ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Predictive inference serving for multi-tenant GPU clusters

Γκατζιούρας Δημήτριος
Α.Μ. : 03116145

Επιβλέπων **:** Δημήτριος Ι. Σούντρης
Καθηγητής ΕΜΠ

Αθήνα
Φεβρουάριος 2023

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Predictive inference serving for multi-tenant GPU clusters

Δημήτριος Γκατζιούρας
Α.Μ. : 03116145

Επιβλέπων : Δημήτριος Ι. Σούντρης
Καθηγητής ΕΜΠ

Τριμελής Επιτροπή Εξέτασης

Δημήτριος Σούντρης	Σωτήρης Ξύδης	Παναγιώτης Τσανάκας
(Υπογραφή)	(Υπογραφή)	(Υπογραφή)
....	....	....
Καθηγητής ΕΜΠ	Καθηγητής ΕΜΠ	Επίκουρος Καθηγητής ΕΜΠ

Ημερομηνία Εξέτασης:
21 Μαρτίου 2023

Copyright ©- All rights reserved Γκατζιούρας Δημήτριος, 2023.
Με επιφύλαξη κάθε δικαιώματος.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ΄ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

(Υπογραφή)

....
Γκατζιούρας Δημήτριος
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.
©2023 - All rights reserved.

Περίληψη

Καθώς η μηχανική μάθηση (ML) συνεχίζει να φέρνει επανάσταση στον τρόπο με τον οποίο οι οργανισμοί επεξεργάζονται και αναλύουν δεδομένα, η ανάγκη για εξειδικευμένους υπολογιστικούς πόρους γίνεται όλο και πιο σημαντική. Οι μονάδες επεξεργασίας γραφικών (GPU) έχουν αναδειχθεί ως μια δημοφιλής λύση για την επιτάχυνση των φορτίων εργασίας ML, προσφέροντας σημαντικά πλεονεκτήματα απόδοσης σε σχέση με τα παραδοσιακά συστήματα που βασίζονται σε CPU. Το νέφος έχει γίνει μια δημοφιλής επιλογή ανάπτυξης για τα φορτία εργασίας ML, προσφέροντας διάφορα πλεονεκτήματα, όπως η επεκτασιμότητα, η ευελιξία και η οικονομική αποδοτικότητα. Το Kubernetes, μια πλατφόρμα ενορχήστρωσης containers ανοικτού κώδικα, έχει γίνει μια δημοφιλής λύση για τη διαχείριση των φορτίων εργασίας που απαιτούν επιτάχυνση σε GPU. Ο συνδυασμός των GPU και του Kubernetes επιτρέπει στους οργανισμούς να κλιμακώνουν τα φορτία εργασίας ML κατά παραγγελία και να κατανέμουν αποτελεσματικά τους πόρους ώστε να ανταποκρίνονται στις απαιτήσεις των φορτίων εργασίας. Ωστόσο, η ανάπτυξη και η διαχείριση των φορτίων εργασίας που βασίζονται σε GPU στο Kubernetes μπορεί να αποτελέσει πρόκληση λόγω της εξειδικευμένης φύσης αυτών των πόρων. Οι οργανισμοί πρέπει να διαχειρίζονται και να κατανέμουν προσεκτικά τους πόρους GPU για να διασφαλίζουν βέλτιστη απόδοση και αποτελεσματική χρήση των πόρων. Αυτό μπορεί να περιλαμβάνει την εφαρμογή πολιτικών και διαδικασιών για την ιεράρχηση της πρόσβασης στους πόρους με βάση τις απαιτήσεις του φόρτου εργασίας, όπως ο τύπος του φόρτου εργασίας, η χρήση των πόρων και οι απαιτήσεις απόδοσης. Επιπλέον, η διασφάλιση της συμβατότητας και της διαλειτουργικότητας μεταξύ διαφορετικών στοιχείων υλικού και λογισμικού είναι κρίσιμη κατά την ανάπτυξη φορτίων εργασίας που βασίζονται σε GPU στο Kubernetes. Το Kubernetes υποστηρίζει διάφορους τύπους GPU, ο καθένας με τις δικές του προδιαγραφές και απαιτήσεις, και η ενσωμάτωση στοιχείων λογισμικού και υλικού τρίτων μπορεί να αποτελέσει πρόκληση.

Σε αυτή τη διατριβή, σχεδιάζουμε ένα σύστημα χρονοπρογραμματισμού GPU με επίγνωση πόρων και παρεμβολών με στόχο τον αποτελεσματικό χρονοπρογραμματισμό και/ή την παράλληλη τοποθέτηση εισερχόμενων εφαρμογών σε διάφορα ετερογενή τμήματα των GPU κέντρων δεδομένων. Ενσωματώνουμε τη λύση στο Kubernetes, ένα από τα πιο ευρέως χρησιμοποιούμενα πλαίσια ενορχήστρωσης νέφους. Δείχνουμε ότι ο χρονοπρογραμματιστής μας μπορεί να καλύψει πιο αποτελεσματικά τους περιορισμούς ποιότητας υπηρεσιών (QoS) των χρηστών, με υψηλότερη αξιοποίηση των πόρων σε σύγκριση με τους σύγχρονους χρονοπρογραμματιστές, για μια ποικιλία φόρτων εργασίας ML στο νέφος, διατηρώντας παράλληλα χαμηλά τις καθυστερήσεις και την κατανάλωση ενέργειας.

Λέξεις Κλειδιά— υπολογιστές νέφους, κάρτα γραφικών, διαχείριση πόρων, δρομολόγηση, παρεμβολή, Κυβερνετες, resource-aware, ετερογένεια

Abstract

As Machine Learning (ML) continues to revolutionize the way organizations process and analyze data, the need for specialized computing resources has become increasingly important. Graphics Processing Units (GPUs) have emerged as a popular solution for accelerating ML workloads, offering significant performance benefits over traditional CPU-based systems. The cloud has become a popular deployment option for ML workloads, offering several benefits such as scalability, flexibility, and cost-effectiveness. Kubernetes, an open-source container orchestration platform, has become a popular solution for managing containerized workloads, including those that require GPU acceleration. The combination of GPUs and Kubernetes enables organizations to scale ML workloads on-demand and efficiently allocate resources to meet workload requirements. However, deploying and managing GPU-based workloads on Kubernetes can be challenging due to the specialized nature of these resources. Organizations must carefully manage and allocate GPU resources to ensure optimal performance and efficient use of resources. This can include implementing policies and procedures to prioritize access to resources based on workload requirements, such as workload type, resource utilization, and performance requirements. Furthermore, ensuring compatibility and interoperability between different hardware and software components is critical when deploying GPU-based workloads on Kubernetes. Kubernetes supports several GPU types, each with their own specifications and requirements, and integrating third-party software and hardware components can be challenging.

In this thesis, we design a resource and interference aware GPU scheduling system with the aim of efficiently scheduling and/or collocating incoming applications on various heterogeneous data center GPU partitions. We integrate our solution with Kubernetes, one of the most widely used cloud orchestration framework. We show that our scheduler can cover users' Quality of Service (QoS) constraints more efficiently, with higher resource utilization compared to the state-of-the-art schedulers, for a variety of ML cloud workloads while maintaining low overheads and energy consumption.

Keywords— cloud computing, GPU, interference, resource management, scheduling, Kubernetes, heterogeneity

Ευχαριστίες

Αρχικά, θα ήθελα να ευχαριστήσω τον επιβλέποντά μου, Καθηγητή Δημήτριο Σούντρη ΕΜΠ, ο οποίος μου έδωσε την ευκαιρία να εκπονήσω τη διπλωματική μου εργασία στο Εργαστήριο Μικροϋπολογιστών και Ψηφιακών Συστημάτων (MicroLab) στο ΕΜΠ.

Επίσης, θα ήθελα να εκφράσω την ευγνωμοσύνη μου στους υποψήφιους διδάκτορες Άγγελο Φερίκογλου, Αχιλλέα Τζενετόπουλο και Δημοσθένη Μασούρο για τη βοήθεια και τη συνεργασία τους καθ ΄όλη τη διάρκεια της διπλωματικής μου εργασίας. Η συνεχής τριβή μας κατά τη διάρκεια της διπλωματικής, με βοήθησε να αποκτήσω γρηγορότερα γνώσεις σχετικές με το αντικείμενο που εξετάσαμε, οι οποίες είναι εφαρμόσιμες σε πολλούς τομείς της σύγχρονης τεχνολογίας και των συστημάτων υπολογιστών. Παράλληλα με εισήγαγαν στην ερευνητική προσέγγιση και εργασία. Θα ήθελα επίσης να ευχαριστήσω όλα τα μέλη του Microlab για το ευχάριστο περιβάλλον εργασίας.

Τέλος, θα ήθελα να ευχαριστήσω τους γονείς μου Σάκη και Ειρήνη και την αδερφή μου Δέσποινα για την υποστηρίξη τους σε ότι επιχειρούσα κατά τη διάρκεια της ζωής και των σπουδών μου.

Contents

Περίληψη **5**

Abstract **7**

Ευχαριστίες **9**

1 Εκτεταμένη Ελληνική Περίληψη **19**
- 1.1 Εισαγωγή . . . 19
- 1.2 Kubernetes . . . 22
- 1.3 Πειραματική Υποδομή . . . 23
- 1.4 Συστήματα Συστάσεων . . . 26
- 1.5 Predictive inference serving for multi-tenant GPU clusters . . . 27
  - 1.5.1 Αξιολόγηση . . . 32
- 1.6 Schedulers . . . 33
  - 1.6.1 Min . . . 33
  - 1.6.2 Max . . . 34
  - 1.6.3 Round Robin . . . 34

2 Introduction **35**
- 2.1 Cloud Computing and Kubernetes . . . 35
- 2.2 Accelerators in Kubernetes . . . 36
- 2.3 Overview . . . 37

3 Related Work **39**
- 3.1 Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters 39
- 3.2 MISO: Exploiting Multi-Instance GPU Capability on Multi-Tenant Systems for Machine Learning . . . 41
- 3.3 Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads . . . 42
- 3.4 KubeShare: A Framework to Manage GPUs as First-Class and Shared Resources in Container Cloud . . . 43

4 The Kubernetes Orchestrator 47
4.1 Virtualization 47
4.2 Virtual Machines 47
4.3 Containers and Docker 48
4.3.1 Orchestration 48
4.3.2 Cloud Orchestration 50
4.4 Kubernetes Control Plane Components 51
4.5 Kubernetes Worker Node(s) Components 52
4.5.1 Other Important Addons 53
4.6 Kubernetes Architecture 53
4.6.1 Cluster 53
4.6.2 Nodes 54
4.6.3 Pods 54
4.6.4 DaemonSet 55
4.6.5 Deployment 55
4.6.6 Service 56
4.6.7 ConfigMaps 56
4.7 Kubernetes Resources 56
4.7.1 Default Resources: CPU and Memory 56
4.7.2 Extended Resources 58
4.7.3 Kubernetes Device Plugins 58
4.8 Kubernetes Scheduler 59
4.8.1 Scheduling overview 59
4.8.2 Kubernetes Scheduling Framework 59
4.8.3 Scheduling and Binding Cycle 60

5 Experimental Infrastructure 61
5.1 System setup 61
5.2 GPU Infrastructure 62
5.2.1 NVIDIA Volta 62
5.2.2 NVIDIA Ampere 64
5.3 GPU Monitoring System 65
5.3.1 NVIDIA GPU Metrics Exporter 65
5.3.2 Prometheus Timeseries Database 66
5.4 Description of Cloud GPU workloads 67
5.4.1 MLPerf Benchmarks 67
5.4.2 MLPerf Inference 68

6 Characterization 73
6.1 Introduction 73
6.2 Utilizing Different Configurations 74

6.3 Fixed workload 76
6.4 Interference 77

7 Recommendation Systems 87
7.1 What Is a Recommendation System? 87
7.2 Collaborative Filtering with SVD 88
7.3 Multivariate feature imputation 89
7.4 Flexibility of Multivariate feature imputation 90
7.5 Evaluation 90

8 Predictive inference serving for multi-tenant GPU clusters 93
8.1 Redis 93
8.2 Exporter 94
8.3 Predicting and Scoring 94
    8.3.1 Score 96
8.4 PostBind and CUDA_VISIBLE_DEVICES 97

9 Evaluation 99
9.1 Description 99
9.2 Schedulers 100
    9.2.1 Min 100
    9.2.2 Max 101
    9.2.3 Round Robin 101
9.3 Schedulers Comparison 101
9.4 QPS Distributions 104

10 Conclusion and Future Work 107
10.1 Summary 107
10.2 Future Work 107
    10.2.1 Development Scope 107
    10.2.2 Research Scope 108

List of Figures

4.1 Container Orchestrator 50
4.2 Kubernetes Architecture 53
4.3 Cluster-Node abstraction level 54
4.4 Node-Pod-Container abstraction levels 54
4.5 Pod Scheduling Context 60

5.1 Experimental Infrastructure Overview 62
5.2 Prometheus Architecture 66
5.3 Load Generator Integration in MLPerf Inference Benchmarks . . 70
5.4 Overall System 71

6.1 Queries per second for all models and all configurations 75
6.2 DCGM_FI_PROF_GR_ENGINE_ACTIVE (in %) 76
6.3 Tensorflow Resnet50 DCGM metrics 76
6.4 GPU Utilization for fixed workload 78
6.5 DCGM_FI_DEV_FB_USED (in MiB) for Fixed Workload . . 79
6.6 Combined QPS for fixed workload 80
6.7 Tensorflow Renset50 and SSD mobilenet fixed workload 81
6.8 QPS, Interference. 2 Partitions A30 82
6.8 QPS, Interference. 2 Partitions A30 83
6.9 QPS, A30, 1 Partition A30 84
6.9 QPS, A30, 1 Partition A30 85
6.10 QPS percentage difference due to Interference on V100 86

7.1 Imputer and SVD Mean Squared Errors 91

8.1 Scores comparison 96

9.1 QPS-SLO percentage difference 102
9.2 QoS Metrics 103
9.3 Errors 104
9.4 Energy consumption 104
9.5 QPS Distributions for high SLOs 105
9.6 QPS Distributions for low SLOs 106

List of Tables

1.1 Virtual Machines Characteristics 23
1.2 Χαρακτηριστικά GPU 24
1.3 MLPerf Inference Benchmarks 26

5.1 Virtual Machines Characteristics 61
5.2 GPU Characteristics 65
5.3 MLPerf Inference Benchmarks 68

Chapter 1

Εκτεταμένη Ελληνική Περίληψη

1.1 Εισαγωγή

Το cloud computing και το Kubernetes είναι δύο ισχυρές τεχνολογίες που έχουν αναβιώσει τον τρόπο με τον οποίο οι επιχειρήσεις διαχειρίζονται και αναπτύσσουν τις IT υποδομές τους. Το cloud computing επιτρέπει στους χρήστες να έχουν πρόσβαση σε εικονικούς υπολογιστικούς πόρους όπως διακομιστές, αποθηκευτικούς χώρους, εφαρμογές και υπηρεσίες, μέσω του διαδικτύου. Το Cloud computing προσφέρει μια σειρά από υπηρεσίες, συμπεριλαμβανομένης της υποδομής ως υπηρεσία (IaaS), της πλατφόρμας ως υπηρεσία (PaaS) και του λογισμικού ως υπηρεσία (SaaS), που επιτρέπουν στους χρήστες να αυξάνουν ή να μειώνουν τους πόρους τους ανάλογα με τις ανάγκες τους, να πληρώνουν μόνο για ό,τι χρησιμοποιούν και να έχουν πρόσβαση στα δεδομένα και τις εφαρμογές τους από οπουδήποτε, ανά πάσα στιγμή, χρησιμοποιώντας οποιαδήποτε συσκευή με σύνδεση στο διαδίκτυο. Το υπολογιστικό νέφος προσφέρει πολλά πλεονεκτήματα σε σχέση με την παραδοσιακή IT υποδομή, όπως μειωμένο κόστος, αυξημένη ευελιξία και επεκτασιμότητα, βελτιωμένη προσβασιμότητα και αυξημένη ασφάλεια και αξιοπιστία. Το Kubernetes, από την άλλη πλευρά, είναι μια πλατφόρμα ενορχήστρωσης containers που απλοποιεί τη διαχείριση και το scaling των εφαρμογών που περιέχουν containers. Το Kubernetes προσφέρει αρκετά πλεονεκτήματα σε σχέση με τις παραδοσιακές μεθόδους ανάπτυξης εφαρμογών, συμπεριλαμβανομένης της βελτιωμένης αξιοποίησης των πόρων, των ταχύτερων χρόνων ανάπτυξης και της αυξημένης ευελιξίας και επεκτασιμότητας. Το Kubernetes αυτοματοποιεί την ανάπτυξη, το scaling και τη διαχείριση των containers, επιτρέποντας στους χρήστες να επικεντρωθούν στην ανάπτυξη των εφαρμογών τους και όχι στη διαχείριση της υποκείμενης υποδομής. Επιπλέον, το Kubernetes προσφέρει μια σειρά χαρακτηριστικών, όπως load balancing, fault tolerance κ.α. που εξασφαλίζουν υψηλή διαθεσιμότητα και αξιοπιστία. Ο συνδυασμός υπολογιστικού νέφους και Kubernetes προσφέρει μια εξαιρετικά επεκτάσιμη και ευέλικτη λύση για τη διαχείριση και την ανάπτυξη υποδομών. Kubernetes μπορεί να χρησιμοποιηθεί για τη διαχείριση εφαρμογών με containers που εκτελούνται στο υπολογιστικό νέφος, επιτρέποντας στους χρήστες να επωφεληθούν από την επεκτασιμότητα και την ευελιξία του του νέφους, ενώ παράλληλα επωφελούνται από τα χαρακτηριστικά αυτοματοποίησης και διαχείρισης που παρέχει το Kubernetes. Ο συνδυασμός του kubernetes και του cloud προσφέρει μια εξαιρετικά επεκτάσιμη και ευέλικτη λύση για τη διαχείριση και την ανάπτυξη εφαρμογών και υπηρεσιών, επιτρέποντας στις επιχειρήσεις να βελτιώσουν τη λειτουργική τους αποδοτικότητα, να μειώσουν το κόστος και να αυξήσουν την ευελιξία. Η χρήση μονάδων επεξεργασίας γραφικών (GPU) και άλλων επιταχυντών στο Kubernetes έχει γίνει ολοένα και πιο δημοφιλής για την επεξεργασία και ανάλυση μεγάλου όγκου δεδομένων. Αυτοί οι εξειδικευμένοι υπολογιστικοί πόροι προσφέρουν σημαντικά πλεονεκτήματα απόδοσης σε σχέση με τα παραδοσιακά συστήματα που βασίζονται σε CPU. Ωστόσο, υπάρχουν αρκετές δυσκολίες και προκλήσεις που σχετίζονται με τη χρήση GPU και άλλων επιταχυντών στο Kubernetes.

Μία από τις κύριες προκλήσεις είναι η αδυναμία διαμοιρασμού αυτών των εξειδικευμένων πόρων μεταξύ πολλαπλών φόρτων εργασίας. Σε αντίθεση με τα συστήματα που βασίζονται σε CPU, οι GPU και άλλοι επιταχυντές συνήθως δεν έχουν σχεδιαστεί για να μοιράζονται μεταξύ πολλαπλών εφαρμογών ταυτόχρονα. Αυτό μπορεί να οδηγήσει σε διαμάχη για πόρους, ιδίως σε περιβάλλοντα με πολλούς χρήστες, όπου αυτοί και φόρτοι εργασίας ανταγωνίζονται για πρόσβαση σε αυτούς τους πόρους. Για την αντιμετώπιση αυτής της πρόκλησης, οι οργανισμοί πρέπει να διαχειρίζονται και να κατανέμουν προσεκτικά αυτούς τους εξειδικευμένους πόρους ώστε να διασφαλίζουν ότι χρησιμοποιούνται αποτελεσματικά. Αυτό μπορεί να περιλαμβάνει την εφαρμογή πολιτικών και διαδικασιών για την ιεράρχηση των προτεραιοτήτων πρόσβασης στους πόρους με βάση τις απαιτήσεις του φόρτου εργασίας, όπως ο τύπος του φόρτου εργασίας, η χρήση των πόρων και οι απαιτήσεις επιδόσεων. Μια άλλη πρόκληση είναι η διασφάλιση της συμβατότητας και της διαλειτουργικότητας μεταξύ ετερογενών στοιχείων υλικού και λογισμικού. Το Kubernetes υποστηρίζει διάφορους τύπους GPU και επιταχυντών, ο καθένας με τις δικές του προδιαγραφές και απαιτήσεις. Η διασφάλιση της συμβατότητας και της διαλειτουργικότητας μεταξύ ετερογενών συστατικών μπορεί να αποτελέσει πρόκληση, ιδίως όταν ενσωματώνονται συστατικά λογισμικού και υλικού τρίτων κατασκευαστών. Επιπλέον, η ανάπτυξη και η διαχείριση φορτίων εργασίας που βασίζονται σε GPU και άλλους επιταχυντές μπορεί να είναι δύσκολη λόγω της εξειδικευμένης φύσης τους. Εφαρμογές και υπηρεσίες που χρησιμοποιούν αυτούς τους πόρους μπορεί να απαιτούν εξειδικευμένες διαμορφώσεις και βελτιστοποιήσεις για να διασφαλιστεί η μέγιστη δυνατή απόδοση. Αποσφαλμάτωση και αντιμετώπιση προβλημάτων που σχετίζονται με φόρτους εργασίας που βασίζονται σε GPU μπορεί επίσης να είναι πρόκληση, απαιτώντας εξειδικευμένες γνώσεις και εμπειρογνωμοσύνη. Συνοψίζοντας, ενώ οι GPU και άλλοι επιταχυντές προσφέρουν σημαντικές επιδόσεις, υπάρχουν αρκετές δυσκολίες και προκλήσεις που σχετίζονται με τη χρήση τους στο Kubernetes. Αυτές οι προκλήσεις περιλαμβάνουν την αδυναμία διαμοιρασμού πόρων μεταξύ πολλαπλών φόρτων εργασίας, τη διασφάλιση της συμβατότητας και της διαλειτουργικότητας μεταξύ των στοιχείων, καθώς και εξειδικευμένες απαιτήσεις ανάπτυξης και διαχείρισης. Η αντιμετώπιση αυτών των προκλήσεων απαιτεί προσεκτικό σχεδιασμό και διαχείριση για τη διασφάλιση της βέλτιστης απόδοσης και της αποτελεσματικής χρήσης των πόρων. Στην παρούσα εργασία, σχεδιάζουμε έναν νέο χρονοπρογραμματιστή GPU που βασίζεται σε προβλέψεις AI μοντέλων σχετικά με τις επιδόσεις των εφαρμογών σε πολλαπλές διαφορετικές διατάξεις GPU, καθώς και τις αναμενόμενες παρεμβολές που παράγονται λόγω της συντοποθέτησης διαφορετικών εφαρμογών στην ίδια διάταξη. Προκειμένου να αποφασιστεί η βέλτιστη διαμόρφωση για κάθε εφαρμογή εκμεταλλευόμαστε επίσης σε πραγματικό χρόνο μετρικές από την GPU που ανακτώνται από διάφορα συστήματα παρακολούθησης. Εντοπίζουμε την αναποτελεσματικότητα των σύγχρονων χρονοπρογραμματιστών GPU του Kubernetes όσον αφορά την ποιότητα των υπηρεσίας (QoS) και τη χρήση των πόρων. Δείχνουμε ότι ο χρονοπρογραμματιστής μας, για την πλειονότητα των φόρτων εργασίας και των σεναρίων χρονοπρογραμματισμού μπορεί να επιτύχει χαμηλότερο χρόνο εκτέλεσης κατά μέσο όρο, καθώς και καλύτερη αξιοποίηση των πόρων, ενώ διασφαλίζει ευελιξία και μένει πιστός στις απαιτήσεις των χρηστών, χωρίς να επιβαρύνεται με υψηλές καθυστερήσεις ή κατανάλωση ενέργειας. Στο κεφάλαιο 2, αναλύουμε άλλες προσεγγίσεις χρονοπρογραμματισμού GPU που έχουν προταθεί μέχρι σήμερα. Στο κεφάλαιο 3, συζητάμε για τις βασικές έννοιες του Kubernetes, του υποκείμενου ενορχηστρωτή containers. Στο κεφάλαιο 4, παρουσιάζουμε τη δική μας πειραματική υποδομή, το σύστημα παρακολούθησης GPU και τη σουίτα που χρησιμοποιήθηκε για εκτέλεση των φόρτων εργασίας και την αξιολόγηση. Στο κεφάλαιο 5, παρουσιάζουμε και συζητάμε τα αποτελέσματα του χαρακτηρισμού των φόρτων εργασίας (ML workloads) που χρησιμοποιήσαμε για τα πειράματά μας και τη δημιουργία των συνόλων δεδομένων που τροφοδοτούν τα μοντέλα τεχνητής νοημοσύνης μας. Στο κεφάλαιο 6, παρέχουμε πληροφορίες σχετικά με τα μοντέλα μας και εξηγούμε τις τεχνικές που χρησιμοποιήσαμε για την παροχή προτάσεων σε βάθος. Στο κεφάλαιο 7, παρουσιάζουμε τον χρονοπρογραμματιστή μας. Στο κεφάλαιο 8, αξιολογούμε το προτεινόμενο πλαίσιό μας και το συγκρίνουμε με πολλαπλές γνωστές επιλογές για προγραμματισμό εργασιών βασισμένων σε GPU στον ενορχηστρωτή Kubernetes, σε διαφορετικά σενάρια και φόρτους εργασίας. Τέλος, στο κεφάλαιο 9, συνοψίζουμε την εργασία μας και προτείνουμε πεδία με, ενδεχομένως, υψηλό ενδιαφέρον για μελλοντική εργασία σε συναφή ερευνητικά πεδία.

1.2 Kubernetes

Το Kubernetes είναι ένα σύστημα ενορχήστρωσης containers ανοιχτού κώδικα που αυτοματοποιεί την ανάπτυξη, την κλιμάκωση και τη διαχείριση εφαρμογών που χρησιμοποιούν container. Αναπτύχθηκε αρχικά από την Google και τώρα συντηρείται από το Cloud Native Computing Foundation (CNCF).

Τα containers είναι ένας ελαφρύς τρόπος συσκευασίας λογισμικού που επιτρέπει στους προγραμματιστές να δημιουργούν, να δοκιμάζουν και να αναπτύσσουν εφαρμογές γρήγορα και αξιόπιστα σε διαφορετικά περιβάλλοντα. Ωστόσο, η διαχείριση των containers σε κλίμακα μπορεί να αποτελέσει πρόκληση, καθώς απαιτεί συντονισμό μεταξύ πολλαπλών containers, σε πολλαπλούς κεντρικούς υπολογιστές ή κόμβους.

Σε αυτό το σημείο έρχεται στο προσκήνιο το Kubernetes. Παρέχει έναν τρόπο διαχείρισης των containers που δεν εξαρτάται από την πλατφόρμα, αφαιρώντας την υποκείμενη υποδομή και επιτρέποντας στους προγραμματιστές να επικεντρωθούν στην κατασκευή και αποστολή των εφαρμογών τους. Το Kubernetes το πετυχαίνει αυτό μέσω μιας σειράς αφαιρέσεων, όπως τα Pods, τις υπηρεσίες και τα Deployments, οι οποίες παρέχουν έναν συνεπή τρόπο για τον ορισμό, την ανάπτυξη και τη διαχείριση containers.

Στον πυρήνα του Kubernetes βρίσκεται η έννοια του cluster, η οποία είναι ένα σύνολο κόμβων που συνεργάζονται για την εκτέλεση εφαρμογών που περιέχονται σε containers. Κάθε κόμβος εκτελεί ένα πρόγραμμα εκτέλεσης container, όπως το Docker, και διαχειρίζεται από έναν πράκτορα του Kubernetes που ονομάζεται kubelet. Το kubelet είναι υπεύθυνο για τη διασφάλιση της σωστής εκτέλεσης των containers στον κόμβο του και για την επικοινωνία με το υπόλοιπο σύστημαγια τον συντονισμό της τοποθέτησης και του χρονοπρογραμματισμού των containers.

Στο Kubernetes, οι εφαρμογές ορίζονται ως συλλογές ενός ή περισσοτέρων containers, ενθυλακωμένες σε ένα Pod. Ένα Pod είναι η μικρότερη μονάδα που μπορεί να αναπτυχθεί στο Kubernetes και αντιπροσωπεύει μια μοναδική περίπτωση μιας εφαρμογής. Τα Pods είναι εφήμερα, που σημαίνει ότι μπορούν να δημιουργηθούν, να καταστραφούν ή να επανεκκινήσουν ανά πάσα στιγμή, και το Kubernetes θα χειριστεί αυτόματα τον προγραμματισμό και τον επαναπρογραμματισμό των containers για να διατηρήσει την επιθυμητή κατάσταση της εφαρμογής.

Οι υπηρεσίες είναι μια άλλη σημαντική αφαίρεση στο Kubernetes, οι οποίες παρέχουν μια σταθερή διεύθυνση IP και όνομα DNS για ένα σύνολο Pods. Οι υπηρεσίες επιτρέπουν στους πελάτες να έχουν πρόσβαση σε μια εφαρμογή που εκτελείται, ακόμη και όταν τα υποκείμενα Pods και containers δημιουργούνται, καταστρέφονται ή μετακινούνται στο σύμπλεγμα.

Τα deployments παρέχουν έναν τρόπο διαχείρισης του κύκλου ζωής μιας εφαρμογής, συμπεριλαμβανομένων των κυλιόμενων ενημερώσεων, των rollbacks και του scaling. Ένα deployment διαχειρίζεται ένα σύνολο αντιγράφων ενός Pod και μπορεί να χρησιμοποιηθεί για να διασφαλιστεί ότι ένας συγκεκριμένος αριθμός αντιγράφων εκτελείται πάντα, για να εκτελεστούν rolling updates της εφαρμογής και για να διαχειριστεί το scaling της εφαρμογής ανάλογα με τη ζήτηση.

Συνοψίζοντας, το Kubernetes είναι μια ισχυρή και ευέλικτη πλατφόρμα για τη διαχείριση εφαρμογών που περιέχουν container, η οποία παρέχει υψηλό επίπεδο αφαίρεσης της υποκείμενης υποδομής. Με τη χρήση του Kubernetes, οι προγραμματιστές μπορούν να επικεντρωθούν στην κατασκευή και αποστολή των εφαρμογών τους, αφήνοντας τις λεπτομέρειες της ενορχήστρωσης και διαχείρισης των containers στην πλατφόρμα.

1.3 Πειραματική Υποδομή

Σε αυτό το κεφάλαιο, περιγράφουμε το cluster που δημιουργήσαμε για τα πειράματά μας, το σύστημα παρακολούθησης της GPU και τη σουίτα αναφοράς MLPerf η οποία χρησιμοποιήθηκε για τη δημιουργία των φόρτων εργασίας.

Για τους κόμβους του cluster μας, δημιουργήσαμε 4 εικονικές μηχανές (VM) (1 Master κόμβος και 3 Worker κόμβοι) πάνω στις φυσικές μηχανές. Οι CPUs των VM αποτελούνται από 4 έως 8 πυρήνες και τα μεγέθη της RAM κυμαίνονται από 8 έως 16 GB. Χρησιμοποιήσαμε το Qemu KVM ως hypervisor. Από τους τρεις εργατικούς κόμβους, ο πρώτος είναι εξοπλισμένος με μια GPU NVIDIA V100 με μνήμη 32 GB και 80 SMs, ο δεύτερος με μια NVIDIA A30 GPU με 24GB μνήμης και 56 SMs, ενώ η τρίτη δεν έχει GPU. Όλες οι εικονικές μηχανές αναπτύσσονται στην υποδομή του εργαστηρίου του ΕΜΠ. Προκειμένου να προσομοιωθεί ένα περιβάλλον νέφους, όλες οι αναφερόμενα φορτία εργασίας που εκτελούνται στο cluster έχουν γίνει containerized χρησιμοποιώντας το Docker. Τα χαρακτηριστικά κάθε εικονικής μηχανής περιγράφονται στον ακόλουθο πίνακα.

Virtual Machines				
Processor	**Role**	**CPU Cores**	**CPU RAM (GB)**	**GPU Access**
Intel(R) Xeon(R) Gold 6138 CPU @ 2.00GHz	Master	8	8	No
Intel(R) Xeon(R) CPU E5-2658A v3 @ 2.20GHz	Worker	4	16	No
Intel(R) Xeon(R) Gold 5218R CPU @ 2.10GHz	Worker	8	16	V100
Intel(R) Xeon(R) Gold 5218R CPU @ 2.10GHz	Worker	8	16	A30

Table 1.1: Virtual Machines Characteristics

Ο συνδυασμός των VM με τα κοντέινερ είναι σήμερα ο συνήθης τρόπος ανάπτυξης συστημάτων νέφους σε κλίμακα, καθώς δημιουργεί τον τέλειο καταλύτη για την αξιοπιστία και την ευρωστία. Πάνω από τα VMs, έχουμε αναπτύξει το

Kubernetes ως ενορχηστρωτή εμπορευματοκιβωτίων, μια από τις πιο δημοφιλείς και πιο συχνά χρησιμοποιούμενες πλατφόρμες σήμερα.

Το σύστημα στο σύνολό του απεικονίζεται στο σχήμα 5.1.

Τα προϊόντα Tesla της εταιρείας NVIDIA εισάγουν μια σειρά επεξεργαστών υπολογιστικών γραφικών που μοιάζουν πολύ με τη σειρά NVIDIA Quadro (συνήθως χρησιμοποιούν το ίδιο τσιπ). Ωστόσο, διαθέτουν απομονωμένη διεπαφή οθόνης. Διατίθενται επίσης σε παθητικά ψυχόμενους τύπους, οι οποίοι είναι ειδικά κατάλληλοι για χρήση σε διακομιστές (rack-mounts).

Οι χρήστες επαγγελματικών εφαρμογών μπορούν, χάρη στην αρχιτεκτονική CUDA, να χρησιμοποιούν επεξεργαστές ροής γραφικών CUDA. Χάρη σε αυτό, είναι δυνατή η χρήση της ακατέργαστης απόδοσης μιας κάρτας γραφικών για συγκεκριμένους υπολογισμούς, γεγονός που μπορεί να αυξήσει σημαντικά την ταχύτητα εργασίας σε σύγκριση με τη χρήση ενός παραδοσιακού επεξεργαστή, οι οποίοι περιορίζονται σημαντικά από τον μικρότερο αριθμό πυρήνων.

