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

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Resource Aware GPU Scheduling in Kubernetes Infrastructure

Άγγελος Α. Φερίκογλου
Α.Μ. : 03114068

Επιβλέπων : Δημήτριος Ι. Σούντρης
Καθηγητής ΕΜΠ

Αθήνα
Ιούλιος 2020

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

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Resource Aware GPU Scheduling in Kubernetes Infrastructure

Άγγελος Α. Φερίκογλου
Α.Μ. : 03114068

Επιβλέπων : Δημήτριος Ι. Σούντρης
Καθηγητής ΕΜΠ

Τριμελής Επιτροπή Εξέτασης

(Υπογραφή)	(Υπογραφή)	(Υπογραφή)
.....	.....	.....
Δημήτριος Σούντρης	Διονύσιος Πνευματικάκος	Γεώργιος Γκούμας
Καθηγητής	Καθηγητής	Επίκουρος Καθηγητής
ΕΜΠ	ΕΜΠ	ΕΜΠ

Ημερομηνία Εξέτασης:
27 Ιουλίου 2020

(Υπογραφή)

.....

Άγγελος Α. Φερίκογλου
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Τα τελευταία χρόνια παρατηρείται η αύξηση της εκτέλεσης εφαρμογών τεχνητής νοημοσύνης και μηχανικής μάθησης σε περιβάλλοντα υπολογιστικού νέφους. Για την αποτελεσματική διαχείριση των υψηλών υπολογιστικών αναγκών, οι διαχειριστές των κέντρων δεδομένων και οι πάροχοι υπηρεσιών νέφους άρχισαν να υιοθετούν την χρήση καρτών γραφικών σε κλίμακα χιλιάδων κόμβων. Δεδομένου ότι οι κάρτες γραφικών άρχισαν να χρησιμοποιούνται σχετικά πρόσφατα σε περιβάλλοντα υπολογιστικού νέφους, εκλείπει η αποδοτική τους διαχείριση. Οι σύγχρονοι δρομολογητές και ενορχηστρωτές αγνοούν τα ιδιαίτερα χαρακτηριστικά τους και τις ιδιότητες των εφαρμογών που τις χρησιμοποιούν. Επιπρόσθετα, παρατηρείται ότι οι χρήστες δεσμεύουν περισσότερους πόρους της κάρτας γραφικών από ότι χρειάζεται γεγονός που οδηγεί σε υποχρησιμοποίηση του πόρου.

Στην παρούσα εργασία σχεδιάζουμε έναν δρομολογητή για κάρτες γραφικών. Ο δρομολογητής έχοντας επίγνωση της τρέχουσας κατάστασης του πόρου είναι σε θέση να τοποθετεί αποδοτικά περισσότερες εφαρμογές στην ίδια κάρτα γραφικών. Ενσωματώνουμε τη λύση μας στον Κυβερνήτη, έναν από τους πιο ευρέως χρησιμοποιούμενους ενορχηστρωτές υπολογιστικών συστημάτων σε περιβάλλοντα νέφους σήμερα. Τέλος, δείχνουμε ότι μπορούμε να πετύχουμε υψηλότερη ταχύτητα εξυπηρέτησης για τις εφαρμογές αλλά και αποδοτικότερη χρήση του ίδιου του πόρου, συγκριτικά με άλλους διαδεδομένους δρομολογητές, για μια πληθώρα διαφορετικών εφαρμογών μηχανικής μάθησης.

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

Abstract

Nowadays, there is an ever-increasing number of Artificial Intelligence (AI) and Machine Learning (ML) workloads pushed and executed on the Cloud. To effectively serve and manage these huge computational demands, data center operators and cloud providers have provisioned GPU resources at the scale of thousands of nodes. Since GPUs are relatively new to the cloud stack, support for efficient GPU management lacks, as state-of-the-art schedulers and orchestrators treat GPUs only as a specific resource constraint while ignoring its unique characteristics and application properties. In addition, users tend to request more GPU resources than they actually need, leading to resource under-utilization.

In this thesis, we design a resource aware GPU scheduling system, able to efficiently colocate applications on the same card arriving at a data center. We integrate our solution with Kubernetes, one of the most widely used cloud orchestration frameworks nowadays. We show that our scheduler can achieve better quality of service (QoS) and higher resource utilization compared to the state-of-the-art schedulers, for a variety of ML cloud representative workloads.

Keywords— cloud computing, GPU, resource management, scheduling, Kubernetes, resource-aware, 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 στον μεταδιδακτορικό ερευνητή Σωτήριο Ξύδη και τον υποψήφιο διδάκτορα Δημοσθένη Μασούρο για τη βοήθεια και τη συνεργασία τους καθ ΄όλη τη διάρκεια της διπλωματικής μου. 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.

Ακόμη θα ήθελα να ευχαριστήσω τους γονείς μου Αλέκο και Βέτα και τον αδερφό μου Σπύρο για την υποστηρίξη τους σε ότι επιχειρούσα κατά τη διάρκεια της ζωής και των σπουδών μου. Τέλος, θα ήθελα να ευχαριστήσω την Φωτεινή, τον Γιάννη, τον Θανάση Μ., τον Θανάση Π., την Θεώνη, τον Ιάσονα και τον Κώστα, τους ανθρώπους που γνώρισα στο πλαίσιο της σχολής και έκαναν αυτά τα 6 χρόνια όμορφα παρόλες τις στιγμές πίεσης και άγχους.

Contents

List of Figures

List of Tables

Εκτεταμένη Ελληνική Περίληψη

1 Εισαγωγή

Την τελευταία δεκαετία, παρατηρήθηκε αύξηση της χρήσης υπολογιστικών συστημάτων νέφους. Η εξέλιξη της τεχνολογίας της εικονικοποίησης βασισμένης σε πακέτα, καθώς και τα πλεονεκτήματα που προσφέρουν οι υπολογιστές νέφους στους χρήστες και στους διαχειριστές, αποτέλεσαν έναυσμα προς αυτή την κατεύθυνση. Οι υποδομές αυτές δίνουν στους χρήστες τη δυνατότητα να εκτελούν διαφορετικά είδη εφαρμογών, πληρώνοντας μόνο τους πόρους που χρησιμοποιούνται σε μια δεδομένη στιγμή. Παράλληλα επιτρέπουν την ανάπτυξη οικονομιών κλίμακας για τους φορείς εκμετάλλευσης, οι οποίοι τους διαμοιράζουν σε διαφορετικούς χρήστες. Η αύξηση του όγκου των εργασιών που φορτώθηκαν και εκτελέστηκαν σε υπολογιστές νέφους, έχουν αναγκάσει τους φορείς εκμετάλλευσης των κέντρων δεδομένων και τους παρόχους υπηρεσιών νέφους, όπως το Google Cloud Platform και το AWS να θέσουν ως σημαντική προτεραιότητα τους το σχεδιασμό ενός συστήματος με γνώμονα την συντοποθέτηση εφαρμογών, καθώς επίσης και τον διαμοιρασμό πόρων μεταξύ διαφορετικών χρηστών. [1]

Ένας πόρος που εμφανίζεται όλο και περισσότερο σε περιβάλλοντα υπολογιστικού νέφους είναι η κάρτα γραφικών. Η αύξηση της εκτέλεσης εφαρμογών μηχανικής μάθησης σε υποδομές νέφους, κατέστησε επιτακτική την ανάγκη να υιοθετηθεί η χρήση καρτών γραφικών σε κλίμακα χιλιάδων κόμβων. Δεδομένου ότι οι κάρτες γραφικών άρχισαν να χρησιμοποιούνται σχετικά πρόσφατα σε περιβάλλοντα υπολογιστικού νέφους, εκλείπει η αποδοτική τους διαχείριση. Οι σύγχρονοι δρομολογητές και ενορχηστρωτές αγνοούν τα ιδιαίτερα χαρακτηριστικά τους και τις ιδιότητες των εφαρμογών που τις χρησιμοποιούν. Επιπρόσθετα, παρατηρείται ότι οι χρήστες δεσμεύουν περισσότερους πόρους της κάρτας γραφικών από αυτούς που πραγματικά χρειάζονται, γεγονός που οδηγεί στην υποχρησιμοποίηση του πόρου.

Στην παρούσα εργασία, σχεδιάζουμε έναν δρομολογητή για κάρτες γραφικών βασισμένο σε real-time μετρικές του πόρου. Εντοπίζουμε την αναποτελεσματικότητα των δύο ευρέως χρησιμοποιούμενων δρομολογητών για κάρτες γραφικών στον Κυβερνήτη σχετικά με την ποιότητα της παρεχόμενης υπηρεσίας (QoS) αλλά και την χρησιμοποίηση του πόρου. Δείχνουμε ότι ο δρομολογητής μας, για την πλειοψηφία των χρησιμοποιούμενων workloads πετυχαίνει χαμηλότερο μέσο χρόνο αναμονής

όσον αφορά τις εφαρμογές αλλά και συνολικό χρόνο εκτέλεσης για τα workloads ενώ εξασφαλίζει υψηλότερη μέση χρήση της μνήμης και των μονάδων παράλληλης επεξεργασίας της κάρτας γραφικών. Τέλος, ο μηχανισμός που προτείνουμε εξασφαλίζει χαμηλότερη μέση κατανάλωση ενέργειας όσον αφορά την κάρτα γραφικών.

2 Κυβερνήτης και Ενορχήστρωση Πακέτων

Πακέτο (Container): Ένα πακέτο (container) [2] είναι μια τυποποιημένη μονάδα λογισμικού η οποία συγκεντρώνει τον κώδικα αλλά και όλες τις εξαρτήσεις του έτσι ώστε η εφαρμογή να μπορεί να εκτελείται γρήγορα και αξιόπιστα σε ποικίλα περιβάλλοντα υπολογιστών. Τα κέντρα δεδομένων σήμερα χρησιμοποιούν αυτή την τεχνολογία εικονικοποίησης καθώς έχει πολλά πλεονεκτήματα συγκριτικά με τις εικονικές μηχανές. Ενδεικτικά αναφέρουμε την ευέλικτη δημιουργία και ανάπτυξη εφαρμογών, τη διευκόλυνση ενός γρηγορότερου κύκλου ανάπτυξης του λογισμικού, τη συνέπεια σχετικά με το περιβάλλον ανάπτυξης και την απομόνωση πόρων.

Ενορχήστρωση: Σε μεγάλες συστοιχίες υπολογιστών υπάρχει η ανάγκη ενορχήστρωσης και διαχείρισης των πακέτων. Η ανάγκη αυτή καλύπτεται από υλοποιήσεις όπως αυτή του Κυβερνήτη [3], ενός έργου ανοιχτού λογισμικού που ξεκίνησε να αναπτύσσεται με πρωτοβουλία της Google. Ο Κυβερνήτης αποτελεί την πιο ευρέως χρησιμοποιούμενη υλοποίηση. Ξεκινώντας από το υψηλότερο επίπεδο αφαίρεσης, η αρχιτεκτονική του περιλαμβάνει έναν ή περισσότερους κόμβους ¨αφέντη' (master) , οι οποίοι είναι το μυαλό του συστήματος, παίρνουν αποφάσεις και αντιδρούν σε διάφορα γεγονότα που λαμβάνουν χώρα σε αυτό. Η άλλη ομάδα κόμβων είναι οι επονομαζόμενοι κόμβοι ¨εργάτες' (workers) , στους οποίους αποστέλλονται και εκτελούνται οι εφαρμογές. Ένας κόμβος master περιέχει: kube-apiserver, etcd, kube-scheduler και kube-controller-manager. Από την άλλη, ένας κόμβος worker περιέχει: kubelet, kube-proxy και Container Runtime.

Οι εφαρμογές αφού τοποθετηθούν μέσα σε πακέτα τοποθετούνται σε Pods τα οποία μπορούν να περιέχουν ένα ή περισσότερα πακέτα ή και μονάδες αποθήκευσης. Ο Κυβερνήτης υποστηρίζει μια πληθώρα υπηρεσιών ενώ παρέχει ποικίλες δυνατότητες στους χρήστες και προγραμματιστές, ευνοώντας την αυτοματοποίηση των αναγκαιών εργασιών. Μια από αυτές είνα ο ενσωματωμένος δρομολογητής εργασιών. Ο τελευταίος βασιζόμενος σε μετρικές υψηλού επιπέδου, αφαιρετικές σε επίπεδο εικονικοποίησης όπως η χρήση των επεξεργαστών και μνήμης, παίρνει αποφάσεις σχετικά με την τοποθέτηση των Pods. Η διαδικάσια με την οποία οι αποφάσεις αυτές λαμβάνονται περιλαμβάνει δύο στάδια. Αρχικά, εξετάζονται όλοι οι υποψήφιοι κόμβοι σχετικά με τη διαθεσιμότητά τους και την ικανότητα τους να εξυπηρετήσουν την εισερχόμενη εφαρμογή. Στη συνέχεια, όσοι από αυτούς κριθούν κατάλληλοι βαθμολογούνται με τη χρήση μιας σειράς από συναρτήσεις

αξιολόγησης.

Ο Κυβερνήτης παρέχει υποστήριξη για τις κάρτες γραφικών των εταιρειών NVIDIA και AMD. Το πρόβλημα με τον μηχανισμό αυτό είναι ότι αγνοεί τα ιδιαίτερα χαρακτηριστικά των καρτών γραφικών αλλά και των εφαρμογών που τις χρειάζονται για την εκτέλεσή τους. Κάθε Pod δεσμεύει πλήρως τον πόρο οδηγώντας σε υποχρησιμοποίηση της κάρτας γραφικών αλλά και σοβαρές παραβιάσεις του QoS.

3 Πειραματικό Περιβάλλον

Το σύστημα που δημιουργήθηκε για την αξιολόγηση του δρομολογητή απαρτίζεται από τρεις εικονικές μηχανές στις οποίες έχει στηθεί ο Κυβερνήτης. Από τους τρεις κόμβους ο ένας λειτουργεί ως κόμβος διαχειριστής ενώ οι άλλοι δύο ως κόμβοι εργάτες. Η κύρια διαφορά των προαναφερθέντων κόμβων είναι ότι ο ένας έχει πρόσβαση σε κάρτα γραφικών.

Για την εξαγωγή των μετρικών της κάρτας δημιουργήθηκε ένα σύστημα που αποτελείται από δύο τμήματα. Αρχικά, ο *NVIDIA GPU Metrics Exporter* [4] λαμβάνει διάφορες μετρικές όπως η χρησιμοποιούμενη μνήμη, το ποσοστό χρήσης των SMs και η κατανάλωση ισχύος και τα εξάγει σε μορφή χρονοσειράς. Με τη σειρά του ο *Prometheus* [5], μια βάση δεδομένων για χρονοσειρές, λαμβάνει αυτές τις χρονοσειρές και επιτρέπει την χρήση τους μέσω της γλώσσας ερωτημάτων PromQL [6]. Οι μετρικές αυτές χρησιμοποιούνται για τις αποφάσεις της τοποθέτησης πολλαπλών εργασιών στην ίδια κάρτα καθώς και για να μας δώσουν μεγαλύτερη εποπτεία όσον αφορά την αξιοποίηση του πόρου.

Το τελευταίο συστατικό του πειραματικού περιβάλλοντος είναι τα workloads που δημιουργήθηκαν για τον έλεγχο του συστήματος. Δεδομένου ότι παρατηρείται η όλο και αυξανόμενη εκτέλεση εφαρμογών μηχανικής μάθησης σε περιβάλλοντα υπολογιστικού νέφους, στραφήκαμε στην σουίτα του MLPerf [7]. Το MLPerf έχει ως στόχο τη δημιουργία δίκαιων και χρήσιμων benchmarks για τη μέτρηση της απόδοσης υλικού, λογισμικού και υπηρεσιών μηχανικής μάθησης. Συγκεκριμένα, χρησιμοποιήσαμε το MLPerf Inference [8], το υποσύνολο της σουίτας που ως στόχο έχει τη μέτρηση του πόσο γρήγορα συστήματα επεξεργάζονται εισόδους και παράγουν αποτελέσματα χρησιμοποιώντας ένα εκπαιδευμένο μοντέλο. Τα workloads που χρησιμοποιήσαμε αφορούν κυρίως image classification και object detection. Κάθε workload αποτελείται από διαφορετικά Inference Engines που χρησιμοποιούν διαφορετικά μοντέλα, σύνολα δεδομένων, backend (ONNX Runtime, Pytorch, Tensorflow κ.λ.π) και εκτελούν ένα διαφορετικό αριθμό από Inference Queries με βάση κάποιο προκαθορισμένο σενάριο.

4 Η υλοποίησή μας

Ο αλγόριθμος που χρησιμοποιήθηκε για την τοποθέτηση πολλαπλών εφαρμογών στην κάρτα γραφικών στηρίχθηκε στο Alibaba GPU sharing scheduler extension [9] καθώς και στην ερευνητική εργασία Kube-Knots [10]. Στη συνέχεια αναφέρουμε τη διαδικασία που ακολουθείται μέχρι τη λήψη της απόφασης της δρομολόγησης καθώς και τα βασικά τμήματα του αλγορίθμου για την τοποθέτηση δύο ή περισσότερων εφαρμογών στην κάρτα γραφικών.

Όποτε ένα Job χρησιμοποιεί τον δρομολογητή μας, το αντίστοιχο Pod εισέρχεται σε μία ουρά προτεραιότητας η οποία καθορίζει τη σειρά με την οποία τα Pods επιλέγονται προς δρομολόγηση. Η προτεραιότητα των εφαρμογών είναι ανάλογη της μνήμης που αιτούνται. Έτσι, ο μηχανισμός μας προσπαθεί πάντα να δρομολογήσει το Pod με τη μεγαλύτερη αίτηση σε μνήμη. Μόλις επιλεγεί ένα Pod από την ουρά ξεκινά η εκτέλεση των τριών τμημάτων του αλγορίθμου τα οποία είναι το Resource Agnostic GPU Sharing Logic, το Correlation Based Prediction και το Peak Prediction. Η εκάστοτε εφαρμογή δρομολογείται αν ικανοποιείται ένα από τα τρία τμήματα. Διαφορετικά, ο δρομολογητής αποθηκεύει το Pod και συνεχίζει με το επόμενο από την ουρά.

Το Resource Agnostic GPU Sharing Logic έχοντας γνώση της μέγιστης ελεύθερης μνήμης της κάρτας γραφικών, ελέγχει αν η αίτηση μνήμης της εφαρμογής μπορεί να ικανοποιηθεί. Σε περίπτωση που αυτό είναι δυνατό, η εφαρμογή δρομολογείται και η ελεύθερη μνήμη μειώνεται κατά την τιμή της αίτησης. Όταν η εφαρμογή ολοκληρωθεί η δεσμευμένη μνήμη απελευθερώνεται. Το πρόβλημα με την προσέγγιση αυτή είναι ότι οι χρήστες συχνά αιτούνται περισσότερη μνήμη από αυτή που πραγματικά χρειάζονται, γεγονός που οδηγεί σε υποχρησιμοποίηση του πόρου. Για να αντιμετωπίσουμε το πρόβλημα αυτό χρησιμοποιήσαμε real-time μετρικές της κάρτας για τη λήψη των αποφάσεων της δρομολόγησης. Αρχικά, χρησιμοποιώντας το Correlation Based Prediction (CBP) πραγματοποιούμε μια πρόβλεψη για την ελεύθερη μνήμη της κάρτας γραφικών με βάση το 80%-ile του σήματος της χρησιμοποιούμενης μνήμης. Στην περίπτωση που η αίτηση μπορεί να ικανοποιηθεί με βάση την πρόβλεψη, η εφαρμογή δρομολογείται. Τέλος, μέσω του Peak Prediction (PP) κάνουμε μια πρόβλεψη, με τη χρήση ενός auto-regressive μοντέλου, για τη μελλοντική χρήση της μνήμης της κάρτας γραφικών. Αν η αίτηση της εφαρμογής μπορεί να ικανοποιηθεί η εφαρμογή δρομολογείται. Η ιδέα για αυτόν τον μηχανισμό είναι ότι μια εφαρμογή δεν δεσμεύει όλη την απαιτούμενη μνήμη αμέσως. Άρα, η εφαρμογή μπορεί να δρομολογηθεί ακόμα και αν δεν μπορεί να ικανοποιηθεί η αίτηση μέσω της πρόβλεψης του CBP. Στο σημείο αυτό πρέπει να γίνει σαφές ότι και οι δύο προτεινόμενοι μηχανισμοί πραγματοποιούν μια πρόβλεψη γεγονός που σημαίνει ότι είναι πιθανή η αποτυχία και επανεκκίνηση των δρομολογούμενων εφαρμογών.

