively prioritize the PCs, we calculate and assign a priority level (PL) to each one and organize them accordingly (as outlined in lines 8-10). Each partition combination is made up of specific hardware partitions, each linked to a queue of grouped tasks. This phase involves a careful selection of the necessary grouped tasks for each partition within the PC, drawing them from the corresponding queue. The tasks in each partition's queue are sorted so that those with higher priority are selected first, promoting a fair scheduling process. Once enough grouped tasks are allocated for each PC, we compute their priority level (PL) using the following formula:

$$PL_p = (CCV_p + SPCO) \quad \forall p \in PC \quad (3.14)$$

The formula incorporates SPCO, short for Switch Partition Combination Overhead, which accounts for the time and computational effort needed to transition between different GPU partition configurations. Since modifying the current partition setup may necessitate a GPU restart, it can significantly impact the efficiency of the method. To mitigate this, SPCO is introduced as a factor that lowers the priority of a specific Partition Combination

(PC), thereby minimizing frequent reconfigurations and enhancing overall performance. Additionally, the Consensus Characteristic Values (CCV) of a partition combination assess how effectively a grouped set of tasks within a PC contributes to fair resource allocation. CCV is determined by identifying the maximum Mean Task Execution Time (MTET) across partitions within a given PC. By prioritizing PCs with lower MTET, CCV helps reduce the overall waiting time for all tasks, leading to more efficient scheduling.

The task allocation algorithm assigns a PL to each PC and arranges them in descending order based on their PL values (lines 11–13). Next, the available GPUs are partitioned sequentially based on the prioritized PCs (lines 15–17). The method then organizes the scheduled tasks within the GPU partitions according to their assigned priority (lines 18–20). As GPU resources become available, the algorithm iteratively cycles through the PCs, determining the next configuration for GPU partitioning until all task groups have been fully processed.

3.6 Performance Evaluation

In this section, we evaluate the proposed methods by conducting extensive experiments across various scenarios. To assess the effectiveness and scalability of our approach, we developed a simulator implementing the three sub-algorithms tailored for a multi-tenant cloud setting. The simulator includes several components: the task grouping module, GPU partitioning module, task placement module, monitoring module, and solver service. The solver serves as a solver for both Task grouping and GPU partitioning. The solver service in the simulator utilizes the CBC MIP solver from the Google OrTools Solver library, which assists in determining the optimal solution for our formulated model. To evaluate real-world scenarios, we incorporated the Alibaba cluster-trace-gpu dataset [96] in our evaluation, which captures AI and machine learning tasks with diverse resource specifications in a heterogeneous GPU cluster. The outcomes of the simulator are compared with four methods: MISO [12], SJF, and FGD [49]. Subsequently, we will outline the simulation setup, detailing the experimental setting and implementation specifics. Afterwards, we will discuss the evaluation outcomes and compare the performance of our proposal with existing solutions.

3.6.1 Experiment Settings

The simulator models a cloud data center environment with a cluster consisting of up to 64 NVIDIA A100-SXM4-40GB GPUs. It is important to highlight that the proposed method relies on the NVIDIA GPU for testing. However, our method is also compatible with other GPU models, including those that do not support hardware partitioning capabilities. The real-world Alibaba cluster-trace-gpu-v2023 dataset contains a total of 7064 GPU requests logged over 145 days. We have analyzed these requests to illustrate active requests

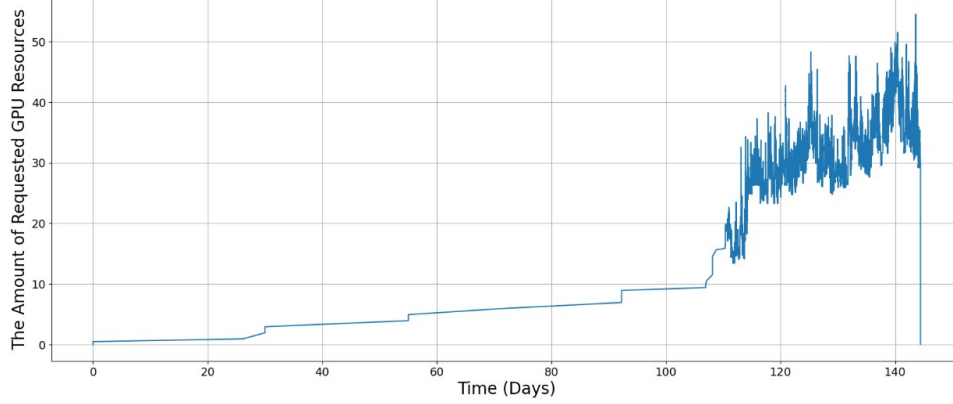


Figure 3.4: Distribution of GPU Resource Requests Over Time in Alibaba

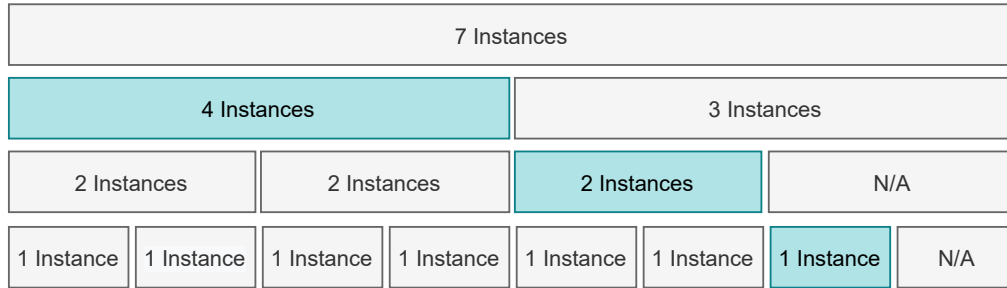


Figure 3.5: A100 MIG Instance configuration policy

throughout the duration. Accordingly, Figure 3.4 visualizes the distribution of the amount of GPU resource requests over time, illustrating the fluctuations in request volume during the observed period. The x-axis represents time in days, converted from seconds, while the y-axis indicates the amount of GPU requests at each time. The continuous line graph highlights trends in GPU request, with peaks representing periods of high demand and troughs indicating lower activity. This visualization provides insights into the temporal behaviour of GPU requests.

To ensure a comprehensive evaluation of scheduling efficiency, we analyzed two key real-world scenarios that assess the impact of user demand and GPU capacity dynamics:

- **Scaling Users:** We varied the number of active users from 1 to 8, simulating different levels of concurrent demand on the system. This scenario helps assess how effectively the scheduling mechanism handles increasing user requests and resource contention.
- **Scaling GPUs:** We adjusted the available GPU resources from 2 to 64, allowing us to analyze the system’s adaptability to changes in GPU availability. This scenario evaluates how effectively the scheduling approach utilizes varying GPU capacities to meet user demands.

A midpoint threshold of 0.5 (Thr=0.5) is established to balance resource utilization and

fair task scheduling.

Using A100 GPU with MIG technology enables the GPU to be divided into a maximum of 7 hardware partitions (instances), which can be sized at 1/7, 2/7, 3/7, 4/7, or the entire 7/7 of the GPU. Each partition possesses its own distinct GPU memory and SM. With the option for seven instances, the A100 can be arranged in a versatile manner, allowing for a single GPU with seven instances totalling 98 SMs and 40 GB of memory or various smaller instances. These configurations can consist of one instance with 14 SMs and 5 GB, two instances with 28 SMs and 10 GB, three instances with 42 SMs and 20 GB, or four instances with 56 SMs and 20 GB, providing adaptable computational capabilities.

In addition to the limitations on the size of instances, there are restrictions on the configuration of MIG instances; only a specific set of instance combinations is permitted in A100. Figure 3.5 showcases the allowed instances along with the configuration policy for A100 MIG instances. The instance selection policy for a valid combination is such that the first instance is chosen from the left side. Once a partition is selected, subsequent selections must avoid instances located below it. As a result, there are 19 different combinations of supported instances [55]. Given that a GPU unit represents the smallest partition in A100 GPU, and each GPU has a capacity of 7 units, the overall GPU capacity for this experiment is up to 448 units distributed across 64 GPUs. The simulator, encompassing resource allocation and scheduling services, is developed using C# 11 and dotnet 8.0.204 on a desktop with an Intel(R) Core (TM) i7-10700 CPU clocked at 2.90GHz and 16 GB of memory.

3.6.2 Evaluation Results

This section evaluates the effectiveness of the proposed method in allocating resources while minimizing both task completion time and resource underutilization. We will assess the proposed method in two real-world scenarios: scaling users and GPUs. To demonstrate its advantages in terms of efficiency and scalability, we will compare the proposed method with FGD [49], SJF, and MISO [12]. It is important to note that our simulation results are not directly comparable to earlier research that did not consider resource isolation, which is a critical factor in resource management and user security within a cloud environment.

3.6.2.1 User Scaling Scenario

In this scenario, the number of active users is increased exponentially from 1 to 8 to simulate rising demand as the number of tasks scales. We evaluated how different scheduling mechanisms handle resource contention by using 7064 requests based on a real-world Alibaba dataset, as shown in Figure 3.4. To focus on a stable task demands, we first removed the initial low-demand period. Additionally, to ensure that the requests come from different users and to simulate concurrent activity, we divided the request duration by the number of users and adjusted each user’s timeline to start from zero. This approach guarantees

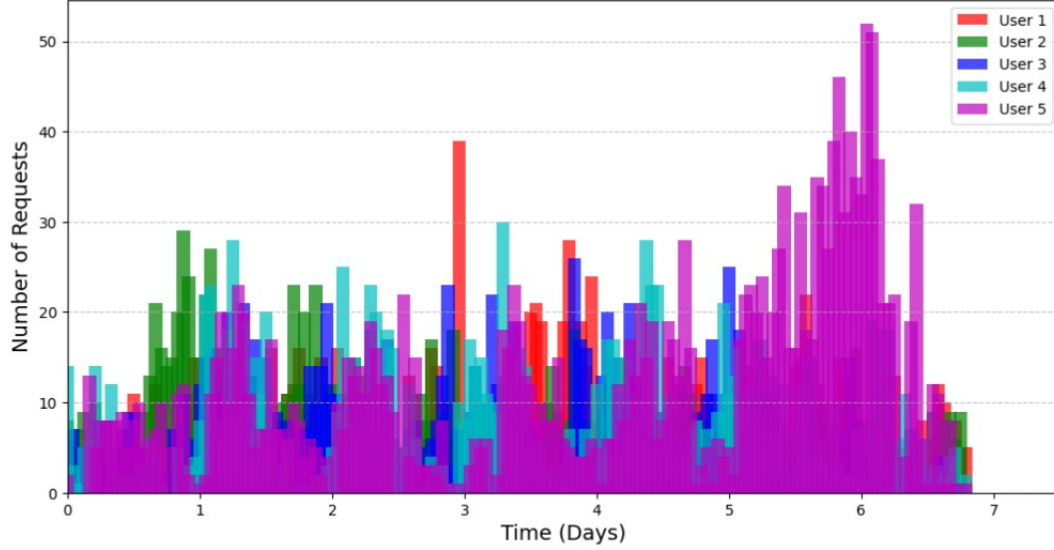


Figure 3.6: Distribution of Requests Over Time by User

that, during the experimental period, all users are actively submitting requests in parallel, reflecting a realistic multi-tenant GPU scheduling scenario. Figure 3.6 illustrates an example with five concurrent users, where each color represents the task requests from a specific user.

In the next scenario, each user can utilize up to 56 GPU units of resources from a total pool of 8 GPUs. The execution times for each task may vary, and the resource requirements for each task range from 1 to 7 units. Figure 3.7 compares resource underutilization across FGD, SJF, and MISO as the number of active users increases. Resource underutilization occurs when resources are not used efficiently or when fragmentation occurs, leading to higher costs and lower productivity. We calculate this resource underutilization by measuring the amount of idle resources, including fragmented parts, during the execution of tasks. This value can be obtained by multiplying the number of idle units by their duration of idleness. By summing these values, we can gain insights into the total amount of unused resources during the execution of all tasks.

As shown in Figure 3.7, the results indicate that the SJF scheduling algorithm results in significant resource waste. This is primarily due to its focus on prioritizing shorter tasks without considering their resource requirements. In addition, while MISO employs hardware-based partitioning, it suffers from inefficiencies arising from limited flexibility in partitioning, which leads to GPU fragmentation. MISO also emphasizes finding the optimal partition for each task but does not consider the constraints of combining different partitions, resulting in further GPU fragmentation.

In contrast, our proposed method utilizes software-based partitioning to co-locate a user’s trusted task within a single hardware partition, maximizing resource utilization. Furthermore, incorporating a solver service in the GPU partitioning sub-algorithm significantly reduces idle resources. Comparison results demonstrate that, although

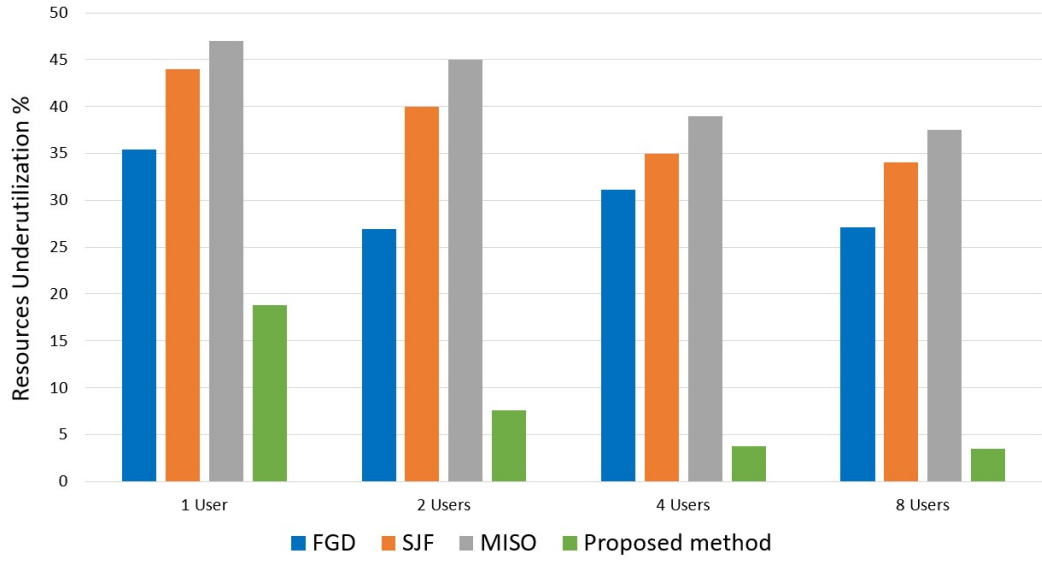


Figure 3.7: Resources Underutilization During Task Execution for different number of users in the cluster

FGD outperforms both SJF and MISO, our proposed method’s dynamic adjustment and rescheduling tasks lead to significant improvements in GPU resource usage. As illustrated in the figure, the comparison results show an enhancement in resource utilization compared to the SJF, FGD, and MISO methods, with a guaranteed minimum improvement of 88.10%.

The maximum normalized task completion time is shown in Figure 3.8, illustrating the effectiveness of the proposed method in reducing task execution time as user demand increases. Our approach results in a lower maximum normalized task completion time compared to both SJF and MISO. While SJF prioritizes shorter tasks to minimize waiting time, it can create resource bottlenecks in a shared GPU environment, ultimately increasing overall task completion time. This happens because executing shorter tasks with high resource demands can leave fewer resources available, blocking the execution of other tasks. In contrast, our proposed fair scheduling method avoids such blocking by considering both the resource requirements and the expected execution time of tasks in the Task Grouping sub-algorithm. In this sub-algorithm, the strategy of grouping tasks with similar execution times into a single partition is crucial for reducing overall task completion time. The Task Placement sub-algorithm, which sorts the grouped tasks along with the list of partition configurations, provides significant advantages in terms of task completion times. On the other hand, constrained by limited partitioning options, MISO fails to optimize task placement, resulting in prolonged execution times. While the FGD is designed for fair resource allocation and performs better than SJF and MISO, it still lacks the efficiency of our proposed approach. The reason is that the proposed approach utilizes dynamic rescheduling, task grouping, and optimized partition placement to improve scheduling

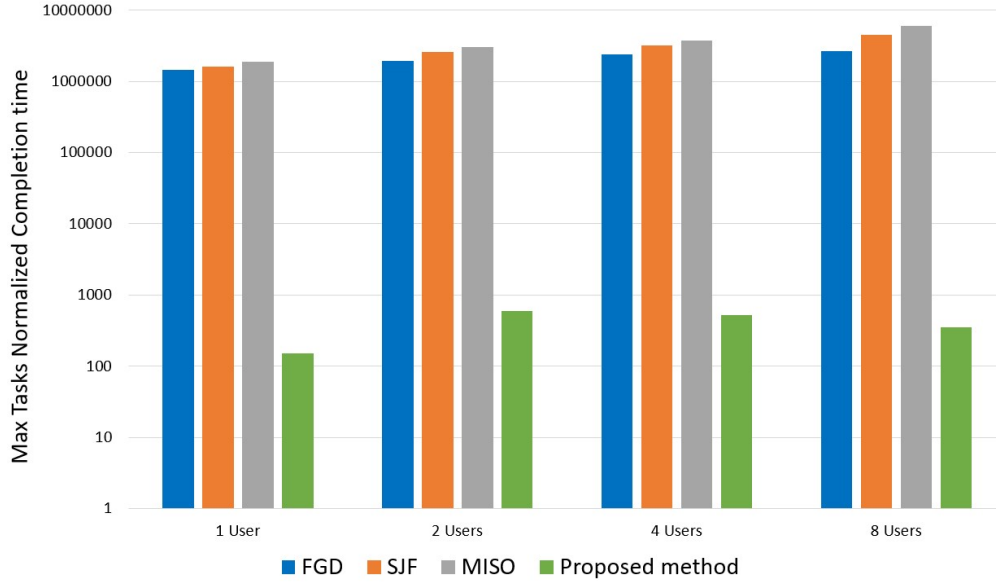


Figure 3.8: Max Tasks Normalization Time During Task Execution for different number of users in the cluster

efficiency and minimize delays. Moreover, the proposed routines in our approach enable the repartitioning and eviction of currently running tasks, leading to fair allocation and shorter task completion times.

3.6.2.2 GPU Scaling Scenario

To evaluate how well scheduling strategies adapt to GPU resource availability, we increased the number of available GPUs exponentially from 2 to 64. This approach enables us to assess how effectively each approach utilizes the additional GPU capacity to meet user demands. Figure 3.9 illustrates the underutilization of resources as GPU capacity increases. The results of the SJF scheduling method indicate inefficient GPU usage because it does not schedule tasks according to available resources, leading to significant idle GPU capacity. Similarly, the MISO method experiences GPU fragmentation due to its dependence on predefined hardware partitions, which restricts its ability to allocate resources efficiently. While the FGD approach performs better than both SJF and MISO, it still does not achieve full utilization. This is primarily due to its fair allocation strategy, which does not always prioritize optimal resource usage. In contrast, our proposed method dynamically adjusts task groupings based on the available GPU resources, significantly reducing underutilization. Our proposed approach still encounters some underutilization because the grouped tasks may not fully utilize a hardware partition. Consequently, some resources may remain idle, potentially leading to delays in task execution.

Finally, Figure 3.10 presents the max normalized task completion time as GPU resources

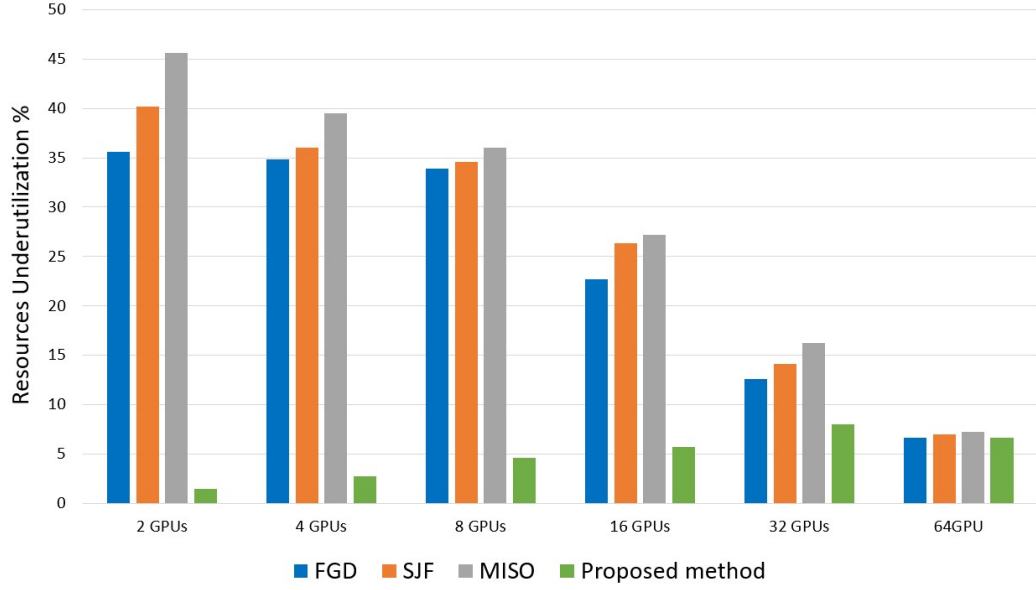


Figure 3.9: Resources Underutilization During Task Execution for different number of GPUs in the cluster

scale. While increasing GPU capacity generally improves task execution times, the effectiveness of this improvement depends on the scheduling approach. SJF and MISO fail to fully leverage additional GPU resources, leading to higher completion times. FGD performs better than these methods but remains suboptimal compared to the proposed approach, which employs a resource-aware task grouping strategy to efficiently distribute tasks across available GPUs. As a result, the proposed method achieves faster task execution. It is worth mentioning that as the number of GPUs increases, such as in the case of 64 GPUs, the performance of the proposed approach stabilizes and shows similar results because there are sufficient resources to handle the load. In this scenario, the scheduling mechanism has less impact, as the system has ample resources to meet the demand, and task completion times are already optimized by the availability of resources.

3.7 Conclusion

In this chapter, we proposed an efficient GPU resource allocation technique integrated with an optimized task scheduling solution to address the challenges of underutilized GPU resources and suboptimal task completion times in multi-tenant cloud environments. Our approach leverages a combination of software- based on partitioning and LP solver optimization to maximize GPU resource utilization while minimizing task completion time. The experimental results demonstrate that our proposed method guarantees at least an 88% improvement in resource utilization, achieving significant reductions in resource wastage

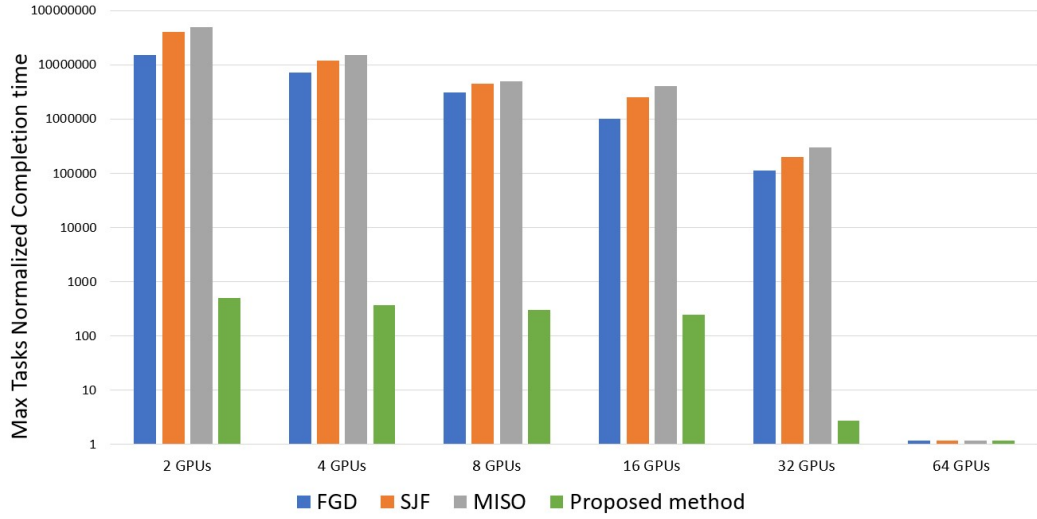


Figure 3.10: Max Tasks Normalization Time During Task Execution for different number of GPUs in the cluster

compared to the FGD, SJF and MISO techniques. Furthermore, our approach shows significant improvement in maximum normalized task completion time. We demonstrated that the proposed method outperforms existing techniques through two scaling scenarios: scaling users and scaling GPUs. In the user scaling scenario, it adapts effectively to increasing demand, while in the GPU scaling scenario, it maximizes resource utilization as the number of GPUs grows. Overall, our proposed method offers substantial improvements over existing techniques, ensuring both optimal resource efficiency and minimized task delays in cloud-based GPU scheduling environments.

Chapter 4

Efficient GPU Resource Allocation for Microservice Applications¹

4.1 Introduction

The emergence of microservice architectures has transformed cloud application design by decomposing monolithic systems into loosely coupled, independently deployable components [97][98]. This modular approach significantly enhances scalability and parallel execution. However, it also introduces new challenges in resource management, particularly for GPU-accelerated workloads where components may have different computational needs, interdependencies, and highly dynamic runtime behaviours. Modern microservice-based applications are highly compute-intensive, especially in domains like real-time video analytics, extended reality, autonomous driving, and scientific simulations [99][100]. These workloads require high throughput and low latency, leading to the use of GPUs to enhance component execution. GPUs enable parallel processing of microservice components through spatial multitasking, allowing multiple components to execute concurrently [26]. However, allocating GPU resources efficiently among components remains challenging due to different computational requirements and component-level dependencies. The first key challenge lies in dynamically and workload-aware allocating GPU resources to microservice components with interdependent execution flows. In synchronous pipelines, where each component waits for its predecessor's output, an imbalance in resource assignment can cause

¹This chapter is based on two published papers and two published Ericsson patent: [5] Hoda Sedighi, D. Gehberger, and R. Glitho, "Workload-aware dynamic GPU resource management in component-based applications," in *IEEE International Conference on Cloud Engineering (IC2E)*, IEEE Computer Society, 2022, pp. 213–220, [4] Hoda Sedighi, D. Gehberger, A. Ebrahimzadeh, F. Wuhib, and R. H. Glitho, "Efficient dynamic resource management for spatial multitasking gpus," in *IEEE Transactions on Cloud Computing*, vol. 13, no. 1, pp. 99–117, 2025, [6] Hoda Sedighi, D. Gehberger and R. H. Glitho, "Dynamic Allocation of Accelerator Processors Between Related Applications," WO2023199099A1, Patent published by *Telefonaktiebolaget LM Ericsson (Publ)*, 2023. Priority date: 2022-04-14, filed: 2022-04-14, published: 2023-10-19, and [7] Hoda Sedighi, R. Glitho, D. Gehberger, H. Gustafsson, and A. A.-E. Hassan, "Transfer of Memory Claim for Data Between Applications," WO2023199096A1, Patent published by *Telefonaktiebolaget LM Ericsson (Publ)*, 2023. Priority 2022-04-14 • Filed 2022-04-14 • Published 2023-10-19.

bottlenecks and reduce overall application throughput [101]. Existing approaches, such as NVIDIA’s MPS, allow spatial sharing of GPU resources but lack dynamic reallocation mechanisms and workload-aware allocating may lead to resource monopolization [27][22]. Other techniques based on kernel prioritization [37], execution time prediction [28], or kernel matching heuristics [54][51] focus on per-kernel optimization, often ignoring component-level interdependencies. Moreover, many rely on offline benchmarking [29], which fails to adapt to real-time changes in workload patterns, especially in cloud environments where task arrival and characteristics evolve dynamically [101]. Additionally, using preemption to allocate resources to specific components can help prevent bottlenecks, similar to preemption-based approaches for dynamic GPU allocation [22][50][21]. However, this method often suffers from context-switching overhead, which can degrade performance. Moreover, it may lead to some components starving of resources, resulting in a lack of fairness and reducing the overall responsiveness of the application [35].

The second critical challenge is the overhead caused by frequent data transfers between CPU and GPU and between interdependent microservice components. In typical GPU execution models, input data must be transferred from the CPU to the GPU, processed, and then returned to the CPU before being passed to the next component [42]. In microservice chains, this back-and-forth communication causes latency and saturates the PCIe bus, becoming a bottleneck for performance-sensitive applications. Although modern GPUs support shared memory models that can reduce such overhead, most existing GPU scheduling and allocation mechanisms do not leverage these capabilities effectively. To address these challenges, there is a clear need for a dynamic, workload-aware GPU resource management framework. Such a system should monitor runtime behaviours, adaptively reallocate GPU resources based on current demands, and respect component interdependencies to maximize throughput while avoiding bottlenecks. Simultaneously, efficient memory-sharing mechanisms are essential to minimize data movement latency between components. Addressing these challenges is crucial for fully leveraging GPU acceleration in microservice-based cloud applications, where response time is critical to maintaining service quality and meeting user expectations.