Ο Tesla V100 έχει σχεδιαστεί από την αρχή για την απλοποίηση της δυνατότητας προγραμματισμού Το NVIDIA NVLink στο Tesla V100 παρέχει 2 φορές υψηλότερη απόδοση σε σύγκριση με την προηγούμενη γενιά Εξοπλισμένο με 640 πυρήνες Tensor, το Tesla V100 παρέχει 125 TeraFLOPS απόδοσης βαθιάς μάθησης Με ένα συνδυασμό βελτιωμένου ακατέργαστου εύρους ζώνης 900 GB/s και υψηλότερης αποδοτικότητας χρήσης DRAM στο 95%, το Tesla V100 προσφέρει.

Η NVIDIA A30 Tensor Core GPU παρέχει μια ευέλικτη πλατφόρμα για κύριους επιχειρηματικούς φόρτους εργασίας, όπως η εξαγωγή συμπερασμάτων AI, η εκπαίδευση και το HPC. Με την υποστήριξη TF32 και FP64 Tensor Core, καθώς και με μια ολοκληρωμένη στοίβα λύσεων λογισμικού και υλικού, η A30 διασφαλίζει ότι οι mainstream εφαρμογές εκπαίδευσης AI και HPC μπορούν να αντιμετωπιστούν γρήγορα. Η Multi-instance GPU (MIG) διασφαλίζει την ποιότητα υπηρεσίας (QoS) με ασφαλείς, κατανεμημένες σε υλικό, κατάλληλου μεγέθους GPU σε όλα αυτά τα φορτία εργασίας για διαφορετικούς χρήστες, αξιοποιώντας βέλτιστα τους υπολογιστικούς πόρους GPU.

Τα χαρακτηριστικά κάθε GPU περιγράφονται στον ακόλουθο πίνακα.

GPUs		
Όνομα GPU	**SMs**	**Μνήμη (GB)**
NVIDIA Tesla V100	80	32
NVIDIA Ampere A30	56	24

Table 1.2: Χαρακτηριστικά GPU

Το DCGM-Exporter είναι ένα εργαλείο βασισμένο στα Go APIs του NVIDIA DCGM που επιτρέπει στους χρήστες να συλλέγουν μετρήσεις GPU και να κατανοούν τη συμπεριφορά του φόρτου εργασίας ή να παρακολουθούν τις GPU σε clusters.

Το dcgm-exporter είναι γραμμένο σε Go και εκθέτει μετρήσεις GPU σε ένα τελικό σημείο HTTP (/metrics) για λύσεις παρακολούθησης.

Αυτές οι μετρήσεις εξάγονται σε μορφή χρονοσειρών προκειμένου να χρησιμοποιηθούν από βάσεις δεδομένων χρονοσειρών όπως οι Influx, Prometheus κ.λπ. Από τη σκοπιά του Kubernetes, ο εξαγωγέας DCGM σχηματίζει ένα DaemonSet που εκκινεί ένα damon Pods σε κάθε κόμβο εξοπλισμένο με GPU. Αυτά τα Pods εκτελούν ερωτήματα μετρήσεων στις GPU των κόμβων χρησιμοποιώντας το NVIDIA Data Center GPU Manager (DCGM) [1]. Τέλος, αυτές οι μετρήσεις αποστέλλονται στο σύστημα παρακολούθησης prometheus [2] το οποίο εκθέτει μια υπηρεσία στο cluster από την οποία οποιοσδήποτε πόρος του cluster μπορεί να έχει πρόσβαση στις μετρήσεις.

The GPU metrics we mainly used in our experiments are presented below.

☐ `DCGM_FI_DEV_FB_FREE`: Ελεύθερη μνήμη framebuffer (σε MiB)

☐ `DCGM_FI_DEV_FB_USED`: Χρησιμοποιούμενη μνήμη framebuffer (σε MiB)

☐ `DCGM_FI_PROF_GR_ENGINE_ACTIVE`: Αναλογία του χρόνου που η μηχανή γραφικών είναι ενεργή (σε %)

☐ `DCGM_FI_DEV_TOTAL_ENERGY_CONSUMPTION`: Συνολική κατανάλωση ενέργειας από την εκκλινηση (σε mJ)

☐ `DCGM_FI_DEV_POWER_USAGE`: Κατανάλωση ενέργειας (σε W)

☐ `DCGM_FI_PROF_DRAM_ACTIVE`: Ποσοστό κύκλων κατά τους οποίους η διεπαφή μνήμης της συσκευής είναι ενεργή στέλνοντας ή λαμβάνοντας δεδομένα (σε %)

Το MLPerf Inference [3, 4] είναι μια σουίτα αναφοράς για τη μέτρηση της ταχύτητας με την οποία τα συστήματα μπορούν να επεξεργάζονται εισόδους και να παράγουν αποτελέσματα χρησιμοποιώντας ένα εκπαιδευμένο μοντέλο. Ακολουθεί μια σύντομη περίληψη των τρεχόντων benchmarks και των μετρικών.

Κάθε σημείο αναφοράς MLPerf Inference ορίζεται από ένα μοντέλο, ένα σύνολο δεδομένων, έναν στόχο ποιότητας και έναν περιορισμό καθυστέρησης. Τα ακόλουθα τρία σημεία αναφοράς βρίσκονται στην έκδοση v0.5 της σουίτας και χρησιμοποιήθηκαν για τη δημιουργία του φόρτου εργασίας.

Σε κάθε benchmark το προ-εκπαιδευμένο μοντέλο έχει οριστεί σε ένα backend όπως Tensorflow, PyTorch, Onnx Runtime κλπ

Area	**Task**	**Backend**	**Model**	**Dataset**	**Quality**
Όραση	Ταξινόμηση εικόνων	Onnxruntime	Resnet50-v1.5	ImageNet (224x224)	FP32 (76.46%)
Όραση	Ταξινόμηση εικόνων	Onnxruntime	MobileNets-v1 224	ImageNet (224x224)	FP32 (71.68%)
Όραση	Ανίχνευση αντικειμένων	Onnxruntime	SSD-MobileNets-v1	COCO (300x300)	FP32 (0,22 mAP)
Όραση	Ταξινόμηση εικόνων	Tensorflow	Resnet50-v1.5	ImageNet (224x224)	FP32 (76.46%)
Όραση	Ταξινόμηση εικόνων	Tensorflow	MobileNets-v1 224	ImageNet (224x224)	FP32 (71.68%)
Όραση	Ανίχνευση αντικειμένων	Tensorflow	SSD-MobileNets-v1	COCO (300x300)	FP32 (0,22 mAP)

Table 1.3: MLPerf Inference Benchmarks

1.4 Συστήματα Συστάσεων

Ένα σύστημα συστάσεων είναι ένας αλγόριθμος τεχνητής νοημοσύνης ή ΤΝ, που συνήθως συνδέεται με τη μηχανική μάθηση, ο οποίος χρησιμοποιεί μεγάλα δεδομένα για να προτείνει ή να συστήσει πρόσθετα προϊόντα στους καταναλωτές. Αυτά μπορεί να βασίζονται σε διάφορα κριτήρια, όπως προηγούμενες αγορές, ιστορικό αναζήτησης, δημογραφικές πληροφορίες και άλλους παράγοντες. Τα συστήματα συστάσεων είναι ιδιαίτερα χρήσιμα, καθώς βοηθούν τους χρήστες να ανακαλύψουν προϊόντα και υπηρεσίες που διαφορετικά μπορεί να μην είχαν βρει μόνοι τους.

Τα συστήματα συστάσεων εκπαιδεύονται ώστε να κατανοούν τις προτιμήσεις, τις προηγούμενες αποφάσεις και τα χαρακτηριστικά των ανθρώπων και των προϊόντων χρησιμοποιώντας δεδομένα που συλλέγονται σχετικά με τις αλληλεπιδράσεις τους. Αυτά περιλαμβάνουν τις εντυπώσεις, τα κλικ, τις συμπάθειες και τις αγορές. Λόγω της ικανότητάς τους να προβλέπουν τα ενδιαφέροντα και τις επιθυμίες των καταναλωτών σε εξαιρετικά εξατομικευμένο επίπεδο, τα συστήματα συστάσεων είναι τα αγαπημένα των παρόχων περιεχομένου και προϊόντων. Μπορούν να οδηγήσουν τους καταναλωτές σχεδόν σε οποιοδήποτε προϊόν ή υπηρεσία που τους ενδιαφέρει, από βιβλία και βίντεο μέχρι μαθήματα υγείας και ρούχα.

Αν και υπάρχει ένας τεράστιος αριθμός αλγορίθμων και τεχνικών συστάσεων, οι περισσότερες εμπίπτουν στις εξής μεγάλες κατηγορίες: συνεργατικό φιλτράρισμα, φιλτράρισμα περιεχομένου και φιλτράρισμα πλαισίου.

Στην παρούσα εργασία, θα επικεντρωθούμε στο συνεργατικό φιλτράρισμα. Οι αλγόριθμοι συνεργατικού φιλτραρίσματος συνιστούν αντικείμενα (αυτό είναι το μέρος του φιλτραρίσματος) με βάση πληροφορίες προτίμησης από πολλούς χρήστες (αυτό είναι το συνεργατικό μέρος). Αυτή η προσέγγιση χρησιμοποιεί την ομοιότητα της συμπεριφοράς των προτιμήσεων των χρηστών, δεδομένης της προηγούμενης αλληλεπίδρασης μεταξύ χρηστών και αντικειμένων, οι αλγόριθμοι σύστασης μαθαίνουν να προβλέπουν τη μελλοντική αλληλεπίδραση.

Οι τεχνικές παραγοντοποίησης πινάκων (MF) αποτελούν τον πυρήνα πολλών δημοφιλών αλγορίθμων, συμπεριλαμβανομένης της ενσωμάτωσης λέξεων και της μοντελοποίησης θεμάτων, και έχουν καταστεί κυρίαρχη μεθοδολογία στο πλαίσιο της σύστασης που βασίζεται σε συνεργατικό φιλτράρισμα.

1.5 Predictive inference serving for multi-tenant GPU clusters

Στην παρούσα εργασία, αναπτύξαμε έναν χρονοπρογραμματιστή για multi-tenant GPU clusters χρησιμοποιώντας το Kubernetes (Κεφάλαιο 4) και το Collaborative Filtering (CF) (Κεφάλαιο 7). Για τη δημιουργία του χρονοπρογραμματιστή χρησιμοποιήσαμε το πλαίσιο χρονοπρογραμματισμού Kubernetes (Κεφάλαιο 4.8). Το πλαίσιο χρονοπρογραμματισμού είναι μια pluggable αρχιτεκτονική για τον χρονοπρογραμματιστή του Kubernetes. Προσθέτει ένα νέο σύνολο από plugin APIs στον υπάρχοντα χρονοπρογραμματιστή. Τα plugins μεταγλωττίζονται στον χρονοπρογραμματιστή. Επεκτείναμε τα plugins Score και PostBind προκειμένου να υλοποιήσουμε τη λογική του χρονοπρογραμματισμού μας. Το πρώτο εμπόδιο που αντιμετωπίζεται κατά τη δημιουργία ενός χρονοπρογραμματιστή για το Kubernetes είναι το γεγονός ότι το Kubernetes δεν διαφημίζει εγγενώς τους πόρους της GPU. Δηλαδή, ο μόνος ευρέως αποδεκτός τρόπος για να αξιοποιήσουμε πόρους GPU είναι με τη χρήση του NVIDIA Device plugin, το οποίο διαφημίζει συσκευές GPU στο σύνολό τους. Ως εκ τούτου, η λεπτομερής κατανομή πόρων GPU, όπως η μνήμη και οι SM (πολυεπεξεργαστές ροής), δεν αποτελεί επιλογή. Για να ξεπεράσουμε αυτό το πρόβλημα δημιουργήσαμε ένα daemonset που ταξινομεί τους κόμβους GPU (κόμβους που περιέχουν GPU) ανάλογα με τους πόρους τους. Στη συνέχεια, αποθηκεύουμε αυτά τα δεδομένα σε ένα Redis deployment το οποίο εκτελούνταν στον κύριο κόμβο, προκειμένου να βελτιστοποιήσουμε τον χρόνο των ερωτημάτων και να ελαχιστοποιήσουμε την επιβάρυνση χρονοπρογραμματισμού. Για το Collaborative Filtering (πρόβλεψη) χρειαζόμασταν μια διαδικασία για τον υπολογισμό του αναμενόμενου QPS (ερωτήματα ανά δευτερόλεπτο). Αυτή θα μπορούσε να είναι μια ξεχωριστή διαδικασία στον κώδικα που δημιουργήθηκε για τον χρονοπρογραμματιστή ή ένα ξεχωριστό container στον χρονοπρογραμματιστή, ωστόσο αποφασίσαμε ότι, δεδομένου ότι το σύνολο δεδομένων μπορεί να είναι σχετικά μεγάλο ή η διαδικασία εξαγωγής συμπερασμάτων να είναι υπολογιστικά έντονη, θα έπρεπε να μεταφέρουμε ολόκληρη τη διαδικασία σε ξεχωριστό κόμβο. Για τον σκοπό αυτό χρησιμοποιήσαμε τον Cheetara, τον δεύτερο κόμβο μας χωρίς GPU εκτός από τον master.

Το Redis (Remote Dictionary Server) είναι ένας αποθηκευτής δομών δεδομένων στη μνήμη, που χρησιμοποιείται ως κατανεμημένη, στη μνήμη βάση δεδομένων κλειδιών-τιμών, κρυφή μνήμη και διαμεσολαβητής μηνυμάτων, με προαιρετική ανθεκτικότητα. Το Redis υποστηρίζει διάφορα είδη αφηρημένων δομών δεδομένων, όπως συμβολοσειρές, λίστες, χάρτες, σύνολα, ταξινομημένα σύνολα, HyperLogLogs, bitmaps, ροές και χωρικούς δείκτες. Σε αυτό το έργο, το Redis χρησιμοποιήθηκε για να διευκολύνει την αποθήκευση προσωρινών δεδομένων σχετικά με την κατάσταση του cluster (UUIDs GPU, εκχωρημένα pods σε τμήματα των GPU κ.λπ.). Η ανάπτυξη του Redis επισημάνθηκε με έναν μαλακό περιορισμό συγγένειας, προκειμένου να ανατεθεί στον κύριο κόμβο, ώστε ο χρονοπρογραμματιστής να έχει ακόμη ταχύτερη πρόσβαση στα δεδομένα.

Προκειμένου να παρακολουθούμε τα χαρακτηριστικά (όπως τα UUIDs των τμημάτων) κάθε GPU του cluster και να τα συσχετίζουμε με τον αντίστοιχο κόμβο, χρειάστηκε να δημιουργήσουμε έναν πόρο Kubernetes που να λειτουργεί ως ενδιάμεσος μεταξύ του χρονοπρογραμματιστή και των GPUs. Για το σκοπό αυτό δημιουργήσαμε ένα daemonset. Κάθε pod του daemonset εκτελεί ένα container το οποίο πρώτα αποφασίζει αν ο κόμβος είναι σε θέση να φιλοξενήσει εφαρμογές GPU και στη συνέχεια, εκτελώντας μια απλή εφαρμογή CUDA, εξάγει τα UUIDS της GPU καθώς και μερικές μετρήσεις και τα αποθηκεύει στο Redis. Στη συνέχεια, ο εξαγωγέας παρακολουθεί την έξοδο της εφαρμογής CUDA για αλλαγές. Κάθε φορά που γίνεται επανεκκίνηση ή συντριβή, ο εξαγωγέας αντλεί αμέσως τα νέα δεδομένα και ενημερώνει το Redis.

1.5.0.1 Πρόβλεψη και βαθμολόγηση

Προκειμένου να παράγουμε συστάσεις και να βαθμολογήσουμε τους κόμβους με βάση τις προβλέψεις για τις επιδόσεις και τις παρεμβολές, δημιουργήσαμε μια ανάπτυξη με έναν μαλακό περιορισμό συγγένειας προκειμένου να την αποφορτίσουμε στον Cheetara (τον δεύτερο κόμβο μας χωρίς GPU). Η ανάπτυξη ενσωματώνει ένα σύστημα συστάσεων που χρησιμοποιεί CF και SVD (singular value decomposition) και είναι προσβάσιμη από το cluster μέσω gRPC. Αξιοποιώντας τα δεδομένα που έχουμε συλλέξει από τον χαρακτηρισμό μας (Κεφάλαιο 6) δημιουργήσαμε 2 σύνολα δεδομένων. Το πρώτο είναι ένας αραιός πίνακας $n \times m$ που χρησιμοποιείται για τον υπολογισμό του αναμενόμενου QPS (ερωτήματα ανά δευτερόλεπτο) όταν η εφαρμογή εκτελείται απομονωμένα σε ένα τμήμα με δυνατότητα GPU. Το n αντιστοιχεί στις εφαρμογές (ισοδύναμο με τους χρήστες για τις παραδοσιακές προσεγγίσεις συστημάτων συστάσεων) και το m αντιστοιχεί στις διαθέσιμες διαμορφώσεις (στοιχεία). Κάθε διαμόρφωση περιγράφεται από

1. GPU
2. τμήμα MIG (εάν υπάρχει) και

3. MPS όρια συσκευής (εάν είναι διαθέσιμα).

Όταν ένα ερώτημα φτάνει στη σύσταση, η σύσταση πρέπει να επιστρέψει ολόκληρη τη σειρά που αντιστοιχεί σε αυτή την εφαρμογή. Εάν η γραμμή έχει ελλιπείς τιμές, χρησιμοποιεί SVD για να εξάγει τα λανθάνοντα χαρακτηριστικά από τον πίνακα και στη συνέχεια υπολογίζει τις αναμενόμενες τιμές με βάση τις ομοιότητες μεταξύ των στοιχείων. Ονομάζουμε την παραπάνω διαδικασία $isolated(x, p)$, όπου x είναι η εφαρμογή, p είναι η διαμόρφωση (κατάτμηση GPU) και isolated(x, p) είναι το QPS που μπορεί να επιτύχει το x όταν εκτελείται απομονωμένα στην κατάτμηση p.

Για να βαθμολογήσουμε τον κόμβο χρειαζόμαστε μια συνάρτηση που να περιγράφει την απόσταση μεταξύ της ζητούμενης τιμής QPS (στόχος επιπέδου εξυπηρέτησης) και της αναμενόμενης τιμής. Ορίζουμε τη μετρική σφάλματος του συστήματος ως την ευκλείδεια απόσταση του ζητούμενου και του παρεχόμενου QPS, κανονικοποιημένη από το πρώτο

$$err(x, X, p) = \frac{|SLO(x) - expected\_value(x, X, p)|}{SLO(x)}$$

, όπου

$$expected\_value(x, X, p) = isolated(x, p) - total\_interference(x, X, p)$$

η αναμενόμενη_τιμή αντιστοιχεί στην αναμενόμενη QPS για την εφαρμογή x όταν είναι τοποθετημένη με το σύνολο των pods X στο διαμέρισμα p και η $SLO(x)$ αναφέρεται στην QPS που απαίτησε ο χρήστης. Για να ορίσουμε το σκορ χρειαζόμαστε κάποια συνάρτηση που μειώνεται όταν η παραπάνω απόσταση γίνεται μεγαλύτερη. Επίσης, για σταθερή απόσταση, οι θετικές τιμές πρέπει να είναι προτιμότερες σε σχέση με τις αρνητικές, αφού πρέπει να καλύψουμε τις ανάγκες των χρηστών. Για το σκοπό αυτό χρησιμοποιήσαμε δύο μέτρα ομοιότητας. Το πρώτο μέτρο ομοιότητας είναι το ακόλουθο

$$score_1(x, X, p) = \frac{1}{1 + (err(x, X, p) + 1)^2}$$

και το δεύτερο

$$score_2(x, X, p) = \frac{1}{1 + err(x, X, p)}$$

, όπου προφανώς το err(x) μπορεί να πάρει οποιαδήποτε θετική τιμή. Το $score_1$ μειώνεται πολύ πιο γρήγορα από το $score_2$ καθώς αυξάνεται το err(x) και επίσης το σύνολο των προορισμών και των δύο συναρτήσεων είναι $[0, 1]$. Στο σχήμα 8.1 μπορούμε να δούμε τις γραφικές παραστάσεις των συναρτήσεων.

1.5.0.2 Score

Έστω X το σύνολο των pods που εκτελούνται στην κατάτμηση p, συμπεριλαμβανομένου του pod που προγραμματίζεται. Ορίζουμε τα σύνολα X_{neg} και X_{pos} έτσι ώστε

$$X_{neg} = \{x | x \in X, expected\_value(x, X, p) - SLO(x) < 0\}$$

είναι το υποσύνολο των pods στο X που δεν μπορούν να καλύψουν το ζητούμενο SLO ενώ

$$X_{pos} = \{x | x \in X, expected\_value(x, X, p) - SLO(x) \geq 0\}$$

είναι το υποσύνολο του X που μπορεί να καλύψει το ζητούμενο SLO. Προφανώς τα X_{neg} και X_{pos} είναι ένα τμήμα του X, δηλαδή

$$X_{neg} \cup X_{pos} = X \ \kappa\alpha\iota \ X_{neg} \cap X_{pos} = \emptyset$$

Η βαθμολογία του τμήματος θα είναι ένας σταθμισμένος μέσος όρος των βαθμολογιών όλων των pods. Αυτές οι βαθμολογίες πρέπει επίσης να είναι μεταξύ 0 και 100, καθώς αυτές είναι οι μόνες αποδεκτές βαθμολογίες από το πλαίσιο χρονοπρογραμματισμού.

Στην αρνητική περίπτωση, η βαθμολογία είναι πιο σημαντική από ό,τι στη θετική περίπτωση. Αυτό οφείλεται στο γεγονός ότι είναι προτιμότερο να παρέχονται περισσότεροι πόροι από αυτούς που απαιτεί ο χρήστης παρά να διακυβεύεται το αίτημα του χρήστη. Για το σκοπό αυτό, στην αρνητική περίπτωση προτιμήσαμε το $score_1(x), x \geq 0$ το οποίο μειώνεται γρήγορα καθώς αυξάνεται η απόσταση $err(x)$, ενώ στη θετική περίπτωση προτιμήσαμε το $score_2(x), x \geq 0$ το οποίο μειώνεται πιο αργά καθώς αυξάνεται το $err(x)$.

Η τελική βαθμολογία για το διαμέρισμα i υπολογίζεται ως γραμμικός συνδυασμός των παραπάνω:

$$score(X,p) = 100 \times (1 - util(p)) \times (k \times \frac{1}{|X_{neg}|} \times \sum_{x \in X_{neg}} score_1(x, X, p) + (1-k) \times \frac{1}{|X_{pos}|} \times \sum_{x \in X_{pos}} score_2(x, X, p))$$

όπου $util(i)$ είναι η στιγμιαία τιμή του τμήματος για τη χρήση της GPU που επιστρέφεται από τον Prometheus και $k = \frac{|X_{neg}|}{|X|}$ είναι μια παράμετρος που βρίσκεται στο διάστημα $[0, 1]$ και ορίζει τη βαρύτητα των αρνητικών περιπτώσεων στη συνολική διαδικασία. Κατά προτίμηση, το k πρέπει να είναι μεγαλύτερο από 50%, δεδομένου ότι οι αρνητικές περιπτώσεις έχουν μεγαλύτερη σημασία, όπως εξηγήθηκε παραπάνω. Τέλος, το σύνολο προορισμού της συνάρτησης βαθμολογίας είναι το διάστημα [0,100].

Εάν $X_{neg} = \emptyset$, τότε ο όρος που περιέχει το $score_1$ παραλείπεται, διαφορετικά εάν $X_{pos} = \emptyset$, τότε ο όρος που περιέχει το $score_2$ παραλείπεται. Στην παραπάνω διαδικασία περιγράψαμε τον τρόπο με τον οποίο μπορούμε να επιλέξουμε μια διαμέριση που ταιριάζει καλύτερα στις ανάγκες μας. Ωστόσο, το ScorePlugin μας δίνει τη δυνατότητα να βαθμολογήσουμε μόνο ολόκληρο τον κόμβο ο οποίος μπορεί να περιέχει πολλαπλά partitions (π.χ. MIG). Έστω P το σύνολο όλων των διαθέσιμων κατατμήσεων GPU στον κόμβο και X_p ένα σύνολο pods που εκτελούνται στην κατάτμηση p. Βαθμολογούμε τον κόμβο n με τη μέγιστη βαθμολογία μεταξύ των βαθμολογιών των κατατμήσεων p

$$final\_score(n) = \max_{p \epsilon P} score(X_p, p)$$

όπου

$$\bigcup_{p \in P} X_p = X$$

Αν ο κόμβος δεν φιλοξενεί άλλες εφαρμογές GPU τη στιγμή που φτάνει το pod, τότε αναδιαμορφώνουμε την GPU έτσι ώστε να μεγιστοποιείται το σκορ (αν $\operatorname{argmax}_p score(X, p) \neq current\ configuration$). Η διαδικασία της αναδιαμόρφωσης διαρκεί περίπου 5 δευτερόλεπτα, πράγμα ασήμαντο σε σύγκριση με τη διάρκεια των φορτίων εργασίας που διαρκούν μερικά λεπτά. Επίσης, όταν η GPU δεν είναι άδεια, αποφασίσαμε να μην την αναδιαμορφώσουμε, δεδομένου ότι τα pods που εκτελούνται θα διακόπτονταν και επομένως θα παραβίαζαν τους στόχους τους.

1.5.0.3 PostBind και CUDA_VISIBLE_DEVICES

Στην προηγούμενη ενότητα περιγράψαμε πώς μπορούμε να επιλέξουμε έναν συγκεκριμένο κόμβο, αλλά όχι πώς να δεσμεύσουμε το pod στο επιθυμητό διαμέρισμα. Για το σκοπό αυτό χρησιμοποιήσαμε μια μεταβλητή περιβάλλοντος που χρησιμοποιείται από τους προγραμματιστές του CUDA για τον έλεγχο της ορατότητας της GPU για τις εφαρμογές CUDA. Η CUDA_VISIBLE_DEVICES είναι μια μεταβλητή περιβάλλοντος που χρησιμοποιείται για να καθορίσει ποιες GPU της NVIDIA θα πρέπει να χρησιμοποιούνται από μια εφαρμογή με δυνατότητα CUDA. Κατά την εκτέλεση ενός προγράμματος CUDA σε ένα σύστημα με πολλαπλές GPU, η CUDA_VISIBLE_DEVICES μπορεί να χρησιμοποιηθεί για να ελέγξει ποιες GPU είναι ορατές στο πρόγραμμα. Από προεπιλογή, όλες οι GPU είναι ορατές, αλλά αυτή η μεταβλητή μπορεί να οριστεί σε μια λίστα με διαχωρισμένα με κόμμα αναγνωριστικά συσκευών GPU για να περιοριστούν οι ορατές συσκευές. Για παράδειγμα, η ρύθμιση CUDA_VISIBLE_DEVICES=0,1 θα περιορίσει το πρόγραμμα να βλέπει μόνο την πρώτη και τη δεύτερη GPU. Αυτό μπορεί να είναι χρήσιμο για διάφορους λόγους, όπως ο περιορισμός των πόρων που χρησιμοποιούνται από ένα πρόγραμμα ή η δυνατότητα ταυτόχρονης εκτέλεσης πολλαπλών προγραμμάτων σε διαφορετικές GPU.

Το πρόσθετο PostBind καλείται μετά την επιτυχή δέσμευση ενός Pod. Προκειμένου το pod να δεσμευτεί σε ένα συγκεκριμένο διαμέρισμα δημιουργήσαμε ένα κενό ConfigMap και το προσαρτήσαμε στο περιβάλλον του pod. Στο πλαίσιο του χρονοπρογραμματιστή, κρατάμε μια δομή δεδομένων στη μνήμη που περιγράφει το βέλτιστο διαμέρισμα για κάθε pod και κάθε κόμβο και μετά την ολοκλήρωση του χρονοπρογραμματισμού συμπληρώνουμε το ConfigMap με τη μεταβλητή περιβάλλοντος CUDA_VISIBLE_DEVICES, με μια τιμή ίση με το UUID του επιλεγμένου τμήματος.

Ακολουθήσαμε ακριβώς την ίδια λογική για τον υπολογισμό και την προσθήκη των βέλτιστων ορίων συσκευών MPS για τη μνήμη και την υπολογιστική χωρητικότητα χρησιμοποιώντας
CUDA_MPS_PINNED_DEVICE_MEM_LIMIT και
CUDA_MPS_ACTIVE_THREAD_PERCENTAGE μεταβλητές περιβάλλοντος αντίστοιχα.

1.5.1 Αξιολόγηση

Για να αξιολογήσουμε τον χρονοπρογραμματιστή μας, εκτελέσαμε μια σειρά πειραμάτων. Κάθε πείραμα είχε ως σκοπό να απεικονίσει μια διαφορετική πτυχή της απόδοσης του χρονοπρογραμματιστή. Ένα πείραμα αποτελείται από ένα σύνολο φόρτων εργασίας (προ-εκπαιδευμένο μοντέλο, backend, σύνολο δεδομένων, σενάριο, αριθμός ερωτημάτων κ.λπ.), τους στόχους (SLO), ένα μέγεθος παρτίδας καθώς και μια κατανομή που περιγράφει τις αφίξεις των εφαρμογών με την πάροδο του χρόνου. Σε κάθε πείραμα οι διαθέσιμοι χρονοπρογραμματιστές τροφοδοτούνται με τους ίδιους ακριβώς φόρτους εργασίας, το μέγεθος της παρτίδας και τις αφίξεις. Στα επόμενα πειράματα, οι φόρτοι εργασίας αναπτύσσονται σε παρτίδες των 8 και εντός των παρτίδων κάθε φόρτος εργασίας έχει μια τυχαία καθυστέρηση μερικών δευτερολέπτων για να προστεθεί ένας απρόβλεπτος παράγοντας. Αποφασίσαμε το μέγεθος της παρτίδας να είναι 8, επειδή διαθέτουμε μόνο 2 GPU στο cluster και αν το μέγεθος της παρτίδας γινόταν μεγαλύτερο, αυτό θα οδηγούσε σε αποτυχίες λόγω ανεπαρκών πόρων και διαφορετικά το cluster θα υπολειτουργούσε. Εκτελέσαμε δύο σειρές πειραμάτων. Στην πρώτη σειρά πειραμάτων στείλαμε 20 pods σε διάστημα περίπου δεκαπέντε λεπτών. Στη δεύτερη, αποφασίσαμε να καταπονήσουμε τους χρονοπρογραμματιστές με τον διπλάσιο αριθμό pods (40) στο ίδιο χρονικό διάστημα, προκειμένου να εξετάσουμε τη συμπεριφορά του χρονοπρογραμματιστή υπό υψηλή πίεση. Και τα δύο πειράματα επαναλήφθηκαν δύο φορές. Την πρώτη φορά επιλέξαμε τα SLOs για κάθε φόρτο εργασίας να είναι στο διάστημα $[0.8 \times SLO_4, 1.2 \times SLO_4]$ όπου SLO_4 ισούται με την αναμενόμενη τιμή του QPS όταν ο φόρτος εργασίας εκτελείται απομονωμένος στο $\frac{1}{4}$ της GPU (στην περίπτωση του A30 1 από τα 4 partitions, στην περίπτωση του V100 περιορισμένος στο 25% των πόρων της GPU με χρήση MPS) από μόνος του. Δηλαδή, επιλέξαμε χαμηλές SLOs. Στη δεύτερη περίπτωση, επιλέξαμε SLOs στο διάστημα $[0.8 \times SLO_1, 1.2 \times SLO_1]$, όπου SLO_1 ισούται με την αναμενόμενη τιμή του QPS όταν ο φόρτος εργασίας εκτελείται μεμονωμένα σε ολόκληρη την GPU. Επιλέξαμε να χωρίσουμε τα πειράματα με αυτόν τον τρόπο, έτσι ώστε να είμαστε σε θέση να εξετάσουμε την ικανότητα του χρονοπρογραμματιστή μας να καταμερίζει τις GPU με δυνατότητα MIG, καθώς και να εξετάσουμε την ικανότητά του να χειρίζεται εκρήξεις και να ανταποκρίνεται στις ανάγκες των χρηστών καθώς αυτές γίνονται όλο και πιο απαιτητικές.

Σε κάθε πείραμα μετράμε τις ακόλουθες μετρικές QoS και χρήσης πόρων GPU χρησιμοποιώντας το API του Kubernetes και τον μηχανισμό παρακολούθησης GPU.

- ☐ Μετρικές QoS
  1. Διάρκεια εκτέλεσης
  2. Πλήθος αποτυχιών
  3. Αριθμός παραβιάσεων SLO
- ☐ Μετρικές χρήσης πόρων GPU
  1. Μέση χρήση GPU (Αναλογία του χρόνου που η μηχανή γραφικών είναι ενεργή (σε %))
  2. Μέση κατανάλωση ενέργειας (W)
  3. Μέση κατανάλωση ενέργειας (J)

1.6 Schedulers

Σε αυτή την ενότητα αναλύουμε τα κύρια χαρακτηριστικά των χρονοπρογραμματιστών που δημιουργήσαμε προκειμένου να συγκρίνουμε τα αποτελέσματά μας.

1.6.1 Min

Πρώτον, προσομοιώσαμε έναν άπληστο χρονοπρογραμματιστή ο οποίος κατανέμει τους λιγότερους δυνατούς πόρους σε κάθε εισερχόμενο φόρτο εργασίας. Αυτό το επιτυγχάνει με την κατάτμηση όλων των GPU με δυνατότητα MIG σε όσο το δυνατόν περισσότερα διαμερίσματα και την κατανομή ενός σε κάθε εφαρμογή. Επιπλέον, για κάθε GPU χωρίς δυνατότητα MIG χρησιμοποίησε τη δυνατότητα MPS (multi process service) για να περιορίσει τους πόρους κάθε εφαρμογής στο 25%. Ο λόγος για τον οποίο θεωρήσαμε ότι το 25% είναι αρκετά μικρό είναι επειδή παρατηρήσαμε ότι αυστηρότερα όρια προκαλούσαν υψηλό ποσοστό αποτυχιών λόγω ανεπαρκούς μνήμης ή/και υπολογιστικής χωρητικότητας. Επίσης, ο χρονοπρογραμματιστής χρησιμοποίησε το NVIDIA Device Plugin προκειμένου να δεσμεύσει τις εφαρμογές σε συσκευές GPU, πράγμα που σημαίνει ότι, όπως εξηγείται στο 4.7.3, οι εκτεταμένοι πόροι του cluster δεν υποστηρίζουν τον διαμοιρασμό μεταξύ των εφαρμογών και επομένως, όταν όλες οι GPU είναι απασχολημένες στο cluster, οι εισερχόμενες εφαρμογές πρέπει να περιμένουν σε μια ουρά πριν ξεκινήσουν την εκτέλεσή τους. Χρησιμοποιήσαμε αυτή τη λογική χρονοπρογραμματισμού προκειμένου να συγκρίνουμε τον χρονοπρογραμματιστή μας με έναν που εκμεταλλεύεται τη γνωστή πρακτική της χρήσης του NVIDIA Device Plugin. Χρησιμοποιήσαμε αυτή τη λογική χρονοπρογραμματισμού προκειμένου να συγκρίνουμε τον χρονοπρογραμματιστή μας με έναν που εκμεταλλεύεται άπληστα τις τεχνολογίες MPS και MIG για να μοιράζεται τους πόρους GPU μεταξύ των εφαρμογών.

