ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ
ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Κατανεμημένο Σύστημα Διαχείρισης Εργασιών Απομακρυσμένης Εκτέλεσης Κώδικα Για Επιταχυντές Γραφικών Σε Συστοιχίες Υπολογιστών

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΚΑΡΚΑΤΣΟΥΛΗΣ Π. ΑΝΤΩΝΙΟΣ

Επιβλέπων : Κοζύρης Νεκτάριος
Καθηγητής Ε.Μ.Π.

Αθήνα, Οκτώβριος 2013

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ
ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Κατανεμημένο Σύστημα Διαχείρισης Εργασιών Απομακρυσμένης Εκτέλεσης Κώδικα Για Επιταχυντές Γραφικών Σε Συστοιχίες Υπολογιστών

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΚΑΡΚΑΤΣΟΥΛΗΣ Π. ΑΝΤΩΝΙΟΣ

Επιβλέπων : Κοζύρης Νεκτάριος
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την ……………………….. 2014.

………………………. …………………………. ………………………….

....

ΑΝΤΩΝΙΟΣ Π. ΚΑΡΚΑΤΣΟΥΛΗΣ

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

Copyright © Αντώνιος Π. Καρκατσούλης, 2013
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 Μετσόβιου Πολυτεχνείου.

Στον Αντώνη

ΠΕΡΙΛΗΨΗ

Οι GPUs έχουν γίνει πλέον κυρίαρχες στον τομέα της υπολογιστικής τεχνολογίας. Η καταλληλότητα τους και η μεγάλη ικανότητα παραλληλοποίησης που προσφέρουν τις έχουν καταστήσει σημαντικότατους υπολογιστικούς πόρους.

Κύρια εφαρμογή τους είναι σε επιστημονικές εφαρμογές, που από μόνες τους έχουν τεράστιες ανάγκες για υπολογιστική ισχύ αλλά και μεγάλες δυνατότητες παραλληλοποίησης.

Για το σκοπό αυτό έχουν δημιουργηθεί πλατφόρμες που να υποστηρίζουν το λεγόμενο General Purpose Graphics Processing (GPGPU), που επιτρέπει πλέον στην GPU να εκτελεί υπολογισμούς σε εφαρμογές γενικότερου σκοπού, σε αντίθεση μέχρι τώρα που ειδικευόταν σε γραφικές εφαρμογές. Να σημειωθεί εδώ ότι σε καμία περίπτωση η GPU δεν αντικαθιστά την CPU. Αρχιτεκτονικοί λόγοι την περιορίζουν από το να χρησιμοποιηθεί ως η κύρια μονάδα επεξεργασίας ενός συστήματος. Αυτό που κάνει είναι απλώς να επιταχύνει συγκεκριμένες εφαρμογές της CPU.

Η ανάγκη για υπολογιστική ισχύ είναι πλέον τόσο μεγάλη, ώστε κατασκευάζονται ολόκληρες συστοιχίες (clusters) υπολογιστών βασισμένες σε GPUs (GPU Clusters). Προφανώς χρειάζεται ειδικό Hardware , αλλά και λογισμικό , για τη σωστή διαχείριση ενός τέτοιου cluster. Για το μεν Hardware υπάρχουν εταιρείες που κατασκευάζουν ισχυρούς επεξεργαστές γραφικών και εξοπλίζουν τα datacenters για το σκοπό αυτό. Για το δε software, απαιτούνται κατάλληλα συστήματα διαχείρισης ώστε να εκμεταλλεύονται όσο το δυνατόν περισσότερο την επεξεργαστική ισχύ των GPUs στις συστοιχίες υπολογιστών. Το πρόβλημα είναι ότι η διαθεσιμότητα GPUs στους κόμβους των συστοιχιών αυτών είναι σχεδόν πάντα περιορισμένη λόγω του κόστους.

Η παρούσα διπλωματική παρουσιάζει το σύστημα rGPU, ένα σύστημα διαχείρισης του cluster το οποίο αναλαμβάνει την διαφανή εκτέλεση προγραμμάτων επιταχυμένων από GPUs σε κόμβους οι οποίοι δεν τις διαθέτουν. Αρχικά γίνεται μια εκτενής ανασκόπηση της βιβλιογραφίας πάνω στον συγκεκριμένο τομέα, έπειτα παρουσιάζεται το προτεινόμενο σύστημα, περιγράφεται αναλυτικά η σχεδίαση και η υλοποίηση του και τέλος αξιολογείται με βάση συγκεκριμένα σενάρια χρήσης.

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ

Επεξεργαστής γραφικών (GPU), Υπολογισμοί γενικού σκοπού σε GPUs (GPGPU), παράλληλη επεξεργασία, σύστημα διαχείρισης, συστοιχία υπολογιστών, CUDA

ABSTRACT

GPUs have become a dominant feature in the field of computer technology. Their always growing scalability and computational power are beginning to classify them as first-class computing resources. Their main application is in the field of science, where applications need huge processing capabilities and offer a huge level of parallelization.

For this purpose, many platforms have been created to support the so-called General Purpose GPU Computing (GPGPU), that enables graphic processors to carry out general purpose calculations, as opposed to the past when they were only used for graphics rendering. However, in no way can the GPU fully replace the CPU. The GPU is a co-processor that accelerates specific parts of a program and architectural reasons prevent it from being used as the primary processing unit of a system.

This need for computational power is so great that GPUs are being installed in High Performance Clusters and Supercomputers. Of course, special hardware, software and technical expertise are required in order to fully utilize these co-processors. As far as hardware is concerned there are many companies that manufacture these powerful special processors. As for the software, apart from the programming tools needed to utilize the GPUs, complex cluster management systems are needed in order to co-ordinate the processing. The problem so far is that, because of the cost, the availability of GPUs in High Performance clusters is limited.

The purpose of this thesis is to introduce a new cluster management system, the purpose of which is to utilize the GPUs in the cluster as much as possible. For this purpose it uses an already developed and optimized system called rCUDA, that enables the remote execution of CUDA programs from nodes not equipped with a GPU. First, an extensive review of the state-of-the-art methods in parallel processing is provided and then the new system is being proposed along with detailed information about its architecture, its implementation and its evaluation process.

KEYWORDS

Graphics Processing Unit (GPU), General Purpose GPU (GPGPU), parallel processing, cluster management system, cluster computing, CUDA

Περιεχόμενα

Κεφάλαιο 1 13
Παράλληλα Συστήματα Επεξεργασίας........ 13
1.1 Ιστορικά........ 13
1.1.1 Στάδια της Υπολογιστικής Τεχνολογίας 14
1.1.2 Αρχιτεκτονικές Μνήμης 16
1.1.3 Κλιμακωσιμότητα (Scalability)........ 18
1.2 Κατηγοριοποίηση Αρχιτεκτονικών Υπολογιστών 19
1.2.1 Αρχιτεκτονικές Κοινής Μνήμης (Shared Memory) 21
1.2.2 Αρχιτεκτονικές Κατανεμημένης Μνήμης (Distributed Memory) 22
1.2.3 Υβριδικές Αρχιτεκτονικές (Hybrid Systems) 22
1.3 Επεξεργαστές Γραφικών (GPUs) 23
1.4 General Purpose GPU Computing (GPGPU) 27
1.4.1 Προγραμματισμός GPGPU 28
1.4.2 Περιορισμοί και προβλήματα στην χρήση των GPUs........ 30
1.5 Η τεχνολογία CUDA 31
1.5.1 Το μοντέλο Υπολογισμού........ 31
1.5.2 Μεταγλώττιση και Εκτέλεση........ 34
1.5.3 Η εναλλακτική της OpenCL........ 36
1.6 GPUs σε clusters και Supercomputers 37
1.6.1 Χαρακτηριστικά των GPU Clusters 37
1.6.2 Απαιτήσεις ισχύος 40
1.6.3 Χρονοδρομολόγηση εργασιών 40
Κεφάλαιο 2 43
rCUDA : Σύστημα απομακρυσμένης εκτέλεσης CUDA 43
2.1 Αρχιτεκτονική 43
2.2 Υλοποίηση – Τρόπος Εκτέλεσης........ 46
2.3 Αξιολόγηση rCUDA........ 48
2.3 Σύγκριση με παρόμοια συστήματα........ 50
2.4 Συμπεράσματα........ 51
Κεφάλαιο 3 53
rGPU : Το προτεινόμενο σύστημα διαχείρισης των απομακρυσμένων εκτελέσεων CUDA .. 53
3.1 Κατανεμημένα Συστήματα........ 53

3.2 rGPU : Σχεδιασμός – Αρχιτεκτονική ... 56
3.2.1 Πελάτης (client) ... 56
3.2.2 Επόπτης (Hypervisor) ... 57
3.2.2 Εξυπηρετητές GPU ... 58
3.3 Τυπικό σενάριο λειτουργίας ... 58
3.4 Αρχιτεκτονικές αποφάσεις – Μειονεκτήματα του συστήματος ... 59
3.5 Εναλλακτικές λύσεις για την διαχείριση των GPU εργασιών ... 61
Κεφάλαιο 4 ... 63
Υλοποίηση του rGPU ... 63
4.1 Πλατφόρμα Ανάπτυξης ... 63
4.2 Υλοποίηση του Επόπτη (Hypervisor) ... 63
4.3 Υλοποίηση του Πελάτη (Client) ... 65
4.4 Ρύθμιση και Εκτέλεση ... 65
Κεφάλαιο 5 ... 69
Αξιολόγηση του συστήματος ... 69
2.1 Μελέτη της κλιμακωσιμότητας (Scalability) του συστήματος ... 69
5.2 Σύγκριση με τα υπάρχοντα συστήματα διαχείρισης ... 71
5.3 Μελέτη εκτέλεσης για διαφορετικά είδη φορτίων ... 72
5.4 Καταλληλότητα ανάλογα με την παραλληλία του προγράμματος ... 73
Κεφάλαιο 6 ... 75
Συμπεράσματα και μελλοντικές κατευθύνσεις ... 75
Προτάσεις για μελλοντική επέκταση ... 76
Βιβλιογραφία ... 80

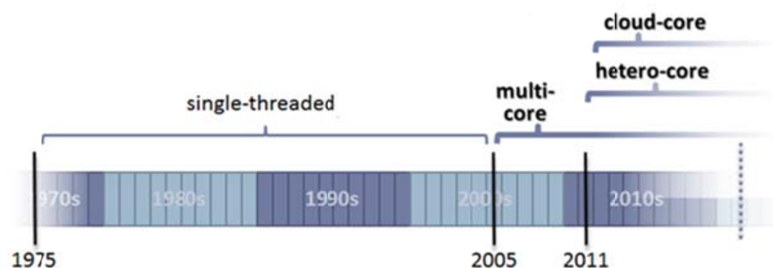
Κεφάλαιο 1

Παράλληλα Συστήματα Επεξεργασίας

1.1 Ιστορικά

Από το 1975 περίπου έως το 2005, η βιομηχανία των υπολογιστών κατάφερε κάτι το αξιοσημείωτο: Μέσα σε 30 χρόνια τοποθετήθηκε ένας προσωπικός υπολογιστής σε κάθε γραφείο, σε κάθε σπίτι και σε κάθε τσέπη. Το 2005, ωστόσο, σχετικά πρόσφατα συγκριτικά με τη συγγραφή της διπλωματικής αυτής, η υπολογιστική τεχνολογία αναγκάστηκε να πάρει μια διαφορετική τροπή, καθώς ήταν η σημαδιακή χρονιά μετάβασης από την σειριακή στην παράλληλη επεξεργασία για τους υπολογιστές. Η βιομηχανία έπρεπε να βρει νέες λύσεις για να συνεχίσει την εξέλιξη της τεχνολογίας με τους ρυθμούς που προέβλεπε ο νόμος του Moore και ταυτόχρονα να αντιμετωπίσει φυσικούς περιορισμούς, όπως π.χ. αύξηση της ισχύος στα κυκλώματα. Οι αλλαγές αυτές θα έπρεπε να είναι ριζικές και να καλύπτουν όλο το φάσμα της στοίβας λογισμικού, από τα λειτουργικά συστήματα μέχρι τα προγραμματιστικά εργαλεία και τις γλώσσες.

Από το 2005 και μετά λοιπόν, ο στόχος της βιομηχανίας έγινε η εγκατάσταση *παράλληλων προσωπικών υπολογιστών* σε κάθε γραφείο, σπίτι και τσέπη. Το 2011 μάλιστα ήταν η χρονιά της ολοκλήρωσης αυτής της μετάβασης, με την άφιξη πολυπύρηνων tablets (Ipad 2, Playbook, Nook Tablet), κινητών τηλεφώνων (Galaxy S II, IPhone 4S). Αυτή τη φορά λοιπόν η μετάβαση διήρκησε μόνο 6 χρόνια και γνωρίζουμε όλοι πολύ καλά ότι δεν υπάρχει γυρισμός στα μονοπύρηνα συστήματα, καθώς τα παράλληλα συστήματα επεξεργασίας αποδίδουν τάξεις μεγέθους καλύτερα και σχεδόν σε κάθε περίπτωση μπορούμε να αναπτύξουμε μια παράλληλη εφαρμογή που να είναι πιο αποδοτική από την αντίστοιχη σειριακή.



Εικόνα 1: Η χρονική εξέλιξη της υπολογιστικής τεχνολογίας, από πλευράς επεξεργασίας.

1.1.1 Στάδια της Υπολογιστικής Τεχνολογίας

Πιο αναλυτικά οι φάσεις που πέρασε η βιομηχανία των υπολογιστών μέχρι το σήμερα είναι:

1975-2005 : Κλασσική σειριακή επεξεργασία

Με τη γέννηση της υπολογιστικής τεχνολογίας, το προφανές μοντέλο που ακολουθήθηκε ήταν αυτό της σειριακής επεξεργασίας, σύμφωνα με τα πρότυπα που έθεσε ο John von Neumann και άλλοι νεωτεριστές της εποχής. Για 30 περίπου χρόνια, η εξέλιξη της βιομηχανίας ακολουθούσε τον νόμο του Moore, σύμφωνα με τον οποίο ο αριθμός των τρανζίστορ σε ένα κύκλωμα διπλασιαζόταν κάθε 18 μήνες. Έτσι, οι επεξεργαστές γίνονταν ολοένα και πιο πολύπλοκοι και η εκτέλεση μιας μονάδας επεξεργασίας ή αλλιώς νήματος (thread) γινόταν συνεχώς πιο γρήγορη. Αυτό ήταν πολύ βολικό για τους προγραμματιστές αφού με ελάχιστο κόπο, έβλεπαν τις εφαρμογές τους να βελτιώνονται σε απόδοση με την άφιξη νέων επεξεργαστών. Μπορούμε να χωρίσουμε την περίοδο αυτή σε δυο υπό-περιόδους, ανάλογα με τις αλλαγές που γίνονταν κάθε φορά στους επεξεργαστές :

- 1970-1980 : Κάθε γενιά επεξεργαστή χρησιμοποιούσε τα επιπλέον διαθέσιμα τρανζίστορ για να εισάγει ένα νέο χαρακτηριστικό (out of order execution, pipelining κτλ.) με σκοπό την γρηγορότερη εκτέλεση κώδικα σε ένα νήμα.
- 1980-2000 : Τα εξτρά τρανζίστορ χρησιμοποιούνταν για να βελτιώσουν πολλές μικρότερες υπομονάδες.

Να σημειωθεί ότι ακόμα και σήμερα βελτιώνουμε την εκτέλεση μονο-νηματικού κώδικα, εκμεταλλευόμενοι την εξέλιξη της τεχνολογίας των υλικών, απλώς όχι με τον μανιώδη ρυθμό που αυτό γινόταν τα πρώτα χρόνια.

2005- : Ομογενή πολυπύρηνα συστήματα

Όταν φτάσαμε στο σημείο, όπου η επιπλέον επίδοση που παίρναμε από την αύξηση των τρανζίστορ σε ένα chip, ήταν πολύ μικρή συγκριτικά με την κατανάλωση ισχύος, η λύση ήταν να τοποθετηθούν πολλαπλοί πυρήνες σε ένα chip. Έτσι μπορούσαμε να συνεχίσουμε να εκμεταλλευόμαστε την εκθετικά αυξανόμενη υπολογιστική ισχύ των υπολογιστών, αλλά σε μια μορφή που ήταν πιο δύσκολα εκμεταλλεύσιμη, αφού το βάρος τώρα έπεσε στους προγραμματιστές, που έπρεπε πλέον να γράψουν *παράλληλα προγράμματα* για να εκμεταλλευτούν το νέο υλικό.

Έτσι άρχισε να αναπτύσσεται το κομμάτι της *Παράλληλης Επεξεργασίας*, τόσο σε θεωρητικό επίπεδο (με νόμους, θεωρίες και τεχνικές), όσο και σε επίπεδο υλοποίησης με την ανάπτυξη παράλληλων Runtimes , όπως το Threading Building Blocks της Intel, παράλληλους debuggers, profilers και ενημερωμένα λειτουργικά συστήματα που τα υποστηρίζουν.

2009- : Ετερογενείς πυρήνες

Ήδη κάθε υπολογιστής διαθέτει παραπάνω από ένα είδος επεξεργαστών, καθώς οι περισσότεροι φορητοί υπολογιστές, οι κονσόλες και τα tablets διαθέτουν εκτός από CPUs και επεξεργαστές γραφικών (Graphics Processing Units ή GPUs) , ικανές να εκτελέσουν κώδικα γενικού σκοπού. Η ανοιχτή ερώτηση που παραμένει στη βιομηχανία σήμερα δεν είναι αν μια εφαρμογή θα μοιράζεται μεταξύ των διαφορετικών τύπων πυρήνων, αλλά το "πόσο

διαφορετικοί” θα πρέπει να είναι αυτοί οι πυρήνες – αν θα έχουν δηλαδή πρακτικά κοινό ρεπερτόριο εντολών και θα αποτελούν μίξη λίγων ισχυρών πυρήνων (πιο αποδοτικών στην εκτέλεση ακολουθιακού κώδικα) και πολλών μικρότερων πυρήνων (για την εκτέλεση παράλληλου κώδικα) ή αν θα είναι πυρήνες με διαφορετικές δυνατότητες που θα υποστηρίζουν μόνο υποσύνολα των γλωσσών προγραμματισμού γενικού σκοπού, όπως είναι η C ή η C++.

Υπάρχουν δύο κύριες κατηγορίες ετερογένειας :

- Μεγάλοι/Γρήγοροι - Μικροί/Αργοί πυρήνες

  Ο μικρότερος βαθμός ετερογένειας είναι όταν όλοι οι πυρήνες είναι γενικού σκοπού με το ίδιο σύνολο εντολών (instruction set), αλλά με μερικούς από αυτούς να είναι πιο ισχυροί από τους άλλους καθώς διαθέτουν περισσότερο υλικό για να επιταχύνουν την εκτέλεση του προγράμματος. Στο μοντέλο αυτό μερικοί κόμβοι είναι πολύπλοκοι, βελτιστοποιημένοι για να τρέχουν τα σειριακά κομμάτια του προγράμματος, ενώ άλλοι μικρότεροι κόμβοι πετυχαίνουν καλύτερες επιδόσεις στα παράλληλα μέρη. Παρότι όμως χρησιμοποιούν το ίδιο instruction set, ο μεταγλωττιστής πολλές φορές θα παράγει διαφορετικό κώδικα.

- Πυρήνες Γενικού – Ειδικού σκοπού

  Πολλές φορές βλέπουμε συστήματα πολλαπλών πυρήνων να έχουν διαφορετικές επεξεργαστικές δυνατότητες, συμπεριλαμβανομένου ότι μερικοί πυρήνες μπορεί να μην έχουν τη δυνατότητα υποστήριξης των κοινών γλωσσών προγραμματισμού όπως η C ή η C++. Το 2006, με την άφιξη της κονσόλας PlayStation 3, ο επεξεργαστής Cell της IBM χάραξε το δρόμο με την ενσωμάτωση διαφορετικών ειδών πυρήνων σε ένα chip. Η ιδέα ήταν ότι υπήρχε ένας επεξεργαστής γενικού σκοπού που υποστηριζόταν από 8 πυρήνες ειδικού σκοπού (SPUs). Επίσης, τα τελευταία χρόνια βλέπουμε την χρήση GPUs ως ειδικών επεξεργαστών που υποστηρίζουν τους επεξεργαστές γενικού σκοπού. Αυτοί οι επεξεργαστές τρέχουν συγκεκριμένα κομμάτια κώδικα πιο γρήγορα, πιο αποδοτικά και πιο φθηνά (π.χ. καταναλώνοντας λιγότερη ισχύ).

Η ετερογένεια ενισχύει την πρώτη τάση (πολυπύρηνα συστήματα), διότι αν μερικοί από τους πυρήνες είναι μικρότεροι, τότε μπορούμε να χωρέσουμε περισσότερους σε ένα chip. Πράγματι, παραλληλία τάξης μεγέθους 100 ή και ακόμα 1000 μπορεί να επιτευχθεί στους περισσότερους υπολογιστές σήμερα (για προγράμματα που μπορούν να εκμεταλλευτούν τις GPUs).
Επιπλέον γνωρίζουμε ότι η μετάβαση στους ετερογενείς πυρήνες είναι μόνιμη, αφού διαφορετικά είδη υπολογισμών τρέχουν από την φύση τους γρηγορότερα σε διαφορετικούς τύπους πυρήνων – συμπεριλαμβανομένου και του γεγονότος ότι διαφορετικά κομμάτια της ίδιας εφαρμογής θα τρέξουν γρηγορότερα σε ένα μηχάνημα με πολλούς τύπους πυρήνων.

2010- : Ελαστικές υπηρεσίες σε Cloud περιβάλλον

Η «Υποδομή ως Υπηρεσία» (Infrastructure as a Service ή IaaS), του Cloud είναι ουσιαστικά η εκμίσθωση κατά ζήτηση υπολογιστικών πόρων με σκοπό την επέκταση των δυνατοτήτων ενός συνηθισμένου μηχανήματος. Πολλές εταιρείες παρέχουν τέτοιου είδους υπηρεσίες στην αγορά με παραδείγματα το Amazon Web Services της ομώνυμης εταιρείας, το Azure της Microsoft και το App Engine της Google.

Το IaaS επίσης ενισχύει την τάση για ετερογένεια, αφού η κύρια ιδέα είναι η ανάπτυξη μεγάλων αριθμών κοινών κόμβων, καθένας από τους οποίους αποτελείται από πολλαπλούς και ετερογενείς πυρήνες.

Οι τεχνολογίες υπολογισμού που περιγράψαμε, όσον αφορά τους επεξεργαστές συνοψίζονται στον παρακάτω πίνακα.

Τεχνολογία	Περίληψη	Στάδια	Επιδράσεις	Παραδείγματα
Μονοπύρηνα Συστήματα	Ολοένα και πιο πολύπλοκοι πυρήνες - Γρηγορότερη Εκτέλεση σειριακού κώδικα	1970-1990 : Προσθήκη ενός μεγάλου χαρακτηριστικού σε κάθε γενιά 1990-2005 : Περισσότερες και μικρότερες βελτιώσεις / γενιά	Το ίδιο πρόγραμμα έτρεχε γρηγορότερα με την επόμενη γενιά επεξεργαστών, χωρίς αλλαγές	Μονοπύρηνη Αρχιτεκτονική x86, ARM
Πολυπύρηνα Συστήματα	Περισσότεροι του ενός πυρήνες / chip.	2005- : Πολλοί μεγάλοι και σύνθετοι πυρήνες 2012-: Περισσότεροι και μικρότεροι πυρήνες	Αλλαγή προγραμματιστικού στυλ : Παράλληλος Προγραμματισμός	SPARC Niagara, x86, Intel MIC
Ετερογενή πολυπύρηνα συστήματα	Πυρήνες διαφορετικών τύπων	Μεγάλοι/Γρήγοροι vs. Μικροί/Αργοί πυρήνες Γενικού Σκοπού (π.χ CPU) vs. Ειδικού σκοπού (π.χ GPU)	Δύσκολο στην εκμετάλλευση : Πρέπει να γραφεί παράλληλος και διαμοιρασμένος κώδικας	Cell (PS3) Intel Xeon + MIC NVIDIA GPUs
Cloud	Υπολογιστική Ισχύς κατά ζήτηση	Πολλοί και Ετερογενείς πυρήνες	Μίσθωση υλικού και πλατφόρμων	Amazon Elastic Cloud Google App Engine

1.1.2 Αρχιτεκτονικές Μνήμης

Μέχρι στιγμής δεν αναφέραμε τίποτα όσον αφορά τη μνήμη. Παρακάτω θα γίνει μια σύντομη αναφορά και ιστορική αναδρομή των διαφορετικών αρχιτεκτονικών μνήμης, που συμβάδιζαν με τις εξελίξεις στο επεξεργαστικό κομμάτι που αναλύθηκε προηγουμένως. Οι σημαντικότερες κατηγορίες είναι :

Ενοποιημένη Μνήμη (Unified Memory)

Πρόκειται για την πιο απλή μορφή αρχιτεκτονικής μνήμης. Ένας ενιαίος χώρος διευθύνσεων που αντιστοιχεί σε έναν επεξεργαστή. Η αρχιτεκτονική αυτή περιγράφει την κυρίαρχη τάση από την εφεύρεση των υπολογιστών, μέχρι τα μέσα της δεκαετίας του 2000. Το προγραμματιστικό μοντέλο είναι απλό : Κάθε δείκτης (ή αναφορά σε αντικείμενο) μπορεί να διευθυνσιοδοτήσει κάθε byte στη μνήμη, ενώ κάθε byte είναι εξίσου “απομακρυσμένο” από

τον (μοναδικό) επεξεργαστικό πυρήνα. Ακόμα και σε αυτή την απλή περίπτωση, ο προγραμματιστής θα πρέπει να είναι προσεκτικός με δύο πράγματα : την *τοπικότητα (locality),* δηλαδή το πόσο καλά τα συχνά χρησιμοποιούμενα δεδομένα χωράνε στην κρυφή μνήμη (cache), καθώς και την *σειρά προσπέλασης* (*access order),* το αν για παράδειγμα τα δεδομένα προσπελαύνονται σειριακά ή τυχαία.

NUMA cache

Εδώ το ενιαίο κομμάτι μνήμης παραμένει, αλλά προστίθενται πολλαπλές κρυφές μνήμες (caches) που αντιστοιχούν στους διαφορετικούς πυρήνες. Η αρχιτεκτονική αυτή συναντάται ευρέως στις σημερινές πολυπύρηνες συσκευές. Το πλεονέκτημα του ενιαίου χώρου διευθύνσεων μνήμης που προσφέρει η Ενοποιημένη Μνήμη παραμένει, ωστόσο ο προγραμματιστής πρέπει να αντιμετωπίσει δύο επιπρόσθετα φαινόμενα που επηρεάζουν την τελική απόδοση : η *τοπικότητα* παίζει ρόλο σε διαφορετικό βαθμό καθώς οι αποστάσεις μεταξύ των caches δεν είναι πια ίδιες και η *διάταξη (layout)* των δεδομένων μετράει επίσης, αφού πρέπει να κρατήσουμε τα αλληλοεξαρτώμενα δεδομένα σε κοντινές caches.

NUMA RAM

Σύμφωνα με αυτή την αρχιτεκτονική η μνήμη διασπάται σε πολλαπλά φυσικά τμήματα, αλλά διατηρεί ακόμα τον ενιαίο χώρο διευθύνσεων. Στην περίπτωση αυτή η πτώση της επίδοσης γίνεται αισθητή όταν μεταφέρονται δεδομένα μεταξύ πυρήνων που αντιστοιχούν σε διαφορετικά τμήματα RAM, καθώς απαιτείται μεταφορά πάνω από τον δίαυλο της μνήμης. Παραδείγματα τέτοιων συστημάτων περιλαμβάνουν Υπολογιστές Συμμετρικής Πολυεπεξεργασίας (SMP) με πολλαπλούς φυσικούς επεξεργαστές καθώς και αρχιτεκτονικές GPU όπως το CUDA, που παρέχουν ενιαίο χώρο διευθύνσεων μεταξύ CPU και GPU, αλλά τμήματα της μνήμης είναι πιο κοντά στη μία ή στην άλλη.

Ασθενής και Ασυνάρτητη Μνήμη (Incoherent and Weak Memory)

Η αρχιτεκτονική αυτή κρατάει ασυγχρόνιστα τα ξεχωριστά τμήματα της μνήμης με την ελπίδα ότι η αίσθηση για τον κάθε επεξεργαστή ότι έχει το αποκλειστικό δικό του κομμάτι θα τους κάνει να τρέξουν πιο γρήγορα, τουλάχιστον μέχρι τη στιγμή που θα πρέπει να συγχρονιστούν ξανά.

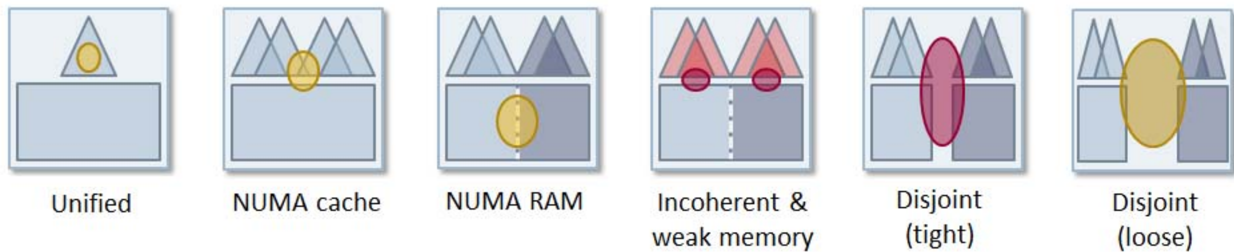
Αποκομμένη – Σφιχτά Συζευγμένη (Disjoint tightly coupled)

Οι διαφορετικοί πυρήνες μπορούν να "βλέπουν" διαφορετικά τμήματα μνήμης, με τη χρήση ενός κοινού διαύλου επικοινωνίας, ενώ τρέχουν σαν μια στενά συνδεδεμένη μονάδα που έχει χαμηλή καθυστέρηση (latency). Τρανό παράδειγμα οι σύγχρονες GPUs , των οποίων η τοπική μνήμη δεν μοιράζεται με την CPU. Η μόνη έννοια του προγραμματιστή πλέον είναι η μεταφορά δεδομένων μεταξύ των διαφορετικών αυτών τμημάτων της μνήμης.

