Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Σχεδίαση και Υλοποίηση Συστήματος Διαμοιρασμού Επιταχυντών Γραφικών σε Εικονικά Περιβάλλοντα

Διπλωματική εργασία

Κωνσταντίνου Παπαζαφειρόπουλου

Επιβλέπων καθηγητής: Νεκτάριος Κοζύρης

Αθήνα,
Μάιος 2015

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Σχεδίαση και Υλοποίηση Συστήματος Διαμοιρασμού Επιταχυντών Γραφικών σε Εικονικά Περιβάλλοντα

Διπλωματική εργασία

Κωνσταντίνου Παπαζαφειρόπουλου

Επιβλέπων καθηγητής: Νεκτάριος Κοζύρης

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 1η Ιουλίου 2015.

___________	___________	___________
Νεκτάριος Κοζύρης Καθηγητής Ε.Μ.Π.	Γεώργιος Γκούμας Λέκτορας Ε.Μ.Π.	Νικόλαος Παπασπύρου Αναπληρωτής Καθηγητής Ε.Μ.Π.

Αθήνα,
Μάιος 2015

Κωνσταντίνος Παπαζαφειρόπουλος
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Copyright © Κωνσταντίνος Παπαζαφειρόπουλος, 2015.
Με επιφύλαξη παντός δικαιώματος. All rights reserved

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

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσοβίου Πολυτεχνείου.

στους φίλους, τους γονείς και την αδερφή μου
που με στηρίζουν όλα αυτά τα χρόνια

σε όλους αυτούς που αγωνίζονται
για μια αξιοπρεπή ζωή

Περίληψη

Το cloud computing, είτε αυτό αφορά σε κατανεμημένα - γεωγραφικά - περιβάλλοντα (grid computing) είτε αφορά υπολογιστικά κέντρα με συστοιχίες (clusters) υπολογιστών, βρίσκεται αναπόφευκτα στο επίκεντρο του ενδιαφέροντος στις μέρες μας. Για οποιαδήποτε εφαρμογή απαιτεί αυξημένη υπολογιστική ισχύ η αποτελεσματικότητα των παραπάνω τεχνολογιών αποκτά ιδιαίτερη σημασία (HPC). Η αποτελεσματική παραλληλοποίηση επαναλαμβανόμενων διαδικασιών που προσφέρει η χρήση GPUs - αντί για CPUs - (GPGPU) σε υπολογιστικά clusters ή/και grid για την επιτάχυνση υπολογιστικά απαιτητικών εργασιών την καθιστούν μια όλο και ευρύτερα υιοθετούμενη λύση. Μάλιστα, ο συνδυασμός των επιδόσεων του GPGPU με τα πλεονεκτήματα που προσφέρουν τα εικονικά περιβάλλοντα (κλιμακωσιμότητα, ελαστικότητα, ασφάλεια, live migration) μπορεί να παρέχει όχι μόνο υψηλές επιδόσεις αλλά και ευελιξία. Ταυτόχρονα όμως, εισάγει και προβλήματα που έχουν εμποδίσει μέχρι στιγμής την καθολική χρήση τέτοιου τύπου λύσεων, παρόλο που λαμβάνουν έντονης ερευνητικής προσοχής. Τα σημαντικότερα είναι η επιβάρυνση στις επιδόσεις που προσθέτει το virtualization αλλά και η αποδοτικότητα ενός τέτοιου συνδυασμού: δεν αρκεί δηλαδή μια τέτοια λύση να προσφέρει μόνο υψηλές επιδόσεις, αλλά πρέπει να είναι και οικονομική (πχ. δυνατότητα χρήσης GPU σε κάποια μόνο μηχανήματα μιας συστοιχίας υπολογιστών). Σκοπός αυτής της εργασίας είναι η μελέτης της δομής και τρόπων για την αποδοτική χρήση τέτοιων συνδυαστικών συστημάτων που αξιοποιούν gpgpu και virtualization. Συγκεκριμένα, εστιάζουμε στη δυνατότητα υπολογιστών χωρίς GPU να εκτελούν αποτελεσματικά εφαρμογές CUDA σε απομακρυσμένους (εικονικούς και μη) υπολογιστές με GPU, δηλαδή στην ταυτόχρονη (αποδοτική) χρήση μιας host gpu από πολλαπλούς (απομακρυσμένους) clients. Σε αυτό το πλαίσιο, παρουσιάζουμε μια υλοποίηση (GPUsockets) “διάφανη”/χωρίς απαιτούμενες μετατροπές από τη μεριά του χρήστη, βασισμένη στο μοντέλο διαχωρισμένου οδηγού, που αξιοποιεί το CUDA driver API, ώστε να είναι εύχρηστη και επεκτάσιμη. Εξετάζουμε την αποδοτικότητά της σε περιβάλλον Xen και, στο βαθμό που τα αποτελέσματα είναι ικανοποιητικά, προτείνουμε βελτιώσεις (πχ. αξιοποιώντας εναλλακτικές λύσεις ενδοεπικοινωνίας των VMs όπως το V4V/V4VSockets) και πιθανές μελλοντικές επεκτάσεις.

Λέξεις-Κλειδιά: GPGPU, Εικονικοποίηση, Αποκακρυσμένη Εκτέλεση σε GPU, GPUsockets

Abstract

Cloud computing, whether it concerns - geographically - distributed environments (grid computing) or computing centers with computer arrays (clusters), lies inevitably in the center of interest nowadays. For any application that requires increased computing power, efficiency of the above technologies is of particular importance (HPC). Efficient parallelization of repetitive tasks offered by the use of GPUs - instead of CPUs - (GPGPU) in computer clusters/grid computing for accelerating compute intensive applications, make it an increasingly adopted solution. The combination of GPU performance with the advantages offered by virtual environments (scalability, elasticity, security, live migration) offers both high performance and flexibility. At the same time it introduces problems that have thus far prevented wide adoption of such solutions despite them gaining great attention in research. Most importantly the performance overhead added by virtualization and the efficiency provided by a combination of that kind: one such solution should not only offer high performance, but also be economical (ie. offering the ability to use a GPU only in certain physical nodes of a computer cluster). The purpose of this thesis is to study the structure and ways to efficiently use such combinatorial systems that leverage GPGPU and virtualization. In particular, we focus on the ability of GPU-less computers to effectively run CUDA applications on remote (virtual or physical) GPU-equiped computers, namely simultaneous (efficient) use of a host GPU by multiple (remote) clients. In this context, we present an implementation (GPUsockets) that is "transparent"/without the need of any user-side modification, based on the split-driver model, that uses CUDA driver API, so that it offers ease of use and extensibility. We evaluate its' efficiency in a Xen environment and, to the extent that the results are satisfying, we propose further improvements (ie. by using alternative solution for inter-VM communication like V4V/V4VSockets) and possible future extensions.

Keywords: GPGPU, Virtualization, GPU Remote Execution, GPUsockets

Ευχαριστίες

Αρχικά θα ήθελα να ευχαριστήσω τον καθηγητή κ.Νεκτάριο Κοζύρη για την ευκαιρία που μου έδωσε να εκπονήσω τη διπλωματική μου εργασία στο συγκεκριμένο εργαστήριο.

Ιδιαίτερες ευχαριστίες οφείλω στον Μεταδιδακτορικό Ερευνητή Αναστάσιο Νάνο για την υπομονή και την υποστήριξή του σε όλη τη διάρκεια της εκπόνησης αυτής της διπλωματικής. Η καθοδήγηση με καίριες παρατηρήσεις και η ενθάρρυνσή του ήταν όχι απλώς σημαντικές αλλά καθοριστικές για την ολοκλήρωσή της.

Για την ψυχολογική υποστήριξη επιβάλλεται να ευχαριστήσω τον Γιάννη, τη Χλόη, τον Κώστα και όλους τους φίλους που με ανέχτηκαν και με βοήθησαν μέχρι την τελευταία στιγμή.

Για τα καλύτερα φοιτητικά χρόνια που θα μπορούσα να περάσω, πρέπει επίσης να ευχαριστήσω τους συμφοιτητές και φίλους που μαζί παλέψαμε και παλεύουμε για να είναι το πανεπιστήμιο, στο οποίο εκπονήθηκε αυτή η διπλωματική, ανοιχτό και δημόσιο. Πρέπει να ευχαριστήσω τους Ανεξάρτητους Αριστερούς Φοιτητές Ηλεκτρολόγους, όχι μόνο γιατί κοντά τους γνώρισα τη συντροφικότητα και τη συλλογικότητα, αλλά κυρίως γιατί με έμαθαν να αγωνίζομαι για ότι θέλω να γίνει καλύτερο. Γιατί με έμαθαν να είμαι λίγο περισσότερο άνθρωπος στη δύσκολη καθημερινότητα του σήμερα. Οι μεγαλύτεροι αγώνες είναι μπροστά μας.

Τέλος, ευχαριστώ τους γονείς μου και την αδερφή μου για την αγάπη, την κατανόησή τους και τη στήριξή τους στις επιλογές μου όλα αυτά τα χρόνια.

Κωνσταντίνος Παπαζαφειρόπουλος

Περιεχόμενα

Περιεχόμενα ix

Κατάλογος σχημάτων xi

1 Εισαγωγή 1

2 Θεωρητικό Υπόβαθρο 3
- 2.1 GPGPU/GPU acceleration 3
- 2.2 Virtualization 8
- 2.3 Sockets - TCP/IP 12
- 2.4 Data serialization 15

3 Σχεδίαση/Υλοποίηση 19
- 3.1 Γενική Περιγραφή 19
- 3.2 Διαδικασία Εξυπηρέτησης CUDA κλήσης 23

4 Πειραματική Αποτίμηση 27
- 4.1 MatSum και Jacobi Stencil 27
- 4.2 Εναλλακτικοί μηχανισμοί ενδοεπικοινωνίας εικονικών μηχανών: V4VSockets/V4V 31

5 Σχετικές Υλοποιήσεις 35
- 5.1 vCUDA 35
- 5.2 rCUDA 36
- 5.3 gVirtuS 36
- 5.4 GviM 36
- 5.5 DS-CUDA 37

6 Σύνοψη 39

Βιβλιογραφία 43

Κατάλογος σχημάτων

2.1 Ενδο-επικοινωνία εικονικών μηχανών στο Xen 11

3.1 Τρόπος λειτουργίας του GPUsockets 20
3.2 Λίστα CUDA συσκευών του backend 22
3.3 Κόμβος της λίστας clients του backend 23
3.4 Δομή μηνύματος του GPUsockets 24
3.5 Δομή πληροφοριών και λίστες frontend 25

4.1 Μέσος χρόνος εκτέλεσης MatSum και Jacobi stencil από 2,3 και 4 clients σειριακά και παράλληλα 29
4.2 Ενδο-επικοινωνία εικονικών μηχανών με V4Vsockets 31
4.3 Κατανομή χρόνου εξυπηρέτησης αιτημάτων MatSum και Jacobi stencil με το συμβατικό μηχανισμό Xen και με V4V/V4VSockets . 32

1 Εισαγωγή

Στη σύγχρονη εποχή, των big data (μεγάλου όγκου/σύνθετων και απαιτητικών σε υπολογιστικούς πόρους δεδομένων) και των απομακρυσμένων υπηρεσιών, το cloud computing, είτε αυτό αφορά σε κατανεμημένα - γεωγραφικά - περιβάλλοντα (grid computing) είτε αφορά υπολογιστικά κέντρα με συστοιχίες (clusters) υπολογιστών, βρίσκεται αναπόφευκτα στο επίκεντρο του ενδιαφέροντος. Ειδικά για επιστημονικές εφαρμογές που βασίζονται σε μαθηματικά μοντέλα, αλλά και γενικά για οποιαδήποτε εφαρμογή απαιτεί αυξημένη υπολογιστική ισχύ, η αποτελεσματικότητα των παραπάνω τεχνολογιών αποκτά ιδιαίτερη σημασία - αναφερόμαστε στο high performance computing (HPC - "υπολογισμούς υψηλών επιδόσεων").

Συγκεκριμένα για τις εφαρμογές που αφορούν (συνήθως απαιτητικά) υπολογιστικά φορτία, μεγάλο ενδιαφέρον έχει παρουσιάσει τα τελευταία χρόνια η επιτάχυνση μέσω GPUs. Η αποτελεσματική παραλληλοποίηση επαναλαμβανόμενων διαδικασιών που προσφέρουν οι GPUs προσφέρει σημαντική αύξηση των επιδόσεων και καθιστά τη χρήση των τελευταίων για "υπολογισμούς γενικού σκοπού" (GPGPU) σημαντική εξέλιξη σε αυτό τον τομέα. Έτσι έχουμε λύσεις όπως η CUDA της NVIDIA και το ανοιχτό OpenCL του Khronos Group, σημειώνοντας παράλληλα ότι μέχρι τη στιγμή της συγγραφής αυτής της διπλωματικής δεν υπάρχει κάποια κοινή χαμηλού επιπέδου διεπαφή (low level interface) για GPGPU.

Ταυτόχρονα, το virtualization, παρόλο που δεν είναι καινούργιο σαν έννοια, έχει δεχτεί τα τελευταία χρόνια ιδιαίτερη προσοχή λόγω της αποδοτικότητας και της ευελιξίας που προσφέρει η χρήση του, ιδιαίτερα σε συστοιχίες υπολογιστών. Οι τεχνολογίες virtualization παρέχουν μια σειρά από πλεονεκτήματα. Καταρχάς επιτρέπουν την αποτελεσματική αξιοποίηση των πόρων σε συστήματα υπολογιστών (μοιράζοντας το διαθέσιμο υλικό σε πολλαπλά εικονικά μηχανήματα), προσφέρουν δηλαδή "κλιμακωσιμότητα" (scalability) και "ελαστικότητα" (elasticity). Ταυτόχρονα, παρέχουν απομόνωση και αυξημένη ασφάλεια στα εικονικά συστήματα, ενώ διευκολύνουν και τη μεταφορά του περιβάλλοντος χρήστη από το ένα μηχάνημα στο άλλο σε πραγματικό χρόνο (relocation και live migration). Παράλληλα όμως έχουν και ένα σημαντικό μειονέκτημα, που είναι και ο κύριος λόγος που δεν χρησιμοποιούνται ευρέως

σε HPC περιβάλλοντα σήμερα: Λόγω της έμμεσης επικοινωνίας με το υλικό και των παραπάνω επιπέδων εξυπηρέτησης και πολύπλεξης των αιτημάτων που εισάγει το virtualization, επιβαρύνονται σημαντικά οι επιδόσεις εφαρμογών που χρησιμοποιούν συχνά συσκευές εισόδου/εξόδου (Ε/Ε - I/O), φυσική μνήμη, CPUs, GPUs κλπ.

Οι επιδόσεις που προσφέρει, λοιπόν, η χρήση GPUs - αντί για CPUs - σε υπολογιστικά clusters ή/και grid για την επιτάχυνση υπολογιστικά απαιτητικών εργασιών, που αφορούν τομείς από τα οικονομικά και την υγεία μέχρι τα μαθηματικά και τη φυσική, την καθιστούν όχι μόνο σημαντική εξέλιξη αλλά και μια όλο και ευρύτερα υιοθετούμενη λύση. Η χρήση GPGPU ειδικά σε εικονικά περιβάλλοντα μπορεί να προσφέρει,συνεπώς,όχι μόνο υψηλές επιδόσεις αλλά και ευελιξία. Παρόλα αυτά, υπάρχουν κάποιοι σημαντικοί λόγοι για τους οποίους αν και τέτοιου τύπου λύσεις λαμβάνουν έντονης ερευνητικής προσοχής δεν χρησιμοποιούνται καθολικά. Αφενός η υλοποίηση μια λύσης με GPUs σε εικονικά περιβάλλοντα χρειάζεται να αντιμετωπίζει την επιβάρυνση στις επιδόσεις που σημειώσαμε παραπάνω. Αφετέρου, οποιαδήποτε από τις λύσεις που αναφέραμε, χρειάζεται να είναι αποδοτική. Δεν αρκεί δηλαδή να προσφέρει μόνο υψηλές επιδόσεις, αλλά ταυτόχρονα να είναι και οικονομική: Η χρήση μιας GPU ανά μηχάνημα σε μια συστοιχία υπολογιστών μπορεί να είναι όχι μόνο οικονομικά δαπανηρή αν σπάνια χρησιμοποιούνται όλες, αλλά και ελάχιστα ευέλικτη.

