ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΑΤΡΩΝ
UNIVERSITY OF PATRAS

Πολυτεχνική Σχολή

Τμήμα Μηχανικών Η/Υ & Πληροφορικής

Διπλωματική Εργασία

Σχεδίαση και υλοποίηση ενός πλαισίου λογισμικού για εξαντλητικές αναζητήσεις σε συστήματα κατανεμημένης μνήμης με συνεπεξεργαστές

Κυριακόπουλος Χρήστος

Α.Μ. 1047204

Επιβλέπων

Γαλλόπουλος Ευστράτιος, Καθηγητής

Μέλος Επιτροπής Αξιολόγησης

Σιούτας Σπύρος, Καθηγητής

Μέλος Επιτροπής Αξιολόγησης

Ψαράκης Εμμανουήλ

Συνεξεταστής

Βενέτης Ιωάννης, Καθηγητής

Πάτρα, 2020

© Copyright συγγραφής Κυριακόπουλος Χρήστος, 2020

© Copyright θέματος Κυριακόπουλος Χρήστος, 2020

Με επιφύλαξη παντός δικαιώματος. All rights reserved.

Η έγκριση της διπλωματικής εργασίας από το Τμήμα Μηχανικών Ηλεκτρονικών Υπολογιστών & Πληροφορικής του Πανεπιστημίου Πατρών δεν υποδηλώνει απαραιτήτως και αποδοχή των απόψεων του συγγραφέα εκ μέρους του Τμήματος.

Πριν τη παρουσίαση της διπλωματικής μου εργασίας, θα ήθελα να ευχαριστήσω…

…τους καθηγητές που με βοήθησαν με τη διπλωματική εργασία: τον κ. Γαλλόπουλο Ευστράτιο, Πρόεδρο του ΤΜΗΥΠ, χάρη στον οποίο έμαθα να λύνω σημαντικά προβλήματα που δεν ήξερα ότι υπάρχουν στον κόσμο των υπολογιστών, και τον κ. Βενέτη Ιωάννη, που με ώθησε στον τομέα της παράλληλης επεξεργασίας στον οποίο και έγινε η εν λόγω εργασία,

… το Τμήμα Μηχανικών Η/Υ και Πληροφορικής και τους καθηγητές του, για την υπερπροσπάθεια που καταβάλουν ώστε να προχωρήσουν την επιστήμη μας ένα βήμα μπροστά,

…και φυσικά όλους τους δικούς μου ανθρώπους, τους φίλους μου, την οικογένειά μου, που με στήριξαν όταν το χρειάστηκα, με βοήθησαν χωρίς να το ζητήσω, και πίστεψαν σε μένα, περισσότερο απ' ότι εγώ.

Περίληψη

Σε πολλές επιστήμες υπάρχουν προβλήματα που δεν έχουν κλειστή λύση, δε μπορούν δηλαδή να λυθούν ακολουθώντας μία πεπερασμένη ακολουθία αυστηρών μαθηματικών βημάτων. Πολλές από αυτές τις περιπτώσεις απαιτούν εξαντλητική αναζήτηση, ώστε να ελεγχθούν οι υποψήφιες λύσεις του μαθηματικού μοντέλου μέχρι να βρεθεί μία ή περισσότερες αποδεκτές λύσεις, ανάλογα με τις ανάγκες της εφαρμογής. Σχεδόν πάντα όμως ο αριθμός των λύσεων είναι τόσο μεγάλος που δεν αρκεί η χρήση ενός απλού επεξεργαστή και πρέπει να χρησιμοποιήσουμε άλλες τεχνικές. Πέραν της χρήσης "έξυπνων" τρόπων για την μείωση της υπολογιστικής πολυπλοκότητας, σήμερα πλέον είναι εφικτό και πρακτικό, ακόμα και για τον μέσο χρήστη, να χρησιμοποιήσει παράλληλη επεξεργασία, καθώς αυτή πλέον υπάρχει σε σύγχρονα laptops, σύγχρονους σταθμούς εργασίας, και βέβαια σε πολύ μεγάλο βαθμό σε many-core συστήματα όπως οι κάρτες γραφικών. Στην εργασία αυτή αξιοποιείται η τελευταία από αυτές τις επιλογές, και αναπτύσσεται ένα ευέλικτο πλαίσιο για την αποτελεσματική χρήση καρτών γραφικών με τεχνολογία CUDA σε κατανεμημένα συστήματα, για την εξαντλητική αναζήτηση σε πολυδιάστατους χώρους. Το Distributed Exhaustive Search Framework (DES Framework) υλοποιεί ένα ιεραρχικό σύστημα επεξεργαστικών στοιχείων σε 2 επίπεδα, χρησιμοποιώντας MPI και OpenMP. Αξιοποιεί πολυπύρηνους επεξεργαστές και σύγχρονες κάρτες γραφικών που υποστηρίζουν GPGPU μέσω CUDA, και δυναμικό loop self-scheduling για τη μεγιστοποίηση της απόδοσης των clusters με ανομοιόμορφη επεξεργαστική ισχύ. Το λογισμικό παρέχει μια εύκολη και ευέλικτη διεπαφή, μέσω της οποίας οι χρήστες του μπορούν να το χρησιμοποιήσουν δίνοντας μόνο τις βασικές παραμέτρους και ένα μικρό τμήμα κώδικα, χωρίς να έχουν καμία επαφή με την επικοινωνία τον συστημάτων και τον διαμοιρασμό του φόρτου εργασίας.

Στα πλαίσια της εργασίας αναπτύχθηκε και χρησιμοποιήθηκε μία παραλλαγή της ήδη γνωστής τεχνικής χρονοδρομολόγησης βρόγχων HPLS, η οποία χρησιμοποιεί δεδομένα πραγματικού χρόνου ώστε να υπολογίσει το πλήθος των στοιχείων προς ανάθεση. Επίσης προτείνεται η τεχνική «Slow-start μέγιστης ανάθεσης» η οποία στοχεύει να καταπολεμήσει μία βασική αδυναμία αρχικοποίησης των τεχνικών δυναμικού loop-scheduling, θέτοντας ένα άνω όριο στο πλήθος στοιχείων των πρώτων αναθέσεων. Το framework αξιοποιήθηκε σε συγκεκριμένη εφαρμογή που προέρχεται από την γεωφυσική και ειδικότερα συγκρίθηκε με πρόσφατες μεθόδους "τοπολογικής αντιστροφής" για την επίλυση προβλημάτων ελαστικής μετατόπισης σεισμικών ρηγμάτων. Αυτές οι μέθοδοι υλοποιήθηκαν πρόσφατα σε κάρτες

γραφικών και η αποτελεσματικότητά τους εξαρτάται άμεσα από την υλοποίηση της εξαντλητικής αναζήτησης. Τα αποτελέσματα της πειραματικής αξιολόγησης έδειξαν ότι το προϊόν της εργασίας μπορεί να είναι ωφέλιμο μόνο σε μεγάλη κλίμακα με πολλά υποσυστήματα, καθώς ανέδειξαν κάποιες αδυναμίες του σε μικρά workloads λόγω του υπολογιστικού overhead που απαιτείται για την υλοποίησή του.

Abstract

In many sciences there are problems that don't have a closed form solution, which means that they can't be solved by following a finite series of mathematical steps. Most of these cases require an exhaustive search, to check every possible solution of a model until enough acceptable solutions are found, depending on the application. Almost always, the number of the candidate solutions is so large, that the use of a single processor is not enough, so we need to employ other techniques. Other than the use of "smart" ways to reduce the computational complexity, today it is possible and practical, even for the average user, to use parallel computing, as it is available in modern laptops, workstations, and of course in many-core systems such as graphics cards. In this thesis the last option was utilized, and a flexible framework was developed, which allows the efficient use of GPUs in distributed systems so that an exhaustive search in such large search domains is possible. Distributed Exhaustive Search Framework (DES Framework) implements a 2-layer hierarchical system of computing elements, using MPI and OpenMP. It takes advantage of multicore CPUs and modern GPUs capable of GPGPU through CUDA, and uses dynamic loop self-scheduling to maximize the efficiency of heterogeneous computer clusters. The framework provides a simple and flexible interface, which can be used by giving only a few basic parameters, as well as an implementation of the model in question. The user never needs to worry about the underlying communication of the computers or the scheduling of the computation.

A new variation of the already known loop-scheduling technique HPLS was developed and used in this thesis, which uses real-time data to calculate the number of elements to be assigned to a computing node. Additionally, a new technique called "Maximum assignment Slow-start" is proposed, which aims to mitigate an important weakness of dynamic loop-scheduling techniques' initialization, by setting an upper limit to the number of elements that can be assigned during the first rounds of each computing node. The designed framework was evaluated in a specific application from geophysics, specifically it was compared to recent methods of topological inversion for solving problems of elastic displacement of seismic faults. These methods were recently implemented using graphics cards, and their efficiency depends on the implementation of the exhaustive search. The results of the experimental evaluation showed that the product of this thesis is beneficial only in large scale with many subsystems, as they revealed weaknesses in smaller workloads, because of the computational overhead of the implementation.

Περιεχόμενα

Λίστα Εικόνων

Λίστα Πινάκων

Λίστα Εξισώσεων

Λίστα Αλγορίθμων

Συντομογραφίες

API	Application Programming Interface
CPU / KME	Central Processing Unit / Κεντρική Μονάδα Επεξεργασίας
CUDA	Compute Unified Device Architecture
DMA	Direct Memory Access
GPGPU	General Purpose computing on Graphics Processing units
GPU	Graphics Processing Unit
LSD	Least Significant Digit
MPI	Message Passing Interface
NUMA	Non-Uniform Memory Access
OpenMP	Open Multi-Processing
OS	Operating System
PCIe	Peripheral Component Interconnect Express
PF	Performance Function
PR	Performance Ratio

Γλωσσάρι ή Απόδοση Όρων

API	Η διεπαφή που προσφέρει μία βιβλιοθήκη μιας γλώσσας προγραμματισμού που επιτρέπει αιτήματα ή ανταλλαγή δεδομένων από τον προγραμματιστή που τη χρησιμοποιεί.
DES – Distributed Exhaustive Search	Το πλαίσιο λογισμικού που υλοποιήθηκε στα πλαίσια της διπλωματικής εργασίας, και επιτρέπει τη χρήση κατανεμημένων συστημάτων με συνεπεξεργαστές για την εκτέλεση εξαντλητικής αναζήτησης.
Debugging	Η διαδικασία εύρεσης λαθών σε κάποιον κώδικα και διόρθωσής τους.
Framework	Μία αφαίρεση ενός λογισμικού που προσφέρει μία γενική λειτουργικότητα και απαιτεί πρόσθετο κώδικα για να λειτουργήσει.
GPGPU	Η χρήση καρτών γραφικών για υπολογισμούς γενικού σκοπού.
GPU Kernel	Ένα τμήμα κώδικα που εκτελεί κάθε GPU thread (βλ. Λεπτομέρειες υπολογισμού σε GPU).
GPU Stream	Μία ουρά λειτουργιών που εκτελούνται στη κάρτα γραφικών (βλ. GPU Streams).
GPU Thread	Ένα υπολογιστικό νήμα που εκτελείται σε μία κάρτα γραφικών.
Hot spot δικτύου	Ένας κόμβος ενός δικτύου που υπόκειται σε μεγαλύτερη κίνηση από τους υπόλοιπους.
Instance	Ένα στιγμιότυπο μίας κλάσης.
MPI rank	Ένα μοναδικό ID που ανατίθεται από το πρότυπο MPI σε κάθε διεργασία του.
On-the-fly	«Κατά τη διάρκεια της εκτέλεσης»
Overhead	Οποιοσδήποτε χρόνος αφιερώνεται σε μία διαδικασία πέρα από τις ζητούμενες πράξεις (αρχικοποιήσεις, μεταφορές δεδομένων, κλπ.).
PCIe	Ένα interface των υπολογιστών για τη γρήγορη επικοινωνία καρτών επέκτασης με το υπόλοιπο σύστημα.

Profiling	Η διαδικασία χρονομέτρησης των επιμέρους τμημάτων μίας εφαρμογής με σκοπό την επιτάχυνσή της.
Single point of failure	Κομβικό σημείο μίας διαδικασίας που αν αποτύχει τότε αποτυγχάνει όλη η διαδικασία.
Struct	Μία δομή μιας γλώσσας προγραμματισμού που επιτρέπει τη διαχείριση πολλών μεταβλητών ως μία.
Throughput	Ο ρυθμός εκτέλεσης πράξεων από κάποιον επεξεργαστή.
Virtual κλάση	Μία κλάση που περιέχει τουλάχιστον μία virtual συνάρτηση.
Virtual συνάρτηση	Μία συνάρτηση που δίνεται μόνο η δήλωσή της και όχι ο ορισμός της, και καλείται να υλοποιηθεί (ορισθεί) από μία υποκλάση.
Διασωλήνωση (pipelining)	Η τεχνική διαίρεσης μίας εντολής σε υποδιαδικασίες με σκοπό την ταυτόχρονη εκτέλεση των επιμέρους υποδιαδικασιών παράλληλα.
Εξαντλητική αναζήτηση	Η εύρεση και αξιολόγηση *κάθε* πιθανής λύσης ενός προβλήματος (βλ. Σημασία του προβλήματος).
Επεξεργαστικό στοιχείο	Οποιοδήποτε υλικό ικανό για ψηφιακή επεξεργασία δεδομένων, όπως ένας επεξεργαστής ή μία κάρτα γραφικών.
Καταχωρητής	Ένα σύνολο από flip-flops που χρησιμοποιούνται εντός του επεξεργαστή ως πολύ γρήγορες προσωρινές μνήμες.
Κρυφή μνήμη (cache)	Μνήμη εντός του επεξεργαστή, πιο γρήγορη από τη κύρια μνήμη, με σκοπό τη γεφύρωση του χάσματος ταχύτητας μεταξύ επεξεργαστή και κύριας μνήμης (βλ. Shared memory).
Μεταφραστής κώδικα (compiler)	Ένα πρόγραμμα που μεταφράζει πηγαίο κώδικα μίας γλώσσας προγραμματισμού σε εκτελέσιμο κώδικα που "καταλαβαίνει" ένας επεξεργαστής.
Συνάρτηση αξιολόγησης	Μία συνάρτηση που δέχεται ως είσοδο ένα σύνολο παραμέτρων ενός μοντέλου (σημείο του χώρου αναζήτησης) και το εξετάζει επιστρέφοντας ένα αριθμητικό αποτέλεσμα.
Συνέπεια μνήμης	Η εξασφάλιση ότι κατά την παράλληλη εκτέλεση εντολών, *κάθε* επεξεργαστής έχει πρόσβαση στα πιο πρόσφατα δεδομένα.
Υπολογιστικό νήμα (thread)	Μία ροή προγράμματος που αποτελείται από εντολές που πρέπει να εκτελεστούν με μία αυστηρά καθορισμένη σειρά.
Χώρος αναζήτησης	Το σύνολο των διανυσμάτων που σχηματίζονται αν θεωρήσουμε ότι κάθε παράμετρος που αναζητούμε είναι ένας ανεξάρτητος όρος ενός διανύσματος (βλ. Αναπαράσταση μοντέλου και χώρου αναζήτησης).

1

Εισαγωγή

1.1 Σημασία του προβλήματος

Σε πολλές επιστήμες υπάρχουν προβλήματα και περίπλοκα μοντέλα που δεν έχουν κλειστή λύση. Αυτό σημαίνει ότι δε μπορούν να λυθούν ακολουθώντας μία σειρά από πεπερασμένα μαθηματικά βήματα. Τέτοια προβλήματα απαιτούν εξαντλητική αναζήτηση, δηλαδή τη παραγωγή και τον έλεγχο κάθε υποψήφιας λύσης μέχρι να βρεθεί μία ή περισσότερες αποδεκτές, ανάλογα με την εφαρμογή. Πολλές φορές, το πλήθος των υποψήφιων λύσεων είναι τόσο μεγάλο που είναι αδύνατο για έναν άνθρωπο να τις ελέγξει όλες σε ρεαλιστικό χρόνο. Σε τέτοιες περιπτώσεις γίνεται προσπάθεια μείωσης των υποψήφιων λύσεων, αποκλείοντας κάποιες χωρίς να χρειαστεί να τις ελέγχουμε. Αυτό μπορεί να γίνει χρησιμοποιώντας ευρετικές μεθόδους, όπως ο εκ των προτέρων αποκλεισμός λύσεων που δεν είναι δυνατές, ή η απόρριψη όμοιων/κοντινών/συμμετρικών λύσεων όταν απορριφθεί μία άλλη. Ακόμα κι έτσι όμως δεν υπάρχει εγγύηση ότι μπορεί να βρεθεί λύση σε λογικό χρονικό πλαίσιο από έναν άνθρωπο. Ως αποτέλεσμα, ο άνθρωπος στράφηκε στη χρήση μηχανών για τη λύση τέτοιων προβλημάτων.

Από τη πρώτη κιόλας μοντελοποίηση του σημερινού ηλεκτρονικού υπολογιστή από τον Alan Turing το 1936 [1], ξεκίνησε η χρήση του για την επίλυση προβλημάτων που απαιτούσαν εξαντλητική αναζήτηση. Μία από τις πρώτες, και ίσως η πιο γνωστή, ήταν η μηχανή που χρησιμοποιήθηκε για την αποκωδικοποίηση των μηνυμάτων της μηχανής Enigma κατά τον $2^{ο}$ παγκόσμιο πόλεμο. Στην επιστήμη της κρυπτογραφίας χρησιμοποιούνται *κλειδιά* για τη κρυπτογράφηση και αποκρυπτογράφηση μηνυμάτων, τα οποία, βάση μαθηματικών

αποδείξεων, μπορούν να βρεθούν μόνο με τυχαία αναζήτηση, δοκιμάζοντας δηλαδή τυχαία κλειδιά μέχρι να βρεθεί το σωστό. Όμοια προβλήματα που εμφανίζονται στην επιστήμη των υπολογιστών συμπεριλαμβάνουν το πρόβλημα του πλανόδιου πωλητή (TSP, εύρεση βέλτιστης σειράς επίσκεψης πόλεων για την ελαχιστοποίηση της συνολικής απόστασης), την εύρεση της μεγαλύτερης κλίκας σε ένα γράφημα, κ.α. Επίσης, τέτοια προβλήματα συναντώνται στη βιοπληροφορική (αναγνώριση και εύρεση μοτίβων σε ακολουθίες DNA), και στον τομέα της τεχνητής νοημοσύνης, καθιστώντας έτσι την εξαντλητική αναζήτηση με χρήση υπολογιστών απαραίτητη.

Τα προβλήματα που μας απασχολούν μπορούμε να τα χωρίσουμε σε 2 κατηγορίες, ανάλογα με τον χώρο αναζήτησής τους (το πεδίο ορισμού τους):

- **Πεπερασμένος χώρος αναζήτησης**: Το πλήθος των υποψήφιων λύσεων είναι πεπερασμένο, όπως σε ένα πρόβλημα μεταθέσεων (π.χ. στο πρόβλημα TSP αναζητούμε τη βέλτιστη σειρά επίσκεψης πόλεων), ικανοποίησης συνθηκών (SAT, όπου αναζητούμε ένα σύνολο λογικών μεταβλητών (true/false) που ικανοποιούν έναν τύπο άλγεβρας Bool), ή στη λύση ενός συστήματος εξισώσεων όπου οι μεταβλητές παίρνουν ακέραιες τιμές σε ένα κλειστό διάστημα.
- **Συνεχής χώρος αναζήτησης**: Η λύση εκφράζεται ως ένα διάνυσμα πραγματικών αριθμών, άρα υπάρχουν άπειρες υποψήφιες λύσεις. Ένα τέτοιο πρόβλημα μπορεί να είναι ένα σύστημα εξισώσεων που οι μεταβλητές του παίρνουν πραγματικές τιμές.

Είναι προφανές ότι τα προβλήματα με συνεχή χώρο αναζήτησης έχουν άπειρες υποψήφιες λύσεις και είναι αδύνατο να τις ελέγξουμε όλες σε πεπερασμένο χρόνο. Επίσης, οι υπολογιστές που έχουμε στη διάθεσή μας δεν έχουν αριθμητική άπειρης ακρίβειας, άρα δε μπορούν να σαρώσουν ένα συνεχές διάστημα, διότι δε μπορούν να παραστήσουν κάθε πιθανό αριθμό μέσα σε αυτό. Για να λύσουμε τέτοια προβλήματα με τη χρήση υπολογιστών κάνουμε διακριτοποίηση του χώρου αναζήτησης και ορίζουμε ένα περιθώριο σφάλματος (threshold) για τη λύση. Η διακριτοποίηση του χώρου αναζήτησης γίνεται διαιρώντας ένα συνεχές διάστημα σε διακριτά σημεία και εξετάζοντας μόνο αυτά. Συνήθως αυτά τα σημεία ισαπέχουν και η απόστασή τους ρυθμίζεται ανάλογα με το πρόβλημα που λύνεται, την ακρίβεια που επιθυμούμε, κλπ. Ως προς το περιθώριο σφάλματος για τη λύση, είναι προφανές ότι αφού δε μπορούμε να εξετάσουμε όλα τα σημεία του άπειρου χώρου αναζήτησης τότε η πιθανότητα να βρεθεί η ακριβής λύση είναι απειροελάχιστη. Άρα θα πρέπει να ορίσουμε ένα διάστημα γύρω

από το επιθυμητό αποτέλεσμα, ώστε να έχουμε ελπίδες να βρούμε μία λύση που μας ικανοποιεί. Για παράδειγμα, αν αναζητούμε τη λύση του $f(\vec{x}) = 0$ θα πρέπει να ψάξουμε για ένα διάνυσμα $\vec{x}$ τέτοιο ώστε $-e \leq f(\vec{x}) \leq e$ όπου e ένας μικρός θετικός αριθμός.

1.2 Συνεισφορά της Διπλωματικής Εργασίας

Το αποτέλεσμα της παρούσας εργασίας είναι το DES (Distributed Exhaustive Search) Framework, ένα πλαίσιο λογισμικού για συστήματα Linux, για εξαντλητική αναζήτηση πολυδιάστατων σημείων σε συστήματα κατανεμημένης μνήμης με συνεπεξεργαστές. Αξιοποιεί πολλαπλούς πολυπύρηνους κεντρικούς επεξεργαστές και πολλαπλές κάρτες γραφικών τεχνολογίας CUDA σε ξεχωριστά συστήματα συνδεδεμένα μεταξύ τους, για να εκτελέσει μία συνάρτηση αξιολόγησης πάνω σε κάθε σημείο ενός πολυδιάστατου χώρου αναζήτησης. Η συνάρτηση αξιολόγησης αποτελεί την υλοποίηση του μοντέλου του προβλήματος που θέλουμε να λύσουμε. Το DES Framework ορίζει την αναπαράσταση ενός τέτοιου μοντέλου και του χώρου αναζήτησής του με έναν εύκολο και κατανοητό τρόπο. Το λογισμικό είναι ανοιχτού κώδικα και ελεύθερα προσβάσιμο από οποιονδήποτε, και η χρήση του δεν απαιτεί καμία γνώση σχετικά με τις επιμέρους λειτουργίες του, όπως τη διακριτοποίηση ενός συνεχούς χώρου αναζήτησης, τον *αποδοτικό* διαμοιρασμό του φόρτου εργασίας στα επιμέρους συστήματα και υποσυστήματα, και τη συλλογή των αποτελεσμάτων. Δίνεται η δυνατότητα χρήσης ετερογενούς συστήματος (που αποτελείται δηλαδή από πολλά συστήματα ανόμοιων επιδόσεων) χωρίς σημαντική επιβάρυνση στον χρόνο εκτέλεσης λόγω της τεχνικής διαμοιρασμού των υπολογισμών. Τα συστήματα αυτά μπορούν να είναι συνδεδεμένα μεταξύ τους τοπικά ή μέσω διαδικτύου. Η υλοποίηση δεν περιορίζει το μέγεθος του χώρου αναζήτησης εφόσον το σύστημα master το επιτρέπει (βλ. Δομή Δικτύου), το πλήθος των συστημάτων του δικτύου, ή το πλήθος των υποσυστημάτων κάθε συστήματος (επεξεργαστές, κάρτες γραφικών).

Κατά την ανάπτυξη του DES Framework, αναπτύχθηκε και χρησιμοποιήθηκε μία παραλλαγή της ήδη υπάρχουσας τεχνικής HPLS [2]. Η νέα τεχνική χρησιμοποιεί μετρήσεις πραγματικού χρόνου ώστε να αξιολογήσει τους υπολογιστικούς κόμβους του δικτύου, σε αντίθεση με την απλή υλοποίηση η οποία βασίζεται σε προϋπολογισμένους χρόνους εκτέλεσης ενός προγράμματος σε όλους τους κόμβους. Έτσι, παρέχεται μία πιο ρεαλιστική εικόνα των

υπολογιστικών ικανοτήτων ενός ετερογενούς συστήματος, άρα δίνεται η δυνατότητα μίας καλύτερης κατανομής του φόρτου εργασίας.

Τέλος, στα πλαίσια της διπλωματικής εργασίας προτείνεται η τεχνική «Slow-start μέγιστης ανάθεσης», η οποία στοχεύει να καταπολεμήσει μία βασική αδυναμία της αρχικοποίησης των τεχνικών δυναμικής χρονοδρομολόγησης βρόχων. Το πρόβλημα παρουσιάζεται όταν το αρχικό μέγεθος ανάθεσης για κάποιους υπολογιστικούς κόμβους ορίζεται πολύ μεγάλο για τις επιδόσεις τους, επιτρέποντας έτσι να μη μείνουν αρκετά στοιχεία προς επεξεργασία για τους γρηγορότερους κόβμους του δικτύου. Η τεχνική που προτείνεται περιορίζει το πλήθος των στοιχείων που επιτρέπεται να ανατεθούν σε κάθε κόμβο κατά τα πρώτα αιτήματά του, με τρόπο τέτοιο ώστε να μη προκαλείται σημαντική συμφόρηση στον master κόμβο, ενώ μειώνεται η πιθανότητα υπερβολικά μεγάλης ανάθεσης που ενδεχομένως να αφήσει γρηγορότερους κόμβους σε αδράνεια.

Κατά την ολοκλήρωση της υλοποίησης έγινε μία πειραματική αξιολόγηση για τη σύγκριση της χρήσης του DES Framework για μοντέλα που έχουν προηγούμενες αποκλειστικές υλοποιήσεις σε GPU. Η αξιολόγηση έδειξε ότι το DES δε προσφέρει κάποια βελτίωση στην απόδοση του υπολογισμού σε ένα μεμονωμένο σύστημα σε σχέση με την αποκλειστική υλοποίηση του αντίστοιχου μοντέλου, αλλά αυτό αντισταθμίζεται από την ευελιξία που προσφέρει για την εύκολη και γρήγορη υλοποίηση εξαντλητικών αναζητήσεων σε κατανεμημένα συστήματα, κάτι που διαφορετικά θα απαιτούσε μεγάλο χρόνο και προσπάθεια για να πραγματοποιηθεί.

1.3 Διάρθρωση της Διπλωματικής Εργασίας

Η δομή της διπλωματικής εργασίας έχει ως εξής: Στο πρώτο κεφάλαιο περιγράφεται το πρόβλημα της εξαντλητικής αναζήτησης, και η συνεισφορά του DES Framework που υλοποιήθηκε στα πλαίσια της εργασίας. Στο δεύτερο κεφάλαιο αναλύονται οι διαθέσιμοι υλικοί πόροι και τα εργαλεία που θα χρησιμοποιηθούν για να λυθεί το πρόβλημα. Στο τρίτο κεφάλαιο αναλύεται το λογισμικό που αναπτύχθηκε και οι νέες τεχνικές χρονοδρομολόγησης βρόχων που προτείνει η διπλωματική εργασία. Περιγράφεται η δομή του δικτύου που δημιουργεί, ο τρόπος που χειρίζεται τα δεδομένα του, και η διεπαφή που προσφέρει στο χρήστη του. Επίσης παρουσιάζεται η μεθοδολογία βάση της οποίας μοιράζει το φόρτο εργασίας στους διαθέσιμους

πόρους, και η τεχνική «Slow-start μέγιστης ανάθεσης» που προτείνεται για την αποδοτική αρχικοποίηση της τεχνικής δυναμικού loop-scheduling που χρησιμοποιείται. Δίνονται λεπτομέρειες της υλοποίησης σχετικές με τη βελτίωση της απόδοσης του framework και τις λειτουργίες που προσφέρει, και παρέχεται μία λεπτομερής περιγραφή των βασικών συναρτήσεων που συνθέτουν το εν λόγω λογισμικό. Στο τέταρτο κεφάλαιο περιγράφεται η πειραματική αξιολόγηση της υλοποίησης, καταγράφονται τα αποτελέσματά της, και αποτυπώνονται τα βασικά συμπεράσματα. Τέλος, στο πέμπτο κεφάλαιο αναφέρονται μερικές μελλοντικές βελτιώσεις που μπορούν να γίνουν πάνω στο DES Framework, που αφορούν θέματα απόδοσης και χρηστικότητας.