1.6.2 Max

Ο επόμενος χρονοπρογραμματιστής που χρησιμοποιήθηκε για αξιολόγηση είναι ένας άπληστος χρονοπρογραμματιστής ο οποίος κατανέμει τους περισσότερους δυνατούς πόρους σε κάθε εισερχόμενο φόρτο εργασίας. Σε αυτό το πείραμα όλες οι GPU χρησιμοποιούνται στο σύνολό τους, πράγμα που σημαίνει ότι οι GPU με δυνατότητα MIG δεν διαμερίζονται και δεν χρησιμοποιείται το MPS. Επίσης, παρόμοια με τον προηγούμενο χρονοπρογραμματιστή, ο χρονοπρογραμματιστής χρησιμοποιεί το NVIDIA Device Plugin προκειμένου να δεσμεύει εφαρμογές σε συσκευές GPU. Χρησιμοποιήσαμε αυτή τη λογική χρονοπρογραμματισμού προκειμένου να συγκρίνουμε τον χρονοπρογραμματιστή μας με έναν που εκμεταλλεύεται τη γνωστή πρακτική της χρήσης του NVIDIA Device Plugin.

1.6.3 Round Robin

Στην τελευταία περίπτωση, αποφασίσαμε να προσομοιώσουμε μια λογική χρονοπρογραμματισμού, η οποία, αφού χωρίσει όλες τις GPUs στα μικρότερα δυνατά κομμάτια, προγραμματίζει pods σε κάθε ένα από τα κομμάτια με τρόπο round robin, υποστηρίζοντας επίσης την παράθεση μεταξύ εφαρμογών που χρησιμοποιούν τεχνολογίες MIG και MPS. Για παράδειγμα, εάν προγραμματίσει δύο εφαρμογές στην ίδια κατάτμηση, θα εφαρμόσει όρια MPS 50% σε κάθε μία από αυτές, εάν προγραμματίσει τρεις εφαρμογές, θα εφαρμόσει όρια MPS 33% σε κάθε μία από αυτές κ.λπ. Ο λόγος για τον οποίο επιλέξαμε να προσομοιώσουμε αυτόν τον χρονοπρογραμματιστή είναι για να έχουμε ένα βασικό κριτήριο για να αξιολογήσουμε την ικανότητα του χρονοπρογραμματιστή μας να αντιμετωπίζει τις παρεμβολές σε σύγκριση με έναν χρονοπρογραμματιστή χωρίς παρεμβολές.

Τα αποτελέσματα των πειραμάτων μπορούν να βρεθούν στο 9 του αγγλικού κειμένου.

Chapter 2

Introduction

2.1 Cloud Computing and Kubernetes

Cloud computing and Kubernetes are two powerful technologies that have revolutionized the way businesses manage and deploy their IT infrastructure.

Cloud computing enables users to access virtualized computing resources such as servers, storage, applications, and services, over the internet. Cloud computing offers a range of services, including Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS), that allow users to scale their resources up or down as needed, pay only for what they use, and access their data and applications from anywhere, at any time, using any device with an internet connection. Cloud computing offers several advantages over traditional IT infrastructure, including reduced costs, increased flexibility and scalability, improved accessibility, and enhanced security and reliability.

Kubernetes, on the other hand, is a container orchestration platform that simplifies the management and scaling of containerized applications. Kubernetes offers several advantages over traditional application deployment methods, including improved resource utilization, faster deployment times, and enhanced flexibility and scalability. Kubernetes automates the deployment, scaling, and management of containerized applications, allowing users to focus on developing their applications rather than managing the underlying infrastructure. Additionally, Kubernetes offers a range of features, including load balancing, automatic failover, and self-healing, that ensure high availability and reliability.

Combining cloud computing and Kubernetes offers a highly scalable and flexible solution for managing and deploying IT infrastructure. Kubernetes can be used to manage containerized applications running on cloud computing platforms, allowing users to take advantage of the scalability and flexibility of the cloud while also benefiting from the automation and management features provided by Kubernetes. The combination of cloud computing and Kubernetes enables users to deploy and manage applications and services in a highly scalable and flexible way, ensuring high availability, scalability, and reliability.

In summary, cloud computing and Kubernetes are two powerful technologies that have transformed the way businesses manage and deploy their IT infrastructure. Their combination offers a highly scalable and flexible solution for managing and deploying applications and services, enabling businesses to improve their operational efficiency, reduce costs, and increase agility.

2.2 Accelerators in Kubernetes

The use of Graphics Processing Units (GPUs) and other accelerators on Kubernetes has become increasingly popular for processing and analyzing large amounts of data. These specialized computing resources offer significant performance benefits over traditional CPU-based systems. However, there are several difficulties and challenges associated with using GPUs and other accelerators on Kubernetes.

One of the main challenges is the inability to share these specialized resources among multiple workloads. Unlike CPU-based systems, GPUs and other accelerators are typically not designed to be shared among multiple applications simultaneously. This can lead to contention for resources, particularly in multi-tenant environments where multiple users and workloads are competing for access to these resources.

To address this challenge, organizations need to carefully manage and allocate these specialized resources to ensure that they are used efficiently and effectively. This can include implementing policies and procedures to prioritize access to resources based on workload requirements, such as workload type, resource utilization, and performance requirements.

Another challenge is ensuring compatibility and interoperability between heterogeneous hardware and software components. Kubernetes supports several GPU and accelerator types, each with their own specifications and requirements. Ensuring compatibility and interoperability between heterogeneous components can be challenging, particularly when integrating third-party software and hardware components.

In addition, deploying and managing GPU-based workloads and other accelerators can be challenging due to their specialized nature. Applications and services that use these resources may require specialized configurations and optimizations to ensure maximum performance and efficiency. Debugging and troubleshooting issues related to GPU-based workloads can also be challenging, requiring specialized knowledge and expertise.

In summary, while GPUs and other accelerators offer significant performance benefits, there are several difficulties and challenges associated with using them on Kubernetes. These challenges include the inability to share resources among multiple workloads, ensuring compatibility and interoperability between components, and specialized deployment and management requirements. Addressing these challenges requires careful planning and management to ensure optimal performance and efficient use of resources.

2.3 Overview

In this work, we design a novel GPU scheduler based on predictions of AI models regarding the performance of applications on multiple different GPU configurations as well as the expected interference produced due to the collocation of different applications on the same configuration. In order to decide the optimal configuration for each application we also exploit real-time GPU metrics retrieved by various monitoring systems. We identify the inefficiency of the state-of-the-art Kubernetes GPU schedulers concerning the quality of service (QoS) and resource utilization. We show that our scheduler, for the majority of workloads and scheduling scenarios, can achieve lower pending and execution time on average as well as better resource utilization while it ensures versatility and sticks to the users' demands without incurring high overhead or energy consumption.

In chapter 3, we analyze other GPU scheduling approaches that have been proposed so far. In chapter 4, we discuss about the basic concepts of Kubernetes, the underlying container orchestrator. In chapter 5, we present our experimental infrastructure, the GPU monitoring system and the suite used for workload execution and benchmarking. In chapter 6, we present and discuss the results of the characterization of workloads (inference engines) we used for our experiments and the creation of the datasets fed in our AI models. In chapter 7, we provide insight on our models and explain the techniques used to provide suggestions in depth. In chapter 8, we present our GPU scheduling system. In chapter 9, we evaluate our proposed framework and compare it to multiple well-known options for scheduling GPU-based workloads on the Kubernetes orchestrator, across different scenarios and workloads. Finally, in chapter 10, we summarize our work and propose fields with, potentially, high interest for future work in related research scopes.

Chapter 3

Related Work

3.1 Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters

An increasing amount of computing is performed in the cloud, primarily due to cost benefits for both the end-users and the operators of datacenters (DC) that host cloud services [5]. Large-scale providers such as Amazon EC2 [6], Microsoft Windows Azure [7], Rackspace[8] and Google Compute Engine [9] host tens of thousands of applications on a daily basis. Several companies also organize their IT infrastructure as private clouds, using management systems such as VMware vSphere [10] or Citrix XenServer [11]. The operator of a cloud service must schedule the stream of incoming applications on available servers in a manner that achieves both fast execution (user's goal) and high resource efficiency (operator's goal), enabling better scaling at low cost. This scheduling problem is particularly difficult as cloud services must accommodate a diverse set of workloads in terms of resource and performance requirements [5]. Moreover, the operator often has no a priori knowledge of workload characteristics. This work, focuses on two basic challenges that complicate scheduling in large-scale DCs: hardware platform heterogeneity and workload interference. Heterogeneity occurs because servers are gradually provisioned and replaced over the typical 15-year lifetime of a DC [5] [12] [13] [14]. At any point in time, a DC may host 3-5 server generations with a few hardware configurations per generation, in terms of the specific speeds and capacities of the processor, memory, storage and networking subsystems. Hence, it is common to have 10 to 40 configurations throughout the DC. Ignoring heterogeneity can lead to significant inefficiencies, as some workloads are sensitive to hardware configurations. A heterogeneity-oblivious scheduler can slow applications down by 22% on average, with some running nearly 2x slower (see Section 4 for methodology). This is not only suboptimal from the user's perspective, but also for the DC operator as workloads occupy servers for significantly longer. Interference is the result of co-scheduling multiple workloads on a single server to increase utilization and achieve better cost efficiency. By co-locating applications a given number of servers can host a larger set of workloads (better scalability). Alternatively, by packing workloads in a small number of servers when the overall load is low, the rest of the servers can be turned off to save energy. The latter is needed because modern servers are not energy proportional and consume a large fraction of peak power even at low utilization [15] [5] [16] [17]. Co-scheduled applications may interfere negatively even if they run on different processor cores because they share caches, memory channels, storage and networking devices [18] [19] [20]. It is shown that an interference-oblivious scheduler will slow workloads down by 34% on average, with some running more than 2x slower. Again, this is undesirable for both users and operators. Finally, a baseline scheduler that is both interference and heterogeneity-oblivious and schedules applications to least-loaded servers is even worse (48% average slowdown), causing some workloads to crash due to resource exhaustion on the server. Previous work has showcased the potential of heterogeneity and interference-aware scheduling [13] [19]. However, techniques that rely on detailed application characterization cannot scale to large DCs that receive tens of thousands of potentially unknown workloads every day [21]. Most cloud management systems have some notion of contention or interference-awareness [22] [20] [23] [24]. However, they either use empirical rules for interference management or assume long-running workloads (e.g., online services), whose repeated behavior can be progressively modeled. This work, targets both heterogeneity and interference and assumes no a priori analysis of the application. Instead, it leverages information the system already has about the large number of applications it has previously seen. Paragon, is an online and scalable datacenter scheduler that is heterogeneity and interference-aware. The key feature of Paragon is its ability to quickly and accurately classify an unknown application with respect to heterogeneity (which server configurations it will perform best on) and interference (how much interference it will cause to co-scheduled applications and how much interference it can tolerate itself in multiple shared resources). Paragon's classification engine exploits existing data from previously scheduled applications and offline training and requires only a minimal signal about a new workload. Specifically, it is organized as a low-overhead recommendation system similar to the one deployed for the Netflix Challenge [25], but instead of discovering similarities in users' movie preferences, it finds similarities in applications' preferences with respect to heterogeneity and interference. It uses singular value decomposition to perform collaborative filtering and identify similarities between incoming and previously scheduled workloads. Once an incoming application is classified, a greedy scheduler assigns it to the server that is the best possible match in terms of platform and minimum negative interference between all co-scheduled workloads. Even though the final step is greedy, the high accuracy of classification leads to schedules that satisfy both user requirements (fast execution time) and operator requirements (efficient resource use). Moreover, since classification is based on robust analytical methods and not merely empirical observation, the paper provides strong guarantees on its accuracy and strict bounds on its overheads.

3.2 MISO: Exploiting Multi-Instance GPU Capability on Multi-Tenant Systems for Machine Learning

GPU technology has been improving at an expedited pace in terms of size and performance, empowering HPC and AI/ML researchers to advance the scientific discovery process. However, this also leads to inefficient resource usage, as most GPU workloads, including complicated AI/ML models, are not able to utilize the GPU resources to their fullest extent – encouraging support for GPU multi-tenancy. In this paper MISO[26] is proposed. MISO is a technique to exploit the Multi-Instance GPU (MIG) capability on the latest NVIDIA datacenter GPUs (e.g., A30, A100, H100) to dynamically partition GPU resources among co-located jobs. MISO's key insight is to use the lightweight, more flexible Multi-Process Service (MPS) capability to predict the best MIG partition allocation for different jobs, without incurring the overhead of implementing them during exploration.

Recent advancement in GPU technology has enabled HPC and AI researchers to leverage GPU computing capabilities for a wide variety of critical science missions, including training of compute-intensive neural network models [27][[28] [29] [30]. While these advances have expedited the scientific discovery process, efficient resource utilization of the powerful GPUs remains a key bottleneck. With innovative progress in computing technology, GPU vendors are making individual GPUs bigger and faster – where an individual GPU can now deliver more than 300 TeraFLOPS of performance and is on the path to becoming a supercomputer of the past by itself [31] [32]. This trend has served the AI/ML models well since the computing requirements of these models are increasing at a rapid pace [33] [34] [35]. Unfortunately, as the paper's experimental characterization and previous works [36] [37] [38] [39] [40] have shown, even these models are not able to fully utilize the GPU computing resources, because various workloads have different resource bottlenecks and performance sensitivity to different resources. Therefore, the "one-size-fits-all" approach of making a single GPU more powerful is not optimal for all workloads and leads to inefficient resource utilization. Recognizing and motivated by these challenges, GPU vendors have recently started offering native GPU resource partitioning capabilities to enable GPU workload co-location [41] [42]. These capabilities allow jobs to share the GPU resources concurrently and, thereby, reduce the cloud computing cost, reduce the long job queue wait time on HPC clusters, and potentially reduce the average job completion time (queue wait time + execution time). While promising, efficiently leveraging GPU partitioning is challenging because configuring a GPU to partition the resources optimally among co-located workloads is

1. cumbersome due to various practical partitioning constraints,
2. prohibitively time-consuming during the exploration process of finding a performance-efficient partition, and
3. incurs overhead.

Therefore, the goal of this paper is to provide a novel method that automatically and quickly partitions GPU resources to achieve overall higher performance. Solutions in this space are expected to become increasingly critical as HPC centers are beginning to deploy modern GPUs with explicit resource partitioning abilities. For example, the NVIDIA A100 GPUs, which have MIG technology support, are a part of many cloud computing offerings, industrial research computing clusters, and academic HPC centers [43] [44] [45] [46]. However currently, we do not have the tools to leverage MIG technology to effectively utilize MIG capabilities for faster execution and higher throughput, and thereby, reducing the cost of renting GPU resources or operating HPC clusters.

3.3 Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads

With widespread advances in machine learning, a number of large enterprises are beginning to incorporate machine learning models across a number of products. These models are typically trained on shared, multi-tenant GPU clusters. Similar to existing cluster computing workloads, scheduling frameworks aim to provide features like high efficiency, resource isolation, fair sharing across users, etc. However Deep Neural Network (DNN) based workloads, predominantly trained on GPUs, differ in two significant ways from traditional big data analytics workloads. First, from a cluster utilization perspective, GPUs represent a monolithic resource that cannot be shared at a fine granularity across users. Second, from a workload perspective, deep learning frameworks require gang scheduling reducing the flexibility of scheduling and making the jobs themselves inelastic to failures at runtime.

This paper [47] studies two main aspects of how locality-aware scheduling affects performance and utilization. First, it studies how waiting for locality constraints can influence queuing delays before training jobs are run. Training jobs need to be gang scheduled, as hyper-parameters are picked for specific GPU count configurations. Given that training jobs take a long time to run, and greater locality improves performance due to the availability of faster interconnects for parallel training [48], the scheduler in Philly waits for appropriate availability of GPUs before beginning to run the training job. Our study shows that as one might expect, relaxing locality constraints reduces queueing delays, especially for jobs that use many GPUs – our emphasis here is not on presenting this as a new insight, but instead on highlighting this using real-world data from production clusters.

Next, it studies how locality-aware scheduling can affect the GPU utilization for distributed training jobs. Even though most GPUs within a cluster are allocated to users, thus suggesting high cluster utilization, this metric alone is misleading. It is shown that the hardware utilization of GPUs in use is only around 52% on average. Two reasons are investigated which contribute to low GPU utilization:

1. the distribution of individual jobs across servers, ignoring locality constraints, increases synchronization overheads, and
2. the collocation or packing of different jobs on same server leads to interference due to contention for shared resources.

Finally, it looks at why jobs might fail to complete successfully and offer a detailed characterization of the causes for such failures in our clusters. Around 30% of jobs are killed or finish unsuccessfully due to failures.

3.4 KubeShare: A Framework to Manage GPUs as First-Class and Shared Resources in Container Cloud

Container has emerged as a new technology in clouds to replace virtual machines (VM) for distributed applications deployment and operation. With the increasing number of new cloud-focused applications, such as deep learning and high performance applications, started to reply on the high computing throughput of GPUs, efficiently supporting GPU in container cloud becomes essential. While GPU virtualization has been extensively studied for VM, limited work has been done for containers. One of the key challenges is the lack of support for GPU sharing between multiple concurrent containers. This limitation leads to low resource utilization when a GPU device cannot be fully utilized by a single application due to the burstiness of GPU workload and the limited memory bandwidth. To overcome this issue, KubeShare [49] was designed and implemented, which extends Kubernetes to enable GPU sharing with fine-grained allocation. KubeShare is the first solution for Kubernetes to make GPU device as a first class resource for scheduling and allocations.

The emergence of containers has revolutionized cloud computing. Due to negligible runtime overhead and much higher deployment density on a physical machine, containers are perceived as a lightweight replacement of virtual machines (VMs) for resource allocation and service deployment. The agility of containers combining with microservice-style software architecture further advances the practice of DevOps [50] to achieve continuous software delivery, productivity, automation, cost saving, in the software delivery process. Therefore, containerization has been increasingly adapted and supported in cloud platform. One of the industry leading solution for containerization is Docker [51], and it has been used by many of the Internet's most predominate systems today [52]. As increasing number of applications are deployed and architected based on containers, the need of an orchestration systems for scheduling, deploying, updating and scaling such container-based applications becomes crucial. Many container-management systems have been developed by industry and open source community, including YARN [53], Mesos [54], Borg [52], etc. But Kubernetes [55] is the most popular one among all by far. Kubernetes is developed and used by Google to power their cloud container service. Kubernetes not only provides powerful tools for developer to manage loosely-coupled and stateless Docker containers without having to interact with the underlying infrastructure, but also has a highly configurable and extensible architecture to support custom cluster operators, including load balancing, container replication, rolling update, volume and network management, etc. While Kubernetes has the strength to support container management, the only computing resources that can be natively recognized and allocated by Kubernetes are the CPU and memory. To attach any other custom devices to a container, including GPU, high-performance NICs, FPGA, a device plugin [56] [51] [57] must be developed and installed following the framework defined by Kubernetes to perform vendor specific initialization and setup for the devices. The device plugin framework successfully separates the vendor-specific code from Kubernetes for better system maintenance and extensibility, but it does not allow resource sharing or fractional allocation on custom devices. This limitation inevitably leads to lower resource utilization when a custom device cannot be fully utilized by a single application or container. The problem is further aggravated over the past decade by the demand surge and the growing price of high performance computing devices, like GPU. With the growing interests from High Performance Computing (HPC) community for container-based computing [58] [59] [60], the problem has drawn attentions from both research and industrial communities in recent years [60] [61] [62]. In this work, KubeShare is presented, which extends

Kubernetes to support GPU sharing with fine-grained allocation and first-class resource management. A first-class resource means that the resource entity can be explicitly identified and selected by both the resource manager and the users. As explained in the paper, it is an essential property to address the performance interference problem in shared resource environment. It is a non-trivial task due to the lack of proper resource description, allocation policy, and architecture support in the existing Kubernetes framework. Until today, only a few attempts [63] [64] [61] have been made recently to support GPU sharing in Kubernetes, but none of them treats GPU as first-class schedulable entities. As a result, their GPU throughput and utilization can easily suffer from resource fragmentation and performance interference problems. In contrast, KubeShare allows users to specify locality constraints on their allocated GPUs, so that a GPU can be shared among containers with less resource contention.

Chapter 4

The Kubernetes Orchestrator

In this chapter we explain why Kubernetes, a container orchestration system, is useful and analyze some of its basic concepts (e.g virtualization, containers[65] etc).

4.1 Virtualization

Virtualization is the process of creating a simulated computing environment that's abstracted from the physical computing hardware—essentially a computer-generated computer. Virtualization allows you to create multiple, virtual computing instances from the hardware and software components of a single machine. Those instances could be a computer in the traditional sense or a storage repository, application, server, or networking configuration. The software that enables virtualization is called a hypervisor. It's a lightweight software layer that sits between the physical hardware and the virtualized environments and allows multiple operating systems (OS) to run in tandem on the same hardware. The hypervisor is the middleman that pulls resources from the raw materials of your infrastructure and directs them to the various computing instances.

4.2 Virtual Machines

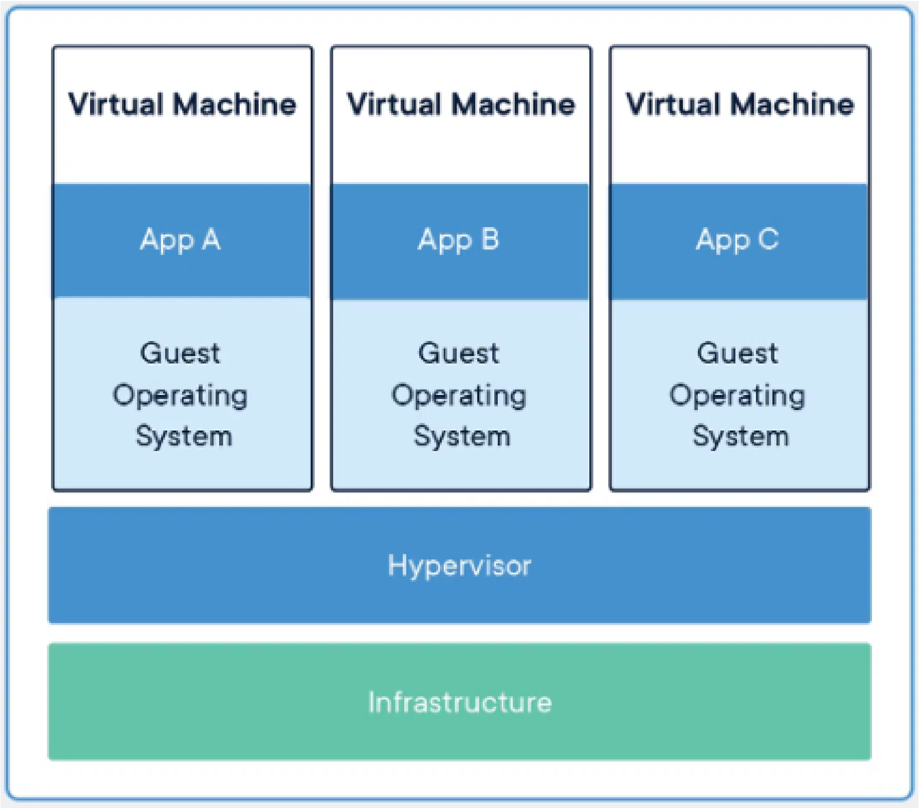
The computer-generated computers that virtualization makes possible are known as virtual machines (VMs)—separate computers running on hardware that is actually contained in one physical computer. Each VM requires its own OS as shown in 4.1a. The OS and any applications running on an individual VM share hardware resources from a single host server, or from a pool of host servers. Thanks to the hypervisor, the hardware resources are virtualized and each VM is isolated from its neighbors. Since the advent of affordable virtualization technology and cloud computing services, IT departments large and small have embraced VMs as a way to lower costs and increase efficiencies. VMs, however, can take up a lot of system resources. Each VM runs not just a full copy of an OS, but a virtual copy of all the hardware that the operating system needs to run. It's why VMs are sometimes associated with the term "monolithic"—they're single, all-in-one units commonly used to run applications built as single, large files. This quickly adds up to a lot of RAM and CPU cycles. They're still economical compared to running separate actual computers, but for some use cases, particularly applications, it can be overkill, which led to the development of containers.

4.3 Containers and Docker

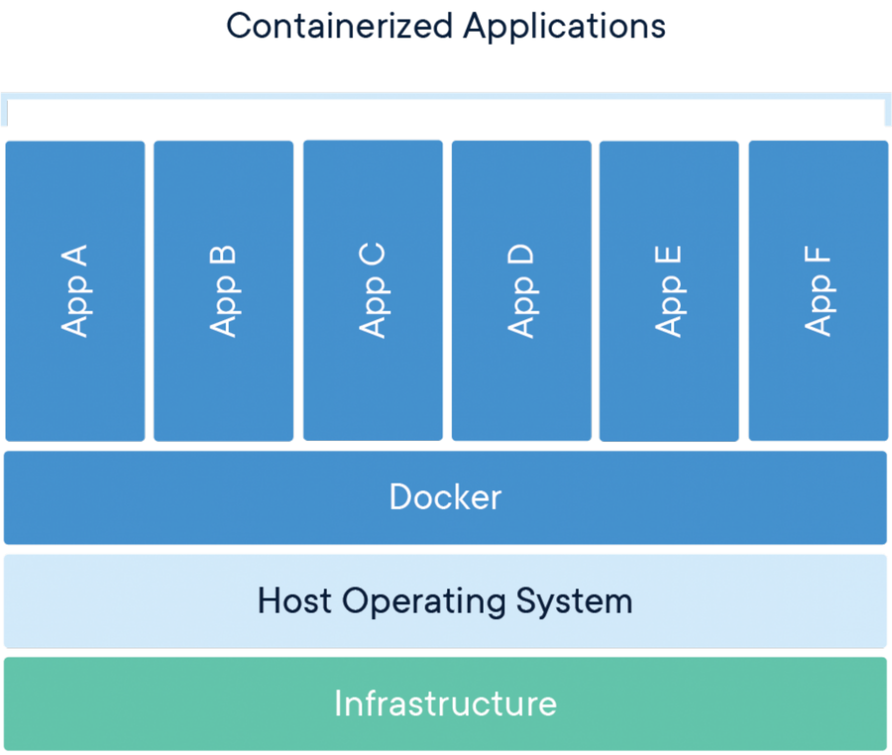
With containers, instead of virtualizing the underlying computer like a VM, just the OS is virtualized. As shown in 4.1b Containers sit on top of a physical server and its host OS—typically Linux or Windows. Each container shares the host OS kernel and, usually, the binaries and libraries, too. Shared components are read-only. Sharing OS resources, such as libraries, significantly reduces the need to reproduce the operating system code—a server can run multiple workloads with a single operating system installation. Containers are thus exceptionally light—they are only megabytes in size and take just seconds to start. What this means in practice is you can put two to three times as many applications on a single server with containers than you can with a VM. Compared to containers, VMs take minutes to run and are an order of magnitude larger than an equivalent container, measured in gigabytes versus megabytes. Container technology has existed for a long time, but the launch of Docker in 2013 made containers essentially industry standard for application and software development. Docker is a software platform that allows you to build, test, and deploy applications quickly. Docker packages software into containers that have everything the software needs to run including libraries, system tools, code, and runtime.

4.3.1 Orchestration

Orchestration [66] is the coordination and management of multiple computer systems, applications and/or services, stringing together multiple tasks in order to execute a larger workflow or process. These processes can consist of multiple tasks that are automated and can involve multiple systems. An example of a container orchestrator can be found in figure 4.1


Virtual Machine
Virtual Machine
Virtual Machine
App A
App B
App C
Guest Operating System
Guest Operating System
Guest Operating System
Hypervisor
Infrastructure

(a) Virtual Machine


Containerized Applications
App A
App B
App C
App D
App E
App F
Docker
Host Operating System
Infrastructure

(b) Container

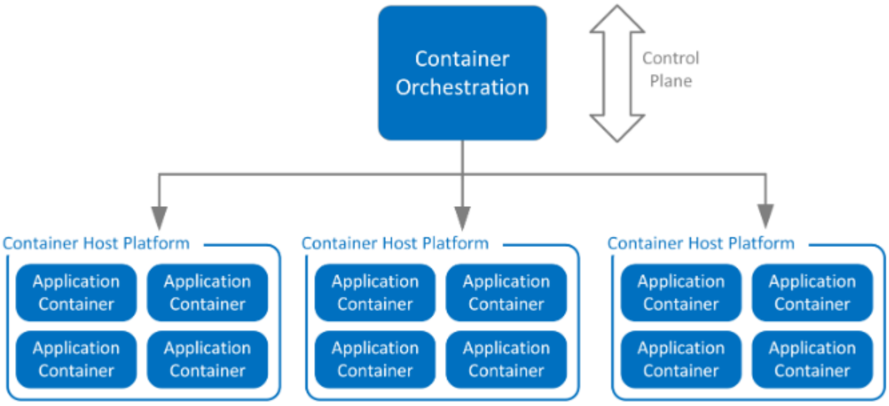

Container Orchestration
Control Plane
Container Host Platform
Application Container
Application Container
Application Container
Application Container
Container Host Platform
Application Container
Application Container
Application Container
Application Container
Container Host Platform
Application Container
Application Container
Application Container
Application Container

Figure 4.1: Container Orchestrator

The goal of orchestration is to streamline and optimize the execution of frequent, repeatable processes and thus to help data teams more easily manage complex tasks and workflows. Anytime a process is repeatable, and its tasks can be automated, orchestration can be used to save time, increase efficiency, and eliminate redundancies. For example, you can simplify data and machine learning with jobs orchestration. Cloud orchestration is the process of automating the tasks that manage connections on private and public clouds. It also integrates automated tasks and processes into a workflow to help you perform specific business functions.

4.3.2 Cloud Orchestration

The rise of cloud computing, involving public, private and hybrid clouds, has led to increasing complexity. This creates a need for cloud orchestration software that can manage and deploy multiple dependencies across multiple clouds. Cloud service orchestration includes tasks such as provisioning server workloads and storage capacity and orchestrating services, workloads and resources.

Remember that cloud orchestration and automation are different things: Cloud orchestration focuses on the entirety of IT processes, while automation focuses on an individual piece. Orchestration simplifies automation across a multi-cloud environment, while ensuring that policies and security protocols are maintained.

However, such an architecture highlights the need for container orchestration, a tool that automates the deployment, management, scaling, networking, and availability of container-based applications.

This is where Kubernetes comes in. Large, distributed containerized applications can become increasingly difficult to coordinate. By making containerized applications dramatically easier to manage at scale, Kubernetes has become a key part of the container revolution. It is a portable, extensible platform that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available. Google open-sourced the Kubernetes project in 2014. Kubernetes builds upon a decade and a half of experience that Google has with running production workloads at scale, combined with best-of-breed ideas and practices from the community.

In the following sections we describe the different components of Kubernetes.

4.4 Kubernetes Control Plane Components

The control plane's components make global decisions about the cluster (for example, scheduling), as well as detecting and responding to cluster events (for example, starting up a new pod when a deployment's replicas field is unsatisfied).

Control plane components can be run on any machine in the cluster. However, for simplicity, set up scripts typically start all control plane components on the same machine, and do not run user containers on this machine. See Creating Highly Available clusters with kubeadm for an example control plane setup that runs across multiple machines.

- □ *kube-apiserver:* The API server is a component of the Kubernetes control plane that exposes the Kubernetes API. The API server is the front end for the Kubernetes control plane.

  The main implementation of a Kubernetes API server is kube-apiserver. kube-apiserver is designed to scale horizontally—that is, it scales by deploying more instances. You can run several instances of kube-apiserver and balance traffic between those instances.

- □ *etcd:*Consistent and highly-available key value store used as Kubernetes' backing store for all cluster data.

  If your Kubernetes cluster uses etcd as its backing store, make sure you have a back up plan for those data.

  You can find in-depth information about etcd in the official documentation[67].

- □ *kube-scheduler:* Control plane component that runs controller processes.

  Logically, each controller is a separate process, but to reduce complexity, they are all compiled into a single binary and run in a single process.

  Some types of these controllers are:

  - – Node controller: Responsible for noticing and responding when nodes go down. Job controller: Watches for Job objects that represent one-off tasks, then creates Pods to run those tasks to completion.

  - EndpointSlice controller: Populates EndpointSlice objects (to provide a link between Services and Pods).
  - ServiceAccount controller: Create default ServiceAccounts for new namespaces.

- ☐ *kube-controller-manager:* Control plane component that runs controller processes.

  Logically, each controller is a separate process, but to reduce complexity, they are all compiled into a single binary and run in a single process.

  Some types of these controllers are:

  - Node controller: Responsible for noticing and responding when nodes go down.
  - Job controller: Watches for Job objects that represent one-off tasks, then creates Pods to run those tasks to completion.
  - EndpointSlice controller: Populates EndpointSlice objects (to provide a link between Services and Pods).
  - ServiceAccount controller: Create default ServiceAccounts for new namespaces.

4.5 Kubernetes Worker Node(s) Components

Node Components run on every node as agents maintaining running pods and providing the Kubernetes runtime environment.

- ☐ *kubelet:* An agent that runs on each node in the cluster. It makes sure that containers are running in a pod. The kubelet takes a set of PodSpecs that are provided through various mechanisms and ensures that the containers described in those PodSpecs are running and healthy.