Σε αυτή την εργασία θέλουμε να ασχοληθούμε με τη δομή και την αποδοτική αξιοποίηση συστημάτων όπως τα παραπάνω. Μέχρι στιγμής, μια σειρά από υλοποιήσεις (τις οποίες παρουσιάζουμε επιγραμματικά στο κεφ.4) έχουν προσπαθήσει να δώσουν λύσεις - μερικά ή συνολικά - στα προβλήματα που περιγράψαμε. Βασισμένοι στο frontend/backend μοντέλο - ένα κομμάτι-εφαρμογή στον/στους server/s και ένα κομμάτι-εφαρμογή στον/στους client/s - που χρησιμοποιούν πολλές από αυτές, παρουσιάζουμε ένα ανοιχτού κώδικα (open-source) framework που επιτρέπει τη χρήση απομακρυσμένων GPGPU επιταχυντών μέσω του (χαμηλότερου επιπέδου) CUDA Driver API. Στη δική μας υλοποίηση η λειτουργία των δύο εφαρμογών είναι “διάφανη” για το χρήστη (δίνει την ψευδαίσθηση της απευθείας χρήσης του εν λόγω API) και η μεταξύ τους επικοινωνία γίνεται με ένα προσαρμοσμένο πρωτόκολλο που χρησιμοποιεί data serialization και είναι βασισμένο σε TCP sockets. Χρησιμοποιώντας αυτή την προσέγγιση προσπαθούμε να μελετήσουμε τον διαμοιρασμό GPGPU επιταχυντών σε εικονικά περιβάλλοντα δίνοντας έμφαση στην κλιμακωσιμότητα (scalability), δηλαδή εν προκειμένω στην ταυτόχρονη (αποδοτική) χρήση μιας host gpu από πολλαπλούς (απομακρυσμένους) clients.

2

Θεωρητικό Υπόβαθρο

2.1 GPGPU/GPU acceleration

Η έννοια GPGPU (General Purpose computing on GPU – Γενικού σκοπού υπολογισμοί σε GPU) χρησιμοποιείται για να περιγράψει τη χρήση μιας GPU (μονάδα επεξεργασίας γραφικών), όχι για υπολογισμούς που αφορούν γραφικά όπως συνηθίζεται, αλλά για την εκτέλεση υπολογιστικών φορτίων – δηλαδή εφαρμογών ή κομματιών εφαρμογών που συνήθως εκτελούνται στη CPU (κεντρική μονάδα επεξεργασίας).

Η αρχιτεκτονική των GPUs, με τους πολλούς πυρήνες και το μεγάλο bandwidth στην ιδιωτική τους μνήμη (private memory), που είναι σχεδιασμένη για την αποτελεσματική παράλληλη εκτέλεση πολλών εργασιών, είναι ιδανική για εφαρμογές που πραγματοποιούν πολλούς υπολογισμούς (όπως μαθηματικά μοντέλα). Αυτός είναι και λόγος που το GPU acceleration (επιτάχυνση με GPU) - δηλαδή η παράλληλη χρήση μιας GPU, μαζί με τη CPU, για την αποτελεσματική εκτέλεση υπολογιστικά δαπανηρών εφαρμογών - γίνεται όλο και πιο δημοφιλές τα τελευταία χρόνια σε διάφορα επιστημονικά πεδία [1, 2].

Τα πιο δημοφιλή και ευρέως χρησιμοποιούμενα frameworks για GPGPU είναι το ανοιχτό OpenCL (του Khronos Group) και η κλειστή (proprietary) CUDA (της NVIDIA).

Το OpenCL (Open Computing Language) είναι ένα ανοιχτό και ατελές (royalty-free) πρότυπο (standard) για γενικού σκοπού παράλληλο προγραμματισμό CPUs, GPUs και άλλων επεξεργαστών. Περιλαμβάνει ένα API (Διεπαφή Προγραμματισμού Εφαρμογών) για το συντονισμό παράλληλων υπολογισμών μεταξύ ετερογενών επεξεργαστών και μία γλώσσα προγραμματισμού που υποστηρίζει διάφορα λειτουργικά συστήματα - βασισμένη στο ISO C99 με επεκτάσεις για παραλληλισμό - με ένα σαφώς προσδιορισμένο υπολογιστικό περιβάλλον για τον προγραμματισμό σε αυτές τις συσκευές [3]. Το πρότυπο αυτό κυκλοφόρησε στην πρώτη του έκδοση πολύ κοντά χρονικά (με διαφορά ενός χρόνου) με την αντίστοιχη πρώτη υλοποίηση της CUDA.

Παρόλο όμως που το OpenCL είναι ανοιχτό, ταχέως εξελισσόμενο και αφορά διάφορες κατηγορίες επεξεργαστών, η (κλειστή) CUDA - τουλάχιστον μέχρι σχετικά πρόσφατα - θεωρούταν πιο ώριμη λύση. Κυρίως – πέρα από τις, περισσότερο ίσως υποκειμενικές, διαφορές απόδοσης και την πληρότητα των βιβλιοθηκών - γιατί, ενώ το ανοιχτό πρότυπο εξελίσσεται, οι αντίστοιχες υλοποιήσεις σε οδηγούς συσκευών (drivers) καθυστερούν ή/και υπολείπονται αρκετά. Αλλά και γιατί η CUDA έχει πιο προηγμένα εργαλεία αποσφαλμάτωσης (debugging) και μελέτης απόδοσης (profiling). Έτσι, η τελευταία έχει κυριαρχήσει μέχρι στιγμής σαν λύση στον τομέα του GPU acceleration, για αυτό και χρησιμοποιείται ως βάση αυτής της εργασίας.

Η CUDA (Compute Unified Device Architecture) είναι μια πλατφόρμα παράλληλου υπολογισμού και προγραμματιστικό μοντέλο της NVIDIA [4] για την αξιοποίηση - αποκλειστικά - των GPUs της τελευταίας για επεξεργασία γενικού σκοπού (GPGPU). Αποτελείται από την αντίστοιχη αρχιτεκτονική στο επίπεδο του υλικού αλλά και από το λογισμικό που εκθέτει την αρχιτεκτονική αυτή, δηλαδή από γλωσσικές επεκτάσεις (language extensions) και APIs για δημοφιλείς γλώσσες όπως η C/C++, από οδηγίες μεταγλωτιστή (compiler directives – βλ. OpenACC) κ.α. [5]

Όσον αφορά τη C/C++ το CUDA toolkit προσφέρει 2 APIs:

- Το Runtime API: Ένα υψηλότερου επιπέδου API. Αυτό είναι και το ευρέως χρησιμοποιούμενο, αφού επιτρέπει με σημαντικά λιγότερο κώδικα την αποτελεσματική διαχείριση και εκτέλεση των απαιτούμενων ενεργειών.
- Το Driver API: Ένα χαμηλότερου επιπέδου API, πάνω στο οποίο είναι βασισμένο το CUDA runtime. Αν και το driver API είναι δυσκολότερο στη χρήση - αφού απαιτεί τη ρητή (explicit) διατύπωση διαδικασιών για τις οποίες φροντίζει με διάφανο προς τον προγραμματιστή τρόπο το runtime – προσφέρει ένα μεγαλύτερο επίπεδο ελέγχου, απαραίτητο για κάποια σενάρια.

Για να γίνει η μεταγλώττιση (compilation) του πηγαίου κώδικα (source code) ενός CUDA C προγράμματος, το CUDA toolkit χρησιμοποιεί τον nvcc. Για ένα πρόγραμμα που χρησιμοποιεί το runtime, ο nvcc ουσιαστικά διαχωρίζει τον κώδικα που θα τρέξει στη gpu (device code) από τον υπόλοιπο κώδικα (host code). Στη συνέχεια, μεταγλωττίζει τον πρώτο είτε σε μορφή assembly (κώδικας ptx) είτε σε δυαδική μορφή (αντικείμενο cubin) και αντικαθιστά τις αντίστοιχες κλήσεις με κλήσεις του runtime. Μετά από αυτή την προεπεξεργασία (preprocess) αφήνει λοιπόν κώδικα έτοιμο να μεταγλωττιστεί από ένα

κοινό C μεταγλωττιστή (όπως ο gcc). Στην περίπτωση του driver API ο nvcc είναι απαραίτητος μόνο για τη μεταγλώττιση του κώδικα για τη gpu, αφού ο κώδικας αυτός καλείται ρητά (μέσω συγκεκριμένων συναρτήσεων) χωρίς να χρειάζεται προεπεξεργασία του υπόλοιπου κώδικα.

Παραθέτουμε παρακάτω ένα απλό πρόγραμμα σε C που υλοποιεί την πρόσθεση δύο διανυσμάτων ακεραίων. Στη συνέχεια, ακολουθούν (σύμφωνα και με αυτά που περιγράψαμε) δύο εκδοχές του σε CUDA C (για έκδοση του CUDA Toolkit >=4.0), για να δείξουμε τις απαραίτητες αλλαγές στον κώδικά και τις απαιτούμενες κλήσεις:

```
void add (cont int *a, const int *b , int *c) {
      int i;

      for (i=0; i<N; i++) {
            c [i] = a[i] + b[i] ;
      }
}

int main(void) {
      int i, a[N], b[N], c[N];

      for (i=0; i<N; i++) {
            a[i] = i;
            b[i] = i * i;
      }
      add(a, b, c);

      return 0;
}
```

Για να παραλληλοποιήσουμε το παραπάνω πρόγραμμα και να το τρέξουμε σε GPU χρησιμοποιώντας το CUDA Runtime API:

```
__global__ void add (int *a, int *b , int *c) {
      int i = blockDim.x * blockIdx.x + threadIdx.x;

      if (i<N) {
            c [i] = a[i] + b[i] ;
      }
}
```

```
int main(void) {
      int i, a[N], b[N], c[N], *dev_a, *dev_b, *dev_c;
      size_t size = N*sizeof(int);

      cudaMalloc((void **)&dev_a, size);
      cudaMalloc((void **)&dev_b, size);
      cudaMalloc((void **)&dev_c, size);

      for (i=0; i<N; i++) {
            a[i] = i;
            b[i] = i * i;


      cudaMemcpy(dev_a, a, size, cudaMemcpyHostToDevice);
      cudaMemcpy(dev_b, b, size, cudaMemcpyHostToDevice);

      add <<<1,N>>> (dev_a, dev_b, dev_c);

      cudaMemcpy(c, dev_c, size, cudaMemcpyDeviceToHost)

      cudaFree(dev_a);
      cudaFree(dev_b);
      cudaFree(dev_c);

      return 0;
}
```

Αντίστοιχα, για το CUDA Driver API:

```
extern \say{C} __global__ void add (int *a, int *b , int *c) {
      int i = blockDim.x * blockIdx.x + threadIdx.x;

      if (i<N) {
            c [i] = a[i] + b[i] ;
      }
}

int main(void) {
      int i, a[N], b[N], c[N];
      size_t size = N*sizeof(int);
      CUdeviceptr dev_a, dev_b, dev_c;
      CUdevice device;
      CUcontext context;
```

```
    CUmodule module;
    CUfunction function;
    void *args[] = {&dev_a, &dev_b, &dev_c};

    cuInit(0);

    cuDeviceGet(&device, 0);
    cuCtxCreate(&context, 0, device);
    cuModuleLoad(&module, ”add_kernel.ptx”);
    cuModuleGetFunction(&function, module, ”add”);

    cuMemAlloc(&dev_a, size);
    cuMemAlloc(&dev_b, size);
    cuMemAlloc(&dev_c, size);

    for (i=0; i<N; i++) {
          a[i] = i;
          b[i] = i * i;
    }

    cuMemcpyHtoD(dev_a, a, size);
    cuMemcpyHtoD(dev_b, b, size);

    cuLaunchKernel(function, 1, 1, 1, N, 1,1, 0, NULL, args, NULL);

    cuMemcpyDtoH(c, dev_c, size);

    cuMemFree(dev_a);
    cuMemFree(dev_b);
    cuMemFree(dev_c);

    cuCtxDestroy(context);

    return 0;
}
```

(ο χειρισμός των λαθών παραλείπεται για λόγους συντομίας και ευκρίνειας)

Σημειώσεις:

- Για την CUDA C ο κώδικας που τρέχει στη gpu αποτελείται από kernels (πυρήνες): τα kernels αυτά αποτελούν επέκταση των συναρτήσεων της

C.

- Το CUDA toolkit συντίθεται από κλειστού κώδικα (closed source) προγράμματα και βιβλιοθήκες. Για να χρησιμοποιηθούν τα διάφορα C APIs παρέχονται βιβλιοθήκες με τη μορφή επικεφαλίδων (headers) και κοινόχρηστων αντικειμένων (shared objects).
- Κατά τη διαδικασία της προεπεξεργασίας, για προγράμματα δηλαδή που είναι γραμμένα με το runtime API, οι κλήσεις των kernels δεν αντικαθίστανται μόνο με ρητές τεκμηριωμένες runtime κλήσεις. Χρησιμοποιούνται και κάποιες ατεκμηρίωτες - "εσωτερικές" - κλήσεις για τις οποίες είναι γνωστοί μόνο οι ορισμοί τους στις αντίστοιχες επικεφαλίδες.

2.2 Virtualization

Η έννοια virtualization (εικονικοποίηση) χρησιμοποιείται στην επιστήμη των υπολογιστών για να περιγράψει ένα μηχανισμό αφαίρεσης με σκοπό να αποκρύπτει, με τη χρήση "εικονικών" υπολογιστικών πόρων, από τους πελάτες των πόρων αυτών λεπτομέρειες για την υλοποίηση ή την κατάσταση των πραγματικών πόρων.

Ο όρος εικονική μηχανή αναφέρεται σε μια μηχανή-λογισμικό που εξομοιώνει μια φυσική μηχανή - στην πρώτη μπορεί δηλαδή να εκτελείται λογισμικό σαν να είναι η δεύτερη. Ο δημοφιλής αρχικός ορισμός των Popek και Goldberg [6] θεωρεί την εικονική μηχανή σαν "ένα αποδοτικό και απομονωμένο αντίγραφο μιας πραγματικής μηχανής". Πλέον η έννοια έχει επεκταθεί ώστε να περιλαμβάνει και υλοποιήσεις που δεν αποτελούν αντίγραφα πραγματικών μηχανών. Εκτός λοιπόν από τις εικονικές μηχανές συστήματος ο όρος περιλαμβάνει και τις εικονικές μηχανές διεργασίας.