2

Πόροι και Εργαλεία

Είναι προφανές ότι προβλήματα που δεν έχουν κλειστή λύση και είναι μεγάλου μεγέθους μπορούν να λυθούν μόνο με τη χρήση υπολογιστών. Δυστυχώς, ακόμα και μετά από τόσα χρόνια ερευνών και μόνιμης αύξησης της απόδοσης ενός επεξεργαστή, ένας απλός επεξεργαστικός πυρήνας δεν είναι αρκετά γρήγορος για να εκτελέσει μία εξαντλητική αναζήτηση μεγάλης κλίμακας σε ρεαλιστικό χρόνο. Πλέον, έχουμε στη διάθεσή μας πολυπύρηνους επεξεργαστές, και κάρτες γραφικών που μπορούν να εκτελέσουν κώδικα γενικής χρήσης. Ένα σύστημα μπορεί να αποτελείται από περισσότερους από έναν επεξεργαστές, και από πολλές κάρτες γραφικών. Επειδή όμως ένα σύστημα δε μπορεί να επεκτείνεται επ' αόριστων, έχουμε στη διάθεσή μας τεχνικές που μας επιτρέπουν να συνδέουμε πολλά τέτοια συστήματα μεταξύ τους ώστε αυτά να συνεργάζονται.

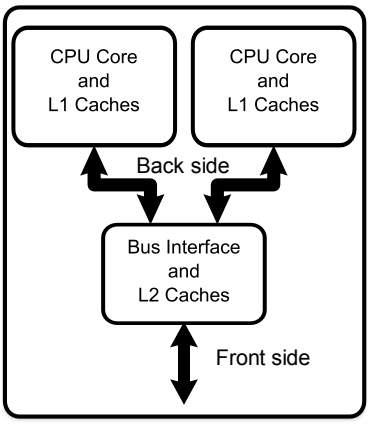
2.1 CPU

Ένας πυρήνας επεξεργαστή είναι το σύνολο των φυσικών κυκλωμάτων που απαιτούνται για να εκτελεστεί οποιαδήποτε λειτουργία του. Ονομαστικά, αυτά είναι το αρχείο καταχωρητών, οι αριθμητικές και λογικές μονάδες, ο αποκωδικοποιητής εντολών, ο μηχανισμός διασωλήνωσης εντολών (pipelining), η κρυφή μνήμη, κ.α.. Τεχνικές όπως το pipelining και η κρυφή μνήμη χρησιμοποιούνται για τη μεγιστοποίηση της απόδοσης ενός επεξεργαστικού πυρήνα, πέρα από τη προφανή λύση που είναι η αύξηση της συχνότητας λειτουργίας. Τέτοιες τεχνικές αναπτύσσονται εδώ και χρόνια και συνεχίζουν να εξελίσσονται. Δυστυχώς ή ευτυχώς, φυσικό επόμενο για την αύξηση της απόδοσης ενός επεξεργαστή είναι η χρήση πολλών επεξεργαστικών πυρήνων στο ίδιο ολοκληρωμένο κύκλωμα. Επεξεργαστές που

περιέχουν πάνω από έναν επεξεργαστικό πυρήνα ονομάζονται **πολυπύρηνοι** (Εικόνα 1 [3]) και είναι ευρέως διαδεδομένοι στις μέρες μας. Προσφέρουν μεγαλύτερη απόδοση σε χαμηλό κόστος, και παρέχουν πραγματική παραλληλία, η οποία μπορεί μόνο να προσομοιωθεί αν έχουμε μόνο έναν πυρήνα.

Σε έναν πολυπύρηνο επεξεργαστή υπάρχουν πολλά αντίγραφα του υλικού που απαρτίζει έναν επεξεργαστικό πυρήνα, τα οποία δουλεύουν ανεξάρτητα μεταξύ τους. Κάθε πυρήνας διατηρεί τα δικά του δεδομένα (καταχωρητές, κρυφή μνήμη), ακολουθεί τη δική του ροή προγράμματος, και μπορεί ακόμα να έχει και διαφορετική συχνότητα λειτουργίας από τους υπόλοιπους ή να είναι και τελείως απενεργοποιημένος. Όλοι οι επεξεργαστικοί πυρήνες έχουν πρόσβαση στην ίδια κύρια μνήμη του συστήματος, και μέσω αυτής μπορούν να ανταλλάζουν δεδομένα και να συγχρονίζονται μεταξύ τους. Σε κάθε δεδομένη στιγμή, κάθε επεξεργαστικός πυρήνας μπορεί να εκτελεί διαφορετικές εντολές από τους υπόλοιπους, ή ίδιες εντολές αλλά σε διαφορετικά δεδομένα. Αυτό μας δίνει τη δυνατότητα να μοιράσουμε τις εντολές που πρέπει να εκτελεστούν σε πολλούς πυρήνες, ώστε να ολοκληρωθούν γρηγορότερα. Για να γίνει κάτι τέτοιο πρέπει φυσικά οι εντολές να το επιτρέπουν, δηλαδή να μην υπάρχουν εξαρτήσεις εκτέλεσης ή δεδομένων μεταξύ τους. Η παραλληλία μεταξύ πυρήνων γίνεται ρητά από τον προγραμματιστή, δηλαδή δε γίνεται αυτόματα από το υλικό ή το λειτουργικό σύστημα, σε αντίθεση με την παραλληλία εντός του πυρήνα που μπορεί να πετυχαίνει η τεχνική του pipelining, ή τη τεχνική γρήγορης εναλλαγής διεργασιών που προσφέρει μία εικονική παραλληλία σε μονοπύρηνα συστήματα.

Με τον κατάλληλο προγραμματισμό και υπό συγκεκριμένες συνθήκες μπορούμε να έχουμε επιτάχυνση μέχρι και ίση με το πλήθος των πυρήνων. Αυτό έχει να κάνει με τη φύση του προβλήματος, και το κατά πόσο η διεργασία μπορεί να παραλληλοποιηθεί. Φυσικά, καθοριστικό ρόλο για την επιτάχυνση έχει και ο τρόπος με τον οποίο μοιράζεται ο φόρτος εργασίας, διότι, παρά το γεγονός ότι οι επεξεργαστικοί πυρήνες δουλεύουν ανεξάρτητα μεταξύ τους, έχουν και πολλά κοινά. Χρησιμοποιούν την ίδια κύρια μνήμη του συστήματος πάνω από το ίδιο κανάλι επικοινωνίας, και κάποια επίπεδα κρυφής μνήμης μπορεί να είναι ενιαία για ολόκληρο τον επεξεργαστή.



Εικόνα 1: Block διάγραμμα ενός 2-πύρηνου επεξεργαστή

Οι σύγχρονοι επεξεργαστές συνήθως έχουν από 4 έως 8 πυρήνες και είναι ιδανικοί για καθημερινή χρήση (Intel i5-10400 6C/12T [4], AMD Ryzen 7 3800X 8C/16T [5], Qualcomm Snapdragon 821 4C/4T (Mobile)) [6] ενώ υπάρχουν και high-end προϊόντα που ξεπερνούν ακόμα και τους 64 πυρήνες (Intel Xeon Phi 7230F 64C/64T [7], AMD Ryzen Threadripper 3990X 64C/128T [8]). Υπάρχουν επίσης υπολογιστικοί πυρήνες που συμπεριφέρονται ως 2 ή περισσότεροι, χωρίς όμως να μπορούν να προσφέρουν πραγματική παραλληλία. Πρόκειται για πολλαπλά νήματα τα οποία εκτελούνται στο ίδιο υλικό, και εναλλάσσονται μεταξύ τους όταν κάποιο από αυτά χρειάζεται πρόσβαση σε πόρους που δεν είναι διαθέσιμοι εκείνη τη στιγμή (όπως για παράδειγμα η πρόσβαση στη κεντρική μνήμη του συστήματος). Η τεχνολογία αυτή ονομάζεται **multithreading**, και χρησιμοποιείται συχνά σε εφαρμογές καθημερινής χρήσης, όπου η γρήγορη ανταπόκριση του συστήματος είναι πιο σημαντική από το συνολικό throughput. Εφόσον το υλικό που εκτελεί τις πραγματικές πράξεις είναι ένα, το multithreading από μόνο του δε συνεισφέρει στο πλήθος των πράξεων που μπορούν να εκτελεστούν στη μονάδα του χρόνου, άρα δε βελτιώνει τη συνολική απόδοση του επεξεργαστή.

Όπως ένας επεξεργαστής μπορεί να αποτελείται από πολλούς πυρήνες, έτσι και ένας υπολογιστής μπορεί να αποτελείται από πολλούς επεξεργαστές. Σε μία τέτοια διαμόρφωση όλοι οι επεξεργαστές έχουν πρόσβαση στην ίδια μνήμη και στα ίδια περιφερειακά συστήματα. Υπάρχουν πολλοί τρόποι με τους οποίους μπορούν αν οργανωθούν τέτοια συστήματα. Η βασικότερη και πιο συνηθισμένη οργάνωση είναι αυτή όπου όλοι οι επεξεργαστές έχουν τις ίδιες ευθύνες, είναι δηλαδή ίδιοι όσο αφορά το λειτουργικό σύστημα. Αυτό ονομάζεται *συμμετρική πολυεπεξεργασία*. Εναλλακτικά, κάποιοι επεξεργαστές μπορεί να δεσμεύονται για κάποιο συγκεκριμένο σκοπό, ή να δεσμεύουν κάποιους συγκεκριμένους πόρους του συστήματος. Τέτοια συστήματα είναι τα συστήματα *ασύμμετρης πολυεπεξεργασίας*, *συστήματα NUMA*, κ.α. Υπάρχουν επίσης τα συστήματα *master/slave πολυεπεξεργασίας*, όπου ένας επεξεργαστής είναι ο master και αναθέτει εργασίες στους υπόλοιπους που θεωρούνται slaves. Σε κάθε σενάριο, οι επεξεργαστές μπορούν να είναι διαφορετικοί μεταξύ τους από φυσικής άποψης (ταχύτητα και αρχιτεκτονική), αν και συνήθως αυτοί που χρησιμοποιούνται σε ένα σύστημα είναι όμοιοι μεταξύ τους, ειδικά σε συστήματα συμμετρικής πολυεπεξεργασίας.

Συστήματα με πολλούς επεξεργαστές χρησιμοποιούνται για συγκεκριμένους σκοπούς, καθώς η χρήση τους δεν είναι πάντα ωφέλιμη. Προβλήματα όπως η ανταλλαγή δεδομένων και η διασφάλιση της συνέπειας της κύριας μνήμης μπορούν να προκαλέσουν μεγάλες

καθυστερήσεις αν δε ληφθούν υπ' όψη κατά τον προγραμματισμό, καθιστώντας το όλο σύστημα μη αποδοτικό.

2.2 GPU

Ένα άλλο είδος επεξεργαστικού στοιχείου που βρίσκεται συχνά σε υπολογιστές είναι οι κάρτες γραφικών. Πρόκειται για κάρτες επέκτασης που συνδέονται σε ένα ήδη υπάρχων σύστημα, και αποτελούνται και αυτές από έναν κεντρικό επεξεργαστή και τη δική τους μνήμη. Όπως δηλώνει το όνομα, η δουλειά μίας κάρτας γραφικών είναι η παραγωγή των γραφικών στην οθόνη του υπολογιστή. Χρησιμοποιούνται προφανώς για εφαρμογές που απαιτούν μεγάλη επεξεργαστική ισχύ γραφικών, όπως το 3D Modeling, επεξεργασία εικόνας και βίντεο, παιχνίδια, κ.α.

Ο υπολογισμός γραφικών έχει διαφορετικές απαιτήσεις, άρα αυτά τα υπολογιστικά υποσυστήματα έχουν διαφορετική αρχιτεκτονική. Για να γίνουν οι απαραίτητοι υπολογισμοί, πρέπει να γίνουν πάρα πολλές αλλά απλές πράξεις, πάνω σε διαφορετικά δεδομένα. Αυτές οι πράξεις είναι ανεξάρτητες μεταξύ τους και μπορούν να εκτελεστούν παράλληλα. Ένας συμβατικός επεξεργαστής δε μπορεί να ανταπεξέλθει, διότι θα πρέπει να εκτελέσει πολλές απλές πράξεις σειριακά. Αντιθέτως, ο επεξεργαστής μίας κάρτας γραφικών αποτελείται από πολλούς πυρήνες (της τάξης των 1000-4000), οι οποίοι όμως είναι πολύ μικρότεροι από τους πυρήνες ενός απλού επεξεργαστή και έχουν πολύ λιγότερες δυνατότητες (όπως φαίνεται στην εικόνα 8). Οι πυρήνες αυτοί λειτουργούν σε χαμηλότερες συχνότητες, είναι ομαδοποιημένοι μεταξύ τους, και όλοι οι πυρήνες μίας ομάδας μπορούν κάθε στιγμή να εκτελούν μόνο μία εντολή πάνω σε διαφορετικά δεδομένα. Η αρχιτεκτονική αυτή επιτρέπει σε μία κάρτα γραφικών να εκτελέσει την ίδια αλληλουχία πράξεων πάνω σε πολλά διαφορετικά δεδομένα. Για παράδειγμα, αν έχουμε 2 διανύσματα 1000 στοιχείων και θέλουμε να τα προσθέσουμε, μία κάρτα γραφικών θα αναθέσει ένα στοιχείο του τελικού διανύσματος σε κάθε πυρήνα και θα εκτελέσει όλες τις πράξεις ταυτόχρονα. Αντιθέτως, ένας κοινός επεξεργαστής θα πρέπει να χωρίσει τα 1000 στοιχεία σε κομμάτια, και ο κάθε πυρήνας να αναλάβει ένα από αυτά, υπολογίζοντας κάθε στοιχείο του σειριακά. Προφανώς, η κάρτα γραφικών θα εκτελέσει τον υπολογισμό πολύ γρηγορότερα.

Παρόλο που αυτή η αρχιτεκτονική δε βολεύει για έναν επεξεργαστή γενικής χρήσης, είναι πολύ χρήσιμη για την επεξεργασία μεγάλου όγκου δεδομένων, όπου θέλουμε να εκτελέσουμε ένα απλό τμήμα κώδικα πάνω σε πολλά δεδομένα. Η χρήση μίας κάρτας γραφικών για την εκτέλεση εντολών γενικού σκοπού ονομάζεται General-Purpose computing on Graphics Processing Units (GPGPU) [9]. Για να επιτευχθεί αυτό πρέπει να γραφεί ειδικός κώδικας και να περάσει από ειδικό μεταφραστή που να λαμβάνει υπ' όψη του τη διαφορετική αρχιτεκτονική του συστήματος. Η πλέον διαδεδομένη γλώσσα προγραμματισμού για GPGPU είναι η OpenCL [10], η οποία είναι ανοιχτού κώδικα και υποστηρίζει τις κάρτες γραφικών της AMD, NVIDIA, Intel, και ARM. Για κάρτες γραφικών της NVIDIA, η εταιρία έχει αναπτύξει τη πλατφόρμα Compute Unified Device Architecture (CUDA) [11] ως υλοποίηση του GPGPU, η οποία και χρησιμοποιείται στη παρούσα εργασία. Η πλατφόρμα CUDA παρέχει βιβλιοθήκες, ειδικό μεταφραστή (compiler), και ισχυρά εργαλεία για debugging και profiling CUDA εφαρμογών. Παρέχεται επίσης ένα εκτενές documentation από την NVIDIA, τόσο για τη χρήση του API της πλατφόρμας όσο και για τις λεπτομέρειες της αρχιτεκτονικής των καρτών και θέματα βελτιστοποίησης του κώδικα.

Αξίζει να σημειωθεί ότι οι πιο σύγχρονες κάρτες γραφικών της NVIDIA υποστηρίζουν αριθμητική *half-precision floating point* [12], [13]. Πρόκειται για μία αναπαράσταση αριθμών κινητής υποδιαστολής η οποία χρησιμοποιεί λιγότερα δυαδικά ψηφία από τις πιο συνηθισμένες: *single-precision* (32bit) και *double-precision* (64bit). Χρησιμοποιώντας λιγότερα bits για την αναπαράσταση αριθμών κινητής υποδιαστολής μειώνεται η ακρίβεια και το εύρος των αριθμών που μπορούν να αναπαρασταθούν, αλλά ταυτόχρονα αυξάνεται η ταχύτητα των πράξεων με αυτούς και μειώνεται ο απαιτούμενος χώρος αποθήκευσής τους. Η μικρότερη απαίτηση σε χωρητικότητα μνήμης μεταφράζεται τόσο σε εξοικονόμηση χώρου στη κύρια μνήμη, όσο και σε περισσότερα δεδομένα διαθέσιμα στη κρυφή μνήμη ενός επεξεργαστή προσφέροντας σημαντική βελτίωση της επίδοσής του [14]. Εφαρμογές οι οποίες απαιτούν μεγάλο όγκο υπολογισμών αλλά δεν έχουν ιδιαίτερες απαιτήσεις σε ακρίβεια μπορούν να επωφεληθούν σημαντικά από την αριθμητική μειωμένης ακρίβειας. Τέτοιες εφαρμογές είναι τα γραφικά υπολογιστών [15], τα νευρωνικά δίκτυα για τεχνητή νοημοσύνη [16], κ.α. Ανά περιπτώσεις, η χρήση μειωμένης ακρίβειας μπορεί να προσφέρει ακόμα και 3x speedup στην εκτέλεση ενός προγράμματος [17]. μεταξύ κάρτας γραφικών και κεντρικής μνήμης απευθείας, χωρίς τη παρέμβαση της ΚΜΕ.

Για την εκτέλεση ενός υπολογιστικού πυρήνα (kernel) στη κάρτα γραφικών, η διαδικασία που ακολουθείται είναι η εξής:

1. Μεταφορά δεδομένων από τη κύρια μνήμη του συστήματος στη μνήμη της κάρτας γραφικών
2. Εκτέλεση του υπολογιστικού πυρήνα (kernel) στον επεξεργαστή της GPU
3. Μεταφορά αποτελεσμάτων από τη μνήμη της κάρτας γραφικών στη κύρια μνήμη.

Η εκτέλεση του kernel στη GPU είναι ασύγχρονη, που σημαίνει ότι ο επεξεργαστής του συστήματος είναι ελεύθερος να συνεχίσει την εκτέλεση του προγράμματος και να πάρει τα αποτελέσματα από τη κάρτα γραφικών όταν τα χρειαστεί.

Οι κάρτες γραφικών της NVIDIA παρέχουν τη δυνατότητα παράλληλων streams για την εκτέλεση μεταφορών δεδομένων και kernels (βλ. GPU Streams). Τα streams μπορούν να χρησιμοποιηθούν για να παραλληλοποιηθούν οι μεταφορές δεδομένων με τις εκτελέσεις των kernels, ή ακόμα και για την παραλληλοποίηση των ίδιων των μεταφορών όταν αυτές είναι προς διαφορετικές κατευθύνσεις. Στο παράδειγμα της πρόσθεσης 2 διανυσμάτων με 1000 στοιχεία το καθένα, θα μπορούσαμε να στείλουμε τα πρώτα 500 στοιχεία και να ξεκινήσουμε το kernel για αυτά, και όσο υπολογίζονται τα πρώτα 500 αποτελέσματα μπορούμε να μεταφέρουμε τα υπόλοιπα 500 στοιχεία των 2 διανυσμάτων. Όταν τελειώσει ο υπολογισμός των πρώτων 500 αποτελεσμάτων, ξεκινάμε την μεταφορά τους προς τη κύρια μνήμη, ενώ παράλληλα ξεκινάμε τον υπολογισμό των επόμενων 500 στοιχείων. Τέλος, μεταφέρουμε τα τελευταία 500 αποτελέσματα. Αυτή η λογική μπορεί να επεκταθεί σε περισσότερα streams.

Ένα σύστημα μπορεί να έχει μία ή περισσότερες κάρτες γραφικών. Οι κάρτες αυτές μπορούν να λειτουργούν παράλληλα και ανεξάρτητα μεταξύ τους, ή μπορούν και να συνεργάζονται. Σε κάθε περίπτωση αυτό είναι στην ευχέρεια της εκάστοτε εφαρμογής, επιλέγοντας κάθε φορά ποια κάρτα θα χρησιμοποιήσει, μεταφέροντας τα κατάλληλα δεδομένα, και καλώντας τον αντίστοιχο υπολογιστικό πυρήνα. Υπάρχει δυνατότητα επικοινωνίας μεταξύ των καρτών γραφικών (εφόσον το υποστηρίζουν και είναι συμβατές μεταξύ τους) μέσω PCIe χωρίς τη παρέμβαση της KME, χρησιμοποιώντας την τεχνολογία GPUDirect της NVIDIA [18]. Επίσης είναι δυνατή η απευθείας σύνδεσης καρτών γραφικών της NVIDIA μέσω SLI [19] ή NVLink [20], επιτρέποντας έτσι μεγαλύτερες ταχύτητες ανταλλαγής δεδομένων, ή ακόμα και άθροιση της διαθέσιμης μνήμης (όπου οι μνήμες των καρτών γραφικών συμπεριφέρονται σαν μία ενιαία μνήμη, προσβάσιμη άμεσα από όλες). Σε περίπτωση που καμία από αυτές τις

τεχνικές δεν είναι διαθέσιμη, μπορεί να γραφεί κώδικας που θα εκτελεστεί στον κεντρικό επεξεργαστή του συστήματος, ο οποίος θα αναλάβει να συντονίσει τις κάρτες γραφικών και να μεταφέρει δεδομένα μεταξύ τους.

2.3 Clusters – Συστήματα κατανεμημένης μνήμης

Ένας υπολογιστής μπορεί να αποτελείται από πολλούς επεξεργαστές, πολλές κάρτες γραφικών, και μνήμη χωρητικότητας ακόμα και πάνω από 1 TB. Όμως, το κόστος ενός τέτοιου συστήματος είναι απαγορευτικό, καθώς αυτό δεν αυξάνεται γραμμικά με την απόδοση. Ακόμα και ένα τέτοιο σύστημα όμως έχει όρια, άρα δεν είναι απαραίτητο ότι θα μπορεί να λύσει προβλήματα μεγάλης κλίμακας. Επίσης, μερικές φορές μπορεί να έχουμε στη διάθεσή μας πολλούς υπολογιστές μέτριων επιδόσεων, αντί για έναν πολύ γρήγορο. Σε τέτοια σενάρια λοιπόν, η πρακτική λύση είναι η σύνδεση πολλών υπολογιστών σε ένα δίκτυο και η συντονισμένη συνεργασία τους για έναν κοινό σκοπό. Τέτοια συστήματα ονομάζονται *συστήματα κατανεμημένης μνήμης*, λόγω του βασικού τους χαρακτηριστικού: αν το δούμε ως ένα σύστημα, τότε η μνήμη του είναι κατανεμημένη σε ξεχωριστά φυσικά συστήματα.

Πλέον έχουμε στη διάθεσή μας εργαλεία, πρωτόκολλα, και τεχνικές για τη συνεργασία υπολογιστικών συστημάτων. Υπάρχουν λύσεις κατανεμημένων συστημάτων που επιτρέπουν στον προγραμματιστή να βλέπει το σύστημα ως έναν ενιαίο υπολογιστή, αλλά συνήθως οι λύσεις που χρησιμοποιούνται ακολουθούν τη λογική της κατανεμημένης μνήμης. Στα συστήματα κατανεμημένης μνήμης κάθε υπολογιστής έχει πρόσβαση μόνο στη δική του κύρια μνήμη. Αυτό σημαίνει ότι για να εκτελέσει κάποιον υπολογισμό πρέπει να αποφασίσει ποιο τμήμα του υπολογισμού θα εκτελέσει, να πάρει από κάπου τα απαραίτητα δεδομένα, και έπειτα να ξεκινήσει τον υπολογισμό. Τέτοια συστήματα προσφέρουν λειτουργίες που δε μπορεί να προσφέρει ένα σύστημα από μόνο του, όπως η επεκτασιμότητα του συστήματος ενώ έχει ήδη ξεκινήσει ο υπολογισμός και ανθεκτικότητα σε αποτυχίες συστημάτων.

Σε αυτά τα δίκτυα υπολογιστών συχνά χρησιμοποιείται η λογική του master/slave. Ένας υπολογιστής (ή ένας επεξεργαστικός πυρήνας ενός υπολογιστή) είναι ο master και όλοι οι υπόλοιποι είναι οι slaves. Ο master υπολογιστής είναι υπεύθυνος για τον συντονισμό των υπολοίπων, το διαμοιρασμό των υπολογισμών, και τη συλλογή τον αποτελεσμάτων. Ανάλογα με την υλοποίηση, ο master μπορεί κι αυτός να εκτελεί μέρος του υπολογισμού όσο δεν έχει

να κάνει κάτι άλλο. Κάθε slave επεξεργαστικό στοιχείο όταν είναι έτοιμο ζητάει ένα κομμάτι υπολογισμού από τον master, υπολογίζει τα αποτελέσματα, τα επιστρέφει, και ξεκινάει από την αρχή. Στο τέλος ο master έχει συγκεντρώσει όλα τα αποτελέσματα και τα επιστρέφει στον χρήστη. Μία τέτοια διαμόρφωση είναι πολύ συνηθισμένη, αλλά έχει και τα αρνητικά της. Το επεξεργαστικό στοιχείο που είναι master μπορεί πολύ εύκολα να είναι bottleneck, κυρίως εάν η διαδικασία της ανάθεσης εργασίας είναι πολύπλοκη ή απαιτούνται πολλές/μεγάλες μεταφορές δεδομένων από/προς τον master. Σε αυτές τις περιπτώσεις οι slaves περιμένουν τον master να ανταποκριθεί, και έτσι σπαταλούν χρόνο ενώ θα μπορούσαν να έκαναν χρήσιμους υπολογισμούς. Επίσης, ο master είναι single point of failure του συστήματος, καθώς αν αυτός αποτύχει τότε όλος ο υπολογισμός καταρρέει.

Εναλλακτικά, υπάρχουν δίκτυα όπου δεν υπάρχει master, και όλοι οι υπολογιστές του δικτύου είναι ισοδύναμοι. Αυτά ονομάζονται δίκτυα Peer-to-peer (P2P) και γίνονται ολοένα και πιο δημοφιλή τα τελευταία χρόνια, καθώς έχουν πολλές χρήσεις στον κόσμο των υπολογιστών. Προσφέρουν μεγάλη ανθεκτικότητα σε σφάλματα των κόμβων του δικτύου, είναι αποκεντρωμένα άρα δεν υπάρχει single point of failure, και μπορούν να προσφέρουν απόλυτη ανωνυμία των χρηστών τους. Η υλοποίηση τέτοιων δικτύων είναι πολύ δυσκολότερη από ένα απλό master/slave δίκτυο, και το overhead του συντονισμού είναι συχνά τόσο μεγάλου που δε προτιμάται για εφαρμογές όπου η επεξεργαστική ισχύς είναι το κύριο ζητούμενο.

Ένα πρόβλημα σε ένα δίκτυο κατανεμημένου υπολογισμού είναι η ανομοιομορφία της υπολογιστικής ισχύος στα συστήματά του. Ένα τέτοιο δίκτυο μπορεί να αποτελείται από διαφορετικούς μεταξύ τους υπολογιστές, όπου κάθε ένας έχει διαφορετική επεξεργαστική ισχύ. Όταν αυτή η διαφορά είναι μεγάλη, τίθεται ζήτημα της κατανομής του φόρτου εργασίας με τρόπο τέτοιο ώστε να αξιοποιούνται στο έπακρο όλα τα συστήματα. Για αυτό, έχουν αναπτυχθεί πολλές τεχνικές προγραμματισμού υπολογισμών (βλ. Load Balancing).

