Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων

Τεχνικές Δυναμικής Διαχείρισης Μνήμης για την Αποδοτική Χρήση Καρτών Γραφικών στη Διαδραστική Ανάπτυξη Εφαρμογών Μηχανικής Μάθησης

Διπλωματικη Εργασια

ΓΕΩΡΓΙΟΣ Δ. ΑΛΕΞΟΠΟΥΛΟΣ

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2021

Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων

Τεχνικές Δυναμικής Διαχείρισης Μνήμης για την Αποδοτική Χρήση Καρτών Γραφικών στη Διαδραστική Ανάπτυξη Εφαρμογών Μηχανικής Μάθησης

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΓΕΩΡΓΙΟΣ Δ. ΑΛΕΞΟΠΟΥΛΟΣ

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 13η Ιουλίου 2021.

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
.....	.....	.....
Νεκτάριος Κοζύρης	Γεώργιος Γκούμας	Διονύσιος Πνευματικάτος
Καθηγητής Ε.Μ.Π.	Αν. Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2021

.....

Γεώργιος Δ. Αλεξόπουλος
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών ΕΜΠ

Περίληψη

Οι GPU σήμερα είναι μια αναγκαιότητα στην επιτάχυνση υπολογισμών Μηχανικής Μάθησης (Machine Learning - ML). Τα φορτία ML περιλαμβάνουν υπολογιστικά απαιτητικές (throughput-intensive) δουλειές training, ευαίσθητες ως προς το latency δουλειές inference καθώς και δουλειές διαδραστικής ανάπτυξης, όπως τα Jupyter Notebook. Μια συνηθισμένη πρακτική είναι η υποβολή των φορτίων ML ως container διαχειριζόμενα από τον ενορχηστρωτή Kubernetes. Ο τρόπος με τον οποίο διαχειρίζεται τις GPU η πλατφόρμα Kubernetes είναι να αναθέτει μια GPU αποκλειστικά σε μια μόνο δουλειά (job). Αυτή η ένα-προς-ένα σχέση μεταξύ καρτών γραφικών και jobs οδηγεί σε σημαντική υποαξιοποίηση της GPU, ειδικά για διαδραστικές δουλειές οι οποίες χαρακτηρίζονται από μεγάλες περιόδους αδράνειας με εμβόλιμες ριπές χρήσης της GPU. Οι τρέχουσες προσεγγίσεις επιτρέπουν την από κοινού χρήση μιας GPU αναθέτοντας ένα κομμάτι της μνήμης GPU σε κάθε συστεγαζόμενη εργασία.

Παρουσιάζουμε την Alexo Shared GPU, έναν μηχανισμό που επιτρέπει σε πολλές διεργασίες να χρησιμοποιούν την ίδια GPU, καθεμιά από αυτές έχουσα όλη τη μνήμη GPU διαθέσιμη. Αυτό το επιτυγχάνουμε εκμεταλλευόμενοι την NVIDIA Unified Memory, μια λειτουργικότητα των μοντέρνων καρτών NVIDIA, η οποία επιτρέπει στην GPU να χρησιμοποιεί την RAM του συστήματος ως χώρο swap. Η κύρια ιδέα είναι να μετατρέπουμε διαφανώς όλες τις κλήσεις συμβατικής εκχώρησης μνήμης GPU μιας εφαρμογής στις αντίστοιχες της Unified Memory, επιτρέποντας έτσι την αυτόματη χρήση των σφαλμάτων σελίδας στην GPU για να εναλλάξουμε τα περιεχόμενα μνήμης μιας αδρανούς διεργασίας με αυτά μιας άλλης που είναι ενεργή. Ακόμα, σχεδιάζουμε έναν μηχανισμό αποφυγής του thrashing, μιας κατάστασης με καταστροφικές για την επίδοση συνέπειες, η οποία εμφανίζεται όταν οι υπολογιστικές ριπές πολλών συστεγαζόμενων διεργασιών στην GPU επικαλύπτονται και η μνήμη είναι oversubscribed. Ο μηχανισμός αυτός δρομολογεί την GPU αποκλειστικά σε μια εκ των ανταγωνιζόμενων διεργασιών τη φορά για ένα κβάντο χρόνου προκειμένου να περιοριστεί το πλήθος των σφαλμάτων σελίδας.

Καθώς είναι δύσκολο να εκτιμήσουμε το speedup του μηχανισμού μας για διαδραστικές εργασίες χωρίς τη διεξαγωγή μιας μελέτης σε περιβάλλον παραγωγής (το οποίο είναι εκτός εμβέλειας αυτής της διπλωματικής), αξιολογούμε το σύστημά μας σε μη-διαδραστικές (συμβατικές) εργασίες, αφού αντιπροσωπεύουν την worst-case περίπτωση. Επιτυγχάνουμε χρόνους εκτέλεσης κάτω του σειριακού ακόμη και για εξαιρετικά εντατικές (ως προς τη χρήση GPU) εργασίες ML training με λόγο χρήσης GPU/CPU ίσο με 90/10. Ακόμα και σε περιπτώσεις όπου η μνήμη είναι κατά πολύ oversubscribed (άθροισμα Working Sets > 200% GPU memory), παρατηρούμε επιταχύνσεις έως και 35% σε σχέση με τη σειριακή εκτέλεση των εργασιών. Ως εκ τούτου, ο μηχανισμός μας μεγιστοποιεί την αξιοποίηση της GPU όχι μόνο για διαδραστικά φορτία, αλλά ακόμα και για συμβατικές/ακολουθιακές εργασίες Μηχανικής Μάθησης. Τέλος, μπορεί εύκολα να εγκατασταθεί σε οποιαδήποτε υπολογιστική συστοιχία Kubernetes με μια μόνο εντολή.

Λέξεις Κλειδιά

Κάρτα Γραφικών, GPU Sharing, Kubernetes, oversubscription μνήμης GPU, Υπολογιστικό Νέφος, Διαμοιρασμός Πόρων, Μηχανική Μάθηση, Jupyter Notebook

Abstract

GPUs today are a necessity in accelerating Machine Learning. ML workloads comprise throughput-intensive training tasks, latency-sensitive inference tasks and interactive development tasks, such as Jupyter Notebook jobs. A common practice today is to deploy ML jobs as containers managed by the Kubernetes orchestrator. Kubernetes' method of handling GPUs is to assign a whole GPU exclusively to a single job. This one-to-one relationship between GPUs and jobs leads to massive GPU underutilization, especially for interactive jobs which are characterized by large idle periods with intermittent bursts of GPU usage. Current solutions enable GPU sharing by statically assigning a fixed slice of GPU memory to each co-located job. These solutions are not suitable for interactive jobs as a) the number of co-located jobs is limited by the size of physical GPU memory and b) they limit users, as they must know the GPU memory demand of their jobs before submitting them for execution, which is impractical.

We present Alexo Shared GPU, a mechanism that enables multiple applications to share the same GPU, each having the whole GPU memory available. We achieve this by leveraging NVIDIA Unified Memory, a feature of modern NVIDIA GPUs which enables applications to use system RAM as swap space for GPU memory. The key idea is to transparently convert all of an application's GPU memory allocations to Unified, thus enabling the automatic use of GPU Page Faults to swap-in/out the memory of an idle process. We also design a mechanism to prevent GPU thrashing, a performance degradation which can occur when the GPU bursts of co-located applications overlap and memory is oversubscribed. This mechanism enforces exclusive use of the GPU in a time-sliced manner between competing processes to limit page faults.

Because it is difficult to quantify the speedup our mechanism offers for interactive tasks without conducting an extensive study in a production environment (which falls out of the scope of this thesis), we evaluate our system on non-interactive (conventional) tasks, as they represent a worst-case scenario. We are able to achieve sub-serial (faster than sequential) execution times even for extremely GPU-intensive ML training tasks (90/10 GPU/CPU ratio). Even in heavily oversubscribed scenarios (Working Sets sum > 200% GPU memory), we observe speedups of up to 35% compared to serial execution. As such, our mechanism not only maximizes GPU utilization for interactive tasks, but can also significantly increase GPU utilization for conventional ML jobs. Our mechanism can be deployed in any Kubernetes cluster with a single command.

Keywords

Graphics Processing Unit, GPU Sharing, Kubernetes, GPU Memory oversubscription, Cloud Computing, Resource Sharing, Machine Learning, Jupyter Notebook

στη μητέρα μου

Ευχαριστίες

Θα ήθελα να εκφράσω την ευγνωμοσύνη μου προς τους ανθρώπους που συνέδραμαν στην ολοκλήρωση αυτής της διπλωματικής εργασίας, αλλά και στην ευρύτερη ακαδημαϊκή μου πορεία. Καταρχήν, ευχαριστώ πολύ τον επιβλέποντα καθηγητή μου κ. Νεκτάριο Κοζύρη, ο οποίος καλλιέργησε μέσω των μαθημάτων του το ενδιαφέρον μου για τα Υπολογιστικά Συστήματα. Επίσης, ευχαριστώ θερμά τον διδάκτορα Βαγγέλη Κούκη, ο οποίος μου έδωσε την ευκαιρία να εργαστώ μέσα στο περιβάλλον της οικογένειας της Arrikto και ακόμη μου ενστάλαξε την νοοτροπία της συνεχούς επιδίωξης της βαθύτερης ουσίας και κατανόησης κάθε κατάστασης με την οποία βρίσκομαι αντιμέτωπος. Μέσω της Arrikto, ήρθα σε επαφή με τους Ιωάννη Ζαρκάδα και Δημήτρη Πουλόπουλο, στους οποίους είμαι ευγνώμων για τις συμβουλές τους. Ευχαριστώ από καρδιάς τους φίλους μου Θανάση, Ηρακλή και Βασίλη για την ανεκτίμητη συντροφιά τους κάθως και τις ατέλειωτες ώρες συζητήσεων που μοιραστήκαμε. Τέλος, τίποτα από όλα αυτά δε θα ήταν εφικτό χωρίς την αγάπη και τη στήριξη της μητέρας και του πατέρα μου, καθώς και των αδελφών μου Νίκου και Γιάννη.

Γεώργιος Αλεξόπουλος,
Ιούλιος 2021

Περιεχόμενα

Κατάλογος Εικόνων

Κατάλογος Πινάκων

Κεφάλαιο 1

Εισαγωγή

Σε αυτό το πρώτο κεφάλαιο, περιγράφουμε το γενικότερο εύρος της εργασίας μας. Παρέχουμε μια σύντομη επισκόπηση του προβλήματος και των συνεπειών του. Στη συνέχεια, εξετάζουμε τις υπάρχουσες ανοιχτού-κώδικα καθώς και εμπορικές προσεγγίσεις, επισημαίνοντας τις προσφορές τους και τα μειονεκτήματά τους. Προχωρώντας, απεικονίζουμε το κενό που προσπαθούμε να καλύψουμε και δίνουμε μια πρώτη σύνοψη της λύσης μας. Τέλος, παρουσιάζουμε τη δομή του παρόντος κειμένου.

1.1 Κίνητρο

1.1.1 Κάρτες Γραφικών και Μηχανική Μάθηση

Για το σύνολο αυτής της διπλωματικής εργασίας, ασχολούμαστε με τους επεξεργαστές γραφικών NVIDIA. Η NVIDIA κατέχει συντριπτικό μερίδιο αγοράς σε κάρτες γραφικών γενικής χρήσης. Οι GPU είναι μαζικά παράλληλοι επεξεργαστές, οι οποίοι χρησιμοποιούνται σε συνδυασμό με την κεντρική μονάδας επεξεργασίας (CPU) ενός συστήματος ως ισχυροί επιταχυντές. Οι κάρτες γραφικών εμφανίστηκαν στην αγορά πριν από περισσότερες από 2 δεκαετίες για να επιτρέψουν την απόδοση εικόνας σε πραγματικό χρόνο, με κύριο στόχο τα γραφικά βιντεοπαιχνιδιών. Σήμερα, οι GPU είναι πανταχού παρούσες και μπορούμε να τις εντοπίσουμε από smartphones, φορητούς υπολογιστές, κέντρα δεδομένων έως και σε υπερυπολογιστές. Η αρχική κινητήρια δύναμη πίσω από την καινοτομία της GPU ήταν η ζήτηση για όλο και πιο αληθοφανή γραφικά [5]. Αν και η επιτάχυνση γραφικών διατηρεί τη θέση της ως το κύριο κίνητρο, η χρήση GPU για υπολογιστικούς σκοπούς γίνεται όλο και πιο διαδεδομένη. Αυτή η προσέγγιση, γνωστή ως Υπολογισμοί Γενικού Σκοπού σε GPU (GPGPU), υιοθετείται όλο και περισσότερο σε εφαρμογές HPC (High Performance Computing). Ένα εξέχον παράδειγμα αυτού είναι η αυξανόμενη χρήση των μονάδων επεξεργασίας γραφικών στην ανάπτυξη και εφαρμογή συστημάτων μηχανικής μάθησης [6]. Τί είναι όμως αυτό που κάνει τις GPU τόσο κατάλληλο ταίρι για τη Μηχανική Μάθηση·

Η φάση εκπαίδευσης (training) ενός νευρωνικού δικτύου είναι μια εξαιρετικά απαιτητική ως προς τους πόρους διαδικασία. Οι είσοδοι τροφοδοτούνται στο δίκτυο και υποβάλλονται σε επεξεργασία στα κρυφά στρώματά του χρησιμοποιώντας βάρη. Αυτά τα βάρη στη συνέχεια προσαρμόζονται από τη λογική του δικτύου για να οδηγήσουν σε μεταγενέστερες καλύτερες προβλέψεις. Τέλος, το μοντέλο παράγει μια πρόβλεψη για την είσοδο που του δόθηκε. Και οι δύο αυτές λειτουργίες περιλαμβάνουν κυρίως πολλαπλασιασμούς πινάκων. Οι πολλαπλα-

σιασμοί πινάκων ακολουθούν εξευτελιστικά παράλληλα (ridiculously parallel) υπολογιστικά μοτίβα, όπως και οι περισσότερες άλλες πράξεις και μετασχηματισμοί της Γραμμικής Άλγεβρας. Καθώς οι GPU είναι μαζικά παράλληλες από τη φύση τους (έχουν μεγάλο αριθμό απλών πυρήνων), μπορούν να εκτελούν πολλές από αυτές τις απλές πράξεις Γραμμικής Άλγεβρας ταυτόχρονα. Επιπλέον, οι υπολογισμοί στη μηχανική μάθηση απαιτούν το χειρισμό τεραστίων ποσοτήτων δεδομένων - αυτό καθιστά το μεγάλο εύρος ζώνης μνήμης (bandwidth) της GPU καταλληλότερο. Ως εκ τούτου, όταν είναι διαθέσιμες, οι GPUs χρησιμοποιούνται για να επιταχύνουν τη διαδικασία ανάπτυξης μοντέλων μηχανικής μάθησης (ML).

1.1.2 Η ροή εργασιών ενός επιστήμονα Μηχανικής Μάθησης

Για να φτάσουμε στη βασική αιτία του προβλήματός μας, πρέπει πρώτα να κατανοήσουμε τα στάδια που περνάει ένας επιστήμονας μηχανικής μάθησης (ΜΜ) προκειμένου να φτάσει τα μοντέλα ΜΜ του στην παραγωγή (production). Παρά την επιλογή μας των προγραμματιστών ΜΜ ως την κύρια περίπτωση χρήσης/εφαρμογής της μελέτης μας και ως αφετηρία αυτής, το τελικό προϊόν μας έχει ένα πολύ ευρύτερο φάσμα εφαρμογών.

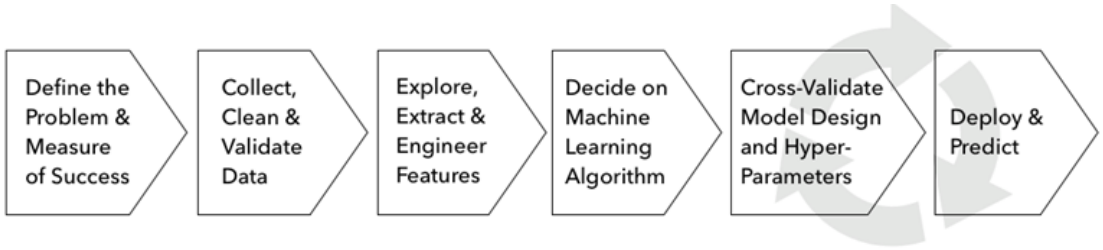

Figure 1.1: *Ροή εργασιών ενός μηχανικού Μηχανικής Μάθησης*

Ακολουθεί μια αριθμημένη λίστα βημάτων που περιγράφουν τη ροή εργασίας ενός επιστήμονα μηχανικής μάθησης:

1. **Εύρεση ενός προβλήματος προς επίλυση**: Αυτονόητο, επίσης υποχρεωτικό.
2. **Απόκτηση δεδομένων**: Αυτό είναι ένα από τα πιο κρίσιμα μέρη της διαδικασίας καθώς τα δεδομένα είναι ένα από τα πιο πολύτιμα ψηφιακά προϊόντα.
3. **Προεπεξεργασία δεδομένων και μεταβλητών**: Αυτό το μέρος περιλαμβάνει την φόρτωση δεδομένων σε δομές γλώσσας υπολογιστή, το φιλτράρισμα των διαθέσιμων δεδομένων (πετώντας λανθασμένες καταχωρήσεις, ακραίες τιμές) και τέλος χειρισμό κατηγορηματικών μεταβλητών, παρέχοντας έναν τρόπο εκχώρησης αριθμητικών τιμών σε αυτές. Τα Jupyter Notebooks είναι ένα πολύ δημοφιλές περιβάλλον ανάπτυξης για αυτό το βήμα.
4. **Δημιουργία ενός βασικού μοντέλου ΜΜ**: Σκιαγράφηση ενός αρχικού μοντέλου και επικύρωση ότι οι υψηλού επιπέδου παραδοχές που έκανε ο επιστήμονας στο πρόβλημα στέκουν. Αυτό το βήμα περιλαμβάνει επίσης σύντομες και επαναλαμβανόμενες ριπές εκπαίδευσης (training) και αξιολόγησης (evaluation), τις οποίες ο επιστήμονας χρησιμοποιεί προκειμένου να αποφασίσει εάν θα δεσμευτεί για μια πιο ολοκληρωμένη εφαρμογή ή θα επιστρέψει στον πίνακα σχεδίασης.
5. **Ανάπτυξη του πλήρους μοντέλου:**

(α') Σε ένα IDE / VSCode / PyCharm ως Python module
(β') Πέρα ως πέρα ανάπτυξη σε Jupyter Notebook διαδραστικά (αυτή είναι και η περίπτωση που μας ενδιαφέρει)

6. **Εκπαίδευση του μοντέλου**: Αυτό είναι το πιο υπολογιστικά απαιτητικό βήμα και συνήθως γίνεται με μη-διαδραστικό τρόπο καθώς υποβάλεται ως δουλειά σε ένα batch σύστημα.
7. **Ικανοποίηση αιτημάτων συμπερασμού (inference) για το μοντέλο**: Χρήση του τελικού, εκπαιδευμένου μοντέλου για πραγματικές προβλέψεις.

Για τα βήματα 4,5 και περιστασιακά 6, οι περισσότεροι επιστήμονες ΜΜ δουλεύουν σε Jupyter Notebook ([7]). Οι εφαρμογές Jupyter Notebook είναι μακροχρόνιες (λάβετε υπ' όψιν αυτό για αργότερα) διαδραστικές διαδικασίες που παρέχουν ένα εξαιρετικό περιβάλλον για την αντιμετώπιση τέτοιων προβλημάτων, που περιλαμβάνουν ενασχόληση με ένα κώδικα κατ' επανάληψη, την πραγματοποίηση αλλαγών, την επαναξιολόγηση, έως ότου το αποτέλεσμα είναι ικανοποιητικό. Αυτό είναι το κύριο κοινό-στόχος των συνεισφορών της δουλείας μας, του οποίου τις ζωές προσπαθούμε να βελτιώσουμε.

Έχοντας υπ' όψιν όλα τα παραπάνω, μπορούμε τώρα να προχωρήσουμε στην διατύπωση του προβλήματος.

1.2 Διατύπωση του προβλήματος

1.2.1 Οι χρήστες

Επιστήμονες της Μηχανικής Μάθησης που κάνουν διαδραστική ανάπτυξη σε ένα περιβάλλον όπως το Jupyter Notebook. Θα προσεγγίσουμε το ζήτημα από την οπτική τους γωνία.

1.2.2 Η τρέχουσα κατάσταση στον upstream Kubernetes

Ο de facto τρόπος χειρισμού GPU σε μια υπολογιστική συστοιχία διαχειριζόμενη από Kubernetes είναι μέσω του μηχανισμού device plugin της NVIDIA [8]. Όταν ένας χρήστης θέλει δουλέψει (μέσω ενός Pod/container) πάνω σε μια GPU στον Kubernetes, δηλώνει nvidia.com/gpu: "X" (όπου X ακέραιος αριθμός) στην αίτηση (request) του και το device plugin εκθέτει αυτόματα την (τις) GPU(s) όταν το container του χρήστη ξεκινά την εκτέλεσή του. Η ανάθεση GPUs σε Pods είναι αποκλειστική, υπό την έννοια πως αν μια GPU «δεθεί» σε ένα Pod από τον δρομολογητή, τότε δεν είναι διαθέσιμη στην υπόλοιπη συστοιχία για όσο χρόνο υπάρχει το Pod. Αναλύουμε διεξοδικά τον μηχανισμό ανάθεσης GPU σε περιβάλλον Kubernetes στην Ενότητα **;;**.

1.2.2.1 Λόγοι πίσω από την αποκλειστική ανάθεση

Οι προγραμματιστές (εργαζόμενοι της NVIDIA) πίσω από το nvidia-device-plugin επέλεξαν την αποκλειστική ανάθεση των GPU σε Pod επειδή:

- Οι διεργασίες που χρησιμοποιούν την ίδια GPU (εκτελούνται στον ίδιο κόμβο) ανταγωνίζονται πάνω στο ίδιο πεπερασμένο (ίσο με το μέγεθος της Κάρτας Γραφικών) σύνολο φυσικής μνήμης για τις εκχωρήσεις τους: Προς το παρόν, οι GPU δεν μπορούν να χειριστούν σφάλματα σελίδας (page faults) για κανονικές (non-Unified) εκχωρήσεις. Δεδομένου ότι οι διεργασίες μπορούν να μεγαλώσουν και να συρρικνώσουν τη μνήμη GPU τους δυναμικά και τα αιτήματα εκχώρησης αντιμετωπίζονται με σειρά προτεραιότητας, μπορεί να εμφανιστούν σενάρια όπου μία ή και οι δύο διαδικασίες μπορεί να αποτύχουν με σφάλματα Out-of-Memory (OOM). Παρόλο που κάθε διεργασία δεν μπορεί να επηρεάσει τα περιεχόμενα της μνήμης GPU οποιασδήποτε άλλης, καθώς κάθε CUDA Context (το αντίστοιχο της διεργασίας για την GPU) διαθέτει ξεχωριστούς πίνακες σελίδων, η NVIDIA δεν παρέχει τρόπο απομόνωσης της χρήσης όγκου μνήμης (σύνολο εκχωρηθείσας μνήμης) GPU μεταξύ πολλών διεργασιών.

- Η δρομολόγηση των εντολών GPU (εκτελέσεις kernel, αντιγραφές μνήμης (memory copies» οι οποίες υποβάλλονται από διαφορετικές διεργασίες καθώς και η εναλλαγή context διαχειρίζονται από τον οδηγό της συσκευής με μη δημοσιευμένο τρόπο. Δεν ύπάρχει καταγεγραμμένος τρόπος πυροδότησης ενός context switch στην GPU. Έτσι, μια από κοινού χρησιμοποιούμενη (shared) GPU δεν μπορεί να προσφέρει εγγυήσεις QoS καθώς και τη χρονικά φραγμένη ικανοποίηση αιτημάτων.

1.2.3 Το πρόβλημα

οι GPUs υποαξιοποιούνται. Δεν υπάρχει επιλογή για συστέγαση δουλειών (Kubernetes Pods) στην ίδια GPU. Αυτό είναι ιδιαίτερα σπάταλο για δουλειές διαδραστικής ανάπτυξης Μηχανικής Μάθησης (Interactive ML Development), όπου είναι και το κύριο αντικείμενο της δουλείας μας, αλλά και για Inference.

Ακολουθούν ορισμένες περιπτώσεις που οι χρήστες εκφράζουν το πρόβλημα:

- "GPUs cannot be shared - GPUs must be shared" from the Jupyter forums [9]

- "Is sharing GPU to multiple containers feasible?" Github Issue on the Kubernetes Repository [10]

1.2.3.1 Παράγοντες επιδείνωσης του προβλήματος

Οι διαδραστικές εργασίες ανάπτυξης ML (Notebooks):

- Δεν πραγματοποιούν προκαθορισμένο όγκο δουλειάς (όπως θα έκανε μια Training δουλειά). Οι χρόνοι εκτέλεσής τους είναι μη φραγμένοι και δεν είναι εκ των προτέρων υπολογίσιμοι.

- είναι εργασιές με μεγάλο χρόνο εκτέλεσης των οποίων τα μοτίβα χρήσης της GPU χαρακτηρίζονται από υπολογιστικές ριπές και από μεγάλες περιόδους αδράνειας (κατά την αναδιαμόρφωση του κώδικα, το debugging και τα διαλείμματα του προγραμματιστή).

Ενώ το πρόβλημα της αποκλειστικής εκχώρησης των GPU μπορεί να λυθεί τετρριμμένα (για παράδειγμα, με τροποποίηση του device plugin για τη διαφήμιση μεγαλύτερου αριθμού nvidia.com/gpu από ότι υπάρχουν φυσικές GPU, το βασικό ζήτημα **είναι αυτό της διαχείρισης της τριβής μεταξύ συστεγαζόμενων εργασιών** (πώς συμπεριφέρονται 2+ διεργασίες στον ίδιο κόμβο, ανεξάρτητα από το Kubernetes) και αυτό είναι δύσκολο να λυθεί.

1.2.3.2 Ένα σενάριο εκτός Kubernetes

Για να δείξουμε παραστατικότερα το πρόβλημα που αντιμετωπίζουμε, ας εξετάσουμε ένα πραγματικό σενάριο στο οποίο ο τρέχων χειρισμός της GPU (μνήμη) είναι αναποτελεσματικός και οδηγεί σε σπατάλη πόρων.

Ας υποθέσουμε ότι οι χρήστες A και B έχουν και οι δύο πρόσβαση σε έναν κοινό server με 1 GPU. Όταν ο χρήστης A κάνει διαδραστική (ML) ανάπτυξη που χρησιμοποιεί GPU σε Jupyter Notebook, τότε η μνήμη GPU που εκχωρεί (πιο συγκεκριμένα η βιβλιοθήκη ML που χρησιμοποιεί) δεν θα απελευθερωθεί έως ότου ο πυρήνας (kernel) IPython [1] (η backend διεργασία του Notebook) τερματιστεί. Αυτό σημαίνει ότι εάν ο χρήστης A τρέξει κάποια κελιά και μετά αποφασίσει να τροποποιήσει τον κώδικά του, ή ακόμη και να πάει για καφέ / περίπατο, η εκχωρηθείσα μνήμη GPU δεν θα είναι διαθέσιμη σε άλλους χρήστες / Notebooks / διεργασίες οποιουδήποτε είδους. Κατά συνέπεια, όταν ο B αποφασίσει να εκκινήσει ένα Notebook και να κάνει επίσης δουλειά, θα έχει πρόσβαση μόνο στην εναπομείνασα μνήμη GPU.

Χειρισμός της μνήμης της GPU από ML Frameworks

Τα ML frameworks προτιμούν να χειρίζονται τη μνήμη της GPU μέσω εσωτερικών sub-allocators. Ως εκ τούτου, ζητούν μνήμη GPU σε μεγάλα κομμάτια και συνήθως ξεπερνούν την πραγματική «ζήτηση». Επιπλέον, το Tensorflow [11] από προεπιλογή εκχωρεί όλη τη μνήμη της GPU. Αυτή η συμπεριφορά μπορεί προαιρετικά να αλλάξει προκειμένου να επιτρέψει τη χρήση της μνήμης GPU να αυξάνεται ανάλογα με τις ανάγκες. Ωστόσο, αυτή η χρήση δεν θα συρρικνωθεί ποτέ. Εάν η «πραγματική» ανάγκη μιας εργασίας ML κυμαίνεται από 500MiB σε 2,5 GiB, και ύστερα πίσω στα 500MiB, τότε η εκχωρηθείσα μνήμη GPU θα είναι 2,5 GiB έως ότου ολοκληρωθεί η διεργασία TF.

Έτσι θα βρεθούν αντιμέτωποι με τα ακόλουθα:

- Πολλά από τα ML μοντέλα τους δεν θα χωράνε στην μνήμη της GPU / δεν θα μπορούν καν να εκκινήσουν (OOM).
- Δεν θα έχουν πρόσβαση στο ίδιο περιβάλλον με την εκπαίδευση (Training), οπότε δεν θα μπορούν να αξιολογήσουν αντικειμενικά τα μοντέλα τους.

[1] Ένας IPython kernel είναι μια διεργασία διερμηνευτή Python που λειτουργεί ως backend για τα Jupyter Notebooks. Το Notebook πακετάρει τον κώδικα χρήστη σε μορφή JSON, τον προωθεί στον IpyKernel ο οποίος με τη σειρά του τον αποτιμά/εκτελεί και τελικά το αποτέλεσμα αποστέλλεται πίσω στο Notebook για εμφάνιση.

1.2.3.3 Χαρακτηριστικά μιας πλήρους λύσης

Κατά συνέπεια, κάθε πλήρης λύση πρέπει να περιλαμβάνει:

- έναν **μηχανισμό** απομόνωσης της χρήσης GPU μεταξύ διεργασιών στον ίδιο κόμβο και τη διευκόλυνση της κοινής χρήσης.
- Έναν τρόπο έκθεσης αυτού του μηχανισμού στο σύστημα Kubernetes, μέσω ειδικών πόρων (custom resources) και μορφοποίηση των αιτημάτων των χρηστών με κατάλληλο τρόπο.

1.3 Επισκόπηση και περιορισμοί υπαρχουσών προσεγγίσεων

1.3.1 Σύνοψη υπαρχουσών προσεγγίσεων

Υπάρχει μια πληθώρα προσεγγίσεων για την κοινή χρήση (sharing) GPU. Οι συγγραφείς του Kubeshare [12] παρέχουν έναν τρόπο απομόνωσης της χρήσης GPU (μνήμης) μεταξύ διεργασιών με διαχωρισμό της μνήμης GPU και εκχώρηση ενός κλάσματος σε κάθε διεργασία. Ως αποτέλεσμα, κάθε διαδικασία έχει ένα εγγυημένο όριο μνήμης GPU, το οποίο δεν μπορεί να υπερβεί. Προσφέρει επίσης μια ενσωμάτωση στον Kubernetes, μέσω της χρήσης ενός προσαρμοσμένου δρομολογητή και ειδικών Annotation στα Pods του χρήστη. Ωστόσο, η χρήση της μεθόδου του Kubeshare για την εκχώρηση GPU απαιτεί σημαντικές αλλαγές σε όλα τα εμπλεκόμενα στοιχεία (δεν είναι μόνο θέμα αιτήματος διαφορετικού πόρου αντί για "nvidia.com/gpu") και επομένως είναι ένα μη εύκολα εφαρμόσιμο σενάριο. Το Aliyun (Alibaba Cloud) Scheduler Extender [13] παρέχει έναν τρόπο παράκαμψης της αποκλειστικής εκχώρησης GPU σε Pods, προσφέροντας ένα εναλλακτικό device plugin (οι χρήστες ζητούν "aliyun.com/gpu"). Οι χρήστες μπορούν επίσης να δηλώσουν αιτήματα μνήμης GPU (GPU Memory Requests) για τα Pod τους, τα οποία χρησιμοποιούνται για το bin-packing (διαμοιρασμό σε «κάδους») από τον δρομολογητή. Ωστόσο, μετά τη δρομολόγηση, το Aliyun δεν προσφέρει κανέναν τρόπο για την επιβολή αυτών των αιτημάτων μνήμης GPU. Οι διεργασίες εξακολουθούν να παρεμβαίνουν, δυνητικά καταστρεπτικά (OOM), μεταξύ τους με τον ίδιο τρόπο που κάνουν δύο διεργασίες GPU σε ένα σενάριο εκτός Kubernetes. Η Alibaba παρέχει έναν proprietary τρόπο για την επιβολή αυτών των ορίων μνήμης σε χρήστες της πλατφόρμας Alibaba Cloud μέσω του cGPU [14], με τρόπο παρόμοιο με το Kubeshare (αυστηρό άνω όριο μνήμης που καθορίζει ο χρήστης πριν τη δρομολόγηση). Υπάρχουν διάφορες άλλες εμπορικές λύσεις ([15], [16], [17], [18]). Το Amazon Elastic Inference ακολουθεί μια εντελώς διαφορετική προσέγγιση και εικονικοποιεί την GPU σε επίπεδο εφαρμογής. Περιορίζεται μόνο σε σενάρια Inference και απαιτεί τη χρήση proprietary λογισμικού της Amazon, με περιορισμένη εφαρμογή στην περίπτωση χρήσης μας. Όλες οι εμπορικές λύσεις που σχετίζονται με τη δική μας περίπτωση χρήσης ακολουθούν το ίδιο σχήμα με το Kubeshare, παρέχοντας έναν τρόπο κλασματικής εκχώρησης της μνήμης της GPU σε διαφορετικές διεργασίες, ώστε να επιτρέψουν την από κοινού χρήση της GPU. Ωστόσο, για να χρησιμοποιήσουν τέτοια σχήματα αποτελεσματικά, οι χρήστες πρέπει να γνωρίζουν την ακριβή χρήση της μνήμης της εφαρμογής τους εκ των προτέρων, κάτι που δεν συμβαίνει όταν αναπτύσσουν μοντέλα ML.

1.3.2 Αδυναμίες υπαρχουσών προσεγγίσεων

Όλες σχεδόν οι υπάρχουσες λύσεις, είτε ανοιχτού-κώδικα είτε εμπορικές/proprietary προσφέρουν τα ίδια ακριβώς πράγματα. Ως εκ τούτου, κάθε μια από αυτές έχει τα ίδια μειονεκτήματα, ειδικά όσον αφορά την περίπτωση χρήσης μας.

- **Σκληρό όριο στη μνήμη GPU για κάθε διεργασία:**
  - Όλες οι υπάρχουσες προσεγγίσεις επιβάλλουν ένα σκληρό όριο στη μνήμη GPU που πρέπει να καθοριστεί κατά το χρόνο υποβολής. Αυτό έρχεται σε αντίθεση με τη διαδραστική και συνεχώς μεταβαλλόμενη φύση μιας δουλειάς τύπου Jupyter Notebook, της οποίας η χρήση μνήμης είναι συνήθως αδύνατο να εκτιμηθεί εκ των προτέρων.
  - Επιπλέον, το σκληρό όριο περιορίζει την ευελιξία της ροής εργασίας του χρήστη όσον αφορά τη δοκιμή σταδιακά μεγαλύτερων μοντέλων. Οι χρήστες θα πρέπει είτε να ζητήσουν μια μεγάλη ποσότητα μνήμης που θα παραμείνει ως επί το πλείστον αχρησιμοποίητη, είτε να βρεθούν αντιμέτωποι με πιθανό σφάλμα OOM.

- **Το πλήθος των συστεγαζόμενων διεργασιών περιορίζεται από το φυσικό μέγεθος της μνήμης της GPU. (όχι oversubscription)**
  - Όπως αναφέραμε, για κανονικές εκχωρήσεις μνήμης CUDA, το άθροισμα αυτών από όλες τις διεργασίες πρέπει να είναι μικρότερο από τη φυσική χωρητικότητα μνήμης GPU. Ως αποτέλεσμα, σε όλα τα υπάρχοντα σχήματα, ο αριθμός των διεργασιών που μπορούν να συστεγαστούν περιορίζεται από αυτήν τη χωρητικότητα μνήμης. Το άθροισμα των κλασμάτων μνήμης που έχουν εκχωρηθεί στις εφαρμογές που βρίσκονται στην ίδια GPU δεν μπορεί να υπερβαίνει τη χωρητικότητα μνήμης ή είναι πολύ πιθανό να προκύψουν σφάλματα OOM. Δεν υπάρχει επιλογή για oversubscription (υπερδέσμευση) της μνήμης.

- **Η υποαξιοποίηση της GPU παραμένει:**
  - Ενώ η εκχώρηση δεν είναι πλέον αποκλειστική μεταξύ Pods και GPUs, ο νέος πόρος που χρησιμοποιείται για το bin packing είναι η μνήμη GPU. Αυτό το νέο κριτήριο, σε συνδυασμό με τη φύση του διαδραστικού φόρτου εργασίας (ριπή υπολογισμού - αδράνεια - επόμενη ριπή) εξακολουθεί να επιτρέπει σενάρια υποαξιοποίησης. Εάν μια GPU έχει μνήμη 4 GB και μια διεργασία A ζητά 2,5, τότε μια άλλη διεργασία B που ζητά 2 δεν θα δρομολογηθεί ανεξάρτητα από την πραγματική χρήση μνήμης της A. Αυτό είναι αποδεκτό σε περιπτώσεις υπολογιστικά εντατικών εργασιών, όπως το ML Training. Ωστόσο, για διαδραστικές εργασίες, η χρήση της GPU από την διεργασία A θα είναι γενικά πολύ χαμηλή καθ όλη τη διάρκεια της (μη υπολογίσιμη) και η GPU στο σύνολό της παραμένει υποαξιοποιημένη.

1.4 Η προσέγγισή μας (alexo-device-plugin)

Όπως τονίσαμε προηγουμένως, το βασικό ζήτημα δεν έγκειται στην δρομολόγηση του Kubernetes, αλλά σε αυτό που ακολουθεί. Ο στόχος μας είναι ένας μηχανισμός που επιτρέπει 2+ διαδραστικές διεργασίες ανάπτυξης ML (Notebooks) να συνυπάρχουν σε μια κοινόχρηστη GPU χωρίς σκληρό άνω όριο μνήμης και στη συνέχεια να την εκθέσουμε στον Kubernetes. Αναπτύξαμε αρχικά έναν μηχανισμό που αντιμετωπίζει το πρόβλημα (libunified) και μετά τον ενσωματώσαμε με τον Kubernetes (alexo-device-plugin).
Ακολουθεί μια αριθμημένη λίστα βημάτων που περιγράφει τη λειτουργία του alexo-device-plugin από την άποψη του χρήστη στον Kubernetes:

1. Ο χρήστης υποβάλλει ένα Pod που ζητά alexo.com/shared-gpu.
2. Το alexo-device-plugin εκθέτει στον Kubernetes τους πόρους alexo.com/shared-gpu και φροντίζει να εισάγει στο container του χρήστη την βιβλιοθήκη μας, libunified.so.
3. Η εφαρμογή του χρήστη ξεκινά την εκτέλεση. Η βιβλιοθήκη μας μετατρέπει όλες τις κανονικές εκχωρήσεις μνήμης (cudaMalloc) στις αντίστοιχες με Unified Memory (cudaMallocManaged). Δηλώνει επίσης (register) την εφαρμογή στον anti-thrashing δρομογητή/daemon.
4. Η εφαρμογή μπορεί να χρησιμοποιήσει όλη τη μνήμη της GPU και να εκτελεστεί παράλληλα με άλλες συστεγαζόμενες εφαρμογές (που ανήκουν στον ίδιο ή σε άλλους χρήστες) στην ίδια GPU.
5. Εφ όσον οι ριπές GPU των συστεγαζόμενων εφαρμογών δεν αλληλεπικαλύπτονται ή το άθροισμα των μεγεθών των Working Sets δεν υπερβαίνουν το μέγεθος της φυσικής μνήμης της GPU, δεν υπάρχει ενδεχόμενο thrashing, επομένως η GPU δρομολογεί αυτόματα (black box) τις εκτελέσεις kernel των χρηστών με τρόπο FCFS. Υπενθυμίζουμε ότι εδώ πρόκειται για την δρομολόγηση σε επίπεδο driver, όχι στον Kubernetes.
6. Εάν οι ριπές στην GPU από διαφορετικές εφαργμογές επικαλύπτονται και επίσης το άθροισμα των Working Sets υπερβαίνει τη χωρητικότητα της μνήμης της GPU, υπάρχει ενδεχόμενο thrashing, το οποίο οδηγεί σε τεράστιες καθυστερήσεις.
7. Εάν ο μηχανισμός anti-thrashing είναι ενεργοποιημένος, σειριοποιεί τη χρήση της GPU μεταξύ των εφαρμογών, παρέχοντας αποκλειστική πρόσβαση στην GPU για ένα ρυθμιζόμενο κβάντο χρόνου (TQ) σε κάθε εφαρμογή που την ζητά. Σε αυτήν την περίπτωση, μια εφαρμογή μπορεί να χρειαστεί να περιμένει τη σειρά της. Σε περίπτωση που η ριπή στη GPU μιας διεργασίας ολοκληρωθεί προτού εκπνεύσει το κβάντο χρόνου, η εφαρμογή (μέσω της libunified) αυτόματα «παραδίδει» την GPU πίσω στον scheduler.
8. Όλες οι εφαρμογές των χρηστών εκτελούνται απρόσκοπτα μέχρι την ολοκλήρωσή τους. Η χρήση της GPU μεγιστοποιείται, με κόστος πιθανών μικρών καθυστερήσεων στην ουρά, περιμένωντας τον anti-thrashing scheduler. Σε περίπτωση που η δικαιότητα (fairness) δεν είναι ένα ζήτημα (π.χ. ένας χρήστης υποβάλλει πολλές εργασίες και θέλει να ελαχιστοποιήσει τον συνολικό χρόνο ολοκλήρωσης), ένα μεγάλο TQ για τον scheduler ελαχιστοποιεί τον χρόνο που ξοδεύεται σε context switches καθώς και τον συνολικό χρόνο ολοκλήρωσης και μεγιστοποιεί την αξιοποίηση της GPU.

1.4.1 Τι προσφέρει

- Κάθε χρήστης (Notebook) μπορεί **να χρησιμοποιήσει ολόκληρη τη μνήμη της GPU**.
- **πολλά Notebooks μπορούν να εκτελέσουν ταυτόχρονα στην ίδια GPU χωρίς όριο μνήμης**. Ο μόνος περιοριστικός παράγοντας είναι το μέγεθος της μνήμης της CPU (RAM) και υποκειμενικά (ανά περίπτωση) η αποδεκτή καθυστέρηση των εκτελέσεων κελιών των Notebooks.
- Ο μηχανισμός μας είναι **διαφανής προς την εφαρμογή χρήστη** (χωρίς τροποποίηση του κώδικα χρήστη ή των frameworks)

1.4.2 Στόχοι

Δεδομένου ότι:

- Το σκληρό όριο της μνήμης GPU είναι περιοριστικό για τους χρήστες
- Η υποαξιοποίηση εξακολουθεί να υφίσταται με όλες τις τελευταίας τεχνολογίας λύσεις, ειδικά εάν τα Notebooks ζητούν μεγάλες ποσότητες μνήμης (πολύ λίγα μπορούν να συστεγαστούν σε έναν μηχανισμό τύπου Kubeshare)

Θέλουμε να ελαχιστοποιήσουμε τον χρόνο αδράνειας της GPU, έχοντας ενεργά πολλά Notebooks και πολυπλέκοντας τη χρήση της μνήμης GPU μεταξύ τους, ενώ παράλληλα μεριμνούμε για την αποφυγή καταστάσεων thrashing.

1.4.3 Γιατί δεν έχει γίνει στο παρελθόν

- Δεν υπάρχει δημόσια μελέτη για την συμπεριφορά της Unified Memory [2] όταν συνυπάρχουν 2+ διεργασίες στην ίδια GPU. Εμείς εντοπίσαμε τη συμπεριφορά (πολιτική αντικατάστασης LRU μεταξύ των διαφορετικών διεργασιών). Η εύρεση αυτού μας οδήγησε να οραματιστούμε τα παρακάτω.
- Δεν υπάρχει καμία μελέτη σχετικά με τη διαφανή μετατροπή όλων των εκχωρήσεων μνήμης GPU από συμβατικές σε Unified Memory καθώς και τη σταθερότητα/επίδοση ενός τέτοιου μηχανισμού. Είμαστε οι πρώτοι που μελετούν την συμπεριφορά εφαρμογών υπό αυτές τις συνθήκες.
- Ο περιορισμός του thrashing σε ένα τέτοιο καινοτόμο μοντέλο εκτέλεσης δεν είναι τετριμμένος και απαιτεί ειδική προσοχή.

1.4.4 Περιορισμοί

- Στην περίπτωση που τα περιεχόμενα της μνήμης GPU μιας διεργασίας δεν βρίσκονται στην GPU όταν αυτή εκτελεί μία από τις εντολές της διεργασίας, υπάρχει κάποια καθυστέρηση (μεταφορές PCIe) για την τοποθέτηση (μέσω page faults) εκ νέου των δεδομένων που χρησιμοποιεί στη φυσική μνήμη της GPU.
- ~1% αυξημένος χρόνος εκτέλεσης από τη χρήση της Unified Memory αυτής καθαυτήν.
- Πιθανό head-of-line (HOL) blocking όταν χρησιμοποιείται μεγάλο κβάντο χρόνου (TQ) για τον anti-thrashing scheduler.

[2] Unified εκχωρήσεις μνήμης: Οι GPU kernels μπορούν να προκαλέσουν σφάλματα σελίδας. Οι σελίδες απομακρύνονται προς τη RAM εάν η φυσική μνήμη της GPU είναι πλήρης. Για κανονικές εκχωρήσεις μνήμης, κάθε εκχωρημένο byte υποστηρίζεται από ένα φυσικό byte στη μνήμη GPU.

1.5 Δομή της Αναφοράς

Η υπόλοιπη αναφορά οργανώνεται ως εξής:

- Στο κεφάλαιο 2 παρέχουμε το απαραίτητο θεωρητικό υπόβαθρο.
- Στο κεφάλαιο 3 παρουσιάζουμε τη μελέτη μας της Unified Memory σε σενάρια ταυτόχρονης εκτέλεσης πολλών διεργασιών. Από όσο ξέρουμε, είναι η πρώτη έρευνα επί του αντικειμένου.
- Στο κεφάλαιο 4 αναλύουμε το σχεδιασμό καθώς και την υλοποίηση της προσέγγισής μας.
- Στο κεφάλαιο 5 αξιολογούμε τη δουλειά μας, συγκρίνοντάς τη με τις τρέχουσες προσεγγίσεις.
- Στο κεφάλαιο 6 ανακεφαλαιώνουμε τις συνεισφορές μας και προτείνουμε μελλοντικές κατευθύνσεις δουλειάς.

Κεφάλαιο 2

Υπόβαθρο

Σε αυτό το κεφάλαιο παρέχουμε το απαραίτητο θεωρητικό υπόβαθρο για το υπόλοιπο της αναφοράς.

2.1 Βασικά στοιχεία πάνω στις GPU

Παρουσιάζουμε μια επισκόπηση των θεμελίων πίσω από τον υπολογισμό με GPU, αναλύουμε τα αρχιτεκτονικά χαρακτηριστικά μιας σύγχρονης GPU και εξηγούμε τις βασικές έννοιες στον προγραμματισμό CUDA. Ενώ στο παρόν επικεντρωνόμαστε στις κάρτες γραφικών NVIDIA και το CUDA API, οι περισσότερες από τις πληροφορίες αυτές ισχύουν γενικά για οποιαδήποτε GPU.

2.1.1 Εισαγωγή στους υπολογισμούς με GPU

2.1.1.1 Παραλληλισμός

Στον σημερινό κόσμο, εκτός από τα παραδοσιακά υπολογιστικά προβλήματα (μηχανική ρευστών, πρόγνωση καιρού, βιοχημεία), υπάρχει μια ολοένα αυξανόμενη ανάγκη για επεξεργασία μεγάλων ποσοτήτων δεδομένων με γρήγορο τρόπο. Καθώς οι μαζικά παράλληλοι επεξεργαστές διατίθενται ευρέως στο εμπόριο, το πεδίο του παραλληλισμού ευδοκιμεί. Κάθε οντότητα στο ψηφιακό τοπίο, είτε πρόκειται για μια μεγάλη εταιρεία είτε για ένα μικρό ερευνητικό εργαστήριο, χρησιμοποιεί παράλληλους υπολογιστές για να ικανοποιήσει τις ανάγκες της. Ως εκ τούτου, ο παραλληλισμός έχει γίνει η κινητήρια δύναμη της αρχιτεκτονικής και του σχεδιασμού συστημάτων.

Υπάρχουν δύο θεμελιώδεις τύποι παραλληλισμού στις εφαρμογές:

- **Παραλληλισμός εργασιών: (Task parallelism)** Υπάρχουν πολλές εργασίες ή λειτουργίες που μπορούν να εκτελεστούν ανεξάρτητα και σε μεγάλο βαθμό παράλληλα. Ο παραλληλισμός εργασιών χειρίζεται τη διανομή αυτών των λειτουργιών σε πολλούς εργάτες (πυρήνες).

- **Παραλληλισμός δεδομένων: (Data parallelism)** Υπάρχουν πολλά δεδομένα τα οποία μπορούμε να επεξεργαστούμε παράλληλα και δίχως εξάρτηση. Στον παραλληλισμό δεδομένων διαχωρίζουμε τα δεδόμενα μεταξύ των νημάτων, με κάθε νήμα να δουλεύει πάνω σε ένα υποσύνολό τους.

Ο προγραμματισμός GPU είναι ιδιαίτερα κατάλληλος για την αντιμετώπιση προβλημάτων (πυκνή γραμμική άλγεβρα, FFT ...) που μπορούν να εκφραστούν ως *data-parallel υπολογισμοί.*

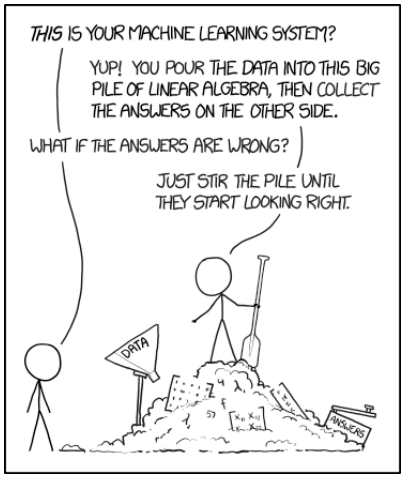

Figure 2.1: *Η μηχανική μάθηση σε μια εικόνα [1]*

Υπάρχει μια πληθώρα εφαρμογών που επεξεργάζονται μεγάλα σύνολα δεδομένων και μπορούν να χρησιμοποιήσουν ένα παράλληλο μοντέλο για να επιταχύνουν τους υπολογισμούς τους.

2.1.1.2 GPU vs CPU

Ιστορικά, οι GPU αναπτύχθηκαν για να χρησιμοποιηθούν ως επιταχυντές γραφικών. Ξεκινώντας από τα μέσα της δεκαετίας του 2000, οι προγραμματιστές άρχισαν να χρησιμοποιούν GPU ως επεξεργαστές γενικής χρήσης. Η χρήση μιας GPU για εκτέλεση υπολογισμών σε εφαρμογές που παραδοσιακά εκτελούνται από CPU ονομάζεται **Υπολογισμός γενικού σκοπού σε Κάρτες Γραφικών (GPGPU)** [19]. Εκείνη την εποχή όμως, οι κατασκευαστές GPU δεν εξέθεταν ένα προγραμματιστικό API για να τους βοηθήσουν σε αυτήν την προσπάθεια. Αρχικά, οι προγραμματιστές «υπερφόρτωναν» τις λειτουργίες των API γραφικών (π.χ. DirectX για την εκτέλεση υπολογισμών γραμμικής άλγεβρας, όπως πολλαπλασιασμούς πινάκων. Η χρήση μιας GPU για υπολογισμούς ήταν προγραμματιστικά περίπλοκη, επιρρεπής σε σφάλματα και γενικά δεν ήταν μια ευχάριστη εμπειρία. Η εισαγωγή του CUDA από την NVIDIA (2007) έδωσε τελικά στους προγραμματιστές τα απαραίτητα εργαλεία για την αξιοποίηση της υπολογιστικής δύναμης της GPU και δημιούργησε ένα νέο πεδίο στον παράλληλο υπολογισμό, GPGPU. Οι GPU είναι πλέον γενικής χρήσης, ισχυροί, πλήρως προγραμματιζόμενοι task- και data- parallel επεξεργαστές. Είναι ιδιαίτερα κατάλληλες για την επίλυση μαζικά παράλληλων υπολογιστικών προβλημάτων.

Η GPU παρέχει πολύ υψηλότερο instruction throughput και εύρος ζώνης μνήμης (memory bandwidth) από την CPU σε μια *παρόμοια τιμή καθώς και επίπεδο ενεργειακής κατανάλωσης*. Ως εκ τούτου, ορισμένες εφαρμογές αξιοποιούν αυτές τις υψηλότερες δυνατότητες για να τρέχουν γρηγορότερα στην GPU από ό,τι στην CPU.

Αυτή η διαφορά στις δυνατότητες μεταξύ της GPU και της CPU υπάρχει επειδή **έχουν σχεδιαστεί με διαφορετικούς στόχους κατά νου**. Ενώ η CPU έχει σχεδιαστεί για να

υπερέχει στην εκτέλεση μιας ακολουθίας λειτουργιών (ένα νήμα) όσο το δυνατόν γρηγορότερα και μπορεί να εκτελέσει παράλληλα μερικές δεκάδες από αυτά τα νήματα, η GPU έχει σχεδιαστεί έτσι ώστε να εκτελεί παράλληλα **χιλιάδες από αυτά (απόσβεση της πιο αργής απόδοσης του ενός νήματος και επίτευξη μεγαλύτερου συνολικού throughput)**.

Η αφιέρωση περισσότερων τρανζίστορ στην επεξεργασία δεδομένων, π.χ. υπολογισμούς κινητής υποδιαστολής, είναι ευεργετική για εξαιρετικά παράλληλους υπολογισμούς. Η GPU μπορεί να αποκρύψει τους χρόνους πρόσβασης στη μνήμη (latency) με υπολογισμούς καθώς διαθέτει επιπλέον υλικό, αντί να βασίζεται σε μεγάλες κρυφές μνήμες δεδομένων (caches) και πολύπλοκο έλεγχο ροής (πρόβλεψη διακλαδώσεων, out-of-order εκτέλεση) για να αποφευχθούν οι μεγάλοι χρόνοι πρόσβασης στην κύρια μνήμη, τεχνικές οι οποίες είναι επίσης ακριβές όσον αφορά τα τρανζίστορ.

Σε γενικές γραμμές, μια εφαρμογή έχει ένα **μείγμα παραλλήλων και σειριακών/ακολουθιακών τμημάτων**, επομένως τα σύγχρονα συστήματα έχουν σχεδιαστεί με συνδυασμό από GPU και CPU, προκειμένου να *μεγιστοποιήσουν τη συνολική απόδοση.*

Μια σύγκριση δίπλα-δίπλα του πυριτίου που κατανέμεται σε κάθε συστατικό δείχνει τις παραπάνω διαφορές.

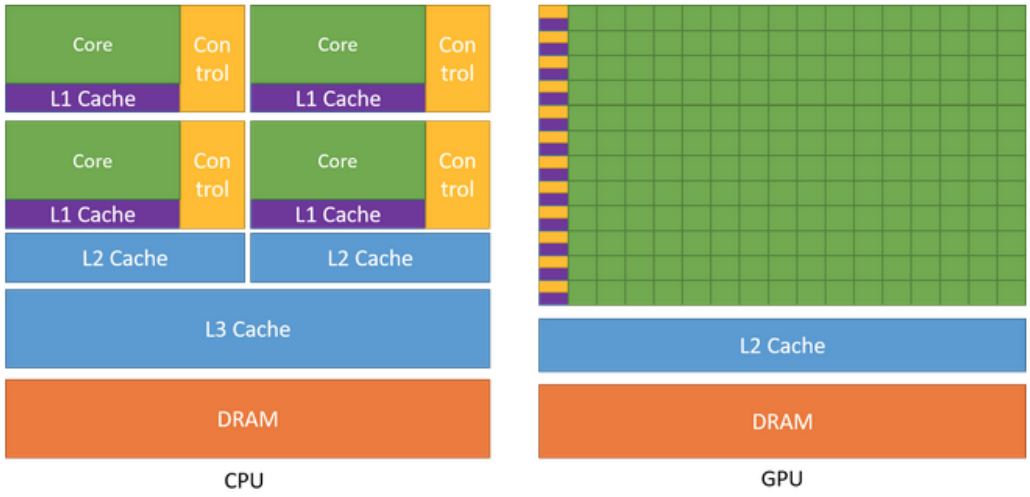

Figure 2.2: *Διαφορές ανάθεσης τρανζίστορ μεταξύ CPU και GPU*

2.1.2 Βασικά στοιχεία πάνω στο Hardware της GPU

Note

Επί του παρόντος, οι προμηθευτές (ειδικά η NVIDIA) κρύβουν τις λεπτομέρειες των αρχιτεκτονικών των GPU για τους δικούς τους λόγους. Θα προσεγγίσουμε το υλικό της GPU από την οπτική γωνία ενός προγραμματιστή εφαρμογών - και δεδομένης της δημοσίως διαθέσιμης πληροφορίας - και όχι από έναν αρχιτέκτονα / προγραμματιστή GPU (reverse-engineered προσέγγιση).

2.1.2.1 Αρχιτεκτονική

Η GPU δεν είναι επί του παρόντος αυτόνομη πλατφόρμα, αλλά συν-επεξεργαστής (co-processor) σε μια CPU. Επομένως, οι GPU πρέπει να λειτουργούν σε συνδυασμό με έναν ηοστ που βασίζεται σε CPU και στον οποίο συνδέονται μέσω διαύλου PCI-Express. Για αυτό, στη βιβλιογραφία, η CPU ονομάζεται *host* και η GPU ονομάζεται *device*.

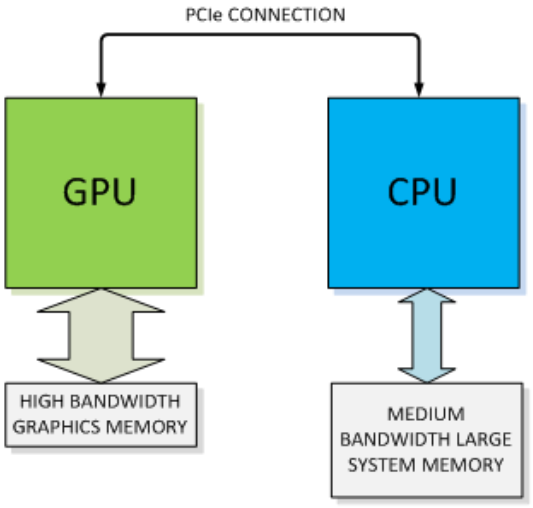

Figure 2.3: *Σύνδεση μεταξύ GPU και CPU*

Η απλοποιημένη άποψη της CUDA για μια GPU περιλαμβάνει τα εξής:

- Α **διεπαφή host** που συνδέει την GPU με το δίαυλο PCI Express. Διαβάζει εντολές GPU (αντίγραφές μνήμης, εκκίνηση kernels) και τις αποστέλλει στις κατάλληλες μονάδες.
- 1-2 **μηχανές αντιγραφής**: για να επικαλύπτονται οι μεταφορές δεδομένων από και προς την GPU με την εκτέλεση υπολογισμών μέσω kernels.
- Α **διεπαφή DRAM** η οποία συνδέει την GPU με την on-chip μνήμη της
- Έναν αριθμό TPC ή GPC (Texture Processing Units ή Graphics Processing Units) καθένα από τα οποία περιέχει κρυφές μνήμες και έναν αριθμό **streaming multiprocessors (SMs)**. Το πλήθος και η διαρρύθμισή τους διαφέρουν ανάλογα με τη γενιά GPU.

Το παρακάτω σχήμα δείχνει την αρχιτεκτονική μιας κάρτας γραφικών NVIDIA γενιάς Fermi:

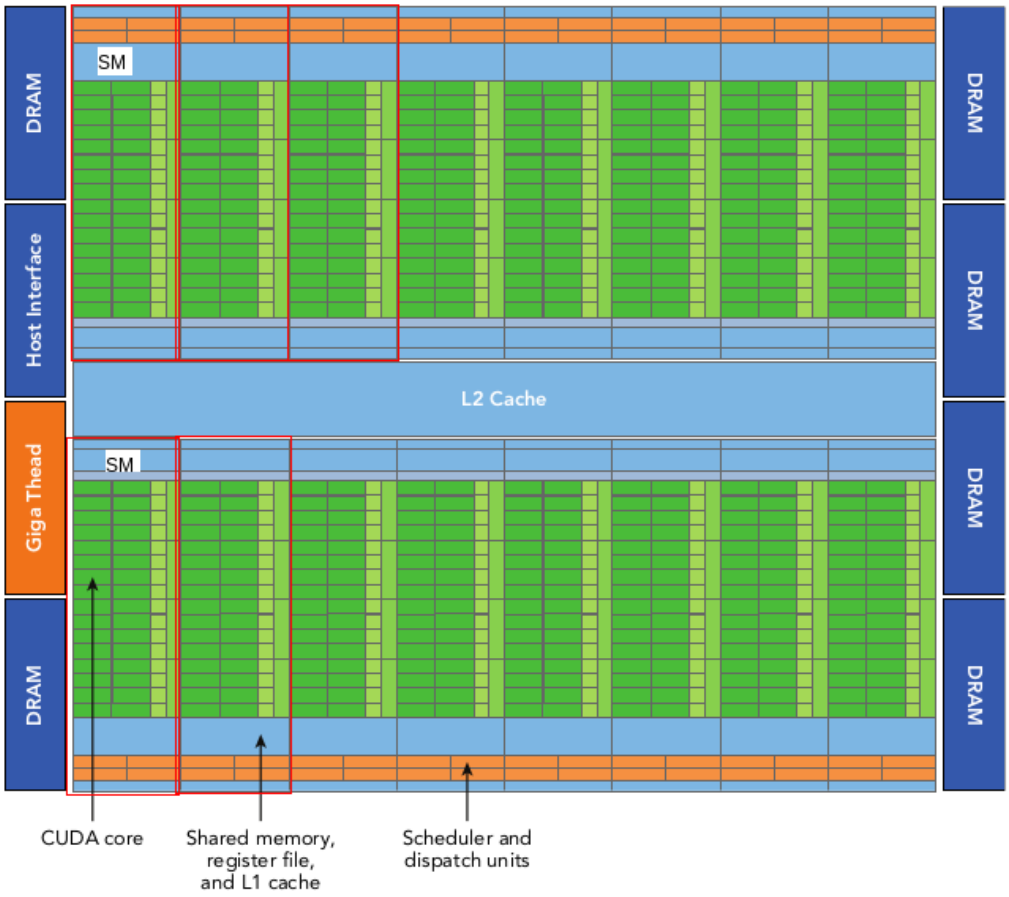

Figure 2.4: *Αρχιτεκτονική μιας Fermi GPU*

2.1.2.2 Ο Streaming Multiprocessor

Η αρχιτεκτονική GPU βασίζεται σε μια συστοιχία από Streaming Multiprocessors (SMs). Ο παραλληλισμός υλικού της GPU επιτυγχάνεται μέσω της αναπαραγωγής αυτού του αρχιτεκτονικού δομικού στοιχείου.