Αποκομμένη – Χαλαρά Συζευγμένη (Disjoint loosely coupled)

Πρόκειται για την αρχιτεκτονική που χρησιμοποιείται στο Cloud, όπου οι επεξεργαστές απλώνονται όχι μόνο σε ξεχωριστά συστήματα, αλλά και σε διαφορετικά δωμάτια, κτίρια και datacenters. Αυτό απομακρύνει τα τμήματα της μνήμης ακόμα περισσότερο και τώρα οι δίαυλοι επικοινωνίας πρέπει να μετατραπούν σε ολόκληρες δικτυακές υποδομές. Οι προγραμματιστές εδώ πρέπει να αντιμετωπίσουν τις προκλήσεις της *αξιοπιστίας* καθώς οι

κόμβοι μπορούν να χαλάσουν και της *καθυστέρησης* ανάλογα με την απόσταση μεταξύ των κόμβων.

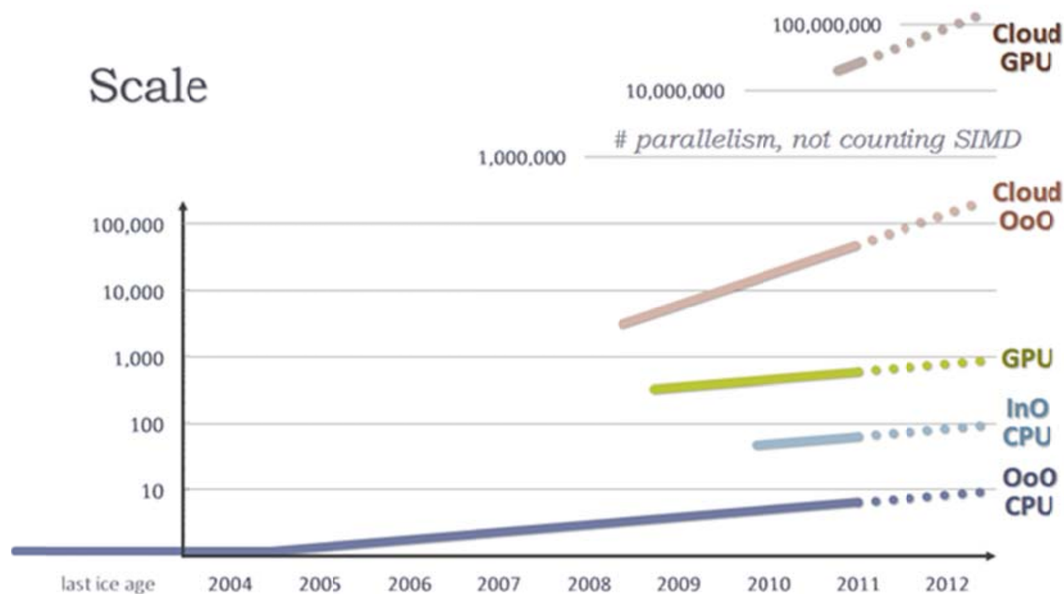


Εικόνα 2: Οι διαφορετικές Αρχιτεκτονικές Μνήμης.
Τα παραλληλόγραμμα αναπαριστούν τη φυσική μνήμη, τα τρίγωνα τις caches και οι ελλείψεις το επίπεδο συνεκτικότητας.

1.1.3 Κλιμακωσιμότητα (Scalability)

Ίσως η σημαντικότερη μετρική των σύγχρονων συστημάτων παράλληλης επεξεργασίας είναι αυτή της επεκτασιμότητας. Εκφράζει την ικανότητα αύξησης της απόδοσης του συστήματος, καθώς προσθέτουμε ολοένα και περισσότερους πόρους σε αυτό. Γενικά υπάρχουν δύο μορφές επεκτασιμότητας : η *οριζόντια* (*scale out*) που σημαίνει συνήθως αύξηση του αριθμού των πόρων (π.χ. προσθήκη περισσότερων κόμβων σε μια συστοιχία υπολογιστών) και η *κάθετη (scale up)* που σημαίνει βελτίωση των ήδη υπαρχόντων πόρων (στο παράδειγμα της συστοιχίας αναβάθμιση των κόμβων με ισχυρότερες CPUs ή περισσότερη και γρηγορότερη μνήμη RAM).

Όσον αφορά τις τεχνολογίες που περιγράφηκαν προηγουμένως, στο παρακάτω διάγραμμα φαίνεται σε γενικές γραμμές πόσο καλά κλιμακώνονται με την πάροδο του χρόνου. Παρατηρούμε ότι οι τεχνολογίες των CPUs και GPUs έχουν κλιμακωσιμότητα που προβλέπεται σχεδόν από τον νόμο του Moore : περισσότερα κυκλώματα κάνουν περισσότερους πόρους διαθέσιμους για τις παράλληλες εφαρμογές. Πρόκειται για κάθετη επεκτασιμότητα. Οι δύο ευθείες στην πάνω πλευρά όμως, που αφορούν το cloud εκμεταλλεύονται την οριζόντια επεκτασιμότητα και πετυχαίνουν πολύ θεαματικά αποτελέσματα. Η ισχύς προέρχεται από πολλούς μικρούς επεξεργαστές αντί για λίγους και ισχυρούς.



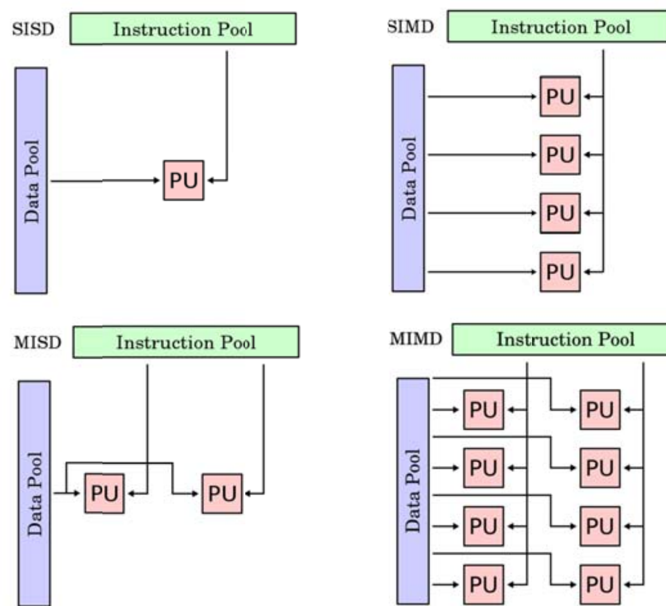
Εικόνα 3: Επεκτασιμότητα (Scalability) για τις διάφορες τεχνολογίες παράλληλης και μη επεξεργασίας.

1.2 Κατηγοριοποίηση Αρχιτεκτονικών Υπολογιστών

Ο Michael Flynn, το 1966 πρότεινε μια κατηγοριοποίηση των Αρχιτεκτονικών Υπολογιστών με βάση τον αριθμό των ταυτόχρονων ροών εκτέλεσης που μπορεί να υποστηρίξει η κάθε μια. Οι 4 αυτές κατηγορίες είναι :

- **Single Instruction, Single Data Stream (SISD)** : Πρόκειται για την σειριακή εκτέλεση εντολών που περιγράφηκε προηγουμένως. Δεν υπάρχει καμία μορφή παραλληλίας, καθώς μια μόνο μονάδα ελέγχου (Control Unit) , φέρνει την επόμενη κάθε φορά εντολή από το ρεύμα των εντολών (Instruction Stream) από τη μνήμη. Έπειτα, η μονάδα ελέγχου παράγει τα κατάλληλα σήματα ελέγχου για να διευθύνει το μοναδικό επεξεργαστικό στοιχείο (processing element) ώστε να επεξεργαστεί μια μονή ροή δεδομένων (single data stream). Πρόκειται για την κλασσική αρχιτεκτονική von Neumann.
- **Single Instruction, Multiple Data Streams (SIMD) :** Αρχιτεκτονική όπου ο υπολογιστής εκμεταλλεύεται πολλαπλές ροές δεδομένων παράλληλα, εκτελώντας την ίδια εντολή σε καθένα από τα επεξεργαστικά στοιχεία. Παράδειγμα είναι οι επεξεργαστές γραφικών (GPUs).
- **Multiple Instruction, Single Data Stream (MISD):** Πολλαπλές εντολές εφαρμόζονται σε μια μοναδική ροη δεδομένων. Πρόκειται για μια σπάνια περίπτωση και χρησιμοποιείται συνήθως για έλεγχο σφαλμάτων σε ετερογενή συστήματα, όπου διαφορετικοί τύποι εντολών πάνω στα ίδια δεδομένα πρέπει να συμφωνήσουν στα αποτελέσματα.
- **Multiple Instruction, Multiple Data Stream (MIMD) :** Πολλαπλά αυτόνομα επεξεργαστικά στοιχεία εκτελούν διαφορετικές εντολές πάνω σε διαφορετικά

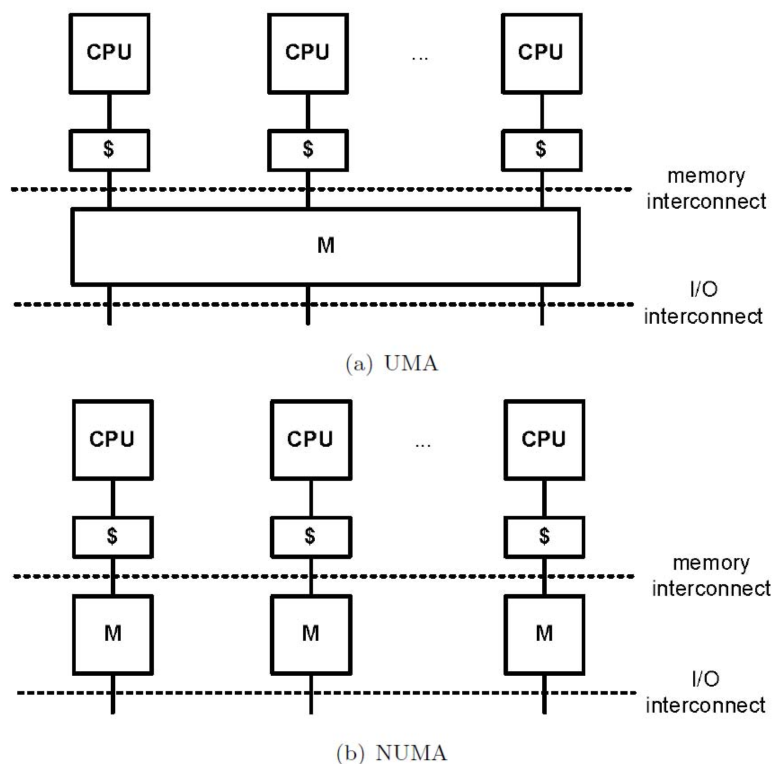
δεδομένα. Παράδειγμα είναι τα κατανεμημένα συστήματα (distributed systems). Αυτά χρησιμοποιούν είτε αρχιτεκτονική κοινής μνήμης, είτε κατανεμημένης μνήμης.



Εικόνα 4: Κατηγοριοποίηση Αρχιτεκτονικών κατά Flynn.

1.2.1 Αρχιτεκτονικές Κοινής Μνήμης (Shared Memory)

Σύμφωνα με το μοντέλο κοινής μνήμης πολλαπλές μονάδες επεξεργασίας προσαρτώνται σε ένα κοινό διάδρομο διασύνδεσης με τη μνήμη. Αν όλα τα τμήματα της μνήμης απέχουν εξίσου από τις CPUs, το σύστημα ονομάζεται UMA (Uniform Memory Access), ενώ αν μερικά κομμάτια της μνήμης βρίσκονται πιο κοντά σε μερικούς επεξεργαστές από κάποιους άλλους, ονομάζεται NUMA (Non-Uniform Memory Access). Τα συστήματα κοινής μνήμης γενικά παρέχουν το επιθυμητό χαρακτηριστικό του κοινού και ενιαίου χώρου διευθύνσεων μνήμης μεταξύ των προγραμμάτων που εκτελούνται σε διαφορετικούς επεξεργαστές. Δομές δεδομένων μπορούν να μοιράζονται μεταξύ των CPUs που μπορούν να επικοινωνούν με τις γνωστές εντολές load/store στην κύρια μνήμη. Έτσι οι αρχιτεκτονικές Κοινής Μνήμης θεωρούνται γενικά παράλληλες πλατφόρμες με μεγάλη ευκολία προγραμματισμού. Από την άλλη πλευρά όμως παρουσιάζουν και αρκετές αδυναμίες, όπως συνθήκες ανταγωνισμού (race conditions) και εξαρτήσεις δεδομένων και αν ο προγραμματιστής δεν είναι προσεκτικός μπορούν να γίνουν αντιπαραγωγικές και εύκολα επιρρεπείς σε σφάλματα.



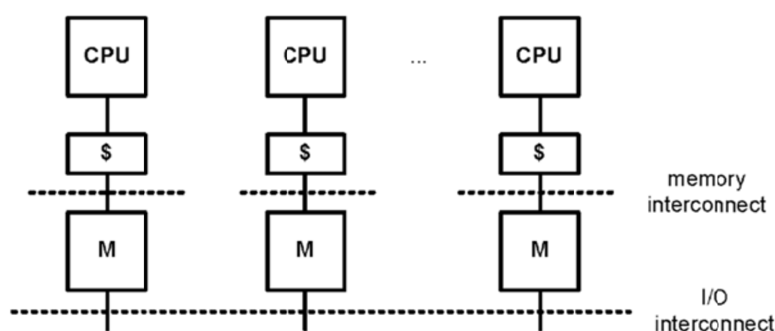
Εικόνα 5: Αρχιτεκτονικές Κοινής Μνήμης (a) UMA και (b) NUMA

Παραδείγματα συστημάτων και εργαλείων κοινής μνήμης είναι τα POSIX Threads, το OpenMP, η Cilk και τα Threading Building Blocks.

1.2.2 Αρχιτεκτονικές Κατανεμημένης Μνήμης (Distributed Memory)

Οι παράλληλες εφαρμογές στην βιομηχανία των υπολογιστών έχουν τεράστιες απαιτήσεις σε επεξεργαστική ισχύ και απαιτούν χιλιάδες, ακόμα και εκατομμύρια επεξεργαστές για να εκτελεστούν αποδοτικά. Είναι προφανές ότι η μικρής κλίμακας αρχιτεκτονική κοινής μνήμης δεν επαρκεί για να καλύψει αυτές τις ανάγκες. Τα συστήματα που χρησιμοποιούνται σε αυτή την περίπτωση ακολουθούν την αρχιτεκτονική της Εικόνας 6, όπου κάθε επεξεργαστικός κόμβος με την δική του ξεχωριστή μνήμη συνδέεται με τους υπόλοιπους είτε μέσω κοινού Gigabit Ethernet δικτύου είτε μέσω πιο αποδοτικών δικτύων, όπως είναι το Myrinet ή το Infiniband.

Έτσι, στα συστήματα κατανεμημένης μνήμης ο κάθε κόμβος δεν έχει άμεση πρόσβαση στην μνήμη των υπολοίπων και έτσι οι επεξεργαστές πρέπει να επικοινωνούν μεταξύ τους μέσω του δικτύου διασύνδεσης. Οι προγραμματιστές πρέπει λοιπόν να υιοθετήσουν μια νέα προσέγγιση ανάπτυξης των εφαρμογών όπου κυρίαρχο στοιχείο είναι η επικοινωνία μεταξύ των επεξεργαστών με τη μορφή μηνυμάτων. Παρότι αυτή η τεχνική έχει αυξημένο βαθμό δυσκολίας και είναι χρονοβόρα, εντούτοις κυριαρχεί στον χώρο των κατανεμημένων συστημάτων.



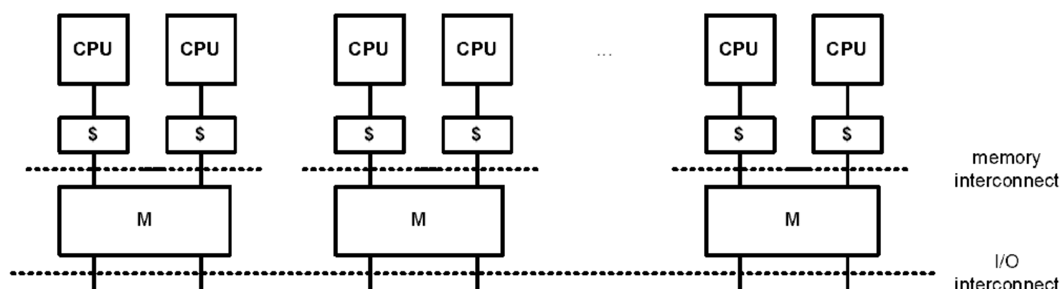
Εικόνα 6: Αρχιτεκτονική Κατανεμημένης Μνήμης

Το κυρίαρχο πρότυπο των κατανεμημένων συστημάτων είναι το MPI (Message Passing Interface). Πρόκειται για μια βιβλιοθήκη που ακολουθεί το προγραμματιστικό μοντέλο ανταλλαγής μηνυμάτων που περιγράφηκε πιο πάνω. Στην πραγματικότητα, η πλειοψηφία των προσομοιώσεων που τρέχουν σε τεράστιους υπερυπολογιστές χρησιμοποιούν το MPI.

1.2.3 Υβριδικές Αρχιτεκτονικές (Hybrid Systems)

Στην πράξη, τα συστήματα παραγωγής χρησιμοποιούν έναν συνδυασμό των παραπάνω αρχιτεκτονικών με σκοπό να εκμεταλλευτούν τα πλεονεκτήματα και των δύο. Στην παρακάτω εικόνα φαίνεται ένα παράδειγμα μιας τέτοιας υβριδικής αρχιτεκτονικής. Γνωρίζοντας την ακριβή αρχιτεκτονική του συστήματος, ο προγραμματιστής μπορεί να γράψει ακόμα πιο αποδοτικό κώδικα. Για παράδειγμα, μπορεί να χωρίσει το κύριο πρόβλημα

του σε υπο-προβλήματα, αναθέτοντας το καθένα σε κάθε κόμβο, και να βελτιστοποιήσει το κάθε υποπρόβλημα για την αρχιτεκτονική κοινής μνήμης του καθενός από αυτούς.



Εικόνα 7: Υβριδική Αρχιτεκτονική

1.3 Επεξεργαστές Γραφικών (GPUs)

Ένας επεξεργαστής γραφικών είναι στην ουσία ένα ηλεκτρονικό κύκλωμα του οποίου ο αρχικός σκοπός ήταν να χειρίζεται και να επιταχύνει την δημιουργία των τελικών εικόνων που θα προβάλλονταν στην οθόνη του υπολογιστή. Χρησιμοποιούνται σε μια πληθώρα συσκευών, όπως για παράδειγμα : ενσωματωμένα συστήματα, κινητά τηλέφωνα, προσωπικούς υπολογιστές, σταθμούς εργασίας και κονσόλες παιχνιδιών. Όσον αφορά τον υπολογιστή, οι GPUs μπορεί να βρίσκονται είτε ως μέρος μιας ξεχωριστής κάρτας PCI, είτε ενσωματωμένες στην μητρική κάρτα , είτε ακόμη και ενσωματωμένες στο κύκλωμα της CPU.

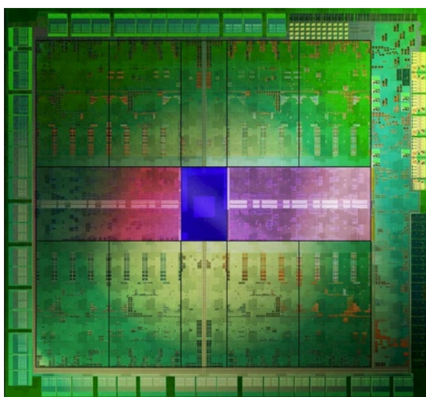
Ιστορικά , ο όρος GPU έγινε δημοφιλής από την εταιρεία NVIDIA , η οποία , το 1999 έφερε στην αγορά την GeForce 256, ως την "πρώτη GPU στον κόσμο". Είχε δυνατότητα επεξεργασίας 10 εκατομμυρίων πολυγώνων το δευτερόλεπτο. Η ανταγωνίστρια εταιρεία ATI ονόμασε το δικό της αντίστοιχο προϊόν VPU (Visual Processing Unit).

Τα τελευταία χρόνια, κυρίως λόγο του ανταγωνισμού των δύο εταιρειών οι επεξεργαστές γραφικών έχουν αυξήσει κατά τάξεις μεγέθους την επεξεργαστική τους ισχύ. Η ισχύς τους, πέρα από την πρόοδο της τεχνολογίας των κυκλωμάτων , προέρχεται από τον πολύ μεγάλο αριθμό επεξεργαστικών πυρήνων τους. Ενδεικτικά ο επεξεργαστής "Kepler GK110" της NVIDIA έχει 2496 επεξεργαστικούς πυρήνες και πετυχαίνει 3.52 Tflops επίδοση, για αριθμούς κινητής υποδιαστολής απλής ακριβείας.

Παράδειγμα Αρχιτεκτονικής (Nvidia Fermi)

Η παρούσα διπλωματική εστιάζει στους επεξεργαστές γραφικών της NVIDIA λόγω ανωτερότητας που παρέχουν από άποψη υλικού σε σχέση με τους ανταγωνιστές της. Συνεπώς θα χρησιμοποιηθεί η τεχνολογία της και όλα τα προγραμματιστικά εργαλεία που παρέχει.

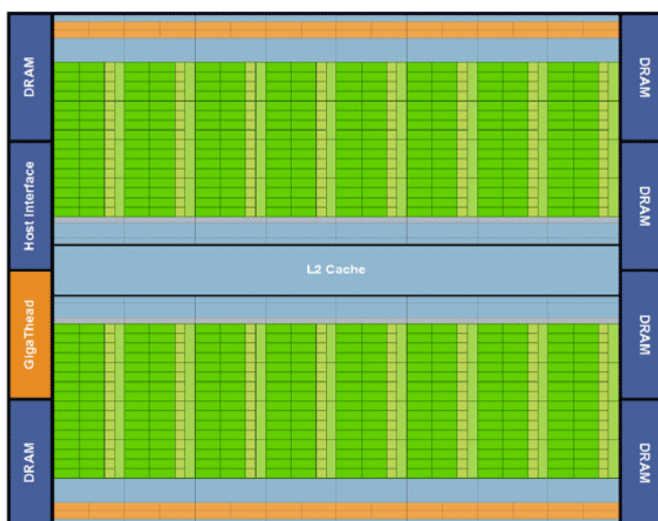
Κατα την συγγραφή της διπλωματικής αυτής ,η πιο πρόσφατη τεχνολογία GPU της εταιρείας φέρει την ονομασία Kepler. Οι GPUs της τεχνολογίας Kepler (GK110) αποτελούνται από 7,1 εκατομμύρια transistor και σχεδιάστηκαν κυρίως για εφαρμογή σε συστήματα High-Performance-Computing.



Εικόνα 8 : Άποψη του κυκλώματος του επεξεργαστή Kepler της Nvidia

Παρόλα αυτά λόγω διαθεσιμότητας και τεχνογνωσίας της αμέσως προηγούμενης αρχιτεκτονικής που φέρει την ονομασία “Fermi”, θα περιγραφεί η τελευταία. Αυτό βεβαίως δεν βλάπτει τη γενικότητα, καθώς το σύστημα που προτείνουμε τελικά είναι ανεξάρτητο της αρχιτεκτονικής.

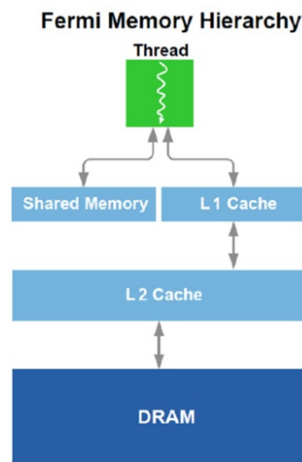
Οι Fermi GPUs αποτελούνται από 3 δισεκατομμύρια τρανζίστορ και περιλαμβάνουν έως 512 επεξεργαστικούς πυρήνες. Ένας επεξεργαστικός πυρήνας (CUDA core) εκτελεί μια πράξη κινητής υποδιαστολής, ή μια ακέραια πράξη ανά παλμό του ρολογιού για το λογαριασμό ενός νήματος (thread). Οι 512 πυρήνες οργανώνονται σε 16 SM (Streaming Multiprocessors), καθένας εκ των οποίων έχει 32 πυρήνες. Η GPU έχει 6 64-bit διεπαφές με την μνήμη, συνολικά δηλαδή 384-bit, υποστηρίζοντας μέχρι 6 GB GDDR5 μνήμης RAM. Υπάρχει μια διεπαφή Host που συνδέει την GPU με την CPU μέσω του διαύλου PCI Express και ένας scheduler που μοιράζει αποδοτικά τις εργασίες στους schedulers του κάθε SM.



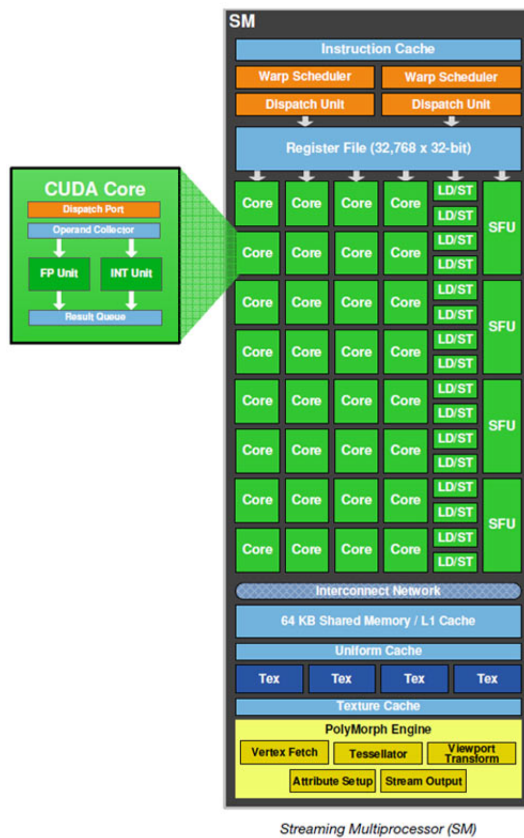
Εικόνα 9 : Αρχιτεκτονική μιας Fermi GPU. Φαίνονται οι 16 SMs και όλες οι διεπαφές που περιγράφηκαν.

Κάθε SM περιλαμβάνει τα παρακάτω :

- **32 CUDA cores** αποτελούμενους από δύο επεξεργαστικά στοιχεία : μια αριθμητική λογική μονάδα (ALU) για ακεραίους και μια για κινητής υποδιαστολής (FPU).
- **16 μονάδες load/store** , οι οποίες επιτρέπουν τον υπολογισμό διευθύνσεων προορισμού και προέλευσης για 16 thread ταυτόχρονα σε κάθε κύκλο ρολογιού. Περαιτέρω μονάδες φορτώνουν και αποθηκεύουν τα δεδομένα κάθε διεύθυνσης στην cache ή στην DRAM.
- **4 Μονάδες ειδικών λειτουργιών (Special Function Units – SFUs)** , που έχουν την ικανότητα να εκτελούν πράξεις όπως : ημίτονο, συνημίτονο και τετραγωνική ρίζα. Κάθε SFU εκτελεί μια εντολή ανά thread.
- Σχεδιασμό για **διπλή ακρίβεια**. Η αριθμητική διπλής ακρίβειας είναι πλέον το βασικό συστατικό σε εφαρμογές HPC , όπως η γραμμική άλγεβρα ή η κβαντική χημεία. Η αρχιτεκτονική Fermi υποστηρίζει μέχρι 16 παράλληλες εκτελέσεις εντολών διπλής ακρίβειας, ανά SM, ανά παλμό ρολογιού.
- **Διπλό Χρονοδρομολογητή (Dual Warp Scheduler)** : Κάθε SM χρονοδρομολογεί ομάδες των 32 threads που ονομάζονται *warps*. Κάθε SM διαθέτει 2 schedulers και δύο μονάδες για την αποστολή των εντολών (Instruction Dispatch Units) , που επιτρέπουν σε δύο warps να ανατεθούν στα SPs και να εκτελεστούν ταυτόχρονα. Να σημειωθεί βέβαια ότι οι πράξεις κινητής υποδιαστολής διπλής ακρίβειας που μας ενδιαφέρουν εδώ δεν μπορούν να ανατεθούν ταυτόχρονα από έναν τέτοιο dual scheduler.
- **64 KB Κοινής μνήμης και L1 Cache**, η οποία μπορεί να ρυθμιστεί ανάλογα με τις απαιτήσεις (16 KB L1 – 48KB Shared ή το αντίθετο). Η κοινή μνήμη επιτρέπει στα threads ενός block να συνεργάζονται, εκμεταλλεύεται την επαναχρησιμοποίηση των on-chip δεδομένων και μειώνει την off-chip κίνηση.

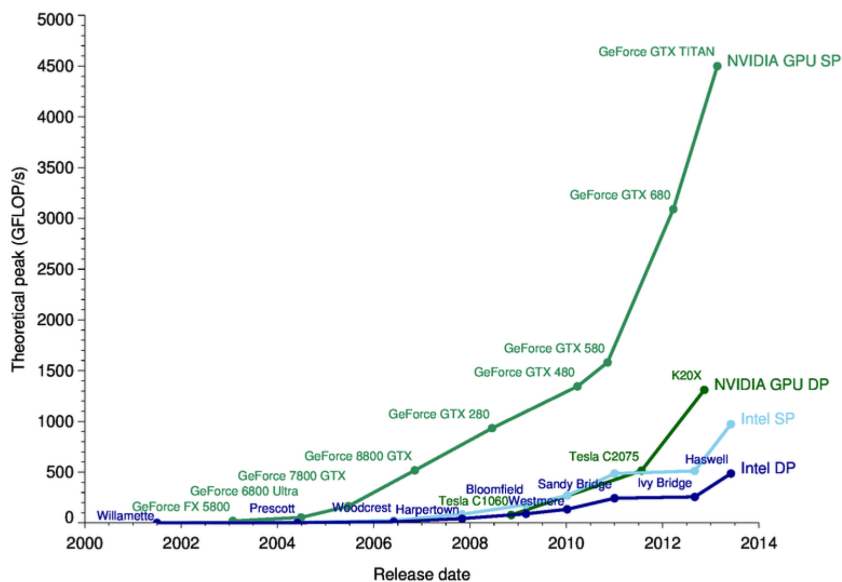


Εικόνα 10 :Η ιεραρχία μνήμης στην αρχιτεκτονική Fermi.