Σαν σύστημα κατανεμημένης μνήμης μπορεί να θεωρηθεί και ένας απλός υπολογιστής που έχει τουλάχιστον μία κάρτα γραφικών. Η κάρτα γραφικών είναι ένας υπολογιστής από μόνη της, διότι έχει τον δικό της επεξεργαστή και τη δική της ανεξάρτητη μνήμη. Για να χρησιμοποιηθεί πρέπει να μεταφερθούν τα απαραίτητα δεδομένα σε αυτή, να γίνουν οι απαραίτητοι υπολογισμοί, και έπειτα να επιστρέψουν τα αποτελέσματα στη κύρια μνήμη του συστήματος. Ακολουθείται δηλαδή η λογική υπολογισμού με κατανεμημένη μνήμη. Άρα

λοιπόν η φιλοσοφία του προγραμματισμού για τέτοια συστήματα δεν εφαρμόζεται μόνο σε δίκτυα υπολογιστών, αλλά και για κάθε έναν υπολογιστή ξεχωριστά.

Το DES Framework που υλοποιήθηκε στη παρούσα εργασία υλοποιεί ένα δίκτυο 2 επιπέδων master/slave, όπως περιγράφεται στο [21]. Συγκεκριμένα, πολλοί υπολογιστές συνδέονται μεταξύ τους και ένας από αυτούς είναι master, ενώ οι υπόλοιποι είναι slaves. Ο master υπολογιστής μπορεί να ρυθμιστεί ώστε να θεωρηθεί ταυτόχρονα και ως slave ώστε να συμμετέχει στον υπολογισμό. Έπειτα, κάθε υπολογιστής θεωρείται ως ένα κατανεμημένο σύστημα, όπου οι κόμβοι του είναι ο επεξεργαστής και οι κάρτες γραφικών του. Άρα κάθε υπολογιστής του δικτύου υλοποιεί αυτόνομα ένα δεύτερο επίπεδο master/slave.

2.4 OpenMP

Ένα από τα βασικά εργαλεία στον τομέα της παραλληλίας είναι το OpenMP [22]. Πρόκειται για ένα API ανοιχτού κώδικα για παράλληλο υπολογισμό σε συστήματα κοινής μνήμης και δίνεται ως επέκταση των γλωσσών προγραμματισμού C, C++, και Fortran. Το OpenMP παρέχει ένα σύνολο εργαλείων και αυτοματισμών που επιτρέπουν σε έναν προγραμματιστή να παραλληλοποιήσει ένα τμήμα σειριακού κώδικα γρήγορα και εύκολα, χωρίς να ασχοληθεί με το χαμηλότερο επίπεδο της δημιουργίας και συγχρονισμού των νημάτων που θα εκτελεστούν. Έχει τη δυνατότητα να παραλληλοποιήσει και να μοιράσει στατικά ή δυναμικά (βλ. Load Balancing) ένα `for` loop σε όλους τους επεξεργαστικούς πυρήνες ενός συστήματος με μία μόνο εντολή.

Χρησιμοποιεί το μοντέλο fork/join, κατά το οποίο το βασικό νήμα μίας εφαρμογής διαιρείται σε πολλά άλλα (fork) (συνήθως ίσα με το πλήθος των πυρήνων του συστήματος) τα οποία τρέχουν παράλληλα. Όταν όλα τα νήματα τελειώσουν τον υπολογισμό που τους ανατέθηκε, αυτά τερματίζουν και το αρχικό νήμα συνεχίζει την εκτέλεσή του (join).

Στο DES Framework το OpenMP παίζει σημαντικό ρόλο στο δεύτερο master/slave επίπεδο, όπου χρησιμοποιείται για να χωριστούν οι υπολογισμοί στους πόρους κάθε συστήματος. Συγκεκριμένα, δημιουργείται ένα νήμα για κάθε επεξεργαστικό στοιχείο του συστήματος (1 για τη CPU και 1 για κάθε GPU) τα οποία ονομάζονται worker threads, και ένα ακόμα για να τα συντονίζει το οποίο ονομάζεται coordinator thread (νήμα συντονιστή). Το coordinator thread επικοινωνεί με τον master υπολογιστή του δικτύου για να ζητήσει ένα

κομμάτι υπολογισμού, έπειτα το αναθέτει στα worker threads, και όταν όλα εκτελέσουν τον υπολογισμό τότε συλλέγει τα αποτελέσματα και τα επιστρέφει στον master. Το coordinator thread λοιπόν είναι ο master του δεύτερου επιπέδου, και τα worker threads είναι αντίστοιχα τα slaves.

2.5 OpenMPI

Το OpenMPI [23] είναι μία υλοποίηση του προτύπου MPI (Message Passing Interface) [24]. Πρόκειται για μία βιβλιοθήκη εργαλείων για την επικοινωνία και ανταλλαγή δεδομένων ανάμεσα σε διεργασίες που εκτελούνται σε διαφορετικά υπολογιστικά συστήματα, δηλαδή συστήματα κατανεμημένης μνήμης. Όπως δηλώνει το όνομα, το MPI βασίζεται στην ανταλλαγή μηνυμάτων ανάμεσα σε κόμβους ενός δικτύου. Υπάρχει η απλή μεταφορά δεδομένων από έναν κόμβο προς έναν άλλο, αλλά προσφέρονται και πιο περίπλοκες αλληλεπιδράσεις, όπως ο διαμοιρασμός δεδομένων σε όλους τους κόμβους του δικτύου και η συλλογή των αποτελεσμάτων από όλους προς έναν, με μία μόνο κλήση στο MPI. Είναι ο πλέον διαδεδομένος μηχανισμός ανταλλαγής μηνυμάτων και παράλληλης επεξεργασίας σε συστήματα κατανεμημένης μνήμης, και χρησιμοποιείται από τους μεγαλύτερους υπερυπολογιστές του κόσμου. Το γενικό σενάριο χρήσης του είναι η συγγραφή ενός κώδικα που το χρησιμοποιεί και η ταυτόχρονη εκτέλεση του ίδιου κώδικα σε πολλά συστήματα. Μέσω της βιβλιοθήκης δίνεται ένα μοναδικό ID (rank) σε κάθε διεργασία που μπορεί να χρησιμοποιηθεί για να διαπιστωθεί σε ποιο πραγματικό σύστημα εκτελείται το πρόγραμμα ώστε αυτό να πράξει ανάλογα.

Το OpenMPI προσφέρει δυνατότητες επικοινωνίας και συγχρονισμού των MPI διεργασιών, και δίνει μία εικονική τοπολογία του δικτύου αναθέτοντας σε κάθε διεργασία ένα μοναδικό ID. Επιτρέπει την απευθείας πρόσβαση στη μνήμη μίας διεργασίας από μία άλλη, χωρίς να είναι απαραίτητο οι 2 διεργασίες να βρίσκονται στο ίδιο φυσικό σύστημα. Παρόλο που παραδοσιακά όλες οι MPI διεργασίες ξεκινούν και τερματίζουν μαζί, το πρότυπο MPI δίνει τη δυνατότητα συμμετοχής νέων διεργασιών σε έναν υπολογισμό που έχει ήδη ξεκινήσει. Έχει υλοποιηθεί για παράλληλο υπολογισμό σε συστήματα υψηλών επιδόσεων, άρα έχει στόχο τη μεγιστοποίηση της απόδοσης των συστημάτων που το χρησιμοποιούν.

Το OpenMPI χρησιμοποιήθηκε στο framework της εργασίας για την υλοποίηση του πρώτου master/slave επιπέδου. Κάθε MPI διεργασία αντιπροσωπεύει ένα σύστημα του δικτύου, με μία από αυτές να δρα ως master και τις υπόλοιπες να λειτουργούν ως slaves. Μπορούμε επίσης να δημιουργήσουμε 2 MPI διεργασίες σε ένα σύστημα, δίνοντας έτσι τη δυνατότητα σε αυτό να είναι master ενώ ταυτόχρονα είναι και slave και συμμετέχει στον υπολογισμό. Αυτό συμφέρει μόνο όταν το εν λόγω σύστημα έχει αρκετούς πόρους ώστε να διαχειριστεί το βάρος της master διεργασίας χωρίς αυτή να επιβαρύνεται από τη συμμετοχή του συστήματος στον υπολογισμό.

2.6 Load Balancing

Ένα από τα πιο υπολογιστικά απαιτητικά τμήματα ενός προγράμματος που εκτελεί επιστημονικούς υπολογισμούς είναι συνήθως ένας ή περισσότεροι βρόχοι `for` [25]. Είναι δηλαδή ένα ή περισσότερα τμήματα κώδικα που εκτελούνται επαναληπτικά πάνω σε κάποια δεδομένα. Ένας βρόχος `for` μπορεί να περιέχει μέσα του και άλλους, οι οποίοι ονομάζονται **εμφωλευμένοι βρόχοι**. Συνήθως αυτές οι επαναλήψεις είναι ανεξάρτητες μεταξύ τους διότι εκτελούνται πάνω σε διαφορετικά δεδομένα. Σε τέτοιες περιπτώσεις είναι δυνατό να τις παραλληλοποιήσουμε, έχοντας έτσι την ευκαιρία να αξιοποιήσουμε πολλά επεξεργαστικά στοιχεία (CPUs, GPUs) ταυτόχρονα, μοιράζοντας τις επαναλήψεις σε αυτά. Στα πλαίσια του κειμένου, κάθε επανάληψη κάποιου κώδικα σε βρόχο `for` θα ονομάζεται **στοιχείο** ή **σημείο**, καθώς θα αναπαριστά ένα σημείο σε κάποιον πολυδιάστατο χώρο (βλ. Αναπαράσταση δεδομένων).

Εδώ προκύπτει ένα σημαντικό **πρόβλημα**:

Πόσα και ποια στοιχεία θα αναθέσουμε σε κάθε επεξεργαστικό στοιχείο;

Η απλούστερη λύση θα ήταν να ισομοιράσουμε τα στοιχεία σειριακά στα διαθέσιμα συστήματα. Αυτό ονομάζεται **στατική χρονοδρομολόγηση βρόχων (static loop scheduling)** και σπάνια είναι η σωστή λύση. Η απάντηση στην παραπάνω ερώτηση λοιπόν δεν είναι απλή. Υπάρχουν 2 παράγοντες που πρέπει να ληφθούν υπ' όψη:

Ποιά η επεξεργαστική ισχύς κάθε επεξεργαστικού στοιχείου;

Ποιά η απαίτηση σε επεξεργαστική ισχύ κάθε στοιχείου προς επεξεργασία;

Κάθε επεξεργαστής και κάρτα γραφικών έχει διαφορετική επεξεργαστική ισχύ λόγω κατασκευαστικών χαρακτηριστικών. Ιδιότητες όπως η συχνότητα λειτουργίας, η κρυφή μνήμη, το πλήθος των πυρήνων, η αρχιτεκτονική, κ.α. προσδιορίζουν την επεξεργαστική ισχύ ενός επεξεργαστή. Επιπρόσθετα, η επεξεργαστική ισχύς ενός *συστήματος* επηρεάζεται και από άλλα χαρακτηριστικά, όπως η ταχύτητα και η χωρητικότητα της μνήμης του και το λειτουργικό του σύστημα. Είναι λοιπόν λάθος να θεωρήσουμε ότι όλα τα συστήματα του δικτύου μας θα αποδίδουν το ίδιο (ή έστω όμοια). Συχνά ένα σύστημα κατανεμημένης μνήμης θα αποτελείται από υπολογιστές ανομοιόμορφης επεξεργαστικής ισχύος. Αυτό θέτει ένα μεγάλο πρόβλημα στην κατανομή των στοιχείων που θέλουμε να επεξεργαστούμε στα συστήματά μας, διότι κάποια συστήματα μπορεί να τελειώσουν νωρίτερα και μετά να είναι αδρανή, ενώ υπάρχουν ακόμα στοιχεία προς επεξεργασία αλλά έχουν ανατεθεί σε άλλα, πιο αργά συστήματα. Αν ισομοιράσουμε τα στοιχεία στα συστήματα σε μία εφαρμογή όπου όλα τα στοιχεία προς επεξεργασία χρειάζονται τον ίδιο αριθμό πράξεων, τότε τα γρηγορότερα συστήματα θα μείνουν αδρανή τον περισσότερο χρόνο, ενώ τα πιο αργά θα δουλέψουν για περισσότερη ώρα.

Πρέπει επίσης να ληφθεί υπόψη ότι κάθε στοιχείο προς επεξεργασία μπορεί να μη χρειάζεται τον ίδιο χρόνο με τα υπόλοιπα. Το τμήμα κώδικα που θέλουμε να εκτελέσουμε επαναληπτικά μπορεί να περιέχει διακλαδώσεις, επηρεάζοντας τον χρόνο εκτέλεσης για κάθε στοιχείο ξεχωριστά. Ακόμα λοιπόν κι αν υποθέσουμε ότι γνωρίζουμε εκ των προτέρων τις επιδόσεις κάθε συστήματος και αναθέσουμε περισσότερα στοιχεία στα συστήματα που είναι γρηγορότερα, υπάρχει πάλι η πιθανότητα να έχουμε σπατάλη των πόρων μας. Αυτό μπορεί να συμβεί στη περίπτωση που στα γρηγορότερα συστήματα ανατεθούν στοιχεία που χρειάζονται λιγότερες πράξεις, άρα αυτά μπορεί πάλι να τελειώσουν νωρίτερα και να είναι αδρανή ενώ τα υπόλοιπα δουλεύουν.

Η λύση στο παραπάνω πρόβλημα είναι η **δυναμική χρονοδρομολόγηση βρόχων (dynamic loop scheduling)** ή **loop self-scheduling** [26]. Με το δυναμικό loop scheduling δεν αναθέτουμε από την αρχή κάθε στοιχείο σε κάποιο σύστημα αλλά το κάνουμε σε βήματα: Θεωρούμε ότι έχουμε ένα απλό δίκτυο master/slave ενός επιπέδου. Ξεκινώντας τον

υπολογισμό, κάθε slave ζητάει από ένα σύνολο στοιχείων από τον master για να τα επεξεργαστεί, και του ανατίθεται. Όταν τελειώσει, επιστρέφει τα αποτελέσματα και ζητάει ένα νέο σύνολο στοιχείων. Η διαδικασία τελειώνει όταν όλα τα στοιχεία έχουν ανατεθεί σε κάποιο σύστημα. Η master διεργασία είναι υπεύθυνη για το πόσα και ποια στοιχεία θα ανατεθούν σε κάθε slave.

Στη γενική περίπτωση δε γνωρίζουμε πόσες πράξεις θα χρειαστεί κάθε στοιχείο, άρα η μόνη λύση μας είναι να τα δίνουμε με "τυχαία" σειρά. Ένα `for` loop εκτελείται με μία συγκεκριμένη σειρά, ανάλογα με το πώς έχει οριστεί, άρα θα χρησιμοποιήσουμε αυτή. Σε εμφωλευμένους βρόχους `for`, το εσωτερικό τμήμα κώδικα εκτελείται για κάθε επανάληψη με αυστηρή σειρά, εκτός εάν υπάρχουν διακλαδώσεις που την αλλάζουν. Κάτι τέτοιο θα σήμαινε ότι υπάρχει εξάρτηση ανάμεσα στις επαναλήψεις των βρόχων, άρα ο κώδικας δε μπορεί να παραλληλοποιηθεί με τον τρόπο που προτείνεται. Θα υποθέσουμε λοιπόν ότι ακόμα και σε εμφωλευμένους βρόχους `for`, το εσωτερικό τμήμα κώδικα εκτελείται για κάθε στοιχείο με μία συγκεκριμένη σειρά, άρα η σειρά ανάθεσης θα είναι αυτή.

Τελικά, η ερώτηση αφορά πλέον μόνο το *"πόσα"* στοιχεία θα αναθέτουμε κάθε φορά σε κάθε σύστημα. Στη βιβλιογραφία έχουν αναφερθεί διάφορες τεχνικές υπολογισμού του μεγέθους του συνόλου των στοιχείων (chunk) που θα αναθέτουμε κάθε φορά.

- **(Pure) Self-Scheduling** [26]**:** Η πιο απλή τεχνική δυναμικού scheduling, όπου κάθε φορά που κάποιος επεξεργαστής ζητάει ένα σύνολο στοιχείων, του ανατίθεται μόνο 1. Αυτό παρέχει το καλύτερο load balancing, αλλά ταυτόχρονα επιφέρει υπερβολικά μεγάλο overhead, καθώς τα συστήματα θα ζητούν συνεχώς νέα στοιχεία για επεξεργασία.
- **Chunk Self-Scheduling** [26]**:** Η άμεση βελτίωση του Pure SS, όπου κάθε φορά αντί να ανατίθεται 1 μόνο στοιχείο, ανατίθενται *k* στοιχεία, όπου *k* μία σταθερά. Όσο μικρότερο είναι το *k*, τόσο μεγαλύτερη η χρονική επιβάρυνση λόγω συχνής επικοινωνίας. Όσο μεγαλύτερο είναι το *k*, τόσο πιο πιθανό είναι γίνει ανομοιόμορφη κατανομή των υπολογισμών, σπαταλώντας έτσι τους πόρους του συστήματος.
- **Guided Self-Scheduling** [25]**:** Η φυσική εξέλιξη του Chunk SS, όπου ο αριθμός *k* δεν είναι σταθερός. Υπολογίζεται πριν από κάθε ανάθεση, και είναι ίσος με το πλήθος των στοιχείων που απομένουν δια το πλήθος των διαθέσιμων επεξεργαστών. Αυτό συνεπάγεται ότι το *k* μειώνεται κατά την εκτέλεση του υπολογισμού, αναθέτοντας

περισσότερα στοιχεία στα πρώτα βήματα, και λιγότερα στα επόμενα. Εδώ μπορεί να δημιουργηθεί πρόβλημα αν οι πρώτες αναθέσεις που δίνουν πολλά στοιχεία γίνουν σε αργά συστήματα, ενώ τα γρηγορότερα συστήματα αναλάβουν λίγα στοιχεία και τελειώσουν νωρίτερα.

- **Heuristic Parallel Loop Scheduling** [2]**:** Η τεχνική HPLS λαμβάνει υπ' όψη της την απόδοση του συστήματος που ζητάει το σύνολο στοιχείων προς επεξεργασία. Χρησιμοποιεί μία **συνάρτηση απόδοσης** (Performance Function, PF) για να υπολογίσει μία **αναλογία απόδοσης** (Performance Ratio, PR) που αντιστοιχεί σε αυτό το σύστημα σε σχέση με τα υπόλοιπα, και έπειτα του αναθέτει ένα πλήθος στοιχείων σύμφωνα με αυτήν την αναλογία. Έτσι, τα συστήματα που αποδίδουν καλύτερα αναλαμβάνουν περισσότερη δουλειά από τα υπόλοιπα. Η PF που προτείνεται στο [2] χρησιμοποιεί τους χρόνους εκτέλεσης ενός συγκεκριμένου προγράμματος σε όλα τα συστήματα ώστε να υπολογίσει την αναλογία που αντιστοιχεί στο κάθε σύστημα κατά την έναρξη του υπολογισμού.

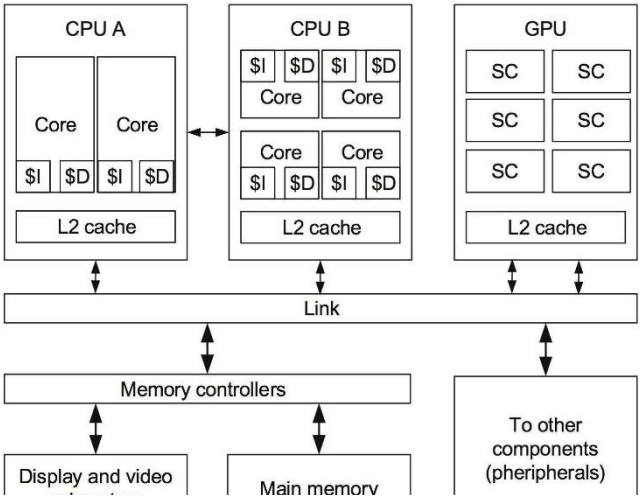
Η τεχνική self-scheduling που χρησιμοποιήθηκε στο DES Framework είναι μία παραλλαγή της HPLS. Συγκεκριμένα, το μέγεθος του chunk υπολογίζεται με βάση μία σταθερά (δοσμένη από τον χρήστη) βεβαρημένη με την αναλογία που δίνει η PF για το συγκεκριμένο σύστημα. Η PF που χρησιμοποιήθηκε αναλύεται στο κεφάλαιο 3.4.1.

3

Distributed Exhaustive Search Framework (DES Framework)

3.1 Γενικά

Στα πλαίσια της εργασίας αναπτύχθηκε ένα πλαίσιο λογισμικού που παρέχει τη δυνατότητα χρήσης πολυπύρηνων συστημάτων με συνεπεξεργαστές για την εκτέλεση εξαντλητικής αναζήτησης σε μοντέλο που παρέχει ο χρήστης. Το DES Framework δίνεται ως βιβλιοθήκη της γλώσσας C++ και δεν απαιτεί καμία παρέμβαση στον κώδικά του από το χρήστη για τη λειτουργία του. Χρησιμοποιήθηκε η C++ διότι είναι μία αντικειμενοστραφής γλώσσα χαμηλού επιπέδου, άρα προσφέρει μεγάλα περιθώρια βελτιστοποίησης της απόδοσης του παραγόμενου κώδικα. Η γλώσσα υποστηρίζει τα βασικά εργαλεία OpenMP και OpenMPI, καθώς και το CUDA API της NVIDIA. Το DES Framework αναπτύχθηκε και ελέγχθηκε για λειτουργικό σύστημα Ubuntu Linux, με τα χαρακτηριστικά που αναφέρονται στο κεφάλαιο 4. Η βιβλιοθήκη είναι σχεδιασμένη ώστε να μπορεί να εκτελείται μέσω MPI σε πολλά συστήματα ταυτόχρονα, ενώ στη συνέχεια όλα τα βήματα της διαδικασίας του υπολογισμού εκτελούνται αυτόματα. Αυτά συμπεριλαμβάνουν τον διαμοιρασμό των δεδομένων στους κόμβους του δικτύου



Εικόνα 2: Block διάγραμμα ενός τυπικού ετερογενούς συστήματος με πολυπύρηνους επεξεργαστές και μία κάρτα γραφικών

(υπολογιστές), τον δυναμικό διαμοιρασμό του φόρτου εργασίας για τη μεγιστοποίηση της αξιοποίησης των υπολογιστικών συστημάτων και την ελαχιστοποίηση του συνολικού χρόνου εκτέλεσης, καθώς και τη συλλογή των αποτελεσμάτων στον master κόμβο. Ο διαμοιρασμός γίνεται λαμβάνοντας υπ' όψη τις δυνατότητες κάθε συστήματος, και χρησιμοποιούνται τεχνικές βελτιστοποίησής του καθώς ο υπολογισμός εκτελείται. Υποστηρίζονται 2 τρόποι επιστροφής αποτελεσμάτων:

- **Όλα τα αποτελέσματα**: Θα επιστραφεί η έξοδος της συνάρτησης αξιολόγησης για **κάθε** σημείο του χώρου αναζήτησης, σε μορφή πίνακα.
- **Λίστα αποδεκτών σημείων**: Θα επιστραφεί μία λίστα με τα σημεία για τα οποία η συνάρτηση αξιολόγησης επέστρεψε ένα *αποδεκτό* αποτέλεσμα. Αυτό αποφασίζεται από τη συνάρτηση `toBool()` που υλοποιεί ο χρήστης.

Ένα σημαντικό χαρακτηριστικό που παρέχει το DES Framework είναι ότι θεωρεί πως η εκτέλεση του υπολογισμού μπορεί να γίνεται σε ένα δίκτυο υπολογιστών, οι οποίοι μπορούν να βρίσκονται οπουδήποτε στον κόσμο και να επικοινωνούν μέσω διαδικτύου, ή να βρίσκονται σε ένα τοπικό δίκτυο. Κάθε υπολογιστής μπορεί να αποτελείται από έναν ή περισσότερους πολυπύρηνους επεξεργαστές, καθώς και καμία, μία ή περισσότερες κάρτες γραφικών τεχνολογίας CUDA. Το κατανεμημένο σύστημα μπορεί να είναι ετερογενές (heterogenous), δηλαδή κάθε υπολογιστικός κόμβος του μπορεί να έχει διαφορετικές δυνατότητες επίδοσης. Ένα τέτοιο σύστημα περιγράφεται από την εικόνα 2 [27]. Ο χρήστης έχει τη δυνατότητα να επιλέξει ποιοι επεξεργαστικοί πόροι θα χρησιμοποιούνται σε κάθε σύστημα που συμμετέχει στον υπολογισμό, και το framework επιλέγει τον καλύτερο τρόπο διαμοιρασμού του φόρτου εργασίας σε αυτούς.

Η χρήση του DES Framework από έναν προγραμματιστή είναι αρκετά απλή. Ο χρήστης παρέχει την υλοποίηση ενός μοντέλου, έναν χώρο αναζήτησης, και μερικές παραμέτρους σχετικά με τη χρήση των πόρων του συγκεκριμένου συστήματος και του υπολογισμού (βλ. Είσοδος Δεδομένων). Το μοντέλο και αντίστοιχα ο χώρος αναζήτησης μπορούν να έχουν πολλές διαστάσεις. Για να ξεκινήσει ο υπολογισμός, ο προγραμματιστείς απλά καλεί μόνο μία συνάρτηση. Όταν ο υπολογισμός τελειώσει, η συνάρτηση αυτή επιστρέφει και το αρχικό πρόγραμμα έχει πρόσβαση στα αποτελέσματα του υπολογισμού από τον master υπολογιστή. Εάν τα αποτελέσματα μπορούν να εκφραστούν δυαδικά (αποδεκτά/μη αποδεκτά), τότε ο χρήστης μπορεί να επιλέξει να επιστραφούν σε μορφή λίστας, η οποία θα περιέχει τα σημεία

του χώρου αναζήτησης για τα οποία η συνάρτηση αξιολόγησης επέστρεψε αποδεκτό αποτέλεσμα. Διαφορετικά, ο χρήστης έχει τη δυνατότητα να ζητήσει τις εξόδους της συνάρτησης αξιολόγησης για *όλα* τα σημεία του χώρου αναζήτησης, τα οποία θα επιστραφούν σε μορφή πίνακα, με πλήθος διαστάσεων ίδιο με τον χώρο αναζήτησης. Η πρώτη περίπτωση είναι ιδιαίτερα απαιτητική ως προς τη μνήμη των slaves διότι πρέπει να δεσμευτεί μνήμη για τη περίπτωση που όλα τα σημεία που θα του ανατεθούν είναι αποδεκτά, ενώ η δεύτερη περίπτωση είναι απαιτητική ως προς τη μνήμη του master διότι πρέπει να δεσμευτεί μνήμη για κάθε σημείο του χώρου αναζήτησης.

3.2 Είσοδος Δεδομένων

Για τη λειτουργία του DES Framework απαιτούνται κάποια δεδομένα. Τα πιο βασικά είναι το μοντέλο του προβλήματος και ο χώρος αναζήτησης. Έπειτα χρειάζονται μερικές παράμετροι για τη χρήση των υπολογιστικών πόρων του κάθε συστήματος και για κάποιες λειτουργίες, που δε μπορούν να προβλεφθούν εκ των προτέρων χωρίς καμία γνώση για το μοντέλο.

Το DES Framework δέχεται την είσοδό του κατά την εκτέλεση του προγράμματος. Αυτό σημαίνει ότι ένα πρόγραμμα μπορεί εκτελέσει πολλές αναζητήσεις αλλάζοντας κάθε φορά το μοντέλο, τη διαστασιμότητα, και το χώρο αναζήτησης, χωρίς την ανάγκη για επαναμετάφραση του κώδικα ή τη κλήση άλλων εκτελέσιμων αρχείων.