Τα βασικά στοιχεία ενός SM είναι:

- CUDA Cores [ALU] [στην GV100: 64 x FP32, 32 x FP64, 64 x INT32, 8 μεικτής-ακρίβειας Tensor Cores]
- Shared Memory/L1 Cache
- Register File (Αρχείο Καταχωρητών) [τάξη μεγέθους 64K 32-bit καταχωρητές]
- Load/Store Units
- Special Function Units [log/exp, sin/cos, and rcp/rsqrt]
- Warp Schedulers [γρήγορη εναλλαγη μεταξύ contexts νημάτων και έκδοση εντολών σε warps που είναι έτοιμα προς εκτέλεση]

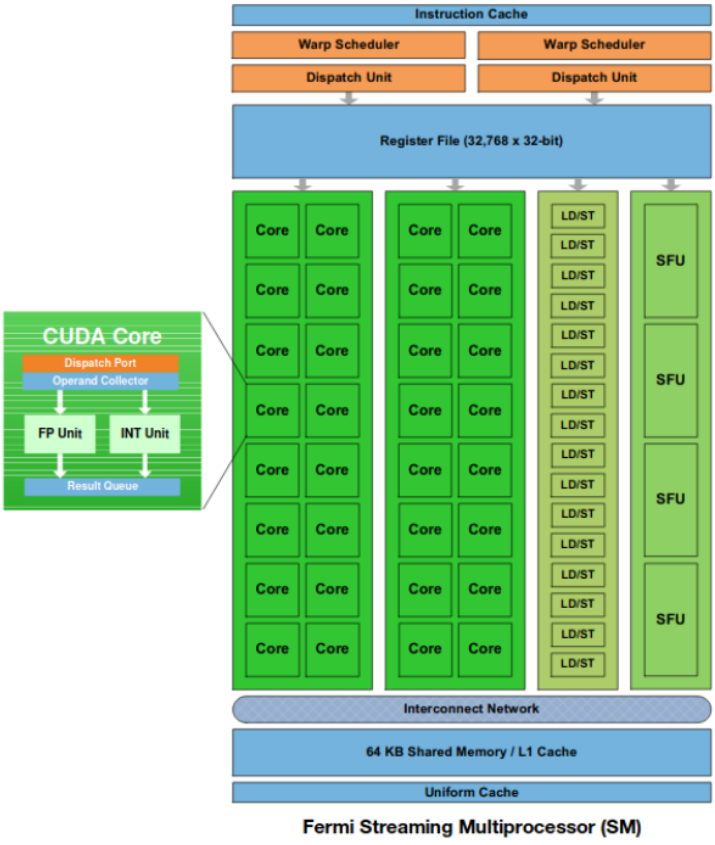

Figure 2.5: *Ένας Fermi Streaming Multiprocessor*

Κάθε SM σε μια GPU έχει σχεδιαστεί για να υποστηρίζει ¨ταυτόχρονη εκτέλεση¨ εκατοντάδων νημάτων (έως 2048 σε αρχιτεκτονικές Volta [20]) και υπάρχουν πολλά SM ανά GPU, επομένως είναι πιθανό να εκτελούνται δεκάδες χιλιάδες νήματα ταυτόχρονα σε μία μόνο GPU. Η CUDA χρησιμοποιεί αρχιτεκτονική **Single Instruction Multiple Thread** (SIMT) για την διαχείριση και εκτέλεση των νημάτων σε ομάδες των 32 που ονομάζονται warps. Όλα τα νήματα σε ένα warp εκτελούν τις ίδιες οδηγίες σε κάθε κύκλο. Κάθε νήμα έχει τη δική του διεύθυνση μετρητή εντολών (instruction address counter) και register file και εκτελεί την τρέχουσα οδηγία στα δικά του δεδομένα.

Η GPU έχει σχεδιαστεί για να έχει αρκετή κατάσταση ώστε να γεμίσει με εκτέλεση άλλων εντολών το latency μνήμης εκατοντάδων κύκλων ρολογιού που μπορεί να χρειαστούν για να φτάσουν τα δεδομένα από τη μνήμη της συσκευής μετά την εκτέλεση μιας εντολής ανάγνωσης και αυτός είναι ο θεμελιώδης τρόπος με τον οποίο οι GPU επιτυγχάνουν τόσο μεγάλη απόδοση.

2.1.2.3 Το μοντέλο μνήμης της CUDA

Για τους προγραμματιστές, υπάρχουν γενικά δύο ταξινομήσεις της μνήμης:

- **Προγραμματιζόμενη (Programmable)**: Ελέγχουμε ρητά ποια δεδομένα τοποθετούνται σε προγραμματιζόμενη μνήμη.
- **Μη προγραμματιζόμενη (Non-programmable)**: Δεν έχουμε έλεγχο επί της τοποθέτησης δεδομένων και βασίζουμε αυτή σε αυτόματες τεχνικές για να επιτύχουμε καλή απόδοση. Οι κρυφές μνήμες είναι ένας εξέχων τύπος μη προγραμματιζόμενης μνήμης.

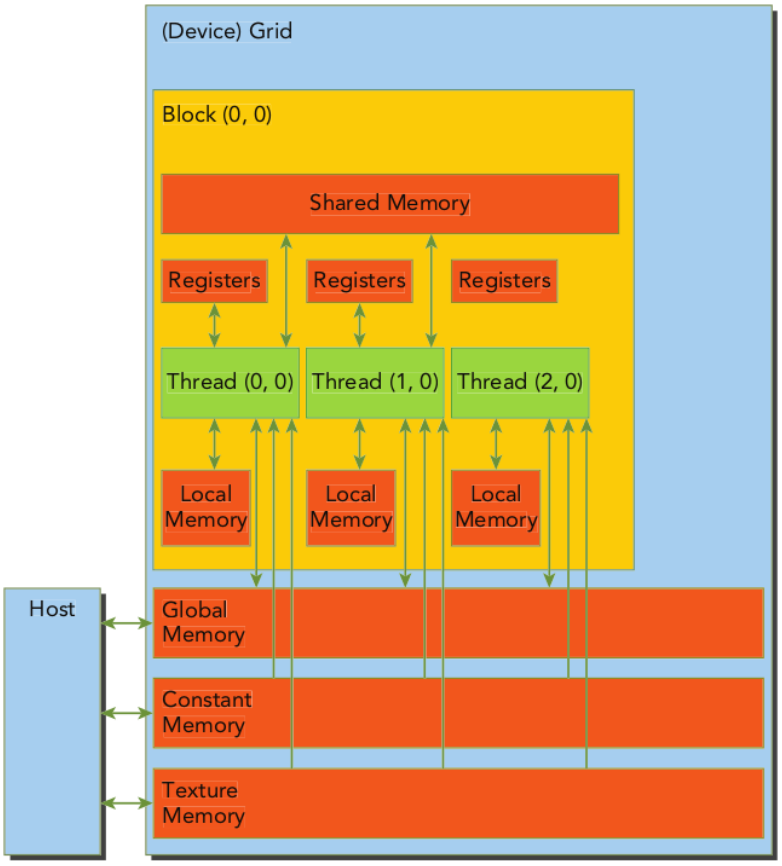

Figure 2.6: *Ιεραρχία Μνήμης στην CUDA*

2.1.2.4 Τι είναι ένας Kernel

Ένα πρόγραμμα CUDA περιλαμβάνει μια μίξη από τους εξής δύο τύπους κώδικα:

- Ο **host κώδικας** εκτελείται στην **CPU**.
- Ο **device κώδικας** εκτελείται στην **GPU**.

Ο μεταγλωττιστής nvcc διαχωρίζει τον host από τον device κώδικα για τη διαδικασία μεταγλώττισης.

Εκφράζουμε έναν kernel ώς ένα ακολουθιακό πρόγραμμα γραμμένο σε γλώσσα C με κάποιες επεκτάσεις της CUDA.

Η τυπική υπολογιστική ροή ενός προγράμματος CUDA ακολουθεί την εξής συνταγή:

1. Εκχώρηση μνήμης σε host και GPU
2. Αρχικοποίηση δεδομένων στον host.
3. Μεταφορά δεδομένων από host σε device.
4. Εκτέλεση ενός ή περισσότερων kernel.
5. Μεταφορά των αποτελεσμάτων από device σε host.

Ο πιο συνηθισμένος τρόπος κλήσης ενός kernel από τον κώδικα είναι με την παρακάτω σύνταξη:

```
Kernel_Name <<< GridSize, BlockSize,  SMEMSize, Stream >>> (arguments,....)
```

Ανακεφαλαιώνοντας:

- Τα νήματα στην GPU ομαδοποιούνται σε (Thread) blocks
- Τα thread blocks ομαδοποιούνται σε ένα grid
- Ένας υπολογιστικός kernel εκτελείται ως ένα grid από blocks από threads.

Thread Blocks:

- Τα TBs ανατίθενται σε SMs από τον δρομολογητή thread-block της GPU βασισμένα στις ανάγκες τους σε πόρους και την χωρητικότητα των SM (για μια βαθύτερη ανάλυση των δρομολογητών TB αναφερθείτε στην εξαιρετική μελέτη του Sreepathi Pai [21])
- Κάθε TB εκτελείται σε ένα ακριβώς SM και δεν μεταναστεύει
- Πολλά διαφορετικά TBs μπορούν να συνυπάρχουν σε ένα SM ανάλογα με τις ανάγκες τους σε μνήμη καθώς και τη χωρητικότητα του SM

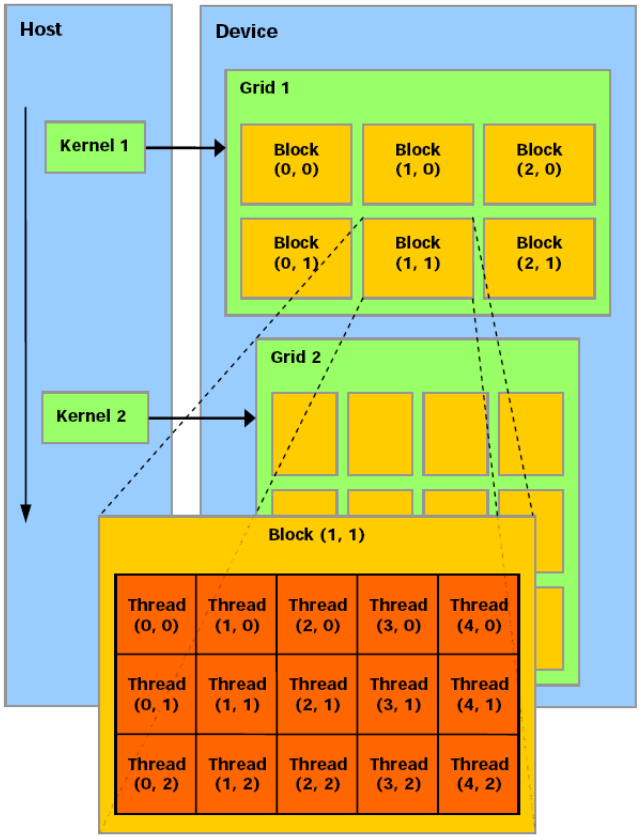

Figure 2.7: *Ιεραρχία νημάτων στην CUDA*

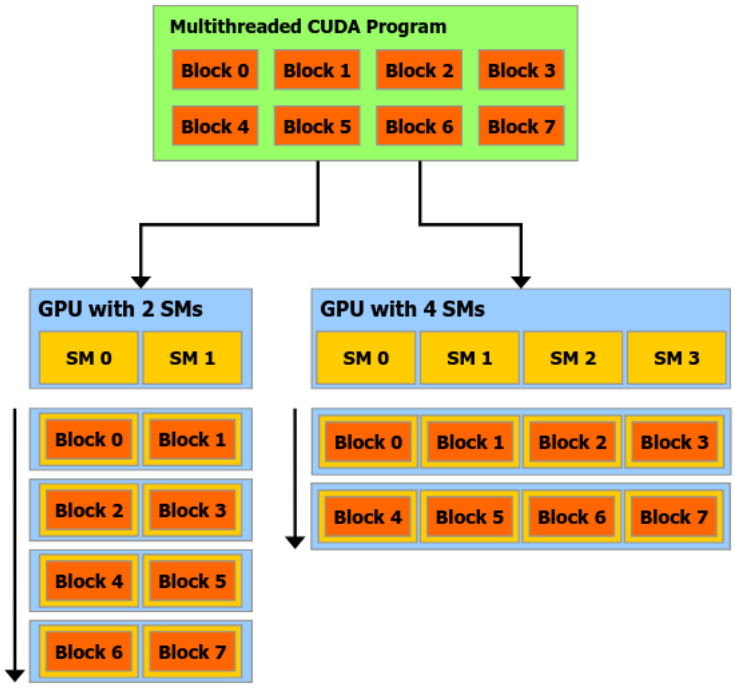

Figure 2.8: *Κλιμακωσιμότητα SM*

2.1.3 Η προγραμματιστική διεπαφή του CUDA

Τώρα θα αναλύσουμε την προγραμματιστική διεπαφή του CUDA από την οπτική ενός προγραμματιστή

2.1.3.1 Προοίμιο: Accelerator Silos

Οι συγγραφείς του AvA (Automatic Virtualization of Accelerators) [2] κάνουν την εξής πολύ εύστοχη παρατήρηση (βέβαια αυτή γίνεται υπό το πρίσμα της εικονικοποίησης, αλλά ισχύει γενικότερα):

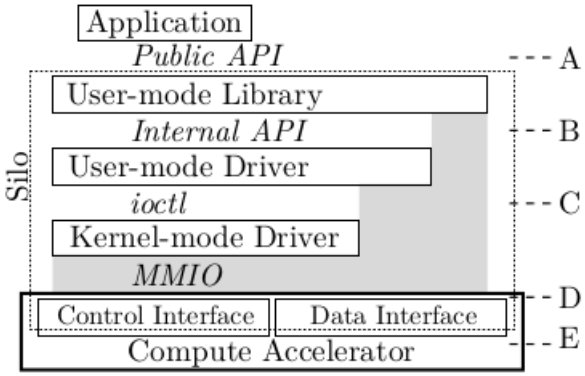

Figure 2.9: *Accelerator Silos [2]*

> *Accelerator stacks comprise layered components that include a user-mode library to support an API framework and a driver to manage the device. Vendors are incentivized to use proprietary interfaces and protocols between layers to preserve forward compatibility, and to use kernel-bypass communication techniques to eliminate OS overheads. However, interposing opaque, frequently-changing interfaces communicating with memory mapped command rings is impractical because it requires inefficient techniques and yields solutions that sacrifice compatibility. Consequently, accelerator stacks are effectively silos, whose intermediate layers cannot be practically separated.*

2.1.3.2 CUDA Runtime και Driver API

Το CUDA προσφέρει δύο προγραμματιστικά API:

1. το **Runtime** API και
2. το **Driver** API

Το Runtime API προσφέρει συναρτήσεις σε C και C++ οι οποίες εκτελούνται στον host και αφορούν την ανάθεση και αποδέσμευση μνήμης, την μεταφορά μνήμης καθώς και την εκτέλεση υπολογιστικών kernels. Είναι μια διεπαφή υψηλού επιπέδου και παρέχει ένα επίπεδο αφαίρεσης προς τον χρήστη, απλοποιώντας την διαχείριση της GPU.

Το Driver API, πάνω στο οποίο είναι βασισμένο το Runtime API για τις περισσότερες από τις λειτουργίες του είναι μια χαμηλοτέρου επιπέδου διεπαφή που προσφέρει λεπτότερο έλεγχο της συσκευής στον χρήστη αλλά είναι προγραμματιστικά δυσκολότερο στη χρήση.

2.1.3.3 Ένα παράδειγμα εφαρμογής CUDA

Εδώ παρουσιάζουμε ένα πρόγραμμα που υλοποιεί **SAXPY** [Single-precision A times X Plus Y, (A*X + Y)] στην GPU χρησιμοποιώντας το Runtime API:

Listing 2.1: SAXPY using the CUDA Runtime API

```
1  #include <stdio.h>
2
3  __global__              //device Function
4  void saxpy(int n, float a, float *x, float *y) {
5  /* find unique thread id - determine data to operate on */
6    int i = blockIdx.x*blockDim.x + threadIdx.x;
7  /* make sure we don't run out of bounds */
8    if (i < n) y[i] = a*x[i] + y[i];
9  }
10
11 int main(void) {
12   int N = 1<<20;
13   float *x, *y, *d_x, *d_y;
14   x = (float*)malloc(N*sizeof(float));
15   y = (float*)malloc(N*sizeof(float));
16
17   cudaMalloc(&d_x, N*sizeof(float));
18   cudaMalloc(&d_y, N*sizeof(float));
19
20   for (int i = 0; i < N; i++) {
21     x[i] = 1.0f;
22     y[i] = 2.0f;
23   }
24
25   cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
26   cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);
27
28   // Perform SAXPY on 1M elements
29   /*
30    * Make sure the # of threads launched are >= N [N+255/256 TB of 256 threads each]
31   */
32   saxpy<<<(N+255)/256, 256>>>(N, 2.0f, d_x, d_y);
33
34   cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);
35
36   cudaFree(d_x);
37   cudaFree(d_y);
```

```
38    free(x);
39    free(y);
40  }
```

2.1.3.4 Contexts

Μπορούμε να σκεφτούμε ένα CUDA context ως την «προβολή» μιας CPU διεργασίας στην GPU. Κάθε διεργασία host αλληλεπιδρά με το CUDA μέσα σε ένα context. Όταν χρησιμοποιούμε το Runtime API, το CUDA χειρίζεται αυτόματα τη δημιουργία και τη διαχείριση του context. Ωστόσο, κατά τη χρήση του Driver API (το οποίο είναι σε χαμηλότερο επίπεδο), πρέπει να δημιουργήσουμε και να χειριστούμε ρητά το GPU context προκειμένου να υποβάλουμε δουλειά στην GPU. Πιο απλά, μια εφαρμογή που θέλει να χρησιμοποιήσει μια GPU δημιουργεί ένα context και στη συνέχεια υποβάλλει εντολές (εκχωρήσεις μνήμης συσκευής, αντιγραφές μνήμης, εκκίνηση kernel) σε αυτό το context.

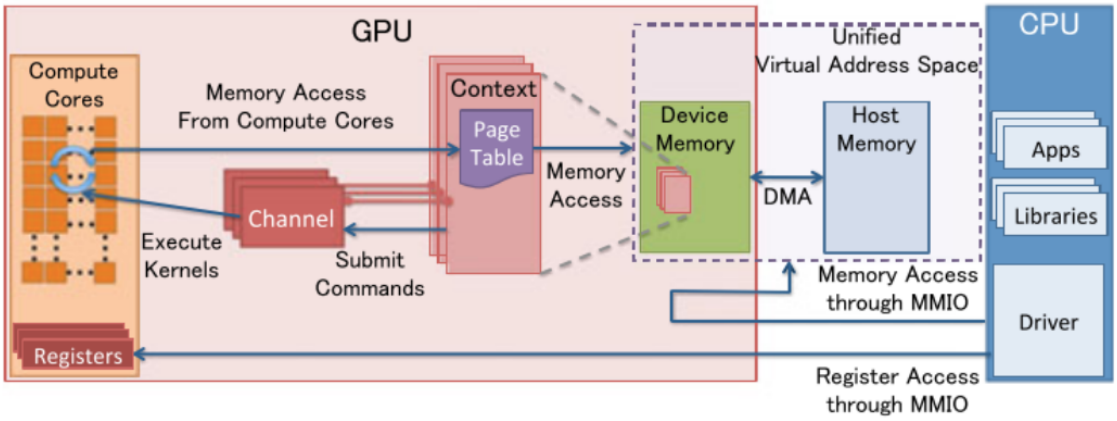

Figure 2.10: *Μοντέλο διαχείρισης πόρων στην GPU*

2.1.3.5 Πολλαπλά context (2 ή περισσότερες εφαρμογές)

Ενώ πολλά context (και οι σχετικοί πόροι τους, όπως οι εκχωρήσεις μνήμης) μπορεί να υπάρχουν ταυτόχρονα σε μια δεδομένη GPU, μόνο **ένα** από αυτά τα context μπορεί να είναι ενεργό οποιαδήποτε δεδομένη στιγμή. Ο driver της NVIDIA χειρίζεται την εναλλαγή των context με μή-γνωστοποιημένο τρόπο.

2.1.3.6 Μοντέλο μνήμης του CUDA API

Η κύρια πληροφορία που εμπεριέχεται σε ένα CUDA context είναι ο πίνακας σελίδων της διεργασίας. Ένα context δεν μπορεί να προσπελάσει μνήμη η οποία ανήκει σε κάποιο άλλο, όπως αντίστοιχα στη CPU μια διεργασία δεν μπορεί να προσπελάσει τη μνήμη μιας άλλης.

Unified Memory

Για τις κανονικές εκχωρήσεις μνήμης (cudaMalloc) ισχύει η ιδιότητα πως κάθε byte το οποίο εκχωρείται σε μια διεργασία υποστηρίζεται από ένα φυσικό byte στη μνήμη της GPU. Μια εξαίρεση σε αυτόν τον κανόνα αποτελούν οι Unified εκχωρήσεις μνήμης. Η Unified

Memory είναι μια νεότερη τεχνολογία της CUDA η οποία υποστηρίζεται από τις σύγχρονες κάρτες γραφικών και επιτρέπει στην GPU να χειρίζονται σφάλματα σελίδας (page faults) καθώς και να χρησιμοποιούν τη RAM του συστήματος σαν swap χώρο. Μας επιτρέπει με διαφανή τρόπο να επεκτείνουμε τη μνήμη της GPU χρησιμοποιώντας την RAM σαν επιπλέον χώρο, πληρώνοντας βέβαια το χρονικό κόστος των σφαλμάτων σελίδας.

2.2 Kubernetes

Σύνοψη του Kubernetes

Ο Kubernetes είναι ένα σύστημα λογισμικού που εκτελεί containers σε κόμβους μιας υπολογιστικής συστοιχίας. Ο Kubernetes βασίζεται στη δήλωση προθέσεων, αντί να εκτελεί προστακτικές εντολές. Αποτελείται από:

1. ένα μόνιμο αποθηκευτικό χώρο (persistent storage - etcd) στον οποίο αποθηκεύει αντικείμενα
2. Ένα σύνολο από Controllers, οι οποίοι δρουν επί των αντικειμένων, προσπαθώντας να ευθυγραμμίσουν την πραγματική κατάσταση του συστήματος με την επιθυμητή, την οποία έχουν δηλώσει οι χρήστες

Όλη η πληροφορία σχετικά με την κατάσταση αποθηκεύεται στον etcd. Με ελάχιστες εξαιρέσεις, όλα τα αντικείμενα αποτελούνται από ένα πεδίο Spec που περιγράφει την επιθυμητή κατάσταση και ένα πεδίο Status το οποίο περιγράφει την πραγματική κατάσταση.

2.2.1 Αρχιτεκτονική

Η πλατφόρμα Kubernetes (γνωστή και ως K8s) ακολουθεί το αρχιτεκτονικό μοντέλο Master-Slave. Αποτελείται από τον Kubernetes Master και πολλαπλά Kubernetes Nodes. Ο Master (control plane) διατηρεί τη βάση δεδομένων etcd καθώς και τον API server. Κάθε κόμβος εκτελεί ένα μικρό πρόγραμμα, το kubelet, το οποίο ρωτά συνεχώς τον API Server και εκτελεί Pods τα οποία έχουν δρομολογηθεί σε αυτό το Node. Επίσης κάθε κόμβος εκτελεί το kube-proxy, το οποίο διαχειρίζεται τη δικτύωση. Τα Pods αποτελούν το θεμελιώδη λίθο του K8s καθώς είναι η μικρότερη δρομολογίσιμη οντότητα. Κάθε Pod εμπεριέχει ένα ή περισσότερα containers. Το μοντέλο δικτύωσης του Kubernetes διαβεβαιώνει πως κάθε Pod βλέπει τον εαυτό του με την ίδια διεύθυνση IP όπως όλα τα υπόλοιπα Pods βλέπουν εκείνο. Μπορεί να συνδεθεί με άλλα Pods καθώς και με το Node το οποίο το φιλοξενεί χωρίς ανάγκη για NAT. Η Εικόνα 2.11 οπτικοποιεί τα προαναφερθέντα.

Ας αναλύσουμε τα επιμέρους συστατικά στοιχεία:

Master:

- **etcd**: Το κεντρικό σημείο αποθήκευσης ενός συστήματος K8s. Ο etcd [22] είναι ένα ισχυρά συνεκτικό, κατανεμημένο key-value store και παρέχει έναν αξιόπιστο τρόπο αποθήκευσης δεδομένων από μια υπολογιστική συστοιχία. Κάθε αντικείμενο του Kubernetes αποθηκεύεται στον etcd.

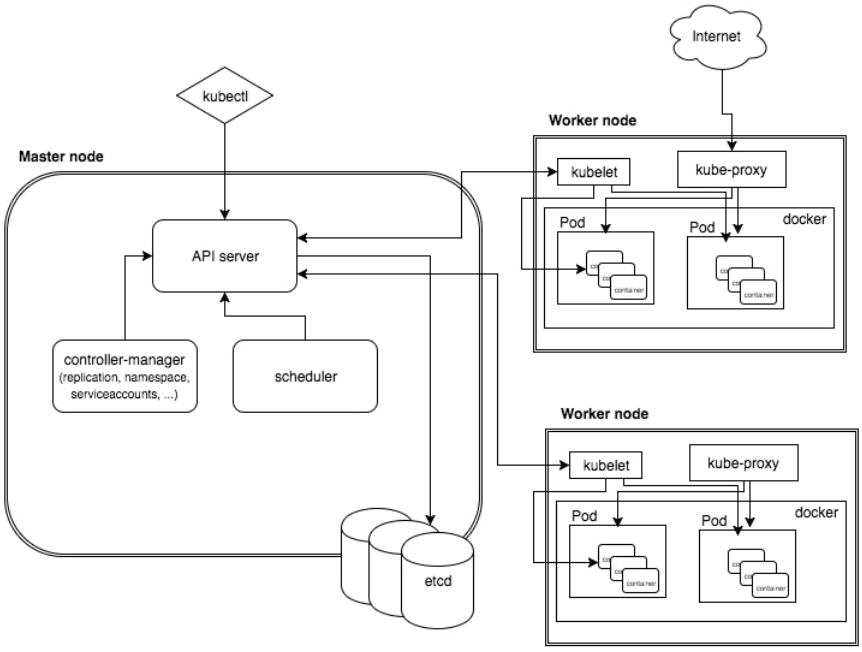

Figure 2.11: *Μια απλοποιημένη όψη της αρχιτεκτονικής του Kubernetes*

- **API Server**: Ο Kubernetes API server χειρίζεται αιτήματα REST σχετικά με τη διαχείριση αντικειμένων (Objects) και είναι η αποκλειστική front-end διεπαφή για τη συστοιχία. Όλες οι τροποποιήσεις δεδομένων στον etcd περνούν μέσα από τον API Server.
- **controller-manager**: Διαχειρίζεται οντότητες (διεργασίες) ελεγκτή (Controller), οι οποίες παρακολουθούν την κατάσταση των αντικειμένων K8s και προσπαθούν να ευθυγραμμίσουν την πραγματική κατάσταση με την επιθυμητή κατάσταση, την οποία δηλώνει ο χρήστης ή άλλοι ελεγκτές.
- **Scheduler**: Αποφασίζει σε ποιον κόμβο θα τρέχει ένα Pod, με βάση τις απαιτήσεις πόρων και την υπόλοιπη χωρητικότητα του κόμβου.

Node:

- **kubelet**: Το kubelet είναι υπεύθυνο για την εκτέλεση container στον κόμβο. Παρακολουθεί τον API Server για να δει εάν πρέπει να εκτελέσει / σταματήσει Pods που έχουν δεθεί σε αυτόν τον κόμβο. Πραγματοποιεί επίσης τακτικά ελέγχους υγείας στα containers που ανήκουν στα Pods του Node, ξεκινώντας ξανά αυτά που είναι μη-υγιή. Τέλος, είναι υπεύθυνο για την καταχώριση του Node και των διαθέσιμων πόρων του στον API Server και τη διαχείριση του Node Object καθ όλη τη διάρκεια ζωής του.
- **kube-proxy**: Επιτρέπει στα container που ζουν στον κόμβο να επικοινωνήσουν με τους τελικούς χρήστες και τον έξω κόσμο. Το kube-proxy δημιουργεί κανόνες iptables που ανακατευθύνουν την IP κίνηση που αποστέλλεται σε μια εξωστρεφή υπηρεσία στο αντίστοιχο Pod/container.

2.2.2 Αντικείμενα (Objects)

Τα αντικείμενα είναι οι μόνιμες οντότητες σε μια συστοιχία K8s και αποθηκεύονται στον etcd. Ο Kubernetes λειτουργεί βάσει της δήλωσης προθέσεων, αντί να εκτελεί επιτακτικές εντολές. Ως εκ τούτου, όταν ένας χρήστης δημιουργεί ένα αντικείμενο, δηλώνει την επιθυμητή κατάσταση στην οποία θέλει να είναι το σύστημα. Τα δύο σημαντικά πεδία σε ένα αντικείμενο είναι το Spec και η Κατάσταση. Το Spec περιγράφει την επιθυμητή κατάσταση που ο χρήστης θέλει να έχει το αντικείμενο. Το πεδίο Status περιγράφει το την τρέχουσα κατάσταση. Το control plane (το οποίο περιλαμβάνει τους Controllers) επιδιώκει συνεχώς και ενεργά την ευθυγράμμιση της τρέχουσας κατάστασης με την επιθυμητή κατάσταση. Οι χρήστες συνήθως ορίζουν αντικείμενα σε YAML, αλλά η διεπαφή γραμμής εντολών (kubectl) το μετατρέπει σε JSON πριν από τη δημιουργία και την αποστολή αιτήματος στον API Server.

Για παράδειγμα, ένα Deployment είναι ένα αντικείμενο που μπορεί να χρησιμοποιηθεί για την αναπαράσταση μιας εφαρμογής που εκτελείται στη συστοιχία. Κατά τη δημιουργία ενός Deployment, μπορούμε να καθορίσουμε στο Spec ότι θέλουμε 3 αντίγραφα (Pods) της εφαρμογής να εκτελούνται. Ο Deployment Controller παρακολουθεί την δημιουργία αντικειμένων Deployment στον API Server και, όταν παρατηρήσει το νέο Deployment, δημιουργεί 3 Pod Objects. Εάν αποτύχει κάποιο από αυτά τα Pods (μια αλλαγή που θα αντικατοπτρίζεται στο πεδίο Status), ο Deployment Controller θα προσπαθήσει να ευθυγραμμίσει τη διαφορά μεταξύ της επιθυμητής-πραγματικής κατάστασης ξεκινώντας ένα νέο Pod.
Παρουσιάζουμε ένα παράδειγμα ενός nginx Deployment, μιας ιδιαίτερα δημοφιλούς εφαρμογής:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # tells deployment to run 2 pods matching the template
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

Στη συνέχεια παρουσιάζουμε και την δήλωση ενός nginx Pod σε YAML:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-example
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

2.3 GPUs στον Kubernetes και device plugin

Ο Kubernetes υποστηρίζει device plugin [23] προκειμένου να επιτρέψει σε (containers που εκτελούνται εντός) Pods να χρησιμοποιήσουν ειδικές συσκευές hardware όπως GPUs. Για να λάβει μια GPU, ένας χρήστης πρέπει να ζητήσει τον πόρο nvidia.com/gpu στο κομμάτι limits του request του. Υπάρχουν όμως κάποιοι περιορισμοί στον τρέχον σχεδιασμό των device plugin:

- Τα Pods δεν μπορούν να μοιραστούν μια κοινή GPU. Δεν υπάρχει overcommiting των GPUs.
- Κάθε container μπορεί να ζητήσει μόνο έναν ακέραιο αριθμό από GPUs. Δεν υπάρχει δυνατότητα να ζητήσει κλάσμα αυτής.

2.3.1 Αρχές λειτουργίας των device plugin

Στη ρίζα τους τα device plugin είναι απλοί gRPC [24] servers οι οποίοι εκτελούνται ως container μέσω ενός Pod ή ως bare metal διεργασίες. Οι αρμοδιότητες ενός device plugin είναι να:

- **ενημερώνει το kubelet για τις συσκευές που διαθέτει το Node** το οποίο διαχειρίζεται
- **το ενημερώνει για αλλαγές στην υγεία τους** και τελικά
- **να απαντά σε αιτήματα (requests)** για συγκεκριμένες συσκευές από το kubelet, δίνοντάς του οδηγίες ως προς το ποιές αλλαγές πρέπει να γίνουν στο configuration file του Pod/container πρωτού το kubelet το προωθήσει στο υποκείμενο CRI [25] container runtime (dockershim, cri-o).

Το ακόλουθο διάγραμμα ακολουθίας παρουσιάζει τις λειτουργίες Initialization και Status Update του nvidia-device-plugin, το οποίο διαχειρίζεται τις NVIDIA GPUs σε ένα σύστημα Kubernetes.

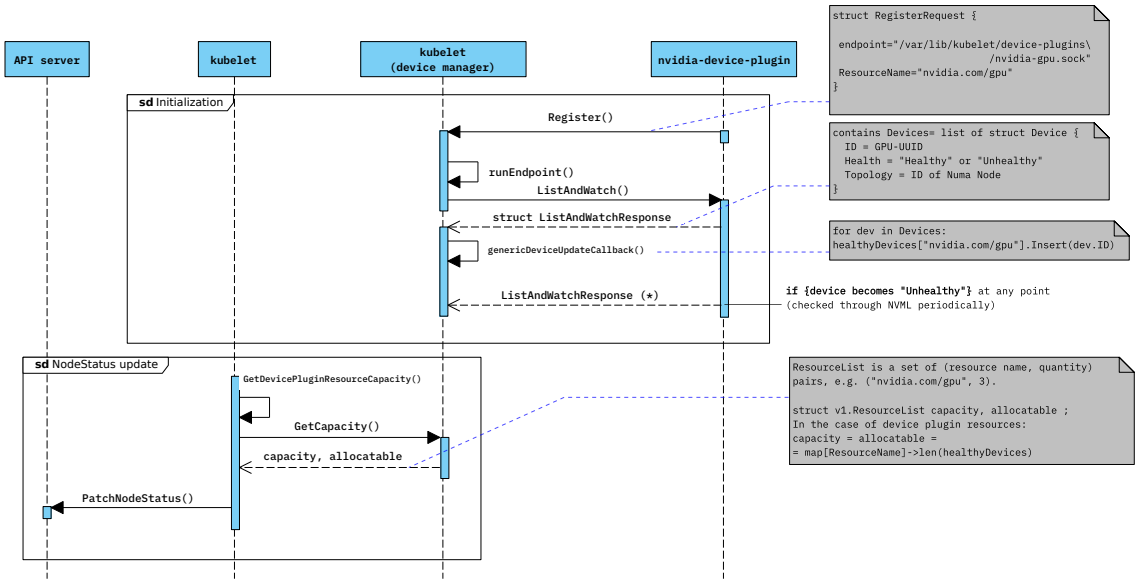

Figure 2.12: *device-plugin: Initialization και Status Update*

Το επόμενο διάγραμμα παρουσιάζει την ακολουθία εκχώρησης συσκευής (device allocation) από το nvidia-device-plugin.

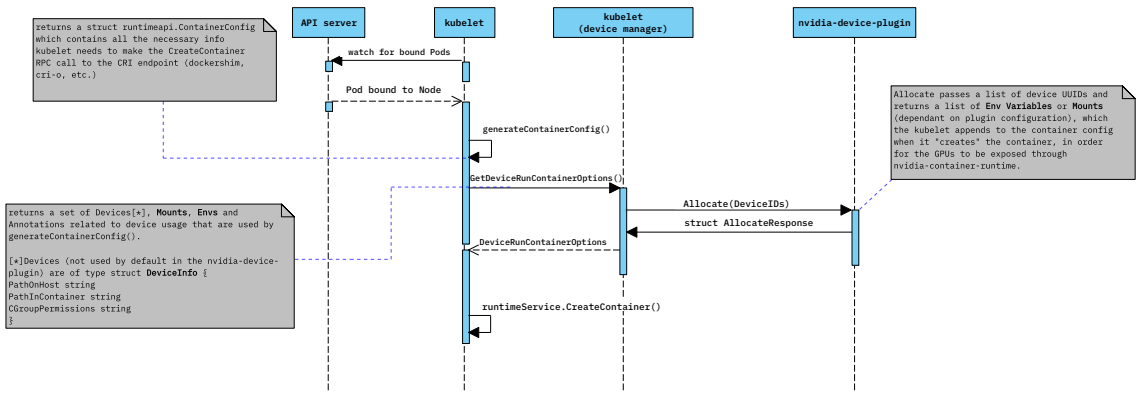

Figure 2.13: *device-plugin: GPU allocation*

Κεφάλαιο 3

Μελέτη της Unified Memory σε multi-process περιπτώσεις

Σε αυτό το κεφάλαιο παρέχουμε, από όσο γνωρίζουμε, την πρώτη μελέτη σχετικά με τη συμπεριφορά της Unified Memory του CUDA σε σενάρια με πολλαπλές διεργασίες (contexts). Χρησιμοποιούμε τον nvprof, τον NVIDIA Profiler [26] για τη μέτρηση σφαλμάτων σελίδας κατά την εκτέλεση στην GPU. Ξεκινάμε εξετάζοντας μια απλή υπόθεση, όπου μια διεργασία εκτελείται στην GPU και η μνήμη δεν είναι oversubscribed και προχωράμε σε πιο περίπλοκες περιπτώσεις, όπου οι υπολογισμοί πολλαπλών διεργασιών αλληλεπικαλύπτονται και η μνήμη GPU είναι oversubscribed. Επαληθεύουμε πειραματικά ότι το CUDA χρησιμοποιεί μια πολιτική αντικατάστασης LRU, καθώς και το γεγονός ότι τα σφάλματα σελίδας που προέρχονται από ένα CUDA context μπορούν να εκδιώξουν σελίδες που ανήκουν σε άλλο. Αυτό το εύρημα υπονοεί ότι κάθε CUDA context μπορεί να χρησιμοποιήσει όλη τη φυσική μνήμη GPU, ακόμη και αν άλλα context έχουν ενεργές αναθέσεις μνήμης. Ο μηχανισμός μας κοινής χρήσης GPU βασίζεται σε μεγάλο βαθμό σε αυτήν την παρατήρηση.

3.1 Σύντομη επανάληψη στην Unified Memory

Η Unified Memory (UM) είναι μια τεχνολογία υλικού / λογισμικού που επιτρέπει στις εφαρμογές να εκχωρούν μνήμη/δεδομένα τα οποία είναι προσβάσιμα από κώδικα που εκτελείται είτε στην CPU είτε στην GPU. Επιτρέπει επίσης στην GPU να χειρίζεται σφάλματα σελίδας. Όταν προκύψει σφάλμα σελίδας, το υποσύστημα Unified Memory (kernel module) ανακτά τη σελίδα που λείπει από τη μνήμη της GPU και επιλέγει μια σελίδα-θύμα για έξωση προς τη RAM. Αυτό σημαίνει ότι όταν η φυσική μνήμη της GPU είναι πλήρης και η μνήμη είναι oversubscribed (το σύνολο εκχωρήσεων μνήμης υπερβαίνει το φυσικό μέγεθός της), η μνήμη GPU λειτουργεί ως ένα είδος cache, χρησιμοποιώντας τη μνήμη RAM του συστήματος ως swap space.

Για κανονικές εκχωρήσεις μνήμης CUDA (όχι UM), κάθε byte εικονικής μνήμης που εκχωρείται πρέπει να υποστηρίζεται από ένα byte φυσικής μνήμης. Αυτό συνεπάγεται ότι το άθροισμα των εκχωρήσεων μνήμης GPU από όλα τα CUDA contexts μπορεί το πολύ να είναι ίσο με το φυσικό μέγεθος μνήμης GPU. Για παράδειγμα, στην περίπτωση μιας NVIDIA P100 με 16 GB μνήμη, εάν μία διεργασία (A) έχει εκχωρήσει 10 GB μέσω cudaMalloc, τότε μια άλλη διεργασία (B) μπορεί να εκχωρήσει (allocate) το πολύ 6 GB μνήμης GPU.

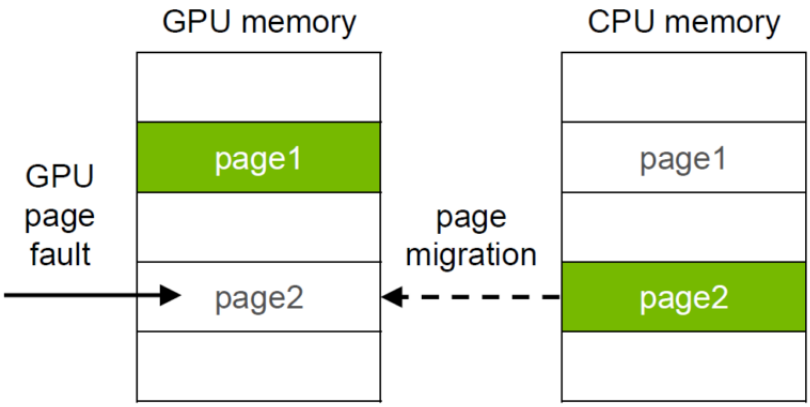

Figure 3.1: *Χειρισμός σφάλματος σελίδας από την Unified Memory*

προ-Pascal συμπεριφορά Unified Memory

Όταν μια εφαρμογή εκκινεί έναν πυρήνα (kernel), το υποσύστημα Unified Memory μεταφέρει όλες τις εκχωρήσεις Unified Memory του context στη μνήμη της GPU προτού ξεκινήσει η εκτέλεση του πυρήνα. Αυτό είναι απαραίτητο επειδή όλες οι γενιές GPU της NVIDIA πριν από την Pascal δεν υποστηρίζουν σφάλματα σελίδας κατά την εκτέλεση ενος kernel.
Παραθέτοντας το Nvidia Developer Blog [27]: «*Σε προ-Pascal GPUs, κατά την εκκίνηση ενός kernel, το CUDA runtime πρέπει να μετεγκαταστήσει όλες τις σελίδες που βρίσκονταν προηγουμένως στη RAM ή σε άλλη GPU στη μνήμη της συσκευής που εκτελεί τον kernel. Δεδομένου ότι αυτές οι παλαιότερες GPU δεν μπορούν να χειριστούν σφάλματα σελίδας, όλα τα δεδομένα πρέπει να βρίσκονται στην GPU σε περίπτωση που ο πυρήνας προσπελάσει κάποια από αυτά (ακόμα κι αν δεν το κάνει).*»

Συμπεριφορά Unified Memory από τη γενιά Pascal και μετά:

Η μεταφορά δεδομένων γίνεται μέσω demand-paging. Όταν ο GPU κώδικας προσπαθεί να προσπελάσει δεδομένα σε σελίδα που δεν βρίσκεται στη μνήμη της GPU, θα προκύψει σφάλμα σελίδας και το υπό εκτέλεση warp θα σταματήσει έως ότου η σελίδα μεταφερθεί στη μνήμη της GPU.
Για μια εξαιρετική σύγκριση της συμπεριφοράς της Unified Memory πριν και μετά τη γενιά Pascal, ανατρέξτε στην απάντηση στο Stackoverflow του Robert Crovella (υπαλλήλου της NVIDIA) [28].

3.2 Το πρόγραμμα αξιολόγησής μας

Εργαζόμαστε σε μια NVIDIA Tesla P100 (Pascal), έτσι τα αποτελέσματά μας ισχύουν για τις αρχιτεκτονικές Pascal, Volta, Turing, Ampere και νεότερες.

Εκτελέσαμε τα πειράματά μας σε μια Nvidia Tesla P100 με 16 GiB φυσικής μνήμης. Σημειώστε ότι δεν είναι και τα 16 GiB διαθέσιμα στις διεργασίες χρήστη. Ένα μικρό μέρος διατίθεται στην GPU για δομές του driver όπως contexts, μετρητές επιδόσεων και άλλες μη-δημοσιευμένες δομές. Στην πράξη, η «χρησιμοποιήσιμη από εφαρμογές» μνήμη είναι ~15.5 GiB, κυμαινόμενη σε βήματα ~100-200 MiB βάσει του πλήθους των ενεργών context.

Το δοκιμαστικό μας πρόγραμμα εκχωρεί NUM_CHUNKS περιοχές μνήμης των CHUNK_-

SIZE byte. Στη συνέχεια τα αρχικοποιεί στη CPU (host) και τελικά καλεί τον kernel "add" για κάθε chunk, ο οποίος αυξάνει την τιμή κάθε στοιχείου κατά 1. Στη συνέχεια, αναμένει το πάτημα οποιουδήποτε πλήκτρου πριν από την εκ νέου εκτέλεση των kernel, προκειμένου να μελετήσουμε τη συμπεριφορά όταν τα δεδομένα βρίσκονται ήδη στην GPU.

Listing 3.1: Το πρόγραμμα αξιολόγησης

```
#include <iostream>
#include <math.h>
#include <cuda_profiler_api.h>
#include <sys/time.h>
#define NUM_CHUNKS 5
#define CHUNK_SIZE 1L<<31
#define NUM_ITERATIONS 1
// define CUDA kernel which increments all the elements of an array
__global__  void add(uint64_t  n, char *x) {
      uint64_t index = blockIdx.x * blockDim.x + threadIdx.x;
      uint64_t stride = blockDim.x * gridDim.x;
      for ( size_t i = index; i < n; i += stride) {
        x[i] = x[i] + x[i];
      }
    }

int main(void) {
struct timeval tv1,tv2;
size_t N = CHUNK_SIZE;
char *x[NUM_CHUNKS];
// Allocate Unified Memory -- accessible from CPU or GPU
// Split the allocations into chunks of 2GiB
  for (int i=0; i<NUM_CHUNKS; i++) {
      cudaMallocManaged(&x[i], N*sizeof(char));
  }
  for (int i=0; i<NUM_CHUNKS; i++) {
    for (size_t k = 0; k < N; k++){
      x[i][k] = 1;
    }
  }
  // Launch kernel on the GPU
  int blockSize = 512;
  int numBlocks = N / blockSize;
  if ((N % blockSize) > 0 ) {numBlocks+=1;}

  for (int j=0; j<NUM_ITERATIONS; j++){
    gettimeofday(&tv1, NULL);
```

```
38      for(int i=0; i<NUM_CHUNKS; i++) {
39        add<<<numBlocks, blockSize>>>(N,x[i]);
40      }
41
42    cudaDeviceSynchronize();
43
44    // Wait for input before doing a single rerun
45    getchar();
46
47    //for (int i=NUM_CHUNKS - 1; i>=0; i--){
48      for (int i=0; i<NUM_CHUNKS; i++) {
49        add<<<numBlocks, blockSize>>>(N,x[i]);
50    }
51    cudaDeviceSynchronize();
52
53  // Wait for GPU to finish before accessing on host
54    cudaDeviceSynchronize();
55
56    return 0;
57  }
```

3.3 Μια Διεργασία - χωρίς oversubscription

Ξεκινάμε μία μόνο διεργασία που λειτουργεί με 5 κομμάτια των 2 GiB το καθένα, συνολικά 10 GiB. Σημειώστε ότι κάθε εκκίνηση πυρήνα επεξεργάζεται 2 GiB, για συνολικά 5 εκτελέσεις πυρήνα για πρόσβαση στο σύνολο των 10 GiB.
Εφόσον πρώτα αγγίζουμε και αρχικοποιούμε τη μνήμη στη CPU, συνολικά 10 GiB από Host to Device σφάλματα σελίδας συμβαίνουν όταν τα δεδομένα προσπελάζονται έπειτα στην GPU κατά την εκτέλεση του kernel.
Δώστε προσοχή στο Total Time για την πρώτη εκτέλεση: 3,82 δευτερόλεπτα. Αυτό περιλαμβάνει και τους χρόνους χειρισμού σφαλμάτων σελίδας. Τη δεύτερη φορά η εκτέλεση διαρκεί μόνο 0,12 δευτερόλεπτα, καθώς τα δεδομένα βρίσκονται ήδη στη μνήμη της GPU. Σε αυτήν την περίπτωση, ολόκληρο το Working Set της εφαρμογής χωρά στη μνήμη GPU.

3.4 Μια διεργασία - oversubscription μνήμης

Ξεκινάμε μία διεργασία που λειτουργεί με 9 κομμάτια των 2 GiB το καθένα, συνολικά 18 GiB. Σημειώστε ότι κάθε εκκίνηση πυρήνα επεξεργάζεται 2 GiB, για ένα σύνολο 9 πυρήνων για πρόσβαση σε ολόκληρα τα 18 GiB.
Κρίνοντας από τα 36 GiB των Host to Device και τα 16 + 2 (+2) * GiB Device to Host σφάλματα σελίδας, μπορούμε να υποθέσουμε ότι χρησιμοποιείται μια πολιτική αντικατάστασης LRU (Least Recently Used), όπως φαίνεται παρακάτω στο Σχήμα 3.4:

```
==6986== NVPROF is profiling process 6986, command: ./no_oversubscribe
[0] Total time = 3.823066 seconds

Single Rerun Total time = 0.124957 seconds
==6986== Profiling application: ./no_oversubscribe
==6986== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:  100.00%  3.94702s        10  394.70ms  24.853ms  775.04ms  add(unsigned long, char*)
      API calls:   92.88%  3.94699s         3  1.31566s  7.4750us  3.82272s  cudaDeviceSynchronize
                    7.08%  300.84ms         5  60.168ms  19.949us  300.67ms  cudaMallocManaged
                    0.02%  693.45us        10  69.344us  5.5990us  415.57us  cudaLaunchKernel
                    0.02%  683.31us         1  683.31us  683.31us  683.31us  cuDeviceTotalMem
                    0.01%  218.19us       101  2.1600us     173ns  91.642us  cuDeviceGetAttribute
                    0.00%  46.508us         1  46.508us  46.508us  46.508us  cuDeviceGetName
                    0.00%  2.9770us         1  2.9770us  2.9770us  2.9770us  cuDeviceGetPCIBusId
                    0.00%  2.3660us         3     788ns     228ns  1.8750us  cuDeviceGetCount
                    0.00%  1.5530us         2     776ns     248ns  1.3050us  cuDeviceGet
                    0.00%     849ns         1     849ns     849ns     849ns  cudaGetLastError
                    0.00%     340ns         1     340ns     340ns     340ns  cuDeviceGetUuid

==6986== Unified Memory profiling result:
Device "Tesla P100-PCIE-16GB (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
  111949  93.665KB  4.0000KB  0.9961MB  10.00000GB   1.217278s  Host To Device
   30732         -         -         -           -   3.719032s  Gpu page fault groups
Total CPU Page faults: 30720
```

Figure 3.2: *UM:Μια διεργασία, χωρίς oversubscription*

```
==24074== NVPROF is profiling process 24074, command: ./oversubscribe_single
[0] Total time = 7.233462 seconds

Single Rerun Total time = 7.794011 seconds
==24074== Profiling application: ./oversubscribe_single
==24074== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:  100.00%  15.0272s        18  834.84ms  757.41ms  1.02775s  add(unsigned long, char*)
      API calls:   98.46%  15.0270s         3  5.00900s  8.9330us  7.79369s  cudaDeviceSynchronize
                    1.53%  233.91ms         9  25.990ms  16.123us  233.72ms  cudaMallocManaged
                    0.01%  775.19us         1  775.19us  775.19us  775.19us  cuDeviceTotalMem
                    0.00%  383.08us        18  21.282us  4.8160us  168.15us  cudaLaunchKernel
                    0.00%  170.80us       101  1.6910us     173ns  68.636us  cuDeviceGetAttribute
                    0.00%  27.187us         1  27.187us  27.187us  27.187us  cuDeviceGetName
                    0.00%  3.0880us         1  3.0880us  3.0880us  3.0880us  cuDeviceGetPCIBusId
                    0.00%  2.3130us         3     771ns     291ns  1.6730us  cuDeviceGetCount
                    0.00%  1.0120us         2     506ns     201ns     811ns  cuDeviceGet
                    0.00%     757ns         1     757ns     757ns     757ns  cudaGetLastError
                    0.00%     367ns         1     367ns     367ns     367ns  cuDeviceGetUuid

==24074== Unified Memory profiling result:
Device "Tesla P100-PCIE-16GB (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
  400281  94.305KB  4.0000KB  0.9961MB  36.00000GB   4.112297s  Host To Device
   10427  2.0000MB  2.0000MB  2.0000MB  20.36523GB   1.788062s  Device To Host
  110592         -         -         -           -  14.618042s  Gpu page fault groups
Total CPU Page faults: 55296
```

Figure 3.3: *UM: Μια διεργασία, oversubscription μνήμης*

[*]: Επειδή η χρησιμοποιούμενη φυσική μνήμη είναι *ελαφρώς μικρότερη από 16 GiB*, παρουσιάζονται ορισμένα επιπλέον σφάλματα σελίδας (αλυσιδωτές 4 KB εξώσεις στο εσωτερικό ενός κομματιού 2 GiB).

Μπορούμε να ενισχύσουμε αυτήν την υπόθεση ξεκινώντας τους kernels της 2ης εκτέλεσης με αντίστροφη σειρά [9-0] (Σχήμα 3.5 παρακάτω):

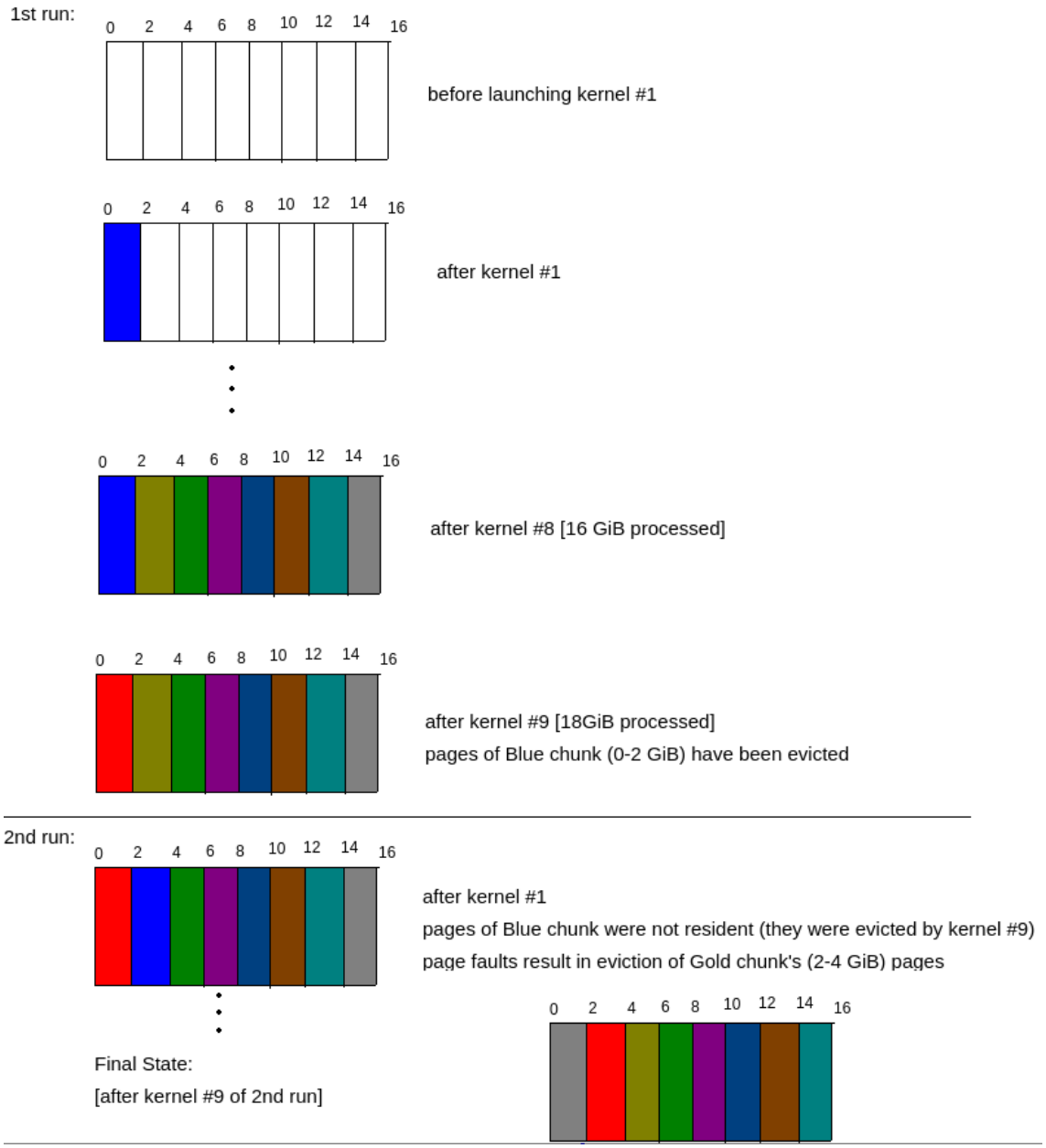

Figure 3.4: *UM: Ακολουθία αντικατάστασης*

Σημειώστε τον μειωμένο χρόνο της δεύτερης εκτέλεσης. Αυτή τη φορά, η πολιτική αντικατάστασης LRU δεν έχει τόσο επιζήμιο αποτέλεσμα, καθώς μόνο τα τελευταία 4 GiB των προσβάσεων προκαλούν σφάλματα σελίδας και εκκενώσεις των επόμενων σελίδων δεδομένων τα οποία ο πυρήνας προσπελάζει αμέσως μετά. Και πάλι, επειδή η χωρητικότητα είναι ελαφρώς μικρότερη από 16 GiB, λαμβάνουμε κάποια επιπλέον σφάλματα σελίδας που δεν ταιριάζουν με αυτό που παρουσιάζεται στο διάγραμμα.

3.5 Δύο διεργασίες - χωρίς oversubscription

Εκτελούμε δύο διεργασίες ταυτόχρονα, A & B. Κάθε μια από αυτές δημιουργεί μόνο 6 GiB Host to Device σφάλματα σελίδας, ίσα με το μέγεθος των δεδομένων τα οποία προσπελάζουν.

```
==28960== NVPROF is profiling process 28960, command: ./oversubscribe_single_reverse_second
[0] Total time = 6.821396 seconds

Single Rerun Total time = 1.890170 seconds
==28960== Profiling application: ./oversubscribe_single_reverse_second
==28960== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:  100.00%  8.71074s        18  483.93ms  24.853ms  981.52ms  add(unsigned long, char*)
      API calls:   97.34%  8.71066s         3  2.90355s  11.548us  6.82106s  cudaDeviceSynchronize
                    2.64%  236.12ms         9  26.235ms  16.856us  235.92ms  cudaMallocManaged
                    0.01%  840.92us         1  840.92us  840.92us  840.92us  cuDeviceTotalMem
                    0.01%  633.78us        18  35.209us  5.2970us  292.22us  cudaLaunchKernel
                    0.00%  251.03us       101  2.4850us     166ns  104.14us  cuDeviceGetAttribute
                    0.00%  34.303us         1  34.303us  34.303us  34.303us  cuDeviceGetName
                    0.00%  2.5480us         1  2.5480us  2.5480us  2.5480us  cuDeviceGetPCIBusId
                    0.00%  1.6270us         3     542ns     207ns  1.1740us  cuDeviceGetCount
                    0.00%     961ns         2     480ns     202ns     759ns  cuDeviceGet
                    0.00%     642ns         1     642ns     642ns     642ns  cudaGetLastError
                    0.00%     332ns         1     332ns     332ns     332ns  cuDeviceGetUuid

==28960== Unified Memory profiling result:
Device "Tesla P100-PCIE-16GB (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
  246520  93.577KB  4.0000KB  0.9961MB  22.00000GB   2.509481s  Host To Device
    3259  2.0000MB  2.0000MB  2.0000MB  6.365234GB  560.0505ms  Device To Host
   67591         -         -         -           -   8.289696s  Gpu page fault groups
Total CPU Page faults: 55296
```

Figure 3.5: *UM: Δεύτερη εκτέλεση με ανάποδη σειρά*

Δεδομένου ότι και τα δύο Working Sets χωρούν στη φυσική μνήμη GPU, δεν υπάρχουν περαιτέρω σφάλματα και οι επαναλήψεις χρειάζονται μόνο ~70 ms, όπως μπορούμε να δούμε στο Σχήμα 3.6.

```
==29882== NVPROF is profiling process 29882, command: ./increment_5g.cu
[0] Total time = 2.285040 seconds

Single Rerun Total time = 0.074728 seconds
==29882== Unified Memory profiling result:
Device "Tesla P100-PCIE-16GB (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
   67569  93.111KB  4.0000KB  0.9922MB  6.000000GB  672.5769ms  Host To Device
   18433         -         -         -           -   2.216746s  Gpu page fault gr
ps
Total CPU Page faults: 18432
```

A

```
==29893== NVPROF is profiling process 29893, command: ./increment_5g.cu
[0] Total time = 2.305340 seconds

Single Rerun Total time = 0.069259 seconds
==29893== Unified Memory profiling result:
Device "Tesla P100-PCIE-16GB (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
   67171  93.663KB  4.0000KB  0.9883MB  6.000000GB  689.1920ms  Host To Device
   18433         -         -         -           -   2.238830s  Gpu page fault g
ps
Total CPU Page faults: 18432
```

B

Figure 3.6: *UM: Δύο διεργασίες, χωρίς oversubscription*

3.6 Δύο διεργασίες - oversubscription μνήμης

Εκτελούμε τις A & B με τον παρακάτω τρόπο και στη συνέχεια κάνουμε τις παρατηρήσεις μας:

1. Η A ξεκινά 5 πυρήνες, καθένας από τους οποίους κάνει increment 2 GiB για συνολικά 10 GiB.
2. Η B ξεκινά 5 πυρήνες, καθένας από τους οποίους κάνει increment 2 GiB για σύνολο 10 GiB. Αυτό έχει ως αποτέλεσμα την έξωση σελίδων μεγέθους ~4,6 GiB που ανήκουν στην A (θυμηθείτε ότι η πραγματική χωρητικότητα μνήμης GPU είναι μικρότερη από 16 GiB). Η LRU πολιτική μας οδηγεί στο συμπέρασμα πως είναι οι περιοχές [0, 4.6] GiB
3. Η διεργασία B τερματίζεται. Το context της έχει καταστραφεί και δεν κατέχει πλέον μνήμη στην GPU.
4. Η A εκτελεί ξανά το σύνολο των 5 πυρήνων. Οι προσβάσεις για την περιοχή [0,4.6] GiB οδηγούν σε σφάλματα σελίδας, επειδή η B έχει ήδη εκδιώξει αυτήν την περιοχή στο βήμα 2. Ωστόσο, αυτά τα σφάλματα σελίδας δεν οδηγούν σε έξωση οποιουδήποτε frame, καθώς η B έχει ήδη τερματιστεί.