Μια εικονική μηχανή διεργασίας εκτελείται μέσα σε ένα λειτουργικό σαν μία συνηθισμένη εφαρμογή και υποστηρίζει μία μόνο διεργασία. Πρόκειται για ένα πρόγραμμα-διερμηνέα που σκοπός του είναι να παρέχει ένα ανεξάρτητο από λειτουργικό σύστημα και υλικό προγραμματιστικό περιβάλλον (πχ. Java Virtual Machine). Μια εικονική μηχανή συστήματος - απλώς "εικονική μηχανή" (virtual machine - VM) στη συνέχεια του κειμένου - από την άλλη, υποστηρίζει την εκτέλεση ενός πλήρους λειτουργικού συστήματος. Ουσιαστικά επιτρέπει την εκτέλεση ενός λειτουργικού συστήματος (ή την ταυτόχρονη εκτέλεση παραπάνω του ενός) εμφωλευμένου σε κάποιο άλλο λειτουργικό, προσομοιώνει δηλαδή τη λειτουργία ενός πραγματικού υπολογιστικού συστήματος.

Για τη συνύπαρξη και τη διαχείριση των εικονικών μηχανών χρησιμοποιείται ο ελεγκτής εικονικής μηχανής (virtual machine monitor - VMM). Πρόκειται πρακτικά για το σύστημα που μεσολαβεί στην επικοινωνία των λειτουργικών των εικονικών μηχανών και του υλικού. Ευθύνεται, μεταξύ άλλων, για την πολύπλεξη των αιτημάτων για πρόσβαση στους πόρους του συστήματος από τα φιλοξενούμενα λειτουργικά (guest OS), αλλά και για τη διασφάλιση της απομόνωσης του περιβάλλοντος εκτέλεσης των εικονικών μηχανών. Μπορούμε να χωρίσουμε τους ελεγκτές εικονικής μηχανής σε δύο τύπους, ανάλογα με την αμεσότητα της επικοινωνίας τους με το υλικό (παρεμβολή ή όχι του λειτουργικού). Στον τύπο I ανήκουν οι ελεγκτές εικονικής μηχανής που τουλάχιστον ένα μέρος τους βρίσκεται σε άμεση επαφή με το υλικό - εδώ υπάρχουν δύο υποτύποι: ο επιβλέπων (hypervisor), που εκτελείται εξολοκλήρου πάνω στο υλικό και το λειτουργικό σύστημα υπηρεσίας (service OS), που ουσιαστικά λειτουργεί ως μια εικονική μηχανή αυξημένων δυνατοτήτων. Στον τύπο II ανήκουν οι ελεγκτές εικονικής μηχανής που εκτελούνται πάνω από ένα ήδη εγκατεστημένο λειτουργικό σύστημα (host OS) - ουσιαστικά σε αυτή την περίπτωση ο ελεγκτής προσφέρει ένα επίπεδο αφαίρεσης στην επικοινωνία μεταξύ host και guest λειτουργικού. Στη συνέχεια αυτής της εργασίας χρησιμοποιούμε εναλλάξιμα τους όρους “hypervisor” και ”ελεγκτής εικονικής μηχανής” για να περιγράψουμε το ίδιο πράγμα (τον τελευταίο) - όπως συνηθίζεται.

Μπορούμε επίσης να διακρίνουμε δύο κατηγορίες τεχνικών virtualization: την πλήρη (full virtualization) και τη μερική (paravirtualization). Στην πρώτη κατηγορία ανήκουν οι τεχνικές οι οποίες περιλαμβάνουν ελεγκτές που προσομοιώνουν πλήρως το υλικό του πραγματικού συστήματος προς τα εικονικά μηχανήματα - το λειτουργικό τους σύστημα δηλαδή μπορεί να λειτουργήσει χωρίς μετατροπές, σαν να χρησιμοποιεί απευθείας το υλικό. Στη δεύτερη κατηγορία, από την άλλη, δεν προσομοιώνεται επακριβώς το υλικό, αλλά χρησιμοποιούνται τροποποιημένοι (σε σχέση με τα κοινά λειτουργικά) οδηγοί ώστε να επικοινωνούν με τα κατάλληλα APIs του ελεγκτή.

Τόσο το Xen όσο και το KVM, οι δύο δημοφιλέστερες ανοιχτού κώδικα πλατφόρμες virtualization, παρέχουν τη δυνατότητα χρήσης εικονικών μηχανών και των δύο παραπάνω κατηγοριών. Ταυτόχρονα, υπάρχουν κατά περίπτωση και paravirtual drivers για κάποιες συσκευές που μπορούν να χρησιμοποιηθούν σε περιβάλλον πλήρους virtualization, συνδυάζοντας πλεονεκτήματα των δύο κατηγοριών. Στη συνέχεια παρουσιάζουμε (κυρίως) τον τρόπο που υλοποιούν τους paravirtual drivers καθώς και τους μηχανισμούς επικοινωνίας μεταξύ των εικονικών μηχανών που χρησιμοποιούν οι προαναφερόμενες πλατφόρμες, και ιδιαίτερα το Xen, το οποίο χρησιμοποιήσαμε για την πειραματική μας αποτίμηση - δεδομένου ότι η απόδοση του framework που υλοποιήσαμε

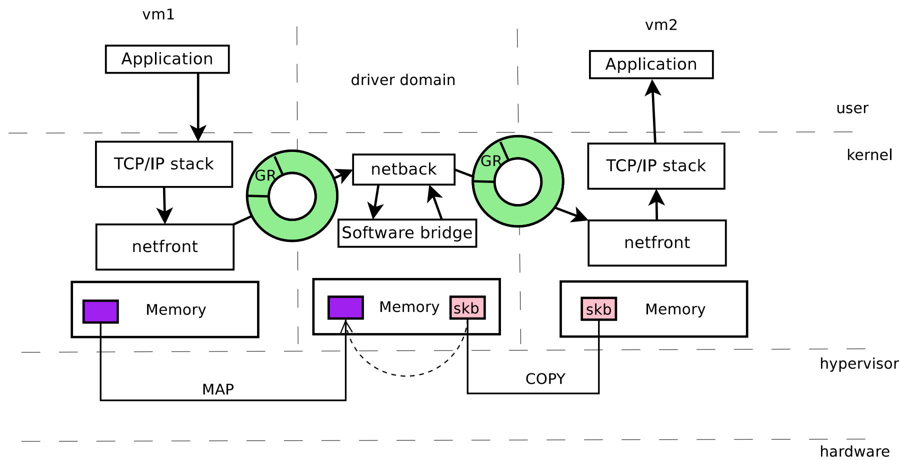
εξαρτάται σε μεγάλο βαθμό από τις επιδόσεις του δικτύου επικοινωνίας host-client, αυτό είναι το κομμάτι που έχει τη μεγαλύτερη σημασία για αυτή την εργασία.

Το Xen είναι ένας τύπου I hypervisor - που βρίσκεται δηλαδή, όπως είπαμε, σε άμεση επαφή με το υλικό. Σημειώνουμε εδώ ότι domain ονομάζουμε (στη συνέχεια) ένα ενεργό (που εκτελείται) στιγμιότυπο μιας εικονικής μηχανής. Πάνω στον hypervisor τρέχουν μια σειρά από εικονικές μηχανές. Το "domain 0" - στο εξής Dom0 - αποτελεί ένα ειδικό domain, με παραπάνω δικαιώματα, το οποίο περιλαμβάνει οδηγούς συσκευών, αλλά και εργαλεία για τη διαχείριση και των έλεγχο των υπόλοιπων εικονικών μηχανών και έχει άμεση επαφή με το υλικό. Τα υπόλοιπα domains (1..N) είναι κοινές εικονικές μηχανές - στο εξής DomUs - που επικοινωνούν με το Dom0 για την προσπέλαση στο υλικό [7].

Οι οδηγοί που υλοποιούν τη διαμοιραζόμενη πρόσβαση στις συσκευές Ε/Ε, όπως και οι περισσότεροι paravirtual drivers στο Xen, βασίζονται στο μοντέλο διαχωρισμένου οδηγού συσκευών (split driver model). Χωρίζονται δηλαδή σε δύο κομμάτια: τον frontend driver που βρίσκεται στο φιλοξενούμενο λειτουργικό σύστημα ενός DomU και τον backend driver που βρίσκεται στο λειτουργικό του Dom0. Ο backend driver παρέχει μια γενική, ανεξάρτητη του υλικού διεπαφή πάνω από τον πραγματικό οδηγό (native driver) και αναλαμβάνει την πολυπλεξία των αιτημάτων. Ο frontend driver, από την άλλη, προσφέρει μια εικονική διεπαφή - ίδια με εκείνη του πραγματικού οδηγού – και είναι συνήθως αρκετά απλός. Για την επικοινωνία μεταξύ frontend και backend, τη μεταφορά δηλαδή δεδομένων και μηνυμάτων μεταξύ των εικονικών μηχανών (intra-node ή inter-domain communication), οι βασικοί μηχανισμοί που χρησιμοποιούνται είναι τα Xen I/O Rings, το XenStore και οι δίαυλοι γεγονότων (event channels).

Τα Xen I/O Rings είναι μοιραζόμενοι χώροι μνήμης σε δομή δακτυλίου (ring buffers), δηλαδή κυκλικές ουρές που αποτελούνται από αιτήματα και απαντήσεις. Για να οριστεί μια σελίδα μνήμης που αρχικοποιείται με δομή δακτυλίου ως διαμοιραζόμενη χρησιμοποιείται ο πίνακας παραχώρησης. Ο πίνακας παραχώρησης (grant table), που ανήκει στον μηχανισμό παραχώρησης σελίδων (grant pages) του Xen, ενημερώνει τον hypervisor για το είδος των δικαιωμάτων που έχει κάθε domain στις σελίδες μνήμης του. Για τη μεταφορά πληροφοριών ελέγχου μεταξύ των DomUs και του Dom0 χρησιμοποιείται ο μηχανισμός XenStore, μια κεντρική βάση δεδομένων ρυθμίσεων, προσβάσιμη από όλα τα domains, αλλαγές στην οποία προκαλούν "γεγονότα" σε εικονικούς οδηγούς συσκευών. Τέλος, για την ενημέρωση και την εγκατάσταση διακοπών τόσο στο Dom0 όσο και στα DomUs δημιουργείται μεταξύ τους ένα σύστημα ενημέρωσης που αποτελείται από έναν δίαυλο γεγονότων, δύο θύρες

(μία σε κάθε άκρο του διαύλου) και δύο συναρτήσεις εξυπηρέτησης (μία σε κάθε θύρα του διαύλου).



Σχήμα 2.1: Ενδο-επικοινωνία εικονικών μηχανών στο Xen

Οι δικτυακές συσκευές Ε/Ε, λοιπόν, στο Xen βασίζονται στο μοντέλο διαχωρισμένου οδηγού συσκευών και τους μηχανισμούς που περιγράψαμε παραπάνω. Οι προεπιλεγμένες ρυθμίσεις χρησιμοποιούν για τη διαδικασία αυτή την κλασσική διεπαφή δικτύου. Για κάθε εικονικό μηχάνημα, το Xen δημιουργεί μια εικονική διεπαφή δικτύου και τη συνδέει με τη γέφυρα (bridge) που έχει ήδη δημιουργήσει πάνω από την πραγματική (native) διεπαφή του δικτύου. Για να φτάσει με αυτό τον τρόπο ένα πακέτο δεδομένων από μία εικονική μηχανή στην πραγματική διεπαφή του δικτύου θα πρέπει να περάσει από κάποια επιπλέον στάδια σε σχέση με ένα περιβάλλον χωρίς virtualization. Έτσι, αφού το πακέτο φτάσει στο κατώτερο επίπεδο της στοίβας TCP/IP θα συναντήσει την εικονική διεπαφή δικτύου (αντί για την πραγματική), δηλαδή τον frontend driver που περιγράψαμε παραπάνω (netfront). Το netfront θα τοποθετήσει στη συνέχεια μέσα στο μοιραζόμενο I/O ring την αντίστοιχη αναφορά (grant reference) του πίνακα παραχώρησης που δείχνει στη σελίδα μνήμης με τα δεδομένα και θα ειδοποιήσει το Dom0, μέσω του διαύλου γεγονότων, να τη διαβάσει. Μόλις πάρει την αναφορά το Dom0 θα αντιστοιχίσει τη σελίδα της σε δικό του χώρο μνήμης (page flipping) και θα βάλει τα δεδομένα σε μια άλλη δομή buffer για τη στοίβα δικτύου. Στην περίπτωση που αυτό το πακέτο δεδομένων προορίζεται για μία άλλη εικονική μηχανή, ο backend driver στο Dom0 χρειάζεται, αφού πάρει την αναφορά στο χώρο μνήμης του δεύτερου μηχανήματος από το I/O ring του τελευταίου, να αντιγράψει τα δεδομένα από το buffer του στο χώρο αυτό. Στη συνέχεια πρέπει να ειδοποιήσει τη δεύτερη εικονική μηχανή για τα δεδομένα που έγραψε, ώστε τελικά να τα πάρει το

netfront της και να τα προωθήσει, μέσω της στοίβας TCP/IP, στο χώρο χρήστη της. Ειδικά στην περίπτωση της επικοινωνίας μεταξύ εικονικών μηχανών, λοιπόν, παρατηρούμε ότι εισάγεται σημαντική επιβάρυνση στο μονοπάτι επικοινωνίας από τα περιττά στάδια.

2.3 Sockets - TCP/IP

Ο μηχανισμός των sockets είναι ο δημοφιλέστερος τρόπος διαδιεργασιακής επικοινωνίας (inter-process communication) σε ένα δίκτυο υπολογιστών. Το κυρίαρχο γενικής χρήσης πρωτόκολλο - ή μάλλον συλλογή πρωτοκόλλων - δικτύου είναι το TCP/IP, παρακάτω λοιπόν θα ασχοληθούμε με sockets σε TCP/IP δίκτυα.

Το TCP/IP είναι μια συλλογή πρωτοκόλλων επικοινωνίας στα οποία βασίζεται το Διαδίκτυο αλλά και πολλά άλλα δίκτυα υπολογιστών. Η ονομασία TCP/IP προέρχεται από τις συντομογραφίες των δυο κυριότερων πρωτοκόλλων που περιέχει το TCP (Transmission Control Protocol) και το IP (Internet Protocol). Η λειτουργικότητα που προσφέρει το TCP/IP οργανώνεται σε 4 στρώματα αφαίρεσης που χρησιμοποιούνται για την ταξινόμηση όλων των συναφών πρωτοκόλλων ανάλογα με το πεδίο εφαρμογής της δικτύωσης στο οποίο εμπλέκονται.

Το socket (υποδοχή) είναι ένα τελικό σημείο (endpoint) επικοινωνίας δύο συστημάτων σε ένα δίκτυο και ορίζεται από την τοπική διεύθυνση IP και την πόρτα (port) σε ένα σύστημα. Για την επικοινωνία επομένως μεταξύ των δύο συστημάτων χρειάζεται ο συνδυασμός δύο sockets τους αντίστοιχα (γνωστός και ως 4-tuple), συνδυασμός που είναι μοναδικός και χαρακτηρίζει συνεπώς και τη σύνδεσή τους. Χαρακτηριστικό του socket είναι επίσης και το πρωτόκολλο μεταφοράς που χρησιμοποιεί: εδώ υπάρχουν οι ευρέως χρησιμοποιούμενες επιλογές (του αντίστοιχου στρώματος του TCP/IP) TCP (connection-oriented, αξιόπιστο) και UDP (connectionless, αναξιόπιστο), αλλά και άλλες όπως το SCTP ή ακόμα και προσαρμοσμένα πρωτόκολλα για ειδικές χρήσεις.

Οι βασικές κλήσεις/συναρτήσεις της C (σύμφωνα με το πρότυπο POSIX.1-2008 [8]) που αφορούν τη δημιουργία και τη σύνδεση ενός socket είναι οι (τα ορίσματα και τα σφάλματα που επιστρέφονται παραλείπονται για λόγους συντομίας):

- socket(): Δημιουργεί ένα νέο, αδέσμευτο (unbound), socket και επιστρέφει έναν περιγραφητή αρχείου (file descriptor) που μπορεί να χρησιμο-

ποιηθεί από επόμενες κλήσεις που διαχειρίζονται τα sockets.