3.2.1 Αναπαράσταση μοντέλου και χώρου αναζήτησης

Το μαθηματικό μοντέλο ενός προβλήματος είναι μία μαθηματική αναπαράσταση που το περιγράφει. Ένα πρόβλημα είναι πιθανό να έχει πολλές διαφορετικές μοντελοποιήσεις. Στα πλαίσια του DES Framework, το μοντέλο ορίζεται ως μία **συνάρτηση αξιολόγησης**, που δέχεται ως όρισμα ένα σύνολο από μεταβλητές παραμέτρους και επιστρέφει έναν αριθμό ως αποτέλεσμα. Προαιρετικά, αυτή η συνάρτηση μπορεί να παίρνει και πρόσθετα δεδομένα τα οποία όμως είναι στατικά κατά τη διάρκεια του υπολογισμού. Αυτό το μοντέλο μπορεί να περιγράφει οτιδήποτε, χωρίς κανέναν περιορισμό. Για να μπορεί να χρησιμοποιηθεί πρέπει μόνο να μπορεί να παρασταθεί ως μία συνάρτηση εκτελέσιμη από ένα ντετερμινιστικό σύστημα. Η έξοδος της συνάρτησης αξιολόγησης για ένα σημείο δε μπορεί να εξαρτάται από

την έξοδο της ίδιας συνάρτησης ενός άλλου σημείου, διότι, λόγω της παραλληλίας που χρησιμοποιείται, δεν υπάρχει καμία εγγύηση για τη σειρά με την οποία θα υπολογιστούν τα αποτελέσματα.

Το σύνολο των μεταβλητών που δέχεται η συνάρτηση αξιολόγησης είναι αριθμοί κινητής υποδιαστολής, και μπορεί να θεωρηθεί ως ένα διάνυσμα. Αν θεωρήσουμε ότι το μοντέλο έχει *D* μεταβλητές, τότε αυτό το διάνυσμα έχει *D* όρους και περιγράφει ένα σημείο στον $\mathbb{R}^D$. Ο **χώρος αναζήτησης** θα είναι ένα πεπερασμένο διακριτό υποσύνολο αυτού του χώρου και θα εκφράζεται ως ένα σύνολο 3άδων [άνω όριο ($high_d$), κάτω όριο (low_d), πλήθος σημείων (N_d)], μία για κάθε διάσταση. Η είσοδος της συνάρτησης αξιολόγησης θα είναι της μορφής $\boldsymbol{X} = [\boldsymbol{x_1}, \boldsymbol{x_2}, \ldots, \boldsymbol{x_D}]$ όπου:

$$x_d = low_d + n\frac{high_d - low_d}{N_d}, με\ n \in [0, N_d)$$

Εξίσωση 1: Υπολογισμός τιμής για μία διάσταση μέσω index

Αυτό δίνει συνολικά

$$N = \prod_{d=1}^{D} N_d$$

Εξίσωση 2: Υπολογισμός συνολικού πλήθος στοιχείων πολυδιάστατου πίνακα

σημεία του χώρου αναζήτησης. Για κάθε ένα από αυτά, θα πρέπει να εκτελεστεί η συνάρτηση αξιολόγησης.

Εσωτερικά της βιβλιοθήκης, το μοντέλο είναι μία κλάση με τις εξής 3 συναρτήσεις, τις οποίες ο χρήστης καλείται να υλοποιήσει:

- `validate_gpu()` και `validate_cpu()`: Αυτές είναι οι συναρτήσεις αξιολόγησης που θα χρησιμοποιηθούν. Η μία θα εκτελείται από κεντρικούς επεξεργαστές γενικής χρήσης και η άλλη από κάρτες γραφικών CUDA. Η διαφοροποίηση γίνεται διότι, όπως αναφέρθηκε στην ενότητα 2.2, οι κάρτες γραφικών χρησιμοποιούν διαφορετική αρχιτεκτονική άρα απαιτούν ειδικά μεταφρασμένο κώδικα. Ανάλογα τη περίπτωση, ίσως να απαιτούν (ελάχιστα) διαφορετική υλοποίηση για λόγους επιδόσεων (π.χ. *half-precision floating point arithmetic*, βλ. GPU).

- `toBool()`: Αυτή η συνάρτηση θα χρησιμοποιηθεί στη περίπτωση που ο χρήστης ζητήσει τα αποτελέσματα σε μορφή λίστας. Το αποτέλεσμα της συνάρτησης αξιολόγησης ενός σημείου θα δοθεί στη συνάρτηση `toBool()` η οποία θα επιστρέψει μία δυαδική τιμή (`true/false`) που θα δείχνει εάν το συγκεκριμένο αποτέλεσμα είναι αποδεκτό, άρα το αντίστοιχο σημείο θα πρέπει να συμπεριληφθεί στη τελική λίστα. Αυτή η συνάρτηση θα εκτελεστεί είτε σε επεξεργαστή είτε σε κάρτα γραφικών, αλλά η υλοποίησή της συνήθως θα είναι αρκετά απλή ώστε να μη χρειαστεί διαχωρισμός του κώδικα. Ο μεταφραστής θα αναλάβει να παράξει εκτελέσιμο κώδικα και για τις 2 αρχιτεκτονικές παίρνοντας μόνο μία υλοποίηση.

3.2.2 Αναπαράσταση δεδομένων

Τα βασικά δεδομένα που διαχειρίζεται το DES Framework είναι τα σημεία του χώρου αναζήτησης τα οποία μπορούν να παρασταθούν ως D-διάστατα διανύσματα. Εσωτερικά της βιβλιοθήκης, αυτό είναι ένας μονοδιάστατος πίνακας D στοιχείων που περιέχει τιμές αριθμητικής *single-precision floating point*. Τα στοιχεία αυτού του πίνακα παίρνουν συγκεκριμένες διακριτές τιμές, όπως δείχνει η εξίσωση 1.

Παρατηρώντας την εξίσωση 1 μπορούμε να δούμε ότι η τιμή για τη διάσταση d εξαρτάται μόνο από μία ακέραια τιμή $n \in [0, N_d)$ καθώς οι υπόλοιποι όροι είναι σταθεροί στα πλαίσια του υπολογισμού. Άρα το διάνυσμα X που κανονικά περιέχει τιμές τύπου *floating-point* μπορεί για απλοποίηση να παρασταθεί με ένα διάνυσμα ακέραιων τιμών $\boldsymbol{I} = [\boldsymbol{n_1}, \boldsymbol{n_2}, \ldots, \boldsymbol{n_D}]$ όπου n_d η ακέραια τιμή n για την εξίσωσης 1 της αντίστοιχης διάστασης d. Για παράδειγμα, αν έχουμε τον 3-διάστατο χώρο με:

- Διάσταση 1: $low_1 = 0,\ high_1 = 1,\ N_1 = 10 \Leftrightarrow step_1 = 0.1$
- Διάσταση 2: $low_2 = 0,\ high_2 = 10,\ N_2 = 10 \Leftrightarrow step_2 = 1$
- Διάσταση 3: $low_3 = 1,\ high_3 = 3,\ N_3 = 4 \Leftrightarrow step_3 = 0.5$

τότε το σημείο $X_1 = [0.2, 3, 1.5]$ θα είχε $I_1 = [2, 3, 1]$ και το σημείο $X_2 = [0.8, 0, 2.5]$ θα είχε $I_2 = [8, 0, 3]$.

Η αναπαράσταση του διανύσματος I μπορεί να απλοποιηθεί κι άλλο. Ας σκεφτούμε αρχικά με ποιο τρόπο αποθηκεύεται ένας πίνακας σε μία μνήμη υπολογιστή, η οποία από τη φύση της είναι μονοδιάστατη (αναφερόμαστε σε μία θέση με έναν ακέραιο αριθμό). Εάν ο

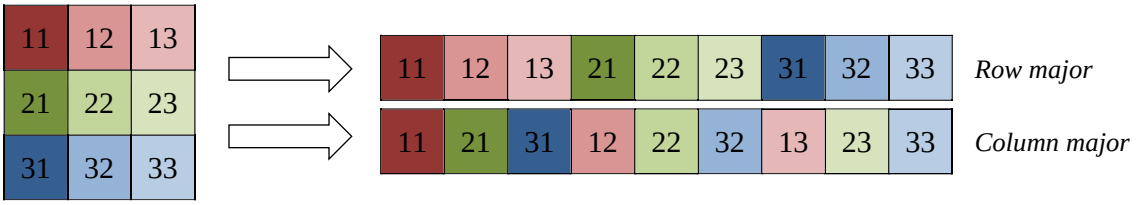
πίνακας έχει μόνο μία διάσταση τότε είναι μία ακολουθία αριθμών, άρα τα στοιχεία του αποθηκεύονται σε διαδοχικές θέσεις μνήμης. Για να τα προσπελάσουμε το i-οστό στοιχείο του πίνακα (*θεωρούμε ότι το πρώτο στοιχείο του πίνακα έχει* $i = 0$*, αλλά υπάρχουν και γλώσσες όπως Fortran και η MATLAB που θεωρούν ότι το πρώτο στοιχείο βρίσκεται στη θέση 1*), χρειαζόμαστε την διεύθυνση όπου είναι αποθηκευμένο το πρώτο στοιχείο, στην οποία προσθέτουμε το i πολλαπλασιασμένο με το μέγεθος κάθε στοιχείου (πολλαπλασιάζουμε με το μέγεθος διότι ένα στοιχείο μπορεί να καταλαμβάνει πάνω από μία θέση μνήμης), και πλέον έχουμε τη διεύθυνση του στοιχείου που ψάχνουμε. Ο αριθμός i ονομάζεται **δείκτης** (index). Σε μία διάσταση λοιπόν η δεικτοδότηση είναι αρκετά απλή.

Στις 2 διαστάσεις η πολυπλοκότητα της απεικόνισης αυξάνει. Ένας διδιάστατος πίνακας $N \times M$ δεν είναι τίποτα παραπάνω από ένα σύνολο N μονοδιάστατων πινάκων μεγέθους M (ή M μονοδιάστατων πινάκων μεγέθους N). Άρα μπορεί να απεικονιστεί σε μία μονοδιάστατη μνήμη με 2 βασικούς τρόπους: Είτε αποθηκεύοντας τις γραμμές του πίνακα σειριακά (row major order), είτε αποθηκεύοντας τις στήλες του πίνακα σειριακά (column major order), όπως φαίνεται στην εικόνα 3. Παραδοσιακά, σε γλώσσες όπως η C χρησιμοποιείται η πρώτη τεχνική, ενώ άλλες γλώσσες όπως η Fortran και η MATLAB χρησιμοποιούν τη δεύτερη. Για ειδικές περιπτώσεις πινάκων (αραιά μητρώα, μητρώα ειδικής μορφής όπως διαγώνια, ζώνης, toeplitz) υπάρχουν άλλες τεχνικές αποθήκευσης, όπως COO, CSR, CSC, [28] οι οποίες όμως δε θα μας απασχολήσουν στα πλαίσια της εργασίας. Άρα λοιπόν, ένας πίνακας $N \times M$ αποθηκεύεται ως ένας μονοδιάστατος πίνακας μεγέθους $N \cdot M$. Η δεικτοδότηση όμως δεν είναι τόσο απλή όσο στο μονοδιάστατο πίνακα. Για να προσπελάσουμε το στοιχείο της γραμμής i και στήλης j του αρχικού πίνακα, πρέπει να βρούμε που ακριβώς στη μονοδιάστατη μνήμη θα έχει τοποθετηθεί. Θεωρώντας ότι χρησιμοποιείται row major απεικόνιση, ο δείκτης υπολογίζεται από τον τύπο:

$$index = i \cdot M + j$$

Εξίσωση 3: Υπολογισμός index για διδιάστατο πίνακα

Ουσιαστικά προσπερνάμε τα πρώτα $i \cdot M$ στοιχεία τα οποία αντιστοιχούν στις πρώτες i γραμμές μεγέθους M του διδιάστατου πίνακα, και έπειτα προχωράμε ακόμη j στοιχεία ώστε να βρούμε αυτό που θέλουμε.



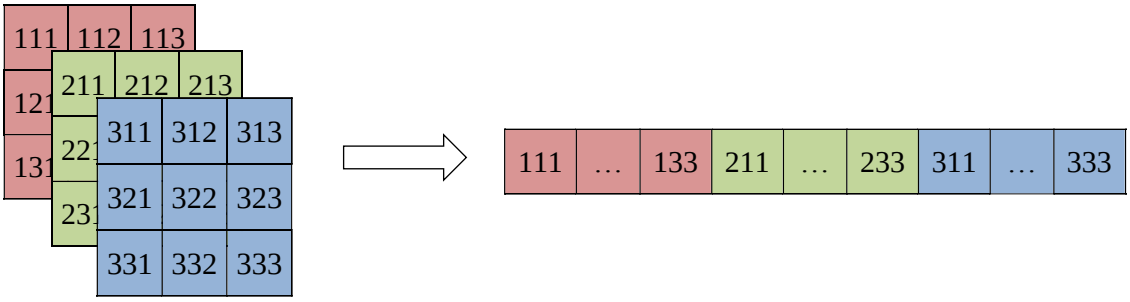
Εικόνα 3: Διδιάστατος πίνακας σε row-major και column-major αναπαράσταση

Προχωρώντας στις 3 διαστάσεις, θα χρησιμοποιήσουμε ακριβώς την ίδια ιδέα αλλά σε 3 επίπεδα. Ένας 3-διάστατος πίνακας $N \times M \times K$ δεν είναι τίποτα παραπάνω από K διδιάστατοι πίνακες $N \times M$, οι οποίοι όπως είδαμε είναι μονοδιάστατοι πίνακες μήκους $N \cdot M$. Ακολουθώντας την ίδια λογική, ο τύπος για τον υπολογισμό του index για το στοιχείο στη θέση (i, j, k) γίνεται:

$$index = k \cdot N \cdot M + j \cdot M + i$$

Εξίσωση 4: Υπολογισμός index για 3-διάστατο πίνακα

θεωρώντας ότι για την απεικόνιση των διδιάστατων πινάκων χρησιμοποιείται row major απεικόνιση. Στην παραπάνω εξίσωση, προσπερνάμε τα πρώτα $k \cdot N \cdot M$ στοιχεία που αντιστοιχούν στους πρώτους k διδιάστατους πίνακες $N \times M$, έπειτα (αφού είμαστε στον διδιάστατο πίνακα που θέλουμε) προσπερνάμε τα $j \cdot M$ στοιχεία που αντιστοιχούν στις πρώτες j γραμμές του πίνακα, και τέλος προχωράμε άλλες i θέσεις για να βρούμε το ζητούμενο στοιχείο. Άρα μπορούμε να αναπαραστήσουμε έναν 3-διάστατο πίνακα σε μία μονοδιάστατη μνήμη, όπως φαίνεται στην εικόνα 4.



Εικόνα 4: Αναπαράσταση 3-διάστατου πίνακα σε μονοδιάστατη μνήμη

Γενικεύοντας, μπορούμε να χρησιμοποιήσουμε τη μονοδιάστατη μνήμη που έχουμε στη διάθεσή μας ώστε να αποθηκεύσουμε τα περιεχόμενα ενός D-διάστατου πίνακα, όπου το D είναι άγνωστο κατά τη συγγραφή και μετάφραση του κώδικα. Υποθέτοντας ότι ο D-διάστατος πίνακας έχει διαστάσεις $N_1 \times N_2 \times \ldots \times N_D$ τότε θα τον απεικονίσουμε σε μία διάσταση χρησιμοποιώντας τον τύπο:

$$index = \sum_{d=1}^{D} \left(n_d \prod_{i=d+1}^{D} N_i \right)$$

Εξίσωση 5: Υπολογισμός index για D-διάστατο πίνακα από διάνυσμα δεικτών

για να μεταφράσουμε το σημείο $(n_1, n_2, \ldots, n_D)$ σε δείκτη μονοδιάστατης απεικόνισης.

Χρησιμοποιώντας τη λογική απεικόνισης ενός πολυδιάστατου πίνακα σε μονοδιάστατο χώρο, το διάνυσμα $I = [n_1, n_2, \ldots, n_D]$ μπορεί να μετασχηματιστεί σε έναν μοναδικό ακέραιο χρησιμοποιώντας την εξίσωση 5. Ο παραπάνω μετασχηματισμός είναι αντιστρέψιμος: μπορούμε να πάρουμε το αρχικό διάνυσμα I με τον αλγόριθμο 1:

```
1  Έστω index = Εξίσωση 5([n1, n2, …, nD])
2  Για κάθε διάσταση d = D:-1:1    (σημαντική η σειρά εκτέλεσης: D -> 1)
3      Βήμα = ∏_{1<i≤d} N_i
4      n_d = index / Βήμα           (ακέραια διαίρεση)
5      index = index % Βήμα         (υπόλοιπο ακέραιας διαίρεσης)
6  end
```

Αλγόριθμος 1: Υπολογισμός διανύσματος δεικτών από μονοδιάστατο index

Έπειτα το διάνυσμα X μπορεί να ανακτηθεί μέσω της εξίσωσης 1, χρησιμοποιώντας τα $n_1, n_2, \ldots, n_D$ που προέκυψαν από τον παραπάνω αλγόριθμο.

Άρα τελικά μπορούμε να παραστήσουμε κάθε σημείο του πεπερασμένου διακριτού χώρου αναζήτησης με έναν μοναδικό αριθμό. Καθώς ο υπολογισμός εκτελείται και οι διεργασίες χρειάζονται να επικοινωνήσουν σύνολα σημείων προς επεξεργασία, αρκούν μόνο 2 αριθμοί: ο μονοδιάστατος δείκτης που αντιστοιχεί στο πρώτο πολυδιάστατο σημείο του συνόλου, και το πλήθος των σημείων. Θεωρώντας ότι γενικά τα σημεία μπορούν να υπολογιστούν με οποιαδήποτε σειρά, η σειρά που χρησιμοποιείται είναι αυτή που δίνεται όταν,

για ένα δεδομένο σημείο, το επόμενό του υπολογίζεται αυξάνοντας το index της πρώτης διάστασης κατά 1, και προωθώντας τα ενδεχόμενα κρατούμενα στις επόμενες.

3.2.3 Παράμετροι εκτέλεσης DES Framework

Για την εκτέλεση του υπολογισμού απαιτούνται οι παρακάτω παράμετροι.

- `batchSize`: Το μέγιστο πλήθος σημείων που μπορεί να ανατεθεί σε ένα slave σύστημα προς επεξεργασία. Στη γενική περίπτωση αυτό θα είναι ένας πολύ μεγάλος αριθμός (όπως `ULONG_MAX`), αλλά δίνεται η δυνατότητα μείωσής του για μείωση της χρήσης της μνήμης ενός συστήματος. Αυτή η μεταβλητή μπορεί να οριστεί ξεχωριστά σε κάθε σύστημα που συμμετέχει στον υπολογισμό κατά την αρχικοποίησή του. Εάν ο master κόμβος του υπολογισμού το κρίνει απαραίτητο μέσω της self-scheduling τεχνικής που θα χρησιμοποιεί, μπορεί να μειώσει το μέγεθος του chunk κάτω από τη δοσμένη τιμή (βλ. Loop Self-scheduling).
- `processingType`: Ορίζει ποιοι επεξεργαστικοί πόροι του συστήματος θα χρησιμοποιηθούν στον υπολογισμό. Αυτή η παράμετρος μπορεί να είναι μία από τις τιμές `CPU`, `GPU`, ή `BOTH`. Ως `CPU` εννοούνται όλοι οι επεξεργαστικοί πυρήνες του κεντρικού επεξεργαστή, ή όλων των επεξεργαστών αν το σύστημα έχει περισσότερους από έναν. Ως `GPU` εννοούνται όλες οι συμβατές κάρτες γραφικών που είναι εγκατεστημένες στο σύστημα. Ως `BOTH` εννοούνται όλοι οι επεξεργαστές και όλες οι κάρτες γραφικών.
- `resultSaveType, saveFile`: Η παράμετρος `resultSaveType` ορίζει σε τι μορφή θα αποθηκευτούν τα αποτελέσματα. Δέχεται τις επιλογές `ALL` και `LIST`. Με την επιλογή `ALL`, θα αποθηκευτούν τα αποτελέσματα της συνάρτησης αξιολόγησης για όλα τα σημεία του χώρου αναζήτησης και θα επιστραφούν στο χρήστη σε μορφή πίνακα. Για μεγάλο χώρο αναζήτησης αυτό απαιτεί μεγάλη χωρητικότητα μνήμης, οπότε δίνεται η παράμετρος `saveFile`, που μπορεί να οριστεί ώστε τα αποτελέσματα να αποθηκευτούν απευθείας σε αρχείο σε κάποιο σκληρό δίσκο. Με την επιλογή `LIST` θα αποθηκευτούν μόνο τα σημεία του χώρου αναζήτησης για τα οποία η συνάρτηση `toBool(result)` επιστρέφει `true` και θα επιστραφεί ένας πίνακας με τα αντίστοιχα διανύσματα X παραθετημένα στη σειρά.

- `slaveBalancing`: Ενεργοποιεί τη χρήση δυναμικού self-scheduling στο επίπεδο τον συστημάτων του δικτύου (βλ. Loop Self-scheduling). Έτσι, η master διεργασία θα αναθέτει διαφορετικό πλήθος σημείων κάθε φορά σε κάθε slave ώστε να εξισορροπήσει το φόρτο εργασίας ανάμεσα στους κόμβους του δικτύου. Αυτή η τεχνική τίθεται σε λειτουργία αφού πρώτα *όλοι* οι κόμβοι έχουν φέρει εις πέρας τουλάχιστον ένα κομμάτι υπολογισμού που τους ανατέθηκε.
- `threadBalancing`: Ενεργοποιεί τη χρήση δυναμικού self-scheduling στο επίπεδο του συστήματος, μεταξύ του επεξεργαστή και των καρτών γραφικών (βλ. Loop Self-scheduling). Όταν αυτή η επιλογή είναι ενεργοποιημένη, κάθε slave σύστημα θα χρησιμοποιεί μία τεχνική δυναμικού self-scheduling ανεξάρτητα από τα υπόλοιπα, ώστε να εξισορροπήσει το φόρτο εργασίας ανάμεσα στους επεξεργαστικούς πόρους της. Αυτή η τεχνική τίθεται σε λειτουργία από τη δεύτερη ανάθεση υπολογισμών που θα δεχτεί το σύστημα.
- `benchmark`: Όταν είναι `true` απενεργοποιεί τις μεταφορές δεδομένων και τις δεσμεύσεις μνήμης ώστε να γίνει χρονομέτρηση του υπολογιστικού μέρους της εκτέλεσης. Μπορεί να χρησιμοποιηθεί για λόγους βελτιστοποίησης άλλων παραμέτρων.
- `dataPtr`, `dataSize`: Το DES Framework δίνει τη δυνατότητα να περαστούν πρόσθετα στατικά δεδομένα στη συνάρτηση αξιολόγησης. Αυτό χρησιμεύει όταν δε θέλουμε να έχουμε hardcoded τα δεδομένα του μοντέλου στον κώδικά του. Αυτά τα δεδομένα θα βρίσκονται μόνιμα στη μνήμη και δεν επηρεάζουν σημαντικά την απόδοση του συστήματος.
- `blockSize`: Ορίζει το μέγεθος των thread blocks που θα χρησιμοποιούνται από τις κάρτες γραφικών (βλ. Λεπτομέρειες υπολογισμού σε GPU). Η μέγιστη τιμή είναι 1024 και γενικά συμφέρει να είναι μεγάλη. Δίνεται η δυνατότητα μείωσής της ώστε να διατεθούν περισσότεροι πόροι σε κάθε GPU thread (περισσότεροι registers). Αυτή η μείωση γίνεται απαραίτητη όταν η διαστασιμότητα του προβλήματος μεγαλώνει πολύ. Εξαρτάται από τις κάρτες γραφικών που χρησιμοποιούνται και μπορεί να ρυθμιστεί ανεξάρτητα για κάθε σύστημα του δικτύου.
- `computeBatchSize`: Το πλήθος των στοιχείων που θα υπολογίζει κάθε GPU thread σε κάθε κλήση του kernel (βλ. Ανάθεση πολλών σημείων ανά GPU thread). Πειραματικά

προσδιορίστηκε ότι τιμές της τάξης των 250-2000 αποδίδουν καλύτερα, αλλά εξαρτάται από το μοντέλο του προβλήματος και τις κάρτες γραφικών.

- `gpuStreams`: Το πλήθος των streams που θα χρησιμοποιούνται κατά την εκτέλεση του υπολογισμού (βλ. GPU Streams). Οι τιμές για αυτή τη παράμετρο κυμαίνονται μεταξύ 1-32, ανάλογα την αρχιτεκτονική της κάρτας και την υλοποίηση του μοντέλου που χρησιμοποιείται. Πειραματικά προσδιορίστηκε ότι τιμές μεταξύ 4-12 αποδίδουν καλύτερα.
- `overrideMemoryRestrictions`: Επιτρέπει τη παράβλεψη των ορίων διαθέσιμης μνήμης για να είναι δυνατή η ανάθεση μεγαλύτερων chunks. Μπορεί να χρησιμοποιηθεί μόνο στη περίπτωση που τα αποτελέσματα αποθηκεύονται σε μορφή λίστας σημείων. Η χρήση του προορίζεται για τη περίπτωση που ο χώρος αναζήτησης είναι πολύ μεγάλος αλλά η τελική λίστα σημείων αναμένεται να είναι πολύ μικρότερη.
- `slowStartBase`, `slowStartLimit`: Οι παράμετροι της τεχνικής Slow-start μέγιστης ανάθεσης.

3.3 Δομή Δικτύου

Η αναζήτηση εκτελείται σε ένα δίκτυο υπολογιστών το οποίο μπορεί να έχει οποιαδήποτε φυσική μορφή. Εικονικά, σχηματίζεται ένα δίκτυο master/slave 2 επιπέδων, το ένα στο επίπεδο του δικτύου, και το άλλο στο επίπεδο του συστήματος.

Το πρώτο επίπεδο χωρίζει τους υπολογιστές μεταξύ τους, ορίζοντας έναν από αυτούς ως master και τους υπόλοιπους ως slaves. Αυτό το επίπεδο υλοποιείται με OpenMPI, όπου ο κάθε υπολογιστής έχει ένα μοναδικό `rank`. Ο υπολογιστής με `rank` 0 είναι ο master. Ο master μπορεί να συμπεριφέρεται και ως slave, και εντός της βιβλιοθήκης θα αντιμετωπίζεται σαν δύο ξεχωριστά συστήματα. Ο master επικοινωνεί με όλους τους κόμβους για να τους αναθέσει εργασίες ή για να συλλέξει αποτελέσματα. Οι slave κόμβοι επικοινωνούν μόνο με τον master και είναι πάντα αυτοί που ξεκινούν την επικοινωνία. Αυτό συμβαίνει όταν βρίσκονται σε αδράνεια και είναι έτοιμοι να αναλάβουν νέους υπολογισμούς, ή έχουν τελειώσει με αυτούς που τους ανατέθηκαν και θέλουν να επιτρέψουν τα αποτελέσματα. Ως «υπολογισμούς»

εννοούμε την εκτέλεση της συνάρτησης αξιολόγησης που αντιστοιχεί στο μοντέλο σε ένα σύνολο σημείων του χώρου αναζήτησης. Εσωτερικά, η πρώτη επικοινωνία μεταφράζεται σε ένα αίτημα `READY` από τον slave προς τον master, που απαντάται με 2 αριθμούς: το πολυδιάστατο σημείο εκκίνησης μετασχηματισμένο στον μονοδιάστατο χώρο, και το πλήθος των σημείων ξεκινώντας από το σημείο εκκίνησης. Έπειτα ο slave επεξεργάζεται τα στοιχεία και επιστρέφει τα αποτελέσματα στον master. Εσωτερικά, αυτό είναι ένα αίτημα `RESULTS` από τον slave προς τον master, που ακολουθείται από τα αποτελέσματα. Αυτά μπορεί να είναι είτε σε μορφή πίνακα είτε σε μορφή λίστας, ανάλογα με το τι έχει επιλέξει ο χρήστης κατά την αρχικοποίηση. Σε περίπτωση που τα αποτελέσματα είναι σε μορφή πίνακα, τότε αυτός είναι ένας μονοδιάστατος πίνακας που αντιστοιχεί σε ένα συνεχόμενο κομμάτι της μονοδιάστατης απεικόνισης του πολυδιάστατου χώρου αναζήτησης, ξεκινώντας από το σημείο εκκίνησης που ανατέθηκε στον slave.