- ☐ *kube-proxy:* A network proxy that runs on each node in the cluster. It enables the Kubernetes service abstraction by maintaining network rules on the host and performing connection forwarding. Kube-proxy is responsible for request forwarding. It allows TCP and UDP stream forwarding or round robin TCP and UDP forwarding across a set of backend functions.

- ☐ *Container Runtime:* The container runtime (e.g. Docker) is the software that is responsible for running containers.

4.5.1 Other Important Addons

- □ *DNS:* Cluster DNS is a DNS server, in addition to the other DNS server(s) in your environment, which serves DNS records for Kubernetes services. Containers started by Kubernetes automatically include this DNS server in their DNS searches.

4.6 Kubernetes Architecture

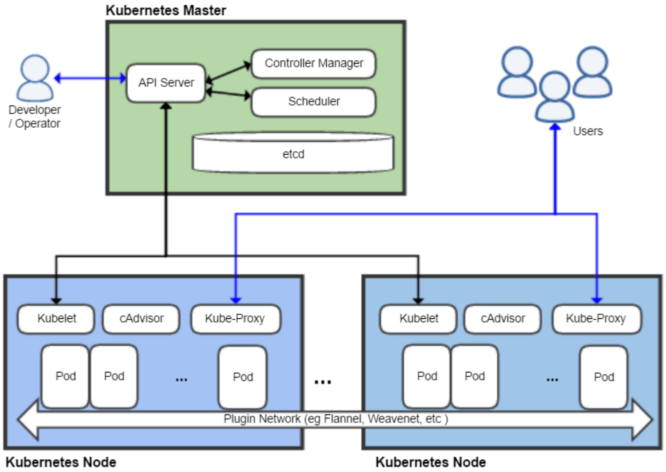

Kubernetes Master
Developer / Operator
API Server
Controller Manager
Scheduler
etcd
Users
Kubelet
cAdvisor
Kube-Proxy
Pod
Pod
...
Pod
...
Kubelet
cAdvisor
Kube-Proxy
Pod
Pod
...
Pod
Plugin Network (eg Flannel, Weavenet, etc)
Kubernetes Node
Kubernetes Node

Figure 4.2: Kubernetes Architecture [1]

Kubernetes's architecture makes use of various concepts and abstractions. Some of these are variations on existing, familiar notions, but others are specific to Kubernetes. As illustrated in figure 4.2 and described before, a Kubernetes cluster is consisted of Nodes. Those nodes are separated into two groups, either Master or Worker nodes. Workloads are executed in Worker Nodes.

4.6.1 Cluster

The highest-level Kubernetes abstraction, the cluster illustrated in figure 4.3, refers to the group of machines running Kubernetes (itself a clustered application) and the containers managed by it. A Kubernetes cluster must have a master, the brain of the system, the node that commands and controls all the other Kubernetes machines in the cluster. A highly available Kubernetes cluster replicates the master's facilities across multiple machines. But only one master at a time runs the job scheduler and controller-manager. The cluster can be set up locally or in the cloud. Most Cloud providers provide a ready-to-use Kubernetes solution.

---

[1] https://en.wikipedia.org/wiki/Kubernetes

[2] https://kubernetes.io/fr/docs/tutorials/kubernetes-basics/create-cluster/cluster-intro/

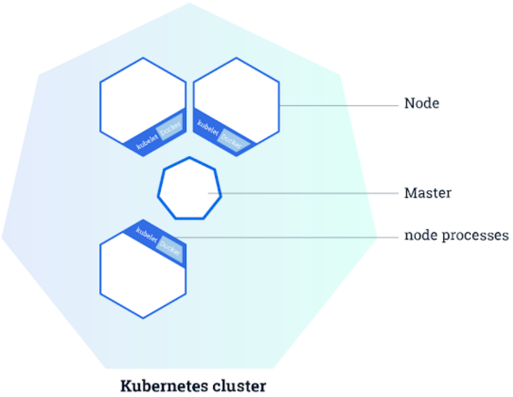

Node
Master
node processes
kubelet
Docker
Kubernetes cluster

Figure 4.3: Cluster-Node abstraction level [2]

4.6.2 Nodes

Each cluster contains Kubernetes nodes. Nodes might be physical machines or VMs. Again, the idea is abstraction: Whatever the application is running on, Kubernetes handles deployment on that substrate. These Nodes can be either Master Nodes or Worker Nodes. A node with its components is presented in figure 4.4.

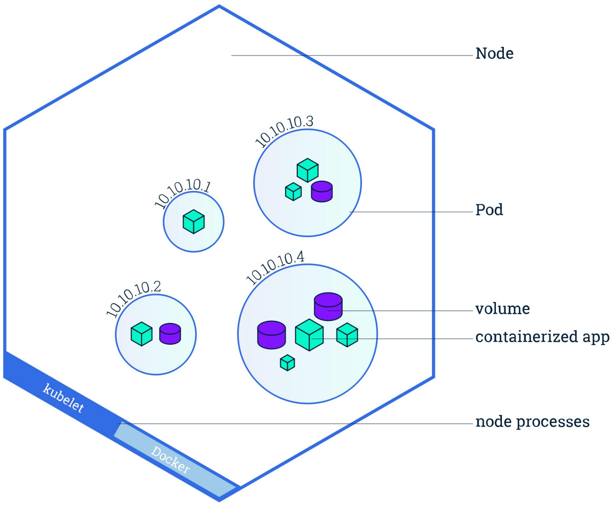

Node
10.10.10.3
10.10.10.1
Pod
10.10.10.4
10.10.10.2
volume
containerized app
kubelet
Docker
node processes

Figure 4.4: Node-Pod-Container abstraction levels [3]

4.6.3 Pods

Nodes run pods, the most basic Kubernetes objects that can be created or managed. Each pod represents a single instance of an application or running process in Kubernetes, and consists of one or more containers as shown in figure 4.4. Kubernetes starts, stops, and replicates all containers in a pod as a group. Pods keep the user's attention on the application, rather than on the containers themselves. Details about how Kubernetes needs to be configured, from the state of up pods, is kept in etcd (distributed key-value store).

---

[3] https://kubernetes.io/docs/tutorials/kubernetes-basics/explore/explore-intro/

Pods are created and destroyed on nodes as needed to conform to the desired state specified by the user in the pod definition. Kubernetes provides an abstraction called a controller for dealing with the logistics of how pods are spun up, rolled out, and spun down. Controllers come in a few different flavors depending on the kind of application being managed. For instance, Job controller is used to ensure that a specified number of the pods will reliably run to completion. Another kind of controller, the deployment, is used to scale an app up or down, update an app to a new version, or roll back an app to a known-good version if there's a problem. Also a deployment will try to reschedule any failed pods. Finally, a deployment tries to provide a guarantee that the required number of pods are running on the cluster.

4.6.4 DaemonSet

A DaemonSet ensures that all (or some) Nodes run a copy of a Pod. As nodes are added to the cluster, Pods are added to them. As nodes are removed from the cluster, those Pods are garbage collected. Deleting a DaemonSet will clean up the Pods it created.

Some typical uses of a DaemonSet are:

- running a cluster storage daemon on every node
- running a logs collection daemon on every node
- running a node monitoring daemon on every node

In a simple case, one DaemonSet, covering all nodes, would be used for each type of daemon. A more complex setup might use multiple DaemonSets for a single type of daemon, but with different flags and/or different memory and cpu requests for different hardware types.

4.6.5 Deployment

As it is described in Kubernetes documentation, a desired state is described in a Deployment, and the Deployment controller changes the actual state to the desired state at a controlled rate. Deployments are defined to create new `ReplicaSets`, or to remove existing Deployments and adopt all their resources with new Deployments. This object offered the easily manageable scalabilty, so as to increase or decrease accordingly the required stress levels, just by changing the replicas of the pods created.

4.6.6 Service

Kubernetes Pods are mortal. They are born and when they die, they are not resurrected. If you use a Deployment to run your app, it can create and destroy Pods dynamically (e.g. when scaling out or in). Each Pod gets its own IP address, however the set of Pods for a Deployment running in one moment in time could be different from the set of Pods running that application a moment later. This leads to the following problem: if a set of Pods (call them "backends") provides functionality to other Pods (call them "frontends") inside your cluster, how do those frontends find out and keep track of which IP address to connect to, so that the frontend can use the backend part of the workload ? A Service is an abstract way to expose an application running on a set of Pods as a network service. Kubernetes gives pods their own IP addresses and a single DNS name for a set of pods and can load-balance across them.

4.6.7 ConfigMaps

A ConfigMap is an API object used to store non-confidential data in key-value pairs. Pods can consume ConfigMaps as environment variables, command-line arguments, or as configuration files in a volume. A ConfigMap allows you to decouple environment-specific configuration from your container images, so that your applications are easily portable.

A ConfigMap is an API object that lets you store configuration for other objects to use. Unlike most Kubernetes objects that have a spec, a ConfigMap has data and binaryData fields. These fields accept key-value pairs as their values. Both the data field and the binaryData are optional. The data field is designed to contain UTF-8 strings while the binaryData field is designed to contain binary data as base64-encoded strings. The name of a ConfigMap must be a valid DNS subdomain name.

4.7 Kubernetes Resources

4.7.1 Default Resources: CPU and Memory

When the user specifies a Pod, he can optionally specify how much CPU and memory (RAM) each container needs. When containers have resource requests specified, the scheduler can make better decisions about which nodes to place Pods on. And when Containers have their limits specified, contention for resources on a node can be handled in a specified manner [4].

---

[4] https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container/

4.7.1.1 Resource Types:

CPU and memory are each a resource type. A resource type has a base unit. CPU is specified in units of cores, and memory is specified in units of bytes. CPU and memory are collectively referred to as compute resources, or just resources. Compute resources are measurable quantities that can be requested, allocated, and consumed. They are distinct from API resources. API resources, such as Pods and Services are objects that can be read and modified through the Kubernetes API server.

Each Container of a Pod can specify one or more of the following:

☐ spec.containers[ ].resources.limits.cpu

☐ spec.containers[ ].resources.limits.memory

☐ spec.containers[ ].resources.requests.cpu

☐ spec.containers[ ].resources.requests.memory

Although requests and limits can only be specified on individual Containers, it is convenient to talk about Pod resource requests and limits. A Pod resource request/limit for a particular resource type is the sum of the resource requests/limits of that type for each Container in the Pod.

4.7.1.2 Meaning of CPU and Memory

Limits and requests in CPU resources are measures in cpu units. One CPU in Kubernetes is equivalent to 1 vCPU or 1 Hyperthread on a bare-metal Intel processor. Also fractional requests are allowed. For example a request of 0.5 cpu (or 500m which can be read as five hundreds millicpu), allocates half of a CPU. CPU is always requested as an absolute quantity, never as a relative quantity; 0.5 is the same amount of CPU on a single-core, dual-core, or a 48-core machine. Regarding to the Memory's requests and limits, they are measured in bytes. Someone can express memory as a plain integer, or as a fixed-point integer. Also the user can use the power-of-two equivalents: Ei, Pi, Ti, Gi, Mi, Ki.

These requests and limits are passed to the container runtime, when the kubelet starts a container of a Pod. When using Docker, there are used the *–cpu-shares* and *–memory* flags accordingly.

When you create a Pod, the Kubernetes scheduler selects a node for the Pod to run on. Each node has a maximum capacity for each of the resource types: the amount of CPU and memory it can provide for Pods. The scheduler ensures that, for each resource type, the sum of the resource requests of the scheduled containers is less than the capacity of the node. Note that although actual memory or CPU resource usage on nodes is very low, the scheduler still refuses to place a Pod on a node if the capacity check fails. This protects against a resource shortage on a node when resource usage later increases, for example, during a daily peak in request rate.

4.7.2 Extended Resources

Extended resources are fully-qualified resource names outside the kubernetes.io domain. They allow cluster operators to advertise and users to consume the non-Kubernetes-built-in resources. For instance, by using this mechanism we can add a graphical processing unit (GPU) in our Kubernetes cluster and let different Pods use it.

There are two steps required to use Extended Resources. First, the cluster operator must advertise an Extended Resource. Second, users must request the Extended Resource in Pods [5].

4.7.3 Kubernetes Device Plugins

Kubernetes provides a device plugin framework that you can use to advertise system hardware resources to the Kubelet.

Instead of customizing the code for Kubernetes itself, vendors can implement a device plugin that you deploy either manually or as a DaemonSet. The targeted devices include GPUs, high-performance NICs, FPGAs, InfiniBand adapters, and other similar computing resources that may require vendor specific initialization and setup.

Following a successful registration, the device plugin sends the kubelet the list of devices it manages, and the kubelet is then in charge of advertising those resources to the API server as part of the kubelet node status update. For example, after a device plugin registers hardware-vendor.example/foo with the kubelet and reports two healthy devices on a node, the node status is updated to advertise that the node has 2 "Foo" devices installed and available.

Then, users can request devices as part of a Pod specification (see container). Requesting extended resources is similar to how you manage requests and limits for other resources, with the following differences:

- ☐ Extended resources are only supported as integer resources and cannot be overcommitted.
- ☐ Devices cannot be shared between containers.

The above limitations are two of the main interests of this thesis. The fact that extended resources can not be overcommited or shared between containers

---

[5] https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/

(only one application can utilize the full GPU at a time) causes GPUs to be underutilized, leading to increased latency, decreased throughput and energy consumption, since a running GPU has high power usage even when not utilized. Namely, the main purpose of this thesis is to design and develop a Kubernetes component, able to utilize the full spectrum of the provided GPU resources.

4.7.3.1 Manage GPUs Using device plugins

As an administrator, you have to install GPU drivers from the corresponding hardware vendor on the nodes and run the corresponding device plugin from the GPU vendor. Here are some links to vendors' instructions:

☐ AMD

☐ Intel

☐ NVIDIA

Once you have installed the plugin, your cluster exposes a custom schedulable resource such as amd.com/gpu or nvidia.com/gpu.

You can consume these GPUs from your containers by requesting the custom GPU resource, the same way you request cpu or memory. However, there are some limitations in how you specify the resource requirements for custom devices.

GPUs are only supposed to be specified in the limits section, which means:

You can specify GPU limits without specifying requests, because Kubernetes will use the limit as the request value by default. You can specify GPU in both limits and requests but these two values must be equal. You cannot specify GPU requests without specifying limits.

4.8 Kubernetes Scheduler

4.8.1 Scheduling overview

A scheduler watches for newly created Pods that have no Node assigned. For every Pod that the scheduler discovers, the scheduler becomes responsible for finding the best Node for that Pod to run on. The scheduler reaches this placement decision taking into account the scheduling principles described below.

4.8.2 Kubernetes Scheduling Framework

The scheduling framework is a pluggable architecture for the Kubernetes scheduler. It adds a new set of "plugin" APIs to the existing scheduler. Plugins are compiled into the scheduler. The APIs allow most scheduling features to be implemented as plugins, while keeping the scheduling "core" lightweight and maintainable.

4.8.3 Scheduling and Binding Cycle

The scheduling cycle selects a node for the Pod, and the binding cycle applies that decision to the cluster. Together, a scheduling cycle and binding cycle are referred to as a "scheduling context".

Scheduling cycles are run serially, while binding cycles may run concurrently.

A scheduling or binding cycle can be aborted if the Pod is determined to be unschedulable or if there is an internal error. The Pod will be returned to the queue and retried.

4.8.3.1 Extension points

In figure 4.5 are represented the scheduling context of a Pod and the extension points that the scheduling framework exposes. In this picture "Filter" is equivalent to "Predicate" and "Scoring" is equivalent to "Priority function".

One plugin may register at multiple extension points to perform more complex or stateful tasks.

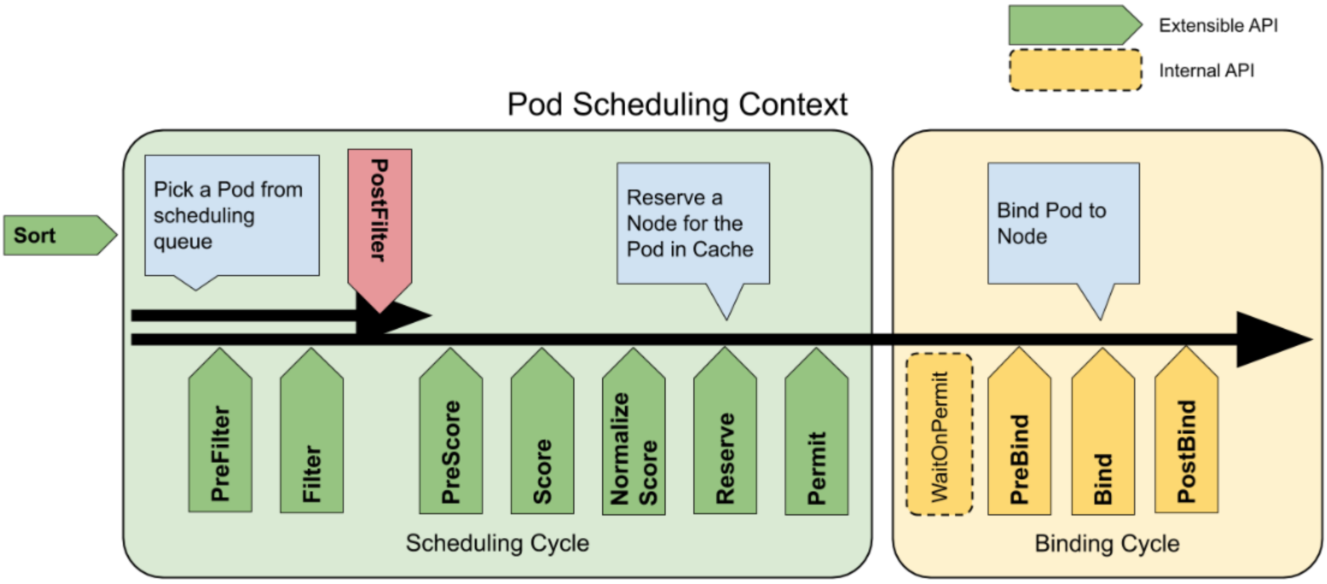

Extensible API
Internal API
Pod Scheduling Context
Sort
Pick a Pod from scheduling queue
PostFilter
Reserve a Node for the Pod in Cache
Bind Pod to Node
PreFilter
Filter
PreScore
Score
Normalize Score
Reserve
Permit
WaitOnPermit
PreBind
Bind
PostBind
Scheduling Cycle
Binding Cycle

Figure 4.5: Pod Scheduling Context [6]

Chapter 5

Experimental Infrastructure

In this chapter, we describe the cluster we have created for our experiments, the GPU monitoring system and the MLPerf benchmark suite [68] which was used for the workload creation.

5.1 System setup

For the nodes of our cluster, we have created 4 virtual machines (VMs) (1 master node and 3 worker nodes) on top of the physical machines. The CPUs of the VMs consist of 4 to 8 cores and the RAM sizes range from 8 to 16 GB. We used `Qemu KVM` as our hypervisor. Of the three worker nodes, the first one is equiped with an NVIDIA V100 GPU with 32GB of memory and 80 SMs, the second one with an NVIDIA A30 GPU with 24GB of memory and 56 SMs while the third one has no GPUs available. All of the virtual machines are deployed on the infrastructure of the laboratory in NTUA. In order to simulate a cloud environment, all the referenced workloads running on the cluster have been containerized using the Docker platform.

Each VM's characteristics are described in the following table.

Virtual Machines				
Processor	**Role**	**CPU Cores**	**CPU RAM (GB)**	**GPU Access**
Intel(R) Xeon(R) Gold 6138 CPU @ 2.00GHz	Master	8	8	No
Intel(R) Xeon(R) CPU E5-2658A v3 @ 2.20GHz	Worker	4	16	No
Intel(R) Xeon(R) Gold 5218R CPU @ 2.10GHz	Worker	8	16	V100
Intel(R) Xeon(R) Gold 5218R CPU @ 2.10GHz	Worker	8	16	A30

Table 5.1: Virtual Machines Characteristics

The combination of VMs with containers is currently the common way of deploying cloud clusters at scale, since it establishes the perfect catalyst for reliability and robustness. On top of the VMs, we have deployed Kubernetes as our container orchestrator, one of the most popular and most used platforms nowadays.

The system as a whole is illustrated in figure 5.1.

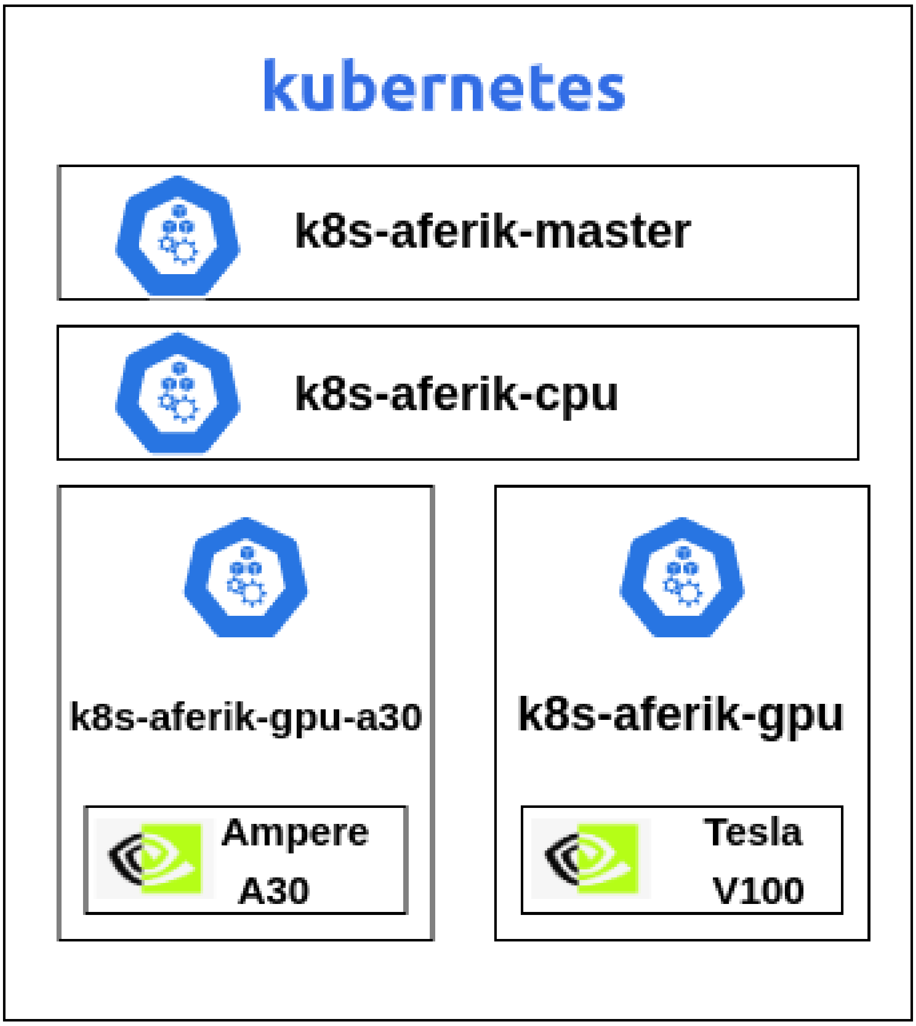

kubernetes
k8s-aferik-master
k8s-aferik-cpu
k8s-aferik-gpu-a30
Ampere A30
k8s-aferik-gpu
Tesla V100

Figure 5.1: Experimental Infrastructure Overview

5.2 GPU Infrastructure

5.2.1 NVIDIA Volta

5.2.1.1 Architecture

Every industry needs AI, and with this massive leap forward in speed, AI can now be applied to every industry. Equipped with 640 Tensor Cores[69], Volta delivers over 125 teraFLOPs per second (TFLOPS) of deep learning performance, over a 5X increase compared to prior generation NVIDIA Pascal™ architecture.

Volta uses next generation revolutionary NVIDIA NVLink™ high-speed interconnect technology. This delivers 2X the throughput, compared to the previous generation of NVLink. This enables more advanced model and data parallel approaches for strong scaling to achieve the absolute highest application performance.

Data scientists are often forced to make trade-offs between model accuracy and longer run-times. With Volta-optimized CUDA and NVIDIA Deep Learning SDK libraries [70] like cuDNN [71], NCCL, and TensorRT[72], the industry's top frameworks and applications can easily tap into the power of Volta running mixed precision[73]. This propels data scientists and researchers towards discoveries faster than before.

5.2.1.2 Volta MPS

The Volta architecture introduced new MPS [41] capabilities. Compared to MPS on pre-Volta GPUs, Volta MPS provides a few key improvements:

- Volta MPS clients submit work directly to the GPU without passing through the MPS server.
- Each Volta MPS client owns its own GPU address space instead of sharing GPU address space with all other MPS clients.
- Volta MPS supports limited execution resource provisioning for Quality of Service (QoS).

This document will introduce the new capabilities, and note the differences between Volta MPS and MPS on pre-Volta GPUs. Running MPS on Volta will automatically enable the new capabilities.

5.2.1.3 NVIDIA Tesla V100

Tesla products from the NVIDIA company introduce a line of computational graphics processors that are very similar to the NVIDIA Quadro series (they usually use the same chip). However, they have an isolated display interface. They're also available in passively-cooled form-factors, which are specifically appropriate for use in servers (rack-mounts).

Users of professional applications can, thanks to the CUDA architecture, use graphical CUDA stream processors. Thanks to this, it is possible to use the raw performance of a graphics card for specific calculations, which can significantly increase work speed compared to the use of a traditional processor, which are significantly limited by the lower number of cores.

Tesla V100 is architected from the ground up to simplify programmability NVIDIA NVLink in Tesla V100 delivers 2X higher throughput compared to the previous generation Equipped with 640 Tensor Cores, Tesla V100 delivers 125 TeraFLOPS of deep learning performance With a combination of improved raw bandwidth of 900 GB/s and higher DRAM utilization efficiency at 95%, Tesla V100 delivers.

5.2.2 NVIDIA Ampere

5.2.2.1 Architecture

First introduced in the NVIDIA Volta™ architecture, NVIDIA Tensor Core technology has brought dramatic speedups to AI, bringing down training times from weeks to hours and providing massive acceleration to inference. The NVIDIA Ampere architecture builds upon these innovations by bringing new precisions—Tensor Float 32 (TF32) and floating point 64 (FP64)—to accelerate and simplify AI adoption and extend the power of Tensor Cores to HPC.

TF32 works just like FP32 while delivering speedups of up to 20X for AI without requiring any code change. Using NVIDIA Automatic Mixed Precision, researchers can gain an additional 2X performance with automatic mixed precision and FP16 by adding just a couple of lines of code. And with support for bfloat16, INT8, and INT4, Tensor Cores in NVIDIA Ampere architecture Tensor Core GPUs create an incredibly versatile accelerator for both AI training and inference. Bringing the power of Tensor Cores to HPC, A100 and A30 GPUs also enable matrix operations in full, IEEE-certified, FP64 precision.

5.2.2.2 Multi Instance GPU

Every AI and HPC application can benefit from acceleration, but not every application needs the performance of a full GPU. Multi-Instance GPU (MIG) is a feature supported on A100 and A30 GPUs that allows workloads to share the GPU. With MIG, each GPU can be partitioned into multiple GPU instances, fully isolated and secured at the hardware level with their own high-bandwidth memory, cache, and compute cores. Now, developers can access breakthrough acceleration for all their applications, big and small, and get guaranteed quality of service. And IT administrators can offer right-sized GPU acceleration for optimal utilization and expand access to every user and application across both bare-metal and virtualized environments.

5.2.2.3 NVIDIA A30

The NVIDIA A30 Tensor Core GPU delivers a versatile platform for mainstream enterprise workloads, like AI inference, training, and HPC. With TF32 and FP64 Tensor Core support, as well as an end-to-end software and hardware solution stack, A30 ensures that mainstream AI training and HPC applications can be rapidly addressed. Multi-instance GPU (MIG) ensures quality of service

(QoS) with secure, hardware-partitioned, right-sized GPUs across all of these workloads for diverse users, optimally utilizing GPU compute resources.

Each GPU's characteristics are described in the following table.

GPUs		
GPU Name	**SMs**	**Memory (GB)**
NVIDIA Tesla V100	80	32
NVIDIA Ampere A30	56	24

Table 5.2: GPU Characteristics

5.3 GPU Monitoring System

5.3.1 NVIDIA GPU Metrics Exporter

DCGM-Exporter is a tool based on the Go APIs to NVIDIA DCGM that allows users to gather GPU metrics and understand workload behavior or monitor GPUs in clusters. dcgm-exporter is written in Go and exposes GPU metrics at an HTTP endpoint (/metrics) for monitoring solutions.

These metrics are exported in time-series format in order to be used by time-series databases like Influx, Prometheus etc. From the Kubernetes perspective, the DCGM exporter forms a DaemonSet that starts a damon Pods on every node equipped with a GPU. These Pods execute metrics queries to the GPUs of the nodes using NVIDIA Data Center GPU Manager (DCGM) [1]. Finally, these metrics are sent to the prometheus monitoring system[2] which exposes a service to the cluster from which any cluster resource can access the metrics.

The GPU metrics we mainly used in our experiments are presented below.

- `DCGM_FI_DEV_FB_FREE`: Framebuffer memory free (in MiB)
- `DCGM_FI_DEV_FB_USED`: Framebuffer memory used (in MiB)
- `DCGM_FI_PROF_GR_ENGINE_ACTIVE`: Ratio of time the graphics engine is active (in %)
- `DCGM_FI_DEV_TOTAL_ENERGY_CONSUMPTION`: Total energy consumption since boot (in mJ)
- `DCGM_FI_DEV_POWER_USAGE`: Power draw (in W)
- `DCGM_FI_PROF_DRAM_ACTIVE`: Ratio of cycles the device memory interface is active sending or receiving data (in %)

5.3.2 Prometheus Timeseries Database

Prometheus is an open-source systems monitoring and alerting toolkit originally built at SoundCloud. Since its inception in 2012, many companies and organizations have adopted Prometheus, and the project has a very active developer and user community. It is now a standalone open source project and maintained independently of any company. To emphasize this, and to clarify the project's governance structure, Prometheus joined the Cloud Native Computing Foundation in 2016 as the second hosted project, after Kubernetes.

Prometheus is based on a multi-dimensional data model with time series data identified by metric name and key/value pairs. It provides PromQL, a flexible query language to leverage the dimensionality. Prometheus does not rely on distributed storage hence each single server node is autonomous. The time series collection happens via a pull model over HTTP while the time series pushing is supported via an intermediate gateway. The Prometheus targets are discovered via service discovery or static configuration. Finally, it provides multiple modes of graphing and dashboarding.

The Prometheus ecosystem consists of multiple components, many of which are optional. The main Prometheus component is the Prometheus server which scrapes and stores time series data. There is a push gateway for supporting short-lived jobs and special-purpose exporters for services like HAProxy, StatsD, Graphite, etc. For the alerts handling an alertmanager component is provided. In addition, Prometheus has client libraries for instrumenting application code while various tools are supported.

This diagram illustrates the architecture of Prometheus and some of its ecosystem components:

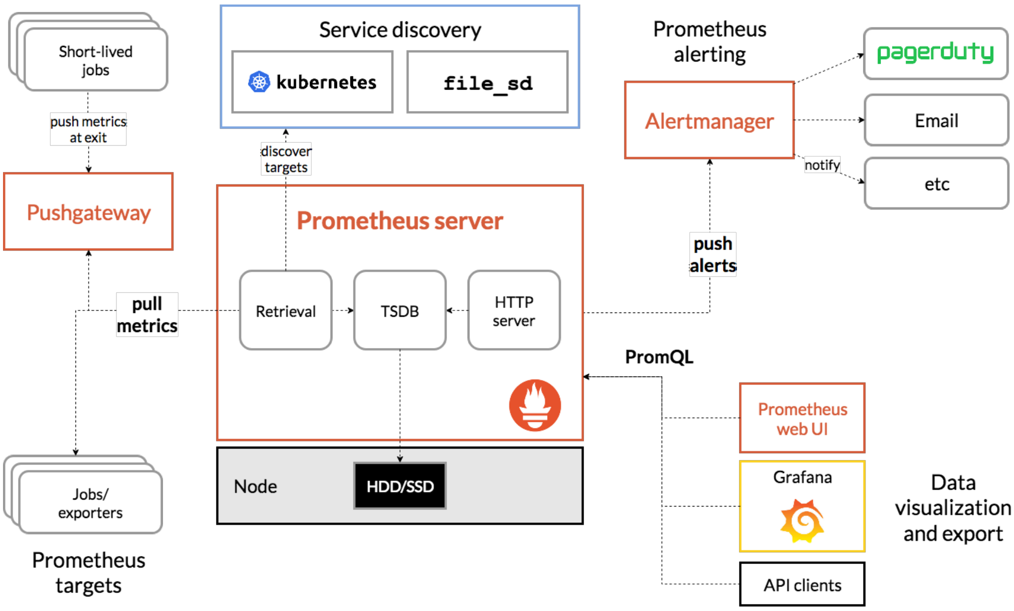

Short-lived jobs
push metrics at exit
Pushgateway
pull metrics
Jobs/ exporters
Prometheus targets
Service discovery
kubernetes
file_sd
discover targets
Prometheus server
Retrieval
TSDB
HTTP server
Node
HDD/SSD
Prometheus alerting
Alertmanager
pagerduty
Email
etc
notify
push alerts
PromQL
Prometheus web UI
Grafana
API clients
Data visualization and export

Figure 5.2: Prometheus Architecture

Prometheus scrapes metrics from instrumented jobs, either directly or via an intermediary push gateway for short-lived jobs. It stores all scraped samples locally and runs rules over this data to either aggregate and record new time series from existing data or generate alerts. Grafana or other API consumers can be used to visualize the collected data.

In our case, the exported GPU metrics are inserted to the Prometheus TSDB every 1 second and the queries are performed with PromQL [74].

5.4 Description of Cloud GPU workloads

Modern data-center server machines accommodate a wide range of workloads, which are basically either batch/best-effort (BE) applications, or user-interactive/latency-critical (LC) applications. The former type of workloads require the highest possible throughput, whereas the latter demand to meet their QoS constraints. Throughout this thesis we focused on latency-critical applications that require GPU resources. For that reason we used MLPerf Inference [4] for creating our workload.

5.4.1 MLPerf Benchmarks

MLPerf is a consortium of AI leaders from academia, research labs, and industry whose mission is to “build fair and useful benchmarks” that provide unbiased evaluations of training and inference performance for hardware, software, and services—all conducted under prescribed conditions. To stay on the cutting edge of industry trends, MLPerf continues to evolve, holding new tests at regular intervals and adding new workloads that represent the state of the art in AI. The MLPerf philosophy is the creation of a widely accepted benchmark suite that will benefit the entire community, including researchers, developers, hardware manufacturers, builders of machine learning frameworks, cloud service providers, application providers, and end users.