- bind(): Αντιστοιχίζει μια τοπική διεύθυνση σε ένα socket που δεν του έχει εκχωρηθεί τέτοια διεύθυνση.
- listen(): Επισημαίνει ένα connection-mode (σε λειτουργία σύνδεσης) socket ως δεχόμενο συνδέσεις.
- accept(): Παίρνει την πρώτη σύνδεση από την ουρά των εκκρεμών συνδέσεων, δημιουργεί ένα νέο, με ίδιου τύπου πρωτόκολλο και οικογένεια διευθύνσεων με το δοσμένο socket και εκχωρεί ένα νέο περιγραφητή αρχείου σε αυτό το socket – το αρχικό socket παραμένει ανοιχτό και μπορεί να δεχτεί (accept) και άλλες συνδέσεις, ενώ ελλείψει εκκρεμούς σύνδεσης η κλήση μπλοκάρει.
- connect(): Επιχειρεί να δημιουργήσει μια σύνδεση για το δοσμένο socket αν αυτό είναι connection-mode ή να θέσει/αλλάξει μόνο την peer διεύθυνσή του (διεύθυνση του socket με το οποίο θέλουμε να επικοινωνήσουμε) αν αυτό είναι connectionless-mode (σε λειτουργία χωρίς σύνδεση).

Για τις περιπτώσεις ασύγχρονων συνδέσεων, προσφέρονται και άλλες συναρτήσεις - pselect(), select(), και poll() - που ελέγχουν την κατάσταση ενός socket (έτοιμο για εγγραφή/ανάγνωση ή με σφάλματα).

Τέλος, να σημειώσουμε τη δυνατότητα ενός socket να είναι είτε blocking είτε non-blocking: Η προεπιλεγμένη συμπεριφορά ενός socket είναι, όταν πχ. έχουμε μια κλήση ανάγνωσης, να "μπλοκάρει" μέχρι να λάβει δεδομένα του αντίστοιχου αιτήματος (blocking socket) - αντίστοιχα για κλήσεις εγγραφής κλπ. Υπάρχει όμως και η δυνατότητα τέτοιες κλήσεις να επιστρέφουν πάντα, χωρίς να περιμένουν να ολοκληρωθεί η αντίστοιχη λειτουργία. Σε αυτή την περίπτωση, η επικοινωνία μεταξύ των sockets γίνεται ασύγχρονα.

Ακολουθεί ένα παράδειγμα μια απλής υλοποίησης για tcp sockets (ο χειρισμός των λαθών παραλείπεται για λόγους συντομίας και ευκρίνειας):

Κώδικας για τον server:

```
int main() {
   int socket_fd, client_sock_fd;
   struct sockaddr_in client_addr;
   struct addrinfo hints, *addr;
   const char *port = PORT;

   memset(&hints, 0, sizeof(hints));
```

```
    hints.ai_family = AF_INET; // Allow IPv4
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE; // For wildcard IP address
    hints.ai_protocol = 0; // Any protocol
    hints.ai_canonname = NULL;
    hints.ai_addr = NULL;
    hints.ai_next = NULL;

    getaddrinfo(NULL, port, &hints, &addr);

    socket_fd = socket(addr->ai_family, addr->ai_socktype, addr->
        ai_protocol);

    bind(socket_fd, (struct sockaddr*)addr->ai_addr, addr->ai_addrlen)
        ;

    listen(socket_fd, 10);

    for (;;) {
       client_sock_fd = accept(socket_fd, (struct sockaddr*)&
           client_addr, &s);

       // Read, Write etc. ...

      close(client_sock_fd);
    }
}
```

Κώδικας για τον client:

```
int main() {
    int socket_fd;
    struct addrinfo hints, *addr;
    const char *server_port = PORT, *server_ip = IP;

    memset(&hints, 0, sizeof(hints));
    hints.ai_family = AF_INET; // Allow IPv4
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE; // For wildcard IP address
    hints.ai_protocol = 0; // Any protocol
    hints.ai_canonname = NULL;
    hints.ai_addr = NULL;
    hints.ai_next = NULL;
```

```
    getaddrinfo(server_ip, server_port, &hints, &addr);

    socket_fd = socket(addr->ai_family, addr->ai_socktype, addr->
        ai_protocol);

    connect(socket_fd, (struct sockaddr*)s_addr->ai_addr, s_addr->
        ai_addrlen);

    // Read, Write etc. ...

    close(socket_fd);
}
```

2.4 Data serialization

Με τον όρο data serialization (σειριοποίηση δεδομένων) αναφερόμαστε στη διαδικασία μετατροπής της πληροφορίας των δεδομένων (νούμερων, κειμένου, αντικειμένων, δομών δεδομένων κλπ) σε μία μορφή ώστε να μπορεί αποθηκευθεί σε κάποιο μέσο ή να αποσταλεί μέσω δικτύου. Από αυτή τη μορφή μπορούν αργότερα να επαναδημιουργηθούν - στον ίδιο ή σε άλλον υπολογιστή - αυτά τα δεδομένα με την αντίστροφη διαδικασία (deserialization - αποσειριοποίηση).

Η χρήση της τεχνικής του serialization, λοιπόν, είναι χρήσιμη - αν όχι απαραίτητη - σε εφαρμογές που τρέχουν σε διαφορετικούς υπολογιστές και χρειάζεται να επικοινωνούν σε ένα δίκτυο. Αφενός, γιατί προσφέρει τη στοιχειώδη αφαίρεση (abstraction) ώστε να μπορούν τα δεδομένα που ανταλλάσσονται να παρασταθούν παντού με τον ίδιο τρόπο, αφού δεν αντιγράφεται απλώς το περιεχόμενο της μνήμης αλλά ανταλλάσσεται η ίδια η πληροφορία σε μια προσυμφωνημένη μορφή (πχ. μια int στη C δεν έχει για τον μεταγλωττιστή απαραίτητα το ίδιο μέγεθος για διαφορετικές αρχιτεκτονικές επεξεργαστών). Αφετέρου, γιατί δίνει τη δυνατότητα δημιουργίας "πρωτοκόλλων" επικοινωνίας που θα μπορούν να χρησιμοποιηθούν από προγράμματα σε οποιαδήποτε γλώσσα - με την προϋπόθεση προφανώς της ύπαρξης της αντίστοιχης βιβλιοθήκης.

Η αναπαράσταση της απαιτούμενης πληροφορίας γίνεται είτε σε μορφή κειμένου (textual - όπως πχ. του xml) είτε σε δυαδική (binary) μορφή. Το serial-

ization σε μορφή κειμένου έχει το πλεονέκτημα ότι μπορεί να διαβαστεί από τον άνθρωπο - άρα είναι ευκολότερο στην αποσφαλμάτωση - και δεν απαιτεί ιδιαίτερη μέριμνα για την αρχιτεκτονική του κάθε επεξεργαστή (πχ. το endianness). Το serialization σε δυαδική μορφή, από την άλλη, απαιτεί λιγότερο χώρο και είναι ταχύτερο - παράγει πολύ μικρότερο overhead - για τις περιπτώσεις που δε χρειαζόμαστε την αναγνωσιμότητα και είναι σημαντική η αποδοτικότητα του κώδικα.

Τα Protocol Buffers είναι μια δημοφιλής βιβλιοθήκη από τη Google για serialization σε δυαδική μορφή δεδομένων. Επίσημα υποστηρίζει Java, C++ και Python [9] ενώ υπάρχει και αντίστοιχη (ανεπίσημη) υλοποίηση για C (protobuf-c).

Για να χρησιμοποιήσει ο προγραμματιστής τα Protocol Buffers αρκεί να προσδιορίσει τη δομή της πληροφορίας που θέλει να κάνει serialize, ορίζοντας τα κατάλληλα protocol buffer "μηνύματα" στα αντίστοιχα .proto αρχεία. Κάθε τέτοιο μήνυμα είναι μια μικρή λογική εγγραφή πληροφορίας, που περιέχει μια σειρά από ζεύγη ονόματος-τιμής. Στη συνέχεια, μπορεί κάποιος να παράγει, χρησιμοποιώντας τον protocol buffer μεταγλωττιστή, τις αντίστοιχες - απλές στη χρήση - συναρτήσεις/κλάσεις πρόσβασης στα δεδομένα [10].

Για παράδειγμα, έστω ότι θέλουμε να ανταλάξουμε μηνύματα που περιέχουν δύο int32 μεταβλητές. Το αντίστοιχο amessage.proto (για την περίπτωση του protobuf-c) θα περιέχει:

```
message AMessage {
      required int32 a=1;
      optional int32 b=2;
}
```

Και αφού παράγουμε τα κατάλληλα .h και .c αρχεία με την:

```
protoc-c --c_out=. amessage.proto
```

μπορούμε να χρησιμοποιήσουμε για την αποστολή και λήψη μηνυμάτων τις amessage__get_packed_size(), amessage__pack() και amessage__unpack() αντίστοιχα ως εξής [11] (κομμάτια του κώδικα που δεν αφορούν άμεσα τη χρήση του PB και ο χειρισμός των λαθών παραλείπονται για λόγους συντομίας και

ευκρίνειας):

1)

```
[...]
#include \say{amessage.pb-c.h}

int main (int argc, const char * argv[]) {
      AMessage msg = AMESSAGE__INIT; // AMessage
      void *buf; // Buffer to store serialized data
      unsigned len; // Length of serialized data

      [...]

      msg.a = atoi(argv[1]);
      if (argc == 3) { msg.has_b = 1; msg.b = atoi(argv[2]); }
      len = amessage__get_packed_size(&msg);

      buf = malloc(len);
      amessage__pack(&msg,buf);

      [...]
      // Write/Send buf (of known length len) using write/fwrite etc.
      [...]

      free(buf); // Free the allocated serialized buffer
      return 0;
}
```

2)

```
[...]
#include \say{amessage.pb-c.h}

int main (int argc, const char * argv[]) {
      AMessage *msg;
      uint8_t buf[MAX_MSG_SIZE];
      size_t msg_len;

      [...]
      // Read packed message to buf and return its length to msg_len.
      [...]
```

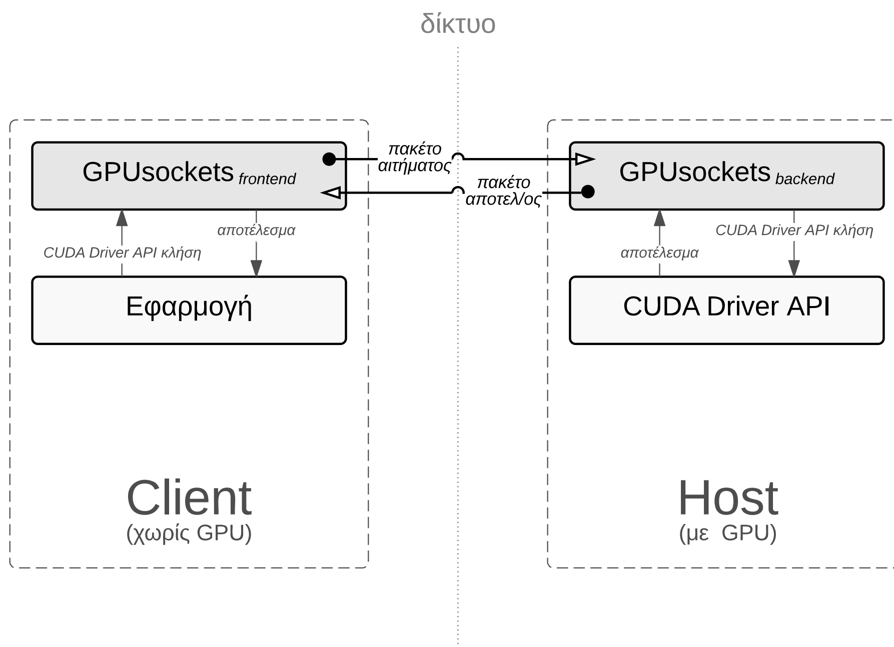
```
    // Unpack the message using protobuf-c.
    msg = amessage__unpack(NULL, msg_len, buf);
    [...]

    // display the message’s fields.
    printf(”Received:␣a=%d”,msg->a); // required field
    if (msg->has_b) // handle optional field
        printf(”␣␣b=%d”,msg->b);

    printf(”\n”);

    // Free the unpacked message
    amessage__free_unpacked(msg, NULL);
    return 0;
}
```

3 Σχεδίαση/Υλοποίηση

Το framework που παρουσιάζουμε σε αυτή την εργασία (GPUsockets) επιχειρεί να δώσει μια αποδοτική λύση: α) για την εκτέλεση υπολογιστικών φορτίων σε απομακρυσμένες GPUs, και β) ιδιαίτερα για το διαμοιρασμό GPUs σε εικονικά περιβάλλοντα βασισμένα σε συστοιχίες υπολογιστών για χρήση με τέτοια φορτία (ιδιαίτερα αναφερόμενα σε HPC). Αυτό που είναι σημαντικό εκτός από τις επιδόσεις σε μια τέτοια λύση είναι η διαφάνεια προς το χρήστη. Έτσι, στη δική μας υλοποίηση, ένα εκτελέσιμο αρχείο γραμμένο για το CUDA driver API μπορεί να εκτελεστεί από έναν οποιοδήποτε κόμβο χωρίς GPU (client), χρησιμοποιώντας μια βιβλιοθήκη που προσφέρει το ίδιο API εικονικά. Οι πληροφορίες και τα δεδομένα της κάθε κλήσης αποστέλλονται από αυτή τη βιβλιοθήκη (που δέχεται ουσιαστικά τις κλήσεις αντί για το CUDA toolkit) χρησιμοποιώντας ένα προσαρμοσμένο πρωτόκολλο, αφού πρώτα τα κωδικοποιήσει σε ένα μήνυμα με συγκεκριμένη μορφή. Μέσω του δικτύου το μήνυμα αυτό φτάνει στον κόμβο με τη GPU (host) όπου μια άλλη εφαρμογή-υπηρεσία του framework το δέχεται, το αποκωδικοποιεί και αφού εκτελέσει την κλήση χρησιμοποιώντας το CUDA toolkit επιστρέφει ένα αντίστοιχο κωδικοποιημένο μήνυμα με το αποτέλεσμα. Τέλος, το αποτέλεσμα επιστρέφεται από τη βιβλιοθήκη στον client, αφού ληφθεί και αποκωδικοποιηθεί το μήνυμα αυτό.

3.1 Γενική Περιγραφή

Αυτά τα δύο μέρη που αναφέραμε παραπάνω, η υπηρεσία εξυπηρέτησης στον host (backend) και η βιβλιοθήκη-μεσολαβητής στους clients (frontend) που λαμβάνει τις κλήσεις προς το CUDA toolkit αποτελούν πρακτικά το μοντέλο διαχωρισμένου οδηγού συσκευών, στο οποίο βασίζεται η υλοποίησή μας. Χρησιμοποιώντας τη βιβλιοθήκη μπορούμε ουσιαστικά να "ξαναγράψουμε" το περιεχόμενο της αντίστοιχης συνάρτησης που καλείται κάθε φορά, ώστε επέμβουμε στη λειτουργικότητά της ανάλογα με τις ανάγκες της υλοποίησής μας. Το προσαρμοσμένο πρωτόκολλο που χρησιμοποιείται για τη μεταξύ τους επικοινωνία, αν και είναι βασισμένο σε TCP sockets, είναι σχεδιασμένο λαμβάνοντας υπόψη τη δυνατότητα μελλοντικής επέκτασης και με άλλους μηχανι-



Σχήμα 3.1: Τρόπος λειτουργίας του GPUsockets

σμούς. Προσφέρει επομένως μια σχετική αφαίρεση και, ουσιαστικά, συντίθεται από δύο επίπεδα.