This chapter investigates the problem of efficient GPU resource allocation in cloud environments, focusing on microservice-based applications executed via spatial multitasking. We analyze how workload fluctuations, component dependencies, and data-transfer overhead affect application throughput, and we discuss the limitations of existing GPU resource management techniques that rely on static partitioning, offline profiling, or per-kernel QoS enforcement. By addressing these limitations, we aim to demonstrate the importance of adaptive scheduling and runtime-aware GPU partitioning strategies for improving throughput and ensuring stable performance in real-time cloud applications.

The rest of this chapter is organized as follows. First, we present an illustrative use case that motivates the need for dynamic GPU resource allocation in microservice-based

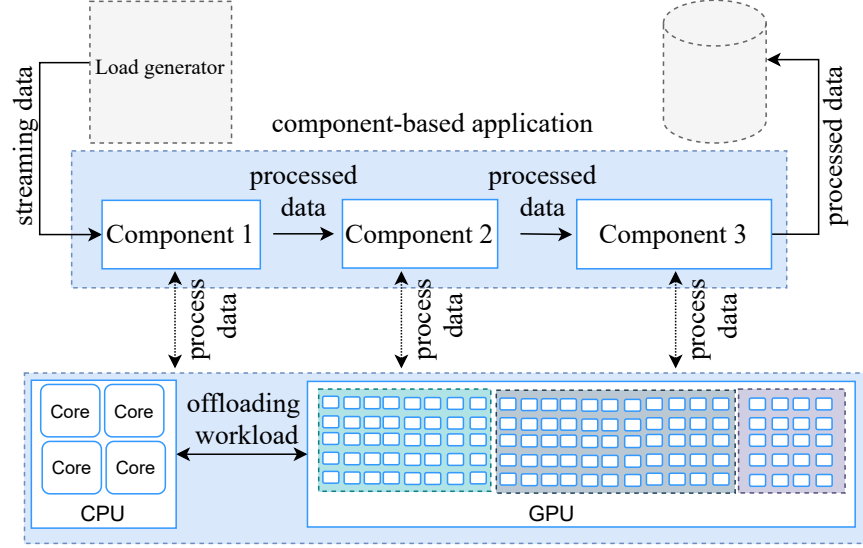


Figure 4.1: Illustrative use case of for GPU resource allocation in for Microservice Applications

cloud applications. Next, we describe the system model and the proposed GPU resource management framework. We then evaluate the proposed approach and compare it against state-of-the-art spatial multitasking techniques. Finally, we summarize the main findings and contributions of this chapter.

4.2 Illustrative Use Case and Motivating Scenario

Figure 4.1 illustrates GPU resource allocation for Microservice Application in a cloud environment, which serves as an example to motivate our research problem. This use case shows a cloud-hosted microservice-based application executes a series of processing stages as a pipeline in the user-space program layer. Each component in the microservice application performs a distinct function. In the microservice-based application with linear dependencies, each component receives data from the preceding component, processes it, and passes it to the next component. This occurs in component-based applications with not only linear service graphs but also service graphs with complex dependencies, such as DAGs. DAGs comprise several linear pipelines, each with a sequence of interconnected components based on the user request type. Many real-world applications, such as speech recognition and text summarization, require advanced features that rely on DAGs with complex interactions between components.

As a real-world example of a microservice with linear dependency, we can consider a component-based object detection application with three components: decoding, enhancement, and object identification. These three components collectively process streaming data, e.g., frames received from a live camera stream. Each component consists

of a sequence of operations, including host and device codes, which are also known as kernel functions [102]. The host codes run on the CPU sequentially and are responsible for managing and launching kernels. After launching the kernel, the device codes, which are primarily compute-intensive workloads, are offloaded to the GPU and executed on the GPU in parallel using thousands of GPU threads. Each component has different computational demands and memory requirements, and these characteristics can change dynamically depending on the input data (e.g., scene complexity) or operational context (e.g., night vs. day). The microservice components are arranged in a synchronous pipeline, where one stage’s output becomes the next’s input. Thus, poor GPU allocation to a single stage (e.g., the detection module) can stall the entire pipeline. For instance, if the object detection module experiences a delay due to insufficient GPU resources, it propagates backpressure upstream and causes idle time downstream, reducing the overall application throughput. Therefore, efficient GPU sharing and adaptive allocation mechanisms are vital to ensure consistent performance. Recent GPUs improve hardware schedulers by enabling spatial multitasking techniques to share the core clusters between different components. However, spatial multitasking by default allows components to run concurrently and access all GPU SMs without restriction [18]. This can pose several challenges when concurrent components have different workloads. If all resources are currently occupied, new components attending to run on the GPU must wait until some resources are released after the currently running components complete their execution. In addition, identical allocation without considering the required resources limits the compute resources and degrades their performance.

GPGPU frameworks like CUDA [103] enable parallel computing on GPUs, providing libraries and APIs to manage computation and data movement [104]. The process of offloading the workload to the GPU involves several steps: (i) allocating memory on the GPU device memory, (ii) transferring data from CPU to GPU, (iii) launching kernel, and (iv) copying results back and freeing memory [16]. Application input is received on the CPU and transferred to the GPU via the PCIe bus. After the execution on the GPU, the result is sent to the CPU memory and transmitted to the following processing step. Some components, such as object detection using deep neural networks, are highly GPU-intensive, whereas others, like data preprocessing, require moderate computing but high I/O throughput. Frequent data transfers between the CPU and GPU (and between components) introduce latency overheads, saturating PCIe bandwidth and further degrading application responsiveness. Although modern GPUs support shared memory and MPS, current scheduling techniques lack robust support for inter-process memory sharing, making them suboptimal for such dynamic, interdependent microservice workloads.

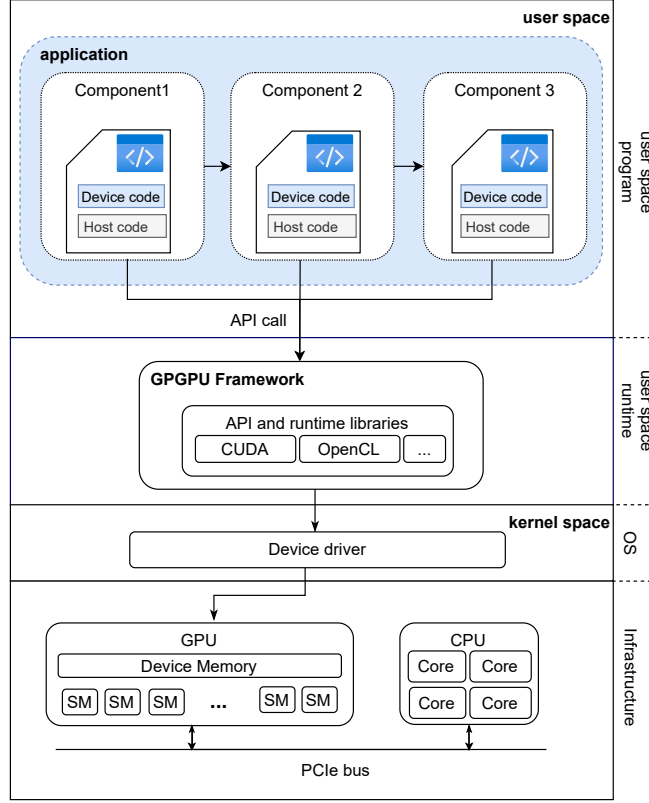


Figure 4.2: System Model

4.3 System Model

Figure 4.2 depicts the system model of an application with parallel programming codes employing a GPU as an accelerator. The model comprises four distinct layers [105]. At the top, the user-space program layer contains real-time microservice-based applications composed of multiple processing stages forming a pipeline. These applications are executed on the CPU and leverage the GPU to offload compute-intensive workloads. Each component consists of a sequence of host code and device code, commonly referred to as kernel functions [102]. Beneath this, the user-space runtime layer provides high-level runtime APIs and libraries used by applications. These APIs are built on top of the driver APIs, abstracting low-level functionality to facilitate kernel execution and data transfer on the GPU. Examples include CUDA and OpenCL, which offer extensive APIs for managing GPU operations [103]. In particular, CUDA divides its functionality into runtime and driver APIs, with the latter offering deeper control through mechanisms such as CUDA contexts and module management [16].

The third layer, the driver API layer, directly interfaces with the GPU hardware. The GPU accesses data stored in its device memory, which must first be transferred from the host memory via the PCI bus. After data transfer, GPU kernels are invoked and executed using thread groups managed by the GPU’s hardware scheduler. These thread groups

are dispatched to compute cores organized in clusters. For instance, in NVIDIA GPUs, thousands of CUDA cores are grouped into SMs, and the scheduler assigns thread blocks to available SMs for parallel execution [16]. The fourth layer is the GPU hardware layer. It handles the actual parallel computation through its compute cores. Spatial multitasking techniques allow different components to share core clusters and execute concurrently. However, this default concurrent execution without resource constraints can introduce performance challenges. When multiple components with varying resource requirements run simultaneously, contention can occur. If sufficient resources are not available, new components must wait until current tasks release resources. Moreover, uniform allocation without considering specific resource needs can limit computational efficiency and degrade performance.

4.4 Problem Formulation

In our system model, applications include a chain of interdependent components with linear or complex dependencies, such as DAGs. DAGs comprise several linear pipelines, each a sequence of interconnected components based on the user request type. Efficient and concurrent execution of components with different workloads on a GPU requires determining the core cluster allocation for each component. Either with a complex or linear dependency, each component c_i is a processing stage of a linear sequence of operations $C = \langle c_1, c_2, \dots, c_N \rangle$ to process the input data that are passed through sequentially. Our main goal is to manage the compute resources and allocate proper resources to the components based on their workload and resource requirements with the ability of dynamic resource reallocation. In modern GPUs, the fundamental unit of computation is the Streaming Multiprocessor (SM), which consists of multiple processing cores, shared memory, and control logic that execute threads in parallel. In this work, SMs are considered as the basic allocatable resource units (also referred to as core clusters). In the following, we develop our problem formulation.

Input Parameters:

- R_{SM} : The total number of SMs, i.e., core clusters, available in the GPU.
- N : The number of components in the application.
- CI_R^i : The number of received inputs in component i .
- CI_P^i : The number of processed inputs in component i .
- T_e : Execution time of the application.

- D_i : The number of dropped inputs in component i per application execution time T_e , which is given by $(CI_R^i - CI_P^i)/T_e$.
- P_i : The number of processed inputs in component i per application execution time T_e , which is given by CI_P^i/T_e .

Decision Variables:

- L_{SM}^i : The number of allocated core clusters to component i .

Objective Function:

We aim to maximize the overall throughput Thr of a given microservice-based application by improving the components' progress in terms of the component input processing rate P_i , $\forall i = 1, 2, 3, \dots, N$ while reducing the data transfer overhead between components. Basically, in applications with linear dependency, the slowest components in the application limit the overall throughput of the component-based application. Moreover, with the complex dependency, the latency of a user request is determined by the CPs, the paths of the maximal duration. However, since CPs change based on the allocated resources, any linear paths in the DAG can potentially become CPs if they encounter a shortage of resources. Therefore, to maximize the throughput of all linear paths in the DAG, we can consider the throughput of each linear path and ensure that all paths are optimized. In a given linear path P with N_p components, throughput can be determined by the minimum processing rate of components:

$$Thr_p = \min_{1 \leq i \leq N_p} f_i(L_{SM}^1, \dots, L_{SM}^i), \quad (4.1)$$

where $f_i(L_{SM}^1, \dots, L_{SM}^i)$ is the input process rate of component i , which depends not only on the allocation of component i , but also on the allocation of its preceding components $i - 1, i - 2, \dots, 1$. Therefore, the overall throughput can be determined as the sum of the throughputs of all linear paths ALP . Thus, our objective function is given as follows:

$$maximize : Thr = \sum_{p \in ALP} Thr_p, \quad (4.2)$$

Determining the relationship between the allocated resources L_{SM} and the corresponding processing rate P_i and precise mapping between them is complicated and requires isolation profiling [28][29], which is quite time-consuming. In this regard, we effectively narrow down the range of viable options by a categorizing technique and fine-tuning the allocated resources by monitoring and analyzing the throughput, processing rate and drop rate.

Constraints:

Constraint (4.3) ensures that the number L_{SM}^i of assigned core clusters to component i is smaller than the GPU capacity R_{SM} and the total number of allocated core clusters does not exceed the GPU capacity R_{SM} :

$$\sum_{i=1}^N L_{SM}^i \leq R_{SM}. \quad (4.3)$$

Given that the GPGPU frameworks provide high-level APIs by having access to GPU commands, our resource allocation optimization stands in the user-space run-time layer and improves the hardcoded scheduler by modifying the allocation of resources between concurrent kernels.

4.5 Proposed Solution

In microservice-based applications with synchronous data communication that employs GPU as an accelerator, dynamic GPU resource allocation and the overhead of regular data transfer between components and over the PCIe bus are critical challenges, which affect the overall application performance. To address these challenges, in this section, we propose a mechanism to share GPU resources between the components of an application and eliminate excessive transferring of data. Figure 4.3 illustrates the overview of our proposed solution, which comprises four main parts: (i) GPU memory manager, (ii) online performance optimizer, (iii) monitoring system, and (iv) GPU resource manager.

4.5.1 GPU Memory Manager

GPU memory manager is a dedicated central system that is responsible for reducing the latency of transferring the data. This system comprises two parts: (1) memory manager server and (2) client libraries. The memory manager server is designed to reduce data passing between components and eliminate frequent copying of the data between CPU and GPU by employing shared GPU memory. The client libraries consist of a set of APIs allowing applications and other system components to access the memory manager server.

The process of offloading workload to a GPU includes allocating GPU memory, transferring the data from the host to the device, copying results back to the host, and de-allocating GPU memory. While the mentioned steps are typical in offloading the workload, in the chain of components with synchronous data communication, the allocated memory can be reused to reduce the cost of transferring and allocating memory. The chain's process consists of multiple sequential steps iteratively applied to the input data. Therefore, a shared allocated memory in the first component can be reused in the chain by only modifying the

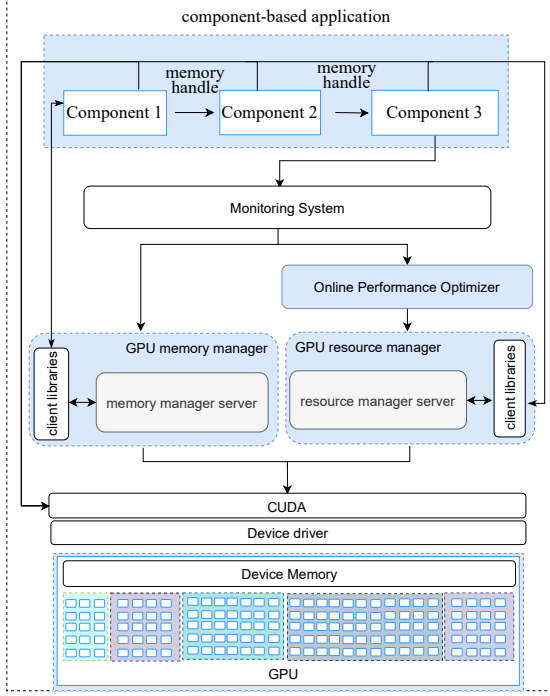


Figure 4.3: Overview of the proposed method.

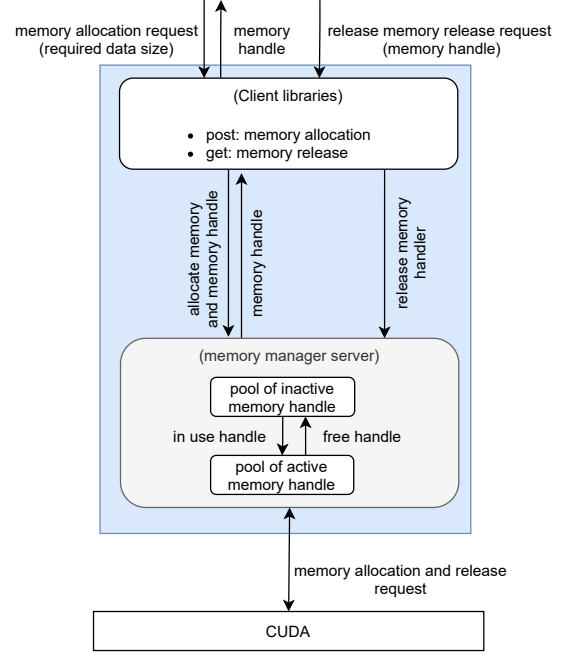


Figure 4.4: Overview of the GPU memory manager.

contents. This implies that the data transfer overhead, including several allocations and de-allocations of memory, transferring the data to GPU, and retrieving the results, can be eliminated using shared GPU memory.

In the GPU memory manager, we manage the shared memory between chain components by an interprocess memory handle, which is a pointer to the allocated memory. Initially, the first component uses the API to send an allocation request to the memory manager server containing the required data size. The first component may request more memory than it needs to reserve additional memory for the rest of the components. As shown in Figure 4.4, after receiving the request, the memory manager server allocates a specific size of memory and a GPU memory claim (handle) corresponding to it using CUDA APIs and exports the related handle to the first component. The component employs the handle information to copy data to allocated GPU memory and passes the handle to the next component after the GPU has processed the data.

The handle object enables other components to access the data that has already been transferred to the GPU memory. After receiving the handle, each component processes and modifies the data and passes the handle to the following components. The last component in the chain is responsible for bringing the processed data back to the CPU as the final output once all components have been executed. Finally, the last component transmits the handle to the monitoring system to collect information related to the components process. To ensure privacy and prevent unauthorized access, we have embedded robust access control

mechanisms within the memory handles. Each handle is assigned specific permissions using Access Control Lists (ACLs). The ACL includes the IDs of components authorized to access or modify the data. Only these authorized components can interact with the handle and access data. This ensures that even if a component receives a handle, it cannot read or modify the data without the appropriate permissions, significantly reducing the risk of unauthorized access or data leakage.

Instead of transferring the entire data, the GPU memory manager transfers only a handle between components. Given that the size of the handle is fixed (64 bytes in CUDA), the data transfer latency between components is fixed and independent of the data size. Moreover, the proposed GPU memory manager prevents frequent data transfer between GPU and CPU, thus decreasing the input processing time and improving the throughput. Importantly, by reducing the data transfer between GPU and CPU, the PCIe bus usage in the system is also decreased significantly. The only concern of this method is the overhead of communicating with the memory manager server. To reduce the overhead, we implement a pool of handles. As a result, instead of only one handle, a set of handles can be requested in the first component. The handles can be stored to be used without any communication overhead. Moreover, given that allocating memory from GPU is a time-consuming process, the memory manager server is designed with two memory pools, namely, inactive pool and active pool. The active pool stores the handles that are currently in use, whereas the inactive pool is responsible for storing the idle handles. After receiving the allocation request from the first component, the memory manager first examines the possibility of retrieving the free handle from the inactive pool. This approach enables faster response without the overhead of allocating new memory from the GPU. If the inactive pool does not have any handles, a new set of memory is allocated from the GPU. Upon completing component operations with the handle, the memory manager moves the handle from the active pool to the inactive pool. The handle will be released if the inactive pool capacity is full.

It is worth noting that having an independent GPU memory manager is essential for ensuring that the reserved shared memory is always usable by other components, even in the case of a failure in one component. Another critical factor for the memory manager to be independent is that if handles are lost due to a component failure, the manager may periodically iterate over the in-used handles and release those that have been allocated for a long time.

4.5.2 Monitoring System

As an essential part of the proposed method, the monitoring system module is designed to collect progressing information of the components in each linear path and monitor the overall application performance. In order to allocate GPU resources to components aiming to improve the application’s overall performance, the progress of each component needs to be

improved. However, in a microservice-based application with synchronous communication, components are interdependent. Thus, enhancing the progress of a component without considering this dependency may cause an unbalanced progress rate between components. The unbalance progress may result in internal queuing and input loss by dropping the inputs between components. With these considerations in mind, we consider two important factors for monitoring the progress of each component: (i) component input processing rate and (ii) component input drop rate. In order to calculate these rates, the monitoring system collects the components' progress information, including the number of processed inputs and the number of received inputs for each component.

To collect and retrieve progress information from components, we extend our proposed shared memory technique and design a monitoring system with the capability of extracting data from the shared memory. Accordingly, we employ the handles not only for data transferring between components but also for carrying progress information. After processing an input, each component updates the number of processed and received inputs, inserts them into the related handle, and sends them to the next component. As shown in Figure 4.3, the last component sends the handle to the monitoring system. As a result, the monitoring system gets access to all the progressing information of each component and collects them dynamically using the handle. The monitoring system calculates the input processing rate P_i and input drop rate D_i of component i running in the specific linear path. The monitoring system also keeps track of all the linear paths by maintaining a list of paths and assigning a unique identifier to each path. This is critical, especially in applications with complex dependencies, as decomposing all linear paths is computationally expensive. Following this approach, we can collect the data related to each linear path as the data flows through components.

We note that the monitoring system is also responsible for calculating the application's progress in the entire runtime, which is the load capacity that the application sustains. The number of processed inputs may also be calculated based on the number of the handles received from the last components in the monitoring system. After retrieving the progressing information, the monitoring system requests the memory manager server to release the allocated memory and its related handle from the GPU.

4.5.3 Online Performance Optimizer

While GPU enhances the performance of compute-intensive components, proper resource-sharing configurations are required to reach to the highest capacity. The online performance optimizer is designed for online resource allocation and dynamic resource reallocation. This module determines the appropriate resource-sharing configuration for the components and the given input stream. The optimization mechanism allocates the most promising resource combination for each linear path according to the component's progress, resource requirements, and contributions to maximize the overall performance. As shown in Figure

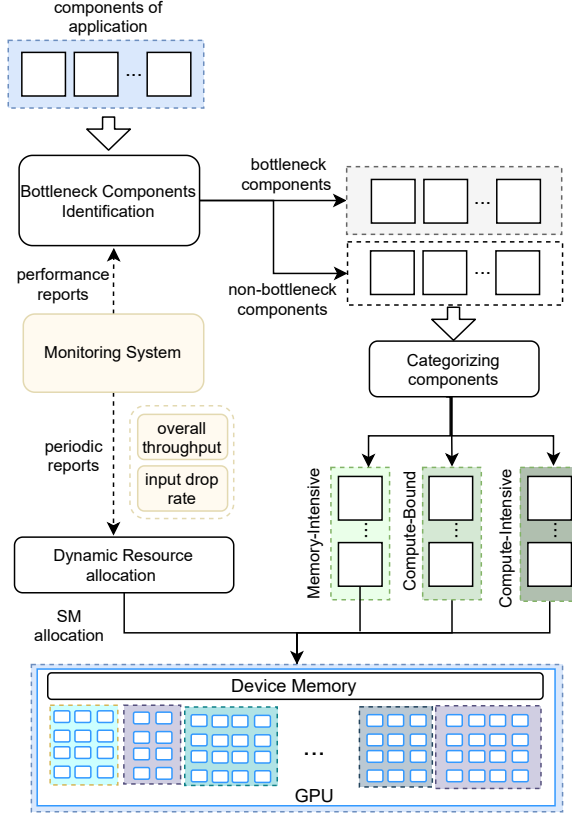


Figure 4.5: Overview of the Online Performance Optimizer.

4.3, the optimizer relies on the collected data by the monitoring system. Given that we aim to optimize the overall performance of the component-based application, the monitoring system periodically reports the collected information, including the application’s throughput and the processing and dropping rates of the components in each linear path, to the optimizer. The period of reports is set based on the complexity and processing rate of the components in the chain. The online performance optimizer makes decisions based on the monitoring reports to determine the optimal GPU resource allocation that yields the highest overall performance and the smallest input drop rate in the components. After making the decision, the results are utilized as the input for the GPU resource manager to allocate the resources to components dynamically.

In the components concurrently executing on the GPU, each component requires assigning a specific portion of SMs proportional to its workload. Allocating more SMs may lead to internal queueing and dropping inputs by processing more inputs than the following components can process. In addition, allocating unnecessary resources to one component may limit the available resources for the other components, which increases their latency. Thus, restricting resources for specific components, even by assigning fewer resources than the component requires for processing, makes more resources available to the resource-intensive components. As a result, the online performance optimizer employs the

input drop rate to prevent wasting resources and imbalance progress in the components to improve the overall performance. The key factor in this approach is balancing the progress between the components in each linear path by finding the proportional resource allocation to prevent imbalance progress.

Algorithm 4.1: BN Identification and SM Portion Assignment

```

1: Inputs: List  $L_c$  of components, number  $N_c$  of components, available compute resources  $R_{SM}$ , list  $P[N_c]$  of
   processing rate for each component, list  $L_{SMAT}[N_c]$  of SM activity value for each component, SM percentage
   unit value  $SM_{unit}$ , list  $PreL_{SM}[N_c]$  of predefined SM portions for each component
2:  $L_{BN} \leftarrow \{\}$  /*list of bottleneck components
3:  $L_{SM}[N_c] \leftarrow \{\}$  /*list of SM portions for each component
4:  $CAT[N_c] \leftarrow \{\}$  /*list of category value for each component
5:  $C_{BN} \leftarrow$  SM portion with maximum access
6:  $Thr \leftarrow$  overall throughput of the application from monitoring report
7: for ( $i = 1$  to  $N_c$ ) do
8:    $L_{SM}[i] \leftarrow R_{SM}/N$ 
9: end for
10: for ( $i = 1$  to  $N_c$ ) do
11:   if ( $(P[i] - Thr) < D_{thr}$ ) then
12:      $L_{BN} \leftarrow L_c[i]$  /*add component to  $L_{BN}$ 
13:   end if
14: end for
15: for ( $i = 1$  to  $N_c$ ) do
16:   if ( $i \notin L_{BN} \ \& \ L_{SMAT}[i] \leq 10\%$ ) then
17:      $CAT[i] \leftarrow low$ ;  $N_{low}++$ ;
18:   end if
19:   if ( $i \notin L_{BN} \ \& \ 10\% < L_{SMAT}[i] \leq 30\%$ ) then
20:      $CAT[i] \leftarrow mid$ ;  $N_{mid}++$ ;
21:   end if
22:   if ( $i \notin L_{BN} \ \& \ 30\% < L_{SMAT}[i]$ ) then
23:      $CAT[i] \leftarrow high$ ;  $N_{high}++$ ;
24:   end if
25: end for
26: for ( $i = 1$  to  $N_c$ ) do
27:   if ( $i \in L_{BN}$ ) then
28:      $L_{SM}[i] \leftarrow C_{BN}$ 
29:   end if
30: end for
31: for ( $i = 1$  to  $N_c$ ) do
32:   if ( $i \notin L_{BN}$ ) then
33:     if ( $CAT[i] = low$ ) then
34:        $L_{SM}[i] \leftarrow I_{low}$  /* calculate normalized initial value for low category
35:     end if
36:     if ( $CAT[i] = mid$ ) then
37:        $L_{SM}[i] \leftarrow I_{mid}$  /* calculate normalized initial value for mid category
38:     end if
39:     if ( $CAT[i] = high$ ) then
40:        $L_{SM}[i] \leftarrow I_{high}$  /* calculate normalized initial value for high category
41:     end if
42:   end if
43: end for
44: AllocationFineTuning ( $L_c[i]$ ) /* allocation fine-tuning of non-bottleneck component

```

Components in the microservice-based application may have different compute resource requirements. Default spatial multitasking allocates equal resources to each concurrent

component, resulting in failure to fulfill component requirements, especially in highly compute-intensive components. Our proposed approach is to find the highly compute-intensive components, known as the bottleneck components, in each linear path and allow them to access most of the GPU resources. Algorithm 4.1 depicts the proposed workload-aware dynamic resource allocation, which aims to maximize the overall performance. As shown in Figure 4.5, our workload-aware dynamic resource allocation comprises four main steps: (1) bottleneck components identification (line 10-14 of Algorithm 1), (2) categorizing components based on compute-intensive characteristics (line 15-25 of Algorithm 1), (3) SM allocation (line 26-43 of Algorithm 1), and (4) dynamic resource reallocation (line 44 of Algorithm 1).

