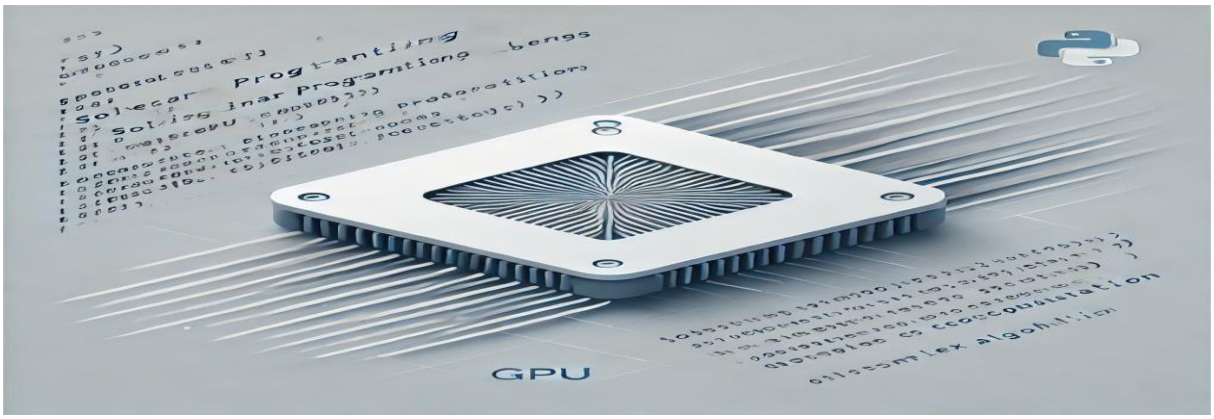
ΔΙΕΘΝΕΣ
ΠΑΝΕΠΙΣΤΗΜΙΟ
ΤΗΣ ΕΛΛΑΔΟΣ

ΣΧΟΛΗ ΜΗΧΑΝΙΚΩΝ

ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΚΑΙ ΗΛΕΚΤΡΟΝΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

«Επίλυση προβλημάτων Γραμμικού Προγραμματισμού σε Python χρησιμοποιώντας επιτάχυνση GPU»



Του φοιτητή
Μπαλάφα-Καραμάτσικου Δημήτριου
Αρ. Μητρώου: 164710

Επιβλέπων
Αντωνίου Ευστάθιος
Βαθμίδα Καθηγητής

Θεσσαλονίκη 2024

Επίλυση προβλημάτων Γραμμικού Προγραμματισμού σε Python χρησιμοποιώντας επιτάχυνση GPU

22117

Δημήτριος Μπαλάφας-Καραμάτσικος
Ευστάθιος Αντωνίου

23 Νοεμβρίου 2022

Ημερομηνία περάτωσης Δ.Ε. 9 Σεπτεμβρίου 2024

Βεβαιώνω ότι είμαι ο συγγραφέας αυτής της εργασίας και ότι κάθε βοήθεια την οποία είχα για την προετοιμασία της είναι πλήρως αναγνωρισμένη και αναφέρεται στην εργασία. Επίσης, έχω καταγράψει τις όποιες πηγές από τις οποίες έκανα χρήση δεδομένων, ιδεών, εικόνων και κειμένου, είτε αυτές αναφέρονται ακριβώς είτε παραφρασμένες. Επιπλέον, βεβαιώνω ότι αυτή η εργασία προετοιμάστηκε από εμένα προσωπικά, ειδικά ως διπλωματική εργασία, στο Τμήμα Μηχανικών Πληροφορικής και Ηλεκτρονικών Συστημάτων του ΔΙ.ΠΑ.Ε.

Περίληψη

Η παρούσα διπλωματική εργασία επικεντρώνεται στην υλοποίηση του αλγορίθμου Simplex στη γλώσσα προγραμματισμού Python, με στόχο την επιτάχυνση των υπολογισμών μέσω GPU (Graphics Processing Unit). Ο αλγόριθμος Simplex αποτελεί μια ευρέως χρησιμοποιούμενη μέθοδο για την επίλυση προβλημάτων Γραμμικού Προγραμματισμού, τα οποία περιλαμβάνουν τη μεγιστοποίηση μιας γραμμικής αντικειμενικής συνάρτησης υπό γραμμικούς περιορισμούς. Δεδομένου ότι η μέθοδος βασίζεται σε πράξεις γραμμικής άλγεβρας, παρουσιάζεται ως κατάλληλη για επιτάχυνση μέσω GPU, εκμεταλλευόμενη την υπολογιστική ισχύ και τον παράλληλο χαρακτήρα των καρτών γραφικών. Στην εργασία αυτή, υλοποιείται ο αλγόριθμος Simplex σε δύο διαφορετικά περιβάλλοντα: αρχικά μέσω της βιβλιοθήκης NumPy για εκτέλεση σε CPU και στη συνέχεια μέσω της βιβλιοθήκης CuPy, η οποία προσφέρει αντίστοιχη λειτουργικότητα αλλά επιτρέπει την εκτέλεση σε GPU. Η σύγκριση της απόδοσης μεταξύ των δύο υλοποιήσεων γίνεται με βάση την ταχύτητα εκτέλεσης σε διάφορα στιγμιότυπα προβλημάτων, τόσο μικρής όσο και μεγάλης κλίμακας. Τα αποτελέσματα της εργασίας δείχνουν ότι η χρήση GPU μπορεί να προσφέρει σημαντική επιτάχυνση, ιδιαίτερα σε προβλήματα μεγάλης κλίμακας, συγκριτικά με την κλασική εκτέλεση σε CPU. Συνολικά, η μελέτη αυτή αναδεικνύει τις δυνατότητες που προσφέρει η χρήση GPU στον τομέα του Γραμμικού Προγραμματισμού και υποδεικνύει περαιτέρω προοπτικές βελτίωσης της απόδοσης αλγορίθμων βελτιστοποίησης.

«Solving Linear Programming Problems in Python using GPU Acceleration»

Dimitrios Balafas - Karamatsikos

Abstract

This thesis focuses on the implementation of the Simplex algorithm in the Python programming language, with the aim of accelerating computations using GPUs (Graphics Processing Units). The Simplex algorithm is a widely used method for solving Linear Programming problems, which involve maximizing a linear objective function subject to linear constraints. Given that the method relies on linear algebra operations, it is well-suited for acceleration via GPUs, leveraging the computational power and parallel nature of graphics cards. In this work, the Simplex algorithm is implemented in twodifferent environments: initially using the NumPy library for execution on the CPU, and subsequently using the CuPy library, which offers similar functionality but enables execution on the GPU. The performance comparison between the two implementations is based on execution speed for various snapshots of problems, both small and large in scale. The results of the study demonstrate that using GPUs can offer significant acceleration, particularly for large-scale problems, compared to traditional CPU execution. Overall, this study highlights the potential benefits of using GPUs in the field of LinearProgramming and suggests further opportunities for improving the performance of optimization algorithms.

Ευχαριστίες

Θα ήθελα να ευχαριστήσω, την οικογένεια μου, τους φίλους αλλά και τους συμφοιτητές μου για την συνεχή στήριξη και την ώθηση που μου προσέφεραν κατά τη διάρκεια των σπουδών μου. Επίσης θα ήθελα να ευχαριστήσω τον επιβλέποντα της διπλωματικής εργασίας, τον Καθηγητή κ. Ευστάθιο Αντωνίου, για την πολύτιμη στήριξη και τις συμβουλές του οι οποίες υπήρξαν καθοριστικές για την ολοκλήρωση της εργασίας μου.