Εικόνα 11: Η πλήρης αρχιτεκτονική ενός Streaming Multiprocessor (SM), στην αρχιτεκτονική Fermi.

Οι επιδόσεις που πετυχαίνουν οι σύγχρονες GPUs είναι πολύ υψηλότερες από αυτές των CPUs, αν και δεν έχουν τις ίδιες επεξεργαστικές δυνατότητες. Μάλιστα, ο η βιομηχανία των GPUs εξελίσσεται τόσο γρήγορα, ώστε η επεξεργαστική ισχύς τους έχει πάψει να ακολουθεί το νόμο του Moore, όπως φαίνεται στο παρακάτω διάγραμμα.



Διάγραμμα 1: Σύγκριση θεωρητικής επεξεργαστικής Ισχύος μεταξύ CPUs και GPUs, για διάφορες γενιές για servers και desktops.

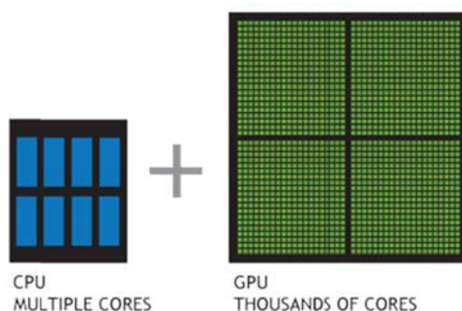
1.4 General Purpose GPU Computing (GPGPU)

Το GPGPU είναι η χρήση της GPU παράλληλα με την CPU για την επιτάχυνση επιστημονικών εφαρμογών και εφαρμογών μηχανικής γενικού σκοπού. Αυτό σημαίνει ότι οι GPUs δεν χρησιμοποιούνται για επεξεργασία γραφικών, τον αρχικό δηλαδή λόγο για τον οποίον κατασκευάστηκαν. Οι εφαρμογές αυτές προέρχονται από διάφορους τομείς όπως :

- Βιοπληροφορική (Bioinformatics)
- Μοριακή δυναμική (Molecular Dynamics)
- Κβαντική Χημεία (Quantum Chemistry)
- Επιστήμη Υλικών (Materials Science)
- Αναλυτικά Μαθηματικά
- Φυσική
- Δυναμική Ρευστών
- Πρόβλεψη κλίματος και καιρού
- Εθνική Άμυνα και Υπηρεσίες Πληροφοριών Στρατού
- Υπολογιστικά Οικονομικά (Computational Finance)
- Σχέδια με υπολογιστή (CAD)
- Υπολογιστική Δομική
- Αυτοματοποίηση Σχεδιασμού Ηλεκτρονικών
- Ανάλυση σε Σεισμολογία

Να σημειωθεί ότι οι εφαρμογές αυτές ανήκουν στην κατηγορία των "embarrassingly parallel" προβλημάτων δηλαδή έχουν τέτοια δομή , ώστε να παραλληλοποιούνται σε πολύ μεγάλο βαθμό.

Αυτό που συμβαίνει στην πράξη είναι ότι η GPU αναλαμβάνει κομμάτια εφαρμογών που τρέχουν στην CPU, τα οποία απαιτούν μεγάλη υπολογιστική ισχύ, ενώ η CPU συνεχίζει την εκτέλεση του δικού της κώδικα. Ο συνδυασμός CPU και GPU είναι πολύ ισχυρός καθώς η μεν CPU αποτελείται από λίγους πυρήνες , που είναι βελτιστοποιημένοι για σειριακή επεξεργασία, ενώ η δε GPU από χιλιάδες μικρούς πυρήνες σχεδιασμένους για παράλληλη επεξεργασία, όπως αναφέρθηκε και στην εισαγωγή. Έτσι τα σειριακά κομμάτια του κώδικα τρέχουν στην CPU και τα παράλληλα (τα λεγόμενα kernels) στην GPU.



Εικόνα 12: Το υπολογιστικό μοντέλο του GPGPU.

1.4.1 Προγραμματισμός GPGPU

Όταν το GPGPU ξεκίνησε τα πρώτα του βήματα, ο μόνος τρόπος προγραμματισμού των επεξεργαστών γραφικών ήταν μέσω του προγραμματισμού της σωλήνωσης γραφικών μέσω γλώσσας μηχανής. Οι μοναδικές διαθέσιμες διεπαφές ήταν αυτές των γραφικών, όπως π.χ. η OpenGL και το Direct3D. Τα τελευταία χρόνια όμως έχουν γίνει προσπάθειες από εταιρείες αλλά και ακαδημαϊκά ιδρύματα να δημιουργηθούν γλώσσες υψηλού αλλά και μεσαίου επιπέδου που να δώσουν μια επιπλέον ώθηση, ευελιξία, ευχρηστία και ευκολίες στην κοινότητα του GPGPU. Θα αναφερθεί παρακάτω με συντομία ένα σημαντικό μέρος αυτών των προσπαθειών, με έμφαση στα πλεονεκτήματα, στα μειονεκτήματα και στις ιδιαιτερότητες της κάθε προσπάθειας.

Γλώσσες Σκίασης (Shaders)

Αυτός ο τρόπος προγραμματισμού είναι ο πλησιέστερος στον τρόπο που προγραμματίζονται τα γραφικά. Οι γλώσσες αυτές μπορούν να χρησιμοποιηθούν για γενικές εφαρμογές με την κατάλληλη απεικόνιση της εφαρμογής που μας ενδιαφέρει σε πρόβλημα επεξεργασίας γραφικών.

Οι σημαντικότερες και πιο διαδεδομένες γλώσσες σκίασης που έχουν χρησιμοποιηθεί για προγραμματισμό GPUs σε γενικές εφαρμογές είναι οι εξής:

C for Graphics (Cg)

Η Cg είναι μια γλώσσα σκίασης που αναπτύχθηκε από την NVIDIA. Έχει παρόμοια σύνταξη με την γλώσσα προγραμματισμού C, περιλαμβάνοντας παράλληλα κάποια στοιχεία από τις C++ και Java και από κάποιες πιο παλιές γλώσσες σκίασης. Είναι ανεξάρτητη πλατφόρμας χάρη στη συμβατότητά της με το πρότυπο OpenGL, ενώ είναι συμβατή και με Direct3D. Για να αντιμετωπίσει διαφορές και αποκλίσεις που υπάρχουν μεταξύ διαφόρων προγραμματιστικών διεπαφών γραφικών περιλαμβάνει έναν αριθμό προτύπων, τα οποία αν ακολουθηθούν εγγυώνται ότι τα προγράμματα που θα γράψουμε σε αυτήν την γλώσσα σκίασης θα τρέξουν στις κάρτες γραφικών που υποστηρίζουν το επιλεγμένο πρότυπο.

High Level Shading language (HLSL)

Αυτή η υψηλού επιπέδου γλώσσα σκίασης εμφανίστηκε μαζί με το DirectX 9. Γλώσσες σκίασης υποστηρίζονταν και από την προηγούμενη έκδοση του DirectX (την 8), αλλά ο μόνος τρόπος να γραφτούν ήταν μέσω γλώσσας μηχανής. Η Cg και η HLSL είναι σχεδόν η ίδια γλώσσα και αναπτύχθηκαν ύστερα από συνεργασία της NVIDIA με την Microsoft, αλλά έχουν διαφορετικά ονόματα για εμπορικούς λόγους. Η διαφορά τους είναι ότι η HLSL μεταγλωττίζεται μόνο σύμφωνα με το πρότυπο Direct3D και όχι με το OpenGL.

OpenGL Shading Language (GLSlang)

Και αυτή η γλώσσα σκίασης είναι βασισμένη στην γλώσσα προγραμματισμού C και έχει διατηρήσει πολλά στοιχεία της παρότι αποτελούν στενωπό στην επίδοση κάποιες φορές. Εμπλουτίζει την C με νέους τύπους διανυσμάτων και πίνακα, δομές που χρησιμοποιούνται κατά κόρον στην επεξεργασία γραφικών, ενώ δανείζεται και κάποια στοιχεία από την C++, όπως για παράδειγμα ο πολυμορφισμός συναρτήσεων. Η GLSlang είναι διαθέσιμη από την έκδοση 1.4 του OpenGL και αποτελεί τον πυρήνα της έκδοσης 2.0 του OpenGL.

Brook

Το BrookGPU είναι ένα ερευνητικό πρόγραμμα του πανεπιστημίου του Stanford με σκοπό να καταστήσει δυνατό σε προγραμματιστές που δεν είχαν άμεση σχέση με τα γραφικά να μπορέσουν να γράψουν αποδοτικές εφαρμογές για GPUs. Είναι μια προσπάθεια που βρήκε μεγάλη απήχηση στην επιστημονική κοινότητα, καθώς με τη χρήση του έχουν γραφτεί πολλές εφαρμογές. Αποτελεί μια γλώσσα προγραμματισμού, η οποία είναι στην ουσία μια επέκταση της γλώσσας C, με επεκτάσεις που έχουν να κάνουν με την χρήση και αξιοποίηση του προγραμματιστικού μοντέλου σε ροές. Για το BrookGPU, οι GPUs είναι μια πλατφόρμα εκτέλεσης κώδικα που μπορεί να παραλληλοποιηθεί. Η κάρτα γραφικών, δηλαδή, αποτελεί έναν συνεπεξεργαστή εκτέλεσης παράλληλου κώδικα. Ο μεταγλωττιστής του Brook ονομάζεται brcc. Οι ροές που χρησιμοποιεί το Brook είναι όμοιες με τους πίνακες της C, με τη διαφορά ότι τα στοιχεία τους μπορούν να επεξεργαστούν την ίδια χρονική στιγμή από διαφορετικά επεξεργαστικά στοιχεία/πυρήνες. Οι πυρήνες υπολογισμού του Brook είναι σαν τις συναρτήσεις της C μόνο που εκτελούνται παράλληλα από πολλούς διαφορετικούς πυρήνες. Ουσιαστικά, αυτό που κάνει το Brook είναι να απεικονίζει ένα πραγματικό πρόβλημα σε πρόβλημα γραφικών. Οι πυρήνες υπολογισμού αποτελούν προγράμματα επεξεργασίας τμημάτων, και οι ροές, είναι δεδομένα της επεξεργασίας υφών. Σε τελικό στάδιο, ένα πρόγραμμα Brook μετατρέπεται σε κλήσεις κάποιου API γραφικών.

Microsoft Accelerator

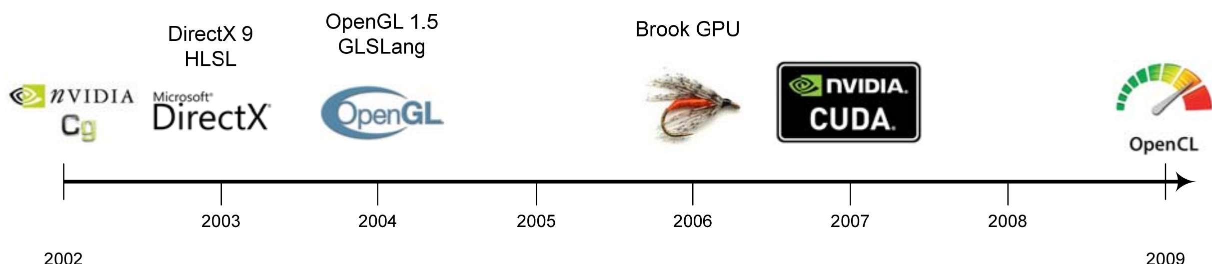
Ο Microsoft Accelerator είναι ένα σύστημα που δημιουργήθηκε από ερευνητές της Microsoft με στόχο να απλοποιήσει την προσπάθεια του GPGPU προσφέροντας μια υψηλού επιπέδου γλώσσα. Σκοπός ήταν αυτή η γλώσσα να είναι κατάλληλη για εφαρμογές με μεγάλο παραλληλισμό σε επίπεδο δεδομένων μέσω μιας βιβλιοθήκης η οποία μπορεί να χρησιμοποιηθεί και από άλλες γλώσσες προγραμματισμού. Ο Accelerator μεταφράζει λειτουργίες που εφαρμόζονται παράλληλα σε σύνολα δεδομένων σε προγράμματα σκίασης εικονοστοιχείων, επιτυγχάνοντας σημαντική επιτάχυνση σε σχέση με κώδικα που τρέχει εξ' ολοκλήρου σε CPUs.

MATLAB

Το MATLAB ή MATrix LABoratory (Εργαστήριο Πινάκων) χρησιμοποιείται ευρέως από ερευνητές για αριθμητικούς υπολογισμούς και σαν μια γλώσσα προγραμματισμού για υπολογιστικές εφαρμογές. Παρ'όλο που χρησιμοποιεί κάποιες βιβλιοθήκες με καλή επίδοση, είναι δυνατόν να επιτευχθούν υψηλές επιδόσεις σε εφαρμογές που τρέχουν σε αυτό εάν γίνει χρήση GPUs. Αυτό μπορεί να επιτευχθεί με χρήση αρχείων με την επέκταση MEX (Matlab EXecutable) τα οποία καλούν υπάρχοντα κομμάτια κώδικα γραμμένα σε C ή FORTRAN. Επίσης το MATLAB μπορεί να επιταχυνθεί και μέσω των βιβλιοθηκών που προσφέρει η τεχνολογία CUDA.

Οι δύο επικρατέστερες τεχνολογίες όμως είναι το CUDA (Compute Unified Device Architecture) της NVIDIA και η ανοιχτού κώδικα OpenCL (Open Computing Language). Το επικρατέστερο αυτή τη στιγμή είναι το CUDA , καθώς η NVIDIA έχει πολύ πιο προηγμένη τεχνολογία από τους ανταγωνιστές της. Όλες πάντως οι τεχνολογίες μοιράζονται κάποια βασικά στοιχεία αρχιτεκτονικής και ακολουθούν την ίδια διαδικασία εκτέλεσης που αποτελείται από :

- **Μεταγλώττιση** : Ειδικοί μεταγλωττιστές (compilers) παράγουν ξεχωριστό κώδικα για την CPU και την GPU αντίστοιχα.
- **Εκτέλεση** : Οι κώδικες εκτελούνται ξεχωριστά με ξεχωριστούς χώρους μνήμης για την GPU και την CPU. Αρχικά τα δεδομένα μεταφέρονται από την CPU στην GPU, έπειτα εκτελείται ο GPU κώδικας (kernel code) και τέλος τα αποτελέσματα μεταφέρονται πίσω στην CPU.
- **Μέθοδοι** : Πρόκειται για γενικότερες μεθόδους που χρησιμοποιούνται στον παράλληλο προγραμματισμό και περιλαμβάνουν :
    - Map : Η μέθοδος αυτή εφαρμόζει μια συγκεκριμένη πράξη (τον πυρήνα ή kernel) σε κάθε επεξεργαστικό στοιχείο. Ας πάρουμε το παράδειγμα του πολλαπλασιασμού κάθε τιμής με μια σταθερά, με σκοπό να αυξήσουμε την φωτεινότητα μιας εικόνας. Το Map είναι αρκετά απλό να υλοποιηθεί στην GPU. Ο προγραμματιστής δεν έχει παρά να χωρίσει το πρόβλημα σε ομάδες pixel και να εφαρμόσει την ίδια πράξη σε όλες αυτές.
    - Reduce: Πρόκειται για την τελική συγκέντρωση των αποτελεσμάτων από πολλές διαφορετικές εκτέλεσης. Γενικά, το Reduce μπορεί να πραγματοποιηθεί σε πολλαπλά βήματα, όπου τα αποτελέσματα από το reduction του προηγούμενου βήματος, χρησιμοποιούνται σαν είσοδος για το επόμενο reduction, μέχρι να μείνει μόνο ένα τελικό αποτέλεσμα.
    - Λοιπές μέθοδοι όπως Scatter, για τον διαμοιρασμό των υπο-προβλημάτων, Gather για την συγκέντρωση τους, Ταξινόμηση, Αναζήτηση και άλλες.



Διάγραμμα 2: Χρονολογική εξέλιξη των γλωσσών προγραμματισμού των GPUs.

1.4.2 Περιορισμοί και προβλήματα στην χρήση των GPUs

Η επεξεργασία γενικού σκοπού στους επεξεργαστές γραφικών είναι μια σχετικά νέα τεχνολογία. Όπως αναφέρθηκε και προηγουμένως οι GPUs χρησιμοποιούνταν αρχικά μόνο για την επεξεργασία γραφικών. Καθώς η τεχνολογία εξελισσόταν, ο μεγάλος αριθμός πυρήνων των GPUs σχετικά με αυτόν τον CPUs οδήγησε στην ανάπτυξη επεξεργαστικών ικανοτήτων για τις GPUs, έτσι ώστε να μπορούν να επεξεργαστούν κάθε είδος δεδομένων. Παρότι όμως οι GPUs έχουν εκατοντάδες ή και χιλιάδες επεξεργαστές, καθένας από αυτούς τρέχει πολύ πιο αργά από ότι οι αντίστοιχοι πυρήνες μιας CPU και έχουν περιορισμένες δυνατότητες (ακόμα και αν είναι πλήρεις κατά Turing ώστε να μπορούν να

προγραμματιστούν για να τρέξουν ακριβώς ό,τι τρέχει μια CPU). Χαρακτηριστικά που απουσιάζουν από τις GPUs συμπεριλαμβάνουν τις διακοπές (interrupts) και την εικονική μνήμη (virtual memory).

Οι GPUs δεν μπορούν να χειριστούν αποδοτικά εφαρμογές που δεν παραλληλοποιούνται σε μεγάλο βαθμό. Πολλές φορές οι ανθρώπινοι πόροι που απαιτούνται για να μεταφραστεί μια υπάρχουσα βελτιστοποιημένη εφαρμογή στην παράλληλη έκδοση της δεν ανταποδίδουν από άποψης βελτίωσης της απόδοσης. Η διαδικασία σχεδιασμού, υλοποίησης και αποσφαλμάτωσης των παράλληλων εφαρμογών είναι συχνά πολύ επίπονες για τους προγραμματιστές.

Επιπροσθέτως, οι εταιρείες παραγωγής καρτών γραφικών, για λόγους ανταγωνισμού και προστασίας των πρωτότυπων τεχνολογιών τους, δεν δίνουν σε ελεύθερη πρόσβαση όλες τις πτυχές των τεχνολογιών που χρησιμοποιούν τα προϊόντα τους. Για παράδειγμα η τεχνολογία CUDA της NVIDIA, που χρησιμοποιήσαμε στο σύστημα που αναπτύξαμε για το σκοπό αυτής της διπλωματικής εργασίας, είναι ιδιόκτητη και κλειστού κώδικα, γεγονός που αποτελεί εμπόδιο για την πλήρη εκμετάλλευση των δυνατοτήτων των GPUs.

Συμπερασματικά, οι GPUs μπορεί να είναι αποδοτικές σαν συν-επεξεργαστές γενικού σκοπού μόνο αν προγραμματιστούν σωστά ώστε να τις εκμεταλλευτούμε πλήρως. Σε αντίθετη περίπτωση αποτελούν σπατάλη υπολογιστικών πόρων και μπορεί να αποβούν ακόμα και ζημιογόνες, λόγω της υψηλής κατανάλωσης ισχύος τους.

1.5 Η τεχνολογία CUDA

Το CUDA (Compute Unified Device Architecture) είναι μια πλατφόρμα παράλληλου προγραμματισμού και ένα προγραμματιστικό μοντέλο από μόνο του, που επιτρέπει μεγάλη αύξηση της υπολογιστικής απόδοσης, εκμεταλλευόμενο την ισχύ της GPU.

Δημιουργήθηκε το 2007 από την εταιρεία NVIDIA και από τότε έχει επεκταθεί μέσα από χιλιάδες εφαρμογών και επιστημονικών μελετών. Ενδεικτικά, κατά τη στιγμή συγγραφής της παρούσας εργασίας το CUDA βρίσκεται εγκατεστημένο σε πάνω από 300 εκατομμύρια συσκευές συμπεριλαμβανομένων από laptops και desktops μέχρι πανίσχυρα clusters και υπερυπολογιστές, ενώ χιλιάδες ερευνητές σε όλο τον κόσμο δουλεύουν πειραματικά πάνω σε αυτό.

1.5.1 Το μοντέλο Υπολογισμού

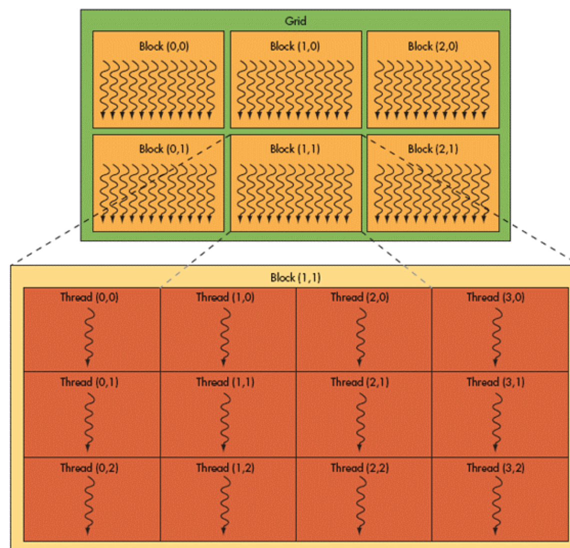
Στο CUDA , όπως και σε όλα τα αντίστοιχα frameworks για προγραμματισμό GPU , η ιδέα είναι ότι επιταχύνονται συγκεκριμένα κομμάτια του προβλήματος από τον επεξεργαστή γραφικών. Τα απαιτητικά σε ισχύ αυτά υπολογιστικά κομμάτια ονομάζονται πυρήνες (kernels) – όρος που προέρχεται περισσότερο από την Επεξεργασία Σήματος , παρά από τα Λειτουργικά Συστήματα. Έτσι ένα πρόγραμμα, μια συνάρτηση ή μια βιβλιοθήκη μπορεί να αποτελείται από έναν ή περισσότερους πυρήνες οι οποίοι προορίζονται για εκτέλεση στην GPU.

Οι πυρήνες αυτοί καλούνται από το κύριο πρόγραμμα που τρέχει στην CPU (ονομάζεται host) και εκτελούνται στην GPU (device). Γράφονται συνήθως σε μια κοινή γλώσσα προγραμματισμού , όπως την C , επεκταμένη με επιπλέον λέξεις κλειδιά, που εκφράζουν ευκολότερα την παραλληλία από το να χρησιμοποιούν συνεχώς βρόχους. Το CUDA ακολουθεί την Αρχιτεκτονική SIMD (Single Instruction Multiple Data), που περιγράφηκε πιο πάνω, δηλαδή το ίδιο σετ εντολών (πυρήνας) εκτελείται πάνω σε διαφορετικά δεδομένα.

Αφού οι πυρήνες μεταγλωττιστούν, αποτελούνται πλέον από πολλά νήματα (*threads*) (όσα έχει δηλώσει ο προγραμματιστής) τα οποία εκτελούν το ίδιο πρόγραμμα παράλληλα : κάθε νήμα εκτελεί ουσιαστικά μια επανάληψη του βρόχου. Για παράδειγμα σε έναν αλγόριθμο επεξεργασίας εικόνας κάθε thread επεξεργάζεται ένα pixel, ενώ συνολικά όλος ο πυρήνας επεξεργάζεται την εικόνα.

Πολλαπλά threads τώρα, οργανώνονται σε *thread blocks* που περιέχουν μέχρι 1536 από αυτά. Όλα τα threads ενός block θα τρέξουν τελικά σε ένα SM, γεγονός που τους επιτρέπει να συνεργάζονται και να μοιράζονται την κοινή μνήμη. Τα blocks χωρίζονται, όπως αναφέραμε και προηγουμένως σε ομάδες των 32 threads που ονομάζονται *warps,* και είναι η μονάδα αυτή που θα σταλεί τελικά για εκτέλεση σε ένα SM. Τέλος τα thread blocks ομαδοποιούνται περαιτέρω σε πλέγματα (grids) καθένα εκ των οποίων εκτελεί έναν μοναδικό πυρήνα.

Όλες οι παραπάνω υπολογιστικές μονάδες έχουν *αναγνωριστικά (identifiers)* που προσδιορίζουν τη σχέση τους με τον πυρήνα. Αυτά τα αναγνωριστικά χρησιμοποιούνται μέσα σε κάθε thread ως δείκτες για τα δεδομένα εισόδου και εξόδου , ώστε να προσδιορίσουν για το καθένα ποιόν ακριβώς υπολογισμό θα πραγματοποιήσουν.



Εικόνα 13: Η οργάνωση των υπολογιστικών στοιχείων στο CUDA.

Εκτέλεση

Κάθε δεδομένη χρονική στιγμή, όσον αφορά την αρχιτεκτονική Fermi, η συσκευή μπορεί να εκτελεί **μόνο μια** εφαρμογή, η οποία φυσικά μπορεί να αποτελείται από πολλούς υπολογιστικούς πυρήνες. Η παράλληλη εκτέλεση πολλών πυρήνων επιτρέπεται , με κάθε πυρήνα να ανατίθεται σε ένα ή περισσότερα SMs κάθε φορά. Έτσι μεγιστοποιείται η χρησιμοποίηση της συσκευής.

Η εναλλαγή (switching) από μια εφαρμογή σε μια άλλη για εκτέλεση είναι πολύ μικρή (της τάξης των 20 μs), διατηρώντας έτσι υψηλή χρησιμοποίηση της συσκευής και παράλληλα αυξάνοντας την παραλληλία. Το switching το αναλαμβάνει ειδικό υλικό στο chip γνωστό με την ονομασία *GigaThread Scheduler*. Ο χρονοδρομολογητής αυτός ελέγχει έως 1536 ενεργά threads σε κάθε SM κατανεμημένα σε έως και 16 πυρήνες.

Γλώσσες Προγραμματισμού

Το CUDA υποστηρίζει καταρχάς την γλώσσα C, όπου είναι γραμμένες και οι περισσότερες εφαρμογές. Για την C έχουν εισαχθεί και κάποιες επεκτάσεις που διευκολύνουν την κατανομή των δεδομένων που θα επεξεργαστεί ένας πυρήνας στις δομές που αναφέρθηκαν παραπάνω.

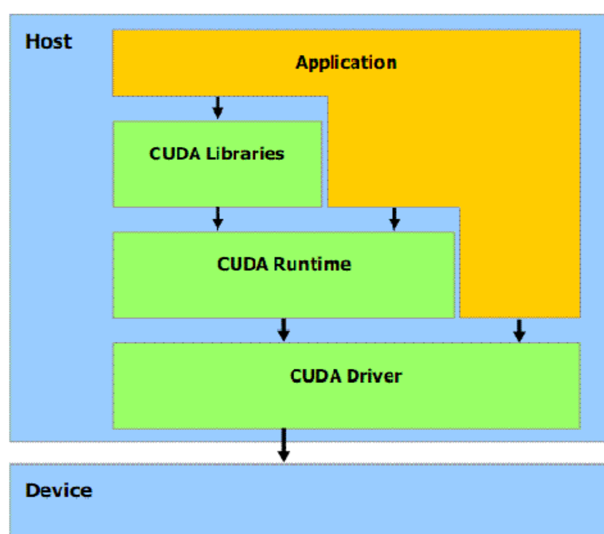
Εικόνα 14: Παράδειγμα συγγραφής προγράμματος σε CUDA C με τις επεκτάσεις.

Επίσης μπορούν να επιταχυνθούν προγράμματα γραμμένα σε FORTRAN, Java, Matlab και Python. Τέλος εκτός από το περιβάλλον ανάπτυξης του CUDA, υποστηρίζεται το πρότυπο OpenCL και το Direct Compute API της Microsoft.

Υλοποίηση

Όσον αφορά τη στοίβα λογισμικού του CUDA αυτή έχει ως εξής :

- To CUDA Driver API για τους χαμηλού επιπέδου χειρισμούς. Είναι κλειστού και η εταιρεία και μόνο έχει τη δυνατότητα επεξεργασίας του.
- To CUDA Runtime API για τον εύκολο προγραμματισμό των εφαρμογών σε CUDA. Είναι προφανώς πιο υψηλού επιπέδου από το Driver API και είναι αυτό στο οποίο τελικώς θα γράψει τον κώδικα του ο προγραμματιστής.
- Τις CUDA libraries που αποτελούν ένα σύνολο με ευρέως χρησιμοποιούμενες συναρτήσεις σε επιστημονικές εφαρμογές, υλοποιημένες και βελτιστοποιημένες για εκτέλεση στις GPUs. Παραδείγματα είναι η CUBLAS για τη Γραμμική Άλγεβρα και η CUFFT για τον μετασχηματισμό Fourier.
- To CUDA Toolkit που πέραν διάφορων χρήσιμων εργαλείων για την ανάπτυξη εφαρμογών σε CUDA, παρέχει και τον Compiler της NVIDIA nvcc που σε συνεργασία με τον gcc παράγει το τελικό εκτελέσιμο αρχείο, το οποίο θα εκτελεστεί τμηματικά στην GPU και την CPU.