5 Αποτελέσματα και Αξιολόγηση

Για την αξιολόγηση του δρομολογητή μας και τη σύγκρισή του με τους διαδεδομένους δρομολογητές Kubernetes default GPU extension και το Alibaba GPU scheduler extension, εκτελέσαμε διάφορα πειράματα. Κάθε πείραμα αποτελείται από ένα workload και το διάστημα *[MIN, MAX]* που καθορίζει τον χρόνο μεταξύ των αφίξεων δύο διαδοχικών Pods. Σε κάθε πείραμα το ίδιο workload τροφοδοτείται σε όλους τους διαθέσιμους δρομολογητές πολλαπλές φορές. Κάθε φορά χρησιμοποιείται διαφορετικό ποσοστό over-provisioning όσον αφορά τη μνήμη της κάρτας γραφικών. Από την άλλη πλευρά, ένα workload αποτελείται από ένα σύνολο από Jobs κάθε ένα εκ των οποίων διαχειρίζεται ένα Pod. Κάθε Pod δημιουργεί ένα διαφορετικό Inference Engine το οποίο καθορίζεται πλήρως από το χρησιμοποιούμενο backend, το εκπαιδευμένο μοντέλο, το σύνολο δεδομένων, το σενάριο, την αίτηση μνήμης κάρτας γραφικών και τον αριθμό από Inference Queries που θα εκτελεστούν.

Στη συνέχεια παρουσιάζονται τα αποτελέσματα για το ομογενές workload (όλα τα Jobs δημιουργούν το ίδιο Inference Engine και εκτελούν το ίδιο πλήθος από Inference Queries) με βάση το ssd-mobilenet για το διάστημα *[20, 40]*. Τέλος, παρουσιάζεται η αξιολόγηση του μοντέλου για το ίδιο διάστημα.

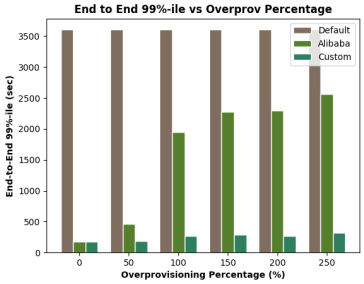
5.1 Ομογενές workload βασισμένο στο ssd-mobilenet με MIN=20 και MAX=40

Η απαιτούμενη μνήμη κάρτας γραφικών για αυτό το είδος Inference Engine είναι περίπου 7 GB. Είναι κατανοητό ότι σε μία κάρτα με 32 GB ελεύθερης μνήμης, μόνο 4 Pods μπορούν να τοποθετηθούν μαζί.

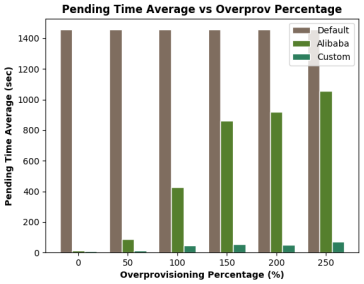
Τα διαγράμματα 1 δείχνουν ότι ο μηχανισμός που προτείνουμε οδηγεί σε καλύτερο end-to-end 99%-ile, pending time average και workload duration συγκριτικά με το Kubernetes default GPU extension και το Alibaba GPU share scheduler extension. Παρόλα αυτά η τοποθέτηση πολλαπλών Pods στην ίδια κάρτα γραφικών οδηγεί σε υψηλότερο μέσο 99%-ile όσον αφορά τα Inference Engines. Επιπρόσθετα, από τα διαγράμματα 2 βλέπουμε ότι ο μηχανισμός μας οδηγεί σε υψηλότερη χρήση της μνήμης και των SMs της κάρτας γραφικών καθώς και σε υψηλότερη μέση κατανάλωση ισχύος. Παρατηρείται επίσης, χαμηλότερη μέση κατανάλωση ενέργειας συγκριτικά με τους διαδεδομένους δρομολογητές.

Για να γίνουν κατανοητά τα παραπάνω αποτελέσματα, στα διαγράμματα 3, 4 και 5 παρουσιάζονται τα σήματα που μετρήθηκαν από τον monitoring μηχανισμό μας και χρησιμοποιήθηκαν για τη δημιουργία των παραπάνω barplots.

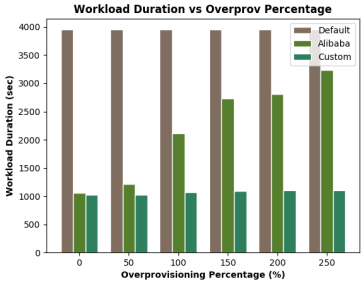
Τέλος, στο παραπάνω πείραμα δεν παρατηρήθηκαν επανεκκινήσεις containers για κανένα ποσοστό over-provisioning.



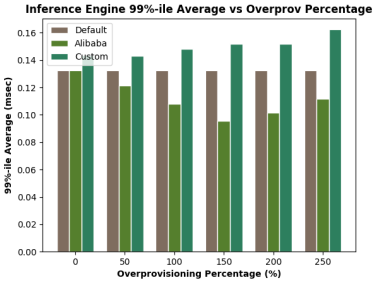
(a) End-to-End 99%-ile



(b) Pending Time Average

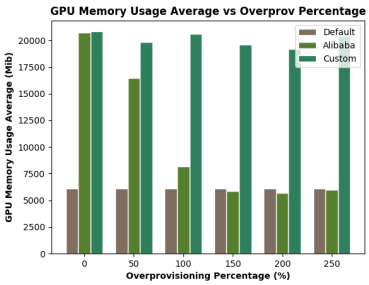


(c) Workload Duration

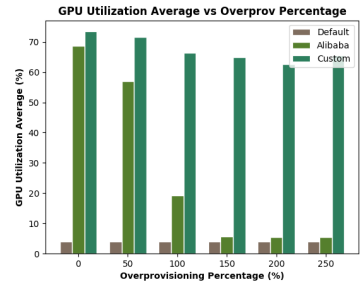


(d) Inference Engine 99%-ile

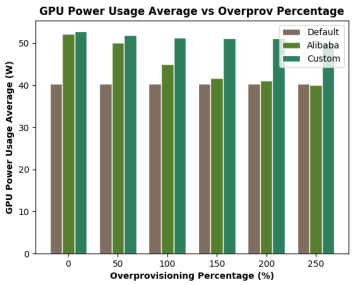
Figure 1: End-to-end 99%-ile, pending time average, workload duration and inference engine 99%-ile συναρτήσει του ποσοστού over-provisioning για το ομογενές ssd-mobilenet workload με MIN=20 και MAX=40



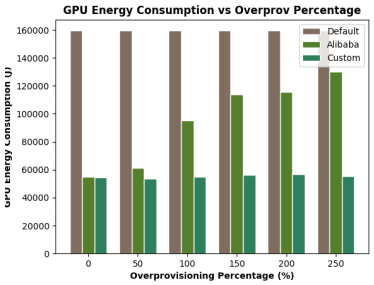
(a) Memory Usage Average



(b) SM Utilization Average

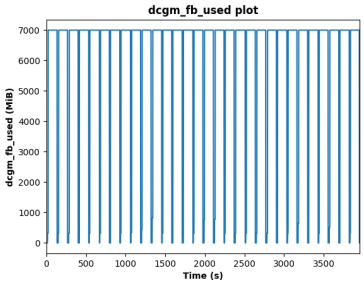


(c) Power Consumption Average

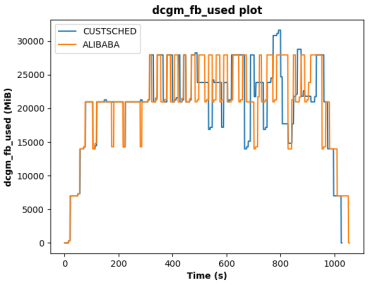


(d) Energy Consumption Average

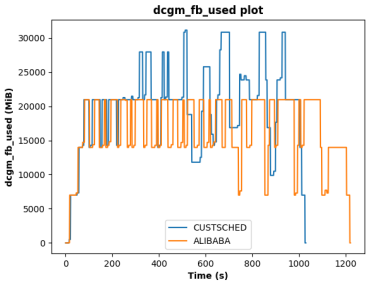
Figure 2: Memory usage average, SM utilization average, power and energy consumption averages συναρτήσει του ποσοστού over-provisioning για το ομογενές ssd-mobilenet workload με MIN=20 και MAX=40



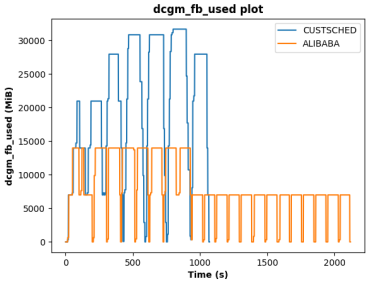
(a) Default



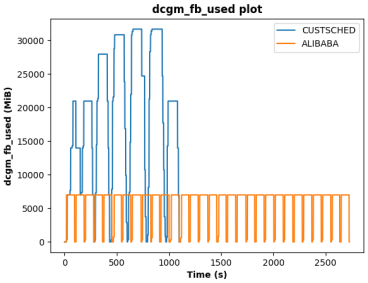
(b) 0% over-provisioning percentage



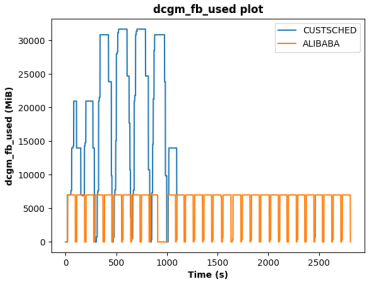
(c) 50% over-provisioning percentage



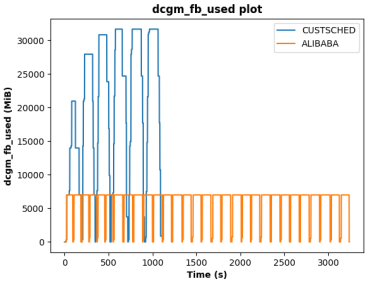
(d) 100% over-provisioning percentage



(e) 150% over-provisioning percentage

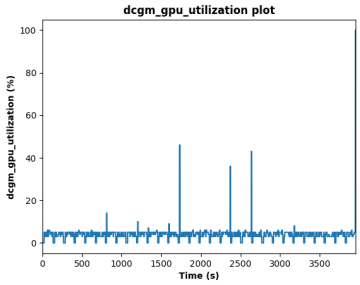


(f) 200% over-provisioning percentage

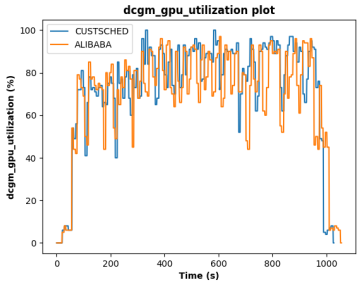


(g) 250% over-provisioning percentage

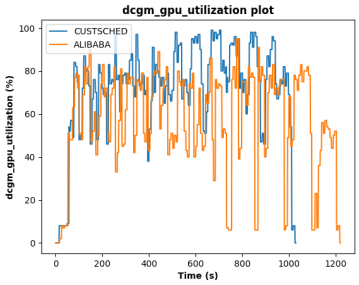
Figure 3: Σήματα κατανάλωσης μνήμης για τον Kubernetes default scheduler, Alibaba GPU sharing scheduler και custom scheduler για ποσοστά over-provisioning ίσα με 0%, 50%, 100%, 150%, 200% και 250% για το ομογενές ssd-mobilenet workload με MIN=20 και MAX=40



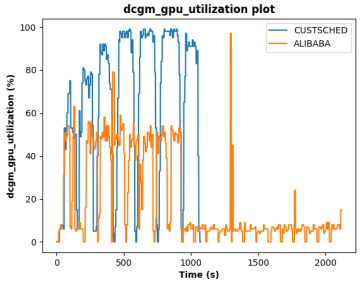
(a) Default



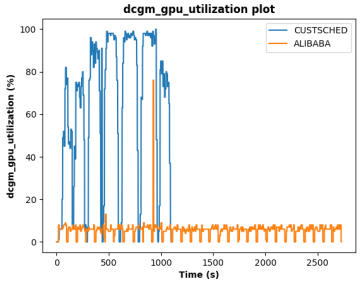
(b) 0% over-provisioning percentage



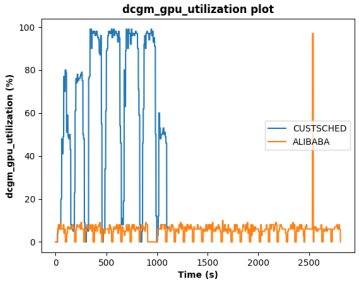
(c) 50% over-provisioning percentage



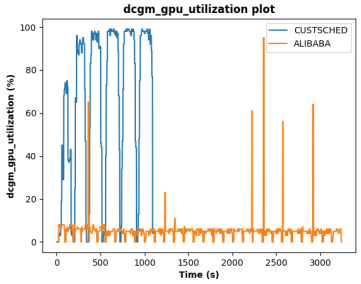
(d) 100% over-provisioning percentage



(e) 150% over-provisioning percentage

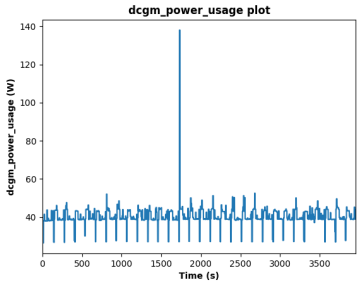


(f) 200% over-provisioning percentage

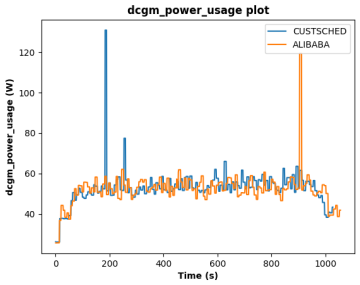


(g) 250% over-provisioning percentage

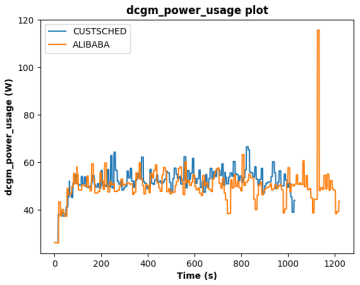
Figure 4: Σήματα κατανάλωσης των SMs για τον Kubernetes default scheduler, Alibaba GPU sharing scheduler και custom scheduler για ποσοστά over-provisioning ίσα με 0%, 50%, 100%, 150%, 200% και 250% για το ομογενές ssd-mobilenet workload με MIN=20 και MAX=40



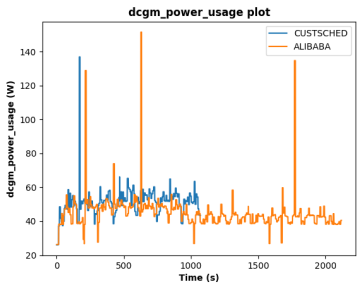
(a) Default



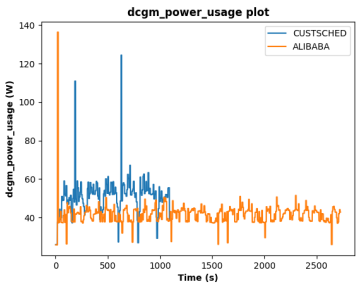
(b) 0% over-provisioning percentage



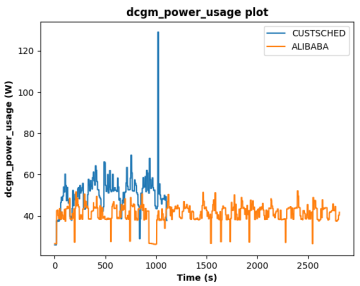
(c) 50% over-provisioning percentage



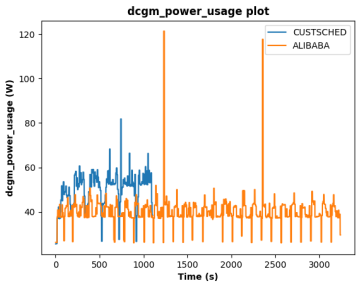
(d) 100% over-provisioning percentage



(e) 150% over-provisioning percentage



(f) 200% over-provisioning percentage



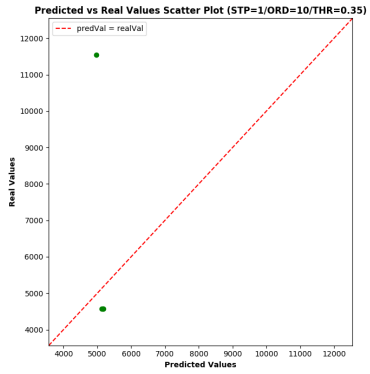
(g) 250% over-provisioning percentage

Figure 5: Σήματα κατανάλωσης ισχύος για τον Kubernetes default scheduler, Alibaba GPU sharing scheduler και custom scheduler για ποσοστά over-provisioning ίσα με 0%, 50%, 100%, 150%, 200% και 250% για το ομογενές ssd-mobilenet workload με MIN=20 και MAX=40

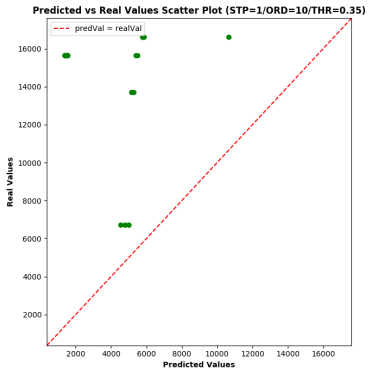
5.2 Αξιολόγηση Μοντέλου

Στην ενότητα αυτή, αξιολογούμε το auto-regressive μοντέλο που χρησιμοποιήθηκε για το Peak Prediction (PP) τμήμα του αλγορίθμου μας. Κατά το PP πραγματοποιούμε Linear Regression (LR) [11] στο σήμα της ελεύθερης μνήμης της κάρτας γραφικών που εξάγεται από τον monitoring μηχανισμό που έχουμε δημιουργήσει.

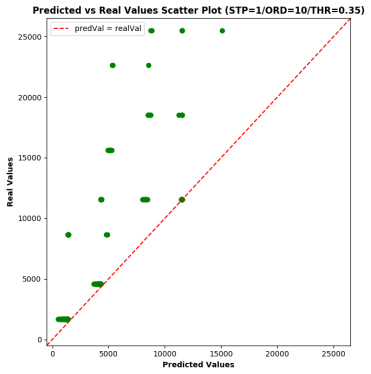
Από τα διαγράμματα 6 και 7 γίνεται αντιληπτό ότι το auto regressive [12] μοντέλο που χρησιμοποιείται για τις προβλέψεις της ελεύθερης μνήμης της κάρτας γραφικών, πραγματοποιεί υποεκτιμήσεις. Αυτή η συμπεριφορά αποτελεί και το κύριο μειονέκτημα του μοντέλου καθώς οι υποεκτιμήσεις οδηγούν σε λανθασμένες αποφάσεις δρομολόγησης και κατά συνέπεια σε παραβιάσεις του QoS και υποχρησιμοποίηση του πόρου.