Περιεχόμενα

Κατάλογος Σχημάτων

Κατάλογος Πινάκων

Συντομογραφίες

Δ.Ε.	Διπλωματική Εργασία
ΔΙΠΑΕ	Διεθνές Πανεπιστήμιο Ελλάδος
Γ.Π.	Γραμμικός Προγραμματισμός
ΕΕ	Επιχειρησιακή Έρευνα
LP	Linear Programming
GPU	Graphic Processing Unit
CPU	Central Processing Unit
GPGPU	General Purpose Graphic Processing Unit
SM	Streaming Multiprocessor
SIMT	Single-Instruction, Multiple-Thread
RAM	Random Access Memory
ROM	Read-Only Memory
API	Application Programming Interface
CAD	Computer Aided Design
HPC	High Performance Computing
BLAS	Basic Linear Algebra Subprograms
CUBLAS	CUDA Basic Linear Algebra Subprograms
LU	Lower Upper

Κεφάλαιο 1ο: Εισαγωγή

Στην παρούσα διπλωματική εργασία, με τίτλο «Επίλυση προβλημάτων Γραμμικού Προγραμματισμού σε Python χρησιμοποιώντας επιτάχυνση GPU», παρουσιάζεται η θεωρία του Γραμμικού Προγραμματισμού, με έμφαση στη μέθοδο Simplex, η οποία αποτελεί έναν από τους πιο διαδεδομένους αλγορίθμους για την επίλυση προβλημάτων βελτιστοποίησης. Στη συνέχεια, αναλύεται η αρχιτεκτονική των καρτών γραφικών και το περιβάλλον CUDA, που επιτρέπει την εκτέλεση παράλληλων υπολογισμών σε GPU. Στο τελευταίο μέρος της εργασίας, υλοποιείται ο αλγόριθμος Simplex με τη χρήση των βιβλιοθηκών Python, συγκεκριμένα της NumPy για εκτέλεση στον επεξεργαστή (CPU) και της CuPy για εκτέλεση στην κάρτα γραφικών (GPU). Τα αποτελέσματα της υλοποίησης συγκρίνονται με εκείνα της βελτιστοποιημένης μεθόδου Simplex, όπως αυτή έχει υλοποιηθεί στο πακέτο HiGHS της βιβλιοθήκης SciPy.

Σε αυτό το κεφάλαιο, εισάγεται το αντικείμενο της διπλωματικής εργασίας, με έμφαση στους στόχους και τη σημασία της επιτάχυνσης αλγορίθμων Γραμμικού Προγραμματισμού με τη χρήση GPU.

Στο δεύτερο κεφάλαιο με τίτλο: «Γραμμικός Προγραμματισμός & Μέθοδος Simplex», παρουσιάζεται η θεωρία του Γραμμικού Προγραμματισμού, συμπεριλαμβανομένης της διαμόρφωσης των μοντέλων και των βασικών υποθέσεων εφαρμογής. Εξετάζεται η γεωμετρική ερμηνεία του προβλήματος και γίνεται αναλυτική περιγραφή της μεθόδου Simplex, με παραδείγματα επίλυσης προβλημάτων. Επιπλέον, συζητούνται άλλοι αλγόριθμοι Γραμμικού Προγραμματισμού και περιγράφεται η Αναθεωρημένη μέθοδος Simplex.

Στο τρίτο κεφάλαιο με τίτλο: «GPU & CUDA» εξετάζεται η αρχιτεκτονική και η λειτουργία των καρτών γραφικών (GPU) και του περιβάλλοντος CUDA. Αναλύεται η εξέλιξη των επεξεργαστών, η δομή και λειτουργία των GPU σε σχέση με τις CPU, καθώς και το προγραμματιστικό μοντέλο CUDA, συμπεριλαμβανομένων των kernels, της ιεραρχίας νημάτων και μνήμης.

Στο τέταρτο κεφάλαιο με τίτλο: «Βιβλιογραφική Έρευνα υλοποίησης αλγορίθμου με επιτάχυνση GPU», παρουσιάζεται η βιβλιογραφική έρευνα σχετικά με την υλοποίηση του αλγορίθμου Simplex με επιτάχυνση GPU. Αναλύεται η στρατηγική υλοποίησης, η διαμόρφωση των βασικών ρουτινών και το πειραματικό περιβάλλον. Τα αποτελέσματα της επιτάχυνσης συγκρίνονται με εκείνα της κλασικής υλοποίησης.

Στο πέμπτο κεφάλαιο με τίτλο «Επιτάχυνση αλγορίθμου Simplex με χρήση της βιβλιοθήκης CuPy» , παρουσιάζεται η υλοποίηση του αλγορίθμου Simplex χρησιμοποιώντας τη βιβλιοθήκη CuPy για επιτάχυνση μέσω GPU. Συγκρίνονται οι επιδόσεις της υλοποίησης με εκείνες της NumPy, με ανάλυση των δομών δεδομένων, των μεθόδων αναλογιών και εκτέλεσης Simplex, καθώς και της διαχείρισης και μεταφοράς δεδομένων. Τα αποτελέσματα της υλοποίησης παρουσιάζονται και αναλύονται.

Τέλος, το έκτο κεφάλαιο της εργασίας με τίτλο: «Συμπεράσματα και μελλοντική εργασία», περιλαμβάνει τα συμπεράσματα που προκύπτουν από την ανάλυση των αποτελεσμάτων, καθώς και προτάσεις για μελλοντική εργασία, όπως η διερεύνηση περαιτέρω βελτιώσεων στην επιτάχυνση. αλγορίθμων Γραμμικού Προγραμματισμού και η ανάπτυξη νέων μεθόδων που να αξιοποιούν πλήρως τις δυνατότητες των σύγχρονων αρχιτεκτονικών GPU.