Το δεύτερο επίπεδο master/slave υλοποιείται στο επίπεδο του κάθε συστήματος. Κάθε σύστημα μπορεί να διαθέτει πολλούς πόρους, όπως πολλούς επεξεργαστικούς πυρήνες και πολλές κάρτες γραφικών. Όλοι αυτοί έχουν διαφορετικές επιδόσεις, άρα οι υπολογισμοί πρέπει να μοιραστούν με τρόπο που να μη δημιουργείται ανισορροπία. Σε κάθε slave σύστημα εκτελούνται πολλά threads. Ένα από αυτά αναλαμβάνει το ρόλο του master και τα υπόλοιπα αναλαμβάνουν από ένα επεξεργαστικό στοιχείο (`CPU`, `GPU0`, `GPU1`, …) και λειτουργούν ως slaves. Αυτό το επίπεδο υλοποιείται με OpenMP, το οποίο δημιουργεί να νήματα και τους αναθέτει ρόλους με βάση τα thread ID τους. Το master thread (στα πλαίσια της βιβλιοθήκης αναφέρεται και ως coordinator thread – νήμα συντονιστή) είναι υπεύθυνο για την επικοινωνία με τον master κόμβο του δικτύου για την ανάθεση υπολογισμών και την επιστροφή αποτελεσμάτων, καθώς και για το διαμοιρασμό του φόρτου εργασίας στα slave threads. Τα slave threads (στα πλαίσια της βιβλιοθήκης αναφέρονται και ως worker threads) περιμένουν τον συντονιστή να τους αναθέσει εργασία, την ολοκληρώνουν, και του επιστρέφουν τα αποτελέσματα. Κάθε worker thread αναλαμβάνει ένα επεξεργαστικό στοιχείο: δημιουργείται ένα νήμα για τον επεξεργαστή (ή τους επεξεργαστές, καθώς αντιμετωπίζονται ως ένας) και ένα για κάθε διαθέσιμη κάρτα γραφικών. Το worker thread που αναλαμβάνει τον/τους επεξεργαστή/ές, είναι υπεύθυνο για να μοιράσει τους υπολογισμούς που θα του ανατεθούν σε όλους τους διαθέσιμους επεξεργαστικούς πυρήνες. Οι επεξεργαστικοί πυρήνες ενός επεξεργαστή είναι όμοιοι μεταξύ τους, και οι επεξεργαστές ενός συστήματος συνήθως είναι ίδιοι μεταξύ τους, άρα μπορούμε να υποθέσουμε ότι όλοι έχουν όμοια επεξεργαστική ισχύ και

τους αναθέτουμε το ίδιο πλήθος σημείων προς επεξεργασία. Έτσι αποφεύγουμε το overhead ενός ακόμα επιπέδου δυναμικού self-scheduling. Μία εναλλακτική επιλογή θα ήταν να δημιουργήσουμε ένα worker thread για κάθε επεξεργαστικό πυρήνα και να αφήσουμε το coordinator thread να αναθέσει υπολογισμούς σε αυτούς. Αυτό όμως θα έδινε ένα έξτρα overhead στον συντονιστή καθώς πρέπει να συντονίσει περισσότερα worker threads, και δε θα ήταν σίγουρο ότι θα δουλέψει διότι κάθε υπολογιστικό νήμα μπορεί ανά πάσα στιγμή να μεταφερθεί σε άλλον φυσικό επεξεργαστικό πυρήνα του συστήματος (αυτό είναι θέμα του λειτουργικού συστήματος).

Στο πρώτο επίπεδο η δουλειά του master μπορεί να είναι χρονοβόρα, και η γρήγορη ανταπόκριση σε αιτήματα είναι κρίσιμη για τη μεγιστοποίηση της απόδοσης του δικτύου. Ο master κόμβος είναι hot spot για το δίκτυο, διότι όλοι οι κόμβοι πρέπει να επικοινωνήσουν μαζί του όταν είναι έτοιμοι για ανάθεση σημείων ή έχουν έτοιμα αποτελέσματα. Εάν για κάποιο λόγο καθυστερήσει να απαντήσει σε κάποιο αίτημα, τότε ο αντίστοιχος slave κόμβος μένει αδρανής μέχρι να λάβει απάντηση, άρα σπαταλάται χρόνος. Για το λόγο αυτό συμφέρει να αναθέτονται μεγάλα κομμάτια υπολογισμού στους slaves, ώστε ο master να μη δέχεται συχνά αιτήματα και έτσι να ελαχιστοποιηθεί η πιθανότητα να έρθουν πολλά αιτήματα ταυτόχρονα.

Στο δεύτερο επίπεδο το overhead του συντονισμού είναι ελάχιστο. Η διαδικασία του διαμοιρασμού των υπολογισμών εξαρτάται μόνο από το πλήθος των worker threads το οποίο είναι σταθερό, και δεν υπάρχουν μεταφορές μνήμης όπως στο πρώτο επίπεδο καθώς στο επίπεδο του συστήματος όλα τα worker threads έχουν πρόσβαση σε κοινή μνήμη. Ο συντονιστής δίνει στα worker threads πρόσβαση σε μία περιοχή μνήμης και τα αποτελέσματα αποθηκεύονται απευθείας εκεί που τα περιμένει αυτός, χωρίς να απαιτείται οποιαδήποτε μεταφορά. Οι κάρτες γραφικών ωστόσο θα πρέπει υποχρεωτικά να κάνουν μία μεταφορά δεδομένων, καθώς μπορούν μόνο να γράφουν στη δική τους μνήμη, ενώ τα αποτελέσματα πρέπει να συγκεντρωθούν στη κύρια μνήμη του συστήματος. Αυτό το αναλαμβάνει το αντίστοιχο worker thread. Το δεύτερο επίπεδο master/slave χρησιμοποιεί το μοντέλο fork/join αντί για αίτημα/απάντηση. Όταν ένα σύνολο σημείων είναι έτοιμο για επεξεργασία, ο συντονιστής αποφασίζει πόσα στοιχεία θα αναλάβει κάθε worker thread. Κάνει τις απαραίτητες ενέργειες και εκκινεί τα worker threads (λειτουργία fork). Όταν όλοι οι υπολογισμοί ολοκληρωθούν, τα worker threads σταματούν και ο συντονιστής συνεχίζει (λειτουργία join). Είναι δουλειά του συντονιστή να μοιράσει κατάλληλα τα σημεία για επεξεργασία στα worker threads ώστε αυτά να τερματίσουν μαζί. Διαφορετικά, κάποιοι πόροι του συστήματος μπορεί

να είναι σε αδράνεια ενώ υπάρχουν ακόμα στοιχεία προς επεξεργασία. Αυτό επιτυγχάνεται μέσω της τεχνικής self-scheduling που χρησιμοποιείται (βλ. Loop Self-scheduling).

3.4 Loop Self-scheduling

3.4.1 Τεχνική HPLS

Μία από τις σημαντικότερες λειτουργίες του DES Framework είναι το δυναμικό loop self-scheduling που χρησιμοποιείται για τον διαμοιρασμό του φόρτου εργασίας στα επιμέρους επεξεργαστικά στοιχεία. Όπως ήδη είδαμε, η δυναμική ανάθεση στοιχείων στους πόρους του δικτύου είναι μία βασική προϋπόθεση για να μεγιστοποιήσουμε την ταχύτητα του υπολογισμού. Ο σωστός διαμοιρασμός των στοιχείων προς επεξεργασία είναι απαραίτητος τόσο στο επίπεδο του δικτύου όσο και στο επίπεδο του κάθε συστήματος, ώστε να περιορίζουμε στο ελάχιστο τους επεξεργαστικούς πόρους που είναι σε αδράνεια. Και στα δύο επίπεδα χρησιμοποιείται μία παραλλαγή της τεχνικής Heuristic Parallel Loop Scheduling (HPLS) [2]. Αυτή η τεχνική χρησιμοποιεί δεδομένα σχετικά με την απόδοση των υπολογιστικών πόρων ώστε να προσδιορίσει το βέλτιστο πλήθος στοιχείων για την επόμενη ανάθεση σε κάθε έναν από αυτούς. Αυτό επιτυγχάνεται χρησιμοποιώντας μία συνάρτηση απόδοσης (Performance Function – PF) που υπολογίζει μία αναλογία απόδοσης (Performance Ratio – PR) για κάθε σύστημα. Η αναλογία απόδοσης τελικά θα χρησιμοποιηθεί για τον υπολογισμό του πλήθους των στοιχείων που θα ανατεθούν σε ένα σύστημα, ή στον υπολογιστικό πόρο ενός συστήματος. Στο [2] η συνάρτηση απόδοσης ορίστηκε ως:

$$PF\_Gen_i = \frac{1/t_i}{\sum_{\forall node j \in S} 1/t_j}$$

Εξίσωση 6: Γενική μορφή της PF

όπου t_i ο χρόνος εκτέλεσης ενός συγκεκριμένου προγράμματος στον κόμβο i. Αυτή η υλοποίηση προϋποθέτει τη γνώση του χρόνου εκτέλεσης κάποιου προγράμματος για *κάθε* σύστημα του δικτύου που συμμετέχει στον υπολογισμό, και δε λαμβάνει υπ' όψη της την απόδοση των συστημάτων για τα συγκεκριμένα δεδομένα και υπό τις συγκεκριμένες συνθήκες υπολογισμού.

Στο DES Framework χρησιμοποιήθηκε μία παραλλαγή της PF_Gen:

$$PF\_DES_i = \frac{score_i}{\sum_{\forall node j \in S} score_j}, score_j = \frac{N_j}{T_j}$$

Εξίσωση 7: PF τεχνικής HPLS

όπου N_j το τελευταίο πλήθος στοιχείων που ανατέθηκαν στον πόρο j, T_j ο χρόνος που χρειάστηκε ο πόρος j για να υπολογίσει τα αποτελέσματα για τα N_j στοιχεία, και S το σύνολο των πόρων στο επίπεδο όπου εφαρμόζεται η HLPS. Για το πρώτο επίπεδο οι πόροι είναι τα συστήματα του δικτύου που συμμετέχουν στον υπολογισμό, ενώ για το δεύτερο επίπεδο οι πόροι είναι τα επεξεργαστικά στοιχεία του συστήματος (CPU, GPUs) όπου εφαρμόζεται το self-scheduling. Το κύριο πλεονέκτημα της PF_DES σε σχέση με την PF_Gen είναι ότι η PF_DES δίνει μία εκτίμηση βασισμένη σε πραγματικά δεδομένα που υπολογίζονται σε κάθε βήμα του υπολογισμού, αντί για στατικά δεδομένα που πρέπει να έχουν υπολογιστεί εκ των προτέρων. Οι μετρήσεις σε πραγματικό χρόνο λαμβάνουν υπ' όψη τους όλους τους παράγοντες που συνεισφέρουν στο χρόνο εκτέλεσης ενός υπολογισμού, συμπεριλαμβανομένων των μεταφορών δεδομένων, της αρχικοποίησης πόρων, τον συντονισμό/διαμοιρασμό των σημείων σε επόμενο επίπεδο, και φυσικά την ίδια την απόδοση των επεξεργαστικών στοιχείων πάνω στο συγκεκριμένο μοντέλο. Αυτό συμβαίνει διότι ο χρόνος T_j στο επίπεδο του δικτύου είναι ο χρόνος που μετράει ο master μεταξύ της ανάθεσης σημείων και της επιτυχούς μεταφοράς των αποτελεσμάτων. Έτσι, στον χρόνο αυτό συμπεριλαμβάνεται οποιαδήποτε καθυστέρηση του δικτύου και του slave συστήματος, δίνοντας έτσι μία πιο ρεαλιστική εκτίμηση της τελικής απόδοσής του.

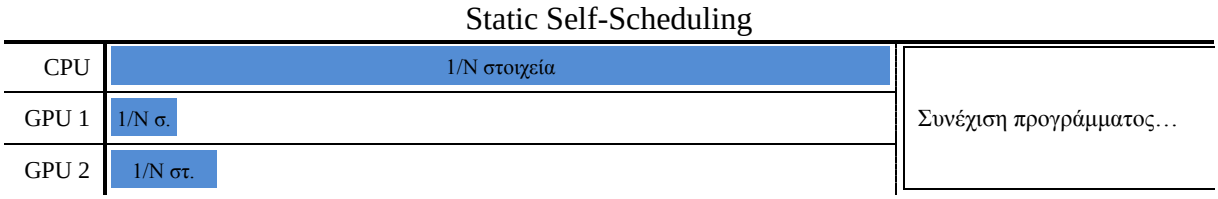
Οι έξοδοι της εξίσωσης 7 για κάθε πόρο του επιπέδου όπου εφαρμόζεται αθροίζονται σε 1 όπως αποδεικνύεται από την εξίσωση 8:

$$\begin{aligned}\sum_{\forall\, node\, i \in S} PF\_DES_i &= \sum_{\forall\, node\, i \in S} \frac{score_i}{\sum_{\forall\, node\, j \in S} score_j} \\ &= \frac{1}{\sum_{\forall\, node\, j \in S} score_j} \sum_{\forall\, node\, i \in S} score_i \\ &= 1 \ \square\end{aligned}$$

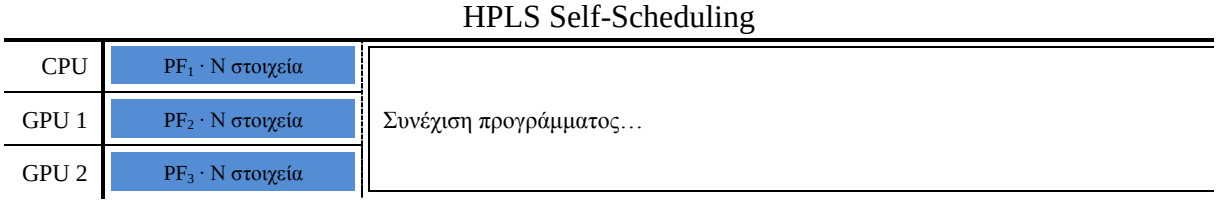
Εξίσωση 8: Απόδειξη αθροίσματος όρων της PF

Η συνάρτηση απόδοσης (είτε PF_DES είτε PF_Gen) δίνει μία αναλογία που αντιστοιχεί στην επεξεργαστική ισχύ ενός πόρου σε σχέση με τους υπόλοιπους του ίδιου επιπέδου. Αν έχουμε N σημεία να μοιράσουμε, τότε αναθέτουμε $N_i = PF_i \cdot N$ σε κάθε πόρο i, δηλαδή όσα του αντιστοιχούν με βάση την απόδοσή του.

Η χρησιμότητα της παραπάνω τεχνικής με τη συγκεκριμένη συνάρτηση απόδοσης φαίνεται έντονα στο δεύτερο επίπεδο του δικτύου, όπου ένα σύνολο σημείων μοιράζεται στους επεξεργαστικούς πόρους ενός συστήματος. Επειδή χρησιμοποιείται το μοντέλο fork/join, όλοι οι επεξεργαστικοί πόροι ξεκινούν τον υπολογισμό ταυτόχρονα. Εφαρμόζοντας τη παραπάνω υλοποίηση της HLPS για να μοιράσουμε τα σημεία σε επεξεργαστή και κάρτες γραφικών, και θεωρώντας αξιόπιστες μετρήσεις χρόνου και μικρές αποκλίσεις στους χρόνους εκτέλεσης της συνάρτησης αξιολόγησης για κάθε σημείο, παρατηρείται ότι όντως οι επεξεργαστικοί πόροι τερματίζουν σε πολύ όμοιους χρόνους, άρα κανένας πόρος δε μένει αδρανής ενώ υπάρχει δουλειά.



Εικόνα 5: Χρονοδιάγραμμα εκτέλεσης με στατική ανάθεση στοιχείων



Εικόνα 6: Χρονοδιάγραμμα εκτέλεσης με χρήση της τεχνικής HPLS

Χωρίς να είναι τόσο ξεκάθαρο, η ίδια τεχνική αποδίδει το ίδιο αποτελεσματικά και στο πρώτο επίπεδο, δηλαδή το επίπεδο του δικτύου. Εκεί, οι εκτελέσεις κάθε συστήματος ξεκινούν και τελειώνουν ανεξάρτητα μεταξύ τους, αλλά πάλι το πλήθος των σημείων που τους ανατίθενται αναλογεί στη μετρημένη απόδοσή τους. Ωστόσο, και στα δύο επίπεδα η λειτουργία της τεχνικής προαπαιτεί προηγούμενες μετρήσεις όλων των πόρων που συμμετέχουν στον υπολογισμό ώστε να υπολογιστούν οι τιμές της PF. Αυτό συνεπάγεται ότι:

- Στο πρώτο επίπεδο η τεχνική μπορεί να τεθεί σε λειτουργία αφού όλα τα συστήματα που συμμετέχουν στον υπολογισμό έχουν ολοκληρώσει τουλάχιστον ένα κομμάτι υπολογισμού, και
- Στο δεύτερο επίπεδο η τεχνική μπορεί να τεθεί σε λειτουργία σε ένα σύστημα μετά τη πρώτη ανάθεση υπολογισμού στο συγκεκριμένο σύστημα.

Να σημειωθεί ότι στο πρώτο επίπεδο δεν υπάρχει ξεκάθαρο πλήθος στοιχείων προς διαμοιρασμό. Δε μπορεί να χρησιμοποιηθεί το συνολικό πλήθος των σημείων του χώρου αναζήτησης διότι αυτό θα μοίραζε όλα τα στοιχεία με βάση τη τεχνική σε όλα τα συστήματα σε ένα βήμα. Για τον ίδιο λόγο δε μπορεί να χρησιμοποιηθεί ούτε το πλήθος τον υπολειπόμενων στοιχείων. Στο DES Framework χρησιμοποιείται το μέγιστο πλήθος ανάθεσης που έχει δώσει ο χρήστης στον κόμβο master κατά την αρχικοποίηση του προγράμματος (`batchSize`).

3.4.2 Slow-start μέγιστης ανάθεσης

Η τεχνική HPLS προϋποθέτει μετρήσεις απόδοσης για κάθε πόρο στο επίπεδο όπου εφαρμόζεται. Αυτό δημιουργεί ένα πρόβλημα στην αρχή του υπολογισμού: Στις πρώτες αναθέσεις δε μπορεί να χρησιμοποιηθεί η τεχνική σε κανένα από τα 2 επίπεδα. Αναγκαστικά θα πρέπει να χρησιμοποιηθεί κάποιος σταθερός αριθμός, ορισμένος εκ των προτέρων, μέχρι να έχουμε αρκετά δεδομένα ώστε να χρησιμοποιήσουμε την HPLS. Αυτό είναι ένα πρόβλημα που αντιμετωπίζουν όλες οι τεχνικές δυναμικού self-scheduling που δεν έχουν γνώση εκ των προτέρων για τα συστήματα που συμμετέχουν στον υπολογισμό.

Το πρόβλημα παρουσιάζεται στο πρώτο επίπεδο όταν τύχει να αναθέσουμε πολλά σημεία σε έναν πόρο που είναι πολύ αργός σε σχέση με τους υπόλοιπους. Σε αυτή τη περίπτωση μπορεί να ολοκληρωθεί ο υπολογισμός από τους υπόλοιπους και μετά να μένουν όλοι σε αδράνεια μέχρι να τελειώσει και ο πρώτος. Παρουσιάζεται επίσης και στο δεύτερο επίπεδο όταν οι πόροι είναι άνισης επεξεργαστικής ισχύος μεταξύ τους, όπου θα ανατεθεί ίδιο πλήθος στοιχείων σε κάθε πόρο και οι πιο γρήγοροι θα τελειώσουν τον υπολογισμό νωρίτερα και θα μένουν σε αδράνεια μέχρι να τελειώσει και ο πιο αργός.

Αυτό το πρόβλημα δε μπορεί να λυθεί απολύτως, καθώς πάντα μπορεί να υπάρχει ένας πολύ αργός πόρος που θα αναλάβει πολλά στοιχεία στην αρχή ενώ δε θα έπρεπε. Η λύση που προτείνεται και χρησιμοποιείται στο DES Framework ονομάστηκε «Slow-start μέγιστης ανάθεσης» και είναι **ο περιορισμός του μέγιστου πλήθους στοιχείων που μπορούν να**

ανατεθούν σε ένα σύστημα για τις πρώτες αναθέσεις. Αυτή η τεχνική εφαρμόζεται στο πρώτο επίπεδο, και είναι σα να εφαρμόζεται και στο δεύτερο, καθώς η ανάθεση λίγων στοιχείων στο πρώτο επίπεδο συνεπάγεται τις αναθέσεις λίγων στοιχείων και στο δεύτερο. Το μέγιστο πλήθος υπολογίζεται για κάθε σύστημα ξεχωριστά, και αυξάνεται εκθετικά με το πλήθος των αναθέσεων που έχουν ολοκληρωθεί από το σύστημα. Αυτή η τεχνική δίνει μικρές αναθέσεις στην αρχή ώστε να υπολογιστούν γρήγορα κάποιοι ενδεικτικοί χρόνοι εκτέλεσης τόσο στο πρώτο όσο και στο δεύτερο επίπεδο, και έπειτα το όριο αυξάνεται αρκετά ώστε να μη χρησιμοποιηθεί.

Για να αποφευχθεί συνωστισμός στον κεντρικό κόμβο master καθώς και περιττό overhead συχνής επικοινωνίας λόγω μικρών αναθέσεων, πρέπει να αποφευχθούν οι πολλές μικρές αναθέσεις. Ταυτόχρονα όμως δε μπορούμε να ορίσουμε αυθαίρετα κάποιο συγκεκριμένο «αρκετά μεγάλο» πλήθος, διότι ο χρόνος που απαιτείται για κάθε στοιχείο εξαρτάται σε μεγάλο βαθμό από το μοντέλο του προβλήματος, το οποίο μας είναι άγνωστο. Έτσι λοιπόν, το όριο επιλέχθηκε να αυξάνεται εκθετικά χρησιμοποιώντας μία αρχική τιμή που θα ορίζεται από τον χρήστη. Η αρχική τιμή θα πρέπει να είναι ένα πλήθος στοιχείων που αναμένεται να ολοκληρωθεί σε λίγα δευτερόλεπτα από τους πιο αργούς υπολογιστικούς κόμβους του δικτύου. Είναι προφανές ότι στην αρχή του υπολογισμού θα προκληθεί συνωστισμός στον master κόμβο λόγω των μικρών αναθέσεων, αλλά δεδομένου ότι το DES Framework προορίζεται για υπολογισμούς μεγάλης κλίμακας σε μεγάλα δίκτυα υπολογιστών, η καθυστέρηση που αναμένεται να προκληθεί μπορεί να θεωρηθεί αμελητέα σε σχέση με το συνολικό χρόνο εκτέλεσης.

Για τον υπολογισμό του μέγιστου μεγέθους ανάθεσης σε έναν κόμβο i προτείνεται ο τύπος:

$$maxChunkSize_i = base \cdot 2^{jobsCompleted_i}, για\ jobsCompleted_i < limit$$

Εξίσωση 9: Υπολογισμός μέγιστου πλήθους στοιχείων προς ανάθεση με Slow-start για έναν κόμβο i

όπου $base$ και $limit$ οι παράμετροι που δίνονται κατά την αρχικοποίηση του προγράμματος, και $jobsCompleted_i$ το πλήθος των αναθέσεων που έχουν ήδη ολοκληρωθεί από τον κόμβο i. Πρακτικά, το όριο ανάθεσης ξεκινάει με μία τιμή που ορίζει ο χρήστης (βλ. Παράμετροι εκτέλεσης DES Framework) και μετά από κάθε ανάθεση η τιμή αυτή διπλασιάζεται. Αυτό συμβαίνει για όσες αναθέσεις έχει ορίσει ο χρήστης, και έπειτα η τεχνική παύει να επεμβαίνει.

Σε κάθε αίτημα `READY` από κάποιο σύστημα (βλ. Συνάρτηση masterProcess()), υπολογίζεται πρώτα το πλήθος των στοιχείων προς ανάθεση χρησιμοποιώντας την τεχνική HPLS, και στη συνέχεια αυτό περιορίζεται με βάση την εξίσωση 9, εφόσον ισχύει ο περιορισμός.

3.5 Iteration D διαστάσεων

Η βιβλιοθήκη έχει γραφεί χωρίς γνώση της διαστασιμότητας του προβλήματος. Όπως είδαμε νωρίτερα, το πλήθος των διαστάσεων εξαρτάται από το μοντέλο που υλοποιεί η συνάρτηση αξιολόγησης, και είναι άγνωστο κατά τη μετάφραση του προγράμματος. Για την εκτέλεση της εξαντλητικής αναζήτησης πρέπει να εξετάσουμε κάθε σημείο του πολυδιάστατου χώρου. Στα πλαίσια της βιβλιοθήκης αυτός ο χώρος διαιρείται σε μικρότερα τμήματα, αλλά κάθε ένα από αυτά αποτελείται από περισσότερα από ένα στοιχεία, άρα η απαίτηση για τη σάρωση ενός πολυδιάστατου χώρου παραμένει.

Σε απλές υλοποιήσεις όπου το πλήθος των διαστάσεων είναι γνωστό, χρησιμοποιούμε εμφωλευμένους βρόχους `for`. Αν για παράδειγμα θέλουμε να προσπελάσουμε κάθε σημείο X ενός 3-διάστατου χώρου, θα χρησιμοποιούσαμε έναν κώδικα της μορφής:

```
1  for x1 in low[1]:step[1]:high[1]
2      for x2 in low[2]:step[2]:high[2]
3          for x3 in low[3]:step[3]:high[3]
4              // Υπολογισμός για το σημείο [x1, x2, x3]
5          end
6      end
7  end
```

Αλγόριθμος 2: Iteration 3-διάστατου πίνακα με βρόχους for

Αν θέλουμε να προσθέσουμε κι άλλες διαστάσεις θα πρέπει να προσθέσουμε κι άλλους βρόχους `for` μέσα στον εσωτερικό ώστε να έχουμε κι άλλες μεταβλητές x_d. Αυτός είναι ένας αποδοτικός τρόπος για το iteration μίας ή πολλών διαστάσεων, διότι ο μεταφρασμένος κώδικας έχει έτοιμη την απαραίτητη δομή με τις κατάλληλες διακλαδώσεις για να δημιουργήσει όλους τους πιθανούς συνδυασμούς $[x_1, x_2, x_3]$.

Στα πλαίσια του DES Framework το πλήθος των διαστάσεων δίνεται κατά την εκτέλεση του προγράμματος, άρα δε γνωρίζουμε εκ των προτέρων πόσες μεταβλητές και πόσους εμφωλευμένους βρόχους `for` χρειαζόμαστε. Στόχος μας είναι να δημιουργήσουμε κάθε πιθανό συνδυασμό $[x_1, x_2, \dots, x_D]$ ξεκινώντας κάθε φορά από κάποιο σημείο εκκίνησης και προχωρώντας για κάποια βήματα, με το σημείο εκκίνησης και τα βήματα να είναι κάθε φορά διαφορετικά. Όπως έχει ήδη αναφερθεί, στο DES Framework το σημείο εκκίνησης είναι ένας αριθμός, ο οποίος είναι το index ενός σημείου του πολυδιάστατου χώρου αναζήτησης σε μία μονοδιάστατη απεικόνιση. Επίσης έχουμε δει από τον αλγόριθμο 1 πώς μπορούμε να ανακτήσουμε το διάνυσμα $I = [n_1, n_2, \dots, n_D]$ (διάνυσμα δεικτών) από το μονοδιάστατο index. Έχοντας στη διάθεσή μας το διάνυσμα I (όπου κάθε όρος αντιστοιχεί στο index της αντίστοιχης διάστασης) μπορούμε να υπολογίσουμε το διάνυσμα $X = [x_1, x_2, \dots, x_D]$, το οποίο θέλουμε να περάσουμε στη συνάρτηση αξιολόγησης, μέσω της εξίσωσης 1. Από τη στιγμή που έχουμε διαθέσιμο το διάνυσμα X, μπορούμε να το ρυθμίζουμε σε κάθε επανάληψη προσθέτοντας το βήμα κάθε διάστασης στην αντίστοιχη μεταβλητή. Όμως, το διάνυσμα X περιέχει τιμές τύπου floating point, και όπως είναι γνωστό κάθε πράξη με αριθμούς κινητής υποδιαστολής από υπολογιστή μπορεί να επιφέρει κάποιο σφάλμα [29]. Εάν ξεκινήσουμε από την ελάχιστη τιμή μίας διάστασης και σε κάθε επανάληψη προσθέτουμε το βήμα της, τότε το σφάλμα θα συσσωρεύεται και μπορεί τελικά να υπολογίζουμε αποτελέσματα για σημεία που δεν ανήκουν στον διακριτό χώρο αναζήτησης. Θα αποφύγουμε λοιπόν να υπολογίζουμε τα σημεία χρησιμοποιώντας το ίδιο το X, και θα χρησιμοποιούμε το διάνυσμα $I = [n_1, n_2, \dots, n_D]$, ώστε σε κάθε επανάληψη να υπολογίζουμε εκ νέου τη τιμή μίας διάστασης χρησιμοποιώντας την εξίσωση 1.