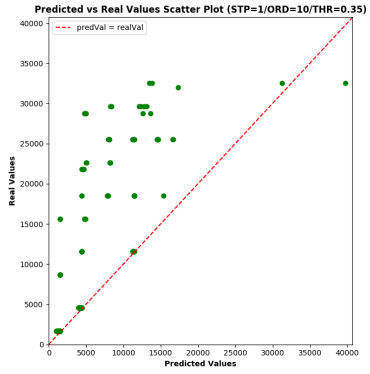
(a) 0% over-provisioning percentage



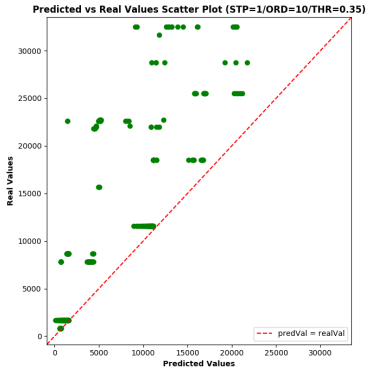
(b) 50% over-provisioning percentage



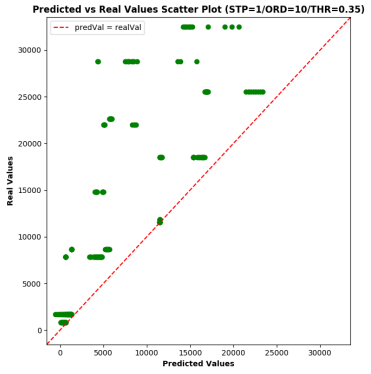
(c) 100% over-provisioning percentage



(d) 150% over-provisioning percentage

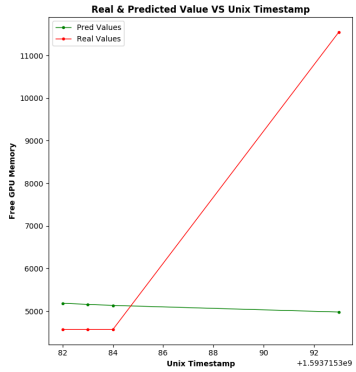


(e) 200% over-provisioning percentage

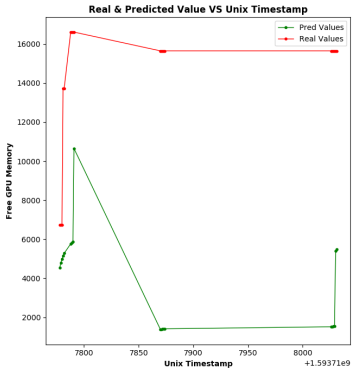


(f) 250% over-provisioning percentage

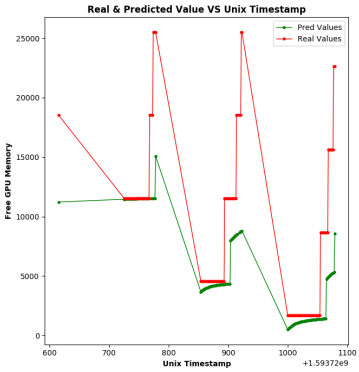
Figure 6: Scatter plot που απεικονίζει την πρόβλεψη της ελεύθερης μνήμη της κάρτας γραφικών συναρτήσει της πραγματικής τιμής για το ομογενές ssd-mobilenet workload με MIN=20 και MAX=40



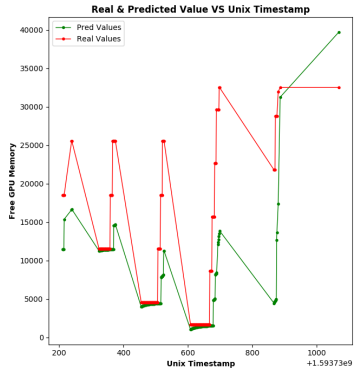
(a) 0% over-provisioning percentage



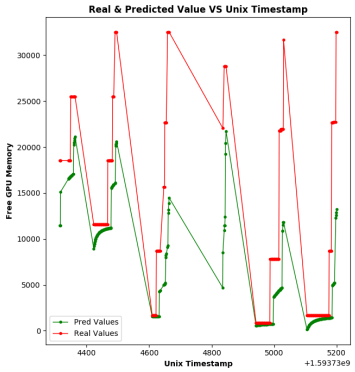
(b) 50% over-provisioning percentage



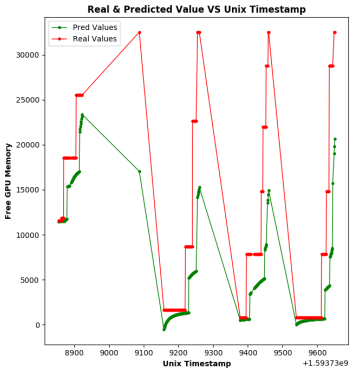
(c) 100% over-provisioning percentage



(d) 150% over-provisioning percentage



(e) 200% over-provisioning percentage



(f) 250% over-provisioning percentage

Figure 7: Line plots που απεικονίζουν την πρόβλεψη της ελεύθερης μνήμη της κάρτας γραφικών και την αντίστοιχη πραγματική τιμή για το ομογενές ssd-mobilenet workload με MIN=20 και MAX=40

6 Σύνοψη και Μελλοντικές Επεκτάσεις

6.1 Σύνοψη

Σε αυτήν τη διπλωματική εργασία, σχεδιάσαμε έναν μηχανισμό ο οποίος λαμβάνοντας υπόψιν real-time μετρικές, επιτρέπει την τοποθέτηση περισσότερων εφαρμογών στην ίδια κάρτα γραφικών. Η υλοποίηση αυτή στηρίχθηκε σε δύο διαδεδομένους δρομολογητές για κάρτες γραφικών στον Κυβερνήτη. Αξιολογήσαμε τον δρομολογητή μας χρησιμοποιώντας workloads, τα οποία απαρτίζονται από διαφορετικά Inference Engines, χρησιμοποιώντας ένα σύνολο από σενάρια. Τέλος, δείξαμε ότι στην πλειοψηφία των σεναρίων ο μηχανισμός που προτείνουμε βελτιώνει την ποιότητα της παρεχόμενης υπηρεσίας ενώ οδηγεί σε καλύτερη εκμετάλλευση της κάρτας γραφικών.

6.2 Μελλοντική Δουλειά

Σαν μελλοντική δουλειά, προτείνουμε από προγραμματιστικής πλευράς την επέκταση του μηχανισμού για πολλαπλούς κόμβους με πολλαπλές κάρτες γραφικών καθώς και την ενσωμάτωσή του στον δρομολογητή του Κυβερνήτη. Τέλος, όσον αφορά την ερευνητική πλευρά, προτείνουμε την χρήση ενός auto-regressive integrated moving average (ARIMA) μοντέλου για την πρόβλεψη της ελεύθερης μνήμης της κάρτας γραφικών καθώς και την χρήση περισσότερων μετρικών όπως το ποσοστό χρήσης των SMs και η κατανάλωση ισχύος για την λήψη των αποφάσεων της δρομολόγησης.

Chapter 1

Introduction

1 Cloud Computing

Cloud computing is a term that has gained widespread use over the last few years. With the exponential increase in data use that has accompanied society's transition into the digital 21st century, it is becoming more and more difficult for individuals and organizations to keep all of their vital information, programs and systems up and running on in-house computer servers.

It operates on a similar principle as web-based email clients, allowing users to access all of the features and files of the system without having to keep the bulk of that system on their own computers. In fact, most people already use a variety of cloud computing services without even realizing it. Gmail, Google Drive, TurboTax, and even Facebook and Instagram are all cloud-based applications. For all of these services, users are sending their personal data to a cloud-hosted server that stores the information for later access. And as useful as these applications are for personal use, they're even more valuable for businesses that need to be able to access large amounts of data over a secure, online network connection. For example, employees can access customer information via cloud-based CRM software like Salesforce from their smartphone or tablet at home or while traveling, and can quickly share that information with other authorized parties anywhere in the world [13].

The increasing use of cloud services has transformed a large part of the IT industry and has radically changed the way users and organizations use the Internet. The adoption of cloud computing has seen explosive growth, both at consumer and enterprise levels and will continue to rise in the future.

The evolution and endorsement of container-based virtualization technology, as well as the advantages that the Cloud computing offers both to users and operators, have acted as enablers towards this direction. Nowadays, it is easier than ever for a user or company to deploy any application in the Cloud. Also, this revolutionary technology made available to users the "pay as you go" feature while economies of scale are enabled for data-center operators who are sharing

their across several users. Finally, the upfront commitment by cloud users is eliminated, allowing companies to start small and increase hardware resources only when there is an increase in their needs.

Cloud provides service-oriented architecture which advocates "everything as a service". Cloud-computing providers offer their "services" according to different models, of which the three standard models are Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). IaaS providers, such as AWS, supply a virtual server instance and storage, as well as APIs that let users migrate workloads to a virtual machine (VM). Users have an allocated storage capacity and can start, stop, access and configure the VM and storage as desired. IaaS providers offer small, medium, large, extra-large and memory- or compute-optimized instances, in addition to customized instances, for various workload needs. In the PaaS model, cloud providers host development tools on their infrastructures. Users access these tools over the internet using APIs, web portals or gateway software. PaaS is used for general software development, and many PaaS providers host the software after it's developed. Common PaaS providers include Salesforce's Lightning Platform, AWS Elastic Beanstalk and Google App Engine. SaaS is a distribution model that delivers software applications over the internet; these applications are often called web services. Users can access SaaS applications and services from any location using a computer or mobile device that has internet access. In the SaaS model, users gain access to application software and databases. One common example of a SaaS application is Microsoft Office 365 for productivity and email services [14, 1].

2 Accelerators in Cloud Infrastructures

In recent years, a growing trend of integrating accelerators in cloud infrastructures is observed. Modern data centers are being provisioned with compute accelerators such as GPUs and FPGAs to catch up with the workload performance demands and reduce the Total Cost of Ownership (TCO). By 2021, traffic within hyperscale datacenters is expected to quadruple with 94% of workloads moving into the cloud according to Cisco's global cloud index. Based on the nature of the application, it can either benefit from the high throughput of GPUs or the low latency present in FPGAs, TPUs, and custom ASICS. This trend is also evident as public cloud service providers like Amazon and Microsoft have started offering GPU and FPGA-based infrastructure services [10].

In particular, emerging intelligent personal assistant (IPA) workloads including speech recognition, image classification, face recognition and natural language processing have recently gained tremendous momentum. Several major

Internet-service companies including Google, Microsoft, Apple and Baidu have all released their IPA services providing a wide range of features. Compared to traditional warehouse scale computer (WSC) applications such as web-search, IPA applications are significantly more computationally demanding. Accelerators, such as GPUs, FPGAs and ASICs, have been shown to be particularly suitable for these IPA applications from both performance and TCO perspectives [15].

With the increase in such workloads, public clouds have provisioned GPU resources at the scale of thousands of nodes in datacenters. Since GPUs are relatively new to the cloud stack, support for efficient GPU management lacks, as state-of-the-art cluster resource orchestrators, like Kubernetes, treat GPUs only as a specific resource constraint while ignoring its unique characteristics and application properties [10]. In addition, it is observed that users tend to request more GPU resources than they actually need. In particular, the majority of applications bind all the resource or request more GPU memory. It is understood that the over-provisioning problem leads to resource under-utilization.

To overcome the above issues, we need to create a scheduler that will be able to understand the resource needs of applications in runtime and colocate them to the same card.

3 Overview

In this thesis, we design a novel GPU scheduler based on real-time GPU metrics monitoring. 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 used workloads, can achieve lower pending time average and overall workload duration while it ensures higher GPU memory usage and SM utilization percentage average and lower energy consumption.

The rest of this thesis is organized as follows. In chapter 2, we analyze other GPU scheduling approaches that have been proposed so far. In chapter 3, we discuss about the basic concepts of Kubernetes, the container orchestrator we used. In chapter 4, we present our experimental infrastructure, the GPU monitoring system and the benchmark for the workload creation. In chapter 5, we present our custom scheduling system which uses real-time metrics in order to decide whether a set of Pods can be co-located in a GPU. In chapter 6, we evaluate our proposed framework and compare it with the state of the art Kubernetes GPU schedulers, across different scenarios and workloads. Finally, in chapter 7, we propose future work in order to improve our scheduling system.

Chapter 2

Related Work

1 Alibaba GPU Sharing Scheduler Extender

Kubernetes provides the capability to schedule NVIDIA GPU containers, but it is generally implemented by binding GPU cards to individual containers. This allows better isolation and ensures that applications using a GPU are not affected by other applications. It is suitable for Deep Learning model training scenarios, but it would be a waste for model prediction scenarios. In order to allow more prediction services to share the same NVIDIA GPU and thus improve the card utilization, the partitioning of the GPU resources is required. The dimension of GPU resource partitioning refers to the partitioning of GPU memory and CUDA Kernel threads.

For fine-grained GPU card scheduling, the Kubernetes community does not currently have a good solution. This is because the Kubernetes definition of extended resources, such as GPUs, only supports the addition and subtraction of integer granularity, but cannot support the allocation of complex resources. For example, it is not possible for a Pod to occupy half of the GPU card in the current Kubernetes architecture design. To overcome the aforementioned issues, the authors in [9] proposed a GPU Share Scheduling solution which relies on the existing working mechanism of Kubernetes. This scheduling mechanism aims to improve the utilization of GPU resources in a cluster and let multiple users share the model development environment. It also allows an application user to run multiple logic tasks on a GPU at the same time.

The basic features of this implementation are:

- The Extended Resource definition of Kubernetes is still used, but the minimum unit to measure the dimension is changed from 1 GPU card to the MiB of GPU memory. If the GPU used by the node is a single-card with 16 GiB memory, then its corresponding resource is 16276 MiB.
- User requirements for shared GPU are typically related to model development and model prediction scenarios. Therefore, in this case, the upper

limit of GPU resources applied by the user cannot exceed 1 card, that is, the upper limit of resources applied is a single card.

In the following pages the architecture of the Alibaba GPU Sharing Scheduler Extender is analyzed.

Firstly, two new extended resources are defined: the first is gpu-mem, corresponding to the GPU memory, and the second is gpu-count, corresponding to the number of GPU cards. Vector resources are described by these two scalar resources, and the vector resources are combined to provide a mechanism to support GPU Share.

The basic architecture diagram is as follows:

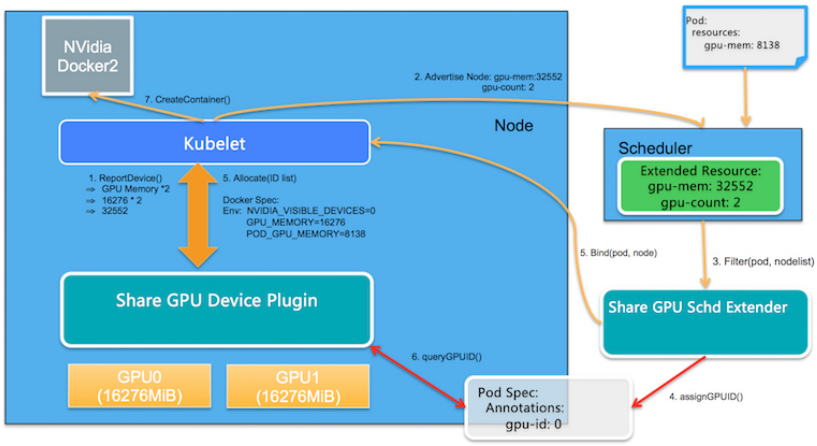


Figure 2.1: Alibaba GPU Share Scheduler Extension Architecture [1]

There are two core function modules that are necessary for this system:

- GPU Share Scheduler Extender: It uses the Kubernetes scheduler extension mechanism to determine whether a single GPU card on the node can provide enough GPU Memory when the global scheduler filters and binds, and record the GPU allocation results to the Pod Spec through annotation at the time of binding for the subsequent filtering to check the allocation results.

- GPU Share Device Plugin: It uses the Device Plugin mechanism, which is called by Kubelet on the node, to allocate the GPU cards and execute based on the allocation result of the Scheduler Extender.

In order to schedule a Pod that requires GPU resources to a specific GPU, the Alibaba GPU Sharing Scheduler Extender follows the steps described below.

[1] https://www.alibabacloud.com/blog/gpu-sharing-scheduler-extender-now-supports-fine-grained-kubernetes-clusters_594926

1. *Resource Reporting*
   GPU Share Device Plugin uses the NVML library to query the number of GPU cards and the memory of each GPU card, and uses ListAndWatch() to report the total memory (quantity memory) of GPUs on the node as an additional Extended Resource to Kubelet. Then, Kubelet reports it to Kubernetes API Server. For example, if a node contains two GPU cards and each card contains 16276 MiB, then from the user's perspective, the GPU resources of the node are 2 * 16276 MiB = 32552 MiB, and the number of GPU cards on the node, which is 2, is also reported as an additional Extended Resource.

2. *Extended Scheduling*
   GPU Share Scheduler Extender can reserve the allocation information in the Pod Spec in the form of annotations while allocating gpu-mem to the Pod, and can determine whether each GPU card contains enough available gpu-mem allocation at the time of filtering based on this information.

   (a) The default Kubernetes scheduler calls the Filter method of the GPU Share Scheduler Extender over HTTP after all the filter actions have been performed. This is because the default scheduler can only determine whether free resources are available that can meet the demand on the whole, and cannot specifically determine whether the demand is met on a single card when computing the Extended Resources. Therefore, it is up to the GPU Share Scheduler Extender to check whether a single card contains available resources.

   The following figure is used as an example. In a Kubernetes cluster composed of 3 nodes that contain 2 GPU cards, when a user applies for gpu-mem = 8138 MiB, the default scheduler scans all nodes and finds that the remaining resources of N1 is 16276 MiB * 2 - 16276 MiB - 12207 MiB = 4069 MiB, which does not meet the resource demands, so N1 node are filtered out. The remaining resources of N2 and N3 nodes are both 8138 MiB, which meets the conditions of the default scheduler from the perspective of overall scheduling. At this time, the default scheduler entrusts the GPU Share Scheduler Extender to perform secondary filtering. In the secondary filtering, the GPU Share Scheduler Extender needs to determine whether the single card meets the scheduling requirements. For N2 node, it is found that the node has 8138 MiB of available resources, but from the perspective of each GPU card, GPU0 and GPU1 have only 4069 MiB of available resources, which cannot meet the demand of 8138 MiB of a single card. N3 Node also has a total of 8138 MiB available

resources, but these available resources all belong to GPU0, meeting the demand of single-card scheduling. As a result, precise conditional filtering can be implemented through the filtering of the GPU Share Scheduler Extender

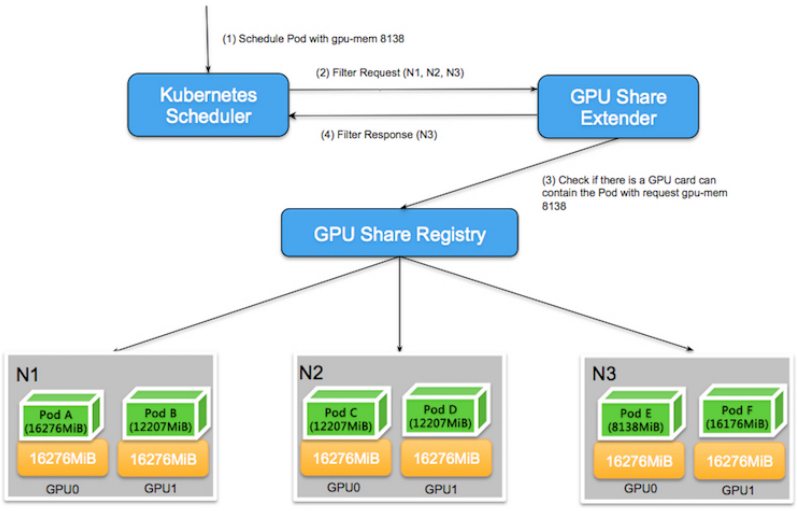


Figure 2.2: Feasible Node Selection

(b) When the scheduler finds a node that meets the condition, it entrusts the bind method of the GPU Share Scheduler Extender to bind the node and the Pod. Here, the Extender needs to perform two operations:

To find the best GPU card ID in the node according to the binpack policy. The "best" here means that for different GPU cards in the same node, and taking the binpack policy as the determinant condition, the GPU card with the least remaining resources and the free resources satisfying the condition is preferentially selected, and saved as ALIYUN_COM_GPU_MEM_IDX in the annotation of the Pod. In addition, the GPU memory applied by the Pod is also saved as ALIYUN_COM_GPU_MEM_POD and ALIYUN_COM_GPU_MEM_ASSUME_TIME to the annotation of the Pod, and the Pod is bound to the selected node at this time.

The Pod annotation for ALIYUN_COM_GPU_MEM_ASSIGNED is also saved and initialized to "false". It means that the Pod is assigned to a GPU card during scheduling, but the Pod is not actually created on the node. ALIYUN_COM_GPU_MEM_ASSUME_TIME represents the time assigned.

If no GPU resources on the allocated node meet the condition, the scheduler does not perform binding at this time and exits directly without reporting an error. The default scheduler will reschedule after ASSUME times out.

As shown in the following figure, when GPU Share Scheduler Extender binds the Pod with gpu-mem 8138 MiB to the selected node N1, it first compares the available resources of different GPUs, which are GPU0 (12207 MiB), GPU1 (8138 MiB), GPU2 (4069 MiB) and GPU3 (16276 MiB). GPU2 are discarded because its remaining resources do not meet the requirements. Among the other 3 GPUs that meet the condition, GPU1 is the GPU card with the least resources left, and the free resources satisfy the condition, so GPU1 is selected.

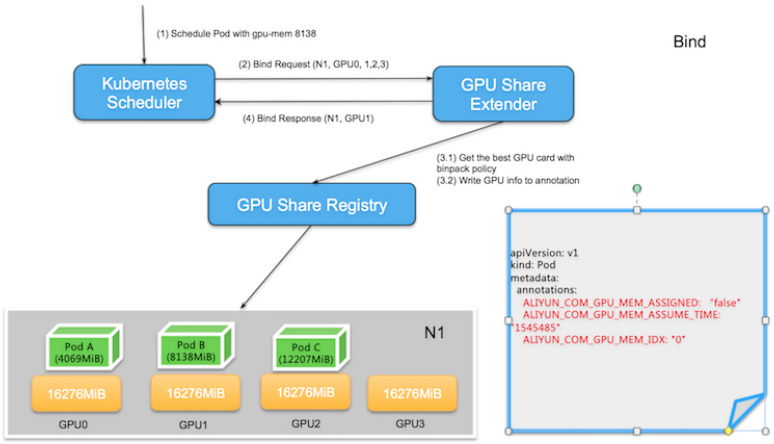


Figure 2.3: GPU Selection

3. *Run on the Node*

When the event that the Pod is bound to the node is received by Kubelet, Kubelet creates a real Pod entity on the node. In this process, Kubelet calls the Allocate method of the GPU Share Device Plugin, and the parameter of the Allocate method is gpu-mem applied by the Pod. In the Allocate method, the corresponding Pod is run according to the scheduling decision of the GPU Share Scheduler Extender.