4.5.3.1 Bottleneck Components Identification

Typically, a components execution on the GPU can be characterized by (i) the execution time of the kernel (i.e., how long the kernel occupies the resources and (ii) the fraction of the resources that are required during the execution. Given that the first parameter directly impacts the processing rate of the components, it represents the contribution of the components in the chain. Therefore we use the execution time to find the bottleneck components and rely on the fraction of the required resources to categorize the remains non-bottleneck components. For each linear path, the chain’s overall processing rate always depends on the components with the lowest processing rate. Decreasing the allocated resources of these components reduces the overall processing rate significantly. As a result, in each linear path, our proposed method decides about the bottleneck components according to the impact of each component on the overall performance.

According to [18], in NVIDIA MPS, one of the recommended methods for assigning an SM portion to each component is $(100\%)/(0.5 \times N)$, where N is the number of the components that are being executed concurrently on the GPU. However, our analysis revealed that this strategy could yield a low performance in the case of having components with different workloads. The reason is that this strategy assigns resources without considering the actual resource requirements, resulting in a lack of resources in the components with more resource requirements. On the other hand, it allocates more resources than required in the components with less resource requirement. Therefore, we assign equal SM portions R_e to the components at the launch time as follows, and then adjust it based on our algorithm:

$$R_e = R_{SM}/N, \quad (4.4)$$

where N and R_{SM} are the number of components in the linear path and the available compute resources, respectively. Based on the MPS technique, R_{SM} is estimated as 200% of all the available resources. This assumption allows the GPU scheduler to overlap components execution when idle resources are available [18].

Next, we employ the process rate P_i of component i and the overall throughput Thr_p in path p of the application to detect the components with the highest impact on the overall performance. The process rate of the bottleneck components is close to the overall throughput of the path. The online performance optimizer receives the performance reports and iteratively compares the application throughput with the process rate of each specific component. A component is labeled as a bottleneck component of the path p if the difference is smaller than a pre-defined threshold D_{thr} :

$$P_i - Thr_p < D_{thr} \quad (4.5)$$

The pre-defined threshold D_{thr} is set as a relative threshold, 10% of the overall throughput Thr_p . The threshold is set based on experiments and could be even smaller. However, due to having a fine tune algorithm, we allow the threshold to be relaxed.

4.5.3.2 Categorizing components based on compute-intensive characteristics

One of the essential parameters that characterize the components executing on the GPU is the fraction of the resources or the number of GPU threads that are required during the execution. In GPUs, this characteristic is typically defined based on two attributes: the number of threads per block and the number of blocks in a resource grid. In order to decide based on the GPU threads, methods with detailed profiling are required. In addition, the input size, which typically impacts the number of threads, may not be known in advance [38].

According to our analyses, the device-level profiling metric, SM activity that is supported in the GPUs, represents the average fraction of the resources that the kernel uses during the execution [106]. A higher value of this metric indicates a higher resource usage. To collect the SM activity information, we execute each component isolated to determine the exact resource requirements without the interference of the other components. At the launch time, the online performance optimizer categorizes all the components into three main categories based on the collected information from offline profiling. The thresholds for SM activity categories are determined based on the results of offline profiling using an iterative process. This process involves collecting baseline data and analyzing various applications with different workloads. Following, initial thresholds are defined for each category and then refined iteratively. This includes evaluating the distribution of applications across predefined categories and adjusting thresholds to achieve a more balanced distribution.

- **Compute-Intensive components:** The components with SM activity of more than 30% marks to be in this category. These components utilize almost all of the available GPU processing cores. Decreasing SM portions of the components in this category highly drop their processing rate. On the other hand, increasing their SM portions or reducing the interference components portions improves their performance.

- **Compute-Bound components:** The components with an SM activity of 10%–30% fall in this category. These components have a moderate SM utilization and utilize a significant portion of the processing cores. In these components changing SM portions impacts their processing rate.
- **Low-Compute-Intensive components:** The components with SM activity less than 10% belong to this category. The component with SM activity percentages below 10% may not fully utilize the GPU’s available processing cores. These components mainly rely on memory transfer, and changing SM portions does not impact their processing rate. This characteristic makes them the appropriate options for releasing the GPU resources.

4.5.3.3 SM Allocation

After specifying the bottlenecks and categorizing the remaining components, SM fractions are determined for all the components of each linear path. Compared to the other components, the bottleneck components require more resources. Therefore, the method grants bottleneck components access to all the SMs. This is essential because components may not execute concurrently on the GPU in practice, and limiting their SM usage affects the application performance directly. In the next step, for the remaining components, the algorithm assigns the compute resources according to the category of each component. More specifically, for each non-bottleneck component, we define a set of various fractions of SM resources ranging from 5% to 85% to have a variety of fraction selection possibilities. The difference between each fraction is SM_{unit} .

Let N_{SM} be the number of pre-defined SM portions for each component. We note that the process of trying all the SM combinations is quite time-consuming, as having N_{NB} non-bottleneck components and N_{SM} SM portions leads to a total of $N_{SM}^{N_{NB}}$ combinations. Therefore, we first set an initial value for the non-bottleneck component according to the components category. Let C_{low} , C_{mid} , and C_{high} represent the initial percentages of SM resources. These initial percentages should be determined based on each category’s SM activity percentages. Therefore, we set these initial values to SM_{unit} , $2 \times SM_{unit}$, and $3 \times SM_{unit}$, respectively. The method assigns C_{high} portion to Compute-Intensive category and C_{mid} and C_{low} portion to Compute-Bound and Low-Compute-Intensive categories, respectively. As a result, limiting the allocated resources of the Compute-Bound and Low-Compute-Intensive categories increases the availability of the resources for the bottleneck components. Assigning the initial values to each component may result in the total allocation being over 100%. Therefore, to ensure that the sum of allocations equals 100%, we need to calculate the normalized initial value for each category, $I_{category}$, based on the number of components in each category and the respective

percentages for each category:

$$I_{category} = \frac{C_{category}}{(N_{low}C_{low} + N_{mid}C_{mid} + N_{high}C_{high})} \times 100 \quad (4.6)$$

where the category can be low, mid, high and N_{low}, N_{mid} , and N_{high} is the number of components in the low, mid and high categories, respectively. We calculate this value and, based on that, map and select the nearest lower pre-defined SM portions for each component. At this point, non-bottleneck components are allocated a fair percentage of the SM. However, the overall performance of the chain can be improved. Algorithm 4.2 depicts the allocation fine-tuning of the non-bottleneck components to enhance the overall performance of the path. The algorithm periodically monitors the non-bottleneck components. In this method, we consider the input drop rate D_i and the overall throughput of the path p , Thr_p , to make a precise decision on assigning SM percentages. After initializing the SM resources of each component, the algorithm monitors the Thr_p and input drop rate D_i of the following component using the periodic monitoring system report.

The algorithm first monitors the input drop rate after assigning the SM portion. If the input drop rate of the next component in the path increases, the resource allocated to the component will be reduced by SM_{unit} . This approach is employed because improving the performance of the component without considering the progress of the next components is a waste of resources, as the output of the component may exceed the computing capacity of the next component. On the other hand, if the input drop rate is low or zero, the resource allocated to the component will be increased by SM_{unit} . Another parameter, Thr_p , reflects the impact of the new SM portion on the application's overall throughput on path p . Adjusting SM will terminate for each component if the chain throughput drops after changing the SM. This will happen if insufficient resources are assigned to the component. Finally, the algorithm normalizes the allocations to ensure the total allocation does not exceed 100%. The algorithm terminates once the appropriate SM portion is selected for all non-bottleneck components in the path. In worst case, the algorithm's complexity for each path is $O(N_{NB} * N_{SM})$ which is much smaller than the original $O(N_{SM}^{N_{NB}})$. It is worth mentioning that to ensure consistent optimized performance in certain conditions the described method might be re-executed periodically to ensure consistent optimized performance.

4.5.3.4 Dynamic Resource Reallocation

As the final step, we monitor concurrent paths in the application to maintain high performance and adjust the SM assignment according to the active paths. When multiple linear paths are executed concurrently, we first divide the total resources between the concurrent paths based on the request load distribution. Then, considering that we have determined the proportional resource allocation between the components in each linear

Algorithm 4.2: Allocation Fine-tuning for Non-Bottleneck Components

```
1: Inputs: Non-bottleneck components  $L_c$  for fine-tuning, List  $D[N_c]$  of input drop rate for each component, SM  
   percentage unit value  $SM_{unit}$   
2:  $Thr_p \leftarrow$  overall throughput of the application on path p from monitoring report  
3:  $Thr_{prv} \leftarrow$  overall throughput of the application on path p before changing SM  
4: for ( $i = 1$  to  $L_c$ ) do  
5:   while ( $Thr_p \geq Thr_{prv}$ ) do  
6:     if ( $(D[i + 1] > 0)$ ) then  
7:       decrease  $L_{SM}[i]$  by  $SM_{unit}$   
8:     else  
9:       increase  $L_{SM}[i]$  by  $SM_{unit}$   
10:    end if  
11:  end while  
12: end for
```

path, we assign the resources to each path component. The online performance optimizer is expected to improve throughput by efficiently allocating resources based on component requirements and contributions. Moreover, it decreases drop rates by considering component progress and its subsequent component.

4.5.4 GPU Resource Manager

To allocate a specific portion of the GPU resources to the components, the GPU resource manager is designed, which receives SM allocation configuration from the online performance optimizer and specifies the components' resource limitations. In order to enforce the limitations, we create various GPU contexts with a specific amount of SMs. Since the process of creating a context is time-consuming, a pool of contexts with different SMs configurations is predefined for each component. During the execution, based on the online performance optimizer results, the GPU resource manager may change the component's context to increase or limit their computational resources. Using this technique, we achieve the goal of dynamically reallocating the SMs for the components. We note that if NVIDIA's MPS tool allows dynamic SM allocations in the future, our proposed solution becomes more efficiently implementable without contexts.

4.6 Performance Evaluation

In this section, we conduct an experimental evaluation of our proposed methods. We first present our prototype, including benchmark setting and implementation details. Next, we explain the experimental settings followed by the evaluation results.

4.6.1 Prototype

We implemented our proposed method along with the related libraries to evaluate its performance. Additionally, we developed microservice-based applications consisting of a chain of components with different workloads that execute concurrently on the GPU.

To comprehensively assess our proposed method, we used a combination of various benchmarks. We utilized the comprehensive Rodinia benchmark suite [107], containing various computational patterns and kernel types. This benchmark offers more variety compared to other benchmark suites such as SHOC[108] and Parboil[109], as demonstrated in benchmark analyses [110]. In addition, we incorporated benchmarks designed explicitly for DNNs and natural language processing (NLP) to better reflect the modern GPU-based applications. These benchmarks include Detectron2 [111] image-to-image processing for object detection with several deep-learning models, NLP Bert [112] for text-to-text processing, Hifi-gan [113], Tacotron2 [114], and WaveGlow [115] for text-to-speech processing.

4.6.2 Benchmark

Applications in the original version of the mentioned benchmarks are independent. To create a chain of interdependent components, we modified the benchmark applications to make them interdependent. The application modification consists of adding a wrapper library to receive data packets, triggering the kernel by receiving data and sending processed data to the next component. We implement an asynchronous workload generator to simulate the frequency of receiving data and generate loads for the application’s components in the chain. The second part of the modification enables applications to support communication with the resource and memory managers to assign proper SM percentages and access data using the handle. The related APIs were added to the initial part of the applications. For fairness, the kernel codes were not modified. It is worth noting that the communication part adds some overhead to the execution time of the application. Although the provided method has overhead, this overhead is negligible compared to the throughput improvement.

4.6.3 Experiment Settings

The experiments were conducted on an NVIDIA A100 GPU with Ampere architecture and 108 SMs. In order to evaluate the performance of our proposed method under various workloads with distinct kernel characteristics, we require components with different kernel complexities. We therefore implemented various chains of components using the combination of different benchmark applications. Table 4.1 summarizes the details of the setups and benchmark applications. In addition, we utilize applications with different input data sizes to change the kernel workload and compute resource requirements. The detail of the benchmark applications configuration and their workload characteristics are presented in Table II [4]. As shown in this table, increasing the data input size of benchmark applications impacts the frequency of applications’ kernels, leading to higher computational intensity. This results in a higher GPU core utilization and a larger number of CPU-to-GPU data transfers. Additionally, in some benchmark applications, such as Gaussian, the number of threads per block depends on the size of the problem. In this case, increasing the input size

Table 4.1: Hardware/software setup and benchmark specifications.

Specification	
Hardware	GPU: NVIDIA A100 CPU: AMD EPYC 7302 16-Core Processor
Software	Driver Version: 470.57.02 CUDA version: 11.4.10 OS: Ubuntu 20.04.4 with 5.4.0-137-generic
Benchmark Suite	
Rodinia Benchmark	LU Decomposition (LUD), Breadth-First Search (BFS), Hotspot (HS), Pyramid Pathfinder (Pathfinder), SRAD, Gaussian, CFD Solver (CFD), Needleman-Wunsch (NW), Stream Cluster (SC), Nearest Neighbor (NN)
DNNs & NLP Benchmarks	Detectron2 , Bert, Hifi-gan, Tacotron2 and WaveGlow

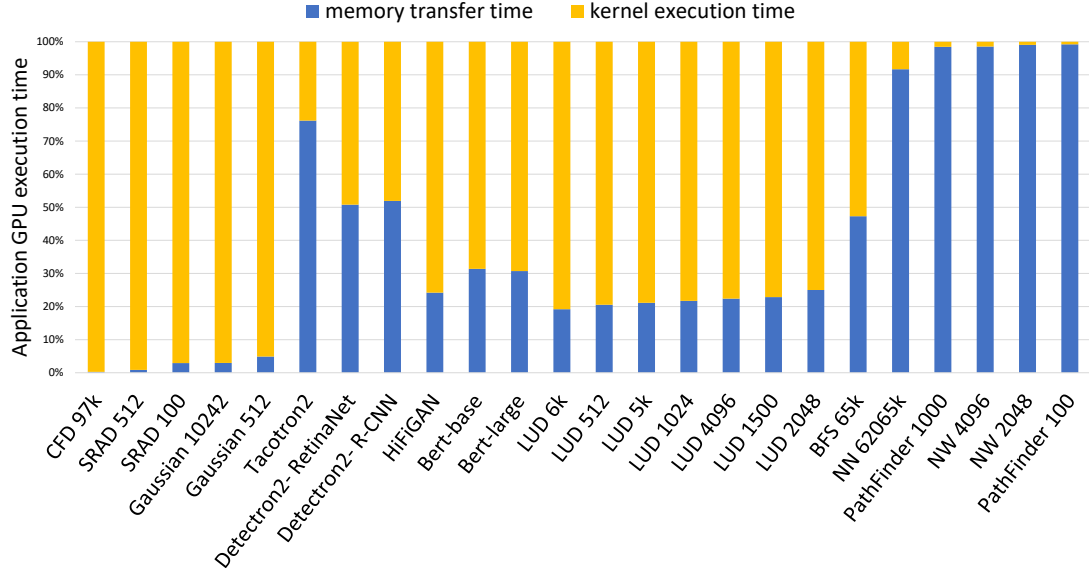


Figure 4.6: Characteristics of benchmark applications in terms of memory data transfer and kernel execution time

increases the thread block size to improve the amount of parallelism in the application, which as a result, increases occupancy and utilization of the GPU resources. Finally, changing the type and size of the model in DNNs and NLP benchmarks, like Bert and Detectron2, increases the repetition of applications’ kernels and GPU resource utilization.

To assess our method, we evaluate several chains of various components using benchmark applications with different types of resource requirements. This demonstrates how two distinct parts of our proposed method can reduce the overhead of data transfer in memory-intensive components and enhance the performance of compute-intensive components. Figure 4.6 illustrates the percentages of time allocated to memory data transfer and kernel execution for each benchmark application. As shown in the figure, some applications require more GPU processing cores to perform more complex computations. Therefore, we categorize benchmark applications based on the percentage of time they spent on

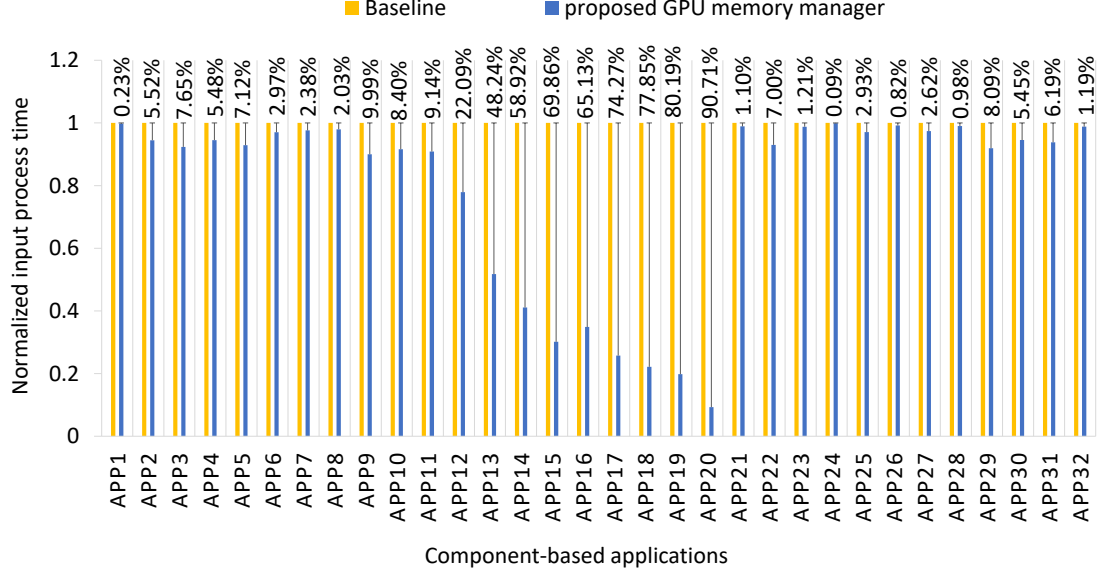


Figure 4.7: Normalized input process time comparison of various applications under consideration.

memory data transfer and kernel execution to ensure that we comprehensively assess our proposed method. Our categorization comprises (i) low compute-intensive components (L), which instead are high memory-intensive, (ii) medium compute-intensive (M), and (iii) high compute-intensive components (H) (see Table 4.2).

Finally, the details of parameter values to assess the performance of our proposed method are as follows. The parameter SM_{unit} determines the accuracy of the fraction selection for each component while, on the other hand, impacts the algorithm's complexity. We conducted a series of experiments and varied the SM_{unit} value from 1% to 20% to observe its impact on accuracy and computational complexity. The results indicated that 10% provided an optimal balance, ensuring precise fraction selection without adding unnecessary complexity. Therefore, the SM_{unit} is assigned 10%. The initial SM percentages C_{low} , C_{mid} , and C_{high} were determined based on SM activity percentages in each category to reflect the SM requirement of each category. The categories range from less than 10%, between 10% and 30%, and more than 30%. Therefore, for our experiments, C_{low} , C_{mid} , and C_{high} are set to 10%, 20%, and 30%, respectively. For the period of monitoring reports and resource reallocation, we utilized a packet-based approach and determined the period based on the completion of a predefined number of packets processed by the chain. After testing the method with various workloads, we found that setting this predefined number to 10 packets provided the optimal settings for this period.

Table 4.2: Component categories based on kernel execution time percentage.

Components	Kernel execution time percentage	Metric
NN 62065k, PathFinder 1000, NW 4096, NW 2048, PathFinder 100	Less than 20%	L
LUD 6k, LUD 512, LUD 5k, LUD 1024, LUD 4096, LUD 1500, LUD 2048, BFS 65k, BERT-b, BERT-l, HiFiGAN, Tac-Wav, Dt-RNet, Dt-CNN	Between 20% and 80%	M
CFD 97k, SRAD 512, SRAD 100, Gaus 10242, Gaus 512	More than 80%	H

Table 4.3: Specifications of the microservice-based applications.

components of application	name	characteristic	category
Gaus(1024),CFD(97k)	App1	H,H	HPP
LUD(2048),LUD(2048),LUD(2048),LUD(2048),BFS(65k),SRAD(512)	App2	M,M,M,M,M,H	HPP
LUD(2048),LUD(2048),LUD(2048),LUD(2048),BFS(65k),SRAD(100)	App3	M,M,M,M,M,H	HPP
LUD(2048),LUD(2048),LUD(2048),LUD(2048),SRAD(512),SRAD(512)	App4	M,M,M,M,M,H	HPP
LUD(2048),LUD(2048),LUD(2048),LUD(2048),SRAD(100),SRAD(100)	App5	M,M,M,M,M,H	HPP
LUD(1500),Gaus(512),BFS(65k),SRAD(512)	App6	M,H,M,H	HPP
LUD(1024),Gaus(512),SRAD(512)	App7	M,H,H	HPP
LUD(4096),LUD(4096),CFD(97k)	App8	M,M,H	HPP
LUD(2048),Gaus(512)	App9	M,H	HPP
LUD(1500),Gaus(512)	App10	M,H	HPP
LUD(1500),LUD(1500),Gaus(512)	App11	M,M,H	HPP
pathfinder(1000),CFD(97k)	App12	L,H	HPP
pathfinder(100),Gaus(512),SRAD(512)	App13	L,H,H	HPP
pathfinder(100),SRAD(512)	App14	L,H	HPP
pathfinder(1000),Gaus(1024)	App15	L,H	HPP
pathfinder1000,Gaus(512),SRAD(512)	App16	L,H,H	HPP
pathfinder(1000),SRAD(512)	App17	L,H	HPP
LUD(1024),pathfinder1000,SRAD(512)	App18	M,L,H	HPP
pathfinder(1000),Gaus(512),NW(2048)	App19	L,H,L	HPP
pathfinder(1000),NW(2048)	App20	L,L	HPP
CFD(97k),Tac-Wav,SRAD(512)	App21	H,M,H	DPP
SRAD(512),Dt-CNN,pathfinder(100)	App22	H,M,L	IAS
NW(2048),HiFiGAN,CFD(97k)	App23	L,M,H	DPP
CFD(97k),HiFiGAN,SRAD(512)	App24	H,M,H	DPP
NW(4096),BERT-b,CFD(97k))	App25	L,M,H	DPP
CFD(97k),BERT-b,Gaus(512)	App26	H,M,H	DPP
NW(4096),BERT-l,SRAD(512)	App27	L,M,H	DPP
CFD(97k),BERT-l,Gaus(512)	App28	H,M,H	DPP
CFD(97k),Dt-RNet,NW(2048)	App29	H,M,L	IAS
CFD(97k),Dt-RNet,SRAD(512)	App30	H,M,H	IAS
CFD(97k),Dt-CNN,pathfinder(100)	App31	H,M,L	IAS
Gaus(512),Tac-Wav,SRAD(512)	App32	H,M,H	DPP

4.6.4 Evaluation Results

In this section, the experimental results of our proposed method are presented in two parts. In the first part, we evaluate the GPU memory manager, which aims to reduce data transfer overhead. The second part of this section presents the interpret results related to the online performance optimizer assessment in terms of throughput and input drop rate.

The impact of using a shared GPU memory is examined by comparing the throughput of the chain using the GPU memory manager with the chain using a baseline method, which uses regular UDP messages to transfer the data [116]. Table 4.3 details the microservice-based applications used in our experiment. We define three main categories for microservice applications to simulate realistic workflow scenarios.

- **Data Processing Pipelines(DPP):** Scenarios involve a sequence of tasks where data is pre-processed, analyzed, and post-processed. The Rodinia benchmarks handle pre-processing, the NLP benchmarks handle processing like text extraction, and the Rodinia benchmarks handle post-processed computationally intensive tasks.

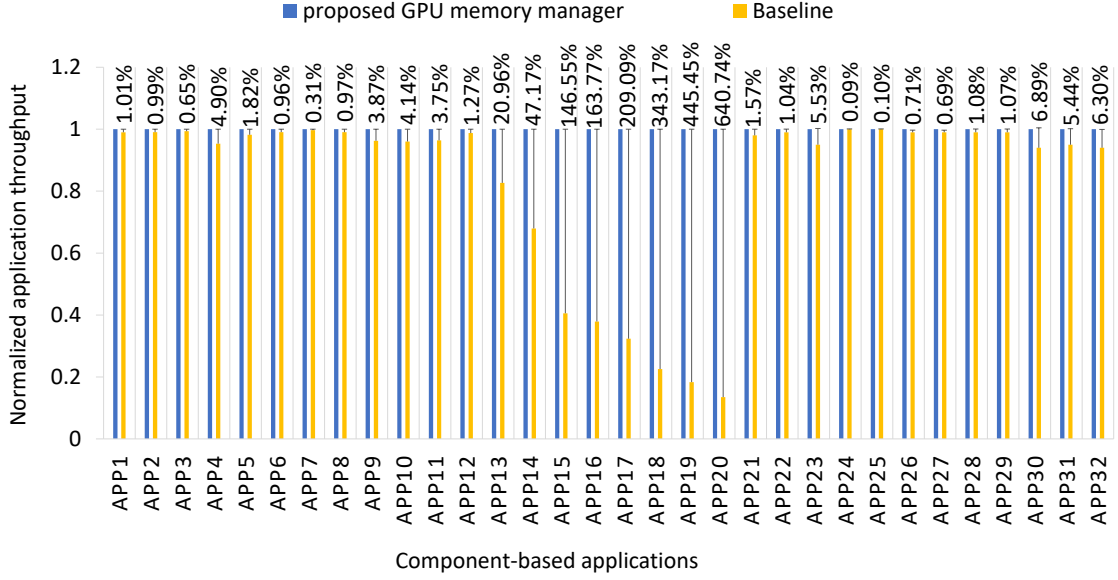


Figure 4.8: Normalized throughput comparison of various applications under consideration.

- **Image Analysis Systems(IAS):** Scenarios focused on visual data analysis. The Rodina benchmarks manage pre-processing, DNN benchmarks are used for tasks like object detection, and then Rodinia benchmarks are employed for high-performance computing tasks.
- **High-Performance Applications(HPP):** Scenarios focus on computational tasks often used in simulations and high-performance applications. They are designed using several Rodinia benchmarks.

The applications listed in Table 4.3 were selected to represent a wide range of real-world scenarios. Applications were selected randomly from Table 4.2 with carefully chosen categories to simulate various workloads across applications. We select as many sets as necessary to reach the point that the proposed method maintains the same level of throughput efficiency. When a component’s output is irrelevant to the needs of the next component, we bypass the actual output and send mock data as an alternative to mimic the expected input format. This ensures that the component can function properly across different scenarios without altering the benchmark code. As shown in Figure 4.6, the compute-intensive applications such as Gaussian rely heavily on GPU processing cores and require the GPU to perform many complex computations. On the other hand, memory-intensive applications such as Pathfinder and NW require the GPU to transfer the data frequently. In this type of applications, minimizing the data transfer improves the performance via efficient memory access. Figure 4.7 presents the input processing time of various microservice-based applications normalized to the baseline. The results indicate that

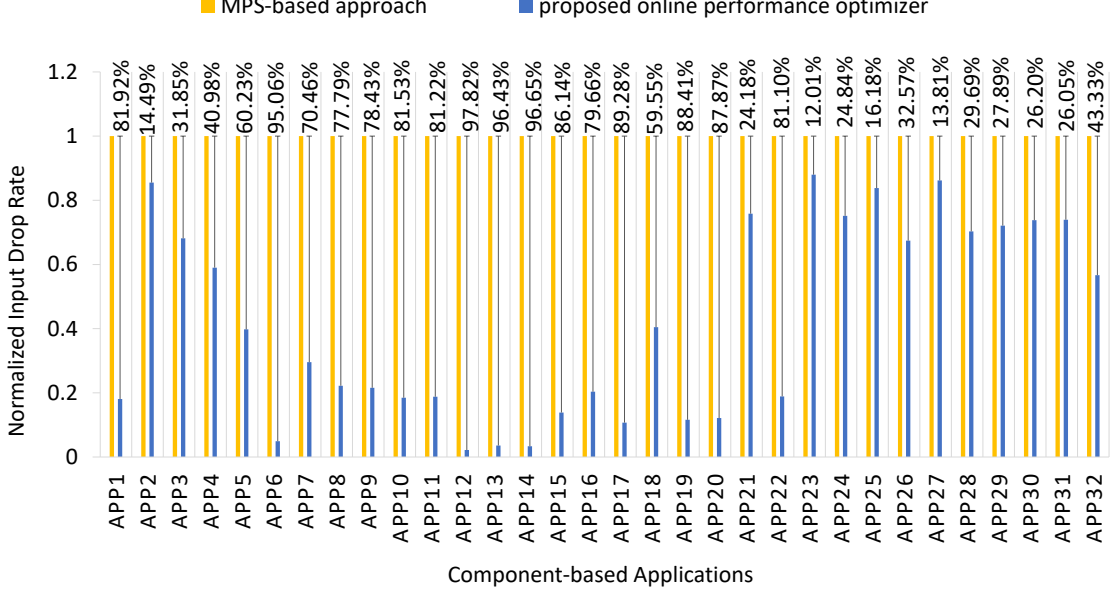


Figure 4.9: Normalized input drop rate comparison of various applications under consideration.

the proposed GPU memory manager achieves up to 90.71% decrease in the processing time of applications which contain components that are classified as highly memory-intensive(L). The reason is that the highly memory-intensive components spend most of their time waiting for data to be transferred between different memories. Decreasing the overhead of data transfer results in shorter processing times. The reduction in processing time is affected by the total time the components in the chain spend on transferring data.