Ας υπολογίσουμε τα σύνολα:
Διεργασία **A**:

- 10 GiB υποχρεωτικές Host to Device μεταφορές για το βήμα #1
- 4.6 GiB Device to Host μεταφορές μετά την έξωση από την B στο βήμα #2
- 4.6 GiB Host to Device μεταφορές για την ανάκτηση και προσπέλαση των σελίδων από το βήμα #4
- **Σύνολο**: 14.6 GiB HtoD και 4.6 GiB DtoH

Διεργασία **B**:

- 10 GiB υποχρεωτικές Host to Device μεταφορές για το βήμα #2
- **Σύνολο**: 10 GiB HtoD

Αυτές οι προκαταρκτικές παραδοχές είναι σύμφωνες με τα αποτελέσματα που αποκτήσαμε από τον nvprof και τα οποία παρουσίαζουμε στην Εικόνα 3.7 παρακάτω.

> Το σημαντικό συμπέρασμα εδώ είναι ότι κατά τον χειρισμό σφαλμάτων σελίδας, frames (σελίδες που βρίσκονται στη μνήμη της GPU) **από οποιοδήποτε context** εκδιώκονται.

```
==31439== Unified Memory profiling result:                                    A
Device "Tesla P100-PCIE-16GB (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
  164692  93.103KB  4.0000KB  0.9922MB  14.62305GB   1.674699s  Host To Device
    2367  2.0000MB  2.0000MB  2.0000MB  4.623047GB  402.7704ms  Device To Host
   44927         -         -         -           -   4.839089s  Gpu page fault grou
ps
Total CPU Page faults: 30720
```

```
==31740== Unified Memory profiling result:                                    B
Device "Tesla P100-PCIE-16GB (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
  111810  93.781KB  4.0000KB  0.9961MB  10.00000GB   1.136773s  Host To Device
   30721         -         -         -           -   4.004468s  Gpu page fault grou
ps
Total CPU Page faults: 30720
```

Figure 3.7: *UM: Δύο διεργασίας, oversubscription μνήμης*

3.7 Δύο διεργασίες - Αρνητική παρεμβολή μνήμης (thrashing)

Όταν το πρόγραμμα της διεργασίας A εκτελείται μόνο του με ένα μέγεθος Working Set ίσο με 10 GiB, τότε μόνο η πρώτη επανάληψη (αυτή που δημιουργεί σφάλματα σελίδας και τοποθέτηση δεδομένων στη μνήμη GPU) έχει μεγάλο χρόνο εκτέλεσης. Οι επακόλουθες επαναλήψεις βρίσκουν τα δεδομένα «έτοιμα» στη μνήμη GPU και χρειάζονται μόνο χιλιοστά του δευτερολέπτου για εκτέλεση. Ας αυξήσουμε τον αριθμό των επαναλήψεων του «increment» σε 100 και ας δούμε πώς συμπεριφέρεται η A όταν τρέχει μόνη της (Εικόνα 3.8):

Τα αποτελέσματα είναι σύμφωνα με την πρόβλεψή μας. Οι επόμενες εκτελέσεις μετά την πρώτη διαρκούν περίπου 115 ms, καθώς τα δεδομένα βρίσκονται ήδη στην GPU και δεν παρουσιάζονται σφάλματα σελίδας.

Ας εξετάσουμε τώρα τι συμβαίνει όταν οι A και B εκτελούν παράλληλα 100 επαναλήψεις του «increment»: Στο Σχήμα 3.9 παρουσιάζουμε τα αποτελέσματα του nvprof για τη διεργασία A. Τη διακόπτουμε μετά από 5 επαναλήψεις, καθώς θα χρειαζόταν πολύς χρόνος για να ολοκληρωθεί. Τα αποτελέσματα ήταν τα ίδια και για τη διεργασία B.

Παρατηρούμε αύξηση 250x στο χρόνο εκτέλεσης. Επειδή οι πυρήνες (kernels) από τις A & B «εκτελούνται» ταυτόχρονα (αν και η GPU εναλλάσσεται μεταξύ των context με time-sliced τρόπο) και η μνήμη είναι oversubscribed (τα Working Sets στο σύνολό τους δεν χωρούν στη μνήμη της GPU) υπάρχει ιδιαίτερος ανταγωνισμός για τη φυσική μνήμη GPU που οδηγεί σε thrashing, κατάταση κατά την οποία ο χειρισμός σφαλμάτων σελίδας επιβραδύνει δραματικά τους ουσιαστικούς υπολογισμούς.

```
==1437== NVPROF is profiling process 1437, command: ./thrashing
Initialization on host complete. Press any key to run GPU kernels

[0] Total time = 3.515558 seconds
[1] Total time = 0.115255 seconds
[2] Total time = 0.115207 seconds
[3] Total time = 0.115230 seconds
[4] Total time = 0.115182 seconds
[5] Total time = 0.115192 seconds
[6] Total time = 0.115204 seconds
[7] Total time = 0.115179 seconds
[8] Total time = 0.115178 seconds
[9] Total time = 0.115201 seconds
[10] Total time = 0.115181 seconds
[11] Total time = 0.115183 seconds
[12] Total time = 0.115205 seconds
[13] Total time = 0.115182 seconds
==1535== Unified Memory profiling result:
Device "Tesla P100-PCIE-16GB (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
  113325  92.527KB  4.0000KB  0.9961MB  10.00000GB   1.129316s  Host To Device
   30720         -         -         -           -   3.741908s  Gpu page fault grou
ps
Total CPU Page faults: 30720
```

Figure 3.8: *UM: Μια διεργασία, 100 επαναλήψεις*

```
==1629== NVPROF is profiling process 1629, command: ./thrashing
Initialization on host complete. Press any key to run GPU kernels

[0] Total time = 9.793950 seconds
[1] Total time = 25.965820 seconds
[2] Total time = 24.063141 seconds
[3] Total time = 25.870272 seconds
[4] Total time = 24.045416 seconds
[5] Total time = 23.742236 seconds
^C==1629== Profiling application: ./thrashing
==1629== Unified Memory profiling result:
Device "Tesla P100-PCIE-16GB (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
  903833  79.493KB  4.0000KB  0.9961MB  68.52057GB   9.834552s  Host To Device
   31177  2.0000MB  2.0000MB  2.0000MB  60.89258GB   6.481411s  Device To Host
  210598         -         -         -           -  66.366487s  Gpu page fault g
ps
Total CPU Page faults: 30720
```

Figure 3.9: *UM: Thrashing, oversubscription μνήμης*

3.8 Prefetching μνήμης στην GPU

Κατά τη χρήση της Unified Memory, μπορούμε να μεταφέρουμε ρητά και εκ των προτέρων περιοχές μνήμης στην GPU χρησιμοποιώντας την κλήση cudaMemPrefetchAsync() [29]. Λειτουργεί με παρόμοιο τρόπο με το να καλούσαμε cudaMemcpy() για να μετακι-

νήσουμε δεδομένα στην GPU σε μη Unified σενάριο. Κάνοντας prefetch δεδομένα στην GPU, αποφεύγουμε τη δημιουργία πολλαπλών σφαλμάτων σελίδας και την εξυπηρέτησή τους ως ανεξάρτητες μονάδες, κάτι που είναι χρήσιμο σε περιπτώσεις όπου γνωρίζουμε εκ των προτέρων ότι μια περιοχή θα χρησιμοποιηθεί από την εφαρμογή. Παρακάτω παρουσιάζουμε τις τροποποιήσεις που κάναμε στο πρωτότυπο πρόγραμμά μας, ώστε να κάνουμε prefetch τη μνήμη στην GPU πριν από την εκκίνηση των υπολογιστικών πυρήνων (kernels).

Listing 3.2: Prefetching memory to the GPU

```
1 // Prefetch GPU memory
2 int device = -1;
3 cudaGetDevice(&device);
4 for (int i = 0; i < NUM_CHUNKS; i++) {
5   cudaMemPrefetchAsync(x[i], N*sizeof(char), device, NULL);
6 }
```

3.8.1 Επανεξέταση της περίπτωσης με μια διεργασία

Ας επανεξετάσουμε την περίπτωση μιας διεργασίας χωρίς oversubscription (10 GiB): Εξετάζουμε τα αποτελέσματα του nvprof, που φαίνονται στο Σχήμα 3.10. Αφού τα δεδομένα κατοικούν ήδη στη GPU όταν ξεκινά η εκτέλεση, δεν υπάρχουν σφάλματα σελίδας στην GPU (το prefetching εξυπηρετείται σε κομμάτια των 2 MB) και ο χρόνος εκτέλεσης είναι ελάχιστος (115 ms ακόμη και για τον πρώτο πυρήνα. Συγκρίνετέ το με την περίπτωση μιας διεργασίας χωρίς prefetching στην αρχή του κεφαλαίου). Δεν πρέπει να ξεχνάμε ότι η διαδικασία του prefetching χρειάζεται και αυτή κάποιο χρόνο, αλλά χρησιμοποιεί λιγότερες συναλλαγές PCIe καθώς μεταφέρει δεδομένα μαζικά, οπότε ο συνολικός χρόνος εκτέλεσης μειώνεται.

```
==1787== NVPROF is profiling process 1787, command: ./prefetch
Initialization on host complete.
Prefetching Complete. Press any key to run the GPU kernels

[0] Total time = 0.115318 seconds
==1787== Unified Memory profiling result:
Device "Tesla P100-PCIE-16GB (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
    5120  2.0000MB  2.0000MB  2.0000MB  10.00000GB   1.178874s  Host To Device
Total CPU Page faults: 30720
```

Figure 3.10: *UM: Prefetching μνήμης στην GPU*

3.8.2 Εξώσεις σελιδών κατά το Prefetching

Παρατηρούμε ότι το Prefetching ακολουθεί την ίδια λογική με τον κανονικό χειρισμό σφαλμάτων σελίδας. Απομακρύνει τα πλαίσια με τρόπο LRU και από οποιοδήποτε context. Για να επαληθεύσουμε τη συμπεριφορά, δημιουργούμε ένα συγκεκριμένο σενάριο ελέγχου. Σε αυτό το πείραμα:

1. η διεργασία A τρέχει μια επανάληψη στα 10 GiB
2. η διεργασία B κάνει prefetch 10 GiB στην μνήμη της GPU και τρέχει μια επανάληψη
3. η A τρέχει μια δεύτερη επανάληψη

Δείχνουμε την έξοδο του παραπάνω πειράματος στην (Εικόνα 3.11):

```
grgalex@grgalex-gpu-devel-p100:~/DEMO_3_5/PREFETCH$ nvprof ./increment_10g
sizeof(char) = 1
==2394== NVPROF is profiling process 2394, command: ./increment_10g
[0] Total time = 4.060820 seconds
Press any key to re-run.
B prefetches 10 GiB
Single Rerun Total time = 9.895979 seconds

==2394== Unified Memory profiling result:
Device "Tesla P100-PCIE-16GB (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
  222234  94.366KB  4.0000KB  0.9961MB  20.00000GB   2.725839s  Host To Device
    5120  2.0000MB  2.0000MB  2.0000MB  10.00000GB   1.025000s  Device To Host
   61441         -         -         -           -  13.738795s  Gpu page fault grou
ps
Total CPU Page faults: 30720
```

A

Figure 3.11: *UM: Αντικατάσταση σελιδών κατά το Prefetching*

Ας αναλύσουμε τι συμβαίνει, βήμα προς βήμα:

1. η A φέρνει 10 GiB σελίδων στην GPU
2. η B κάνει prefetch 10 GiB. Διώχνει περίπου 5 GiB από τη διεργασία A (πιο συγκεκριμένα τις περιοχές [0, 5G])
3. η A ξανατρέχει. Ο πρώτος πυρήνας της προσπελάζει την περιοχή [0, 2G] η οποία δεν βρίσκεται πια στη μνήμη της GPU
4. η A σταδιακά εκδιώκει την δική της περιοχή [5, 7G] λόγω της πολιτικής LRU
5. αυτή η αλυσιδωτή αντίδραση οδηγεί τελικά σε άλλα 10 GiB DtoH σφάλματα για συνολικά 20 GiB. Αυτό συμβαίνει επειδή η A προσπελάζει τις σελίδες που έχει ήδη εκδιώξει στο προηγούμενο βήμα.

Το κύριο συμπέρασμα εδώ είναι ότι **το prefetching αντιμετωπίζεται με τον ίδιο ακριβώς τρόπο (σχετικά με την LRU πολιτική και τις εξώσεις) με τον οποίο αντιμετωπίζονται οι μεταφορές σελίδων που προκύπτουν από κανονικά σφάλματα σελίδας.**

Κεφάλαιο 4

Η προσέγγισή μας

Έχοντας δει πώς συμπεριφέρεται η Unified Memory σε σενάρια πολλαπλών διεργασιών στο προηγούμενο κεφάλαιο, σκιαγραφούμε σιγά-σιγά τον μηχανισμό μας. Στόχος μας είναι να επιτρέψουμε σε δύο ή περισσότερες εφαρμογές χρηστών να χρησιμοποιούν την ίδια GPU στον Kubernetes και όπου κάθε χρήστης μπορεί να χρησιμοποιήσει ολόκληρη τη μνήμη GPU. Η κύρια περίπτωση χρήσης μας είναι αυτή της διαδραστικής ανάπτυξης ML σε Jupyter Notebooks, όπου οι υπολογισμοί GPU έρχονται σε ριπές και υπάρχουν περίοδοι αδράνειας. Ωστόσο, όπως θα δούμε στη συνέχεια, ο μηχανισμός μας βρίσκει εφαρμογή ακόμη και σε μη διαδραστικές περιπτώσεις.
Ας περιγράψουμε τα βήματα που ακολουθήσαμε για να δημιουργήσουμε τον μηχανισμό κοινής χρήσης GPU:

1. Παροχή ενός μηχανισμού για την **μετατροπή όλων των εκχωρήσεων μνήμης CUDA σε Managed (Unified Memory)** μέσω hooking των CUDA API.
2. Διαβεβαίωση **πως αυτός ο μηχανισμός διατηρεί την ορθή εκτέλεση των εφαρμογών**
3. **Μέτρηση του overhead μιας εφαρμογή το οποίο προκύπτει από τη χρήση Unified Memory αυτής καθαυτής** (όταν εκτελείται μόνη της). Σύγκριση των non-unified με τα unified αποτελέσματα για μια πληθώρα από ML benchmarks.
4. **Απόδειξη και εκτίμηση του thrashing** όταν 2+ χρήστες δουλεύουν ταυτόχρονα στην ίδια GPU και οι υπολογιστικές ριπές επικαλύπτονται.
5. **Αντιμετώπιση του thrashing** μέσω υλοποίησης ενός time-slicing μηχανισμού (daemon/scheduler). Το κβάντο χρόνου πρέπει να είναι αρκετά μεγάλο ώστε να δικαιολογεί το κόστος του preemption (και άρα της εναλλαγής περιεχομένων μνήμης).
6. **Ενσωμάτωση του μηχανισμού στην πλατφόρμα Kubernetes** (alexo-device-plugin)

4.1 Διαφανής μετατροπή κλήσεων εκχώρησης μνήμης σε Unified Memory

[Ενώ το επίκεντρο αυτής της ενότητας είναι το hooking κλήσεων του CUDA API, όλες οι πληροφορίες που παρουσιάζονται ισχύουν για το hooking οποιασδήποτε συνάρτησης και καλύπτουν όλες τις επιλογές σύνδεσης / φόρτωσης.]
Θέλουμε να μετατρέψουμε τις κλήσεις εκχώρησης μνήμης cudaMalloc(), cuMemAlloc() στις αντίστοιχες Unified, cudaMallocManaged(), cuMemAllocManaged() προκειμένου να πετύχουμε τα ακόλουθα:

- oversubscription της μνήμης της GPU
  - Οι εκχωρήσεις μνήμης δεν έχουν πλέον 1-1 αντιστοίχιση με την φυσική μνήμη
- να επιτρέψουμε σε κάθε συστεγαζόμενη εφαρμογή να χρησιμοποιεί όλη την φυσική μνήμη της GPU
  - κάθε εφαρμογή μπορεί να εκδιώξει σελίδες από άλλες (αδρανείς) εφαρμογές, από την GPU προς τη RAM του συστήματος

Αναλύουμε τη συμπεριφορά των συστεγαζόμενων εφαρμογών που χρησιμοποιούν Unified Memory στο προηγούμενο κεφάλαιο. Προκειμένου να κάνουμε wrap μια συνάρτηση, αναγκάζοντας την εφαρμογή χρήστη να χρησιμοποιεί μια δική μας έκδοση πρέπει να κάνουμε τα ακόλουθα:

- Δημιουργία μιας shared library η οποία ορίζει μια συνάρτηση με το ίδιο όνομα (π.χ. cudaMalloc). Αυτή η συνάρτηση εσωτερικά καλή την «πραγματική» εκδοχή της εν λόγω συνάρτησης. Ο δείκτης προς την πραγματική συνάρτηση αποκτάται μέσω των συναρτήσεων dlopen() και dlsym().
- Εξασφάλιση ότι η εφαρμογή καλεί τη δική μας έκδοση της συνάρτησης. Για απλές περιπτώσεις, το επιτυγχάνουμε μέσω της μεταβλητής περιβάλλοντος LD_PRELOAD [30] η οποία εξασφαλίζει ότι το σύμβολο της συνάρτησης επιλύεται στη δική μας εκδοχή, συνδέοντας την shared library μας με την εφαρμογή πριν από όλες τις άλλες. Για πιο περίπλοκες περιπτώσεις όπως δυναμικά φορτωμένες βιβλιοθήκες (dynamically loaded shared libraries) πρέπει επίσης να κάνουμε wrap τη συνάρτηση dlsym() καθώς δεν αρκεί απλά να συνδέεται πρώτη η βιβλιοθήκη μας με την εφαρμογή.

4.1.0.1 Τύποι CUDA εφαρμογών και επιλογές σύνδεσης (linking)

- Αμιγές Driver API (καλεί μόνο συναρτήσεις του Driver API)
  - δυναμική σύνδεση (dynamically linked) με την libcuda.so
- Αμιγές Runtime API (καλεί μόνο συναρτήσεις του Ρυντιμε API)
  - στατική σύνδεση με την libcudart.a
  - δυναμική σύνδεση με την libcudart.so
- Μεικτό API
  - στατικά συνδεδεμένο Runtime (libcudart.a), δυναμικά συνδεδεμένο Driver API (libcuda.so)

- δυναμικά συνδεδεμένο Runtime (libcudart.so), δυναμικά συνδεδεμένο Driver (libcuda.so)
- δυναμικά συνδεδεμένο Runtime (libcudart.so), δυναμικά φορτωμένο (loaded) Driver (libcuda.so)
- δυναμικά **φορτωμένο** Runtime (libcudart.so), δυναμικά **φορτωμένο** Driver (libcuda.so)

Το Tensorflow χρησιμοποιεί την επιλογή (δ) της κατηγορίας μεικτού API. Το PyTorch χρησιμοποιεί την επιλογή (γ).

Στη συνέχεια παρουσιάζουμε τον κωδικά της συνάρτησης wrapper για την κλήση cudaMalloc.

```
1  void *real_cudaMallocManaged = NULL;  //global
2
3  cudaError_t cudaMalloc ( void** devPtr, size_t size ) {
4    cudaError_t result = cudaSuccess;
5
6    if (!real_cuMemAllocManaged) {
7      void *cudart_handle;
8      cudart_handle = dlopen("libcudart.so", RTLD_LAZY);
9      real_cudaMallocManaged = (void *)real_dlsym( cudart_handle, CUDA_SYMBOL_STRING
       (cudaMallocManaged));
10     dlclose(cudart_handle);
11     // just decrements the handle reference counter, does not unload libcudart
12   }
13
14   result = ((cudaError_t (*) ( void** devPtr, size_t size,   unsigned int  flags )
        )real_cudaMallocManaged) (devPtr,   size, cudaMemAttachGlobal);
15
16   return result;
```

Μπορούμε τώρα να προχωρήσουμε στη δοκιμή του τρόπου λειτουργίας του μηχανισμού (libunified.so) μας και στη διασφάλιση της σταθερότητας (οι εφαρμογές χρήστη τρέχουν σωστά) αυτού.

4.2 Επικύρωση της σταθερότητας του μηχανισμού μετατροπής

Πρέπει να διασφαλίσουμε ότι οποιαδήποτε εφαρμογή της οποίας τις εκχωρήσεις μνήμης μετατρέπουμε διαφανώς σε Managed (Unified Memory) μέσω του μηχανισμού που παρουσιάσαμε στο προηγούμενο κεφάλαιο:

- δεν οδηγεί σε σφάλματα προκύπτοντα από την χρήση της Unified Memory
- έχει τα ίδια αποτελέσματα εκτέλεσης (από άποψη διατήρησης ορθότητας) με την αρχική

Δεδομένου ότι δεν μπορούμε να αποδείξουμε μαθηματικά την ορθότητα του μηχανισμού μετατροπής μας, η μόνη επιλογή είναι να εκτελέσουμε μια μεγάλη ποικιλία εφαρμογών συνδεδεμένες με την libunified.so και να παρατηρήσουμε τη συμπεριφορά εκτέλεσης.

Για το σκοπό αυτό διεξάγουμε μια σειρά πειραμάτων που περιλαμβάνουν:

- Αμιγείς εφαρμογές CUDA
- προγράμματα Tensorflow
- προγράμματα PyTorch

Πιο συγκεκριμένα, εξετάσαμε τα εξής:

- Official CUDA Samples [31]
- Official Tensorflow Benchmarks [32]
- Official PyTorch Benchmarks [33]
- AI-Benchmark Suite [ETH Zurich] (Tensorflow) [34]
- Altis GPU Benchmarks [UT Austin] (state-of-the-art CUDA Benchmark Suite) [35]

Τα πειραματικά μας αποτελέσματα ενισχύουν την υπόθεση ότι η μετατροπή όλων των κλήσεων εκχώρησης μνήμης σε Unified Memory διατηρεί την ορθότητα εκτέλεσης της εφαρμογής. Όλα τα πειράματα διεξήχθησαν σε ένα Google Cloud VM με επεξεργαστή Intel Xeon 8-core, 52 GiB RAM και Nvidia Tesla P100 GPU με μνήμη 16 GB.

4.2.1 CUDA Samples

Τα δείγματα CUDA αποτελούνται από 50 εφαρμογές που καλύπτουν ένα ευρύ φάσμα λειτουργικοτήτων CUDA. Κάνουμε clone το Github Repository [31], εισερχόμαστε στον κατάλογο με τα δείγματα και εκτελούμε make. Στη συνέχεια, μπαίνουμε στον κατάλογο κάθε εφαρμογής και εκτελούμε κάθε πρόγραμμα, μία φορά στην αρχική του κατάσταση και μία φορά με ενεργοποιημένη τη βιβλιοθήκη μετατροπής μας (libunified).

Για παράδειγμα, για να εκτελέσουμε το matrixMul εισερχόμαστε στο directory matrixMul και εκτελούμε `LD_PRELOAD=libunified.so ./matrixMul`.

Ορισμένες δοκιμές δεν υποστηρίζονται στην NVIDIA P100, καθώς απαιτούν πιο πρόσφατα χαρακτηριστικά υλικού όπως τα Tensor Cores, τα οποία υπάρχουν στις γενιές Volta, Turing ή Ampere.

Από τις δοκιμές που μπορούν να εκτελεστούν στην P100, όλες ήταν επιτυχημένες εκτός από το simpleIPC. Η εφαρμογή simpleIPC χρησιμοποιεί την οικογένεια λειτουργιών CUDA Interprocess Communication. Χρησιμοποιούνται για να εκθέσουν μια περιοχή μνήμης από

ένα CUDA context σε ένα άλλο και δεν λειτουργούν για cudaMallocManaged εκχωρήσεις, σύμφωνα με την τεκμηρίωση της Nvidia [36]. Έχοντας εξετάσει προσεκτικά τον πηγαίο κώδικα, μπορούμε να πούμε ότι τα Tensorflow και Pytorch δεν χρησιμοποιούν καθόλου το μηχανισμό CUDA IPC, οπότε δεν μας αφορά.

4.2.2 Επίσημα Tensorflow Benchmarks

Κάνουμε clone το official Tensorflow Benchmark repository ([32]), πηγαίνουμε στον κατάλογο scripts/tf_cnn_benchmarks και εκτελούμε:

- `python run_test.py --full_tests`
- `LD\_PRELOAD=libunified.so python run_test.py --full_tests`

Και στις δύο περιπτώσεις, και οι 232 δοκιμές ολοκληρώθηκαν με επιτυχία. Δεν πραγματοποιείται ρητή επικύρωση εξόδου από τα TF benchmarks, ωστόσο η ακρίβεια του μοντέλου ελέγχεται και, καθώς δεν υπάρχουν σφάλματα και στις περισσότερες περιπτώσεις τα τυχαία seeds είναι σταθερά, μπορούμε να συμπεράνουμε ότι διατηρείται η ορθότητα.

4.2.3 Επίσημα PyTorch Benchmarks

Κλωνοποιούμε το repository Pytorch Benchmark ([33]) και εκτελούμε μόνο τα benchmarks ενός κόμβου:

- `pytest --ignore-machine-config test_bench.py`
- `LD_PRELOAD=libunified.so pytest --ignore-machine-config test_bench.py`

Και οι 104 δοκιμές ολοκληρώνονται επιτυχώς.

4.2.4 AI-Benchmark

Το AI-Benchmark είναι μια σουίτα benchmark ML (χρησιμοποιεί Tensorflow) που αναπτύχθηκε από τον Andrey Ignatov [37] του ETH Zurich, με αρχικό στόχο τη δοκιμή της απόδοσης των Smartphone και Mobile SoCs. Χρησιμοποιούμε την έκδοση GPU για υπολογιστές στις δοκιμές μας, ακολουθώντας τις οδηγίες στον επίσημο ιστότοπο AI-Benchmark [34]. Εγκαθιστούμε το πακέτο Python «ai-benchmark» και εκτελούμε το benchmark μέσω του κάτωθι script:

```
import ai_benchmark
benchmark = ai_benchmark.AIBenchmark()
results = benchmark.run(precision="high")
```

Ορίζουμε το precision ίσο με high για να εκτελέσουμε 10 φορές περισσότερες επαναλήψεις από την κανονική (default) περίπτωση, όπως τονίζεται στο documentation του project [3]

[3] https://pypi.org/project/ai-benchmark/

4.2.5 Altis GPU Benchmarks

Το Altis είναι μια σύγχρονη σουίτα benchmarking CUDA που αναπτύχθηκε από το SCEA Lab στο UT Austin. Το σχετικό paper δημοσιεύτηκε το 2020 [35] και το έργο είναι ανοιχτού κώδικα [38]. Η σουίτα benchmark Altis χωρίζεται σε τρία επίπεδα. Κάθε επίπεδο αντιπροσωπεύει benchmarks των οποίων οι τομείς εστίασης κυμαίνονται από χαρακτηριστικά χαμηλού επιπέδου, όπως εύρος ζώνης διαύλου έως από άκρο σε άκρο απόδοση πραγματικών εφαρμογών. Η ίδια κατηγοριοποίηση υιοθετείται και από το Scalable Heterogeneous Computing (SHOC) [39] Benchmark Suite. Η δομή των επιπέδων είναι ως εξής:

- Επίπεδο 0: Μέτρηση χαμηλού επιπέδου χαρακτηριστικών του hardware. Αυτό το επίπεδο περιέχει απλά benchmarks όπως το maxflop και το busBandwidth.
- Επίπεδο 1: Περιλαμβάνει βασικούς παράλληλους αλγορίθμους που εμφανίζονται συχνά και χρησιμοποιούνται σε πυρήνες πραγματικών εφαρμογών.
- Λεελ 2: πιο περίπλοκοι πυρήνες (kernels) εφαρμογών του πραγματικού κόσμου, που συχνά εντοπίζονται στη βιομηχανία.

Αφού κλωνοποιήσαμε το repository του project, εκτελέσαμε καθεμία από τις εφαρμογές των επιπέδων {0,1,2} στην αρχική τους μορφή και για άλλη μια φορά με εισηγμένη την libunified. Όλες οι εκτελέσεις ολοκληρώθηκαν επιτυχώς.

4.3 Μέτρηση του overhead στην επίδοση του μηχανισμού μετατροπής

Έχουμε πλέον επαρκή στοιχεία ότι οι εφαρμογές των οποίων τις εκχωρήσεις μνήμης μετατρέπουμε διαφανώς σε Managed (Unified Memory) μέσω του μηχανισμού μας εκτελούνται σωστά. Τώρα, πρέπει να εκτιμήσουμε το overhead απόδοσης, που απορρέει από το γεγονός ότι οι εκχωρήσεις μνήμης GPU είναι πλέον Unified.

Εκτελέσαμε τα ακόλουθα benchmarks και συλλέξαμε μετρήσεις:

- (PyTorch) Ryujaehun's pytorch-gpu-benchmark [40]
- (PyTorch) Official PyTorch benchmarks [33]
- (Tensorflow) Official Tensorflow benchmarks (tf_cnn_benchmarks) [32]
- (Tensorflow) AI-Benchmark [34]

Ακολουθήσαμε την εξής μεθοδολογία για κάθε benchmark:

1. Εκτελέσαμε το original (απείραχτο) benchmark αρκετές φορές (5 για το ai-benchmark, 10 για τα άλλα) για να μειώσουμε την τυχαιότητα και αποθηκεύσαμε τις εξόδους σε αρχεία.
2. Επαναλάβαμε το παραπάνω βήμα με την μεταβλητή περιβάλλοντος LD_PRELOAD=libunified.so, δηλαδή μετατρέποντας όλες τις εκχωρήσεις μνήμης σε Unified
3. Κάναμε parse τα αρχεία σε Python, συλλέξαμε τους χρόνους εκτέλεσης ανά μοντέλο και batch size και υπολογίσαμε τον μέσο όρο των εκτελέσεων για κάθε μοντέλο. Το βήμα αυτό το κάναμε για τις Stock αλλά και Unified μορφές εκτελέσεων.
4. Υπολογίσαμε το average latency slowdown (T(unified)/T(stock)) για κάθε μοντέλο. Για τα tf_cnn_benchmarks λαμβάνουμε images/sec σαν έξοδο του benchmark οπότε υπολογίζουμε το average throughput slowdown μέσω του τύπου Q(stock)/Q(unified).
5. Υπολογίσαμε τη συνολική μέση επιβράδυνση (Total Average Slowdown) ανά benchmark στο σύνολο των μοντέλων του.

Συνοψίζουμε τα τελικά αποτελέσματά μας στον Πίνακα 4.1.

Benchmark	Total Average Slowdown	stdev
PyTorch torchbenchmark	0.9994	0.0464
AI-Benchmark	1.0192	0.0261
tf_cnn_benchmarks	1.0173	0.0131
ryujaehun-pytorch-gpu-benchmark	1.0004	0.0306

Table 4.1: *Overall Slowdown της libunified*

Παρατηρούμε μέγιστη επιβάρυνση 1,9 % και μέση επιβάρυνση 0,9 % (που είναι αμελητέα) στις 4 σουίτες benchmark. Θεωρούμε ότι αυτό το κόστος (επίδοσης) είναι αποδεκτό και καταλήγουμε στο συμπέρασμα ότι μπορούμε να προχωρήσουμε με την ενσωμάτωση

του μηχανισμού μετατροπής (libunified) στον Kubernetes, αφού πρώτα αντιμετωπίσουμε τα ενδεχόμενα thrashing.

4.4 Παροχή ενός μηχανισμού για την αντιμετώπιση του thrashing

Έχοντας επαληθεύσει ότι η libunified.so λειτουργεί σωστά και έχει ελάχιστο overhead, έχουμε παράσχει έναν μηχανισμό που επιτρέπει την κοινή χρήση GPU. Πολλές εφαρμογές χρηστών μπορούν πλέον να εκτελούνται ταυτόχρονα στην ίδια GPU. Κάθε μία από αυτές τις εφαρμογές χρήστη μπορεί να εκχωρήσει και να χρησιμοποιήσει ολόκληρη τη μνήμη GPU. Ωστόσο, όταν η μνήμη είναι oversubscribed, και λόγω του γεγονότος ότι εμείς το επιτρέπουμε μέσω της χρήσης της Unified Memory, ενδέχεται να προκύψουν σφάλματα σελίδας. Ενώ στη γενική μας περίπτωση χρήσης της διαδραστικής ανάπτυξης, οι ριπές GPU από συστεγαζόμενες εφαρμογές χρήστη δεν αλληλεπικαλύπτονται, σίγουρα μπορεί να υπάρξουν περιπτώσεις όπου συμβαίνει αυτό. Πρέπει να χειριστούμε αυτές τις περιπτώσεις προσεκτικά και να αποτρέψουμε τα υπερβολικά σφάλματα σελίδας που προκαλούνται από τη συνεχή μεταφορά δεδομένων από και προς την GPU λόγω των σφαλμάτων.
Ως εκ τούτου, σε αυτήν την ενότητα θα:

- Επαληθεύσουμε ότι μπορούν να παρουσιαστούν περιπτώσεις thrashing υπό τον μηχανισμό sharing (libunified) τον οποίο παρέχουμε
- Παράσχουμε έναν μηχανισμό για την αποφυγή του thrashing, σειριοποιώντας την υποβολή (και συνεπώς εκτέλεση) πυρήνων GPU και ρητών αντιγραφών μνήμης από τις ανταγωνιζόμενες διεργασίες σε χρονικά παράθυρα (time quanta) με τρόπο round-robin.

4.4.1 Υπόβαθρο

Το Thrashing [41] ορίζεται ως μια κατάσταση στην οποία ο χρόνος που αφιερώνεται στο χειρισμό page-faults ξεπερνά τον χρόνο που αφιερώνεται κάνοντας χρήσιμους υπολογισμούς. Η Unified Memory ενεργοποιεί τα σφάλματα σελίδας στη μνήμη GPU, χρησιμοποιώντας τη μνήμη RAM του συστήματος ως swap space. Όταν η Μνήμη GPU είναι oversubscribed, το άθροισμα των εκχωρήσεων μνήμης των διεργασιών GPU υπερβαίνει τη φυσική χωρητικότητα GPU, έτσι μπορεί να συμβούν σφάλματα σελίδας και επακόλουθη έξωση σελίδων.

Ωστόσο, στην ειδική περίπτωση χρήσης των διαδραστικών φορτίων εργασίας ML (Jupyter), παρόλο που πολλές διεργασίες έχουν εκχωρήσει μνήμη GPU, τις περισσότερες φορές μόνο μία διεργασία θα εκκινεί ενεργά πυρήνες (θα χρησιμοποιεί ενεργά) (σ)την GPU. Ως εκ τούτου, αυτή η διεργασία θα υποστεί περιορισμένο μόνο αριθμό σφαλμάτων σελίδας (και στη συνέχεια οι άλλες διεργασίες θα υποστούν εξώσεις) κατά την ανάκτηση του συνόλου εργασίας της στη φυσική μνήμη GPU στην αρχή της υπολογιστικής ριπής. Θα ολοκληρώσει την ριπή της δουλειάς της από την αρχή έως το τέλος χωρίς να υποστεί έξτρα σφάλματα σελίδας.

Ακόμη και όταν περισσότερες από μία διεργασίες υποβάλλουν δουλειά στην GPU ταυτόχρονα, επιπλέον σφάλματα σελίδας (εκτός από τα αρχικά που περιγράφουμε παραπάνω) θα συμβούν μόνο εάν το άθροισμα των δεδομένων που φέρνουν στη μνήμη GPU για τη συγκεκριμένη υπολογιστική ριπή υπερβαίνει τη φυσική χωρητικότητα της GPU μνήμης. Για μια NVIDIA Tesla P100 με μνήμη 16 GB, αυτό σημαίνει ότι: M_bst (A) + M_bst (B) > 16 GiB, όπου το M_bst δηλώνει το «μέγεθος μνήμης που απαιτείται για αυτή την υπολογιστική

ριπή», και τα A και B είναι τα ονόματα των διεργασιών. Το M_bst είναι συνήθως μικρότερο από το μέγεθος της εκχωρηθείσας μνήμης της διεργασίας, αφού:

- Τα ML Frameworks συνήθως υπερεκτιμούν την πραγματική ανάγκη σε GPU μνήμη και δεν συρρικνώνουν ποτέ τις εκχωρήσεις (allocations) τους. (το TF κανονικά εκχωρεί όλη τη μνήμη)
- δεν χρησιμοποιείται όλη η εκχωρηθείσα μνήμη σε κάθε υπολογιστική ριπή

Θα παράσχουμε τώρα ορισμένα στοιχεία σχετικά με τις GPU, τα οποία επηρεάζουν την εκτίμησή μας για το thrashing:

1. Δεν μπορούμε να ελέγξουμε/σταματήσουμε έναν πυρήνα που τρέχει στην GPU. Ένας πυρήνας (kernel) είναι μια συνάρτηση που εκτελείται στην GPU. Οι εκκινήσεις πυρήνα είναι ασύγχρονες και μια διεργασία μπορεί μόνο να υποβάλει ερώτημα σχετικά με την ολοκλήρωση της εργασίας που υποβλήθηκε στην GPU καλώντας την συνάρτηση cudaDeviceSynchronize().
2. Μια εφαρμογή ML ξεκινά χιλιάδες πυρήνες GPU, καθένας από τους οποίους είναι μικρός (μερικά χιλιοστά του δευτερολέπτου)

Θα αναφερθούμε σε αυτά τα 2 σημεία αργότερα, όταν παίρνουμε τις αποφάσεις μας για το πώς να χειριστούμε το thrashing.

4.4.2 dogbreed: Δημιουργώντας ένα σενάριο thrashing στην GPU

Τροποποιήσαμε το dogbreed-v2 Kale example [42], κάνοντάς το να χρησιμοποιεί ένα βαρύτερο μοντέλο ML (Resnet152) και αλλάξαμε το batch και image size, προσπαθώντας να δημιουργήσουμε ένα «βαρύ» Notebook. Πρέπει να έχουμε κατά νου ότι, όταν συγκρίνουμε τους χρόνους εκτέλεσης, δεν ξοδεύεται όλη η ώρα στον υπολογισμό GPU καθώς εκτελείται επίσης και ο κώδικας CPU. Η παράλληλη εκτέλεση δύο Notebook μειώνει το χρόνο που αφιερώνεται στον υπολογισμό της CPU και, σε γενικές γραμμές, οδηγεί σε μειωμένο συνολικό χρόνο εκτέλεσης, παρόλο που οι πυρήνες GPU από διαφορετικές διεργασίες δεν μπορούν να εκτελεστούν παράλληλα. Στην περίπτωση του dogbreed-resnet152, σχεδόν όλος ο χρόνος αφιερώνεται σε υπολογισμούς GPU, καθώς το κύριο έργο γίνεται μέσω επαναληπτικής εκπαίδευσης του μοντέλου Resnet. Μετατρέπουμε το Notebook σε python script για να διευκολύνουμε τη δοκιμή (αλλά εξακολουθούμε να αναφέρονται στο σενάριο ως "Notebook"), καθώς καταργούμε την ανάγκη χρήσης GUI. Θα αναφερόμαστε επίσης σε αυτό το τροποποιημένο dogbreed με το Resnet152 ως "dogbreed" στο εξής. Εδώ είναι το dogbreed-thrashing πρόγραμμά μας:

Listing 4.1: dogbreed-thrashing Python script

```
1 import os
2 import numpy as np
3 import tensorflow as tf
4 import matplotlib.pyplot as plt
5 from tensorflow.keras.preprocessing.image import ImageDataGenerator
6 from glob import glob
```

```
7  from PIL import Image
8  from PIL import ImageFile
9  import time
10
11 ImageFile.LOAD_TRUNCATED_IMAGES = True
12
13 LR = 6e-4
14 BATCH_SIZE = 64
15 NUMBER_OF_NODES = 256
16 EPOCHS = 5
17 IMG_SIZE = 224
18
19 def get_train_generator():
20     data_datagen = ImageDataGenerator(
21         rescale=1./255,
22         width_shift_range=.2,
23         height_shift_range=.2,
24         brightness_range=[0.5,1.5],
25         horizontal_flip=True
26     )
27     return data_datagen.flow_from_directory(
28         "dogImages/train/",
29         target_size=(IMG_SIZE, IMG_SIZE),
30         batch_size=BATCH_SIZE,
31     )
32
33 def get_valid_generator():
34     data_datagen = ImageDataGenerator(rescale=1./255)
35     return data_datagen.flow_from_directory(
36         "dogImages/valid/",
37         target_size=(IMG_SIZE, IMG_SIZE),
38         batch_size=BATCH_SIZE
39     )
40
41 def get_test_generator():
42     data_datagen = ImageDataGenerator(rescale=1./255)
43     return data_datagen.flow_from_directory(
44         "dogImages/test/",
45         target_size=(IMG_SIZE, IMG_SIZE),
46         batch_size=BATCH_SIZE
47     )
48
49 dog_classifier = tf.keras.applications.ResNet50V2(
```

```
    weights="imagenet",
    input_shape=(IMG_SIZE, IMG_SIZE, 3)
)

def is_dog(data):
    probs = dog_classifier.predict(data[0])
    preds = tf.argmax(probs, axis=1)
    return ((preds >= 151) & (preds <= 268))

train_generator = get_train_generator()
batch = train_generator.next()
predictions = is_dog(batch)

n_dog = np.sum(predictions)
dog_percentage = n_dog/BATCH_SIZE

print('{:.0%} of the files have a detected dog'.format(dog_percentage))

start_time = time.time()
resnet_body = tf.keras.applications.ResNet152V2(
    weights="imagenet",
    include_top=False,
    input_shape=(IMG_SIZE, IMG_SIZE, 3)
)
resnet_body.trainable = True
inputs = tf.keras.layers.Input(shape=(IMG_SIZE, IMG_SIZE, 3))
x = resnet_body(inputs, training=True)
x = tf.keras.layers.Flatten()(x)
outputs = tf.keras.layers.Dense(133, activation="softmax")(x)
resnet_model = tf.keras.Model(inputs, outputs)
resnet_model.compile(
    optimizer=tf.optimizers.Adam(learning_rate=LR),
    loss=tf.losses.categorical_crossentropy,
    metrics=["accuracy"]
)
train_generator = get_train_generator()
valid_generator = get_valid_generator()

resnet_model.fit(train_generator, epochs=EPOCHS,
    validation_data=valid_generator
)

test_generator = get_test_generator()
```

```
93 test_loss_resnet, test_accuracy_resnet = resnet_model.evaluate(test_generator)
94
95 print(f"The accuracy in the test set is {test_accuracy_resnet:.3f}.")
96 print(test_accuracy_resnet)
97 print("--- %s seconds ---" % (time.time() - start_time))
```

- Πρώτα εκτελέσαμε ένα αντίγραφο του Notebook και μετρήσαμε τον χρόνο εκτέλεσης και τη χρήση μνήμης GPU.
- Στη συνέχεια, εκτελέσαμε δύο αντίγραφα του Notebook παράλληλα, σε διάφορες συνθήκες.

Χρησιμοποιήσαμε τις παραμέτρους [EPOCHS = 5, IMG_SIZE = 224] για όλες τις δοκιμές. Ο αριθμός των Epochs δεν επηρεάζει το αποτύπωμα μνήμης, επηρεάζει μόνο τον αριθμό των επαναλήψεων και ο χρόνος εκτέλεσης ακολουθεί μια σχεδόν γραμμική σχέση με τον αριθμό των Epochs. Λαμβάνουμε το «GPU Memory Usage» από το εργαλείο nvidia-smi. Για τα σενάρια "Solo", μόνο μία διεργασία εκτελείται μόνη της. Δείχνουμε τα αποτελέσματα στον Πίνακα 4.2

Batch Size	(Peak) GPU Memory Usage (MiB)	Execution Time (s)
16	11196	752.8
32	11196	752.8
40	11196	750.7
64	15806	750.4

Table 4.2: *Solo dogbreed runs*

Δύο Notebook εκτελούμενα παράλληλα:

Η χρήση μνήμης GPU μεγιστοποιείται στα 16259 MiB για batch size > = 32 και αναπόφευκτα συμβαίνουν σφάλματα σελίδας. Ορίζουμε τον χρόνο εκτέλεσης ως τον συνολικό χρόνο ολοκλήρωσης, που είναι ο συνολικός χρόνος έως ότου ολοκληρωθούν όλες οι εργασίες. Δείχνουμε τα αποτελέσματά μας στον Πίνακα 4.3

Batch Size	(Peak) GPU Memory Usage (MiB)	Execution Time (s)
16	13679	1082
32	16259	1123
40	16259	2002
64	16259	11234

Table 4.3: *Δύο παράλληλες εκτελέσεις dogbreed*

- Στην περίπτωση που το batch size = 32, παρόλο που η συνολική ποσότητα μνήμης που

χρησιμοποιείται είναι 2 * 11196 = 22392 MiB, δεν υπάρχει σημαντική επιβράδυνση κατά την εκτέλεση δύο Notebook.

- Αυτό οφείλεται στο γεγονός ότι παρόλο που η συνολική εκχωρηθείσα μνήμη είναι > 16GiB, τα Working Sets για τα Notebooks είναι αρκετά μικρά, ώστε να μην προκαλούν μεγάλο αριθμό σφαλμάτων σελίδας.
- Σημειώνουμε ότι για Batch Size = 64, ο συνολικός χρόνος εκτέλεσης είναι ~11000 sec, δηλαδή 14,6 φορές ο χρόνος εκτέλεσης solo. Σε αυτήν την περίπτωση, μια τεράστια αναλογία χρόνου αφιερώθηκε στη διαχείριση σφαλμάτων σελίδας και ο χρόνος εκτέλεσης είναι πολύ μεγαλύτερος από ό,τι εάν οι υπολογισμοί GPU εκτελούνταν σειριακά.
- Επειδή, όπως σημειώσαμε παραπάνω, οι Εφαρμογές ML ξεκινούν χιλιάδες μικρούς πυρήνες, δεν χάνουμε παντελώς τον έλεγχο της GPU (όπως συνέβη σε προηγούμενο Κεφάλαιο, όπου ξεκινούσαμε έναν ενιαίο, τεράστιο πυρήνα, που επαναλαμβανόμενα προσπέλαυνε τις ίδιες περιοχές μνήμης ξανά και ξανά).
- Ως εκ τούτου, σε ένα σενάριο όπου λαμβάνει χώρα ένας μεγάλο αριθμός σφαλμάτων σελίδας, μπορούμε να ανακουφίσουμε την κατάσταση με την ελεγχόμενη εκκίνηση πυρήνων μιας εφαρμογής, επιτρέποντας ταυτόχρονα σε κάποια άλλη εφαρμογή να κάνει τους υπολογισμούς της χωρίς εμπόδια.

4.5 Anti-thrashing Μηχανισμός

Ο μηχανισμός anti-thrashing μας βασίζεται στην ιδέα ενός global GPU lock. Μόνο η διεργασία, η οποία κρατά το global lock μπορεί να δουλέψει στην GPU. Ένας δρομολογητής διαχειρίζεται το lock. Λαμβάνει αιτήματα από τους πελάτες-clients (διεργασίες), εκχωρεί το lock σε μια διεργασία για ένα κβάντο χρόνου και ανακτά το lock από τη διεργασία όταν παρέλθει το TQ.

4.5.1 Επισκόπηση

Παραθέτουμε την αριθμημένη λίστα βημάτων η οποία παρουσιάζει τη λειτουργία του anti-thrashing μηχανισμού μας, από την οπτική γωνία μιας νεο-δημιουργηθείσας διεργασίας χρήστη:
[Χρησιμοποιούμε μια διεργασία Jupyter Notebook ως την ερφαρμογή που εκτελεί ο χρήστης. Τα (a), (b) βήματα μπορούν να συμβούν σε οποιαδήποτε σειρά. Τα (i), (ii) βήματα συμβαίνουν με τη δηλωθείσα σειρά.]

1. Η διεργασία IPykernel (Jupyter backend) εκκινείται
2. Ο ld.so φορτώνει την libunified.so, αφού εμπεριέχεται στη μεταβλητή περιβάλλοντος LD_PRELOAD.
3. Η διεργασία καλεί την cuInit() (αυτή είναι πάντοτε η πρώτη κλήση συνάρτησης CUDA)
4. Η hooked εκδοχή της cuInit καλεί την συνάρτηση initializer μας
5. Η συνάρτηση initializer μας δημιουργεί τα νήματα A & B.

(α΄) Το νήμα Α εγγράφεται με τον daemon:
message_type="REGISTER"
data=[ID, client thread B socket path]
(β΄) Το νήμα client B ακούει στο client socket path (μοναδικώς προσδιορισθέν από το ID) για μηνύματα από τον daemon.

6. Ο daemon στέλνει ένα SCHED_ON ή SCHED_OFF μήνυμα στο client socket (νήμα B). Αυτό το σενάριο περιγράφει μια περίπτωση όπου ο scheduler είναι ενεργοποιημένος, οπότε ο daemon στέλνει ένα SCHED_ON μήνυμα.
7. Ο κώδικας χρήστη καλεί την cuLaunchKernel() (τα ίδια ισχύουν και για τις cuMemcpy() συναρτήσεις), για την οποία η libunified.so έχει εγκαταστήσει ένα hook (ας το ονομάσουμε cuLaunchKernel_hook)
8. Η cuLaunchKernel_hook() ελέγχει την τιμή του have_lock. Αν είναι ψευδής, τότε θέτει την need_lock = true. Έπειται μπλοκάρει έως ότου η have_lock γίνει αληθής.
   (i) Το client νήμα Α παρατηρεί ότι η need_lock είναι αληθής
   (ii) Στέλνει ένα REQ_LOCK μήνυμα στον δαεμον
   (iii) Στη συνέχεια αναμένει έως ότου λάβει ένα LOCK_OK μήνυμα από τον daemon.
   (iv) Το client νήμα B θέτει την have_lock σε True και την need_lock σε False.
9. Η cuLaunchKernel_hook() ξεμπλοκάρει αφού η μεταβλητή have_lock είναι True. Συνεχίζει με το κυρίως σώμα της συνάρτησης και υποβάλλει δουλειά στην GPU.
10. Όσο η μεταβλητή have_lock είναι αληθής, κάθε cuLaunchKernel και cuMemcpy κλήση εκτελείται απρόσκοπτη.
11. Το client νήμα B λαμβάνει μια DROP_LOCK εντολή από τον daemon. Θέτει την have_lock σε ψευδή (ώστε να μην υποβάλλεται νέα δουλειά από την διεργασία στην GPU) και καλεί την cuCtxSynchronize() (η οποία είναι blocking) ώστε να εξασφαλίσει πως οποιαδήποτε δουλειά έχει κατατεθεί προηγουμένως στην GPU είναι ολοκληρωμένη. Τελικά στέλνει μια LOCK_RELEASED απάντηση στον daemon.
12. **Τα βήματα 7-11** επαναλαμβάνονται μέχρι τον τερματισμό της διεργασίας.
13. Κατά την έξοδο του προγράμματος, ο client απελευθερώνει το lock σε περίπτωση που το κρατά ακόμα.

4.5.2 Σχεδιασμός

Παρουσιάζουμε τώρα τη λογική όλων των στοιχείων σε διαγράμματα ροής:

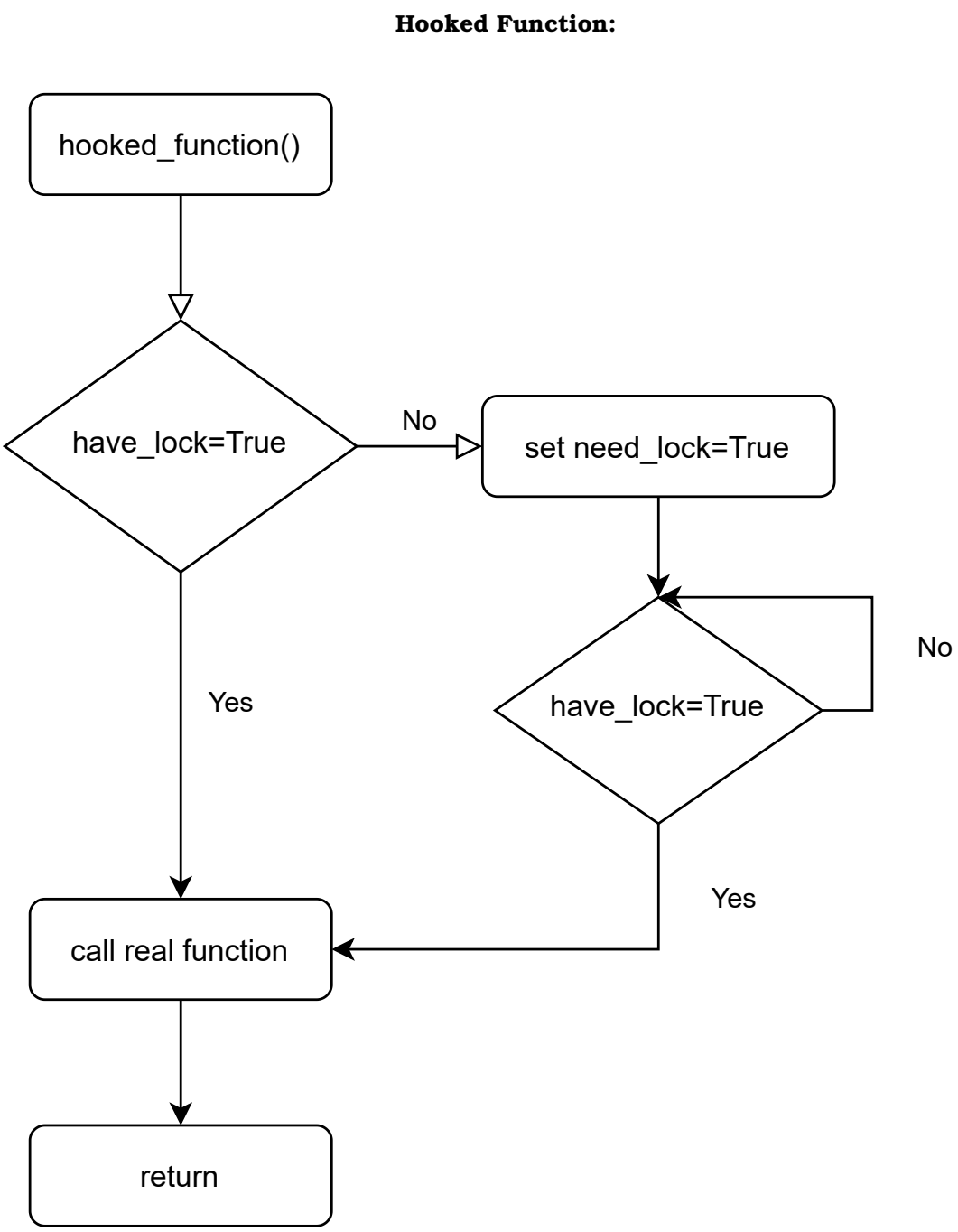

Figure 4.1: *Διάγραμμα ροής Hooked συνάρτησης*

Client:

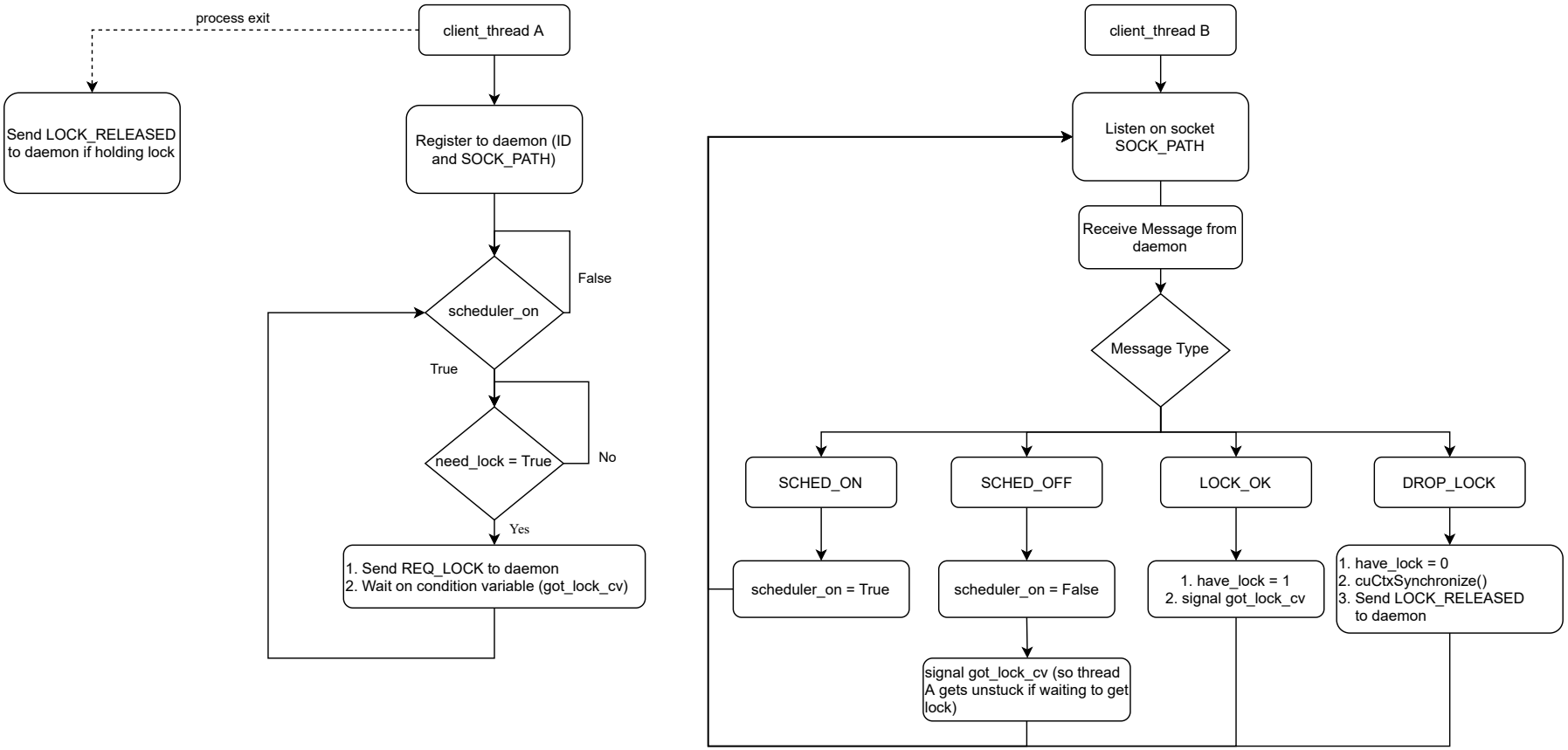

Figure 4.2: *Διάγραμμα ροής Client*

Δαεμον:

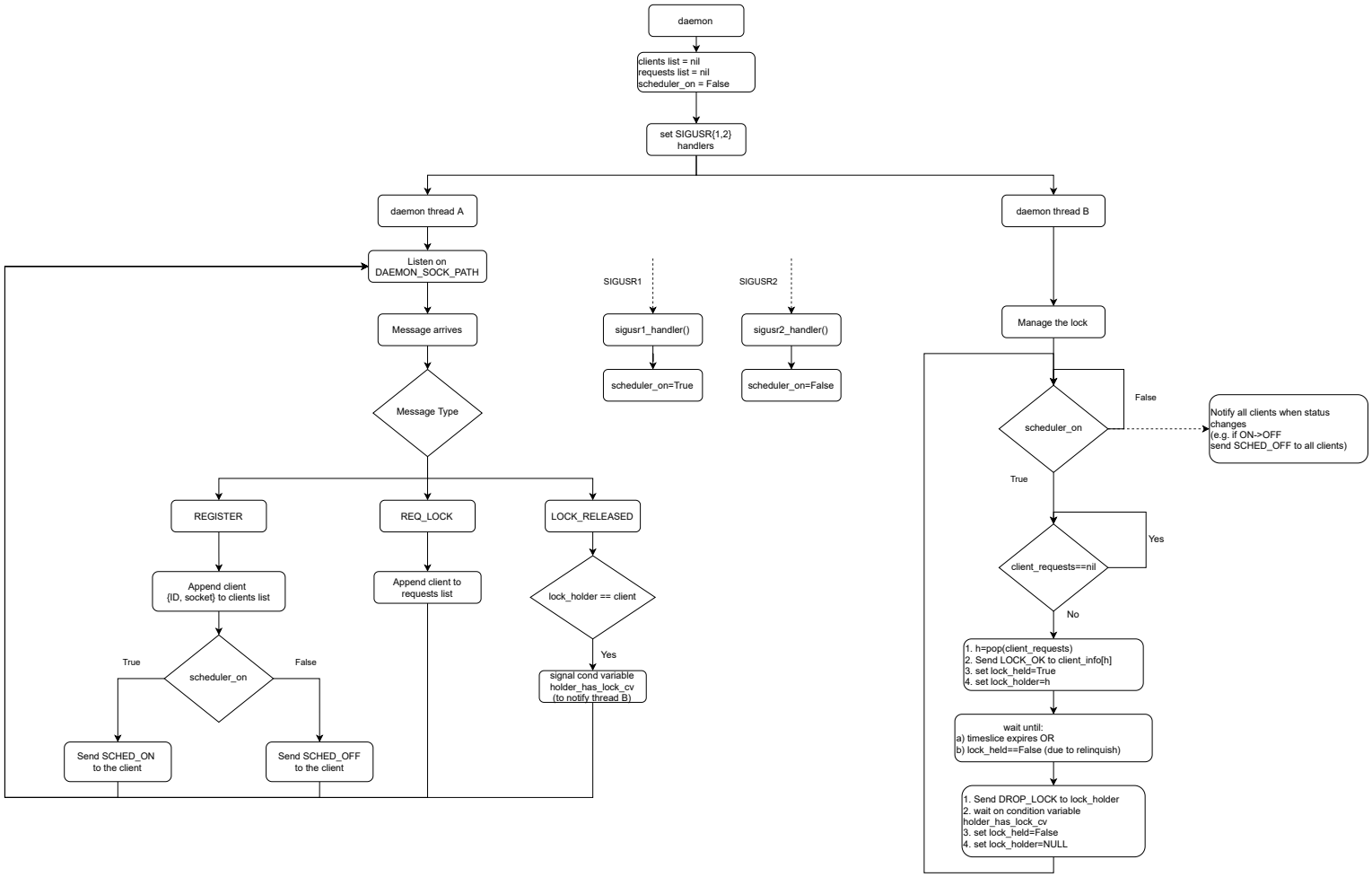

Figure 4.3: *Διάγραμμα ροής Daemon*

4.5.3 Υλοποίηση

Έχουμε ήδη υλοποιήσει την libunified.so ως κοινόχρηστη βιβλιοθήκη γραμμένη στη γλώσσα C. Επιλέξαμε να συνεχίσουμε σε C, καθώς προσφέρει καλύτερη απόδοση και λεπτομερή έλεγχο. Για την επικοινωνία μεταξύ client και daemon, επιλέξαμε να χρησιμοποιήσουμε UNIX Sockets (SOCK_STREAM - το ισοδύναμο του TCP) καθώς όλες οι διεργασίες βρίσκονται στον ίδιο φυσικό κόμβο. Ο anti-thrashing scheduler ανεπτύχθη ως ξεχωριστή διεργασία και όλη η επικοινωνία μεταξύ του scheduler και του client (νήματα) πραγματοποιείται μέσω UNIX Sockets που ζουν κάτω από το directory "/tmp/libunified". Μπορούμε να ενεργοποιήσουμε και να απενεργοποιήσουμε τον δρομολογητή στέλνοντας ένα σήμα USR1 και USR2 αντίστοιχα στη διεργασία daemon (scheduler). Αφού ενεργοποιηθεί, ο δρομολογητής ειδοποιεί αμέσως όλες τις υπό εκτέλεση διεργασίες και αρχίζει να επεξεργάζεται τις εισερχόμενες αιτήσεις για χρήση της GPU.