(a) All the GPU Share Pods in this node with Pending status and ALIYUN_COM_GPU_MEM_ASSIGNED set to false are listed.

(b) The Pod with the same number of ALIYUN_COM_GPU_MEM_POD (in the Pod Annotation) and Allocate applications is selected. If multiple Pods meet the condition, the POD with the earliest ALIYUN_COM_GPU_MEM_ASSUME_TIME is selected.

(c) The ALIYUN_COM_GPU_MEM_ASSIGNED in the Pod Annotation is set to true, and the GPU information in the Pod Annotation is converted into an environment variable and returned to Kubelet to truly create the Pod.

The previously described process is shown in the following diagram.

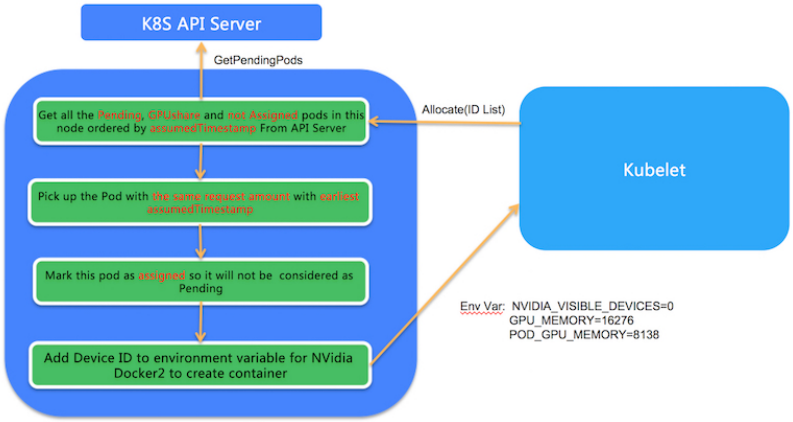


Figure 2.4: Container Creation Steps

2 Kube-Knots

Kube-Knots [10] is a GPU-aware resource orchestration layer that tries to address the GPU orchestration problem. It can dynamically harvest spare compute cycles through dynamic container orchestration enabling co-location of latency-critical and batch workloads together while improving the overall resource utilization.

Kube-Knots is based on two techniques to schedule datacenter-scale workloads: *Correlation Based Prediction* (CBP) and *Peak Prediction* (PP). CBP uses real-time GPU memory data in order to estimate the real memory usage while PP uses real-time GPU memory data to predict the peak memory usage in the future. Kube-Knots improves both average and 99th percentile cluster-wide GPU utilization in case of HPC workloads. In addition, it improves the average Job completion times (JCT) of deep learning workloads when compared to state-of-the-art GPU agnostic schedulers. This leads to cluster-wide energy savings while the end-to-end quality of service (QoS) violations for latency-critical queries are reduced.

3 GPU Scheduling Approaches

The GPU resource management in data center scale is an emerging research problem. For that reason various scheduling frameworks have been proposed. These frameworks are classified in the following four categories.

- *GPU-aware runtime systems*: Quite recently, researchers have proposed GPU runtime changes to enable better scheduling of the GPU tasks either by predicting task behavior or reordering queued tasks. Ukidave et al. [16]

optimized for workloads which under-utilize the device memory and bandwidth. Chen et al. proposed Baymax [15], a runtime mechanism which does workload batching and kernel reordering to improve the GPU utilization. Other techniques such as interference driven resource management proposed by Phull et al. [17], predict the interference on a GPU to do safe co-location of GPU tasks. They do not consider memory bandwidth contention nor over-commitment challenges as they assume sequential execution of the GPU tasks while performing static profiling of applications to ensure safe co-locations. Most of these approaches aim to increase utilization of an individual GPU node and do not scale at the cluster level as they depend on offline training models for node-level run time prediction.

- *Node-level GPU-aware scheduling*: Recent works [18, 19] have proposed docker-level container sharing solutions. In this approach, multiple containers can be made to fit in the same GPU as long as the active working set size of all the containers are within the GPU physical memory capacity. These scheduling techniques over-commit resources for individual containers to ensure crash free execution. This may lead to severe internal memory fragmentation.

- *Distributed GPU scheduling for DNN applications*: Distributed DNN training based applications have started taking advantage of multiple GPUs in a cluster. There are emerging schedulers such as Gandiva [20] and Optimus [21] that focus on prioritizing the GPU tasks that are critical for the DNN model accuracy in case of parameter server-based architecture. These schedulers are designed to cut down on the DNN model exploration time and do not scale well to other application domains as it needs domain-specific knowledge and the application progress metrics. Domain-agnostic scheduling and consolidation is important in case of a public datacenter with multiple-tenants sharing the GPUs bacause they might run a mix of batch and latency jobs of various application domains and not just the DLT (Deep Learning Training) jobs.

- *Hardware support for virtualizing GPUs*: There has been extensive work on providing hardware support for GPU virtualization and preemption. Gupta et al. [22] implemented a task queue in the hypervisor to allow virtualization and preemption of GPU tasks. Tanasic et al. [23] proposed a technique that improves the performance of high priority processes by enabling preemptive scheduling on GPUs. Aguilera et al. [24] proposed a technique to guarantee QoS of high priority tasks by spatially allocating them on more SMs in a GPU. All these techniques require vendors to add extra hardware extensions.

Chapter 3

The Kubernetes Orchestrator

In this chapter we explain why Kubernetes, a container orchestration system, is useful and analyze some of its basic concepts.

1 Docker containers and Orchestration

Virtualization technology increases efficiency in data centers by enabling servers to run multiple operating systems and applications with different requirements and dependencies. Server consolidation has been the focus of virtualization, requiring hardware abstraction to create an environment that can run multiple operating systems. Applications run on virtual machines abstracted away from the hardware. As shown in figure 3.1, Virtual Machines on the left are created on the top of a hypervisor. In Virtual Machines, a complete Operating System is installed. As a result, every VM acts like a guest host. On the other hand containers, presented on the right include a container engine, which creates and manages containers. Note that virtualization via containers is also known as containerization. As shown containerization technology runs multiple containers on a common underlying kernel which are abstracted away into logical partitions. Linux containers with the docker packaging format allow a user to bundle application code with its runtime dependencies, and deploy in a container. A frequently asked question is if someone should use Virtual Machines or containers for his infrastructure setup.

In the following subsections, those two technologies are described in more detail.

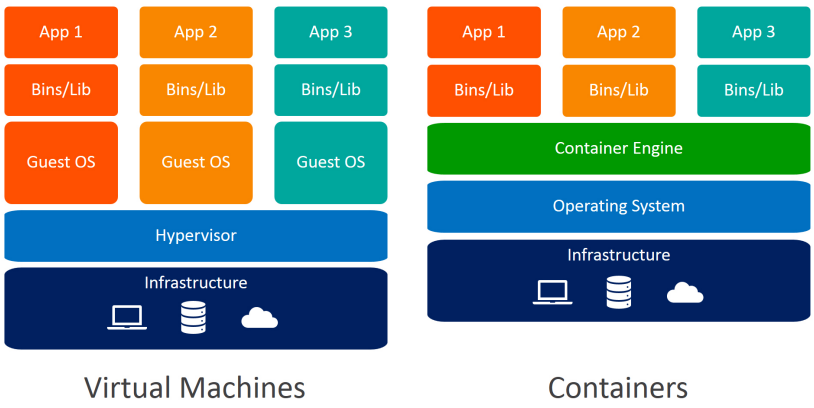


Figure 3.1: Virtual Machines and Containers [1]

1.1 Virtual Machines

Virtual Machines (VMs) provide a virtualized hardware environment where a guest OS is able to run one or more applications. They enable users to create multiple OS instances over the same machine using a hypervisor. User has the flexibility to allocate CPU, Memory and Disk resources into different VMs. This technology unbounds applications from the machine installed OS. Virtualization has matured to include many resilient capabilities such as live migration, high availability, SDN, and storage integration which, to date, are not as mature with containerization. Virtualization also provides a higher level of security by running the workload inside a guest operating system that is completely isolated from the host operating system.

1.2 Containers

A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. A Docker [2] container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings.

Container images become containers at runtime and in the case of Docker containers - images become containers when they run on Docker Engine. Available for both Linux and Windows-based applications, containerized software will always run the same, regardless of the infrastructure. Containers isolate software from its environment and ensure that it works uniformly despite differences for instance between development and staging.

[1] https://www.bmc.com/blogs/containers-vs-virtual-machines/

Container technology is:

- *Standard:* Docker created the industry standard for containers, so they could be portable anywhere.

- *Lightweight:* Containers share the machine's OS system kernel and therefore do not require an OS per application, driving higher server efficiency and reducing server and licensing costs. The overhead of booting managing and maintaining a guest OS environment is avoided. Their lightweight nature leads towards to greater start-up speed.

- *Agile application creation and deployment:* Increased ease and efficiency of container image creation and deployment with quick and easy rollbacks (due to image immutability). In fact, it is the application packaging and deployment capability that is revolutionizing DevOps by providing the capability for developers and operations to work side by side enabling continuous development, integration and deployment. At the same time environment consistency across development, testing and production is provided, as it runs the same on a laptop as it does in the cloud.

- *Resource isolation:* Containers can be deployed with a fixed amount of resources available. Such techniques control and prevent greedy resources usage.

1.3 Orchestration

The answer to the previous question about which virtualization technology is better to use is that they should be used both. They are in fact complementary technologies. Containers support VM-like separation of concerns but with far less overhead and far greater flexibility. As a result, containers have reshaped the way people think about developing, deploying, and maintaining software. In such a hybrid containerized architecture, the different services that constitute an application are packaged into separate containers and deployed across a cluster of virtual machines as illustrated in figure 3.2.

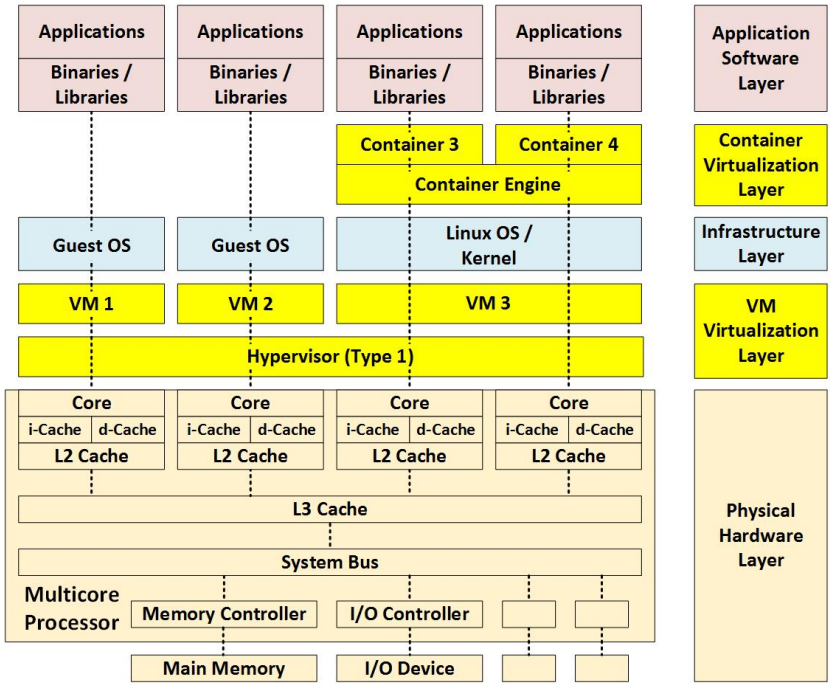


Figure 3.2: Hybrid Containerized Architecture [2]

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.

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.

[2] https://insights.sei.cmu.edu/sei_blog/2017/09/virtualization-via-containers.html

In the following sections we describe the different components of that container orchestrator.

2 Kubernetes Master Node(s) Components

Master components provide the cluster's control plane. Master components make global decisions about the cluster (for example, scheduling), and they detect and respond to cluster events (for example, starting up a new pod when a replication controller's replicas field is unsatisfied). The basic Kubernetes master components are listed below.

- *kube-apiserver:* Component on the master that exposes the Kubernetes API. It is the front-end for the Kubernetes control plane.
- *etcd:* Consistent and highly-available key value store used as Kubernetes' backing store for all cluster data.
- *kube-scheduler:* Component on the master that watches newly created pods that have no node assigned, and selects a node for them to run on.
- *kube-controller-manager:* Component on the master that runs controllers. 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.

3 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.

3.1 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 Kubernetes Architecture

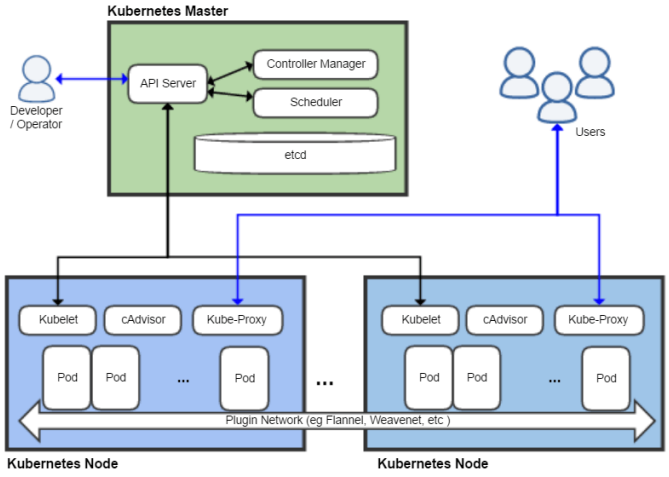


Figure 3.3: Kubernetes Architecture [3]

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 3.3 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.

4.1 Cluster

The highest-level Kubernetes abstraction, the cluster illustrated in figure 3.4, 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.

[3] https://en.wikipedia.org/wiki/Kubernetes
[4] https://kubernetes.io/fr/docs/tutorials/kubernetes-basics/create-cluster/cluster-intro/

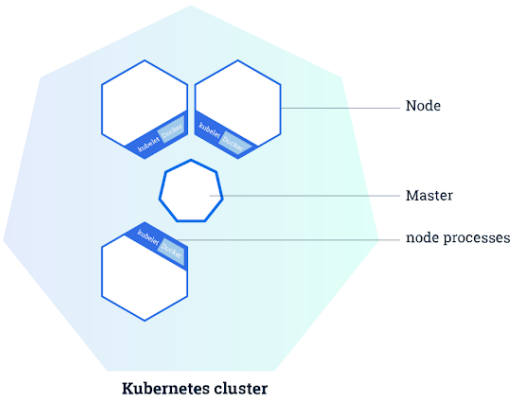


Figure 3.4: Cluster-Node abstraction level [4]

4.2 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 3.5.

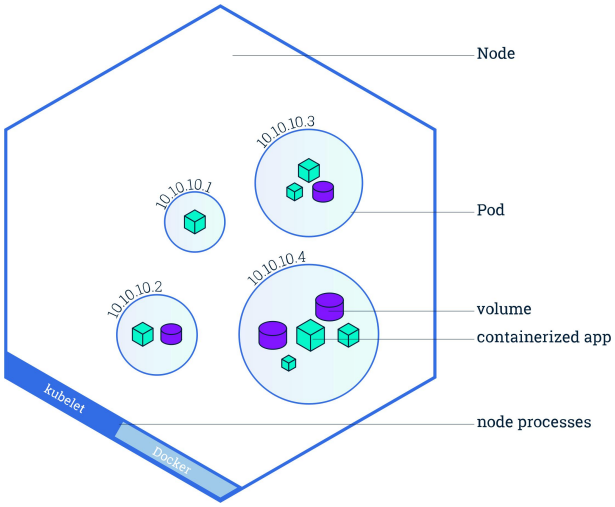


Figure 3.5: Node-Pod-Container abstraction levels [5]

4.3 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 3.5. 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

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

containers themselves. Details about how Kubernetes needs to be configured, from the state of up pods, is kept in etcd (distributed key-value store).

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.

4.4 Job

A Job creates one or more Pods and ensures that a specified number of them successfully terminate. As pods successfully complete, the Job tracks the successful completions. When a specified number of successful completions is reached, the task (i.e. Job) is complete. Deleting a Job will clean up the Pods it created.

A simple case is to create one Job object in order to reliably run one Pod to completion. The Job object will start a new Pod if the first Pod fails or is deleted (for example due to a node hardware failure or a node reboot). A Job can be used to run multiple Pods in parallel.

4.5 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 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.

4.7 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.

5 Kubernetes Resources

5.1 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 [6].

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

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

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.

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.

5.2 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 [7].

6 Kubernetes Scheduler

The Kubernetes Scheduler is a core component of Kubernetes: After a user or a controller creates a Pod, the Kubernetes Scheduler, monitoring the Object Store for unassigned Pods, will assign the Pod to a Node. Then, the kubelet, monitoring the Object Store for assigned Pods, will execute the Pod. [8]

For each unscheduled Pod, Kubernetes scheduler tries to find a node across the cluster according to a set of rules. There are two steps before a destination node of a Pod is chosen. The first step is *Node Filtering*. In this phase, the scheduler determines the set of feasible placements, which is the set of nodes that meet a set of given constraints. All filter functions, also called predicates, must yield true for the Node to host the Pod. The second step is *Node Prioritizing*. In this phase, with only the feasible Nodes remaining, Kubernetes scheduler using a set of predefined rating functions, determines the viability of each Node. The Pod will be scheduled in the one with the highest viability.

Kubernetes scheduler uses this technique for two reasons. Firstly, it needs to make sure that no pod will be scheduled in a Node that is unable to handle it taking into account its Quality of Service (QoS), which can be declared with a few configurations in the yaml file that creates the pod. Secondly, that way it will run the second part of the algorithm (prioritization functions) across a much less set of nodes, which will consume less system resources and less time.

In the following image we can see the steps of the scheduling algorithm.

[7] https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/
[8] https://medium.com/@dominik.tornow/the-kubernetes-scheduler-cd429abac02f

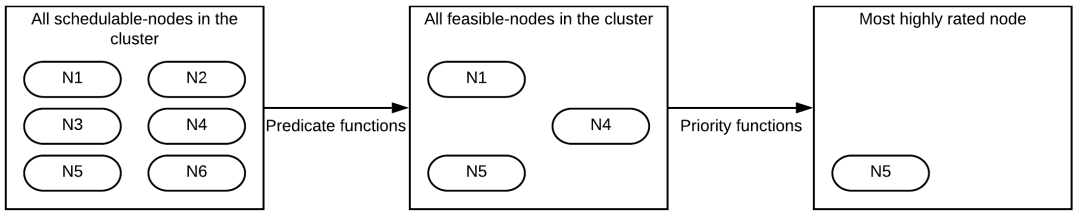


Figure 3.6: Node filtering and ranking

6.1 GPU Extension for Kubernetes Scheduler

Kubernetes includes experimental support for managing AMD and NVIDIA GPUs (graphical processing units) across several nodes [25]. After installing the GPU drivers from the corresponding hardware vendor on the nodes, the corresponding device plugin can be used. When the above conditions are true, Kubernetes will expose amd.com/gpu or nvidia.com/gpu as a schedulable resource.

The GPUs can be used from containers by requesting <vendor>.com/gpu just like the cpu and memory resources. However, there are some limitations in how you specify the resource requirements when using GPUs:

- 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.
- Containers (and Pods) do not share GPUs. There's no overcommitting of GPUs.
- Each container can request one or more GPUs. It is not possible to request a fraction of a GPU.

It is clear that with the default Kubernetes GPU extension only one Pod can use the GPU resource at a time.

Chapter 4

Experimental Infrastructure

In this chapter, we describe the cluster we have created for our experiments, the GPU monitoring system and the MLPerf benchmark suite [7] which was used for the workload creation.

1 System setup

For the nodes of our cluster, we have deployed 3 virtual machines (VMs) (1 master node and 2 worker nodes) on top of the physical machines. The cores of these VMs range from 8 up to 16 and the RAM size from 8 GB up to 24 GB. We used `KVM` as our hypervisor. One of the worker nodes has access to a NVIDIA V100 GPU. To simulate a cloud environment, all the referenced workloads running on the cluster have been containerized, utilizing the Docker platform.