Next, we evaluate the performance of the GPU memory manager on the overall application throughput. Figure 4.8 depicts the normalized throughput of applications over the baseline. As the results show, the throughput comparison indicates that the GPU memory manager results in up to a 6.4 times higher throughput in the test cases. Considering that the compute-intensive components spend most of their time performing computation on data, the results show less improvement in the chain without high memory-intensive components. The results demonstrate that in some test cases like app12, the improvement is less than the input process improvement. The reason is that although the proposed method is expected to improve the throughput by having a low compute-intensive component in the chain, the improvement depends on the difference between the input processing rate of the components. For instance, in app12, using a shared GPU memory improves the process rate of the Pathfinder. However, the next component CFD is highly compute-intensive and cannot process the packets as fast as the previous component. Therefore, in such cases, increasing the process rate of the component increases the drop rate between components. Next, we provide more results on how our proposed method

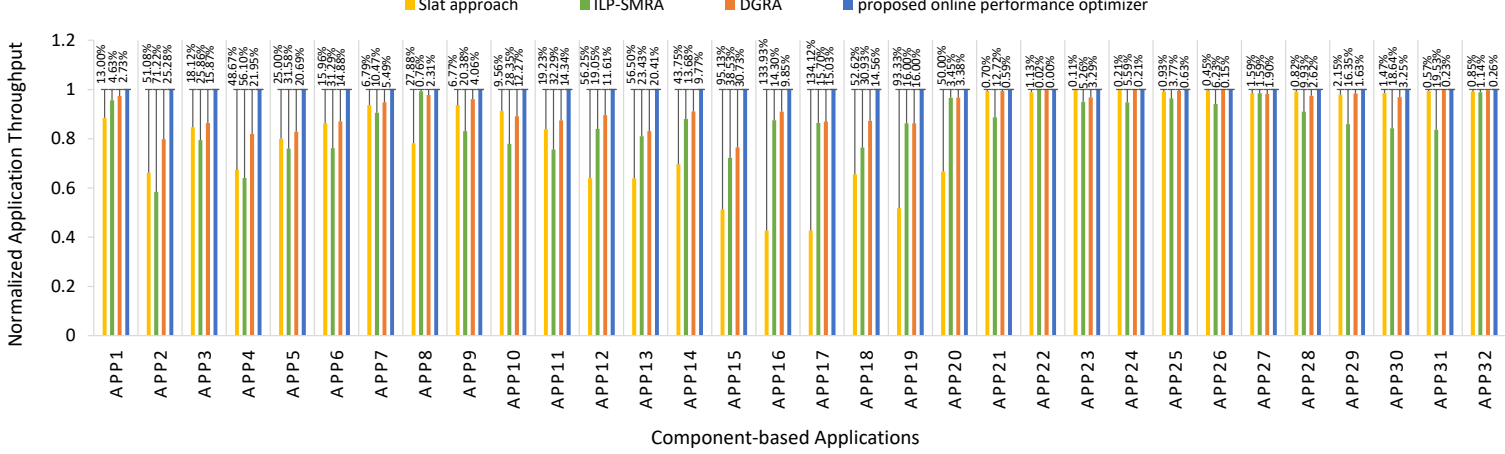


Figure 4.10: Normalized application throughput comparison of various applications under consideration.

Table 4.4: Components categories based on SM activity percentage.

Components	SM activity	description
LUD(512), BFS(65K), BFS(1M), SRAD(512), SRAD100, Gaus(256), NW(2048), SC, NN(62065k), BERT-b, BERT-l, HiFiGAN, Dt-RNet, Dt-CNN	Less than 10%	Low-Compute-Intensive
LUD (1024), LUD (1500), LUD (2048), Gaus (512), Tac-Wav	Between 10% and 30%	Compute-Bound
LUD (4096), LUD(5k), LUD(6k), Pathfinder (1000), Pathfinder (100), Gaus (1024), CFD (97k)	More than 30%	Compute-Intensive

reduces the drop rate between components.

In the following, we evaluate the performance of the online performance optimizer. The online performance optimizer considers the input drop rate metric to decide about allocating compute resources between components and attempts to reduce the drop rate to improve the application’s throughput. Figure 4.9 presents the normalized results of application input drop rate comparisons over the MPS-based approach as the default spatial GPU resource sharing [18]. In the MPS-based method, all the components get access to the entire computing resources of the GPU. The evaluation results on test case applications show a decreasing drop rate of 57.93% on average. The dropped rate decreasing depends on the different processing rates between components in the application. For instance, app12 consists of two components: a pathfinder(1000), which is a memory-intensive component, and CFD(97k), which is a compute-intensive component. These components have significantly different processing rates. As shown in the figure, the drop rate for this app is the highest at 97.82%. In contrast, in App 2, which consists of components with similar input processing rates, the drop rate is 14.49%, one of the smallest reductions in the figure.

To evaluate our online performance optimizer’s ability in terms of throughput, we compare the throughput result of our system with the ILP-SMRA method [54], the Slate

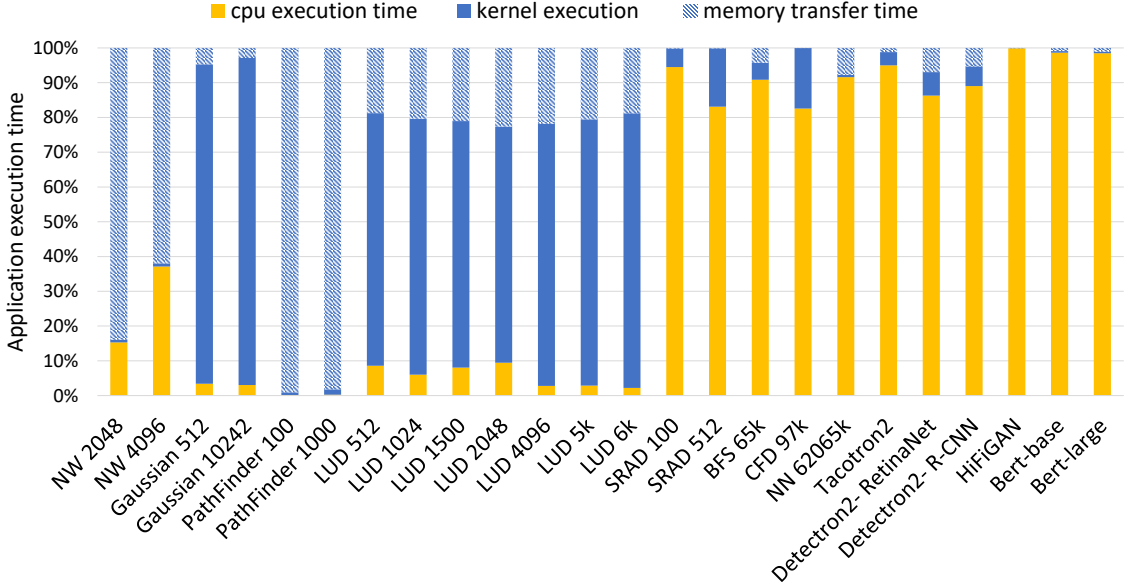


Figure 4.11: Benchmark applications characteristics in term of CPU and GPU execution time.

approach [51], and dynamic GPU resource allocation(DGRA) approach[5]. The ILP-SMRA method first assigns equal compute resources to the co-located applications. Then, it evaluates the Instructions Per Cycle (IPC) and bandwidth usage to reallocate GPU resources between applications. On the other hand, the Slate approach identifies and runs complementary kernels in parallel to improve system throughput. A set of kernels are selected as complementary if their concurrent execution enhances system throughput. Finally, DGRA allocates computing resources equally among the microservice application components and then adjusts the resource allocation based on performance improvements through iterative processes. To provide a controlled experiment with accurate and reliable results, we use the GPU memory manager in all four methods for memory-sharing purposes and ensure that the performance evaluation comparisons are solely related to the online performance optimizer. Our proposed method employs the SM activity characteristic to categorize components in terms of compute resource usage and allocate resources based on the components' resource requirements. The results of categorizing components by our method are presented in Table 4.4. Figure 4.10 presents the normalized application throughput comparison under different workload scenarios using 32 sets of microservice-based applications shown in Table 4.3. As observed in the figure, the comparison results show up to 134.12%, 71.22%, and 30.73% throughput improvements compared to the Slate approach, ILP-SMRA method, and DGRA, respectively.

The comparison results over the ILP-SMRA indicate that the advantages of our method in improving throughput are evident when considering the components' resource

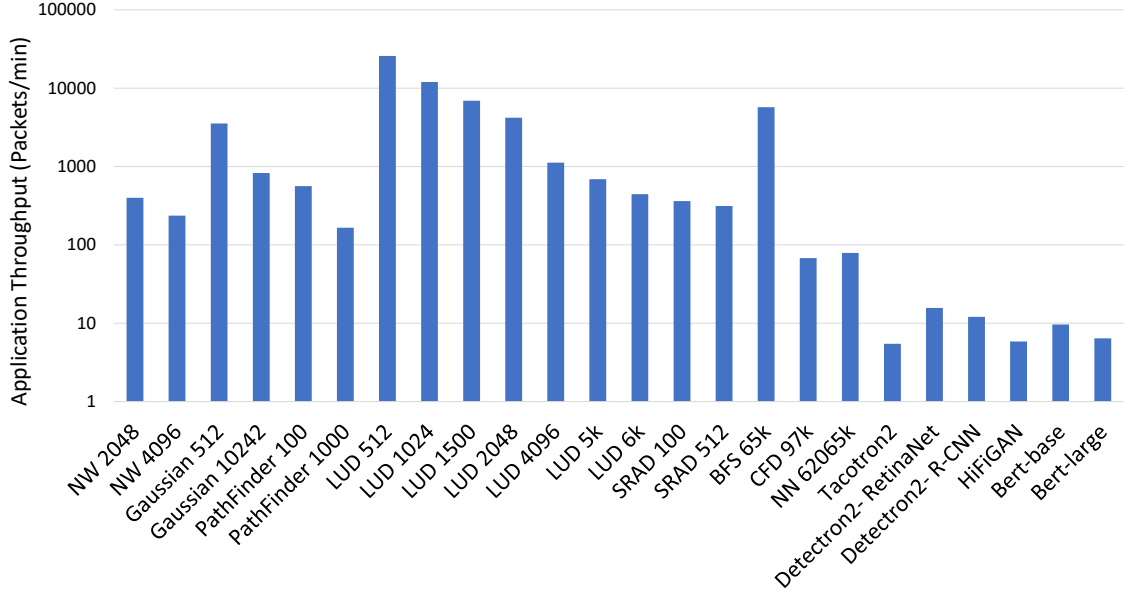


Figure 4.12: Throughput of benchmark applications on a logarithmic scale.

requirement, input processing rate and input drop rate. Using the input processing rate metric allows us to assess the impact of both GPU and CPU execution time. Figure 4.11 provides the Benchmark application characteristics regarding CPU and GPU execution time. It indicates that in some components, the CPU execution time is a significant portion of the component’s execution time. Therefore, it is crucial to consider CPU execution time when allocating resources to improve overall application throughput. In addition, our method considers the input drop rate when allocating resources. Figure 4.12 shows the normalized input processing rate of benchmark applications used as components in the test applications. The different processing rates between components in the application lead to input drops, which adversely affect throughput. On the other hand, the Slate approach executes the proper set of kernels simultaneously to maximize the system throughput. Considering the linear dependency in a given component-based application, Slate tries to maximize the throughput by executing all components together. As a result, by executing all the components and allowing them to access all the shared GPU resources using the spatial multitasking method, the Slate approach behaves similarly to the MPS-based approach. Thus, throughput is degraded in the Slate approach, as it allows all components to access GPU resources unrestrictedly without considering their resource requirements. Finally, the DGRA approach produces results that are closest to those yielded by our method by using an iterative process to find the appropriate resource fraction. However, removing certain resource fractions without thorough analysis leads to missed opportunities for improved resource allocation. Additionally, the absence of component categorization results in a uniform initial distribution of resources. This causes testing a broader range of resource

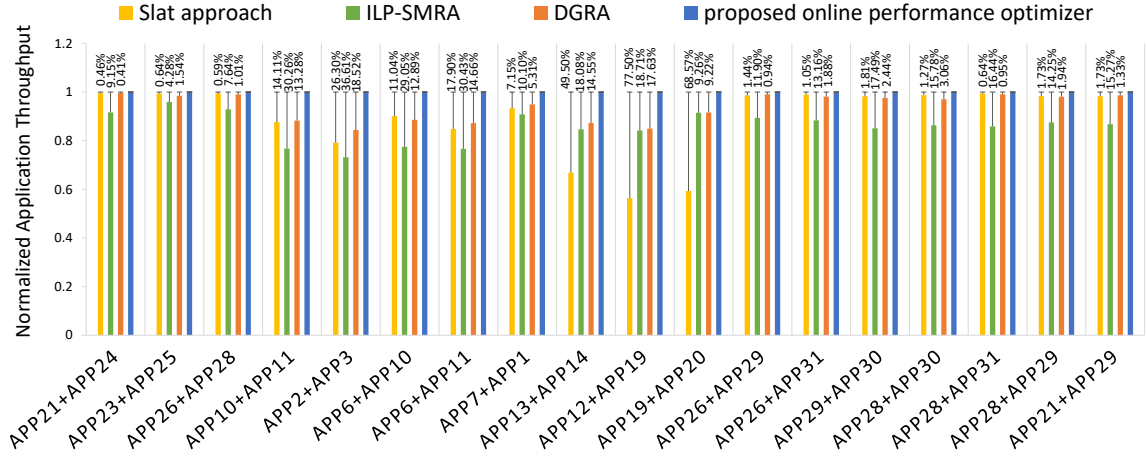


Figure 4.13: Normalized DAG application throughput comparison of various applications under consideration.

fractions, increasing the risk of incorrect allocations.

To assess the performance of our method on applications with the DAG structure, we generated DAG applications by combining linear applications with different categories from Table 4.3. These DAG applications exhibit different structures, such as splitting into two parallel paths that later converge (Diamond Structure) or dividing into two parallel branches (Tree Structure). Figure 4.13. presents the normalized application throughput comparison. The comparison results show up to 77.5%, 36.1%, and 18.52% throughput improvements compared to the Slate approach, ILP-SMRA method, and DGRA, respectively. The worst result is related to the Slate approach since it allows all components to access GPU resources unrestrictedly.

Our proposed method introduces some overhead to the component’s execution time. The main overhead is due to the communication with the memory and resource manager to allocate GPU memory and assign proper SM percentages, respectively. Considering that the component should wait for the servers’ response, this overhead also indicates the overhead of processing time in the servers. A pool of handles is designed for the first component to reduce the communication overhead. In addition, two memory pools are designed in the memory manager server to decrease the overhead of allocating memory from the GPU. However, communication with the GPU memory manager and GPU resource manager is still required in components and exposes additional overhead. To evaluate the communication overhead, we compare the execution times of the original and modified versions of the Rodina benchmark applications. In this experiment, we consider an application chain with only one component. We execute the application multiple times iteratively and calculate the communication overhead of communicating with the memory and resource manager during the execution time. The results of the overhead evaluation are presented in Figure 4.14. According to the results, the proposed method introduces up to a 4.1% overhead, which is

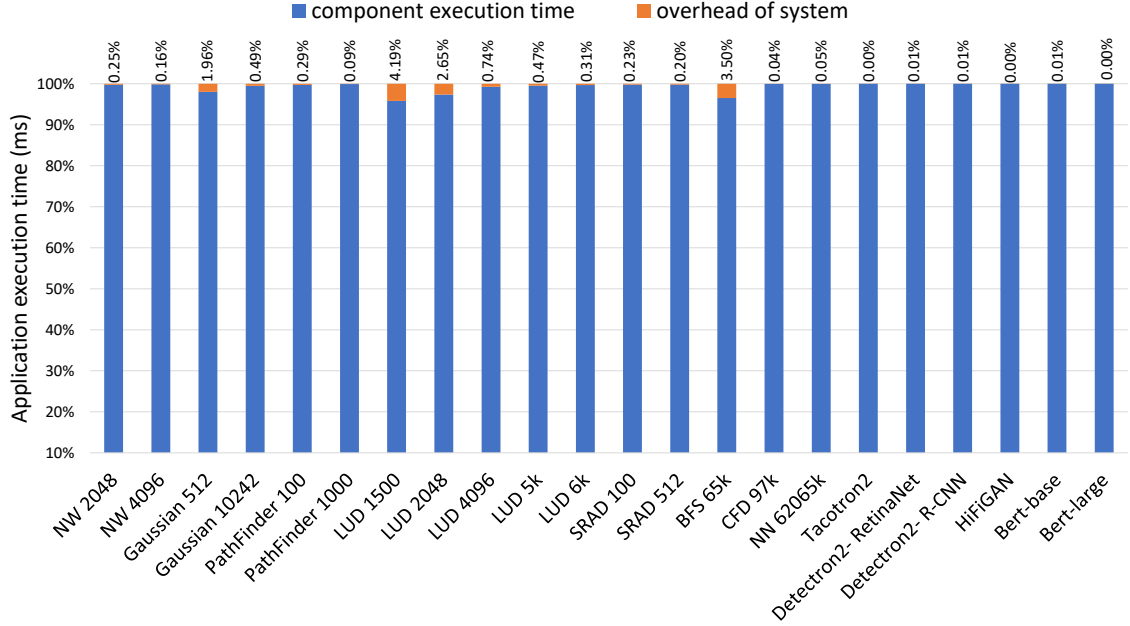


Figure 4.14: Overhead of the proposed method in benchmark applications.

significantly lower compared to the 134.12%, 71.22%, and 30.73% throughput improvements over the Slate, ILP-SMRA, and DGRA approach, respectively. We also observe that the overhead varies from one application to another. The reason is that communicating with servers occurs when the components process the input. As shown in Figure 4.12, different applications are associated with different execution times and processing rates. Accordingly, communication with memory and resource managers is more frequent in applications with faster processing rates, which results in higher communication overheads.

4.7 Conclusion

In this chapter, we proposed an efficient GPU resource management technique and a GPU memory-sharing solution to address the challenges of executing concurrent components of microservice-based applications on a GPU via spatial multitasking. We proposed a low-overhead approach that dynamically allocates the GPU computational resources using an online performance optimizer. Our evaluation results show that the proposed dynamic resource allocation method in various applications provides a significant performance improvement of up to 134.12% compared to a state-of-the-art spatial multitasking technique. Furthermore, we demonstrated that using shared memory results in a 6x throughput improvement over the baseline UDP-based technique. signing a method to further speed up the execution of microservice-based applications across multiple GPUs and addressing the challenges of optimizing performance with multiple GPUs can be an interesting research direction.

Chapter 5

Priority-based GPU Resource Allocation in Clouds¹

5.1 Introduction

GPUs, known for their extensive parallel processing capabilities, have become essential for handling compute-intensive workloads in cloud environments. They significantly enhance deep learning training and inference, data analytics, and emerging network services [117],[118]. However, GPUs are often expensive and typically available in limited quantities. To mitigate this, a range of GPU multitasking mechanisms have been proposed that allow multiple tasks to share a single device. These mechanisms, however, bring new difficulties. The first challenge is enforcing tenant isolation while preserving a configured priority policy across workload classes (e.g., SLA-critical vs. best-effort), so that higher-priority tenants receive predictable service without being impacted by co-located lower-priority tasks. Software-based sharing techniques allow multiple tasks to run simultaneously on a GPU and provide fine-grained control over their access to compute and memory [21], [5]. However, these techniques often suffer from weak isolation. If one task misbehaves or becomes overloaded, it can disrupt the performance of other tasks. On the other hand, hardware-level partitioning divides a GPU into smaller, more independent units, each with its own resources [12], [44]. This approach enhances isolation but reduces flexibility, as static partition layouts rarely align with the varying sizes of tasks, potentially leaving some capacity unused. In a dynamic multi-tenant cloud environment where workloads fluctuate over time, GPU allocation needs to adapt both in how devices are partitioned and how they are shared, while simultaneously guaranteeing isolation and honoring class-level priorities without sacrificing utilization.

The second challenge is achieving fairness within each priority class under heterogeneous

¹This chapter is based on an under-review journal paper:[8] Hoda Sedighi, F. Wuhib, and R. Glitho, “Priority-Aware Scheduling and Isolation-Preserving GPU Partitioning in the Cloud,” Submitted to *IEEE Transactions on Network and Service Management (TNSM)*, 2026

workload types and arrival patterns, while maintaining high GPU utilization. Many deployments involve continuous workloads that are expected to run indefinitely while adhering to strict SLAs. Examples of these include virtualized Radio Access Network (RAN) functions, such as virtual Distributed Units (vDUs) or PHY-layer acceleration, as well as long-lived online services like large language model (LLM) inference [119]. Alongside these continuous workloads, there are also time-limited tasks such as rApps, batch analytics, or ML training jobs. These time-limited tasks can be paused, preempted, or delayed to make room for urgent tasks. Providers typically assign higher priority to continuous, SLA-critical services and treat training or batch workloads as lower-priority, preemptible workloads. Traditional scheduling policies, like FIFO, strict priority queues, or Shortest-Job-First [12], [46], are often inadequate in this context. These methods might overlook differences in GPU demand, leading to inefficient capacity usage, or they might over-serve short or high-priority tasks, resulting in lower-priority tenants experiencing unpredictable service. Moreover, even within the same task class, interference and workload diversity can cause slowdowns unless fairness is explicitly enforced. A practical scheduler must support a two-layer concept of fairness: across different classes (e.g., continuous vs. time-limited, high-priority vs. best-effort), ensuring that high-priority workloads are consistently preferred; and within each class, ensuring that tasks share resources in a manner that prevents significant slowdowns for any tenant.

To overcome these challenges, this chapter introduces a priority-aware GPU-sharing framework designed for multi-tenant clusters managed by container orchestration systems. The main contribution of this work is meeting both goals (i) cross-class prioritization with isolation guarantees and (ii) intra-class fairness across mixed workload types, all while keeping GPUs highly utilized. We propose a priority-aware GPU-sharing framework for mixed workloads, using score-based scheduling and rescheduling to improve fairness, reduce waste, and protect high-priority tasks. In addition, we integrated the proposed method with orchestration and implemented it in both a simulator and a prototype.

The rest of this chapter is organized as follows. First, we present an illustrative use case followed by the system model and the problem formulation. Next, we present the proposed method and its components. Next, we evaluate the proposed method against state-of-the-art baselines and detail the experimental results. Finally, we summarize the contributions and findings.

5.2 Illustrative Use Case and Motivating Scenario

Modern cloud platforms increasingly rely on GPU-accelerated clusters that are shared among multiple users and services. Within the same pool of GPUs, providers are now co-locating network functions, such as RAN virtual network functions (VNFs), alongside a variety of AI workloads, including data analytics pipelines, large language model training

Table 5.1: Workload Classes, Types, and Runtime Characteristics

Service Class	Workload Type	Run Duration	Preemptability	Example Workloads
Class 1 (Guaranteed)	Continuous	Long	Non-preemptable	RAN processing chains (e.g., DU/PHY functions)
Class 1 (Guaranteed)	Time-limited	Short	Non-preemptable	Real-time rApps for RAN optimization
Class 2 (Best-Effort)	Continuous	Long	Preemptable	Generative AI pipelines (inference or training)
Class 2 (Best-Effort)	Time-limited	Short	Preemptable	AI model training / fine-tuning jobs

and inference, and other generative applications. Despite the differences, these workloads compete for a common pool of GPU accelerators in the data center. Figure 5.1 illustrates a conceptual model of a cloud environment composed of an orchestrator control plane, a scheduler, controllers, a container runtime, and GPU device plugins.

At the top of the model is the orchestrator control plane, which includes the scheduler and a set of controllers. This control plane provides an API that serves as the entry point for workload submissions and abstracts the underlying cluster. The scheduler selects suitable nodes and assigns workloads based on the available resources, including multiple GPUs per node. Meanwhile, the controllers manage the lifecycle of the workload, ensuring that workloads are created, monitored, and able to recover from failures. At the node level, each node hosts one or more GPUs and executes workloads via a container runtime (e.g., containerd). Finally, device plugins play a crucial role in discovering, advertising, and managing GPU resources for container allocation[20]. These components work together to manage a pool of multi-GPU nodes, allowing multiple users to submit various types of workloads, such as training, inference, and batch jobs, for execution across the cluster. The workflow operates as follows: Users submit their applications using the Orchestrator API, and the orchestrator then forwards accepted requests to the scheduler. The scheduler selects appropriate nodes based on resource availability and placement constraints. The selected nodes then instantiate the workloads, while controllers continuously monitor execution.

Workloads executing on shared GPU resources vary in terms of run duration, tolerance to interruptions, and the strictness of their performance guarantees. Based on these parameters, we propose a workload categorization summarized in Table 5.1. In our setting, we consider two service classes. *Class 1 (Guaranteed)* includes workloads that are sensitive to delays and require high reliability. These workloads must receive their requested GPU resources upon acceptance. Examples of Class 1 workloads include RAN-related processing (e.g., centralized or distributed baseband functions). An example of the latter is rApps that continuously optimize radio performance and cannot tolerate long waiting times. *Class 2 (Best-Effort)* includes opportunistic workloads that utilize available spare GPU capacity. Examples in this category include AI model training jobs and generative AI workloads, including both inference and training. These tasks can run for long periods but are preemptible and can be postponed when higher-priority services need resources.

In addition to service classes, we consider two types of workloads: *continuous* and *time-limited*. Continuous workloads run for long periods or indefinitely. In Class 1, this

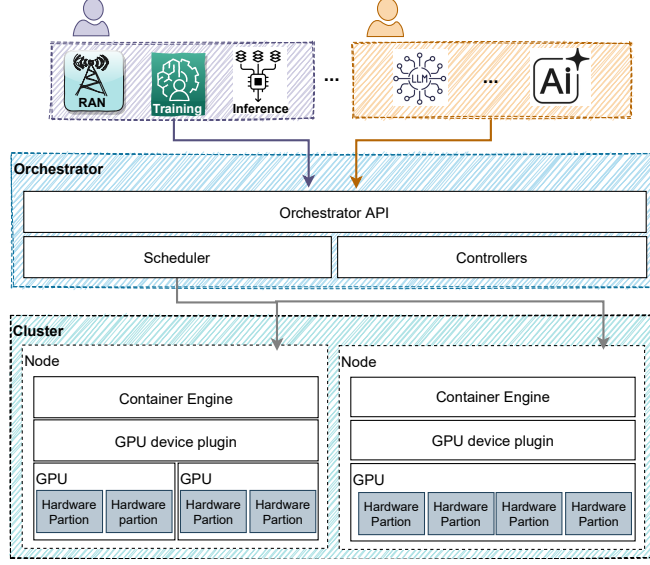


Figure 5.1: System model

includes non-preemptable RAN functions that must maintain a stable performance level once activated. In Class 2, we include long-running generative AI pipelines that can be preempted but still occupy GPUs for significant durations. In contrast, time-limited workloads are completed within a relatively short timeframe. Short, real-time applications (rApps) fall into the time-limited category of Class 1. They are bursty yet guaranteed and must be served quickly when they arise. Meanwhile, short training or fine-tuning tasks are part of the time-limited segment of Class 2. They are offered on a best-effort basis and can be delayed if higher-priority tasks are present.