4.5.3.1 Έλεγχος του lock σε hooked συναρτήσεις

Έχουμε τροποποιήσει την υπάρχουσα βιβλιοθήκη μας. Οι παρεμβαλλόμενες συναρτήσεις τώρα προχωρούν με τη δουλειά τους μόνο εάν ο συγκεκριμένος client έχει το global lock. Εισάγουμε μια κλήση στην hook_check_lock() στην εκκίνηση του kernel και στις λειτουργίες αντιγραφής μνήμης προκειμένου να εξασφαλίσουμε πως η διεργασία μας κατέχει το lock.

Listing 4.2: Lock-checking logic for hooked functions

```
1  void hook_check_lock(void){
2
3          if (!(have_lock)) {
4                  need_lock = 1;
5  // Handle contention; many application threads may call CUDA functions
6                  pthread_mutex_lock(&need_lock_mutex);
7                  pthread_cond_signal(&need_lock_cv);
8                  pthread_mutex_unlock(&need_lock_mutex);
9                  pthread_mutex_lock(&have_lock_mutex);
10 // wait until we acquire the lock
11                 pthread_cond_wait(&received_lock_cv, &have_lock_mutex);
12                 pthread_mutex_unlock(&have_lock_mutex);
13         }
14
15         did_work = 1;
16         return ;
17 }
```

4.5.3.2 Επικοινωνία

Σχεδιάσαμε και υλοποιήσαμε το δικό μας πρωτόκολλο επικοινωνίας πάνω από Unix SOCK_STREAM sockets. Παρουσιάζουμε όλα τα πιθανά σενάρια επικοινωνίας στον Πίνακα 4.4:

Source	Destination	Message Type	data (/notes)
cA	dA	REGISTER	(ID, SOCKPATH)
cA	dA	REQ_LOCK	(ID)
cA	dA	LOCK_RELEASED	(ID)
dB	cB	LOCK_OK	-
dB	cB	DROP_LOCK	-
dA	cB	SCHED_ON	(immediately after REGISTER)
dA	cB	SCHED_OFF	(immediately after REGISTER)
dB	cB	SCHED_ON	(on scheduler status change)
dB	cB	SCHED_OFF	(on scheduler status change)

Table 4.4: *Πρωτόκολλο επικοινωνίας*

Χρησιμοποιούμε ένα κοινό packed [5] struct Message για κάθε ανταλλαγή πληροφορίας (μήνυμα).

Listing 4.3: Our Message struct

```
#define MSG_TYPE_LEN 16
#define MSG_DATA_LEN 40

struct message {
        uint64_t id;
        char type[MSG_TYPE_LEN];
        char data[MSG_DATA_LEN];
} __attribute__((__packed__));
```

4.5.4 Το κβάντο χρόνου του δρομολογητή

Το TQ δηλώνει το χρονικό διάστημα για το οποίο ο scheduler δίνει το lock σε έναν client. Μπορούμε να αλλάξουμε δυναμικά το TQ του δρομολογητή ανά πάσα στιγμή στέλνοντας ένα CHANGE_TQ μήνυμα στο daemon socket. Χρησιμοποιούμε ένα ξεχωριστό πρόγραμμα για αυτό το σκοπό, το οποίο παίρνει το νέο κβάντο χρόνου ως όρισμα και στέλνει το προαναφερθέν μήνυμα στον δαίμονα.

- Ένα μικρότερο κβάντο χρόνου σημαίνει περισσότερη διαδραστικότητα, καθώς οι πελάτες με μικρότερες ριπές GPU δεν θα πρέπει να περιμένουν πίσω από άλλους με

[5] https://gcc.gnu.org/onlinedocs/gcc-4.0.2/gcc/Type-Attributes.html

μεγαλύτερες και μπορούν να επιστρέψουν τα αποτελέσματα στον χρήστη πιο γρήγορα. Έτσι, ένα μικρότερο TQ ανακουφίζει το Head-of-line (HOL) μπλοκάρισμα.

- Ωστόσο, κάθε φορά που το lock αλλάζει τα χέρια, ο νέος κάτοχος πρέπει να φέρει εκ νέου τα δεδομένα του στην GPU (και να εκδιώξει δεδομένα από άλλον) πριν κάνει δουλειά. Αυτό οφείλεται στο γεγονός ότι εξετάζουμε σενάρια όπου η μνήμη GPU είναι oversubscribed, και τα Working Sets όλων των client δεν χωρούν στη μνήμη GPU. Ως εκ τούτου, ένα μικρότερο TQ οδηγεί σε μεγαλύτερο αριθμό σφαλμάτων σελίδας που μεταφράζεται σε ένα μεγαλύτερο χρόνο αλλαγής context καθώς και σε μεγαλύτερο συνολικό χρόνο ολοκλήρωσης (TCT).
- Ένα μεγαλύτερο TQ οδηγεί σε μικρότερο overhead για context switching (αφού συμβαίνει λιγότερες φορές) καθώς και συνολικό χρόνο εκτέλεσης, με το μειονέκτημά του να είναι ένα μειωμένο επίπεδο εμπειρίας και απόκρισης χρήστη.

4.6 Ενσωμάτωση με τον Kubernetes

Τώρα που έχουμε έτοιμο τον μηχανισμό μας, ήρθε η ώρα να τον ενσωματώσουμε με τον Kubernetes, έτσι ώστε οι χρήστες να μπορούν εύκολα να τον εγκαταστήσουν στη συστοιχία τους και να αποκομίσουν τα οφέλη που προσφέρουμε. Έχουμε ήδη αναλύσει πώς λειτουργεί το nvidia-device-plugin, που είναι ο de facto τρόπος χειρισμού GPU στον Kubernetes. Μια γρήγορη αναθεώρηση των διαγραμμάτων ακολουθίας θα είναι χρήσιμη για την κατανόηση του υπολοίπου αυτής της ενότητας. Θέλουμε να εμμείνουμε σε αυτό το σχέδιο και να το αλλάξουμε με έναν ελάχιστο τρόπο για να εκθέσουμε τον μηχανισμό μας. Η δημιουργία του δικού μας device-plugin θα μας επιτρέψει να διαφημίσουμε μια κοινόχρηστη GPU ως ένα extended resource, επιτρέποντας στους χρήστες να ζητήσουν αυτόν τον πόρο και να ενσωματώσουν εύκολα τον μηχανισμό μας στα Pods τους. Για το σκοπό αυτό, σχεδιάσαμε και υλοποιήσαμε το alexo-device-plugin.

Listing 4.4: alexo-device-plugin Daemonset YAML

```
1  apiVersion: apps/v1
2  kind: DaemonSet
3  metadata:
4    name: alexo-device-plugin
5    namespace: kube-system
6  spec:
7    template:
8      metadata:
9        labels:
10         name: alexo-device-plugin-ds
11     spec:
12       tolerations:
13       - key: alexo.com/shared-gpu
14         operator: Exists
15         effect: NoSchedule
16       priorityClassName: "system-node-critical"
17       containers:
```

```
18        - image: alexo-device-plugin:v0.0.2-ubuntu16.04-bd0fc6a
19          name: alexo-device-plugin-ctr
20          #args: ["--shared-gpus-per-gpu=3"]
21          #   env:
22          #- name: LIBUNIFIED_SRC
23          #  value: "/path/to/libunified/src"
24          #- name: LIBUNIFIED_DST
25          #  value: "/path/to/libunified/dst"
26          securityContext:
27            allowPrivilegeEscalation: false
28            capabilities:
29              drop: ["ALL"]
30          volumeMounts:
31            - name: device-plugin
32              mountPath: /var/lib/kubelet/device-plugins
33            - name: host-libunified
34              mountPath: /usr/lib/arr-gpushare
35          resources:
36            limits:
37              nvidia.com/gpu: 1
38        - image: alexo-device-plugin:v0.0.2-daemon-ubuntu16.04-bd0fc6a
39          name: alexo-device-plugin-gpu-scheduler
40          securityContext:
41            allowPrivilegeEscalation: false
42            capabilities:
43              drop: ["ALL"]
44          volumeMounts:
45            - name: host-scheduler-socks
46              mountPath: /tmp/libunified
47        volumes:
48          - name: device-plugin
49            hostPath:
50              path: /var/lib/kubelet/device-plugins
51          - name: host-libunified
52            hostPath:
53              path: /usr/lib/arr-gpushare
54              type: DirectoryOrCreate
55          - name: host-scheduler-socks
56            hostPath:
57              path: /tmp/libunified
58              type: DirectoryOrCreate
```

Το Daemonset [6] μας εμπεριέχει 2 Pods (το κάθε Pod εκτελεί ένα μόνο container):

- alexo-device-plugin: αυτό το Pod υλοποιεί την λογική του device-plugin, επικοινωνόντας με το kubelet και χειρίζόμενο τα Allocate requests για τον πόρο "alexo.com/shared-gpu". Επίσης εγκαθιστά την libunified.so στο directory /usr/lib/arr-gpushare.
- gpu-scheduler: Αυτό το Pod υλοποιεί τον anti-thrashing scheduler μας. Εκτελεί το daemon/scheduler container. Επικοινωνεί με τις εφαρμογές χρηστών μέσω UNIX Sockets τα οποία υπάρχουν στο directory /tmp/libunified. Δηλώνουμε ρητά το VolumeMount για αυτό το directory στο YAML αρχείο.

Στη συνέχεια παραθέτουμε μια αριθμημένη λίστα βημάτων που δείχνει τις ενέργειες που κάνει το device plugin μας.

1. Διαβάζει την τιμή της μεταβλητής NVIDIA_VISIBLE_DEVICES από το περιβάλλον του και τη χρησιμοποιεί μετέπειται ώστε να εκθέσει την GPU στα Pod των χρηστών. Αυτή είναι η πραγματική GPU, την οποία διαχειρίζεται το nvidia-device-plugin.
2. Εγκαθιστά την libunified.so στον κόμβο στην τοποθεσία /usr/lib/arr-gpushare κατά την εκκίνηση. (Ύστερα δίνει οδηγίες στο kubelet να κάνει mount το συγκεκριμένο μονοπάτι στα Pods των χρηστών που ζητούν τη shared-gpu μας.)
3. Διαφημίζει τον πόρο alexo.com/shared-gpu στο cluster (default 1000 συσκευές)
4. Απαντά σε Allocate Requests από το Kubelet, δίνοντάς του οδηγίες να:

   (α΄) Κάνει mount την shared library μας, libunified.so, μέσα στο file system του Pod. Η βιβλιοθήκη μας υλοποιεί την πλευρά του client του anti-thrashing μηχανισμού μας, όπως εξηγούμε στην προηγούμενη ενότητα.

   (β΄) Κάνει mount το directory των socket του μηχανισμού μας, ούτως ώστε οι διεργασίες να μπορούν να επικοινωνήσουν με το δρομολογητή.

   (γ΄) Θέτει τη μεταβλητή περιβάλλοντος LD_PRELOAD στο μονοπάτι /usr/lib/arr-gpushare/libunified.so

   (δ΄) Θέτει τη μεταβλητή περιβάλλοντος NVIDIA_VISIBLE_DEVICES ίση με το GPU-UUID που αποκτάμε μέσω του alexo-device-plugin Pod, το οποίο ζητά μια "nvidia.com/gpu" προκειμένου να εκθέσουμε την GPU (σύμφωνα με τις αρχές λειτουργίας του nvidia-container-runtime) στο Pod του χρήστη το οποίο ζητά "alexo.com/shared-gpu"

Οι χρήστες μπορούν τώρα να εγκαταστήσουν το alexo-device-plugin με μια εντολή kubectl apply: `$ kubectl apply -f alexo-device-plugin.yaml`

[6] Ένα DaemonSet εξασφαλίζει πως όλοι (οι μερικοί) κόμβοι εκτελούν ένα αντίγραφο κάποιου Pod.

Κεφάλαιο 5

Αξιολόγηση

Ο μηχανισμός μας επιτρέπει νέα σενάρια, στα οποία πολλές εφαρμογές χρηστών μπορούν να μοιράζονται την ίδια GPU χωρίς περιορισμούς στη μνήμη. Σε αυτό το κεφάλαιο θα συγκρίνουμε τον μηχανισμό μας με την τελευταία λέξη της τεχνολογίας. Για τις νέες περιπτώσεις που υποστηρίζονται μόνο μέσω του μηχανισμού μας, μπορούμε να τον συγκρίνουμε μόνο με τον εαυτό του εκτός από το baseline της σειριακής εκτέλεσης.

Οι διαδραστικές εφαρμογές (όπως τα Jupyter Notebooks) δεν έχουν πεπερασμένο χρόνο εκτέλεσης (ούτε έχουν προκαθορισμένη δομή · ο χρήστης μπορεί να αλλάξει δυναμικά τα κελιά κώδικα). Ως εκ τούτου, δεν υπάρχει απλός τρόπος για τον καθορισμό ποσοτικών μετρήσεων για τα χαρακτηριστικά εκτέλεσης τους, ειδικά ως προς το χρόνο ολοκλήρωσης. Για να αξιολογήσουμε το μηχανισμό μας, θα βασιστούμε σε μη διαδραστικά (conventional) προγράμματα. Εξετάζουμε μόνο σενάρια στα οποία εκτελούνται παράλληλα δύο διεργασίες, υποβάλλοντας δουλειά στην GPU. Το σενάριο που θα εξετάσουμε είναι το χειρότερο (πιο υπολογιστικά εντατικό) για την περίπτωση διαδραστικού φόρτου εργασίας και παρέχει ένα σαφές μέσο ποσοτικοποίησης και αξιολόγησης των χαρακτηριστικών απόδοσης του μηχανισμού μας. Η συμπεριφορά της απόδοσης κάτω από αληθινά διαδραστικά φορτία μπορεί να είναι μόνο καλύτερη από αυτή που θα μετρήσουμε σε αυτό το κεφάλαιο.

Είμαστε πολύ ικανοποιημένοι αφού ο μηχανισμός μας αποδίδει εξαιρετικά καλά ακόμη και για μη διαδραστικά (conventional) φορτία, στα οποία δεν υπάρχουν περίοδοι αδράνειας για τις διεργασίες. Αυτό σημαίνει ότι ο μηχανισμός μας παρέχει έναν τρόπο μεγιστοποίησης της αξιοποίησης της GPU ακόμη και τέτοιου είδους δουλειές και μπορεί να φανεί χρήσιμος σε ένα πολύ ευρύτερο φάσμα εφαρμογών.

5.1 Εργαλεία, Μεθοδολογία και Περιβάλλον

Πραγματοποιήσαμε τα πειράματά μας στο Google Cloud Platform. Χρησιμοποιήσαμε έναν μοναδικό κόμβο, εξοπλισμένο με CPU 16 πυρήνων (Intel Xeon 2,3 GHz - Haswell), 104 GB μνήμης RAM και GPU NVIDIA Tesla P100 (GP100GL) με μνήμη 16 GB. Η σουίτα αξιολόγησής μας περιλαμβάνει 4 ML εφαρμογές γραμμένες σε Tensorflow που περιέχουν ένα συνδυασμό μερών CPU και GPU. Το τμήμα CPU κάθε εφαρμογής εκπαιδεύει ένα μοντέλο ResNet152v2 [43] για λίγα μόνο βήματα (καθώς η εκπαίδευση στην CPU είναι μια τάξη μεγέθους πιο αργή από ό,τι στην GPU). Το τμήμα GPU εκπαιδεύει επίσης ένα μοντέλο ResNet152v2 για 5 Epochs.

Αρχικά δημιουργήσαμε 2 βασικές εφαρμογές, με τη μεγαλύτερη διαφορά τους να είναι η χρήση μνήμης GPU. Η εφαρμογή small χρησιμοποιεί περίπου 7 GB μνήμης GPU καθ όλη τη διάρκεια της εκτέλεσης, γεγονός που καθιστά δυνατή τη συστέγαση δύο small εφαρμογών στην ίδια GPU υπό μηχανισμό τύπου Kubeshare και καθιστά δυνατή τη σύγκριση απόδοσής του με τον μηχανισμό μας. Η εφαρμογή big χρησιμοποιεί περίπου 15 GB μνήμης GPU. Αυτό σημαίνει ότι οποιαδήποτε άλλη υπάρχουσα προσέγγιση εκτός από τη δική μας δεν μπορεί να τρέξει δύο big εφαρμογές στην ίδια GPU παράλληλα. Αυτή η υπόθεση προκαλεί επίσης thrashing κάτω από τον μηχανισμό μας, οπότε και ελέγχει τα όριά του υπό μέγιστη πίεση.

Δημιουργούμε δύο περαιτέρω εφαρμογές από κάθε μια εκ των big και small βάσεων. Διαφοροποιούμε το μείγμα υπολογιστικών τμημάτων CPU / GPU (αναφορικά με το χρόνο εκτέλεσης) για να δοκιμάσουμε και να εξετάσουμε μια ποικιλία σεναρίων εργασίας, είτε πιο βαριά ως προς τη χρήση GPU είτε πιο ισορροπημένα. Επιλέγουμε τις αναλογίες 90/10 (90 GPU, 10 CPU) και 50/50. Ορίζουμε τους λόγους ως προς το χρόνο εκτέλεσης όταν μια διαδικασία εκτελείται «μόνη» στο σύστημα.

Για το υπόλοιπο αυτής της αξιολόγησης θα θεωρήσουμε ταυτόσημους τους όρους Working Set Size (WSS) και Memory Usage όταν μιλάμε για GPU. Το μέγεθος του Working Set είναι ένας πιο λεπτός όρος και σε σενάρια ML αντιπροσωπεύει τη μνήμη GPU που απαιτείται για κάθε επαναληπτικό βήμα training. Επομένως, μπορεί να είναι μικρότερο από τη μέγιστη χρήση μνήμης της εφαρμογής, καθώς, όπως είπαμε, τα πλαίσια ML υπερεκτιμούν την πραγματική ζήτηση και δεν μειώνουν ποτέ τη χρήση της μνήμης τους. Αυτό που περιμένουμε είναι ότι όταν το άθροισμα των WSS δύο εφαρμογών υπερβαίνει τη φυσική μνήμη GPU, θα εμφανιστεί thrashing και με αυτό κατά νου κατασκευάζουμε τις big εφαρμογές. Μπορούν να υπάρξουν περιπτώσεις όπου το thrashing δεν συμβαίνει, ακόμη και όταν το συνολικό άθροισμα των εκχωρήσεων μνήμης μεταξύ των διεργασιών είναι μεγαλύτερο από τη μνήμη της GPU. Αυτό συμβαίνει επειδή το άθροισμα του WSS εξακολουθεί να χωράει στη μνήμη GPU. Δείχνουμε το τελικό μας σύνολο 4 εφαρμογών στον Πίνακα 5.1

name	GPU Working Set Size	GPU/CPU ratio
small_90	7.2 GB	90/10
small_50	7.2 GB	50/50
big_90	15.3 GB	90/10
big_50	15.3 GB	50/50

Table 5.1: *Τα προγράμματα αξιολόγησής μας*

Εκτελέσαμε όλες τις εμφανίσεις των πειραμάτων μας ως container σε ένα Debian host. Για να μετρήσουμε τους χρόνους εκτέλεσης του Kubeshare, υποβάλαμε τα container σαν (Share)Pods στον Kubernetes και καθορίσαμε το αίτημα μνήμης GPU στο πεδίο "Annotations", σύμφωνα με τις επίσημες οδηγίες repository του Kubeshare [44]. Δεν μπορούμε να εκτελέσουμε τα big προγράμματα μέσω του Kubeshare, καθώς δεν μπορεί να κάνει oversubscribe τη μνήμη GPU.

5.2 Αποτελέσματα

5.2.1 Επισκόπηση

Δείχνουμε τις μετρήσεις μας στον Πίνακα 5.2. Αρχικά εκτελέσαμε καθεμία από τις 4 εφαρμογές μας (solo) σε stock (NVIDIA) mode, ύστερα με την libunified με το anti-thrashing απενεργοποιημένο και τέλος με τον μηχανισμό anti-thrashing. Αυτό μοντελοποιεί τον τρόπο με τον οποίο οι ριπές GPU μιας διαδραστικής εφαρμογής θα συμπεριφέρονταν όταν αυτή έτρεχε μόνη της και δίνει μια εκτίμηση της γενικής επιβάρυνσης του μηχανισμού μας, εξαλείφοντας τα σφάλματα σελίδας ή τις παρεμβολές από εφαρμογές που βρίσκονται σε συστεγασμό.

Στη συνέχεια, καθορίσαμε το baseline για τον χρόνο εκτέλεσης 2 αντιγράφων κάθε προγράμματος που εκτελούνται παράλληλα πολλαπλασιάζοντας τον solo (stock) χρόνο επί 2 και λαμβάνοντας έτσι τον σειριακό (serial) χρόνο εκτέλεσης. Αυτός υποδηλώνει το χρόνο που θα χρειαζόταν για την εκτέλεση δύο αντιγράφων ενός προγράμματος το ένα μετά το άλλο. Στη συνέχεια, δοκιμάσαμε το Kubeshare για τις small εφαρμογές (όπως αναφέραμε προηγουμένως, δεν μπορεί να εκτελέσει δύο αντίγραφα των big εφαρμογών, καθώς το συνολικό WSS υπερβαίνει τη φυσική μνήμη GPU). Δοκιμάσαμε επίσης την libunified χωρίς anti-thrashing και τελικά λάβαμε μετρήσεις με τον anti-thrashing δρομολογητή ενεργοποιημένο και για διάφορες τιμές του κβάντου χρόνου (TQ). Λάβετε υπόψη ότι το client μέρος του μηχανισμού μας απελευθερώνει αυτόματα το lock πίσω στον δρομολογητή μετά από 5 δευτερόλεπτα χωρίς υποβολή εργασίας στη GPU.

Method	**small_50 (s)**	**small_90 (s)**	**big_50 (s)**	**big_90 (s)**
solo (stock)	1318	692	1383	719
solo (libunified no anti-thrashing)	1332	711	1385	724
solo (libunified w/ anti-thrashing)(60)	1339	712	1390	734
serial (2*solo-stock)	**2636**	**1384**	**2766**	**1438**
2 instances in Parallel for all below				
Kubeshare	1724	1078	---	---
libunified (no anti-thrashing)	1772	1128	*11757 (thrashing)*	*11434 (thrashing)*
anti-thrashing(1000)	2053	1361	2043	1380
anti-thrashing(500)	2053	1363	2070	1400
anti-thrashing(400)	2030	1352	2081	1405
anti-thrashing(300)	2010	1354	2085	1418
anti-thrashing(200)	2007	1360	2092	1423
anti-thrashing(100)	2010	1351	2100	1435
anti-thrashing(60)	2020	1348	2122	1468
anti-thrashing(50)	1995	1351	2145	1473
anti-thrashing(40)	1993	1343	2156	1485
anti-thrashing(30)	2017	1366	2170	1521
anti-thrashing(20)	2007	1360	2204	1583
anti-thrashing(15)	1983	1341	2305	1668
anti-thrashing(10)	2009	1360	2465	1821
anti-thrashing(5)	1982	1334	3496	2788

Table 5.2: *Μετρήσεις χρόνων εκτέλεσης για τα "small" και "big"*

Επίσης, απεικονίζουμε τα αποτελέσματα του παραπάνω πίνακα στα σχήματα 5.1, 5.2, 5.3. Σημειώνουμε το κβάντο χρόνου (TQ) του scheduler anti-thrashing σε παρένθεση στις ετικέτες του σχήματος. Στις επόμενες ενότητες θα σχολιάσουμε τις μετρήσεις μας και θα δηλώσουμε τις παρατηρήσεις μας.

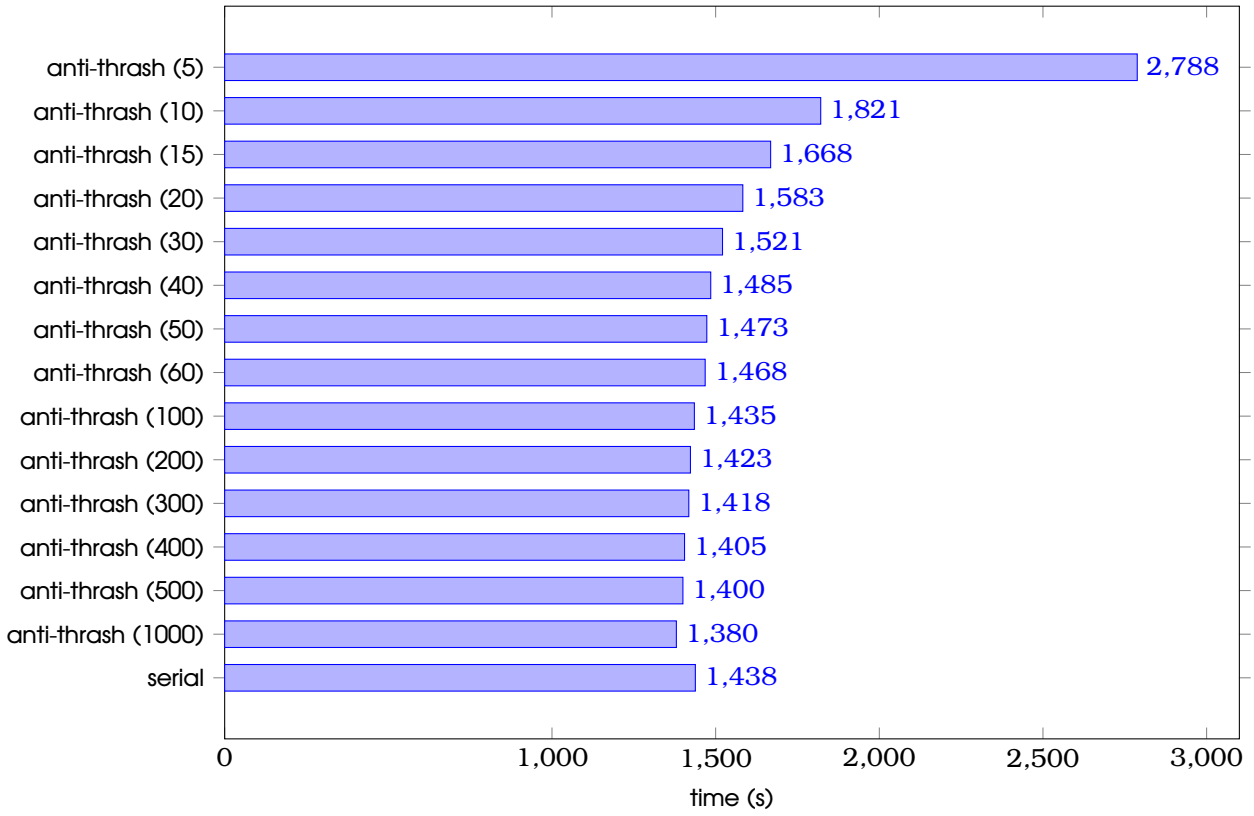

Figure 5.1: *Χρόνοι εκτέλεσης για το big_90*

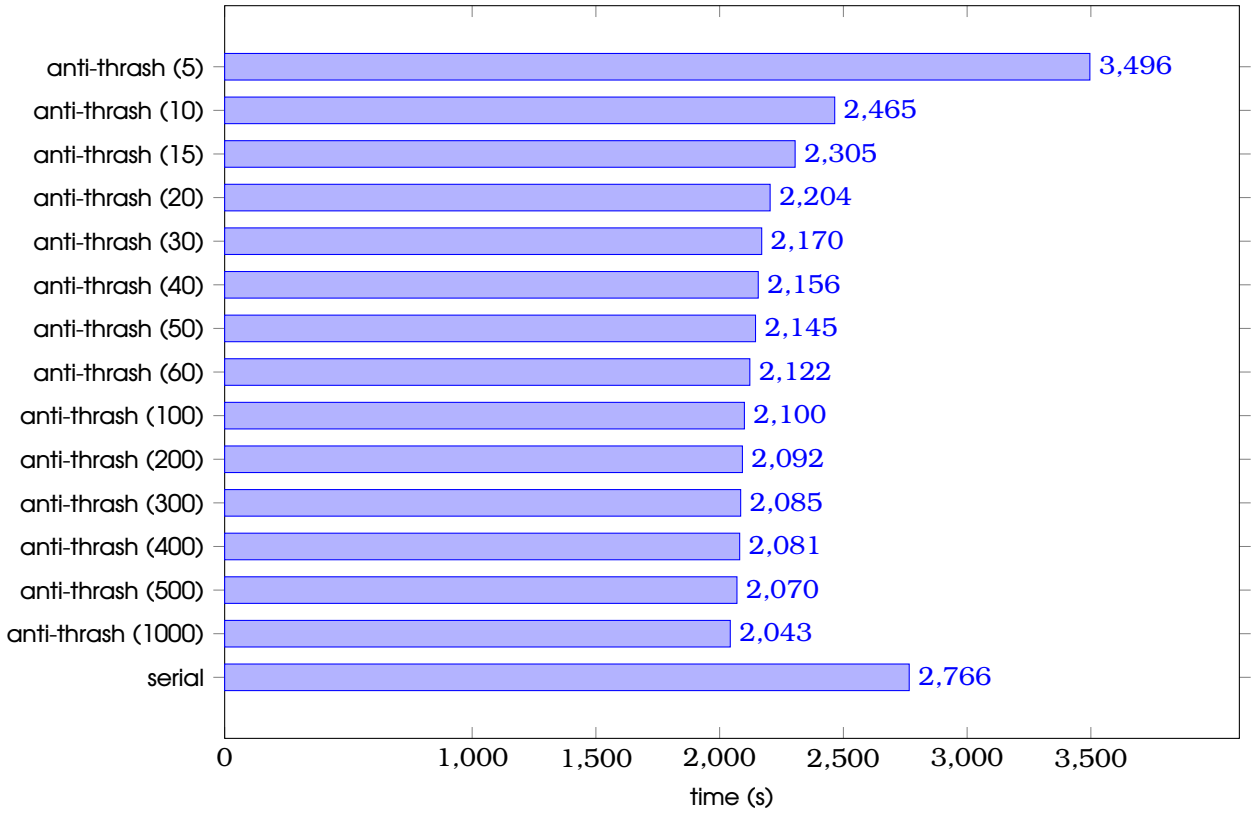

Figure 5.2: *Χρόνοι εκτέλεσης για το big_50*

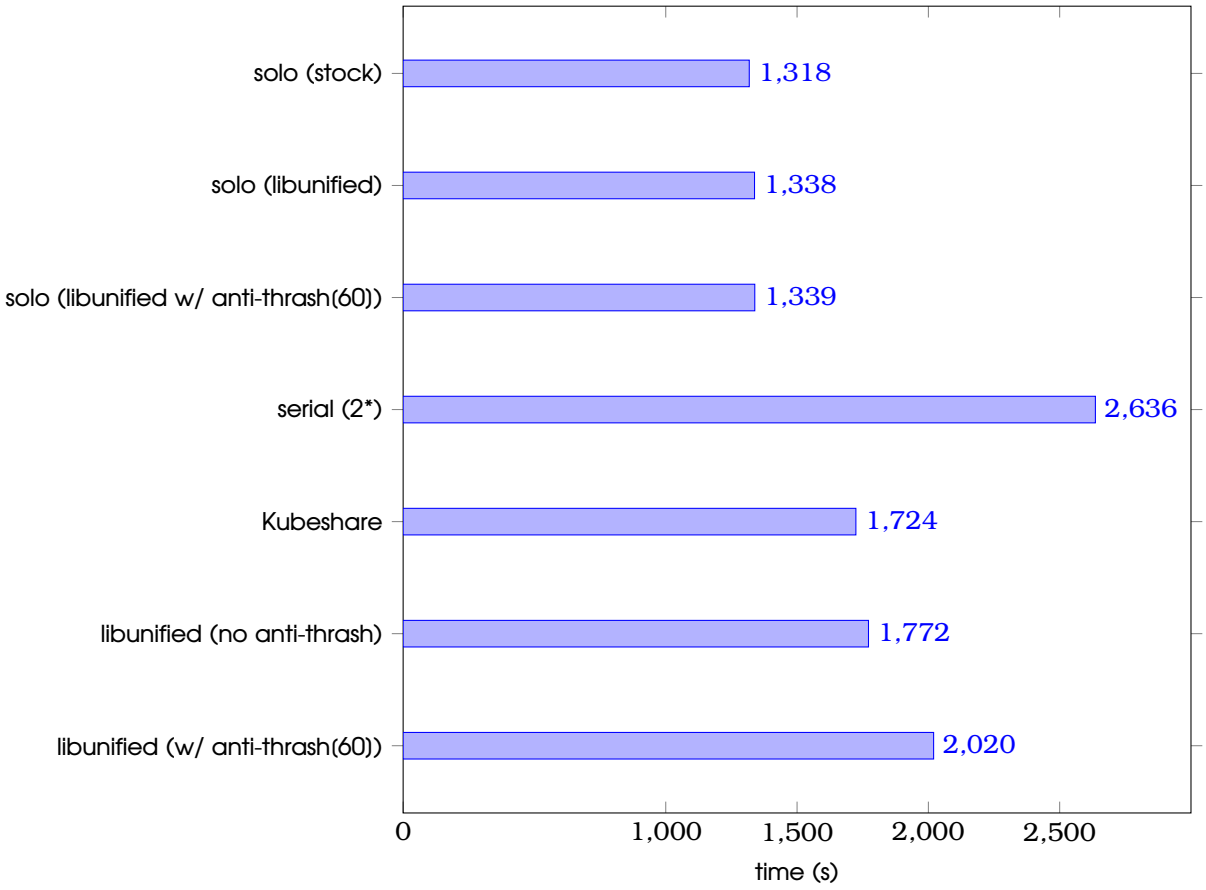

Figure 5.3: *Χρόνοι εκτέλεσης για το small_50*

5.2.2 Μικρό WSS

Για τις περιπτώσεις small_50 και small_90 δεν συμβαίνουν σφάλματα σελίδας καθόλη την παράλληλη εκτέλεση των δύο πανομοιότυπων προγραμμάτων (εξάλλου θα ήταν fatal στις non-Unified περιπτώσεις). Για την περίπτωση 50/50, είμαστε σε θέση να επιτύχουμε σημαντική επιτάχυνση, καθώς τα μέρη της CPU τρέχουν εντελώς παράλληλα. Πρέπει να σημειώσουμε εδώ ότι όταν δύο διαφορετικές διεργασίες (contexts) υποβάλλουν εργασία στην GPU, η GPU δρομολογεί τους πυρήνες των context με μη δημοσιευμένο τρόπο. Δεν μπορεί να εκτελέσει ταυτόχρονα πυρήνες από δύο διαφορετικά contexts, ωστόσο χειρίζεται τη δρομολόγηση (εναλλαγή context) με πιο αποτελεσματικό τρόπο από ό,τι ο scheduler anti-thrashing. Ως εκ τούτου, η πιο αποτελεσματική μέθοδος σε αυτήν την περίπτωση είναι το Kubeshare. Ωστόσο, κατά τη χρήση του Kubeshare πρέπει να δηλώσουμε τα όρια μνήμης της διαδικασίας εκ των προτέρων, κάτι που συνήθως δεν είναι εφικτό ούτε και βολικό. Ο χρόνος εκτέλεσης της libunified (χωρίς ενεργοποιημένο το anti-thrashing) είναι παραπλήσιος, ο ελαφρά αυξημένος χρόνος του προερχόμενος από το γεγονός ότι χρησιμοποιεί την Unified Memory (χωρίς βέβαια σφάλματα σελίδας). Εκτιμήσαμε ότι αυτό το γενικό overhead είναι περίπου ˜1 % σε προηγούμενο κεφάλαιο. Η λιγότερο αποδοτική μέθοδος είναι η libunified (anti-thrash), στην οποία είναι ενεργοποιημένος ο scheduler anti-thrashing και κάθε διεργασία χρησιμοποιεί την GPU για ένα χρονικό διάστημα. Αυτό είναι αναμενόμενο, καθώς σε αυτές τις small εφαρμογές δεν υπάρχει thrashing και ως εκ τούτου δεν χρειάζεται να σειριοποιήσουμε την

εργασία στην GPU από τις διεργασίες. Ο default black-box (driver-level) δρομολογητής της NVIDIA (ο οποίος χρησιμοποιείται στις περιπτώσεις Kubeshare και libunified) χειρίζεται τους υποβληθέντες GPU kernels με πιο αποτελεσματικό τρόπο από ό,τι η συντηρητική προσέγγιση του δρομολογητή μας. Λάβετε υπόψη, ωστόσο, ότι αυτή η κατηγορία small εφαρμογών είναι σπάνια και επίσης ότι το Kubeshare υποστηρίζει τη συστέγαση μόνο όταν το άθροισμα των εκχωρήσεων μνήμης είναι μικρότερο από τη μνήμη GPU (στην περίπτωση αυτή δεν θα μπορούσε να υποστηρίξει ένα τρίτο αντίγραφο που εκτελείται παράλληλα ενώ η libunified το υποστηρίζει).

5.2.3 Μεγάλο WSS

Το Kubeshare δεν μπορεί να υποστηρίξει την εκτέλεση δύο big εφαρμογών παράλληλα, καθώς το συνολικό άθροισμα WSS ισούται με περίπου 30 GB και ξεπερνά κατά πολύ το μέγεθος της φυσικής μνήμης GPU. Κατά την εκτέλεση δύο εφαρμογών big παράλληλα, εμφανίζεται thrashing στην περίπτωση της απλής libunified, εξ ου και οι εξαιρετικά μεγάλοι χρόνοι εκτέλεσης στον Πίνακα 5.2. Ωστόσο, με ενεργοποιημένο τον μηχανισμό anti-thrashing, μπορούμε να επιτύχουμε χρόνους πολύ κάτω από τον σειριακό χρόνο εκτέλεσης για την περίπτωση 50/50 και συγκρίσιμο με τους σειριακούς χρόνους εκτέλεσης για την περίπτωση 90/10. Έχοντας μεγαλύτερο Time Quantum ελαχιστοποιείται ο συνολικός χρόνος ολοκλήρωσης (TCT), καθώς εμφανίζονται λιγότερα swaps (Σφάλματα σελίδας), δηλαδή το lock αλλάζει χέρια λιγότερο συχνά. Όταν το TQ γίνεται πολύ μικρό (< 10 δευτερόλεπτα), οι χρόνοι αρχίζουν να αυξάνονται απότομα, καθώς ο χρόνος εκτέλεσης κυριαρχείται πάλι από τα Σφάλματα σελίδας, καθώς η κάθε εφαρμογή δεν έχει αρκετό χρόνο για να εκτελέσει τους ουσιαστικούς υπολογισμούς της. Ο μηχανισμός μας λειτουργεί άριστα και καταφέρνει να εκτελέσει δύο εφαρμογές παράλληλα, με WSS > 30 GB, σε χρόνους κάτω του σειριακού. Καμία υπάρχουσα λύση δεν επιτρέπει την εκτέλεση εφαρμογών των οποίων το άθροισμα WSS υπερβαίνει τη φυσική μνήμη GPU, πόσο μάλλον να το επιτύχει με χρόνο κάτω από τον σειριακό.

Κεφάλαιο 6

Συμπεράσματα

Το ταξίδι μας φτάνει στο τέλος του. Θα επαναλάβουμε τις συνεισφορές μας, δείχνοντας ποια είναι η νέα κατάσταση όσον αφορά την κοινή χρήση GPU στον Kubernetes, αυτή τη φορά λαμβάνοντας υπόψη και την προσέγγισή μας. Τέλος, θα κλείσουμε αυτήν την εργασία αναφέροντας τις μελλοντικές επεκτάσεις (Future Work) που μπορούν να γίνουν επί του μηχανισμού μας.

6.1 Ένα νέο state of the art

Σχεδιάσαμε, υλοποιήσαμε και αξιολογήσαμε το alexo-device-plugin. Ας επανεξετάσουμε τι προσφέρει:

- κάθε εφαρμογή μπορεί **να χρησιμοποιήσει ολόκληρη τη μνήμη της GPU**.
- Πολλές **εφαρμογές μπορούν να εκτελέσουν ταυτόχρονα στην ίδια GPU χωρίς όρια μνήμης**. Ο μόνος περιοριστικός παράγοντας είναι η μνήμη της CPU (RAM).
- Είναι **διαφανής για τις εφαρμογές χρήστη**· καμία τροποποίηση δεν απαιτείται στον κώδικα χρήστη ή στα frameworks που χρησιμοποιεί.
- Έχει **ελάχιστο overhead** όταν μια εφαρμογή εκτελείται μόνη της στην GPU.
- **Αποτρέπει αποτελεσματικά το thrashing**, μεγιστοποιώντας την αξιοποίηση της GPU και ελαχιστοποιώντας τον συνολικό χρόνο εκτέλεσης ακόμα και όταν τα Working Sets είναι κατά πολύ μεγαλύτερα του μεγέθους μνήμης της GPU.
- Μπορεί να εγκατασταθεί στον Kubernetes με **μια μόνο** kubectl apply **εντολή**.

Όπως είδαμε στο προηγούμενο κεφάλαιο, παρόλο που ο αρχικός μας στόχος ήταν μόνο να επιτρέψουμε την κοινή χρήση GPU για διαδραστικές εργασίες ML για την αύξηση της αξιοποίησης της GPU. και είναι η περίπτωση που η εφαρμογή μας ευδοκιμεί απολύτως, **ο μηχανισμός μας μπορεί να χρησιμοποιηθεί εξίσου καλά σε μη διαδραστικές εργασίες**. Παρέχουμε μια ποιοτική σύγκριση με το state-of-the-art στον Πίνακα 6.1

[*]: Δεν μπορούμε να ορίσουμε ένα «εγγυημένο» κλάσμα για την Unified Memory με τον ίδιο τρόπο που μπορούμε για τις κανονικές εκχωρήσεις μνήμης CUDA. Δεν επιβάλουμε σκληρό όριο στο μέγεθος των εκχωρήσεων μνήμης από μια διεργασία, επομένως υπό αυτή την έννοια δεν προσφέρουμε «εγγυημένα» κλάσματα μνήμης.

Mechanism	Can co-locate	co-location constraint	guaranteed memory fraction	K8s Integration
nvidia-device-plugin	No	-	yes (whole mem)	Seamless
Aliyun Scheduler Extender	Yes	Physical GPU memory	No	Average
Kubeshare	Yes	Physical GPU memory	Yes	Hard
alexo-device-plugin	Yes	RAM + GPU Memory	No*	Seamless

Table 6.1: *Σύγκριση με το state of the art*

6.2 Μελλοντικές Επεκτάσεις

Μπορούμε επιτέλους να ολοκληρώσουμε αυτήν την διπλωματική με τις μελλοντικές ερευνητικές κατευθύνσεις του μηχανισμού μας. Σκοπεύουμε να τις επιδιώξουμε ενεργά τους επόμενους μήνες/χρόνια.

- Χρήση και μελέτη επί πραγματικών φορτίων εργασίας σε περιβάλλον παραγωγής.
- Επέκταση του alexo-device-plugin για να υποστηρίζει πολλαπλές GPU ανά κόμβο.
- Δημιουργία ευριστικών για την αυτόματη ενεργοποίηση/απενεργοποίηση του μηχανισμού anti-thrashing αντί της χειροκίνητης ενεργοποίησης. Μια ευριστική μπορεί να είναι ο όγκος των μεταφορών PCIe σε ένα sliding time window.
- Υποστήριξη μετεγκατάστασης εφαρμογών από μία GPU σε άλλη (εντός του ίδιου κόμβου αρχικά, μεταξύ διαφορετικών κόμβων σε μεταγενέστερο στάδιο). Η χρήση της Unified Memory μας διευκολύνει σε αυτό το κομμάτι, καθώς όλες οι GPU σε έναν κόμβο έχουν κοινό χώρο διευθύνσεων CUDA. Παρόλα αυτά, δεν είναι καθόλου τετριμμένη υπόθεση.

Chapter 7

Introduction

In this first chapter, we outline the scope of our work. We provide a brief overview of the problem at hand and its implications. Then, we go over the existing open-source as well as commercial approaches, highlighting their offerings and their drawbacks. Moving on, we illustrate the gap that we are attempting to fill and give a 10000 foot view of our solution. Finally, we present the structure of this thesis.

7.1 Motivation

7.1.1 GPUs and Machine Learning

For the entirety of this thesis, we deal with NVIDIA Graphics Processors. NVIDIA holds the overwhelming market share when it comes to general-purpose GPU computing. GPUs are massively parallel co-processors, which are used alongside the main CPU platform in a system as powerful accelerators. GPUs were introduced more than 2 decades ago to enable real-time rendering, the main target being video-game graphics. Today, GPUs are ubiquitous, and are found everywhere, from Smartphones, laptops, data centers to supercomputers. The initial driving force behind GPU innovation was the demand for ever more pristine graphics [5]. While graphics acceleration maintains its place as the main incentive, GPU usage in non-graphics computing is becoming increasingly more prevalent. This approach, known as general purpose computation on GPUs (GPGPU), is being increasingly adopted in HPC (High Performance Computing) applications. A prominent example of this is the growing use of Graphics Processing Units in the development and deployment of machine learning systems [6]. But what is it that makes GPUs such a good fit for powering ML models?

The training phase of a neural network is an extremely resource-intensive task. Inputs are fed to the network and processed in its hidden layers using weights. These weights are then adjusted by the network's logic to lead to subsequently better predictions. Finally, the model outputs a prediction on the input. Both of these operations mostly comprise matrix multiplications. Matrix multiplications follow ridiculously parallel computational patterns, as most linear algebra operations do. As GPUs are massively parallel by nature (they have a large number of simple cores), they can perform many of these straightforward linear algebra operations simultaneously. Additionally, computations in machine learning need to handle huge amounts of data — this makes the GPU's large memory bandwidth

most suitable. As such, when they are available, GPUs are utilized to speed up the ML development process.

7.1.2 An ML Scientist's Workflow

In order to get to the root cause of our problem, we must first understand the stages that an ML Scientist goes through to get their models into production. While we choose ML developers as the main use-case of our study and its starting point, our final product has a much broader application spectrum.

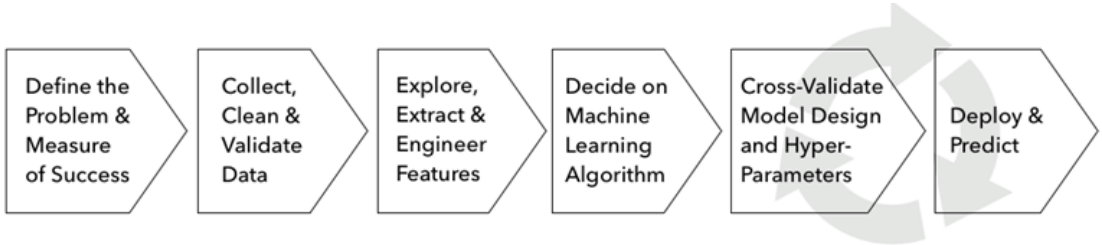

Figure 7.1: *Workflow of an ML Engineer*

Here is a numbered list of steps that outline the workflow of a Machine Learning scientist:

1. **Find a problem to solve**: Self explanatory, also mandatory.
2. **Acquire data**: This is one of the most critical parts of the process, and data are one of the most valuable digital commodities.
3. **Preprocess data and variables**: This part comprises loading data in computer language structures, filtering available data (tossing away erroneous entries, outliers) and finally handling categorical variables; deriving a means to assign numerical values to them. Jupyter Notebooks are a very popular development environment for this step.
4. **Create a baseline ML model**: Sketch out a rugged model and validate that the high-level assumptions the scientist made on the problem hold. This step also comprises short and repetitive bursts of training and evaluation, which the scientist as a means to decide whether to commit to a more robust implementation or go back to the drawing board.
5. **Develop the full model:**
   (a) In an IDE/VSCode/PyCharm as a Python module
   (b) Go all the way in Jupyter Notebook interactively (this is what we are interested in)
6. **Train the model**: This step is the most computationally intensive one and is usually non-interactive.
7. **Serve Inference requests for the model**: Use the final trained model to make real predictions

For Steps 4,5 and occasionally 6, most ML Scientists work with Jupyter Notebooks ([7]). Jupyter Notebook applications are long-running (keep this in mind for later) interactive processes that provide an excellent environment for tackling these kinds of problems, which involve iterating over a code, making changes, re-evaluating, until the result is satisfactory. This is the main target audience of our thesis' contributions, whose lives we are trying to improve.

Having all the above in mind, we can now move on to our problem statement.

7.2 Problem Statement

7.2.1 The user(s)

ML scientists doing interactive development on a Jupyter Notebook; we will approach this issue from their point of view.

7.2.2 The current state in upstream Kubernetes

The de facto way of handling GPUs in a Kubernetes cluster is via NVIDIA's device-plugin mechanism [8]. When a user wants to operate (via a Pod/container) on a GPU on Kubernetes, they specify `nvidia.com/gpu: "X"` (where X is an integer) in their request (the Pod object they create) and the device plugin automatically exposes the GPU(s) when their container starts executing. The assignment of GPUs to Pods is exclusive, meaning that if a GPU is bound to a Pod, it remains unavailable to the rest of the cluster while that Pod is running. We thoroughly analyze how Kubernetes handles GPUs in Section 8.6.3.

7.2.2.1 Reasons behind exclusive assignment

The developers (NVIDIA employees) behind the `nvidia-device-plugin` chose the approach of exclusive assignment of GPUs to Pods because:

- Processes using the same GPU (running on the same node) compete for the same pool of physical GPU memory for their allocations: Currently, GPUs cannot handle page-faults for normal (non-Unified) allocations. We cover this in Section 8.1.3.6. Since processes can grow and shrink their GPU memory dynamically and the allocation requests are handled in a first-come-first-served order, there can be scenarios where one or both processes can fail with Out-of-Memory (OOM) errors. While each process cannot interfere with the GPU memory contents of another as each context owns distinct page tables, NVIDIA provides no way to isolate the GPU memory volume usage (as in size) between many processes.
- The scheduling of GPU commands (kernel launches, memory copies) issued by different processes as well as the context switching are both handled in an undisclosed manner by the device driver. There is no documented way of triggering a context switch on the GPU. As such, QoS guarantees such as time-bound handling of requests cannot be satisfied when working on a shared GPU.

7.2.3 The Problem

GPUs are underutilized; there is no option to co-locate tasks (Kubernetes Pods) on the same GPU. This is especially wasteful for interactive Machine Learning development tasks, which are our main focus, as well as inference.

Here are some instances of users voicing the problem:

- "GPUs cannot be shared - GPUs must be shared" from the Jupyter forums [9]
- "Is sharing GPU to multiple containers feasible?" Github Issue on the Kubernetes Repository [10]

7.2.3.1 Aggravating Factors

ML Development Tasks (Notebooks):

- do not perform a pre-determined amount of work (as a training task would); their execution times cannot be calculated/bound.
- are long-running tasks with GPU usage patterns that are characterized by bursts and generally have large idle periods (during code refactoring/debugging/developer breaks)

While the problem of exclusive assignment of GPUs can be solved trivially (for example by tweaking device-plugin to advertise a greater number of `nvidia.com/gpu` than physical GPUs, the **core issue is that of managing the friction between co-located tasks** (how 2+ processes on the same node behave, irrespective of Kubernetes) and that is hard to solve.

7.2.3.2 A non-Kubernetes scenario

To illustrate the problem at hand, let's examine a real-world scenario in which the current handling of GPU (memory) is inefficient and leads to resource waste.

Assume users A and B both have access to a common server with 1 GPU. When user A is doing interactive (ML) development that uses a GPU on a Jupyter Notebook, then the GPU memory they (specifically the ML framework) allocate will not be freed until the IPython kernel [1] (the Notebook's backend process) is terminated. This means that if they run some cells and afterwards decide to tweak their code, or even go for a coffee/walk, the allocated GPU memory will remain unavailable to other users/notebooks/processes of any kind. Consequently, when B decides to launch a Notebook and do work as well, they will have access only to the remainder of GPU memory.

ML Frameworks' handling of GPU Memory

ML frameworks prefer to handle GPU memory through internal sub-allocators. As such, they request GPU memory in large chunks and usually overshoot real "demand". Additionally, Tensorflow [11] by default allocates all GPU memory. This behavior can be optionally altered in order to allow GPU memory usage to grow as needed. Still that usage will never shrink. If an ML job's "actual" need fluctuates from 500MiB to 2.5 GiB, then back to 500MiB, then the allocated GPU memory will be 2.5 GiB until the TF process terminates.

Thus they will be faced with the following:

- Many of their ML models won't fit in GPU memory / won't even launch [OOM].
- They won't have access to the same environment as when training, so they won't be able to objectively assess their models.

7.2.3.3 Characteristics of a complete solution

Consequently, any complete solution must comprise:

- a **mechanism** to isolate the GPU usage between processes on the same node and facilitate sharing
- a K8s-specific way of exposing that mechanism (via custom resources, request formatting) to the users of the cluster. In one word, a K8s **integration**.

[1] An IPython kernel is a Python interpreter process that acts as the backend for Jupyter Notebooks. The notebook packs user code in JSON format, forwards it to the kernel which in turns evaluates it and finally the result is sent back to the notebook to be displayed.

7.3 Overview and limitations of existing approaches

7.3.1 Summary of existing approaches

There exists a plethora of approaches towards GPU Sharing. The authors of Kubeshare [12] provide a way to isolate the GPU (memory) usage between processes by partitioning GPU memory and assigning a fraction to each process. As a result, each process has a guaranteed quota of GPU memory, which it cannot exceed. It also offers a Kubernetes integration, via the use of a custom scheduler and special Annotations on the user Pods. However, using Kubeshare's method for assigning GPUs requires major changes in all involved components (it's not only a matter of requesting a different resource instead of "nvidia.com/gpu") and is therefore an unrealistic scenario. Aliyun (Alibaba Cloud) Scheduler Extender [13] provides a way to circumvent the exclusive assignment of GPUs to Pods by offering an alternative device plugin (users request "aliyun.com/gpu"). Users can also state GPU memory requests for their Pod, which are used for bin-packing by the scheduler. However, post-scheduling, Aliyun offers no way to enforce these GPU memory requests. The processes still interfere, potentially destructively (OOM), with each other in the same way two GPU processes do in a non-Kubernetes scenario. Alibaba provides a proprietary way to enforce these memory limits when using their cloud infrastructure via cGPU [14], in a manner similar to Kubeshare (hard memory limit specified before launch). A variety of other commercial solutions ([15], [16], [17], [18]) exist. Amazon Elastic Inference takes a completely different approach and virtualizes the GPU at the application level. It is restricted only to inference scenarios and requires use of proprietary Amazon software, having limited application to our use-case. All commercial solutions that are pertinent to our use-case follow the same scheme as Kubeshare, providing a way to fractionally assign GPU memory to different processes, to enable GPU Sharing. To use this effectively however, users have to know the precise memory usage of their application beforehand, which is not the case when developing ML models.

7.3.2 Shortcomings of existing approaches

All of the existing solutions, either open-source or commercial/proprietary promise to offer the very same things. As such, every single one of them has the same shortcomings, especially with regards to our use-case.

- **Hard limit on GPU memory for each process:**
  - All of the existing approaches impose a hard limit on GPU memory that has to be specified during submission time. This contradicts the interactive and ever-changing nature of a Jupyter Notebook job, whose memory usage is usually impossible to estimate beforehand.
  - Additionally, the hard limit restricts the flexibility of the user's workflow with regard to testing out incrementally larger models. They will either have to request a large amount of memory that will mostly remain unused, or be faced with a potential OOM error.

- **Number of co-located processes limited by physical GPU memory (no oversubscription)**
  - As we mentioned, for normal CUDA memory allocations, the sum of them across all processes must be smalle than physical GPU memory capacity. As a result, under all existing schemes, the number of processes that can be co-located is limited by this memory capacity. The sum of memory fractions assigned to the co-located applications cannot exceed memory capacity or OOM errors will occur. There is no option for memory oversubscription.
- **GPU Underutilization persists:**
  - While assignment is no longer exclusive between Pods and GPUs, the new resource that is used for binpacking is GPU memory. This new criterion, coupled with the nature of interactive workloads (burst of computation -> idle -> next burst) still allows for underutilization scenarios. If a GPU has 4 GB of memory and a job A requests 2.5, then a job B requesting 2 won't be scheduled regardless of A's actual memory usage. This is acceptable in cases of computationally intensive tasks, such as ML Training. However, for interactive tasks, job A's utilization of the GPU will generally be very low throughout it's non-a-priori-calculable duration and the GPU as a whole remains underutilized.

7.4 Our approach (alexo-device-plugin)

As we highlighted before, the core issue does not lie in K8s scheduling, but in what comes after. Our goal is a mechanism that enables 2+ interactive ML development processes (Notebooks) to co-exist on a shared GPU without a hard memory limit and then expose it in K8s. We first developed a mechanism that tackles the problem (`libunified`) and then integrated it with Kubernetes (`alexo-device-plugin`).

Here is a numbered list of steps that describes the operation of `alexo-device-plugin` from a user's point of view in Kubernetes:

1. User submits a Pod requesting `alexo.com/shared-gpu`
2. `alexo-device-plugin` exposes the GPU into the container and injects our `libunified.so` shared library.
3. User application starts execution. Our shared library converts all normal memory allocations (`cudaMalloc`) to their Unified Memory counterpart (`cudaMallocManaged`). It also registers the application with the anti-thrashing scheduler.
4. The application can use all GPU memory and run alongside other co-located applications (belonging to the same or other users) on the same GPU.
5. As long as the GPU bursts of the co-located applications don't overlap or the working set sizes don't exceed physical GPU memory, there is no possibility of thrashing, so the GPU automatically handles execution requests in a FCFS manner.
6. If GPU bursts of different applications overlap and the sum of WSS exceeds physical GPU capacity, there can be potential thrashing, leading to overall slowdown.
7. If the anti-thrashing mechanism is enabled, it serializes usage of the GPU among applications, giving sole access to the GPU for a configurable time quantum (TQ) to an application that requests it. In that case, an application may have to wait for its turn. In the case that the GPU work of a burst is done before the TQ elapses, it automatically "hands" the GPU back to the scheduler.
8. All user applications run seamlessly to completion. The utilization of the GPU is maximized, at the cost of potential minor queuing delays. In the case where fairness is not an issue (e.g. one user submits many jobs and wants to minimize the total completion time), a large TQ for the scheduler minimizes the context switching overhead and total completion time and maximizes GPU utilization.

7.4.1 What it offers

- each user (Notebook) can **use the whole GPU memory**.
- **many Notebooks can execute on the same GPU concurrently without hard memory limits**. The only limiting factor is host memory (RAM) size and subjectively (per-case) the acceptable latency of cell executions.
- Our mechanism is **transparent to the user application** (no modification to user code or frameworks)

7.4.2 Goals

Given that:

- Hard limit of GPU memory is restrictive for the users
- Underutilization still persists with all state-of-the-art solutions, especially if Notebooks request large amounts of memory (very few can be co-located in a Kubeshare-style mechanism)

We want to minimize GPU idle time by having many Notebooks active and multiplexing their use of GPU Memory, while also catering to preventing thrashing.