Each VM's characteristics are described in the table.

Virtual Machines			
VM Name	**Cores**	**RAM (GB)**	**GPU Access**
kube-master	8	8	No
kube-cpu	8	16	No
kube-gpu	16	24	Yes

Table 4.1: 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 4.1.

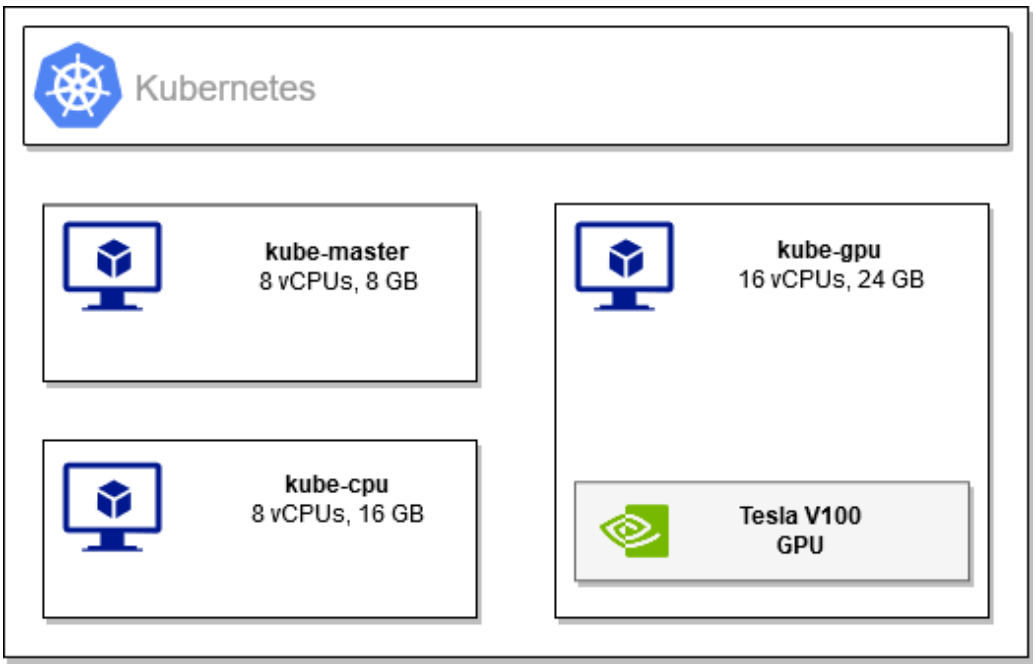


Figure 4.1: Experimental Infrastructure Overview

2 GPU Monitoring System

As a first step, we need to get insight of real-time GPU metrics. The monitoring system we have used consists of the NVIDIA GPU Metrics Exporter [4] and the Timeseries Database Prometheus [5]. These metrics will help us have a better understanding concerning the resource utilization.

2.1 NVIDIA GPU Metrics Exporter

The NVIDIA GPU metrics exporter exports various metrics from the GPU resources of a specified cluster node. These metrics are exported in timeseries format in order to be used in a timeseries database like Influx, Prometheus e.t.c. From the Kubernetes perspective, the NVIDIA GPU metrics exporter is a DaemonSet that creates a set of Pods in the nodes with the GPU resources. These Pods execute metrics queries to the GPUs of the node using NVIDIA Data Center GPU Manager (DCGM) [4]. Finally, an endpoint is exposed from which all the exported metrics can be found.

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

- `dcgm_fb_used`: Used framebuffer memory in Mib
- `dcgm_gpu_utilization`: Simultaneous multiprocessors utilization percentage
- `dcgm_power_usage`: Per second power usage in Watts

2.2 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, e.t.c. 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:

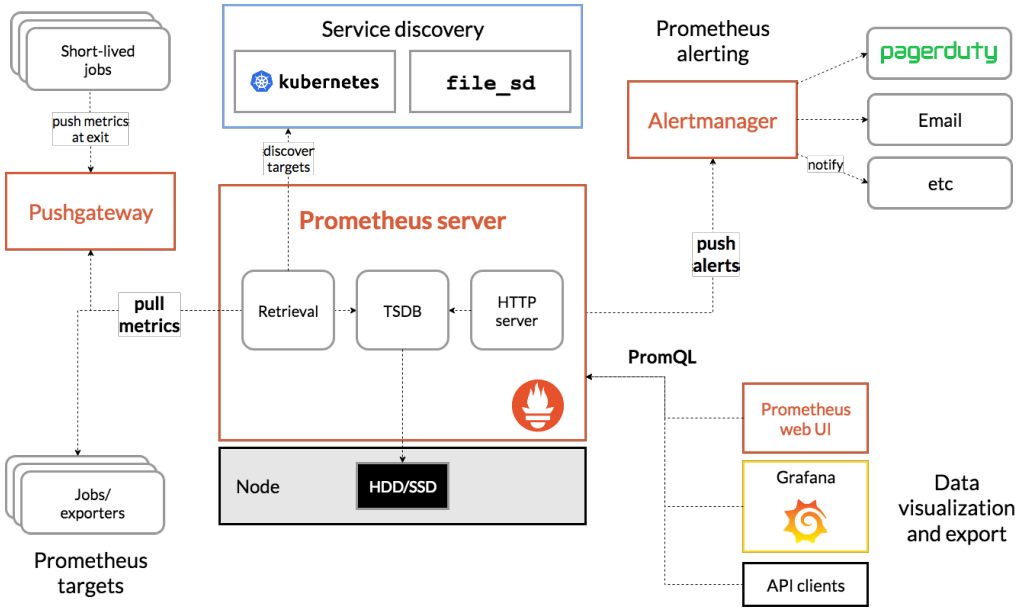


Figure 4.2: Prometheus Architecture

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 [6].

3 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 diploma we focused on latency-critical applications that require GPU resources. For that reason we used MLPerf Inference [8] for creating our workload.

3.1 MLPerf Benchmarks

The mission of MLPerf is to build fair and useful benchmarks for measuring training and inference performance of machine learning (ML) hardware, software, and services. 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 via fair and useful measurement.
- 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 diploma, 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.

3.2 MLPerf Inference

MLPerf Inference [26, 8] 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**	**Model**	**Dataset**	**Quality**	**Server latency constraint**	**Multi-Stream latency constraint**
Vision	Image classification	Resnet50-v1.5	ImageNet (224x224)	99% of FP32 (76.46%)	15 ms	50 ms
Vision	Image classification	MobileNets-v1 224	ImageNet (224x224)	98% of FP32 (71.68%)	10 ms	50 ms
Vision	Object detection	SSD-MobileNets-v1	COCO (300x300)	99% of FP32 (0.22 mAP)	10 ms	50 ms

Table 4.2: MLPerf Inference Benchmarks

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

The key component of the MLPerf Inference Benchmark is the Load Generator [27]. 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.

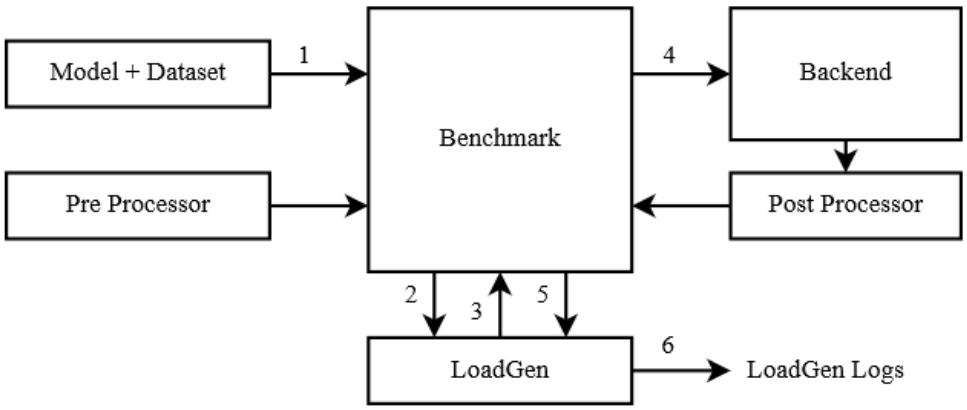


Figure 4.3: Load Generator Integration in MLPerf Inference Benchmarks

As shown in figure 4.3 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 workload consists of a set of Pods, controlled by Jobs, that start the previously described MLPerf Inference Benchmarks. We created a container image that given the appropriate parameters starts the different benchmarks. The scenario we used was Single Stream. We also modified the default query number in Single Stream scenario in order to create Pods that execute different number of queries. The number of queries ranges from 64 up to 32768. In figure 4.4 the inference engine architecture is presented.

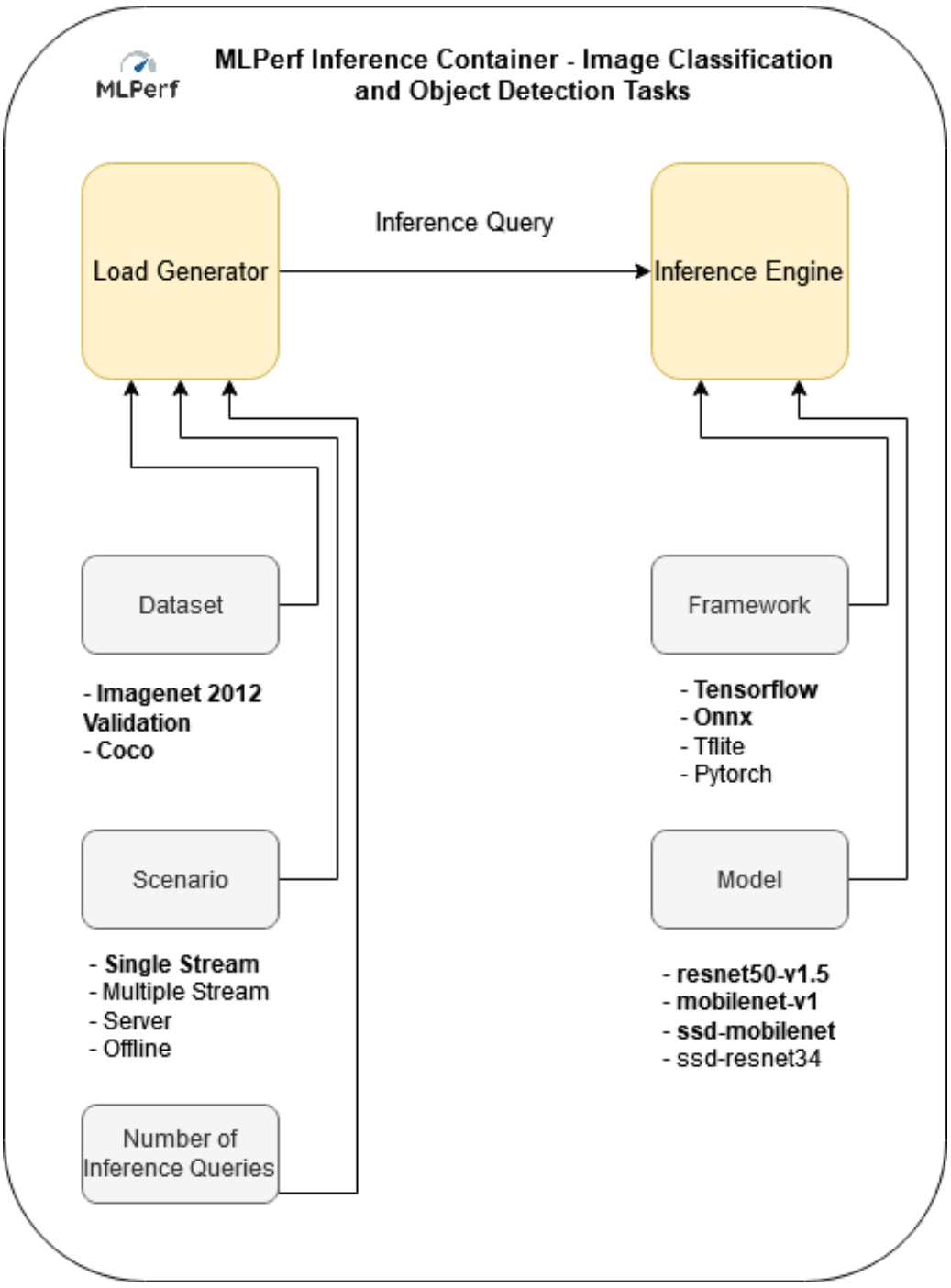


Figure 4.4: Inference Engine Architecture

In figure 4.5, we present all the components of our proposed system.

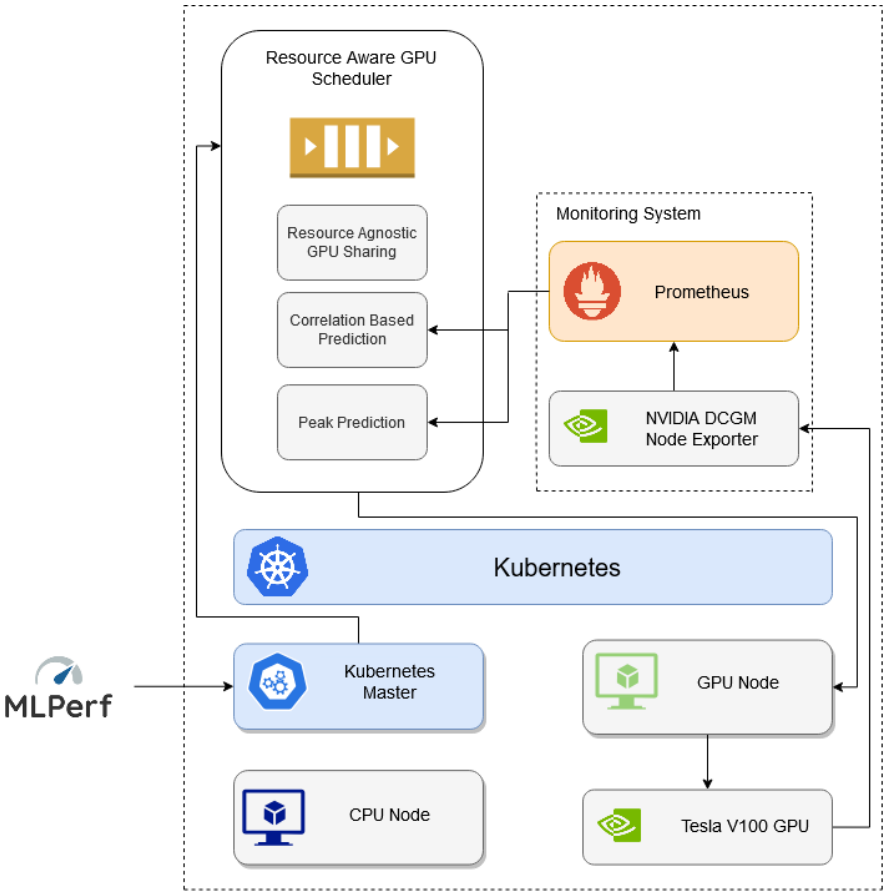


Figure 4.5: Overall System

Chapter 5

Resource Aware GPU Scheduling

In this chapter, we analyze the algorithm that is used for the Pod colocation. The scheduling logic concerning the *node selection*, the *binding* and the *scheduled event emission* is based on Marton Sereg's random scheduler [28]. The proposed GPU collocation mechanism is based on the *Alibaba GPU share scheduler extension* and the *Kube-Knots* paper.

1 Algorithm

When a Job is scheduled from our custom scheduler, the corresponding Pod enters a *priority queue* which defines the Pod scheduling order. The Pod priority is proportional to the corresponding GPU memory request. In that way our scheduling mechanism always tries to schedule the Pods with the bigger memory requests.

If a Pod is chosen to be scheduled, the three following collocation mechanisms are successively executed:

1. *Resource Agnostic GPU Sharing Logic*
2. *Correlation Based Prediction*
3. *Peak Prediction*

The Pod is scheduled only if one of them is satisfied. Otherwise, the scheduler continues with the next Pod in the *priority queue*.

In the following subsections we analyze the previously mentioned colocation mechanisms.

1.1 Resource Agnostic GPU Sharing Logic

The custom scheduler holds a variable ($AVAILABLE\_GPU\_MEMORY$) that is used as an indicator of the available GPU memory. This variable is initialized to the maximum available memory of the used card in *kube-gpu* node. If the

Pod memory request is smaller than the value of this variable, the request can be satisfied and the Pod can be scheduled. Whenever a Pod is scheduled, its value is decreased by the amount of the memory request. Note that this is the *Alibaba GPU Share Scheduler Extension* logic for a cluster with one GPU node that is equipped with one card, as described in chapter 2.

Resource Agnostic GPU Sharing Logic does not face the memory over-provisioning problem. It is not possible to know a priory that the amount of requested memory is actually the amount that the Pod needs to run properly. For example, when a Pod requests more GPU memory than it actually needs and the request can be satisfied, it is possible that a new Pod may not be able to be colocated although it actually is. As we previously mentioned this leads to resource under-utilization.

In our proposed scheduler, we tried to overcome this problem by using real-time memory usage data by our GPU monitoring system. The monitoring system data are collected by performing a query to Prometheus time series database. The range of the query is defined by the scheduling timestamp of the last scheduled Pod and the timestamp of the query execution.

It is important to understand that with the proposed mechanisms there is a risk of Pod failure. That happens because both of these algorithms try to estimate the free GPU memory. The goal is to find the optimal parameters that minimize the container restarts, maximize the resource utilization and minimize the QoS violations of the scheduled Pods. These mechanisms are described below.

1.2 Correlation Based Prediction

Correlation Based Prediction (CBP) provides an estimation for the real memory consumption on a GPU node. The estimation is defined from the 80%-ile of the GPU memory usage rather than the maximum usage.

The basic idea of this algorithm is that GPU jobs, on an average, have stable resource usage for most of their execution, except for the times when the resource demand surges. In addition, the whole allocated capacity is used for a small portion of the execution time while the jobs are provisioned for the peak utilization. CBP scheduler virtually resizes the running Pods for a common case, letting more pending Pods to be colocated.

In order to have an accurate estimation, low signal variability is required. The signal variability is calculated using the coefficient of variation (CV) metric [29]. CV is defined as the ratio of the standard deviation (σ) to the mean (μ). This metric shows the extent of variability in relation to the mean of the data set values. The closer the CV value is to zero, the lower the variability of the signal is. The following examples aim to help us understand the meaning of

CV.

- A data set of *[100, 100, 100]*, with σ equal to 0 and μ equal to 100, has CV equal to 0.
- A data set of *[1, 5, 6, 8, 10, 40, 65, 88]*, with σ equal to 32.9 and μ equal to 27.9, has CV equal to 1.18.

It is obvious that the second data set has higher variability from the first one.

If CV is lower than a defined value, the memory usage is defined by calculating the 80%-ile of the signal. The free GPU memory estimation is equal to the difference of the maximum available GPU memory (MAX_AVAIL_GPU_MEMORY) and the memory usage estimation.

Finally, if the Pod memory request can be satisfied the Pod is scheduled. If the request cannot be satisfied the Peak Prediction algorithm takes over.

1.3 Peak Prediction

Peak Prediction (PP) relies on the temporal nature of peak resource consumption within an application. For example, a Pod that requires GPU resources will not allocate all the memory it needs at once. So although the GPU memory request cannot be satisfied at the scheduling time, it may be satisfied in the near future. As we explained, the Pod can be scheduled without interfering with other running Pods because it will not allocate all the needed GPU memory the moment it is sent.

The memory usage prediction is based on an auto regressive model (AR) [12]. For an accurate prediction the auto correlation value of order k is calculated. If the auto correlation value is larger than a defined value, auto regression of order 1 is performed. In particular, two vectors from the input signal are created (suppose that the input signal has N elements), the independent variable vector and the dependent variable vector. The independent variable vector holds the values of the input signal from index 0 to $N-k-1$ while the dependent variable vector holds the values from index k to $N-1$. Then Linear Regression [11] is performed in order to calculate the slope and intercept of the best fitting line. Finally, the GPU memory usage prediction is found by using the last signal value as an input to our model.

If the Pod GPU memory request can be satisfied from PP the Pod is scheduled. Otherwise, the Pod is saved and our algorithm tries to schedule the next Pod from the priority queue.