At any given time, the system may host long-lived RAN, continuous generative AI workloads, urgent short-duration rApps, and best-effort training jobs, all competing for the same limited GPU resources. Without careful management, best-effort training or generative workloads can interfere with guaranteed services. Additionally, inflexible partitioning decisions can result in unused GPU capacity in partitions that do not have any active tasks. Figure 5.2 illustrates an example in a single node equipped with one GPU of normalized capacity 1.0 that can be divided into hardware partitions. The node hosts four tasks belonging to different classes and types:

- G_{RAN} : a Class 1, continuous, non-preemptable RAN processing chain that requires 0.2 GPU capacity;
- G_{rApp} : a Class 1, time-limited, non-preemptable rApp that must complete quickly and requires 0.4 GPU capacity;
- B_{GenAI} : a Class 2, continuous, preemptable generative AI workload (e.g., long-running inference or training) that requires 0.5 GPU capacity;

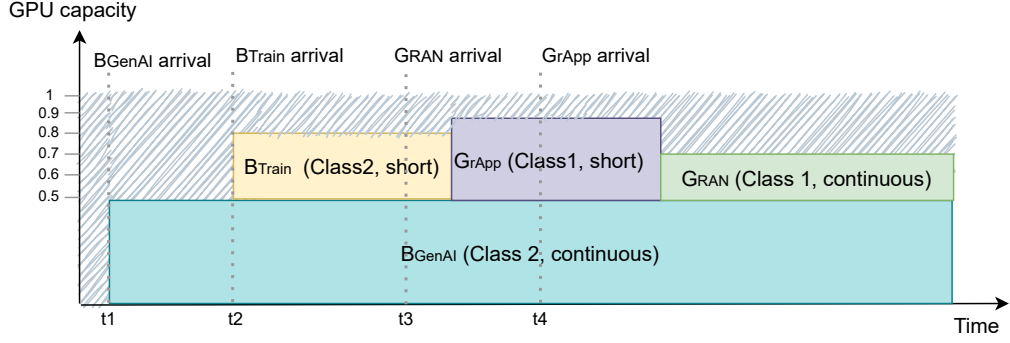


Figure 5.2: Time-based view of GPU utilization across two priority classes, showing interaction between guaranteed and best-effort workloads

- B_{Train} : a Class 2, time-limited, preemptable AI training job that requires 0.3 GPU capacity.

In the scenario depicted in Figure 5.2, the orchestrator employs a straightforward first-fit scheduling policy. The vertical axis represents the total GPU capacity, while the horizontal axis indicates time. The first workload to arrive is the best-effort continuous workload, denoted as B_{GenAI} , which is allocated a substantial hardware partition of size 0.5. Following this, the best-effort training task, B_{Train} , arrives and is assigned a second partition of size 0.3. This allocation leaves a small residual fragment of 0.2 GPU capacity, which cannot be reconfigured without relocating the existing tasks. When the high-priority continuous task G_{RAN} (which requires 0.4) and the short, guaranteed application G_{rApp} (which requires 0.2) arrive, there is no contiguous partition large enough to accommodate G_{RAN} . Consequently, both G_{RAN} and G_{rApp} must wait, as the best-effort tasks effectively tie up a significant portion of the GPU, even though both best-effort tasks are preemptable. This situation highlights a combination of priority inversion (where Class 2 workloads block Class 1) and fragmentation, resulting in unused capacity that could be allocated to more critical workloads.

Figure 5.3 illustrates another scenario involving a Class 2 (best-effort) workload for two users, A and C. Both users are requesting resources for time-limited training tasks that share the same priority class, but their tasks differ in size and duration. Tenant C submits numerous small tasks, each requiring 0.4 GPU capacity and having a short runtime. In contrast, Tenant A submits fewer but larger tasks, each needing 0.7 GPU capacity and lasting longer. With a strict priority policy applied across classes and a FIFO approach within each class, A's long task is scheduled to run on the GPU. As a result, C's shorter tasks must wait until a continuous 0.7 GPU share becomes available before they can start. This situation means that A's tasks experience minimal slowdowns, while C's tasks face significantly longer normalized completion times, despite being in the same priority class. This highlights the need for a fairness metric for each class, such as a normalized completion-time metric, to ensure that resource allocation does not unfairly disadvantage

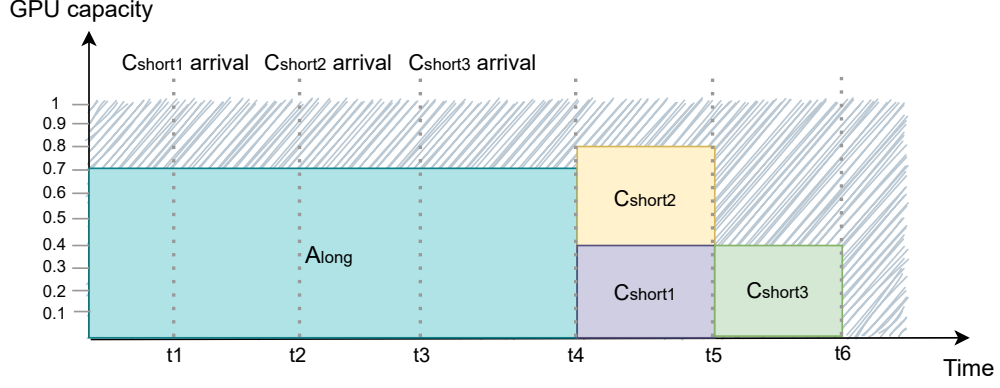


Figure 5.3: Time-based view of GPU utilization within one class showing unfair slowdowns

specific tenants.

These scenarios highlight two key challenges that motivate our framework: (i) the scheduler must differentiate between workloads of varying class priorities and between continuous and time-limited tasks (including preemptible and non-preemptible tasks). It must use this information to make informed decisions about which tasks to admit, preempt, or evict, ensuring that high-priority services are protected while still allowing lower-priority jobs to make progress; and (ii) GPU partitioning and fairness metrics need to be coordinated with these priority-aware decisions. This coordination is essential to prevent scenarios where tasks dominate large partitions, leading to fragmented capacity and delays for short tasks.

5.3 System model

We consider a GPU-enabled cloud cluster managed by a container orchestration platform. The cluster contains a set of physical GPUs, indexed by $g \in \{1, \dots, G\}$. Each GPU g has a normalized capacity of 1 and can be divided into a maximum of K_g hardware partitions (e.g., slices or instances), each with capacity $c_{gk} \in (0, 1]$. When software-level sharing is enabled, partitions can host one or multiple tasks. The cluster serves a set of tenants $u \in \{1, \dots, U\}$, each submitting GPU-accelerated tasks. For each task i , we denote $u(i)$ as its corresponding tenant, $r_i \in (0, 1]$ as the requested GPU capacity, and a_i as the arrival time of the task. The platform maintains an estimated service time $\hat{s}_i$ for each task, which is derived from prediction models that analyze task descriptors (such as resource requests or historical traces). The scheduler subsequently uses this information to assess completion times and potential slowdowns.

In addition to resource demand, each task has specific semantic attributes that influence scheduling decisions. We associate each task i with a *priority class* $q(i) \in \{1, \dots, Q\}$, where lower numbers indicate higher priority (for instance, Class 1 is for guaranteed services, while Class 2 is for best-effort workloads). We also differentiate between *workload type* $t(i) \in \{\text{continuous}, \text{time-limited}\}$, which shows whether the task represents a long-running

service (like RAN processing or online inference) or a finite computation (such as training or analytics). A preemptibility flag $p(i) \in \{0, 1\}$ indicates if the task can be paused or evicted: continuous guaranteed tasks are usually non-preemptable ($p(i) = 0$), while many best-effort, time-limited tasks can be preempted ($p(i) = 1$). The scheduler selects a set of runnable tasks at any given time and assigns them to GPU partitions, taking into account each GPU’s total capacity.

We use time-based performance metrics to evaluate and guide scheduling. For each task i , let C_i represent its completion time. The flow time is defined as $F_i = C_i - a_i$, where a_i is the arrival time of the task. Our system enforces a two-layer policy based on this model: first, tasks of higher priority should be prioritized when there is competition for capacity across priority classes; second, within each class, tasks from different tenants should experience balanced normalized completion times to ensure that no tenant is unfairly penalized. The next section will formalize the objective and constraints that arise from this model.

5.4 Problem Formulation

Building on the presented system model, our objective is to design a resource management policy that jointly decides (i) how to allocate GPU partitions to tasks over time and (ii) which tasks to admit, preempt, or evict. This approach aims to protect high-priority services, respect isolation constraints, and ensure fair resource sharing within each priority class. The environment we are dealing with is dynamic, as tasks arrive over time with varying GPU demands, priority classifications, workload types (either continuous or time-limited), and preemption properties. As demonstrated by our motivating scenarios, simplistic policies can lead to issues such as priority inversion (high-priority tasks being blocked by low-priority tasks even though the latter are preemptable) and significant slowdowns for specific tenants within the same class when long tasks monopolize the GPU. Therefore, the challenge is to coordinate GPU partitioning and task scheduling to maintain cross-class priority ordering while achieving balanced normalized completion times within each class. Following, we present our problem formulation. A summary of the notation is provided in Table 5.2.

Input parameters: Let $\mathcal{I}$ denote the set of tasks under consideration. For each task $i \in \mathcal{I}$ we are given the following information:

- Task’s tenant $u(i)$, representing the corresponding tenant for the task
- Task’s priority class $q(i) \in \{1, \dots, Q\}$, where lower numbers indicate higher priority
- Task’s workload type $t(i) \in \{\text{continuous}, \text{time-limited}\}$

Table 5.2: Summary of main notations

Annotation	Description
G	Number of physical GPUs in the cluster
g	Index of a GPU, $g \in \{1, \dots, G\}$
K_g	Number of hardware partitions on GPU g
c_{gk}	Capacity of partition k on GPU g (normalized)
U	Number of tenants in the system
u	Index of a tenant, $u \in \{1, \dots, U\}$
$\mathcal{I}$	Set of all tasks in the system
i	Index of a task, $i \in \mathcal{I}$
$u(i)$	Tenant to which task i belongs
$q(i)$	Priority class of task i (1 = highest)
$t(i)$	Workload type of task i (continuous / time-limited)
$p(i)$	Preemptibility flag of task i (1 = preemptable)
r_i	Requested GPU capacity of task i
a_i	Arrival time of task i
$\hat{s}_i$	Estimated service time of task i
C_i	Completion time of task i
F_i	Flow time of task i , $F_i = C_i - a_i$
ρ_i	Normalized completion time of task i , $\rho_i = F_i / \hat{s}_i$
$x_{i,gk}(t)$	Binary variable: task i runs on partition k of GPU g at time t

- Task's GPU demand $r_i \in (0, 1]$, representing the fraction of a GPU
- The arrival time a_i , the instant at which the task becomes known to the scheduler
- The estimated service time $\hat{s}_i$ is the time required to complete the task if it is allocated the requested GPU.
- The preemptibility flag $p(i) \in \{0, 1\}$, where $p(i) = 1$ denotes that the task may be paused or evicted (typical for best-effort workloads), while $p(i) = 0$ denotes a non-preemptable task (typical for guaranteed services);

During execution, each task i is assigned to a GPU partition. Considering that $\hat{s}_i$ represents the estimated completion time for the task and C_i denotes its actual completion time.

Decision variables: We divide time into discrete decision moments, denoted as $t \in \mathcal{T}$. For each GPU g , partition k , and task i , we define a binary variable $x_{i,gk}(t)$. This variable equals 1 if task i is running on partition k of GPU g at time t , and 0 otherwise. The scheduling policy manages the changes in $x_{i,gk}(t)$ over time, including the initiation of tasks, the pausing or eviction of preemptible tasks, and the reconfiguration of GPU partitions.

Objective: Our primary objective is to implement a two-layer approach to fairness while respecting task priority. Higher-priority tasks should not be delayed excessively by lower-priority workloads. Additionally, tasks from different tenants within the same priority class should have balanced normalized completion times. To incorporate class-level priority, we associate with each priority class $q \in \{1, \dots, Q\}$ a weight $w_q \in \mathbb{R}_{>0}$ such that

$$w_1 > w_2 > \dots > w_Q. \quad (5.1)$$

where larger weights correspond to higher-priority classes.

The flow time for each task is defined as $F_i = C_i - a_i$, which takes into account both waiting and execution time. We also define the normalized completion time as follows:

$$\rho_i = \frac{F_i}{\hat{s}_i}, \quad (5.2)$$

which shows how much slower the task completes compared to its predicted execution time. Additionally, to capture the evolution of task progress over time, we define the instantaneous normalized flow time for task i at time t as follows:

$$\rho_i(t) = \begin{cases} (\frac{t - a_i}{\hat{s}_i}) \sum_{g=1}^G \sum_{k=1}^{K_g} x_{i,gk}(t), & \text{if } a_i \leq t \leq C_i, \\ 0, & \text{otherwise,} \end{cases} \quad (5.3)$$

To assess the worst normalized completion time for each priority class at time t , we define the class-wise worst priority-weighted normalized completion time as follows:

$$Z_q(t) = \max_{i:q(i)=q} (w_q \rho_i(t)), \quad q = 1, \dots, Q. \quad (5.4)$$

In this equation, $q(i)$ represents the priority class of task i . The notation $\max_{i:q(i)=q}$ indicates that the maximum is computed over all tasks i for which the class index satisfies $q(i) = q$.

Over a finite period of time, $[0, \Delta]$, we calculate the time-averaged class-wise objective for each class as:

$$\bar{Z}_q = \frac{1}{\Delta} \int_0^\Delta Z_q(t) dt, \quad q = 1, \dots, Q. \quad (5.5)$$

In this equation, $\bar{Z}_q$ for class q represents the worst priority-weighted normalized completion time observed over time. Our scheduling problem involves selecting time-varying allocations $x_{i,gk}(t)$ to lexicographically minimize the vector $(\bar{Z}_1, \bar{Z}_2, \dots, \bar{Z}_Q)$. This means we first aim to reduce $\bar{Z}_1$ (the highest priority class), then $\bar{Z}_2$, while keeping $\bar{Z}_1$ optimal, and so on. Therefore, our objective is to:

$$\min_{\{x_{i,gk}(t)\}} (\bar{Z}_1, \bar{Z}_2, \dots, \bar{Z}_Q) \quad (\text{lexicographic order}). \quad (5.6)$$

This approach ensures that higher-priority classes are protected, allowing improvements for lower-priority classes only if they do not negatively impact the worst normalized completion time of any higher-priority class. Preemptibility and capacity constraints restrict the set of feasible schedules (for example, non-preemptible tasks cannot be interrupted once they have started). The scheduler then uses the remaining flexibility to minimize the class-specific objectives $(Z_1, \dots, Z_Q)$ while ensuring that the GPUs remain well utilized.

Constraints: The decision variables are subject to the following constraints for all decision times t . The first constraint is the Capacity constraints. For each GPU g , considering that c_{gk} represents the capacity of the partition k

$$\sum_{k=1}^{K_g} \sum_{i \in \mathcal{I}} x_{i,gk}(t) r_i \leq \sum_{k=1}^{K_g} c_{gk}, \quad (5.7)$$

We define constraint 5.7 to ensure that the total allocated capacity does not exceed the sum of the partition capacities. Additionally, we need to implement isolation and placement constraints. Each task can run on only one partition at a time,

$$\sum_g \sum_k x_{i,gk}(t) \leq 1, \quad \forall i \in \mathcal{I}, \quad (5.8)$$

and cannot be scheduled before it arrives,

$$x_{i,gk}(t) = 0 \quad \text{for all } t < a_i. \quad (5.9)$$

Furthermore, regarding the preemptibility constraint, if $p(i) = 0$ and task i has already started (i.e., $t < C_i$), it cannot be interrupted; in other words, once a non-preemptable task is admitted, $x_{i,gk}(t)$ must remain equal to 1 in a given partition until the task is completed. Tasks with $p(i) = 1$ can be paused or migrated, as long as capacity and isolation constraints are maintained.

Finally, we establish the priority ordering constraint. When capacity is limited, tasks with smaller $q(i)$ values should be prioritized over those with larger $q(i)$ values. Specifically, any schedule where a lower-priority task is executed while a higher-priority task is waiting is considered suboptimal and dominated under the objective above. The goal of these formulations is to develop a dynamic scheduling and partitioning policy that determines the appropriate values of $x_{i,gk}(t)$ over time. This approach should meet the specified constraints while optimizing the normalized completion times for each class. In the following section, we outline our priority-aware framework and the algorithms that implement it.

5.5 Proposed Solution

This section presents our priority-aware GPU sharing framework, designed to address the two challenges identified in the introduction: (i) enforcing isolation and priority ordering across different classes while effectively managing GPU partitioning and task placement, and (ii) achieving fairness within each class under continuous task arrivals and varying workload types. Our framework functions as a control loop integrated with the orchestration system. Specifically, it focuses on the internal logic of the scheduler component in the Figure 5.1. At each decision point, it observes the current tasks and the state of GPU partitions, updates priority-aware fairness metrics, and determines which tasks to start, continue, preempt, or keep pending. Under specific circumstances, the framework may modify the GPU partition layout if it detects ongoing fragmentation or a breach of priority. The method is organized around four main elements:

- **Priority-Aware Task Scoring Module:** This component assigns a global priority score to each task.
- **Task Grouping and Partition Feasibility Mechanism:** This mechanism creates task groups for each tenant and assesses their feasibility in co-location, considering the current partition layout and the specific set of supported partition configurations.
- **Priority-Aware Scheduling Algorithm:** Utilizing the outputs from the previous modules, this algorithm determines task admissions and, when necessary, preempts lower-priority tasks that can be interrupted.
- **Partition Manager:** This component reconfigures GPU partition layouts when task groups remain unfeasible under the current partition configurations. Its goal is to reduce unused capacity while ensuring tenant isolation and priority guarantees.

Figure 5.4 provides a detailed view of the scheduler in Figure 5.1, illustrating the proposed priority-aware scoring, grouping, scheduling, and partitioning modules.

5.5.1 Priority-Aware Task Scoring

In the proposed scheduler framework, the Priority-Aware Task Scoring module is responsible for calculating a score for each task to guide decision-making. The main approach of this module is summarized in Algorithm 5.1. At each decision point t , all tasks from all tenants are conceptually organized into a single global queue. Each task i is assigned a discrete priority class index $q(i)$, which ranges from $q(i) \in \{1, \dots, Q\}$. Here, smaller indices represent higher-priority classes. For instance, $q = 1$ may represent guaranteed service, whereas $q = Q$ indicates best-effort service in a typical configuration. The framework, however, is flexible enough to accommodate any number of classes, with $Q \geq 2$. For every task i , the Priority-Aware Task Scoring module uses:

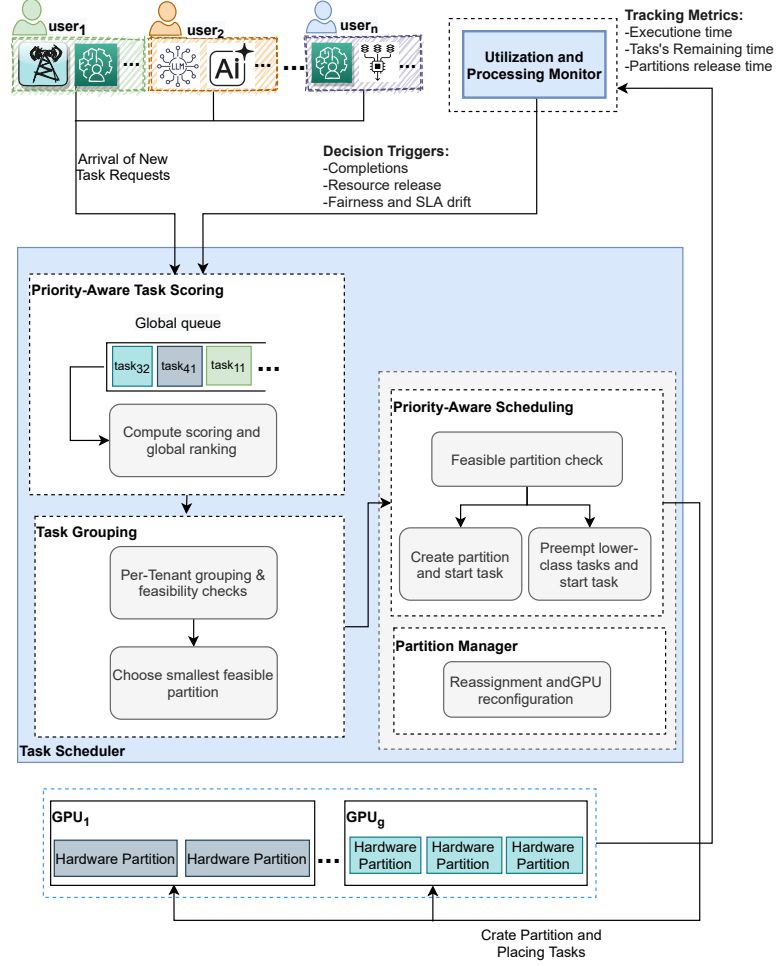


Figure 5.4: Overview of the proposed priority-aware GPU sharing framework implemented within the scheduler component of the orchestrator control plane shown in Figure 5.1

- The task's normalized completion ratio at time t , $\rho_i(t)$, calculated from equation (5.3).
- The priority class index of the task, represented as $q(i)$.
- The workload type of the task, $t(i)$, which indicates whether the task is *continuous* (long-running and SLA-oriented) or *time-limited* (such as batch processing).
- The preemptibility flag of the task, denoted as $p(i)$, where $p(i) = 1$ indicates that the task can be preempted, and $p(i) = 0$ indicates that it cannot.

In our proposed algorithm, priority is conveyed through two layers. First, we associate each class q with a global class score $G^{(q)} > 0$, and set

$$G_i = G^{(q(i))}. \quad (5.10)$$

These constants control the separation between classes in the score space, with smaller values typically assigned to higher-priority classes. For example, when dealing with two classes, we can set $G^{(1)} \ll G^{(2)}$ (e.g., $G^{(1)} = 1$, $G^{(2)} = 10^3$). This approach ensures that best-effort tasks receive significantly greater weight compared to guaranteed tasks. Additionally, within each class, we assign a configurable base score B_i that refines the ordering of tasks within that class and captures the workload type. Specifically, we define:

$$B_i = \begin{cases} B^{(q(i), \text{cont})}, & \text{if } t(i) = \text{continuous}, \\ B^{(q(i), \text{batch})}, & \text{if } t(i) = \text{time-limited}, \end{cases} \quad (5.11)$$

where $B^{(q, \text{cont})}$ and $B^{(q, \text{batch})}$ represent constants for class q . This setup allows continuous tasks and time-limited tasks within the same class to receive slightly different weights. For instance, it allows prioritizing continuous SLA services or, conversely, granting a small advantage to short batch tasks. Finally, the combined priority weight of task i is defined as follows:

$$w_i = G_i \cdot B_i, \quad (5.12)$$

This represents a priority weight for each task. In the optimization model discussed in Section III, priority is indicated by the class weights w_q . Here, w_i serves as a refined weight for task i that combines the importance at the class level (related to $w_{q(i)}$) with intra-class differentiation and workload type. We ultimately define the global scoring of task i at time t as follows:

$$S_i(t) = \frac{w_i}{\rho_i(t) + \varepsilon} + \eta(1 - p(i)), \quad (5.13)$$

Where $\varepsilon > 0$ is a small constant that helps avoid division by zero for newly arrived tasks with $\rho_i(t) = 0$, and $\eta > 0$ is a small constant used for tie-breaking. Intuitively, w_i represents the cost of delaying task i based on its class and intra-class priority, while $\rho_i(t)$ indicates how long it has already waited and run compared to its estimated completion time. For a fixed w_i , the term $w_i/(\rho_i(t) + \varepsilon)$ decreases as $\rho_i(t)$ grows. This means that tasks with a higher normalized completion time receive lower scores and, therefore, become more urgent. The additional term $\eta(1 - p(i))$ slightly penalizes non-preemptable tasks (where $p(i) = 0$), ensuring that, when two tasks have similar priority weight and normalized completion ratio, the preemptable one is favoured in the ordering. Across different classes, the scale of $G^{(q)}$ ensures that higher-priority classes tend to have lower scores when their normalized completion times are comparable. This prevents low-priority tasks from preempting running guaranteed services.

In the scheduler algorithm, the scheduler utilizes $S_i(t)$ as a global urgency indicator and

Algorithm 5.1: Priority-Aware Task Scoring

```
1: Input Active task set  $\mathcal{I}$ ; parameters  $\varepsilon, \eta$ ; class constants  $G^{(q)}$ ; base constants  $B^{(q, \text{cont})}$  and  $B^{(q, \text{batch})}$ 
2: Output Scores  $\{S_i(t)\}$  and a global ordered list  $L(t)$  of pending tasks
3: for each task  $i \in \mathcal{I}$  do
4:   Compute  $\rho_i(t)$  using (5.3)
5:    $G_i \leftarrow G^{(q(i))}$  (5.10)
6:   if  $t(i) = \text{continuous}$  then
7:      $B_i \leftarrow B^{(q(i), \text{cont})}$ 
8:   else
9:      $B_i \leftarrow B^{(q(i), \text{batch})}$ 
10:  end if
11:   $w_i \leftarrow G_i \cdot B_i$  (5.12)
12:   $S_i(t) \leftarrow \frac{w_i}{\rho_i(t) + \varepsilon} + \eta(1 - p(i))$  (5.13)
13: end for
14:  $L(t) \leftarrow$  pending tasks in non-decreasing order of  $S_i(t)$ 
15: return  $\{S_i(t)\}$  and  $L(t)$ 
```

sorts tasks in non-decreasing order of $S_i(t)$. This means that tasks with smaller $S_i(t)$ values are prioritized and examined first when creating candidates for allocations. Conversely, tasks with larger $S_i(t)$ values are considered as prime candidates for preemption when capacity needs to be freed and $p(i) = 1$. This approach allows us to integrate class-level priority, workload type, intra-class preferences, and fairness signals (through $\rho_i(t)$) into a single scalar ranking across all users before grouping tasks. Although the fairness objective outlined in Section 5.4 is expressed in terms of the class-wise worst priority-weighted normalized completion time $Z_q(t)$, the score shown in equation (5.13) serves as a practical heuristic derived from the same components.

The Priority-Aware Task Scoring module is invoked whenever the scheduler is triggered. In our design, this occurs during specific events, such as task arrivals, task completions, or changes in resource availability (e.g., when partitions are released). The module is also invoked periodically if it detects a decrease in fairness or a violation of SLA. During each of these occasions, our method utilizes heuristics based on the global scores $S_i(t)$ and the overall prioritization. The resulting rankings are then forwarded to the grouping and placement components, which are described in the following subsections. Regarding the complexity analysis, let $N = |\mathcal{I}(t)|$ be the number of tasks visible at decision point t . If tasks are not already organized in a priority queue, generating the global ordering requires $O(N)$ time, and the sorting process itself costs $O(N \log N)$. Therefore, the overall complexity of the Priority-Aware Task Scoring algorithm is $O(N \log N)$.

5.5.2 Task Grouping and GPU Partition Selection

Given the per-task scores and the current GPU partition state, the Task Grouping is responsible for creating candidate task groups that can be co-located on GPUs without

violating capacity or isolation constraints. This approach enables optimal use of GPU resources and prevents fragmentation. Hardware partitions (e.g., MIG instances in NVIDIA GPUs) serve as the primary isolation boundary between tenants. For any specific GPU model, the set of acceptable hardware partitions is fixed and typically limited. The GPU exposes only a discrete set of capacities $\{C^{(1)}, C^{(2)}, \dots, C^{(L)}\}$ with limited granularity. Considering that, even the smallest hardware partition $C^{(1)}$ can exceed the resource demand of an individual task. As a result, placing one task per partition would either waste capacity or require an impractical number of partitions. To address this, it is preferable to co-locate multiple tasks from the same tenant within a single hardware partition and to shape their combined resource demand so that it fits an available hardware partition as tightly as possible. Additionally, if a partition's capacity significantly exceeds the total demand of the tasks assigned to it, the leftover space cannot be allocated to other tenants. This situation results in internal fragmentation, leaving a considerable portion of the partition idle until the tasks are completed. Therefore, the grouping logic is crucial in reducing both fragmentation and underutilization of GPU resources.

At each decision point t , after calculating $S_i(t)$ as described in the previous subsection, the framework constructs *tenant-scoped groups* and assigns each group a *group-level urgency* that will be used by the scheduler. Importantly, *group construction is capacity-aware*: although admission decisions occur later, the grouping step uses the *currently available partition residuals* (and the discrete partition catalog) to decide which target capacity a tenant group should pack toward, thereby reducing fragmentation and stranded capacity.