Κεφάλαιο 2ο: Γραμμικός Προγραμματισμός & Μέθοδος Simplex

2.1 Εισαγωγή

Ο Γραμμικός Προγραμματισμός (ΓΠ) αποτελεί την πιο δημοφιλή μαθηματική μέθοδο της Επιχειρησιακής Έρευνας (ΕΕ) και γενικότερα της Διοικητικής Επιστήμης. Μια πρώτη ερμηνεία για την Επιχειρησιακή Έρευνα είναι ότι οι λήψεις των αποφάσεων σε μια επιχείρηση, γίνονται με γνώμονα διαφόρων μαθηματικών μοντέλων με σκοπό τη βέλτιστη κατανομή των οικονομικών πόρων που αυτή διαθέτει [3]. Η διαδρομή της Επιχειρησιακής Έρευνας ξεκινάει εν μέσω του Β' Παγκοσμίου Πολέμου, όπου οι συμμαχικές δυνάμεις[1], υπό την πίεση του πολέμου, δημιούργησαν ομάδες επιστημόνων και ερευνητών, κυρίως μαθηματικών και οικονομολόγων με σκοπό να διαχειριστούν με τον καλύτερο δυνατό τρόπο τους πόρους τους, δηλαδή την καλύτερη κατανομή του ανθρώπινου ή του υλικού δυναμικού, χρησιμοποιώντας μαθηματικές τεχνικές [1], [4]. Λίγο αργότερα, το 1947, ο μαθηματικός George Bernard Dantzig, ο οποίος ήταν επικεφαλής του έργου «Scientific Computation of Optimum Programs» της πολεμικής αεροπορίας των Η.Π.Α., αποτύπωσε τη γενική μορφή του ΓΠ και ανακάλυψε τη μέθοδο Simplex, που επιλύει τα γραμμικά προβλήματα [2]. Ο Dantzig στο συγκεκριμένο έργο, ασχολήθηκε με θέματα όπως την βέλτιστη ανάπτυξη και συντήρηση του στρατιωτικού δυναμικού αλλά και τους χρόνους εφοδιασμού [3]. Οι συγκεκριμένες μαθηματικές τεχνικές για την επίλυση αυτών των προβλημάτων ήταν γραμμικά εκφρασμένες και τις αποκαλούσαν προγράμματα. Έτσι προέκυψε το όνομα Γραμμικός Προγραμματισμός, ο οποίος δεν έχει καμία σχέση με τον προγραμματισμό εφαρμογών λογισμικού, όπως είναι γνωστός σήμερα [3]. Μεγάλη επιρροή για τη διαμόρφωση της κατάστασης στους κλάδους εφαρμογής της ΕΕ στην μετά 1947 εποχή, είχαν μελέτες που δημοσιευτήκαν με πρωτοπόρους τους John Von Neumann, Leonid Kantorovic, Wassily Leontief και Tjalling Koopmans [2]. Πιο συγκεκριμένα ο Von Neumann δημοσίευσε μία εργασία για τη Θεωρία των Παιγνίων το 1928 και λίγο αργότερα, άλλη μία για την σταθερή οικονομική ανάπτυξη. Το 1936 ο Leontief δημοσίευσε την εργασία του για το μοντέλο εισροών-εκροών, ενώ ακόμα παλαιότερα, το 1823 υπήρξαν δημοσιεύσεις από τον Joseph Fourier (με την γνωστή σειρά Fourier) και το 1911 από τον διάσημο Βέλγο μαθηματικό, Charles Jean de la Valée Poussin [2]. Η μεγάλη επιτυχία της ΕΕ και πιο συγκεκριμένα του ΓΠ, στον στρατιωτικό τομέα, οδήγησαν τις επιχειρήσεις να ενστερνιστούν αυτές τις μαθηματικές τεχνικές για να μεγιστοποιήσουν τα κέρδη ή να ελαχιστοποιήσουν τα κόστη τους [1]. Η ραγδαία εξέλιξη αυτού του κλάδου συνοδεύτηκε από την επανάσταση της πληροφορικής, καθώς μόνο με την χρήση Η/Υ θα μπορούσαν να επιλυθούν προβλήματα μεγάλης κλίμακας με πολλά δεδομένα [1]. Η ανάπτυξη της πληροφορικής οδήγησε τον ΓΠ και γενικότερα το φάσμα της ΕΕ να βρίσκει εφαρμογές σε πολλούς τομείς, όπως είναι το μάρκετινγκ, η υγεία, η εκπαίδευση, τα χρηματοοικονομικά κ.α. [1]. Τα εργαλεία του ΓΠ που χρειάζονται για να εφαρμοστεί στους παραπάνω αλλά και σε κάθε τομέα που το επιδιώκει είναι τρία. Πρώτον, η διατύπωση των πραγματικών προβλημάτων σε μαθηματικά μοντέλα, δεύτερον, οι τεχνικές που θα χρησιμοποιηθούν για την επίλυση των παραπάνω μοντέλων, δηλαδή οι αλγόριθμοι και τρίτον, οι μηχανές για την εκτέλεση των αλγορίθμων, δηλαδή τα λογισμικά και οι Η/Υ [2]. Ο ΓΠ, χρησιμοποιείται για την εύρεση της βέλτιστης λύσης σε διάφορα προβλήματα για μία επιχείρηση σε οικονομικά και διοικητικά θέματα [1]. Ουσιαστικά, πρόκειται για ένα εργαλείο για την σωστότερη λήψη αποφάσεων από τις επιχειρήσεις. Διάφορα προβλήματα, αποτελούν οι πόροι όπου σε αρκετές περιπτώσεις μπορεί να είναι περιορισμένοι, είτε χρειάζονται καλύτερη διαχείρισή για το καλύτερο δυνατό αποτέλεσμα. Η κάθε επιχείρηση, θέτει ορισμένους στόχους που θέλει να πετύχει, κάτω όμως υπό δεδομένους περιορισμούς που είναι ανάγκη να ικανοποιούνται. Οι περιορισμοί αυτοί, αφορούν τους διαθέσιμους πόρους, όπως την σωστή κατανομή του ανθρώπινου δυναμικού, των διαθέσιμων κεφαλαίων αλλά και του εξοπλισμού της επιχείρησης, κ.α. [1]. Η επιχείρηση,

αποφασίζοντας με συγκεκριμένα κριτήρια, μπορεί να επιδιώκει στόχους όπως, η μεγιστοποίηση του κέρδους από τις πωλήσεις της ή την ελαχιστοποίηση του κόστους παραγωγής της. Η επίλυση των προβλημάτων και γενικότερα η επίτευξη των στόχων μιας επιχείρησης γίνεται κυρίως με τη μέθοδο Simplex, όπου έχει κυριαρχήσει σε αυτόν τον τομέα. Θεωρείται η πιο γνωστή και αποτελεσματική μέθοδος του Γραμμικού Προγραμματισμού και βρίσκει εφαρμογή σε προβλήματα μεγάλης κλίμακας από κάθε μορφής επιχείρηση ή δραστηριότητας [3].

2.2 Θεωρία Γραμμικού Προγραμματισμού

Ο Γραμμικός Προγραμματισμός είναι μια μαθηματική μέθοδος βελτιστοποίησης που επιλύει προβλήματα με ποσότητες που πρέπει είτε να μεγιστοποιηθούν είτε να ελαχιστοποιηθούν, έχοντας όμως και περιορισμούς. Ο Γραμμικός Προγραμματισμός, θεωρείται η πιο γνωστή μέθοδος της Επιχειρησιακής Έρευνας για μοντέλα όπου οι αντικειμενικές συναρτήσεις αλλά και οι περιορισμοί είναι γραμμικής μορφής. Ο ΓΠ αποτελείται από τρία μέρη: 1) Τις μεταβλητές απόφασης, που θα καθορίσουν την αντικειμενική συνάρτηση, 2) ο στόχος του κάθε προβλήματος, δηλαδή την μεγιστοποίηση ή την ελαχιστοποίηση και 3) τους περιορισμούς που πρέπει να ικανοποιούνται για το αποτέλεσμα. Η αντικειμενική συνάρτηση είναι μία μαθηματική έκφραση που συνδυάζει τις μεταβλητές απόφασης και τους συντελεστές της, για να επιτευχθεί η βέλτιστη λύση. Το αποτέλεσμα ενός προβλήματος ΓΠ προκύπτει από την μέγιστη ή ελάχιστη τιμή στη γραμμική συνάρτηση $f: f(x) = c_1 \times x_1 + \cdots \; c_n \times x_n$. Οι περιορισμοί αποτελούν τα όρια στο μοντέλο σχετικά με τους διαθέσιμους πόρους. Οι περιορισμοί μπορεί να αποτυπωθούν είτε ως εξισώσεις είτε ως ανισότητες. Επιπλέον, στους περιορισμούς, θα προστεθούν και αυτοί της μη αρνητικότητας με τις μεταβλητές απόφασης x_i ($x_1, x_2 \geq 0$), γιατί δεν γίνεται φυσικά μεγέθη όπως ποσότητες να είναι μικρότερα του 0. Η λύση του προβλήματος εντοπίζεται εντός των πλαισίων των γραμμικών ανισοτήτων, δηλαδή η εύρεση των τιμών των μεταβλητών x_i , που ταυτόχρονα ικανοποιούν και τους περιορισμούς. Στη συνέχεια, γίνεται αντικατάσταση των τιμών των μεταβλητών στην αντικειμενική συνάρτηση f, και το αποτέλεσμα, γραφικά (βλ. σχήμα 2.1), θα είναι το σημείο στο οποίο τέμνονται οι δύο ευθείες. Όλα τα σημεία από το σημείο τομής των δύο ευθειών και κάτω είναι στο χώρο εφικτών λύσεων. Ο χώρος εφικτών λύσεων είναι η σκιαγραφημένη περιοχή που ικανοποιεί όλους τους περιορισμούς και όποια τιμή βρίσκεται μέσα σε αυτό είναι υποψήφια, για τη βέλτιστη λύση του προβλήματος (βλ. σχήμα 2.2) [5].

2.2.1 Μοντέλο Γραμμικού Προγραμματισμού

Το πρότυπο μοντέλο του ΓΠ, διαμορφώνεται, με τις γραμμικές ανισότητες ή εξισώσεις για x_i (για $i = 1,2,3, \ldots n$), που μεγιστοποιούν ή ελαχιστοποιούν την γραμμική αντικειμενική συνάρτηση f: $max/minf(x) = c_1x_1 + c_2x_2 + c_3x_3 + \cdots + c_nx_n$:

$$a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \leq, =, \geq b_1$$
$$a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \leq, =, \geq b_2$$
$$\ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots$$
$$a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \leq, =, \geq b_m$$
$$x_1 \geq 0, x_2 \geq 0, \ldots, x_n \geq 0$$

Με πραγματικούς συντελεστές, a_{ij}, b_i, c_j, για i =1,2,…,n και j=1,2,…,m. Οι περιορισμοί μπορεί να έχουν πρόσημο, είτε μεγαλύτερο ή ίσο(≥) , είτε μικρότερο ή ίσο (≤), είτε ίσο (=) [4].