The main goals of MLPerf project are:

- ☐ Accelerate progress in ML
- ☐ Serve both the commercial and research communities
- ☐ Enable fair comparison of competing systems yet encourage innovation to improve the state-of-the-art of ML
- ☐ Enforce replicability to ensure reliable results
- ☐ Keep benchmarking effort affordable so all can participate

MLPerf began in February 2018 with a series of meetings between engineers and researchers from Baidu, Google, Harvard University, Stanford University and the University of California Berkeley. MLPerf launched the Training benchmark suite on May 2nd, 2018 and published the first Training results, including results from Google, Intel, and NVIDIA, on December 12, 2018. MLPerf launched the Inference benchmark suite on June 24th, 2019.

As we mentioned before, in this thesis, we focus on workloads that consist of latency-critical applications. Because of this choice we used the MLPerf Inference rather than the MLPerf Training benchmarks.

5.4.2 MLPerf Inference

MLPerf Inference [3, 4] is a benchmark suite for measuring how fast systems can process inputs and produce results using a trained model. Below is a short summary of the current benchmarks and metrics.

Each MLPerf Inference benchmark is defined by a model, a dataset, a quality target, and a latency constraint. The following three benchmarks are in version v0.5 of the suite and were used for the workload creation.

Area	**Task**	**Backend**	**Model**	**Dataset**	**Quality**
Vision	Image classification	Onnxruntime	Resnet50-v1.5	ImageNet (224x224)	FP32 (76.46%)
Vision	Image classification	Onnxruntime	MobileNets-v1 224	ImageNet (224x224)	FP32 (71.68%)
Vision	Object detection	Onnxruntime	SSD-MobileNets-v1	COCO (300x300)	FP32 (0.22 mAP)
Vision	Image classification	Tensorflow	Resnet50-v1.5	ImageNet (224x224)	FP32 (76.46%)
Vision	Image classification	Tensorflow	MobileNets-v1 224	ImageNet (224x224)	FP32 (71.68%)
Vision	Object detection	Tensorflow	SSD-MobileNets-v1	COCO (300x300)	FP32 (0.22 mAP)

Table 5.3: MLPerf Inference Benchmarks

In each benchmark the pretrained model is set on a backend like Tensorflow, PyTorch, Onnx Runtime etc.

The key component of the MLPerf Inference Benchmark is the Load Generator [75]. The Load Generator is a reusable module that efficiently and fairly measures the performance of inference systems. It generates traffic for scenarios as formulated by a diverse set of experts in the MLPerf working group. The scenarios emulate the workloads seen in mobile devices, autonomous vehicles, robotics, and cloud-based setups. Although the Load Generator is not model or dataset aware, its strength is in its reusability with logic that is.

The following is a diagram of how the Load Generator can be integrated into an inference system, resembling how the used MLPerf reference models are implemented.

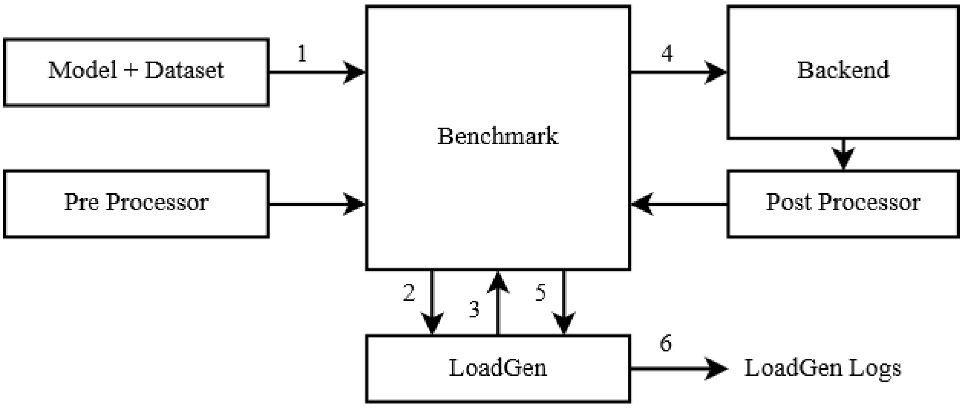


Figure 5.3: Load Generator Integration in MLPerf Inference Benchmarks

As shown in figure 5.3 the Benchmark knows the model, dataset, and preprocessing (1). The Benchmark hands dataset sample IDs to Load Generator (2). Load Generator starts generating queries of sample IDs (3). These queries are translated to backend requests (4). The result is post processed and forwarded to the Load Generator (5). Finally, Load Generator outputs logs for analysis (6).

After analyzing the benchmark architecture the only thing left to define is the way queries are sent to the backend. In order to enable representative testing of a wide variety of inference platforms and use cases, MLPerf has defined four different scenarios as described below. A given scenario is evaluated by a standard load generator generating inference requests in a particular pattern and measuring a specific metric.

From the Kubernetes perspective the wowwrkload consists of a set of Pods, each running a single MLPerf inference task. One container image was created per inference engine (Framework + Model) containing all the needed software to support the execution of this particular engine and afterwards those images were ran on Kubernetes with the appropriate parameters describe the Dataset, the Scenario, Number of Queries etc. The scenarios used for this project were Single Stream and Multi Stream. We also modified the default query number in order to create Pods that execute different number of queries. The number of queries ranges from 270 up to 22000.

In figure 5.4, we present all the components of our proposed system.

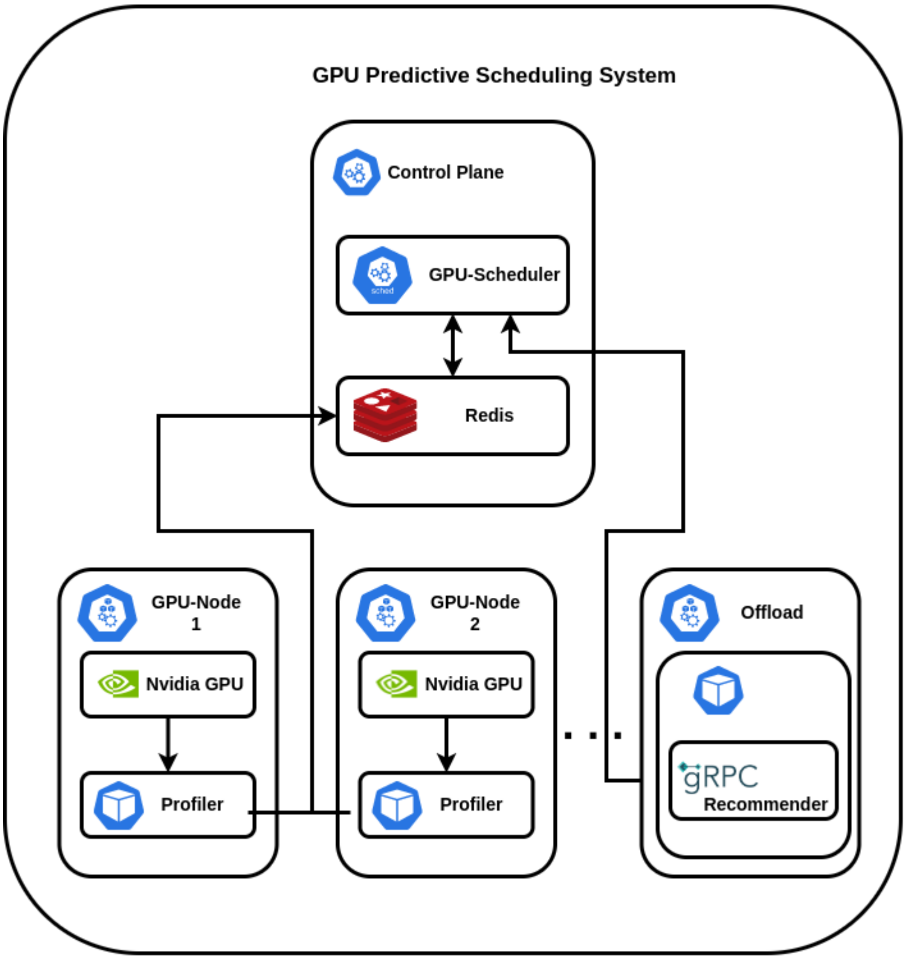

GPU Predictive Scheduling System
Control Plane
GPU-Scheduler
sched
Redis
GPU-Node 1
Nvidia GPU
Profiler
GPU-Node 2
Nvidia GPU
Profiler
Offload
gRPC
Recommender

Figure 5.4: Overall System

Chapter 6

Characterization

6.1 Introduction

Kubernetes promises high scalability, flexibility and cost-effectiveness to satisfy emerging computing requirements. To efficiently provision gpu resources in the cloud, system administrators need the capabilities of characterizing and predicting workload on the machines.

In this thesis, we use data traces obtained from inference tasks on pre-trained neural models running on two real data center GPUs (NVIDIA V100, NVIDIA A30) located on the premises of the laboratory to develop such capabilities. First, we search for repeatable patterns by exploring the execution trace of each workload in all different configurations. Then, we study the batching capabilities of each GPU as well as the intensity of the model and dataset loading operations by executing each inference engine with x queries on each GPU full, the with $x/2$ queries on one GPU partition halved in terms of memory and GPU capacity etc. Finally, we utilize the data traces across the GPUs to detect similarities and differences between the Volta (V100) and Ampere (A30) architectures.

The models and datasets used in the following experiments are part of the MLPerf Inference Classification and Object Detection benchmarks and the partitioning means:

- in the case of A30, partitioning the GPU compute and memory resources utilizing the MIG feature
- in the case of V100, partitioning the GPU compute and memory resources utilizing the MPS CUDA_MPS_PINNED_DEVICE_MEM_LIMIT and CUDA_MPS_ACTIVE_THREAD_PERCENTAGE environment variables.

Finally, each experiment has been ran at least ten times and all of the figures appended correspond to the averaged values.

6.2 Utilizing Different Configurations

In this experiment, we use the workloads described in section 5 using the SingleStream scenario from MLPerf to analyze and study the execution of each model in all possible GPU cofingurations. In our case, we first considered the whole server as one configuration. Then we partition each GPU in half using either MPS of MIG and consider this as another configuration and so on.

In figure 6.1 we can see the number of queries that each engine-configurations pair achieved. Queries are selected in a way such that the median latency of a task is about half a minute. In $x - axis$ we can see the configurations. xP means GPU resources are equally split in x partitions, and A30/V100 describe the GPU.

From figure 6.1 we observe that, as expected, the QPS of each application is degrading as GPU resources are further partitioned. One rather interesting note here is that smaller applications are less affected by resource partitioning. This leads to the conclusion that GPUs are under-utilized when used as a whole to execute one such application since it could produce results as close as 1% to the optimal in a fraction of the GPU. Finally, we observe that in some cases even in large models the QPS decrease only by a marginal amount and this is because each model has different key architectural bottlenecks.

In figure 6.3 we can see that both for V100 and A30, FB_USED (memory used) tends to decrease for smaller fractions of the GPU, and total energy consumption tends to increase. This is because power does not have significant changes when the GPU is empty and when it is loaded and therefore we need to waste more energy when running on smaller partitions for the same job since it would take more time. As far as memory is concerned when running on smaller fractions of the card's memory, the applications tend to save memory, leading to lower mean memory utilization. However, the most important conclusion which emerges from figure 6.2b is that GPU Utilization (GR_ENGINE_ACTIVE) decreases for smaller partitions of the GPU which means that relatively small GPU workloads tend to exhaust GPU resources due to other bottlenecks. GPU Utilization is equal to the percent of time over the past sample period during which one or more kernels was executing on the GPU. The sample period may be between 1 second and 1/6 second depending on the product. The reason why the metrics related to V100 for GPU Utilization are increasing is because GPU metrics that DCGM provides for V100 refer to the whole GPU, whereas A30 refer to the specific partition. Also we can see that GPU utilization increases in smaller MPS "partitions" possibly leading to the conclusion that limited SMs are sufficient and that MPS manages clients demands in such a way that if limits are as close as possible to the real constraints GPU utilization is highest. Furthermore, supposing that MPS fully isolates GPU clients' address spaces and

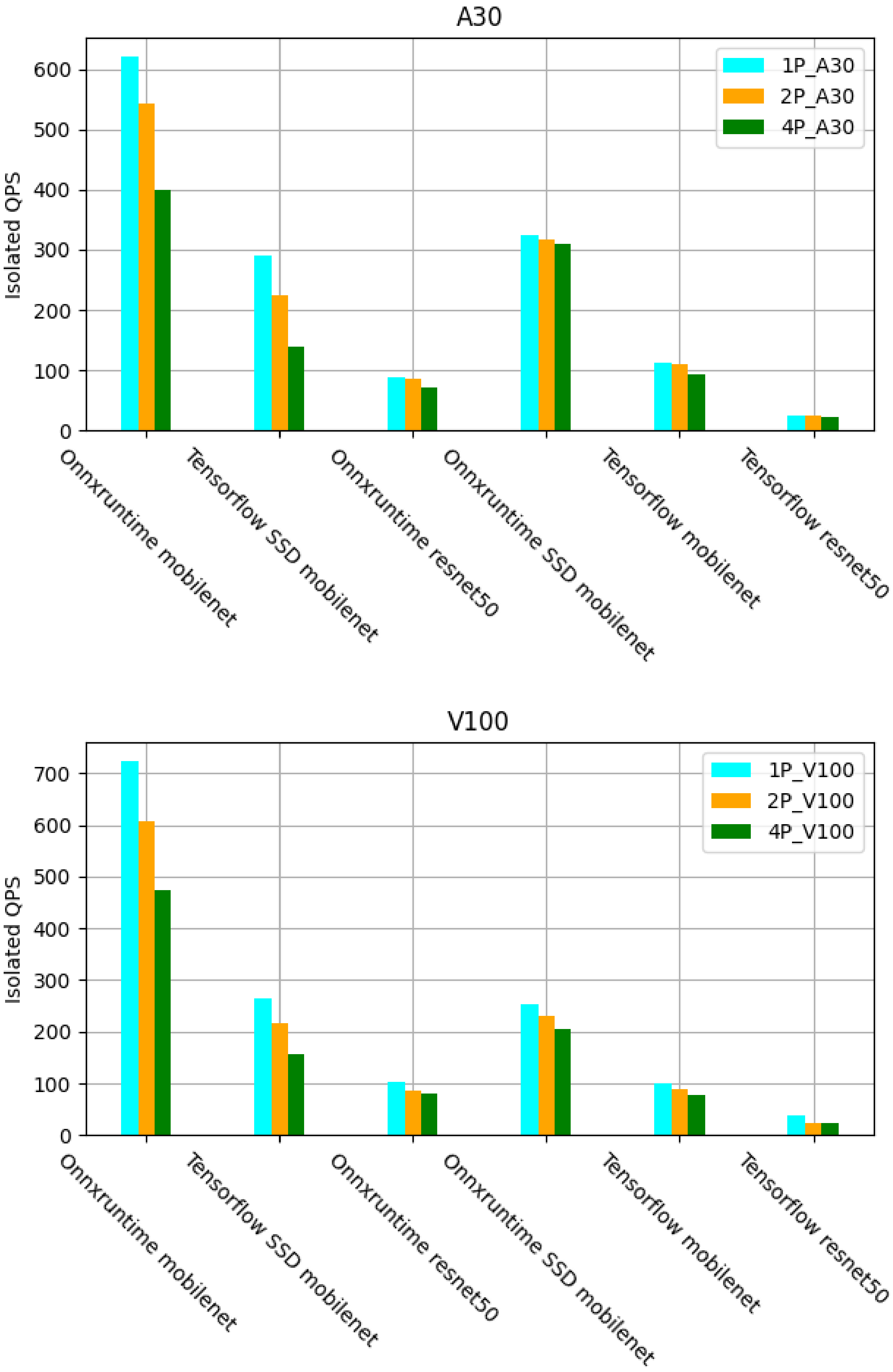
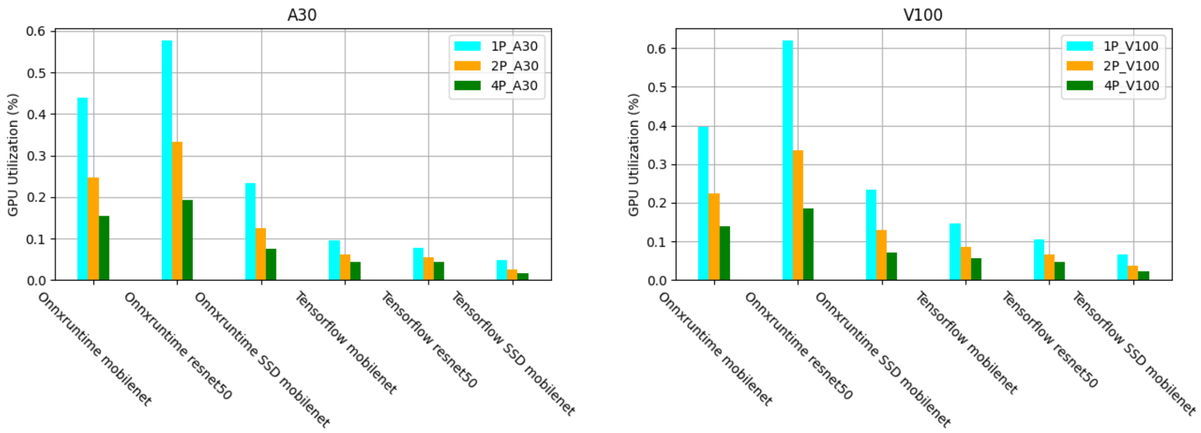
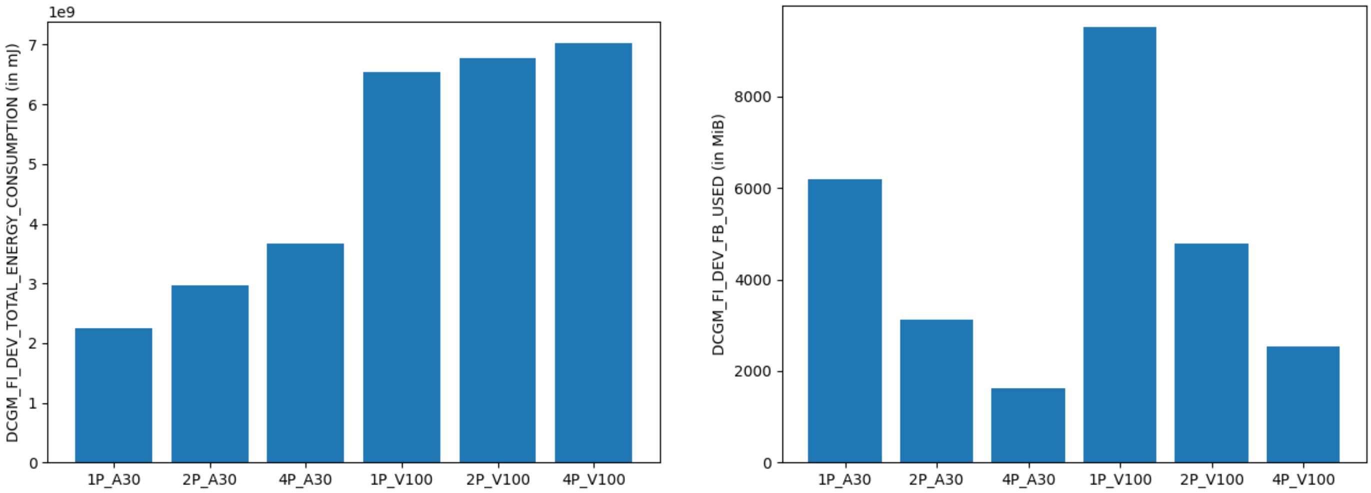


Figure 6.1: Queries per second for all models and all configurations [1]



(a) (b)

Figure 6.2: DCGM_FI_PROF_GR_ENGINE_ACTIVE (in %)



(a) DCGM_FI_DEV_TOTAL_ENERGY_CONSUMPTION (in mJ)

(b) DCGM_FI_DEV_FB_USED (in MiB)

Figure 6.3: Tensorflow Resnet50 DCGM metrics

limits the SMs' utilization, it is safe to assume that approximately all the rest GPU SMs and memory are free. Therefore GPU resources are underutilized and the need for a better, heterogeneous aware and interference tolerant scheduling algorithm arises.

6.3 Fixed workload

In this experiment we examine the ability of the Ampere and Volta GPUs to host multiple applications at the same time and the potential overheads and bottlenecks that appear from computation and/or memory heavy applications. The idea of the experiment is to gradually decrease the intensity of the applications by decreasing the expected number of queries while also decreasing the provided resources and increasing parallelism.

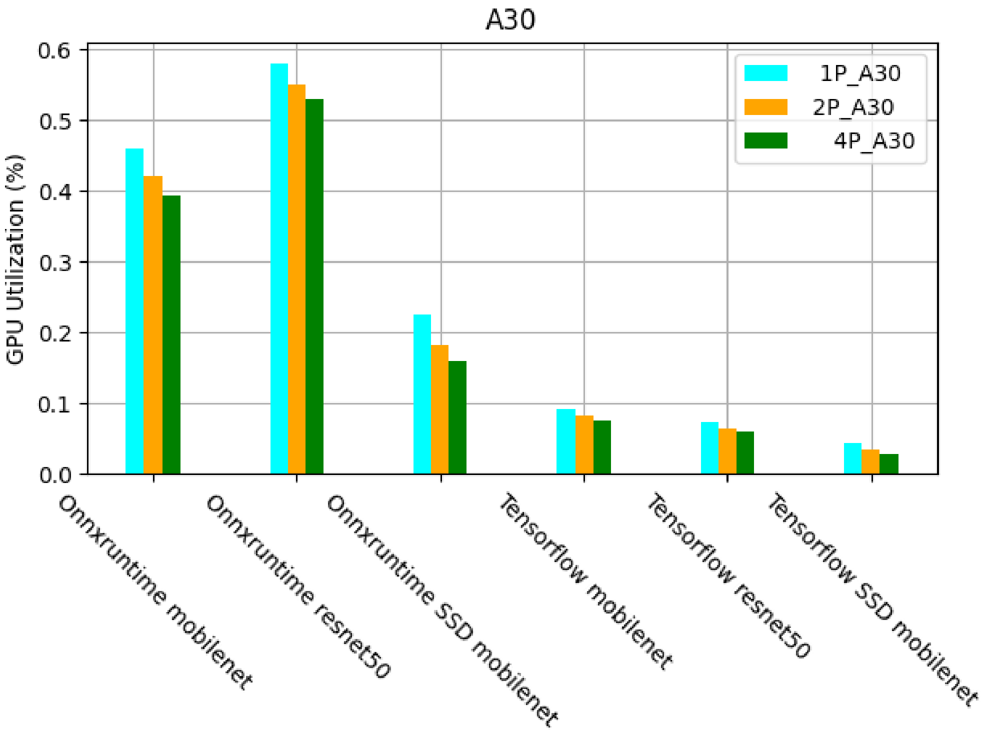
In figure 6.5 xP designates that x pods (applications) were executed for this related benchmark, A30/V100 designate the GPU they were ran on and backend/model are described from y labels as explained before. As expected, the mean frame buffer memory used by in the case of 2 pods is almost double the memory of one pod and so on. This is because in our experiments one of the most memory consuming process is the loading of the model which happens more frequently for greater batch sizes. In the case of GPU utilization we see that when batching applications in the case of A30, we tend to have lower utilization for more pods whilst for V100 when co-scheduling increasing batch size mostly means decrease of utilization and only in few cases we see an decrease. This can be strongly related to MIG which partitions the GPU into multiple GPU instances, fully isolated and secured at the hardware level with their own high-bandwidth memory, cache, and compute cores.

To gain further insight in the purpose of the experiment we append figures 6.6a and 6.6b which describe the total number of queries executed per second among the pods in each batch. Clearly in both cases the queries made per second follow an almost linear advance when increasing batch size. In figures 6.7a and 6.7b validate that the end-to-end system throughput (namely, the gpu-k8s-scheduler) also follows the same pattern.

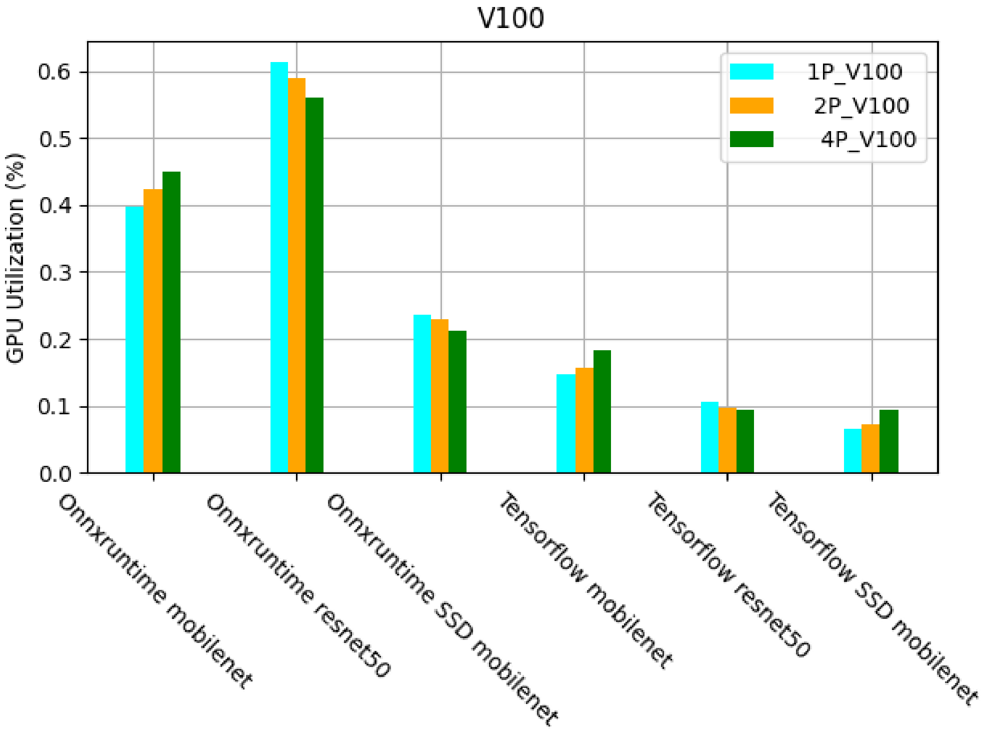
6.4 Interference

In Kubernetes environments, interference between collocated pods can degrade performance, violating the quality of service (QoS) guarantees that many cloud workloads require. In this experiment we analyze the results of the last experiment for workload characterization. Purpose of this experiment is to examine the level of interference caused and tolerated by the pods when collocated with each other in the same GPU configuration. For the needs of the experiment we used the $Offline$ scenario from MLPerf in order to send all the inference queries at the same and thus produce as much interference as possible.

First, we are going to take a look at how applications, ran on different MIG partitions in A30, respond to interference. In figures 6.8 we can see the number of queries that each inference achieved when collocated with each of the rest engines. The collocated inference engine is depicted in the labels of x-axis. The first thing that we notice in these figures is that there is a small fluctuation in the values of QPS when one engine is collocated with another GPU application. What this means is that even when utilizing MIG architecture in order to isolate the execution of the kernels in terms of memory and GPU engines, there are always unpredictable sources of interference degrading the system's performance. However, it is obvious that among the experiments done to measure

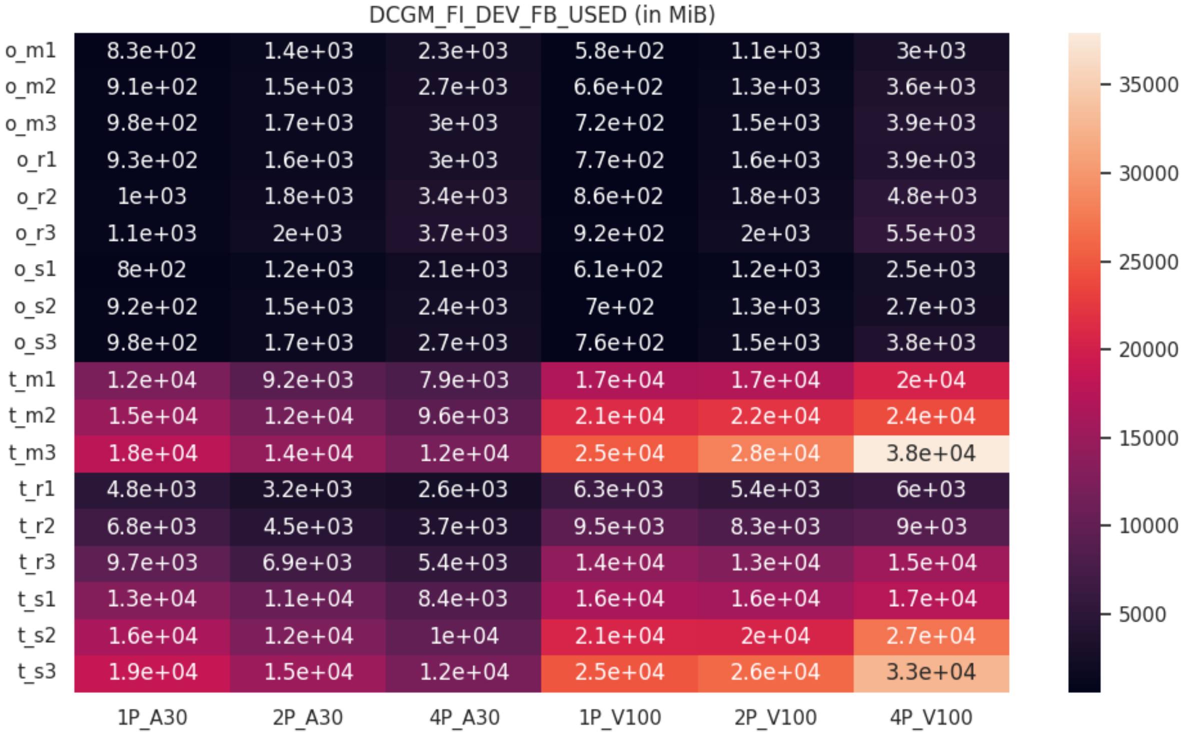


(a)



(b)

Figure 6.4: GPU Utilization for fixed workload



DCGM_FI_DEV_FB_USED (in MiB)

	1P_A30	2P_A30	4P_A30	1P_V100	2P_V100	4P_V100
o_m1	8.3e+02	1.4e+03	2.3e+03	5.8e+02	1.1e+03	3e+03
o_m2	9.1e+02	1.5e+03	2.7e+03	6.6e+02	1.3e+03	3.6e+03
o_m3	9.8e+02	1.7e+03	3e+03	7.2e+02	1.5e+03	3.9e+03
o_r1	9.3e+02	1.6e+03	3e+03	7.7e+02	1.6e+03	3.9e+03
o_r2	1e+03	1.8e+03	3.4e+03	8.6e+02	1.8e+03	4.8e+03
o_r3	1.1e+03	2e+03	3.7e+03	9.2e+02	2e+03	5.5e+03
o_s1	8e+02	1.2e+03	2.1e+03	6.1e+02	1.2e+03	2.5e+03
o_s2	9.2e+02	1.5e+03	2.4e+03	7e+02	1.3e+03	2.7e+03
o_s3	9.8e+02	1.7e+03	2.7e+03	7.6e+02	1.5e+03	3.8e+03
t_m1	1.2e+04	9.2e+03	7.9e+03	1.7e+04	1.7e+04	2e+04
t_m2	1.5e+04	1.2e+04	9.6e+03	2.1e+04	2.2e+04	2.4e+04
t_m3	1.8e+04	1.4e+04	1.2e+04	2.5e+04	2.8e+04	3.8e+04
t_r1	4.8e+03	3.2e+03	2.6e+03	6.3e+03	5.4e+03	6e+03
t_r2	6.8e+03	4.5e+03	3.7e+03	9.5e+03	8.3e+03	9e+03
t_r3	9.7e+03	6.9e+03	5.4e+03	1.4e+04	1.3e+04	1.5e+04
t_s1	1.3e+04	1.1e+04	8.4e+03	1.6e+04	1.6e+04	1.7e+04
t_s2	1.6e+04	1.2e+04	1e+04	2.1e+04	2e+04	2.7e+04
t_s3	1.9e+04	1.5e+04	1.2e+04	2.5e+04	2.6e+04	3.3e+04

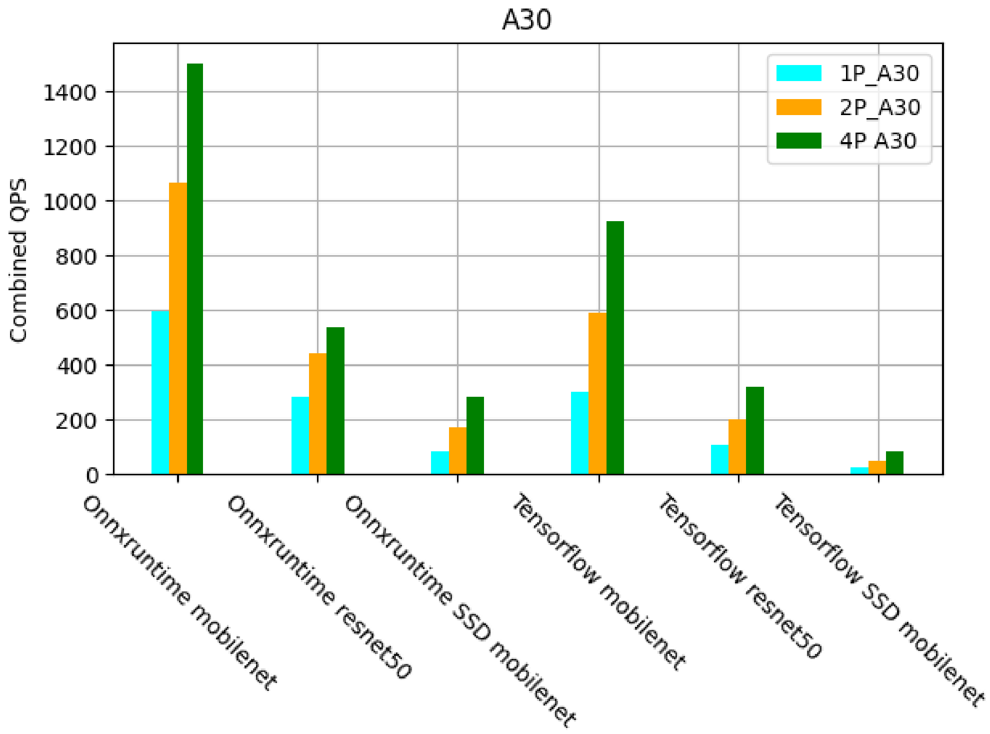
(a)

Figure 6.5: DCGM_FI_DEV_FB_USED (in MiB) for Fixed Workload

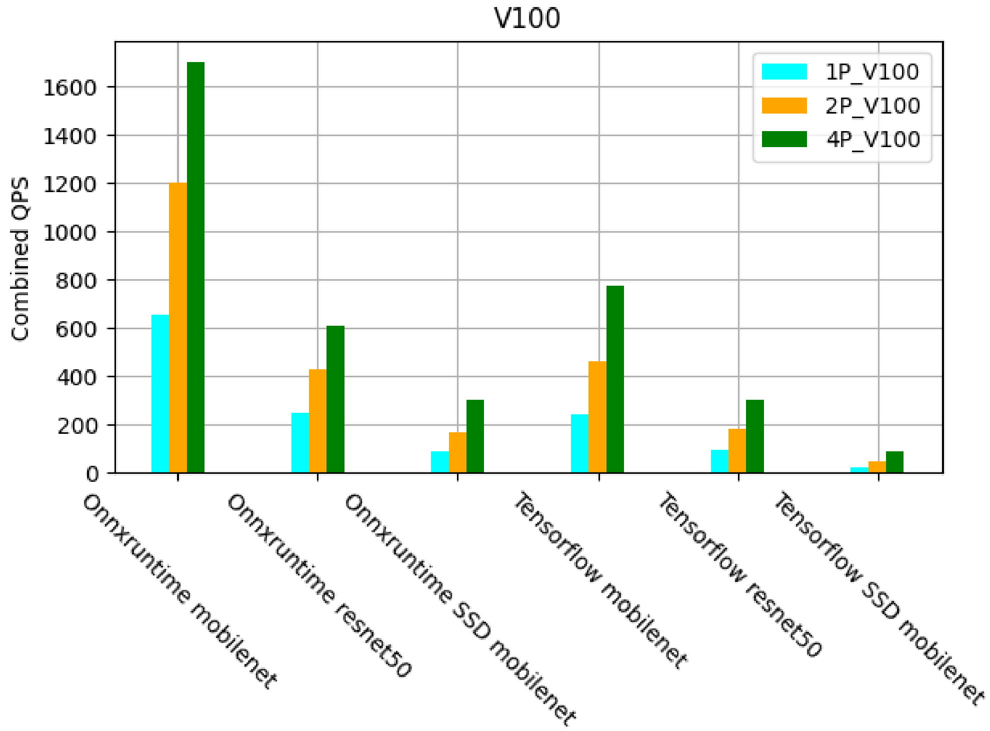
the interference, this case is the most beneficial since each instance's processors have separate and isolated paths through the entire memory system - the on-chip crossbar ports, L2 cache banks, memory controllers, and DRAM address busses are all assigned uniquely to an individual instance.