In Capacity-aware per-tenant grouping and group-score assignment, we first receive all tasks in $\mathcal{I}$ in non-decreasing order of $S_i(t)$, ensuring that tasks with lower scores (indicating higher urgency) are processed first. We then scan this list and create groups as follows:

1. Take the first task i in the global list that is not yet assigned to any group. Let $u = u(i)$ be its tenant and $q = q(i)$ its priority class. We construct a group G that is *tenant- and class-consistent*, i.e., G may only contain tasks from tenant u and class q .
2. Select a target capacity based on current residuals. Let $\mathcal{P}(t)$ denote the set of existing GPU partitions at time t and let $\text{res}_{gk}(t)$ be the residual capacity on partition $(g, k) \in \mathcal{P}(t)$ after accounting for currently running workloads. Define the set of residual capacities that are currently usable by tenant u as

$$\mathcal{C}_u^{\text{free}}(t) = \{\text{res}_{gk}(t) : (g, k) \in \mathcal{P}(t), \text{res}_{gk}(t) \geq r_i, \}.$$

If $\mathcal{C}_u^{\text{free}}(t) \neq \emptyset$, set the target capacity to the tightest currently feasible residual:

$$\tilde{C}_G(t) = \min \mathcal{C}_u^{\text{free}}(t). \quad (5.14)$$

Otherwise, fall back to the discrete catalog of supported partition profiles

$\{C^{(1)}, \dots, C^{(L)}\}$ (with $C^{(1)} < \dots < C^{(L)}$) and set

$$\tilde{C}_G(t) = \min \left\{ C^{(\ell)} : C^{(\ell)} \geq r_i \right\}, \quad (5.15)$$

and mark G as *potentially requiring re-partitioning or preemption* at admission time.

3. Initialize $G \leftarrow \{i\}$ and then consider subsequent tasks j in the global list that belong to the same tenant $u(j) = u$, $q(j) = q$, and j is not yet grouped. Among these candidates, we select tasks that both:

- have score and estimated (remaining) execution times similar to those of i , and
- keep the aggregate demand within the selected target capacity:

$$R(G) + r_j \leq \tilde{C}_G(t), \text{ where } R(G) = \sum_{\ell \in G} r_\ell. \quad (5.16)$$

If so, add j to the group and update R_u :

$$G_u \leftarrow G_u \cup \{j\}, \quad R_u \leftarrow R_u + r_j. \quad (5.17)$$

4. Stop extending G_u when either (i) adding another task of tenant u would exceed $\tilde{C}_G(t)$, or (ii) there are no more suitable tasks from u remaining in the global list.
5. Assign a group-level urgency score. After G is formed, define its urgency as the most urgent member score:

$$U_G(t) = \min_{i \in G} S_i(t), \quad (5.18)$$

and define a group-level preemptibility flag

$$p(G) = \min_{i \in G} p(i), \quad (5.19)$$

so that $p(G) = 1$ only if *all* tasks in the group are preemptable.

This grouping has three benefits: (i) it preserves tenant isolation by construction (each G is tenant-scoped), (ii) it reduces fragmentation by packing urgent same-tenant tasks toward a tight currently feasible residual when one exists (otherwise toward the smallest feasible catalog profile), and (iii) it provides a single scalar $U_G(t)$ so that subsequent admission and preemption decisions can operate at the group level. Groups that cannot be made feasible under the current layout are flagged as requiring either re-partitioning and/or preemption of lower-priority groups with $p(G) = 1$; these flags are processed by the scheduling and partition-management logic.

In the context of complexity analysis, let $N = |\mathcal{I}|$ denote the number of tasks, $P = |\mathcal{P}(t)|$ represent the number of active partitions, and M signify the number of groups formed. The

process of grouping tasks involves a scan that takes $O(N)$ time, assuming each task is assigned only once. To select $\tilde{C}_G(t)$, it is necessary to check the currently usable residuals. If this is implemented naively, it requires $O(P)$ time per group, leading to a total time complexity of $O(N + MP)$.

5.5.3 Priority-Aware Scheduling Algorithm

Finding the optimal grouping, partitioning, and placement that minimises the fairness objective is NP-hard. It combines features of multi-dimensional bin packing (due to discrete partition capacities and a limited set of admissible partitions per GPU) with priority-aware scheduling under preemption. Our method therefore relies on heuristics guided by per-task scores $S_i(t)$ and the partition-size structure. In particular, we first score tasks, then form tenant-scoped groups that *inherit* a group-level urgency $U_G(t)$, and finally schedule *groups* (rather than individual tasks) to balance system-wide urgency, tenant isolation, and fragmentation control.

The primary role of the scheduler is to determine, at each decision point, which groups should be started immediately, which should remain pending, and when it is appropriate to preempt lower-priority groups in order to accommodate higher-priority ones. Additionally, given the hardware’s limited granularity, the scheduler and partition manager must decide how to configure the GPUs using acceptable partition profiles and determine when to reconfigure them to meet increased demand.

Algorithm 5.2 outlines the main responsibility of the scheduling process. This algorithm is triggered by significant events, such as the arrival or completion of tasks, changes in resource availability (for instance, when a partition is released), and periodically to detect fairness or SLA degradation. As shown in algorithm 5.2, the outer loop processes classes in order of priority, starting with the highest and progressing to the lowest. This approach aligns with the lexicographic objective outlined in Section III, which involves $(\bar{Z}_1, \dots, \bar{Z}_Q)$. Within each class, groups are prioritized based on the urgency score $U_G(t)$ from equation (5.18). These groups are processed in non-decreasing order, meaning that those with the most urgent tasks are handled first. This method ensures strict class priority while maintaining fairness within each class based on task scores.

For admitting each high-priority group G , the scheduler first attempts to find the target capacity $\tilde{C}_G(t)$ using admissible partition profiles. If that fails, it considers preemption only of lower-priority groups that are explicitly preemptable ($p(H) = 1$). Eviction candidates are chosen to minimize disruption by preempting groups with larger $U_H(t)$ first (least urgent), preempting only the minimum number needed to admit G . Non-preemptable groups ($p(H) = 0$) are never evicted. If admission is still infeasible, G remains pending until capacity becomes available or the partition manager reshapes the layout.

Regarding the Complexity, let $N = |\mathcal{I}|$ be the number of active tasks, $M = |\mathcal{G}(t)|$ the number of tenant-scoped groups produced by the grouping step, and $P = |\mathcal{P}(t)|$

Algorithm 5.2: Priority-Aware Scheduling and Preemption Control

```

1: Initialize task set  $\mathcal{I} \leftarrow \emptyset$  and partition state.
2: while cluster is running do
3:   Observe trigger (arrival, completion, release, periodic check); update  $\mathcal{I}$ .
4:   Compute  $\{S_i(t)\}$  for all  $i \in \mathcal{I}$  (Algorithm 5.1).
5:    $\mathcal{G}(t) \leftarrow$  run Grouping Procedure on  $\mathcal{I}$ 
6:   for  $q = 1$  to  $Q$  do
7:      $\mathcal{G}_q \leftarrow \{G \in \mathcal{G}(t) : q(G) = q\}$ .
8:     Sort  $\mathcal{G}_q$  in non-decreasing order of  $U_G(t)$ .
9:     for each group  $G \in \mathcal{G}_q$  in that order do
10:      if  $G$  can be admitted under current layout then
11:        Start all tasks in  $G$  on an isolation-respecting partition.
12:      else
13:        Attempt to create a partition of size  $\tilde{C}_G(t)$ .
14:        if reconfiguration succeeds then
15:          Start  $G$  on the newly created/resized partition.
16:        else
17:          Identify running lower-priority groups eligible for eviction:
18:           $\mathcal{J} \leftarrow \{H : q(H) > q, p(H) = 1\}$ .
19:          if  $\mathcal{J} \neq \emptyset$  then
20:            Sort  $\mathcal{J}$  by non-increasing  $U_H(t)$  (evict least-urgent first).
21:            Preempt a minimal prefix of  $\mathcal{J}$  to free capacity for  $G$ .
22:            Re-attempt admission for  $G$ ; start if feasible.
23:          else
24:            Keep  $G$  pending and signal the partition manager for a future reconfiguration attempt.
25:          end if
26:        end if
27:      end for
28:    end for
29: end while

```

the number of currently instantiated partitions in the cluster. Per decision epoch, the scheduler (Algorithm 5.2) first computes per-task scores and orders them in $O(N \log N)$ time (Algorithm 5.1), and then partitions groups by priority class in $O(N + MP)$. Sorting groups within classes costs

$$\sum_q O(|\mathcal{G}_q| \log |\mathcal{G}_q|) \leq O(M \log M).$$

For each group, admission checking and partition selection tests require $O(P)$ time in the worst case. If admission fails, the scheduler identifies lower-priority running groups eligible for eviction in $O(R)$ time, sorts them by urgency in $O(R \log R)$ time, and then preemptively evicts a minimal prefix. R is the number of eviction-eligible running lower-priority groups. In the worst case, such eviction handling may be triggered for every group, yielding an overall cost of $O(M(M + M \log M))$. Therefore, total per-epoch worst-case complexity is

$$O(N \log N + MP + M^2 \log M).$$

Since each group contains at least one task, we have $M \leq N$, and thus the worst-case per-epoch complexity is bounded by $O(N^2 \log N)$. In practice, however, the number of groups and eviction candidates is typically much smaller than the number of tasks.

5.5.4 GPU Repartitioning and Scheduling Interaction

The fast-timescale scheduling loop (Algorithm 5.2) operates on the *current* partition layout at each decision epoch. However, because the hardware exposes only a *discrete* catalog of admissible partition profiles (and a restricted set of profile combinations per GPU), the layout itself may become mismatched with the evolving mix of demands. Reconfiguring partitions is also costly, it may require draining or preempting work on the affected slices. Our framework therefore adopts a two-timescale design:

- **Fast timescale (group admission and preemption on a fixed layout).** At each trigger (arrival, completion, release, periodic check), we compute per-task scores and run the grouping procedure, which assigns each tenant group G a group-level urgency score (e.g., $U_G(t)$) and a target partition profile based on the current residuals and the discrete catalog. The scheduler then performs group-level admission decisions: it starts the most urgent feasible groups on existing partitions and, when necessary, preempts lower-priority *running groups* (or tasks within those groups) that are explicitly preemptable to free sufficient capacity for a higher-priority group. The fast-timescale loop does not reshape the partition layout; it only admits, continues, or preempts work under the current layout constraints.
- **Slow timescale (layout reconfiguration and global reassignment).** A partition manager runs periodically or when the system exhibits persistent infeasibility (e.g., high-priority groups remain unplaceable across multiple epochs) or sustained fragmentation (capacity becomes stranded due to mismatches between group demands and the discrete partition configurations). When triggered, the partition manager executes the same priority-aware scoring and grouping logic, but over the entire active task set (running and pending), rather than only newly arrived or currently unplaced work. In effect, it recomputes group scores/urgencies and a consistent global grouping plan, then selects a new partition layout (from the admissible catalog and GPU combination rules) that improves feasibility for high-priority groups while reducing stranded capacity for best-effort groups. Because reconfiguration is disruptive, it is applied conservatively: changes are scheduled when impacted slices are idle or occupied only by lower-priority preemptable work, so that reshaping has bounded impact on protected workloads.

This coordination between fast-timescale group scheduling and slow-timescale partition management is essential to avoid persistent priority inversion caused by fragmentation. When high-priority workloads arrive, the fast scheduler increases their relative urgency and, where possible, admits the corresponding groups by using available residual capacity or by preempting lower-priority, preemptable work. Over longer horizons, the partition manager reshapes the layout to reduce the likelihood that high-priority groups remain blocked by unsuitable slice sizes under the hardware’s limited granularity.

5.6 Prototype Description

In the previous section, we introduced the proposed method and presented a high-level overview of the system design. In this section, we describe how that design is realized within a Kubernetes environment and detail the architectural components that enable GPU-aware scheduling and dynamic MIG allocation.

Figure 5.5 illustrates the complete architecture of the implemented system, which is integrated within the Kubernetes environment. The architecture is divided into two main components: the control plane and the worker nodes. The control plane includes the standard Kubernetes API server (kube-apiserver), the controller manager (kube-controller-manager), and etcd. To support our proposed method, we implemented a custom GPU scheduler (gpu-scheduler). The GPU scheduler selects nodes using an HTTP-based endpoint known as the GPU-scheduler API. In addition, we have added several lightweight components to extend Kubernetes and support fine-grained GPU management. These include a Custom Resource Definition (CRD) manager, known as Pod Watcher, which maintains the allocation state through a dedicated CRD called NodeGPUState. Additionally, we implemented a GPU partitioning manager that configures MIG or hardware partitions on worker nodes. An HTTPS-based mutating admission webhook is also implemented to inject GPU-specific runtime settings into pods. The worker nodes run the conventional kubelet and container runtime but are enhanced with the partitioning manager to ensure that the hardware state remains synchronized with CRD updates.

To clearly separate component responsibilities from runtime behaviour, we first outline each custom component and its function within the architecture. Next, we present the complete execution flow, reflecting the interactions shown in Figure 5.5.

5.6.1 Architectural Components

Our implemented system extends Kubernetes with several lightweight components that collectively enable GPU-aware scheduling and dynamic MIG allocation, while preserving compatibility with the native control plane. These components work alongside the standard Kubernetes modules, adding additional logic to interpret GPU requirements, perform node-level allocation, and inject GPU-specific runtime configurations. Below is a summary of the major components.

- **GPU-Scheduler:** The core component of our implementation is a custom scheduler called GPU-Scheduler. This scheduler utilizes an HTTP service as its GPU Scheduler Endpoint. During the scheduling cycle, the GPU-Scheduler invokes this endpoint to interpret the GPU requirements specified in the pod annotations. Users specify their requirements with two fields: 'priority-class,' which indicates the task's priority, and 'gpu.request', which determines the required GPU resources. Using this information, the GPU-Scheduler evaluates potential candidate nodes based on the GPU state

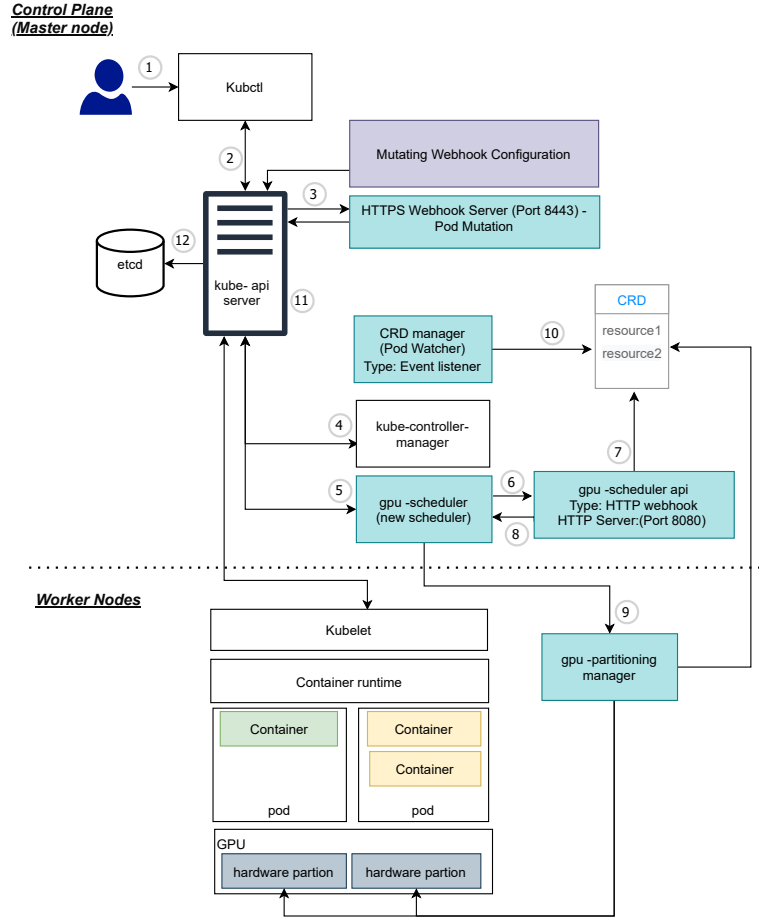


Figure 5.5: Prototype Architecture.

stored in the NodeGPUState CRD and various allocation algorithms. Based on this evaluation, the GPU-Scheduler Endpoint filters out non-viable nodes and ranks the remaining candidates. Then it informs the scheduler which nodes and the GPU slices have sufficient GPU resources available. If no suitable candidates are found, it notifies the GPU partition manager to create the requirements.

- **Mutating Admission Webhook:** The mutating admission webhook participates in Kubernetes' admission control pipeline and is triggered during both CREATE and UPDATE events for pods. When a user submits a request to the API server, during the Create event, the webhook inspects the new pod for GPU-related annotations (specifically, `gpu.request`) to determine whether a GPU allocation request has been made. Upon CREATE events, the webhook creates an environment field for the pod with a placeholder key such as `NVIDIA_VISIBLE_DEVICES` as the name and ConfigMap as the value. Using a placeholder (ConfigMap) for the environment value is necessary because certain fields in Kubernetes pods are immutable. Since the value for `NVIDIA_VISIBLE_DEVICES`, which identifies the specific GPU slices allocated

to the pod, is defined after the pod is created, using a placeholder allows us to update this value without altering immutable fields. After the pod is created and the scheduler determines which GPU slices to allocate, the Pod Watcher updates the placeholder with the actual ID of the allocated GPU slice (for example, via a ConfigMap volume mount). This approach avoids the need to rely on the Webhook UPDATE event, which cannot modify immutable fields.

- **CRD Manager (Pod Watcher):** The Pod Watcher is a controller that monitors pod update events and detects when a pod has been assigned to a node. After the scheduler selects an appropriate node and GPU slice (e.g, MIG instance), the Pod Watcher records the allocation in the NodeGPUState CRD. This process ensures that allocation decisions remain consistent and are visible to other components. Additionally, it updates a per-pod ConfigMap (created earlier by the webhook) by replacing the placeholder value with the allocated MIG instance UUID(s). This makes the final device binding visible to the container without altering the immutable pod specification.
- **GPU Partitioning Manager:** The GPU partitioning manager operates on worker nodes that are equipped with GPUs. Its primary responsibility is to create, resize, or remove MIG partitions based on allocation decisions noted in the CRD. The manager ensures that the physical GPU setup aligns with the logical state outlined in NodeGPUState. This component enables dynamic reconfiguration of GPU resources without restarting the kubelet or container runtime.
- **NodeGPUState CRD:** The NodeGPUState CRD encodes the GPU information, available MIG instances, and active allocations for each node. It serves as the shared state between the GPU-Scheduler Endpoint, Pod Watcher, and the GPU partitioning manager. Each instance provides detailed information, including the GPU profile, total and minimum slice capacity, MIG instance identifiers, and allocation bindings. By maintaining this state, the system minimizes its dependence on low-level device plugin interfaces while ensuring complete visibility for scheduling and resource management.

These components work together to divide responsibilities. The interactions among them follow the numbered steps illustrated in Figure 5.5. A detailed, step-by-step mapping of this execution flow is provided in Table 5.3. In the following subsection, we will describe the complete end-to-end workflow.

5.6.2 Execution Flow

Figure 5.5 depicts the runtime interaction between the components described in Section 5.6.1. Additionally, Table 5.3 provides a clear mapping of the numbered steps,

Table 5.3: End-to-end execution flow corresponding to the numbered steps in Figure 5.5.

Step	Component	Description
1	User / kubectl	User submits workload to the cluster (e.g., kubectl apply -f pod.yaml).
2	kube-apiserver	API server receives the request and begins processing the pod creation event.
3	Mutating Webhook (CREATE)	Webhook is invoked for the CREATE admission request and creates ConfigMap with a placeholder
4	Control Plane Controllers	The Controller Manager and scheduler observe the new Pending pod.
5	GPU-Scheduler	The GPU scheduler begins a scheduling cycle for the pod requiring GPU resources.
6-7	GPU-Scheduler Endpoint	Scheduler calls the GPU-Scheduler endpoint. The GPU-Scheduler endpoint reads the NodeGPUState CRD to identify feasible nodes.
8	GPU-Scheduler	The GPU-Scheduler API informs the scheduler of the appropriate node and GPU partition, enabling it to assign a node to the pod.
9	GPU Partition Manager	If no MIG partition fulfill the request, the GPU partition manager creates or reconfigures partitions, and the CRD state is updated accordingly.
10	Pod Watcher (CRD Manager)	Pod Watcher records the exact allocation (MIG instance, capacity) into the NodeGPUState. It also updates the per-pod ConfigMap by replacing the placeholder with the allocated MIG UUID.
11-12	Kubelet	Kubelet on the target node retrieves the pod spec and starts the container. The container receives NVIDIA_VISIBLE_DEVICES by resolving the updated ConfigMap value at startup.

identifying the corresponding component and its role in the workflow. In this subsection, we will walk through the execution flow, following these numbered steps from initial pod submission to container startup with an assigned MIG partition.

- **Request Submission and Initial Pod Creation:** In step 1, the user submits a task using kubectl (e.g., kubectl apply -f pod.yaml). This request is then sent to the Kubernetes API server in step 2. During the CREATE admission process, the webhook identifies the GPU request annotation and creates a per-pod ConfigMap. This ConfigMap contains a placeholder for device binding, with NVIDIA_VISIBLE_DEVICES as the name and the ConfigMap as the value (step 3). The API server then saves the new pod object in etcd with a status of Pending, where the GPU requirements are specified only through annotations.
- **GPU-Aware Scheduling and Partition Management:** The creation of the Pending pod triggers the control-plane controllers. In step 4, the controller manager and the custom GPU scheduler observe the new pod. Since the pod specifies the GPU scheduler as its schedulerName, the GPU scheduler takes responsibility for the allocation decision in step 5. In step 6, the GPU-scheduler calls the GPU-Scheduler HTTP API to identify a suitable node and GPU slice. The GPU-Scheduler Endpoint reads the NodeGPUState CRD and, in step 7, filters and prioritizes candidate nodes based on GPU profile, capacity, existing allocations, and the scheduling algorithms. In step 8, the GPU-Scheduler API informs the scheduler about the suitable node

and the corresponding GPU partition. Based on this information, the scheduler can assign a node to the pod. If the CRD indicates that no existing partition meets the request, step 9 involves invoking the GPU partitioning manager to create or adjust MIG partitions and update the CRD. After that, a subsequent scheduling cycle can successfully bind the pod by setting its nodeName in the API server.

- **Allocation Finalization via CRD and ConfigMap Update:** Once the scheduler assigns a node to the pod, the Pod Watcher (the CRD manager) monitors this scheduling decision. In step 10, it detects the transition of the pod from an unscheduled state to a scheduled state. The Pod Watcher then updates the NodeGPUState CRD with a specific allocation, which includes the selected MIG partitions and its allocated capacity. Additionally, during this phase, it modifies the previously created ConfigMap for the pod by replacing placeholders with the actual UUID of MIG partitions and sets corresponding allocation based on the pod’s name.
- **Container Startup:** Finally, in step 11, the kubelet on the selected node communicates with the Api server to retrieve the pod specification and starts the container. During container creation, Kubernetes resolves the NVIDIA_VISIBLE_DEVICES variable by reading the updated ConfigMap, ensuring the container starts with the correct MIG partition identifiers.

5.7 Performance Evaluation

This section evaluates the proposed priority-aware GPU management framework along two complementary axes. First, we report results from a Kubernetes-based prototype to validate that the end-to-end control loop preserves the framework’s core guarantees in a real orchestration workflow (priority ordering, tenant isolation, and correct GPU binding). Because the prototype is designed to demonstrate feasibility rather than large-scale throughput, these results are intentionally limited to the main goals and implementation overheads. Second, we use a trace-driven simulator to quantify scalability and scheduling behaviour under realistic, large workloads derived from the Alibaba GPU trace [96].

5.7.1 Kubernetes Prototype Evaluation

5.7.1.1 Testbed and methodology

We deploy the prototype described in Section 5.6.1 on a Kubernetes cluster equipped with NVIDIA A100-SXM4-40GB GPUs. The A100 features hardware-based GPU partitioning, allowing users to choose from a specific menu of partitions: A single device can be divided into a maximum of seven isolated partitions, enabling fractional allocations such as 1/7, 2/7, 3/7, 4/7, or the entire GPU. Each partition provides dedicated accelerator resources,

including both compute and memory, which enhances isolation compared to software-only sharing.

A key implication of this hardware interface is that partition sizes and layouts are not arbitrarily composable. There is only a limited catalogue of valid partition combinations for the A100. When a partition is selected, it restricts which additional partitions can coexist on the same GPU. As a result, the prototype scheduler and GPU partitioning manager must operate within these device-level constraints, selecting from the 19 supported layout combinations for the A100 when creating, resizing, or removing partitions. This limitation is central to our evaluation because it directly influences whether a pending task can be admitted immediately, whether preemption is necessary, and whether reconfiguring a partition is worthwhile. Furthermore, the A100 supports software-level sharing (MPS), allowing multiple tasks of a tenant to share the same hardware partition when appropriate.

We evaluate the prototype using controlled workload mixes with two priority classes (guaranteed and best-effort) and two workload types (continuous and time-limited). Each task is submitted as a Kubernetes pod annotated with its class, type, GPU resource requirement, and the name of the scheduler. The control loop then performs node selection (GPU-Scheduler Endpoint), allocation recording (NodeGPUState CRD via the Pod Watcher), partition enforcement (GPU partitioning manager), and runtime injection via the ConfigMap-based admission path. The goal of these experiments is to demonstrate that: (i) the configured class ordering is enforced at admission and placement decisions; (ii) tenant isolation is preserved; and (iii) the orchestration overhead remains modest relative to task runtimes.

5.7.1.2 Metrics

We concentrate on the following metrics, which align with the objectives of our proposed method.

Priority-order preservation. We measure two key metrics: (a) the *priority inversion rate*, which is defined as the fraction of decision periods during which a best-effort task is allowed to run or continues to run on capacity that could have been allocated to a pending guaranteed task (given the constraints of non-preemptability), and (b) the *guaranteed admission delay*, which refers to the time from the submission of a pod to the start of that pod for guaranteed tasks under conditions of contention.

Tenant isolation correctness. We validate isolation by checking that a pod can only observe the GPU partition assigned to its tenant allocation and cannot access partitions allocated to other namespaces/tenants. In the prototype, this is verified by comparing the injected partition identifiers (resolved through ConfigMap injection) with the *NodeGPUState* allocation record observed at runtime.

Table 5.4: Baseline prototype results (no contention).

Metric	Guaranteed Task 1 (100)	Guaranteed Task 2 (200)
Submission time	0 ms	500 ms
Scheduling time	180 ms	720 ms
Allocation found	200 ms	750 ms
ConfigMap updated	220 ms	770 ms
Pod running	350 ms	900 ms
Admission delay	350 ms	400 ms

Table 5.5: Contention scenario with preemption.

Pod	Class	Cap.	Preempts	MIG	Adm. delay
besteffort-1	best-effort	200	–	2g.10gb	300 ms
besteffort-2	best-effort	300	–	3g.20gb	800 ms
guaranteed-1	guaranteed	300	besteffort-2	3g.20gb	580 ms
guaranteed-2	guaranteed	200	besteffort-1	2g.10gb	620 ms

Orchestration overhead. We report the end-to-end admission delay (submission-to-running) and the latency of key control-loop steps visible in the prototype timeline: scheduling latency (including GPU-Scheduler API call), allocation materialization in *NodeGPUState*, ConfigMap creation/update, and (when applicable) preemption latency (decision to target removal).

5.7.1.3 Results and discussion

In this section we reort both *illustrative scenario runs* and *aggregate prototype metrics*, including priority preservation, admission delay, and isolation collected across a larger series of scheduling decisions (Tables 5.4–5.7).

(A) Illustrative scenarios: In the first scenario, we have a no-contention baseline with sequential guaranteed pods. The first guaranteed task arrives at time 0 with a requested GPU resource capacity of 100 (1g.10gb). The second guaranteed task arrives at time 500 and requires a GPU capacity of 200 (2g.20gb). It is important to note that, since we are using an A100 GPU with a max 7-instance capacity, we assume the GPU’s total capacity is 700 and interpret the request accordingly. The results show that both pods are admitted and reach the Running state, with admission delays of 350 ms and 400 ms, respectively (mean 375 ms), as shown in Table 5.4.

In the second scenario, which involves contention, best-effort pods occupy the available partitions. Newly arriving guaranteed pods trigger targeted preemption: each guaranteed pod preempts a best-effort target on the required partition (see Table 5.5). In these

Table 5.6: Mixed workload scenario with 10 tasks over 30 seconds.