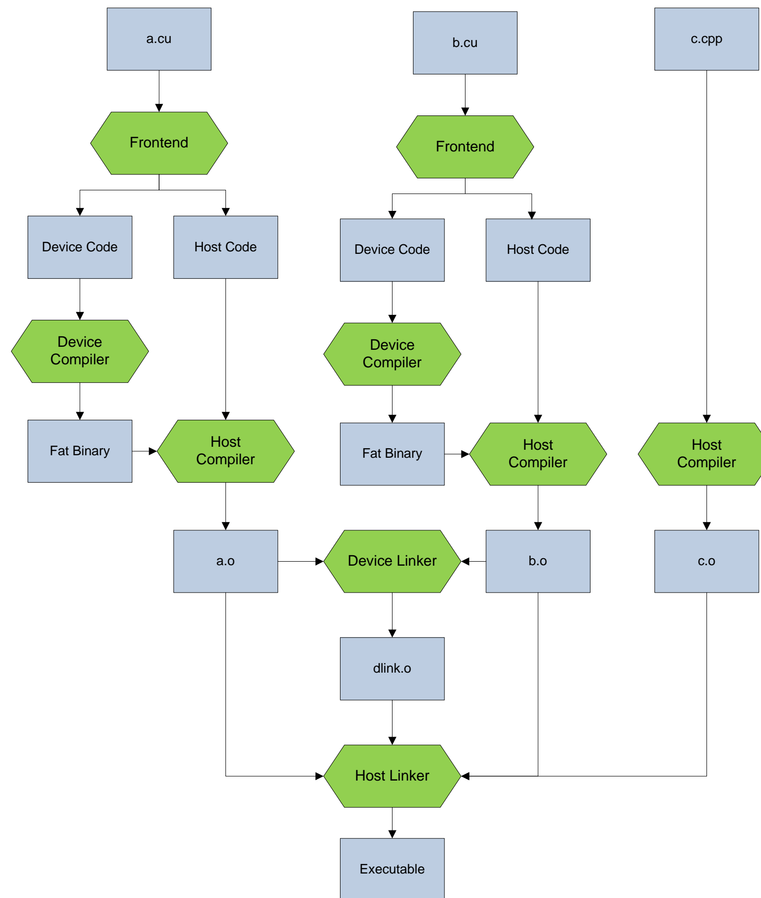
Εικόνα 15: Η στοίβα λογισμικού του CUDA.

1.5.2 Μεταγλώττιση και Εκτέλεση

Οι CUDA εφαρμογές αποτελούνται από μια μίξη συμβατικού κώδικα για CPU, γραμμένο σε C, C++, Fortran ή άλλες γλώσσες υψηλού επιπέδου με συναρτήσεις της GPU. Σύμφωνα με τη διαδικασία μεταγλώττισης, οι συναρτήσεις που πρόκειται να εκτελεστούν στην GPU διαχωρίζονται και μεταγλωττίζονται από τον ιδιόκτητο μεταγλωττιστή της NVIDIA (nvcc), ενώ οι υπόλοιπες συναρτήσεις μεταγλωττίζονται με τους κλασσικούς μεταγλωττιστές που είναι διαθέσιμοι στο κάθε σύστημα (π.χ. gcc). Έπειτα οι συναρτήσεις της GPU ενσωματώνονται σαν εικόνες στο αντικείμενο αρχείο που παράχθηκε. Τέλος, κατά τη διάρκεια της σύνδεσης (linking), συγκεκριμένες βιβλιοθήκες εκτέλεσης CUDA

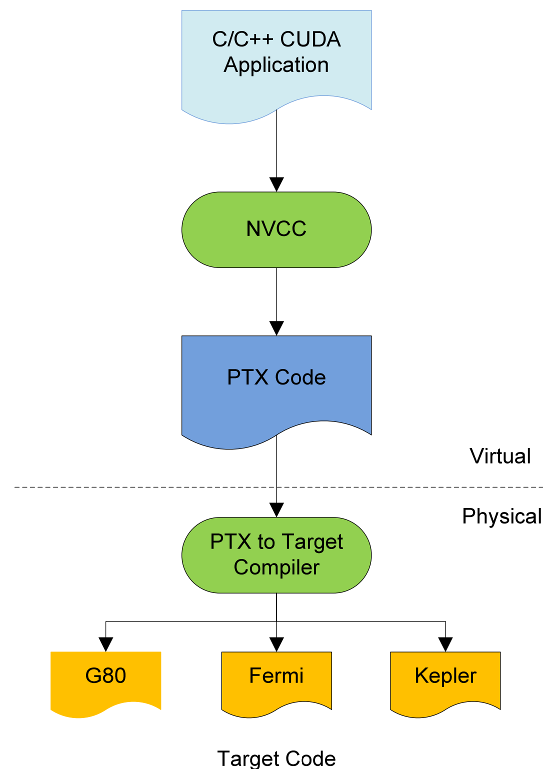
αναλαμβάνουν υποστήριξη για χειρισμό των GPUs, όπως δέσμευση χώρου μνήμης στη συσκευή επιταχυντή ή μεταφορά από τη μνήμη συσκευής στη μνήμη του συστήματος.

Σχηματικά η διαδικασία μεταγλώττισης και εκτέλεσης στην απλοποιημένη της μορφή φαίνεται στο παρακάτω παράδειγμα :



Εικόνα 16: Ενδεικτική πορεία μεταγλώττισης και σύνδεσης ενός CUDA προγράμματος. Θεωρούμε οτι ο κώδικας αποτελείται από 3 αρχεία πηγαίου κώδικα , 2 εκ των οποίων περιλαμβάνουν και συναρτήσεις της GPU (a.cu, b.cu).

Να σημειωθεί εδώ ότι όσον αφορά τον κώδικα που θα εκτελεστεί στη συσκευή, αφού ο nvcc τον διαχωρίσει από τον κώδικα της CPU, παράγει μια ενδιάμεση μορφή που ονομάζεται PTX Assembly (Parallel Thread eXecution) που είναι γενικός για όλες τις αρχιτεκτονικές και έπειτα, ανάλογα με την ακριβή πλατφόρμα που θα εκτελεστεί ο κώδικας, γίνεται ένα επιπλέον στάδιο μεταγλώττισης με στόχο την παραγωγή του τελικού εκτελέσιμου κώδικα της συσκευής. Ο μεταγλωττιστής nvcc δίνει την δυνατότητα στον προγραμματιστή να παράγει τα ενδιάμεσα .ptx αρχεία, τα οποία όμως έχουν νόημα σε χρήση μόνο στο CUDA Driver API.



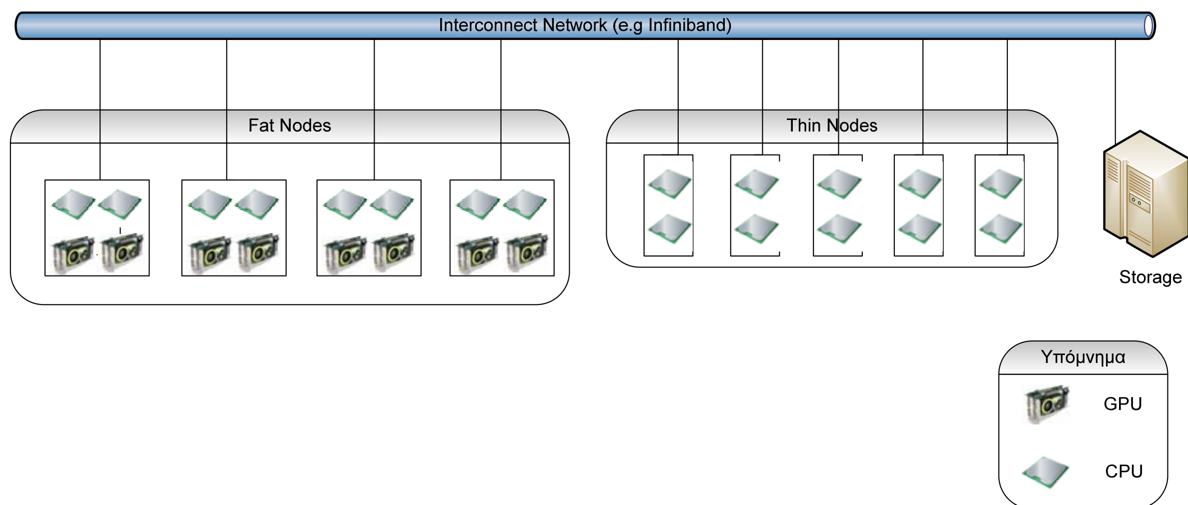
Εικόνα 17: Γενικότερο διάγραμμα των σταδίων της μεταγλώττισης ενός CUDA προγράμματος.

1.5.3 Η εναλλακτική της OpenCL

Η OpenCL (Open Computing Language) είναι μια πρόσφατα ανεπτυγμένη βιβλιοθήκη και σύνολο προτύπων που στοχεύουν τόσο στις GPUs, όσο και στις CPUs και δυνητικά και σε άλλους τύπους επιταχυντών. Για τη διατήρηση του προτύπου υπεύθυνο είναι το Khronos Group, το οποίο διατηρεί και το πρότυπο OpenGL για τα γραφικά. Σε αντίθεση με το CUDA, η OpenCL υλοποιείται μόνο ως μια βιβλιοθήκη. Στην πράξη αυτό εναποθέτει το φορτίο της συμπίεσης και της αποσυμπίεσης των παραμέτρων των πυρήνων, μαζί με άλλες παρόμοιες εργασίες στα χέρια του προγραμματιστή της εφαρμογής, ο οποίος πρέπει να υλοποιήσει όλα τα παραπάνω. Ένα άλλο αποτέλεσμα αυτής της προσέγγισης είναι ότι οι πυρήνες GPU δεν μεταγλωττίζονται μαζί με την υπόλοιπη εφαρμογή, αλλά κατά την εκτέλεση, από την ίδια την βιβλιοθήκη της OpenCL. Αυτό επιτυγχάνεται με την αποστολή του πηγαίου κώδικα του πυρήνα σαν ένα σύνολο από συμβολοσειρές (strings) στο κατάλληλο OpenCL API. Όταν οι πυρήνες μεταγλωττιστούν, η καλούσα εφαρμογή οφείλει να διαχειριστεί αυτούς τους πυρήνες. Στην πράξη, απαιτείται πολύ περισσότερος κώδικας από ότι σε μια αντίστοιχη CUDA εφαρμογή, αν και όλες αυτές οι λειτουργίες είναι σχετικά απλές στην διαχείριση τους. Κατά τα άλλα, οι βασικές αρχές είναι ίδιες με αυτές του προγραμματισμού σε CUDA.

1.6 GPUs σε clusters και Supercomputers

Μια συστοιχία από GPUs (GPU Cluster) είναι ουσιαστικά μια συστοιχία υπολογιστών όπου τουλάχιστον ένας κόμβος είναι εξοπλισμένος με μια GPU. Συγκεντρώνοντας την υπολογιστική ισχύ των μοντέρνων επεξεργαστών γραφικών μέσω του GPGPU , πετυχαίνουμε πολύ γρήγορους υπολογισμούς. Οι μεγάλης κλίμακας συστοιχίες GPU γίνονται ολοένα και πιο δημοφιλείς στην επιστημονική κοινότητα, αφού βρίσκουν εφαρμογή σε μεγάλο αριθμό προβλημάτων.



Εικόνα 18: Τυπικό παράδειγμα διάταξης ενός cluster με GPUs.

Η κατηγοριοποίηση των GPU clusters όσον αφορά το Hardware έχει ως εξής :

- Ομοιογενή (homogeneous) : Κάθε GPU είναι ακριβώς της ίδιας αρχιτεκτονικής κλάσης και μοντέλου (για παράδειγμα ένα cluster που περιέχει 100 Nvidia K20X, όλες με την ίδια ποσότητα μνήμης).
- Ετερογενή (heterogeneous) : Μπορεί να χρησιμοποιηθεί Hardware και από τους δύο κύριους κατασκευαστές GPUs (Nvidia και ATI). Ακόμα και αν χρησιμοποιούνται διαφορετικά μοντέλα του ίδιου κατασκευαστή (π.χ. Fermi και Kepler NVIDIA κάρτες), το cluster θεωρείται ετερογενές.

1.6.1 Χαρακτηριστικά των GPU Clusters

<u>Διασύνδεση</u>

Ο τύπος διασύνδεσης που χρησιμοποιείται εξαρτάται κυρίως από τον αριθμό των κόμβων της συστοιχίας. Οι κυριότεροι τύποι που χρησιμοποιούνται είναι τα : Gigabit Ethernet, Scalable Coherent Interface (SCI), Myrinet, Quadrics καθώς και το Infiniband. Στους υπερυπολογιστές (supercomputers) χρησιμοποιούνται συνήθως δίκτυα διασύνδεσης κατασκευασμένα κατόπιν

παραγγελίας και βελτιστοποιημένα κάθε φορά για το αντίστοιχο υλικό. Παρακάτω δίνεται μια σύντομη αναφορά στο Infiniband.

Το Infiniband [1] είναι μια τεχνολογία στρώματος δικτύου που δημιουργήθηκε με σκοπό να καλύψει το κενό που είχε αρχίσει να δημιουργείται μεταξύ των ολοένα και αυξανόμενων σε ισχύ επεξεργαστικών μονάδων και των συστημάτων εισόδου-εξόδου που αποτελούσαν σημείο συμφόρησης για τους εξυπηρετητές οι οποίοι έτρεχαν απαιτητικές εφαρμογές. Η αρχιτεκτονική του βασίζεται σε ένα σειριακό switched fabric, το οποίο, εκτός από το ότι ορίζει ταχύτητες μεταξύ 2.5 και 30 Gbits/sec, ξεπερνάει και τους περιορισμούς που έχουν να κάνουν με την επεκτασιμότητα, την κλιμακωσιμότητα και την ανοχή σε σφάλματα, που παρουσιάζει η κλασσική αρχιτεκτονική του κοινού διαύλου (shared bus). Υπεύθυνος για την συντήρηση του προτύπου είναι ο οργανισμός Infiniband Trade Association (ITA), που συγκροτείται από τους 7 πρωτοπόρους της βιομηχανίας των υπολογιστών : Dell, Compaq, Hewlett-Packard, IBM, Intel, Microsoft και Sun. Η πρώτη έκδοση των προδιαγραφών του Infiniband εκδόθηκε το 2000.

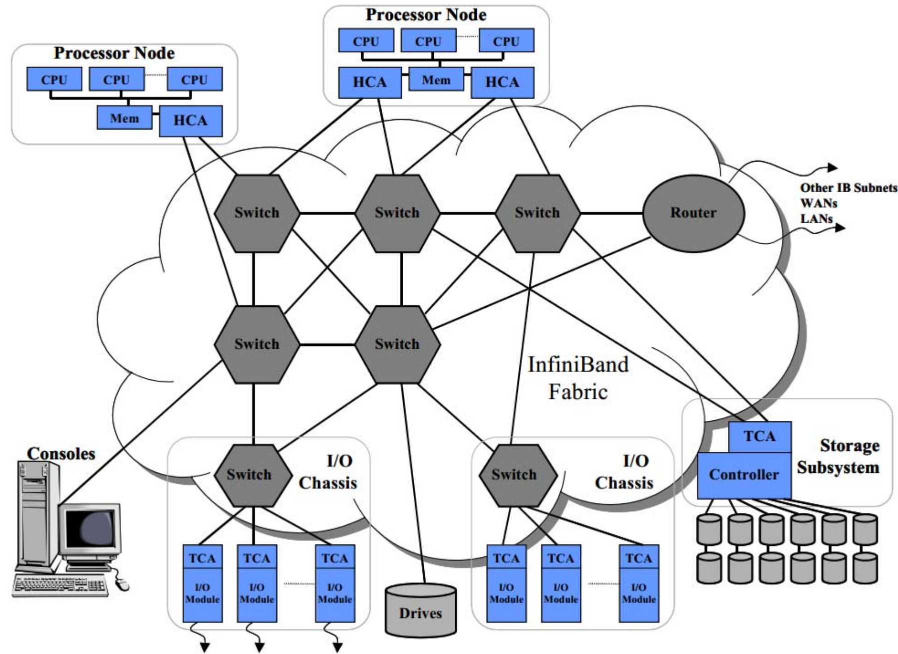
Η βασική διαφορά με την αρχιτεκτονική κοινού διαύλου είναι ότι στο Infiniband οι συνδέσεις μεταξύ των κόμβων (υπολογιστές, συστήματα RAID, συσκευές I/O) είναι point-to-point , πράγμα που επιτρέπει σε αυτούς να χρησιμοποιήσουν όλο το εύρος ζώνης της σύνδεσης, χωρίς να υπάρχει περίπτωση συμφόρησης στο δίαυλο. Παρακάτω παρουσιάζεται ένας πίνακας με τα βασικότερα πλεονεκτήματα του Infiniband σε σχέση με την αρχιτεκτονική κοινού διαύλου.

Χαρακτηριστικό	Infiniband Fabric	Δίαυλος
Τοπολογία	Με διακόπτες	Κοινού Διαύλου
Αριθμός pins	Μικρός	Μεγάλος
Αριθμός συσκευών	Πολλές	Λίγες
Μέγιστο μήκος σήματος	Χιλιόμετρα	Εκατοστά
Αξιοπιστία	Ναι	Όχι
Κλιμακώσιμο	Ναι	Όχι
Ανοχή σε σφάλματα	Ναι	Όχι

Πίνακας 1:Σύγκριση μεταξύ των αρχιτεκτονικών Infiniband Fabric και Shared Bus

Επίσης, ένα πολύ σημαντικό χαρακτηριστικό του Infiniband, που του επιτρέπει να πετυχαίνει πολύ υψηλές ταχύτητες είναι η ικανότητα του να μεταφέρει δεδομένα μεταξύ κατανεμημένων εφαρμογών σε διαφορετικούς κόμβους, χωρίς τη διαμεσολάβηση του επεξεργαστή. Χρησιμοποιεί κατευθείαν την χαμηλού επιπέδου υποδομή του δικτύου και μεταφέρει δεδομένα από και προς την απομακρυσμένη μνήμη χωρίς να τα αντιγράφει πρώτα στους buffers του λειτουργικού συστήματος. Η λειτουργία αυτή ονομάζεται RDMA (Remote Direct Memory Access).

Το Infiniband fabric, ο πυρήνας δηλαδή του δικτύου, μπορεί να αποτελείται από ένα switch ή από ένα ολόκληρο δίκτυο από switches. Όλες οι συνδέσεις τερματίζουν σε έναν προσαρμογέα καναλιού (channel adapter). Κάθε επεξεργαστής περιέχει έναν Host Channel Adapter (HCA) και κάθε συσκευή έναν Target Channel Adapter (TCA). Παρακάτω φαίνεται ένα παράδειγμα διασύνδεσης με Infiniband.



Εικόνα 19 : Παράδειγμα κατανεμημένου συστήματος διασυνδεδεμένου με Infiniband.

<u>Λογισμικό</u>

Τα βασικά συστατικά του λογισμικού που χρειάζονται σε κάθε κόμβο ενός GPU cluster είναι τα εξής :

1. Λειτουργικό σύστημα.
2. GPU Drivers για κάθε τύπο GPU που βρίσκεται στους αντίστοιχους κόμβους.
3. Κάποιο clustering API , όπως το MPI (Message Passing Interface).
4. Ένα σύστημα διαχείρισης πόρων, με σκοπό την αποδοτική κατανομή τους σε όλους τους χρήστες.

Να σημειωθεί εδώ ότι η χρήση επιταχυντών με GPUs στις συστοιχίες υπολογιστών ξεκίνησε τα τελευταία χρόνια και σε πολλές περιπτώσεις δεν έχουν μελετηθεί ακριβώς οι προϋποθέσεις για την πλήρη χρησιμοποίηση τους. Κάποια επιστημονικά άρθρα [2] έχουν προτείνει μερικές από αυτές. Αρχικά θα πρέπει οι GPUs τελευταίας γενιάς να εκμεταλλεύονται πλήρως το κανάλι διασύνδεσης με τον επεξεργαστή (σ.σ. τον δίαυλο PCI Express) καθώς και το δίκτυο διασύνδεσης μεταξύ των κόμβων, που προτιμάται να είναι Infiniband, ώστε να εξισωθεί η ταχύτητα μεταφοράς των δεδομένων μεταξύ των επεξεργαστών με την ταχύτητα μεταφοράς από τους επεξεργαστές στις κάρτες. Υπάρχουν, παρόλα αυτά και τεχνολογίες στους σύγχρονους επιταχυντές, που επιτρέπουν την μεταφορά δεδομένων μεταξύ των επιταχυντών , αλλά και μεταξύ των επιταχυντών και άλλων συσκευών όπως δίσκων SSD, χωρίς την διαμεσολάβηση του επεξεργαστή. Μια τέτοια τεχνολογία είναι το GPUDirect [3] της NVIDIA.

Τέλος, η μνήμη του κόμβου θα πρέπει να είναι σε μέγεθος τουλάχιστον όση αυτή στον επιταχυντή και μια ένα-προς-ένα αναλογία μεταξύ πυρήνων CPU και GPU είναι επιθυμητή

από πλευράς ανάπτυξης λογισμικού, καθώς διευκολύνει την ανάπτυξη κατανεμημένων εφαρμογών βασισμένων στο MPI.

1.6.2 Απαιτήσεις ισχύος

Οι απαιτήσεις ισχύος σε μια συστοιχία υπολογιστών μεγάλης κλίμακας είναι ένας από τους σημαντικότερους παράγοντες λειτουργίας, κυρίως λόγω του κόστους που αυτές συνεπάγονται. Ιδιαίτερα αν λάβουμε υπόψη το γεγονός ότι τα συστήματα αυτά λειτουργούν ακατάπαυστα, συμπεραίνουμε ότι η μείωση της συνολικής κατανάλωσης ισχύος είναι ένας σημαντικότατος στόχος.

Τα περισσότερα clusters, συμπεριλαμβανομένου και του διαθέσιμου για την ανάπτυξη του συστήματος μας, περιλαμβάνουν κυρίως τρείς τύπους κόμβων. Αρχικά, υπάρχουν οι λεγόμενοι "thin nodes", που αποτελούν απλά επεξεργαστικά συστήματα με CPU, RAM και πιθανώς και αποθηκευτικό χώρο. Προαιρετικά, μπορεί να είναι εγκατεστημένα κάποια "service nodes", τα οποία αναλαμβάνουν βοηθητικές λειτουργίες, όπως διαχείριση του δικτύου διασύνδεσης και δημιουργία αντιγράφων ασφαλείας. Τέλος, τα "fat nodes", που περιέχουν επιπλέον επιταχυντές όπως GPUs και συν-επεξεργαστές. Τα τελευταία είναι αρκετά ενεργοβόρα αλλά και ακριβά και συνήθως γίνεται ένας συμβιβασμός μεταξύ τιμής και απόδοσης κατά την εγκατάσταση τους. Ενδεικτικά, οι επιταχυντές που χρησιμοποιήσαμε για τις μετρήσεις μας (NVIDIA Tesla M2050) καταναλώνουν περίπου 255 W ισχύος η καθεμία, αν και η κατανάλωση ισχύος είναι γενικά εξαρτώμενη από το φορτίο και μπορεί να φτάσει σε πολύ υψηλότερα επίπεδα. Αν λοιπόν, λάβουμε υπόψη μας ότι ένας κόμβος χωρίς επιταχυντή καταναλώνει περί τα 500 W ισχύος, κατανοούμε ότι είναι άμεση η ανάγκη για μείωση του αριθμού των επιταχυντών σε συνδυασμό με την αποδοτικότερη χρησιμοποίηση τους.

Για το λόγο αυτό έχουν προταθεί πολλές λύσεις, που κυρίως υλοποιούν τεχνικές εικονικοποίησης (virtualization). Οι περισσότερες από αυτές χρησιμοποιούν λύσεις λογισμικού (software virtualization), επιτρέποντας σε πολλές εικονικές μηχανές να εκμεταλλεύονται τον ίδιο επιταχυντή. Σαν παραδείγματα θα μπορούσαμε να αναφέρουμε το ευρέως διαδεδομένο VMware, το ανοιχτού κώδικα Xen, καθώς και τα Virtual PC της Microsoft, VirtualBox της Oracle και το Kernel Based Virtual Machine (KVM). Ωστόσο, όλες αυτές οι λύσεις εικονικοποιούν ένα ολόκληρο σύστημα και δεν λύνουν το πρόβλημα που θέσαμε παραπάνω, καθώς οι επιταχυντές πρέπει να προσφέρονται ανεξάρτητα και απευθείας στην συστοιχία για χρήση. Από την άλλη έχει αναπτυχθεί και μια σειρά από λύσεις λογισμικού που μιμούνται την λειτουργία της GPU και δεν απαιτούν φυσική εγκατάσταση, που έχουν όμως το πολύ σοβαρό πρόβλημα των χαμηλών επιδόσεων.

1.6.3 Χρονοδρομολόγηση εργασιών

Υπάρχουν πολλοί παράγοντες που επηρεάζουν την γενική απόδοση ενός GPU cluster, αλλά ο σημαντικότερος είναι ίσως η σωστή χρονοδρομολόγηση και διαχείριση των εργασιών, με σκοπό την κατά το δυνατόν μεγαλύτερη εκμετάλλευση των GPUs. Δουλεύοντας πιο

“έξυπνα”, οι οργανισμοί και οι εταιρείες μπορούν να εξοικονομήσουν χρήματα και να αυξήσουν την παραγωγικότητα τους.

Για τη μέτρηση της απόδοσης του cluster συχνά λαμβάνονται υπόψη τεχνικές προδιαγραφές (όπως π.χ. ο αριθμός των πυρήνων) καθώς και αποτελέσματα από προγράμματα αξιολόγησης (benchmarks) , όπως για παράδειγμα το πακέτο LINPACK. Παρόλα αυτά η πραγματική απόδοση θα πρέπει να μετράται με βάση φορτία από πραγματικές εφαρμογές. Εκεί τον πιο σημαντικό ρόλο τον παίζει η σωστή χρονοδρομολόγηση και όχι η αύξηση του μεγέθους του cluster. Τις τελευταίες δυο δεκαετίες τεχνικές όπως το back-fill scheduling [4] , το checkpoint/restart [5] και το session scheduling της IBM [6] έχουν επιτρέψει σε πολλά HPC clusters σήμερα να αξιοποιούν σχεδόν το σύνολο των πόρων τους.

Οι περισσότεροι σύγχρονοι χρονοδρομολογητές αντιμετωπίζουν τους διαθέσιμους πόρους (μνήμη, πυρήνες και εύρος ζώνης) σαν καταναλώσιμες οντότητες. Η χρονοδρομολόγηση με βάση τους πόρους όχι μόνο αυξάνει το ποσοστό αξιοποίησης, αλλά και βελτιώνει την συνολική απόδοση, καθώς οι εφαρμογές έχουν λιγότερες πιθανότητες να αποτύχουν λόγω περιορισμένων πόρων. Μια τέτοια πολιτική θα μπορούσε να ακολουθηθεί και για τις GPUs, να τις αντιμετωπίζουμε δηλαδή ως έναν καταναλώσιμο πόρο. Ωστόσο, αυτή η όψη είναι υπεραπλουστευμένη και αγνοεί κάποια σημαντικά χαρακτηριστικά των GPUs.

Σε αντίθεση λοιπόν με την CPU και τη μνήμη, που είναι αφηρημένοι πόροι και διαχειρίζονται από τα λειτουργικά συστήματα, οι GPUs πρέπει να διαχειρίζονται ως ξεχωριστές οντότητες. Ο χρονοδρομολογητής πρέπει να είναι ενήμερος για την κατάσταση και το context κάθε GPU. Η αντιμετώπιση της GPU ως έναν απλό δυαδικό πόρο σημαίνει ότι μόνο ένα φορτίο GPU μπορεί να εκτελείται κάθε χρονική στιγμή σε αυτήν. Αυτό περιορίζει τον ταυτόχρονο αριθμό GPU εφαρμογών που μπορούν να τρέξουν, στον αριθμό των διαθέσιμων GPUs στο cluster. Από την άλλη βέβαια τα σύγχρονα frameworks για GPU Computing , όπως το CUDA επιτρέπουν στις εφαρμογές να χρησιμοποιήσουν την GPU σε δυο διαφορετικές λειτουργίες : την SHARED , όπου αναγκάζει την εφαρμογή να ελευθερώσει την GPU όσο δεν την χρησιμοποιεί, έτσι ώστε άλλες εφαρμογές να μπορούν να τρέξουν και την EXCLUSIVE, η οποία δεσμεύει την GPU καθ’όλη τη διάρκεια εκτέλεσης της εφαρμογής.

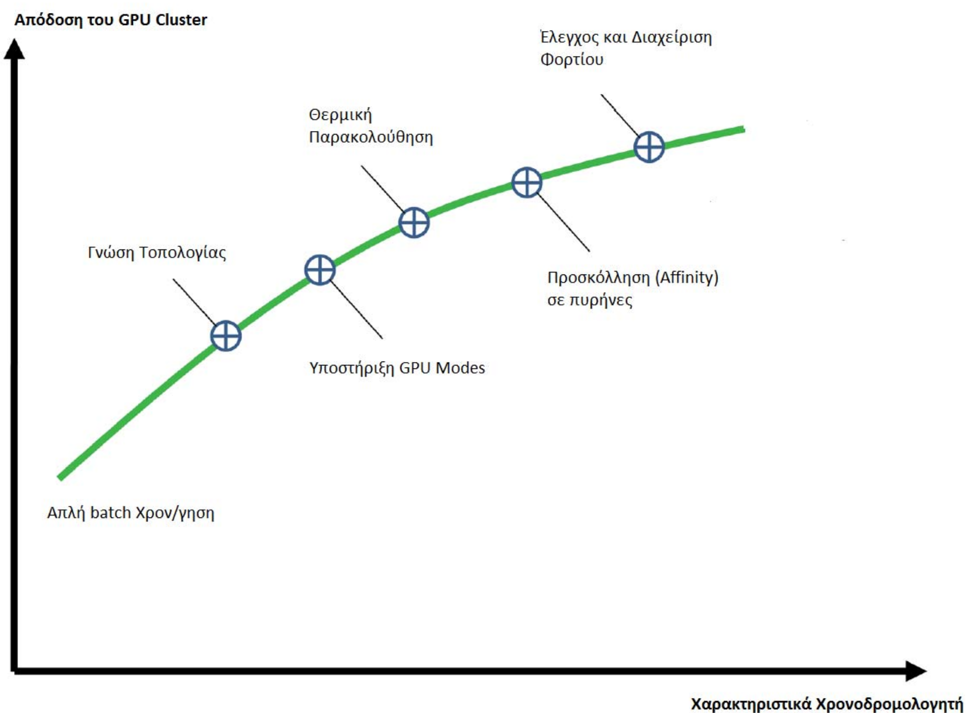
Τέλος, διαφορετικές GPU εφαρμογές μπορούν να παρουσιάσουν διαφορετικά χαρακτηριστικά όσον αφορά τον τρόπο που χρησιμοποιούν την CPU και την GPU. Για αλγορίθμους που είναι αρκετά απαιτητικοί σε πόρους GPU, για παράδειγμα, άλλα απαιτούν μόνο έναν πυρήνα CPU, ο χρονοδρομολογητής θα πρέπει να έχει την ευελιξία να μπορεί να το τοποθετήσει άλλα μη-GPU φορτία στην ίδια CPU, ώστε να αυξήσει την χρησιμοποίηση.