In order to be able to better realise the sensitivity of applications we decided re-execute this experience, this time by scheduling all applications on the same GPU partition. In figure 6.9 we can see the results. Here, it is crystal clear that some favoured when co-scheduled with some applications in some cases achieving only marginal interference when in other cases their performance is dramatically impaired.

In the figures 6.10 we can see the deterioration of QPS when running two GPU pods on V100 concurrently. Here we can clearly observe the negative impact of interference. For example, in figure 6.10b we see that Onnx runtime with Resnet50 achieves almost 15 queries per second when collocated with Onnx runtime Mobilenet and more than 20 queries per second when collocated with Tensorflow Resnet50. As expected the QPS for each engine is lower than when ran fully isolated on the card (e.g. for Onnx Resnet50 the QPS is approximately equal to 22). Another important note here is that intense models, e.g Resnet, tend to influence system efficiency more when collocated with other workloads.

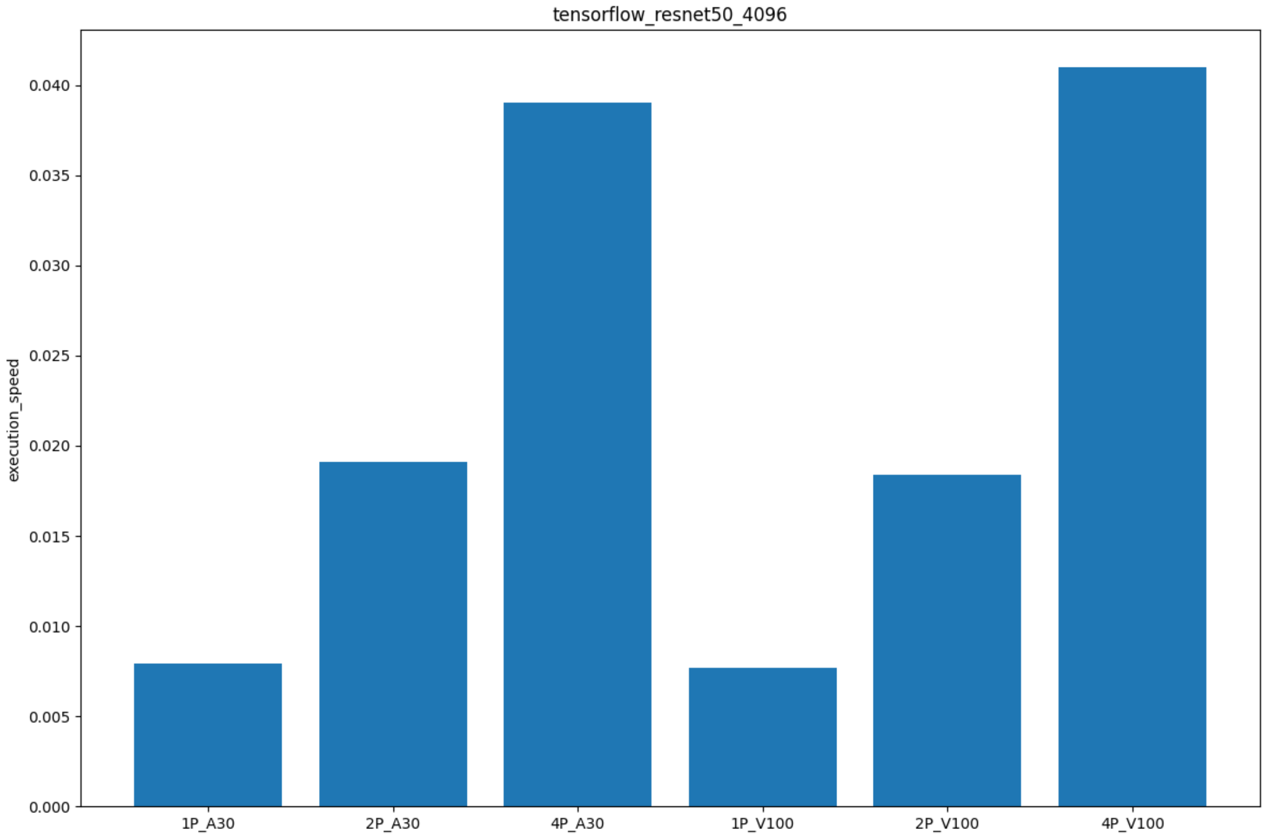


(a) A30

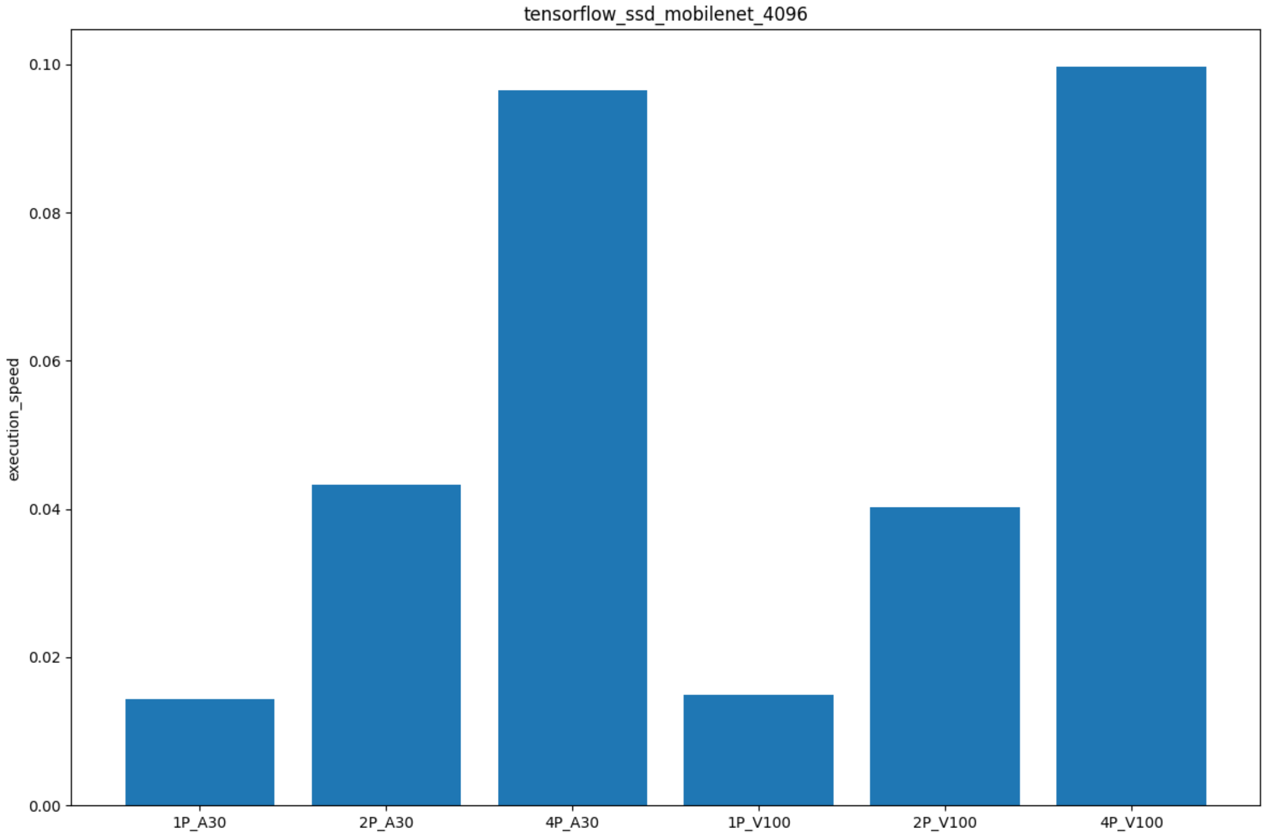


(b) V100

Figure 6.6: Combined QPS for fixed workload

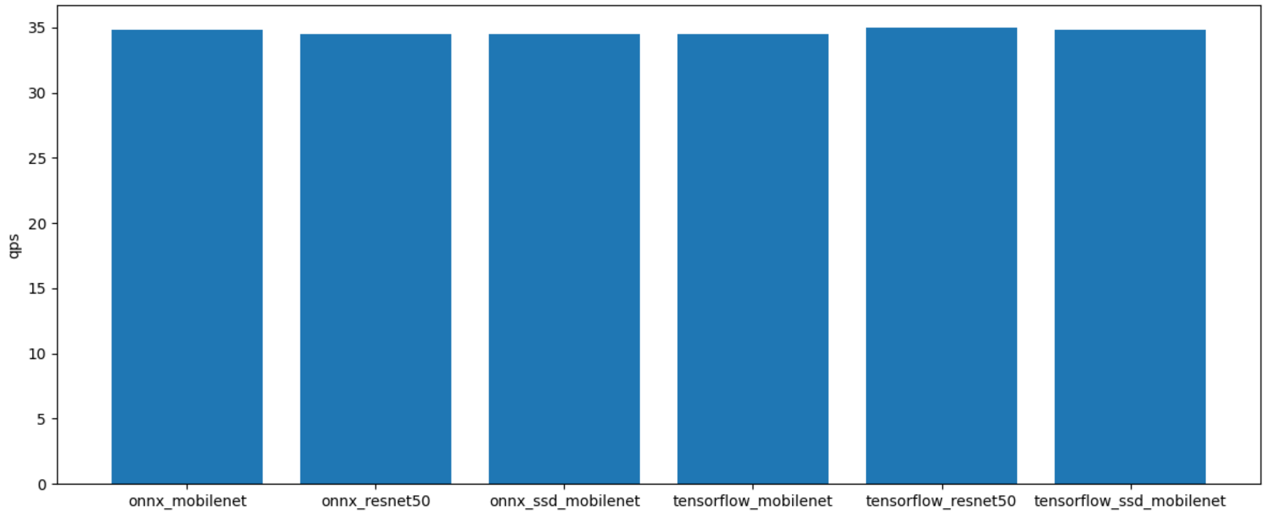


(a) Tensorflow Resnet50 Execution Speed

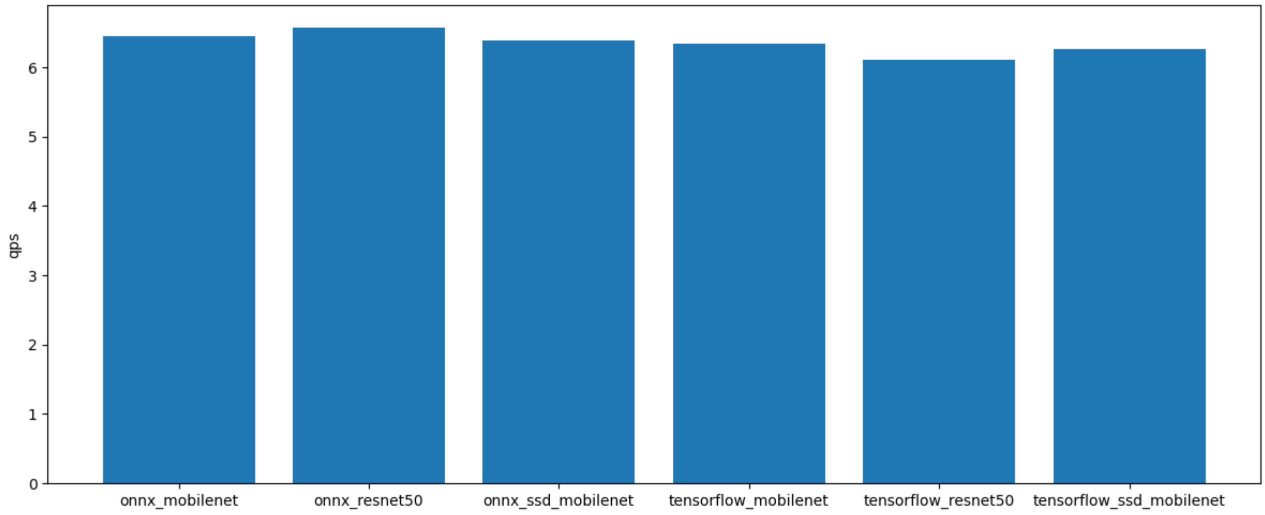


(b) Tensorflow SSD Mobilenet Execution Speed

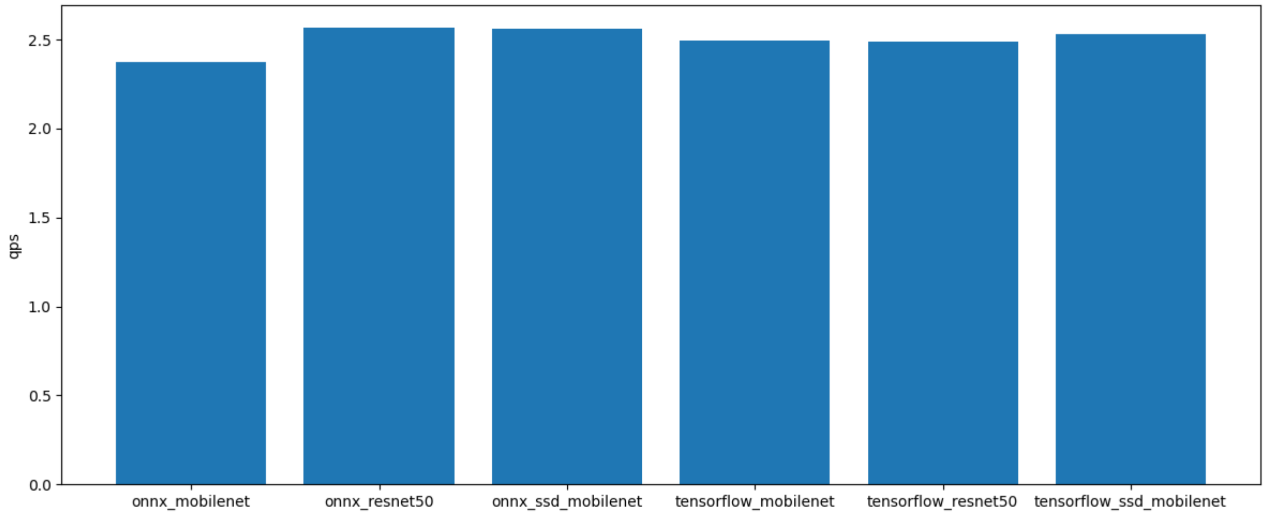
Figure 6.7: Tensorflow Renset50 and SSD mobilenet fixed workload



(a) Onnx runtime Mobilenet

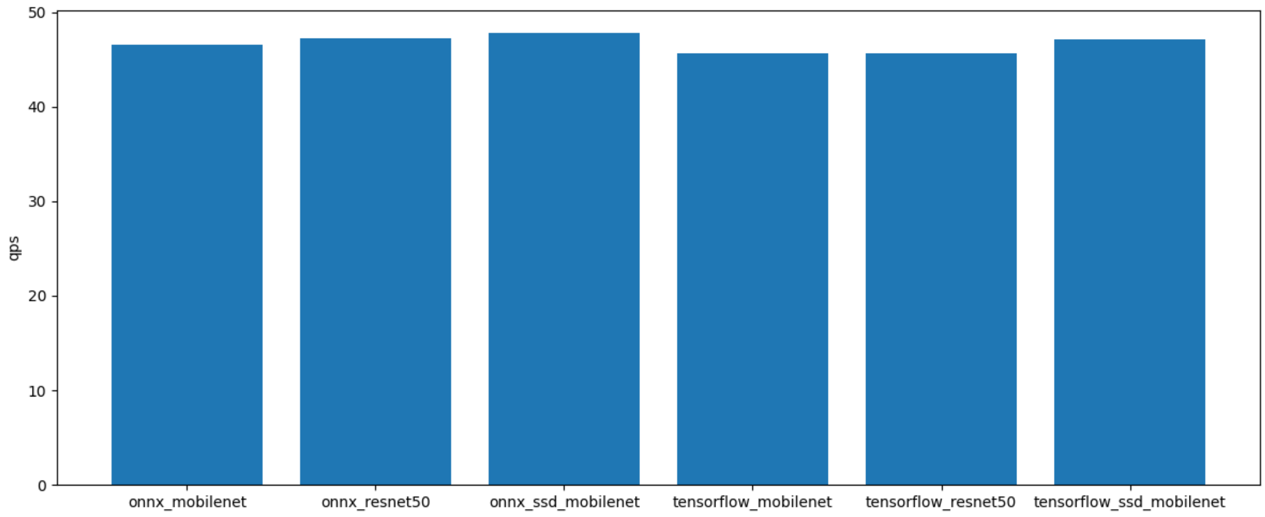


(b) Onnx runtime Resnet50

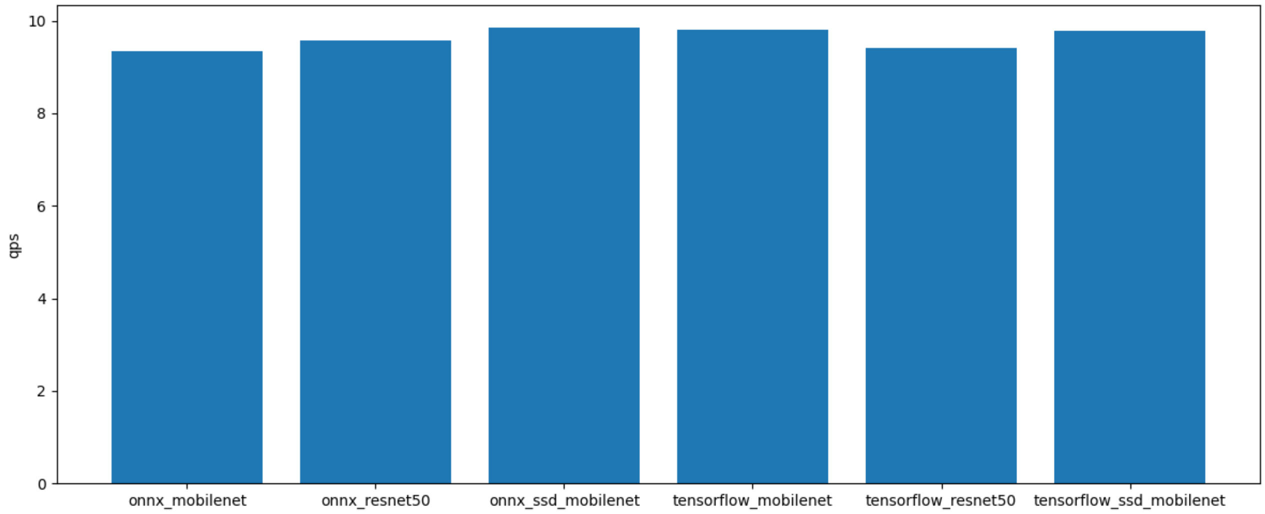


(c) Onnx runtime SSD-mobilenet

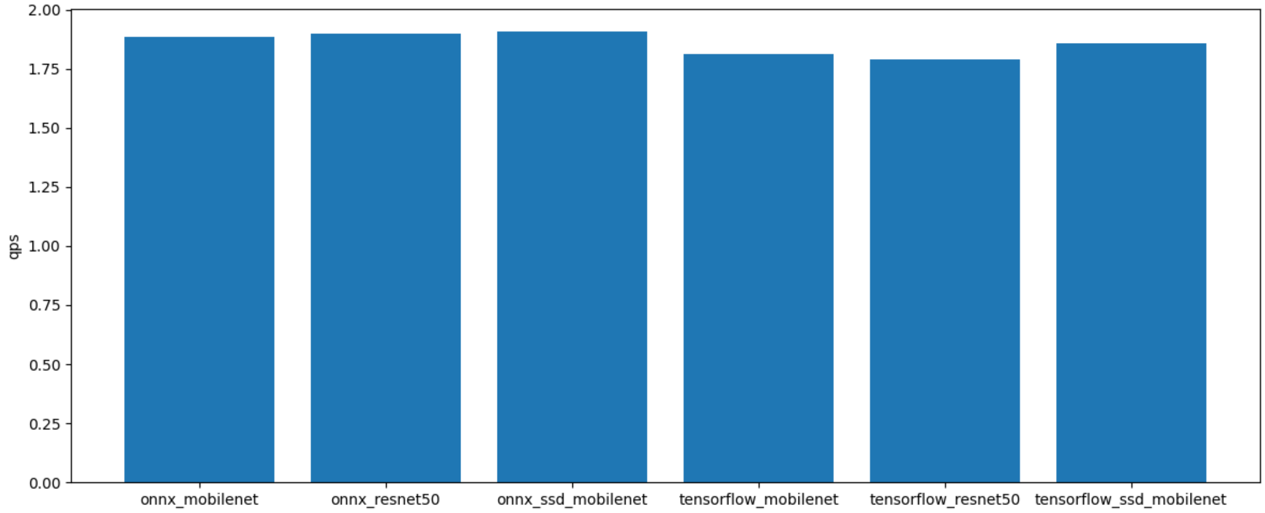
Figure 6.8: QPS, Interference. 2 Partitions A30



(d) Tensorflow Mobilenet

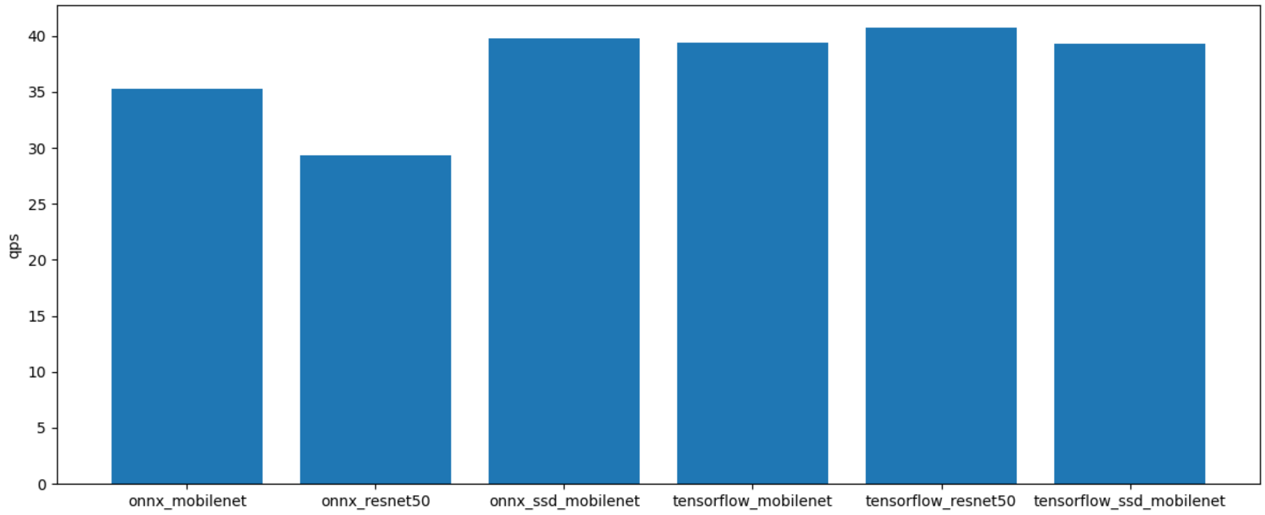


(e) Tensorflow Resnet50

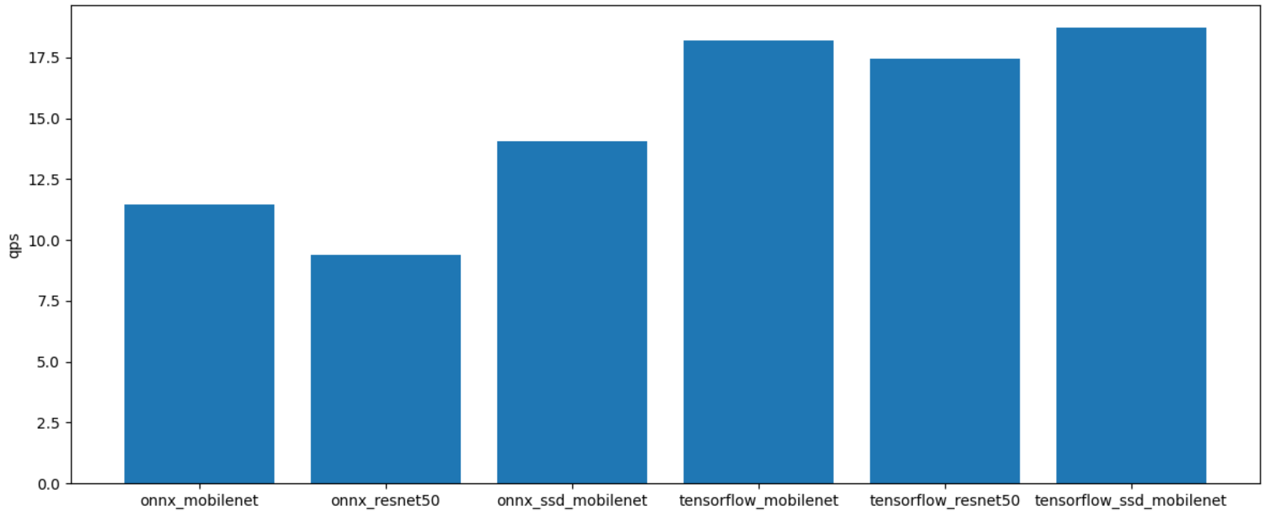


(f) Tensorflow SSD-mobilenet

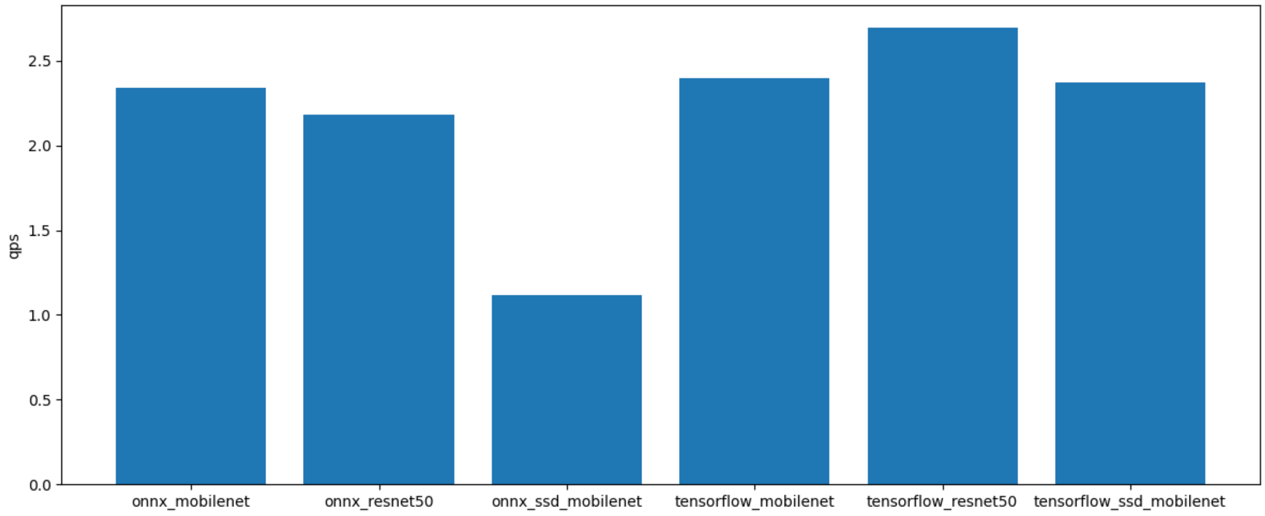
Figure 6.8: QPS, Interference. 2 Partitions A30