#	Class	User	Capacity	Arrival time	Status	Adm. delay	Preempted
1	best-effort	user-bob	300	0s	Preempted	320 ms	By #7 at 18.4s
2	guaranteed	user-alice	100	3.2s	Running	450 ms	–
3	best-effort	user-charlie	100	5.7s	Preempted	350 ms	By #5 at 11.6s
4	guaranteed	user-alice	100	9.1s	Running	380 ms	–
5	guaranteed	user-alice	100	11.6s	Running	720 ms	Preempted #3
6	best-effort	user-bob	200	15.9s	Pending	–	–
7	guaranteed	user-alice	300	18.4s	Running	890 ms	Preempted #1
8	best-effort	user-charlie	100	21.6s	Pending	–	–
9	guaranteed	user-alice	100	25.7s	Running	420 ms	–
10	best-effort	user-bob	100	28.2s	Pending	–	–

Table 5.7: Priority-order preservation and preemption metrics.

Metric	Value	Upper bound
Priority inversion rate	0.8% (2/250)	< 5%
Preemption success rate	96% (24/25)	> 90%
Preemption latency (mean)	185ms	< 500ms
Guaranteed task starvation	0% (0/50)	0%

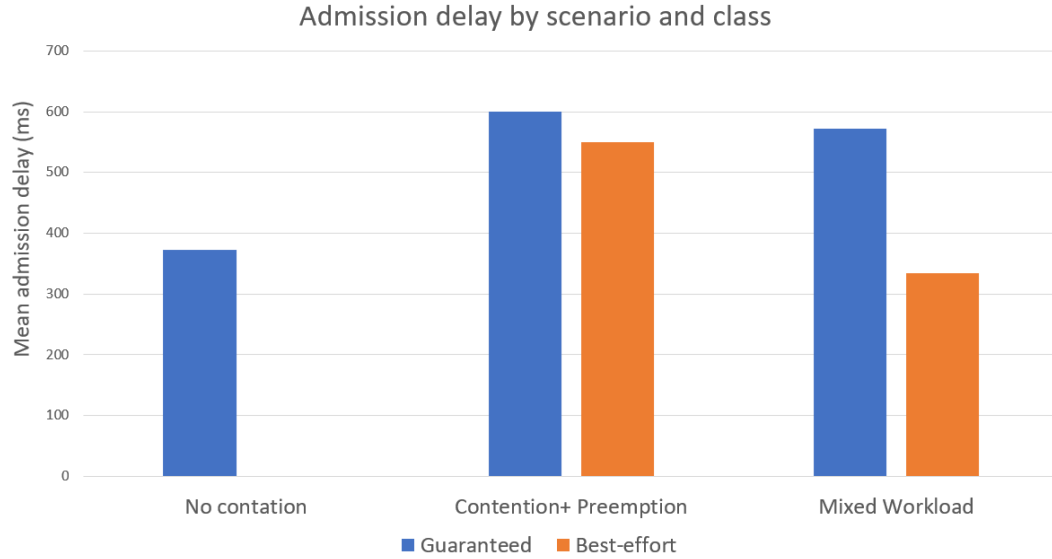


Figure 5.6: Mean admission delay by scenario and class.

controlled runs, no priority inversions are observed because preemption is allowed and is completed before the guaranteed pod is admitted. In the final scenario, we conduct an experiment with a mixed workload including 10 tasks with randomized arrival times. The results indicate that guaranteed tasks consistently receive priority. All guaranteed tasks are successfully admitted, resulting in a 100% success rate. However, some best-effort tasks remain pending or are preempted when there is sustained demand (see Table 5.6). This example demonstrates the intended behaviour: best-effort tasks do not delay the admission of guaranteed tasks.

(B) Aggregate priority preservation and preemption: In another scenario involving 250 scheduling decisions that included both guaranteed and best-effort workloads under contention, the scheduler successfully maintained the configured priority ordering. The rate of priority inversion was 0.8% (2 out of 250 epochs), which is below 5% (see Table 5.7). The two inversions occurred only during short transitional periods when a preemption had been initiated, but the target pod had not yet terminated and released its partition resources. Once the preemption was completed, the pending guaranteed pod was admitted. Preemption was successful in 96% of attempts (24 out of 25) and had an average latency of 185 ms. Throughout the evaluation period, no guaranteed tasks experienced starvation.

(C) Isolation and overhead: Isolation checks confirm that the partition identifiers observed inside containers match the allocations recorded in NodeGPUState and remain specific to each tenant. We report zero instances of cross-tenant identifier exposure across the prototype runs discussed in this section. Additionally, the prototype’s orchestration overhead remains modest. In a contention-free scenario, the allocation materialization and injection steps are brief relative to the overall admission delay, which is primarily determined by scheduling and pod startup times. When contention occurs, the additional overhead mainly arises from preemption; nevertheless, guaranteed admission still occurs within sub-second limits. Figures 5.6 summarize these results.

5.7.2 Trace-Driven Scalable Evaluation with Alibaba Workloads

5.7.2.1 Simulator and workload preparation

To assess scalability beyond what can be practically achieved in a small orchestration deployment, we have developed a simulator that incorporates the proposed control loop. This control loop includes priority-aware scoring, per-tenant grouping, checks for partition feasibility under discrete partition configurations, preemption of explicitly preemptable tasks, and periodic reconfiguration of partitions. The simulator utilizes the same decision-making logic as outlined in the proposed method section, enabling us to explore a larger number of tasks and GPUs.

We use Alibaba’s cluster-trace-gpu dataset as the source of the task arrivals, requested GPU capacity, and observed runtimes. The trace spans approximately 145 days of activity and includes over 7000 GPU-dependent tasks, providing realistic submission patterns and highly varied GPU demand. Since the dataset does not explicitly label the attributes required by our model, such as priority class, intra-class priority, workload type, and preemptibility, we enhance each task in the trace with additional labels through a controlled mapping process. Specifically, we categorize each task into a priority class: guaranteed (highest priority) or best effort. Additionally, we assign an intra-class priority score ranging from $\{1, \dots, 10\}$ to reflect the relative urgency of tasks within the same class. Moreover, we classify tasks as preemptable or non-preemptable based on their assigned class and the

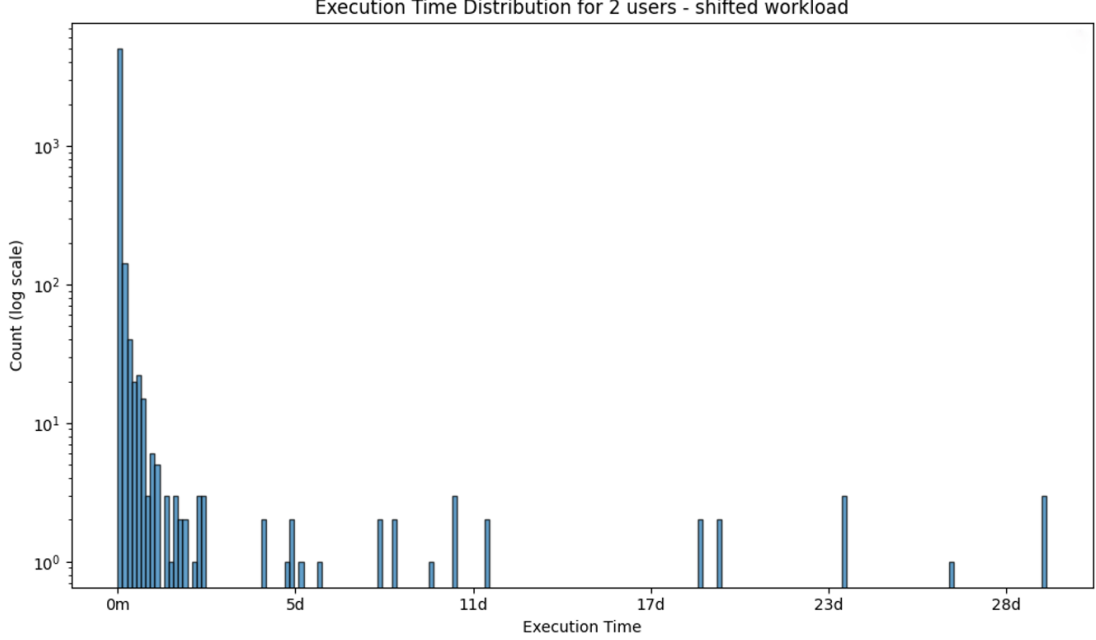


Figure 5.7: Distributions of the execution time for two users

specific scenario configuration.

To simulate a multi-tenant environment, we partition the trace stream across different tenants (e.g., two users). We then adjust each tenant’s timeline to start at zero and concurrently replay the resulting submission streams. This approach creates overlapping GPU demand over an extended duration (e.g., more than 14 days) while maintaining the original arrival patterns. In addition, it results in a heavy-tailed distribution, which places significant stress on scheduling systems. Figure 5.7 illustrates the distribution of the workload when we divide the trace stream between two tenants. To model mixed workload types, we establish a duration threshold derived from the empirical distribution of the trace. Specifically, tasks with a duration exceeding 1800 seconds (30 minutes) are classified as long-running (continuous), while those with a duration of 1800 seconds or less are considered time-limited. This classification scheme is informed by the trace distribution shown in Figure 5.7, where approximately 65% to 70% of tasks are completed within 30 minutes. In contrast, a smaller percentage of tasks run for hours, days, and, in some cases, even weeks. Finally, we use multiple class mixes by adjusting the proportions of guaranteed and best-effort tasks, including 70/30, 50/50, 10/90, and 5/95 distributions.

5.7.2.2 Metrics

In the experimental results, we present metrics that directly address the challenges outlined in the abstract and introduction. The following are the metrics we examine:

Fairness Metric: For each task, we calculate the average and maximum normalized

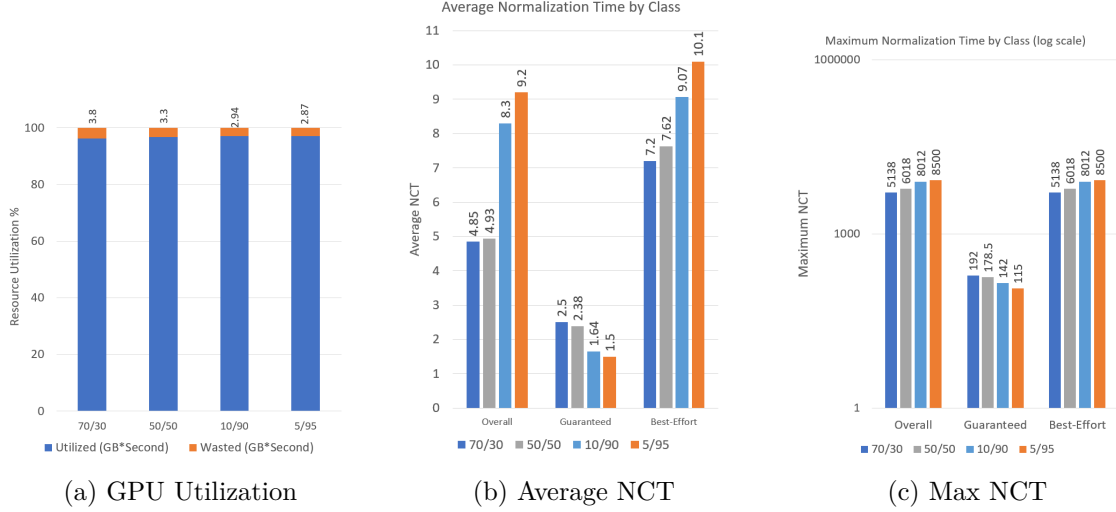


Figure 5.8: Scalability experiment results with 8 GPUs and mixed workload: (a) GPU utilization, (b) average NCT, (c) maximum NCT.

completion times as fairness metrics.

Utilization and unused capacity: In the experimental results, we present the GPU utilization over time. To connect utilization with the discrete partition structure and fragmentation, we report the amount of unused capacity on active GPUs. This information illustrates how well the allocations can satisfy pending demand.

5.7.2.3 Scalability experiments

We evaluate two scalability dimensions:

Scaling the number of GPUs. We vary the cluster size from a small number of GPUs to a larger configuration of up to 16 GPUs while replaying the same arrival process. This experiment aims to examine whether the algorithm continues to preserve class ordering and fairness while effectively utilizing the additional capacity.

Scaling contention and class mix. We adjust the proportions of guaranteed and best-effort tasks. This experiment assesses whether the two-layer policy guarantees reliable service for assigned tasks, avoiding significant slowdowns in either task class across various proportions of the class types.

5.7.2.4 Results

In this section we provide the simulator results to quantify scalability and scheduling behaviour under realistic workload.

(A) Class Mix Scaling Examination: In this scenario, we utilize 8 GPUs and adjust the proportions of best-effort and guaranteed workloads. The purpose of these experiments is to evaluate whether the framework can maintain priority ordering and protect guaranteed

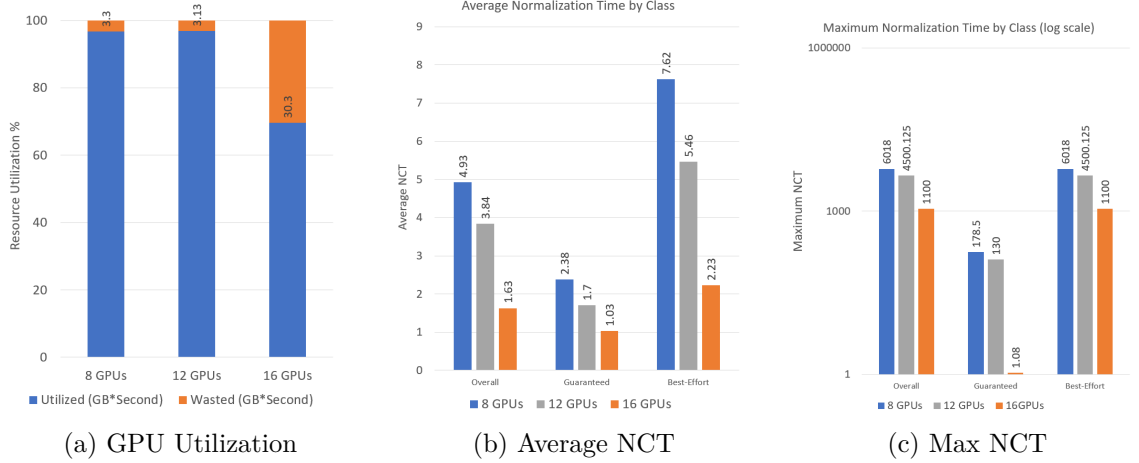


Figure 5.9: Scalability experiment results with GPU scaling 50/50 mix workload: (a) GPU utilization, (b) average NCT, (c) maximum NCT.

workloads while keeping the GPUs busy under varying levels of best-effort pressure. To explore both best-effort-dominant and guaranteed-dominant scenarios, we compare four mixes: 70/30, 50/50, 10/90, and 5/95 (representing Guaranteed percentage/Best-Effort percentage). The results demonstrate that GPU underutilization remains low, with utilization consistently high (around ~ 96 – 97%), as illustrated in Figure 5.8a. This indicates that the scheduler effectively keeps GPUs occupied by admitting lower-priority tasks when appropriate, while still adhering to class priority, even during significant shifts in workload domination. Figure 5.8b presents results related to the average normalized completion time (NCT). Across all workload mixes, the guaranteed class remains protected, though its average NCT increases as the guaranteed fraction becomes dominant, due to higher intra-class contention. When best-effort workloads dominate, both the average and maximum NCTs (Figure 5.8c) for best-effort tasks increase. This indicates that best-effort tasks are experiencing longer wait times. As the system becomes increasingly dominated by best-effort traffic (in the 10/90 and 5/95 scenarios), additional delays primarily impact the best-effort tasks, leading to higher average and maximum NCT values. In contrast, guaranteed tasks continue to be served relatively well. This outcome aligns with the policy’s intended design, which enforces strict priority between workload classes to prevent best-effort tasks from overtaking guaranteed tasks, while also ensuring fairness among tasks within the same class based on NCT to avoid starvation.

(B) GPU Scaling Examination: This experiment evaluates whether the framework scales effectively with increasing GPU capacity and whether it continues to achieve its objectives. In this experiment, we utilize a workload comprised of a 50% mix. Moving from 8 to 16 GPUs reduces contention, as shown in Figure 5.9. This figure illustrates improvements in both maximum NCT and average NCT for both classes, indicating that the scheduler is not merely protecting guaranteed tasks, but also improving within-class

fairness and completion-time efficiency when resources allow. However, the underutilization plot reveals an important trade-off in the system: when capacity exceeds what the workload can continuously utilize (for example, at 16 GPUs), utilization drops significantly. This indicates that the system becomes demand-limited even as the scheduler continues to respect priority and reduce NCT. Overall, these results support our claims regarding the proposed method: the framework maintains class-based priority behaviour, improves fairness-driven completion times (NCT) within each class, and achieves high utilization when demand is sufficient, while ensuring that best-effort workloads do not degrade guaranteed service.

(C) Examining GPU Scaling Using Different Approaches: In this experiment, we used a 50% workload mix, increased the number of GPUs, and compared the Max-NCT of guaranteed tasks across different approaches. As the cluster capacity grew from 8 to 16 GPUs, contention decreased, leading to improvements in Max-NCT for all methods. However, the rate of improvement varied based on whether the baseline could (a) protect guaranteed tasks, (b) avoid starvation and long wait times, and (c) adapt partitioning decisions to minimize fragmentation. We specifically compared our approach with MISO [12], GFS [90], and dynamic fairness scheduling (DFS) [2]. Figure 5.10 shows the Max-NCT of various approaches as GPU resources increase. In MISO, static partitioning and FIFO scheduling do not support preemption or reconfiguration, meaning the scheduler must wait for partitions to drain, which increases Max-NCT. Additionally, relying solely on hardware partitioning can lead to significant fragmentation, which increases task wait times for finding available resources. In MISO, Max-NCT is high, particularly with 8 GPUs, because FIFO scheduling lacks class protection. Early-arriving long best-effort tasks can occupy large slices and block subsequent guaranteed tasks. This leads to guaranteed tasks remaining pending, even though some capacity is allocated to lower-priority work. While Max-NCT decreases as the number of GPUs increases (due to more available space reducing blocking), it rarely approaches the results achieved by our approach, which enforces explicit priority.

In GFS, strict priority reduces the Max-NCT for guaranteed tasks, as these tasks are prioritized above all others. However, this approach lacks intra-class fairness, which can worsen fairness across tasks within the same category. Furthermore, while strict priority generally reduces Max-NCT for guaranteed tasks, it does not group tasks by their users and requirements. This lack of task grouping results in inefficient resource utilization, leading to significant fragmentation and idle GPU capacity. This fragmentation restricts the scheduling of additional tasks, ultimately increasing the Max-NCT for all tasks, particularly for guaranteed ones. Dynamic and fair scheduling in DFS tends to reduce the overall Max-NCT compared to other approaches, as it actively balances slowdown among tasks. However, it does not consistently protect guaranteed workloads.

Under heavy contention, the Max-NCT may still be affected by situations where guaranteed tasks are delayed behind best-effort tasks that appear fair on a global scale. As a result, DFS generally exhibits a non-trivial Max-NCT for guaranteed tasks, even with

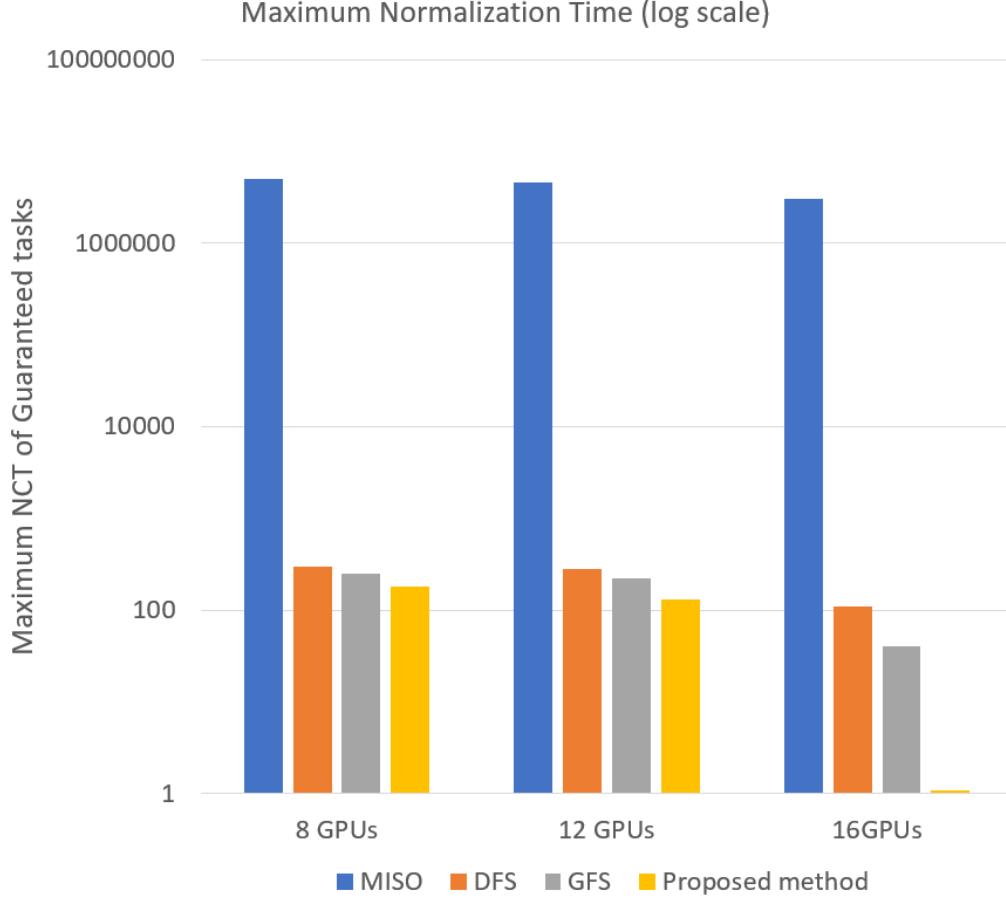


Figure 5.10: Max-NCT during Task execution for different numbers of GPUs in the cluster

higher GPU counts. This is because the policy intentionally balances slowdown across all tasks, allowing or continuing best-effort tasks even when guaranteed tasks are pending. In contrast, our priority-aware, fairness-based approach significantly reduces Max-NCT, even with limited GPUs available. By ensuring that best-effort tasks do not delay guaranteed tasks, we maintain strict priority. Additionally, ensuring fairness within each task class prevents problematic FIFO sequences and helps improve the Max-NCT for guaranteed tasks. As the number of GPUs increases towards 16, the Max-NCT decreases because the system encounters less contention, requiring fewer preemption and reconfiguration decisions.

The results of the proposed method demonstrate significant performance improvements over baseline approaches, including MISO, GFS, and DFS, across all GPU configurations. In the 8-GPU setup, the proposed method reduces the MAX-NCT metric by 28.6% compared to GFS and by 40.5% compared to DFS. With 12 GPUs, the improvements increase to 40.9% over GFS and 54.4% over DFS. In the 16-GPU configuration, the proposed method achieves reductions of 97.3% compared to GFS and 99.0% compared to DFS. In all scenarios, the proposed method outperforms MISO by more than 99.99%, highlighting its superior scalability and efficiency as the number of GPUs increases.

5.8 Conclusions

This chapter presents a priority-aware GPU sharing framework for multi-tenant cloud clusters, designed to address two practical requirements. First, it enforces tenant isolation while maintaining a configured priority among different classes. Second, it achieves fairness within each class under varying workload types while ensuring high utilization. Our approach combines three key components: (i) a priority-aware scoring mechanism that globally ranks tasks across tenants, (ii) a per-tenant grouping and partition feasibility step that maps tasks onto a limited set of hardware slice profiles, and (iii) a scheduler that admits tasks and applies targeted preemption. We implemented this framework in two complementary forms. First, we created a Kubernetes-integrated prototype that includes the complete control loop, along with node filtering and ranking via a GPU Scheduler. Experiments with the prototype confirm that the system preserves the configured priority ordering with low orchestration overhead. Second, we evaluated the scheduling policy at scale using the Alibaba GPU trace and the simulator. The results demonstrate that our proposed design not only maintains priority but also improves fairness. Overall, our experimental findings indicate that this method achieves approximately 97% GPU utilization. Additionally, it consistently outperforms all baseline methods, including MISO, GFS, and DFS. On average, our proposed approach improves normalized task completion times by about 76%, with improvements ranging from 28.6% to over 99.9% across various GPU configurations and baselines.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Efficient GPU resource allocation in cloud environments is essential for supporting modern compute-intensive workloads such as deep learning training, real-time inference, and microservice-based applications. However, managing GPUs in cloud systems remains challenging due to limited GPU availability, heterogeneous workload demands, and dynamic multi-tenant execution. Key issues include GPU underutilization, resource fragmentation, performance interference between co-located tenants, and the difficulty of ensuring fairness and predictable QoS under fluctuating workload arrivals. These challenges are further intensified in microservice-based applications, where pipeline dependencies, bottleneck components, and frequent CPU–GPU data transfers can significantly degrade end-to-end throughput and latency.

This thesis addresses these challenges by proposing efficient and adaptive GPU resource management frameworks for cloud environments, targeting improved utilization, fairness, and QoS guarantees under heterogeneous and dynamic workloads. The main contributions are summarized as follows:

In Chapter 3, we introduced an efficient GPU resource allocation technique integrated with an optimized task scheduling framework to address GPU underutilization and suboptimal task completion times in multi-tenant cloud environments. Our approach combines software- and hardware-based partitioning with an optimization-based scheduling method to improve utilization while reducing task delays. Experimental evaluation demonstrates that the proposed approach outperforms existing scheduling baselines and provides more efficient resource usage in dynamic multi-tenant settings.

In Chapter 4, we addressed the challenges of executing concurrent components of microservice-based applications on GPUs using spatial multitasking. We proposed a low-overhead dynamic GPU resource management technique based on an online performance optimizer to allocate GPU compute resources efficiently. In addition, we introduced a shared-memory-based communication mechanism to reduce data transfer overhead

between microservice components. The evaluation results confirm that the proposed approach significantly improves throughput and efficiency compared to state-of-the-art GPU multitasking methods.

In Chapter 5, we tackled the problem of priority-based GPU scheduling in multi-tenant cloud clusters hosting mixed workload classes. We proposed a priority-aware GPU sharing framework that enforces isolation and class-level priorities while maintaining fairness under heterogeneous workload types. The framework integrates priority-aware scoring, feasibility-based partition mapping, and targeted preemption to achieve predictable performance for critical workloads. Experimental results demonstrate that the proposed method maintains high GPU utilization while improving fairness and task completion performance compared to existing priority-based scheduling approaches.

6.2 Future Work

This thesis presents significant advancements in GPU resource allocation and scheduling for multi-tenant cloud environments. However, several important research directions remain unexplored. Future work should aim to improve energy efficiency, predictive intelligence, and system resilience, while ensuring high utilization, fairness, and QoS guarantees in cloud infrastructures.

6.2.1 Energy- and Cost-Aware GPU Resource Management

The techniques proposed in this thesis primarily optimize GPU utilization, fairness, and performance. However, GPUs are among the most power-hungry and expensive resources in modern data centers. As cloud providers increasingly seek sustainable and cost-efficient operations, energy-aware GPU scheduling becomes a critical research direction.

Future research should investigate allocation frameworks that jointly optimize performance and energy consumption. In particular:

- **Cost-Effective Resource Allocation:** Design frameworks that balance performance, energy consumption, and operational costs, enabling cloud providers to optimize resource usage while minimizing expenses.
- **Cost-Performance Trade-off Modeling:** Incorporate operational cost models into scheduling decisions, balancing SLA requirements with energy expenditure.

By integrating energy-awareness into GPU management, cloud systems can achieve improved sustainability without compromising QoS guarantees.

6.2.2 Predictive and Learning-Based GPU Scheduling

The resource allocation mechanisms proposed in this thesis rely on runtime monitoring and optimization-based decision-making. While effective, they primarily react to observed

workload behavior. A promising research direction is the development of predictive and learning-driven scheduling frameworks that anticipate workload dynamics before contention occurs.

Future work may explore:

- **Workload Demand Forecasting:** Leveraging historical workload patterns to predict future GPU demand and proactively adjust resource allocation.
- **Adaptive Priority Weighting:** Dynamically refining priority and fairness policies based on system behavior and tenant workload characteristics.
- **Learning-Based Preemption Strategies:** Applying reinforcement or online learning techniques to optimize preemption decisions under dynamic conditions.

Incorporating predictive intelligence into GPU scheduling can reduce reactive reallocations, improve stability under bursty workloads, and enhance overall system efficiency.

6.2.3 Resilient and Fault-Tolerant GPU Scheduling

Although this thesis focuses on efficient and fair GPU allocation under dynamic workloads, large-scale cloud systems must also address hardware failures, driver instability, and unexpected resource disruptions. GPUs and their associated software stacks may experience resets, contention anomalies, or node-level failures, which can impact SLA-critical workloads.