Ένα άλλο χαρακτηριστικό που θα πρέπει να διαθέτει ένας σύγχρονος χρονοδρομολογητής είναι η γνώση σχετικά με την τοπολογία των GPUs στην συστοιχία. Έτσι, αν κάποιος πελάτης ζητήσει να εκτελέσει ένα πρόγραμμα CUDA π.χ. το οποίο εκτείνεται σε περισσότερες από μια GPUs, ο χρονοδρομολογητής θα πρέπει να είναι αρκετά ευέλικτος ώστε να το στείλει για εκτέλεση σε “κοντινούς” για το δίκτυο κόμβους και αν είναι δυνατόν σε GPUs στους κόμβους, ώστε να ελαχιστοποιηθεί το κόστος επικοινωνίας.

Επίσης ένα σημαντικό χαρακτηριστικό είναι η διαχείριση των εξαιρέσεων. Οι σύγχρονες κάρτες γραφικών διαθέτουν σημαντική ποσότητα ενσωματωμένης μνήμης και τα σφάλματα μεταφοράς δεν μπορούν εύκολα και αυτόματα να διορθωθούν. Σημειώνεται εδώ ότι οι περισσότερες από τις μνήμες που χρησιμοποιούνται υποστηρίζουν ECC (Error-Correcting Codes), καθώς τα υπό εκτέλεση προγράμματα απαιτούν μεγάλο ποσοστό ακρίβειας, αφού χειρίζονται επιστημονικά δεδομένα. Ο χρονοδρομολογητής λοιπόν θα πρέπει να παρακολουθεί τη διαδικασία του ελέγχου σφαλμάτων και να αποφεύγει να στέλνει εργασίες σε GPUs που παρουσιάζουν υπερβολικό αριθμό λαθών.

Ένα τελευταίο στοιχείο των χρονοδρομολογητών που μπορεί να αυξήσει την απόδοση της συστοιχίας είναι η παρακολούθηση της κατανάλωσης ισχύος και της θερμοκρασίας των GPUs. Σύμφωνα με τις τελευταίες τάσεις στις εγκαταστάσεις GPUs στις συστοιχίες, τοποθετούνται έως και 4 σε έναν μόνο κόμβο. Με περίπου 1200 W κατανάλωση ισχύος και 12 ανεμιστήρες ψύξης, οι θερμοκρασίες που αναπτύσσονται σε αυτά τα συστήματα είναι πολύ υψηλές. Ο χρονοδρομολογητής σε ένα τέτοιο περιβάλλον πρέπει σαφώς να λαμβάνει υπόψη του μετρικές όπως η θερμοκρασία της GPU και η ταχύτητα περιστροφής των ανεμιστήρων με στόχο να αποφευχθεί η υπερφόρτωση συστημάτων που έχουν ήδη αυξημένη θερμοκρασία. Όταν τα συστήματα τερματίζουν την λειτουργία τους λόγω υπερφόρτωσης, σπαταλούνται υπολογιστικοί πόροι της συστοιχίας, αφού οι εκτελούμενες εργασίες πρέπει να μεταφερθούν σε άλλους υγιείς κόμβους και να επανεκκινηθούν.

Όλα τα παραπάνω χαρακτηριστικά οδηγούν τελικά στην αύξηση της απόδοσης της συστοιχίας, όπως φαίνεται και στο παρακάτω διάγραμμα.



Διάγραμμα 3: Απόδοση των GPU clusters με την προσθήκη διαφόρων χαρακτηριστικών στον χρονοδρομολογητή.

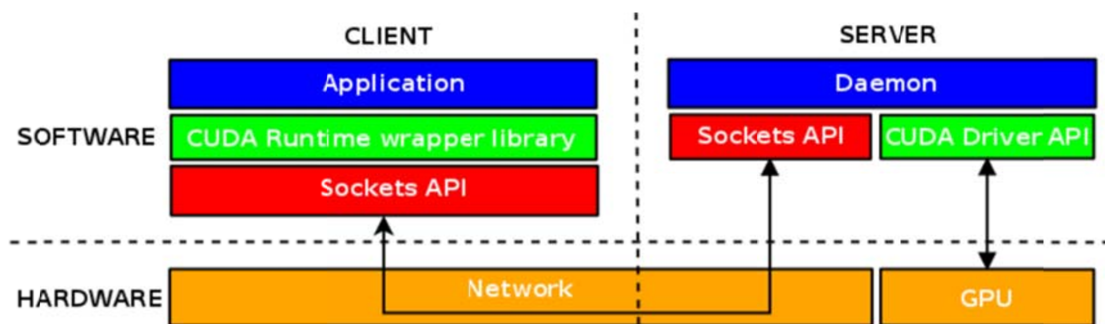
Κεφάλαιο 2

rCUDA : Σύστημα απομακρυσμένης εκτέλεσης CUDA

Το rCUDA [7] είναι ένα σύστημα που επιτρέπει στο CUDA να εκτελείται απομακρυσμένα πάνω από ένα κοινό δίκτυο. Δίνει λοιπόν τη δυνατότητα σε κόμβους μιας συστοιχίας να εκτελούν κώδικα CUDA, ακόμα και αν δεν διαθέτουν επιταχυντή γραφικών εγκατεστημένο. Έχει επίσης μελετηθεί [8] και μια δεύτερη χρήση του, όπου αναλαμβάνει τον διαμοιρασμό μίας GPU που βρίσκεται σε έναν κόμβο, στις διάφορες εικονικές μηχανές που τρέχουν πάνω σε αυτόν.

2.1 Αρχιτεκτονική

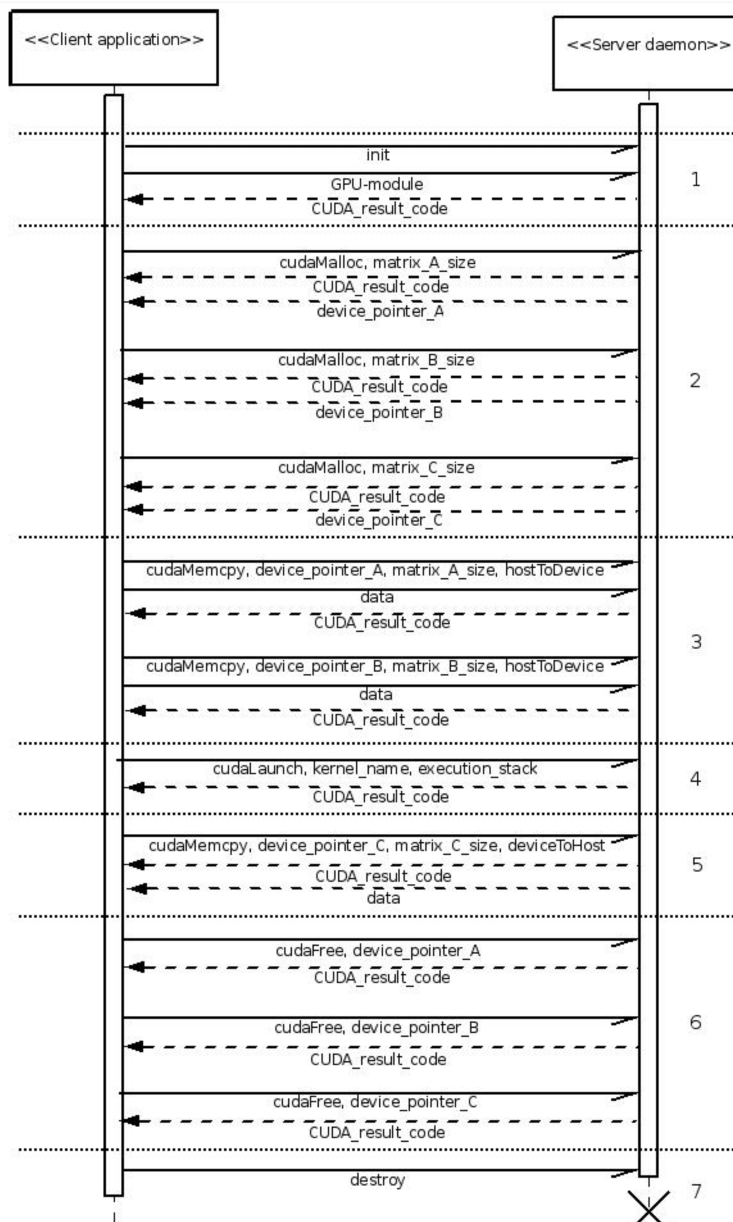
Το rCUDA χρησιμοποιεί την αρχιτεκτονική πελάτη – εξυπηρετητή. Πελάτες είναι ουσιαστικά όλοι οι κόμβοι της συστοιχίας, ενώ εξυπηρετητές είναι οι κόμβοι που έχουν εγκατεστημένους επιταχυντές γραφικών. Οι πελάτες, αντί να καλούν το υψηλού επιπέδου CUDA Runtime API, καλούν ένα wrapper αυτού το οποίο αναλαμβάνει να προωθήσει τις κλήσεις CUDA μέσω του δικτύου σε έναν εξυπηρετητή που “ακούει” σε συγκεκριμένη θύρα. Ο εξυπηρετητής αυτός αναλαμβάνει να εκτελέσει τον κώδικα χρησιμοποιώντας το πραγματικό Runtime CUDA API και επιστρέφει τα αποτελέσματα.



Εικόνα 20: Η αρχιτεκτονική του rCUDA.

Πιο συγκεκριμένα, για κάθε κλήση CUDA του πελάτη αποστέλλεται ένα μήνυμα αίτησης για εκτέλεση στον αρμόδιο εξυπηρετητή. Κάθε τέτοιο μήνυμα ξεκινάει με ένα αναγνωριστικό 32 bit, που προσδιορίζει την συνάρτηση που πρέπει να κληθεί, ενώ το υπόλοιπο κομμάτι του εξαρτάται από την συνάρτηση και προσδιορίζει τις συγκεκριμένες παραμέτρους για κάθε κλήση. Ο εξυπηρετητής πάντα στέλνει πίσω έναν κωδικό ελέγχου (επίσης 32 bit) και πιθανόν περισσότερα δεδομένα, αναλόγως την καλούμενη συνάρτηση.

Η διαδικασία εκτέλεσης ενός πυρήνα απαιτεί τις επόμενες φάσεις, που φαίνονται στην εικόνα 21 , μέσω του παραδείγματος του πολλαπλασιασμού πινάκων C = A·B.

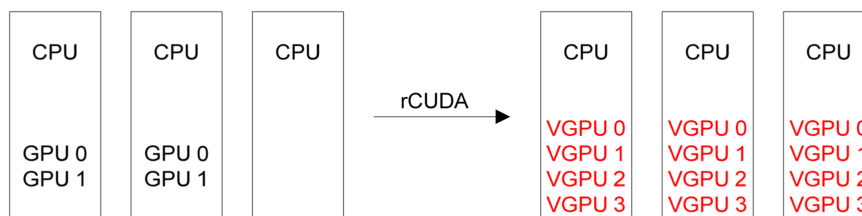


Εικόνα 21: Ακολουθιακό διάγραμμα των επικοινωνιών του rCUDA κατα τη διάρκεια εκτέλεσης ενός πολλαπλασιασμού πινάκων C=A X B.

1) **Στάδιο Αρχικοποίησης :** Ο πελάτης εγκαθιστά την σύνδεση με τον απομακρυσμένο εξυπηρετητή, εντοπίζει και στέλνει την κατάλληλη μονάδα (module) για αίτηση υπηρεσιών από την GPU, που αποτελείται από τον κώδικα (kernel code) που πρέπει να εκτελεστεί μαζί με άλλες σχετικές πληροφορίες, όπως στατικές μεταβλητές.
2) **Ανάθεση μνήμης :** Ο πελάτης ζητάει ανάθεση μνήμης στην GPU για τα δεδομένα που θα χειριστεί ο πυρήνας.
3) **Μεταφορά δεδομένων εισόδου :** Αν υπάρχουν επιπλέον δεδομένα στην είσοδο του προγράμματος πελάτη, πρέπει να μεταφερθούν και αυτά στον εξυπηρετητή.
4) **Εκτέλεση πυρήνα :** Απομακρυσμένη εκτέλεση του κώδικα πυρήνα.
5) **Μεταφορά δεδομένων εξόδου :** Αφού τερματίσει η εκτέλεση του πυρήνα, τα δεδομένα εξόδου πρέπει να μεταφερθούν πίσω στον πελάτη.
6) **Απελευθέρωση μνήμης :** Η δεσμευμένη από την GPU μνήμη απελευθερώνεται.
7) **Στάδιο τερματισμού :** Η εφαρμογή πελάτη κλείνει το socket επικοινωνίας με τον εξυπηρετητή. Ο daemon που τρέχει στον εξυπηρετητή σταματάει να εξυπηρετεί την συγκεκριμένη εκτέλεση και απελευθερώνει όλους τους δεσμευμένους πόρους.

Σε αυτό το σημείο αξίζει να σημειώσουμε ότι για κάθε απομακρυσμένη εκτέλεση ο εξυπηρετητής δημιουργεί μια ξεχωριστή διεργασία, για να εκτελέσει όλες τις κλήσεις από μια συγκεκριμένη απομακρυσμένη εφαρμογή σε ένα μόνο GPU context. Έτσι επιτυγχάνεται πολυπλεξία των επεξεργαστών γραφικών, καθώς δημιουργούνται πολλά ξεχωριστά contexts και ο χρονοδρομολογητής του κάθε επιταχυντή αναλαμβάνει την ταυτόχρονη εκτέλεση τους. Με τη δημιουργία ξεχωριστών διεργασιών εξασφαλίζεται επίσης και η επιβίωση των υπόλοιπων εξυπηρετητών σε περίπτωση αποτυχίας μερικών (π.χ. λόγω εσφαλμένης κλήσης CUDA).

Το αποτέλεσμα είναι ότι πλέον δίνεται η δυνατότητα σε διαφορετικούς κόμβους να τρέχουν ταυτόχρονα CUDA εφαρμογές εκμεταλλευόμενοι το σύνολο των επιταχυντών που υπάρχουν στο σύστημα. Επίσης, επιτρέπει σε ένα CUDA πρόγραμμα να χρησιμοποιήσει το σύνολο των GPUs που υπάρχουν στο cluster , δημιουργώντας εικονικές GPUs στον πελάτη. Αυτό φαίνεται γραφικά στο παρακάτω διάγραμμα.



Εικόνα 22: Δημιουργία εικονικών GPUs από το rCUDA. Ο 3ος κόμβος δεν διαθέτει καμία φυσική GPU, παρόλα αυτά μπορεί να χρησιμοποιήσει έως και 4 στο CUDA πρόγραμμά του.

2.2 Υλοποίηση – Τρόπος Εκτέλεσης

Το rCUDA είναι ένα λογισμικό κλειστού κώδικα, με αποτέλεσμα να μην είναι γνωστός ο ακριβής τρόπος υλοποίησης του. Επιστημονικά άρθρα των δημιουργών του όμως αποκαλύπτουν κάποια στοιχεία της αρχιτεκτονικής του. Στο σύστημα μας χρησιμοποιείται ως "μαύρο κουτί" για την διμερή επικοινωνία μεταξύ του πελάτη και του εξυπηρετητή που διαθέτει GPU.

Η τελευταία έκδοση του rCUDA κατά τη στιγμή της συγγραφής αυτής της εργασίας (4.0) υλοποιεί όλες τις συναρτήσεις του CUDA Runtime API 5.0, εκτός από αυτές που σχετίζονται με την επεξεργασία γραφικών, πράγμα το οποίο δεν καθιστά περιορισμό, αφού οι εφαρμογές GPGPU, δεν χρησιμοποιούν σχεδόν ποτέ γραφικά αυτές καθ'αυτές. Το rCUDA προορίζεται για το λειτουργικό σύστημα Linux μόνο (32 και 64 bit αρχιτεκτονικές).

Το πακέτο του rCUDA αποτελείται από 3 συστατικά : Ένα εκτελέσιμο που τρέχει στον εξυπηρετητή και προσφέρει GPU υπηρεσίες (*rCUDAd*), μια βιβλιοθήκη πελάτη που προωθεί τις κλήσεις CUDA στον εξυπηρετητή (*libcudart.so)* , καθώς και βιβλιοθήκες για την επικοινωνία μεταξύ πελάτη-εξυπηρετητή τόσο για συνδέσεις βασισμένες στο TCP (*rCUDAcommTCP.so),* όσο και για το Infiniband (*rCUDAcommIB.so).*

Όσον αφορά τον εξυπηρετητή, τρέχει τον δαίμονα *rCUDAd* , ο οποίος αρχικοποιεί όλες τις GPUs και περιμένει να "ακούσει" για αιτήσεις από τους πελάτες σε συγκεκριμένη θύρα (η προεπιλογή είναι η θύρα 8308). Αν χρησιμοποιείται δίκτυο διασύνδεσης Infiniband, πρέπει πρώτα να θέσουμε τη μεταβλητή περιβάλλοντος *RCUDAPROTO=IB* , για να φορτωθεί η αντίστοιχη βιβλιοθήκη επικοινωνίας. Στις επιλογές εκτέλεσης του rCUDAd περιλαμβάνονται:

- ✓ Εκτέλεση σε συγκεκριμένες συσκευές του κόμβου μόνο, αν διατίθενται πάνω από μια.
- ✓ Επιλογή για διαδραστική λειτουργία (αντί για daemon).
- ✓ Επιλογή μέγιστου ταυτόχρονου αριθμού servers (υπενθυμίζουμε ότι για κάθε CUDA πρόγραμμα που τρέχει δημιουργείται ξεχωριστή διεργασία).
- ✓ Επιλογή θύρας επικοινωνίας.

Από την πλευρά του πελάτη, αυτό που απαιτείται είναι να φορτώνει τη βιβλιοθήκη *libcudart.so.5* κάθε φορά που θέλει να εκτελέσει ένα CUDA πρόγραμμα. Να τονιστεί πάλι ότι δεν πρόκειται για την αυθεντική βιβλιοθήκη του CUDA Runtime API, που παρέχει η NVIDIA, άλλα ένα σύνολο από wrappers για τις συναρτήσεις της που απλά προωθεί κατάλληλα τις CUDA κλήσεις. Η "πραγματική" βιβλιοθήκη φορτώνεται στους εξυπηρετητές που διαθέτουν GPUs και τρέχουν το rCUDAd. Οι μεταβλητές περιβάλλοντος που προσδιορίζουν την επικοινωνία είναι οι εξής :

- *RCUDA_DEVICE_COUNT :* Προσδιορίζει πόσες GPUS «βλέπει» ο πελάτης συνολικά μέσα στο cluster. (π.χ. *RCUDA_DEVICE_COUNT=6).* Αυτές είναι διαθέσιμες στον προγραμματιστή για την εφαρμογή του (π.χ. με *cudaSetDevice()* μπορεί να τις χειριστεί χωριστά).
- *RCUDA_DEVICE_X :* Προσδιορίζει τη θέση της GPU X μέσα στο cluster. Ο προσδιορισμός γίνεται μέσω της διεύθυνσης IP. Για παράδειγμα αν ο πελάτης «βλέπει» 3 GPUs, δύο εκ των οποίων βρίσκονται στον κόμβο με διεύθυνση IP

192.168.0.1 και η τρίτη στον 192.168.0.2 πρέπει να τεθούν : *RCUDA_DEVICE_0=192.168.0.1*
RCUDA_DEVICE_1=192.168.0.1:1
RCUDA_DEVICE_2=192.168.0.2

- *RCUDAPROTO=IB :* Τίθεται , όπως και στον server όταν χρησιμοποιείται Infiniband μόνο.

Παρακάτω παρατίθενται ένα παραδείγματα εκτέλεσης του rCUDA :

rCUDA Server (IP: 147.104.4.173) 1 GPU Available	**rCUDA Client**
$ **./rCUDAd -v -i** rCUDAd v4.0.1 rCUDAd[21241]: Using rCUDAcommTCP.so communications library. rCUDAd[21241]: Server daemon succesfully started. rCUDAd[21244]: Trying device 0... rCUDAd[21244]: CUDA initialized on device 0. rCUDAd[21244]: Connection established with 147.102.4.148. rCUDAd[21284]: Trying device 0... rCUDAd[21244]: '__cudaRegisterFatBinary' received. rCUDAd[21244]: 'cudaGetDeviceProperties' received. rCUDAd[21244]: 'cudaEventCreate(WithFlags)' received. rCUDAd[21244]: 'cudaEventCreate(WithFlags)' received. rCUDAd[21244]: 'cudaHostAlloc' received. rCUDAd[21284]: CUDA initialized on device 0. rCUDAd[21244]: 'cudaMalloc' received. rCUDAd[21244]: 'cudaEventRecord' received. rCUDAd[21244]: 'cudaMemcpyAsync' received. rCUDAd[21244]: 'cudaMemcpyAsync' received. rCUDAd[21244]: 'cudaMemcpyAsync' received. rCUDAd[21244]: 'cudaMemcpyAsync' received. rCUDAd[21244]: 'cudaMemcpyAsync' received. rCUDAd[21244]: 'cudaMemcpyAsync' received. rCUDAd[21244]: 'cudaMemcpyAsync' received. rCUDAd[21244]: 'cudaMemcpyAsync' received. rCUDAd[21244]: 'cudaMemcpyAsync' received. rCUDAd[21244]: 'cudaMemcpyAsync' received. rCUDAd[21244]: 'cudaEventRecord' received. rCUDAd[21244]: 'cudaDeviceSynchronize' received. rCUDAd[21244]: 'cudaEventElapsedTime' received. rCUDAd[21244]: 'cudaEventDestroy' received. rCUDAd[21244]: 'cudaEventDestroy' received. rCUDAd[21244]: 'cudaFreeHost' received. rCUDAd[21244]: 'cudaFree' received. rCUDAd[21244]: Remote application finished.	**$ export LD_LIBRARY_PATH=~/rcuda/framework/rCUDAl/:$LD_LIBRARY_PATH** **$ export RCUDA_DEVICE_COUNT=1** **$ export RCUDA_DEVICE_0=147.102.4.173** **$ ./bandwidthTest** [CUDA Bandwidth Test] - Starting... Running on... Device 0: Tesla M2050 Quick Mode Host to Device Bandwidth, 1 Device(s) PINNED Memory Transfers Transfer Size (Bytes) Bandwidth(MB/s) 33554432 111.8 Device to Host Bandwidth, 1 Device(s) PINNED Memory Transfers Transfer Size (Bytes) Bandwidth(MB/s) 33554432 113.8 Device to Device Bandwidth, 1 Device(s) PINNED Memory Transfers Transfer Size (Bytes) Bandwidth(MB/s) 33554432 104864.7 Result = PASS

2.3 Αξιολόγηση rCUDA

Για να αποδειχθεί η αποτελεσματικότητα του rCUDA με σκοπό να ενσωματωθεί στο τελικό μας σύστημα, διεξήχθη μια σειρά από μετρήσεις στα συστήματα του εργαστηρίου. Επιλέχθηκαν benchmarks από την επίσημη ιστοσελίδα του CUDA και συγκρίθηκε ο χρόνος εκτέλεσης χρησιμοποιώντας την κλασσική ροή εκτέλεσης του CUDA με τον χρόνο εκτέλεσης χρησιμοποιώντας το rCUDA για να τρέξουμε απομακρυσμένα τις εντολές.

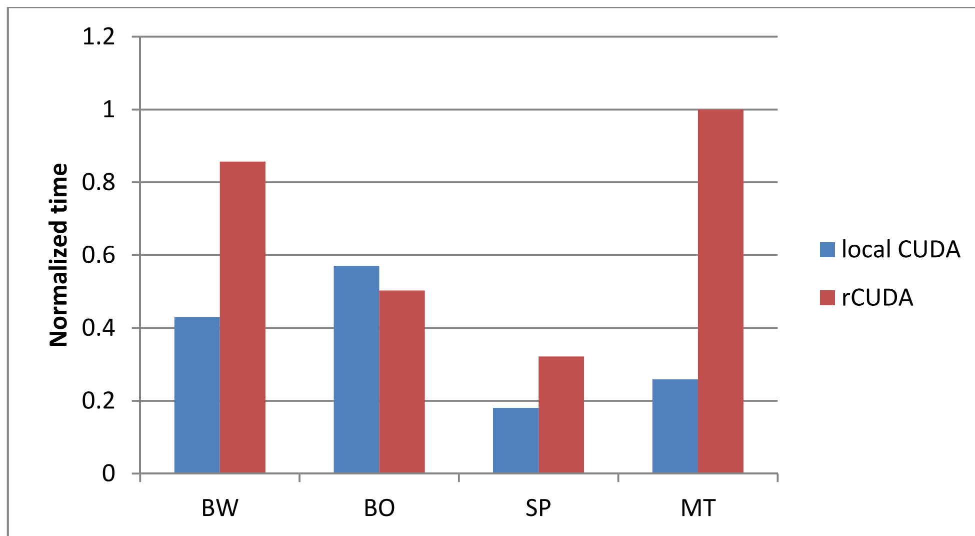
Τα πειράματα εκτελέστηκαν στη συστοιχία υπολογιστών του εργαστηρίου CSLab (Computing Systems Laboratory) του Εθνικού Μετσόβιου Πολυτεχνείου. Η συστοιχία αυτή αποτελείται από 39 κόμβους διαφόρων χαρακτηριστικών. Για τη συγκεκριμένη μέτρηση χρησιμοποιήθηκαν μόνο 2 από αυτούς. Ο rCUDA server ήταν ένας κόμβος με δύο Intel Xeon X5650, 48GB RAM και μια κάρτα NVIDIA Tesla M2050, τεχνολογίας Fermi με 448 πυρήνες CUDA και 3 GB μνήμης, ενώ ο client διέθετε 2 επεξεργαστές Intel Xeon E5335, 8 GB μνήμης RAM και κανέναν επιταχυντή.

Τα benchmarks επιλέχθηκαν από τη σουίτα των δοκιμαστικών προγραμμάτων που δίνει στη δημοσιότητα η NVIDIA και ήταν τα εξής :

- Bandwidth Test (BT) : Το απλό αυτό πρόγραμμα μετράει την ταχύτητα μεταφοράς δεδομένων μεταξύ των GPUs και μεταξύ GPU και CPU.
- Binomial Options (BO) : Πρόκειται για ένα δοκιμαστικό πρόγραμμα από το πεδίο της Οικονομίας. Επιλέχθηκε καθώς είναι ιδιαίτερα απαιτητικό σε υπολογισμούς.
- Scalar Product (SP) : Υπολογίζει το γινόμενο δοσμένων διανυσμάτων.
- Matrix Transpose (MT) : Πρόκειται για το γνωστό πρόβλημα της αντιμετάθεσης πινάκων. Διαφορετικές υλοποιήσεις που αναδεικνύουν τη σημασία του σωστού προγραμματισμού των GPUs περιλαμβάνονται σε αυτό το πακέτο.

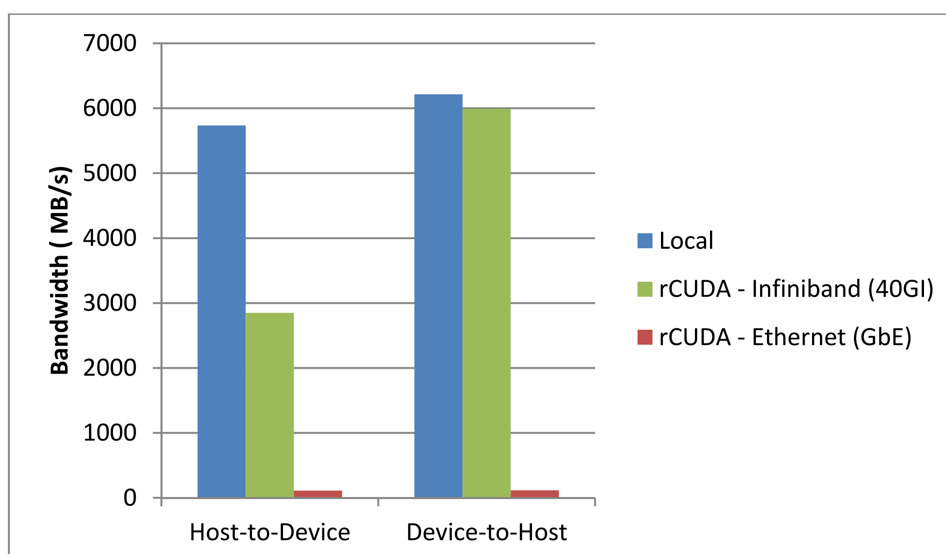
Πρέπει να σημειωθεί εδώ ότι κατά την πρώτη εκτέλεση των πειραμάτων τα αποτελέσματα έδειξαν ότι το rCUDA σχεδόν σε όλες τις περιπτώσεις ήταν πιο γρήγορο, ένα μη αναμενόμενο αποτέλεσμα καθώς η απόδοση του rCUDA θα έπρεπε να περιορίζεται από το εύρος ζώνης του δικτύου. Μια πιο προσεκτική έρευνα έδειξε ότι αυτό συνέβαινε λόγω κάποιων αρχικοποιήσεων του CUDA Driver που πραγματοποιούσε στον εξυπηρετητή με την GPU. Για να πάρει κανείς τα ουσιαστικά αποτελέσματα πρέπει να ξεκινήσει τη μέτρηση του χρόνου αμέσως μετά την αρχικοποίηση του CUDA Driver. Ο τελευταίος σύμφωνα με τα επίσημα έγγραφα της NVIDIA αρχικοποιείται κατά την πρώτη κλήση του CUDA Runtime API. Οπότε, τοποθετώντας μια κλήση αυτού χωρίς αντίκτυπο στο πρόγραμμα, όπως είναι π.χ η *cudaFree(0)*, ακριβώς πριν τη μέτρηση του χρόνου μας δίνει αξιόπιστες μετρήσεις.

Το κάθε πείραμα εκτελέστηκε 10 φορές και υπολογίστηκε ο μέσος όρος των χρόνων. Τα αποτελέσματα φαίνονται παρακάτω :



Διάγραμμα 4: Σύγκριση του χρόνου εκτέλεσης μεταξύ τοπικής εκτέλεσης CUDA και απομακρυσμένης εκτέλεσης με το rCUDA.