Στο περιβάλλον ενός υπολογιστικού πόρου (δηλαδή ενός worker thread), η διαδικασία υπολογισμού ξεκινάει με ένα μονοδιάστατο index. Χρησιμοποιώντας τον αλγόριθμο 1 και την εξίσωση 1, μπορούμε να εφαρμόσουμε την εξής λύση:

```
1  Μετατροπή του μονοδιάστατου index σε διάνυσμα indices I (O(D))
2  Μετατροπή του I σε διάνυσμα X (O(D))
3  Υπολογισμός της συνάρτησης αξιολόγησης για το σημείο X
4  Αύξηση του μονοδιάστατου index κατά 1
5  Αν το index δε ξεπέρασε το όριο, μετάβαση στο βήμα 1
6  Τέλος
```

Αλγόριθμος 3: Απλή υλοποίηση D-διάστατου iteration με μονοδιάστατο index

Αυτή είναι μία απλή και λειτουργική λύση που ξεκινάει από το index που δόθηκε ως σημείο εκκίνησης, το μετατρέπει σε διάνυσμα I και μετά σε διάνυσμα X, και εκτελεί τη συνάρτηση αξιολόγησης. Έπειτα αυξάνει το index κατά 1 και επαναλαμβάνει την ίδια διαδικασία, για τόσες φορές όσο είναι το πλήθος των σημείων που έχουν ανατεθεί. Κάθε τέτοια επανάληψη έχει πολυπλοκότητα $O(D)$. Αν κοιτάξουμε αναλυτικά την παραπάνω διαδικασία, θα παρατηρήσουμε ότι πολλές πράξεις επαναλαμβάνονται σε κάθε επανάληψη, ενώ δε χρειάζονται. Στη λύση με τις εμφωλευμένες `for` η διαδικασία που ακολουθείται είναι η εξής:

```
1  Αρχικοποίηση x1
2  Αν το x1 έφτασε στο όριό του τότε μετάβαση στο βήμα 11
3  Αρχικοποίηση x2
4  Αν το x2 έφτασε στο όριό του τότε μετάβαση στο βήμα 10
5  Αρχικοποίηση x3
6  Αν το x3 έφτασε στο όριό του τότε μετάβαση στο βήμα 9
7  // Υπολογισμός με [x1, x2, x3]
8  Αύξηση x3, μετάβαση στο βήμα 6
9  Αύξηση x2, μετάβαση στο βήμα 4
10 Αύξηση x1, μετάβαση στο βήμα 2
11 Τέλος
```

Αλγόριθμος 4: Ανάλυση $3^{ων}$ εμφωλευμένων βρόχων for με code branches

Ο αλγόριθμος 4 υλοποιεί τη λογική εκτέλεσης για 3 εμφωλευμένους βρόχους `for`. Η παραπάνω ανάλυση δείχνει ότι σε κάθε επανάληψη υπολογίζονται μόνο οι μεταβλητές που *πρέπει* να αλλάξουν για τη συγκεκριμένη επανάληψη. Η μεταβλητή `x3` αλλάζει σε κάθε επανάληψη, αλλά η μεταβλητή `x2` αυξάνεται και ελέγχεται μόνο όταν η `x3` φτάσει στο όριό της, και η `x1` αυξάνεται και ελέγχεται μόνο όταν η `x2` φτάσει στο όριό της. Αντιθέτως, ο αλγόριθμος 3 υπολογίζει όλες τις μεταβλητές σε κάθε επανάληψη, ενώ σπάνια χρειάζεται. Αυτό προσφέρει μία πολύ μεγάλη επιβάρυνση στον υπολογισμό και είναι απαγορευτικό.

Στο DES Framework χρησιμοποιήθηκε μία γενίκευση του αλγορίθμου 4 ως προς τις διαστάσεις που κάνει iterate. Ο νέος αλγόριθμος χωρίζεται σε δύο μέρη: την αρχικοποίηση (αλγόριθμος 5) και το βήμα (αλγόριθμος 6). Η αρχικοποίηση γίνεται κατά την έναρξη του υπολογισμού ενός worker thread, και έπειτα επαναλαμβάνονται τα βήματα *"υπολογισμός για X → βήμα"* για τόσες φορές όσα είναι τα στοιχεία που ανατέθηκαν στο συγκεκριμένο worker thread.

```
1  υπόλοιπο = index εκκίνησης
2  for d in 1:D                                    (για κάθε διάσταση...)
3      βήμα_διάστασης = ∏_{1<i≤d} N_i
4      n_d = υπόλοιπο / βήμα_διάστασης             (ακέραια διαίρεση)
5      υπόλοιπο = υπόλοιπο - n_d * βήμα_διάστασης  (αποφυγή πράξης MOD)
6      x_d = low_d + n_d * (high_d – low_d)/N_d (υπολογισμός αρχικού x_d, εξίσωση 1)
7  end
```

Αλγόριθμος 5: Αρχικοποίηση

```
1  for d in 1:D      (για κάθε διάσταση...)
2      n_d += 1        (αύξηση μεταβλητής τρέχουσας διάστασης)
3      if n_d < N_d    (αν δε ξεπεράσαμε το όριο της διάστασης...)
4          x_d += (high_d – low_d)/N_d (πρόσθεση του όρου της εξίσωσης 1 που
   πολλαπλασιάζεται με n_d, διότι το n_d αυξήθηκε κατά 1)
5          Διακοπή βρόχου for    (τέλος αύξησης μεταβλητών)
6      else
7          n_d = 0      (αρχικοποίηση τρέχουσας διάστασης)
8          x_d = low_d  (αρχικοποίηση τρέχουσας διάστασης)
9      end
10 end
```

Αλγόριθμος 6: Βήμα

Κατά την αρχικοποίηση, υπολογίζουμε και αποθηκεύουμε τις τιμές του διανύσματος δεικτών $I = [n_1, n_2, \dots, n_D]$ και του διανύσματος $X = [x_1, x_2, \dots, x_D]$. Ο αλγόριθμος αρχικοποίησης είναι ο αλγόριθμος 1 με μία μικρή αλλαγή στην εντολή που χρησιμοποιεί τη πράξη MOD. Ο λόγος για την αλλαγή είναι επειδή η συγκεκριμένη πράξη προκαλούσε μετρήσιμη καθυστέρηση στον αλγόριθμο κατά την εκτέλεσή του σε κάρτα γραφικών, διότι η πράξη MOD είναι πιο ακριβή λόγω της αρχιτεκτονικής που χρησιμοποιείται. Το πρόβλημα γίνεται ιδιαίτερα εμφανές όταν πολλά GPU threads επιχειρούν να εκτελέσουν τη πράξη ταυτόχρονα, και οι φυσικές μονάδες επεξεργασίας δεν είναι αρκετές. Τότε, κάποια GPU threads αναγκάζονται να περιμένουν τη σειρά τους, καθυστερώντας έτσι όλη την εκτέλεση του υπολογισμού. Η πράξη MOD μπορεί να γίνει με μία ακέραια διαίρεση (η οποία ούτως ή άλλως χρειάζεται στον συγκεκριμένο αλγόριθμο) και έναν πολλαπλασιασμό, καθώς ισχύει ότι:

$$A\ mod\ B\ =\ A\ -\ B\cdot(A\ div\ B)$$

Εξίσωση 10: Υπολογισμός πράξης MOD με πολλαπλασιασμό και ακέραια διαίρεση

όπου mod το υπόλοιπο της ακέραιας διαίρεσης και div το πηλίκο της ακέραιας διαίρεσης. Στην εντολή 5 του αλγορίθμου 5, θέλουμε να αποθηκεύσουμε στη μεταβλητή `υπόλοιπο` το αποτέλεσμα της πράξης `υπόλοιπο MOD βήμα`. Μαθηματικά, αυτό ισούται με

$$υπόλοιπο\ -\ n_d\cdot βήμα$$

όπου n_d το πηλίκο της ακέραιας διαίρεσης $υπόλοιπο\ div\ βήμα$. Αξίζει να σημειωθεί ότι η πράξη mod μπορεί να εκτελεστεί με πολύ απλές πράξεις στο επίπεδο των δυαδικών ψηφίων των αριθμών, όταν ο διαιρέτης της πράξης είναι δύναμη του 2. Συγκεκριμένα, όταν το n είναι δύναμη του 2, τότε

- $i/n\ =\ i\ \gg\ \log_2 n$
- $i\ mod\ n\ =\ i\ \&\ (n\ -\ 1)$

όπου $\gg$ η δεξιά λογική ολίσθηση και & η λογική πράξη AND. Δυστυχώς στα πλαίσια του framework δε μπορεί να ισχύει πάντα κάτι τέτοιο, άρα δε μπορούμε να χρησιμοποιήσουμε αυτές τις διευκολύνσεις.

Μετά τη φάση της αρχικοποίησης, το διάνυσμα X περιέχει το πρώτο σημείο που ανατέθηκε στο συγκεκριμένο worker thread και καλείται η συνάρτηση αξιολόγησης για αυτό. Έπειτα, πρέπει να υπολογίσουμε το επόμενο διάνυσμα X. Αυτό επιτυγχάνεται με το δεύτερο μέρος του αλγορίθμου, τον αλγόριθμο 6. Εκεί, πρέπει να υπολογίσουμε το επόμενο X αποφεύγοντας περιττές πράξεις, όπως ακριβώς θα γινόταν αν είχαμε εμφωλευμένους βρόχους `for`. Ο αλγόριθμος ξεκινάει από τη πρώτη διάσταση, η οποία στα πλαίσια του λογισμικού θεωρείται το LSD (λιγότερο σημαντικό ψηφίο) και αυξάνεται πρώτο (είναι δηλαδή σαν το `x3` του αλγορίθμου 4). Το n_1 αυξάνεται κατά 1, και γίνεται ο έλεγχος για το αν ξεπέρασε το όριο. Σε περίπτωση που δεν το έχει ξεπεράσει ακόμα, ρυθμίζουμε κατάλληλα το x_1 του διανύσματος X και προχωράμε στην εκτέλεση της συνάρτησης αξιολόγησης. Εάν το n_1 έχει φτάσει το N_1, αρχικοποιούμε τη πρώτη διάσταση θέτοντας το $n_1 = 0$ και το $x_1 = low_1$, και προχωράμε στην επόμενη διάσταση, όπου η διαδικασία επαναλαμβάνεται. Αναπόφευκτα, ένας από τους ελέγχους θα αποφασίσει ότι κάποιο όριο δεν έχει ξεπεραστεί και ο αλγόριθμος θα τερματίσει. Η διαδικασία αυτή αυξάνει και ελέγχει μόνο τις μεταβλητές που αλλάζουν και αγνοεί τις

υπόλοιπες, όπως θα γινόταν και με τις εμφωλευμένες `for`. Σε αντίθεση με τον αλγόριθμο 3 που εκτελείται σε χρόνο $O(D)$, ο αλγόριθμος 6 εκτελείται σε amortized χρόνο $O(1)$, διότι κατά μέσο όρο θα τελειώνει σε μία επανάληψη, εκτός από τις περιπτώσεις που η πρώτη διάσταση φτάσει στο όριό της και πρέπει να προωθηθεί κρατούμενο στις επόμενες.

Στις πραγματικές υλοποιήσεις των παραπάνω αλγορίθμων, όροι όπως το βήμα index κάθε διάστασης ($\prod_{1<i\leq d} N_i$) και το πραγματικό βήμα κάθε διάστασης $(high_d - low_d)/N_d$ είναι σταθεροί κατά την εκτέλεση του προγράμματος, άρα υπολογίζονται μία φορά κατά την αρχικοποίηση του προγράμματος και έπειτα χρησιμοποιούνται έτοιμοι.

3.6 Λεπτομέρειες υπολογισμού σε CPU

Κάθε σύστημα που συμμετέχει στον υπολογισμό μπορεί να έχει έναν ή περισσότερους επεξεργαστές, και κάθε ένας από αυτούς μπορεί να έχει έναν ή περισσότερους επεξεργαστικούς πυρήνες. Το DES Framework καλείται να μοιράσει τη δουλειά που ανατίθεται σε ένα σύστημα σε όλους τους διαθέσιμους επεξεργαστικούς πυρήνες, εφόσον ο χρήστης το έχει επιλέξει. Αυτό γίνεται στο δεύτερο επίπεδο του δικτύου που δημιουργείται μέσω OpenMP και συντονίζεται από ένα από τα threads (νήμα συντονιστή). Το νήμα συντονιστή θα αναθέσει ένα μέρος των υπολογισμών σε κάθε worker thread, όπου ένα από αυτά θα αναλάβει τους επεξεργαστές του συστήματος.

Στη πλειοψηφία των περιπτώσεων, σε ένα σύστημα που έχει πολλούς επεξεργαστές, αυτοί είναι ίδιοι μεταξύ τους. Αντίστοιχα, οι επεξεργαστικοί πυρήνες κάθε επεξεργαστή είναι ίδιοι μεταξύ τους, με μερικές εξαιρέσεις όπου κάποιοι πυρήνες είναι χαμηλότερης ισχύος για λόγους κατανάλωσης ενέργειας. Αυτοί χρησιμοποιούνται κυρίως σε κινητές συσκευές με περιορισμένη αυτονομία, οπότε μπορούμε να θεωρήσουμε ότι δε θα χρησιμοποιηθούν σε συνεργασία με το DES Framework. Επίσης, όλοι οι υπόλοιποι παράγοντες που επηρεάζουν την απόδοση των επεξεργαστών είναι ίδιοι για όλους, διότι χρησιμοποιούν τον ίδιο συντονιστή, την ίδια μνήμη, και το ίδιο λειτουργικό σύστημα. Τελικά, μπορούμε να θεωρήσουμε ότι όλοι οι επεξεργαστικοί πυρήνες ενός συστήματος είναι όμοιοι μεταξύ τους, άρα μπορούμε να ισομοιράσουμε τα σημεία προς επεξεργασία σε αυτούς. Το worker thread που αναλαμβάνει τους επεξεργαστές χρησιμοποιεί το OpenMP για να δημιουργήσει τόσα νήματα όσα είναι και οι διαθέσιμοι επεξεργαστικοί πυρήνες. Έπειτα, κάθε τέτοιο νήμα υπολογίζει το σύνολο των

σημείων που του αντιστοιχεί και ξεκινάει τον υπολογισμό. Τα σημεία που ανατέθηκαν στο συγκεκριμένο worker thread ισομοιράζονται στους επεξεργαστικούς πυρήνες με σειριακό τρόπο (ο πρώτος πυρήνας θα πάρει τα πρώτα N/P σημεία, ο δεύτερος θα πάρει τα επόμενα N/P, κ.ο.κ, όπου N το πλήθος των στοιχείων που ανατέθηκαν στο worker thread και P το πλήθος των επεξεργαστικών πυρήνων).

3.7 Λεπτομέρειες υπολογισμού σε GPU

Ο υπολογισμός με χρήση GPU θέτει πολλές προκλήσεις, αλλά και πολλές ευκαιρίες. Η κάρτα γραφικών είναι ένα ανεξάρτητο σύστημα μέσα στον υπολογιστή, με τη δική του μνήμη και τον δικό του επεξεργαστή. Η αρχιτεκτονική μιας κάρτας γραφικών διαφέρει σημαντικά από αυτή ενός επεξεργαστή γενικού σκοπού. Για τη χρήση μίας CUDA GPU για υπολογισμούς γενικής χρήσης ακολουθείται η εξής διαδικασία:

1. Επιλογή GPU
2. Δέσμευση απαιτούμενης μνήμης για δεδομένα και αποτελέσματα στη GPU
3. (Προαιρετικό) Μεταφορά δεδομένων από το host στη GPU
4. Εκτέλεση ενός υπολογιστικού πυρήνα (kernel) στη GPU
5. Αναμονή για τερματισμό kernel
6. Μεταφορά αποτελεσμάτων από τη GPU στο host
7. Αποδέσμευση μνήμης στη GPU

Το **kernel** που εκτελείται σε μία GPU είναι ένα τμήμα κώδικα που εκτελεί κάθε GPU thread. Στους κοινούς επεξεργαστές συχνά δημιουργούνται τόσα νήματα όσα και οι διαθέσιμοι επεξεργαστικοί πυρήνες, και κάθε ένα από αυτά αναλαμβάνει ένα μέρος του υπολογισμού που πρέπει να γίνει. Αντιθέτως, σε μία κάρτα γραφικών δημιουργούνται χιλιάδες νήματα, τα οποία συχνά αντιστοιχίζονται σε ένα στοιχείο προς επεξεργασία, όπως για παράδειγμα τα pixels μίας εικόνας. Τα νήματα οργανώνονται σε blocks, τα οποία αποτελούν ένα ενιαίο grid. Τα blocks και αντίστοιχα τα grids μπορούν να έχουν μονοδιάστατη, 2-διάστατη, ή 3-διάστατη μορφή, και κάθε νήμα ενός block έχει ένα μοναδικό μονοδιάστατο, 2-διάστατο, ή 3-διάστατο ID. Αυτή είναι απλά μία απεικόνιση των νημάτων για ευκολότερη αντιστοίχιση σε στοιχεία προς επεξεργασία, και δεν έχει καμία φυσική διαφορά το αν ένα block θα είναι π.χ. 1x1024, 32x32, ή 32x8x4. Στο DES Framework το πλήθος των διαστάσεων του προβλήματος είναι μεταβλητό,

και μπορεί να είναι πάνω από 3. Αυτό συνεπάγεται ότι δε μπορούμε να εκμεταλλευτούμε την έτοιμη απεικόνιση των GPU threads σε 1, 2, ή 3 διαστάσεις. Ακόμα κι αν το CUDA API πρόσφερε απεικόνιση νημάτων σε περισσότερες διαστάσεις, πάλι δε θα μπορούσε να χρησιμοποιηθεί, λόγω του ότι η διαστασιμότητα είναι άγνωστη κατά τη συγγραφή και μετάφραση του προγράμματος. Για λόγους απλότητας, το λογισμικό χρησιμοποιεί μονοδιάστατα blocks και grids, και έπειτα χρησιμοποιεί τη μονοδιάστατη ταυτότητα του κάθε νήματος για να αναφερθεί σε σημεία του χώρου αναζήτησης, μέσω των μεθόδων που αναλύθηκαν νωρίτερα (εξίσωση 5 και αλγόριθμος 1). Κάθε block μπορεί συνολικά να έχει το πολύ 1024 threads, και το όριο μειώνεται ανάλογα με τους πόρους που απαιτεί κάθε νήμα (π.χ. το πλήθος των registers που χρειάζεται). Κατά την κλήση του kernel, ο host προσδιορίζει πόσα θα είναι τα blocks του grid, και πόσα threads θα έχει κάθε block. Το μέγεθος του block δίνεται από τον χρήστη με την παράμετρο `blockSize` κατά την αρχικοποίηση του προγράμματος. Έπειτα, χρησιμοποιώντας το `blockSize` και το πλήθος των σημείων που ο χρήστης έχει επιλέξει ότι θα ανατίθενται σε κάθε GPU thread, υπολογίζεται το πλήθος των blocks που θα χρειαστούν, και πραγματοποιείται η κλήση του kernel.

Στο δεύτερο master/slave επίπεδο του δικτύου δημιουργείται ένα worker thread για κάθε κάρτα γραφικών. Αυτό αρχικοποιεί την αντίστοιχη GPU δεσμεύοντας την απαραίτητη μνήμη και μεταφέροντας τα απαραίτητα δεδομένα, και περιμένει για ανάθεση εργασίας από τον συντονιστή. Κατά την ανάθεση καλεί το GPU kernel με τις κατάλληλες παραμέτρους και περιμένει να τερματίσει. Όταν το kernel τερματίσει, μεταφέρει τα αποτελέσματα από τη μνήμη της GPU στη μνήμη του host όπου τα περιμένει ο συντονιστής, και περιμένει την επόμενη ανάθεση. Όταν λάβει μήνυμα από τον συντονιστή ότι ο υπολογισμός έχει τελειώσει, το worker thread αποδεσμεύει τη μνήμη στη GPU και τερματίζει. Η κλήση ενός kernel είναι μία χρονοβόρα διαδικασία. Απαιτεί μεταφορές στη μνήμη και επικοινωνία με τη κάρτα γραφικών, άρα για κάθε κλήση έχουμε ένα σημαντικό overhead. Επίσης, σε κάθε εκκίνηση του kernel, κάθε GPU thread πρέπει να κάνει κάποια αρχικοποίηση, η οποία επίσης προσθέτει ένα overhead στον όλο υπολογισμό. Είναι λοιπόν σημαντικό να ολοκληρώσουμε όσο το δυνατόν περισσότερη δουλειά σε κάθε κλήση, ώστε ο χρόνος που σπαταλάται να είναι αμελητέος σε σχέση με τον χρόνο που γίνονται χρήσιμοι υπολογισμοί.

3.7.1 Ανάθεση πολλών σημείων ανά GPU thread

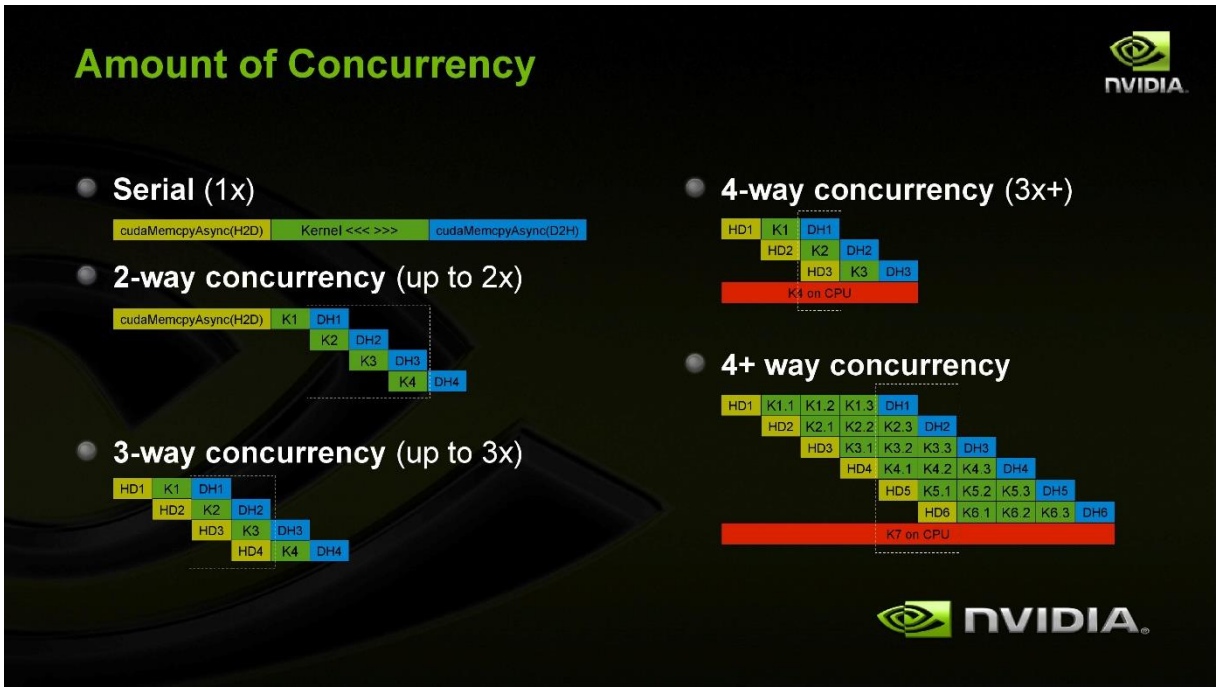
Όπως είδαμε παραπάνω, σε κάθε κλήση του υπολογιστικού πυρήνα στη κάρτα γραφικών κάθε νήμα πρέπει να κάνει κάποια αρχικοποίηση. Αυτή συμπεριλαμβάνει τον υπολογισμό του σημείου εκκίνησης και του πλήθους των σημείων προς επεξεργασία, την εκτέλεση του αλγορίθμου 5, και μεταφορές δεδομένων από τη κύρια μνήμη της κάρτας γραφικών στη κοινή μνήμη των threads (βλ. Shared memory). Για να μειωθεί η επιβάρυνση της αρχικοποίησης σε σχέση με τον χρόνο εκτέλεσης της συνάρτησης αξιολόγησης για ένα σημείο, ανατίθενται πολλά σημεία σε κάθε thread αντί για μόνο ένα, ώστε αυτή η διαδικασία να εκτελείται πιο σπάνια σε σχέση με το πλήθος των σημείων που πρέπει να επεξεργαστούν. Το βέλτιστο πλήθος των σημείων εξαρτάται από τα κατασκευαστικά χαρακτηριστικά της κάρτας γραφικών και το μοντέλο του προβλήματος (τη συνάρτηση αξιολόγησης που εκτελείται στη GPU), άρα δε μπορεί να προσδιοριστεί ακριβώς για όλες τις περιπτώσεις. Για το λόγο αυτό, το πλήθος δίνεται από το χρήστη με τη παράμετρο `computeBatchSize` κατά την αρχικοποίηση του προγράμματος.

Σε κάθε εκτέλεση του kernel, κάθε νήμα που εκτελείται στη κάρτα γραφικών αναλαμβάνει να υπολογίσει τη συνάρτηση αξιολόγησης για `computeBatchSize` συνεχόμενα σημεία. Κάθε νήμα αναλαμβάνει το ίδιο πλήθος σημείων, εκτός ίσως από ένα που μπορεί να αναλάβει λιγότερα, διότι το συνολικό πλήθος των στοιχείων μπορεί να μη διαιρείται ακριβώς με το `computeBatchSize`.

3.7.2 GPU Streams

Οι κάρτες γραφικών της NVIDIA υποστηρίζουν τη χρήση πολλαπλών GPU streams [30], [31]. Ένα GPU stream είναι μία ουρά λειτουργιών (FIFO – First In First Out) που εκτελούνται στη κάρτα γραφικών. Αυτές οι λειτουργίες μπορεί να είναι μεταφορές δεδομένων και εκτελέσεις υπολογιστικών πυρήνων. Κανονικά, χρησιμοποιώντας μόνο ένα stream, όλες αυτές οι λειτουργίες εκτελούνται σειριακά. Κάποιες όμως από αυτές ίσως να μπορούν να παραλληλοποιηθούν, όπως μία μεταφορά με έναν υπολογιστικό πυρήνα, πολλές μεταφορές μεταξύ τους, ή ακόμα και πολλά kernels μεταξύ τους. Μία κάρτα γραφικών CUDA μπορεί να εκτελέσει ταυτόχρονα πολλά streams, εφ' όσων υπάρχουν οι απαραίτητοι πόροι. Υπάρχει λοιπόν η δυνατότητα να διαιρέσουμε τον υπολογισμό σε κομμάτια και να παραλληλοποιήσουμε τη διαδικασία *"μεταφορά δεδομένων → υπολογισμός → μεταφορά*

αποτελεσμάτων" ώστε να αξιοποιήσουμε στο έπακρο τη κάρτα γραφικών και να πετύχουμε μικρότερους χρόνους εκτέλεσης. Αυτό φαίνεται στην εικόνα 7.



Εικόνα 7: Αναπράσταση παραλληλίας με χρήση Streams

Στα πλαίσια του framework, η εκτέλεση ενός υπολογιστικού πυρήνα δεν απαιτεί μεταφορά δεδομένων προς τη GPU, αλλά απαιτεί τη μεταφορά αποτελεσμάτων προς το host. Μπορούμε λοιπόν να χωρίσουμε τους υπολογισμούς που έχουν ανατεθεί σε μία κάρτα γραφικών σε πολλά kernels και να τα στείλουμε σε πολλά streams. Έτσι θα έχουμε πολλά kernels να εκτελούνται ταυτόχρονα, άρα θα έχουμε μεγαλύτερη χρησιμοποίηση της κάρτας γραφικών. Στα ίδια streams μπορούμε να δρομολογήσουμε τις μεταφορές αποτελεσμάτων ώστε και αυτές να παραλληλοποιηθούν με τις εκτελέσεις των υπολογιστικών πυρήνων, άρα να μειωθεί κι άλλο ο χρόνος εκτέλεσης.