Future research should explore mechanisms for enhancing resilience in GPU resource management. Specifically:

- **Failure-Aware Scheduling:** Designing allocation frameworks that detect GPU faults and dynamically migrate or reassign affected workloads.
- **Checkpoint-Aware Preemption:** Integrating lightweight checkpointing strategies to minimize recomputation costs when tasks are preempted or interrupted.
- **SLA-Preserving Recovery Policies:** Ensuring that high-priority workloads maintain predictable performance even during hardware disruptions.

By incorporating fault tolerance and recovery mechanisms into GPU scheduling frameworks, cloud infrastructures can achieve greater robustness and service continuity under real-world operational conditions.

6.2.4 Multi-GPU Execution for Microservice-Based Applications

The microservice-based resource management techniques proposed in this thesis primarily focus on optimizing performance within a single GPU using spatial multitasking. However, a key advantage of microservice architectures is their natural ability to scale across multiple servers and accelerators. In large-scale deployments, executing microservice pipelines across multiple GPUs can significantly improve throughput and reduce end-to-end latency, particularly when individual pipeline stages become compute-intensive bottlenecks.

Future research should explore GPU resource management frameworks that extend dynamic allocation and optimization across multiple GPUs. Specifically:

- **Pipeline-Aware Multi-GPU Scheduling:** Design scheduling strategies that distribute microservice components across GPUs while considering dependency constraints and end-to-end pipeline performance.
- **Cross-GPU Data Transfer Optimization:** Develop mechanisms to reduce communication overhead between pipeline stages deployed on different GPUs, such as GPU-to-GPU communication optimization and placement-aware scheduling.
- **Distributed Online Performance Optimization:** Extend online optimizers to operate across multiple GPUs, enabling dynamic allocation decisions that adapt to workload fluctuations at cluster scale.

By supporting multi-GPU execution for microservice-based applications, cloud platforms can further improve scalability and performance while preserving the adaptability and efficiency achieved by single-GPU optimization techniques.

Bibliography

- [1] H. Sedighi, F. Wuhib, and R. H. Glitho, “Dynamic task scheduling and adaptive gpu resource allocation in the cloud,” *IEEE Transactions on Network and Service Management*, vol. 23, pp. 1112–1127, 2026.
- [2] H. Sedighi, F. Wuhib, and R. Glitho, “Efficient task scheduling and allocation of gpu resources in clouds,” in *2024 15th International Conference on Network of the Future (NoF)*, pp. 115–123, 2024.
- [3] H. Sedighi, F. Wuhib, and R. Glitho, “A method for sharing gpu resources in multi-tenant cloud.” WO2025219744A1, Patent published by Telefonaktiebolaget LM Ericsson (Publ), 2025. Priority 2024-04-16, filed: 2024-04-16, published: 2025-10-23.
- [4] H. Sedighi, D. Gehberger, A. Ebrahimzadeh, F. Wuhib, and R. H. Glitho, “Efficient dynamic resource management for spatial multitasking gpus,” *IEEE Transactions on Cloud Computing*, vol. 13, no. 1, pp. 99–117, 2025.
- [5] H. Sedighi, D. Gehberger, and R. Glitho, “Workload-aware dynamic GPU resource management in component-based applications,” in *2022 IEEE International Conference on Cloud Engineering (IC2E)*, pp. 213–220, IEEE Computer Society, 2022.
- [6] H. Sedighi, R. Glitho, and D. Gehberger, “Dynamic allocation of accelerator processors between related applications.” WO2023199099A1, Patent published by Telefonaktiebolaget LM Ericsson (Publ), 2023. Priority date: 2022-04-14, filed: 2022-04-14, published: 2023-10-19.
- [7] H. Sedighi, R. Glitho, D. Gehberger, H. Gustafsson, and A. A.-E. Hassan, “Transfer of memory claim for data between applications.” WO2023199096A1, Patent published by Telefonaktiebolaget LM Ericsson (Publ), 2023. Priority date: 2022-04-14, filed: 2022-04-14, published: 2023-10-19.
- [8] H. Sedighi, F. Wuhib, and R. H. Glitho, “Priority-aware scheduling and isolation-preserving gpu partitioning in the cloud,” *IEEE Transactions on Network and Service Management*, 2026.

- [9] A. Habermaier and A. Knapp, “On the correctness of the simt execution model of gpus,” in *European Symposium on Programming*, pp. 316–335, Springer, 2012.
- [10] Q. Weng, W. Xiao, Y. Yu, W. Wang, C. Wang, J. He, Y. Li, L. Zhang, W. Lin, and Y. Ding, “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)*, pp. 945–960, 2022.
- [11] H. Zhao, Z. Han, Z. Yang, Q. Zhang, F. Yang, L. Zhou, M. Yang, F. C. Lau, Y. Wang, Y. Xiong, *et al.*, “HiveD: Sharing a GPU cluster for deep learning with guarantees,” in *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, pp. 515–532, 2020.
- [12] B. Li, T. Patel, S. Samsi, V. Gadepally, and D. Tiwari, “Miso: exploiting multi-instance GPU capability on multi-tenant GPU clusters,” in *Proceedings of the 13th Symposium on Cloud Computing*, pp. 173–189, 2022.
- [13] J. Yao, S. Zhang, Y. Yao, F. Wang, J. Ma, J. Zhang, Y. Chu, L. Ji, K. Jia, T. Shen, A. Wu, F. Zhang, Z. Tan, K. Kuang, C. Wu, F. Wu, J. Zhou, and H. Yang, “Edge-cloud polarization and collaboration: A comprehensive survey for ai,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 7, pp. 6866–6886, 2023.
- [14] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, *et al.*, “Above the clouds: A berkeley view of cloud computing,” tech. rep., Technical Report UCB/EECS-2009-28, EECS Department, University of California . . . , 2009.
- [15] X. Chen, Y. Wang, Y. Liang, Y. Xie, and H. Yang, “Run-time technique for simultaneous aging and power optimization in gpgpus,” in *Proceedings of the 51st Annual Design Automation Conference*, pp. 1–6, 2014.
- [16] “CUDA C++ Programming Guide .” http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, 2023. [Online].
- [17] C.-H. Hong, I. Spence, and D. S. Nikolopoulos, “Gpu virtualization and scheduling methods: A comprehensive survey,” *ACM Comput. Surv.*, vol. 50, June 2017.
- [18] “NVIDIA. Multi-process service.” https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf, 2024. [Online; accessed February-2024].
- [19] “Nvidia multi-instance GPU user guide..” <https://docs.nvidia.com/datacenter/tesla/mig-user-guide/>, 2023. [Online].
- [20] “device-plugins.” <https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/device-plugins/>, 2024.

- [21] C. Zhao, W. Gao, F. Nie, and H. Zhou, “A survey of GPU multitasking methods supported by hardware architecture,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 6, pp. 1451–1463, 2022.
- [22] C. Li, A. Zigerelli, J. Yang, Y. Zhang, S. Ma, and Y. Guo, “A dynamic and proactive GPU preemption mechanism using checkpointing,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 1, pp. 75–87, 2020.
- [23] C. Hartmann and U. Margull, “GPUart - an application-based limited preemptive GPU real-time scheduler for embedded systems,” *Journal of Systems Architecture*, vol. 97, pp. 304–319, 2019.
- [24] G. Chen, Y. Zhao, X. Shen, and H. Zhou, “EffiSha: A software framework for enabling efficient preemptive scheduling of GPU,” in *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, p. 3–16, 2017.
- [25] H. Zhou, G. Tong, and C. Liu, “GPES: a preemptive execution system for GPGPU computing,” in *Proc. IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 87–97, 2015.
- [26] J. T. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte, “The case for GPGPU spatial multitasking,” in *Proc. IEEE International Symposium on High-Performance Comp Architecture*, pp. 1–12, 2012.
- [27] C. Yu, Y. Bai, H. Yang, K. Cheng, Y. Gu, Z. Luan, and D. Qian, “SMGuard: A flexible and fine-grained resource management framework for GPUs,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 12, pp. 2849–2862, 2018.
- [28] Q. Chen, H. Yang, J. Mars, and L. Tang, “Baymax: QoS awareness and increased utilization for non-preemptive accelerators in warehouse scale computers,” *SIGPLAN Not.*, vol. 51, no. 4, p. 681–696, 2016.
- [29] P. Aguilera, K. Morrow, and N. S. Kim, “Fair share: Allocation of GPU resources for both performance and fairness,” in *Proc. IEEE 32nd International Conference on Computer Design (ICCD)*, pp. 440–447, 2014.
- [30] P. Di Francesco, P. Lago, and I. Malavolta, “Migrating towards microservice architectures: An industrial survey,” in *2018 IEEE International Conference on Software Architecture (ICSA)*, pp. 29–2909, 2018.
- [31] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding, “Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study,” *IEEE Transactions on Software Engineering*, vol. 47, no. 2, pp. 243–260, 2018.

- [32] I. Oumoussa and R. Saidi, “Evolution of microservices identification in monolith decomposition: A systematic review,” *IEEE Access*, vol. 12, pp. 23389–23405, 2024.
- [33] K. Gupta, J. A. Stuart, and J. D. Owens, “A study of persistent threads style GPU programming for GPGPU workloads,” in *Proc. IEEE Innovative Parallel Computing (InPar)*, pp. 1–14, 2012.
- [34] B. Wu, X. Liu, X. Zhou, and C. Jiang, “FLEP: Enabling flexible and efficient preemption on GPUs,” *SIGPLAN Not.*, vol. 52, no. 4, p. 483–496, 2017.
- [35] C. Basaran and K.-D. Kang, “Supporting preemptive task executions and memory copies in GPGPUs,” in *Proc. IEEE Euromicro Conference on Real-Time Systems*, pp. 287–296, 2012.
- [36] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero, “Enabling preemptive multiprogramming on GPUs,” *SIGARCH Comput. Archit. News*, vol. 42, no. 3, p. 193–204, 2014.
- [37] P. Aguilera, K. Morrow, and N. S. Kim, “QoS-aware dynamic resource allocation for spatial-multitasking GPUs,” in *Proc. IEEE Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 726–731, 2014.
- [38] W. Zhang, Q. Chen, N. Zheng, W. Cui, K. Fu, and M. Guo, “Toward QoS-Awareness and improved utilization of spatial multitasking GPUs,” *IEEE Transactions on Computers*, vol. 71, no. 4, pp. 866–879, 2022.
- [39] W. Zhang, K. Fu, N. Zheng, Q. Chen, C. Li, W. Zheng, and M. Guo, “CHARM: Collaborative host and accelerator resource management for GPU datacenters,” in *Proc. IEEE International Conference on Computer Design (ICCD)*, pp. 307–315, 2021.
- [40] F. Xu, J. Xu, J. Chen, L. Chen, R. Shang, Z. Zhou, and F. Liu, “igniter: Interference-aware gpu resource provisioning for predictable dnn inference in the cloud,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 3, pp. 812–827, 2023.
- [41] W. Gao, Z. Ouyang, P. Sun, and T. Zhang, “Icefrog: A layer-elastic scheduling system for deep learning training in gpu clusters,” *IEEE Transactions on Parallel and Distributed Systems*, pp. 1–16, 2025.
- [42] B. Hu, X. Yang, and M. Zhao, “Energy-minimized scheduling of intermittent real-time tasks in a cpu-gpu cloud computing platform,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, pp. 2391–2402, 2023.
- [43] B. Wu, Z. Zhang, Z. Bai, X. Liu, and X. Jin, “Transparent GPU sharing in container clouds for deep learning workloads,” in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pp. 69–85, 2023.

- [44] C. Tan, Z. Li, J. Zhang, Y. Cao, S. Qi, Z. Liu, Y. Zhu, and C. Guo, “Serving dnn models with multi-instance gpus: A case of the reconfigurable machine scheduling problem,” *arXiv preprint arXiv:2109.11067*, 2021.
- [45] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, “Dominant resource fairness: Fair allocation of multiple resource types,” in *8th USENIX symposium on networked systems design and implementation (NSDI 11)*, 2011.
- [46] Z. Ye, P. Sun, W. Gao, T. Zhang, X. Wang, S. Yan, and Y. Luo, “Astraea: A fair deep learning scheduler for multi-tenant GPU clusters,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 11, pp. 2781–2793, 2021.
- [47] J. Gu, M. Chowdhury, K. G. Shin, Y. Zhu, M. Jeon, J. Qian, H. Liu, and C. Guo, “Tiresias: A GPU cluster manager for distributed deep learning,” in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pp. 485–500, 2019.
- [48] K. Mahajan, A. Balasubramanian, A. Singhvi, S. Venkataraman, A. Akella, A. Phanishayee, and S. Chawla, “Themis: Fair and efficient GPU cluster scheduling,” in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pp. 289–304, 2020.
- [49] Q. Weng, L. Yang, Y. Yu, W. Wang, X. Tang, G. Yang, and L. Zhang, “Beware of fragmentation: Scheduling {GPU-Sharing} workloads with fragmentation gradient descent,” in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pp. 995–1008, 2023.
- [50] S. Garg, K. Kothapalli, and S. Purini, “Share-a-GPU: Providing simple and effective time-sharing on GPUs,” in *Proc. IEEE 25th International Conference on High Performance Computing (HiPC)*, pp. 294–303, 2018.
- [51] T. Allen, X. Feng, and R. Ge, “Slate: Enabling workload-aware efficient multiprocessing for modern GPGPUs,” in *Proc. IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 252–261, 2019.
- [52] S. K. Saha, Y. Xiang, and H. Kim, “STGM: Spatio-temporal GPU management for real-time tasks,” in *Proc. IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pp. 1–6, 2019.
- [53] W. Thomas, S. Toraskar, and V. Singh, “Dynamic optimizations in GPU using roofline model,” in *Proc. IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–5, 2021.

- [54] S. R. Punyala, T. Marinakis, A. Komaee, and I. Anagnostopoulos, “Throughput optimization and resource allocation on GPUs under multi-application execution,” in *Proc. IEEE Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 73–78, 2018.
- [55] Q. Chen, H. Yang, M. Guo, R. Kannan, J. Mars, and L. Tang, “Prophet: Precise QoS prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers,” *ACM SIGPLAN Notices*, vol. 52, pp. 17–32, 2017.
- [56] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, “{FIRM}: An intelligent fine-grained resource management framework for {SLO-Oriented} microservices,” in *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, pp. 805–825, 2020.
- [57] A. Ilyushkin, A. Ali-Eldin, N. Herbst, A. V. Papadopoulos, B. Ghit, D. Epema, and A. Iosup, “An experimental performance evaluation of autoscaling policies for complex workflows,” in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, pp. 75–86, 2017.
- [58] Y. Ahn, J. Choi, S. Jeong, and Y. Kim, “Auto-scaling method in hybrid cloud for scientific applications,” in *The 16th Asia-Pacific Network Operations and Management Symposium*, pp. 1–4, IEEE, 2014.
- [59] S. Kato, J. Aumiller, and S. Brandt, “Zero-copy I/O processing for low-latency GPU computing,” in *Proc. ACM/IEEE International Conference on Cyber-Physical Systems*, p. 170–178, 2013.
- [60] Y. Fujii, T. Azumi, N. Nishio, S. Kato, and M. Edahiro, “Data transfer matters for GPU computing,” in *Proc. IEEE International Conference on Parallel and Distributed Systems*, pp. 275–282, 2013.
- [61] S. Rennich, “CUDA C/C++ Streams and Concurrency.” <http://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf>, 2011. [Online].
- [62] W.-K. Lee, R. C.-W. Phan, B.-M. Goi, L. Chen, X. Zhang, and N. N. Xiong, “Parallel and high speed hashing in GPU for telemedicine applications,” *IEEE Access*, vol. 6, pp. 37991–38002, 2018.
- [63] S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, and R. Rajkumar, “RGEM: A responsive GPGPU execution model for runtime engines,” in *Proc. IEEE Real-Time Systems Symposium*, pp. 57–66, 2011.

- [64] B. v. Werkhoven, J. Maassen, F. Seinstra, and H. Bal, “Performance models for CPU-GPU data transfers,” in *Proc. IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pp. 11–20, 2014.
- [65] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero, “Enabling preemptive multiprogramming on gpus,” *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 193–204, 2014.
- [66] Y. Liang, H. P. Huynh, K. Rupnow, R. S. M. Goh, and D. Chen, “Efficient GPU spatial-temporal multitasking,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 3, pp. 748–760, 2014.
- [67] P. Aguilera, K. Morrow, and N. S. Kim, “Qos-aware dynamic resource allocation for spatial-multitasking gpus,” in *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 726–731, IEEE, 2014.
- [68] J. Bakita and J. H. Anderson, “Hardware compute partitioning on nvidia gpus for composable systems,” in *37th Euromicro Conference on Real-Time Systems (ECRTS 2025)*, pp. 21–1, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2025.
- [69] X. Zhao, G. Zhang, L. Wang, and H. Dai, “Ugpu: Dynamically constructing unbalanced gpus for enhanced resource efficiency,” in *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, pp. 1356–1369, 2025.
- [70] Y. Zhang, H. Yu, C. Han, C. Wang, B. Lu, Y. Li, Z. Jiang, Y. Li, X. Chu, and H. Li, “Sgdcrc: Software-defined dynamic resource control for concurrent dnn inference on nvidia gpus,” in *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pp. 267–281, 2025.
- [71] Y. Zhao, X. Liu, S. Liu, X. Li, Y. Zhu, G. Huang, X. Liu, and X. Jin, “Muxflow: Efficient and safe gpu sharing in large-scale production deep learning clusters,” *arXiv preprint arXiv:2303.13803*, 2023.
- [72] M. Li, W. Xiao, H. Yang, B. Sun, H. Zhao, S. Ren, Z. Luan, X. Jia, Y. Liu, Y. Li, *et al.*, “Easyscale: Elastic training with consistent accuracy and improved utilization on gpus,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–14, 2023.
- [73] A. Islam and M. Ghose, “Delta: Deadline aware energy and latency-optimized task offloading and resource allocation in gpu-enabled, pim-enabled distributed heterogeneous mec architecture,” *Journal of Systems Architecture*, vol. 159, p. 103335, 2025.

- [74] S. Wang, S. Chen, Y. Shi, G. Yao, and M. Liu, “Adagap: An adaptive gap-aware resource allocation strategy for gpu sharing in heterogeneous clusters,” *Future Generation Computer Systems*, p. 107883, 2025.
- [75] Z. Mo, H. Xu, and W. C. Lau, “Optimal resource efficiency with fairness in heterogeneous gpu clusters,” in *Proceedings of the 25th International Middleware Conference*, pp. 36–48, 2024.
- [76] T. Fu, Z. Yang, Z. Ye, C. Ma, Y. Han, Y. Luo, X. Wang, and Z. Wang, “A survey on the scheduling of dl and llm training jobs in gpu clusters,” *Chinese Journal of Electronics*, vol. 34, no. 3, pp. 881–905, 2025.
- [77] X. He, S. Pang, S. Qiao, H. Gui, S. Yu, J. J. Rodrigues, S. Mumtaz, and Z. Lyu, “Real-time scheduling of cpu/gpu heterogeneous tasks in dynamic iot systems: Enhancing gpu and memory efficiency,” *IEEE Transactions on Mobile Computing*, 2025.
- [78] B. Jiang, J. Yao, H. Mu, X. Su, and Z. Tang, “Dike: Enhancing fairness and efficiency in gpu clusters for deep learning,” in *ICASSP 2025-2025 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 1–5, IEEE, 2025.
- [79] Y. Zhang, L. Luo, G. Sun, H. Yu, and B. Li, “Deadline-aware online job scheduling for distributed training in heterogeneous clusters,” *IEEE Transactions on Cloud Computing*, 2025.
- [80] H. Ayadi, A. An, Y. Shao, H. Pourmedheji, J. Deng, J. X. Huang, M. Feiman, and H. Zhou, “Topology-aware gpu job scheduling with deep reinforcement learning and heuristics,” *Journal of Parallel and Distributed Computing*, p. 105138, 2025.
- [81] Z. Mo, H. Xu, and W. C. Lau, “Fast and fair training for deep learning in heterogeneous gpu clusters,” in *Proceedings of the 39th ACM International Conference on Supercomputing*, pp. 324–338, 2025.
- [82] A. Qiao, S. K. Choe, S. J. Subramanya, W. Neiswanger, Q. Ho, H. Zhang, G. R. Ganger, and E. P. Xing, “Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning,” in *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, 2021.
- [83] X. Wang, Y. Li, F. Guo, Y. Xu, and J. C. Lui, “Dynamic gpu scheduling with multi-resource awareness and live migration support,” *IEEE Transactions on Cloud Computing*, vol. 11, no. 3, pp. 3153–3167, 2023.
- [84] F. Strati, X. Ma, and A. Klimovic, “Orion: Interference-aware, fine-grained gpu sharing for ml applications,” in *Proceedings of the Nineteenth European Conference on Computer Systems*, pp. 1075–1092, 2024.

- [85] C. Li, Y. Sun, L. Jin, L. Xu, Z. Cao, P. Fan, D. Kaeli, S. Ma, Y. Guo, and J. Yang, "Priority-based pcie scheduling for multi-tenant multi-gpu systems," *IEEE Computer Architecture Letters*, vol. 18, no. 2, pp. 157–160, 2019.
- [86] J. Cao, Y. Guan, K. Qian, J. Gao, W. Xiao, J. Dong, B. Fu, D. Cai, and E. Zhai, "Crux: Gpu-efficient communication scheduling for deep learning training," in *Proceedings of the ACM SIGCOMM 2024 Conference*, pp. 1–15, 2024.
- [87] A. Gheibi-Fetrat, S. Safari, A. Ahmadi-Tonekaboni, S. Hessabi, and H. Sarbazi-Azad, "Padel: Priority-based real-time scheduling for gpus," *IEEE Embedded Systems Letters*, 2025.
- [88] M. Tsenos and V. Kalogeraki, "Exploring gpu-based workload scheduling techniques for edge computing," in *2025 IEEE International Conference on Cloud Engineering (IC2E)*, pp. 248–258, IEEE, 2025.
- [89] M. Kim, "Guaranteeing that multilevel prioritized dnn models on an embedded gpu have inference performance proportional to respective priorities," *IEEE Embedded Systems Letters*, vol. 14, no. 2, pp. 83–86, 2022.
- [90] J. Duan, S. Xu, S. Qian, D. Yang, K. Wang, C. Liao, Y. Yu, Q. Hua, H. Hu, Q. Wang, *et al.*, "Gfs: A preemption-aware scheduling framework for gpu clusters with predictive spot instance management," *arXiv preprint arXiv:2509.11134*, 2025.
- [91] R. Chab, F. Li, and S. Setia, "Algorithmic techniques for gpu scheduling: A comprehensive survey," *Algorithms*, vol. 18, no. 7, p. 385, 2025.
- [92] J. Y.-T. Leung, M. Dror, and G. H. Young, "A note on an open-end bin packing problem," *Journal of Scheduling*, vol. 4, no. 4, pp. 201–207, 2001.
- [93] "Resource Management for Pods and Containers." <https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/>, 2023.
- [94] G. Yeung, D. Borowiec, R. Yang, A. Friday, R. Harper, and P. Garraghan, "Horus: Interference-aware and prediction-based scheduling in deep learning systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 1, pp. 88–100, 2022.
- [95] B. Wu, Z. Zhang, Z. Bai, X. Liu, and X. Jin, "Transparent GPU sharing in container clouds for deep learning workloads," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, (Boston, MA), pp. 69–85, USENIX Association, Apr. 2023.
- [96] "Alibaba Cluster Trace." <https://github.com/alibaba/clusterdata>, 2023.
- [97] C. Fetzer, "Building critical applications using microservices," *IEEE Security & Privacy*, vol. 14, no. 6, pp. 86–89, 2016.

- [98] G. Blinowski, A. Ojdowska, and A. Przybyłek, “Monolithic vs. microservice architecture: A performance and scalability evaluation,” *IEEE Access*, vol. 10, pp. 20357–20374, 2022.
- [99] I. Fontana de Nardin, R. da Rosa Righi, T. R. Lima Lopes, C. André da Costa, H. Y. Yeom, and H. Köstler, “On revisiting energy and performance in microservices applications: A cloud elasticity-driven approach,” *Parallel Computing*, vol. 108, p. 102858, 2021.
- [100] J. Yu, K. Guo, Y. Hu, X. Ning, J. Qiu, H. Mao, S. Yao, T. Tang, B. Li, Y. Wang, and H. Yang, “Real-time object detection towards high power efficiency,” in *Proc. IEEE Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 704–708, 2018.
- [101] L. Weerasinghe and I. Perera, “Evaluating the inter-service communication on microservice architecture,” in *Proc. IEEE International Conference on Information Technology Research (ICITR)*, pp. 1–6, 2022.
- [102] S. Heldens, P. Hijma, B. Van Werkhoven, J. Maassen, and R. V. van Nieuwpoort, “Lightning: Scaling the GPU programming model beyond a single GPU,” in *Proc. IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 492–503, 2022.
- [103] C.-L. Su, P.-Y. Chen, C.-C. Lan, L.-S. Huang, and K.-H. Wu, “Overview and comparison of OpenCL and CUDA technology for GPGPU,” in *Proc. IEEE Asia Pacific Conference on Circuits and Systems*, pp. 448–451, 2012.
- [104] W. Shin, K.-H. Yoo, and N. Baek, “Large-scale data computing performance comparisons on sycl heterogeneous parallel processing layer implementations,” *Applied Sciences*, vol. 10, no. 5, p. 1656, 2020.
- [105] S. Kato, S. Brandt, Y. Ishikawa, and R. Rajkumar, “Operating systems challenges for GPU resource management,” in *Proc. International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pp. 23–32, 2011.
- [106] “NVIDIA Data Center GPU Manager.” <https://docs.nvidia.com/datacenter/dcgm/latest/user-guide/feature-overview.html>, 2022. [Online; accessed October-2022].
- [107] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, “Rodinia: A benchmark suite for heterogeneous computing,” in *Proc. IEEE International Symposium on Workload Characterization (IISWC)*, pp. 44–54, 2009.

- [108] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, “The scalable heterogeneous computing (shoc) benchmark suite,” in *Proceedings of the 3rd workshop on general-purpose computation on graphics processing units*, pp. 63–74, 2010.
- [109] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu, “Parboil: A revised benchmark suite for scientific and commercial throughput computing,” *Center for Reliable and High-Performance Computing*, vol. 127, no. 7.2, 2012.
- [110] P. Carvalho, R. Cruz, L. M. Drummond, C. Bentes, E. Clua, E. Cataldo, and L. A. Marzulo, “Kernel concurrency opportunities based on gpu benchmarks characterization,” *Cluster Computing*, vol. 23, pp. 177–188, 2020.
- [111] Y. Wu, A. Kirillov, F. Massa, W.-Y. Lo, and R. Girshick, “Detectron2.” <https://github.com/facebookresearch/detectron2>, 2019.
- [112] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” *arXiv preprint arXiv:1810.04805*, 2018.
- [113] J. Kong, J. Kim, and J. Bae, “Hifi-gan: Generative adversarial networks for efficient and high fidelity speech synthesis,” *Advances in neural information processing systems*, vol. 33, pp. 17022–17033, 2020.
- [114] J. Shen, R. Pang, R. J. Weiss, M. Schuster, N. Jaitly, Z. Yang, Z. Chen, Y. Zhang, Y. Wang, R. Skerrv-Ryan, *et al.*, “Natural tts synthesis by conditioning wavenet on mel spectrogram predictions,” in *2018 IEEE international conference on acoustics, speech and signal processing (ICASSP)*, pp. 4779–4783, IEEE, 2018.
- [115] R. Prenger, R. Valle, and B. Catanzaro, “Waveglow: A flow-based generative network for speech synthesis,” in *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 3617–3621, IEEE, 2019.
- [116] A. Sill, “The design and architecture of microservices,” *IEEE Cloud Computing*, vol. 3, no. 5, pp. 76–80, 2016.
- [117] J. Paul, S. Lu, B. He, *et al.*, “Database systems on gpus,” *Foundations and Trends® in Databases*, vol. 11, no. 1, pp. 1–108, 2021.
- [118] S. Mittal and S. Vaishay, “A survey of techniques for optimizing deep learning on gpus,” *Journal of Systems Architecture*, vol. 99, p. 101635, 2019.
- [119] L. Kundu, X. Lin, E. Agostini, V. Ditya, and T. Martin, “Hardware acceleration for open radio access networks: A contemporary overview,” *IEEE Communications Magazine*, vol. 62, no. 9, pp. 160–167, 2024.