The basic parts of our GPU collocation algorithm are presented below.

```
//ResourceAgnosticGPUSharingLogic
if pod.GPUMemRequest ≤ AVAILABLE_GPU_MEMORY then
    schedulePod(pod);
end
//CorrelationBasedPrediction
usedGPUMemTimeseries ← getGPUMetricTS(usedGPUMem);
freeGPUMemTimeseries ← getGPUMetricTS(freeGPUMem);
CV ← calcCV(usedGPUMemTimeseries);
if CV ≤ CV_THR then
    usedGPUMemEstim ← calc80Perc(usedGPUMemTimeseries);
    freeGPUMemEstim ←
     MAX_AVAIL_GPU_MEMORY − usedGPUMemEstim;
    if pod.GPUMemRequest ≤ freeGPUMemEstim then
        schedulePod(pod);
    end
end
//PeakPrediction
autocorValue ← calcAutocorrelation(k, freeGPUMemTimeseries);
if autocorValue ≤ AUTOCORRELATION_THR then
    freeGPUMemPrediction ← AR1(freeGPUMemTimeseries);
    if pod.GPUMemRequest ≤ freeGPUMemPrediction then
        schedulePod(pod);
    end
end
```

Algorithm 1: GPU Collocation Algorithm

In figure 5.1, we describe the custom scheduler algorithm in a more visual way.

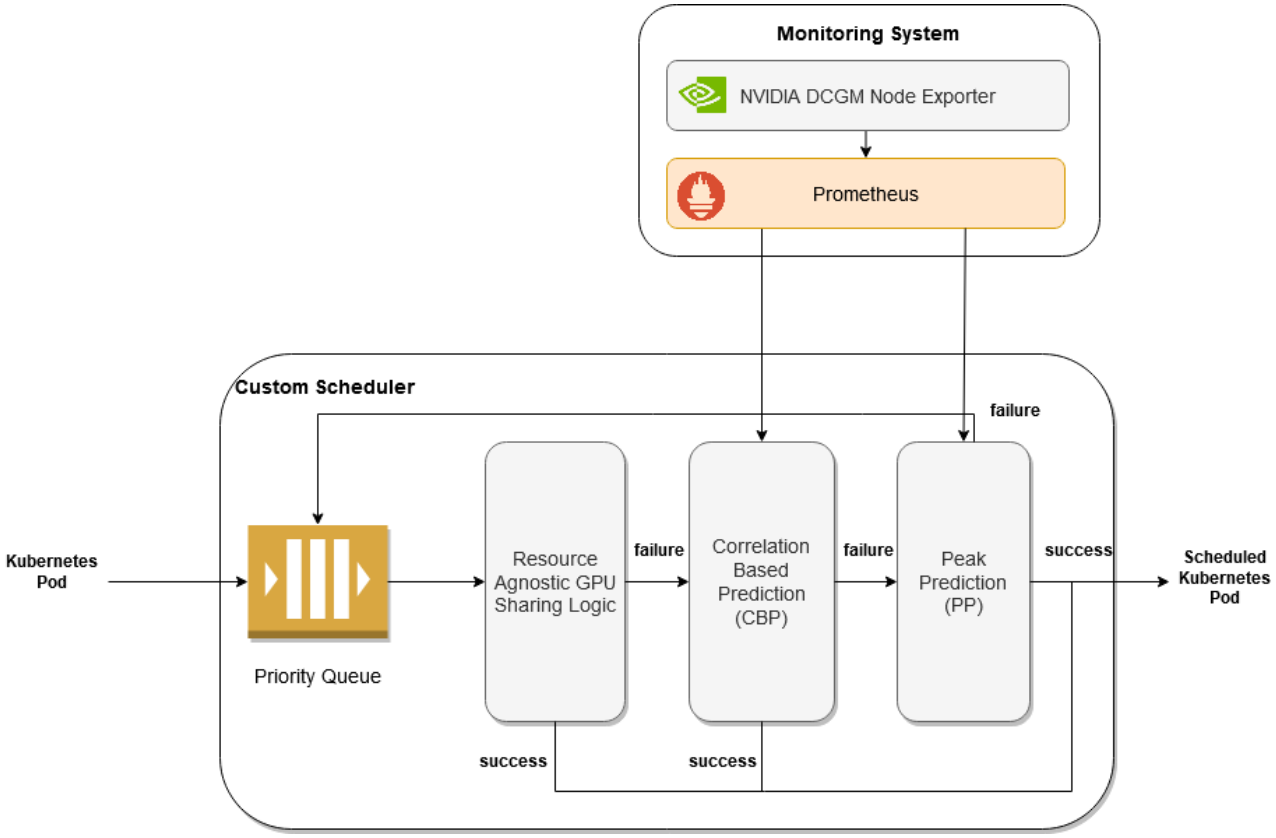


Figure 5.1: Custom Scheduler Algorithm

Chapter 6

Evaluation

In this chapter, we use our experimental infrastructure to evaluate our custom GPU scheduling mechanism using a set of different experiments. In addition, we compare it with Kubernetes Default GPU Extension and Alibaba GPU Share Scheduler Extension.

1 Experiments Description

In order to evaluate our custom scheduler we executed various experiments. Each of them gives us insight about different metrics concerning the QoS and the GPU resource utilization.

An experiment consists of a workload and a range that defines the arrival time between two consecutive Pods. In each experiment the exact same workload with the exact same Pod arrival times are fed to our available schedulers multiple times. Each time, different GPU memory over provisioning percentage is used. The over provisioning percentages range from 0 % up to 250 %.

The arrival time between two consecutive Pods is defined by two different values *MIN* and *MAX*. The arrival time between two consecutive Pods is a randomly chosen number in seconds in *[MIN, MAX]* interval. A workload consists of a number of Kubernetes Jobs where each of them handles one Pod. Each Pod creates a different inference engine by using the MLPerf Inference container we described in section 3. An inference engine is fully defined from the used backend (e.g. Tensorflow, PyTorch, ONNX Runtime e.t.c.), the pretrained model, the dataset, the scenario, the GPU memory request and the number of inference queries that are going to be executed.

We executed two series of experiments. In the first one the workloads consist of Jobs that use the same inference engine and execute the same number of inference queries. We created these workloads in order to understand the impact of the different schedulers in the average 90%-ile and 99%-ile of the inference engines. The two used workloads are described in table 6.1. For each of the workloads we used the intervals *[20, 40]* and *[10, 20]*.

Homogeneous Workloads				
Backend	**Model**	**Dataset**	**Scenario**	**Queries per Pod (Job Number)**
Tensorflow	ssd-mobilenet	COCO (resized 300x300)	Single Stream	1024 (30)
Tensorflow	resnet50	Imagenet Subset	Single Stream	8192 (30)

Table 6.1: Homogeneous Workloads

In the second one, the workloads consist of different inference engines and execute different number of queries. The two used workloads are described in table 6.2. For each of these workloads we used the intervals *[20, 40]* and *[10, 20]*.

Heterogeneous Workload				
Backend	**Model**	**Dataset**	**Scenario**	**Queries per Pod (Job Number)**
ONNX Runtime	mobilenet	Imagenet Subset	Single Stream	1024 (2), 4096 (2), 16384 (2)
Tensorflow	resnet50	Imagenet Subset	Single Stream	4096 (2), 8192 (2), 32768 (2)
Tensorflow	ssd-mobilenet	COCO (Resized 300x300)	Single Stream	512 (2), 1024 (2), 2048 (2)
Tensorflow	ssd-mobilenet (Quantized Finetuned)	COCO (Resized 300x300)	Single Stream	128 (2), 256 (2), 512 (2)
Tensorflow	ssd-mobilenet (Symmetrically Quantized Finetuned)	COCO (Resized 300x300)	Single Stream	64 (2), 512 (2), 2048 (2)

Table 6.2: Heterogeneous Workload

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. *End-to-End 99%-ile*
2. *Workload Duration*
3. *Pending Time Average*
4. *Inference Engine 99%-ile*

□ **GPU Resource Utilization Metrics**

1. *Average SM Utilization Percentage*
2. *Average Memory Utilization (MiB)*
3. *Average Power Consumption (W)*
4. *Average Energy Consumption (J)*

2 Custom Scheduler Results & Scheduler Comparison

2.1 Homogeneous Workload - SSD-MOBILENET

In this subsection, we present the scheduler comparison for the *homogeneous ssd-mobilenet workload*.

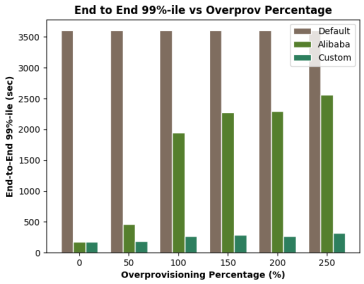
The required GPU memory for this kind of inference engine is approximately *7 GB*. It is understood that in a card with *32 GB* memory, only *4* Pods can be collocated.

MIN = 20 & MAX = 40

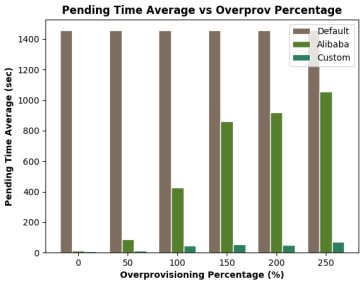
Figure 6.1 shows the end-to-end 99%-ile, the pending time average, the overall workload duration and the inference engine 99%-ile average for all the available schedulers for different over-provisioning percentages. It is clear that our custom scheduler offers better end-to-end 99%-ile, pending time average and overall workload duration from Kubernetes default GPU scheduler extension and Alibaba GPU sharing scheduler extension. Although it has a better behaviour concerning these metrics, the colocation of multiple Pods leads to higher inference engine 99%-ile average.

In order to understand the above results, we should analyze the way each available mechanism schedules Pods to the GPU card. Kubernetes default GPU scheduler extension allocates the whole GPU resource for each Pod, leading to severe increase of the pending time average. The Alibaba GPU share scheduler extension uses a resource agnostic colocation mechanism to schedule Pods in the same card. The success of this scheduler depends on the memory over-provisioning percentage. In particular, for over-provisioning percentage equal to 0% (7 GB memory request per Pod) 4 Pods can be collocated, for 50% (10 GB memory request per Pod) 3 Pods can be collocated, for 100% (14 GB memory request per Pod) 2 Pods can be collocated and for 150%, 200% and 250% each Pod allocates the whole GPU resource. As a result, Alibaba GPU share scheduler extension has similar results with our custom scheduler for over-provisioning percentages equal to 0% and 50%. Our custom scheduler

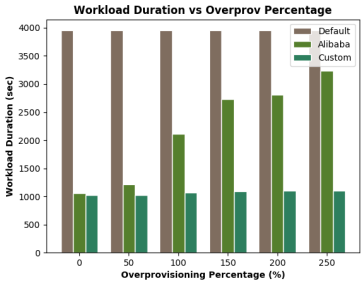
handles the memory over-provisioning problem in a better way because of its resource aware nature. From the presented diagrams we observe that our custom scheduler has similar behavior concerning the QoS metrics for all the different over-provisioning percentages.



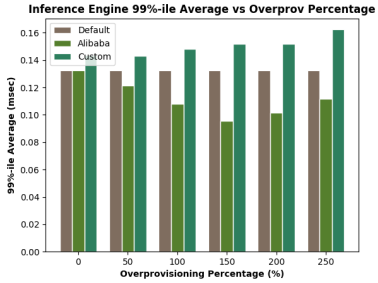
(a) End-to-End 99%-ile



(b) Pending Time Average



(c) Overall Workload Duration



(d) Inference Engine 99%-ile

Figure 6.1: End-to-end 99%-ile, pending time average, workload duration and inference engine 99%-ile vs over-provisioning percentage for ssd-mobilenet homogeneous workload with MIN=20 and MAX=40

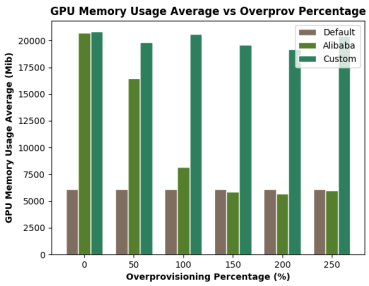
Figure 6.2 shows the memory usage, the SM utilization percentage and the power and energy consumption average for all the available schedulers for different over-provisioning percentages. It is obvious that our collocation mechanism leads to higher memory usage, SM utilization and power consumption, on average. It also has lower energy consumption average from Kubernetes default GPU extension and Alibaba GPU share scheduler extension.

In particular, Kubernetes default GPU extension has the lower resource utilization because each Pod allocates the whole GPU resource. Alibaba GPU share scheduler extension has similar results with our custom scheduler only for 0% and 50% over-provisioning percentages. As we previously explained that happens because for these over-provisioning percentages this scheduler can collocate Pods. The higher the over-provisioning percentage is the closer the resource utilization is to Kubernetes default GPU extension. Finally, we observe that our custom scheduler has similar behavior concerning the resource utilization for all the different over-provisioning percentages.

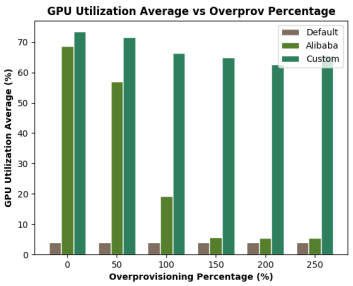
To get better insight about the resource utilization, figures 6.3, 6.4 and 6.5 present the GPU memory consumption, the SM utilization and the power

usage signals we used to create the bar plots of figure 6.2. For each metric the Kubernetes default scheduler results and the comparison of the Alibaba GPU sharing scheduler and the custom scheduler for all the different over-provisioning percentages are shown.

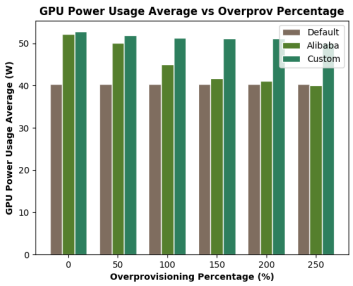
In this experiment, there were not container restarts for none of the different over-provisioning percentages.



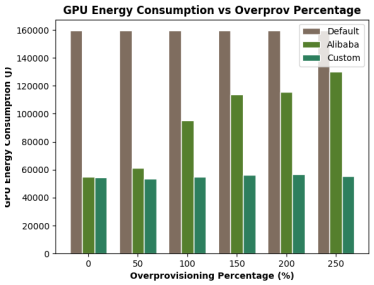
(a) Memory Usage Average



(b) SM Utilization Average

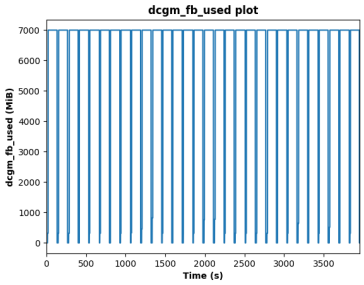


(c) Power Consumption Average

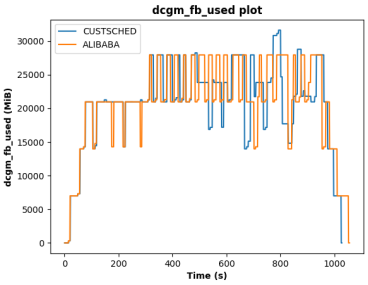


(d) Energy Consumption Average

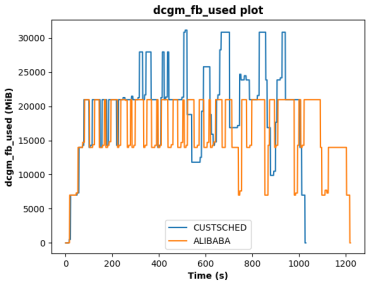
Figure 6.2: Memory usage average, SM utilization average, power and energy consumption averages vs over-provisioning percentage for ssd-mobilenet homogeneous workload with MIN=20 and MAX=40



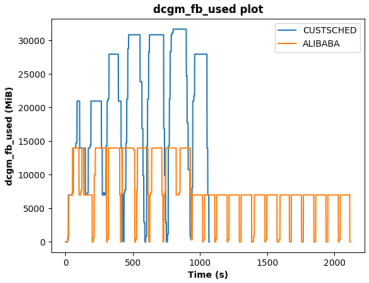
(a) Default



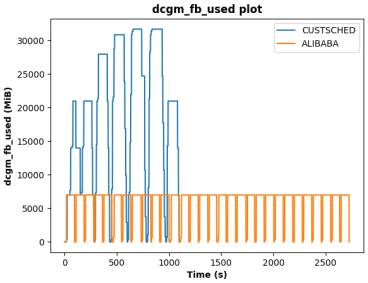
(b) 0% over-provisioning percentage



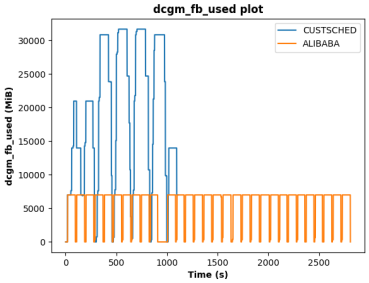
(c) 50% over-provisioning percentage



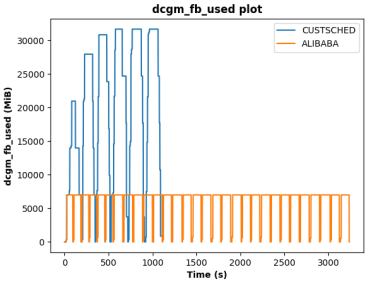
(d) 100% over-provisioning percentage



(e) 150% over-provisioning percentage

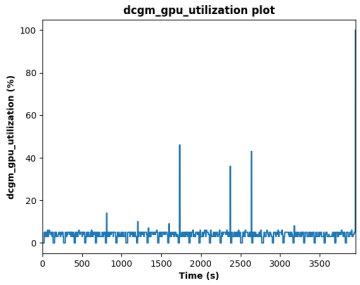


(f) 200% over-provisioning percentage

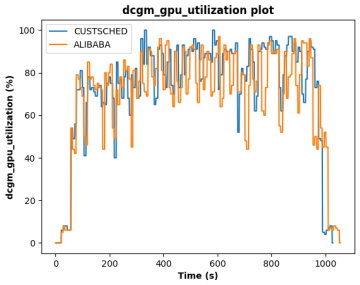


(g) 250% over-provisioning percentage

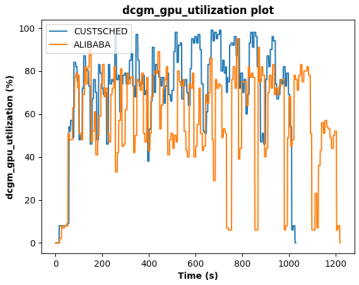
Figure 6.3: Memory usage signals for Kubernetes default scheduler, Alibaba GPU sharing scheduler and custom scheduler for over-provisioning percentages equal to 0%, 50%, 100%, 150%, 200% and 250% for ssd-mobilenet homogeneous workload with MIN=20 and MAX=40



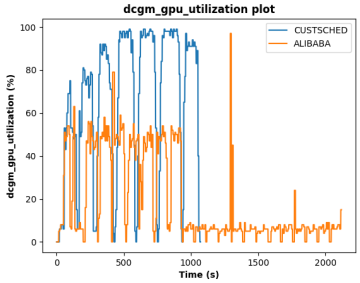
(a) Default



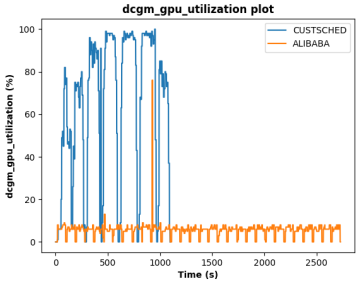
(b) 0% over-provisioning percentage



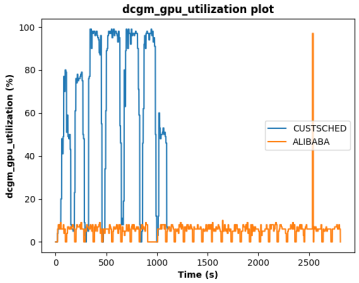
(c) 50% over-provisioning percentage



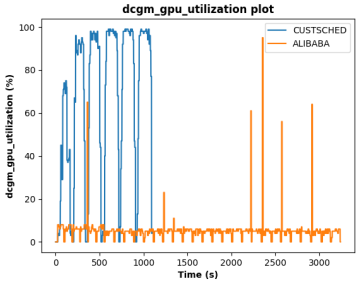
(d) 100% over-provisioning percentage



(e) 150% over-provisioning percentage

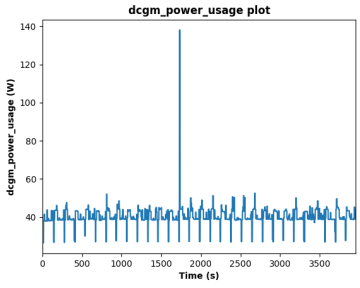


(f) 200% over-provisioning percentage

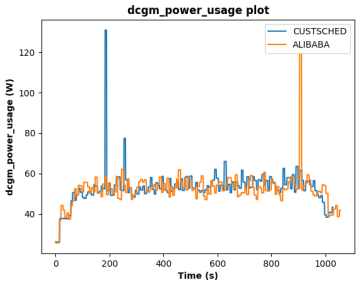


(g) 250% over-provisioning percentage

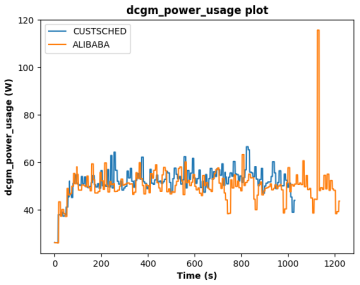
Figure 6.4: SM utilization signals for Kubernetes default scheduler, Alibaba GPU sharing scheduler and custom scheduler for over-provisioning percentages equal to 0%, 50%, 100%, 150%, 200% and 250% for ssd-mobilenet homogeneous workload with MIN=20 and MAX=40