7.4.3 Why it hasn't been done before

- No public study exists of Unified Memory behavior with 2+ co-located processes; we discovered this behavior (LRU-style eviction across different processes). Finding this led us to envision the point below.
- No study exists on transparently converting all Memory allocations to Unified Memory and the stability/performance of that conversion (we are the first to study the behavior of applications under this scheme)
- Controlling thrashing in such a novel execution scheme is not trivial and requires special attention.

7.4.4 Limitations

- In the case where the GPU memory contents of the process are not resident when the GPU executes one of its commands, there is some delay (PCIe transfers) for making the data it uses resident on the GPU again.
- ~1% flat execution time overhead due to using Unified Memory
- Potential head-of-line (HOL) blocking when we use a large time quantum for the scheduler

7.5 Thesis Structure

The rest of the document is organized as follows:

- In Chapter 8 we provide the necessary theoretical background: We start from the fundamentals of GPU computing, gradually making our way to understanding the way GPUs are used in contemporary container-based environments, in our case Kubernetes.
- In Chapter 9 we present the existing approaches towards GPU Sharing.
- In Chapter 10 we present our study on Unified Memory behavior in multi-process scenarios. To the best of our knowledge, this is the first comprehensive study of Unified Memory under multi-tenancy.

Unified memory allocations: GPU kernels can page-fault. Pages are evicted to Host RAM if GPU physical memory is full. For normal memory allocations each byte allocated is backed by a physical byte in GPU memory.

- In Chapter 11 we analyze the design and implementation of our mechanism.
- In Chapter 12 we evaluate our work, comparing it to the previous state and finally briefly comment on the criteria behind choosing a scheduler TQ. We also hint at other potential use-cases, apart from our main focus (interactive jobs).
- In Chapter 13 we provide a summary of our contributions as well as possible future work directions.

Chapter 8

Background

In this chapter we provide the necessary theoretical background for the rest of thesis.

8.1 GPU Basics

We present an overview of the foundations behind GPU computing, analyze the architectural characteristics of a modern GPU and explain the key concepts in CUDA programming. While we focus on NVIDIA Graphics Card and the CUDA API in the present, most of the information provided generally applies to any GPU.

8.1.1 Introduction to GPU computing

8.1.1.1 Parallelism

In today's world, apart from the traditional computational problems (fluid mechanics, weather forecast, biochemistry), there is an ever increasing need to sift through large amounts of data in a quick manner. With massively parallel processors being widely commercially available, the field of parallelism is thriving. Every single entity in the digital landscape, be it a large corporation or a small research laboratory, employs parallel computing to satisfy its needs. As such, parallelism has become a driving force of architectural and systems design.

There are two fundamental types of parallelism in applications:

- **Task parallelism:** There are many tasks or functions that can be executed independently and largely in parallel. Task parallelism handles distributing these functions across multiple workers (cores).

- **Data parallelism:** There are many data items that can be concurrently operated on. Data-parallel processing partitions data across threads, with each thread working on a portion of the data.

GPU programming is especially well-suited to address problems (dense linear algebra, FFT...) that can be expressed as *data-parallel computations*.

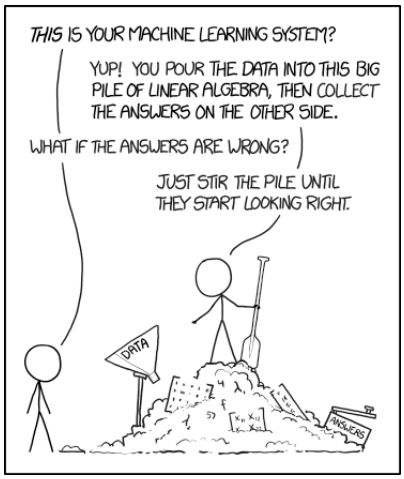

Figure 8.1: *Machine learning in a nutshell [1]*

There is a plethora of applications that process large data sets and can use a data-parallel model to speed up their computations.

8.1.1.2 Flynn's Taxonomy

In the context of parallel programming, we can classify the computing architectures into four different classes. Michael J. Flynn [45] introduced this categorization in 1966 and has been in use ever since.

1. **S**ingle **I**nstruction,**S**ingle **D**ata (SISD): A traditional uniprocessor machine. Being a sequential computer, it does not exploit task nor data parallelism.

2. **S**ingle **I**nstruction,**M**ultiple **D**ata (SIMD): A multiprocessor machine capable of executing the same instruction on multiple cores, operating on different data streams. GPUs are to an extent examples of SIMD systems.

3. **M**ulitple **I**nstructions,**S**ingle **D**ata (MISD): A MISD computing system is a multiprocessor machine capable of executing different instructions on multiple cores, all operating on the same data set. This architecture is uncommon and is generally used for fault tolerance.

4. **M**ulitple **I**nstructions,**M**ultiple **D**ata (MIMD): A MIMD computing system consists of multiple autonomous processors simultaneously executing different instructions on different data. MIMD architectures include multicore processors and distributed systems.

8.1.1.3 GPU vs CPU

Historically, GPUs were developed to be used as graphics accelerators. Starting from the mid 2000s, programmers started utilizing GPUs for general-purpose computing. The use of a GPU to perform computation in applications traditionally performed by CPUs is called **General-purpose computing on graphics processing units** [19]. At that time however, GPU manufacturers did not expose a programming API to assist them in this

endeavor. Initially, programmers "overloaded" the use of the Graphics APIs (e.g. `DirectX`) to perform linear algebra calculations, such as matrix multiplications. Using a GPU for computations was programmatically complex, error-prone and generally not a pleasant experience. The introduction of NVIDIA's CUDA (2007) finally gave developers the necessary tools to harness the Graphics Processing Unit's computing power and spawned a new field in parallel computing; GPGPU. GPUs are now general-purpose, powerful, fully programmable task and data parallel processors. They are particularly suited to solving massively parallel computational problems.

A GPU provides a much higher instruction throughput and memory bandwidth than the CPU within a *similar price and power envelope*. As such, certain applications leverage these higher capabilities to run faster on the GPU than on the CPU.

This difference in capabilities between the GPU and the CPU exists because they are **designed with different goals in mind**. While the CPU is designed to excel at executing a sequence of operations (a thread) as fast as possible and can execute a few tens of these threads in parallel, the GPU is designed to excel at executing **thousands** of them in parallel (amortizing the slower single-thread performance to achieve greater throughput).

Devoting more transistors to data processing, e.g. floating-point computations, is beneficial for highly parallel computations; the GPU can hide memory access latencies with computation, instead of relying on large data caches and complex flow control (branch prediction, out of order execution) to avoid long memory access latencies, both of which are expensive in terms of transistors.

In general, an application has a **mix of parallel and sequential parts**, so systems are designed with a mix of GPUs and CPUs in order to *maximize overall performance*.

A side-by-side comparison of the silicon allocated to each component illustrates the above differences.

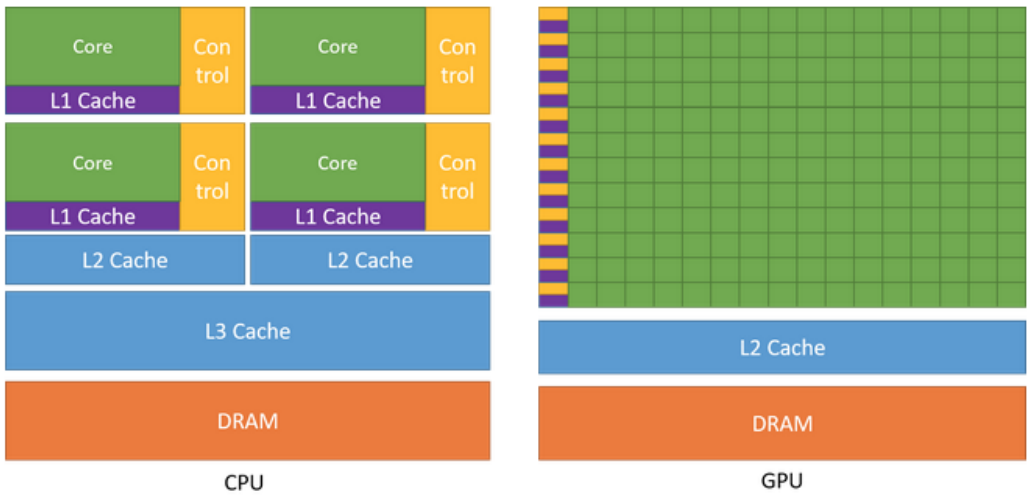

Figure 8.2: *GPU vs CPU transistor allocation*

8.1.2 Hardware Basics

Note

Currently, vendors (especially NVIDIA) hide the details of GPU architectures for their own reasons. We are going to approach GPU hardware from an application developer's perspective - given publicly documented information - and not from a GPU architect/driver developer's (reverse-engineered).

8.1.2.1 Architecture

A GPU is currently not a standalone platform but a co-processor to a CPU. Therefore, GPUs must operate in conjunction with a CPU-based host through a PCI-Express bus. That is why, in GPU computing terms, the CPU is called the *host* and the GPU is called the *device*.

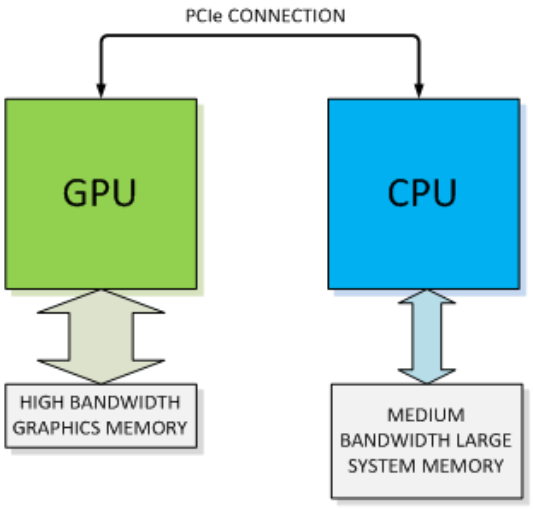

Figure 8.3: *GPU and CPU connection*

CUDA's simplified view of the GPU includes the following:

- A **host interface** that connects the GPU to the PCI Express bus. It reads GPU commands (memory copies, kernel launches) and dispatches them to the appropriate units.
- 1-2 **copy engines**; to overlap device-host and host-device transfers with kernel execution
- A **DRAM interface** that connects the GPU to its device memory
- A number of TPCs or GPCs (texture processing clusters or graphics processing clusters) each of which contains caches and a number of **streaming multiprocessors (SMs)**. These vary by GPU generation.

The following figure shows the architecture of a Fermi-generation Graphics Processing Unit:

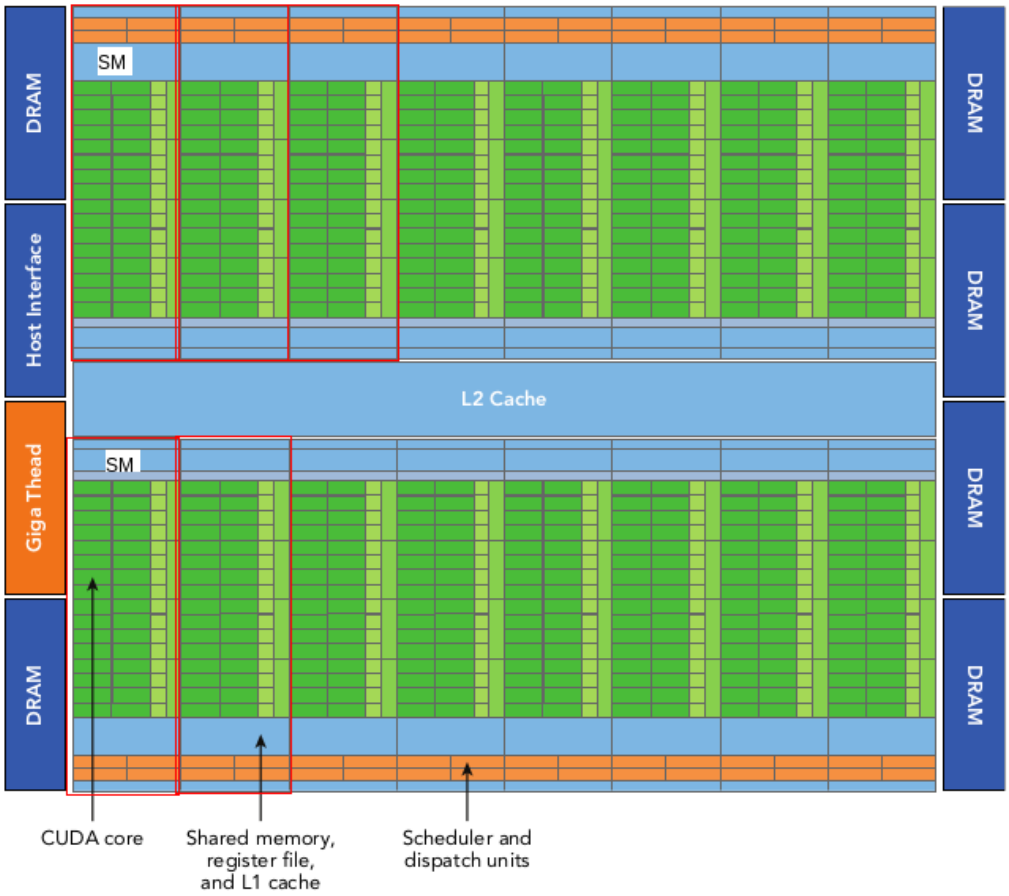

Figure 8.4: *Fermi GPU Architecture*

8.1.2.2 The Streaming Multiprocessor

The GPU architecture is built around an array Streaming Multiprocessors (SMs). GPU hardware parallelism is achieved through the replication of this architectural building block.

The key components of an SM are:

- CUDA Cores [ALU] [on GV100: 64 x FP32, 32 x FP64, 64 x INT32, 8 mixed-precision Tensor Cores]
- Shared Memory/L1 Cache
- Register File [order of 64K 32-bit registers]
- Load/Store Units
- Special Function Units [log/exp, sin/cos, and rcp/rsqrt]
- Warp Schedulers [quickly switch contexts between threads and issue instructions to warps that are ready to execute]

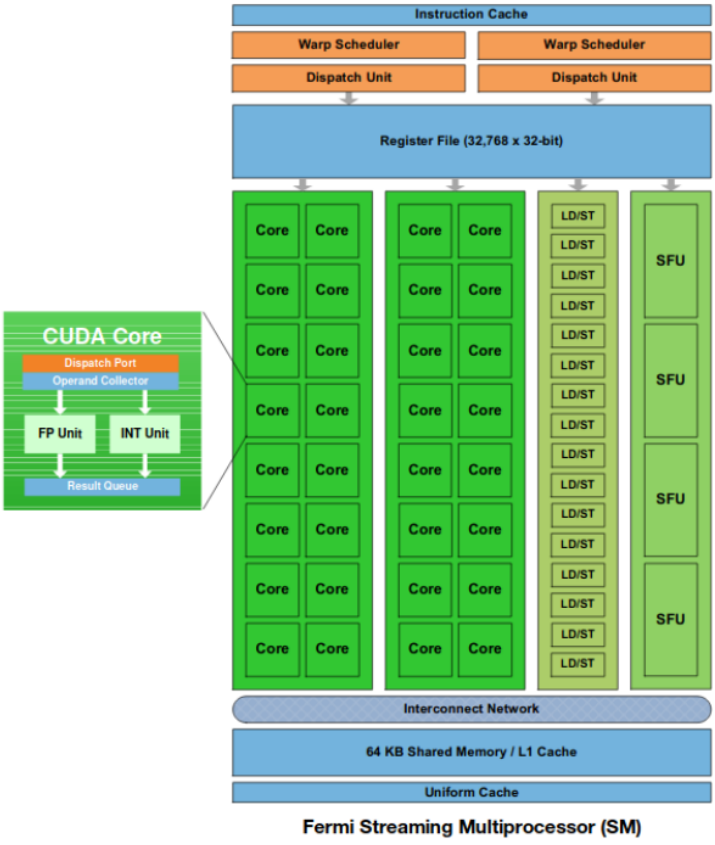

Figure 8.5: *Fermi Streaming Multiprocessor*

Each SM in a GPU is designed to support "concurrent execution" of hundreds of threads (up to 2048 in Volta [20]), and there are multiple SMs per GPU, so it is possible to have tens of thousands of threads executing concurrently on a single GPU. CUDA employs a **Single Instruction Multiple Thread** (SIMT) architecture to manage and execute threads in groups of 32 called warps. All threads in a warp execute the same instruction in each cycle. Each thread has its own instruction address counter and register state, and executes the current instruction on its own data.

The GPU is designed to have enough state to cover both the execution latency and the memory latency of hundreds of clock cycles that it may take for data from device memory to arrive after a read instruction is executed and this is fundamentally how GPUs achieve such large throughput.

8.1.2.3 The CUDA Memory Model

To programmers, there are generally two classifications of memory:

- **Programmable**: We explicitly control what data is placed in programmable memory.
- **Non-programmable**: We have no control over data placement, and rely on automatic techniques to achieve good performance. Caches are a prominent type of non-programmable memory.

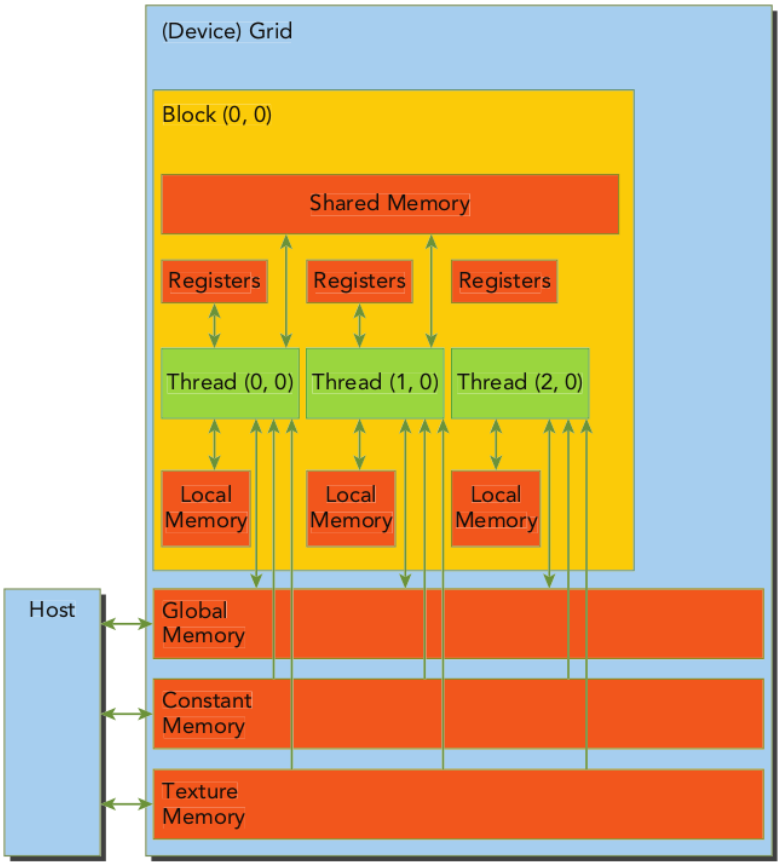

Figure 8.6: *CUDA Memory hierarchy*

The CUDA memory model exposes the following types of **programmable** memory:

- Registers (fastest/per-thread)
- Local Memory (stores register spills - same latency as global memory)

  The name "local memory" is misleading: Register values spilled to local memory reside in the same physical location as global memory, so local memory accesses are characterized by high latency and low bandwidth when compared to on-chip stores.

- Shared memory (fast, on-chip, per thread-block):
- Constant memory (read-only, initialized by host, accessible by all kernels)
- Texture memory (per-SM, read-only)
- Global memory (largest, off-chip, accessible from all kernels)

There are four types of caches in GPU devices:

- L1, L2
- Read-only constant, texture

8.1.2.4 What is a Kernel?

A CUDA program consists of a mixture of the following two parts:

- The **host code** runs on **CPU**.
- The **device code** runs on **GPU**.

NVIDIA's CUDA `nvcc` compiler separates the device code from the host code during the compilation process. The host code is written in standard C (there are bindings for other languages too) and it is compiled with an ordinary compiler. The code that runs on the device is written in C, using some CUDA-specific extensions to mark the data-parallel functions which execute on the GPU, which are called kernels. The same GPU kernel is run by hundreds or thousands of threads in parallel. The CUDA compiler, `nvcc` further compiles the kernel code. The application must also link to the CUDA Runtime Libraries so as to be able to issue commands, such as kernel launches and memory copies, to the GPU.

We express a kernel as a sequential program. A kernel is defined using the `__global__` declaration specifier. Behind the scenes, CUDA manages scheduling programmer-written kernels on GPU threads.

A typical processing flow of a CUDA program follows the pattern below:

1. Declare and allocate host and device memory.
2. Initialize host data.
3. Transfer data from the host to the device.
4. Execute one or more kernels.
5. Transfer results from the device to the host.

A kernel is most commonly launched in the following way:

```
Kernel_Name <<< GridSize, BlockSize,  SMEMSize, Stream >>> (arguments,....)
```

Quoting the NVIDIA Docs on kernel launch syntax [46]:
The execution configuration is specified by inserting an expression of the form `<<<Dg, Db, Ns, S>>>` between the function name and the parenthesized argument list, where:

- `Dg` is of type `dim3` (see [47]) and specifies the dimension and size of the grid, such that `Dg.x * Dg.y * Dg.z` equals the **number of blocks to launch**
- `Db` is of type `dim3` and specifies the dimension and size of each block, such that `Db.x * Db.y * Db.z` equals the **number of threads per block**

- `Ns` is of type `size_t` and specifies the number of bytes in shared memory that is dynamically allocated per block for this call in addition to the statically allocated memory; this dynamically allocated memory is used by any of the variables declared as an external array. Ns is an optional argument which defaults to 0.
- `S` is of type `cudaStream_t` and specifies the associated stream; S is an optional argument which defaults to 0.

As an example, a function declared as:

```
__global__ void Func (float* param)
```

must be called like this:

```
Func<<< Dg, Db, Ns >>>(param);
```

Summary:

- Threads are grouped into (thread) blocks Figure 8.7
- Thread blocks are grouped into a grid
- A kernel is executed as a grid of blocks of threads

Thread Blocks:

- TBs are assigned to SMs by the GPU thread-block scheduler based on their resource requirements and the SM capacity (for an in-depth view of TB Schedulers refer to this excellent study by Sreepathi Pai [21])
- Each TB (threadblock) is executed by one SM and does not migrate
- Several concurrent TBs can reside on one SM depending on the blocks' memory requirements and the SM's memory resources

Thread Blocks can execute in any order, concurrently or sequentially; this independence between blocks enables **scalability** (a kernel scales across any number of SMs - see Figure 8.8 below)

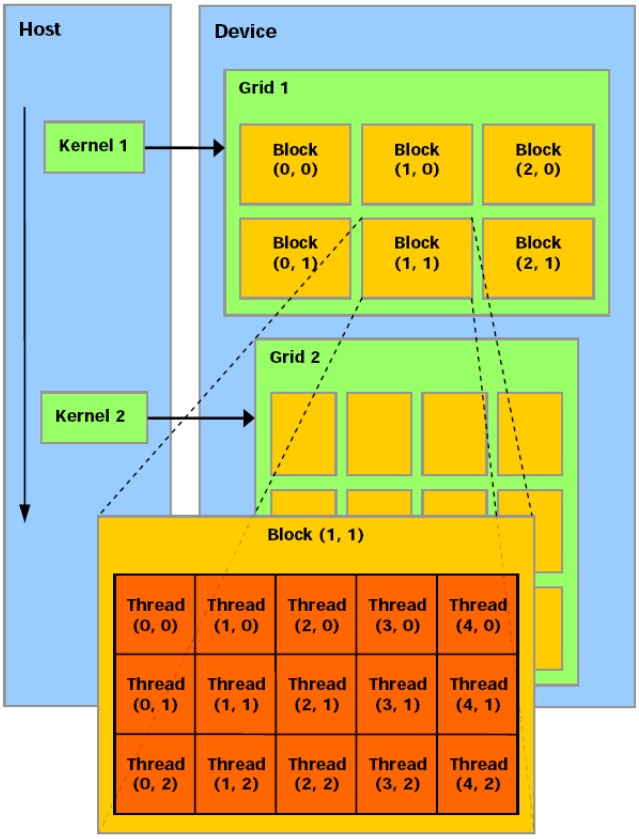

Figure 8.7: *CUDA thread hierarchy*

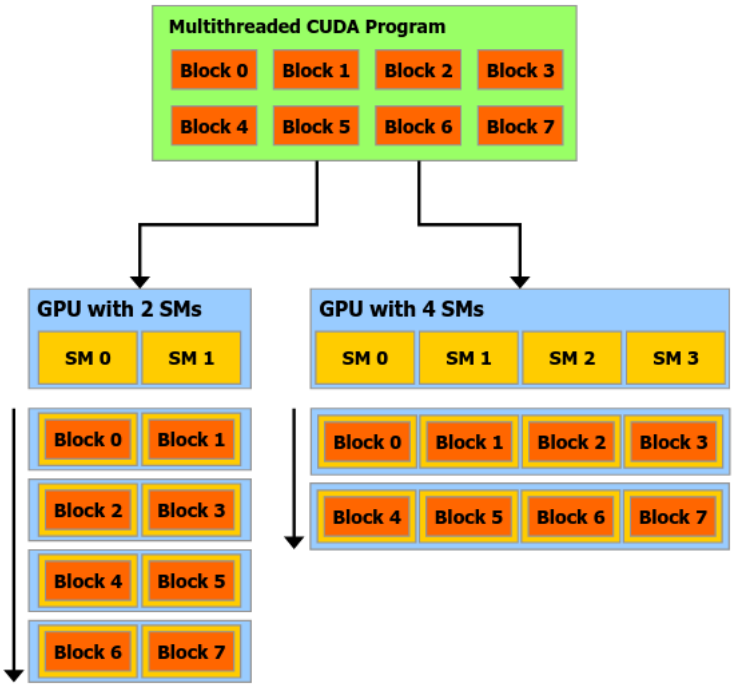

Figure 8.8: *SM Scalability*

Threads rely on the following two unique coordinates to distinguish themselves from each other:

- `blockIdx.{x,y,z}` (block index within a grid)
- `threadIdx.{x,y,z}` (thread index within a block)

Kernel functions can access these intrinsic variables, which are initialized by the GPU (black box) when the kernel begins execution. Based on the coordinates, programmers can partition data between the parallel threads.
Suppose we launch `myKernel<<4, 8>>(arguments,...)` (4 thread blocks * 8 threads each). Then the built-in variables will take the following values for the threads:

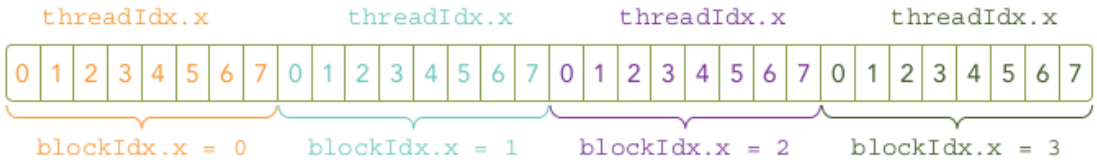

Figure 8.9: *Thread Block indexing*

Listing 8.1: Minimal example of thread indexing in CUDA.

```
1  // A and B are both vectors of size N
2  // Kernel definition
3  __global__ void VecAdd(float* A, float* B, float* C)
4  {
5      int i = threadIdx.x;
6      C[i] = A[i] + B[i];  //each of the N threads performs a pair-wise addition
7  }
8
9  int main()
10 {
11     ...
12     // Kernel invocation with N threads
13     VecAdd<<<1, N>>>(A, B, C);  // 1 ThreadBlock of N threads
14     ...
15 }
```

Warps:

Warps are the basic unit of execution in an SM. When we launch a grid of thread blocks, the thread blocks in the grid are distributed among SMs. Once a thread block is scheduled to an SM, threads in the thread block are further partitioned into warps of 32 threads (note that the size of warps is implementation specific and can change in the future). From the hardware perspective, a thread block is a one-dimensional collection of warps.

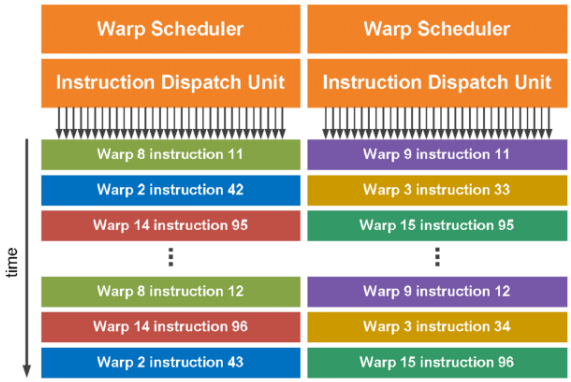

Figure 8.10: *Warp Scheduler*

Because compute resources are partitioned among warps and kept on-chip during the entire lifetime of the warp, switching warp contexts is very fast (warp contexts are completely different from device contexts, which we will cover later and are part of the programming API). A large number of warps need to be active in order to hide the latency caused by warps stalling, waiting for their operands.

Warp divergence occurs when threads within a warp take different code paths. Different if-then-else branches are executed serially (the threads in a warp that don't take the branch are marked to execute a NOOP - Figure 8.11). Different warps can execute different code with no penalty on performance. When a warp executes an instruction that accesses global memory the memory controller coalesces the memory accesses of the threads within the warp into as few transactions as possible as global memory accesses are extremely costly cycle-wise.

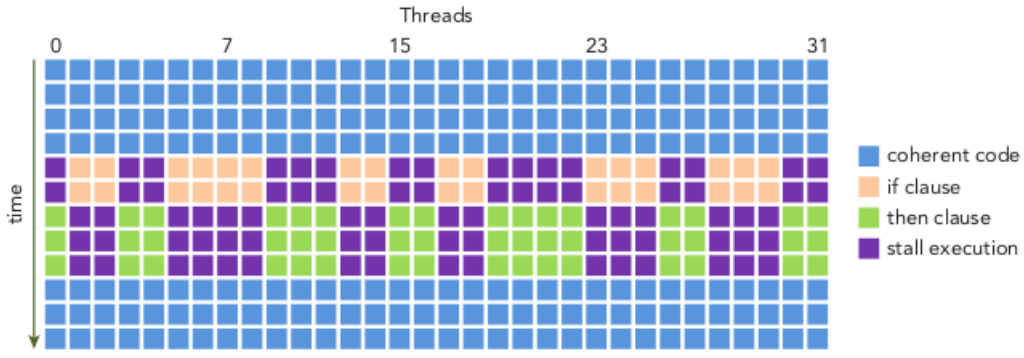

Figure 8.11: *Warp divergence in CUDA*

8.1.3 CUDA Programming Interface

We will now analyze the CUDA programming API from a programmer's point of view.

8.1.3.1 Preamble: Accelerator Silos

The authors of AvA (Automatic Virtualization of Accelerators) [2] hit the nail on the head with their simple observation (it's made from a virtualization standpoint, but applies in general):

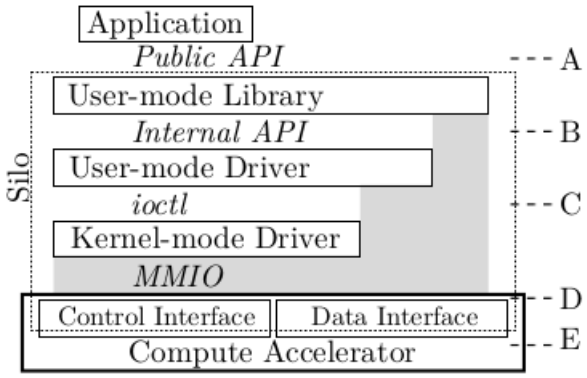

Figure 8.12: *Accelerator Silos [2]*

> *Accelerator stacks comprise layered components that include a user-mode library to support an API framework and a driver to manage the device. Vendors are incentivized to use proprietary interfaces and protocols between layers to preserve forward compatibility, and to use kernel-bypass communication techniques to eliminate OS overheads. However, interposing opaque, frequently-changing interfaces communicating with memory mapped command rings is impractical because it requires inefficient techniques and yields solutions that sacrifice compatibility. Consequently, accelerator stacks are effectively silos, whose intermediate layers cannot be practically separated.*

8.1.3.2 CUDA Runtime & Driver API

CUDA offers two programming interfaces:

1. the **Runtime** API and
2. the **Driver** API

The CUDA Runtime API provides C and C++ functions that *execute on the host* to allocate and deallocate device memory, transfer data between host memory and device memory, launch computational kernels, manage systems with multiple devices, etc. It is a high level interface that offers implicit initialization, context (more on contexts later) and module management. Functions of the Runtime API are prefixed with `cuda` (e.g. `cudaMalloc`, `cudaMemcpy`). Applications need to be linked to `libcudart.so` either statically or dynamically. A complete description of the runtime can be found in the CUDA reference manual [48].

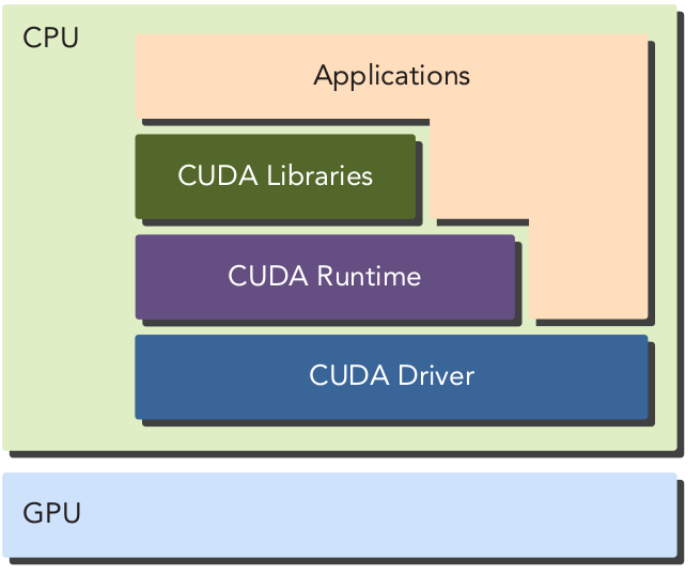

Figure 8.13: *CUDA Runtime and Driver API*

The Runtime API is *built on top of* a lower-level C API, the CUDA Driver API, which is also accessible by the application. The driver API provides an additional level of control by exposing lower-level concepts such as CUDA contexts - the analogue of host processes for the device - and CUDA modules - the analogue of dynamically loaded libraries for the device. Most applications do not use the driver API as they do not need this additional level of control and when using the runtime, context and module management are implicit, resulting in more concise code. As the runtime is interoperable with the driver API, most applications that need some driver API features can default to use the runtime API and only use the driver API where needed. Functions of the Driver API are prefixed with `cu*` (e.g. `cuMemAlloc`, `cuMemcpyDtoH`).

The Runtime API *dynamically loads* `libcuda.so`, the Driver API shared library, using `dlopen()`, so that it can invoke its functions internally. More information on the Driver API can be found in the relevant sections of the CUDA programming guide [49].

8.1.3.3 An Example Application

As we mentioned before, a typical processing flow of a CUDA program follows the pattern:

CUDA Workflow

1. Declare and allocate host and device memory [`malloc()` + `cudaMalloc()`]
2. Copy data from CPU memory to GPU memory [`cudaMemcpy(...,cudaMemcpyHostToDevice)`]
3. Invoke kernels to operate on the data stored in GPU memory [`foo<<...>>()`]
4. Copy data back from GPU to CPU memory [`cudaMemcpy(...,cudaMemcpyDeviceToHost)`]
5. Release host and device memory [`free()` + `cudaFree()`]

Below we show a program performing **SAXPY** [Single-precision A times X Plus Y, `(A*X + Y)`] on the GPU using the Runtime API: (the original can be found at [50])

Listing 8.2: SAXPY using the CUDA Runtime API

```
#include <stdio.h>

__global__              //device Function
void saxpy(int n, float a, float *x, float *y) {
/* find unique thread id - determine data to operate on */
  int i = blockIdx.x*blockDim.x + threadIdx.x;
/* make sure we don't run out of bounds */
  if (i < n) y[i] = a*x[i] + y[i];
}

int main(void) {
  int N = 1<<20;
  float *x, *y, *d_x, *d_y;
  x = (float*)malloc(N*sizeof(float));
  y = (float*)malloc(N*sizeof(float));

  cudaMalloc(&d_x, N*sizeof(float));
  cudaMalloc(&d_y, N*sizeof(float));

  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);

  // Perform SAXPY on 1M elements
  /*
   * Make sure the # of threads launched are >= N [N+255/256 TB of 256 threads each]
  */
  saxpy<<<(N+255)/256, 256>>>(N, 2.0f, d_x, d_y);

  cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);

  cudaFree(d_x);
  cudaFree(d_y);
  free(x);
  free(y);
}
```

8.1.3.4 Contexts

We can think of a CUDA context as the "projection" of a CPU process on the GPU. Each host process does all CUDA work within a context. When using the Runtime API, CUDA automatically handles creation and management of the context. However, when using the Driver API (which is the lower-level one), we must explicitly create and manipulate the GPU context in order to submit work to the GPU. Informally put, an application that wants to utilize a GPU creates a context and then issues commands (device memory allocations, cudaMemcpies, kernel launches) to that context.

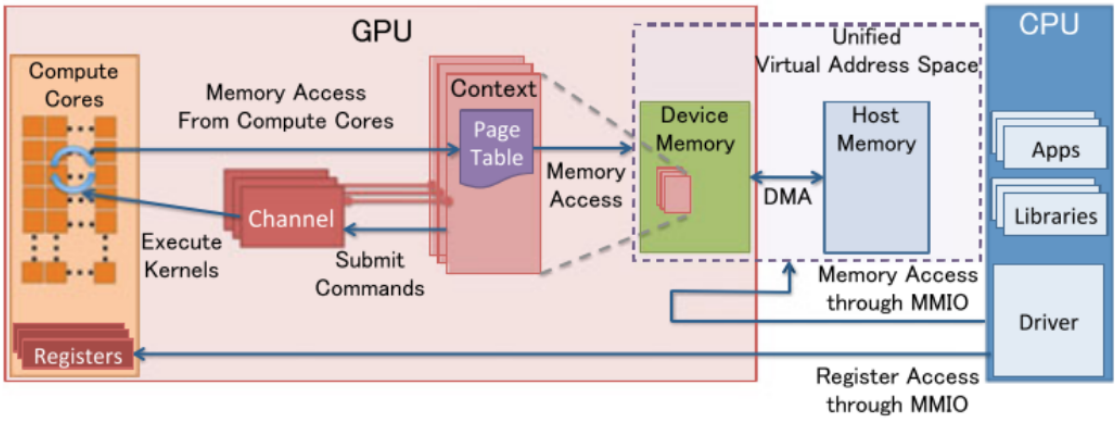

Figure 8.14: *GPU Resource Management Model*

More specifically (as far as we know from the documentation) the context contains:

1. All **memory allocations** (device memory, host memory, CUDA arrays)
2. Modules
3. CUDA streams
4. CUDA events
5. Texture and surface references
6. Device memory for kernels that use local memory
7. **Internal resources for debugging, profiling, and synchronization**

> **Note**
>
> *A CUDA Context is a driver level construct, the GPU itself doesn't know anything about contexts. All it knows is that it has a command queue, sent to it from the driver, that it will run through and execute. However, this does not affect the user's perception of contexts.*

For an analysis of the CPU-GPU command submission mechanism please refer to the very interesting blog post by Insu Jang [51] as well as S. Kato's work [52], other works from Kato and Tanasic's work [53].

8.1.3.5 Multiple Contexts (2 or more applications)

While multiple contexts (and their associated resources such as global memory allocations) can exist concurrently on a given GPU, only **one** of these contexts can execute work at any given moment; contexts sharing the same GPU are time-sliced. Creating additional contexts incurs memory overhead for per-context data and time overhead for context switching, but it is the only way for different applications to work on the same GPU. Time-slicing is handled by the driver in an undisclosed manner and there is no official documentation on it. We cover this in more detail in the relevant chapter.

8.1.3.6 Memory model (w.r.t. CUDA API)

The key information that a context holds is the **address space (page tables)** of the process on the GPU (we will see below that the host-device address spaces become one under UVA). These address spaces are unique per context, and, in a similar fashion to CPU processes, a context cannot access another context's memory.

Unified Virtual Addressing

It used to be the case that the address space of a CUDA context was separate and distinct from the CPU address space used by CUDA host code. However, in all modern GPUs (Compute Capability >=2.0) Unified Virtual Addressing (`UVA`) is used. `UVA` is enabled by default and there is no documented way to disable it. When `UVA` is in effect, the CPU and GPU(s) share the same address space; every host or device memory allocation has a unique address within the process. This does not mean however, that the CPU can directly read from and write to GPU memory and vice-versa. Only special types of CUDA memory (pinned non-pageable memory, peer-to-peer memory, Managed allocations) have this property.

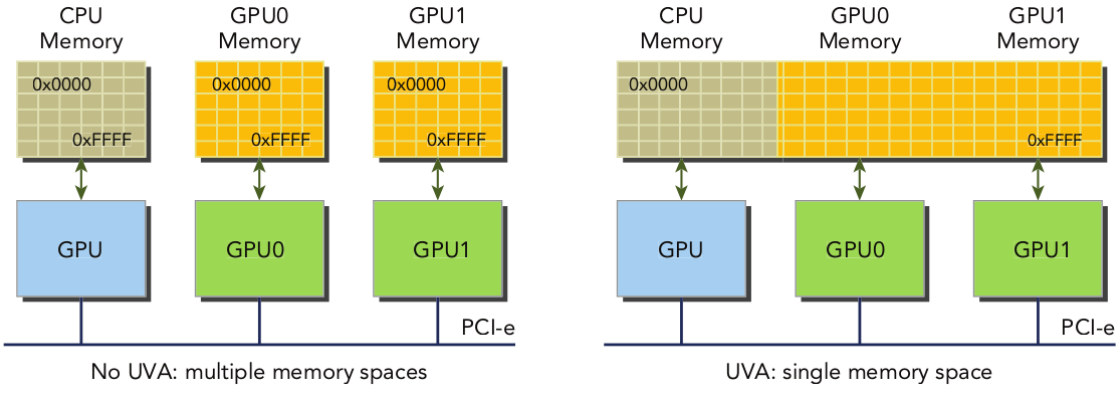

Figure 8.15: *Unified Virtual Addressing*

Unified Memory

An exception to the 1-1 relationship between physical and virtual device memory pages are Unified Memory Allocations (`cudaMallocManaged()`). With CUDA 6.0, a new feature called 'Unified Memory' was introduced to simplify memory management in the CUDA programming model. Unified Memory creates a pool of managed memory, where each

allocation from this memory pool is accessible on **both the CPU and GPU with the same memory address (pointer)**. The underlying system automatically migrates data in the unified memory space between the host and device. This data movement is transparent to the application and dispels the need to have distinct pointers for host RAM and GPU memory. We can thus transparently "extend" GPU device memory, using the host RAM as a backing store. Unified Memory depends on Unified Virtual Addressing (UVA) support. Unified Virtual Addressing does not automatically migrate data to and from the GPU; we must explicitly allocate the memory as Unified to enable this behavior.

Important Note

GPUs do not support demand paging for non-UM allocations, so every byte of virtual memory allocated by CUDA must be backed by a byte of physical memory.

For an easy introduction to Unified Memory with an example refer to the Nvidia Developer Blog [27].

8.1.3.7 Streams

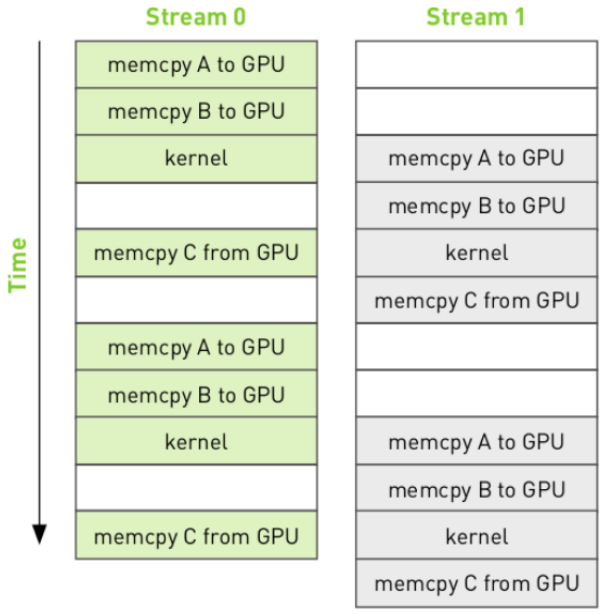

Figure 8.16: *CUDA Streams Overlapping*

A CUDA Stream encapsulates a sequence of (asynchronous) CUDA operations that the device executes in the order they are issued by the host. These operations usually comprise data transfers and kernel launches. We can have multiple active streams per context and, while the GPU preserves the ordering of the operations within a stream, the execution of operations belonging to different streams can overlap. By dispatching kernel execution and data transfer into separate streams, we can overlap these operations and reduce total execution time (see Figure 8.16). Every stream belongs to a context. Note that overlapping can happen only between streams of the same CUDA context, as work

from only a single CUDA context is processed at a given moment.

For further information on CUDA Streams refer to Lei Mao's excellent blog post [54] as well as NVIDIA's training presentation [55].

8.1.4 CUDA Libraries

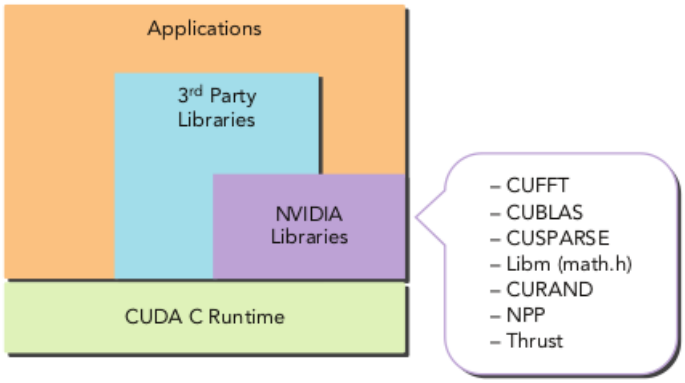

Figure 8.17: *CUDA Libraries*

To augment the abilities of CUDA developers, NVIDIA and other institutions provide domain-specific CUDA libraries that can be used as building blocks for more complex applications. These libraries have been optimized by CUDA experts and designed to have high-level, highly-usable APIs with standardized data formats to facilitate their ability to plug in to existing applications. CUDA libraries sit on top of the CUDA runtime, providing a simple, familiar, and domain-specific interface for both host applications and third-party libraries. ML frameworks make extensive use of these libraries, for their highly optimized linear algebra implementations.

Indicatively:

- `cuSPARSE` includes a range of general-purpose sparse linear algebra routines.
- `cuBLAS` includes CUDA ports of all functions in the standard Basic Linear Algebra Subprograms (BLAS) library for Levels 1, 2, and 3.
- `cuFFT` includes methods for performing fast Fourier transforms (FFTs) and their inverse.
- `cuRAND` includes methods for rapid random number generation using the GPU.

CUDA Software Stack overview:

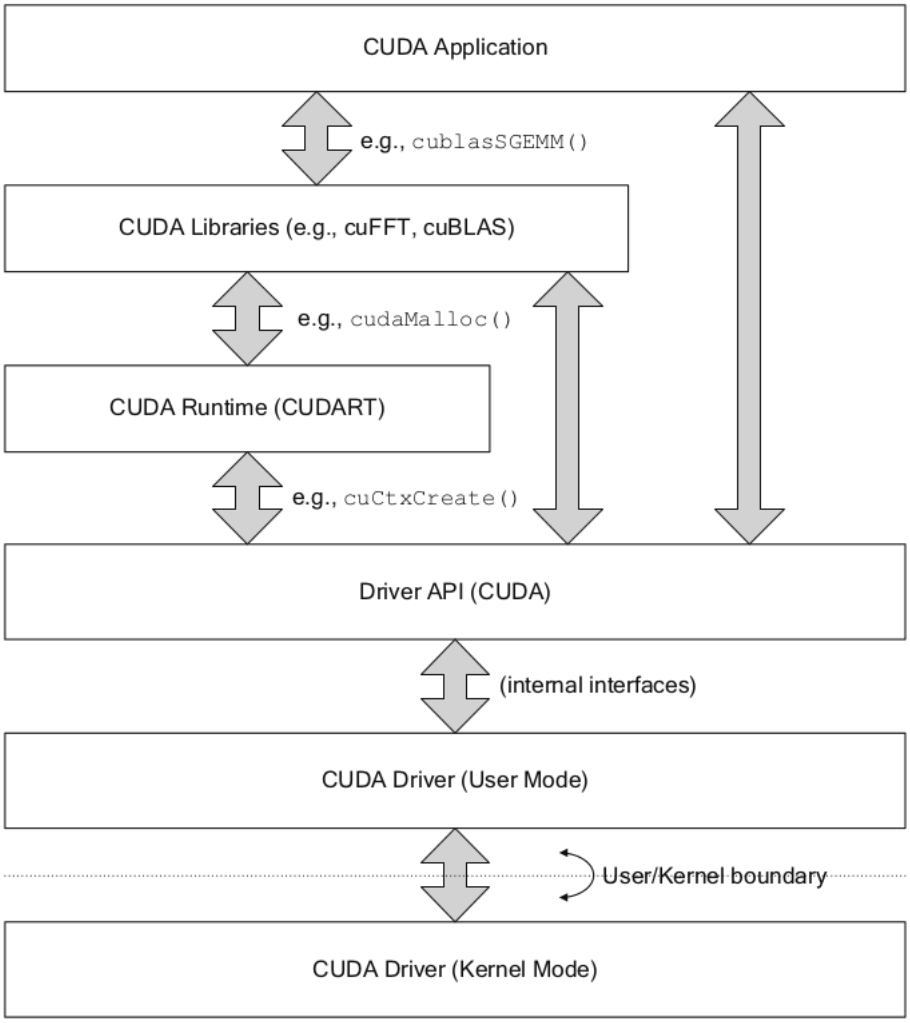

Figure 8.18: *CUDA Software Layers*

8.1.5 CUDA Compilation Process

Compilation works as follows (adapted from NVIDIA documentation):

1. `nvcc` preprocesses the input for device compilation and compiles it to CUDA binary (cubin) [56]) and/or (PTX) [57] intermediate code, which are placed in a "fatbinary" [58].
2. It preprocesses the input once again for host compilation and embeds the fatbinary. It also transforms CUDA-specific C(++) extensions into standard C(++) constructs (e.g. kernel launch syntax `<<<...>>>` is replaced by a set of Runtime API function calls).
3. The C(++) host compiler compiles the synthesized host code with the embedded fatbinary into a host object.

4. The CUDA Runtime system inspects the embedded fatbinary (when the host program launches device code) to obtain an appropriate image for the current GPU.

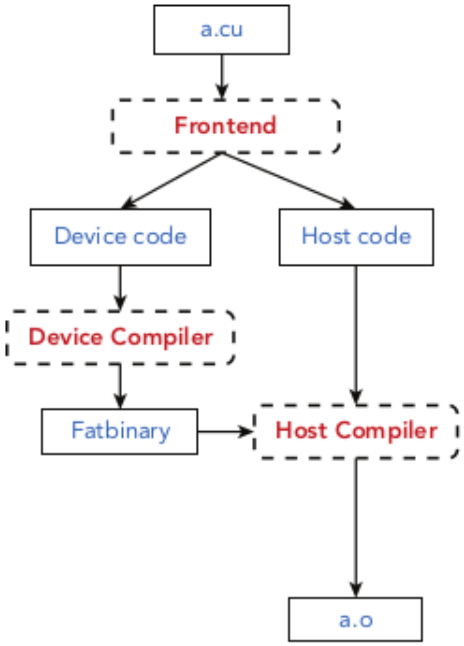

Figure 8.19: *CUDA Compilation Process*

For a deeper analysis of the CUDA Compilation Process see the excellent explanation from Vincent Jordan [59] as well as the Official NVIDIA documentation [60].

8.2 Concurrent execution of multiple processes in CUDA

As we mentioned before, to use the GPU, each application creates a (CUDA) context. This context contains, among others, the page tables, which describe its GPU memory allocations. We are going to use the terms context and application interchangeably henceforth. While multiple contexts can concurrently exist on the GPU, only one context can be active at a given moment [61][62]. This means that only a single context's operations (kernel launches, memory copies) are executed on the GPU at any moment. The GPU handles context switching internally and in an undisclosed manner, however we know that this is a time-sliced mechanism [63]. Modern GPUs (Pascal and newer) can preempt running kernels, so the GPU can switch contexts during kernel execution [64]. The documentation on MPS (Multi-process service, a mechanism from NVIDIA that funnels different contexts into one - only mainly used for different MPI ranks of the same application running on the same machine) explicitly states:
"The GPU also has a time sliced scheduler to schedule work from work queues belonging to different CUDA contexts. Work launched to the compute engine from work queues belonging to different CUDA contexts cannot execute concurrently. This can cause underutilization of the GPU's compute resources if work launched from a single CUDA context is not sufficient to use up all resource available to it."

Figure 8.20, from the MPS manual [63] shows two MPI processes concurrently executing using the GPU. Note the absence of overlap in kernel executions between the different contexts.

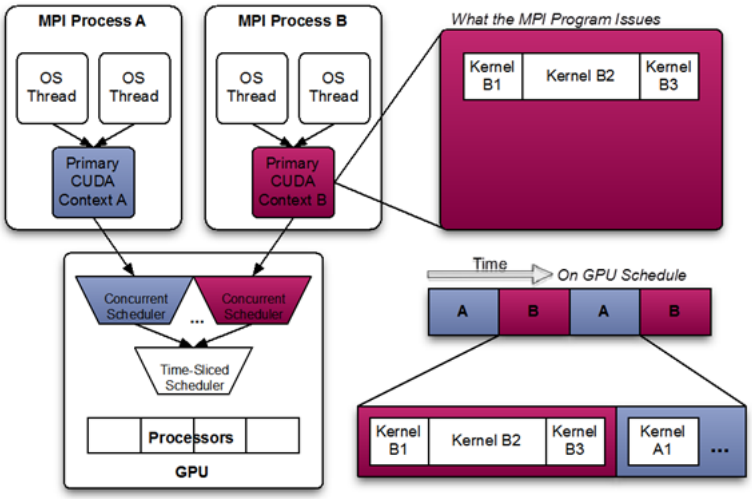

Figure 8.20: *Scheduling GPU work from different contexts*

A **limiting factor when co-locating processes is GPU memory** (VRAM). Unless using Unified Memory, total memory consumption from all contexts (CPU processes) cannot be greater than available VRAM. Most existing GPU applications do not use Unified Memory, and those that do, only use it as a means to oversubscribe GPU memory within a single context.

8.3 OS-level virtualization and containers

Operating-system-level virtualization refers to an operating system feature in which kernel services are used to allow the existence of multiple isolated user-space instances. In other words, the kernel provides us with tools that as a whole enable us to replicate the operating system's functionality for each isolated instance. There exist many OS-level virtualization implementations, such as BSD jails, LXC and Docker. In the latter two, the instances are called containers. Docker [65] played a critical role in popularizing containers and solidifying their status as the primary means of lightweight virtualization.

8.3.1 What is a container?

From the 10000 foot view, a container is a standard unit of software delivery that allows engineering teams to ship software reliably and automatically. In order to get a firmer grasp of their capabilities and limitations, we must examine the mechanisms that power containers behind-the-scenes and enable us to reap their benefits.

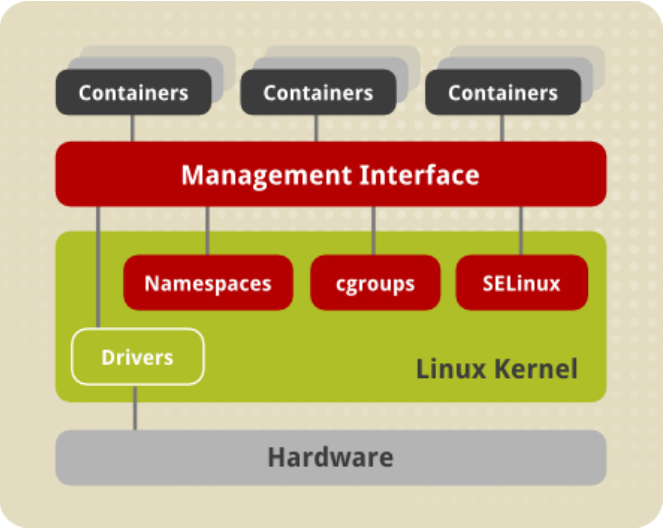

Figure 8.21: *Container Architecture*

8.3.1.1 cgroups

According to the Linux Kernel documentation [66]: "*cgroup is a mechanism to organize processes hierarchically and distribute system resources along the hierarchy in a controlled and configurable manner.*"

A cgroup (control group) comprises a set of processes which are bound by the same criteria and are assigned a set of parameters or limits. The organization of the groups can be hierarchical, in the sense that every group inherits its configuration from its parent. The Linux kernel exposes a variety of relevant controllers (subsystems) through the `cgroup` interface. As an example, the `memory` controller limits memory usage and `cpuacct` accounts CPU usage. Linux also exposes cgroups in the virtual file system, mounting the controller hierarchy under `/sys/fs/cgroup`.

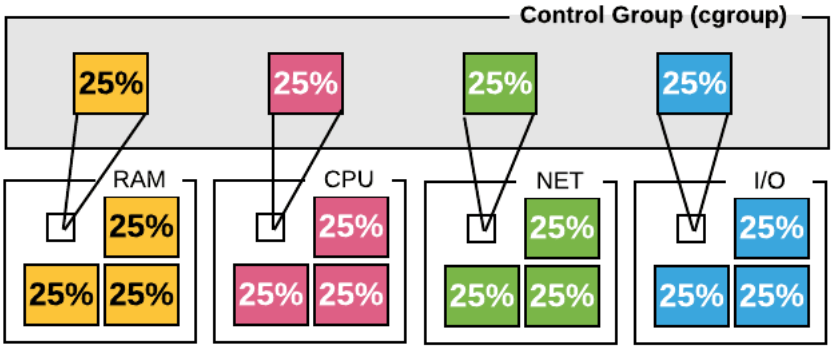

Figure 8.22: *Using cgroups to limit resource usage [3]*

8.3.1.2 Namespaces

Namespaces enable creating an abstraction of a particular global system resource and make it appear as a separated instance to processes within a namespace. Examples of resources include `pid`, `ipc` and `network`.

Putting the above together

It is clear that if we combine cgroups (to control resource consumption) and Namespaces (to isolate resource usage) we can run a process in the system that has its own unique file-system, its own IP address, and that has adjustable CPU and Memory usage. This is a primitive form of a container but even more robust examples of container runtimes stem from these basic roots.

8.3.2 Container Images

Here we must note that a container is an isolated *running* process with all the above desirable characteristics. Because containers, as a technology, emerged to cover the need of reproducibility, vendors need a way to ship a frozen version of the container which the user will execute. This frozen version is called a container image. A container image comprises a compressed set of Union mount Filesystem layers which, when decompressed, produce the final filesystem that the container will use as its root.

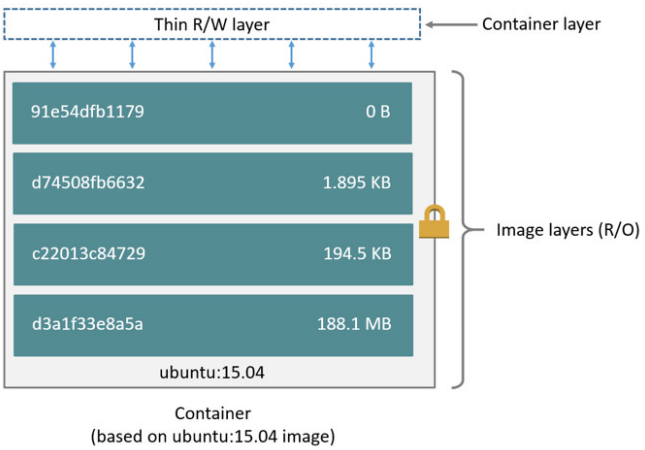

Figure 8.23: *Container Image Layers*

8.3.3 Open Contaner Initiative

The Open Container Initiative (OCI) [67] is a project started by Docker in June 2015 which designs standards for container technologies. OCI defines the following two standards:

- the **Runtime Specification** which specifies how to run an unpacked filesystem bundle (rootfs + a configuration file)
- the **Image Specification** which specifies how to create an OCI image, which contains the necessary files (or references to layers) as well as runtime configuration information as well as instructions on how to unpack this image into a Runtime Bundle.

For a short and thorough explanation of the OCI Specs, refer to the excellent blog post from Alibaba's Bin Chen [68].

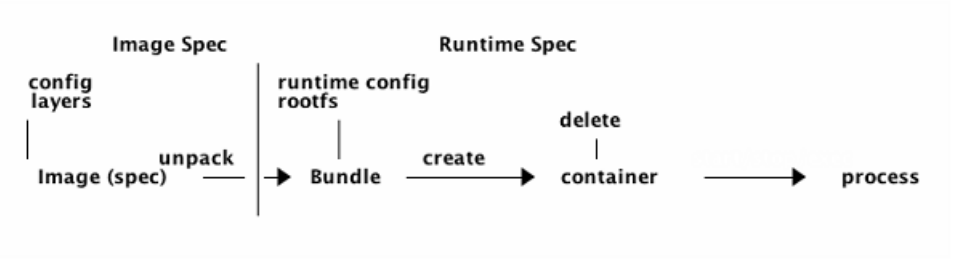

Figure 8.24: *Overview of the OCI lifecycle*

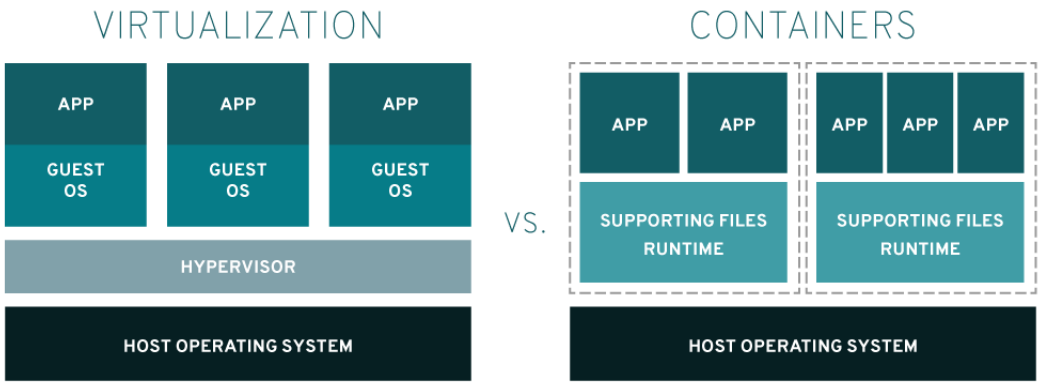

Figure 8.25: *Containers vs Virtual Machines [4]*

8.3.4 Comparison to VMs

While both containers and Virtual Machines are forms of virtualization, they have some significant differences. Each VM must run a completely separate guest OS and emulate its hardware devices. VMs with different operating systems can execute on the same host and each VM has its own distinct image on disk - measuring tens of gigabytes. VMs also provide strict security and isolation between workloads.