Από το παραπάνω διάγραμμα συμπεραίνουμε ότι η απόδοση του rCUDA είναι σε γενικές γραμμές χειρότερη από αυτή της τοπικής εκτέλεσης του CUDA. Μεγάλη διαφορά στην επίδοση χρόνου φαίνεται να υπάρχει στο bandwidth test δηλαδή στις εφαρμογές που απαιτείται μεταφορά μεγάλου όγκου δεδομένων πάνω από το δίκτυο. Τα μηχανήματα στο εργαστήριο συνδέονται με Gigabit Ethernet, γεγονός που αποτελεί στενωπό για το σύστημα, ειδικά όταν οι εφαρμογές μας έχουν αρκετή μεταφορά δεδομένων , όπως ακριβώς συμβαίνει στο Bandwidth Test, το οποίο περιλαμβάνει μεγάλο αριθμό από κλήσεις *cudaMemCpy()* μεταξύ του επεξεργαστή και του επιταχυντή. Παρόλα αυτά η ύπαρξη μιας back-to-back σύνδεσης Infiniband μεταξύ δύο μηχανημάτων μας επέτρεψε να διεξάγουμε μετρήσεις και για τέτοιας μορφής σύνδεση. Τα αποτελέσματα φαίνονται παρακάτω και αποδεικνύουν την υπόθεση μας, ότι δηλαδή η τεχνολογία αλλά και κυρίως η ταχύτητα του δικτύου διασύνδεσης επηρεάζει άμεσα την απόδοση του συστήματος.



Διάγραμμα 5: Αποτελέσματα του Bandwidth Test για μεταφορά δεδομένων από τον host στην απομακρυσμένη GPU και αντίστροφα, για τους δύο τύπους δικτύου.

Να σημειωθεί εδώ ότι η αρχική ιδέα ήταν να υλοποιηθεί από την αρχή ένα σύστημα παρόμοιο με το rCUDA για να χρησιμοποιηθεί για την απομακρυσμένη εκτέλεση των CUDA εντολών. Ωστόσο, η απόδοση, η λειτουργικότητα και η ευελιξία που παρέχει το ήδη υλοποιημένο rCUDA μας οδήγησαν στην απόφαση να το χρησιμοποιήσουμε ως έχει.

Το βασικό πλεονέκτημα που προκύπτει από τη χρήση του rCUDA είναι η εξοικονόμηση ενέργειας μέσω της μείωσης του αριθμού των GPUs στις συστοιχίες. Μετρήσεις [7], έχουν δείξει ότι σε συστοιχία υπολογιστών παραγωγής, η συνολική κατανάλωση ενέργειας μπορεί να μειωθεί έως και 20% μειώνοντας τον αριθμό των επεξεργαστών γραφικών και χρησιμοποιώντας το rCUDA, με ελάχιστη μείωση της συνολικής απόδοσης. Η επιπλέον κίνηση που δημιουργείται στο δίκτυο λόγω του μεγάλου όγκου επικοινωνίας δεν αυξάνει την κατανάλωση στον εξοπλισμό δικτύωσης, αφού στις σύγχρονες συστοιχίες υπολογιστών οι συνδέσεις διατηρούνται συνεχώς ζωντανές με στόχο τη μείωση του latency.

Αριθμός GPUs	Κατανάλωση (W)	Εξοικονόμηση (W)	Ποσοστό
100	80622.0	0.0	0.0 %
50	70943.0	9679.0	12.0 %
25	66103.5	14518.5	18.0 %
10	63199.8	17422.2	21.6 %

Πίνακας 2 : Εξοικονόμηση ενέργειας από τον περιορισμό στον αριθμό των GPUs σε ένα cluster με 100 κόμβους.

2.3 Σύγκριση με παρόμοια συστήματα

Το rCUDA δεν είναι το μοναδικό σύστημα το οποίο επιτρέπει τη δημιουργία virtual GPUs. Παρακάτω αναφέρονται συνοπτικά αντίστοιχες τα γνωστότερα που υπάρχουν :

- gVirtuS: Δημιουργήθηκε για να κάνει διαθέσιμες τις GPUs σε εικονικές μηχανές. Χρησιμοποιείται κυρίως σε Private clouds καθώς και επίσης στο γνωστό λογισμικό OpenStack.
- DS-CUDA: Middleware, που επιτρέπει, ό,τι ακριβώς και το rCUDA, δηλαδή την πρόσβαση σε NVIDIA GPUs πάνω από ένα δίκτυο.
- vCUDA
- GViM
- GridCUDA
- V-GPU

Για τα τελευταία δεν θα αναφέρουμε τίποτα, καθώς η υλοποίησεις τους έχουν σταματήσει σε παλαιότερες εκδόσεις του CUDA. Όσο για τα gVirtuS και DS-CUDA, μετρήσεις από τους δημιουργούς του rCUDA δείχνουν ότι αποδίδουν πολύ χειρότερα από το τελευταίο, κυρίως στη μεταφορά δεδομένων από και προς τη συσκευή. Αυτό οφείλεται κυρίως στην βελτιστοποίηση του rCUDA σε αυτό το κομμάτι.

2.4 Συμπεράσματα

Σε γενικές γραμμές το rCUDA είναι ένα αξιόπιστο σύστημα πελάτη – εξυπηρετητή που μας επιτρέπει να πραγματοποιούμε απομακρυσμένες CUDA κλήσεις. Το μοναδικό μειονέκτημα του συστήματος αυτού είναι η επίδοση, καθώς κάθε κλήση πρέπει να “διέλθει” μέσα από το δίκτυο διασύνδεσης. Υπάρχει σημαντική βελτίωση με ταχύτερες τεχνολογίες όπως το Infiniband, αλλά αυτό ανεβάζει σημαντικά το κόστος κάνοντας έτσι το σύστημα αποδοτικό μόνο σε υπερυπολογιστές μεγάλης κλίμακας. Σε γενικές γραμμές πάντως, μια έρευνα που διεξήχθη [9] έδειξε ότι η λύση της εικονικής GPU με τον τρόπο που την υλοποιεί το rCUDA είναι μια βιώσιμη επιλογή όταν τα προβλήματα είναι απαιτητικά από πλευράς υπολογισμών.

Η τελευταία έκδοση 4.0.1 μάλιστα έχει υποστεί αρκετές βελτιστοποιήσεις, κυρίως στο κομμάτι της μεταφοράς δεδομένων, όπου υλοποιεί ένα custom πρωτόκολλο επικοινωνίας και χρησιμοποιεί την τεχνολογία GPUDirect για να αποφύγει περιττές αντιγραφές στη μνήμη, καθώς και pinned θέσεις μνήμης για ταχύτερη μεταφορά των δεδομένων.

Συμπερασματικά λοιπόν και έπειτα από την αξιολόγηση που διεξήγαμε, πρόκειται για ένα αρκετά αποδοτικό σύστημα που υλοποιεί την ιδέα της απομακρυσμένης εκτέλεσης GPU κώδικα για τους κόμβους που δε διαθέτουν επιταχυντές. Για το λόγο αυτό επιλέξαμε να ενσωματωθεί στο τελικό σύστημα, παρά το γεγονός ότι πρόκειται για ένα “μαύρο κουτί”, δηλαδή ένα κλειστού κώδικα σύστημα.

Κεφάλαιο 3

rGPU : Το προτεινόμενο σύστημα διαχείρισης των απομακρυσμένων εκτελέσεων CUDA

Η τελική υλοποίηση αυτής της διπλωματικής εργασίας είναι η ανάπτυξη ενός κατανεμημένου συστήματος, που εκμεταλλευόμενο το rCUDA, θα κατανέμει αποδοτικά την εκτέλεση CUDA εργασιών στην συστοιχία. Πρωταρχικός στόχος ήταν να παραμείνει ανέπαφο το προγραμματιστικό στυλ. Έτσι, το σύστημα είναι διαφανές στον τελικό χρήστη ο οποίος κατά την εκτέλεση των εφαρμογών του δεν χρειάζεται να τροποποιήσει απολύτως τίποτα. Η διαφορά του με τα κοινά συστήματα διαχείρισης εργασιών βρίσκεται όχι μόνο στον διαδραστικό χαρακτήρα του (σε αντίθεση με την κατά δέσμες (batch) επεξεργασία που προσφέρουν τα περισσότερα από αυτά), αλλά και στο σύστημα παρακολούθησης των πόρων που είναι ειδικά σχεδιασμένο για τις GPUs.

3.1 Κατανεμημένα Συστήματα

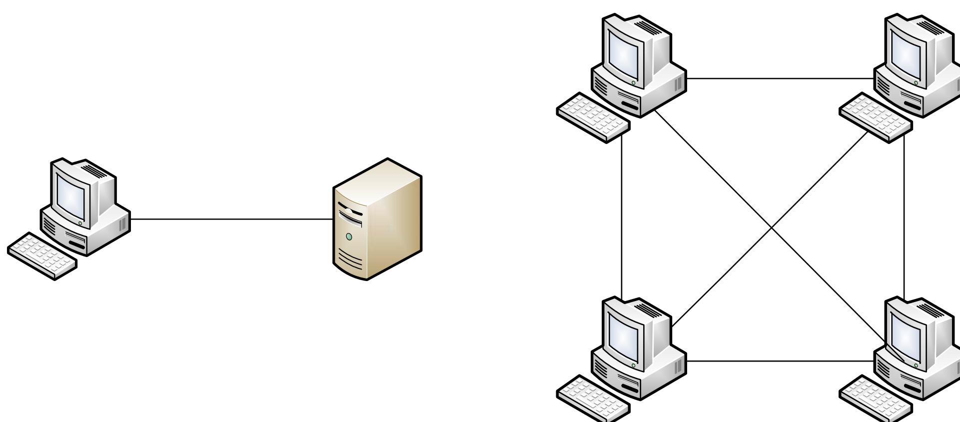
Στην επιστήμη των υπολογιστών, ένα κατανεμημένο σύστημα (distributed system) είναι ένα σύστημα λογισμικού, του οποίου τα συστατικά βρίσκονται σε διασυνδεδεμένους υπολογιστές και των οποίων η επικοινωνία και ο συντονισμός γίνονται κυρίως με μεταβίβαση μηνυμάτων. Όλα αυτά τα συστατικά αλληλεπιδρούν μεταξύ τους, με σκοπό να πετύχουν έναν κοινό στόχο. Υπάρχουν και άλλες εναλλακτικές στον μηχανισμό ανταλλαγής μηνυμάτων και περιλαμβάνουν την Απομακρυσμένη Εκτέλεση Διεργασιών (Remote Procedure Call ή RPC) ή και τα Συστήματα Ουρών (Message Queues). Ένας σημαντικός στόχος των κατανεμημένων συστημάτων είναι διαφάνεια στην τοποθεσία των πόρων. Μπορεί δηλαδή ένας πόρος του συστήματος (π.χ. ένα αρχείο) να εμφανίζεται ως μια λογική οντότητα, αλλά παρόλα αυτά να είναι κατανεμημένος σε πολλά διαφορετικά φυσικά σημεία (π.χ. σε πολλούς δίσκους αποθήκευσης).

Βασικά χαρακτηριστικά των κατανεμημένων συστημάτων, πέρα από τον κοινό σκοπό που πρέπει να έχουν όλα τα υπο-συστήματα που τα αποτελούν, περιλαμβάνουν :

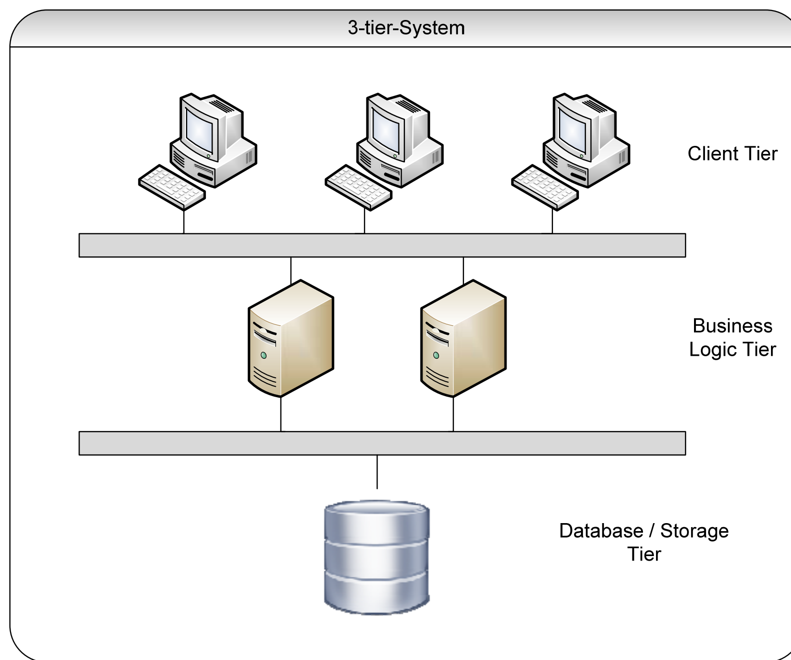
- Ανεκτικότητα σε σφάλματα : Το συνολικό σύστημα θα πρέπει να είναι σε θέση να λειτουργεί αδιάλειπτα, ακόμα και αν ορισμένες από τις μονάδες που το αποτελούν αποτύχουν.
- Η δομή του συστήματος (τοπολογία δικτύου, καθυστέρηση δικτύου, αριθμός υπολογιστών) δεν είναι γνωστή εκ των προτέρων. Το σύστημα πρέπει να εξακολουθεί να λειτουργεί ακόμη και αν αυτά για κάποιο λόγο μεταβληθούν.
- Κάθε υπολογιστής "βλέπει" μόνο ένα περιορισμένο κομμάτι του συστήματος και γνωρίζει μόνο ένα μέρος της εισόδου.

Οι βασικές κατηγορίες στις οποίες υποπίπτουν τα κατανεμημένα συστήματα είναι οι εξής :

- **Πελάτη Εξυπηρετητή (Client –Server) :** Ο κώδικας που τρέχει στον πελάτη επικοινωνεί με τον εξυπηρετητή σε μια ένα-προς-ένα σχέση.
- **Αρχιτεκτονική 3 επιπέδων (3-tier) :** Τα συστήματα αυτά μετακινούν την νοημοσύνη του πελάτη σε ένα μεσαίο επίπεδο, αφήνοντας στο χαμηλότερο επίπεδο τα πιο τεχνικά ζητήματα. Πρόκειται για την αρχιτεκτονική που χρησιμοποιείται στις περισσότερες εφαρμογές του διαδικτύου.
- **Αρχιτεκτονική n επιπέδων (n-tier) :** Πρόκειται για αρχιτεκτονική που χρησιμοποιείται σε web εφαρμογές οι οποίες προωθούν τις αιτήσεις σε εξωτερικές υπηρεσίες.
- **Συζευγμένες (highly coupled) :** Πρόκειται για αρχιτεκτονική που έχει αναλυθεί εκτενώς στα πλαίσια αυτής της εργασίας. Αναφέρεται σε συστοιχίες υπολογιστών που συνεργάζονται για την επίλυση ενός κοινού προβλήματος.
- **Ομότιμη (peer-to-peer)** : Σε αυτή την αρχιτεκτονική δεν υπάρχει συγκεκριμένο μηχάνημα που να παρέχει υπηρεσίες στα υπόλοιπα ή να διαχειρίζεται κεντρικά τους πόρους. Αντιθέτως, όλες οι ευθύνες ανατίθενται ομοιόμορφα σε όλα τα μηχανήματα τα οποία μπορεί να είναι είτε πελάτες είτε εξυπηρετητές.



Εικόνα 23 : Παραδείγματα Κατανεμημένων Συστημάτων. Αριστερά το μοντέλο Client - Server και δεξιά το peer-to -peer.

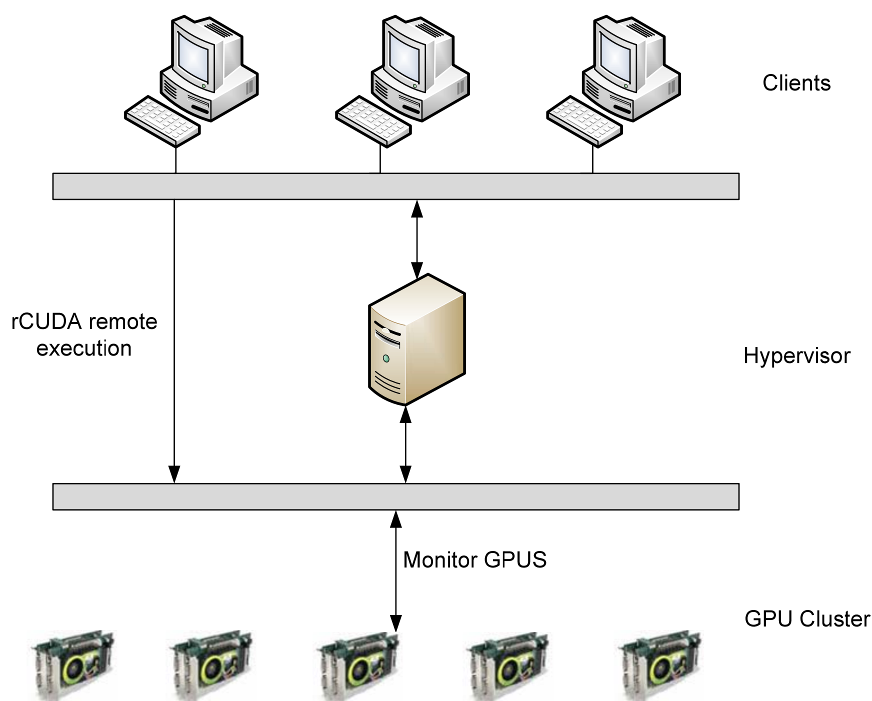


Εικόνα 24 : Παράδειγμα ενός κατανεμημένου συστήματος 3 επιπέδων.

3.2 rGPU : Σχεδιασμός – Αρχιτεκτονική

Το σύστημα που παρουσιάζεται είναι κατανεμημένο και βασίζεται στην αρχιτεκτονική Client – Server για τις διμερείς επικοινωνίες του rCUDA.

Παρακάτω παρουσιάζεται η αρχιτεκτονική του συστήματος.



Εικόνα 25: Διάγραμμα που απεικονίζει την αρχιτεκτονική του προτεινόμενου συστήματος.

Παρακάτω περιγράφονται αναλυτικά τα συστατικά του συστήματος :

3.2.1 Πελάτης (client)

Ο πελάτης μπορεί να είναι οποιοσδήποτε κόμβος στο cluster είτε διαθέτει GPU είτε όχι. Είναι αυτός που ενδιαφέρεται να τρέξει ένα CUDA πρόγραμμα. Θεωρητικά, αν το σύστημα που τρέχει ο πελάτης είχε εγκατεστημένη GPU, θα μπορούσε να γίνει τοπική εκτέλεση με καλύτερες επιδόσεις. Επειδή όμως κάτι τέτοιο θα έδινε πλεονέκτημα σε μερικούς χρήστες έναντι κάποιων άλλων, ακόμα και μια τέτοιου είδους αίτηση θα πρέπει να περάσει από το σύστημα ελέγχου. Έχει φορτωμένες τις βιβλιοθήκες του rCUDA για την απομακρυσμένη εκτέλεση του CUDA κώδικα. Ο πελάτης κάνει αίτηση για εκτέλεση CUDA και ένας ειδικός εξυπηρετητής (ο hypervisor), αναλαμβάνει να του απαντήσει σε ποίον κόμβο πρέπει να συνδεθεί για να πραγματοποιήσει την απομακρυσμένη εκτέλεση.

3.2.2 Επόπτης (Hypervisor)

Πρόκειται για τον κύριο εξυπηρετητή του συστήματος. Επιλέγεται από τον διαχειριστή και οι βασικές του λειτουργίες είναι οι εξής :

- Είναι υπεύθυνος για την αρχικοποίηση του συστήματος. Διαβάζει ένα αρχείο ρυθμίσεων (configuration) και ελέγχει τους κόμβους του συστήματος. Το αρχείο αυτό περιέχει πληροφορίες για τους κόμβους, όπως η διεύθυνση IP, το όνομα τους και η διαθεσιμότητα GPUs. Ένα παράδειγμα του αρχείου αυτού φαίνεται παρακάτω

```
ID    Name       IP Address      GPUs
0     xenon5      147.102.4.69     0
1     xenon6      147.102.4.70     0
2     xenon7      147.102.4.71     0
3     xenon8      147.102.4.72     0
4     clone26     147.102.4.154    0
5     clone29     147.102.4.157    0
6     termi2      147.102.4.172    0
7     termi3      147.102.4.173    1
8     termi4      147.102.4.174    1
9     termi6      147.102.4.176    1
```

Το παραπάνω αρχείο δηλώνει ότι υπάρχουν 10 κόμβοι διαθέσιμοι και ότι οι κόμβοι *termi3, termi4* και *termi6* διαθέτουν από μια GPU ο καθένας. Να σημειωθεί εδώ ότι λέγοντας GPU εννοείται NVIDIA GPU με δυνατότητα εκτέλεσης CUDA. Για την διευκόλυνση του διαχειριστή συστήματος, δίνεται και ένα script που παράγει αυτόματα το αρχείο αυτό. Σε κάθε περίπτωση όμως θα πρέπει να ελεγχθεί χειροκίνητα.

- Παρακολουθεί το φορτίο των GPUs στους servers και αναλόγως κατανέμει τις διάφορες αιτήσεις για εκτέλεση CUDA εργασιών. Να σημειωθεί εδώ ότι για λόγους απόδοσης η λειτουργία αυτή είναι παθητική. Δηλαδή, ο επόπτης παρακολουθεί τις GPUs ακόμα και αν δεν έχει γίνει κάποια αίτηση από πελάτη. Ο λόγος για τον οποίο συμβαίνει αυτό εξηγείται στο τμήμα 3.3, όπου εκθέτουμε τις αρχιτεκτονικές αποφάσεις που λάβαμε. Για τη παρακολούθηση των GPUs χρησιμοποιείται το εργαλείο NVIDIA SMI [10], το οποίο είναι ένα εργαλείο γραμμής εντολών φτιαγμένο για ακριβώς αυτό τον σκοπό. Για παράδειγμα δίνοντας την εντολή *"nvidia-smi –q –i 0 –d UTILIZATION"* μπορούμε να ενημερωθούμε για το ποσοστό χρησιμοποίησης της GPU υπ' αριθμόν 0 που βρίσκεται εγκατεστημένη στο σύστημα. Αντίστοιχες εντολές υπάρχουν και για την παρακολούθηση της χρήσης της μνήμης συσκευής.

Όσον αφορά το σύστημα λήψης των αποφάσεων δέχεται ως είσοδο τις τελευταίες μετρήσεις χρησιμοποίησης και διαλέγει την GPU με το ελάχιστο φορτίο είτε βάσει της τελευταίας μέτρησης, είτε προσδιορίζοντας έναν σταθμισμένο μέσο όρο των τελευταίων μετρήσεων. Η τελική τιμή που αναφέρεται ως "φορτίο" για κάθε συσκευή είναι συνδυασμός του φορτίου στην GPU και της χρησιμοποίησης της μνήμης της με βάση έναν παράγοντα (P_FACTOR).

Έτσι αυτό υπολογίζεται ως εξής :

$$Utilization = (GPU\ Utilization) \times P_FACTOR + (Memory\ Utilization) \times (1 - P_FACTOR)$$

Με αυτό τον τρόπο πετυχαίνουμε ισοκατανομή του φορτίου στις GPUs. Γενικά είναι προτιμότερο να δίνεται περισσότερη έμφαση στη χρήση της GPU, από ότι στην χρήση της μνήμης, οπότε ο παράγοντας P_FACTOR προτιμάται να κυμαίνεται από 0.6 έως 0.9.

3.2.2 Εξυπηρετητές GPU

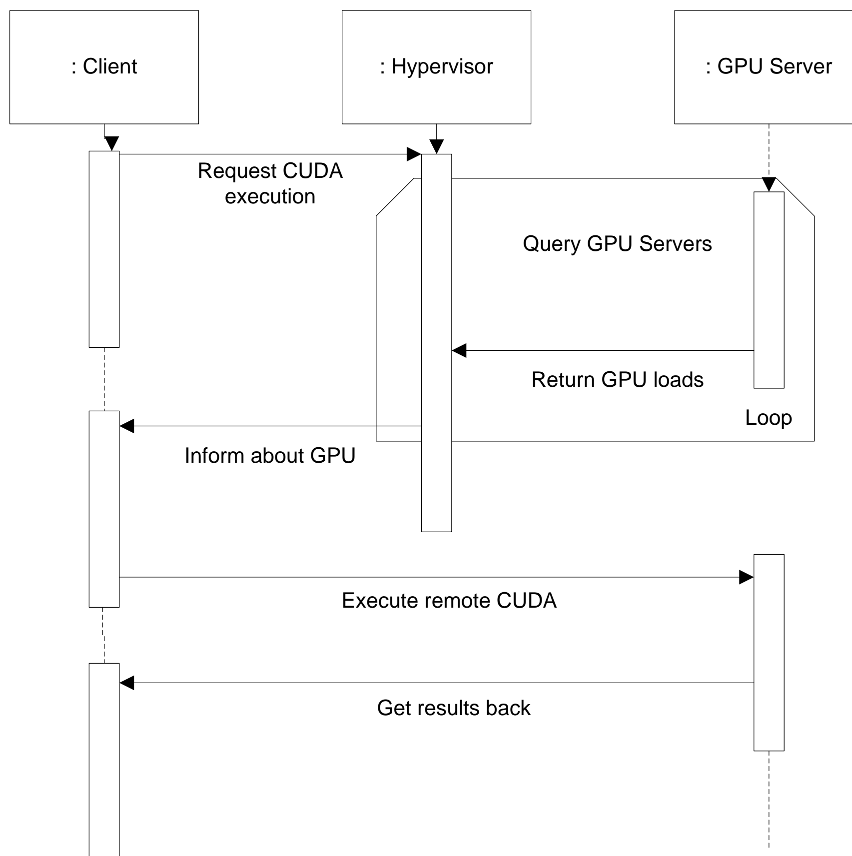
Πρόκειται για την συστοιχία των κόμβων. Μας ενδιαφέρουν οι κόμβοι που έχουν επιταχυντές οι οποίοι αναφέρονται ως “GPU Servers”. Οι κόμβοι αυτοί πρέπει να έχουν εγκατεστημένο το rCUDA (το κομμάτι του server) και να τρέχουν τον rCUDA daemon περιμένοντας για απομακρυσμένες κλήσεις CUDA.

Για καλύτερη κατανομή φορτίου προτιμάται το cluster να είναι ομοιογενές, δηλαδή όλες οι GPUs να είναι ίδιας αρχιτεκτονικής. Αυτό συμβαίνει γιατί το σύστημα μας διαχειρίζεται όλες τις GPU ισάξια, οπότε αν είναι διαφορετικές δεν θα είναι δίκαιο. Ο διαχειριστής του συστήματος είναι υπεύθυνος για όλες τις αρχικοποιήσεις με τα κατάλληλα scripts που δίνονται. Όπως προαναφέρθηκε η διασύνδεση των κόμβων της συστοιχίας μπορεί να γίνεται είτε με Gigabit Ethernet, είτε με πιο προηγμένες μορφές όπως το Infiniband.

Οι κόμβοι που διαθέτουν GPUs είναι προφανώς περιορισμένοι και όλο το υπολογιστικό φορτίο πέφτει πάνω τους. Κάθε αίτηση για εκτέλεση που δέχεται ένας εξυπηρετητής από έναν απομακρυσμένο πελάτη δημιουργεί μια ξεχωριστή διεργασία rCUDA daemon. Με αυτό τον τρόπο επιτυγχάνεται πολυπλεξία των ροών εκτέλεσης στην GPU καθώς και αυξημένη αξιοπιστία αφού αν αποτύχει μια διεργασία, οι επόμενες συνεχίζουν κανονικά την εκτέλεσή τους.

3.3 Τυπικό σενάριο λειτουργίας

Το τυπικό σενάριο λειτουργίας του συστήματος πραγματοποιείται σε ένα σύνολο διασυνδεδεμένων κόμβων, είτε βρίσκονται σε μια συστοιχία υπολογιστών, είτε έχουν μια κοινή διασύνδεση δικτύου και δεν είναι στενά συζευγμένοι. Εκκινείται ο επόπτης σε ένα από τα συστήματα το οποίο θεωρείται κεντρικό και αμέσως πραγματοποιεί όλες τις απαραίτητες αρχικοποιήσεις στους εξυπηρετητές με GPUs και στους πελάτες. Το σύστημα είναι διαφανές στον χρήστη, αφού ο πελάτης που επιθυμεί να τρέξει ένα CUDA πρόγραμμα δεν αλλάζει την γνωστή διαδικασία. Σχηματικά ένα σενάριο λειτουργίας φαίνεται παρακάτω :



Διάγραμμα 6 : Διάγραμμα ακολουθίας (sequence diagram) της λειτουργίας του συστήματος.

3.4 Αρχιτεκτονικές αποφάσεις – Μειονεκτήματα του συστήματος

Κατά την διάρκεια του σχεδιασμού του συστήματος ήταν αναγκαίο να λάβουμε κάποιες αποφάσεις, οι οποίες θα επηρέαζαν την πολυπλοκότητα του, αλλά και την ταχύτητα του τελικού προϊόντος. Σε αρκετές περιπτώσεις βρέθηκε μια συμβιβαστική λύση, ενώ πολλές αποφάσεις λήφθηκαν με κριτήριο τη δυσκολία υλοποίησης. Στο παρακάτω κομμάτι γίνεται ένας σχολιασμός αυτών τον αποφάσεων καθώς και πιθανά σημεία αδυναμίας του προτεινόμενου συστήματος.