Το υψηλότερο επίπεδο αναλαμβάνει το πακετάρισμα/ξε-πακετάρισμα των δεδομένων που θέλουμε να στείλουμε/λάβουμε αντίστοιχα στο/από το δίκτυο. Συγκεκριμένα, κατά το πακετάρισμα λαμβάνει τα καθαρά δεδομένα, τα κάνει serialize και επιστρέφει το αποτέλεσμα σε μορφή μηνύματος-πακέτου (στο εξής απλώς “μήνυμα”, η δομή του οποίου περιγράφεται παρακάτω). Η αντίστροφη διαδικασία, δοσμένων των serialized καθαρών δεδομένων ενός μηνύματος, εφαρμόζει το αντίστοιχο deserialize. Το επίπεδο αυτό υλοποιείται με τις συναρτήσεις encode_message() και decode_message().

Το χαμηλότερο επίπεδο έχει την ευθύνη της αποστολής/λήψης των μηνυμάτων που παράγονται από το υψηλότερο. Έτσι, ρόλος του είναι η αποστολή ενός πακεταρισμένου μηνύματος γράφοντάς το στο αντίστοιχο TCP socket, και αντίστροφα, η ανάγνωση ενός μηνύματος από το TCP socket και η εξαγωγή των - serialized - καθαρών δεδομένων που μεταφέρει. Αυτό το επίπεδο υλοποιείται με τις συναρτήσεις send_message() και receive_message().

Τα μηνύματα που ανταλλάσσονται μεταξύ του frontend και του backend αποτελούνται πρακτικά από δύο κομμάτια: τα (μεταβλητού μήκους) serialized δε-

δομένα/περιεχόμενο του μηνύματος και το σταθερού μήκους (uint32_t) μέγεθος των δεδομένων αυτών. Ο λόγος που στην αρχή του κάθε μηνύματος βρίσκεται ένα πεδίο με το - γνωστού μήκους - μέγεθος του πραγματικού περιεχομένου του μηνύματος είναι για να ξέρει η μεριά που το λαμβάνει ακριβώς πόσα bytes να περιμένει. Έτσι, αφού διαβαστεί η γνωστού μήκους μεταβλητή που έχει το μέγεθος των δεδομένων, το frontend/backend γνωρίζει ακριβώς ποιο είναι το μέγεθος της πληροφορίας που πρέπει να διαβάσει στη συνέχεια και συνεπώς πόσο ακριβώς χώρο μνήμης χρειάζεται να δεσμεύσει και πότε να σταματήσει να περιμένει άλλα δεδομένα. Το serialized περιεχόμενο του μηνύματος δημιουργείται χρησιμοποιώντας το Protocol Buffers όπως εξηγήσαμε στο κεφ. 2.2 και περιλαμβάνει τον τύπο του μηνύματος (πχ. εντολή CUDA προς εκτέλεση - CUDA_CMD) και το "ωφέλιμο φορτίο" (payload, πχ. αν πρόκειται για CUDA_CMD περιέχει μεταβλητές και ορίσματα συναρτήσεων). Να σημειώσουμε, τέλος, ότι όλα τα δεδομένα που στέλνονται/λαμβάνονται στο δίκτυο ακολουθούν συγκεκριμένη - γνωστή - αναπαράσταση (network byte order/big endian), η αποκωδικοποίησή τους είναι ανεξάρτητη, δηλαδή, της αρχιτεκτονικής των host και client υπολογιστικών συστημάτων.

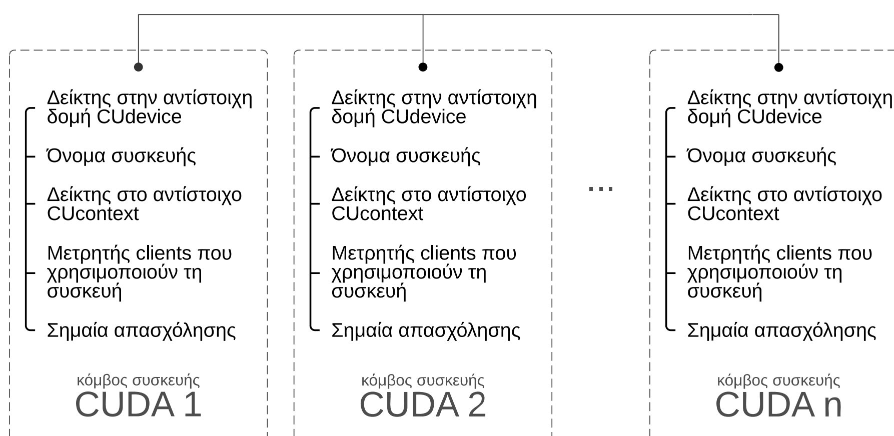
Για να δημιουργηθεί η σύνδεση μεταξύ των δυο κομματιών του μοντέλου διαχωρισμένου οδηγού συσκευών είναι απαραίτητες κάποιες διαδικασίες αρχικοποίησης στον host και στον κάθε client. Έτσι, στην υλοποίησή μας με sockets το backend αρχικά δημιουργεί και κάνει διαθέσιμο ένα socket, ενώ το frontend στον κάθε client δημιουργεί - για κάθε εντολή - το δικό του socket και συνδέεται με αυτό του backend για να γίνει δυνατή η αποστολή και λήψη μεταξύ τους αιτημάτων και απαντήσεων. Η IP και η πόρτα του host που χρησιμοποιείται για αυτή την επικοινωνία δίνεται μέσω των μεταβλητών περιβάλλοντος GPUSOCK_SERVER και GPUSOCK_PORT, αλλιώς επιχειρείται η χρήση προκαθορισμένων τιμών που έχουν οριστεί στο αρχείο επικεφαλίδας common.h πριν τη μεταγλώττιση.

Στη συνέχεια, για να εξυπηρετηθούν αυτά τα αιτήματα χρειάζεται να αρχικοποιηθούν και να χρησιμοποιηθούν κάποιες επιπλέον δομές εκατέρωθεν:

- Το backend διατηρεί δύο (διπλά συνδεδεμένες) λίστες: μία που αφορά τις διαθέσιμες συσκευές CUDA και μία που αφορά τους clients που είναι συνδεδεμένοι. Η πρώτη περιέχει πληροφορίες για όλες τις συσκευές CUDA που έχουν αναγνωριστεί ως διαθέσιμες στο σύστημα του host. Η λίστα αυτή συμπληρώνεται κατά την εκκίνηση του backend: αφού αρχικοποιηθεί για χρήση το CUDA driver API, αναγνωρίζεται και αρχικοποιείται η κάθε προς χρήση συσκευή και οι αντίστοιχες απαραίτητες πληροφορίες (όπως πχ. αναγνωριστικό) τοποθετούνται σαν κόμβος σε

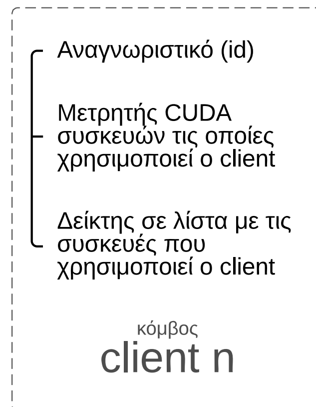
αυτή τη λίστα. Η δεύτερη λίστα περιέχει πληροφορίες για τους clients που συνδέονται στο backend και αρχικοποιείται και αυτή κατά την εκκίνηση. Κάθε φορά που το frontend κάποιου client συνδέεται με επιτυχία στο backend, το τελευταίο προσθέτει σε αυτή τη λίστα έναν κόμβο με ένα αναγνωριστικό και στη συνέχεια ανανεώνει αναλόγως τον αριθμό και τους αντίστοιχους δείκτες για τις συσκευές CUDA που απασχολεί ο client. Τα δεδομένα που αποθηκεύονται στις δύο αυτές δομές χρησιμοποιούνται από το backend για την αποτελεσματική εξυπηρέτηση των αιτημάτων-εντολών που δέχεται, αλλά και για τον διαμοιρασμό των συσκευών που διαχειρίζεται.

- Το frontend, χρειάζεται από τη μεριά του να διατηρεί μια δομή που περιέχει πληροφορίες για τη σύνδεσή του με το backend αλλά και (διπλά συνδεδεμένες) λίστες με "αντιστοιχίσεις" για τους δείκτες (C pointers) που είναι απαραίτητοι στις διαφορές εντολές που καλείται να εκτελέσει. Εφόσον όλες οι κλήσεις στο CUDA toolkit εκτελούνται στην πραγματικότητα στον host, όλοι οι δείκτες που χρησιμοποιούνται από αυτές αφορούν σε χώρους μνήμης του τελευταίου - δεν έχουν δηλαδή, σαν μεταβλητές, "νόημα" για τον client. Όμως για να είναι σε θέση το backend να ακολουθήσει τη ροή των CUDA εντολών του frontend, είναι απαραίτητο ο client να δέχεται σε κάποια μορφή τους δείκτες αυτούς ώστε να τους ξαναστείλει σε όποιες επόμενες κλήσεις τους χρειάζονται. Για να επιτευχθεί αυτό - και να αποφευχθούν τα προβλήματα της πιθανής διαφοράς μεγέθους των δεικτών server/client - το frontend δημιουργεί για κάθε δείκτη που απαιτείται από τις προς εκτέλεση κλήσεις, έναν κόμβο στη λίστα του. Ο κόμβος αυτός περιέχει τον ίδιο το δείκτη σε μορφή (μη προσημασμένου) ακεραίου (όπως τον έχει λάβει από το backend) και



Σχήμα 3.2: Λίστα CUDA συσκευών του backend

ένα αναγνωριστικό. Με αυτό τον τρόπο, στη συνέχεια, μπορεί χρησιμοποιώντας το αναγνωριστικό αυτό να αποστείλει αιτήματα στον host που θα περιλαμβάνουν τον αντίστοιχο δείκτη χωρίς να χρειαστεί ποτέ να χρησιμοποιηθεί η πραγματική περιοχή μνήμης στην οποία αναφέρεται από τον client.



Σχήμα 3.3: Κόμβος της λίστας clients του backend

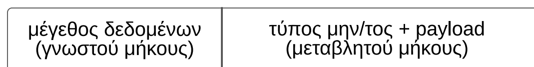
Για να είναι αποδοτική (όπως θα δείξουμε στο κεφ.5 παρακάτω) η εξυπηρέτηση των αιτημάτων υλοποιήσαμε την πολύπλεξή τους στο backend χρησιμοποιώντας τη βιβλιοθήκη pthread, που επιτρέπει τη χρήση νημάτων POSIX (POSIX threads). Έτσι, κάθε σύνδεση frontend (client) που δημιουργείται επιτυχώς, εξυπηρετείται (ασύγχρονα) στο δικό της, ξεχωριστό νήμα.

Τέλος, η υλοποίησή μας υποστηρίζει το CUDA driver API (>4.0) για δύο κυρίως λόγους. Πρώτον, γιατί το runtime API, παρόλο που θεωρείται το πιο χρησιμοποιούμενο μιας και προσφέρει ένα υψηλότερο επίπεδο αφαίρεσης, χρησιμοποιεί ατεκμηρίωτες/εσωτερικές μεθόδους και συναρτήσεις για να προσφέρει αυτή την ευκολία, που μπορούν να αλλάξουν χωρίς ειδοποίηση σε οποιαδήποτε επόμενη έκδοση του CUDA toolkit. Δεύτερον, και πιο σημαντικά, γιατί η (χαμηλού επιπέδου) δομή του driver API προσομοιάζει σε άλλα προγραμματιστικά μοντέλα επιταχυντών, επιτρέπει έτσι τη μελλοντική επέκταση της λύσης μας για την υποστήριξη frameworks όπως το OpenCL.

3.2 Διαδικασία Εξυπηρέτησης CUDA κλήσης

Στη συνέχεια θα παρουσιάσουμε τη διαδικασία εξυπηρέτησης μια κλήσης CUDA όπως θα συνέβαινε σε ένα υπολογιστικό σύστημα/client (χωρίς GPU) χρησι-

μοποιώντας τη δική μας προσέγγιση και ένα άλλο υπολογιστικό σύστημα/host με GPU. Το GPUsockets μπορεί να χρησιμοποιηθεί για ένα οποιοδήποτε εκτελέσιμο είναι φτιαγμένο για το CUDA driver API, απλά προ-φορτώνοντας (με LD_PRELOAD) την κατάλληλη βιβλιοθήκη (libcudawrapper.so) με παραμέτρους τα στοιχεία - IP/hostname και πόρτα - που δηλώνουν την τοποθεσία του backend. Προφανώς πρώτα έχουμε φροντίσει ώστε να εκτελείται το backend στον host με τα ίδια στοιχεία. Κατά την εκτέλεση της πρώτης CUDA κλήσης, δηλαδή πάντα της cuInit() η οποία χρησιμοποιείται κανονικά πριν από κάθε άλλη κλήση για την αρχικοποίηση του CUDA driver API, το frontend αρχικοποιεί τη δομή με τις πληροφορίες που αναφέραμε στην προηγούμενη υποπαράγραφο. Ακολούθως, δημιουργεί ένα socket, συνδέεται με το ανοιχτό socket του host και παίρνει ένα αναγνωριστικό (id) από τον τελευταίο, το οποίο αποθηκεύει.



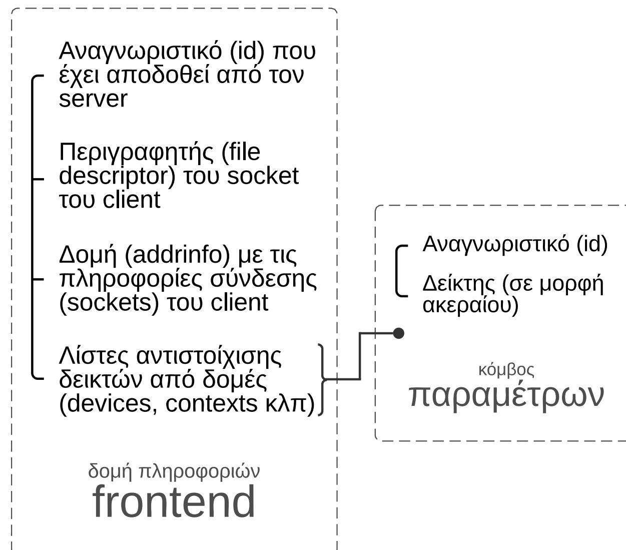
Σχήμα 3.4: Δομή μηνύματος του GPUsockets

Για την εκτέλεση μιας οποιασδήποτε άλλης κλήσης ακολουθείται η εξής διαδικασία: Αφού ελεγχθεί η διαθεσιμότητα (get_server_connection()) της σύνδεσης με τον host, στη συνέχεια τα ορίσματα της υπό εκτέλεση συνάρτησης που χρειάζεται να του αποσταλούν (που αποτελούν δηλαδή δεδομένα εισόδου) γίνονται serialize με τη χρήση του Protocol Buffers και αποθηκεύονται σε ένα μήνυμα με τη δομή που φαίνεται στο σχήμα 3.4.

Το στάδιο αυτό υλοποιείται με τη συνάρτηση send_cuda_cmd() η οποία πρακτικά αποτελείται από δύο διαδικασίες. α) Δοσμένων των προς αποστολή δεδομένων, εκτελεί το ίδιο το serialization (βλ. και κεφ. 2.2) και δημιουργεί σε έναν buffer με τη χρήση της pack_cuda_cmd() και της encode_message() το μήνυμα που θα αποσταλεί στον host. β) Χρησιμοποιώντας τον buffer αυτόν και με την send_message() αποστέλλει το μήνυμα μέσω της σύνδεσης των TCP sockets που έχει ήδη δημιουργηθεί.