(a) Onnx runtime Mobilenet

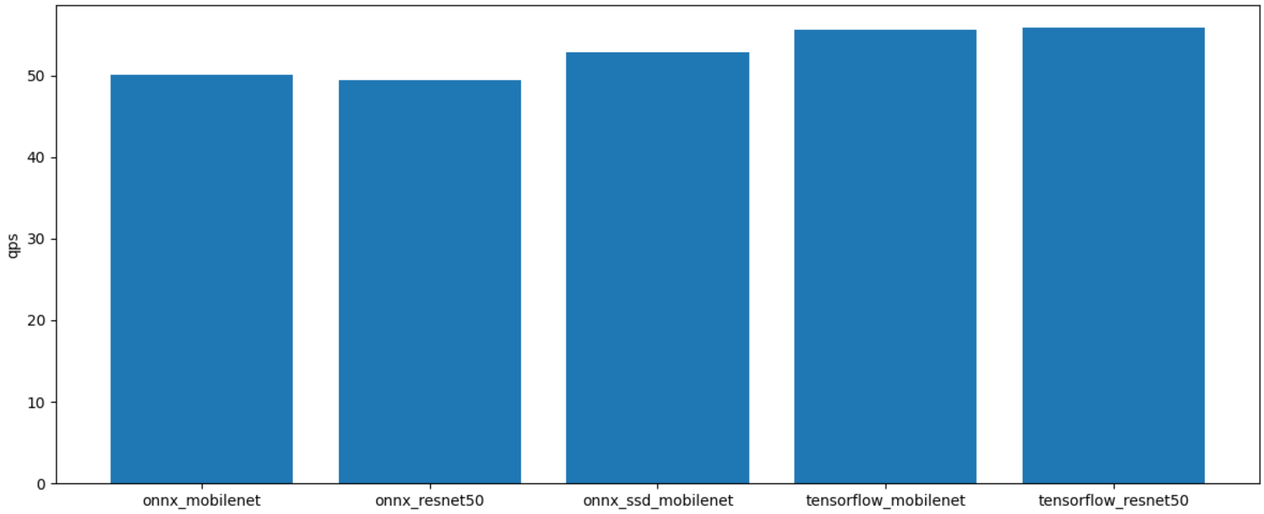


(b) Onnx runtime Resnet50

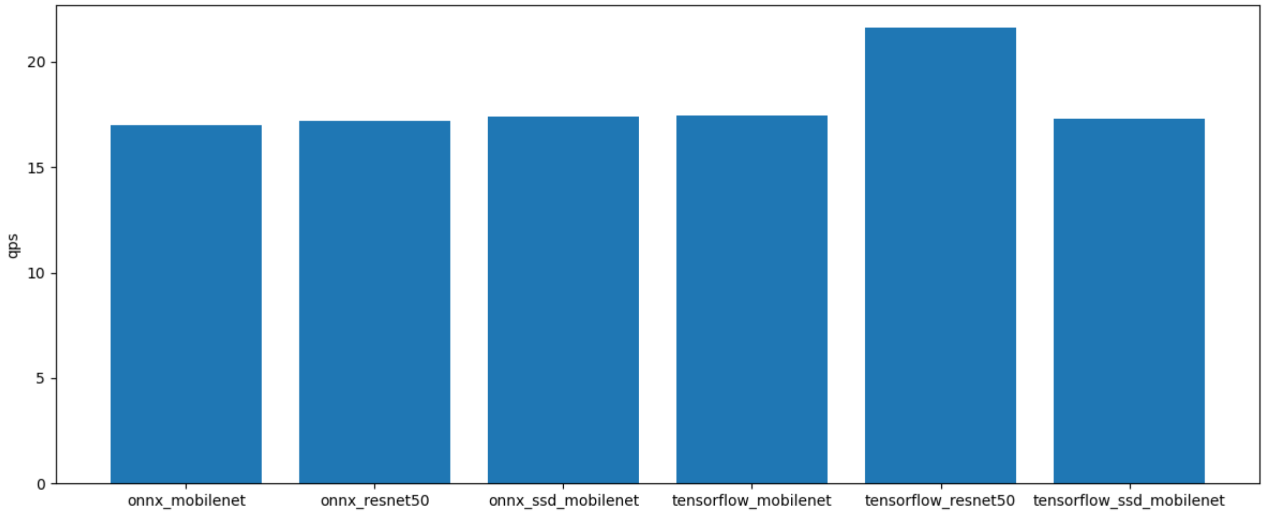


(c) Onnx runtime SSD-mobilenet

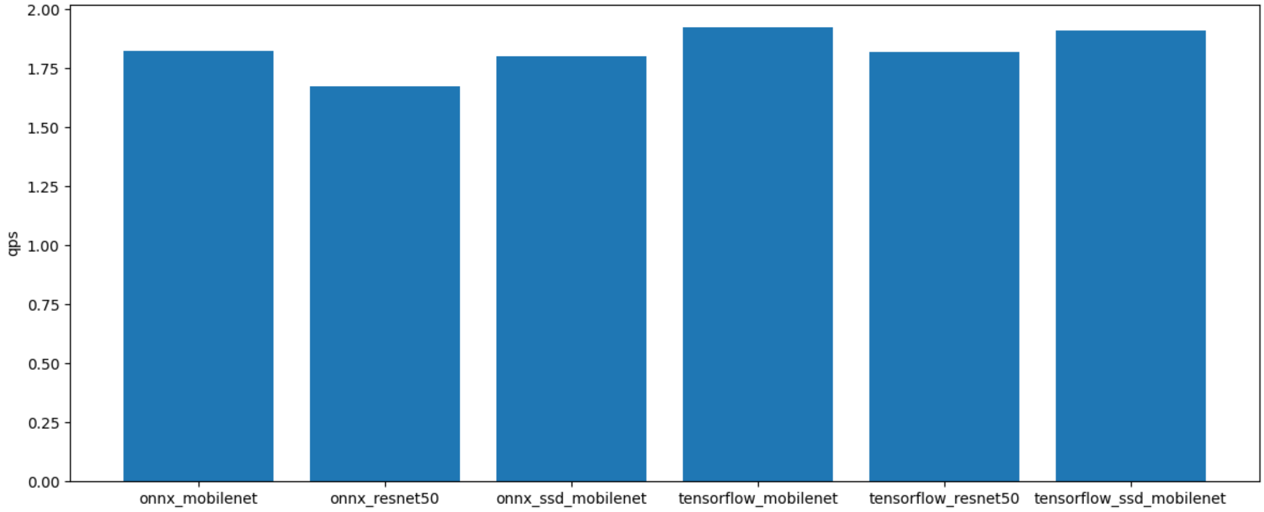
Figure 6.9: QPS, A30, 1 Partition A30



(d) Tensorflow Mobilenet

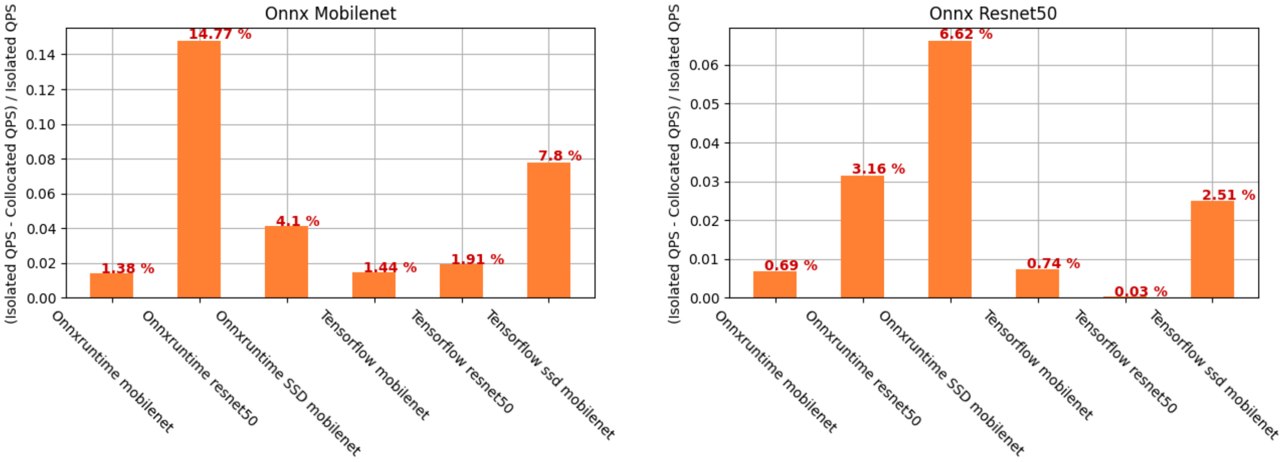


(e) Tensorflow Resnet50



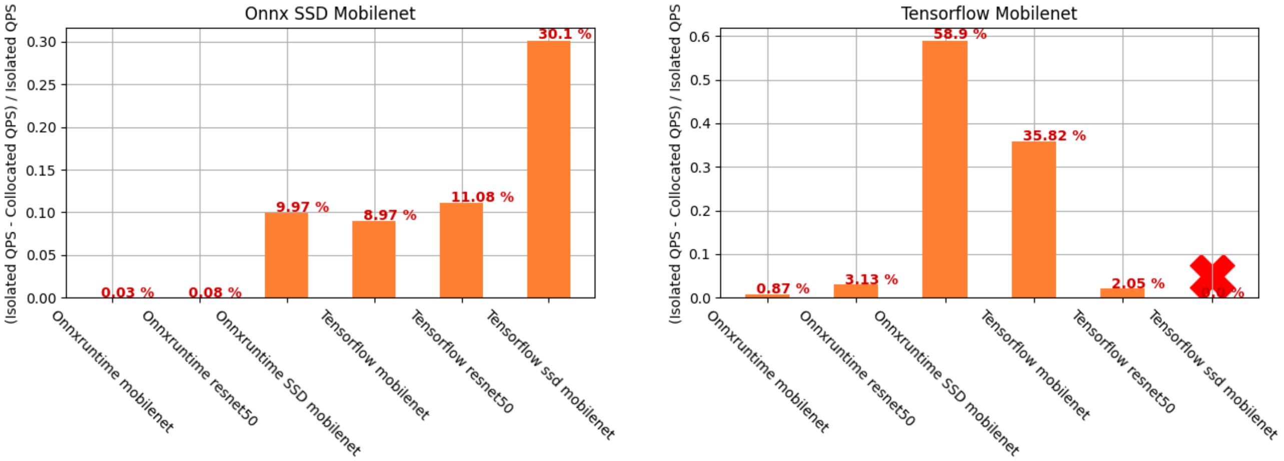
(f) Tensorflow SSD-mobilenet

Figure 6.9: QPS, A30, 1 Partition A30



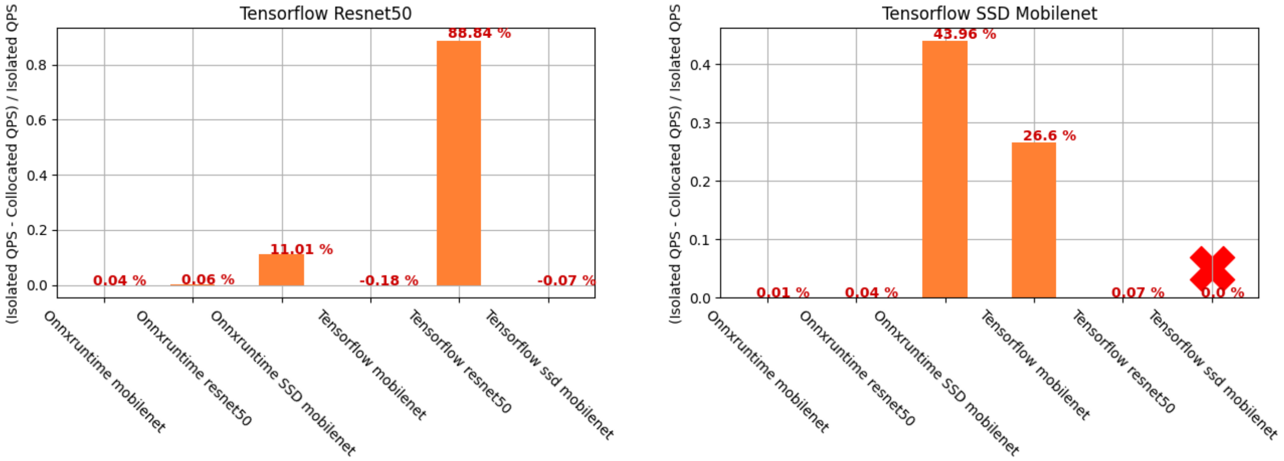
(a) Onnx Mobilenet

(b) Onnx Resnet50



(c) Onnx SSD-mobilenet

(d) Tensorflow Mobilenet



(e) Tensorflow Resnet50

(f) Tensorflow SSD-mobilenet

Figure 6.10: QPS percentage difference due to Interference on V100

Chapter 7

Recommendation Systems

7.1 What Is a Recommendation System?

A recommendation system is an artificial intelligence or AI algorithm, usually associated with machine learning, that uses Big Data to suggest or recommend additional products to consumers. These can be based on various criteria, including past purchases, search history, demographic information, and other factors. Recommendation systems are highly useful as they help users discover products and services they might otherwise have not found on their own.

Recommendation systems are trained to understand the preferences, previous decisions, and characteristics of people and products using data gathered about their interactions. These include impressions, clicks, likes, and purchases. Because of their capability to predict consumer interests and desires on a highly personalized level, Recommendation systems are a favorite with content and product providers. They can drive consumers to just about any product or service that interests them, from books to videos to health classes to clothing.

While there are a vast number of recommendation algorithms and techniques, most fall into these broad categories: collaborative filtering, content filtering and context filtering.

In this work, we will be focusing on collaborative filtering. Collaborative filtering algorithms recommend items (this is the filtering part) based on preference information from many users (this is the collaborative part). This approach uses similarity of user preference behavior, given previous interactions between users and items, recommendation algorithms learn to predict future interaction.

Matrix factorization (MF) techniques are the core of many popular algorithms, including word embedding and topic modeling, and have become a dominant methodology within collaborative-filtering-based recommendation.

7.2 Collaborative Filtering with SVD

The singular value decomposition of a matrix A is the factorization of A into the product of three matrices

$$\mathbf{A} = \mathbf{U}\mathbf{D}\mathbf{V}^T$$

where the columns of U and V are orthonormal and the matrix D is diagonal with positive real entries. The SVD is useful in many tasks. Here we mention some examples. First, in many applications, the data matrix A is close to a matrix of low rank and it is useful to find a low rank matrix which is a good approximation to the data matrix . We will show that from the singular value decomposition of A, we can get the matrix B of rank k which best approximates A; in fact we can do this for every k. Also, singular value decomposition is defined for all matrices (rectangular or square) unlike the more commonly used spectral decomposition in Linear Algebra. The reader familiar with eigenvectors and eigenvalues (we do not assume familiarity here) will also realize that we need conditions on the matrix to ensure orthogonality of eigenvectors. In contrast, the columns of V in the singular value decomposition, called the right singular vectors of A, always form an orthogonal set with no assumptions on A. The columns of U are called the left singular vectors and they also form an orthogonal set. A simple consequence of the orthogonality is that for a square and invertible matrix A, the inverse of A is

$$\mathbf{V}\mathbf{D}^{-1}\mathbf{U}^T$$

. To gain insight into the SVD, treat the rows of an (n × d) matrix A as n points in a d-dimensional space and consider the problem of finding the best k-dimensional subspace with respect to the set of points. Here best means minimize the sum of the squares of the perpendicular distances of the points to the subspace. We begin with a special case of the problem where the subspace is 1-dimensional, a line through the origin. We will see later that the best-fitting k-dimensional subspace can be found by k applications of the best fitting line algorithm. Finding the best fitting line through the origin with respect to a set of points $\{x_i \mid 1 \leq i \leq n\}$ in the plane means minimizing the sum of the squared distances of the points to the line. Here distance is measured perpendicular to the line. The problem is called the best least squares fit. In the best least squares fit, one is minimizing the distance to a subspace. An alternative problem is to find the function that best fits some data. Here one variable y is a function of the variables $x_1, x_2, ..., x_d$ and one wishes to minimize the vertical distance, i.e., distance in the y direction, to the subspace of the x_i rather than minimize the perpendicular distance to the subspace being fit to the data. Returning to the best least squares fit problem, consider projecting a point x_i onto a line through the origin. Then ${x_{i_1}}^2 + {x_{i_2}}^2 + \cdots + {x_{id}}^2 = (length \quad of \quad projection)^2 + (distance \quad of \quad point \quad to \quad line)^2$. Thus $(distance \quad of \quad point \quad to \quad line)^2 = {x_{i_1}}^2 + {x_{i_2}}^2 + \cdots + {x_{id}}^2 - (length \quad of \quad projection)^2$. To minimize the sum of the squares of the distances to the line, one could minimize

$\Sigma_{i=1}^{n}(x_{i1}^2 + x_{i2}^2 + \cdots + x_{id}^2)$ minus the sum of the squares of the lengths of the projections of the points to the line. However, $\Sigma_{i=1}^{n}(x_{i1}^2 + x_{i2}^2 + \cdots + x_{id}^2)$ is a constant (independent of the line), so minimizing the sum of the squares of the distances is equivalent to maximizing the sum of the squares of the lengths of the projections onto the line. Similarly for best-fit subspaces, we could maximize the sum of the squared lengths of the projections onto the subspace instead of minimizing the sum of squared distances to the subspace.

7.3 Multivariate feature imputation

Multiple imputation [76] is the method of choice for complex incomplete data problems. Missing data that occur in more than one variable presents a special challenge. Two general approaches for imputing multivariate data have emerged: joint modeling (JM) and fully conditional specification (FCS), also known as multivariate imputation by chained equations (MICE). Schafer (1997) developed various JM techniques for imputation under the multivariate normal, the log-linear, and the general location model. JM involves specifying a multivariate distribution for the missing data, and drawing imputation from their conditional distributions by Markov chain Monte Carlo (MCMC) techniques. This methodology is attractive if the multivariate distribution is a reasonable description of the data. FCS specifies the multivariate imputation model on a variable-by-variable basis by a set of conditional densities, one for each incomplete variable. Starting from an initial imputation, FCS draws imputations by iterating over the conditional densities. A low number of iterations (say 10–20) is often sufficient. FCS is attractive as an alternative to JM in cases where no suitable multivariate distribution can be found. In the statistics community, it is common practice to perform multiple imputations, generating, for example, m separate imputations for a single feature matrix. Each of these m imputations is then put through the subsequent analysis pipeline (e.g. feature engineering, clustering, regression, classification). The m final analysis results (e.g. held-out validation errors) allow the data scientist to obtain understanding of how analytic results may differ as a consequence of the inherent uncertainty caused by the missing values. The above practice is called multiple imputation. Scikit-learn provides the implementation of and "IterativeImputer" inspired by the R MICE package, but differs from it by returning a single imputation instead of multiple imputations. However, IterativeImputer can also be used for multiple imputations by applying it repeatedly to the same dataset with different random seeds.

IterativeImputer class models each feature with missing values as a function of other features, and uses that estimate for imputation. It does so in an iterated round-robin fashion: at each step, a feature column is designated as output y and the other feature columns are treated as inputs X. A regressor is fit on (X, y) for known y. Then, the regressor is used to predict the missing values of y. This is done for each feature in an iterative fashion, and then is repeated for max_iter imputation rounds. The results of the final imputation round are returned.

7.4 Flexibility of Multivariate feature imputation

There are many well-established imputation packages in the R data science ecosystem: Amelia, mi, mice, missForest, etc. missForest is popular, and turns out to be a particular instance of different sequential imputation algorithms that can all be implemented with Multivariate feature imputation by passing in different regressors to be used for predicting missing feature values. In the case of missForest, this regressor is a Random Forest.

7.5 Evaluation

In this section we evaluate the perfomance of Collaborative Filtering with SVD and Multivariate Imputation techniques. For the first case we used numpy.linalg library for the decomposition, we used euclidean similarity for similarity measure. Euclidean similarity is equal to

$$distance \quad based \quad similarity = \frac{1}{(1 + d(i_1, i_2))}$$

$d(i_1, i_2) = \sqrt{\sum_{j=1}^{n} (i_{1_j} - i_{2_j})^2}$, where n is equal to the dimension of the items.

For the case of multivariate imputation we used IterativeImputer from the sklearn.impute library and set max_iter to 10. max_iter is the maximum number of imputation rounds to perform before returning the imputations computed during the final round. For this experiment we used a small rating dataset from 18 users for 6 items.

In figure 7.1 we can see the mean squared error for SVD and IterativeImputer in a common plot. Here, we use the leave-one-out cross validation technique. This means that in each iteration of the experiment we used a single row as the test dataset and the rest to fit our model. From the row used for testing we progressively delete elements and observed the performance of each model. This was repeated ten times for each row and averaged among for all results with the same row and number of missing values. Here, x-axis represents the

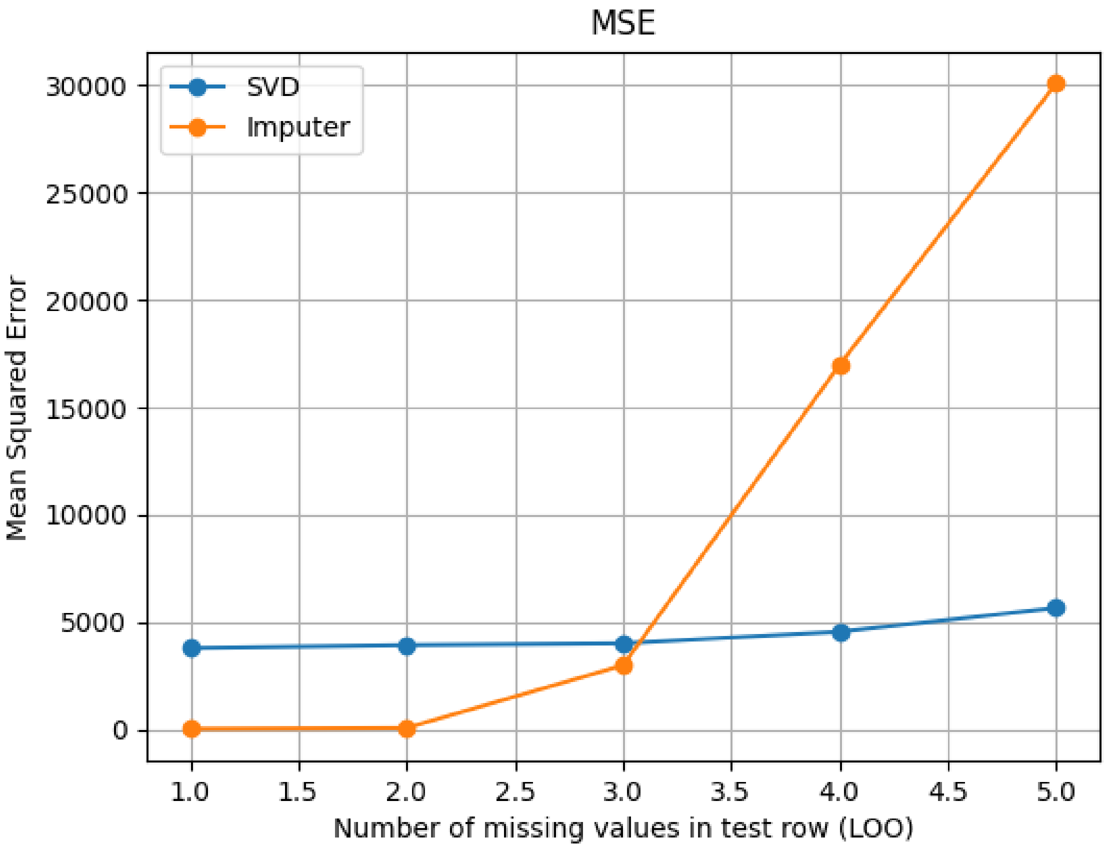


Figure 7.1: Imputer and SVD Mean Squared Errors

number of missing values and y-axis the average MSE for all rows. What we understand from this figure is that the sparser the matrix is the better the results from SVD are and the opposite for Multivariate Imputation. In fact, results from SVD have very small variance and have a mean value close to 5000 when imputer performs very good for small number of missing values but its performance degrades very quickly when increasing the number more 50%. This could be due to the fact that Multivariate Imputation uses Markov chain Monte Carlo (MCMC) which is useful when the multivariate distribution is a reasonable description of the data.

Chapter 8

Predictive inference serving for multi-tenant GPU clusters

In the present work, we developed a scheduler for multi-tenant GPU clusters using Kubernetes (Chapter 4) and Collaborative Filtering (CF) (Chapter 7). For the creation of the scheduler we used the Kubernetes Scheduling Framework (Section 4.8). The scheduling framework is a pluggable architecture for the Kubernetes scheduler. It adds a new set of "plugin" APIs to the existing scheduler. Plugins are compiled into the scheduler. We extended the Score plugins and the PostBind plugins in order to implement our scheduling logic. The first barrier you face when creating a scheduler for Kubernetes is the fact that Kubernetes does not natively advertise GPU resources. Namely, the only vulgate way to utilize GPU resources is by using the NVIDIA device plugin which advertises GPU devices as a whole. Therefore, fine-grained allocation of GPU resources such as memory and SMs (streaming multiprocessors) is not an option. To overcome this issue we created a daemonset classified the GPU nodes (nodes containing GPUs) according to their resources. Then, we saved these data in a Redis deployment which was running in the master node in order to optimize the query time and minimize the scheduling overhead. For the collaborative filtering (prediction) we needed a procedure to compute the expected QPS (queries per second). This could have been a separate process in the code created for the scheduler or a separate container in the deployment for the scheduler, however we decided that, since the dataset could be relatively large or the inference process compute intense, we should offload the whole procedure into a separate node. For this purpose we used Cheetara, our second non-GPU node apart from master.

8.1 Redis

Redis (Remote Dictionary Server) is an in-memory data structure store, used as a distributed, in-memory key–value database, cache and message broker, with optional durability. Redis supports different kinds of abstract data structures, such as strings, lists, maps, sets, sorted sets, HyperLogLogs, bitmaps, streams, and spatial indices. In this project, Redis was used to facilitate the storage of temporary data regarding the state of the cluster (GPU UUIDs, assigned pods to GPU partitions etc). The Redis deployment was marked with a soft affinity constraint in order to be assigned on the master node so that the scheduler can access the data even faster.

8.2 Exporter

In order to monitor the characteristics (such as partitions' UUIDs) of each GPU in the cluster and correlate them with the corresponding node we needed to create a Kubernetes resource acting as an intermediary between the scheduler and GPUs. For this purpose we created a daemonset. Each pod of the daemonset runs a container which first decides whether the node is able to host GPU applications and then by running a simple CUDA application it exports the GPU UUIDS as well as a few metrics and saves them in Redis. The exporter then watches the output of the CUDA application for changes. Whenever a restart or a crush happens, exporter immediately pulls the new data and updates Redis.

8.3 Predicting and Scoring

In order to produce recommendations and rate the nodes based on the predictions for performance and interference we created a deployment with a soft affinity constraint in order to offload it on Cheetara (our second non-GPU node). The deployment incorporates a recommendation system using CF and SVD (singular value decomposition) and is accessible by the cluster using gRPC. Utilizing the data we have collected from our characterization (Chapter 6) we created 2 datasets. The first one is a sparse $n \times m$ matrix used to compute the expected QPS (queries per second) when the application is running isolated on a GPU enabled partition. n corresponds to the applications (equivalent to users for traditional recommendation system approaches) and m corresponds to the available configurations (items). Each configuration is described by

1. GPU
2. MIG partition (if available) and
3. MPS device limits (if available).

When a query comes to the recommender, the recommender has to return the whole row corresponding to this application. If the row has missing values, it uses SVD to extract the latent features from the matrix and then computes the expected values based on similarities between items. We call the above procedure $isolated(x, p)$, where x is the application, p is the configuration (GPU partition) and isolated(x, p) is the QPS x can achieve when ran isolated on partition p.

The second dataset is used to compute the interference generated among the pods. It is a matrix $A_{n \times n}$ where n corresponds to the applications and A_{ij} is equal to the interference suffered (QPS degradation) by application i and produced by application j. The prediction for missing values are exactly as above. We define the partition's interference as the sum of interference suffered from the collocated pods as follows

$$total\_interference(x, X, p) = \sum_{y \in X, y \neq x} interference(x, y, p)$$

, where X is the set of collocated pods and $interference(x, y, p)$ is equal to the QPS degradation returned from the above recommendation system for pod x when collocated with pod y on partition p.

In order to score the node we need a function to describe the distance between the demanded QPS value (Service Level Objective) and the expected value. We define the system's error metric as the euclidean distance of the requested and provided QPS, normalized by the first

$$err(x, X, p) = \frac{|SLO(x) - expected\_value(x, X, p)|}{SLO(x)}$$

, where

$$expected\_value(x, X, p) = isolated(x, p) - total\_interference(x, X, p)$$

expected_value corresponds to the expected QPS for application x when collocated with the set of pods X on partition p and $SLO(x)$ refers to the QPS the user demanded. To define the score we need some function that declines when the above distance gets larger. Also, for fixed distance, positive values must be preferable compared to negative values since we must cover the users' needs. For this purpose we used two similarity measures. The first similarity measure is the following

$$score_1(x, X, p) = \frac{1}{1 + (err(x, X, p) + 1)^2}$$

and the second one

$$score_2(x, X, p) = \frac{1}{1 + err(x, X, p)}$$

, where obviously err(x) can take any positive value. $score_1$ declines much faster than $score_2$ as err(x) increases and also both functions' set of destinations are $[0, 1]$. In figure 8.1 we can see the graphs of the functions.

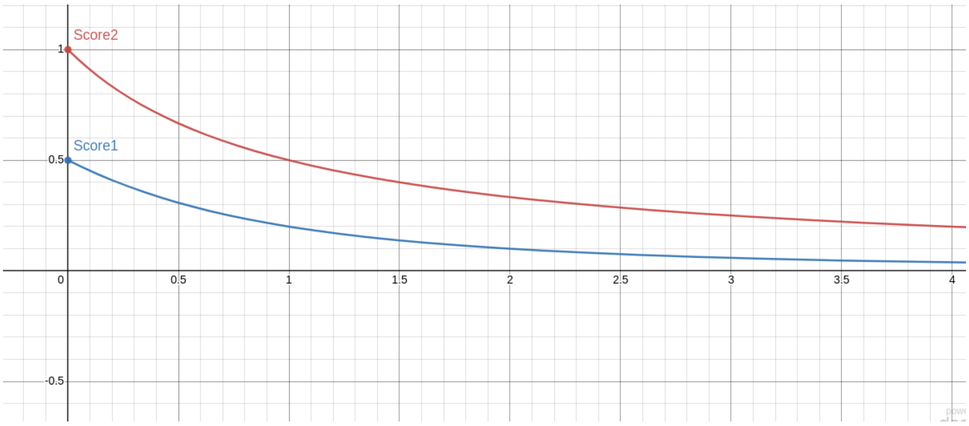


Figure 8.1: Scores comparison

8.3.1 Score

Let X be the set of pods running on partition p, including the pod that is being scheduled. We define the sets X_{neg} and X_{pos} such that

$$X_{neg} = \{x | x \in X, expected\_value(x, X, p) - SLO(x) < 0\}$$

is the subset of pods in X that can not cover the demanded SLO whereas

$$X_{pos} = \{x | x \in X, expected\_value(x, X, p) - SLO(x) \geq 0\}$$

is the subset of X that can cover the demanded SLO. Obviously X_{neg} and X_{pos} are a partition of X, namely

$$X_{neg} \cup X_{pos} = X \ and \ X_{neg} \cap X_{pos} = \emptyset$$

The score of the partition will be a weighted average of the scores of all pods. These scores must also be between 0 and 100 since those are the only accepted scores by the scheduling framework.

In the negative case, the score is more important than in the positive case. This is because it is preferable to provide more resources than the user demands than to compromise the user's request. For this purpose, in the negative case we preferred the $score_1(x), x \geq 0$ which rapidly declines as distance $err(x)$ increases, whereas in the positive case we preferred $score_2(x), x \geq 0$ which declines slower as $err(x)$ increases.

The final score for partition i is computed as a linear combination of the above:

$$score(X,p) = 100 \times (1 - util(p)) \times (k \times \frac{1}{|X_{neg}|} \times \sum_{x \in X_{neg}} score_1(x, X, p) + (1-k) \times \frac{1}{|X_{pos}|} \times \sum_{x \in X_{pos}} score_2(x, X, p))$$

where $util(i)$ is the instantaneous partition's value for GPU utilization returned from Prometheus and $k = \frac{|X_{neg}|}{|X|}$ defines the weight of the negative instances in the total procedure. Finally, the score function's set of destination is the interval [0,100].

If $X_{neg} = \emptyset$, then the term with $score\_1$ is omited, else if $X_{pos} = \emptyset$, then the term with $score\_2$ is omited. In the above procedure we described how we can choose a partition which best suits our needs. However, the ScorePlugin only gives us the option to score the whole node which can hold multiple partitions (e.g. MIG). Let P be the set of all available GPU partitions on the node and X_p be a set of pods running on partition p. We score node n with the maximum score among the scores of partitions p

$$final\_score(n) = \max_{p \epsilon P} score(X_p, p)$$

where

$$\bigcup_{p \in P} X_p = X$$

If the node does not host any other GPU applications at the time the pod arrives, then we reconfigure the GPU such that the score is maximized (if $\underset{p}{\text{argmax}}\, score(X, p) \neq current\ configuration$). The process of reconfiguration lasts an approximate of 5 seconds which is insignificant compared to the duration of the workloads which last few minutes. Also, when the GPU is not empty, we decided not to reconfigure it, since the running pods would be interrupted and therefore would violate their objectives.

8.4 PostBind and CUDA_VISIBLE_DEVICES

In the previous section we described how we can choose a certain node, but not how to bind the pod on the desired partition. For this purpose we utilized an environment variable used by CUDA developers to control GPU visibility for CUDA applications. CUDA_VISIBLE_DEVICES is an environment variable used to specify which NVIDIA GPUs should be used by a CUDA-enabled application. When running a CUDA program on a system with multiple GPUs, CUDA_VISIBLE_DEVICES can be used to control which GPUs are visible to the program. By default, all GPUs are visible, but this variable can be set to a comma-separated list of GPU device IDs to limit the visible devices. For example, setting CUDA_VISIBLE_DEVICES=0,1 would limit the program to only see the first and second GPUs. This can be useful for a variety of reasons, such as limiting the resources used by a program or allowing multiple programs to run simultaneously on different GPUs.

The PostBind plugin is called after a Pod is successfully bound. In order for the pod to be bound on a specific partition we created an empty ConfigMap and appended it in the pod's environment. Within the scheduler, we hold an in memory data structure describing the optimal partition for each pod and each node and after the scheduling is finished we populate the ConfigMap with the

CUDA_VISIBLE_DEVICES environment variable, with a value equal to the UUID of the selected partition.

We followed the exact same logic for computing and appending the optimal MPS device limits for memory and compute capacity utilizing CUDA_MPS_PINNED_DEVICE_MEM_LIMIT and CUDA_MPS_ACTIVE_THREAD_PERCENTAGE environment variables respectively.

Chapter 9

Evaluation

In this chapter, we use our experimental infrastructure to evaluate our scheduling mechanism using a set of different experiments. We implement a set of different scheduling logics and compare them to our system.

9.1 Description

In order to evaluate our scheduler we executed a sequence of experiments. Each experiment had the purpose of illustrating a different aspect of the performance of the scheduler. An experiment consists of a set workloads (pre-trained model, backend, dataset, scenario, number of queries etc), the objectives (SLO), a batch size as well as a distribution describing the applications' arrivals over time. In each experiment the exact same workloads, batch size and arrivals are fed to the available schedulers. In the following experiments, the workloads are deployed in batches of 8 and within the batches each workload has a random delay of few seconds in order to add some unpredictability. We decided that the batch size be 8, because we only have 2 GPUs in the cluster and if the batch size got larger, it would result in failures due to insufficient resources and otherwise the cluster would be underutilized. However, later on, we will examine the cluster's response to different batch size and compare the benefits of each one. We executed two series of experiments. In the first series of experiments we sent 20 pods over a period of roughly fifteen minutes. In the second one, we decided to stress the schedulers with the double number of pods (40) in the same time interval in order to examine the behaviour of the scheduler under high pressure. Both of the experiments were repeated two times. The first time we chose SLOs for each workload to be in the interval $[0.8 \times SLO_4, 1.2 \times SLO_4]$ where SLO_4 is equal to the expected value of QPS when the workload is running isolated on the $\frac{1}{4}$ of the GPU (in the case of A30 1 out of 4 partitions, in the case of V100 limited to the 25% of the GPU resouces using MPS) by itself. Namely, we picked Low SLOs. In the second case, we picked SLOs in the interval $[0.8 \times SLO_1, 1.2 \times SLO_1]$, where SLO_1 is equal to the expected value of QPS when the workload is running isolated on the whole GPU. We chose to divide the experiments in this way, so that we are able to to examine the ability of our scheduler to partition MIG enabled GPUs, as well as, examine its ability to handle bursts and meet users needs as they get more and more demanding.

In each experiment we measure the following QoS and GPU resource utilization metrics using the Kubernetes API and the GPU monitoring mechanism.

- □ QoS Metrics
  1. Execution Duration
  2. Number of errors occurred
  3. Number of SLO violations
- □ GPU Resource Utilization Metrics
  1. Average GPU utilization (Ratio of time the graphics engine is active (in %))
  2. Average Power Consumption (W)
  3. Average Energy Consumption (J)

9.2 Schedulers

In this section we analyze the main characteristics of the schedulers we created in order to compare our results.

9.2.1 Min

First, we simulated a greedy scheduler which allocates the least possible resources to each incoming workload. It accomplishes this by partitioning all MIG enabled GPUs in as many partitions as possible and allocating one to each application. Furthermore, for each non MIG enabled GPU it used the MPS (multi process service) capability to limit the resources of each application to 25%. The reason we considered 25% to be small enough is because we observed that stricter limits incurred a high percentage of failures due to insufficient memory and/or compute capacity. Also, the scheduler used the NVIDIA Device Plugin in order to tie applications on GPU devices, meaning that, as explained in 4.7.3, the cluster's extended resources do not support sharing between applications and therefore when all GPUs are busy in the cluster, incoming applications must wait in a queue before starting their execution. We used this scheduling logic in order to compare our scheduler to one which exploits the well-known practice of using the NVIDIA Device Plugin. We used this scheduling logic in order to compare our scheduler to one which greedily exploits MPS and MIG technologies to share GPU resources among applications.