Για να χρησιμοποιηθούν πολλά streams, οι υπολογισμοί που έχουν ανατεθεί στη κάρτα γραφικών χωρίζονται σε τόσα κομμάτια όσα είναι τα streams. Σε κάθε stream εισάγεται ένα kernel που θα υπολογίζει ένα από αυτά, και έπειτα μία μεταφορά δεδομένων για να επιστραφούν τα αποτελέσματα στην κύρια μνήμη του συστήματος. Αυτό μπορεί να προσφέρει σημαντική βελτίωση στον χρόνο εκτέλεσης του υπολογισμού, αλλά μπορεί και να δημιουργήσει τόσο μεγάλο overhead που τελικά να μη συμφέρει. Αυτό εξαρτάται από τον υπολογισμό που θα εκτελείται, και κατά πόσο αυτός είναι ικανός να αξιοποιήσει τη συγκεκριμένη κάρτα γραφικών στο έπακρο. Εξαρτάται επίσης από τις τιμές που δίνονται στις

παραμέτρους `blockSize` και `computeBatchSize`, διότι αυτές επηρεάζουν τη χρησιμοποίηση της κάρτας γραφικών.

Το DES Framework υποστηρίζει 2 λειτουργίες σχετικά με την αποθήκευση των αποτελεσμάτων: Αποθήκευση της εξόδου της συνάρτησης αξιολόγησης για όλα τα σημεία του χώρου αναζήτησης, και αποθήκευση μιας λίστας με τα σημεία για τα οποία η έξοδος της συνάρτησης αξιολόγησης θεωρείται αποδεκτή από το μοντέλο. Στη πρώτη περίπτωση η χρήση πολλών streams παραλληλοποιεί τόσο την εκτέλεση των kernels όσο και τις μεταφορές των αποτελεσμάτων. Στη δεύτερη περίπτωση ωστόσο δε επιτρέπεται η παραλληλοποίηση των μεταφορών, διότι τότε υπάρχει μία κοινή περιοχή μνήμης στην οποία όλα τα GPU threads μπορούν να αποθηκεύσουν σημεία, και η προσπέλαση αυτής συντονίζεται με ατομικές εντολές. Ως αποτέλεσμα, ή λίστα που δημιουργεί προσωρινά η κάρτα γραφικών είναι ενιαία για όλα τα νήματα, άρα και για όλες τις κλήσεις του υπολογιστικού πυρήνα. Άρα ο host πρέπει να περιμένει να τελειώσουν όλοι οι υπολογισμοί, ώστε να γνωρίζει το τελικό μέγεθος της λίστας και να κάνει την απαραίτητη μεταφορά.

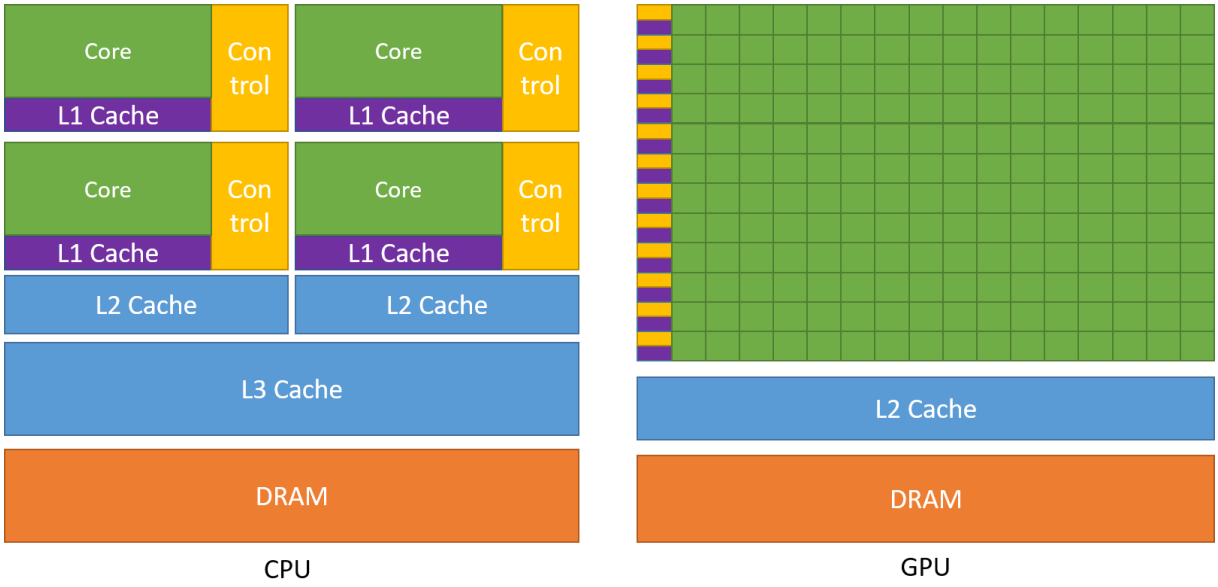
3.7.3 Shared memory

Η αρχιτεκτονική μίας κάρτας γραφικών είναι ιδιαίτερη τόσο στον επεξεργαστή της όσο και στη μνήμη της. Σε ένα απλό σύστημα που χρησιμοποιεί έναν επεξεργαστή γενικής χρήσης υπάρχει μία κύρια μνήμη με την οποία μπορούν να επικοινωνήσουν όλοι οι επεξεργαστικοί πυρήνες, άρα και όλα τα νήματα που εκτελούνται σε αυτούς. Λόγω της σημαντικά χαμηλότερης ταχύτητας της κύριας μνήμης σε σχέση με τους σημερινούς επεξεργαστές, υπάρχουν διαφορετικά επίπεδα **κρυφής μνήμης** (μνήμες cache) «ανάμεσα» στη CPU και τη μνήμη, ώστε να γεφυρώσουν αυτό το χάσμα ταχύτητας. Το κύριο χαρακτηριστικό της κρυφής μνήμης είναι ότι είναι πολύ γρηγορότερη από τη κύρια μνήμη λόγω διαφορετικής αρχιτεκτονικής, αλλά είναι πιο μικρή λόγω του κόστους υλοποίησής της. Αυτός είναι και λόγος που υπάρχουν διαφορετικά επίπεδα κρυφής μνήμης αντί για ένα ενιαίο. Υπάρχει κρυφή μνήμη επιπέδου L1 η οποία είναι η πιο γρήγορη από όλες, αλλά είναι μικρότερης χωρητικότητας, και έπειτα υπάρχουν οι μνήμες L2, L3, κλπ. οι οποίες είναι πιο αργές, αλλά μεγαλύτερης χωρητικότητας. Τα δεδομένα που χρησιμοποιούνται πιο συχνά από τον επεξεργαστή αποθηκεύονται στις κρυφές μνήμες για πιο γρήγορη προσπέλαση. Όσο πιο συχνά αναμένεται να χρησιμοποιηθούν, αποθηκεύονται σε τόσο πιο «κοντινή» μνήμη. Αυτό θέτει προβλήματα συνέπειας της μνήμης, διότι οι αλλαγές

που γίνονται στις κρυφές μνήμες είναι πιθανό να μην αντικατοπτρίζονται στη κύρια. Για παράδειγμα, μπορεί ένα νήμα να αλλάξει μία τιμή στη κρυφή μνήμη, και μετά από λίγο ένα άλλο νήμα να ζητήσει αυτή τη τιμή από τη κύρια μνήμη. Εάν η κύρια μνήμη δεν έχει ενημερωθεί για την αλλαγή, το δεύτερο νήμα θα πάρει λάθος τιμή. Υπάρχουν μηχανισμοί εντός του επεξεργαστή και άλλοι που είναι προγραμματισμένοι στο λειτουργικό σύστημα που αποτρέπουν τέτοιες ασυνέπειες. Αυτοί λειτουργούν χωρίς καμία καθοδήγηση από τον προγραμματιστή που γράφει μία εφαρμογή και μπορούμε πάντα να θεωρούμε ότι υπάρχει συνέπεια της μνήμης.

Η κάρτες γραφικών CUDA χρησιμοποιούν διαφορετική λογική για τις μνήμες τους. Υπάρχει η κύρια μνήμη (γνωστή και ως global memory) η οποία είναι προσβάσιμη ανά πάσα στιγμή από κάθε GPU thread, αλλά η προσπέλαση σε αυτή είναι σημαντικά χρονοβόρα. Όπως οι επεξεργαστές γενικής χρήσης έχουν κρυφές μνήμες, έτσι και οι κάρτες γραφικών έχουν τη **κοινή μνήμη** (shared memory) [32], [33], γνωστή και ως L2 cache. Η κοινή μνήμη στις κάρτες γραφικών έχει σημαντικά μικρότερη χωρητικότητα από τη κύρια μνήμη (συνήθως 48KB), και είναι ξεχωριστή για κάθε thread block που εκτελείται κάθε στιγμή στην κάρτα. Για λόγους υλοποίησης και απόδοσης, δεν υπάρχουν αυτοματοποιημένοι μηχανισμοί που να εγγυούνται τη συνέπεια της κοινής μνήμης με τη κύρια. Αυτό σημαίνει ότι καμία αλλαγή στη κύρια μνήμη δεν αντικατοπτρίζεται στη κοινή, και το αντίθετο. Για τη διαχείριση της κοινής μνήμης ευθύνεται πλήρως ο προγραμματιστής που γράφει κώδικα χρησιμοποιώντας το CUDA API. Κατά την εκκίνηση του kernel, κάθε thread block πρέπει να διαβάσει από τη κύρια μνήμη τα δεδομένα που θέλει να έχει στη κοινή μνήμη και να τα αποθηκεύσει εκεί, και έπειτα να τα χρησιμοποιήσει. Σε περίπτωση που η κοινή μνήμη χρησιμοποιηθεί για εγγραφή αποτελεσμάτων, τα νήματα πρέπει να εκτελέσουν ρητά τη μεταφορά των δεδομένων στη κύρια μνήμη, απ' όπου ο host μπορεί τελικά να τα ανακτήσει.

Στη κοινή μνήμη τοποθετούνται δεδομένα που προσπελαύνονται συχνά από τα GPU threads, ώστε τα νήματα να έχουν πολύ γρήγορη πρόσβαση σε αυτά λόγω της υψηλότερης ταχύτητας της shared μνήμης. Επίσης, με αυτόν τον τρόπο ελαχιστοποιούνται οι συγκρούσεις που προκαλούνται όταν πολλά νήματα προσπαθούν να διαβάσουν από την ίδια θέση μνήμης, καθώς η κοινή μνήμη είναι προσβάσιμη μόνο από ένα thread block και όχι από όλα τα νήματα που εκτελούνται εκείνη τη στιγμή στη κάρτα γραφικών.



Εικόνα 8: Διαφορά αρχιτεκτονικής κρυφής μνήμης μεταξύ CPU και GPU

Στα πλαίσια του framework το kernel η κοινή μνήμη χρησιμοποιείται για να αποθηκευτούν τα στατικά δεδομένα του μοντέλου, τα οποία δίνονται στη συνάρτηση αξιολόγησης μαζί με το διάνυσμα X. Φυσικά, το kernel που εκτελείται και καλεί τη συνάρτηση αξιολόγησης χρειάζεται κι αυτό δεδομένα από τη κύρια μνήμη, τα οποία θα μπορούσαν επίσης να αποθηκευτούν στη κοινή μνήμη. Όμως, ο περισσότερος χρόνος εκτέλεσης αφιερώνεται στην εκτέλεση της συνάρτησης αξιολόγησης και όχι στον υπόλοιπο κώδικα του kernel, και είναι αναμενόμενο ότι η συνάρτηση αξιολόγησης θα χρειάζεται πάντα τα δεδομένα της. Άρα πάρθηκε η απόφαση να χρησιμοποιηθεί η shared μνήμη για τα δεδομένα του μοντέλου, κάτι που όντως προσέφερε βελτίωση στους χρόνους εκτέλεσης.

3.8 Λεπτομέρειες Υλοποίησης κώδικα

Το DES Framework παρέχεται ως μία βιβλιοθήκη της C++. Εκεί υλοποιείται η κλάση `ParallelFramework` που αναπαριστά ένα instance του framework, η κλάση `Model` που είναι η virtual κλάση του μοντέλου που πρέπει να υλοποιήσει ο χρήστης, και οποιαδήποτε λειτουργία και βοηθητική συνάρτηση χρειάζεται κατά την εκτέλεση του υπολογισμού. Η χρήση του DES Framework είναι αρκετά απλή:

1. Δημιουργούμε μια υποκλάση της `Model` και **υλοποιούμε τις virtual συναρτήσεις** `validate_cpu()` και `validate_gpu()`. Αυτές είναι οι συναρτήσεις αξιολόγησης που

θα χρησιμοποιηθούν είτε από επεξεργαστή γενικής χρήσης είτε από κάρτα γραφικών, αντίστοιχα. Σε περίπτωση που θέλουμε τα αποτελέσματα σε μορφή λίστας σημείων, πρέπει να **υλοποιηθεί η συνάρτηση** `toBool()` η οποία αποφασίζει αν ένα σημείο πρέπει να συμπεριληφθεί στη τελική λίστα με βάση την έξοδο της συνάρτησης αξιολόγησης για αυτό. Διαφορετικά η `toBool()` μπορεί να οριστεί ως κενή συνάρτηση.

2. **Δημιουργούμε ένα αντικείμενο** `ParallelFramework`, όπου γίνεται η αρχικοποίηση του MPI. Μετά από τη κλήση του δημιουργού της κλάσης, μπορούμε να χρησιμοποιήσουμε τη **συνάρτηση** `getRank()` του αντικειμένου `ParallelFramework` ώστε να προσδιορίσουμε σε ποιο σύστημα εκτελείται ο κώδικας και να κάνουμε τυχόν ειδικές ρυθμίσεις. Προφανώς, μπορούμε πάντα να χρησιμοποιήσουμε οποιονδήποτε άλλο τρόπο για τον προσδιορισμό των χαρακτηριστικών του συστήματος.
3. **Δημιουργούμε ένα αντικείμενο** `ParallelFrameworkParameters` το οποίο είναι ένα `struct` που περιέχει όλες τις παραμέτρους που απαιτούνται για να εκτελεστεί ο υπολογισμός. Αυτές είναι αυτές που αναλύθηκαν στο κεφάλαιο 3.2.3. Κάποιες από αυτές (όπως οι παράμετροι σχετικά με την εκτέλεση σε GPU) έχουν αρχικές τιμές και ρυθμίζονται από τον χρήστη μόνο αν χρειάζεται.
4. **Δημιουργούμε ένα αντικείμενο** `Limit` **για κάθε διάσταση του προβλήματος**. Αυτό είναι ένα `struct` που περιέχει τις μεταβλητές `lowerLimit`, `upperLimit`, `N`, `step`. Ο χρήστης καλείται να συμπληρώσει τις 3 πρώτες, και το `step` ορίζεται εσωτερικά κατά την αρχικοποίηση.
5. **Καλούμε τη συνάρτηση** `init()` στο αντικείμενο `ParallelFramework` που δημιουργήσαμε νωρίτερα, δίνοντας ως παραμέτρους έναν πίνακα με τα `Limit structs` και το αντικείμενο `ParallelFrameworkParameters`.
6. **Καλούμε τη συνάρτηση** `run()` για το αντικείμενο `ParallelFramework`, δίνοντάς της την υλοποιημένη υποκλάση της `Model`.
7. Όταν η συνάρτηση `run()` επιστρέψει, **καλούμε τη συνάρτηση** `getResults()` **ή** `getList()` για να πάρουμε έναν pointer προς τα αποτελέσματα στη μορφή που ζητήθηκαν .

Ο κώδικας που θα υλοποιεί τη παραπάνω διαδικασία θα πρέπει να εκτελεστεί μέσω MPI σε όλα τα συστήματα που θα συμμετέχουν στον υπολογισμό. Η συνάρτηση `run()` θα ξεκινήσει τον υπολογισμό σε όλα τα συστήματα. Θα επιστρέψει στο κάθε σύστημα σε διαφορετικό χρόνο, ο οποίος για τα slaves είναι όταν ζητήσουν σημεία προς επεξεργασία ενώ

ο master δεν έχει άλλα σημεία να αναθέσει, και για τον master είναι όταν θα έχει λάβει όλα τα αποτελέσματα. Κατά την επιστροφή από τη `run()`, μόνο η master διεργασία έχει πρόσβαση στα αποτελέσματα. Οι slaves μπορούν να χρησιμοποιήσουν τη συνάρτηση `getRank()` για να διαπιστώσουν ότι δεν είναι master, και συνήθως θα τερματιστούν οικειοθελώς.

Κατά την εκτέλεση της συνάρτησης `run()`, το `rank` της διεργασίας χρησιμοποιείται για να προσδιοριστεί αν πρόκειται για master ή για slave, και αντίστοιχα καλούνται οι συναρτήσεις `masterProcess()` ή `slaveProcess()`. Όταν αυτές επιστρέψουν, ο υπολογισμός έχει τελειώσει και ο έλεγχος του προγράμματος επιστρέφει στον αρχικό κώδικα.

3.8.1 Συνάρτηση masterProcess()

Η συνάρτηση `masterProcess()` υλοποιεί το βασικό επαναληπτικό βρόχο του master κόμβου. Επαναληπτικά, περιμένει ένα αίτημα `READY` ή `RESULTS` από κάποιο slave κόμβο και το εξυπηρετεί, μέχρις ότου δεν υπάρχουν άλλα στοιχεία να αναθέσει, έχει λάβει αποτελέσματα για όλα τα σημεία του χώρου αναζήτησης, και έχει ενημερώσει όλους τους slaves ότι ο υπολογισμός έχει τελειώσει και πρέπει να τερματίσουν.

Μετά από το αίτημα `READY`, ο ίδιος slave κόμβος αναμένεται να στείλει το μέγιστο πλήθος σημείων που μπορεί να αναλάβει. Αυτό εξαρτάται από τη διαθέσιμη μνήμη του και μπορεί να παρακαμφθεί από τη παράμετρο `overrideMemoryRestrictions` κατά την αρχικοποίηση του προγράμματος στον master κόμβο. Έπειτα, ο master υπολογίζει ένα σύνολο σημείων χρησιμοποιώντας την τεχνική loop-scheduling που αναλύθηκε στην ενότητα 3.4 και στέλνει το σημείο εκκίνησης και το πλήθος σημείων στον αιτών κόμβο. Ακριβώς μετά την αποστολή αποθηκεύεται η χρονική στιγμή που έγινε η ανάθεση ώστε, όταν ο κόμβος ολοκληρώσει τον υπολογισμό και επιστρέψει τα αποτελέσματα, ο master να μπορεί να υπολογίσει τον συνολικό χρόνο εκτέλεσης και να τον χρησιμοποιήσει στη τεχνική HPLS.

Μετά από αίτημα `RESULTS`, ο ίδιος slave αποστέλλει τα αποτελέσματα του υπολογισμού που του ανατέθηκε, τα οποία ο master αποθηκεύει σε μία προσωρινή τοποθεσία. Ο master έπειτα μεταφέρει τα αποτελέσματα στη τελική περιοχή μνήμης, ενημερώνει τα στατιστικά στοιχεία του κόμβου, και τέλος υπολογίζει τις νέες αναλογίες για τη τεχνική HPLS εν όψει του νέου χρόνου υπολογισμού του slave κόμβου. Η συνάρτηση `masterProcess()` τερματίζει όταν όλα τα σημεία του χώρου αναζήτησης έχουν ανατεθεί, όλα τα αποτελέσματα έχουν ληφθεί, και όλοι οι slave κόμβοι έχουν ενημερωθεί για το τέλος του υπολογισμού και έχουν τερματίσει.

3.8.2 Συνάρτηση slaveProcess()

Η συνάρτηση `slaveProcess()` εκτελείται σε όλα τα slave συστήματα, και χρησιμοποιεί το OpenMP για να δημιουργήσει τα κατάλληλα νήματα για την υλοποίηση του δεύτερου επιπέδου master/slave. Αρχικά δημιουργείται η υποδομή για την επικοινωνία των worker threads με το νήμα συντονιστή. Χρησιμοποιώντας τη παράμετρο `processingType` που δόθηκε από τον χρήστη κατά την αρχικοποίηση της master διεργασίας, δημιουργούνται τα κατάλληλα worker threads και τους ανατίθενται οι πόροι που θα χρησιμοποιήσουν. Επίσης δημιουργείται ένα ακόμα νήμα που θα έχει τον ρόλο του συντονιστή. Τα worker threads υλοποιούνται από τη συνάρτηση `computeThread()`, και το νήμα συντονιστή υλοποιείται από τη συνάρτηση `coordinatorThread()`. Η `slaveProcess()` τερματίζει όταν ενημερωθεί από τον master κόμβο ότι δεν υπάρχουν άλλα στοιχεία προς επεξεργασία, αφού πρώτα τερματίσει όλα τα worker threads..

3.8.3 Συνάρτηση coordinatorThread()

Η συνάρτηση `coordinatorThread()` υλοποιεί τη λογική του νήματος συντονιστή. Η δουλειά της είναι να επικοινωνεί με τη διεργασία master για να της ανατεθούν σημεία του χώρου αναζήτησης, να τα μοιράζει στα worker threads, να συλλέγει τα αποτελέσματα, και τέλος να τα επιστρέφει στο master σύστημα.

Η πρώτη της δουλειά είναι να υπολογίσει το μέγιστο πλήθος σημείων που μπορούν να ανατεθούν στο συγκεκριμένο σύστημα. Αυτό εξαρτάται από τη διαθέσιμη μνήμη του συστήματος και ενδεχομένως των καρτών γραφικών που χρησιμοποιούνται. Κάθε επεξεργαστικός πόρος που χρησιμοποιείται έχει στη διάθεσή του κάποια χωρητικότητα μνήμης η οποία καθορίζει το μέγιστο πλήθος στοιχείων που μπορεί να αναλάβει. Κατά την εκτέλεση του υπολογισμού, σε ένα σύστημα θα ανατεθεί ένα πλήθος σημείων το οποίο έπειτα θα μοιραστεί στους διαθέσιμους πόρους του. Λόγω του ότι η ταχύτητα ενός πόρου δεν εξαρτάται από το μέγεθος της μνήμης του, θέλουμε να έχουμε τη δυνατότητα να αναθέσουμε οποιοδήποτε ποσοστό του υπολογισμού σε οποιοδήποτε πόρο. Έτσι, το μέγιστο πλήθος ορίζεται με βάση τη *μικρότερη* διαθέσιμη μνήμη του κάθε επεξεργαστικού πόρου. Επίσης, το μέγιστο πλήθος περιορίζεται και από τον περιορισμό του MPI που μπορεί να μεταφέρει μόνο $2^{31} - 1$ στοιχεία σε κάθε επικοινωνία.

Μετά από τον υπολογισμό του μέγιστου πλήθους στοιχείων, ξεκινάει ένας βρόχος ο οποίος τερματίζει μόνο όταν ο master αναφέρει ότι δεν υπάρχουν σημεία για ανάθεση. Το πρώτο βήμα το βρόχου είναι το αίτημα `READY` προς τον master, που προκαλεί την ανάθεση ενός συνόλου σημείων του χώρου αναζήτησης στο συγκεκριμένο σύστημα. Έπειτα, το νήμα συντονιστή καλείται να μοιράσει αυτό το σύνολο σημείων στους διαθέσιμους πόρους του συστήματος, δηλαδή στα ενεργά worker threads. Αυτό γίνεται χρησιμοποιώντας τη τεχνική HPLS που αναλύθηκε παραπάνω, εκτός από την πρώτη ανάθεση όπου, λόγω αδυναμίας της HPLS, τα σημεία ισομοιράζονται. Έπειτα τα worker threads ενημερώνονται να ξεκινήσουν τον υπολογισμό μέσω σημαφόρων. Καθώς αυτά δουλεύουν, το νήμα συντονιστή μένει αδρανές. Αυτό δε προκαλεί κάποια μείωση της απόδοσης του συστήματος, καθώς ο επεξεργαστικός πυρήνας που είχε αναλάβει την εκτέλεση του coordinator thread χρησιμοποιείται από κάποιο worker thread για χρήσιμη δουλειά. Όταν όλα τα worker threads ολοκληρώσουν τον υπολογισμό που τους ανατέθηκε, ενημερώνουν το coordinator thread μέσω ενός σημαφόρου. Σε αυτό το σημείο τα worker threads έχουν αποθηκεύσει τα ζητούμενα αποτελέσματα σε μνήμη που τους ανέθεσε το νήμα συντονιστή. Με τα αποτελέσματα έτοιμα, ο συντονιστής τα στέλνει στη master διεργασία, ρυθμίζει τις αναλογίες για τη τεχνική HPLS εν όψει των νέων χρόνων εκτέλεσης των worker threads, και η διαδικασία ξεκινάει από την αρχή.

Όταν η master διεργασία απαντήσει στο αίτημα `READY` με 0 σημεία προς επεξεργασία, ο συντονιστής διακόπτει τον κύριο βρόχο του, ενημερώνει τα worker threads με όμοιο τρόπο ότι πρέπει να τερματίσουν, και η συνάρτηση `coordinatorThread()` επιστρέφει.

3.8.4 Συνάρτηση computeThread()

Για την εκτέλεση του υπολογισμού σε ένα σύστημα, κάθε επεξεργαστικός πόρος αντιστοιχίζεται σε ένα worker thread, άρα υπάρχει ένα νήμα που εκτελεί τη συνάρτηση `computeThread()` για αυτό. Στη περίπτωση πολλών επεξεργαστών και πολλών επεξεργαστικών πυρήνων, αυτοί θεωρούνται όμοιοι και αντιστοιχίζονται σε ένα μόνο worker thread. Η συνάρτηση `slaveProcess()` δημιουργεί τα κατάλληλα worker threads και καλεί τη συνάρτηση `computeThread()` για κάθε ένα από αυτά. Αυτά εκτελούνται παράλληλα μεταξύ τους και επικοινωνούν μόνο με το νήμα συντονιστή.

Κάθε worker thread που αντιστοιχεί σε κάρτα γραφικών πρέπει να την αρχικοποιήσει. Αρχικά επιλέγει τη κάρτα που του αντιστοιχεί, δεσμεύει τη μνήμη που απαιτείται, αποφασίζει

αν θα χρησιμοποιείται κοινή μνήμη ανάλογα με το μέγεθός της και το μέγεθος των δεδομένων που έδωσε ο χρήστης για το μοντέλο, και αντιγράφει τα απαραίτητα δεδομένα στη μνήμη της GPU. Επίσης δημιουργεί στη μνήμη της GPU ένα instance της υλοποιημένης κλάσης του μοντέλου που παρέχει ο χρήστης, ώστε να μπορεί να κληθεί η συνάρτηση αξιολόγησης από το kernel που θα εκτελεστεί στον επεξεργαστή της κάρτας γραφικών.

Όταν το worker thread είναι έτοιμο περιμένει να του ανατεθούν σημεία για επεξεργασία από τον συντονιστή. Αυτός ο συγχρονισμός γίνεται μέσω ενός σημαφόρου στον οποίο έχει πρόσβαση το worker thread και ο συντονιστής. Όταν υπάρχουν έτοιμα δεδομένα προς υπολογισμό, αυτά δίνονται στο worker thread και αυτό χρησιμοποιεί τον πόρο που του έχει ανατεθεί για να υπολογίσει τα ζητούμενα αποτελέσματα.