Οι παραπάνω παραλλαγές, περιλαμβάνουν όλα τα προβλήματα ΓΠ, που ανήκουν στην κατηγορία κανονικής μορφής προβλημάτων ΓΠ [1].

Το μοντέλο του ΓΠ, μπορεί να γραφεί και με την χρήση πινάκων. Πιο συγκεκριμένα [1]:

$$\max f(x) = C \times X$$

Με περιορισμούς δομής

$$A \times X \leq B$$

Και τους περιορισμούς μη αρνητικότητας

$$X \geq 0$$

Όπου:

$$A = \begin{bmatrix} a_{11} & a_{12} & . & . & a_{1n} \\ a_{21} & a_{22} & . & . & a_{2n} \\ . & . & . & . & . \\ . & . & . & . & . \\ a_{m1} & a_{m2} & . & . & a_{2n} \end{bmatrix}, \quad X = \begin{bmatrix} x_1 \\ x_2 \\ . \\ . \\ x_n \end{bmatrix} \qquad B = \begin{bmatrix} b_1 \\ b_2 \\ . \\ . \\ b_n \end{bmatrix} \qquad C = [c_1, c_2, \dots c_n].$$

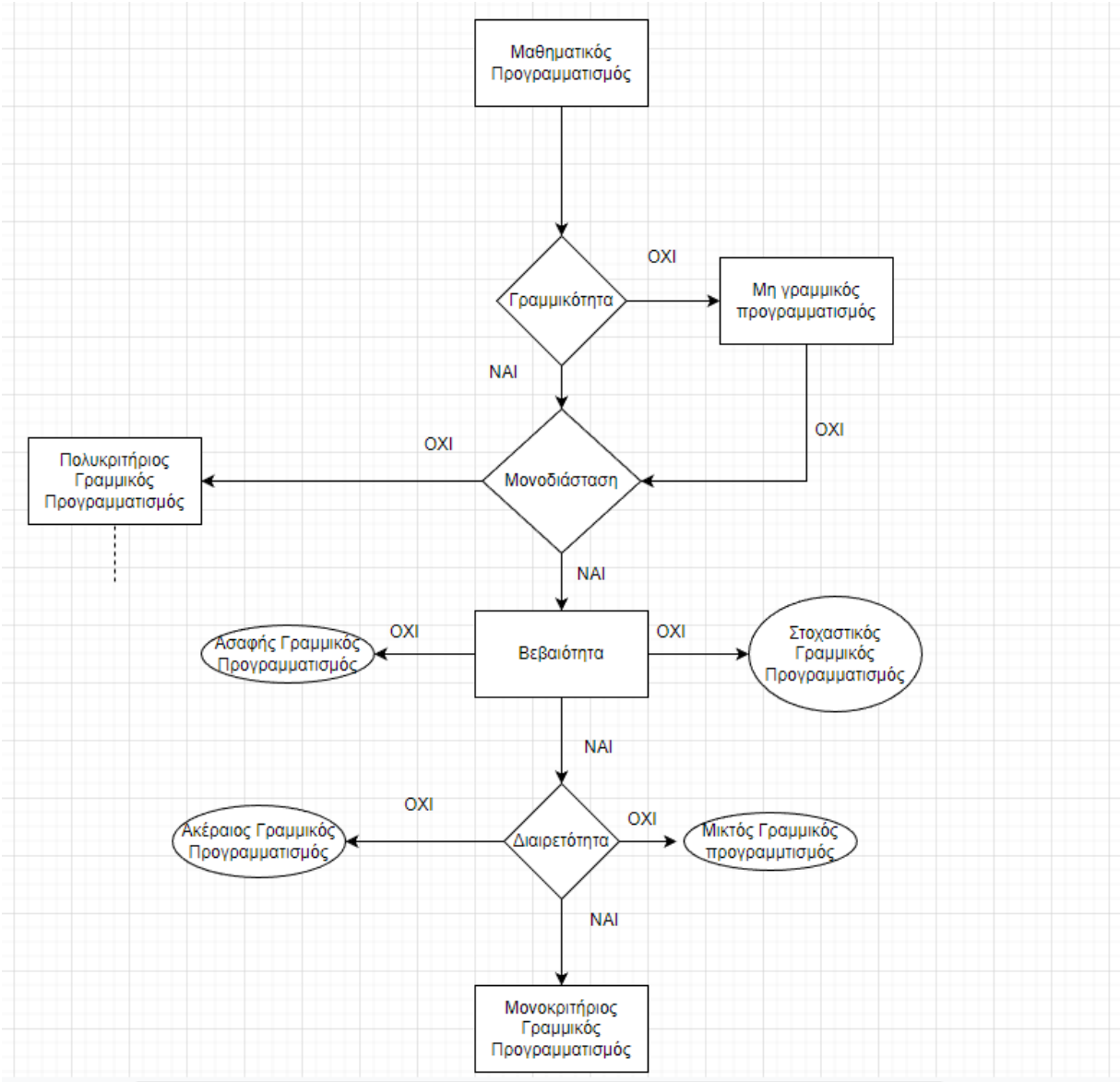
Ένα Γραμμικό Πρόγραμμα, λέγεται ότι είναι στην κανονική του μορφή, όταν αναγράφεται όπως στην παραπάνω μορφή.

2.2.2 Διαμόρφωση μοντέλου ΓΠ

Η διαδικασία αντιμετώπισης προβλημάτων με τη βοήθεια του ΓΠ, μπορεί να συνοψισθεί σε μερικά στάδια [1] – [4]. Αρχικά, γίνεται η διαμόρφωση του προβλήματος. Η διαμόρφωση του προβλήματος, δεν είναι η ίδια για κάθε κατάσταση. Κάθε πρόβλημα είναι διαφορετικό και η παραμικρή λεπτομέρεια το καθιστά μοναδικό. Υπάρχουν, μία σειρά από ενέργειες που είναι ανάγκη να γίνουν για την διαμόρφωση του μοντέλου, αλλά η εκτέλεση τους διαφέρει από πρόβλημα σε πρόβλημα. Ουσιαστικά, η διαδικασία βασίζεται αποκλειστικά στην λογική και οι γνώσεις που χρειάζεται να έχει αυτός που καλείται να αντιμετωπίσει το πρόβλημα, προέρχονται από αρκετούς κλάδους των θετικών επιστημών [1]. Οι ενέργειες που είναι απαραίτητες για την διαμόρφωση του μοντέλου, προϋποθέτουν, την αναγνώριση των βασικών χαρακτηριστικών του προβλήματος αλλά και την συγκέντρωση των απαραίτητων παραδοχών που επιτρέπουν την χρήση του ΓΠ και την πλήρη κατανόηση του προβλήματος [1]. Τα επόμενα στάδια αφορούν την δημιουργία του μοντέλου, όπου η αρχή γίνεται με τον καθορισμό των μεταβλητών απόφασης, οι οποίες μοντελοποιούν το ζητούμενο του προβλήματος. Με τον καθορισμό των μεταβλητών απόφασης, είναι δυνατή η διατύπωση της γραμμικής ή γραμμικών αντικειμενικών συναρτήσεων (εάν το πρόβλημα ανήκει στην κατηγορία της πολυκριτήριας βελτιστοποίησης) αλλά και των περιορισμών δομής, με γνώμονα τους στόχους που μπορεί να έχει μία επιχείρηση όπως η μεγιστοποίηση του κέρδους της. Το επόμενο στάδιο περιλαμβάνει το τεχνικό μέρος της διαδικασίας, όπου ο αναλυτής χρησιμοποιεί μαθηματικές μεθόδους και αλγόριθμους για την επίλυση του προβλήματος. Όσον αφορά προβλήματα βελτιστοποίησης γραμμικής μορφής και πιο συγκεκριμένα όταν υπάρχει μόνο ένα κριτήριο βελτιστοποίησης, τότε η χρήση της μεθόδου Simplex, κρίνεται απαραίτητη, ενώ σε πολυκριτήρια μοντέλα ΓΠ, θα χρησιμοποιηθούν πιο εξειδικευμένες τεχνικές, βασισμένες όμως στην μέθοδο Simplex. [4]. Το τελευταίο στάδιο, αφορά την αξιολόγηση του αποτελέσματος, για παράδειγμα σε περίπτωση που τα αποτελέσματα δεν ικανοποιούν την επιχείρηση, τότε μπορεί να γίνει αναθεώρηση μερικών παραγόντων όπως η αλλαγή πολιτικής που μπορεί να σημαίνει αύξηση αντικειμενικών συναρτήσεων, ή η αύξηση του προϋπολογισμού των πόρων κ.α. [4].