On the other hand, containers share the underlying OS, so executables must be binary compatible with the operating system. For example, you cannot run a Windows container in a Linux machine, while nothing stops you from running a Windows VM. Containers are bundled only with a minimal set of necessary dependencies (libraries, misc files) and container images usually are (this depends on the containerized application) orders of magnitude smaller than VM images.

8.4 Using GPUs in (OCI) containers

8.4.1 Introduction

In order to use (nvidia) GPUs to run and develop cuda applications within containers, the following must be present as a bare minimum:
(an analysis of a sample CUDA application's interaction with the system can be found in this post by Zygmunt Krynicki [69])

- `/dev/` character device files (`nvidia0`, `nvidiactl`, `nvidia-uvm`, `nvidia-uvm-tools`)
- NVIDIA libraries (`libcuda.so` as a minimum to run CUDA applications)

The above are necessary to expose the nvidia driver to the container. More specifically, the nvidia driver comprises multiple kernel modules (omitting the graphics related ones):

```
$ lsmod | grep nvidia
nvidia_uvm            1089536  0
nvidia_modeset        1114112  1 nvidia_drm
nvidia               20459520  20 nvidia_uvm,nvidia_modeset
```

It also provides a collection of user-level driver libraries that enable the applications to communicate with the kernel modules and therefore the GPU devices:

```
$ ldconfig -p | grep -E 'nvidia|cuda'
libnvidia-ml.so (libc6,x86-64) => /usr/lib/nvidia-361/libnvidia-ml.so
libnvidia-glcore.so.361.48 (libc6,x86-64) => /usr/lib/nvidia-361/libnvidia-glcore.so.361.48
libnvidia-compiler.so.361.48 (libc6,x86-64) => /usr/lib/nvidia-361/libnvidia-compiler.so.361.48
libcuda.so (libc6,x86-64) => /usr/lib/x86_64-linux-gnu/libcuda.so
...
```

For a precise list of the installed components of the nvidia driver refer to the NVIDIA Driver Documentation [70]. For compute applications we aren't particularly interested in the graphics components listed therein.

The **CUDA toolkit**, required to develop applications is installed inside the container, usually by leveraging nvidia/cuda images as the base for image builds (note the ENV variables for later).

Listing 8.3: nvidia/cuda:10.1-base Dockerfile

```
FROM ubuntu:18.04
LABEL maintainer "NVIDIA CORPORATION <cudatools@nvidia.com>"
RUN apt-get update && apt-get install -y --no-install-recommends \
    gnupg2 curl ca-certificates && \
    curl -fsSL https://developer.download.nvidia.com/compute/cuda/repos/ubuntu1804/
    x86_64/7fa2af80.pub | apt-key add - && \
    echo "deb https://developer.download.nvidia.com/compute/cuda/repos/ubuntu1804/
    x86_64 /" > /etc/apt/sources.list.d/cuda.list && \
    echo "deb https://developer.download.nvidia.com/compute/machine-learning/repos/
    ubuntu1804/x86_64 /" > /etc/apt/sources.list.d/nvidia-ml.list && \
    apt-get purge --autoremove -y curl \
    && rm -rf /var/lib/apt/lists/*

ENV CUDA_VERSION 10.1.243
ENV CUDA_PKG_VERSION 10-1=$CUDA_VERSION-1

# For libraries in the cuda-compat-* package: https://docs.nvidia.com/cuda/eula/index.
    html#attachment-a
RUN apt-get update && apt-get install -y --no-install-recommends \
    cuda-cudart-$CUDA_PKG_VERSION \
    cuda-compat-10-1 \
    && ln -s cuda-10.1 /usr/local/cuda && \
    rm -rf /var/lib/apt/lists/*

# Required for nvidia-docker v1
RUN echo "/usr/local/nvidia/lib" >> /etc/ld.so.conf.d/nvidia.conf && \
    echo "/usr/local/nvidia/lib64" >> /etc/ld.so.conf.d/nvidia.conf

ENV PATH /usr/local/nvidia/bin:/usr/local/cuda/bin:${PATH}
ENV LD_LIBRARY_PATH /usr/local/nvidia/lib:/usr/local/nvidia/lib64

# nvidia-container-runtime
ENV NVIDIA_VISIBLE_DEVICES all
ENV NVIDIA_DRIVER_CAPABILITIES compute,utility
ENV NVIDIA_REQUIRE_CUDA "cuda>=10.1 brand=tesla,driver>=396,driver<397 brand=tesla,
    driver>=410,driver<411 brand=tesla,driver>=418,driver<419"
```

One of the *early solutions* was to install the NVIDIA driver in the container and mount the device files manually, as documented in this answer on StackOverflow [71]. This approach led to many issues [72] as the NVIDIA driver inside the container had to match the precise driver on the host machine (whose kernel modules it would utilize to access the GPU). This meant that the Docker images were not portable and had to be build locally on each machine, thus defeating one of the main purposes of Docker.

8.4.2 nvidia-docker-{1,2}

In 2015, nvidia released `nvidia-docker1` [73]. This initial release included all the functionalities (exposing device files, libraries) and acted as a thin wrapper over Docker while also utilizing a daemon process to detect the aforementioned files. We will not delve into the details of this initial implementation as it was deprecated with the release of `nvidia-docker2` [74], which is just a small part of the Nvidia Container Toolkit [75] stack. The functionalities have been broken down into smaller pieces, with the purpose of enabling support of all OCI runtimes and *not being merely bound to Docker*.

For example, we can now work with any other high-level container runtime such as `Podman` [76] and just instruct it to use `nvidia-contaner-runtime`.

```
|root@legion|:[/root]> podman --runtime /usr/bin/nvidia-container-runtime run nvidia/cuda:10.2-base nvidia-smi
Mon Nov  9 13:20:36 2020
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 440.82       Driver Version: 440.82       CUDA Version: 10.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  GeForce GTX 105...  On   | 00000000:01:00.0 Off |                  N/A |
| N/A   40C    P8    N/A /  N/A |      0MiB /  4042MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

Figure 8.26: *using Podman to run a GPU container*

We show NVIDIA Container Toolkit's architecture in figure 8.27:
`nvidia-docker2` is **the only Docker specific component** in the hierarchy. It merely installs `nvidia-container-runtime` as the (default) runtime for Docker in `/etc/docker/daemon.json`.

Listing 8.4: Setting nvidia-container-runtime in Docker's daemon.json

```
"default-runtime": "nvidia",
"runtimes": {
    "nvidia": {
        "path": "/usr/bin/nvidia-container-runtime",
        "runtimeArgs": []
    }
}
```

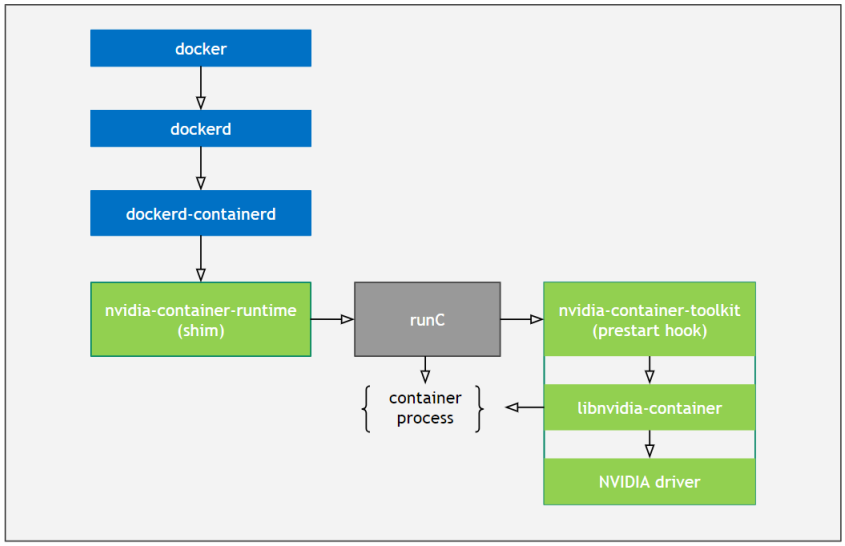

Figure 8.27: *NVIDIA Container Toolkit architecture*

8.4.3 nvidia-container-runtime

`nvidia-container-runtime` [77] is then invoked by `containerd` [78] and pointed to the `OCI runtime bundle`.

Note

- `containerd` is Docker's mid-level runtime: it's job is mainly to convert an OCI `Image` to an OCI `runtime bundle` and call `runc` [79] or another low level runtime
- An `OCI Runtime bundle` comprises a `rootfs` directory and a `config.json` configuration file.

It opens 'config.json' and modifies it, adding `nvidia-container-runtime-hook` (a symlink to [`nvidia-container-toolkit`) as a Prestart Hook.

Listing 8.5: Adding nvidia-container-toolkit PreStart Hook

```
1  func addNVIDIAHook(spec *specs.Spec) error {
2          path, err := exec.LookPath("nvidia-container-runtime-hook")
3          spec.Hooks.Prestart = append(spec.Hooks.Prestart, specs.Hook{
4                  Path: path,
5                  Args: append(args, "prestart"),
6          })
```

Listing 8.6: config.json with Prestart Hook injected

```
"hooks": {
    "prestart": [{
        "path": "/usr/bin/nvidia-container-runtime-hook",
        "args": ["/usr/bin/nvidia-container-runtime-hook","prestart"]
      }]
}
```

`nvidia-container-runtime` then `execve`'s (see `man 2 execve`) `runc`, which uses the modified spec (config.json) with the prestart hook inserted. `runc` executes the `Prestart Hooks` [80] after it creates the container Namespaces, but before it executes the user-specified program. Figure 8.28 shows the lifecyle of and OCI-conformant container:

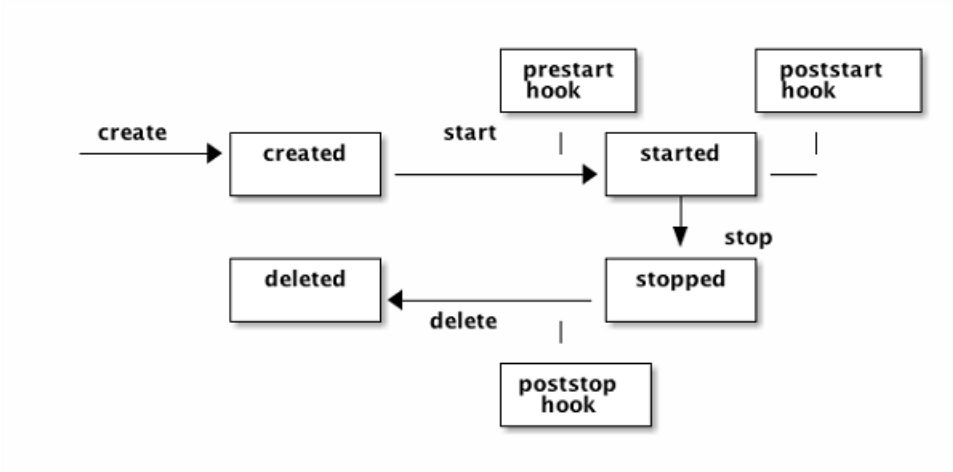

Figure 8.28: *OCI Container lifecycle*

8.4.4 nvidia-container-toolkit

`nvidia-container-toolkit`'s purpose is to invoke `nvidia-container-cli` [81] with the right arguments. To formulate those arguments, `nvidia-container-toolkit` examines the `config.json` configuration file and looks for specific Environment Variables (`NVIDIA_VISIBLE_DEVICES`) to determine the following:

1. devices to expose: `NVIDIA_VISIBLE_DEVICES` or VolumeMounts (see below for that)
2. driver capabilities: `NVIDIA_DRIVER_CAPABILITIES`
3. requirements (CUDA driver version, architecture) `NVIDIA_REQUIRE_XXX`

The ENV variables' usage is documented nicely in nvidia-contaner-runtime's README [82].

Recently, NVIDIA introduced a **new method of declaring devices to expose**, in order to **avoid accidental and/or unauthorized access** to GPUs in a cluster, especially in Kubernetes. The relevant design document [83] explains that the main motivation behind this change is the need to ensure that `nvidia-k8s-device-plugin` [8] is the sole purveyor of GPUs in a given cluster.

Note

At the moment, most GPU container images have the aforementioned ENV variables already set. As a consequence, submitting them even without requesting `nvidia.com/gpu` resources **still results in them being provided access to a GPU** by `nvidia-container-toolkit`.

8.4.5 nvidia-container-cli

`nvidia-container-cli` leverages `libnvidia-container` [81] in order to mount the necessary libraries, expose the device files and perform all actions needed in order for the container to be able to access the GPU. Time constraints did not allow for a detailed analysis of its source code, which is written in C and is platform agnostic. In short, it enters the newly created namespaces and mounts the `/dev/` files and nvidia driver libraries.

The relevant **strace** output is quite illustrative:

Listing 8.7: Strace output of nvidia-container-cli

```
22314 execve("/usr/bin/nvidia-container-cli", ["/usr/bin/nvidia-container-cli",
"--load-kmods", "configure", "--ldconfig=@/sbin/ldconfig", "--device=all",
"--compute", "--utility",
"--require=cuda>=10.2,brand=tesla,driver>=396,driver<397",
"--pid=22287",
"/var/lib/docker/overlay2/xxx/merged"], 0xc0000ae5c0)

Note here that /var/lib/docker/overlay2/<hash>/merged points to the rootfs (merged
image layers) of the container.
...
22314 openat(AT_FDCWD, "/proc/22287/ns/mnt", O_RDONLY) = 5
22314 openat(-1, "/var/lib/docker/overlay2/xxx/merged", O_RDONLY|O_NOFOLLOW|O_PATH|
    O_DIRECTORY) = 5
22314 openat(5, "dev", O_RDONLY|O_NOFOLLOW|O_PATH) = 6
22314 stat("/dev/nvidia0", {st_mode=S_IFCHR|0666, st_rdev=makedev(0xc3, 0), ...}) = 0
22314 stat("/var/lib/docker/overlay2/xxx/merged/dev", {st_mode=S_IFDIR|0755, st_size
    =400, ...}) = 0
22314 openat(AT_FDCWD, "/var/lib/docker/overlay2/xxx/merged/dev/nvidia0", O_RDONLY|
    O_CREAT|O_NOFOLLOW,    0644) = 5
22314 mount("/dev/nvidia0", "/var/lib/docker/overlay2/xxx/merged/dev/nvidia0", NULL,
    MS_BIND, NULL) = 0
22314 mount(NULL, "/var/lib/docker/overlay2/xxx/merged/dev/nvidia0", NULL,
                        MS_RDONLY|MS_NOSUID|MS_NOEXEC|MS_REMOUNT|MS_BIND, NULL) = 0
...
22314 openat(AT_FDCWD, "/sys/fs/cgroup/devices/docker/XXX/devices.allow", O_WRONLY|
    O_CREAT|O_APPEND,      0666) = 5
22314 write(5, "c 195:0 rw", 10)        = 10
Afterwards:
$ (ls -l /dev/) crw-rw-rw-   1 root root    195,   0 Nov  5 11:42 nvidia0
```

8.4.6 Docker "–gpus" option

Since Docker 19.03, Docker supports the `--gpus` option [84]. When we use "`docker run --gpus all ...`", `docker-cli` (the user command line interface) parses it and converts it into a `DeviceRequest` which it then passes over to dockerd. Dockerd (Moby [85]) interprets the `DeviceRequest`, sets the `NVIDIA_VISIBLE_DEVICES` variable in the OCI spec and finally adds the `nvidia-prestart-hook` in the relevant section of `config.json`. Consequently, the only parts of the hierarchy that need to be present are `nvidia-container-toolkit` and `nvidia-container-cli`. The flow from there onwards is the same as in the general case we analyzed above.

8.5 Kubernetes

8.5.1 Introduction

Containers provide a way for applications to run inside isolated, immutable and reproducible environments. Launching a container is trivial that virtually every developer does on a regular basis. The logistical problem arises when the number of applications (and users) grows significantly. In that case, managing a significant number of physical nodes that run user containers, executing health checks on them and ensuring that containers recover from failure is no trivial task. Kubernetes satisfies this need, in addition to providing ways to scale apps dynamically, ways for different containers to communicate with each other and share underlying storage. It is a managing platform for containerized workloads, and is ubiquitous in today's cloud computing landscape.

Kubernetes in a nutshell

Kubernetes is a software system that runs containers on Nodes. Kubernetes works on the basis of declaration of intent, instead of carrying out imperative commands. It consists of:

1. a persistent storage (etcd), in which it stores objects
2. a set of Controllers, which act on those objects, trying to reconcile the actual state of the cluster to the desired state

All state is stored in etcd. With only a few exceptions, `Objects` consist of a `Spec`, which describes the **desired state** and a `Status`, which describes the **current state**. Kubernetes `Controllers` monitor Objects and try to reconcile the actual state of the cluster with the desired state.

8.5.2 Architecture

Kubernetes (also known as K8s) follows the Master-Slave architectural pattern and comprises the Kubernetes `Master` and multiple Kubernetes `Nodes`. The Master (Control Plane) maintains the `etcd` database in addition to the API server. Each Node runs an instance of `kubelet`, which queries the API server and runs Pods bound to the Node and an instance of `kube-proxy` to control networking. Pods are the basic building block of K8s, and they are the smallest schedulable (to Nodes) entity. Each Pod encompasses one *or more* containers. The K8s networking model ensures that each Pod views itself with the same IP address as the rest of the Pods see it; it can also connect with other Pods as well as the Node that hosts it without any need for NAT. Figure 8.29 visualizes what we just mentioned.

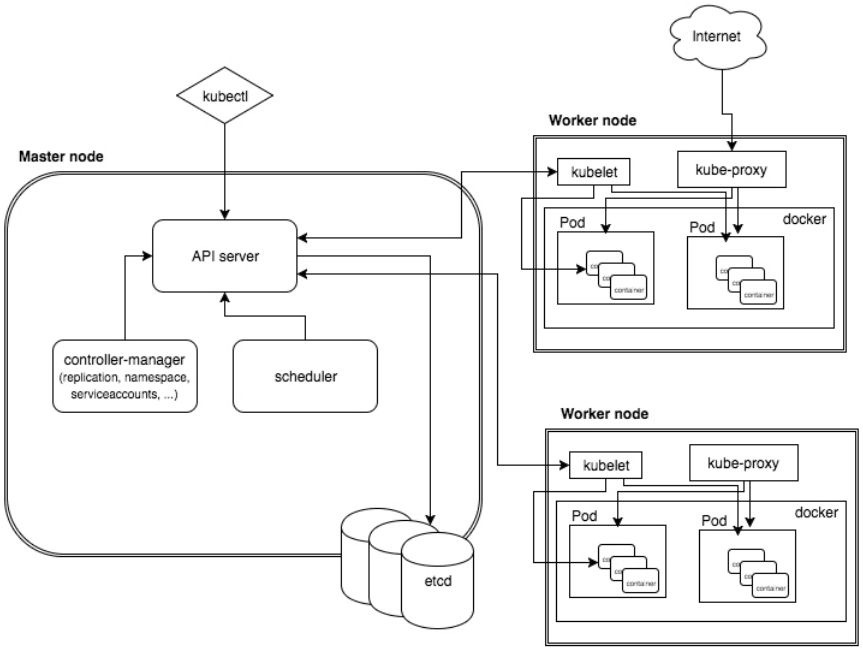

Figure 8.29: *A simplified view of Kubernetes' Architecture*

Let's analyze the individual components:

Master:

- **etcd**: The central storage of K8s. `etcd` [22] is a strongly consistent, distributed key-value store that provides a reliable way to store data belonging to a cluster of machines. Every single piece of persistent data in K8s is stored in etcd. In order to avoid having a Single Point of Failure (SPOF), K8s can run multiple instance of Master to ensure fault-tolerance.
- **API Server**: The Kubernetes API server serves all `REST` requests related to managing K8s Objects and is the sole front-end to the cluster. All modifications to data in etcd must go through the API Server.
- **controller-manager**: Manages controller entities (processes), which monitor the state of K8s Objects and try to reconcile the actual state with the desired state, which the user or other controllers declare.
- **Scheduler**: Decides which Node a Pod will run on, based on its resource requirements and the Node's remaining capacity.

Node:

- **kubelet**: The `kubelet` is responsible for running containers on the Node. It watched the API Server to see if it needs to run/stop Pods that have been assigned by the Scheduler to that Node. It also regularly performs health-checks on the Node's Pods' Containers, restarting those that are unhealthy. Finally, it is responsible to registering the `Node` and its available resources to the API Server and managing the Node Object throughout its lifetime.

- **kube-proxy**: Enables the resident container workloads to be accessed by end-users. `kube-proxy` creates custom `iptables` rules that redirects traffic sent to a Service IP (which is outwards-facing) to the respective Pod/Container.

8.5.3 Objects

Objects are the persistent entities in a K8s cluster and are stored in etcd. Kubernetes works on the basis of declaration of intent, instead of carrying out imperative commands. As such, when a user creates an object they declare the desired state they want the system to be in. The two important fields in an Object are its `Spec` and its `Status`. The Spec describes the desired characteristics that the user wants the object to have; the *desired state*. The Status describes the *current state*; The control plane (which includes controllers) continually and actively manages reconciling the current state with the desired state. Users usually define objects in YAML, but the command-line interface (`kubectl`) converts it to JSON before forming and sending a request to the API Server.

For example, a Deployment is an object that can be used to represent an application running on the cluster. When creating a Deployment, we can specify in the `Spec` that we want 3 replicas (Pods) of the application up and running. The Deployment Controller watches for Deployment Objects on the API Server, and, when it notices the new Deployment, it creates 3 Pod Objects. If any of those Pods fail (a change which will be reflected in the `Status` field) the Deployment Controller will try to reconcile the difference between desired-actual state by launching a replacement instance.

Here is an example of an `nginx` Deployment, which is a popular kind of Object:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # tells deployment to run 2 pods matching the template
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

8.5.4 Pods

Pods serve as the basic building block for more complex K8s Objects, as they are the smallest deployable entity in the Kubernetes API. A Pod comprises one or more container(s), which as a whole represent an application (e.g. an nginx server). Apart from its containers and the execution parameters the Spec provides for them, a Pod also encapsulates storage, and network identity for the application. Containers belonging to a Pod are automatically co-located and co-scheduled on the same Node by `kubelet`. These containers can communicate and coordinate with each other for the entirety of the Pod's lifetime.
Here is a simple example of a Pod in YAML:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-example
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

8.5.5 Controllers

In the introduction to Kubernetes, we stated that in essence K8s is simply a collection of Objects and Controllers. Having covered the former, it's now time to explain how Controllers come into play.

Controllers

Controllers are pieces of software that are responsible for managing a specific K8s Object. They execute a continuous control loop which tries to reconcile the `Status` of an Object with its `Spec` - or in simple words the actual state with the desired state. They play a central role in a Kubernetes system. Here is a breakdown of the loop that a Controller responsible for Objects of kind X runs:

1. Observe the current `Status` (current state) of objects of kind X
2. Calculate the difference from the `Spec` (desired state) (e.g. 3 nginx Pods should be running but only 2 are)
3. Take action to reconcile the existing differences (e.g. Create 1 more nginx Pod)

Virtually everything in Kubernetes is a Controller (e.g. kubelet, kube-proxy, Replication controller, ServiceAccounts controller). Let's do a specific high-level case-study of the Scheduler. Weird as it may seem, the Scheduler is also a Controller.

8.5.5.1 The Kubernetes Scheduler

In the case of the `Scheduler` the **desired state** is that the `NodeName` of a Pod's Spec (`pod.Spec.Nodename`) is not empty. This is an exception to the usual *modus operandi* of K8s Controllers, in which they calculate the desired state by comparing an Object's `Status` to its `Spec`. The Scheduler obtains the **current state** by examining the length of `pod.Spec.Nodename` and checking if it is greater than 0.

In order to schedule a given Pod, the Scheduler follows these steps, which comprise the **scheduling process**. In this brief analysis, we are only focusing on the Resource aspect of scheduling (which is the most important) and ignoring other factors such as affinity/anti-affinity labels.

1. It **calculates the remaining allocatable capacity of each Node** in the following manner:
   (a) Reads the `Status` field of the `Node` Object and initializes a `NodeInfo` cache with the total `Allocatable` resources of the Node.
   (b) Reads the `Pod.spec.containers.resources` fields of each Pod that is running on the Node and calculates the total resources of each individual Pod.
   (c) Stores the sum of total Pod resource requirements of each Node in the `Requested` field of the NodeInfo struct.

2. It **finds candidate Nodes** by checking if there are enough resources to fit the Pod's containers:
   (a) Checks the `podRequest` (the sum of resource requirements of all containers in the given Pod) against NodeInfo.Allocatable (total capacity of the Node) minus nodeInfo.Requested (current occupancy) to determine if the Pod fits.
   (b) If the Pod fits, it adds the Node to a list of candidate Nodes.

3. It **chooses one of the candidate Nodes to schedule the Pod on**:
   (a) Assigns a score to each Node in the candidate Nodes list (the least remaining resources the better).
   (b) Picks the Node with the highest score.
   (c) **Updates the Pod Object** in `etcd`, setting `Spec.NodeName` to that of the selected Node.

Now that we have a basic understanding of how Kubernetes operates, we can begin to slowly make our way to the gist of this thesis.

8.6 GPUs in Kubernetes and device plugins

8.6.1 Overview

Kubernetes supports `Device Plugins` [23] in order to let (containers running in) Pods access specialized hardware resources such as GPUs. To consume a GPU, a user must specify `nvidia.com/gpu` in the limits section of their request. However, there are some limitations according to the official documentation [86] 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.

8.6.2 Device Plugins in a nutshell

At their core, device plugins are simple gRPC [24] servers that can run in a container deployed through the pod mechanism or in bare metal mode. A device plugin's responsibilities are to:

- **inform the kubelet of the devices present on the node** that it manages
- **notify it of any change in their health** and finally
- **respond to requests** for specific devices from the kubelet, instructing it on **what additions need to be made to the configuration file** of the pod/container before kubelet forwards it to the underlying CRI [25] container runtime (dockershim, cri-o)

Since we concern ourselves with GPUs, we are going into more detail about `nvidia-device-plugin` [8] but the analysis made applies in general to every device plugin deployed in K8s.

8.6.3 nvidia-device-plugin

The NVIDIA device plugin for Kubernetes is a Daemonset that allows users to automatically:

- Expose the number of GPUs on each nodes of their cluster
- Keep track of the health of their GPUs
- Run GPU enabled containers in their Kubernetes cluster.

8.6.3.1 Initialization

The following sequence diagram (8.30) illustrates the first two parts (registration and device advertisement) of the aforementioned responsibilities:

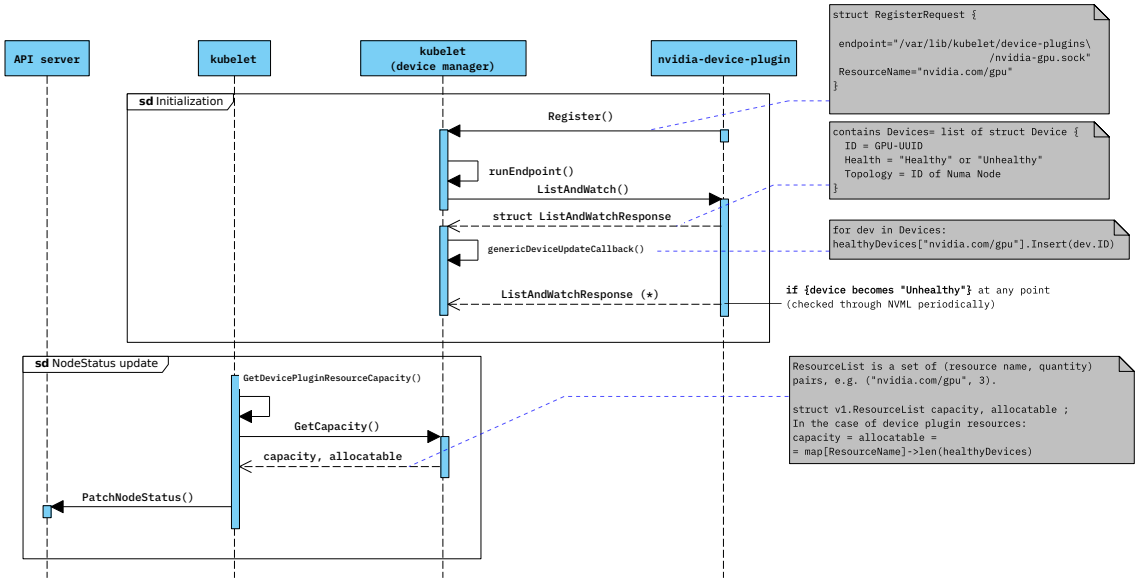

Figure 8.30: *device-plugin Initialization and Status Update*

- The device plugin registers the `ResourceName` it's managing (in this case "nvidia.com/gpu") as well as the endpoint it is listening on to the `kubelet`. The "subsystem" of kubelet tasked with handling devices is the `device manager`. Its responsibilities comprise acting as a middleman between the "core" of the kubelet (the part of it that interacts with the API server) and the various device plugins. It facilitates practical device exposure in Kubernetes while also *mitigating the need for adding device-specific code in the K8s core*.

- Kubelet's device-manager creates an endpoint which manages the remainder of the device plugin's interactions (gRPC communication) with kubelet. An endpoint maps to a single registered device-plugin and caches device states reported by it.

- The device manager then makes the `ListAndWatch()` call, which returns a list of `Device UUIDs` along with the health of each device. The **kubelet** then **adds the number of healthy devices to the NodeStatus update** that it periodically sends to the API Server. Specifically, the NodeStatus update only contains the `ResourceName` (in this case "nvidia.com/gpu") and an integer count of the devices.

8.6.3.2 Device Allocation

The sequence diagram below (8.31) illustrates the actions taken between the binding of a GPU Pod to a Node by the `Scheduler` and the creation of the corresponding container.

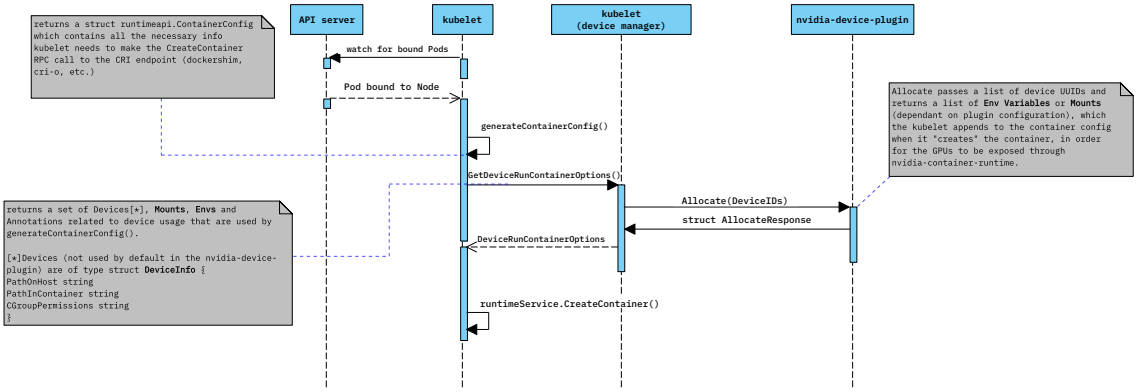

Figure 8.31: *device-plugin GPU allocation*

When kubelet retrieves a `PodSpec` (when it realizes a Pod has been bound to the node it manages on the API Server) it goes through the following actions in order to get it running. First, it generates a `PodSandbox`, that is the environment within which the container is going to exist. Then, it begins to create the containers specified in the `PodSpec`.

In order to create/start a Container, the kubelet first needs to generate the required `ContainerConfig`:

Listing 8.8: ContainerConfig structure

```
type ContainerConfig struct {
        // Metadata of the container. This information will uniquely identify the
        // container, and the runtime should leverage this to ensure correct
        // operation. The runtime may also use this information to improve UX, such
        // as by constructing a readable name.
        Metadata *ContainerMetadata
        // Image to use.
        Image *ImageSpec
        // Command to execute (i.e., entrypoint for docker)
        Command []string
        // Args for the Command (i.e., command for docker)
        Args []string
        // Current working directory of the command.
        WorkingDir string
        // List of environment variables to set in the container.
        Envs []*KeyValue
        // Mounts for the container.
        Mounts []*Mount
        // Devices for the container.
        Devices []*Device
        // Key-value pairs that may be used to scope and select individual resources.
        Labels map[string]string
        // Unstructured key-value map that may be used by the kubelet to store and
```

```
        // retrieve arbitrary metadata.
        Annotations map[string]string
        // Path relative to PodSandboxConfig.LogDirectory for container to store
        // the log (STDOUT and STDERR) on the host.
        LogPath string
}
```

Kubelet uses this ContainerConfig in the (CRI) runtime service's `CreateContainer()` RPC call. We will only concern ourselves in the present with the device-related parts of this procedure. More specifically, a fairly lengthy chain of calls finally reaches the device manager subsystem of kubelet, which, as we mentioned is in charge of device exposure. The device manager's job is to provide the necessary `DeviceRunContainerOptions` that are going to be integrated into the CRI run options, in the fashion seen below:

Listing 8.9: ContainerConfig structure

```
func (cm *containerManagerImpl) GetResources(pod *v1.Pod, container *v1.Container) (*
    kubecontainer.RunContainerOptions, error) {
        opts := &kubecontainer.RunContainerOptions{}
        devOpts, err := cm.deviceManager.GetDeviceRunContainerOptions(pod, container)

        opts.Devices = append(opts.Devices, devOpts.Devices...)
        opts.Mounts = append(opts.Mounts, devOpts.Mounts...)
        opts.Envs = append(opts.Envs, devOpts.Envs...)
        opts.Annotations = append(opts.Annotations, devOpts.Annotations...)

        return opts, nil
}
```

We are now going to analyze how the device manager queries the device plugin for the aforementioned information. It iterates over the `Limits` section of the Container Spec, identifying device-plugin-managed resources. Then, it needs to decide which GPU-UUIDs it will request from the device-plugin. The ContainerSpec **only specifies the number of GPUs**. It is up to the device manager to decide which physical GPUs it will allocate to the container. It is important to stress here that the **device plugin has no knowledge of which GPUs have been allocated**. Its only reason for existence is to receive a GPU ID and provide instructions as to how to expose it to the running container. Back to our dilemma, the device manager can either ask the device plugin for advice on which GPU IDs to request, or just randomly pick the N first from its list of healthy devices. Once the device IDs have been decided, the kubelet makes the `Allocate()` RPC call to the device plugin. The way device plugin handles the request is fairly simple: Depending on the strategy used, it either returns Environment variables or a specific set of Mounts [83] that have a symbolic meaning. The decision on which information to return is based on the value of the device plugin's `DeviceListStrategy`:

Listing 8.10: DeviceListStrategy of nvidia-device-plugin

```
const (

        deviceListAsVolumeMountsHostPath          = "/dev/null"
        deviceListAsVolumeMountsContainerPathRoot = "/var/run/nvidia-container-devices"
)

func (m *NvidiaDevicePlugin) Allocate(ctx context.Context, reqs *pluginapi.AllocateRequest
    ) (*pluginapi.AllocateResponse, error) {
        responses := pluginapi.AllocateResponse{}
        for _, req := range reqs.ContainerRequests {
                response := pluginapi.ContainerAllocateResponse{}

                if *deviceListStrategyFlag == DeviceListStrategyEnvvar {
                        response.Envs = m.apiEnvs(m.deviceListEnvvar, req.DevicesIDs) // m
    .deviceListEnvvar="NVIDIA_VISIBLE_DEVICES"
                }
                if *deviceListStrategyFlag == DeviceListStrategyVolumeMounts {
                        response.Envs = m.apiEnvs(m.deviceListEnvvar, []string{
    deviceListAsVolumeMountsContainerPathRoot})
                        response.Mounts = m.apiMounts(req.DevicesIDs)
                }

                responses.ContainerResponses = append(responses.ContainerResponses, &
    response)
        }

        return &responses, nil
}
```

where the **corresponding method for generating EnvVars** is:

Listing 8.11: ContainerConfig structure

```
func (m *NvidiaDevicePlugin) apiEnvs(envvar string, filter []string) map[string]string {
        return map[string]string{
                envvar: strings.Join(filter, ","),
        }
}
```
```
for the case of volume mounts, the 'NVIDIA_VISIBLE_DEVICES' variable takes on a special
    value '"/var/run/nvidia-container-devices"' and the mounts are set as follows:
```
```go


func (m *NvidiaDevicePlugin) apiMounts(filter []string) []*pluginapi.Mount {
```

```
12          var mounts []*pluginapi.Mount
13
14          for _, id := range filter {
15                  mount := &pluginapi.Mount{
16                          HostPath:      deviceListAsVolumeMountsHostPath,  //   "/dev/null"
17                          ContainerPath: filepath.Join(
        deviceListAsVolumeMountsContainerPathRoot, id),
18      // basically "/var/run/nvidia-container-devices" + ID
19                  }
20                  mounts = append(mounts, mount)
21          }
22
23          return mounts
24  }
```

This sums up how nvidia-device-plugin facilitates the exposure of GPUs in containers.

Summary

Nvidia-container-toolkit (8.4.4) expects specific **Environment Variables** (or Volume Mounts) to be set in the OCI spec in order to expose the GPUs (mount the libraries, device files). Consequently, containers that run in pods on K8s and need access to a GPU have to somehow get those variables set. One solution was to have them hard-coded in the submitted images. Another solution, the more modern one, which also provides nicer manageability, is to use the device plugin approach, which does this job for us. So, we just request X nvidia GPUs in our `PodSpec` and the device plugin takes care of setting `NVIDIA_VISIBLE_DEVICES` for us and finally we not-so-magically get access to GPUs in our container.

Chapter 9

Existing Approaches

In this chapter, we analyze existing solutions towards GPU sharing in Kubernetes and highlight their pros and cons. We also discuss Replay Technique, which is a promising solution without a Kubernetes integration. However, after extensive testing, we conclude (with the authors' confirmation) that Replay Technique-based approaches do not work with modern CUDA versions and as such is not viable.

9.1 Public (Open-source) Solutions

Aliyun Scheduler extender

This solution from Alibaba Cloud [13] lifts the restriction of exclusive GPU assignment to Pods and implements bin packing based on the GPU memory requested in the Pod Spec. However, it does nothing beyond the scheduling phase which means that, during application execution, no measures are taken to isolate the GPU (memory) usage between the processes and OOM errors can still happen as in the stock NVIDIA case.

Contributions:

- enables users to specify a GPU memory request ("'aliyun.com/gpu-mem"') in their PodSpec
- extends K8s to allow bin-packing based on GPU memory; removes exclusive assignment of GPUs to Pods by using a custom device plugin

Pros:

- GPUs are no longer exclusively assigned to Pods
- Exposes a method to bin-pack based on GPU memory requests

Cons:

- **Does not enforce memory limits** on the Pods/processes; a Pod can allocate more memory than its request, breaking isolation
- GPUs can only be requested in the cluster through Aliyun's method
- `nvidia-device-plugin` (the de facto method of requesting GPUs) must be removed from the cluster; this breaks compatibility.

Kubeshare

Kubeshare [12] is a work from researchers at the National Tsing Hua University of Taiwan. The authors try to alleviate the lack of GPU sharing in Kubernetes while also providing a means to isolate the GPU usage between co-located processes. This is (to the best of our knowledge) the most complete existing solution to the problem we are trying to tackle, however it still leaves much room for GPU underutilization, as we will see below. The exact same approach is taken by Tencent's GPU Manager [87], albeit in a less sophisticated manner.

Contributions:

- enables users to specify GPU memory (hard limit) and compute requests and does bin-packing based on them
- provides a mechanism to enforce GPU memory requests; each process owns a specific amount of GPU memory that the user specifies in the Pod Spec - this is a hard limit that cannot change after launch.
- provides a mechanism to enforce GPU compute time-slicing (limits GPU command issuing from different process contexts using a Token-based approach; only the process/context holding the Token can issue work to the GPU; the Token is given to each process for a time-slice relative to their compute partition fraction)

Pros:

- enforces Memory limits; Processes cannot allocate more than their requested GPU memory. Thus, memory volume usage is isolated between co-located processes.
- makes a meaningful attempt at enforcing Compute Slicing, especially given our limited knowledge of the GPU context scheduling mechanism.
- Provides a way to schedule (binpack) on both compute and memory per GPU card.
- Ability to co-schedule on a specific gpu of a Node (device-plugins only treat GPUs are integer quantities by default).

.

Cons:

- the memory limit is hard; If the process tries to allocate more than the limit, it gets a CUDA OOM error and is terminated.
- Requested memory is not available to other processes, whether or not it is actually used by the Pod it's been assigned to.
- There is no study on the overhead (or the efficiency) of the Compute Slicing mechanism
- Kubeshare uses a custom resource (SharePod) to encapsulate the actual Pods that are to be run. We have identified two problems that this creates. Any component that creates Pods (e.g., Argo Workflows, StatefulSet controller) needs to be changed. As a result, integrating SharePod resources (and Kubeshare's approach) with existing logic requires major changes.

9.2 Commercial Solutions

Below we will very briefly go over some of the existing Commercial Solutions that offer the option to share a GPU between multiple user applications/processes/Pods. We mostly present this section to outline the importance of the problem, by showing the amount of effort (for some companies, it is their main service) that has been put into trying to solve GPU Sharing in Kubernetes.

Run:AI

Run:AI [17] offers many services to increase cluster GPU utilization. Among them is the option to share a GPU between Pods and speed up ML experimentation, the exact same as our goal.
Here is a rough overview of the company's relevant offerings:

- platform built on top of K8s
- offers dynamic scheduling (no static assignment of GPUs to user groups)
- guaranteed quotas for each job; jobs can use more GPUs if they are idle and also scale down when other users want to make use of their quota
- users must change their code to use Run:ai versions of Keras or Torch if they want elasticity
- can submit jobs via Run:ai CLI or directly to K8s
- supports fractional GPU allocation (memory) with enforcement (similar to Kubeshare)

Amazon Elastic Inference

Amazon's Elastic Inference [15] takes a totally different approach and virtualizes the GPU at the application level. This means that when a user calls a function from Amazon's custom TF library, they will be given the illusion that they are running on a GPU at the application level, and the call will be redirected to a back-end that handles maintaining this illusion. In the back-end, it is possible that Amazon uses some kind of GPU Sharing, to serve Elastic Inference calls from multiple instances.
Here is an overview of how it works:

- Only for inference and only for TF, PyTorch, MXNET
- users must work with custom ML framework images
- network-attached "GPU" (virtualized at application level, not CUDA)
- can only be attached during instance (EC2) launch

Bitfusion (VMware)

vSphere Bitfusion [16] exposes remote GPUs to CPU processes. It also offers the option to partition GPU memory between co-located processes. We won't consider this as an approach that tries to tackle the exact same problem as us, since it mainly handles remote GPU virtualization (similar to rCUDA [88] from the Polytechnic University of Valencia) which is a completely different area of study.
Here's a brief outline:

- Network attached GPUs (remote) ; interpose CUDA API calls and execute them remotely
- Supports fractional GPUs with memory limit
- K8s integration

Alibaba Cloud (Aliyun) cGPU

cGPU [14] is the enforcement mechanism that is paired with the Aliyun Scheduler Extender (which is open-source). This is an approach very similar to Kubeshare, enforcing hard memory limits on the applications. cGPU is only available when using Kubernetes in Alibaba Cloud's infrastructure.

9.3 Replay Technique: A non Kubernetes-integrated solution

9.3.1 Overview

We use the term "Replay Technique" to refer to a family of GPU mechanisms that are based on the same underlying concept: maintaining a ledger of CUDA calls made by the application to allow dynamic deletion/re-creation of the resources it owns. The goals of these types of approaches are twofold:

- Checkpoint/Restart support for GPU applications (NVCR [89]). Create a storable image of GPU memory to be used in conjunction with a CPU-process checkpointing tool (BLCR) to enable fault-tolerance. They originally developed this for the Tsubame supercomputer.
- Transparent suspend/resume to enable execution of multiple applications and possible migration to another GPU.(MobileCUDA [90])

For our problem, we are interested in the latter application of Replay Technique.
These are the steps that MobileCUDA takes to temporarily "freeze" an application and make the GPU memory available to other applications:

1. Send all data from Device to Host
2. Destroy CUDA context (for Driver API implementation).
3. Suspend Application
4. Reallocate CUDA resources (context, memory)
5. Restore the data on the device
6. Resume application execution

9.3.2 Main idea

These are the logical steps that describe the main logic behind Replay Technique:

- Observe that under certain circumstances CUDA returns the same device pointers when reallocating the same pointer to device memory after releasing it. We illustrate a trivial example in Figure 9.1. The `cudaFree()` call is part of the back-up mechanism. In this trivial case, there are no actions performed between the allocation of memory and the (transparent) release. When actions (CUDA calls) are done, as in a real user program, we try to nullify their side-effects on the CUDA memory allocation pointer logic so as to maintain the observed repetition of device pointers returned.
- Record a set of API calls that can potentially affect device memory allocation during the replay. Store the calls and their arguments in a `Replay Log`. (This must include memory allocation calls)
- During back-up/suspend:
  - Freeze the application (usually the trigger-point is in a CUDA call, so wait there)
  - Back up memory (cudaMemcpy to CPU RAM)
  - Free GPU memory allocations

- When restoring an application:
  - Replay all API calls in the Replay Log to ensure CUDA returns the same addresses to memory (re)allocation calls.
  - Copy memory back to the GPU
  - Let application continue

```
[#1] Device memory Pointer values:
            d_p = 0x7f6717600000
            d_c = 0x7f6714000000
cudaFree(d_c)
cudaFree(d_p)
[#2] Device memory Pointer values:
            d_p = 0x7f6717600000
            d_c = 0x7f6714000000
```

Figure 9.1: *Trivial example of Replay technique*

9.3.3 Problems with Replay Technique

After reallocation, the address of device memory allocations may differ from the previous one. Addresses cannot be opaque to the applications, they need to be directly accessible. (they are used as arguments when launching kernels, whose function prototypes are determined by the user) The **memory address of a device memory region must be the same as it was before back-up/checkpointing**.

Figure 9.2 presents a table from the NVCR [89] paper (2011) that showcases which CUDA data types can be opaque. For example, a `CUdevice` variable is only used in calls to the Runtime/Driver APIs, so an interposition library can transparently modify it before calling the real function.

CUDA Data Type	Actual Data Type	Explanation	Can be opaque?
CUdevice	int	Identifier to specify GPU device.	YES
CUdeviceptr	unsigned int	Pointer address for device memory.	NO
CUcontext	struct CUctx_st *	CUDA context must be created for each GPU device.	YES
CUmodule	struct CUmod_st *	Identifier of a CUDA module. CUDA module is created from PTX/CUBIN/fatCUBIN.	YES
CUfunction	struct CUfunc_st *	Identifier of a kernel function in CUDA modules.	YES
CUarray	struct CUarray_st *	Identifier of a special array for texture mapping.	YES
CUtexref	struct CUtexref_st *	Identifier of a texture reference in CUDA modules.	YES
CUevent	struct CUevent_st *	Identifier of CUDA event used for timing and synchronization.	YES
CUstream	struct CUstream_st *	Identifier of CUDA stream used to specify task dependencies.	YES

Figure 9.2: *NVCR: Opaque and non-opaque CUDA data types*

None of the implementations of Replay Technique (NVCR, MobileCUDA) work with today's CUDA versions, as replaying fails to yield the same CUDA memory pointer. We contacted the authors of MobileCUDA, Taichiro Suzuki and Prof. Akira Nukada at the University of Tsukuba, and they confirmed that this is the case. Prof. Nukada hinted that an updated replay-technique-based implementation is under way, however the sole focus of their work will be towards supporting fault-tolerance for long-running jobs and

not GPU sharing. As such, Replay Technique is not a valid solution to tackle the problem of GPU Sharing in multi-tenant scenarios, and will not be, for the foreseeable future.

Chapter 10

Our study of Unified Memory in multi-process scenarios

In this chapter we provide what is, to the best of our knowledge, the first study on the behavior of CUDA's Unified Memory in scenarios with multiple processes (contexts). We use `nvprof`, the NVIDIA Profiler [26] to measure Page Faults during GPU execution. We start by examining a simple case, where one process is executing on the GPU and memory is not oversubscribed and move on to more complex cases, where multiple processes' work overlaps and GPU memory is oversubscribed. We experimentally verify that CUDA uses an LRU eviction policy, as well as the fact that page-faults emanating from one CUDA context can evict pages belonging to another. This finding implies that each CUDA context can make use of all physical GPU memory, even if other contexts have active memory allocations. Our GPU Sharing mechanism is largely based on this observation.

10.1 Unified Memory refresher

Unified Memory (UM) is a hardware/software technology which allows applications to allocate data that can be accessed from code running on either CPU or GPU. It also allows GPU code to page-fault. When a page-fault occurs, the Unified Memory subsystem (kernel module) fetches the missing page to GPU memory and chooses a victim page to evict to host RAM. This means that when GPU physical memory is full and memory is oversubscribed (total memory allocations exceed physical size) the GPU memory acts as a sort of cache, with the system RAM as a backing store.

For normal (non-UM) CUDA memory allocations, every byte of virtual memory allocated must be backed by a byte of physical memory. An implication of this is that the sum of GPU memory allocations across all CUDA contexts can at most be equal to the physical GPU memory size. As an example, in the case of an NVIDIA P100 with 16 GB of memory, if one process (A) has allocated 10 GB via `cudaMalloc`, then another process (B) can at most allocate 6 GB of GPU memory.

Pre-Pascal Unified Memory behavior:

When an application launches a kernel, the Unified Memory subsystem migrates all Unified Memory allocations of the application context to GPU memory before the kernel starts execution. This is necessary because all Pre-Pascal generations of NVIDIA GPUs do not

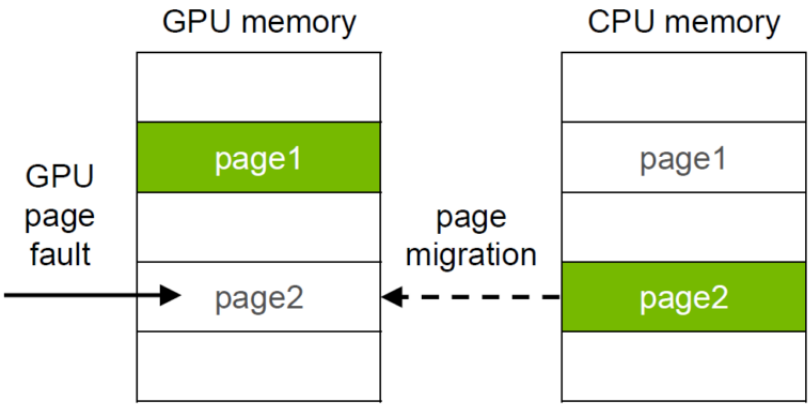

Figure 10.1: *Unified Memory Page Fault handling*

support page-faults during kernel execution.
Quoting the Nvidia Developer Blog [27]: "*On pre-Pascal GPUs, upon launching a kernel, the CUDA runtime must migrate all pages previously migrated to host memory or to another GPU back to the device memory of the device running the kernel. Since these older GPUs can't page fault, all data must be resident on the GPU just in case the kernel accesses it (even if it won't).*"

Post-Pascal Unified Memory behavior:

Data migration occurs via demand-paging. When the GPU device code attempts to access data in a particular page that is not resident in GPU memory, a page fault will occur and the warp will stall until the page is fetched to GPU memory.

For an excellent comparison of pre- and post- Pascal behavior of Unified Memory refer to the SO answer from Robert Crovella (Nvidia employee) [28].

10.2 Our evaluation program

We are working on a P100 (Pascal) GPU so our results apply to Pascal, Volta, Turing and Ampere architectures.

We ran our experiments on an Nvidia Tesla P100 with 16 GiB of physical memory. Note that not all 16 GiB are available to user processes. A small portion is set aside on the GPU for driver constructs such as contexts, performance counters and other undisclosed structures. In practice, the "program-usable" memory is ~15.5 GiB, fluctuating in ~100-200 MiB steps based on existing context count.

Our test program allocates `NUM_CHUNKS` memory regions of `CHUNK_SIZE` bytes. Then it initializes them on the CPU (host) and finally launches the `add<<<...>>>` kernel on each chunk, which increments each element's value by 1. It then waits for any keystroke before re-running the kernels, to test the execution behavior when data is already resident on the GPU.

Listing 10.1: Our evaluation program

```
#include <iostream>
#include <math.h>
#include <cuda_profiler_api.h>
#include <sys/time.h>
#define NUM_CHUNKS 5
#define CHUNK_SIZE 1L<<31
#define NUM_ITERATIONS 1
// define CUDA kernel which increments all the elements of an array
__global__  void add(uint64_t  n, char *x) {
      uint64_t index = blockIdx.x * blockDim.x + threadIdx.x;
      uint64_t stride = blockDim.x * gridDim.x;
      for ( size_t i = index; i < n; i += stride) {
        x[i] = x[i] + x[i];
      }
    }

int main(void) {
struct timeval tv1,tv2;
size_t N = CHUNK_SIZE;
char *x[NUM_CHUNKS];
// Allocate Unified Memory -- accessible from CPU or GPU
// Split the allocations into chunks of 2GiB
  for (int i=0; i<NUM_CHUNKS; i++) {
      cudaMallocManaged(&x[i], N*sizeof(char));
  }
  for (int i=0; i<NUM_CHUNKS; i++) {
    for (size_t k = 0; k < N; k++){
      x[i][k] = 1;
    }
  }
  // Launch kernel on the GPU
  int blockSize = 512;
  int numBlocks = N / blockSize;
  if ((N % blockSize) > 0 ) {numBlocks+=1;}

  for (int j=0; j<NUM_ITERATIONS; j++){
    gettimeofday(&tv1, NULL);
    for(int i=0; i<NUM_CHUNKS; i++) {
      add<<<numBlocks, blockSize>>>(N,x[i]);
    }
```

```
41
42    cudaDeviceSynchronize();
43
44    // Wait for input before doing a single rerun
45    getchar();
46
47    //for (int i=NUM_CHUNKS - 1; i>=0; i--){
48      for (int i=0; i<NUM_CHUNKS; i++) {
49        add<<<numBlocks, blockSize>>>(N,x[i]);
50    }
51    cudaDeviceSynchronize();
52
53  // Wait for GPU to finish before accessing on host
54    cudaDeviceSynchronize();
55
56    return 0;
57  }
```

10.3 Single Process - no oversubscription

We launch a single process that works with 5 chunks of 2 GiB each, totaling 10 GiB. Note that each kernel launch processes 2 GiB, for a total of 5 kernel launches to access the entire 10 GiB.

```
==6986== NVPROF is profiling process 6986, command: ./no_oversubscribe
[0] Total time = 3.823066 seconds

Single Rerun Total time = 0.124957 seconds
==6986== Profiling application: ./no_oversubscribe
==6986== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:  100.00%  3.94702s        10  394.70ms  24.853ms  775.04ms  add(unsigned long, char*)
      API calls:   92.88%  3.94699s         3  1.31566s  7.4750us  3.82272s  cudaDeviceSynchronize
                    7.08%  300.84ms         5  60.168ms  19.949us  300.67ms  cudaMallocManaged
                    0.02%  693.45us        10  69.344us  5.5990us  415.57us  cudaLaunchKernel
                    0.02%  683.31us         1  683.31us  683.31us  683.31us  cuDeviceTotalMem
                    0.01%  218.19us       101  2.1600us     173ns  91.642us  cuDeviceGetAttribute
                    0.00%  46.508us         1  46.508us  46.508us  46.508us  cuDeviceGetName
                    0.00%  2.9770us         1  2.9770us  2.9770us  2.9770us  cuDeviceGetPCIBusId
                    0.00%  2.3660us         3     788ns     228ns  1.8750us  cuDeviceGetCount
                    0.00%  1.5530us         2     776ns     248ns  1.3050us  cuDeviceGet
                    0.00%     849ns         1     849ns     849ns     849ns  cudaGetLastError
                    0.00%     340ns         1     340ns     340ns     340ns  cuDeviceGetUuid

==6986== Unified Memory profiling result:
Device "Tesla P100-PCIE-16GB (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
  111949  93.665KB  4.0000KB  0.9961MB  10.00000GB   1.217278s  Host To Device
   30732         -         -         -           -   3.719032s  Gpu page fault groups
Total CPU Page faults: 30720
```

Figure 10.2: *UM: Single Process, no oversubscription*

Since we first touch the memory on the CPU, a total of 10 GiB of `Host To Device` page

faults happen, when the data is afterwards accessed on the GPU during kernel execution.

Pay attention to the `Total Time` for the first execution: 3.82 sec. This includes the page-fault handling times. The second time around the execution takes only 0.12 sec, as data is now resident in GPU memory. In this case, the whole working set of the application fits in GPU memory.

10.4 Single Process - memory oversubscription

We launch a single process that works with 9 chunks of 2 GiB each, totaling 18 GiB. Note that each kernel launch processes 2 GiB, for a total of 9 kernel launches to access the entire 18 GiB.

```
==24074== NVPROF is profiling process 24074, command: ./oversubscribe_single
[0] Total time = 7.233462 seconds

Single Rerun Total time = 7.794011 seconds
==24074== Profiling application: ./oversubscribe_single
==24074== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:  100.00%  15.0272s        18  834.84ms  757.41ms  1.02775s  add(unsigned long, char*)
      API calls:   98.46%  15.0270s         3  5.00900s  8.9330us  7.79369s  cudaDeviceSynchronize
                    1.53%  233.91ms         9  25.990ms  16.123us  233.72ms  cudaMallocManaged
                    0.01%  775.19us         1  775.19us  775.19us  775.19us  cuDeviceTotalMem
                    0.00%  383.08us        18  21.282us  4.8160us  168.15us  cudaLaunchKernel
                    0.00%  170.80us       101  1.6910us     173ns  68.636us  cuDeviceGetAttribute
                    0.00%  27.187us         1  27.187us  27.187us  27.187us  cuDeviceGetName
                    0.00%  3.0880us         1  3.0880us  3.0880us  3.0880us  cuDeviceGetPCIBusId
                    0.00%  2.3130us         3     771ns     291ns  1.6730us  cuDeviceGetCount
                    0.00%  1.0120us         2     506ns     201ns     811ns  cuDeviceGet
                    0.00%     757ns         1     757ns     757ns     757ns  cudaGetLastError
                    0.00%     367ns         1     367ns     367ns     367ns  cuDeviceGetUuid

==24074== Unified Memory profiling result:
Device "Tesla P100-PCIE-16GB (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
  400281  94.305KB  4.0000KB  0.9961MB  36.00000GB   4.112297s  Host To Device
   10427  2.0000MB  2.0000MB  2.0000MB  20.36523GB   1.788062s  Device To Host
  110592         -         -         -           -  14.618042s  Gpu page fault groups
Total CPU Page faults: 55296
```

Figure 10.3: *UM: Single Process, memory oversubscription*

Judging by the 36 GiB of `Host to Device` and the 16 + 2 (+2)* GiB of `Device to Host` transfers, we can hypothesize that an LRU (Least Recently Used) eviction policy is employed, as depicted below in Figure 10.4:
[*] : Because the usable physical memory is *slightly less than 16 GiB (as we mention in the GPU Basics section)*, some additional page faults (chained 4 KB evictions within a 2 GiB chunk) occur.

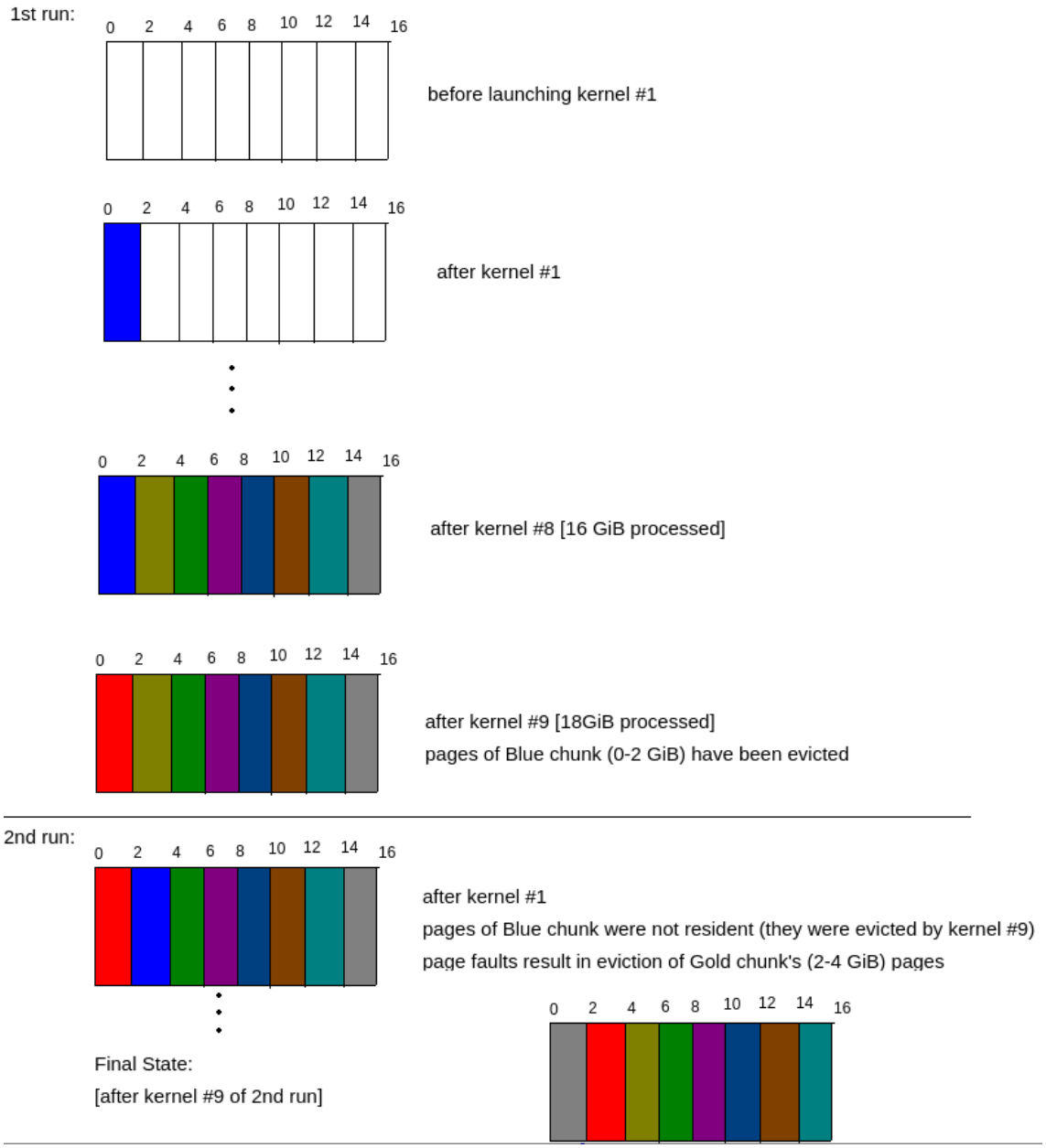

Figure 10.4: *UM: Eviction Sequence*

We can strengthen this assumption by launching the kernels of the 2nd run in reverse order [9-0] (Figure 10.5 below):

Note the reduced execution time of the second run. This time, the LRU replacement policy does not have such a detrimental effect, as only the last 4 GiB of accesses trigger page faults and subsequent page evictions of data that the kernel will access immediately afterwards. Again, due to capacity being slightly less than 16 GiB, we get some additional page faults that do not match with what is presented in the diagram.