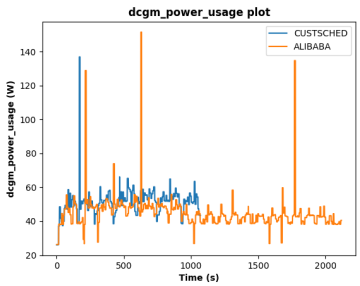
(a) Default



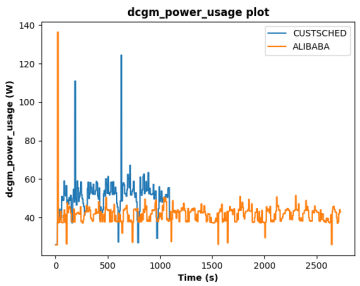
(b) 0% over-provisioning percentage



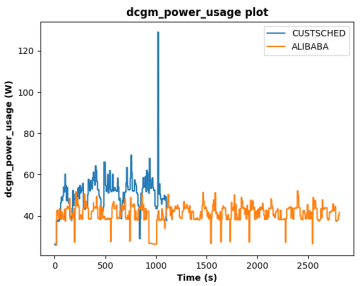
(c) 50% over-provisioning percentage



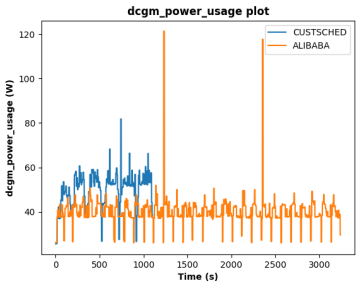
(d) 100% over-provisioning percentage



(e) 150% over-provisioning percentage



(f) 200% over-provisioning percentage



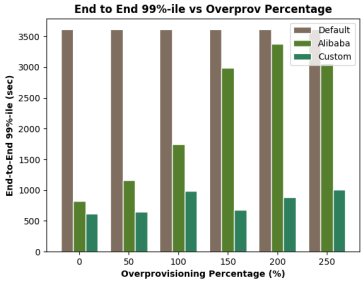
(g) 250% over-provisioning percentage

Figure 6.5: Power usage signals for Kubernetes default scheduler, Alibaba GPU sharing scheduler and custom scheduler for over-provisioning percentages equal to 0%, 50%, 100%, 150%, 200% and 250% for ssd-mobilenet homogeneous workload with MIN=20 and MAX=40

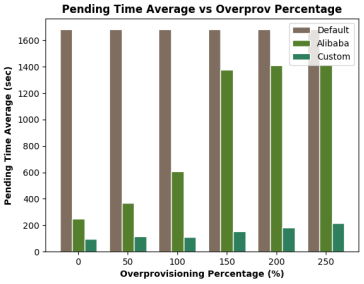
MIN = 10 & MAX = 20

In this experiment, the basic results concerning the Kubernetes default scheduler extension, the Alibaba GPU share scheduler extension and the custom scheduler comparison are the same although the system stress is higher.

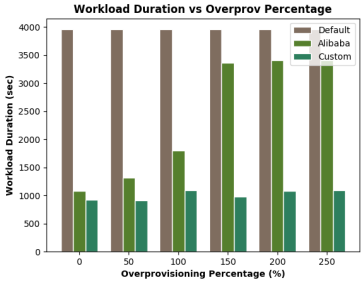
Figure 6.6 shows that the higher stress level leads to higher pending time average and thus higher end-to-end 99%-ile and overall workload duration for all of the available schedulers. It is also observed that the inference engine 99%-ile average values for Kubernetes default scheduler extension and Alibaba GPU share scheduler extension are similar to those presented in figure 6.1. As for the resource utilization, figure 6.7 shows that the metrics' averages do not differentiate much from the ones in figure 6.2 for Kubernetes default scheduler extension and Alibaba GPU share scheduler extension.



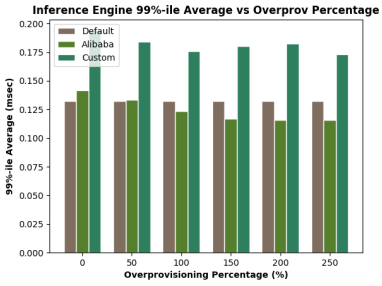
(a) End-to-End 99%-ile



(b) Pending Time Average



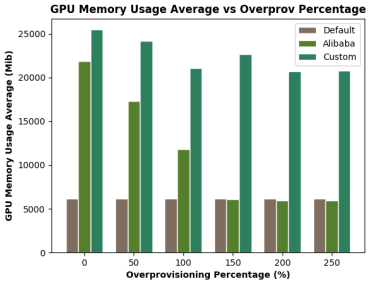
(c) Overall Workload Duration



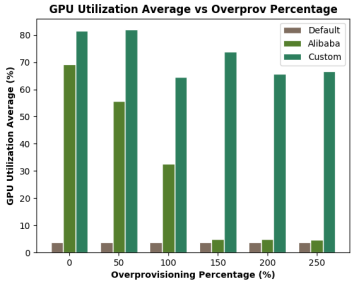
(d) Inference Engine 99%-ile

Figure 6.6: End-to-end 99%-ile, pending time average, workload duration and inference engine vs over-provisioning percentage for ssd-mobilenet homogeneous workload with MIN=10 and MAX=20

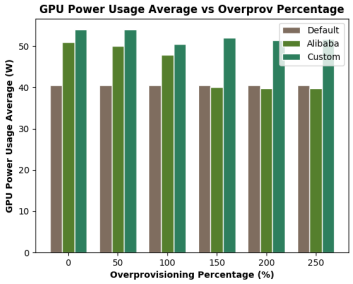
The major difference, for our custom scheduler, is the container restarts because of the higher stressing level. Figure 6.8 reveals that our scheduler in certain scenarios may collocate Pods that should not be collocated, leading to Pod failure due to lack of available memory. These container restarts cause the slightly higher values in the resource utilization metrics for our custom scheduler in figure 6.7. It is also observed that higher over-provisioning percentages lead to lower number of container restarts. That is because of the higher memory requests which are harder to satisfy and thus less Pods are collocated.



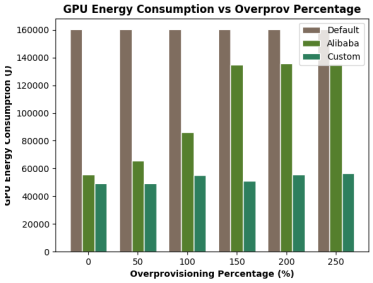
(a) Memory Usage Average



(b) SM Utilization Average

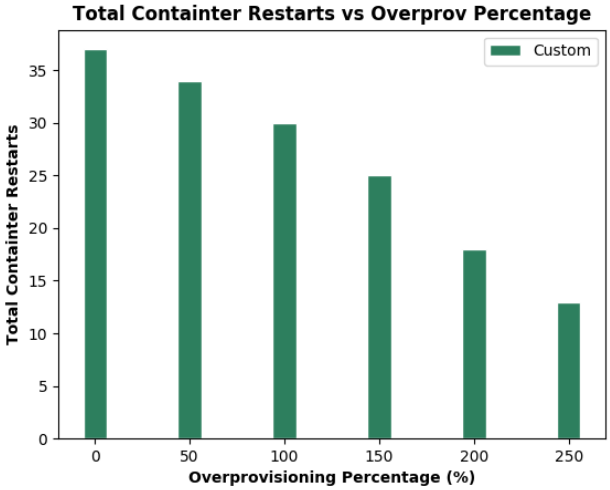


(c) Power Consumption Average



(d) Energy Consumption Average

Figure 6.7: Memory usage average, SM utilization average, power and energy consumption averages vs over-provisioning percentage for ssd-mobilenet homogeneous workload with MIN=10 and MAX=20



(a) Container Restarts

Figure 6.8: Container restarts vs over-provisioning percentage for ssd-mobilenet homogeneous workload with MIN=10 and MAX=20

2.2 Homogeneous Workload - RESNET

In this subsection, we present the scheduler comparison for the *homogeneous resnet workload*.

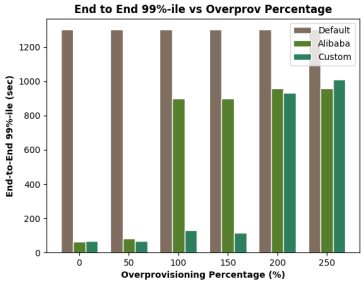
The required GPU memory for this kind of inference engine is approximately *9 GB*. It is understood that in a card with *32 GB* memory, only *3* Pods can be collocated.

MIN = 20 & MAX = 40

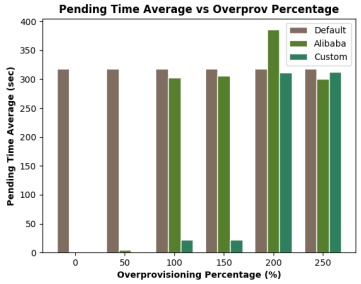
Figures 6.9 and 6.10 show that the results concerning the schedulers' comparison do not differentiate from the homogeneous workload ssd-modilenet results for 0%, 50%, 100% and 150% over-provisioning percentages. It is also observed that all the available schedulers have the same results for 200% and 250% over-provisioning percentages. In order to understand this result we should get insight about how the over-provisioning percentage affects the available schedulers for this kind of workload.

As we mentioned before, the Kubernetes default GPU scheduler extension allocates the whole GPU resource for each Pod, hence it is not affected by the over-provisioning percentage. The Alibaba GPU share scheduler extension collocates Pods based on their memory request. As a result, the mechanism success depends on the memory over-provisioning percentage. In particular, for over-provisioning percentage equal to 0% (9 GB memory request per Pod) 3 Pods can be collocated, for 50% (13 GB memory request per Pod) 2 Pods can be collocated and for 100%, 150%, 200% and 250% each Pod allocates the whole GPU resource. This scheduler has similar results with our custom scheduler for over-provisioning percentages equal to 0% and 50% while for higher over-provisioning percentages its results are closer to the Kubernetes default GPU scheduler extension. Finally, our custom scheduler is less affected by the over-provisioning percentage because of its resource aware nature. Although our custom scheduler is less affected by getting real-time data about the memory usage of the running Pods, the over-provisioning problem remains. It should be clear that it is not possible to know a priory the real GPU request of a Pod. That leads to workload serialization although the Pod real memory request can be satisfied.

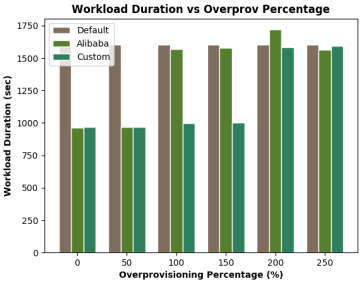
In this experiment, there were not container restarts for none of the different over-provisioning percentages.



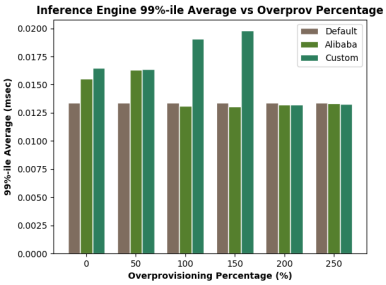
(a) End-to-End 99%-ile



(b) Pending Time Average

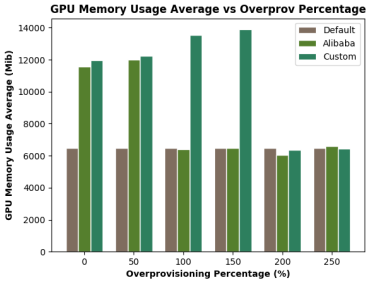


(c) Overall Workload Duration

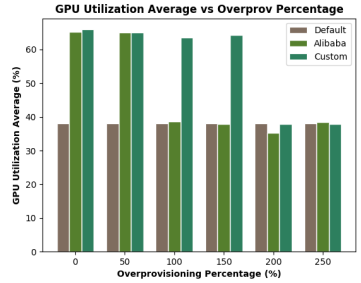


(d) Inference Engine 99%-ile

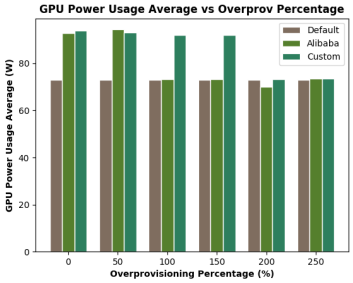
Figure 6.9: End-to-end 99%-ile, pending time average, workload duration and inference engine vs over-provisioning percentage for resnet homogeneous workload with MIN=20 and MAX=40



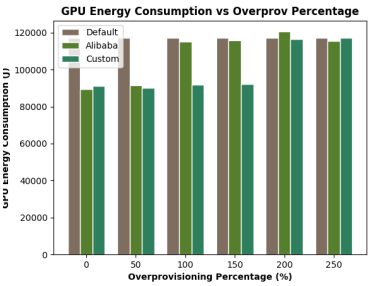
(a) Memory Usage Average



(b) SM Utilization Average



(c) Power Consumption Average

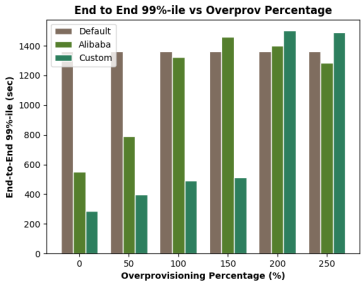


(d) Energy Consumption Average

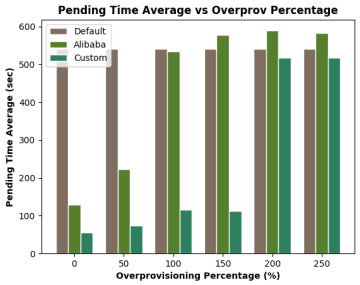
Figure 6.10: Memory usage average, SM utilization average, power and energy consumption averages vs over-provisioning percentage for resnet homogeneous workload with MIN=20 and MAX=40

MIN = 10 & MAX = 20

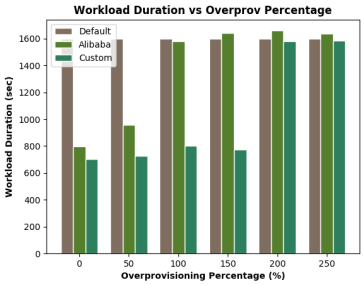
Figures 6.11, 6.12 and 6.13 present the different scheduler results for a higher stressing level. The higher stressing level affects the Kubernetes default GPU extension, the Alibaba GPU share scheduler extension and our custom scheduler the same way it affects these schedulers when the ssd-mobilenet homogeneous workload is used.



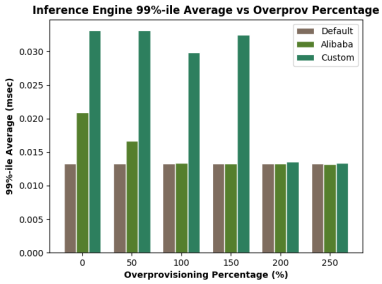
(a) End-to-End 99%-ile



(b) Pending Time Average

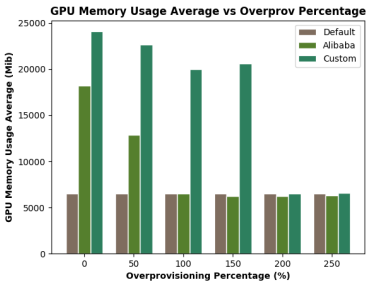


(c) Overall Workload Duration

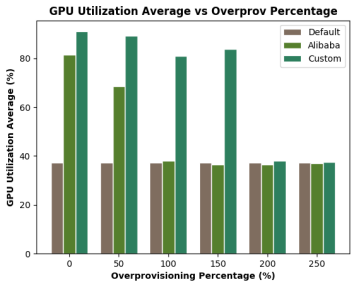


(d) Inference Engine 99%-ile

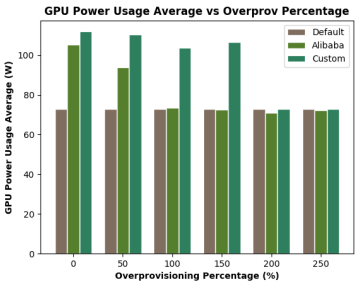
Figure 6.11: End-to-end 99%-ile, pending time average, workload duration and inference engine vs over-provisioning percentage for resnet homogeneous workload with MIN=10 and MAX=20



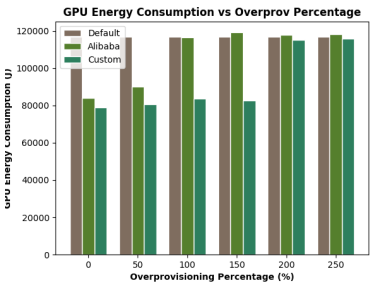
(a) Memory Usage Average



(b) SM Utilization Average

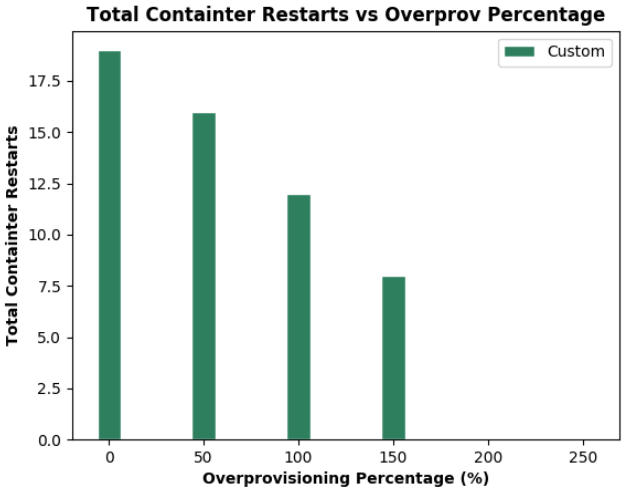


(c) Power Consumption Average



(d) Energy Consumption Average

Figure 6.12: Memory usage average, SM utilization average, power and energy consumption averages vs over-provisioning percentage for resnet homogeneous workload with MIN=10 and MAX=20



(a) Container Restarts

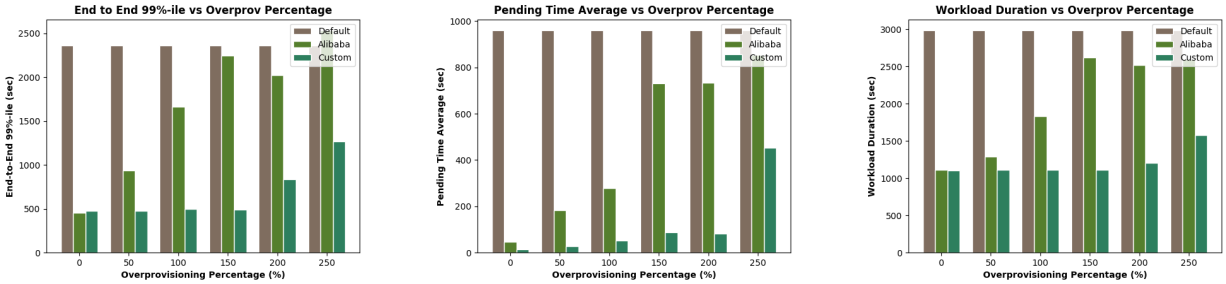
Figure 6.13: Container restarts vs over-provisioning percentage for resnet homogeneous workload with MIN=10 and MAX=20

2.3 Heterogeneous Workload

In this subsection, we present the scheduler comparison for the heterogeneous workload. This workload consists of Pods that create different inference engines which perform a different number of inference queries.

MIN = 20 & MAX = 40

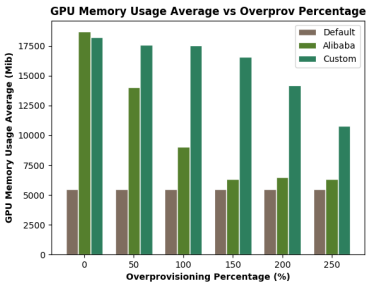
Figures 6.14 and 6.15 show that our custom scheduler offers better QoS and GPU resource utilization metrics compared with Kubernetes default GPU scheduler extension and Alibaba GPU share scheduler extension. We can also observe its consistent behaviour for the vast majority of the over-provisioning percentages.