2.2.3 Προϋποθέσεις Εφαρμογής ΓΠ

Για την επίλυση ενός προβλήματος με το κλασικό μοντέλο του ΓΠ, είναι ανάγκη να ισχύουν, τέσσερις υποθέσεις, οι οποίες θα αναφερθούν. Η πρώτη υπόθεση είναι αυτή της γραμμικότητας. Με τον όρο γραμμικότητα, εννοείται ότι οι σχέσεις που περιγράφουν την αντικειμενική συνάρτηση και τους περιορισμούς είναι γραμμικές. Η υπόθεση της γραμμικότητας χωρίζεται σε δύο υποθέσεις, αυτή της αναλογικότητας και αυτή της προσθετικότητας. Αρχικά, η υπόθεση της αναλογικότητας μπορεί να αναλυθεί, εάν για παράδειγμα μια επιχείρηση παραγωγής ελαιόλαδου, για την παραγωγή μιας μονάδας προϊόντος Π_1: 1 λίτρο ελαιόλαδο, απαιτούνται 5 κιλά ελιές, τότε $5 \times x_1$ (x_1: ποσότητα προϊόντος Π_1 που θα παραχθεί) απαιτούνται για την παραγωγή x_1 μονάδων προϊόντος Π_1 για κάθε $x_1 \geq 0$ [3]. Η υπόθεση της προσθετικότητας αναφέρεται στις διαφορετικές ποσότητες για την παραγωγή μιας μονάδαςπροϊόντος αθροιστικά. Για παράδειγμα για την παραγωγή μιας μονάδας προϊόντος Π_1 απαιτούνται $5x_1 + 6x_2 + 2x_3$. Σε περίπτωση που δεν ισχύει η υπόθεση της γραμμικότητας, τότε το πρόβλημα ανήκει στη κατηγορία του μη γραμμικού προγραμματισμού. Η δεύτερη υπόθεση είναι αυτή της διαιρετότητας, όπου δίνεται ότι όλες οι τιμές των συντελεστών της αντικειμενικής συνάρτησης αλλά και των περιορισμών θα είναι ακέραιοι και μη αρνητικοί αριθμοί [3]. Στη περίπτωση αυτή το πρόβλημα ανήκει στην κατηγορία του ακέραιου γραμμικού προγραμματισμού και είναι πιθανό να είναι και στη κατηγορία των μη γραμμικών προβλημάτων, αρά και να χρειάζεται διαφορετική αντιμετώπιση από τηνμέθοδο Simplex [4]. Στην περίπτωση όπου μερικές μόνο μεταβλητές θα έχουν ακέραιες τιμές, ενώ άλλες θα ανήκουν στους πραγματικούς μη αρνητικούς αριθμούς, τότε το πρόβλημα ανήκει στην κατηγορία του μικτού γραμμικού προγραμματισμού [4]. Η τρίτη υπόθεση είναι αυτή της βεβαιότητας, όπου όλα τα δεδομένα του προβλήματος, δηλαδή οι τιμές των συντελεστών της αντικειμενικής συνάρτησης και οι τιμές των περιορισμών και των συντελεστών τους είναι με βεβαιότητα γνωστά και δεν θα καθοριστούν από πιθανοθεωρητικές κατανομές [3]. Στη περίπτωση που δεν είναι γνωστοί με βεβαιότητα ορισμένοι συντελεστές αλλά επηρεάζονται από στατιστικές μεθόδους, τότε το πρόβλημα ανήκει στην κατηγορία του στοχαστικού προγραμματισμού, ενώ όταν μερικά δεδομένα είναι ασαφή, τότε το πρόβλημα ανήκει στην κατηγορία του ασαφή προγραμματισμού και ομοίως με τον ακέραιο προγραμματισμό, αντιμετωπίζεται με άλλες τεχνικές εκτός της Simplex. Τέλος, η υπόθεση της μονοδιάστασης, όπου είναι ανάγκη να υπάρχει μόνο μία αντικειμενική συνάρτηση. Στη περίπτωση που ισχύει αυτό, τότε το πρόβλημα ανήκει στη κατηγορία του μονοκριτήριου γραμμικού προγραμματισμού, ενώ σε αντίθετη περίπτωση ανήκει στη κατηγορία του πολυκριτήριου γραμμικού προγραμματισμού [4].



Σχήμα 2.1 Κατηγοριοποίηση προβλημάτων μαθηματικού προγραμματισμού [4]

2.2.4 Παράδειγμα διαμόρφωσης προβλήματος Γραμμικού Προγραμματισμού

Αφού αναφέρθηκαν η θεωρία αλλά και το μοντέλο του Γραμμικού Προγραμματισμού, είναι ανάγκη να εφαρμοσθούν και να παρουσιαστούν η θεωρία και οι κανόνες του, με την μορφή της επίλυσης ενός παραδείγματος, ώστε να γίνουν καλύτερα κατανοητές οι έννοιες που έχουν αναφερθεί.

Μια εταιρεία παραγωγής επίπλων, παράγει καρέκλες και τραπέζια. Κάθε τραπέζι απαιτεί 4 ώρες ξυλουργικής εργασίας και 2 ώρες εργασίας για να βαφτεί, ενώ μια καρέκλα απαιτεί 3 ώρες και 1 ώρα εργασίας αντίστοιχα. Οι διαθέσιμες ώρες απασχόλησης για την ξυλουργική εργασία και του βαψίματος, είναι 240 και 100 αντίστοιχα. Κάθε τραπέζι έχει έσοδα των 200 ευρώ ενώ κάθε καρέκλα των 80 ευρώ. Να διατυπωθεί το πρόβλημα σε μοντέλο γραμμικού προγραμματισμού.

Η μοντελοποίηση του προβλήματος έχει 3 βήματα.

Βήμα 1ο: Καθορισμός μεταβλητών απόφασης:

- Τα προϊόντα που παράγονται είναι το τραπέζι και η καρέκλα. Άρα, δύο θα είναι και οι μεταβλητές απόφασης. Το τραπέζι θα αναγράφεται ως «Τ» και η καρέκλα ως «Κ».