Η πρώτη απόφαση αφορά τη λειτουργία του επόπτη (hypervisor), του οποίου ο ρόλος, όπως προαναφέραμε είναι η παρακολούθηση των πόρων και η καθοδήγηση των πελατών ως προς το που να εκτελέσουν τα προγράμματά τους. Ο επόπτης ρωτάει συνεχώς τους κόμβους που διαθέτουν GPUs για την κατάσταση τους και η διαδικασία αυτή απαιτεί κάποιο εύλογο χρονικό διάστημα. Αυτό οφείλεται στο ότι ο μόνος τρόπος να πάρει κανείς πληροφορίες για την κατάσταση των GPUs είναι μέσω του εργαλείου NVIDIA-SMI, το οποίο χρησιμοποιείται μόνο ως εκτελέσιμο. Μετρήσεις δείχνουν ότι απαιτούνται περίπου 2.7 sec, χρόνος καθόλου αμελητέος για ένα διαδραστικό σύστημα. Ακόμα και αν δημιουργηθούν ξεχωριστές διεργασίες/νήματα για την παρακολούθηση κάθε GPU ο χρόνος παραμένει ο ίδιος. Τέλος, ακόμα και η χρήση της βιβλιοθήκης NVML (Nvidia Management Library), μιας βιβλιοθήκης της C που εκθέτει τη λειτουργικότητα του εργαλείου, δεν βελτιώνει την απόδοση. Για το

λόγο αυτό επιλέχθηκε η λειτουργία του επόπτη να είναι παθητική, δηλαδή να επικοινωνεί συνεχώς με τους εξυπηρετητές GPU (ανά κατάλληλα χρονικά διαστήματα που ρυθμίζονται από τον διαχειριστή), να ενημερώνεται για την κατάσταση τους και να την αποθηκεύει εσωτερικά. Έτσι μπορεί να απαντήσει άμεσα στους πελάτες τη στιγμή που τον ρωτούν. Το μοναδικό ίσως μειονέκτημα είναι ότι υπάρχει μια συνεχής, ίσως και άσκοπη μερικές φορές σύνδεση με τους εξυπηρετητές, που θα μπορούσε να επιβαρύνει το δίκτυο, αλλά που είναι ωστόσο αμελητέα.

Η δεύτερη απόφαση αφορά το κατά πόσο το τελικό σύστημα θα αλλάζει το προγραμματιστικό στυλ ή το στυλ εκτέλεσης των προγραμμάτων CUDA. Για παράδειγμα κατά την απομακρυσμένη εκτέλεση με το rCUDA, ο προγραμματιστής δεν χρειάζεται να αλλάξει τίποτα καθώς χρησιμοποιείται μια τροποποιημένη έκδοση της Runtime βιβλιοθήκης του CUDA, η οποία αναλαμβάνει να προωθήσει τις κλήσεις με διαφανή τρόπο. Λόγω περιορισμών που θέτει ο τρόπος εκτέλεσης του rCUDA όμως, το οποίο απαιτεί να τεθούν οι μεταβλητές περιβάλλοντος που υποδεικνύουν στον πελάτη σε ποιόν εξυπηρετητή θα γίνει η απομακρυσμένη εκτέλεση, η μόνη λύση θα ήταν η παρακολούθηση των ανοιχτών sessions των πελατών από τον επόπτη και η ενημέρωση των φλοιών που θα γίνει η εκτέλεση των προγραμμάτων. Έτσι όταν ο πελάτης θα τρέξει το CUDA πρόγραμμα του, θα γνωρίζει ήδη που πρέπει να κάνει την απομακρυσμένη εκτέλεση. Επιχειρήθηκε μια τέτοιου είδους υλοποίηση, η οποία χρησιμοποιεί τα *screens* του Linux, ωστόσο για να λειτουργήσει σωστά απαιτούνται αρκετές αρχικοποιήσεις και τροποποιήσεις στους πελάτες. Για το λόγο αυτό καταλήξαμε σε μια δεύτερη υλοποίηση στην οποία ο πελάτης πρέπει να δηλώσει ρητά ότι χρησιμοποιεί το σύστημα επικοινωνίας με τον επόπτη. Αντί λοιπόν να τρέξει *./myCudaProgram* θα τρέξει *rgpu_client ./myCudaProgram* και το πρόγραμμα πελάτη *rgpu_client* θα αναλάβει την επικοινωνία.

Η τρίτη απόφαση αφορούσε το ποσοστό της νοημοσύνης που θα "μοιραζόταν" ανάμεσα στον πελάτη και τον επόπτη. Αποφασίστηκε ο επόπτης να πραγματοποιεί το μεγαλύτερο μέρος των εργασιών (αρχικοποιήσεις, παρακολούθηση των GPUs, έλεγχος για σφάλματα), να διατηρεί όλα τα αρχεία με τις ρυθμίσεις και να αποστέλλει ότι χρειάζεται κάθε φορά στον πελάτη, ο οποίος δεν γνωρίζει ουσιαστικά τίποτα για το σύστημα μέχρι να πάρει την πληροφορία από τον επόπτη. Αυτό προσθέτει ελάχιστο φόρτο στο δίκτυο, άλλα αφαιρεί αρμοδιότητες από τον πελάτη, του οποίου η κύρια επιθυμία είναι η εκτέλεση του CUDA προγράμματος.

Το μοναδικό ίσως σημείο αμφιβολίας για το σύστημα είναι η μοναδική παρουσία του επόπτη, ο οποίος συγκεντρώνει αρκετές ευθύνες, ανοιχτές συνδέσεις και επεξεργαστικό φόρτο. Παρά το γεγονός ότι η υλοποίηση του περιλαμβάνει και ξεχωριστά νήματα, παραμένει ένα μοναδικό σημείο στο σύστημα, αποτυχία του οποίου οδηγεί στην κατάρρευση του συνόλου.

3.5 Εναλλακτικές λύσεις για την διαχείριση των GPU εργασιών

Η ιδέα της χρονοδρομολόγησης των GPU εργασιών δεν είναι πρωτόγνωρη. Πολλές μεγάλες εταιρείες ανάπτυξης λογισμικού παρέχουν λύσεις διαχείρισης των εργασιών που απαιτούν GPU στα πλαίσια ενός cluster. Οι περισσότερες από αυτές τις λύσεις αποτελούν επεκτάσεις των παραδοσιακών συστημάτων διαχείρισης των πόρων που τοποθετούν τις υπό εκτέλεση εργασίες σε ουρές και τις χρονοδρομολογούν με βάση συγκεκριμένα κριτήρια. Όπως αναφέραμε και προηγουμένως όμως, οι GPUs αποτελούν έναν ιδιαίτερο πόρο που πρέπει να διαχειρίζεται διαφορετικά από τα υπόλοιπα κομμάτια υλικού. Όσον αφορά λοιπόν το λογισμικό ανοικτού κώδικα στον τομέα αυτό, υπάρχει ένα τεράστιο κενό.

Κατά τη συγγραφή της διπλωματικής αυτής εργασίας οι μοναδικές επιλογές ανοικτού κώδικα είναι οι:

<u>Torque (PBS)</u>

Ουσιαστικά επέκταση του γνωστού συστήματος διαχείρισης cluster ώστε να αντιλαμβάνεται τις GPUs και να χρονοδρομολογεί τις εργασίες πιο αποδοτικά. Στα θετικά συγκαταλέγονται η ευκολία χρήσης και εγκατάστασης, καθώς και το γεγονός ότι το συγκεκριμένο σύστημα είναι καθιερωμένο στην αγορά και ευρέως διαδεδομένο. Παρόλα αυτά ο υπάρχων χρονοδρομολογητής του Torque είναι αρκετά απλός στην υλοποίηση του και δεν λαμβάνει υπόψη του πιο εξελιγμένα χαρακτηριστικά. Ο εξελιγμένος χρονοδρομολογητής του Torque, εν ονόματι Maui δεν υποστηρίζει επίσης GPUs.

<u>SLURM</u>

Ο χρονοδρομολογητής SLURM υποστηρίζει εν γένει τις GPUs, ωστόσο είναι αρκετά πολύπλοκος ως σύστημα και απαιτεί αρκετό κόπο για την σωστή εγκατάσταση και ρύθμισή του. Από όλες τις διαθέσιμες υλοποιήσεις ανοικτού κώδικα παρόλα αυτά, ο SLURM αποτελεί την πιο προηγμένη και αξιόπιστη λύση. Είναι αποκριτικός σε μεγάλο επίπεδο και διαθέτει προηγμένες μεθόδους κατανομής των φορτίων.

<u>Sun Grid Engine</u>

Πρόκειται για το batch σύστημα υποβολής της Sun, προσφάτως Oracle. Δεν διαθέτει εγγενή υποστήριξη για χρονοδρομολόγηση GPUs.

Κεφάλαιο 4

Υλοποίηση του rGPU

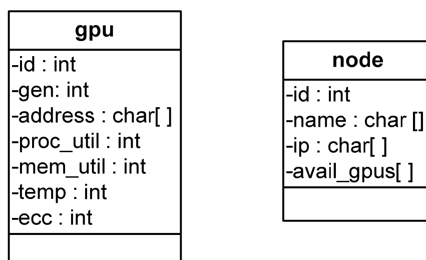
Στο παρακάτω κεφάλαιο περιγράφεται σε γενικές γραμμές η διαδικασία της υλοποίησης του συστήματος σύμφωνα με την σχεδίαση που περιγράψαμε προηγουμένως. Εφόσον πρόκειται για κατανεμημένο σύστημα, αποτελείται από διάφορα συστατικά που τρέχουν σε διαφορετικά συστήματα. Από τα 3 συστατικά (πελάτης, επόπτης και εξυπηρετητής GPU) , που περιγράφηκαν πιο πάνω υλοποιούνται ουσιαστικά ο πελάτης και ο επόπτης, αφού η λειτουργία των εξυπηρετητών GPU υλοποιείται με το rCUDA.

4.1 Πλατφόρμα Ανάπτυξης

Το σύστημα rGPU αναπτύχθηκε για το λειτουργικό σύστημα Linux και έχει δοκιμαστεί συγκεκριμένα στην 32-bit έκδοση του Ubuntu 12.04 LTS. Επειδή όμως δεν χρησιμοποιεί λειτουργίες που εξαρτώνται από το λειτουργικό σύστημα, είναι συμβατό και με άλλες εκδόσεις του Linux. Ο λόγος που επιλέγεται το εν λόγω λειτουργικό σύστημα είναι ότι είναι εγκατεστημένο στο μεγαλύτερο ποσοστό των συστημάτων που στοχεύει η εφαρμογή, δηλαδή σε clusters και datacenters. Για την ανάπτυξη χρησιμοποιήθηκε η γλώσσα προγραμματισμού C και συγκεκριμένα η υλοποίηση ANSI C, καθώς είναι η καθιερωμένη γλώσσα για την υλοποίηση τέτοιων συστημάτων. Επίσης χρησιμοποιούνται και scripts κονσόλας (bash scripts) κυρίως για ρυθμίσεις.

4.2 Υλοποίηση του Επόπτη (Hypervisor)

Η λειτουργία του επόπτη περιγράφεται στην ενότητα 3.2.2. Υλοποιείται ως ένας daemon, που έχει όμως και τη δυνατότητα να τρέξει σε διαδραστική λειτουργία για περισσότερες πληροφορίες. Οι δύο κύριες δομές που κρατάει αποθηκευμένες είναι οι *gpu* και *node.* Τα διαγράμματα κλάσης τους φαίνονται παρακάτω :



Εικόνα 26 : Διαγράμματα κλάσης των δύο βασικών δομών του επόπτη.

Παρακάτω περιγράφονται τα attributes των δομών :

gpu

Attribute	Περιγραφή
id	Ένα μοναδικό αναγνωριστικό για την κάθε GPU.
gen	Η γενιά της GPU (0 για G80, 1 για Fermi και 2 για Kepler)
node_name	Το όνομα του κόμβου που διαθέτει την GPU.
address	Η διεύθυνση της GPU όπως ακριβώς την απαιτεί το rCUDA για τη λειτουργία του. Είναι της μορφής <IP κόμβου>:<id μέσα στον κόμβο>.
proc_util	Ποσοστιαία χρήση της GPU.
mem_util	Ποσοστιαία χρήση της μνήμης συσκευής.
temp	Η θερμοκρασία της GPU σε βαθμούς Κελσίου (μόνο για Kepler).
ecc	Ο συνολικός αριθμός ECC λαθών στην GPU (μόνο για Kepler).

node

Attribute	Περιγραφή
id	Ένα μοναδικό αναγνωριστικό για τον κάθε κόμβο.
name	Το όνομα του κόμβου.
ip	Η διεύθυνση IP του κόμβου.
avail_gpus	Ο αριθμός των διαθέσιμων GPUs που βρίσκονται στον κόμβο.

Οι βασικές συναρτήσεις που υλοποιεί ο επόπτης είναι:

```
void * monitor (struct gpu **)
```

Πρόκειται για το σύστημα παρακολούθησης των GPUs. Εκτελείται μέσω του κώδικα επόπτη ως ξεχωριστό thread και ανανεώνει τις πληροφορίες για τις GPUs σε τακτά χρονικά διαστήματα.

```
void parse_nodefile (nodeinfo **, gpuinfo **, int *, int *)
```

Αναλαμβάνει να διαβάσει το αρχείο με τις ρυθμίσεις (configuration file) και να αποθηκεύσει στις εσωτερικές δομές όλες τις απαραίτητες πληροφορίες.

```
int * get_utils (gpuinfo **, int)
```

Επιστρέφει έναν πίνακα με τους πιο κατάλληλους για εκτέλεση εξυπηρετητές GPU, με βάση το φορτίο τους.

```
void print_node_info (nodeinfo **, gpuinfo **, int, int)
```

Εκτυπώνει πληροφορίες σχετικά με τους συνδεδεμένους κόμβους στο standard output.

Η κύρια λειτουργία του επόπτη βρίσκεται σε ένα μεγάλο loop, όπου έχοντας πρώτα ανοίξει μια συγκεκριμένη πόρτα όπου ακούει για αιτήσεις από τους πελάτες, απαντάει σε κάθε μια από αυτές διαλέγοντας τους κατάλληλους εξυπηρετητές για εκτέλεση. Η υλοποίηση της επικοινωνίας γίνεται με τα γνωστά sockets του UNIX. Να σημειωθεί εδώ ότι υπάρχει δυνατότητα εκτέλεσης τόσο σε διαδραστική λειτουργία (Interactive mode) ,όπου τα μηνύματα τυπώνονται στην οθόνη, όσο και ως διεργασία παρασκηνίου (daemon), όπου όλα τα γεγονότα καταγράφονται στο syslog.

4.3 Υλοποίηση του Πελάτη (Client)

Το πρόγραμμα πελάτη αναλαμβάνει απλώς να συνδεθεί στο socket του εξυπηρετητή, να λάβει τις κατάλληλες πληροφορίες για την εκτέλεση του CUDA προγράμματος και να εκκινήσει με βάση αυτές την απομακρυσμένη εκτέλεση σε έναν ή περισσότερους εξυπηρετητές GPU. Η υλοποίηση του είναι σχετικά απλή και δεν παρατίθενται λεπτομέρειες.

4.4 Ρύθμιση και Εκτέλεση

Στο παρακάτω τμήμα παρουσιάζονται οι βασικές οδηγίες για την ρύθμιση και την εκτέλεση του συστήματος. Τα βήματα που πρέπει να ακολουθηθούν είναι :

1. Δημιουργία του αρχείου *nodes.config* που περιγράφει την διάταξη των κόμβων. Παράδειγμα του αρχείου έχει δωθεί σε προηγούμενο κεφάλαιο. Διατίθεται και το script *config_create.sh* ,το οποίο παράγει αυτόματα το συγκεκριμένο αρχείο.

2. Εκτέλεση του *./rgpu_server* στον επόπτη. Πρέπει πρώτα να τεθεί η μεταβλητή περιβάλλοντος RCUDA_PATH στην κατάλληλη τοποθεσία όπου βρίσκεται ο rCUDA daemon στους εξυπηρετητές (θεωρείται ότι είναι σε όλους η ίδια). Οι υπόλοιπες επιλογές για το πρόγραμμα του επόπτη είναι :

-D	Εκτέλεση ως daemon.
-s	Χρήση του syslog για καταγραφή των συμβάντων.
-v	Εκτύπωση της έκδοσης
-p <port>	Χρήση συγκεκριμένης θύρας για την επικοινωνία (default 1091).
-m <AVG ,NOW>	Τρόπος καταγραφής μετρήσεων από τους servers.

-h Εκτύπωση βοήθειας.
-v Αναλυτική εκτύπωση μηνυμάτων (verbose mode)

3. Εκτέλεση του *rgpu_client* στον πελάτη αφού πρώτα τεθεί η μεταβλητή περιβάλλοντος RGPU_HYPERV=<όνομα επόπτη> και προστεθεί το path της βιβλιοθήκης πελάτη rCUDA (…/rCUDAl/) στο LD_LIBRARY_PATH. Οι επιλογές για το πρόγραμμα πελάτη είναι οι εξής :

   -h Εκτύπωση βοήθειας.
   -p <port> Χρήση συγκεκριμένης θύρας για την επικοινωνία (Μόνο σε περίπτωση που έχει χρησιμοποιηθεί διαφορετική θύρα από την προεπιλεγμένη στον hypervisor).
   -r <εκτελέσιμο> Το CUDA εκτελέσιμο.
   -v Αναλυτική εκτύπωση μηνυμάτων (verbose mode)

Hypervisor (rGPU Server)	**rGPU Client 1**	**rGPU Client 2**
$ ./rgpu_server rgpud started by User 10162 List of available nodes ------------------------------- ID	Name	IP	GPUs | ---| 0	xenon5	147.102.4.69	0 | 1	xenon6	147.102.4.70	0 | 2	xenon7	147.102.4.71	0 | 3	xenon8	147.102.4.72	0 | 4	clone2	147.102.4.130	0 | 5	clone17	147.102.4.14	0 | 6	clone18	147.102.4.146	0 | 7	clone29	147.102.4.157	0 | 8	termi2	147.102.4.172	0 | 9	termi3	147.102.4.173	1 | 10	termi4	147.102.4.174	1 | 11	termi11	147.102.4.165	0 | 12	termi12	147.102.4.166	0 | 13	nehalem	147.102.3.131	0 | List of available GPUs ---------------------- ID	Hostname | --------------| 0	termi3 1	termi4 Monitor thread created with mode AVG	**$ export LD_LIBRARY_PATH=/home/users/akarkat/rcuda/framework/rCUDA1/:$LD_LIBRARY_PATH**	**$ export LD_LIBRARY_PATH=/home/users/akarkat/rcuda/framework/rCUDA1/:$LD_LIBRARY_PATH**
Received request from node with IP: 147.102.4.190 ←	**$ ./rgpu_client -r benchs/bin/x86_64/linux/release/bandwidthTest** Communication with server established. Waiting to receive..	**$ ./rgpu_client -r benchs/bin/x86_64/linux/release/bandwidthTest**
Sending Util Table... 0 147.102.4.173 1 147.102.4.174 →	Running rCUDA.. [CUDA Bandwidth Test] - Starting... Running on...	
Received request from node with IP: 147.102.4.191 ←	Device 0: Tesla M2050 Quick Mode Host to Device Bandwidth, 1 Device(s) PINNED Memory Transfers Transfer Size (Bytes)	Communication with server established. Waiting to receive..
Sending Util Table... 1 147.102.4.174 0 147.102.4.173 →	Bandwidth(MB/s) 33554432 111.7 Device to Host Bandwidth, 1 Device(s) … … …	Running rCUDA.. [CUDA Bandwidth Test] - Starting... Running on... Device 0: Tesla M2050 Quick Mode Host to Device Bandwidth, 1 Device(s) … … …

Εικόνα 27: Παράδειγμα εκτέλεσης του συστήματος σε μέρος του cluster του εργαστηρίου που διαθέτει 2 GPUs. Ο πρώτος πελάτης κάνει αίτηση για εκτέλεση CUDA και ο επόπτης του απαντάει διαλέγοντας ως πρώτο GPU Server τον termi3. Κατά τη διάρκεια της εκτέλεσης ένας δεύτερος πελάτης κάνει αίτηση. Ο επόπτης θα διαλέξει τώρα τον termi4 καθώς ο 3 είναι απασχολημένος.

Κεφάλαιο 5

Αξιολόγηση του συστήματος

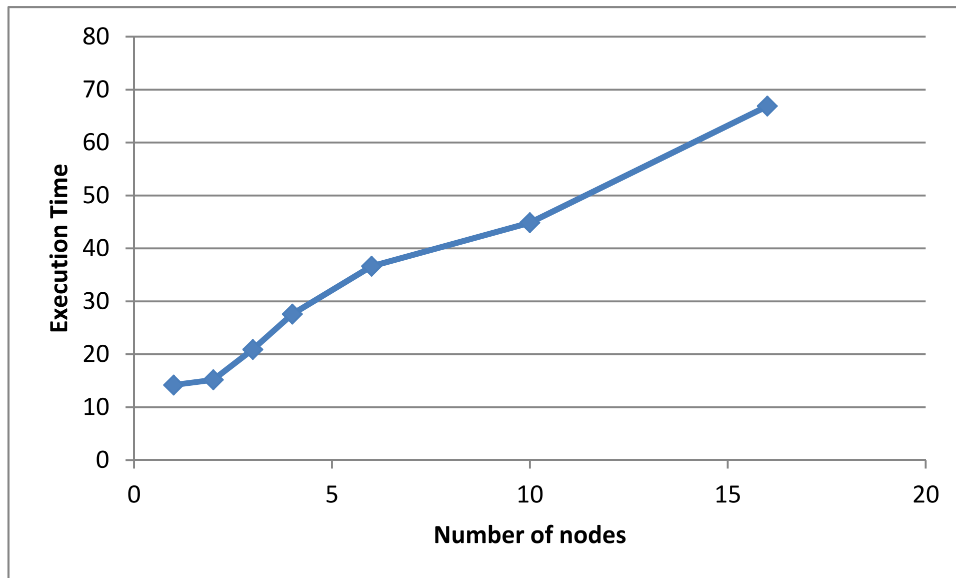
Στο παρόν κεφάλαιο πραγματοποιείται η αξιολόγηση του προτεινόμενου συστήματος ώστε να αποδειχθεί η αποτελεσματικότητα του. Η αξιολόγηση αυτή κινείται γύρω από τρείς κυρίως άξονες με σκοπό να καλύψει όλα τα δυνατά ενδεχόμενα. Αυτοί οι άξονες είναι :

- Μελέτη της Κλιμακωσιμότητας (Scalability) του συστήματος.
- Σύγκριση με κάποιον υπάρχοντα χρονοδρομολογητή κατά την εκτέλεση CUDA διεργασιών.
- Μελέτη της εναλλαγής μεταξύ απαιτητικών σε μνήμη και απαιτητικών σε επεξεργαστική ισχύ εφαρμογών.

2.1 Μελέτη της κλιμακωσιμότητας (Scalability) του συστήματος

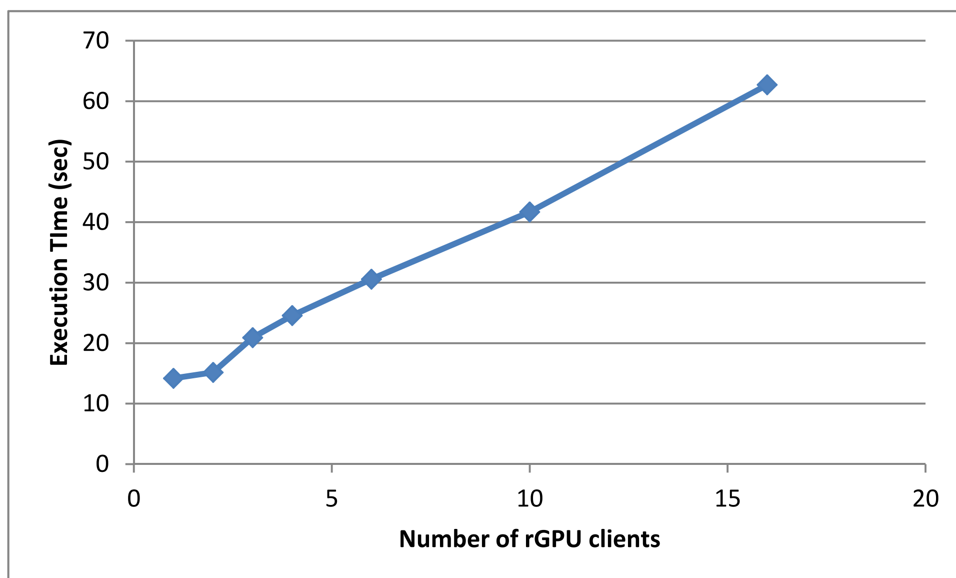
Στο πρώτο κομμάτι των μετρήσεων σκοπός ήταν η μέτρηση της κλιμακωσιμότητας του συστήματος, δηλαδή του πόσο καλά ανταποκρίνεται ο Hypervisor καθώς αυξάνεται ο αριθμός των πελατών που πρέπει να εξυπηρετήσει. Για το σκοπό αυτό υλοποιήσαμε 3 διαφορετικά πειράματα.

Στο πρώτο πείραμα βάλαμε διαφορετικούς κόμβους της συστοιχίας να κάνουν από μια αίτηση για εκτέλεση ενός CUDA προγράμματος και μετρήσαμε τον μέσο όρο του χρόνου που απαιτείται για το κάθε πρόγραμμα να ολοκληρωθεί. Με τον τρόπο αυτό μπορούμε να δούμε την απόκριση του συστήματος σε ένα πραγματικό σενάριο λειτουργίας, όπου πολλοί πελάτες επιθυμούν να τρέξουν ταυτόχρονα τα προγράμματά τους. Το πρόγραμμα που εκτελέστηκε είναι το Matrix Multiplication, από το επίσημο site της NVIDIA. Τα αποτελέσματα φαίνονται στο Διάγραμμα 7. Παρατηρούμε ότι ο χρόνος εκτέλεσης της κάθε διεργασίας σαφώς αυξάνεται όταν ο αριθμός των κόμβων-αιτήσεων γίνει μεγαλύτερος από αυτόν των διαθέσιμων GPUs, αφού πρέπει υποχρεωτικά να γίνει πολυπλεξία των ροών εκτέλεσης. Στο παράδειγμα της εκτέλεσης όπου διαθέταμε 2 GPUs βλέπουμε ότι ο χρόνος εκτέλεσης για έναν και για δύο κόμβους είναι πρακτικά ο ίδιος. Σε αντίθετη περίπτωση, ο χρόνος εκτέλεσης αυξάνει σχεδόν γραμμικά με τον αριθμό των κόμβων / αιτήσεων.



Διάγραμμα 7: Χρόνος Εκτέλεσης για αυξανόμενο αριθμό κόμβων που πραγματοποιούν από μια αίτηση ο καθένας.

Στο δεύτερο κομμάτι, θέσαμε έναν μόνο κόμβο να πραγματοποιεί συνεχόμενες αιτήσεις στον Hypervisor για εκτέλεση ενός CUDA προγράμματος. Αυτό το σενάριο καλύπτει την περίπτωση που ο πελάτης θέλει "απλώσει" κατά το δυνατόν περισσότερο την εκτέλεση των προγραμμάτων του όταν έχει στη διάθεση του αρκετούς πόρους από το cluster. Τα αποτελέσματα, για το ίδιο benchmark όπως προηγουμένως, φαίνονται παρακάτω. Παρατηρούμε ότι ακολουθεί σχεδόν την ίδια πορεία, όπως ήταν αναμενόμενο, αφού και στην προηγούμενη περίπτωση η ανεξάρτητη μεταβλητή είναι ουσιαστικά ο αριθμός των rGPU clients. Εδώ, ωστόσο παρατηρούμε μια μικρή βελτίωση, που οφείλεται πιθανώς στην μείωση της επικοινωνίας κατά την απομακρυσμένη εκτέλεση.

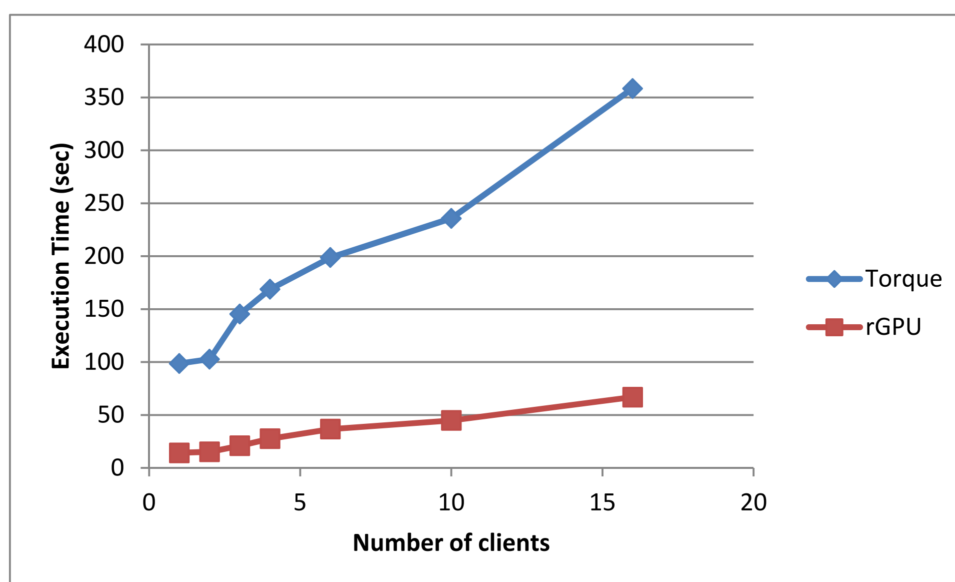


Διάγραμμα 8: Χρόνος Εκτέλεσης για αυξανόμενο αριθμό rGPU clients που εκτελούνται σε έναν κόμβο της συστοιχίας.