```
==28960== NVPROF is profiling process 28960, command: ./oversubscribe_single_reverse_second
[0] Total time = 6.821396 seconds

Single Rerun Total time = 1.890170 seconds
==28960== Profiling application: ./oversubscribe_single_reverse_second
==28960== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:  100.00%  8.71074s        18  483.93ms  24.853ms  981.52ms  add(unsigned long, char*)
      API calls:   97.34%  8.71066s         3  2.90355s  11.548us  6.82106s  cudaDeviceSynchronize
                    2.64%  236.12ms         9  26.235ms  16.856us  235.92ms  cudaMallocManaged
                    0.01%  840.92us         1  840.92us  840.92us  840.92us  cuDeviceTotalMem
                    0.01%  633.78us        18  35.209us  5.2970us  292.22us  cudaLaunchKernel
                    0.00%  251.03us       101  2.4850us     166ns  104.14us  cuDeviceGetAttribute
                    0.00%  34.303us         1  34.303us  34.303us  34.303us  cuDeviceGetName
                    0.00%  2.5480us         1  2.5480us  2.5480us  2.5480us  cuDeviceGetPCIBusId
                    0.00%  1.6270us         3     542ns     207ns  1.1740us  cuDeviceGetCount
                    0.00%     961ns         2     480ns     202ns     759ns  cuDeviceGet
                    0.00%     642ns         1     642ns     642ns     642ns  cudaGetLastError
                    0.00%     332ns         1     332ns     332ns     332ns  cuDeviceGetUuid

==28960== Unified Memory profiling result:
Device "Tesla P100-PCIE-16GB (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
  246520  93.577KB  4.0000KB  0.9961MB  22.00000GB   2.509481s  Host To Device
    3259  2.0000MB  2.0000MB  2.0000MB  6.365234GB  560.0505ms  Device To Host
   67591         -         -         -           -   8.289696s  Gpu page fault groups
Total CPU Page faults: 55296
```

Figure 10.5: *UM: Second Iteration in Reverse Order*

10.5 Two Processes - no oversubscription

We run two processes, A & B simultaneously. Each one of them only generate 6 GiB of `Host to Device` page-faults, equal to the size of data they access. Since both working sets fit in physical GPU memory, there are no further faults and the re-runs take only ~70ms, as we can see in Figure 10.6.

```
==29882== NVPROF is profiling process 29882, command: ./increment_5g.cu
[0] Total time = 2.285040 seconds

Single Rerun Total time = 0.074728 seconds
==29882== Unified Memory profiling result:
Device "Tesla P100-PCIE-16GB (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
   67569  93.111KB  4.0000KB  0.9922MB  6.000000GB  672.5769ms  Host To Device
   18433         -         -         -           -   2.216746s  Gpu page fault gr
ps
Total CPU Page faults: 18432
```

A

```
==29893== NVPROF is profiling process 29893, command: ./increment_5g.cu
[0] Total time = 2.305340 seconds

Single Rerun Total time = 0.069259 seconds
==29893== Unified Memory profiling result:
Device "Tesla P100-PCIE-16GB (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
   67171  93.663KB  4.0000KB  0.9883MB  6.000000GB  689.1920ms  Host To Device
   18433         -         -         -           -   2.238830s  Gpu page fault g
ps
Total CPU Page faults: 18432
```

B

Figure 10.6: *UM: Two Processes, no oversubscription*

10.6 Two Processes - memory oversubscription

We execute A & B in the manner listed below and then state our observations:

1. A launches 5 kernels, each of which increments 2 GiB for a total of 10 GiB.
2. B launches 5 kernels, each of which increments 2 GiB for a total of 10 GiB. This results in eviction of ~4.6 GiB pages belonging to A (remember actual GPU memory capacity is less than 16 GiB). LRU suggests these are regions [0, 4.6] GiB
3. B terminates. Its context is destroyed and it longer owns any GPU memory.
4. A re-runs the set of 5 kernels. Accesses for the region [0,4.6] GiB result in page-faults, because B has already evicted this region in step 2. However, those page faults don't result in eviction of any frame, as B has terminated.

Let's calculate the totals:
Process **A**:

- 10 GiB mandatory Host to Device transfers for step #1
- 4.6 GiB Device to Host transfers after it gets evicted by B on step #2
- 4.6 GiB Host to Device transfers to re-fetch and access the evicted pages on step #4
- **Total**: 14.6 GiB HtoD and 4.6 GiB DtoH

Process **B**:

- 10 GiB mandatory Host to Device transfers for step #2
- **Total**: 10 GiB HtoD

These preliminary assumptions are in line with the results we obtained from `nvprof` and show in Figure 10.7 below.

A

```
==31439== Unified Memory profiling result:
Device "Tesla P100-PCIE-16GB (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
  164692  93.103KB  4.0000KB  0.9922MB  14.62305GB   1.674699s  Host To Device
    2367  2.0000MB  2.0000MB  2.0000MB  4.623047GB  402.7704ms  Device To Host
   44927         -         -         -           -   4.839089s  Gpu page fault grou
ps
Total CPU Page faults: 30720
```

B

```
==31740== Unified Memory profiling result:
Device "Tesla P100-PCIE-16GB (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
  111810  93.781KB  4.0000KB  0.9961MB  10.00000GB   1.136773s  Host To Device
   30721         -         -         -           -   4.004468s  Gpu page fault grou
ps
Total CPU Page faults: 30720
```

Figure 10.7: *UM: Two Processes, memory oversubscription*

The main takeaway here is that when handling a page-fault, frames (resident pages) are **evicted from any context** on the GPU.

10.7 Two Processes - Negative Memory Interference (thrashing)

When A is running alone with a working set of 10 GiB, then only the first iteration (the one generating page faults and fetching data to GPU memory) has a long execution time. Subsequent iterations find the data resident in GPU memory and only take milliseconds to execute. Let's increase the number of "increment" iterations to 100 and see how A behaves when running alone (Figure 10.8):

```
==1437== NVPROF is profiling process 1437, command: ./thrashing
Initialization on host complete. Press any key to run GPU kernels

[0] Total time = 3.515558 seconds
[1] Total time = 0.115255 seconds
[2] Total time = 0.115207 seconds
[3] Total time = 0.115230 seconds
[4] Total time = 0.115182 seconds
[5] Total time = 0.115192 seconds
[6] Total time = 0.115204 seconds
[7] Total time = 0.115179 seconds
[8] Total time = 0.115178 seconds
[9] Total time = 0.115201 seconds
[10] Total time = 0.115181 seconds
[11] Total time = 0.115183 seconds
[12] Total time = 0.115205 seconds
[13] Total time = 0.115182 seconds
==1535== Unified Memory profiling result:
Device "Tesla P100-PCIE-16GB (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
  113325  92.527KB  4.0000KB  0.9961MB  10.00000GB   1.129316s  Host To Device
   30720         -         -         -           -   3.741908s  Gpu page fault grou
ps
Total CPU Page faults: 30720
```

Figure 10.8: *UM: One process, 100 iterations*

The results are in line with our prediction. Subsequent runs after the first only take ~115 ms, as data is already resident on the GPU and no page faults occur.

Let's now examine what happens when A and B execute 100 iterations of "increment" in parallel: In Figure 10.9 we present the results of nvprof for process A. We interrupted it after 5 iterations as it was going to take an extremely long time to complete. The results were the same for process B.

We observe a ~250x increase in execution time. Because the kernels from A & B are "executing" at the same time (albeit the GPU switches between the contexts in a time-sliced manner) and memory is oversubscribed (their working sets as a whole don't fit in GPU memory) there is extreme contention for the physical GPU memory that leads to thrashing, where page-fault handling slows down meaningful computations almost to a complete halt.

```
==1629== NVPROF is profiling process 1629, command: ./thrashing
Initialization on host complete. Press any key to run GPU kernels

[0] Total time = 9.793950 seconds
[1] Total time = 25.965820 seconds
[2] Total time = 24.063141 seconds
[3] Total time = 25.870272 seconds
[4] Total time = 24.045416 seconds
[5] Total time = 23.742236 seconds
^C==1629== Profiling application: ./thrashing
==1629== Unified Memory profiling result:
Device "Tesla P100-PCIE-16GB (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
  903833  79.493KB  4.0000KB  0.9961MB  68.52057GB   9.834552s  Host To Device
   31177  2.0000MB  2.0000MB  2.0000MB  60.89258GB   6.481411s  Device To Host
  210598         -         -         -           -  66.366487s  Gpu page fault g
ps
Total CPU Page faults: 30720
```

Figure 10.9: *UM: Thrashing, memory oversubscription*

10.8 Prefetching Memory to the GPU

When using Unified Memory, we can explicitly prefetch regions to GPU memory by using `cudaMemPrefetchAsync()` [29]. This functions in a similar way to how we would call `cudaMemcpy()` to move data to the GPU in a non-unified scenario. By prefetching data to the GPU, we avoid triggering multiple page-faults and serving them independently, which is useful in cases where we know that a region is going to be needed by the application in advance. Below we show the modifications we made to the example program so as to prefetch memory to the GPU before launching the computational kernels.

Listing 10.2: Prefetching memory to the GPU

```
1 // Prefetch GPU memory
2 int device = -1;
3 cudaGetDevice(&device);
4 for (int i = 0; i < NUM_CHUNKS; i++) {
5   cudaMemPrefetchAsync(x[i], N*sizeof(char), device, NULL);
6 }
```

10.8.1 Revisiting the single process case

Let's revisit the single process case with no oversubscription (10 GiB): We examine the results of nvprof, shown in Figure 10.10. As data is already resident on the GPU when execution starts, there are no page-faults on the GPU (the prefetches are handled in 2 MB chunks) and execution time is minimal (115 ms even for the first kernel; compare the to the single process case without prefetch in the beginning of this chapter). We should not forget that the prefetching procedure still takes some time, but it uses less PCIe transactions as it transfers data in bulk, so the total execution time is shortened.

```
==1787== NVPROF is profiling process 1787, command: ./prefetch
Initialization on host complete.
Prefetching Complete. Press any key to run the GPU kernels

[0] Total time = 0.115318 seconds
==1787== Unified Memory profiling result:
Device "Tesla P100-PCIE-16GB (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
    5120  2.0000MB  2.0000MB  2.0000MB  10.00000GB   1.178874s  Host To Device
Total CPU Page faults: 30720
```

Figure 10.10: *UM: Prefetching Memory to the GPU*

10.8.2 Eviction during Prefetching

We observe that Prefetching follows the same logic as normal page-fault handling. It evicts frames in an LRU manner and from any context. To verify the behavior, we create a specific scenario to test against. In this experiment:

1. A runs an iteration on 10 GiB
2. B prefetches 10 GiB to GPU memory and runs an iteration
3. A runs a second iteration

We show the output of the above experiment below (Figure 10.11):

```
grgalex@grgalex-gpu-devel-p100:~/DEMO_3_5/PREFETCH$ nvprof ./increment_10g
sizeof(char) = 1
==2394== NVPROF is profiling process 2394, command: ./increment_10g
[0] Total time = 4.060820 seconds
Press any key to re-run.
B prefetches 10 GiB
Single Rerun Total time = 9.895979 seconds
==2394== Unified Memory profiling result:
Device "Tesla P100-PCIE-16GB (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
  222234  94.366KB  4.0000KB  0.9961MB  20.00000GB   2.725839s  Host To Device
    5120  2.0000MB  2.0000MB  2.0000MB  10.00000GB   1.025000s  Device To Host
   61441         -         -         -           -  13.738795s  Gpu page fault grou
ps
Total CPU Page faults: 30720
```

A

Figure 10.11: *UM: Eviction during Prefetching*

Let's analyze what happens, step by step:

1. A brings in 10 GiB pages
2. B prefetches 10 GiB; it evicts ~5 GiB from A (specifically regions [0-5G] of A)
3. A re-runs; its first kernel tries to access its region [0-2G]; it is no longer resident
4. A gradually "evicts" its own region [5-7G], based on the LRU policy
5. this chain reaction eventually leads to another 10 GiB of DtoH page-faults for a total of 20 GiB. This is because A needs to access the pages it already evicted in the previous step.

The main takeaway here is that **Prefetching is treated equally (w.r.t. LRU, eviction)** to page transfers that originate from page faults.

Chapter 11

Our Approach

Having seen how Unified Memory behaves under multi-tenancy in the previous Chapter, we slowly devise our mechanism. Our goal is to enable two or more user applications to use the same GPU on Kubernetes where each user can utilize the whole GPU memory. Our main use-case is that of interactive ML development in Jupyter Notebooks, where GPU computations come in bursts and there are idle periods. As we will see down the road, however, our mechanism is applicable to many more scenarios.
Let's outline the steps we took to create our GPU Sharing mechanism:

1. Provide a mechanism to **convert all CUDA memory allocations of an application to Managed (Unified Memory)** by hooking the CUDA APIs.
2. Ensure **this mechanism does not break applications**
3. **Measure the overhead an application suffers stemming from its use of Unified Memory** (when running alone). Compare the non-unified to the unified results on a set of ML benchmarks.
4. **Prove and measure thrashing** when 2+ users run their workloads at the same time (GPU bursts overlap).
5. **Eliminate thrashing** by implementing a time-slicing mechanism (daemon/scheduler). The time quantum must be long enough to justify the preemption (memory swapping) cost.
6. **Integrate the whole mechanism with Kubernetes** (alexo-device-plugin)

11.1 Transparently convert all memory allocation calls to Unified Memory

[While the focus of this section is hooking CUDA API calls, all the information presented applies to hooking any function and covers all options of linking/loading.]

We want to convert allocation calls `cudaMalloc()`, `cuMemAlloc()` to their Managed counterparts `cudaMallocManaged()`, `cuMemAllocManaged()` in order to achieve the following:

- Enable oversubscription of total GPU memory
  - Memory allocations no longer have a 1-1 correspondence with physical GPU memory
- Enable each co-located application to utilize all physical GPU memory
  - each application can evict pages from other (idle) applications when transferring data from host RAM to GPU.

We analyze the behavior of co-located applications that use Unified Memory with regard to eviction in the previous chapter.

11.1.1 Summary

In order to hook a function and force the user application to call our custom version we must:

- Create a shared library that defines a function with the same name (e.g. `cudaMalloc`). This internally calls the "real" function. The pointer to the real function is obtained through `dlopen()` + `dlsym()`
- Make sure the application calls our version of the function. For simple cases, we achieve this through the `LD_PRELOAD` trick [30], making sure the symbol resolves to our function by linking our shared library to the application before all others. For more complex cases (dynamically loaded libs) we must also hook `dlsym()` because linking before is not enough on its own.

11.1.1.1 Types of CUDA applications and linking options

- Pure Driver API (only calls Driver API functions)
  - dynamically linked to `libcuda.so`
- Pure Runtime API (only calls Runtime API functions)
  - statically linked to `libcudart.a`
  - dynamically linked to `libcudart.so`
- Mixed API
  - statically linked Runtime (`libcudart.a`), dynamically linked Driver (`libcuda.so`)
  - dynamically linked Runtime (`libcudart.so`), dynamically linked Driver (`libcuda.so`)
  - dynamically linked Runtime (`libcudart.so`), dynamically loaded Driver (`libcuda.so`)
  - dynamically **loaded** Runtime (`libcudart.so`), dynamically **loaded** Driver (`libcuda.so`)

Tensorflow uses the **Mixed API** option (d). Pytorch uses **Mixed API** option (c).
Refer to the excellent StackOverflow answer [91] for an explanation of the differences between dynamically linked and dynamically loaded shared libraries.

11.1.2 How we hook dynamically linked Runtime API calls: cudaMalloc()

The mechanism relies on the `LD_PRELOAD` environment variable and a shared C library (we call it `libunified.so`) to hook a function and we outline the steps taken below:

1. Define cudaMalloc() in our library. Because of `LD_PRELOAD`, the dynamic linker binds symbols (in this case the `cudaMalloc` function) provided by `libunified.so` before other dynamic libraries (e.g. `libcudart.so`). As a result, the user application calls our version of `cudaMalloc`. The `void *devPtr` device pointer is set by `cudaMallocManaged` instead of `cudaMalloc`.
2. Since we want to convert application calls of `cudaMalloc` into calls of `cudaMallocManaged`, our (`libunified.so`) version of `cudaMalloc` uses `dlsym()` to obtain the address of `cudaMallocManaged` in `libcudart.so` and call it.
3. Our `libunified.so`'s `cudaMalloc` returns the error value of `libcudart.so`'s 'cudaMallocManaged'.

```
cudaError_t cudaMalloc ( void** devPtr, size_t size ) {
  cudaError_t result = cudaSuccess;

// obtain the pointer to the real cudaMallocManaged (see below)

  result = ((cudaError_t (*) ( void** devPtr, size_t size, unsigned int flags ) )
    real_cudaMallocManaged) (devPtr, size, cudaMemAttachGlobal);

  return result;
}
```

11.1.2.1 How to obtain the address of cudaMallocManaged

When a hook function `foo` wants to call the "real" version as part of its body (for example if we wanted to hook `cudaMalloc()` just to log the arguments and return value), then the address of the real function (e.g. `libcudart.so`'s `cudaMalloc`) can be obtained by calling `dlsym(RTLD_NEXT, "foo")`. Quoting the man page on `dlsym` [92]: *"[...] The latter [RTLD_NEXT] will find the next occurrence of a function in the search order after the current library. This allows one to provide a wrapper around a function in another shared library."*

The `RTLD_NEXT` flag of `dlsym` does not imply that the symbol to be looked up must have been defined in the current object. See the paragraph "Symbol Fishing" of [30] for more information. In `cudaMalloc` of our `libunified.so` we want to obtain the address of the function `cudaMallocManaged()`:

One option is to use `dlsym(RTLD_NEXT, "cudaMallocManaged")` and it works correctly. However, the use of `RTLD_NEXT` has some dangers due to its dependence on load order which are detailed in this great OptumSoft article [93]. The following code snippet showcases the `RTLD_NEXT` approach:

```
void *real_cudaMallocManaged;
real_cudaMallocManaged = (void *)real_dlsym(RTLD_NEXT, CUDA_SYMBOL_STRING(
    cudaMallocManaged));
result = ((cudaError_t (*) ( void** devPtr, size_t size, unsigned int  flags ) )
    real_cudaMallocManaged) (devPtr, size, cudaMemAttachGlobal);
```

Another option, the safer one, is to explicitly search `libcudart.so` for the address of `cudaMallocManaged`. The following code snippet showcases this:

```
void *cudart_handle;
void *real_cudaMallocManaged;
cudart_handle = dlopen("libcudart.so", RTLD_LAZY);
real_cudaMallocManaged = (void *)real_dlsym(cudart_handle, CUDA_SYMBOL_STRING(
    cudaMallocManaged));
result = ((cudaError_t (*) ( void** devPtr, size_t size, unsigned int  flags ) )
    real_cudaMallocManaged) (devPtr, size, cudaMemAttachGlobal);
dlclose(cudart_handle);
```

By only assigning the pointer value to `real_cudaMallocManaged` once, we can optimize the above implementation, as shown below [this is our final version]:

```
1  void *real_cudaMallocManaged = NULL;  //global
2
3  cudaError_t cudaMalloc ( void** devPtr, size_t size ) {
4    cudaError_t result = cudaSuccess;
5
6    if (!real_cuMemAllocManaged) {
7      void *cudart_handle;
```

```
8       cudart_handle = dlopen("libcudart.so", RTLD_LAZY);
9       real_cudaMallocManaged = (void *)real_dlsym( cudart_handle, CUDA_SYMBOL_STRING
        (cudaMallocManaged));
10      dlclose(cudart_handle);
11      // just decrements the handle reference counter, does not unload libcudart
12    }
13
14    result = ((cudaError_t (*) ( void** devPtr, size_t size,   unsigned int  flags )
         )real_cudaMallocManaged) (devPtr,   size, cudaMemAttachGlobal);
15
16    return result;
```

We verify through the experiment below that the returned handle points to the same `libcudart.so` that the program has been dynamically linked to:

```
void *cudart_handle, *cudart_handle2;
cudart_handle = dlopen("libcudart.so", RTLD_LAZY | RTLD_NOLOAD);
cudart_handle2 = dlopen("libcudart.so", RTLD_LAZY);
```

We **verify that the returned handles are identical** through gdb (or printing):

```
dlsym Value returned $1 = (void *) 0x7ffff7fce530
dlsym Value returned $2 = (void *) 0x7ffff7fce530
```

We are therefore very certain that our way of obtaining the address of `cudaMallocManaged` is optimal.

11.1.3 How we hook dynamically linked Driver API calls: cuMemAlloc()

Exactly the same procedure as above where:

- instead of `cudaMalloc` we hook `cuMemAlloc`
- instead of `cudaMallocManaged` we call `cuMemAllocManaged` and
- instead of `libcudart.so` we use `libcuda.so`

11.1.4 How we hook dynamically loaded API calls

In cases where the user application obtains the function pointers to CUDA functions via `dlopen()` + `dlsym()`, `LD_PRELOAD` no longer causes our versions of the functions to be called. In this scenario we need to do the following:

1. hook `dlsym()` and have it return our versions of the function we want to interpose (cuMemAlloc, cudaMalloc)
2. Provide a `real_dlsym()` so our hooked functions can obtain the pointers to and call the real versions once they record useful information.

In our case, since we eventually make a call to another symbol (cuMemAllocManaged, cudaMallocManaged) this is not strictly necessary. Nevertheless, further down the road we are going to do bookkeeping and actual function interposition (where our hook invokes the very same function of the original library).

```
1  #define _GNU_SOURCE
2  // defining _GNU_SOURCE enables us to call __libc_dlsym()
3  // includes omitted to reduce length
4  void *__libc_dlsym(void *map, const char *name);
5  void *__libc_dlopen_mode(const char *name, int mode);
6  typedef void *(*fnDlsym)(void *, const char *);
7
8  static void *real_dlsym(void *handle, const char *symbol) {
9    static fnDlsym internal_dlsym ;
10   internal_dlsym = (fnDlsym)__libc_dlsym(__libc_dlopen_mode("libdl.so.2",
       RTLD_LAZY), "dlsym");
11   return (*internal_dlsym)(handle, symbol);
12 }
13
14 /*
15  ** interposed functions
16  */
17 void *dlsym(void *handle, const char *symbol) {
18 // Return early if it is not a CUDA driver symbol
19   if (strncmp(symbol, "cu", 2) != 0) {
20     return (real_dlsym(handle, symbol));
21   }
22
23   if (strcmp(symbol, CUDA_SYMBOL_STRING(cuMemAlloc)) == 0) {
24     return (void *)(&cuMemAlloc);
25   }
26
27   else if (strcmp(symbol, CUDA_SYMBOL_STRING(cudaMalloc)) == 0) {
28     return (void *)(&cudaMalloc);
29   }
30   return (real_dlsym(handle, symbol));
31 }
```

This hooking method was adapted from the CUDA sample cuHook [94]. It is also akin to the method used in Gemini (Kubeshare) [95].

After the user application has obtained a pointer to our versions of the functions we want to hook, the procedure onwards is exactly the same as in the cases of dynamically linked API calls (do custom work, obtain original function pointer, call it, return).

11.1.5 How we hook statically linked Runtime API calls

The Runtime API internally calls Driver API functions to implement most of its functionalities. Specifically, for memory allocation, `cudaMalloc()` internally calls `cuMemAlloc()`. The Runtime API uses `dlopen()` + `dlsym()` to extract symbol locations of the Driver API functions.
Here is a **GDB backtrace** that proves the above statement.
[Notice the call to `dlopen("libcuda.so.1")`].

```
#0  __dlopen (file=0x5555555b23b6 "libcuda.so.1", mode=2) at dlopen.c:75 [mode =
    RTLD_NOW]
#1  0x000055555556e146 in cudart::globalState::loadDriverInternal() ()
#2  0x000055555556f173 in cudart::__loadDriverInternalUtil() ()
#3  0x00007ffff7f89997 in __pthread_once_slow (once_control=0x5555555c8830 <cudart::
    globalState::loadDriver()::loadDriverControl>,
    init_routine=0x55555556f150 <cudart::__loadDriverInternalUtil()>) at pthread_once.
    c:116
#4  0x00005555555abca9 in cudart::cuosOnce(int*, void (*)()) ()
#5  0x00005555555705a3 in cudart::globalState::initializeDriver() ()
#6  0x0000555555594102 in cudaMalloc ()
#7  0x000055555555aac2 in main ()
```

When it is possible, we choose to treat `cudaMalloc` and `cuMemAlloc` distinctly with regards to conversion. In the case above, where CUDA Runtime is statically linked, our only choice is to hook `cuMemAlloc` and convert it to `cuMemAllocManaged`. This falls under the case "How we hook dynamically loaded API" calls above. We haven't noticed any deviation in behavior when only replacing `cuMemAlloc` and ignoring `cudaMalloc`, however we still deal with them separately when we can, just to be on the safer side.

We can now convert any CUDA application's memory allocation calls to Unified transparently.

11.1.6 An example: Tensorflow

As we mentioned in the introduction of this section, Tensorflow dynamically loads both CUDA Runtime as well as Driver API libraries. It extracts the pointer to CUDA Driver and Runtime API functions manually by using `dlopen()` and `dlsym()`. Let's examine the specific parts of its source code which handle this procedure.

- `cuMemAlloc` wrapper calls `LoadSymbol()`:

```
1  CUresult CUDAAPI cuMemAlloc(CUdeviceptr *dptr, size_t bytesize) {
2    using FuncPtr = CUresult(CUDAAPI *)(CUdeviceptr *, size_t);
3    static auto func_ptr = LoadSymbol<FuncPtr>("cuMemAlloc_v2");
4    if (!func_ptr) return GetSymbolNotFoundError();
5    return func_ptr(dptr, bytesize);
```

```
6 }
```

- `LoadSymbol()` eventually calls `GetSymbolFromLibrary()` which uses `dlsym()` to extract the symbol location:

```
1  Status GetSymbolFromLibrary(void* handle, const char* symbol_name,
2                              void** symbol) {
3    // Check that the handle is not NULL to avoid dlsym's RTLD_DEFAULT
       behavior.
4    if (!handle) {
5      *symbol = nullptr;
6    } else {
7      *symbol = dlsym(handle, symbol_name);
8    }
9    if (!*symbol) {
10     return errors::NotFound(dlerror());
11   }
12   return Status::OK();
13 }
```

- We also provide a GDB trace which showcases the `dlopen()` call that precedes the various `dlsym` calls for the Driver API:

```
#0  __dlopen (file=0x7fffffffc3e0 "libcuda.so.1", mode=2) at dlopen.c:75
#1  0x00007fff9d8136fd in tensorflow::internal::LoadLibrary(char const*, void**)
     () from libtensorflow_framework.so.2
#2  0x00007fff9d80de76 in tensorflow::(anonymous namespace)::PosixEnv::
    LoadLibrary(char const*, void**) () from libtensorflow_framework.so.2
#3  0x00007fffaaaa2814 in stream_executor::internal::GetDsoHandle() from
    _pywrap_tensorflow_internal.so
#4  0x00007fffaaaa48d4 in stream_executor::internal::DsoLoader::
    GetCudaDriverDsoHandle() from _pywrap_tensorflow_internal.so
#5  0x00007fffaaa990bd in LoadSymbol() from _pywrap_tensorflow_internal.so
#6  0x00007fffaaa9935c in cuInit () from _pywrap_tensorflow_internal.so
#7  0x00007fffaa96f58f in stream_executor::gpu::InternalInit() () from
    _pywrap_tensorflow_internal.so
#8  0x00007fffaa96f701 in stream_executor::gpu::GpuDriver::Init() from
    _pywrap_tensorflow_internal.so
```

We can now move on to testing how our mechanism works and ensuring it doesn't break user applications.

11.2 Validate the stability of the conversion mechanism

We must ensure that any application whose memory allocations we transparently convert to Managed (Unified Memory) by the mechanism presented in the previous chapter:

- does not encounter any errors stemming from its use of Unified Memory
- has the same execution results as the original

Since we cannot mathematically prove the correctness of our conversion mechanism, the only option is to run a wide a variety of applications under the conversion scheme and observe their execution behavior. To this end we conduct a series of experiments targeting:

- Pure CUDA applications
- Tensorflow programs
- PyTorch programs

More specifically, we tested the following:

- Official CUDA Samples [31]
- Official Tensorflow Benchmarks [32]
- Official PyTorch Benchmarks [33]
- AI-Benchmark Suite [ETH Zurich] (Tensorflow) [34]
- Altis GPU Benchmarks [UT Austin] (state-of-the-art CUDA Benchmark Suite) [35]

Our experimental results strengthen the assumption that our converting all memory allocation calls to Unified Memory ones maintains application execution correctness. All experiments were conducted on a Google Cloud VM with an Intel Xeon 8-core CPU, 52 GiB RAM and an Nvidia Tesla P100 GPU with 16 GB of memory.

11.2.1 CUDA Samples

The CUDA Samples consist of 50 applications covering a wide range of CUDA functionalities. We clone the Samples' Github Repository [31], enter the source directory and run `make`. Then we enter each application directory and run every program, once in its original state and once with our conversion hook library enabled.

For example, to run 'matrixMul' we enter the matrixMul directory and execute: `./matrixMul` and `LD_PRELOAD=libunified.so ./matrixMul`

Some tests are not supported on the P100, as they require more recent hardware features such as Tensor Cores, which are present on Volta, Turing or Ampere GPUs.

Of the tests that can run on P100, all were successful except for `simpleIPC`. `simpleIPC` uses the CUDA Interprocess Communication family of functions. They are used to expose a memory region from one CUDA context to another, and do not function for `cudaMallocManaged` allocations, as per Nvidia documentation [36]. After examining the source code carefully, we can say that Tensorflow and Pytorch do not use CUDA IPC at all.

11.2.2 Official Tensorflow Benchmarks

We clone the official Tensorflow Benchmark repository ([32]), enter the `scripts/tf_cnn_benchmarks` directory and execute:

- `python run_test.py --full_tests`
- `LD\_PRELOAD=libunified.so python run_test.py --full_tests`

In both cases, all 232 tests complete successfully. No explicit output validation is performed by the TF benchmarks, however the model accuracy is tested and, since there are no errors and in most cases the random seeds are fixed, we can conclude that correctness is maintained.

11.2.3 Official PyTorch Benchmarks

We clone the Pytorch Benchmark repository ([33]) and run only the single node benchmarks:

- `pytest --ignore-machine-config test_bench.py`
- `LD_PRELOAD=libunified.so pytest --ignore-machine-config test_bench.py`

All 104 tests complete successfully.

11.2.4 AI-Benchmark

AI-Benchmark is an ML (Tensorflow) benchmarking suite developed by Andrey Ignatov [37] at ETH Zurich, with an initial goal of testing the performance of Smartphones/Mobile SoCs. We use the desktop GPU version for our testing, following the instructions on the AI-Benchmark website [34]. We install the 'ai-benchmark' Python package and execute the benchmark by running the following script (`run_ai_benchmark`):

```
import ai_benchmark
benchmark = ai_benchmark.AIBenchmark()
results = benchmark.run(precision="high")
```

We set 'precision="high"' in order to execute 10 times more runs than regular execution, as highlighted in the project's documentation

11.2.5 Altis GPU Benchmarks

`Altis` is a state-of-the-art CUDA Benchmarking suite developed by the SCEA Lab at UT Austin. The paper was released in 2020 [35] and the project is open-source [38]. The Altis benchmark suite is divided into three levels. Each level represents benchmark applications whose behaviors of interest range from low-level characteristics such as bus bandwidth to end-to-end performance of real world applications. This categorization is adopted from the Scalable Heterogeneous Computing (SHOC) [39] Benchmark Suite. The level structure is:

https://pypi.org/project/ai-benchmark/

- Level 0: Measures low level characteristics of the targeted hardware. This level includes simple benchmarks such as maxflop and bus bandwidth.
- Level 1: Includes basic parallel algorithms which are commonly seen in parallel computing and used in kernels of real applications.
- Level 2: more complicated real-world application kernels, often found in industry.

After cloning the project's repository, we executed each of the level {0,1,2} applications in its original form and once again with our hook library injected. All runs were successful.

11.3 Assess the performance overhead of the conversion mechanism

We now have sufficient evidence that applications whose memory allocations we transparently convert to Managed (Unified Memory) through our mechanism execute correctly. Now, we must assess the performance overhead, stemming from the fact that GPU memory allocations are now Unified, that these applications experience.

We ran the following benchmarks and collected execution metrics:

- (PyTorch) Ryujaehun's pytorch-gpu-benchmark [40]
- (PyTorch) Official PyTorch benchmarks [33]
- (Tensorflow) Official Tensorflow benchmarks (tf_cnn_benchmarks) [32]
- (Tensorflow) AI-Benchmark [34]

We followed the below methodology for each benchmark:

1. Run the original benchmark a number of times (5 for ai-benchmark, 10 for the rest) to reduce randomness and save the outputs to files.
2. Repeat the step above with `LD_PRELOAD=libunified.so`, converting all memory allocations to Unified
3. Parse the output files in Python, collect the execution times per model/batch_size and calculate the mean of the runs for each model. Do this step for {stock,unified}.
4. Calculate the average latency slowdown `T(unified)/T(stock)` for each model. (for tf_cnn_benchmarks we get images/sec (throughput) as the benchmark output, so we use `Q(stock)/Q(unified)` to get the average throughput slowdown.
5. Calculate the total average slowdown per benchmark.

We summarize our final results in Table 11.1.

Benchmark	Total Average Slowdown	stdev
PyTorch torchbenchmark	0.9994	0.0464
AI-Benchmark	1.0192	0.0261
tf_cnn_benchmarks	1.0173	0.0131
ryujaehun-pytorch-gpu-benchmark	1.0004	0.0306

Table 11.1: *Overall Slowdown from libunified*

We observe a maximum overhead of 1.9% and an average overhead of 0.9% (which is negligible) over the 4 benchmark suites. We consider this overhead to be acceptable, and conclude that we can proceed with the integration of our hooking mechanism to Kubernetes as well as alleviating the thrashing problems that may arise from inter-application interference in sharing scenarios.

We will now go over the procedure and results for each individual benchmark.

11.3.1 Ryujae pytorch-gpu-benchmark

We used the following script to obtain the results for 10 different runs:

```
#!/bin/bash
for i in {0..10}
do
        ./test.sh >"./RUNS_STDOUT_ERR_STOCK/run_$i.txt" 2>&1
done

for j in {0..10}
do
        LD_PRELOAD=libunified.so ./test.sh >"./RUNS_STDOUT_ERR_UNIFIED/run_$j.txt"
    2>&1
done
```

We then trimmed the output of the runs to only keep the benchmarking times. Finally, we wrote a Python script to obtain the slowdown for each model of the benchmark, as well as the average slowdown for the benchmark as a whole. We present the table of results per model in Table 11.2 below. We omit the Table of Slowdowns per model for {training, inference} for brevity.

11.3.2 AI-Benchmark

We introduced `AI-Benchmark` in the previous chapter, as we also used it to test the stability of the conversions. In this case, to make performance measurements, we ran the benchmarking program 5 times and proceeded to parse the output in Python and calculate the slowdowns observed. Table 11.3 shows the average slowdown per model-name, mode, batch-size, and input-size.

Model Name	Average Slowdown
mnasnet0_5	1.0033
mnasnet0_75	1.0028
mnasnet1_0	1.0031
mnasnet1_3	1.0059
resnet18	0.9975
resnet34	0.9973
resnet50	1.0192
resnet101	1.0254
resnet152	1.0301
resnext50_32x4d	0.9717
resnext101_32x8d	0.9551
wide_resnet50_2	1.0086
wide_resnet101_2	1.0089
densenet121	1.0033
densenet169	1.0071
densenet201	1.0033
densenet161	1.0036
squeezenet1_0	0.9928
squeezenet1_1	0.9941
vgg11	0.9979
vgg11_bn	0.9987
vgg13	0.9984
vgg13_bn	0.9990
vgg16	0.9986
vgg16_bn	0.9986
vgg19_bn	0.9996
vgg19	0.9990
shufflenet_v2_x0_5	1.0032
shufflenet_v2_x1_0	1.0107
shufflenet_v2_x1_5	0.9981
shufflenet_v2_x2_0	0.9773

Table 11.2: *Ryujae GPU benchmark Slowdowns*

Model Name	mode	batch size	input size	Slowdown
MobileNet-V2	inference	50	224x224	1.0412
MobileNet-V2	training	50	224x224	1.0127
Inception-V3	inference	20	346x346	1.0582
Inception-V3	training	20	346x346	1.0049
Inception-V4	inference	10	346x346	1.0229
Inception-V4	training	10	346x346	1.0069
Inception-ResNet-V2	inference	10	346x346	1.0188
Inception-ResNet-V2	training	8	346x346	0.9947
ResNet-V2-50	inference	10	346x346	1.0299
ResNet-V2-50	training	10	346x346	1.0009
ResNet-V2-152	inference	10	256x256	1.0285
ResNet-V2-152	training	10	256x256	1.0053
VGG-16	inference	20	224x224	1.0099
VGG-16	training	2	224x224	1.0029
SRCNN	inference	10	512x512	0.9996
SRCNN	inference	1	1536x1536	1.0073
SRCNN	training	10	512x512	1.0109
VGG-19	inference	10	256x256	1.0311
VGG-19	inference	1	1024x1024	1.0074
VGG-19	training	10	224x224	0.9969
ResNet-SRGAN	inference	10	512x512	1.0204
ResNet-SRGAN	inference	1	1536x1536	1.0213
ResNet-SRGAN	training	5	512x512	1.0045
ResNet-DPED	inference	10	256x256	1.0108
ResNet-DPED	inference	1	1024x1024	1.0135
ResNet-DPED	training	15	128x128	1.0038
U-Net	inference	4	512x512	1.0083
U-Net	inference	1	1024x1024	1.0077
U-Net	training	4	256x256	1.0059
Nvidia-SPADE	inference	5	128x128	1.0104
Nvidia-SPADE	training	1	128x128	1.0047
ICNet	inference	5	1024x1536	0.9758
ICNet	training	10	1024x1536	1.0190
PSPNet	inference	5	720x720	1.0085
PSPNet	training	1	512x512	1.0043
DeepLab	inference	2	512x512	1.0155
DeepLab	training	1	384x384	0.9997
Pixel-RNN	inference	50	64x64	1.0678
Pixel-RNN	training	10	64x64	1.1187
LSTM-Sentiment	inference	100	1024x300	1.0643
LSTM-Sentiment	training	10	1024x300	1.0831
GNMT-Translation	inference	1	1x20	1.0471

Table 11.3: *AI-Benchmark Slowdowns*

11.3.3 Official Tensorflow benchmarks (tf-cnn-benchmarks)

The `tf-cnn-benchmarks` repository contains TensorFlow implementations of several popular convolutional models, and is designed to be as fast as possible. We used the following shell script to obtain the results for the 5 different runs for each different model.

```
#!/bin/bash
export TF_CPP_MIN_LOG_LEVEL=3
ALL_MODELS=( "vgg11" "vgg19" "lenet" "googlenet" "overfeat" "alexnet" "trivial" "
    inception3" "inception4" "resnet50" "resnet101" "resnet152" "resnet50_v2" "
    resnet101_v2" "resnet152_v2" )

for model in ${ALL_MODELS[*]}
do
        for bsize in  16 32 64 128
        do
                for i in {0..4}
                do
                        echo "------------------RUN #${i}------------------" >>"
    stock_${model}_${bsize}.txt"
                        python tf_cnn_benchmarks.py --num_gpus=1 --batch_size ${
    bsize} --num_batches 100  --model=${model} >>"stock_${model}_${bsize}.txt"
    2>&1
                done
        done
done

for model in ${ALL_MODELS[*]}
do
        for bsize in 16 32 64 128
        do
                for i in {0..4}
                do
                        echo "------------------RUN #${i}------------------" >>"
    unified_${model}_${bsize}.txt"
                        LD_PRELOAD=libunified.so python tf_cnn_benchmarks.py --
    num_gpus=1 --batch_size ${bsize} --num_batches 100  --model=${model} >>"
    unified_${model}_${bsize}.txt" 2>&1
                done
        done
done
```

We then parsed these results in Python and calculated the total slowdowns. We present the Slowdowns per model-name, mode, and batch-size in Table 11.4

Model Name	mode	batch size	Slowdown
alexnet	training	16	1.0357
alexnet	training	32	1.0430
alexnet	training	64	1.0533
alexnet	training	128	1.0613
googlenet	training	16	0.9955
googlenet	training	32	1.0016
googlenet	training	64	1.0258
googlenet	training	128	1.0263
inception3	training	16	1.0003
inception3	training	32	1.0024
inception3	training	64	1.0038
inception3	training	128	1.0010
inception4	training	16	1.0052
inception4	training	32	1.0021
inception4	training	64	1.0033
lenet	training	16	1.0266
lenet	training	32	1.0208
lenet	training	64	1.0220
lenet	training	128	1.0074
overfeat	training	16	1.0121
overfeat	training	32	1.0180
overfeat	training	64	1.0259
overfeat	training	128	1.0231
resnet101	training	16	1.0129
resnet101	training	32	1.0159
resnet101	training	64	1.0124
resnet101	training	128	1.0369
resnet101_v2	training	16	1.0147
resnet101_v2	training	32	1.0148
resnet101_v2	training	64	1.0184
resnet101_v2	training	128	1.0175
resnet152	training	16	1.0118
resnet152	training	32	1.0100
resnet152	training	64	1.0095
resnet152_v2	training	16	1.0103
resnet152_v2	training	32	1.0104
resnet152_v2	training	64	1.0125
resnet50	training	16	1.0165
resnet50	training	32	1.0146
resnet50	training	64	1.0185
resnet50	training	128	1.0229
resnet50_v2	training	16	1.0180
resnet50_v2	training	32	1.0166
resnet50_v2	training	64	1.0299
resnet50_v2	training	128	1.0313
trivial	training	16	1.0449
trivial	training	32	1.0369
trivial	training	64	1.0197
trivial	training	128	1.0153
vgg11	training	16	1.0129
vgg11	training	32	1.0109
vgg11	training	64	1.0106
vgg11	training	128	1.0103
vgg19	training	16	1.0082
vgg19	training	32	1.0075
vgg19	training	64	1.0056
vgg19	training	128	1.0115

Table 11.4: *tf-cnn-benchmarks Slowdowns*

11.3.4 Official PyTorch benchmarks (torchbenchmark)

Pytorch Benchmarks, abbreviated as `torchbenchmark`, is a collection of open-source benchmarks used to evaluate Pytorch's performance. We did a single run for each of {stock, unified} with 100 iterations internally for each model:

```
#!/bin/bash

pytest ../test_bench.py --ignore_machine_config -k "cuda" \\
--benchmark-min-rounds=100 \\
--benchmark-json="runs_stock_${i}.json" \\
>"runs_stock_${i}_out.txt" 2>&1

LD_PRELOAD=libunified.so pytest ../test_bench.py \\
--ignore_machine_config -k "cuda" \\
--benchmark-min-rounds=100 \\
--benchmark-json="runs_unified_${i}.json" \\
>"runs_unified_${i}_out.txt" 2>&1
```

We then processed the output files in Python and calculated the average slowdowns. We present these results per model-name and mode (inference, training) in Table 11.5

Model Name	mode	Slowdown
BERT_pytorch-cuda-jit	train	0.9396
BERT_pytorch-cuda-jit	eval	0.9863
BERT_pytorch-cuda-eager	train	0.9773
BERT_pytorch-cuda-eager	eval	0.9384
Background_Matting-cuda-jit	train	1.0119
Background_Matting-cuda-eager	train	1.0193
LearningToPaint-cuda-jit	train	1.0631
LearningToPaint-cuda-jit	eval	0.9942
LearningToPaint-cuda-eager	train	1.0379
LearningToPaint-cuda-eager	eval	0.9991
Super_SloMo-cuda-jit	train	0.9961
Super_SloMo-cuda-jit	eval	0.9929
Super_SloMo-cuda-eager	train	0.9953
Super_SloMo-cuda-eager	eval	0.9941
alexnet-cuda-jit	train	0.9970
alexnet-cuda-jit	eval	0.9710
alexnet-cuda-eager	train	0.9971
alexnet-cuda-eager	eval	0.9995
attention_is_all_you_need_pytorch-cuda-jit	train	0.9983
attention_is_all_you_need_pytorch-cuda-jit	eval	0.9940
attention_is_all_you_need_pytorch-cuda-eager	train	0.9959
attention_is_all_you_need_pytorch-cuda-eager	eval	0.9944
demucs-cuda-jit	train	0.9927
demucs-cuda-jit	eval	1.0128
demucs-cuda-eager	train	0.9909
demucs-cuda-eager	eval	1.0156
densenet121-cuda-jit	train	0.9732
densenet121-cuda-jit	eval	0.9911
densenet121-cuda-eager	train	0.9831
densenet121-cuda-eager	eval	1.0578
dlrm-cuda-eager	train	1.0326
dlrm-cuda-eager	eval	1.0072
fastNLP-cuda-jit	train	1.0487
fastNLP-cuda-jit	eval	0.9435
fastNLP-cuda-eager	train	0.9837
fastNLP-cuda-eager	eval	1.0881
maml-cuda-eager	train	1.0692
maml-cuda-eager	eval	1.0723
mnasnet1_0-cuda-jit	train	0.9638
mnasnet1_0-cuda-jit	eval	0.9549
mnasnet1_0-cuda-eager	train	0.9630
mnasnet1_0-cuda-eager	eval	0.9647
mobilenet_v2-cuda-jit	train	1.0156
mobilenet_v2-cuda-jit	eval	0.9657
mobilenet_v2-cuda-eager	train	1.0107
mobilenet_v2-cuda-eager	eval	0.9744
moco-cuda-jit	train	0.9213
moco-cuda-jit	eval	1.0286
moco-cuda-eager	train	0.9255
moco-cuda-eager	eval	1.0357
pytorch_CycleGAN_and_pix2pix-cuda-jit	eval	1.0122
pytorch_CycleGAN_and_pix2pix-cuda-eager	train	0.8814
pytorch_CycleGAN_and_pix2pix-cuda-eager	eval	1.0143
pytorch_mobilenet_v3-cuda-jit	train	1.0940
pytorch_mobilenet_v3-cuda-jit	eval	1.0386
pytorch_mobilenet_v3-cuda-eager	train	1.0986
pytorch_mobilenet_v3-cuda-eager	eval	1.0016
pytorch_stargan-cuda-jit	train	1.0598
pytorch_stargan-cuda-jit	eval	1.1004
pytorch_stargan-cuda-eager	train	1.0660
pytorch_stargan-cuda-eager	eval	1.0687
pytorch_struct-cuda-jit	train	1.0138
pytorch_struct-cuda-jit	eval	1.0284
pytorch_struct-cuda-eager	train	1.0190
pytorch_struct-cuda-eager	eval	0.9390
resnet18-cuda-jit	train	1.0010
resnet18-cuda-jit	eval	1.0542
resnet18-cuda-eager	train	0.9973
resnet18-cuda-eager	eval	0.9989
resnet50-cuda-jit	train	0.8876
resnet50-cuda-jit	eval	0.9882
resnet50-cuda-eager	train	0.8872
resnet50-cuda-eager	eval	1.0110
resnext50_32x4d-cuda-jit	train	0.9309
resnext50_32x4d-cuda-jit	eval	1.0197
resnext50_32x4d-cuda-eager	train	0.9296
resnext50_32x4d-cuda-eager	eval	0.9736
shufflenet_v2_x1_0-cuda-jit	train	0.9402
shufflenet_v2_x1_0-cuda-jit	eval	1.0564
shufflenet_v2_x1_0-cuda-eager	train	0.9648
shufflenet_v2_x1_0-cuda-eager	eval	0.9522
squeezenet1_1-cuda-jit	train	0.9866
squeezenet1_1-cuda-jit	eval	1.0332
squeezenet1_1-cuda-eager	train	0.9830
squeezenet1_1-cuda-eager	eval	0.9428
tacotron2-cuda-eager	train	1.0638
tacotron2-cuda-eager	eval	1.0806
vgg16-cuda-jit	train	1.0009
vgg16-cuda-jit	eval	1.0031
vgg16-cuda-eager	train	1.0009
vgg16-cuda-eager	eval	1.0026
yolov3-cuda-eager	eval	0.9382

Table 11.5: *Pytorch's torchbenchmark Slowdowns*

11.4 Provide a mechanism to prevent thrashing

Having verified that `libunified.so` works correctly and has minimal overhead, we have provided a mechanism that enables GPU Sharing. Many user applications can now run concurrently on the same GPU (concurrently as in their lifetimes overlap; kernel execution cannot overlap). Each of these user applications can allocate and utilize the whole GPU memory. However, when memory is oversubscribed, and due to us enabling it through the use of Unified Memory, page faults can occur. While in our general use-case of interactive development GPU bursts from co-located user applications do not overlap, there can definitely be cases where it does. We need to handle those cases carefully and prevent excessive page faults caused by the constant back and forth transfer of data to and from the GPU due to the page faults.
As such, in this section we will:

- Verify that thrashing scenarios can occur under our sharing mechanism (libunified)
- Provide a mechanism to alleviate thrashing, by serializing the submission (and therefore execution) of GPU kernels and memory copies from the competing processes in sliding windows (time quanta) in a round-robin manner.

11.4.1 Background

Thrashing [41] is defined as a situation in which time spent handling page-faults overwhelms time spent doing useful computations. `Unified Memory` enables page-faults in GPU Memory, using system RAM as the swapping space. When we oversubscribe GPU Memory, the sum of memory allocations of the GPU process(es) exceeds physical GPU capacity, so page-faults and subsequent eviction of pages can happen. We analyze a custom-made thrashing scenario to showcase thrashing when using Unified Memory in Chapter 4.

However, in our specific use-case of interactive ML workloads (Jupyter), while many processes have allocated GPU memory, most of the time only one process will be actively launching kernels (doing computation) on the GPU. As such, this process will only suffer a limited amount of page-faults (and subsequently the other processes will suffer evictions) when fetching its working set to physical GPU memory at the beginning of its computational burst. It will complete its burst of work from start to finish without suffering any extraneous page-faults.

Even when more than one processes submit work to the GPU concurrently, extraneous page-faults (apart from the initial ones we describe above) will only occur if the sum of the data they fetch to GPU memory for that particular computational burst exceeds physical GPU memory capacity. For an NVIDIA Tesla P100 with 16 GB of memory, this means that: `M_bst(A) + M_bst(B) > 16 GiB`, where `M_bst` denotes "size of memory fetched for this computational Burst", A and B are the process names.`M_bst` is usually smaller than the size of memory allocations of the process, as:

- ML Frameworks tend to overshoot actually memory requirements and never shrink their allocations (TF allocates all memory by default)

- not all allocated memory is used for each computational burst

We will now provide some facts specific to GPUs, which affect our assessment of thrashing:

1. We cannot control/stop a running kernel on the GPU. A kernel is a function that executes on the GPU. Kernel launches are asynchronous, and a process can only query for the completion its work submitted to the GPU (`cudaDeviceSynchronize()`).
2. An ML Application launches thousand of GPU kernels, each of which is short (~a few milliseconds)

We will refer to these 2 points later, when making our decisions on how to handle thrashing.

11.4.2 How many kernels do ML applications launch?

Regarding point (2) above (ML applications launch thousands of kernels):

- We measured the amount of kernels run both via `nvprof` and by augmenting libunified.so with a (mutex-guarded) `num_cu_kernels` counter. We increment this by one for every kernel launch issued by the application through our interposition function.

Our evaluation program, `dog-breed-resnet152` (we will provide details on it in the next section) launches a total of 37500 CUDA kernels. Regarding the "tilde (~)" on the number above, it is strange is that, different execution instances of `dog-breed-resnet152` launch a slightly different amount of CUDA Kernels. We investigated this observation further, creating a bash script that runs `nvprof dog-breed-resnet152` 10 times and filters the count of `cudaLaunchKernel` and `cuLaunchKernel` calls. `nvprof` counts the `cudaLaunchKernel` and `cuLaunchKernel` calls separately, even though we know (and will prove below) that the former (Runtime API) calls the latter (Driver API) internally. So, the `cuLaunchKernel` calls that `nvprof` reports are explicit calls to that function, not internal invocations by `cudaLaunchKernel`.

We show our experimental results in Table 11.6. The `num_cu_kernels` value is the one our `libunified.so` reports and is the amount of `cuLaunchKernel` calls. As we said before, `nvprof` counts pure `cuLaunchKernel` calls separately from `cudaLaunchKernel` ones, although `cuda*` calls `cu*` internally. The sum of `cudaLaunchKernel` plus `cuLaunchKernel` calls reported by `nvprof` equals our `num_cu_kernels`.

This also proves that our `libunified.so` hooks every single Kernel launch that the application makes.

	libunified	nvprof counts		
Run id	num_cu_kernels	cudaLaunchKernel	cuLaunchKernel	(nvprof) Total
1	37476	35134	2342	37476
2	37468	35126	2342	37468
3	37350	35008	2342	37350
4	37475	35133	2342	37475
5	37499	35157	2342	37499
6	37366	35024	2342	37366
7	37453	35111	2342	37453
8	37474	35132	2342	37474
9	37461	35119	2342	37461
10	37464	35122	2342	37464

Table 11.6: *libunified vs nvprof kernel launch counts*

11.4.3 dogbreed: Constructing a thrashing scenario in ML

We modified the `dogbreed-v2` Kale example [42], making it use a heavier ML model (Resnet152) and changed the batch/image sizes, trying to create a "heavy" Notebook. We must keep in mind that, when comparing execution times, not all time is spent on GPU computation and CPU code is executed as well. Running two Notebooks in parallel reduces the time spent on CPU computing and in the general case leads to a reduced Total Execution Time, even though GPU kernels from different processes cannot execute in parallel. In the case of `dogbreed-resnet152`, nearly all time is spent on GPU computations, as the main work is done by iteratively training our resnet model (there is no explicit CPU work apart from the control logic of the Python program). We convert the Notebook to a python script to ease testing (but still refer to the script as "Notebook"), since we abolish the need to use a UI. We will also refer to this modified dogbreed with Resnet152 as simply "dogbreed" henceforth. Here is our `dogbreed-thrashing` program:

Listing 11.1: dogbreed-thrashing Python script

```
1  import os
2  import numpy as np
3  import tensorflow as tf
4  import matplotlib.pyplot as plt
5  from tensorflow.keras.preprocessing.image import ImageDataGenerator
6  from glob import glob
7  from PIL import Image
8  from PIL import ImageFile
9  import time
10
11 ImageFile.LOAD_TRUNCATED_IMAGES = True
12
```