Αφού αποσταλεί το μήνυμα στον host, εκτελείται η get_cuda_cmd_result() η οποία αναμένει την απάντηση με το αποτέλεσμα. Τα βήματα που αποτελούν αυτή τη διαδικασια είναι τα εξής: Αρχικά λαμβάνεται το μήνυμα, τα (serialized) δεδομένα του οποίου αποθηκεύονται από τη receive_message() σε έναν buffer. Στη συνέχεια, η decode_message() αναλαμβάνει να κάνει τα δεδομένα deserialize και αφού ολοκληρωθεί επιτυχώς αυτή η διαδικασία το επιστρέφεται το αναμενόμενο αποτέλεσμα.

Τέλος, αν το αποτέλεσμα περιέχει κάποιον δείκτη, αυτός αντιστοιχίζεται με



Σχήμα 3.5: Δομή πληροφοριών και λίστες frontend

ένα (εκ των προτέρων) γνωστού μεγέθους αναγνωριστικό και η αντιστοίχηση αυτή αποθηκεύεται σε μια λίστα, η δομή της οποίας φαίνεται στο σχήμα 3.5.

Από τη μεριά του backend, από τη στιγμή που δημιουργείται η σύνδεση των TCP sockets αρχίζει η διαδικασία λήψης και επεξεργασίας του αιτήματος του frontend ώστε να αποσταλεί στον client η κατάλληλη απάντηση. Μόλις δημιουργηθεί επιτυχώς η σύνδεση ξεκινάει ένα καινούργιο pthread (ανά client) ώστε να συνεχίσει εκεί η εξυπηρέτηση του αιτημάτων. Αφού ακολουθώντας την ίδια διαδικασία με το frontend - receive_message() και decode_message() - λάβει το backend τα καθαρά δεδομένα του αιτήματος, τα επεξεργάζεται ώστε να διαβάσει τα αντίστοιχα ορίσματα και τον τύπο της CUDA κλήσης που πρέπει να εκτελεστεί χρησιμοποιώντας την process_cuda_cmd(). Στη συνέχεια εκτελεί την κλήση με το CUDA Driver API και πακετάρει και αποστέλλει το αποτέλεσμα με τη χρήση των pack_cuda_cmd(), encode_message() και send_message(), αντίστοιχα με το frontend. Αν η κλήση είναι η cuInit() που αναφέραμε παραπάνω, προσθέτει τον client στη λίστα που διατηρεί και του αποδίδει ένα αναγνωριστικό, το οποίο και του επιστρέφει. Τέλος, μετά την ολοκλήρωση της επεξεργασίας ελέγχει αν ο client έχει ολοκληρώσει τυπικά την εκτέλεση της εφαρμογής του με την get_client_status() για να απελευθερώσει τους αντίστοιχους πόρους, αλλιώς αναμένει την ανάγνωση του επόμενου μηνύματος/αιτήματος του αντίστοιχου frontend.

4

Πειραματική Αποτίμηση

Για να αναλύσουμε τη συμπεριφορά και τις επιδόσεις του framework που υλοποιήσαμε και παρουσιάσαμε στο Κεφ.3, χρησιμοποιήσαμε δύο GPGPU εφαρμογές γραμμένες για το CUDA driver API: μία που εκτελεί μια απλή πράξη μεταξύ πινάκων (MatSum) και μία υπολογιστικά απαιτητική που εκτελεί την επαναληπτική μέθοδο Jacobi (Jacobi stencil). Σκοπός είναι η αποτίμηση της προσέγγισής μας σε σχέση με την απευθείας εκτέλεση σε CUDA GPU σε Xen περιβάλλον με πολλαπλά VMs, αλλά και του τρόπου με τον οποίο επηρεάζεται από τον συμβατικό μηχανισμό επικοινωνίας των VMs, αξιοποιώντας εναλλακτικές μεθόδους επικοινωνίας.

4.1 MatSum και Jacobi Stencil

Το MatSum εκτελεί την πρόσθεση δύο πινάκων (arrays της C) με δοσμένο αριθμό στοιχείων/μέγεθος. Η διαδικασία είναι σχετικά απλή: αφού αρχικοποιηθεί το driver api και γίνουν οι απαραίτητες ενέργειες για την παραχώρηση της αντίστοιχης CUDA συσκευής, context κλπ., αρχικοποιούνται οι αντίστοιχες μεταβλητές στον host (cpu) και στη gpu, εκτελείται το CUDA kernel που υλοποιεί την παράλληλη πρόσθεση στοιχείο-στοιχείο των δύο πινάκων στη gpu και στη συνέχεια το αποτέλεσμα αντιγράφεται πίσω στον host και ελέγχεται.

Το Jacobi stencil εκτελεί, για δοσμένο αριθμό επαναλήψεων και μέγεθος πίνακα, τον αλγόριθμο Jacobi (που παρουσιάζεται παρακάτω σε μορφή ψευδοκώδικα), δημοφιλή επαναληπτική μέθοδο που χρησιμοποιείται για την επίλυση μερικών διαφορικών εξισώσεων Laplace και Poisson σε δύο διαστάσεις. Όπως φαίνεται και από τον κώδικα (όπου T ο αριθμός των επαναλήψεων και N οι διαστάσεις του πίνακα), οι T επαναλήψεις πρέπει να εκτελεστούν σειριακά, αφού οι εσωτερικοί τους υπολογισμοί μεταβάλλουν γειτονικά στοιχεία του πίνακα. Ο φόρτος, λοιπόν, της εκτέλεσης της εφαρμογής επηρεάζεται περισσότερο από τον αριθμό T των εξωτερικών επαναλήψεων - αφού πρέπει μετά από κάθε επανάληψη, η επόμενη να περιμένει να έχουν γραφτεί όλες οι τιμές της προηγούμενης - ενώ η παραλληλοποίηση εξαρτάται από το μέγεθος N των

διαστάσεων του πίνακα - αφού οι εσωτερικές επαναλήψεις αφορούν υπολογισμούς που μπορούν να εκτελεστούν παράλληλα, υπό την προϋπόθεση ότι σε κάθε χρονικό βήμα το αποτέλεσμα γράφεται στην κύρια μνήμη και ξαναδιαβάζεται στο επόμενο (εφόσον επηρεάζονται γειτονικά στοιχεία). Το σημαντικά διαφορετικό, επομένως, βήμα σε σχέση με το MatSum, είναι το πιο σύνθετο CUDA kernel που υπολογίζει τη συνάρτηση A[t][i][j] και η εκτέλεσή του - ουσιαστικά η πολλαπλή κλήση του με διαφορετικά ορίσματα - για την υλοποίηση των εσωτερικών επαναλήψεων.

```
for (t = 1; t < T && !converged; t++) {
      for (i = 1; i < N; i++) {
            for (j = 1; j < N; j++) {
                  A[t][i][j] = 0.25*(A[t-1][i-1][j]+A[t-1][i+1][j]+A[
                      t-1][i][j-1]+A[t-1][i][j+1]);
            }
      }
      converged = check_convergence();
}
```

	Τύπος / Χαρακτηριστικά
Επεξεργαστές (CPUs)	2 x Intel Xeon X5650 @ 2.67GHz
Μητρική Πλακέτα (M/B)	Supermicro X8DTG-D (Intel 5520)
Κεντρική Μνήμη (RAM)	12 x 4GB DIMMs @ 1333Mhz
Κάρτα Γραφικών (GPU)	NVIDIA Tesla M2050 (Mem: 2687MB)

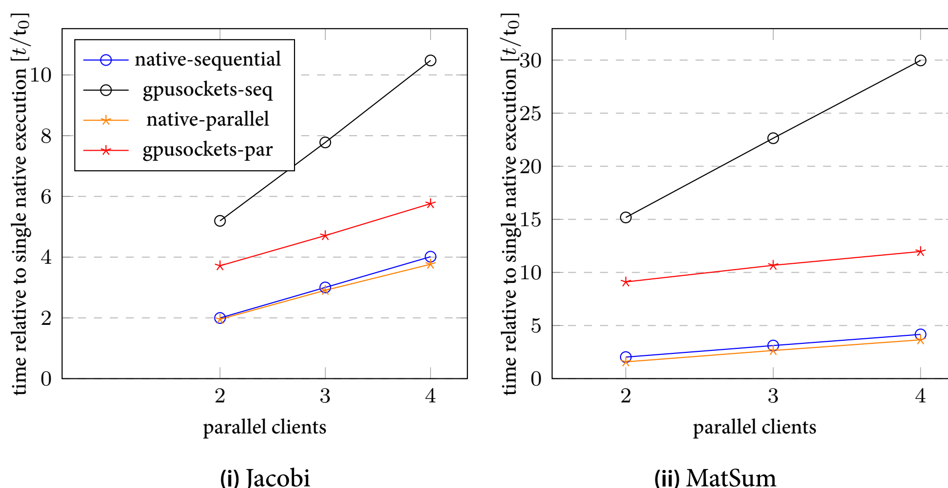
(i) Τεχνικές προδιαγραφές υλικού

	Έκδοση
Linux Kernel	3.15.6 SMP PREEMPT x86_64
Protocol Buffers	2.6.1
protobuf-c	1.1.1
Xen	4.5-unstable (cset: git:c39e1ff-dirty)
NVIDIA Driver	340.29
NVIDIA CUDA Toolkit	7.0

(ii) Εκδόσεις βασικότερου λογισμικού

Πίνακας 4.1: Χαρακτηριστικά του υλικού και του λογισμικού που χρησιμοποιήθηκε για τα πειράματα

Οι προδιαγραφές του μηχανήματος στο οποίο εκτελέστηκαν τα πειράματα ακολουθούν παρακάτω στον πιν. 4.1i, ενώ οι εκδόσεις του βασικότερου λογισμικού που χρησιμοποιήθηκε βρίσκονται στον πιν. 4.1ii. Για να πάρουμε με-



Σχήμα 4.1: Μέσος χρόνος εκτέλεσης MatSum και Jacobi stencil από 2,3 και 4 clients σειριακά και παράλληλα

τρήσεις χρησιμοποιήσαμε 5 Xen VMs(1-5). Στο VM1 έγινε απευθείας ανάθεση της GPU (passthrough) και χρησιμοποιήθηκε για τις απευθείας μετρήσεις και για την εκτέλεση του backend, ενώ στα υπόλοιπα - χωρίς GPU - VMs (2-5) εκτελέστηκαν οι εφαρμογές και το αντίστοιχο frontend. Χρησιμοποιήσαμε χρονόμετρα (timers) στον κώδικα για να πάρουμε ακριβείς χρόνους εκτέλεσης στις δύο εφαρμογές, αλλά και αναλυτικούς χρόνους για τις διάφορες ενέργειες στο backend. Λόγω της απλότητας του MatSum, έχουμε φροντίσει στον κώδικά του να εκτελούνται σειριακά οι εντολές που αξιοποιήσαμε στις μετρήσεις, ώστε να πάρουμε αξιόπιστα αποτελέσματα. Αρχικά, εκτελέσαμε το MatSum και το Jacobi stencil απευθείας στη GPU (VM1) και σε ένα VM χωρίς GPU. Τα αποτελέσματα φαίνονται στον πιν. 4.2.

Για να μελετήσουμε πώς μοιράζεται το υπολογιστικό φορτίο εφαρμογών από πολλαπλούς clients ταυτόχρονα, εκτελέσαμε τις δύο εφαρμογές στα VM2-5. Θέλοντας να αξιολογήσουμε την απόδοση της χρήσης παράλληλων νημάτων στο backend, πραγματοποιήσαμε το παραπάνω πείραμα εκτελώντας τις εφαρμογές αρχικά σειριακά και στη συνέχεια παράλληλα. Η σύγκριση φαίνεται στα

	MatSum	**Jacobi**
native	0.000852	1.532332
gpusockets	0.005739	3.901227

Πίνακας 4.2: Χρόνος εκτέλεσης MatSum και Jacobi stencil απευθείας στη GPU και από VM χωρίς GPU *(sec)*

γραφ. 4.1 για την παράλληλη χρήση σε 2, 3 και 4 clients διαδοχικά.

Ένα πρώτο συμπέρασμα που μπορούμε να βγάλουμε από τα παραπάνω γραφήματα, είναι ότι και στις 2 περιπτώσεις η παράλληλη (ασύγχρονη) εξυπηρέτηση των αιτημάτων στο backend, αυξάνει σημαντικά τις επιδόσεις όταν εκτελούνται ταυτόχρονα εφαρμογές από διαφορετικούς clients. Αυτό που πρέπει επίσης να παρατηρήσουμε εδώ είναι ότι η τελική υλοποίηση του framework μας προσφέρει κλιμακωσιμότητα, αφού οι επιδόσεις για πολλαπλά VMs είναι ανάλογες με αυτές της απευθείας εκτέλεσης. Ταυτόχρονα, η δυνατότητα παράλληλης εξυπηρέτησης πολλαπλών clients προσφέρει ευελιξία και αποτελεσματικότερο και πιο "δίκαιο" (όσον αφορά το χρόνο που απασχολεί ο κάθε client) μοίρασμα των διαθέσιμων υπολογιστικών πόρων της GPU.

λήψη	**αποκωδικοποίηση**	**πραγματική επεξεργασία**	**αποστολή**	**κωδικοποίηση**	**συνολικά εξυπηρέτησης**
537.778	20.317	443.787	71.185	23.598	1096.665

Πίνακας 4.3: Ενδεικτικός χρόνος εκτέλεσης σταδίων εξυπηρέτησης ενός μηνύματος *(usec)*

Στη συνέχεια, για να αναλύσουμε περαιτέρω τις επιδόσεις και ιδιαίτερα την επιβάρυνση που εμφανίζει η λύση μας, εξετάσαμε τους χρόνους που είχαν καταγράψει τα χρονόμετρα στο backend κατά τη διάρκεια του παραπάνω πειράματος. Ενδεικτικά παραθέτουμε στον πιν. 4.3 την κατανομή - κατά μέσο όρο - του χρόνου στις διάφορες εργασίες του backend κατά την εξυπηρέτηση κάθε αιτήματος μιας εκτέλεσης του Jacobi stencil (αντίστοιχα νούμερα παρατηρούνται και στις υπόλοιπες περιπτώσεις και των δύο εφαρμογών). Να σημειώσουμε ότι προφανώς οι χρόνοι στο frontend των clients θα μπορούσαν επίσης να χρησιμοποιηθούν και θα εμφανίζονταν αντίστοιχοι, αλλά επιλέξαμε να πάρουμε όλες τις αναλυτικές μετρήσεις στο backend για λόγους ευκολίας.