- Εάν πρόκειται για επεξεργαστή γενικής χρήσης, δημιουργούνται τόσα νήματα όσα και οι διαθέσιμοι επεξεργαστικοί πυρήνες, τα σημεία που ανατέθηκαν στο worker thread ισομοιράζονται σε αυτούς, και ο υπολογισμός ξεκινά. Όταν ολοκληρωθεί, ενημερώνεται ο συντονιστής, και η διαδικασία ξεκινάει από την αρχή.
- Εάν πρόκειται για κάρτα γραφικών, τα σημεία μοιράζονται στα streams που έχει ορίσει ο χρήστης και γίνονται οι κλήσεις των υπολογιστικών πυρήνων. Ανάλογα με τον τρόπο αποθήκευσης που έχει επιλεχθεί, οι μεταφορές των αποτελεσμάτων είτε δρομολογούνται σε κάθε stream μετά την εκτέλεση του αντίστοιχου kernel, είτε γίνεται μία ενιαία όταν τερματίσουν όλα τα streams. Όταν γίνει η μεταφορά των αποτελεσμάτων στη μνήμη του host, ενημερώνεται ο συντονιστής και η διαδικασία ξεκινάει από την αρχή.

Η συνάρτηση `computeThread()` γνωρίζει ότι πρέπει να τερματίσει όταν της ανατεθούν `-1` στοιχεία για επεξεργασία. Τότε ελευθερώνει ότι πόρους έχουν δεσμευτεί και επιστρέφει.

4

Πειραματική Αξιολόγηση

4.1 Προετοιμασία

Το πλαίσιο που σχεδιάστηκε υλοποιήθηκε και αξιοποιήθηκε σε συγκεκριμένη εφαρμογή που προέρχεται από την γεωφυσική και ειδικότερα συγκρίθηκε με πρόσφατες μεθόδους τοπολογικής αντιστροφής για την επίλυση προβλημάτων ελαστικής μετατόπισης σεισμικών ρηγμάτων και μοντελοποίησης μαγματικής πηγής Mogi και Okada. Αυτές οι μέθοδοι υλοποιήθηκαν πρόσφατα σε κάρτες γραφικών [34] και η αποτελεσματικότητά τους εξαρτάται άμεσα από την επίδοση της εξαντλητικής αναζήτησης στο σύστημα. Για την αξιολόγηση του DES Framework χρησιμοποιήθηκαν αυτές οι υλοποιήσεις σε GPU των 4 μοντέλων Mogi 1, Mogi 2, Okada 1, και Okada 2, τα οποία χρησιμοποιούν 4, 8, 10, και 20 παραμέτρους αντίστοιχα. Οι συναρτήσεις που χρησιμοποιούνται για να προσδιοριστεί αν ένα σύνολο παραμέτρων είναι αποδεκτό είναι μη γραμμικές και εξαιρετικά ευαίσθητες σε αλλαγές, άρα η εξαντλητική αναζήτηση είναι η μόνη λύση. Από τις έτοιμες υλοποιήσεις χρησιμοποιήθηκαν τα kernels που εκτελούνταν σε CUDA GPU, τα οποία ήταν προσαρμοσμένα στο κάθε μοντέλο, άρα χρησιμοποιούσαν εμφωλευμένους βρόχους `for` για να εξετάσουν κάθε πιθανό συνδυασμό των μεταβλητών του προβλήματος. Ο κώδικας του εσωτερικού βρόχου `for` για κάθε μοντέλο δόθηκε ως συνάρτηση αξιολόγησης στο DES Framework. Δημιουργήθηκαν τα αντίστοιχα 4 μοντέλα και κάθε ένα χρησιμοποιήθηκε για αναζήτηση σε αρκετούς διαφορετικούς χώρους αναζήτησης. Για κάθε τεστ ενός μοντέλου, οι χώροι αναζήτησης είχαν το ίδιο πλήθος διαστάσεων (αναπόφευκτο), τα ίδια όρια, αλλά διαφορετική ανάλυση κάθε ενός από αυτά, δίνοντας έτσι διαφορετικό μέγεθος του χώρου αναζήτησης.

Οι έτοιμες υλοποιήσεις ήταν φτιαγμένες ώστε να χρησιμοποιούν μία κάρτα γραφικών, και το framework δοκιμάστηκε τόσο με μία κάρτα γραφικών, όσο και με τη GPU σε συνεργασία με τη CPU ενός συστήματος. Τα χαρακτηριστικά του συστήματος όπου εκτελέστηκαν τα πειράματα φαίνονται παρακάτω:

- **CPU**: Intel i5-7400, 64bit, 4C/4T @ 3.5 Ghz, 46.87 Gflop/s [35]
- **GPU**: Nvidia GeForce GTX 1050Ti, 768 CUDA cores, 4GB GDDR5 112 GB/s
- **RAM**: 16GB DDR4 2400Mhz, 19.2 GB/s
- **OS, kernel:** Ubuntu 20.04.1 LTS, 5.4.0-42-generic kernel
- **gcc:** 7.5.0
- **nvcc:** CUDA compilation tools, v10.2.89
- **CUDA:** CUDA 10.2, Driver 440.100
- **OpenMPI:** 4.0.3

Οι εκτελέσεις των αναζητήσεων έγιναν με τις εξής παραμέτρους:

- `batchSize`: `ULONG_MAX` (το μέγιστο δυνατό)
- `slaveBalancing`: αδιάφορο (αφού υπάρχει μόνο ένα σύστημα)
- `threadBalancing`: `true`, μόνο για τη περίπτωση που χρησιμοποιήθηκε και η CPU
- `GPU Block size`: 256 (μέγιστο επιτρεπόμενο λόγω υψηλής χρήσης registers)
- `GPU Computing batch size`: 200 (επιλέχθηκε πειραματικά για τα συγκεκριμένα μοντέλα)
- `GPU Streams`: 8 (επιλέχθηκε πειραματικά για τα συγκεκριμένα μοντέλα)
- `overrideMemoryRestrictions`: `true` (γνωρίζουμε ότι τα αποτελέσματα είναι πολύ λίγα σε σχέση με τον χώρο αναζήτησης άρα δεχόμαστε να εκτελέσουμε τον υπολογισμό με ενδεχομένως ανεπαρκή μνήμη – κατά τα πειράματα η μνήμη που τελικά χρειαζόταν για τα αποτελέσματα ήταν ελάχιστη, και οι έτοιμες υλοποιήσεις χρησιμοποιούσαν την ίδια παραδοχή)
- `slowStartBase`, `slowStartLimit`: 5000000, 5 (προεπιλεγμένες τιμές, επιλέχθηκαν πειραματικά)

4.2 Αποτελέσματα

Τα αποτελέσματα παρουσιάζονται στον παρακάτω πίνακα:

Πίνακας 1: Αποτελέσματα πειραματικής αξιολόγησης

Μοντέλο	**Σημεία χώρου αναζήτησης**	**Αιτήματα slave σε master**	**Χρόνος DES (CPU & GPU)**	**Χρόνος DES (GPU)**	**Χρόνος TOPINV (Baseline)**
MOGI 1 (4 διαστάσεις)	120960	1	0.1s	0.1s	0.1s
	1621620	1	0.1s	0.1s	0.1s
	9979200	2	0.2s	0.1s	0.2s
	155297880	6	0.4s	0.4s	0.4s
	2450288880	6	4.5s	4.8s	4.3s
	24374428572	6	39.0s	44.0s	42.0s
MOGI 2 (8 διαστάσεις)	4435968	1	0.3s	0.2s	0.1s
	114412452	5	1.0s	1.0s	0.9s
	343874160	6	2.5s	2.7s	2.6s
	1255565220	6	7.8s	8.3s	8.7s
	4330418820	6	26.6s	28.5s	28.3s
	27605453400	6	249.0s	268.0s	207.0s
OKADA 1 (10 διαστάσεις)	7257600	2	0.6s	0.4s	0.2s
	33264000	3	1.4s	1.3s	0.8s
	107775360	5	3.6s	3.6s	2.3s
	810810000	6	19.0s	19.4s	15.0s
	2318916600	6	100.0s	108.0s	47.0s
	7894341000	6	340.0s	377.0s	171.0s
OKADA 2 (20 διαστάσεις)	68024448	4	1.8s	1.7s	1.0s
	283435200	6	8.6s	6.6s	4.2s
	2902376448	6	340.0s	133.0s	87.0s

4.3 Παρατηρήσεις

Στον παραπάνω πίνακα φαίνεται αμέσως ότι, στη καλύτερη περίπτωση, το DES Framework αποδίδει το ίδιο με τους χρόνους των έτοιμων υλοποιήσεων, αλλά γενικότερα παρουσιάζει χαμηλότερη αποδοτικότητα. Με εξαίρεση το μοντέλο Mogi 1 και κάποιες περιπτώσεις του Mogi 2, το DES παρουσιάζει καθυστερήσεις που φαίνονται να αυξάνονται με το πλήθος των σημείων του χώρου αναζήτησης, καθώς και με το πλήθος των διαστάσεων. Μία γενική αύξηση του χρόνου εκτέλεσης είναι αναμενόμενη λόγω της έξτρα λειτουργικότητας που υλοποιεί το DES. Η αρχικοποίηση του MPI Communicator, η εφαρμογή τεχνικής loop-scheduling, η ευελιξία στο πλήθος των συστημάτων που μπορούν να συμμετέχουν και στο πλήθος των διαστάσεων, είναι λειτουργίες που απαιτούν υπολογιστικό χρόνο.

Στα πειράματα με μικρούς χώρους αναζήτησης, μεταξύ των εκτελέσεων μόνο με GPU και με συμμετοχή της CPU φαίνεται καθαρά η καθυστέρηση του διαμοιρασμού του φόρτου εργασίας και η ανάγκη για την τεχνική Slow-start μέγιστης ανάθεσης. Ο χρόνος εκτέλεσης μόνο με τη GPU είναι μικρότερος όταν το κόστος του χρήσιμου υπολογισμού είναι συγκρίσιμο με το κόστος της ανάθεσης σε πολλούς πόρους και τη καθυστέρηση της "λάθους" πρώτης ανάθεσης, η οποία δίνει στον επεξεργαστή περισσότερα στοιχεία απ' ότι πρέπει. Αυτό είναι ένα «απαραίτητο κακό» για την υποστήριξη πολλών συστημάτων και πολλών επεξεργαστικών πόρων ανά σύστημα, δηλαδή το όλο concept του DES Framework. Ο διαμοιρασμός του φόρτου εργασίας γίνεται και στα 2 επίπεδα σε σταθερό χρόνο ως προς από το μέγεθος του χώρου αναζήτησης και τη διαστασιμότητα του προβλήματος, αλλά εξαρτάται γραμμικά από το πλήθος των slaves στα 2 επίπεδα λόγω της ανάγκης υπολογισμού των scores για τη τεχνική HPLS.

Από τις παραπάνω μετρήσεις φαίνεται επίσης η μείωση της απόδοσης του DES καθώς το πλήθος των διαστάσεων αυξάνεται. Γνωρίζοντας ότι κατά την εκτέλεση του μεγαλύτερου μέρους υπολογισμού η κύρια διαφορά με τον κώδικα των έτοιμων υλοποιήσεων είναι ο τρόπος που υλοποιείται το iteration στα σημεία προς επεξεργασία, μπορούμε να συμπεράνουμε ότι οι αλγόριθμοι 5 και 6 είναι αυτοί που δίνουν τη καθυστέρηση που βλέπουμε στις μετρήσεις. Προφανώς οι αλγόριθμοι 5 και 6 προσφέρουν μία καλύτερη προσέγγιση από την απλή λύση του αλγορίθμου 3, αλλά φαίνεται να μην είναι αρκετή.

Το πλήθος των αιτημάτων από slave προς master που φαίνεται στον πίνακα δείχνει πόσες φορές έγινε ανάθεση σημείων προς τον μοναδικό slave (1ου επιπέδου) που συμμετείχε

στον υπολογισμό. Εκτελέσεις που έχουν το ίδιο πλήθος αναθέσεων, έχουν και την ίδια καθυστέρηση λόγω αναθέσεων ή επικοινωνίας μέσω MPI. Λόγω της τεχνικής Slow-start μέγιστης ανάθεσης στα πρώτα αιτήματα γινόταν μικρές αναθέσεις. Έπειτα, λόγω του ότι υπήρχε μόνο ένα σύστημα που συμμετείχε στον υπολογισμό, χρησιμοποιούνταν η παράμετρος `batchSize` που δόθηκε κατά την αρχικοποίηση, η οποία ήταν η μέγιστη δυνατή (`ULONG_MAX`). Αφού η παράμετρος `slowStartLimit` ήταν 5, τότε στην 6η ανάθεση χρησιμοποιούνταν πάντα το `batchSize` ως μέγεθος ανάθεσης, το οποίο πάντα έδινε όλα τα διαθέσιμα σημεία για επεξεργασία. Από αυτό μπορούμε να συμπεράνουμε ότι στις εκτελέσεις που έχουν πάνω από 5 αναθέσεις, οποιαδήποτε διαφορά στον χρόνο εκτέλεσης αφορά αποκλειστικά το μέγεθος του χώρου αναζήτησης και το πλήθος των διαστάσεων.

4.4 Συμπέρασμα

Από τις παραπάνω παρατηρήσεις είναι προφανές ότι το DES Framework δε προσφέρει κάποια βελτίωση απόδοσης σε σχέση με μία μεμονωμένη υλοποίηση ενός μοντέλου. Δηλαδή για κάθε μοντέλο που μπορεί να δοθεί στο DES προς αναζήτηση, μπορεί να γραφεί κώδικας που να το υλοποιεί καλύτερα στο επίπεδο του ενός συστήματος. Αυτό όμως αντισταθμίζεται από την ευελιξία που παρέχεται από τη βιβλιοθήκη. Το DES Framework προσφέρει έτοιμο το μεγαλύτερο μέρος του κώδικα αναζήτησης με CPU ή/και GPU, αφαιρώντας τις λεπτομέρειες χαμηλού επιπέδου για τη χρήση GPU και για τον διαμοιρασμό δουλειάς σε πολυπύρηνα ετερογενή συστήματα. Κυριότερα, το DES Framework προσφέρει άμεσα και με ελάχιστη δουλειά από τον προγραμματιστή τη δυνατότητα χρήσης πολλών συστημάτων για την εκτέλεση της αναζήτησης, εφαρμόζοντας τεχνικές διαμοιρασμού του φόρτου εργασίας σε αυτά ώστε να αξιοποιούνται όλα στο έπακρο. Επίσης, μέσω του συγκεκριμένου πλαισίου λογισμικού ορίζεται ένα πρότυπο που μπορεί να εξελιχθεί περαιτέρω, προσφέροντας περισσότερες λειτουργίες και βελτιώνοντας στις υπάρχουσες και τη γενικότερη απόδοση του συστήματος.

5

Μελλοντική Δουλειά

5.1 *Πρόβλημα λόγω μεταβλητής διαστασιμότητας*

Στα πλαίσια του DES Framework το πλήθος των διαστάσεων του μοντέλου είναι παραμετροποιήσιμο κατά την εκτέλεση του προγράμματος. Αυτό συνεπάγεται ότι κατά τη μετάφραση δε μπορούν να χρησιμοποιηθούν εμφωλευμένοι βρόχοι `for` ή διάφορα optimizations του compiler όπως loop unrolling που, υπό άλλες συνθήκες, θα μπορούσαν να εφαρμοστούν. Αυτό δημιουργεί ένα πρόβλημα απόδοσης, που η ενότητα 3.5 προσπαθεί να λύσει. Έγινε μία προσπάθεια γενίκευσης του αλγορίθμου 4 με τους αλγορίθμους 5 και 6. Η πολυπλοκότητα των αλγορίθμων 5 και 6 είναι ίδιας τάξης με αυτή του 4, αλλά δεν έχουν τους ίδιους συντελεστές. Αυτό προκαλεί μία αισθητή καθυστέρηση στην εκτέλεση του υπολογισμού, και σε μελλοντική δουλειά θα ήταν ένα καλό σημείο εκκίνησης για τη βελτίωση της απόδοσης του λογισμικού.

5.2 *Βελτίωση Self-Scheduling*

Στο DES Framework χρησιμοποιείται δυναμικό loop self-scheduling για το διαμοιρασμό του φόρτου εργασίας στα συστήματα του δικτύου και στους πόρους του κάθε συστήματος. Χρησιμοποιήθηκε μία παραλλαγή της τεχνικής HPLS με μία συνάρτηση αναλογίας που βασίστηκε στο score κάθε κόμβου, το οποίο ορίστηκε ως ο λόγος του πλήθους των σημείων της τελευταίας ανάθεσης προς το χρόνο εκτέλεσης του αντίστοιχου υπολογισμού. Αυτή λειτούργησε καλύτερα από άλλες τεχνικές που αναφέρθηκαν, και φυσικά ήταν σαφώς

καλύτερη από τη στατική ανάθεση. Παρόλα αυτά, η τεχνική παρουσιάζει μία αδυναμία κατά την εκκίνηση του υπολογισμού, καθώς δεν υπάρχουν οι απαραίτητες μετρήσεις για τον υπολογισμό της συνάρτησης αναλογίας. Για τη καταπολέμηση του προβλήματος χρησιμοποιήθηκε η τεχνική Slow-start μέγιστης ανάθεσης η οποία περιορίζει το μέγιστο μέγεθος ανάθεσης στην αρχή του υπολογισμού, προκειμένου να γίνουν οι απαραίτητες μετρήσεις πριν αρχίσουν να δίνονται μεγάλα chunks. Αυτό έδειξε αρκετή βελτίωση, κυρίως σε μικρούς χώρους αναζήτησης. Καθώς ο χώρος αναζήτησης μεγαλώνει, το πρόβλημα δεν έχει τόση μεγάλη επίδραση στο συνολικό χρόνο εκτέλεσης. Να σημειωθεί ότι αν το `batchSize` που έχει δώσει ο χρήστης είναι λιγότερο από N/P, όπου N το μέγεθος του χώρου αναζήτησης και P το πλήθος των συστημάτων που συμμετέχουν στον υπολογισμό, τότε θα παρουσιαστεί το πρόβλημα που παρουσιάζουν και οι τεχνικές Chunk Self-Scheduling και Guided Self-Scheduling, δηλαδή θα είναι δυνατή η ανάθεση όλων (ή του μεγαλύτερου μέρους) των στοιχείων σε ένα slave σύστημα και τα υπόλοιπα θα μείνουν αδρανή. Η βελτίωση της παραπάνω τεχνικής είναι ένα ακόμα σημείο όπου η συνολική αποδοτικότητα της βιβλιοθήκης θα μπορούσε να αυξηθεί μελλοντικά. Τέτοιες βελτιώσεις θα μπορούσαν να είναι η χρήση μίας εντελώς άλλης τεχνικής, η χρήση προηγούμενης γνώσης για το μοντέλο και τη συνάρτηση αξιολόγησης, η χρήση προηγούμενης γνώσης για τα συστήματα που συμμετέχουν στον υπολογισμό, και η υπόθεση εκ των προτέρων ότι οι κάρτες γραφικών είναι πιο γρήγορες από τους επεξεργαστές γενικού σκοπού ώστε η εκάστοτε τεχνική να αρχικοποιηθεί καλύτερα. Γενικά όμως, η ιδέα της χρήσης πραγματικών μετρήσεων σε πραγματικό χρόνο φάνηκε να δουλεύει εξαιρετικά, και καλό θα ήταν να παραμείνει.

5.3 Ανοχή σφαλμάτων και σύσταση νέων κόμβων on-the-fly

Ένας τομέας που η παρούσα υλοποίηση σίγουρα υστερεί είναι η ανοχή αποτυχιών κόμβων του δικτύου. Σε περίπτωση που ένας από τους κόμβους του συστήματος αποτύχει (αποσυνδεθεί, κλείσει, αντιμετωπίσει κάποιο σφάλμα και τερματιστεί η εκτέλεση του προγράμματος), τα αποτελέσματα θα είναι ελλειπή. Συγκεκριμένα, η master διεργασία θα περιμένει απ’ αόριστων τα αποτελέσματα από τον αποτυχημένο κόμβο, και δε θα τερματίσει ποτέ. Σε μελλοντική δουλειά, η υλοποίηση μπορεί να προσαρμοστεί ώστε να αντιμετωπίζει τέτοια σφάλματα, και να αναθέτει τους χαμένους υπολογισμούς σε άλλους κόμβους.

Η δομή του δικτύου είναι master/slave, η οποία εξ' ορισμού θέτει ένα single point of failure. Εάν συμβεί κάτι στον master κόμβο τότε όλος ο υπολογισμός καταρρέει. Μία δομή κατανεμημένου υπολογισμού, όπου όλοι οι κόμβοι είναι ίσοι μεταξύ τους, μπορεί να δώσει τη δυνατότητα σε οποιονδήποτε κόμβο να αποτύχει ανά πάσα στιγμή, άρα θα έλυνε το πρόβλημα. Τέτοιες υλοποιήσεις είναι φυσικά πιο περίπλοκες και θέτουν περισσότερες προκλήσεις, όπως μεγάλα overhead στον συγχρονισμό των κόμβων, πιο χρονοβόρα επικοινωνία, σπατάλη πόρων λόγω πλεονασμού δεδομένων και αποτελεσμάτων, κ.α., αλλά ίσως να είναι αποδεκτές εάν η ανοχή σφαλμάτων είναι σημαντική για κάποια εφαρμογή.

Επίσης χρήσιμη θα ήταν η δυνατότητα σύστασης νέων συστημάτων στο δίκτυο υπολογισμού μετά την έναρξή του. Αυτό σε συνδυασμό με την ανοχή σφαλμάτων θα μπορούσε να βοηθήσει με τη χρήση του λογισμικού σε μεγάλα δίκτυα (π.χ. διαδίκτυο) όπου οι κόμβοι μπορούν ανά πάσα στιγμή να πάψουν να είναι προσβάσιμοι και να επανασυνδεθούν αργότερα.

Βιβλιογραφία - Αναφορές

[1] A. Turing, "On Computable Numbers, with an Application to the Entscheidungsproblem," *Proc. Lond. Math. Soc.*, vol. 42, no. 1, pp. 230–265, 1936.

[2] C.-T. Yang, W.-C. Shih, and S.-S. Tseng, "Dynamic partitioning of loop iterations on heterogeneous PC clusters," *J. Supercomput.*, vol. 44, no. 1, pp. 1–23, Apr. 2008, doi: 10.1007/s11227-007-0146-0.

[3] "Multi-core processor," *Wikipedia*. May 26, 2020, Accessed: Jul. 13, 2020. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Multi-core_processor&oldid=958877109.

[4] "Intel® Core™ i5-10400 Processor," *Intel*. https://www.intel.com/content/www/us/en/products/processors/core/i5-processors/i5-10400.html (accessed Sep. 05, 2020).

[5] "AMD Ryzen™ 7 3800X." https://www.amd.com/en/products/cpu/amd-ryzen-7-3800x (accessed Sep. 05, 2020).

[6] "Qualcomm Snapdragon 821 Mobile Platform," *Qualcomm*, Oct. 02, 2018. https://www.qualcomm.com/products/snapdragon-821-mobile-platform (accessed Sep. 05, 2020).

[7] "Intel® Xeon Phi™ Processor 7230F (16GB, 1.30 GHz, 64 core) Product Specifications." https://ark.intel.com/content/www/us/en/ark/products/95828/intel-xeon-phi-processor-7230f-16gb-1-30-ghz-64-core.html (accessed Sep. 05, 2020).

[8] "AMD Ryzen™ Threadripper™ 3990X Processor." https://www.amd.com/en/products/cpu/amd-ryzen-threadripper-3990x (accessed Sep. 05, 2020).

[9] "General-purpose computing on graphics processing units," *Wikipedia*. Jul. 04, 2020, Accessed: Jul. 13, 2020. [Online]. Available: https://en.wikipedia.org/w/index.php?title=General-purpose_computing_on_graphics_processing_units&oldid=966044535.

[10] "OpenCL - The Open Standard for Parallel Programming of Heterogeneous Systems," *The Khronos Group*, Jul. 21, 2013. https://www.khronos.org// (accessed Jul. 14, 2020).

[11] "CUDA Zone," *NVIDIA Developer*, Jul. 18, 2017. https://developer.nvidia.com/cuda-zone (accessed Jul. 14, 2020).

[12] "Mixed-Precision Programming with CUDA 8," *NVIDIA Developer Blog*, Oct. 19, 2016. https://developer.nvidia.com/blog/mixed-precision-programming-cuda-8/ (accessed Sep. 08, 2020).

[13] "754-2019 - IEEE Standard for Floating-Point Arithmetic." https://standards.ieee.org/content/ieee-standards/en/standard/754-2019.html (accessed Sep. 08, 2020).

[14] "Performance Benefits of Half Precision Floats," *Intel*. https://www.intel.com/content/www/us/en/develop/articles/performance-benefits-of-half-precision-floats.html (accessed Sep. 08, 2020).

[15] "Half-precision floating-point format," *Wikipedia*. Sep. 06, 2020, Accessed: Sep. 08, 2020. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Half-precision_floating-point_format&oldid=977105152.

[16] "Training with Mixed Precision," p. 46.

[17] N. Ho and W. Wong, "Exploiting half precision arithmetic in Nvidia GPUs," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, 2017, pp. 1–7.

[18] "GPUDirect," *NVIDIA Developer*, Oct. 06, 2015. https://developer.nvidia.com/gpudirect (accessed Jul. 14, 2020).

[19] "NVIDIA SLI." https://www.geforce.co.uk/hardware/technology/sli (accessed Jul. 14, 2020).

[20] "NVLink High-Speed GPU Interconnect | NVIDIA Quadro." https://www.nvidia.com/en-us/design-visualization/nvlink-bridges/ (accessed Jul. 14, 2020).

[21] C.-C. Wu, L.-F. Lai, C.-T. Yang, and P.-H. Chiu, "Using hybrid MPI and OpenMP programming to optimize communications in parallel loop self-scheduling schemes for multicore PC clusters," *J. Supercomput.*, vol. 60, no. 1, pp. 31–61, Apr. 2012, doi: 10.1007/s11227-009-0271-z.

[22] "OpenMP." https://www.openmp.org/ (accessed Jul. 13, 2020).

[23] "Open MPI: Open Source High Performance Computing." https://www.open-mpi.org/ (accessed Jul. 13, 2020).

[24] D. W. Walker, "Standards for message-passing in a distributed memory environment," Oak Ridge National Lab., TN (United States), ORNL/TM-12147; CONF-9204185-Summ., Aug. 1992. Accessed: Jul. 13, 2020. [Online]. Available: https://www.osti.gov/biblio/10170156.

[25] C. D. Polychronopoulos and D. J. Kuck, "Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers," *IEEE Trans. Comput.*, vol. C–36, no. 12, pp. 1425–1439, Dec. 1987, doi: 10.1109/TC.1987.5009495.

[26] T. Rauber and G. Rünger, *Parallel programming: for multicore and cluster systems*, Second edition. Berlin Heidelberg: Springer, 2013.

[27] "Block diagram of a typical heterogenous architecture." https://www.embedded.com/wp-content/uploads/contenteetimes-images-design-embedded-2017-ec2-7-c.jpg (accessed Jul. 13, 2020).

[28] "Sparse matrix," *Wikipedia*. Aug. 24, 2020, Accessed: Sep. 08, 2020. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Sparse_matrix&oldid=974657894.

[29] D. Goldberg, "What Every Computer Scientist Should Know about Floating-Point Arithmetic," *ACM Comput Surv*, vol. 23, no. 1, pp. 5–48, Mar. 1991, doi: 10.1145/103162.103163.

[30] S. Rennich, "CUDA Streams and Concurrency," [Online]. Available: https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf.

[31] "CUDA Streams," *NVIDIA Developer Blog*, Jan. 23, 2015. https://developer.nvidia.com/blog/gpu-pro-tip-cuda-7-streams-simplify-concurrency/ (accessed Jul. 14, 2020).

[32] "CUDA C++ Programming Guide." http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (accessed Jul. 14, 2020).

[33] "Using Shared Memory in CUDA C/C++," *NVIDIA Developer Blog*, Jan. 29, 2013. https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/ (accessed Jul. 14, 2020).

[34] I. E. Venetis, V. Saltogianni, S. Stiros, and E. Gallopoulos, "Multivariable inversion using exhaustive grid search and high-performance GPU processing: a new perspective," *Geophys. J. Int.*, vol. 221, no. 2, pp. 905–927, May 2020, doi: 10.1093/gji/ggaa042.

[35] "Details for Component Intel Core i5-7400," *SiSoftware Official Live Ranker*. http://ranker.sisoftware.co.uk/show_device.php?q=c9a598d1bfcbaec2e2a1cebcd9f990a588bf8bbb8badcaf7daebcdbf82b294fdc0f1d7bf82b791e9d4e5c3a6c3fecee89ba69e&l=en (accessed Jul. 14, 2020).