```
LR = 6e-4
BATCH_SIZE = 64
NUMBER_OF_NODES = 256
EPOCHS = 5
IMG_SIZE = 224

def get_train_generator():
    data_datagen = ImageDataGenerator(
        rescale=1./255,
        width_shift_range=.2,
        height_shift_range=.2,
        brightness_range=[0.5,1.5],
        horizontal_flip=True
    )
    return data_datagen.flow_from_directory(
        "dogImages/train/",
        target_size=(IMG_SIZE, IMG_SIZE),
        batch_size=BATCH_SIZE,
    )

def get_valid_generator():
    data_datagen = ImageDataGenerator(rescale=1./255)
    return data_datagen.flow_from_directory(
        "dogImages/valid/",
        target_size=(IMG_SIZE, IMG_SIZE),
        batch_size=BATCH_SIZE
    )

def get_test_generator():
    data_datagen = ImageDataGenerator(rescale=1./255)
    return data_datagen.flow_from_directory(
        "dogImages/test/",
        target_size=(IMG_SIZE, IMG_SIZE),
        batch_size=BATCH_SIZE
    )

dog_classifier = tf.keras.applications.ResNet50V2(
    weights="imagenet",
    input_shape=(IMG_SIZE, IMG_SIZE, 3)
)

def is_dog(data):
    probs = dog_classifier.predict(data[0])
```

```
    preds = tf.argmax(probs, axis=1)
    return ((preds >= 151) & (preds <= 268))

train_generator = get_train_generator()
batch = train_generator.next()
predictions = is_dog(batch)

n_dog = np.sum(predictions)
dog_percentage = n_dog/BATCH_SIZE

print('{:.0%} of the files have a detected dog'.format(dog_percentage))

start_time = time.time()
resnet_body = tf.keras.applications.ResNet152V2(
    weights="imagenet",
    include_top=False,
    input_shape=(IMG_SIZE, IMG_SIZE, 3)
)
resnet_body.trainable = True
inputs = tf.keras.layers.Input(shape=(IMG_SIZE, IMG_SIZE, 3))
x = resnet_body(inputs, training=True)
x = tf.keras.layers.Flatten()(x)
outputs = tf.keras.layers.Dense(133, activation="softmax")(x)
resnet_model = tf.keras.Model(inputs, outputs)
resnet_model.compile(
    optimizer=tf.optimizers.Adam(learning_rate=LR),
    loss=tf.losses.categorical_crossentropy,
    metrics=["accuracy"]
)
train_generator = get_train_generator()
valid_generator = get_valid_generator()

resnet_model.fit(train_generator, epochs=EPOCHS,
    validation_data=valid_generator
)

test_generator = get_test_generator()
test_loss_resnet, test_accuracy_resnet = resnet_model.evaluate(test_generator)

print(f"The accuracy in the test set is {test_accuracy_resnet:.3f}.")
print(test_accuracy_resnet)
print("--- %s seconds ---" % (time.time() - start_time))
```

- We first executed one copy of the Notebook and measured the execution time and GPU Memory Usage.
- We then ran two copies of the Notebook in parallel, under different configurations.

We summarize our results below:

Solo Notebook Execution:

We used `[EPOCHS=5, IMG_SIZE=224]` for all runs. The number of Epochs does not affect the memory footprint, it only affects the number of iterations and execution time has a near-linear scaling relationship with the number of Epochs. We obtain 'GPU Memory Usage' from `nvidia-smi`. For these "Solo" runs, only one process is running alone. We show the results in Table 11.7

Batch Size	(Peak) GPU Memory Usage (MiB)	Execution Time (s)
16	11196	752.8
32	11196	752.8
40	11196	750.7
64	15806	750.4

Table 11.7: *Solo dogbreed runs*

Two Notebooks Running in Parallel:

GPU Memory Usage is maxed out at 16259 MiB for batch sizes >= 32 and page-faults unavoidably happen. We define Execution Time as the Total Job Completion time, which is the overall time until all jobs have finished. We show our results in Table 11.8

Batch Size	(Peak) GPU Memory Usage (MiB)	Execution Time (s)
16	13679	1082
32	16259	1123
40	16259	2002
64	16259	11234

Table 11.8: *Two parallel dogbreed executions*

- For the case where Batch Size = 32, even though the total amount of memory used is 2*11196=22392 MiB, there is no significant slowdown when running two Notebooks.
- This is due to the fact that even though the total memory used is > 16 GiB, the working sets for the Notebooks are sufficiently small so as not to cause a large amount of page-faults.
- We note that for Batch Size = 64, Total Execution Time is ~11000 sec, which is 14.6x the solo execution time. In this case, an extreme amount of time was spent handling page-faults and the execution time is far greater than if the GPU computations were run serialized.

- Because, as we noted above, ML Applications launch thousands of small kernels, we don't lose control over the GPU (as happened in Section 10.7 where we launch a single, huge kernel, that iterates over the same memory regions over and over).
- Hence, in a scenario where a large amount of page-faults is taking place, we can alleviate the situation by throttling the kernel launches of one application, while allowing the other to do its computations unhindered.

11.5 Anti-thrashing Mechanism

Our anti-thrashing mechanism is based on the concept of a global GPU lock. Only the process that holds the global lock can do work on the GPU. A scheduler manages the lock; it receives requests from the clients (processes), assigns the lock to a process for a time quantum and retrieves the lock from the process when the TQ elapses.

11.5.1 Entities

First, let's define the entities in our anti-thrashing mechanism:

- **global lock:** Only the process that holds this lock can submit work to the GPU (kernel launches, memory copies).
- **daemon (scheduler):** A separate process that communicates with the clients. The daemon manages the global lock, giving it to applications (`client-threads`) that request it (for a timeslice). It also receives back the lock from the client threads. The daemon operates in a similar fashion to a cooperative scheduler in a Cooperative Multitasking scheme [96].
- **client:** `libunified.so` (linked to each application) spawns this set of threads when CUDA is initialized by the program. It communicates with the `daemon` in order to obtain or release the global lock. It requests the lock from the daemon and releases it when `daemon` orders it. Before releasing the lock it makes sure all currently submitted (to the driver; by past launches) GPU work is complete, and sends an ACK to the scheduler that it no longer has the lock. It also must release the lock when its parent process terminates.
- **bool have_lock:** Per-process variable that indicates whether the process holds the global lock or not. The hooked versions of CUDA API functions that can affect memory state w.r.t. thrashing (`cuLaunchKernel()`, `cuMemCpy()`) check the value of have_lock before they call the real CUDA function that submits work to the GPU. `client` updates the value of `have_lock` after acquiring the global lock from the daemon.
- **bool need_lock:** The hooked functions set this variable to "True" to inform their `client` that the lock is needed. They then block until `have_lock` is True.

11.5.2 Overview

Here is the **numbered list of steps** showing the operation of the anti-thrashing mechanism, from the point of view of a newly created user process:
[We use a Jupyter Notebook as the application that the user runs. The (a), (b) parts of a step can happen in any order. The (i), (ii) parts of a step happen in order.]

1. IPykernel (Jupyter backend) process starts
2. `ld.so` loads `libunified.so`, as it is contained in the "LD_PRELOAD" env variable
3. Process calls `cuInit()` (this is always the first CUDA call it makes)
4. Hooked version of cuInit calls our initializer function
5. Initializer function spawns client threads A & B.
   (a) Client thread A registers itself to the daemon:
   message_type="REGISTER"
   data=[ID, client thread B socket path]
   (b) Client thread B listens on client socket path (uniquely derived from ID) for messages from the `daemon`
6. `daemon` sends a `SCHED_ON` or `SCHED_OFF` message to the client socket (thread B). The scenario describes a situation where the scheduler is enabled, so `daemon` sends a `SCHED_ON` message.
7. User code calls `cuLaunchKernel()` (the same applies for `cuMemcpy()`, for which `libunified.so` has set up a hook (let's call it `cuLaunchKernel\_hook`)
8. `cuLaunchKernel_hook()` checks the value of `have_lock`. It is false, so it sets `need_lock`=True. It then blocks until `have_lock` becomes True.
   (i) client thread A notices that `need_lock` is True
   (ii) client thread A sends a `REQ_LOCK` message to `daemon`
   (iii) client thread B eventually receives a `LOCK_OK` message from `daemon`
   (iv) client thread B sets `have_lock`=True and `need_lock`=False
9. `cuLaunchKernel_hook()` unblocks as `have_lock` is now True. It proceeds with the main body of the function and submits work to the GPU.
10. While `have_lock`=True, all `cu_{LaunchKernel, MemCpy}` functions execute unhindered.
11. client thread B receives a release order (`DROP_LOCK`) from daemon. It sets `need_lock`, `have_lock` to False (so no new work is submitted), calls `cuCtxSynchronize()` (which is blocking) to ensure that all outstanding work submitted to the GPU by the process is complete and finally sends a `LOCK_RELEASED` response to the `daemon`.
12. **Steps 7-11** repeat until the user process exits.
13. At program exit, `client` releases the lock if it holds it, by sending a `LOCK_RELEASED` message to daemon. We set an `atexit()` function for this.

man 3 atexit: https://man7.org/linux/man-pages/man3/atexit.3.html

11.5.3 Design

We now present the logic of all components in flow diagrams:

Hooked Function:

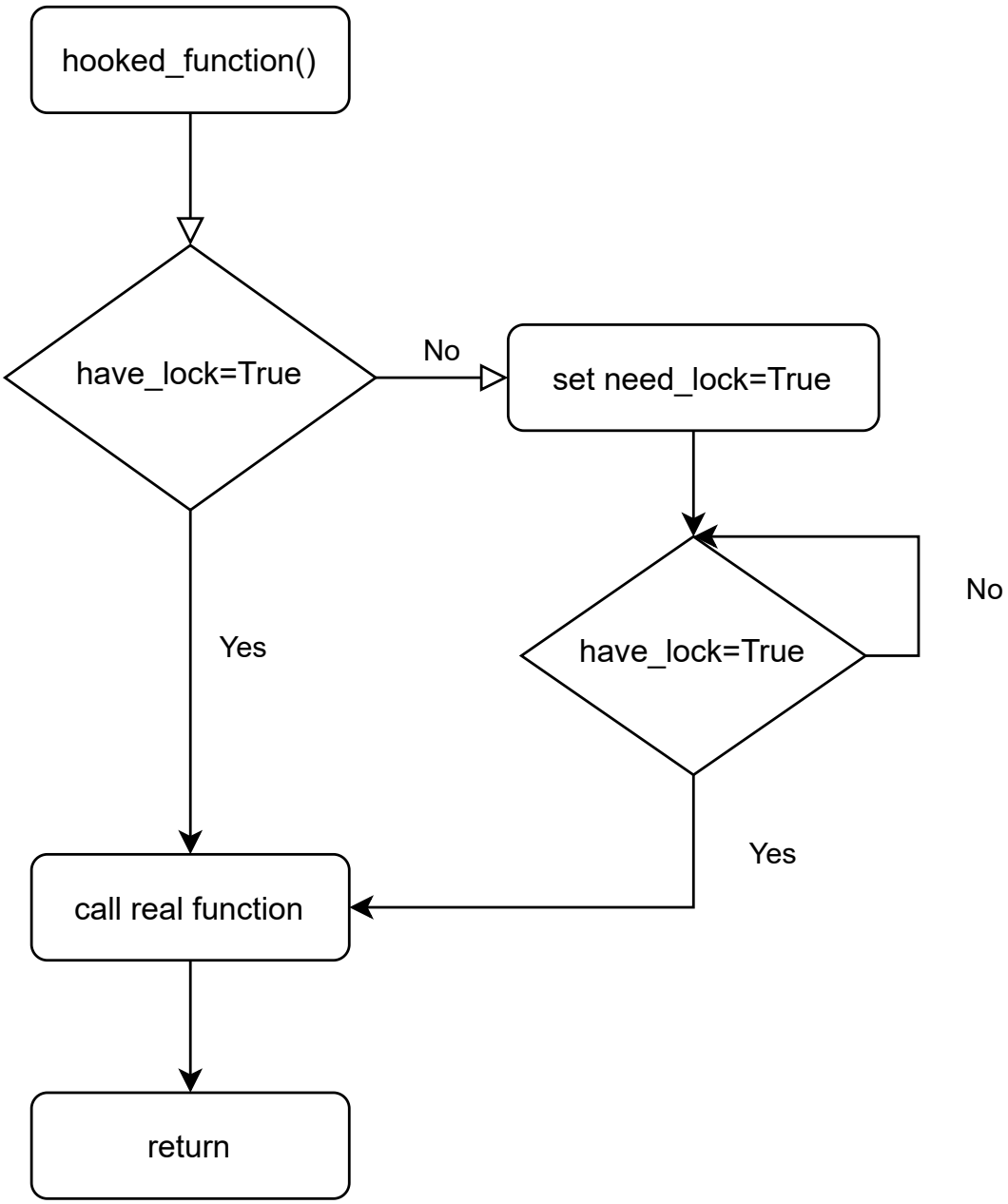

Figure 11.1: *Hooked function flow diagram*

Client:

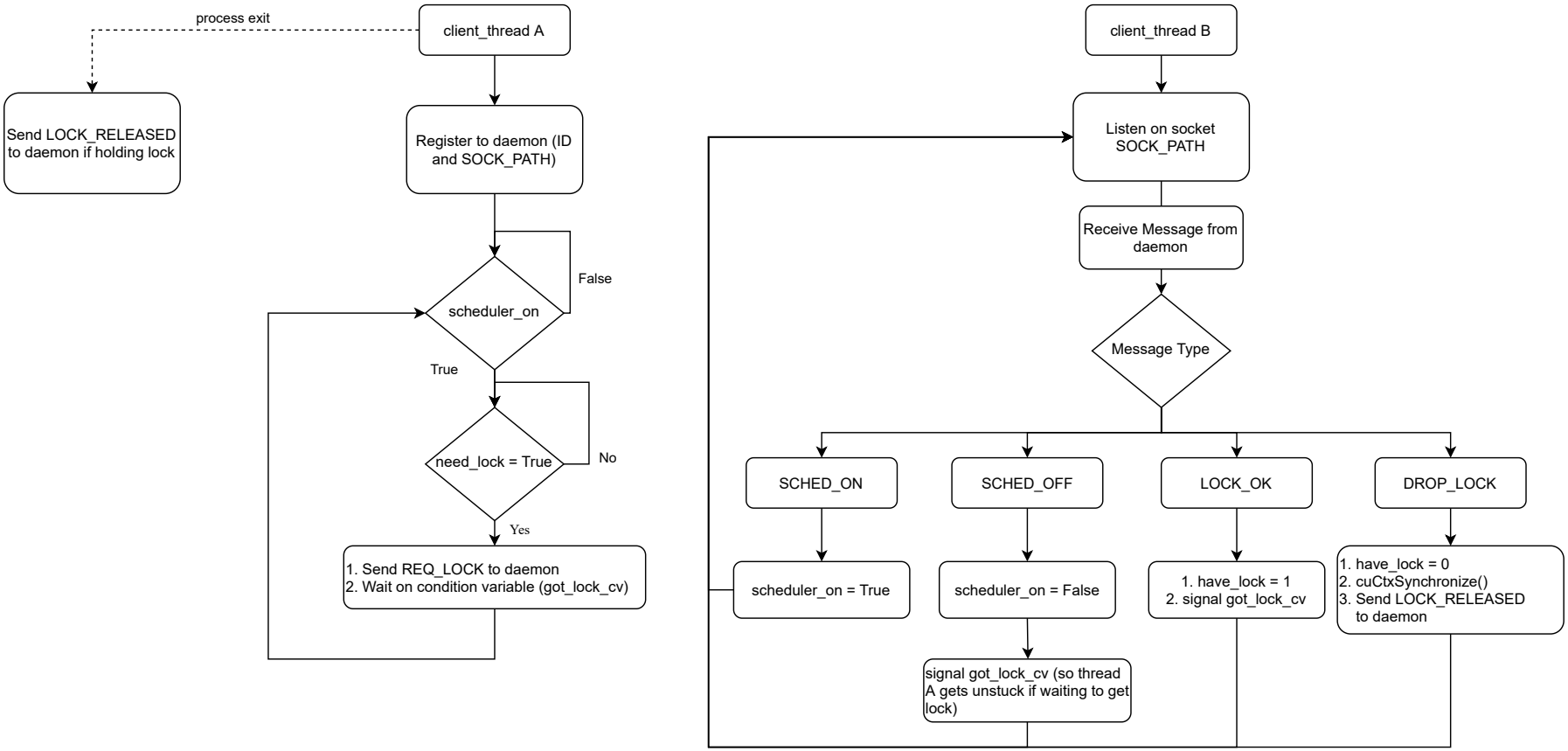

Figure 11.2: *Client flow diagram*

Daemon:

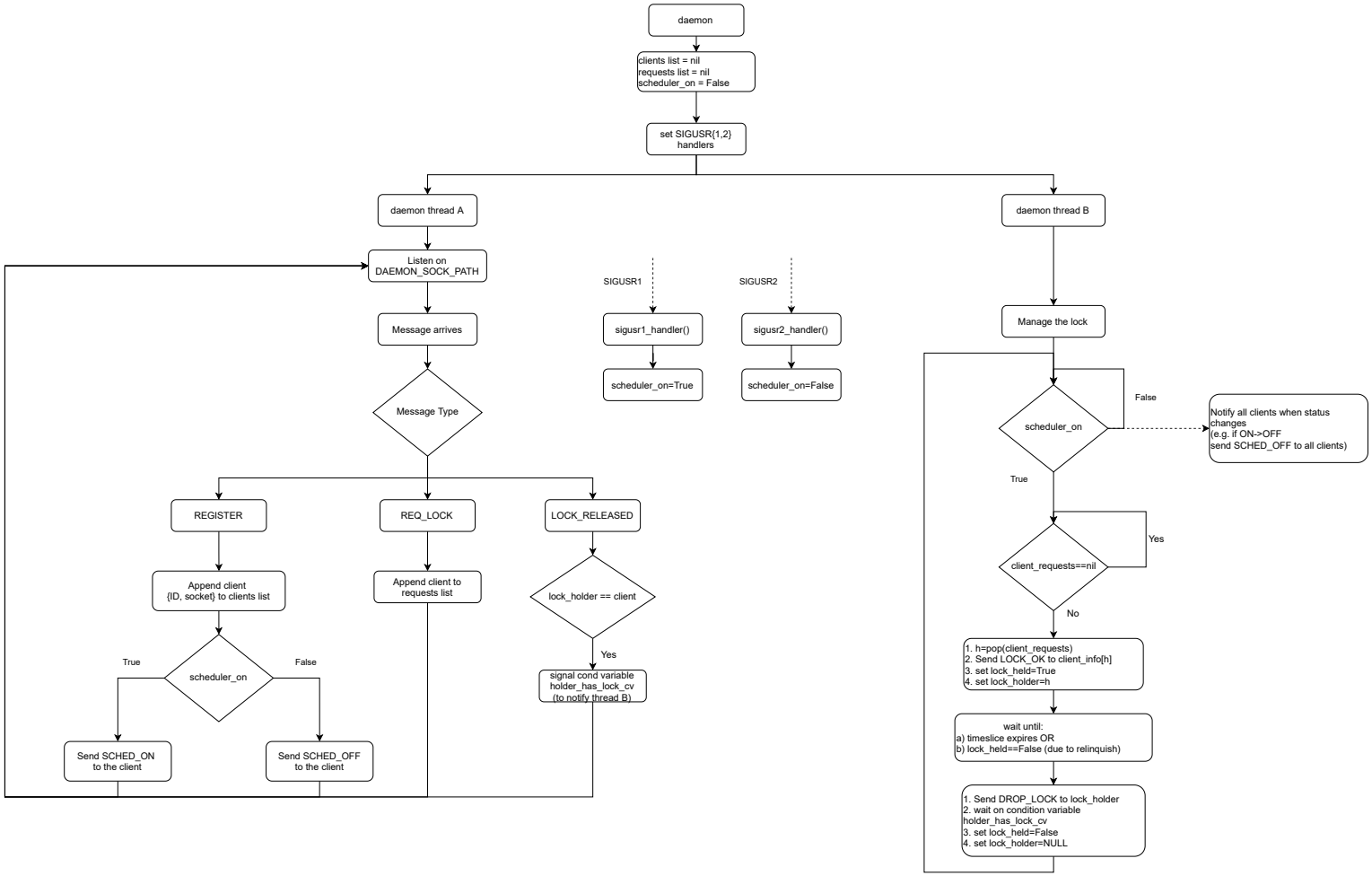

Figure 11.3: *Daemon flow diagram*

11.5.4 Implementation

We have already implemented `libunified.so` as a shared library written in C. We chose to stick to C, as it offers better performance and fine-grained control. For the communication between `client` and `daemon`, we chose to use UNIX Sockets (`SOCK_STREAM` - the equivalent of TCP) as all processes reside in the same physical node. Our anti-thrashing scheduler is deployed as a separate process and all communication between the scheduler and the client(threads) occurs over UNIX Sockets living under the `/tmp/libunified` directory. We can dynamically turn the scheduler ON and OFF by sending a USR1 and USR2 signal respectively to the daemon (scheduler) process. After being turned on, the scheduler immediately notifies all executing processes that it is on and begins processing incoming GPU requests.

11.5.4.1 Checking lock in hooked functions

We tweaked our existing interposition library. The interposed functions now only proceed with their work if the particular client has the global lock. We insert a call to `hook_check_lock()` in the kernel launch and memory copy functions to ensure the lock is held.

Listing 11.2: Lock-checking logic for hooked functions

```
1  void hook_check_lock(void){
2
3          if (!(have_lock)) {
4                  need_lock = 1;
5  // Handle contention; many application threads may call CUDA functions
6                  pthread_mutex_lock(&need_lock_mutex);
7                  pthread_cond_signal(&need_lock_cv);
8                  pthread_mutex_unlock(&need_lock_mutex);
9                  pthread_mutex_lock(&have_lock_mutex);
10 // wait until we acquire the lock
11                 pthread_cond_wait(&received_lock_cv, &have_lock_mutex);
12                 pthread_mutex_unlock(&have_lock_mutex);
13         }
14
15         did_work = 1;
16         return ;
17 }
```

11.5.4.2 Communication

We designed and implemented our custom communication protocol of UNIX `SOCK_STREAM` sockets. We present all possible communication scenarios in Table 11.9, where we use the following abbreviations:

- `c{A,B}`: client threads `{A,B}`
- `d{A,B}`: daemon threads `{A,B}`

Source	Destination	Message Type	data (/notes)
cA	dA	REGISTER	(ID, SOCKPATH)
cA	dA	REQ_LOCK	(ID)
cA	dA	LOCK_RELEASED	(ID)
dB	cB	LOCK_OK	-
dB	cB	DROP_LOCK	-
dA	cB	SCHED_ON	(immediately after REGISTER)
dA	cB	SCHED_OFF	(immediately after REGISTER)
dB	cB	SCHED_ON	(on scheduler status change)
dB	cB	SCHED_OFF	(on scheduler status change)

Table 11.9: *Communication Protocol*

We use a single packed `struct Message` for every exchange of information (every message).

Listing 11.3: Our Message struct

```
#define MSG_TYPE_LEN 16
#define MSG_DATA_LEN 40

struct message {
        uint64_t id;
        char type[MSG_TYPE_LEN];
        char data[MSG_DATA_LEN];
} __attribute__((__packed__));
```

- After writing exactly 64 Bytes successfully, the writer waits on a `read()` call to the socket FD. Because the `Message` struct has a fixed size, reading exactly 64 Bytes (=`sizeof(struct Message)`) from a socket means that we have received a full message, thus we don't need to use delimiters.
- The reader signifies this (the fact that they have received the message) to the writer by closing the socket FD. The `read()` call that the writer made above returns 0, (and the writer knows that they have written a whole message) so they understand that the message was received successfully by the other end.

https://gcc.gnu.org/onlinedocs/gcc-4.0.2/gcc/Type-Attributes.html

11.5.5 The scheduler time quantum

The `TQ` denotes the length of time that the `daemon` gives the lock to a `client` for. We can dynamically change the scheduler `TQ` at any time by sending a `CHANGE_TQ` message to the daemon socket. We have implemented a separate program for that, which takes the new time quantum as an argument and sends the aforementioned message to the daemon.

- A smaller time quantum means more interactivity, as clients with shorter GPU bursts will not have to wait behind others with larger ones and can return the results to the user quicker. Thus, a smaller TQ alleviates head-of-line blocking.
- However, each time the lock swaps hands, the new holder must fetch their data to the GPU (and evict data from another) before doing work. This is due to the fact that we are considering scenarios where GPU memory is oversubscribed, and the working sets of all clients do not fit in GPU memory. As such, a smaller TQ leads to a larger amount of Page Faults which translates to a larger overall context switching overhead and a larger Total Completion Time (TCT).
- A larger TQ leads to a smaller total context switching overhead (as it happens fewer times) and TCT, its drawback being a lowered level of user experience and responsiveness.

11.6 Integration with Kubernetes

Now that we have our mechanism ready, it's time to integrate it with Kubernetes, so users can easily deploy it in their cluster and reap the benefits we offer. We have already analyzed how `nvidia-device-plugin`, which is the de facto way of handling GPUs in Kubernetes, works in Section 8.6.3. A quick revision of the sequence diagrams therein will be really helpful for grasping the rest of this section. We want to stick with this design and alter it in a minimal way to expose our mechanism. Creating our own device plugin will allow us to advertise a shared GPU as an extended resource, enabling the users to request this resource and automatically have our mechanism injected into their Pods. To that end, we designed and implemented `alexo-device-plugin`.

Listing 11.4: alexo-device-plugin Daemonset YAML

```
1  apiVersion: apps/v1
2  kind: DaemonSet
3  metadata:
4    name: alexo-device-plugin
5    namespace: kube-system
6  spec:
7    template:
8      metadata:
9        labels:
10         name: alexo-device-plugin-ds
11     spec:
12       tolerations:
13       - key: alexo.com/shared-gpu
```

```
14          operator: Exists
15          effect: NoSchedule
16        priorityClassName: "system-node-critical"
17        containers:
18        - image: alexo-device-plugin:v0.0.2-ubuntu16.04-bd0fc6a
19          name: alexo-device-plugin-ctr
20          #args: ["--shared-gpus-per-gpu=3"]
21          #   env:
22          #- name: LIBUNIFIED_SRC
23          #  value: "/path/to/libunified/src"
24          #- name: LIBUNIFIED_DST
25          #  value: "/path/to/libunified/dst"
26          securityContext:
27            allowPrivilegeEscalation: false
28            capabilities:
29              drop: ["ALL"]
30          volumeMounts:
31            - name: device-plugin
32              mountPath: /var/lib/kubelet/device-plugins
33            - name: host-libunified
34              mountPath: /usr/lib/arr-gpushare
35          resources:
36            limits:
37              nvidia.com/gpu: 1
38        - image: alexo-device-plugin:v0.0.2-daemon-ubuntu16.04-bd0fc6a
39          name: alexo-device-plugin-gpu-scheduler
40          securityContext:
41            allowPrivilegeEscalation: false
42            capabilities:
43              drop: ["ALL"]
44          volumeMounts:
45            - name: host-scheduler-socks
46              mountPath: /tmp/libunified
47        volumes:
48          - name: device-plugin
49            hostPath:
50              path: /var/lib/kubelet/device-plugins
51          - name: host-libunified
52            hostPath:
53              path: /usr/lib/arr-gpushare
54              type: DirectoryOrCreate
55          - name: host-scheduler-socks
56            hostPath:
57              path: /tmp/libunified
58              type: DirectoryOrCreate
```

Our Daemonset (Listing 11.4) comprises 2 Pods (each Pod running a single container):

- `alexo-device-plugin`: This Pod implements the device plugin logic, communicates with Kubelet and handles `Allocate` requests for `"alexo.com/shared-gpu"`. It also installs `libunified.so` under `/usr/lib/arr-gpushare`.

- `gpu-scheduler`: This Pod comprises our anti-thrashing scheduler. It executes the `daemon/scheduler` container. It communicates with the client applications' components via UNIX sockets in the `/tmp/libunified` directory. We specify the VolumeMount for this directory explicitly in the YAML.

Below we provide a numbered list that illustrates the actions taken by our device plugin.

alexo-device-plugin:

1. Reads the value of `NVIDIA_VISIBLE_DEVICES` from its own environment, uses it later to expose this GPU into the user Pods that request the shared GPU. This is the real GPU, managed by nvidia-device-plugin, which is the "backend" to our shared-gpu.

2. Installs `libunified.so` under `/usr/lib/arr-gpushare` in the node during startup. (It then instructs Kubelet to mount this directory into the user Pods that request our shared-gpu.)

3. Advertises the `alexo.com/shared-gpu` resource to the cluster (by default 1000 devices)

4. Handles `Allocate` requests from Kubelet, instructing it to:

   (a) mount our shared GPU library `libunified.so` at `usr/lib/arr-gpushare/libunified.so` inside the user Pod's FS. Our shared library also implements the client side of the anti-thrashing mechanism, as we explain in the previous section.

   (b) mount the anti-thrashing mechanism socket directory (host's `/tmp/libunified`) at the Pod's `/tmp/libunified`, so the process can communicate with the scheduler

   (c) set the environment variable `LD_PRELOAD` to `/usr/lib/arr-gpushare/libunified.so`

   (d) set the environment variable `NVIDIA_VISIBLE_DEVICES` to the `GPU-UUID` obtained from the `alexo-device-plugin` Pod ("nvidia.com/gpu), in order to expose the GPU (nvidia-container-runtime; see Section 8.4.3) to the container that is requesting `alexo.com/shared-gpu`.

Users can now install our `alexo-device-plugin` with a single `kubectl apply` command: `$ kubectl apply -f alexo-device-plugin.yaml`

A DaemonSet ensures that all (or some) Nodes run a copy of a Pod.

Chapter 12

Evaluation

Our mechanism makes new scenarios possible, in which multiple user applications can share the same GPU without memory constraints. In this chapter we will compare our mechanism against the state of the art. For the novel cases only supported through our mechanism, we can only compare it against itself in addition to the serial execution baseline.

Interactive applications (such as a Jupyter Notebooks) do not have a finite execution time (nor do they have a predetermined structure; the user can change the code cells dynamically). As such, there is no straightforward way to define quantitative measurements on their execution characteristics as a whole, especially completion-time wise. In order to assess our mechanism, we will rely on non-interactive (ordinary) programs. We only examine scenarios in which two processes execute in parallel, also submitting work to the GPU. By doing this, we disregard scenarios in which two interactive jobs execute in parallel and any of them has idle periods which are (sometimes) statistically filled in by the GPU bursts of the other. While what we will examine is the worst-case (most compute-intensive) scenario in the case of interactive workloads, it provides an unambiguous means of quantifying and assessing the performance characteristics of our mechanism. Its performance behavior under real-world interactive workloads can only be better than what we will measure in this chapter.

We were extremely pleased to measure that our mechanism performs exceptionally well even for non-interactive (ordinary) workloads, where there are no idle periods for the process. This means that our mechanism provides a way to maximize GPU utilization even for these workloads, and can be applied to a much wider range of GPU jobs.

12.1 Tools, Methodology and Environment

We conducted our experiments on Google Cloud Platform. We used a singled node, equipped with a 16-core CPU (Intel Xeon 2.3 GHz - Haswell), 104 GB of RAM and an NVIDIA Tesla P100 (GP100GL) GPU with 16 GB of device memory. Our evaluation suite comprises 4 ML applications written in Tensorflow containing a mix of CPU and GPU computational parts. The CPU part of each application trains a ResNet152v2 [43] model for just a few steps (since training on the CPU is an order of magnitude slower than on the GPU). The GPU part also trains a ResNet152v2 model for 5 Epochs.

We first create 2 baseline applications, their major difference being their GPU memory usage. The `"small"` application uses approximately 7 GB of GPU memory throughout its execution, which makes it possible to co-locate two `small` applications on the same GPU in a Kubeshare-style mechanism and compare its performance against our mechanism. The `"big"` application uses approximately 15 GB of GPU memory throughout its execution. This means that any other existing approach except our own cannot run two `big` applications on the same GPU. This case also causes thrashing under our mechanism, so it tests its limits, performance-wise, under maximum stress.

We further create two new applications from the `big` and `small` baselines. We vary the mix between CPU/GPU computational parts (w.r.t. to execution time) to try and examine a variety of workloads, either more GPU-heavy or more balanced. We choose the 90/10 (90 GPU, 10 CPU) and 50/50 ratios. We define the ratios by execution time and when a process is running "alone" in the system.

For the rest of this evaluation we will conflate the terms Working Set Size (WSS) and Memory Usage when talking about GPUs. Working Set Size is a more delicate term, and in ML scenarios can mean the GPU memory that is needed for each training step. Therefore, it can be smaller than the peak Memory Usage of the application, since, as we said ML frameworks overshoot actual demand and never scale down their memory usage. What we expect is that when the sum of WSS of two applications exceeds physical GPU memory, thrashing will occur and that is how we construct the `big` applications. There can be cases where thrashing does not occur, even when the total sum of allocations between the processes is larger than GPU memory. This is because the sum of WSS still fits into GPU memory. We show our final set of 4 applications in Table 12.1

name	GPU Working Set Size	GPU/CPU ratio
small_90	7.2 GB	90/10
small_50	7.2 GB	50/50
big_90	15.3 GB	90/10
big_50	15.3 GB	50/50

Table 12.1: *Our evaluation programs*

We ran all instances of our experiments as containers on a Debian host. In order to measure Kubeshare's execution times, we deployed the containers as (Share)Pods in Kubernetes and specified the GPU memory request in the "Annotations" field, as per Kubeshare's official repository instructions [44]. We cannot execute the `"big"` programs under Kubeshare, as it cannot oversubscribe GPU memory.

12.2 Results

12.2.1 Overview

We show our measurements in Table 12.2. We first executed each of our 4 applications alone (solo) in stock (NVIDIA) mode, with our `libunified` with anti-thrashing turned off and finally with our anti-thrashing mechanism on. This models the way the GPU bursts of an interactive application would behave when running alone and gives an estimate of the overhead of our mechanism in and of itself, eliminating Page Faults or interference from co-located applications.

Afterwards, we established our baseline for the execution time of 2 copies of each program running in Parallel by multiplying the solo (stock) time by 2 and obtaining the serial execution time. This denotes the time it would take to run two instances of a program one after another. Then, we tested `Kubeshare` for the `small` applications (as we mentioned before, it cannot run two copies of the `big` applications, as the total WSS exceeds physical GPU memory). We also tested our `libunified` mechanism without anti-thrashing support and finally enabled the anti-thrashing scheduler and measured times for various TQ (time quantum) values. Keep in mind that the `client` part of our mechanism automatically releases the lock back to the scheduler after 5 seconds of no GPU work having been submitted.

Method	**small_50 (s)**	**small_90 (s)**	**big_50 (s)**	**big_90 (s)**
solo (stock)	1318	692	1383	719
solo (libunified no anti-thrashing)	1332	711	1385	724
solo (libunified w/ anti-thrashing)(60)	1339	712	1390	734
serial (2*solo-stock)	**2636**	**1384**	**2766**	**1438**
2 instances in Parallel for all below				
Kubeshare	1724	1078	---	---
libunified (no anti-thrashing)	1772	1128	*11757 (thrashing)*	*11434 (thrashing)*
anti-thrashing(1000)	2053	1361	2043	1380
anti-thrashing(500)	2053	1363	2070	1400
anti-thrashing(400)	2030	1352	2081	1405
anti-thrashing(300)	2010	1354	2085	1418
anti-thrashing(200)	2007	1360	2092	1423
anti-thrashing(100)	2010	1351	2100	1435
anti-thrashing(60)	2020	1348	2122	1468
anti-thrashing(50)	1995	1351	2145	1473
anti-thrashing(40)	1993	1343	2156	1485
anti-thrashing(30)	2017	1366	2170	1521
anti-thrashing(20)	2007	1360	2204	1583
anti-thrashing(15)	1983	1341	2305	1668
anti-thrashing(10)	2009	1360	2465	1821
anti-thrashing(5)	1982	1334	3496	2788

Table 12.2: *Execution Time Measurements for "small" and "big"*

We also visualize the results of the table above in Figures 12.1, 12.2, 12.3. We note the anti-thrashing scheduler's time quantum (TQ) in parentheses in the figure labels. In the next sections we will comment on our measurements and state our observations.

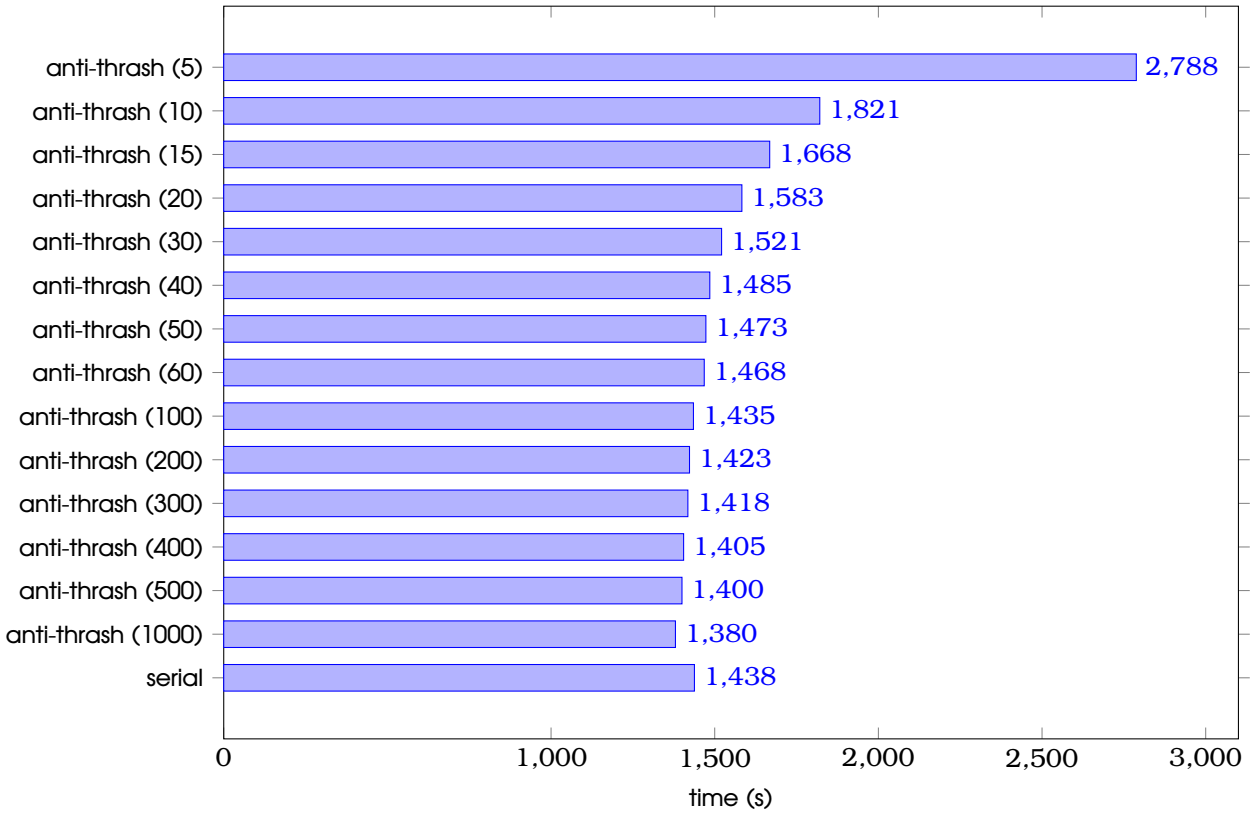

Figure 12.1: *Execution times for big_90*

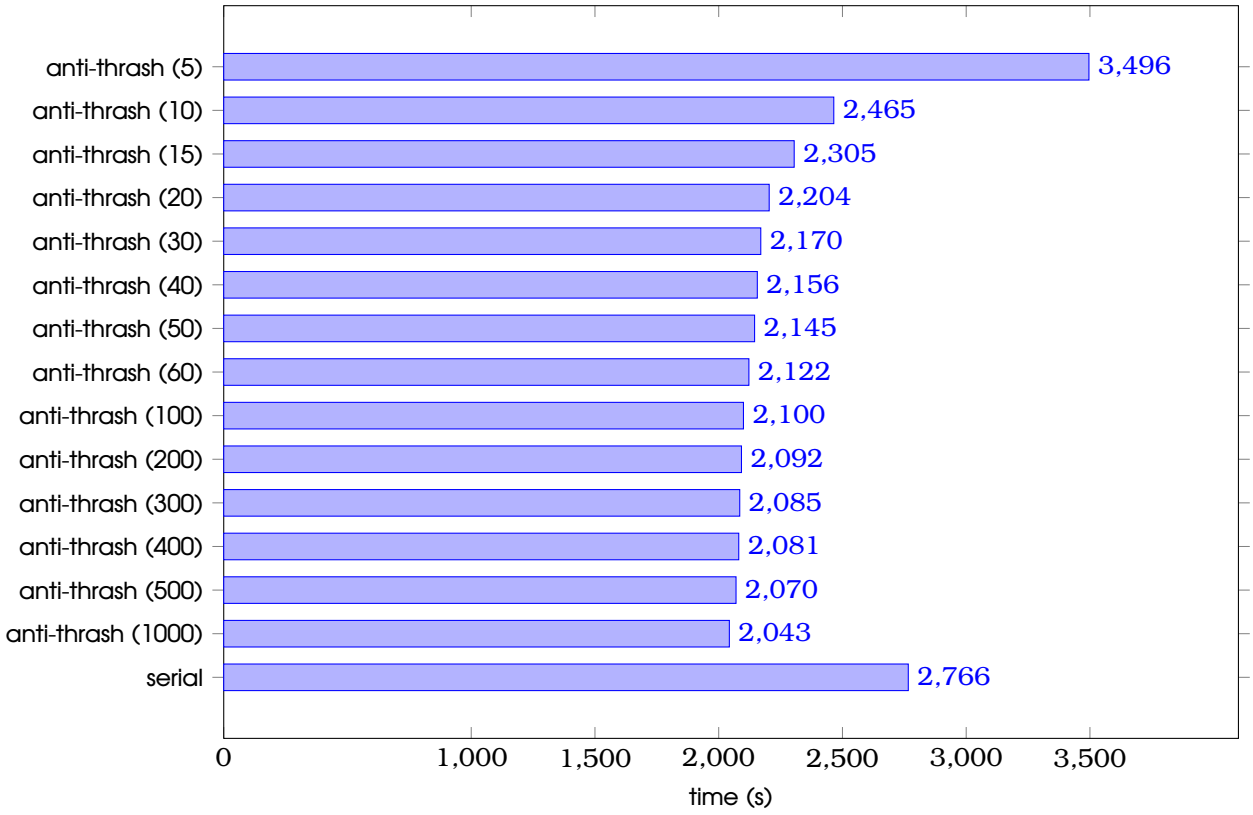

Figure 12.2: *Execution times for big_50*

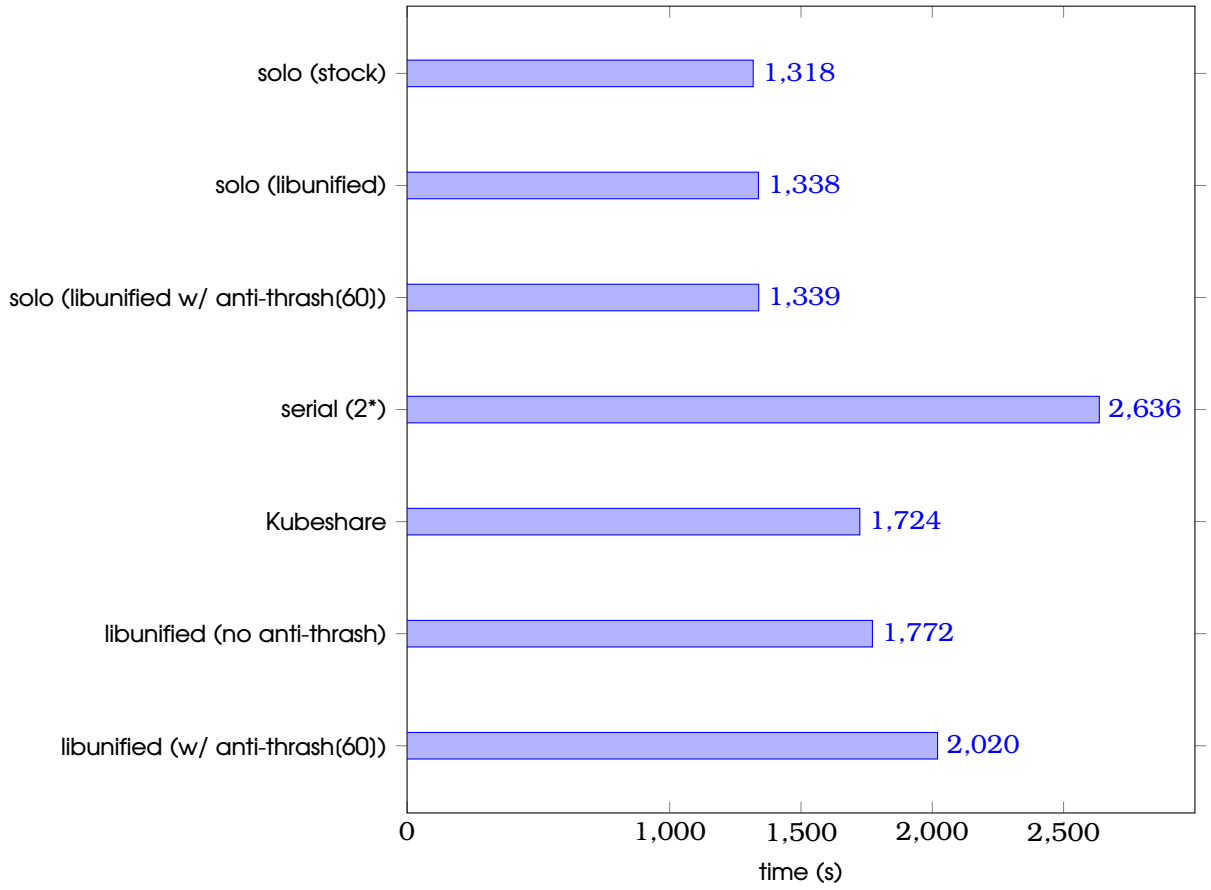

Figure 12.3: *Execution times for small_50*

12.2.2 Small WSS

For the `small` WSS cases (small_90 and small_50), no Page Faults happen throughout the parallel execution of the two identical programs (they would be fatal in the non-unified cases after all). For the 50/50 case, we are able to achieve significant speedup, as the CPU parts run completely in parallel. We must note here that when two different processes (contexts) submit work to the GPU, the GPU schedules this work in an undisclosed manner. It cannot execute kernels from two different contexts concurrently, however it handles the scheduling (context-switching) in a more efficient manner than our anti-thrashing scheduler. Hence, the most performant method in this case is Kubeshare. However, when using Kubeshare we must declare the memory limits of the process beforehand, which is most always not the case. The execution time of our `libunified` (without anti-thrashing enabled) comes at a close second, its overhead stemming from the fact that it uses Unified Memory in and of itself(and not Page Faults). We estimated this overhead in Section 11.3 of Chapter 5 to be around ~1 %. The least performant method is `libunified (anti-thrash)`, in which the anti-thrashing scheduler is enabled and each process uses the GPU for a time slice. This is expected, as in these `small` applications there is no thrashing and therefore no need to serialize GPU work from the processes. The default black-box NVIDIA driver (which is used in the Kubeshare and libunified cases) handles the GPU work in a more efficient manner than the conservative approach of our

anti-thrashing scheduler. Keep in mind, however, that this category of `small` applications is rare and also that Kubeshare supports co-location only when the sum of Memory allocations is smaller than GPU memory (in this case it could not support a third copy running in parallel).

12.2.3 Big WSS

Kubeshare cannot support two copies of `big_{90,50}` running in parallel, as the WSS equals roughly 30 GB. When executing two copies of a `big_{90,50}` program in parallel, thrashing occurs under our simple `libunified`, hence the extremely large execution times in Table 12.2. However, with our anti-thrashing mechanism enabled, we are able to achieve times well below the serial execution time even for the 50/50 case, and comparable to the serial execution times for the 50/50 case. Having a larger Time Quantum minimizes the Total Completion Time (TCT), as less swaps (Page Faults) occur when the scheduler lock changes hands. When the TQ becomes too small (< 10 sec), timing measurements begin to rise sharply, as execution time is once again dominated by Page Faults since each application is not given enough time to perform its meaningful computations. Our mechanism performs excellently and manages to execute two applications in parallel, with a WSS > 30 GB, in sub-serial times. No existing solution permits running applications whose sum of WSS exceeds GPU physical memory, let alone achieve that with a sub-serial time.

12.3 Recommending default policies (TQ, scheduler on/off)

Regarding the scheduler time quantum, as we said in Section 11.5.5:

- A smaller time quantum means more interactivity, as clients with shorter GPU bursts will not have to wait behind others with larger ones and can return the results to the user quicker. Thus, a smaller TQ alleviates head-of-line blocking.
- However, each time the lock swaps hands, the new holder must fetch their data to the GPU (and evict data from another) before doing work. This is due to the fact that we are considering scenarios where GPU memory is oversubscribed, and the working sets of all clients do not fit in GPU memory. As such, a smaller TQ leads to a larger amount of Page Faults which translates to a larger overall context switching overhead and a larger Total Completion Time (TCT).
- A larger TQ leads to a smaller total context switching overhead (as it happens fewer times) and TCT, its drawback being a lowered level of user experience and responsiveness.

As such, and given our measurements, we consider a TQ of 20 sec to be a sensible default value, as it strikes the best balance between performance and user experience/interactivity. For non-interactive workloads, especially when fairness is not an issue (for example 2 applications owned by the same user), a huge TQ will lead to the shortest overall TCT (remember than clients voluntarily release the lock after 5 seconds of not having done GPU work).

Additionally, we recommend it a prudent default to always have the anti-thrashing scheduler enabled. Even though for the cases where the total WSS of the co-located applications is smaller than GPU memory we incur a ~10% overhead, we have no oracle knowledge and cannot (at the moment) predict the future, especially for interactive jobs. Given that we don't have a mechanism through which to decide when the anti-thrashing mechanism should be switched on (thrashing "detection" is part of our future work), it's safer to always have it on. Remember that this "overhead", is only incurred when the GPU parts of the applications overlap (as was the case in all our measurements). When only one application submits work to the GPU, the overhead of the anti-thrashing mechanism is minimal/non-existent (See Figure12.3 - row `solo (w/ anti-thrash)`).

Chapter 13

Concluding Remarks

Our journey is reaching its end. We will restate our contributions, illustrating what the new state with regard to GPU Sharing in Kubernetes is, this time taking our approach into account as well. Finally, we will close this thesis by mentioning Future Work that can be done to enrich our mechanism.

13.1 A new state of the art

We have designed, implemented and evaluated `alexo-device-plugin`. Let's review what it offers once more:

- each application can **use the whole GPU memory**.
- Many **applications can execute on the same GPU concurrently without hard memory limits**. The only limiting factor is host memory (RAM).
- It is **transparent to user applications**; no modification must be made to user code or frameworks.
- It has **minimal overhead** when a single application is executing alone on the GPU.
- It **averts thrashing effectively**, maximizing GPU utilization and minimizing Total completion time even when Working Set sizes far exceed GPU memory capacity.
- It can be deployed in a K8s cluster with a **single** `kubectl apply` **command**.

As we saw in the previous chapter, even though our initial goal was only to enable GPU Sharing for interactive ML jobs to increase GPU utilization; and is the use-case our implementation absolutely thrives at, **our mechanism can be used just as well with non-interactive jobs** (throughput-oriented). We provide a qualitative comparison with the state of the art in Table 13.1

Mechanism	Can co-locate	co-location constraint	guaranteed memory fraction	K8s Integration
nvidia-device-plugin	No	-	yes (whole mem)	Seamless
Aliyun Scheduler Extender	Yes	Physical GPU memory	No	Average
Kubeshare	Yes	Physical GPU memory	Yes	Hard
alexo-device-plugin	Yes	RAM + GPU Memory	No*	Seamless

Table 13.1: *Comparison with state of the art*

[*]: We cannot define a "guaranteed" fraction for Unified Memory in the same way that we can for nomal CUDA allocations. We do not impose a hard limit on the size of memory allocations from a process, so in that sense we do not offer "guaranteed" memory fractions.

13.2 Future Work

We can finally wrap this thesis up with the future research directions of our mechanism. We plan on pursuing these actively over the next months/years.

- Perform a real-world deployment/analysis on actual user workloads in a production environment.
- Extend `alexo-device-plugin` to support multiple GPUs per node.
- Create heuristics to automatically enable/disable the anti-thrashing mechanism instead of doing it manually. One heuristic can be the volume of PCIe transfers over a sliding time window.
- Support job migration from one GPU to another (intra-node first, inter-node at a later stage). Our use of Unified Memory eases this task, as multiple GPUs in a node have a common CUDA address space. Still, it is far from trivial.

Bibliography

[1] *Machine Learning*. https://xkcd.com/1838/. Accessed: 31-05-2021.

[2] Hangchen Yu, Arthur Michener Peters, Amogh Akshintala και Christopher J. Rossbach. *AvA: Accelerated Virtualization of Accelerators*. *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, σελίδες 807–825. ACM, 2020.

[3] *A game changer for containers; cgroups v2*. https://acloudguru.com/blog/engineering/a-game-changer-for-containers-cgroups, 2021. Accessed: 31-05-2021.

[4] *Containers vs VMs*. https://www.redhat.com/en/topics/containers/containers-vs-vms. Accessed: 31-05-2021.

[5] J. Montrym και H. Moreton. *The GeForce 6800*. *IEEE Micro*, 25(2):41–51, 2005.

[6] *NVIDIA Corp. Inside volta: The world's most advanced data center GPU.* https://devblogs.nvidia.com/parallelforall/inside-volta/. Accessed: 31-05-2021.

[7] Jeffrey M. Perkel. *Why Jupyter is data scientists' computational notebook of choice*. *Nature*, 563(7729):145–146, 2018.

[8] *nvidia-device-plugin: A Kubernetes device-plugin*. https://github.com/NVIDIA/k8s-device-plugin/. Accessed: 31-05-2021.

[9] *GPUs can not be shared, but GPUs must be shared*. https://discourse.jupyter.org/t/gpus-can-not-be-shared-but-gpus-must-be-shared/1348/6. Accessed: 31-05-2021.

[10] *Is sharing GPU to multiple containers feasible?* https://github.com/kubernetes/kubernetes/issues/52757. Accessed: 31-05-2021.

[11] *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*, 2015.

[12] Ting An Yeh, Hung Hsin Chen και Jerry Chou. *KubeShare: A Framework to Manage GPUs as First-Class and Shared Resources in Container Cloud*. *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '20, 2020.

[13] *GPU Sharing Scheduler Extender in Kubernetes*. https://github.com/AliyunContainerService/gpushare-scheduler-extender. Accessed: 31-05-2021.

[14] *Alibaba Cloud: cGPU*. https://www.alibabacloud.com/help/doc-detail/171786.htm?spm=a2c63.p38356.879954.7.4c6e3cdat0S0f5#task-2536612. Accessed: 31-05-2021.

[15] *Amazon Elastic Inference*. https://docs.aws.amazon.com/sagemaker/latest/dg/ei.html. Accessed: 31-05-2021.

[16] *GPU Virtualization with vSphere Bitfusion*. https://www.vmware.com/solutions/business-critical-apps/hardwareaccelerators-virtualization.html. Accessed: 31-05-2021.

[17] *Run:AI*. https://www.run.ai/. Accessed: 31-05-2021.

[18] *Backend.AI*. https://backend.ai/. Accessed: 31-05-2021.

[19] Wikipedia contributors. *General-purpose computing on graphics processing units — Wikipedia, The Free Encyclopedia*, 2021. [Online; accessed 31-May-2021].

[20] *NVIDIA Tesla V100 GPU Architecture*. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf. Accessed: 31-05-2021.

[21] *How the Fermi Thread Block Scheduler works*. https://www.cs.rochester.edu/~sree/fermi-tbs/fermi-tbs.html. Accessed: 31-05-2021.

[22] *etcd: A distributed, reliable key-value store for the most critical data of a distributed system*. https://etcd.io/. Accessed: 31-05-2021.

[23] *Kubernetes Device Manager Proposal*. https://github.com/kubernetes/community/blob/master/contributors/design-proposals/resource-management/device-plugin.md. Accessed: 31-05-2021.

[24] *gRPC: A high performance, open source universal RPC framework*. https://grpc.io/. Accessed: 31-05-2021.

[25] *Introducing Container Runtime Interface (CRI) in Kubernetes*. https://kubernetes.io/blog/2016/12/container-runtime-interface-cri-in-kubernetes/. Accessed: 31-05-2021.

[26] *NVIDIA Profiler (nvprof)*. https://docs.nvidia.com/cuda/profiler-users-guide/index.html. Accessed: 31-05-2021.

[27] *Unified Memory for CUDA Beginners*. https://developer.nvidia.com/blog/unified-memory-cuda-beginners/. Accessed: 31-05-2021.

[28] Robert Crovella. *Stackoverflow: pre-Pascal vs post-Pascal Unified Memory behavior*. https://stackoverflow.com/a/40011988. Accessed: 31-05-2021.

[29] *cudaMemPrefetchAsync Documentation*. https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html. Accessed: 31-05-2021.

[30] *THE LD_PRELOAD trick*. http://www.goldsborough.me/c/low-level/kernel/2016/08/29/16-48-53-the_-ld_preload-_trick/. Accessed: 31-05-2021.

[31] *CUDA samples*. https://github.com/NVIDIA/cuda-samples. Accessed: 31-05-2021.

[32] *Official Tensorflow Benchmarks*. https://github.com/tensorflow/benchmarks. Accessed: 31-05-2021.

[33] *Official PyTorch Benchmarks*. https://github.com/pytorch/benchmark. Accessed: 31-05-2021.

[34] Andrey Ignatov. *AI-benchmark suite*. https://ai-benchmark.com/tests.html. Accessed: 31-05-2021.

[35] Bodun Hu και Christopher J. Rossbach. *Altis: Modernizing GPGPU Benchmarks*. *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2020.

[36] *CUDA Interprocess Communication*. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#interprocess-communication. Accessed: 31-05-2021.

[37] Andrey Ignatov, Radu Timofte, Andrei Kulik, Seungsoo Yang, Ke Wang, Felix Baum, Max Wu, Lirong Xu και Luc Van Gool. *AI Benchmark: All About Deep Learning on Smartphones in 2019*, 2019.

[38] *Altis GPGPU benchmarking suite source repository*. https://github.com/utcs-scea/altis. Accessed: 31-05-2021.

[39] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju και Jeffrey S. Vetter. *The Scalable Heterogeneous Computing (SHOC) Benchmark Suite*. *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, 2010.

[40] *ryujaehun: Pytorch GPU benchmark*. https://github.com/ryujaehun/pytorch-gpu-benchmark. Accessed: 31-05-2021.

[41] Wikipedia contributors. *Thrashing (computer science) — Wikipedia, The Free Encyclopedia*, 2021. [Online; accessed 13-June-2021].

[42] *Kale examples: Dog Breed Classification v2*. https://github.com/kubeflow-kale/kale/blob/d90310dbebc765c68915df0cf832a7a5d1ec1551/examples/dog-breed-classification/dog-breed-v2.ipynb. Accessed: 31-05-2021.

[43] *Keras: ResNet152v2*. https://www.tensorflow.org/api_docs/python/tf/keras/applications/resnet_v2/ResNet152V2. Accessed: 31-05-2021.

[44] *Kubeshare: Share GPU between Pods in Kubernetes*. https://github.com/NTHU-LSALAB/KubeShare. Accessed: 31-05-2021.

[45] Wikipedia contributors. *Flynn's taxonomy — Wikipedia, The Free Encyclopedia*, 2021. [Online; accessed 31-May-2021].

[46] *CUDA Kernel Launch Syntax*. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#execution-configuration. Accessed: 31-05-2021.

[47] *CUDA dim3 reference*. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#dim3. Accessed: 31-05-2021.

[48] *CUDA Runtime API*. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#cuda-c-runtime. Accessed: 31-05-2021.

[49] *CUDA Driver API*. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#driver-api. Accessed: 31-05-2021.

[50] *An Easy Introduction to CUDA C and C++*. https://developer.nvidia.com/blog/easy-introduction-cuda-c-and-c/. Accessed: 31-05-2021.

[51] *GPU Architecture Overview*. https://insujang.github.io/2017-04-27/gpu-architecture-overview/. Accessed: 31-05-2021.

[52] S. Kato. *Implementing Open-Source CUDA Runtime*. *Proceedings of the 54th Programming Symposium*, 2013.

[53] Ivan Tanasic, Isaac Gelado, Javier Cabezas, Alex Ramirez, Nacho Navarro και Mateo Valero. *Enabling preemptive multiprogramming on GPUs*. *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, σελίδες 193–204, 2014.

[54] *CUDA Streams*. https://leimao.github.io/blog/CUDA-Stream/. Accessed: 31-05-2021.

[55] *CUDA Streams and Concurrency*. https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf. Accessed: 31-05-2021.

[56] *CUDA Binary Utilities*. https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html. Accessed: 31-05-2021.

[57] *CUDA Parallel Thread Execution (PTX)*. https://docs.nvidia.com/cuda/parallel-thread-execution/index.html. Accessed: 31-05-2021.

[58] *Understanding Fat Binaries and JIT caching*. https://developer.nvidia.com/blog/cuda-pro-tip-understand-fat-binaries-jit-caching/. Accessed: 31-05-2021.

[59] *Development environment for multi platform CUDA software*. https://vjordan.info/thesis/nvidia_gpu_archi/devel_env.xhtml. Accessed: 31-05-2021.

[60] *NVCC Documentation*. https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html. Accessed: 31-05-2021.

[61] Ivan Tanasic. *Towards Multiprogrammed GPUs*. Διδακτορική Διατριβή, 2017. pages 11-12.

[62] *GPU Sharing among different CUDA contexts*. https://forums.developer.nvidia.com/t/gpu-sharing-among-different-application-with-different-cuda-context/53057/4. Accessed: 31-05-2021.

[63] *NVIDIA Multi-process service*. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf. Accessed: 31-05-2021.

[64] *NVIDIA GP100 Pascal Whitepaper*. https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf. Accessed: 31-05-2021.

[65] Dirk Merkel. *Docker: Lightweight Linux Containers for Consistent Development and Deployment*. *Linux J.*, 2014.

[66] Tejun Heo (Linux Kernel Developer). *Control Group v2*. https://www.kernel.org/doc/Documentation/cgroup-v2.txt. Accessed: 31-05-2021.

[67] *Open Container Initiative*. https://opencontainers.org/. Accessed: 31-05-2021.

[68] *Alibaba Cloud: Open Container Initiative (OCI) Specifications*. https://www.alibabacloud.com/blog/open-container-initiative-oci-specifications_594397. Accessed: 31-05-2021.

[69] *Snapcraft: Hello World CUDA Analysis*. https://forum.snapcraft.io/t/hello-world-cuda-analysis/11250. Accessed: 31-05-2021.

[70] *NVIDIA Driver: Listing of Installed Components*. https://download.nvidia.com/XFree86/Linux-x86_64/450.51/README/installedcomponents.html. Accessed: 31-05-2021.

[71] *StackOverflow: Using GPU from a docker container?* https://stackoverflow.com/a/40191878. Accessed: 31-05-2021.

[72] *NVIDIA-docker version 1.0 Github Wiki*. https://github.com/NVIDIA/nvidia-docker/wiki/NVIDIA-driver-(version-1.0). Accessed: 31-05-2021.

[73] *nvidia-docker-1 Github Respository*. https://github.com/NVIDIA/nvidia-docker/tree/v0.0.0-poc. Accessed: 31-05-2021.

[74] *nvidia-docker-2 Github Respository*. https://github.com/NVIDIA/nvidia-docker/. Accessed: 31-05-2021.

[75] *NVIDIA Container Toolkit*. https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/overview.html. Accessed: 31-05-2021.

[76] *Podman: A tool for managing OCI containers and pods*. https://podman.io/. Accessed: 31-05-2021.

[77] *Github: NVIDIA Container Runtime*. https://github.com/nvidia/nvidia-container-runtime. Accessed: 31-05-2021.

[78] *Github: containerd*. https://github.com/containerd/containerd. Accessed: 31-05-2021.

[79] *Opencontainers' runc*. https://github.com/opencontainers/runc. Accessed: 31-05-2021.

[80] *OCI Runtime Spec Prestart Hooks*. https://github.com/opencontainers/runtime-spec/blob/master/config.md#prestart. Accessed: 31-05-2021.

[81] *Github: libnvidia-container*. https://github.com/NVIDIA/libnvidia-container. Accessed: 31-05-2021.

[82] *NVIDIA Container Runtime Environmental Variables*. https://github.com/NVIDIA/nvidia-container-runtime/blob/master/README.md#environment-variables-oci-spec. Accessed: 31-05-2021.

[83] *Read list of GPU devices from volume mounts instead of NVIDIA_VISIBLE_DEVICES*. https://docs.google.com/document/d/1uXVF-NWZQXgP1MLb87_kMkQvidpnkNWicdpO2l9g-fw/. Accessed: 31-05-2021.

[84] *Docker: specifying resource constraints - GPU*. https://github.com/docker/docker.github.io/blob/master/config/containers/resource_constraints.md#gpu. Accessed: 31-05-2021.

[85] *Moby*. https://github.com/moby/moby/blob/master/daemon/nvidia_linux.go. Accessed: 31-05-2021.

[86] *Kubernetes Docs: Schedule GPUs*. https://kubernetes.io/docs/tasks/manage-gpus/scheduling-gpus/. Accessed: 31-05-2021.

[87] *tkestack: GPU Manager*. https://github.com/tkestack/gpu-manager. Accessed: 31-05-2021.

[88] José Duato, Antonio J. Peña, Federico Silla, Rafael Mayo και Enrique S. Quintana-Ortí. *rCUDA: Reducing the number of GPU-based accelerators in high performance clusters*. *2010 International Conference on High Performance Computing Simulation*, σελίδες 224–231, 2010.

[89] Akira Nukada, Hiroyuki Takizawa και Satoshi Matsuoka. *NVCR: A Transparent Checkpoint-Restart Library for NVIDIA CUDA*. *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, σελίδες 104–113, 2011.

[90] Taichiro Suzuki, Akira Nukada και Satoshi Matsuoka. *Efficient Execution of Multiple CUDA Applications Using Transparent Suspend, Resume and Migration*. *Euro-Par 2015: Parallel Processing*, σελίδες 687–699. Springer Berlin Heidelberg, 2015.

[91] *StackOverflow: Dynamic Linking vs Dynamic Loading*. https://stackoverflow.com/questions/10052464/difference-between-dynamic-loading-and-dynamic-linking. Accessed: 31-05-2021.

[92] *dlsym(3) man page*. https://linux.die.net/man/3/dlsym. Accessed: 31-05-2021.

[93] *Dangers of using dlsym() with RTLD_NEXT*. http://optumsoft.com/dangers-of-using-dlsym-with-rtld_next/. Accessed: 31-05-2021.

[94] *CUDA Examples, cuHook*. https://github.com/phrb/intro-cuda/blob/d38323b81cd799dc09179e2ef27aa8f81b6dac40/src/cuda-samples/7_CUDALibraries/cuHook/libcuhook.cpp#L59. Accessed: 31-05-2021.

[95] *Gemini: an efficient GPU resource sharing system with fine-grained control for Linux platforms*. `https://github.com/NTHU-LSALAB/Gemini/`. Accessed: 31-05-2021.

[96] Wikipedia contributors. *Cooperative multitasking — Wikipedia, The Free Encyclopedia*, 2021. [Online; accessed 14-June-2021].