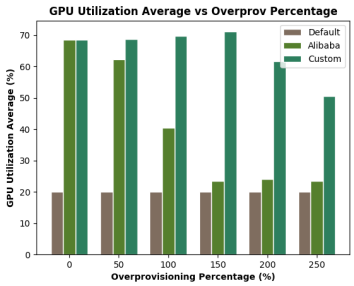
(a) End-to-End 99%-ile (b) Pending Time Average (c) Overall Workload Duration

Figure 6.14: End-to-end 99%-ile, pending time average and workload duration vs over-provisioning percentage for heterogeneous workload with MIN=20 and MAX=40

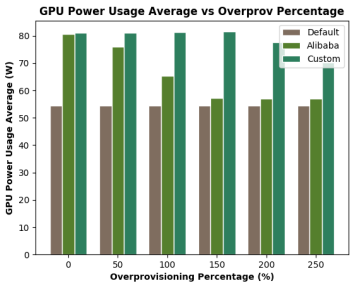
From figure 6.16 we observe the occurrence of container restarts. Container restarts did not occur for the same stressing level in the homogeneous workloads. That happens because of the different memory requests and durations which are specified from the different inference engines and number of executed queries. The workload diversity creates more collocation opportunities while it increases the risk of container restarts.



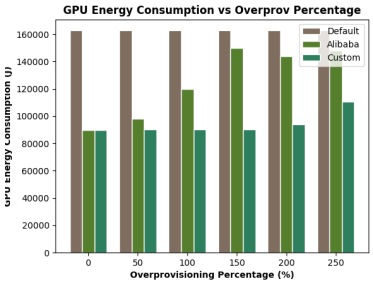
(a) Memory Usage Average



(b) SM Utilization Average

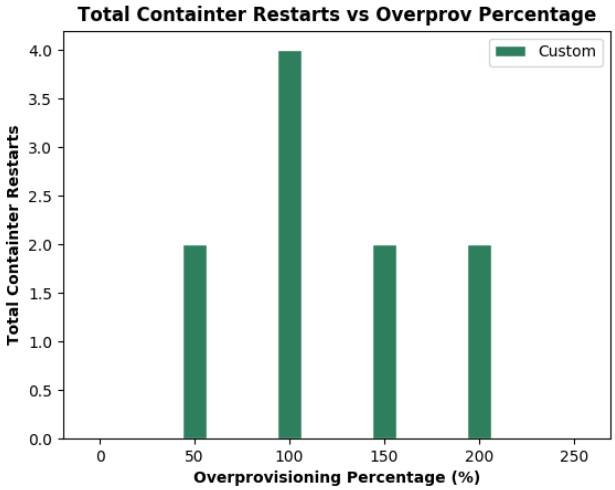


(c) Power Consumption Average



(d) Energy Consumption Average

Figure 6.15: Memory usage average, SM utilization average, power and energy consumption averages vs over-provisioning percentage for heterogeneous workload with MIN=20 and MAX=40

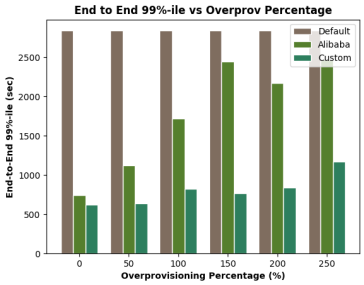


(a) Container Restarts

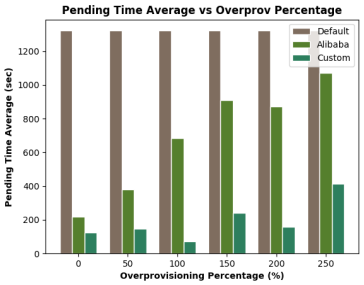
Figure 6.16: Container restarts vs over-provisioning percentage for resnet homogeneous workload with MIN=20 and MAX=40

MIN = 10 & MAX = 20

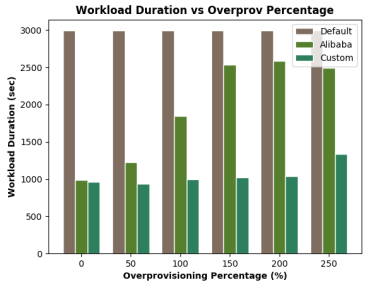
Figures 6.17, 6.18 and 6.19 present the different scheduler results for a higher stressing level. The higher stressing level affects the Kubernetes default GPU extension, the Alibaba GPU share scheduler extension and our custom scheduler the same way it affects these schedulers when the homogeneous workloads were used.



(a) End-to-End 99%-ile

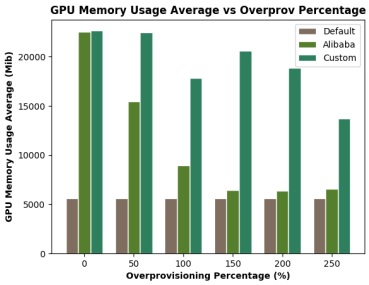


(b) Pending Time Average

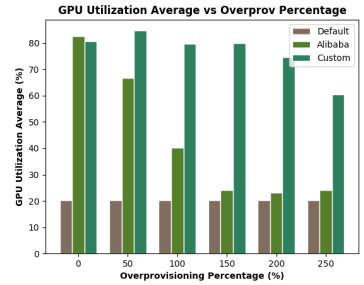


(c) Overall Workload Duration

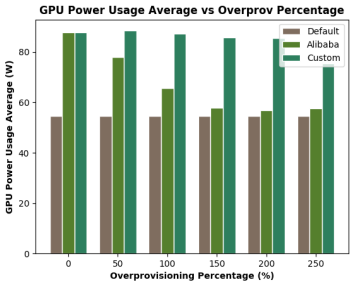
Figure 6.17: End-to-end 99%-ile, pending time average and workload duration vs over-provisioning percentage for heterogeneous workload with MIN=10 and MAX=20



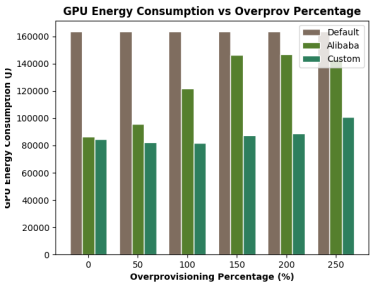
(a) Memory Usage Average



(b) SM Utilization Average

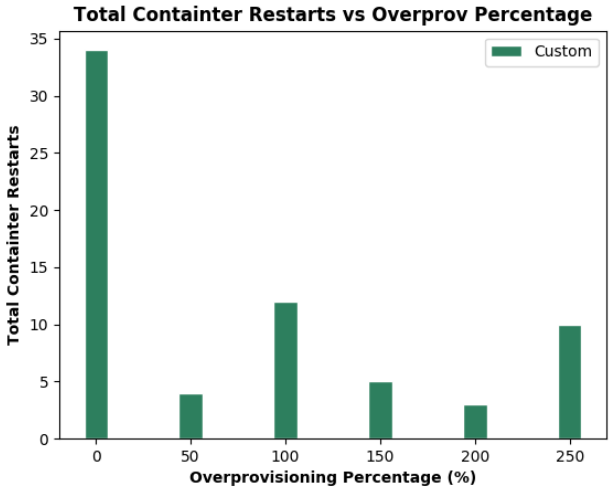


(c) Power Consumption Average



(d) Energy Consumption Average

Figure 6.18: Memory usage average, SM utilization average, power and energy consumption averages vs over-provisioning percentage for heterogeneous workload with MIN=10 and MAX=20



(a) Container Restarts

Figure 6.19: Container restarts vs over-provisioning percentage for heterogeneous workload with MIN=10 and MAX=20

2.4 Model Evaluation

In this subsection, we evaluate the auto regressive model that is used in the *Peak Prediction* part of our algorithm. As we previously explained, we perform *Linear Regression (LR)* on the free GPU memory signal that is exported from the monitoring system.

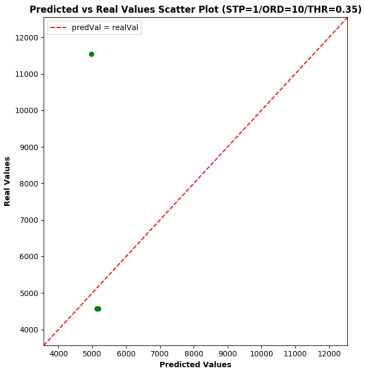
In figures 6.21 and 6.20 we present the results for the ssd-mobilenet homogeneous workload with MIN=20 and MAX=40 for all the different over-provisioning percentages. Although we do not present the model evaluation results for all the executed experiments, the following observations are representative of the model behavior.

For each over-provisioning percentage we present two different diagrams. The first one shows the predicted memory usage versus the actual memory usage scatter plot. In this diagram, the closer the points are to the diagonal line, the better our LR model behaves. The second one shows the line plots for the predicted memory usage and the actual memory usage versus time. In this diagram, the similarity of these two lines indicates that our predictions are accurate.

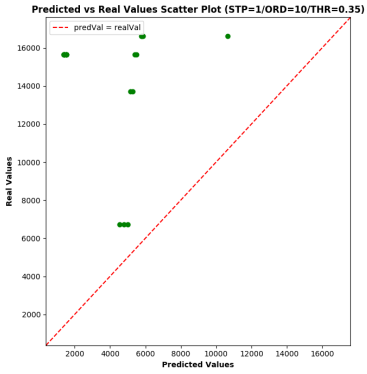
Figure 6.20 shows that the used auto regressive model makes an underestimation of the free GPU memory usage in most of its predictions (the majority of the points are located in the upper left corner). This is the main disadvantage of the auto regressive model. The free GPU memory underestimation leads to incorrect Pod collocation decisions and hence to quality of service violations and resource under utilization. It should be clear that the Linear Regression model is a simplistic approach concerning the free GPU memory prediction.

This model is chosen because of the small size of the input data set and the fast calculation of the slope and intercept of the best fitting line.

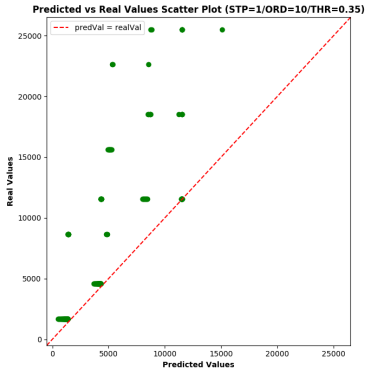
Figure 6.21 gives us insight about whether the LR model follows the actual free memory trend or not. It is obvious that although LR underestimates the free memory usage, in most of the cases it follows the trend of the actual memory usage.



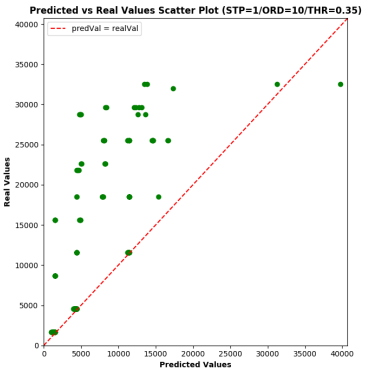
(a) 0% over-provisioning percentage



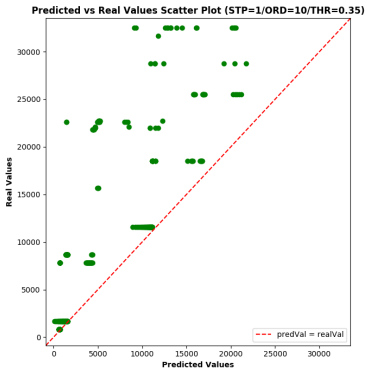
(b) 50% over-provisioning percentage



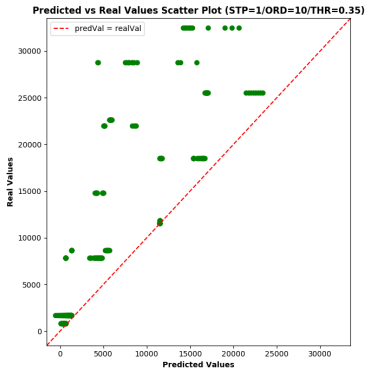
(c) 100% over-provisioning percentage



(d) 150% over-provisioning percentage

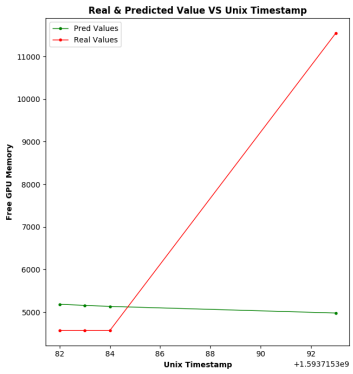


(e) 200% over-provisioning percentage

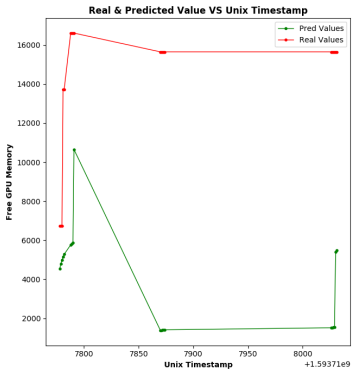


(f) 250% over-provisioning percentage

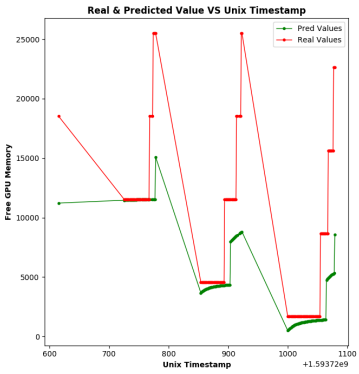
Figure 6.20: Free GPU memory prediction versus actual value scatter plot for ssd-mobilenet homogeneous workload with MIN=20 and MAX=40



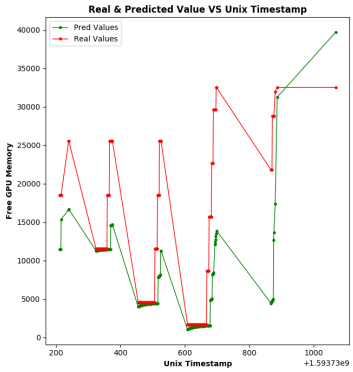
(a) 0% over-provisioning percentage



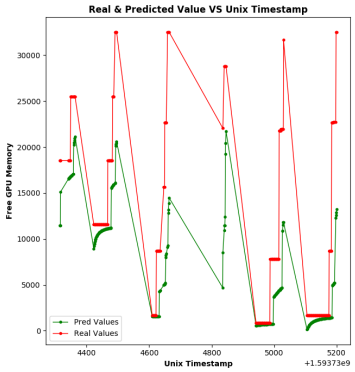
(b) 50% over-provisioning percentage



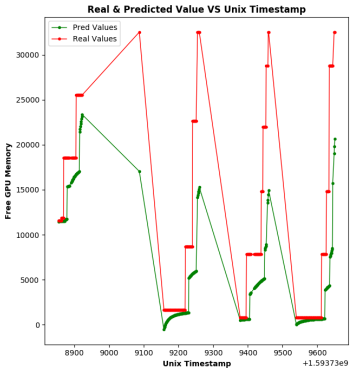
(c) 100% over-provisioning percentage



(d) 150% over-provisioning percentage



(e) 200% over-provisioning percentage



(f) 250% over-provisioning percentage

Figure 6.21: Free GPU memory prediction and actual value line plots for ssd-mobilenet homogeneous workload with MIN=20 and MAX=40

Chapter 7

Conclusion and Future Work

1 Summary

In this thesis, we designed a resource aware GPU collocation mechanism for Kubernetes based on state-of-the-art Kubernetes GPU schedulers. We evaluated the Pod collocation using workloads that consist of inference engines using different scenarios. We showed that in most of the scenarios, our custom scheduler improves the defined quality of service metrics while it increases the GPU resource utilization.

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.

2.1 Development Scope

Regarding to the development of the system, future work could include the expansion of the system for multiple nodes with multiple GPU resources. Furthermore, such a system could be plugged-in Kubernetes project. Metrics extracted from the system will be communicated to the Kubernetes API, as custom resources. Also, the proposed collocation mechanism can be implemented in kube-scheduler native code. Regarding to heterogeneity, the code can be extended to take into consideration other modern system resources such as FPGAs.

2.2 Research Scope

On the other hand, we also suggest some research subjects as proposed future work. First of all, an auto-regressive integrated moving average (ARIMA) model could be used to predict the free GPU memory. Additionally, alongside with the Correlation Based Prediction (CBP) and the Peak Prediction (PP), a system can be used to extract features from the submitted Jobs and make an estimation of the Job duration. This algorithm extension will allow better collocation decisions and hence less quality of service (QoS) violations and better resource utilization. Finally, the SM utilization and the power usage signals can be used for our scheduling decisions. This extension will lead to better collocation decisions and less Pod failures.

Bibliography

[1] A. Tzenetopoulos and D. Soudris, "Interference Aware Container Orchestration in Kubernetes Cluster," NTUA, Tech. Rep., 2019. [Online]. Available: http://artemis.cslab.ece.ntua.gr:8080/jspui/handle/123456789/17454

[2] "Docker Official Website." [Online]. Available: https://www.docker.com/

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

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

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

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

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

[8] 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.

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

[10] P. Thinakaran, J. R. Gunasekaran, B. Sharma, M. T. Kandemir, and C. R. Das, "Kube-Knots: Resource Harvesting through Dynamic Container Orchestration in GPU-based Datacenters," *Proceedings - IEEE International Conference on Cluster Computing, ICCC*, vol. 2019-Septe, pp. 1–13, 2019.

[11] Wikipedia contributors, "Linear regression — Wikipedia, the free encyclopedia," 2020, [Online; accessed 16-July-2020]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Linear_regression&oldid=965415129

[12] W. contributors, "Autoregressive model — Wikipedia, the free encyclopedia," 2020, [Online; accessed 16-July-2020]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Autoregressive_model&oldid=967175472

[13] "Benefits of Cloud Computing." [Online]. Available: https://www.salesforce.com/products/platform/best-practices/benefits-of-cloud-computing/

[14] "Cloud Computing." [Online]. Available: https://searchcloudcomputing.techtarget.com/definition/cloud-computing

[15] Q. Chen, H. Yang, J. Mars, and L. Tang, "Baymax: QoS awareness and increased utilization for non-preemptive accelerators in warehouse scale computers," *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, vol. 02-06-Apri, pp. 681–696, 2016.

[16] Y. Ukidave, X. Li, and D. Kaeli, "Mystic: Predictive scheduling for gpu based cloud servers using machine learning," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2016, pp. 353–362.

[17] R. Phull, C.-H. Li, K. Rao, H. Cadambi, and S. Chakradhar, "Interference-driven resource management for gpu-based heterogeneous clusters," in *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 109–120. [Online]. Available: https://doi.org/10.1145/2287076.2287091

[18] D. Kang, T. J. Jun, D. Kim, J. Kim, and D. Kim, "Convgpu: Gpu management middleware in container based virtualized environment," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, Sep. 2017, pp. 301–309.

[19] J. Gleeson and E. de Lara, "Heterogeneous GPU reallocation," *9th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2017, co-located with USENIX ATC 2017*, 2017.

[20] 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," *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018*, pp. 595–610, 2007.

[21] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo, "Optimus: An Efficient Dynamic Resource Scheduler for Deep Learning Clusters," *Proceedings of the 13th EuroSys Conference, EuroSys 2018*, vol. 2018-January, 2018.

[22] V. Gupta, K. Schwan, N. Tolia, V. Talwar, and P. Ranganathan, "Pegasus: Coordinated scheduling for virtualized accelerator-based systems," 06 2011.

[23] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero, "Enabling preemptive multiprogramming on gpus," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, 2014, pp. 193–204.

[24] P. Aguilera, K. Morrow, and N. S. Kim, "Qos-aware dynamic resource allocation for spatial-multitasking gpus," in *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2014, pp. 726–731.

[25] "Kubernetes GPU Scheduling (Kubernetes Official Website)." [Online]. Available: https://kubernetes.io/docs/tasks/manage-gpus/scheduling-gpus/

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

[27] "MLPerf Inference Load Generator." [Online]. Available: https://mlperf.github.io/inference/loadgen/index.html

[28] "Random Kubernetes Scheduler Analysis (Banzai Cloud Blog)." [Online]. Available: https://banzaicloud.com/blog/k8s-custom-scheduler

[29] Wikipedia contributors, "Coefficient of variation — Wikipedia, the free encyclopedia," 2020, [Online; accessed 16-July-2020]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Coefficient_of_variation&oldid=967405166