Βήμα 2ο: Διαμόρφωση αντικειμενικής συνάρτησης:

- Στην εκφώνηση αναφέρεται ότι κάθε κατηγορία προϊόντος έχει 200 και 80 ευρώ κέρδος αντίστοιχα. Άρα, ο σκοπός της αντικειμενικής συνάρτησης είναι να μεγιστοποιηθεί το συνολικό κέρδος. Συνεπώς:

$$Max: Z = 200T + 80K$$

Βήμα 3ο: Εύρεση των περιορισμών:

- Οι περιορισμοί στο συγκεκριμένο παράδειγμα, είναι οι ώρες ξυλουργικής εργασίας και εργασίας βαψίματος, όπου είναι 240 και 100 ώρες αντίστοιχα. Στο παράδειγμα αναφέρεται, ότι για την κατασκευή ενός τραπεζιού, απαιτούνται 4 ώρες ξυλουργικής εργασίας και 2 ώρες εργασίας βαψίματος, ενώ για την κατασκευή μίας καρέκλας απαιτούνται 3 ώρες ξυλουργικής εργασίας και 1 ώρες εργασίας βαψίματος. Συνεπώς:

$$4T + 3K \leq 240$$
$$2T + 1K \leq 100$$

  Και φυσικά, οι περιορισμοί μη αρνητικότητας, όπου όλες οι μεταβλητές πρέπει να είναι μη είναι αρνητικές:

$$T, K \geq 0$$

2.3 Γεωμετρική ερμηνεία – Γραφική επίλυση

Για την καλύτερη κατανόηση της θεωρίας του ΓΠ, είναι ανάγκη να εφαρμοσθούν και στη πράξη, ορισμένοι κανόνες, ιδιότητες αλλά κυρίως το μοντέλο του ΓΠ, το οποίο στην γραφική του μορφή, δείχνει τη χρησιμότητα του. Θα χρησιμοποιηθεί το προηγούμενο πρόβλημα, που μοντελοποιήθηκε θεωρητικά, αλλά σε αυτή την ενότητα με τη βοήθεια του λογισμικού GeoGebra και πιο συγκεκριμένα της Graphing Calculator έκδοσης, θα αναπαρασταθεί γραφικά το μοντέλο του συγκεκριμένου προβλήματος, και θα γίνει μία συνοπτική αναφορά σε ειδικές περιπτώσεις ορισμένων προβλημάτων όπου έχουν ιδιαίτερες λύσεις ή δεν έχουν και καμία λύση.

Η αντικειμενική συνάρτηση και οι περιορισμοί είναι οι εξής:

$$maxZ = 200x_1 + 80x_2$$

$$4x_1 + 3x_2 \leq 240$$

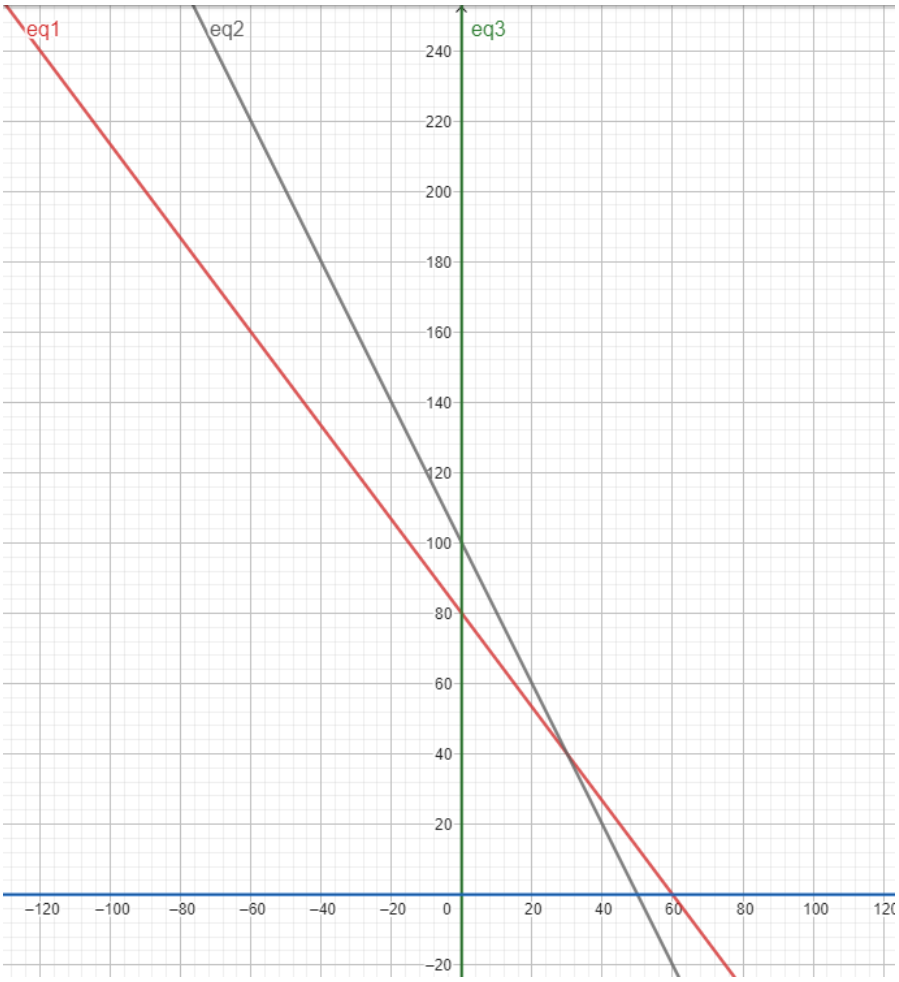
$$2x_1 + 1x_2 \leq 100$$

$$x_1 \geq 0$$

$$x_2 \geq 0$$

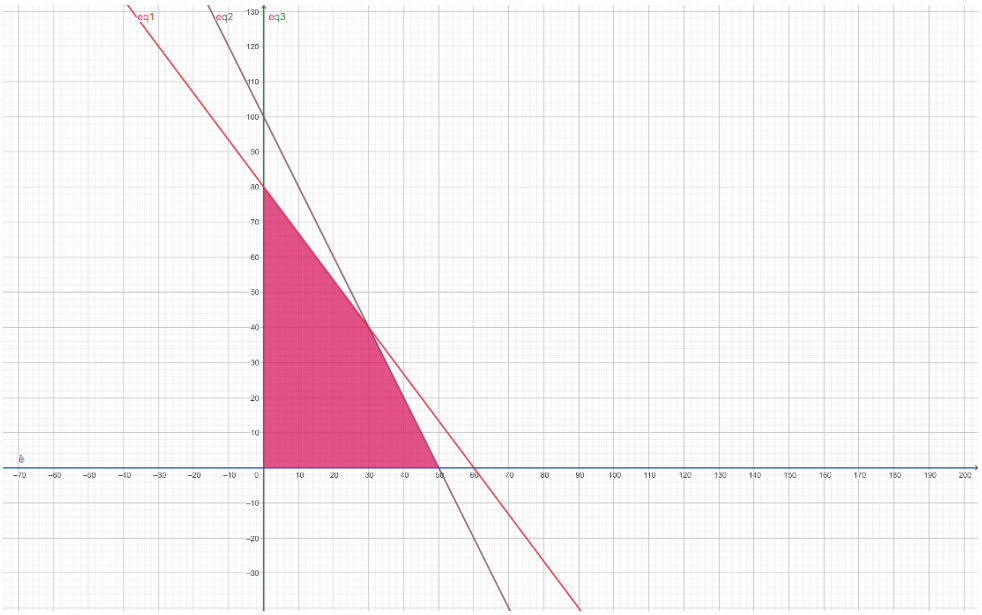
Το πρώτο βήμα για την γραφική επίλυση είναι η μετατροπή των ανισοτήτων σε ισότητες, για να χαραχθούν οι ευθείες στον άξονα και πιο συγκεκριμένα στο 1ο τεταρτημόριο, καθώς όλες οι τιμές είναι μη αρνητικές. Οι ευθείες που αντιστοιχούν σε κάποιον περιορισμό, ονομάζονται περιοριστικές ευθείες και το σημείο στο οποίο τέμνονται οι δύο περιοριστικές ευθείες, ονομάζεται κορυφή. Στη συνέχεια θα μηδενιστούν εναλλάξ οι δύο μεταβλητές των περιορισμών για την εύρεση των ευθειών. Με αυτές τις

ενέργειες οι τιμές του πρώτου περιορισμού είναι, T= 60, K = 80 και ο δεύτερος, T= 50 και K=100, όπως φαίνεται στο παρακάτω σχήμα 2.2.



Σχήμα 2.2: Γραφική αναπαράσταση περιορισμών

Από τον σχεδιασμό προκύπτει, πως από την μετατροπή των περιορισμών σε ισότητες σχεδιάζονται οι ευθείες και έτσι αποκλείεται το ημι-επίπεδο που δεν ικανοποιεί τις ανισότητες. Έτσι παραμένει η περιοχή που ικανοποιεί όλους τους περιορισμούς και σε αυτή βρίσκονται όλες οι εφικτές λύσεις. Εφικτές είναι οι λύσεις που ικανοποιούν τους δύο περιορισμούς και αυτούς της μη-αρνητικότητας και είναι όλα τα σημεία που βρίσκονται εντός ή στο σύνορο της εφικτής περιοχής που είναι χρωματισμένη με μωβ στο σχήμα 2.3.