5.2 Σύγκριση με τα υπάρχοντα συστήματα διαχείρισης

Στο κομμάτι αυτό επιχειρείται μια σύγκριση του rGPU με τον καθιερωμένο τρόπο εκτέλεσης CUDA προγραμμάτων. Σύμφωνα με αυτόν, για την εκτέλεση ενός CUDA προγράμματος, ο χρήστης ζητάει τους απαιτούμενους πόρους από το σύστημα διαχείρισης (π.χ. το Torque Cluster Management System) και έπειτα εκτελεί ό,τι χρειάζεται στα υποδεδειγμένα μηχανήματα. Σε καμία περίπτωση το προτεινόμενο σύστημα δεν αντικαθιστά τα πολύπλοκα αυτά συστήματα, αλλά υπάρχουν περιπτώσεις όπου λειτουργεί πιο αποδοτικά. Ένα παράδειγμα είναι το εξής : Όταν ένα CUDA πρόγραμμα χρειαστεί για κάποιο λόγο να χρησιμοποιήσει μεγαλύτερο αριθμό GPUs, από αυτές που βρίσκονται εγκατεστημένες σε έναν κόμβο, τα κλασσικά συστήματα αποτυγχάνουν, εφόσον δεν χρησιμοποιηθεί κάποιο πρωτόκολλο ανταλλαγής μηνυμάτων όπως το MPI. Αυτό περιορίζει τον μέγιστο αριθμό GPUs που χρησιμοποιούνται από ένα πρόγραμμα, στον μέγιστο αριθμό που μπορούν να τοποθετηθούν σε έναν κόμβο, ο οποίος κατά τη στιγμή της συγγραφής αυτής της εργασίας είναι 4. Με το rGPU, ωστόσο, που χρησιμοποιεί το rCUDA, οι GPUs γίνονται εικονικές και ο περιορισμός αυτός αίρεται. Αν π.χ. η συστοιχία διαθέτει 64 GPUs, το πρόγραμμα CUDA μπορεί να τις εκμεταλλευτεί όλες και να τις χρησιμοποιήσει σαν να ήταν εγκατεστημένες όλες στο ίδιο μηχάνημα.

Παρόλο λοιπόν που η σύγκριση δεν γίνεται εκ των πραγμάτων επί ίσοις όροις, διεξήγαμε ένα πείραμα όπου συγκρίνουμε υποβολές προγραμμάτων CUDA μέσω του rGPU, με αντίστοιχες υποβολές μέσω του Torque, ο οποίος βρίσκεται εγκατεστημένος στα μηχανήματα. Να σημειώσουμε ότι το configuration του Torque που χρησιμοποιήσαμε δεν είναι GPU-aware, δηλαδή αν ζητήσουμε δύο κόμβους με GPUs και υποβάλλουμε δύο ξεχωριστές αιτήσεις για εκτέλεση CUDA, μοιραία τοποθετούνται στην ίδια GPU. Για να είναι δίκαιη λοιπόν η σύγκριση υποβάλλονταν κάθε φορά ξεχωριστά Torque PBS scripts για κάθε κόμβο με GPU. Τα αποτελέσματα για αυξανόμενο αριθμό clients , σε σύγκριση με υποβολές με το προτεινόμενο rGPU φαίνονται στο παρακάτω διάγραμμα.

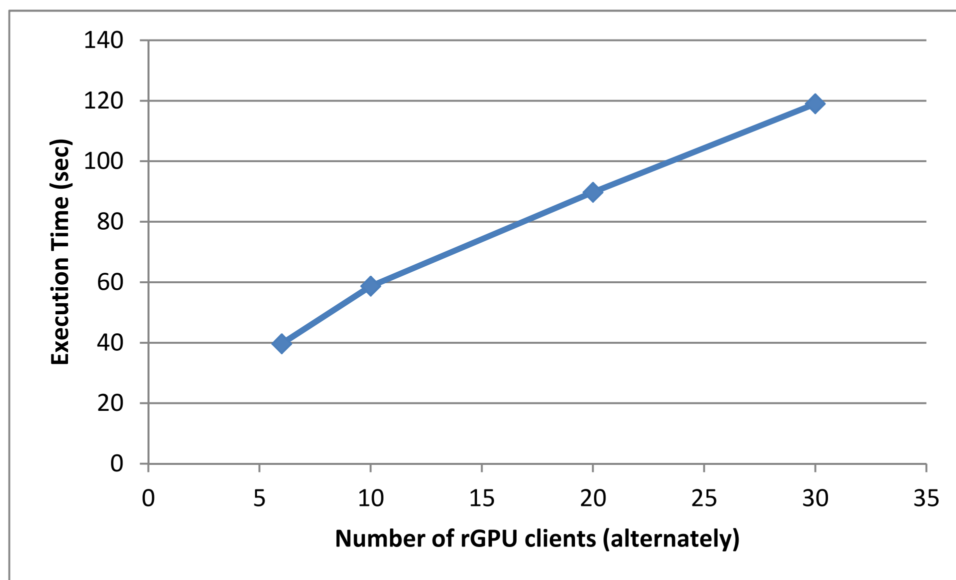


Διάγραμμα 9: Σύγκριση μέσου χρόνου εκτέλεσης μιας διεργασίας για υποβολή στο Torque και το rGPU για αυξανόμενο αριθμό rGPU clients.

Από το παραπάνω διάγραμμα συμπεραίνουμε ότι ο χρόνος εκτέλεσης με χρήση του rGPU είναι μια τάξη μεγέθους μικρότερος από την αντίστοιχη υποβολή μέσω του Torque. Τονίζουμε ξανά βέβαια ότι η σύγκριση δεν είναι απολύτως δίκαιη, αφού το σύστημα Torque είναι πολύπλοκο και οι χρόνοι μετρήσεις συχνά περιλαμβάνουν τυχαία αναμονή σε ουρές.

5.3 Μελέτη εκτέλεσης για διαφορετικά είδη φορτίων

Το τελευταίο πείραμα που εκτελέστηκε είχε σκοπό να μετρήσει την απόκριση του συστήματος, όταν η είσοδος ήταν φορτία διαφορετικών απαιτήσεων. Πιο συγκεκριμένα μελετήθηκαν δύο είδη προγραμμάτων : απαιτητικά σε υπολογισμούς και απαιτητικά σε προσπελάσεις μνήμης. Τα πρώτα είναι προγράμματα που περιλαμβάνουν μεγάλο αριθμό υπολογισμών που πρέπει να εκτελεστούν στην GPU, ενώ τα δεύτερα απαιτούν αρκετή μεταφορά δεδομένων από και προς τη μνήμη συσκευής. Υπενθυμίζεται ότι το rGPU, κατα την διαδικασία κατανομής των φορτίων, λαμβάνει υπόψη τόσο την χρησιμοποίηση της GPU, όσο και την διαθέσιμη μνήμη συσκευής. Παρακάτω φαίνεται ένα σενάριο κατά το οποίο ένας πελάτης επιθυμεί να τρέξει τέτοιου είδους φορτία εναλλάξ.



Διάγραμμα 10: Χρόνος εκτέλεσης για αυξανόμενο αριθμό εκτελούμενων rGPU Clients (εναλλάξ απαιτητικά σε μνήμη και σε υπολογισμούς).

Παρατηρούμε ότι το σύστημα έχει φυσιολογική συμπεριφορά και έχει τη δυνατότητα να υπερκαλύπτει την εκτέλεση απαιτητικών σε υπολογισμούς προγραμμάτων, με άλλα που είναι απαιτητικά σε μνήμη.

5.4 Καταλληλότητα ανάλογα με την παραλληλία του προγράμματος

Με σκοπό να αναδείξουμε την καταλληλότητα του rGPU για τα διαφορετικά είδη CUDA προγραμμάτων που πρόκειται να εκτελεστούν σε αυτό, χωρίζουμε τα τελευταία νοητά σε 4 κατηγορίες ανάλογα με τον βαθμό παραλληλίας τους και σχολιάζουμε την πιθανή συμπεριφορά του:

Χαμηλός βαθμός παραλληλίας

Πρόκειται για τα προγράμματα που δεν χρησιμοποιούν ιδιαίτερα ή και καθόλου την επιτάχυνση που προσφέρει η GPU. Σε τέτοιου είδους προβλήματα συνίσταται η χρήση των παραδοσιακών μεθόδων του HPC, δηλαδή εφαρμογές που βασίζονται σε κάποιο πρότυπο ανταλλαγής μηνυμάτων για CPUs όπως το MPI.

Υψηλός βαθμός παραλληλίας

Τα προγράμματα αυτά χρησιμοποιούν τις GPUs σε πολύ μεγάλο βαθμό και έχουν πολύ υψηλή αναλογία μεταξύ μεταφοράς δεδομένων και εκτέλεσης στην GPU. Επίσης δεν είναι κατάλληλες για χρήση με το σύστημα που προτείνουμε καθώς εισάγεται σημαντικό overhead λόγω της επικοινωνίας. Σε αυτή την περίπτωση, η μόνη λύση είναι να εξοπλίσουμε μερικούς κόμβους με όσο το δυνατόν περισσότερες GPUs και να τρέξουμε την εφαρμογή σε αυτούς.

Μέτριος βαθμός παραλληλίας

Πρόκειται για εφαρμογές που παρουσιάζουν ποσοστό παραλληλίας από 40 έως 80 %. Σε αυτή την περίπτωση οι GPUs χρησιμοποιούνται μόνο σε μερικά τμήματα της εφαρμογής, ενώ παραμένουν ανενεργές τον υπόλοιπο χρόνο, σπαταλώντας έτσι άσκοπα πόρους και ενέργεια. Σε αυτή την περίπτωση το rGPU μπορεί να φανεί χρήσιμο, καθώς μπορεί να κατανέμει αποτελεσματικά το φορτίο στις διαθέσιμες GPUs και να επικαλύψει την εκτέλεση των εφαρμογών.

Εφαρμογές που απαιτούν πολλαπλές GPUs

Μερικές εφαρμογές επεκτείνονται σε περισσότερες της μιας GPUs και παρόλο που συχνά οι μεταφορές δεδομένων μεταξύ αυτών αποτελούν στενωπό της επίδοσης, κλιμακώνονται ικανοποιητικά αν σχεδιαστούν προσεκτικά. Εδώ το προτεινόμενο rGPU είναι μονόδρομος, καθώς όπως αναφέραμε και προηγουμένως είναι η μοναδική λύση για να χρησιμοποιήσουμε περισσότερες GPUs, από όσες είναι εγκατεστημένες σε έναν κόμβο.

Κεφάλαιο 6

Συμπεράσματα και μελλοντικές κατευθύνσεις

Στην παρούσα διπλωματική εργασία παρουσιάσαμε και στη συνέχεια υλοποιήσαμε ένα αυτόνομο και ολοκληρωμένο κατανεμημένο σύστημα διαχείρισης των μονάδων επεξεργαστών γραφικών σε μια συστοιχία υπολογιστών, εν ονόματι rGPU. Το σύστημα αυτό βασίζεται τόσο στο ήδη υλοποιημένο rCUDA, το οποίο ουσιαστικά επιτρέπει τη δημιουργία εικονικών GPUs σε όλους τους κόμβους της συστοιχίας, όσο και σε ένα ειδικά σχεδιασμένο σύστημα παρακολούθησης των πόρων και λήψης αποφάσεων. Μπορεί να βρει εφαρμογή τόσο σε συστοιχίες με περιορισμένο αριθμό GPUs που πρέπει να εξυπηρετήσουν μεγάλο αριθμό πελατών, όπως π.χ. σε ένα ακαδημαϊκό ίδρυμα όπου οι σπουδαστές μελετούν τις GPUs στα πλαίσια ενός μαθήματος, όσο και σε μεγαλύτερες κλίμακες, όπου μειώνοντας τον αριθμό των GPUs μπορούμε να εξοικονομήσουμε τα κόστη απόκτησης, συντήρησης και ψύξης, ενώ παράλληλα να διατηρήσουμε τα ίδια επίπεδα υπολογιστικής ισχύος.

Αρχικά αξιολογήσαμε το rCUDA για να αποδείξουμε την καταλληλότητα του για ενσωμάτωση στο τελικό σύστημα. Έπειτα περιγράψαμε τις διαδικασίες σχεδίασης και υλοποίησης του συστήματος μας, καθώς και τις αρχιτεκτονικές αποφάσεις που αναγκαστήκαμε να πάρουμε. Το σύστημα που παρουσιάσαμε επιτυγχάνει στο να κατανέμει αποδοτικά τις διαφορετικές εργασίες προς εκτέλεση, χάρη στον ειδικά σχεδιασμένο χρονοδρομολογητή, ο οποίος λαμβάνει υπόψην του έναν αριθμό από παράγοντες που σχετίζονται με τον φόρτο στους επεξεργαστές γραφικών. Αυτοί περιλαμβάνουν : το ποσοστό χρησιμοποίησης του επεξεργαστή γραφικών, την θερμοκρασία του, την διαθέσιμη μνήμη συσκευής, τον αριθμό των σφαλμάτων, καθώς και τις γενικότερες δυνατότητες της GPU. Τέλος, αξιολογήσαμε το σύστημα με βάση κάποια προκαθορισμένα πειράματα που αντιπροσωπεύουν σενάρια λειτουργίας υπό πραγματικές συνθήκες. Αποδείξαμε ότι το κατανεμημένο rGPU κλιμακώνεται ικανοποιητικά και κατανέμει ομοιόμορφα τα υπό εκτέλεση φορτία στους διαθέσιμους πόρους. Η διαδικασία ρύθμισης και εκτέλεσης του είναι αρκετά απλή και δεν χρειάζονται ιδιαίτερες γνώσεις για την χρήση του.

Προφανώς, δεν διαθέτει την λειτουργικότητα και τις δυνατότητες ενός ολοκληρωμένου συστήματος διαχείρισης για συστοιχίες, ωστόσο σε συγκεκριμένες περιπτώσεις μπορεί να αποδώσει καλύτερα από ένα τέτοιο, αφού δεν λαμβάνει υπόψη του την GPU ως ένα δυαδικό πόρο, άλλα επιτρέπει την πολυπλεξία πολλών ροών εκτέλεσης σε αυτήν. Επίσης, είναι περιορισμένο σε μια μόνο αρχιτεκτονική : το CUDA της NVIDIA και μάλιστα σε συγκεκριμένες εκδόσεις αυτού. Γενικά πάντως πιστεύουμε ότι, ακόμα και σε αυτή την

πρωτόγονη μορφή του, μπορεί να φανεί χρήσιμο σε αρκετές περιπτώσεις καθώς και επίσης να επιδεχθεί πλήθος βελτιώσεων, οι οποίες περιγράφονται παρακάτω.

Προτάσεις για μελλοντική επέκταση

Αρχικά, το σύστημα μπορεί να επεκταθεί ώστε να υποστηρίζει και άλλες αρχιτεκτονικές GPGPU, πέραν του CUDA, όπως π.χ. το OpenCL. Έτσι θα μπορέσει να καλύψει όλες τις πιθανές εγκαταστάσεις σε κατανεμημένα συστήματα υπολογιστών. Η λειτουργία του συστήματος, ωστόσο, εξαρτάται άμεσα από το rCUDA, το οποίο όχι μόνο στοχεύει αποκλειστικά την αρχιτεκτονική του CUDA, αλλά είναι επίσης και κλειστού κώδικα. Έχοντας όμως γνώση της λειτουργίας του, μπορεί να αναπτυχθεί ένα αντίστοιχο σύστημα ανοικτού κώδικα που να φιλοξενεί όλες τις τεχνολογίες GPGPU. Στην πραγματικότητα, πραγματοποιήθηκε μια μικρή υλοποίηση ενός τέτοιου συστήματος, η οποία επισυνάπτεται μαζί με την παρούσα διπλωματική.

Βελτίωση θα μπορούσε να γίνει επίσης και στη διαδικασία με την οποία γίνεται η χρονοδρομολόγηση των εργασιών υπό εκτέλεση. Πιο συγκεκριμένα, υπάρχουν πολλοί παράγοντες που πρέπει να ληφθούν υπόψη για να κατανεμηθεί σωστά το φορτίο στις GPUs. Πέρα από τη χρησιμοποίηση (utilization) του επεξεργαστή της GPU και της μνήμης συσκευής, η οποία λαμβάνεται υπόψη, σημαντικό ρόλο παίζουν και : η τοπολογία των GPUs (προγράμματα που χρησιμοποιούν πολλαπλές GPUs θα πρέπει να τοποθετούνται σε "κοντινές" συσκευές), η θερμική συμπεριφορά τους (για αποφυγή επιπλοκών) καθώς και γνώση πιο συγκεκριμένων ρυθμίσεων της συσκευής. Επίσης, θα μπορούσαν να υποστηριχτούν και ανομοιογενή clusters , προσθέτοντας διαφορετικά βάρη στις GPUs, ανάλογα με τις δυνατότητες τους. Σε κάθε περίπτωση θα πρέπει να γίνει μια ολοκληρωμένη μελέτη για να προσδιοριστούν οι παράγοντες βαρύτητας καθενός από αυτά τα χαρακτηριστικά.

Ένας ακόμα τρόπος για να βελτιωθεί η απόδοση του συστήματος θα ήταν ο χρονοδρομολογητής να αποκτήσει γνώση των φορτίων που εκτελούνται κάθε φορά έτσι ώστε να έχει μεγαλύτερη ευελιξία πάνω τους. Για παράδειγμα θα μπορούσε να τοποθετεί φορτία που γνωρίζει ότι απαιτούν ελάχιστο υπολογιστικό χρόνο μπροστά από άλλα πιο χρονοβόρα στις ουρές, ώστε ο μέσος χρόνος αναμονής των πελατών να μειωθεί σημαντικά. Αυτό βέβαια προϋποθέτει την κατάλληλη είσοδο από τον χρήστη σχετικά με τους απαιτούμενους πόρους εκ των προτέρων όπως συμβαίνει με τα περισσότερα εμπορικά συστήματα διαχείρισης εργασιών για clusters.

Τέλος, η υλοποίηση του rGPU από μόνη της, επιδέχεται πολλές βελτιώσεις κυρίως σε θέματα απόδοσης και σταθερότητας. Η υλοποίηση που κατατίθεται είναι μεν λειτουργική, αλλά βρίσκεται σε αρκετά πρώιμο στάδιο και είναι επιρρεπής σε αστοχίες. Μια ιδέα θα ήταν για παράδειγμα η τροποποίηση του Hypervisor, ώστε να τρέχει και αυτός κατανεμημένα στο σύστημα και να μην αποτελεί πια μοναδικό σημείο ελέγχου, που συγκεντρώνει όλη την λειτουργικότητα. Σε κάθε περίπτωση ο κώδικας μπορεί να βελτιωθεί ώστε να έχει πιο σταθερή συμπεριφορά και να λάβει υπόψη του πολλά διαφορετικά σενάρια χρήσης.

Παράρτημα Α

Βιβλιοθήκη (Header File) του rGPU

```
/*******************************************
              Header File for rgpu
       Author: Antonis Karkatsoulis
              akarkat@cslab.ece.ntua.gr
*******************************************/

#include <stdlib.h>

#define MAX_NODES 64
#define MAX_GPUS 64
#define MONITOR_REFRESH 1
#define P_FACTOR 0.8

/* ------------------------------  Monitor modes  ------------------------*
** AVG : Keeps the average of some of the last measurements of utilizations
*
** NOW : Returns the utilization the moment it is requested            */

enum monitor_mode
{
       AVG,
       NOW

};

/* ----------------------------   GPU struct  ------------------------------
*
** Contains info about the GPUs available in the cluster :             *
** node_name : The name of the node the GPU is installed in.
*
** id       : A local id for different GPUs on the same node.
*
** address   : The IP address of the node + the port that connects to this
*
**           specific device.                                          *
** proc_util : The GPU processor utilization in %.                     *
** mem_util  : The GPU memory utilization in %.
*
** -----------------------------------------------------------------------
-*/

typedef struct gpu {
       char node_name[255];
       unsigned int id;
       unsigned int gen;
       char address[32];
       int proc_util;
       int mem_util;
       int temp;  //Temp and ecc measurents available only for newer Tesla
Models
       int ecc;
} gpuinfo;

/* -------------------------- Node struct  -------------------------------
*
** Contains info about all the nodes present in the cluster :
*
** id        : A specific id for each node.                            *
```

```
** name       : The node hostname.                                   *
** ip         : The IP address of the node.                          *
** avail_gpus : The available COMPATIBLE NVIDIA gpus.                *
** screen_ids : A table with all the open screens of this node.
*
** num_screens: The number of open screens for this node.            *
** ---------------------------------------------------------------------------
-*/

typedef struct node {
	unsigned int id;
	char name[255];
	char ip[32];
	int avail_gpus;
} nodeinfo;

/* Function prototypes */
void * monitor (gpuinfo **);
int usage (FILE *, char *);
void parse_nodefile (nodeinfo **, gpuinfo **, int *, int *);
int add_screen_id (nodeinfo **, int, int);
int * get_utils(gpuinfo **, int);
int export_vars (nodeinfo **, int, char *);
void inthandler (int);
int CheckForData (int);
void print_node_info (nodeinfo **, gpuinfo **, int, int);
```

Παράρτημα Β

Αρχείο Σεναρίου (script) για την παραγωγή του αρχείου ρυθμίσεων nodes.config

```
## This shell script creates the necessary configuration file
## when run in a node of a cluster that has access to every other
node
## It also tries to recognize all the GPU resources available.
## Tested only on cslab@ntua cluster. The user should always check
the final
## configuration file produced for errors!
## Note :  This script uses SSH to connect to the machines and obtain
information about the GPUS
##       SSH keys are required in all the machines for the script to
run smoothly.


#!/bin/bash

queue -a | awk '{if (NR>3) print $1}' > nodes

id=0
for line in $(cat nodes)
do
      name=$line
      echo "Obtaining information about node \"$name\".."
      ip=`nslookup $name | awk '{if (NR==5) print $2}'`
      echo "Checking GPU configuration.."
      nr_gpus=`ssh $name 'nvidia-smi -L 2>/dev/null | grep GPU | wc -
l'`
      if [ $? -eq 255 ]
      then
            continue
      fi
      echo "Found : $nr_gpus GPUs"
      echo "Done"
      echo $id $name $ip $nr_gpus >> nodes.config
      id=`expr $id + 1`

      echo "Complete"
done

rm -f nodes
```

Παράρτημα Γ

Εναλλακτικός τρόπος υλοποίησης του συστήματος

Ο τρόπος υλοποίησης που προτάθηκε εξαρτάται σχεδόν αποκλειστικά από έργα ιδιόκτητου κώδικα τα οποία πολλές φορές αποτελούν τροχοπέδη για την περαιτέρω ανάπτυξη και τη βελτιστοποίηση από τον προγραμματιστή. Πιο συγκεκριμένα τόσο το rCUDA, όσο και το πρόγραμμα οδήγησης (Driver) της NVIDIA για το CUDA είναι κομμάτια κλειστού κώδικα και στο κοινό εκτίθενται μόνο μέσω κάποιων APIs.

Υπάρχουν, ωστόσο, εναλλακτικές λύσεις ανοικτού κώδικα που αντικαθιστούν τα παραπάνω συστήματα και αποδίδουν τα ίδια αποτελέσματα. Όσον αφορά το rCUDA, τονίσαμε στο Κεφάλαιο 2 ότι υπάρχουν αρκετές άλλες λύσεις που υλοποιούν την λειτουργία της απομακρυσμένης εκτέλεσης CUDA κώδικα. Η πιο κατάλληλη ίσως είναι το gVirtuS [11] , το οποίο επιτρέπει σε μια εικονική μηχανή (VM) να έχει πρόσβαση σε GPUs με ένα διαφανή τρόπο και με ελάχιστο επιπλέον κόστος. Το gVirtuS είναι ανεξάρτητο τεχνολογίας GPU, δικτύου διασύνδεσης και συστήματος παρακολούθησης (Hypervisor) των VMs. Ο τρόπος με τον οποίον το gVirtuS υλοποιεί την απομακρυσμένη εκτέλεση είναι παρόμοιος με αυτόν του rCUDA, δηλαδή χωρίζει τη στοίβα εκτέλεσης του CUDA προγράμματος μεταξύ πελάτη και εξυπηρετητή και χρησιμοποιεί μια δομή που την ονομάζει Communicator για να υλοποιήσει ταχεία επικοινωνία μεταξύ των δύο πλευρών. Υποστηρίζει πληθώρα από Hypervisors όπως Xen, VMware και KVM/QEMU.

Όσον αφορά τον Driver της NVIDIA, υπάρχει και εδώ εναλλακτική λύση και ονομάζεται *Nouveau Driver* [12] . Ο Driver αυτός έχει δημιουργηθεί με αντίστροφη μηχανική από τον αυθεντικό της NVIDIA και επιτρέπει στο λειτουργικό σύστημα να αλληλεπιδρά με τις κάρτες της εταιρείας μέσω ανοικτού κώδικα. Την διαχείριση του Nouveau την έχει αναλάβει το X.org Foundation μαζί με το Freedesktop.org και η άδεια είναι από το MIT. Χρησιμοποιείται σαν ο προεπιλεγμένος Driver για αρκετές διανομές του Linux, όπως τα Ubuntu, το Fedora κ.ο.κ.

Ένα τελευταίο σύστημα το οποίο θα μπορούσε να χρησιμοποιηθεί προς αυτό το σκοπό είναι το *Gdev* [13]. Το σύστημα αυτό παρέχει επιθυμητές λειτουργίες, όπως το Virtualization των GPUs και υλοποίηση του CUDA Driver API, ενώ χρησιμοποιεί τον Nouveau Driver που αναφέραμε προηγουμένως ως υποδομή. Έτσι χρησιμοποιώντας το Gdev μπορεί κάποιος να τροποποιήσει το ανοικτού πλέον CUDA Driver API και να προσθέσει το κομμάτι της επικοινωνίας μεταξύ πελάτη και GPU εξυπηρετητή, διατηρώντας τον Hypervisor που προτείνεται στην παρούσα διπλωματική εργασία για την διαχείριση των εικονικών GPUs.

Βιβλιογραφία

[1] M. Technologies, «Introduction to Infiniband».

[2] J. J. E. G. S. M. T. S. Volodymyr V. Kindratenko, «GPU Clusters for High-Performance Computing».

[3] «NVIDIA GPUDirect,» [Ηλεκτρονικό]. Available: https://developer.nvidia.com/gpudirect.

[4] «Backfill Scheduling,» [Ηλεκτρονικό]. Available: http://docs.adaptivecomputing.com/maui/8.2backfill.php.

[5] «Linux Checkpoint/Restart,» [Ηλεκτρονικό]. Available: https://ckpt.wiki.kernel.org/index.php/Main_Page#Project_Info.

[6] IBM, «IBM Platform Session».

[7] J. Duato, A. Pena, F. Silla, R. Mayo και E. Quintana-Ortí, «rCUDA: Reducing the number of GPU-based accelerators in high performance clusters».

[8] A. F. J. R. E. Q.-O. J.Duato, «A new Approach to rCUDA».

[9] A. J. P. F. S. R. M. E. S. Q.-O. Jose Duato, «Modeling the CUDA Remoting Virtualization Behavior in High Performance Networks».

[10] «NVIDIA System Management Interface,» [Ηλεκτρονικό]. Available: https://developer.nvidia.com/nvidia-system-management-interface.

[11] R. M. G. A. G. C. Giulio Giunta, «A GPGPU Transparent Virtualization Component for High Performance Computing Clouds».

[12] T. N. Driver. [Ηλεκτρονικό]. Available: http://nouveau.freedesktop.org/wiki/.

[13] M. M. C. M. a. S. B. Shinpei Kato, «Gdev: First-Class GPU Resource Management in the Operating System».

[14] «NVIDIA GPUDirect,» NVIDIA, [Ηλεκτρονικό]. Available: https://developer.nvidia.com/gpudirect.

[15] «General-purpose computing on graphics processing units,» [Ηλεκτρονικό]. Available: http://en.wikipedia.org/wiki/General-purpose_computing_on_graphics_processing_units.

[16] IBM, «Improving the efficiency of GPU clusters».

Λίστα Εικόνων

Εικόνα 1: Η χρονική εξέλιξη της υπολογιστικής τεχνολογίας, από πλευράς επεξεργασίας. .. 13
Εικόνα 2: Οι διαφορετικές Αρχιτεκτονικές Μνήμης. 18
Εικόνα 3: Επεκτασιμότητα (Scalability) για τις διάφορες τεχνολογίες παράλληλης και μη επεξεργασίας.. 19
Εικόνα 4: Κατηγοριοποίηση Αρχιτεκτονικών κατά Flynn. 20
Εικόνα 5: Αρχιτεκτονικές Κοινής Μνήμης (a) UMA και (b) NUMA....................................... 21
Εικόνα 6: Αρχιτεκτονική Κατανεμημένης Μνήμης 22
Εικόνα 7: Υβριδική Αρχιτεκτονική 23
Εικόνα 8 : Άποψη του κυκλώματος του επεξεργαστή Kepler της Nvidia 24
Εικόνα 9 : Αρχιτεκτονική μιας Fermi GPU. 24
Εικόνα 10 :Η ιεραρχία μνήμης στην αρχιτεκτονική Fermi.. 25
Εικόνα 11: Η πλήρης αρχιτεκτονική ενός Streaming Multiprocessor (SM), στην αρχιτεκτονική Fermi.. 26
Εικόνα 12: Το υπολογιστικό μοντέλο του GPGPU. 27
Εικόνα 13: Η οργάνωση των υπολογιστικών στοιχείων στο CUDA...................................... 32
Εικόνα 14: Παράδειγμα συγγραφής προγράμματος σε CUDA C με τις επεκτάσεις. 33
Εικόνα 15: Η στοίβα λογισμικού του CUDA... 34
Εικόνα 16: Ενδεικτική πορεία μεταγλώττισης και σύνδεσης ενός CUDA προγράμματος. 35
Εικόνα 17: Γενικότερο διάγραμμα των σταδίων της μεταγλώττισης ενός CUDA προγράμματος.. 36
Εικόνα 19 : Παράδειγμα κατανεμημένου συστήματος διασυνδεδεμένου με Infiniband....... 39
Εικόνα 20: Η αρχιτεκτονική του rCUDA... 43
Εικόνα 21: Ακολουθιακό διάγραμμα των επικοινωνιών του rCUDA κατα τη διάρκεια εκτέλεσης ενός πολλαπλασιασμού πινάκων C=A X B. 44
Εικόνα 22: Δημιουργία εικονικών GPUs από το rCUDA. 45
Εικόνα 23 : Παραδείγματα Κατανεμημένων Συστημάτων.. 54
Εικόνα 24 : Παράδειγμα ενός κατανεμημένου συστήματος 3 επιπέδων. 55
Εικόνα 25: Διάγραμμα που απεικονίζει την αρχιτεκτονική του προτεινόμενου συστήματος.56
Εικόνα 26 : Διαγράμματα κλάσης των δύο βασικών δομών του επόπτη. 64
Εικόνα 27: Παράδειγμα εκτέλεσης του συστήματος σε μέρος του cluster του εργαστηρίου που διαθέτει 2 GPUs. 67