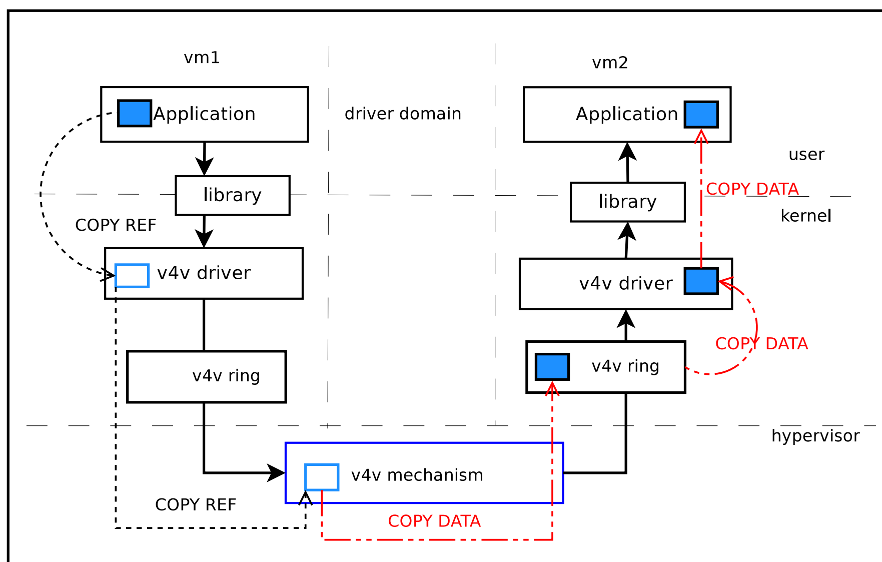
Όπως φαίνεται στον πίνακα, ο επιπλέον της πραγματικής επεξεργασίας/εκτέλεσης στη GPU χρόνος ξοδεύεται κατά ένα σημαντικό ποσοστό (και μάλιστα στη συγκεκριμένη περίπτωση και μεγαλύτερο αυτού της επεξεργασίας) στην ίδια τη λήψη/αποστολή δεδομένων μεταξύ client και host. Η απόδοση της προσέγγισής μας επομένως εξαρτάται σε μεγάλο βαθμό από τις επιδόσεις του μηχανισμού επικοινωνίας των δικτυακών συσκευών του client/host - και εν προκειμένω από τις επιδόσεις της επικοινωνίας μέσω TCP sockets.

Τέλος, παρατηρώντας ότι οι χρόνοι κωδικοποίησης/αποκωδικοποίησης των μηνυμάτων είναι αναλογικά μικροί (αν και επιδέχονται πιθανώς κάποιας βελτίωσης) επιλέξαμε να εξετάσουμε περαιτέρω τη σχέση των παραπάνω αποτε-

λεσμάτων με τις επιδόσεις του συμβατικού μηχανισμού επικοινωνίας του Xen για τις δικτυακές συσκευές (που βασίζεται στη συλλογή πρωτοκόλλων επικοινωνίας TCP/IP).

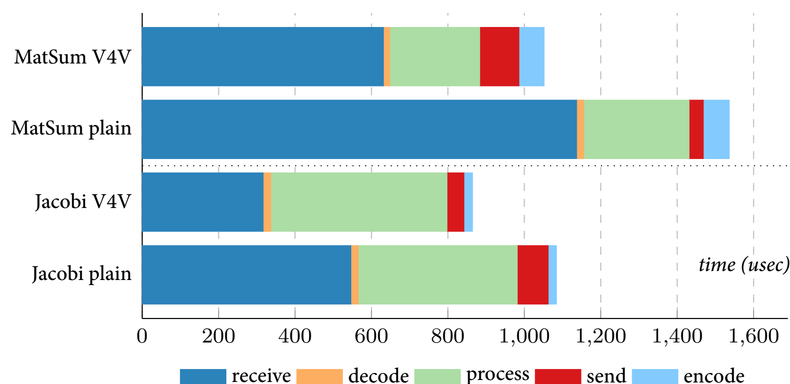
4.2 Εναλλακτικοί μηχανισμοί ενδοεπικοινωνίας εικονικών μηχανών: V4VSockets/V4V

Για να αποτιμήσουμε την απόδοση του προαναφερθέντος μηχανισμού επικοινωνίας, χρησιμοποιήσαμε τη βιβλιοθήκη V4VSockets που βασίζεται στον εναλλακτικό μηχανισμό ενδοεπικοινωνίας εικονικών μηχανών V4V. Το V4V είναι μια υπό εξέλιξη λύση που δεν έχει ενσωματωθεί ακόμα στο Xen, παρόλα αυτά επιχειρεί να υλοποιήσει ένα πιο αποδοτικό μοντέλο επικοινωνίας μεταξύ των VMs του Xen. Ακολουθώντας την σχεδιαστική αρχή διάκρισης μηχανισμού και πολιτικής το V4V αποτελείται από δύο μέρη. Το πρώτο μέρος είναι υλοποιημένο μέσα στον hypervisor και παρέχει έναν μηχανισμό για τη μεταφορά δεδομένων από το ένα VM στο άλλο. Το δεύτερο μέρος (πολιτική) αποτελεί ένα module στον πυρήνα των εικονικών μηχανών και υλοποιεί τη συλλογή πρωτοκόλλων επικοινωνίας TCP/IP.



Σχήμα 4.2: Ενδο-επικοινωνία εικονικών μηχανών με V4Vsockets

Το V4VSockets [12] είναι μια βιβλιοθήκη που στοχεύει στην αποδοτικότερη χρήση των TCP sockets, “μεταφράζοντας” τις κλήσεις που τα αφορούν σε αι-



Σχήμα 4.3: Κατανομή χρόνου εξυπηρέτησης αιτημάτων MatSum και Jacobi stencil με το συμβατικό μηχανισμό Xen και με V4V/V4VSockets

τήματα/απαντήσεις προς/από το V4V [13]. Αξιοποιεί έτσι τις αυξημένες επιδόσεις που προσφέρει το τελευταίο προς όφελος των εφαρμογών που βασίζονται σε TCP sockets, οι οποίες μπορούν - χωρίς τροποποιήσεις από τη μεριά του χρήστη - να χρησιμοποιήσουν τον μηχανισμό του V4V. Οι βασικές υπερ-κλήσεις (hypercalls) [14] του τελευταίου που χρησιμοποιούνται για την επικοινωνία είναι οι εξής:

- register/unregister: Χρησιμοποιείται κατά τη δημιουργία ενός socket για να παρέχει τον απαραίτητο χώρο μνήμης για τη μεταφορά δεδομένων.
- send: Διαθέτει τις σχετικές δομές στον hypervisor κατά την εκπομπή μιας κλήσης αποστολής (send call), ώστε εκείνος με την σειρά του να εκτελέσει τη μεταφορά.
- notify: Με τη σωστή τοποθέτηση των δεδομένων ή όταν υπάρχει κάποια ειδοποίηση που χρειάζεται προσοχή, ο πυρήνας (kernel) του VM εκπέμπει αυτή την κλήση και ο hypervisor στη συνέχεια προχωράει στα απαραίτητα βήματα ώστε να ολοκληρωθεί η αντίστοιχη ενέργεια (πχ. λήψη κλήσεων).

Το V4VSockets βασίζεται στη δομή v4v_ring, η οποία αποτελείται από ένα προ-εκχωρημένο (pre-allocated) μοιραζόμενο χώρο μνήμης σε δομή δακτυλίου (ring buffer) που εξομοιώνει ουσιαστικά το δικτυακό μέσο. Ο χώρος αυτό ακολουθεί το γενικό μοντέλο παραγωγού-καταναλωτή με δύο δείκτες, έναν για χρήση από το VM και έναν για χρήση από τον hypervisor. Η δομή αυτή εκχωρείται στον πυρήνα του VM και καταχωρείται (register) στον hypervisor με την κλήση συστήματος bind() - που μεταφράζεται σε μια υπερ-κλήση register δημιουργώντας έναν μοιραζόμενο χώρο μνήμης μεταξύ hypervisor-VM. Για να μεταφερθούν δεδομένα από ένα VM σε ένα άλλο, ακολουθώντας την

κλήση συστήματος sendmsg(), ο πυρήνας του VM-αποστολέα δημιουργεί ένα iovector (διάνυσμα ε/ε) από τα δεδομένα στο χώρο χρήστη. Αφού τα πακετάρει σε ένα μήνυμα V4V, εκπέμπει μια υπερ-κλήση send. Στη συνέχεια, ο hypervisor αντιγράφει τα δεδομένα στο ring του VM-παραλήπτη, στο οποίο VM έχει ήδη αρχικοποιηθεί μια διαδικασία λήψης χρησιμοποιώντας την κλήση συστήματος accept() και περιμένει εισερχόμενα πακέτα σε συγκεκριμένη πόρτα. Τέλος, ακολουθώντας μια κλήση συστήματος recvmsg(), ο πυρήνας του VM-παραλήπτη αντιγράφει τα δεδομένα από το ring σε έναν τοπικό buffer.

Χρησιμοποιώντας, λοιπόν, το V4V στους clients και στον host και τη βιβλιοθήκη V4VSockets με το MatSum και το Jacobi stencil, συγκρίναμε τη διαφορά στην κατανομή του χρόνου εξυπηρέτησης των αιτημάτων σε σχέση με τον συμβατικό μηχανισμό επικοινωνίας του Xen για δικτυακές συσκευές. Όπως παρατηρούμε στο γραφ. 4.3, το ποσοστό του χρόνου λήψης/αποστολής στην περίπτωση του V4V μειώνεται σημαντικά. Μπορούμε επομένως να συμπεράνουμε ότι, ειδικά για την περίπτωση του Xen, η επιβάρυνση που προσθέτει η υλοποίησή μας μπορεί να παρουσιάσει αισθητή μείωση αξιοποιώντας έναν αποδοτικότερο μηχανισμό ενδοεπικοινωνίας των VMs.

5

Σχετικές Υλοποιήσεις

Όπως η δική μας εφαρμογή, υπάρχουν μια σειρά από προσεγγίσεις που ασχολούνται με το ζήτημα του virtualization - ή/και του remote execution - σε συσκευές CUDA. Οι περισσότερες από αυτές περιλαμβάνουν ένα κατανεμημένο middleware με μία εφαρμογή στον client (frontend) και μία στον host (backend) που προσφέρει τις GPGPU υπηρεσίες, που παρεμβάλλονται όμως σε διαφορετικά επίπεδα της CUDA και με διαφορετικό ενδεχομένως τρόπο, ενώ υλοποιούν διαφορετικά και την μεταξύ τους επικοινωνία. Παρόλο που ακολουθούν κάποιες κοινές (άρα) αρχές λειτουργίας, στοχεύουν στο να λύσουν αποδοτικά διαφορετικά - αν και όχι απαραίτητα ανεξάρτητα - προβλήματα (πχ. διαμοιρασμός gpu σε εικονικά περιβάλλοντα, απομακρυσμένη εκτέλεση σε gpu μη εικονικού συστήματος σε δίκτυα με χαμηλές επιδόσεις κλπ.) και επομένως έχουν διαφορετικούς περιορισμούς. Παρακάτω παρουσιάζουμε τις γνωστότερες από αυτές:

5.1 vCUDA [15]

Αποτελείται από τρία modules (μονάδες) στο user space (χώρος χρήστη): μια βιβλιοθήκη η οποία περιλαμβάνει μια δομή δεδομένων που αναπαριστά μια εικονική gpu στον client και μια εφαρμογή στον server. Η βιβλιοθήκη στον client λαμβάνει τις κλήσεις προς το runtime API της CUDA και προωθεί τα αιτήματα στην εφαρμογή του server που τα εκτελεί και επιστρέφει τα αποτελέσματα. Για την επικοινωνία μεταξύ των εφαρμογών χρησιμοποιείται το πρωτόκολλο XML-RPC, που όμως επιβαρύνει αρκετά τις συνολικές επιδόσεις αυτής της λύσης, αφού αυξάνει αισθητά το χρόνο που ξοδεύεται στα στάδια κωδικοποίησης και αποκωδικοποίησης (βλ. και 2.4 παραπάνω), όπως φαίνεται από τις μετρήσεις. Το framework αυτό είναι γραμμένο σε C και υλοποιεί ένα αδιευκρίνιστο υποσύνολο της (παρωχημένης) έκδοσης 1.1 του CUDA runtime API.

5.2 rCUDA [16]

Αποτελείται πάλι από δύο βασικές εφαρμογές, μια wrapper (“περιτύλιγμα”) βιβλιοθήκη στον client και μια υπηρεσία (service) στον server. Και εδώ, σε γενικές γραμμές, η βιβλιοθήκη στον client λαμβάνει τις κλήσεις στο CUDA runtime API και προωθεί τα αντίστοιχα αιτήματα στην υπηρεσία του server περιμένοντας το αποτέλεσμα. Η επικοινωνία μεταξύ των δύο εφαρμογών γίνεται με τη χρήση ενός προσαρμοσμένου πρωτοκόλλου βασισμένου σε TCP sockets, ενώ έχει προστεθεί και ένα module που χρησιμοποιεί InfiniBand Verbs για να εκμεταλλευτεί τις καλύτερες επιδόσεις αυτού του χαμηλού επιπέδου API σε συστήματα που χρησιμοποιούν IB [17, 18]. Το framework αυτό αναπτύσσεται ενεργά και προσφέρει βελτιώσεις στην επίδοση των επικοινωνιών – πχ. χρησιμοποιεί pipelined επικοινωνίες Μέχρι τη στιγμή της συγγραφής αυτής της διπλωματικής, το rCUDA υποστηρίζει το πιο πρόσφατο 6.5 CUDA Runtime API [19]. Παρόλο που αυτή είναι η πιο “ώριμη” από τις λύσεις που παραθέτουμε, είναι κλειστό λογισμικό (προσφέρονται οι εφαρμογές μόνο σε μεταγλωττισμένη μορφή).

5.3 gVirtuS [20]

Ακολουθεί ένα μοντέλο frontend/backend όπως και τα προηγούμενα 2 frameworks. Εδώ ο στόχος είναι κυρίως εικονικά περιβάλλοντα, με έμφαση στα VMware και KVM, οπότε η επικοινωνία γίνεται με ένα πρωτόκολλο βασισμένο σε unix sockets σε εικονικές μηχανές χρησιμοποιώντας μια συσκευή QEMU συνδεδεμένη στον εικονικό δίαυλο PCI (virtual PCI bus) - υπάρχει και μια βασική υλοποίηση που χρησιμοποιεί tcp/ip, για δοκιμαστικούς λόγους. Το gVirtus είναι γραμμένο σε C++ και υλοποιεί ένα υποσύνολο της (επίσης παρωχημένης) έκδοσης 3.2 του CUDA runtime API.

5.4 GviM [21]

Είναι σχεδιασμένο με αντίστοιχη λογική, αλλά εστιάζει συγκεκριμένα σε εικονικές μηχανές του Xen. Αξιοποιεί μηχανισμούς του Xen για την επικοινωνία μεταξύ των εφαρμογών, όπως shared-memory (μοιραζόμενης μνήμης) buffers. Η λύση αυτή υποστηρίζει ένα υποσύνολο του CUDA runtime API 1.1.

5.5 DS-CUDA [22]

Υλοποιεί παρόμοιο μοντέλο με τα παραπάνω, μόνο που εδώ στόχος είναι κυρίως η δυνατότητα χρήσης πολλαπλών απομακρυσμένων (εικονικών) GPUs από ένα σύστημα (και όχι η χρήση όσο το δυνατόν λιγότερων GPUs σε ένα cluster όπως στα προηγούμενα). Εδώ η επικοινωνία μεταξύ frontend και backend γίνεται με τη χρήση του χαμηλού επιπέδου InfiniBand API, αλλά δευτερευόντως υποστηρίζεται και επικοινωνία βασισμένη σε tcp sockets. Ενδιαφέρον χαρακτηριστικό αυτής της λύσης είναι ότι ενσωματώνει ένα μηχανισμό ανεκτικό στα σφάλματα, που μπορεί να εκτελέσει δευτερεύοντες υπολογισμούς χρησιμοποιώντας πολλαπλές GPUs. Και αυτό το framework δεν υλοποιεί πλήρως το CUDA runtime API (δεν υποστηρίζει πχ. ασύγχρονες λειτουργίες), εδώ της έκδοσης 4.1.