Σχήμα 2.3: Εφικτή περιοχή

Στη συνέχεια για τιμές του Z σχεδιάζεται η αντικειμενική συνάρτηση, ώστε να εντοπιστεί η βέλτιστη λύση στην εφικτή περιοχή, που μεγιστοποιεί την αντικειμενική συνάρτηση. Μετατρέπεται η Z σε εξίσωση ευθείας και έστω $x_1 = T \; και \; X_2 = K$,δηλαδή :

$$x_2 = -\frac{200}{80}\, x_1 + \frac{1}{200} Z \;\Rightarrow x_2 = \; -\frac{5}{2} x_1 + \frac{1}{200} Z$$

Στις περισσότερες περιπτώσεις, η βέλτιστη λύση βρίσκεται σε ένα γωνιακό σημείο της εφικτής περιοχής και εφόσον υπάρχει μόνο μία λύση και όχι άπειρες τότε το σημείο είναι πάντα μία γωνία της εφικτής περιοχής. Το γωνιακό σημείο μετατοπίζεται ανάλογα με την κλίση της αντικειμενικής όπου στην περίπτωση αυτή είναι το $-5/2$ (συντελεστής διεύθυνσης) [7].

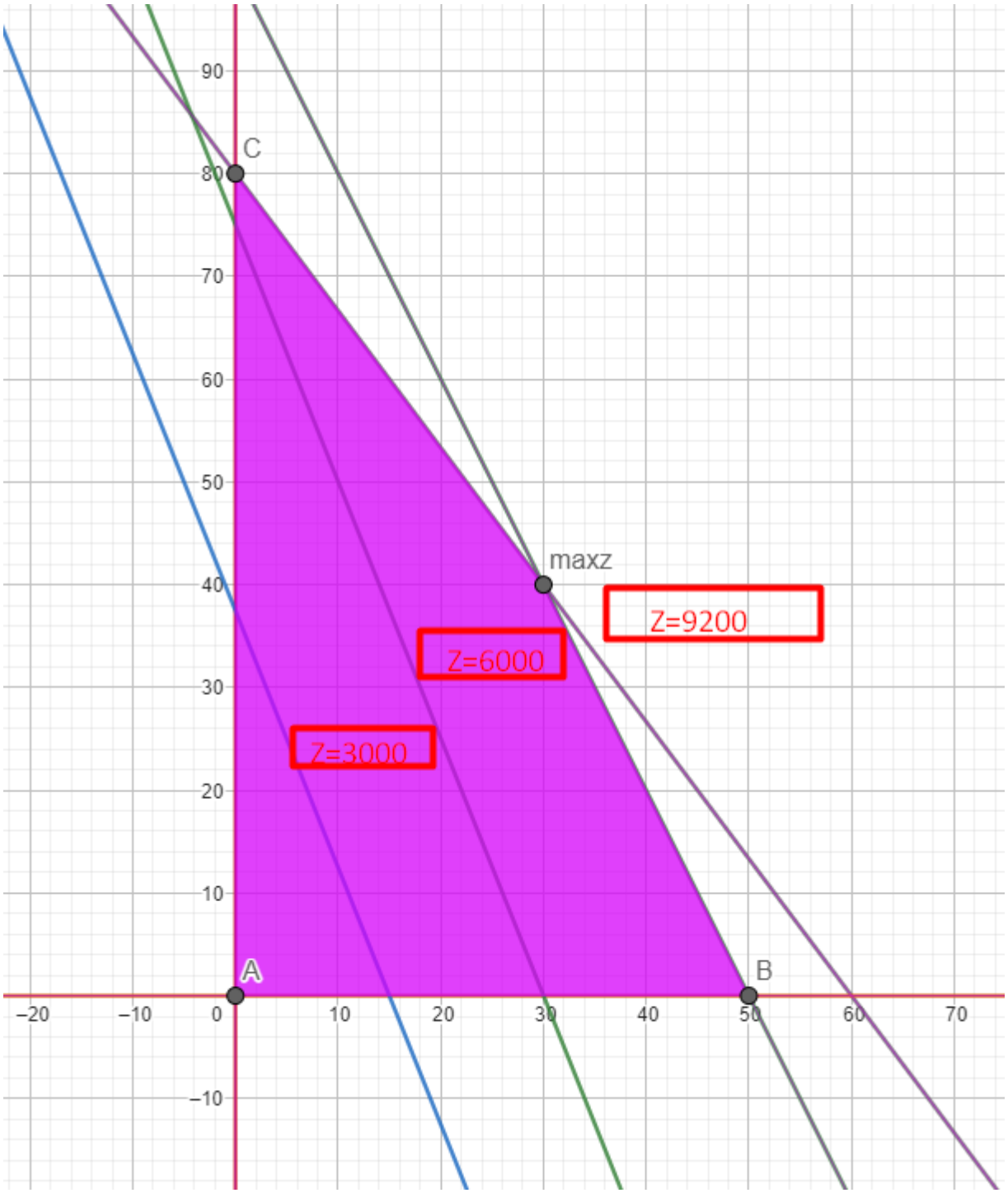
Για την εύρεση του γωνιακού σημείου που μεγιστοποιεί την αντικειμενική συνάρτηση, είναι ανάγκη να επιλυθεί το σύστημα με τους δύο περιορισμούς:

$$4x_1 + 3x_2 = 240$$

$$2x_1 + 1x_2 = 100$$

Από την επίλυση του συστήματος προκύπτει ότι οι τιμές που μεγιστοποιούν την Z είναι οι :

$x_1 = 30, x_2 = 40$ και έτσι η μέγιστη τιμή της Z είναι $Z = 200x_1 + 80x_2 \Rightarrow Z = 200 * 30 + 80 * 40$ $\Rightarrow Z = 9{,}200.$



Σχήμα 2.4 : Η κορυφή maxz που βρίσκεται η βέλτιστη λύση και μεγιστοποιεί την Z

Η γραφική επίλυση του συγκεκριμένου παραδείγματος, είχε μία μοναδική λύση που μεγιστοποιεί την αντικειμενική συνάρτηση. Για διαφορετικές τιμές της αντικειμενικής συνάρτησης, εφόσον ο συντελεστής διεύθυνσης παραμένει ίδιος (δηλαδή οι τιμές των μεταβλητών), προκύπτουν ευθείες από την αντικειμενική συνάρτηση παράλληλες μεταξύ τους. Υπάρχουν όμως, και κάποιες ειδικές περιπτώσεις, όπου υπάρχουν άπειρες βέλτιστες λύσεις ή και καμία βέλτιστη λύση. Παρακάτω θα γίνει μία σύντομη περιγραφή αυτών των ειδικών περιπτώσεων με παραδείγματα και γραφική επίλυση.