9.2.2 Max

The next scheduler used for evaluation is a greedy scheduler which allocates the most possible resources to each incoming workload. In this experiment all GPUs are used as a whole, meaning that MIG enabled GPUs are not partitioned and MPS is not used. Also, similarly to the previous scheduler, the scheduler used the NVIDIA Device Plugin in order to tie applications on GPU devices. We used this scheduling logic in order to compare our scheduler to one which exploits the well-known practice of using the NVIDIA Device Plugin.

9.2.3 Round Robin

In the last case, we decided to simulate a scheduling logic which, after partitioning all GPUs in the smallest possible chunks, it scheduled pods on each of the chunks in a round robin fashion, also supporting collocation between applications utilizing both MIG and MPS technologies. For example, if it were to schedule two applications in the same partition it would apply 50% MPS limits to each one of them, if it were to schedule three application it would apply 33% MPS limits to each one of them etc. The reason why we chose to simulate this scheduler is so that we have a baseline to assess the ability of our scheduler to deal with interference compared to a interference-oblivious scheduler.

9.3 Schedulers Comparison

In the following graphs, our scheduler is designated by "My-sched", Min scheduler by "min", Max scheduler by "max" and Round Robin scheduler by "round robin".

In figure 9.1 we can see the percentage difference between the expected QPS (SLO) and the achieved QPS, namely

$$\frac{|Achieved \quad QPS - SLO|}{SLO}$$

Obviously, our objective is to minimize this difference. We can see that "min" and "round robin" schedulers perform better when the SLO demands are low since they offer less resources to pods and the achieved QPS's are expected to be lower while for "max" scheduler the opposite applies. Here we can clearly see that our scheduler performs better in all of the case regardless of the SLO demands. The average percentage difference among all the conducted experiments of our scheduler is 2.92x lower compared to "min" scheduler, 3.03x lower compared to "round robin" scheduler and finally 3.62x lower compared to "max" scheduler. This metric is important because apart from demonstrating our schedulers ability to cover the users' needs (which will also be validated by the rest of the metrics) it also ensures that our scheduler covers their needs in an optimal way, without overpovisioning resources to do so and therefore has more free resources to host newly deployed applications.

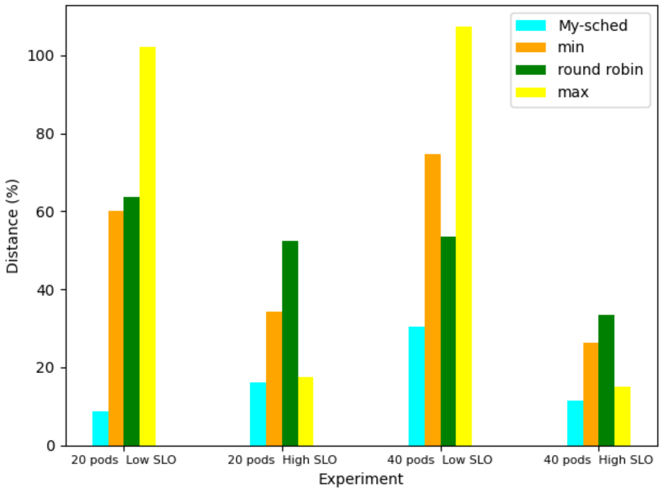
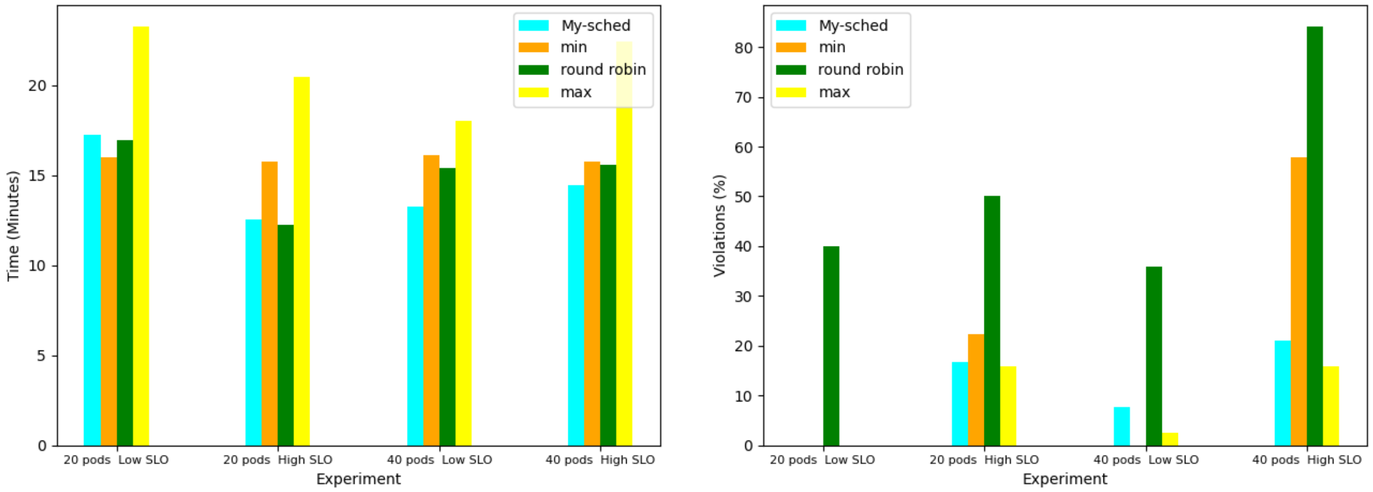


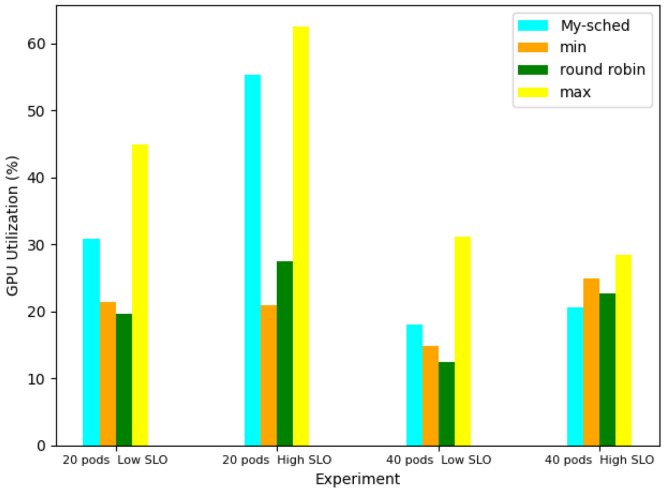
Figure 9.1: QPS-SLO percentage difference

In figure 9.2a we can see the total duration of the experiment on each of the schedulers. We can see that as expected the "max" scheduler takes the most time to complete in all of the experiment since it uses the NVIDIA device plugin and does not support collocation. With regard to the "round robin" scheduler, we can see that the duration of the experiments in most cases is lower than our scheduler's. However, the reason why this happens is because as we can see in figure 9.2b the violations of this scheduler as well as the failures (9.3) are really high in all cases. As far as the "min" scheduler is concerned, as pointed out before, the "min" scheduler performs better in low SLOs. In this case, we can see that "min" scheduler has lower duration in the case of 20 pods and low SLOs. However, as we can see in figure 9.1, the QPS achieved by the "min" scheduler in average is much higher than the demanded QPS, therefore meaning that provided resources are higher and thus the whole system's versatility is significantly degraded. Precisely, our scheduler is 1.1x faster than "min", 1.04x faster than "round robin" and 1.5x faster than "max" scheduler. The total number of pods whose QoS constraints were violated for our scheduler is equal to 13 out of 120 (including failures).



(a) Duration

(b) Violations



(c) Utilization

Figure 9.2: QoS Metrics

In figure 9.2c we can see the GPU utilization for each experiment and for each of the schedulers. We can see that our scheduler achieves better resource utilization comparing to "min" and "round robin" schedulers but worse compared to "max" scheduler. Why this happens, is because "max" scheduler always provide full GPU resources for each application leading to higher utilization but also leading to 1.4x the energy consumption of our scheduler and 1.5x the average duration.

At this stage, we will take a look at the failures that occurred in the case of our scheduler and "round robin" scheduler. The reason why we ignore the cases of "max" and "min" schedulers is because in these cases the number of errors is insignificant due to the fact that NVIDIA device plugin over-provisions resources, making it unlikely for the memory and/or compute capacity to not be sufficient for the pod. In figure 9.3 we can see the number of errors occurred for our scheduler as well as the "round robin" scheduler. The total number of errors for our scheduler is 3.4x lower compared to "round robin" (5 out of 120 applications).

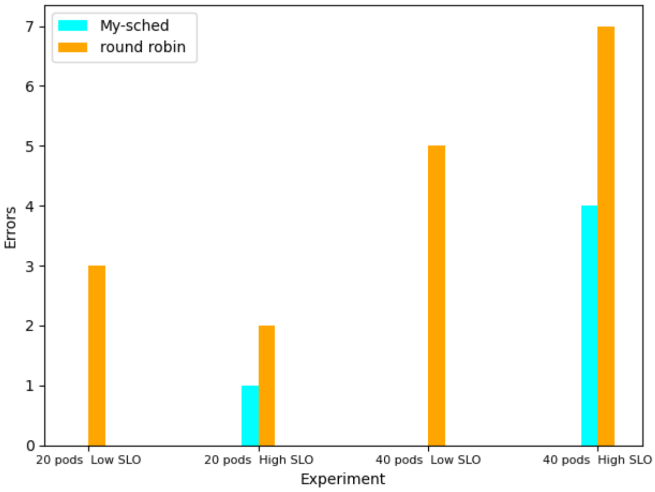
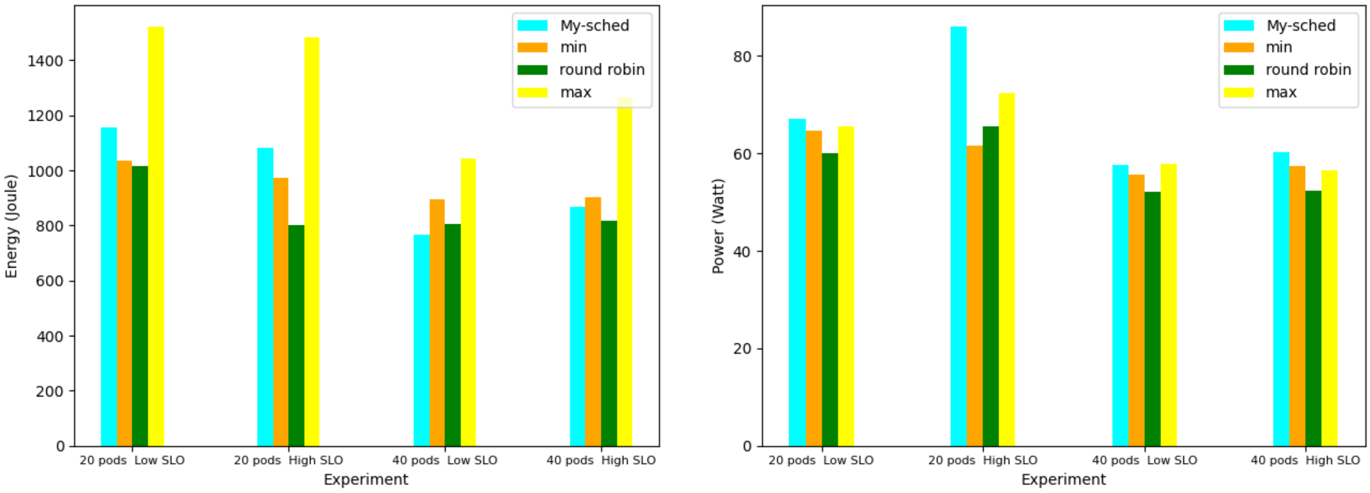


Figure 9.3: Errors

In the case of energy consumption (figure 9.4) our scheduler performs slightly worse than "min" and "round robin" schedulers (1.01x and 1.1x higher energy consumption respectively) while consuming 1.4x less energy compared to the "max" scheduler. However, the energy consumption degradation in the cases of "min" and "round robin" are marginal when compared to the improvements showcased above.



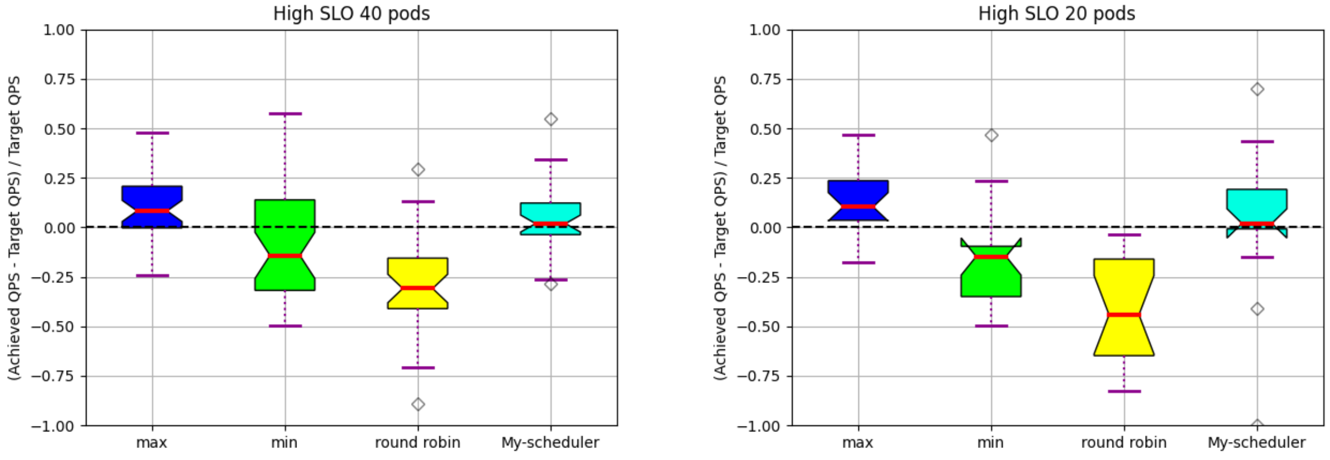
(a) Energy

(b) Power

Figure 9.4: Energy consumption

9.4 QPS Distributions

In figure 9.5 we can see the distribution of the percentage differences between Achieved QPS and SLO for the experiments with high SLOs. As expected both "min" and "round robin" schedulers failed to have a positive mean value while almost all the occurrences lie in the negative half plane. In the case of "max" scheduler, we can see that all the occurrences lie in the positive half plane. However, as explained before, the mean value is significantly greater than zero and occurrences can be found even at 25% higher than demanded due to over-provisioning, resulting in an underutilized cluster with few applications occupying a large quota of the total resources for a potential large time interval. This leads to long queuing times which in turn can comprise fairness or even lead to resource starvation. In the case of our scheduler we can see that the mean value of percentage difference is fractionally higher than 0, while almost all non zero occurrences lie in the positive half, however with significantly lower variation compared to the other schedulers. On an average, only 7 out of 60 (11.6%) applications had an average number of queries per second lower than the SLO, while even those applications that were unable to cover the users' requests had a distance not greater than 5% from the respective requests. For "round robin" scheduler 54 out of 60 applications (except for one) failed to meet their constraints, with an average negative distance of 31.6%. For "min" scheduler 24 out of 60 failed to meet their constraints, with an average negative distance of 26.1%.

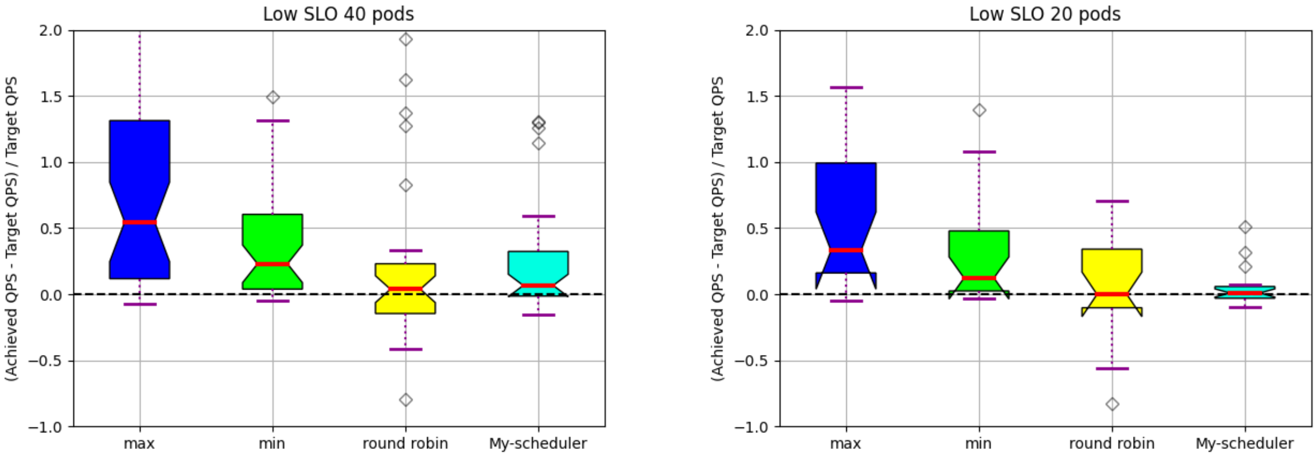


(a) High SLO 40 pods

(b) High SLO 20 pods

Figure 9.5: QPS Distributions for high SLOs

In figure 9.6 we can see the distribution of the percentage differences between Achieved QPS and SLO for the experiments with low SLOs. Here, we can see that all of the schedulers were able to cover most of the user constraints due to them being fairly low. However, there are some things that should be pointed out as well. Comparing our scheduler to an interference oblivious scheduler which supports collocation (round robin) we come to the conclusion that while our scheduler manages to cover users' needs for 96.15% of the applications while also supporting collocation, "round robin" scheduler only manages to cover users' needs for 62.05% of the pods due to unforeseen negative interference produced and suffered by collocated applications. Furthermore, it is clear that distributions related to "round robin" hold a much larger variation compared to the ones related to our scheduler. As far as the "max" scheduler is concerned, we observe that, as explained before, while all occurrences lie in the positive half plane, their values are absurdly large, resulting in erratic delays and stiffness in the cluster. Regarding the "min" scheduler we can see that all of the user needs are covered. Nevertheless, at a glance, we can conclude that in accordance with "max" scheduler, it also features a high variation, thus also resulting in rigidness. In contrast to the rest of the schedulers, in the case of our scheduler the variance as low as 18% compared to the variances of "max" and "min" schedulers which are equal to 340% and 259% respectively.



(a) Low SLO 40 pods

(b) Low SLO 20 pods

Figure 9.6: QPS Distributions for low SLOs

Chapter 10

Conclusion and Future Work

10.1 Summary

In this thesis, we designed a resource and interference aware GPU scheduler based on state-of-the-art Kubernetes container orchestrator. First, we characterized our workloads based on metrics obtained from prometheus time series database(TSDB) regarding their performance and after analysis of their results we were able to extract some useful conclusions related to their interference and durability to heterogeneity. We evaluated the system using workloads that consist of inference engines with different backends, models, number of queries, scenarios etc. We show that our scheduler, for the majority of workloads and scheduling scenarios, can achieve lower pending and execution time on average as well as better resource utilization while it ensures versatility and sticks to the users' demands without incurring high overhead or energy consumption.

10.2 Future Work

The analysis, observations and proposals described in this thesis were an immature attempt to face the GPU sharing problem in Kubernetes infrastructures. In the following subsections, future work is suggested. We categorize those suggestions into two groups, the ones related to development optimizations and the ones related to further research opportunities.

10.2.1 Development Scope

Regarding the development of the system, future work could include designing a custom device plugin which, similarly to our work, using MPS and MIG technologies would offer the ability to the user to reserve GPU resources in a fine-grained manner. Regarding heterogeneity, the code can be extended to take into consideration other modern system resources such as FPGAs, TPUs, ASICs etc. Regarding the prediction system, a wider variety of AI/ML models and their respective parameters could be explored in order to better profile the applications. Finally, in order to train our models the data used were metrics obtained by statically profiling the incoming workloads. Therefore an option is to extend the recommender system in order to be able to get a trace of an unknown application and add it to the dataset while regularly retraining the models to incorporate the data from newly arrived applications.

10.2.2 Research Scope

In this section, we suggest some research subjects as proposed future work. First of all, a Neural Collaborative Filtering model could be used to predict the performance and interference of incoming workloads. Another option is that since multivariate imputation performs better for low sparsity and SVD works better for high sparsity, a mix of the two could be used to optimize the accuracy of the predictions. These extensions will allow better collocation decisions and hence less quality of service (QoS) violations and better resource utilization. Finally, more fine-grained/low-level metrics could be utilized in order to explore the sensitivity of each application to the interference on each component of the GPU architecture.

Bibliography

[1] "NVIDIA Data Center GPU Manager (NVIDIA Official Website)." [Online]. Available: https://developer.nvidia.com/dcgm

[2] "Prometheus Official Website." [Online]. Available: https://prometheus.io/docs/introduction/overview/

[3] "MLPerf Inference (MLPerf Official Website)." [Online]. Available: https://mlperf.org/inference-overview/{#}overview

[4] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, R. Chukka, C. Coleman, S. Davis, P. Deng, G. Diamos, J. Duke, D. Fick, J. S. Gardner, I. Hubara, S. Idgunji, T. B. Jablin, J. Jiao, T. S. John, P. Kanwar, D. Lee, J. Liao, A. Lokhmotov, F. Massa, P. Meng, P. Micikevicius, C. Osborne, G. Pekhimenko, A. T. R. Rajan, D. Sequeira, A. Sirasao, F. Sun, H. Tang, M. Thomson, F. Wei, E. Wu, L. Xu, K. Yamada, B. Yu, G. Yuan, A. Zhong, P. Zhang, and Y. Zhou, "Mlperf inference benchmark," 2019.

[5] L. A. B. Urs Hoelzle, *The Datacenter as a Computer.* Morgan Claypool Publishers, May 25, 2009.

[6] Amazon ec2. [Online]. Available: http://aws.amazon.com/ec2/

[7] Microsoft azure. [Online]. Available: http://www.windowsazure.com/

[8] Rackspace. [Online]. Available: http://www.rackspace.com/

[9] Google compute engine. [Online]. Available: cloud.google.com/compute

[10] Vmware vsphere. [Online]. Available: http://www.vmware.com/products/vsphere/

[11] Xenserver. [Online]. Available: http://www.citrix.com/products/xenserver/overview.html

[12] A. Kansal, C. Kozyrakis, S. Sankar, and K. Vaid, "Server engineering insights for large-scale online services," *IEEE Micro*, July 2010. [Online]. Available: https://www.microsoft.com/en-us/research/publication/server-engineering-insights-for-large-scale-online-services/

[13] J. Mars, L. Tang, and R. Hundt, “Heterogeneity in “homogeneous” warehouse-scale computers: A performance opportunity,” *IEEE Computer Architecture Letters*, vol. 10, no. 2, pp. 29–32, 2011.

[14] R. Nathuji, C. Isci, and E. Gorbatov, “Exploiting platform heterogeneity for power efficient data centers,” in *Fourth International Conference on Autonomic Computing (ICAC’07)*, 2007, pp. 5–5.

[15] L. A. Barroso, “Warehouse-scale computing: entering the teenage decade,” 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=2019527

[16] J. Leverich and C. Kozyrakis, “On the energy (in)efficiency of hadoop clusters,” *Operating Systems Review*, vol. 44, pp. 61–65, 03 2010.

[17] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch, “Power management of online data-intensive services,” in *Proceedings of the 38th ACM International Symposium on Computer Architecture*, 2011.

[18] S. Govindan, J. Liu, A. Kansal, and A. Sivasubramaniam, “Cuanta: Quantifying effects of shared on-chip resource interference for consolidated virtual machines,” Tech. Rep. MSR-TR-2011-55, May 2011. [Online]. Available: https://www.microsoft.com/en-us/research/publication/cuanta-quantifying-effects-of-shared-on-chip-resource-interference-for-consolidated-

[19] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, “Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations,” in *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2011, pp. 248–259.

[20] R. Nathuji, A. Kansal, and A. Ghaffarkhah, “Q-clouds: Managing performance interference effects for qos-aware clouds,” vol. 35, 04 2010, pp. 237–250.

[21] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. Haq, and L. Rigas, “Windows azure storage: a highly available cloud storage service with strong consistency,” 10 2011, pp. 143–157.

[22] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, “Mesos: A platform for Fine-Grained resource sharing in the data center,” in *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*. Boston, MA: USENIX Association, Mar.

2011. [Online]. Available: https://www.usenix.org/conference/nsdi11/mesos-platform-fine-grained-resource-sharing-data-center

[23] N. Vasić, D. Novaković, S. Miučin, D. Kostic, and R. Bianchini, "Dejavu: Accelerating resource allocation in virtualized environments," *ACM SIGPLAN Notices*, vol. 47, 04 2012.

[24] Xen. [Online]. Available: http://www.xen.org/

[25] R. Bell, Y. Koren, Y. Research, and I. Volinsky, "The bellkor 2008 solution to the netflix prize," *ATT Research*, 01 2008.

[26] S. S. V. G. D. T. Baolin Li, Tirthak Patel, "Miso: Exploiting multi-instance gpu capability on multi-tenant systems for machine learning," 2022. [Online]. Available: https://arxiv.org/abs/2207.11428

[27] J. M. Wozniak, R. Jain, P. Balaprakash, J. Ozik, N. T. Collier, J. Bauer, F. Xia, T. Brettin, R. Stevens, J. Mohd-Yusof, C. G. Cardona, B. V. Essen, and M. Baughman, "Candle/supervisor: a workflow framework for machine learning applied to cancer research," *BMC Bioinformatics*, vol. 19, no. 18, p. 491, Dec 2018. [Online]. Available: https://doi.org/10.1186/s12859-018-2508-4

[28] M. Raissi, P. Perdikaris, and G. Karniadakis, "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations," *Journal of Computational Physics*, vol. 378, pp. 686–707, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0021999118307125

[29] J. Shlomi, P. Battaglia, and J.-R. Vlimant, "Graph Neural Networks in Particle Physics," 7 2020.

[30] V. Fung, J. Zhang, E. Juarez, and B. G. Sumpter, "Benchmarking graph neural networks for materials chemistry," *npj Computational Materials*, vol. 7, no. 1, p. 84, Jun 2021. [Online]. Available: https://doi.org/10.1038/s41524-021-00554-0

[31] A100. nvidia a100 tensor core gpu datasheet, 2021.

[32] D. Chen, N. Eisley, P. Heidelberger, S. Kumar, A. Mamidala, F. Petrini, R. Senger, Y. Sugawara, R. Walkup, B. Steinmacher-Burow, A. Choudhury, Y. Sabharwal, S. Singhal, and J. J. Parker, "Looking under the hood of the ibm blue gene/q network," in *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012, pp. 1–12.

[33] D. Narayanan, M. Shoeybi, J. Casper, P. LeGresley, M. Patwary, V. Korthikanti, D. Vainbrand, P. Kashinkunti, J. Bernauer, B. Catanzaro, A. Phanishayee, and M. Zaharia, “Efficient large-scale language model training on gpu clusters using megatron-lm,” in *SC21: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–14.

[34] N. Dryden, R. Böhringer, T. Ben-Nun, and T. Hoefler, “Clairvoyant prefetching for distributed machine learning i/o,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC ’21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3458817.3476181

[35] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS’12. Red Hook, NY, USA: Curran Associates Inc., 2012, p. 1097–1105.

[36] Q. Hu, P. Sun, S. Yan, Y. Wen, and T. Zhang, “Characterization and prediction of deep learning workloads in large-scale gpu datacenters,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC ’21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3458817.3476223

[37] U. Gupta, C.-J. Wu, X. Wang, M. Naumov, B. Reagen, D. Brooks, B. Cottel, K. Hazelwood, M. Hempstead, B. Jia, H.-H. S. Lee, A. Malevich, D. Mudigere, M. Smelyanskiy, L. Xiong, and X. Zhang, “The architectural implications of facebook’s dnn-based personalized recommendation,” in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 488–501.

[38] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang, “Analysis of Large-Scale Multi-Tenant GPU clusters for DNN training workloads,” in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 947–960. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/jeon

[39] 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). Renton, WA: USENIX Association, Apr. 2022, pp. 945–960. [Online]. Available: https://www.usenix.org/conference/nsdi22/presentation/weng

[40] B. Li, R. Arora, S. Samsi, T. Patel, W. Arcand, D. Bestor, C. Byun, R. B. Roy, B. Bergeron, J. Holodnak, M. Houle, M. Hubbell, M. Jones, J. Kepner, A. Klein, P. Michaleas, J. McDonald, L. Milechin, J. Mullen, A. Prout, B. Price, A. Reuther, A. Rosa, M. Weiss, C. Yee, D. Edelman, A. Vanterpool, A. Cheng, V. Gadepally, and D. Tiwari, "Ai-enabling workloads on large-scale gpu-accelerated system: Characterization, opportunities, and implications," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022, pp. 1224–1237.

[41] "Multi-Process Service." [Online]. Available: https://docs.nvidia.com/deploy/mps/index.html

[42] Mig. nvidia multi-instance gpu user guide. [Online]. Available: https://docs.nvidia.com/datacenter/tesla/pdf/NVIDIA_MIG_User_Guide.pdf

[43] Aws. aws to offer nvidia a100 tensor core gpu-based amazon ec2 instances. [Online]. Available: https://www.nvidia.com/en-us/data-center/a100/

[44] Google. a2 vms now ga—the largest gpu cloud instances with nvidia a100 gpus. [Online]. Available: https://cloud.google.com/blog/products/compute/a2-vms-with-nvidia-a100-gpus-are-ga

[45] Top500. top 500 list november 2021. [Online]. Available: https://www.top500.org/lists/top500/2021/11/

[46] Microsoft. microsoft expands its ai-supercomputer lineup with general availability of the latest 80gb nvidia a100 gpus in azure. [Online]. Available: https://azure.microsoft.com/en-us/blog/microsoft-expands-its-aisupercomputer-lineup-with-general-availability-of-the-lates

[47] J. Li, C. Zhang, Y. Cheng, J. Zhang, M. Xie, J. Huang, L. Wu, and J. Yuan, "Analysis of large-scale multi-tenant gpu clusters for dnn training workloads," in *Proceedings of the 2018 ACM International Conference on Measurement and Modeling of Computer Systems*, 2018, pp. 40–52.

[48] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang, F. Yang, and L. Zhou, "Gandiva: Introspective cluster scheduling for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 595–610. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/xiao

[49] X. Zhang, C. Wang, J. Liu, X. Gu, Y. Sun, and B. He, "Kubeshare: A framework to manage gpus as first-class and shared resources in container cloud," in *Proceedings of the 2019 IEEE International Conference on Cloud Computing (CLOUD).* IEEE, 2019, pp. 138–147.

[50] A. Mairh, A. Farooq, and Z. Li, "Container and microservice driven design for cloud infrastructure devops," in *2018 26th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP).* IEEE, 2018, pp. 251–259.

[51] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

[52] A. Verma, L. Ahuja, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proceedings of the Tenth European Conference on Computer Systems.* ACM, 2015, p. 18.

[53] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, and S. Seth, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing.* ACM, 2013, p. 5.

[54] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, 2011, pp. 22–22.

[55] "Kubernetes (Official Website)." [Online]. Available: https://kubernetes.io/

[56] Intel. intel device plugin for kubernetes. [Online]. Available: https://github.com/intel/intel-device-plugins-for-kubernetes

[57] Tencent. rdma device plugin for kubernetes.

[58] M. E. Papka, M. O'Connor, K. Kloster, K. Barker, M. R. Norman, S. Coghlan, J. M. Brase, and R. R. McCune, "Container solutions for hpc systems: A case study of using shifter on blue waters," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW).* IEEE, 2016, pp. 184–193.

[59] G. M. Kurtzer, V. Sochat, and M. W. Bauer, "Singularity: Scientific containers for mobility of compute," in *Proceedings of the practice and experience in advanced research computing on rise of the machines (learning, adaptation, and complexity).* ACM, 2017, pp. 1–8.

[60] A. Bishop, X. Liu, H. S. Oral, B. Raju, S. Sen, and F. Wang, "Enabling diverse software stacks on supercomputers using high performance virtual clusters," in *2016 IEEE International Conference on Cluster Computing (CLUSTER).* IEEE, 2016, pp. 353–362.

[61] Y. Zhang, H. Wang, X. Zhang, F. Liu, and W. Chen, "Gaiagpu: Sharing gpus in container clouds," in *2018 IEEE International Conference on Cloud Computing (CLOUD).* IEEE, 2018, pp. 170–177.

[62] Y. Wang, X. Zhang, and X. Li, "Convgpu: Gpu management middleware in container based virtualized environment," in *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom).* IEEE, 2018, pp. 179–186.

[63] "Alibaba GPU Sharing Scheduler Extender (Alibaba Cloud Blog)." [Online]. Available: https://www.alibabacloud.com/blog/594926

[64] Deepomatic. a shared gpu nvidia k8s device plugin. [Online]. Available: https://github.com/Deepomatic/shared-gpu-nvidia-k8s-device-plugin

[65] [Online]. Available: https://www.backblaze.com/blog/vm-vs-containers/

[66] orchestration. [Online]. Available: https://www.databricks.com/glossary/orchestration

[67] "Kubernetes official documentation." [Online]. Available: https://kubernetes.io/docs/home/

[68] "MLPerf Official Website." [Online]. Available: https://mlperf.org/

[69] "Tensor Cores." [Online]. Available: https://developer.nvidia.com/tensor-cores

[70] "NVIDIA Deep Learning SDK libraries." [Online]. Available: https://developer.nvidia.com/deep-learning-software

[71] "NVIDIA cuDNN." [Online]. Available: https://developer.nvidia.com/cudnn

[72] "NVIDIA TensorRT." [Online]. Available: https://developer.nvidia.com/tensorrt

[73] "Automatic Mixed Precision for Deep Learning." [Online]. Available: https://developer.nvidia.com/automatic-mixed-precision

[74] "Querying Prometheus (Official Site)." [Online]. Available: https://prometheus.io/docs/prometheus/latest/querying/basics/

[75] “MLPerf Inference Load Generator.” [Online]. Available: https://mlperf.github.io/inference/loadgen/index.html

[76] S. van Buuren and K. Groothuis-Oudshoorn, “mice: Multivariate imputation by chained equations in r,” *Journal of Statistical Software*, vol. 45, no. 3, p. 1–67, 2011. [Online]. Available: https://www.jstatsoft.org/index.php/jss/article/view/v045i03