Να σημειώσουμε επίσης ότι υπάρχουν κι άλλες λύσεις, ιδιαίτερα για OpenCL, όπως τα SnuCL [23], VCL [24], VOCL [25]. Τέλος να αναφέρουμε και τη δυνατότητα ανάθεσης μιας GPU - αποκλειστικά - σε ένα εικονικό μηχάνημα που προσφέρουν γνωστά περιβάλλοντα virtualization όπως το Xen [26] και το KVM/QEMU [27, 28] VGA pass-through.

6

Σύνοψη

Το cloud computing, είτε αυτό αφορά σε κατανεμημένα - γεωγραφικά - περιβάλλοντα (grid computing) είτε αφορά υπολογιστικά κέντρα με συστοιχίες (clusters) υπολογιστών, βρίσκεται αναπόφευκτα στο επίκεντρο του ενδιαφέροντος στις μέρες μας. Για οποιαδήποτε εφαρμογή απαιτεί αυξημένη υπολογιστική ισχύ, αλλά και ιδιαίτερα για επιστημονικές εφαρμογές που βασίζονται σε μαθηματικά μοντέλα, η αποτελεσματικότητα των παραπάνω τεχνολογιών αποκτά ιδιαίτερη σημασία (HPC). Η αποτελεσματική παραλληλοποίηση επαναλαμβανόμενων διαδικασιών που προσφέρει η χρήση GPUs - αντί για CPUs - (GPGPU) σε υπολογιστικά clusters ή/και grid για την επιτάχυνση υπολογιστικά απαιτητικών εργασιών, που αφορούν τομείς από τα οικονομικά και την υγεία μέχρι τα μαθηματικά και τη φυσική, την καθιστούν όχι μόνο σημαντική εξέλιξη αλλά και μια όλο και ευρύτερα υιοθετούμενη λύση. Μάλιστα, ο συνδυασμός των επιδόσεων του GPGPU με τα πλεονεκτήματα που προσφέρουν τα εικονικά περιβάλλοντα (κλιμακωσιμότητα, ελαστικότητα, ασφάλεια, live migration) μπορεί να παρέχει όχι μόνο υψηλές επιδόσεις αλλά και ευελιξία. Ταυτόχρονα όμως, εισάγει και προβλήματα που έχουν εμποδίσει μέχρι στιγμής τέτοιου τύπου λύσεις να χρησιμοποιηθούν καθολικά, παρόλο που λαμβάνουν έντονης ερευνητικής προσοχής. Τα σημαντικότερα είναι η επιβάρυνση στις επιδόσεις που προσθέτει το virtualization αλλά και η αποδοτικότητα που μπορούν να προσφέρει ένας συνδυασμός σαν τον παραπάνω: δεν αρκεί δηλαδή μια τέτοια λύση να προσφέρει μόνο υψηλές επιδόσεις, αλλά πρέπει να είναι και οικονομική (πχ. δυνατότητα χρήσης GPU σε κάποια μόνο μηχανήματα μιας συστοιχίας υπολογιστών).

Με αφορμή τα παραπάνω ασχοληθήκαμε με τη δομή και τρόπους για την αποδοτική χρήση τέτοιων συνδυαστικών συστημάτων που αξιοποιούν GPGPU και virtualization.

Πριν παρουσιάσουμε τη δική μας προσέγγιση στα παραπάνω προβλήματα, μελετήσαμε τις τεχνολογίες στις οποίες βασίζεται η συγκεκριμένη εργασία. Περιγράψαμε την έννοια του GPGPU (χρήση GPU για υπολογισμούς "γενικού σκοπού") και τα διαθέσιμα, δημοφιλή και ευρέως χρησιμοποιούμενα frameworks για GPGPU (το ανοιχτό OpenCL του Khronos Group και την κλειστή-proprietary CUDA της NVIDIA). Εξηγήσαμε τους λόγους που επιλέξαμε την

CUDA και παρουσιάσαμε τις βασικές αρχές λειτουργίας της και τις διεπαφές (runtime και driver API) που προσφέρει. Επίσης, ασχοληθήκαμε με την έννοια του virtualization και τα πλεονεκτήματα και τα μειονεκτήματα που έχει ένας τέτοιος μηχανισμός αφαίρεσης. Δώσαμε τις κατηγορίες που μπορούμε να διακρίνουμε (τύπου I/II - paravirtualization/full virtualization) και τα χαρακτηριστικά τους, και περιγράψαμε τη δομή και τους βασικούς μηχανισμούς ενδοεπικοινωνίας του Xen, μιας από τις δημοφιλέστερες πλατφόρμες για εικονικά περιβάλλοντα, που χρησιμοποιήσαμε για τους σκοπούς αυτής της εργασίας. Αναφερθήκαμε στη φιλοσοφία του μηχανισμού των sockets σε ένα δίκτυο υπολογιστών και αναλύσαμε τον τρόπο λειτουργίας των TCP sockets, στα οποία βασίστηκε το πρωτόκολλο επικοινωνίας της λύσης που στη συνέχεια προτείναμε. Τέλος, παρουσιάσαμε την τεχνική του data serialization και τη χρήση της, εξηγήσαμε τους λόγους που μπορεί να βελτιώσει (αν χρησιμοποιηθεί σωστά) το προαναφερθέν πρωτόκολλο επικοινωνίας και παραθέσαμε το τρόπο λειτουργίας του Protocol Buffers, της βιβλιοθήκης της Google που αξιοποιήσαμε και υλοποιεί την παραπάνω τεχνική.

Στη συνέχεια, περιγράψαμε το σχεδιασμό και την υλοποίηση της δικής μας προσέγγισης ενός framework που επιχειρεί να δώσει μια αποδοτική λύση: α) για την εκτέλεση υπολογιστικών φορτίων σε απομακρυσμένες GPUs, και β) ιδιαίτερα για το διαμοιρασμό GPUs σε εικονικά περιβάλλοντα βασισμένα σε συστοιχίες υπολογιστών για χρήση με τέτοια φορτία. Σκοπός ήταν κυρίως η αποδοτική αξιοποίηση συστοιχιών υπολογιστών χωρίς τη χρήση καρτών γραφικών σε όλους τους κόμβους/υπολογιστές. Συγκεκριμένα, εστιάσαμε στη δυνατότητα υπολογιστών χωρίς GPU να εκτελούν αποτελεσματικά εφαρμογές CUDA σε απομακρυσμένους (εικονικούς και μη) υπολογιστές με GPU, δηλαδή στην ταυτόχρονη (αποδοτική) χρήση μιας host gpu από πολλαπλούς (απομακρυσμένους) clients. Στόχος της υλοποίησής μας ήταν να είναι "διάφανη"/χωρίς απαιτούμενες μετατροπές από τη μεριά του χρήστη και να αξιοποιεί το χαμηλότερου επιπέδου CUDA driver API, ώστε να είναι εύχρηστη και επεκτάσιμη. Για του λόγους αυτούς, η προσέγγιση που χρησιμοποιήσαμε βασίστηκε στο διαδεδομένο μοντέλο διαχωρισμένου οδηγού συσκευών, και αποτελείται από δύο μέρη, τη λειτουργία των οποίων περιγράψαμε αναλυτικά: την υπηρεσία εξυπηρέτησης στον host με GPU (backend) και τη βιβλιοθήκη-μεσολαβητή στους clients χωρίς GPU (frontend). Το πρωτόκολλο επικοινωνίας που χρησιμοποιείται για την ανταλλαγή δεδομένων μεταξύ των δύο αυτών μερών είναι βασισμένο στη χρήση TCP sockets και data serialization, είναι όμως φτιαγμένο με την απαραίτητη αφαίρεση ώστε να είναι μελλοντικά επεκτάσιμο και με άλλους μηχανισμούς.

Ακολούθως, παραθέσαμε συνοπτικά μια σειρά από υλοποιήσεις, βασισμένες

στο frontend/backend μοντέλο που αξιοποιήσαμε και εμείς. Δώσαμε τα βασικά τους χαρακτηριστικά και τους τομείς στους οποίους έχουν εστιάσει επιχειρώντας να δώσουν λύσεις - μερικά ή συνολικά - στα προβλήματα που περιγράψαμε ξεκινώντας αυτό το κεφάλαιο.

Τέλος, για την αποτίμηση του framework που παρουσιάσαμε κάναμε μια σειρά από πειράματα για να αναλύσουμε τη συμπεριφορά και τις επιδόσεις του, χρησιμοποιώντας μια CUDA GPU σε Xen περιβάλλον. Δείξαμε ότι η προσέγγισή μας, αξιοποιώντας τη χρήση παράλληλων νημάτων στο backend, προσφέρει: α) κλιμακωσιμότητα (scalability) και β) ευελιξία όσον αφορά την αξιοποίηση και το διαμοιρασμό μιας "απομακρυσμένης" GPU (host) από πολλαπλά VMs (clients). Όπως φάνηκε από την ανάλυση της κατανομής του χρόνου εξυπηρέτησης των αιτημάτων, διαπιστώσαμε ότι η επιβάρυνση μιας τέτοιας υλοποίησης στον χρόνο εκτέλεσης εξαρτάται σημαντικά από τις επιδόσεις του δικτυακού μηχανισμού επικοινωνίας που αφορά την ίδια την αποστολή/λήψη δεδομένων μεταξύ host και clients. Στη δική μας περίπτωση, η χρήση, μέσω του V4VSockets, του V4V ως εναλλακτικού μηχανισμού ενδοεπικοινωνίας μεταξύ των VMs στο Xen οδήγησε σε αξιόλογη μείωση του επιπλέον χρονικού κόστους της "απομακρυσμένης" εκτέλεσης.

Συνεπώς, η υλοποίησή μας ανταποκρίνεται στα δεδομένα του προβλήματος και μπορεί να γίνει αποδοτικότερη είτε α) συνδυάζοντάς τη με τη χρήση βελτιωμένων μηχανισμών ενδοεπικοινωνίας host-clients (όπως τα V4VSockets/ V4V στο Xen) είτε β) με την περαιτέρω επέκταση εγγενών δυνατοτήτων της. Ενδεικτικά, κομμάτια προς διερεύνηση με περιθώρια βελτίωσης - αξιοποιώντας τον αφαιρετικό τρόπο που είναι δομημένη η λύση μας - υπάρχουν: α) στην επέκταση της βάσης του προσαρμοσμένου πρωτοκόλλου επικοινωνίας host-clients για την υποστήριξη αποδοτικότερων ή/και πιο ειδικών μηχανισμών από τα TCP sockets (όπως InfiniBand verbs), αλλά και β) με την εν γενει εισαγωγή πιο αποδοτικού pipeline (σωλήνωσης) για την ταχύτερη εκτέλεση ανεξάρτητων CUDA εντολών (τεχνική της οποίας επωφελείται και το ίδιο το CUDA toolkit), ειδικά όσον αφορά τις εντολές μνήμης.

Βιβλιογραφία

[1] Luebke-D. Govindaraju-N. Harris M. Kruger J. Lefohn-A.E. Purcell-T.J. Owens, J.D. A survey of general-purpose computation on graphics hardware. In *Computer Graphics Forum*, volume 26, pages 80–113, 2007.

[2] Haldar-J.P. Tsao-S.C. Hwu W.-m.W. Liang Z.-P. Sutton-B.P. Stone, S.S. Accelerating advanced mri reconstructions on gpus. In *CF '08: Proceedings of the 2008 conference on Computing frontiers*, pages 261–272. ACM, 2008.

[3] URL https://www.khronos.org/registry/cl/sdk/2.0/docs/man/xhtml/.

[4] URL http://www.nvidia.com/object/cuda_home_new.html.

[5] *CUDA C Programming Guide (PG-02829-001_v6.5)*.

[6] Goldberg-R.P. Popek, G.J. Formal requirements for virtualizable third generation architectures. 17(7), July 1974.

[7] . URL http://wiki.xenproject.org/wiki/Xen_Overview.

[8] URL http://pubs.opengroup.org/onlinepubs/9699919799/.

[9] . URL https://developers.google.com/protocol-buffers.

[10] . URL https://developers.google.com/protocol-buffers/docs/overview.

[11] URL https://github.com/protobuf-c/protobuf-c/wiki/Examples.

[12] Gerangelos-S. Alifieraki-I. Koziris N. Nanos, A. V4vsockets: low-overhead intra-node communication in xen. In *Proceedings of the 5th International Workshop on Cloud Data and Platforms*. ACM, 2015.

[13] V4v implementation, . URL http://lists.xen.org/archives/html/xen-devel/2013-05/msg02711.html.

[14] . URL http://wiki.xenproject.org/wiki/Hypercall.

[15] Chen-H. Sun-J. Li K. Shi, L. vcuda: Gpu accelerated high performance computing - in virtual machines. 61, 2012.

[16] Pena-A.J. Silla-F. Fernandez J.C.-Mayo R. Quintana-Ortí E.S. Duato, J. Enabling cuda acceleration within virtual machines using rcuda. In *Proceedings of the 2011 International Conference on High Performance Computing (HiPC)*, 2011.

[17] Mayo-R. Quintana-Ortí E.S. Silla-F. Duato J.-Peña A.J. Reaño, C. Influence of infiniband fdr on the performance of remote gpu virtualization. In *Proceedings of the IEEE Cluster 2013 Conference, Indianapolis, IN (USA)*, 2013.

[18] Silla-F. Peña-A.J. Shainer G.-Schultz S. Castelló-A. Quintana-Ortí E.S. Duato J. Reaño, C. Poster: Boosting the performance of remote gpu virtualization using infiniband connect-ib and pcie 3.0. In *Proceedings of the IEEE Cluster 2014 Conference, Madrid, Spain*, 2014.

[19] URL http://www.rcuda.net/index.php/latest-release.html.

[20] Montella-R. Agrillo-G. Coviello G. Giunta, G. A gpgpu transparent virtualization component for high performance computing clouds. In *Euro-Par 2010 - Parallel Processing*, volume 6271 of LNCS. Springer Berlin/Heidelberg, 2010.

[21] Gavrilovska-A. Schwan-K. Kharche H.-Tolia N. Talwar-V. Ranganathan-P. Gupta, V. Gvim: Gpu-accelerated virtual machines. In *3rd Workshop on System-level Virtualization for High Performance Computing, NY, USA*. ACM, 2009.

[22] Kawai-A. Nomura-K. Yasuoka K. Yoshikawa K. Narumi-T. Oikawa, M. Ds-cuda: A middleware to use many gpus in the cloud environment. In *High Performance Computing, Networking, Storage and Analysis (SCC)*, 2012.

[23] Seo-S. Lee-J. Nah J. Jo G. Lee-J. Kim, J. Snucl: An opencl framework for heterogeneous cpu/gpu clusters. In *Proceedings of the 26th ACM International Conference on Supercomputing*. ACM, 2012.

[24] Ben-Nun T.-Levy E. Shiloh A. Barak, A. A package for opencl based heterogeneous computing on clusters with many gpu devices. In *Workshop on Parallel Programming and Applications on Accelerator Clusters (PPAAC)*, 2010.

[25] Balaji-P. Zhu-Q. Thakur R. Coghlan S. Lin-H. Wen-G. Hong J. Feng W. Xiao, S. Vocl: An optimized environment for transparent virtualization of graphics processing units. In *Proceedings of the 1st Innovative Parallel Computing (InPar)*, 2012.

[26] . URL http://wiki.xenproject.org/wiki/XenVGAPassthrough.

[27] URL https://www.kernel.org/doc/Documentation/vfio.txt.

[28] URL http://events.linuxfoundation.org/sites/events/files/slides/KVM%20Forum%202014%20-%20VFIO,%20OVMF,%20GPU,%20and%20You%20-%20Alex%20Williamson.pdf.