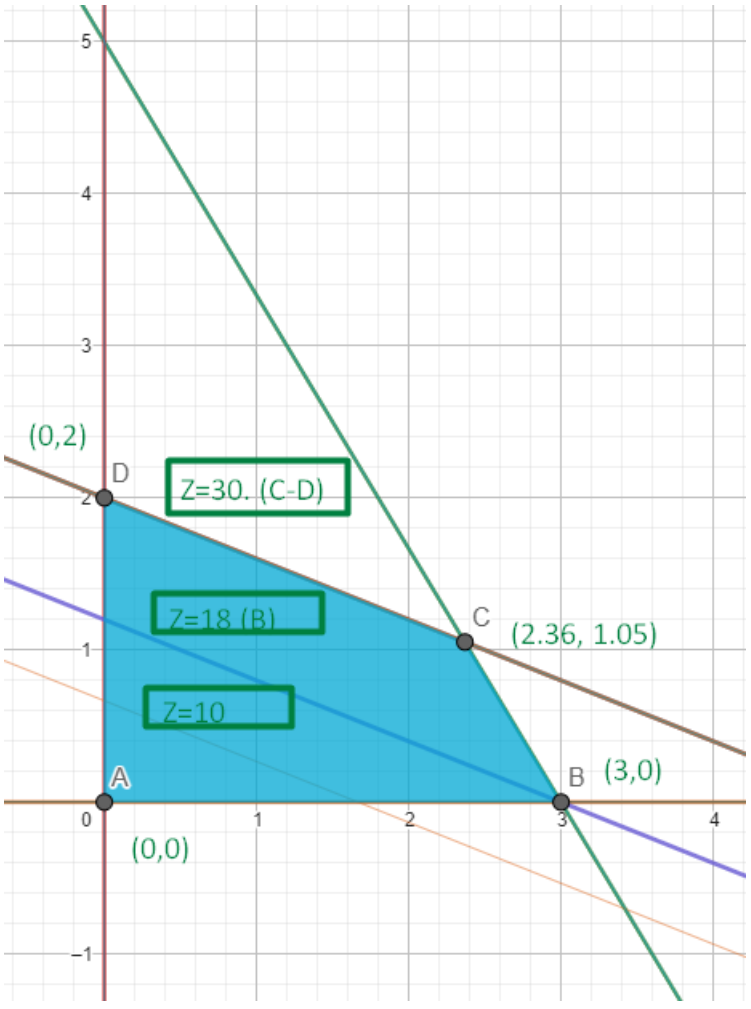
2.3.1 Ειδικές Περιπτώσεις

Μία από τις ειδικές περιπτώσεις είναι όταν ένα πρόβλημα έχει άπειρες λύσεις. Παρουσιάζεται, όταν ο συντελεστής διεύθυνσης ευθείας της αντικειμενικής συνάρτησης είναι ίδιος με αυτόν ενός περιορισμού. Αυτό σημαίνει ότι οι ευθείες είναι παράλληλες και οι βέλτιστες λύσεις είναι άπειρες και βρίσκονται όλες πάνω σε ένα ευθύγραμμο τμήμα. Μία άλλη ειδική περίπτωση είναι αυτή του αδύνατου προβλήματος όπου δεν υπάρχει καμία λύση. Πιο συγκεκριμένα αυτό συμβαίνει όταν οι περιορισμοί είναι ασυμβίβαστοι, δηλαδή σχηματικά δεν μπορεί να χαραχθεί ένα πολύγωνο όπου θα υπάρχει εφικτή περιοχή και μέσα σ ’αυτή θα περιέχεται η άριστη λύση. Τέλος, μία ακόμη περίπτωση είναι αυτή του μη φραγμένου προβλήματος όπου εάν σε ένα πρόβλημα είτε μεγιστοποίησης είτε ελαχιστοποίησης δεν υπάρχει φραγμένη περιοχή τότε η τιμή της αντικειμενικής συνάρτησης είναι αρκετά πιθανό να οδηγηθεί σε ανυπολόγιστη αύξηση ή μείωση.

- Άπειρες Λύσεις:

Έστω αντικειμενική συνάρτηση $Z = 6x + 15y$ με περιορισμούς μη-αρνητικότητας $x, y \geq 0$ και περιορισμούς δομής $5\chi + 3y \leq 15$ και $2x + 5y \leq 10$, όπου ζητείται η μεγιστοποίηση της Z.

Επαναλαμβάνοντας τη διαδικασία που έχει αναφερθεί για την γραφική επίλυση του προβλήματος προκύπτει το παρακάτω σχήμα:



Σχήμα 2.5 : Άπειρες λύσεις στο ευθύγραμμό τμήμα C-D

Παρατηρείται ότι, η εφικτή περιοχή είναι το μπλε πολύγωνο (σχήμα 2.5) και έχει προκύψει από τους περιορισμούς, με κορυφές, A-B-C-D. Για την εύρεση της βέλτιστης λύσης, γίνεται αντικατάσταση των μεταβλητών της Z, με τα σημεία της κορυφής C.

$$Z = 6x + 15y \Rightarrow Z = 6 * 2.36 + 15 * 1.05 \Rightarrow Z=30$$

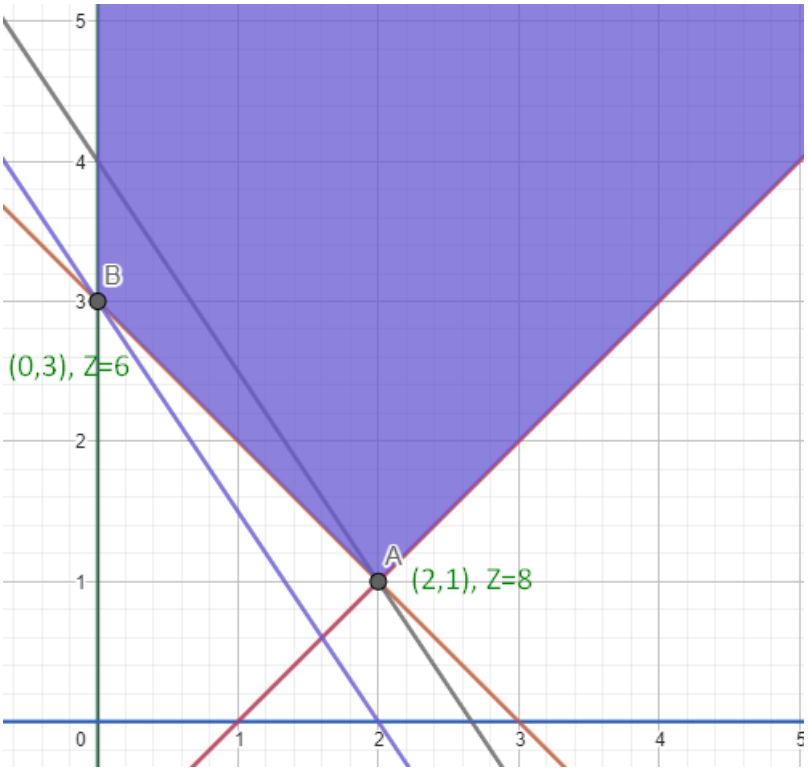
Για διάφορες τιμές της αντικειμενικής συνάρτησης προκύπτουν ευθείες παράλληλες μεταξύ τους (σχήμα 2.5). Αντικαθιστώντας τις μεταβλητές απόφασης της Z, με τα σημεία κορυφής C, η ευθεία της αντικειμενικής συνάρτησης ταυτίζεται με το ευθύγραμμό τμήμα C-D, του πολυγώνου A-B-C-D, που αποτελεί την εφικτή περιοχή οπού εντός της, εάν βρίσκεται η ευθεία της αντικειμενικής συνάρτησης, υπάρχουν λύσεις (εφικτές). Στην περίπτωση όπου Z=30, η λύση είναι η βέλτιστη και στο συγκεκριμένο παράδειγμα, επειδή η ευθεία της Z, ταυτίζεται με το ευθύγραμμό τμήμα C-D, σημαίνει ότι υπάρχουν άπειρες βέλτιστες λύσεις σε οποιοδήποτε σημείο του ευθυγράμμου τμήματος C-D. Ως απόδειξη, με αντικατάσταση τιμών των μεταβλητών απόφασης της Z, με το σημείο της κορυφής D, αποδεικνύεται ότι:

$$Z = 6x + 15y \Rightarrow Z = 6 * 0 + 15 * 2 \Rightarrow Z=30$$

- Μη-φραγμένο πρόβλημα:

Έστω αντικειμενική συνάρτηση $Z = 3x + 2y$ με περιορισμούς μη-αρνητικότητας $x, y \geq 0$ και περιορισμούς δομής $x - y \leq 1$ και $x + y \geq 3$, όπου ζητείται η μεγιστοποίηση της Z.

Επαναλαμβάνοντας τη διαδικασία που έχει αναφερθεί για την γραφική επίλυση του προβλήματος προκύπτει το παρακάτω σχήμα:



Σχήμα 2.6 : Μη-φραγμένη περιοχή

Από τα δεδομένα του προβλήματος, προκύπτουν οι ευθείες των περιορισμών, όπου έχουν κορυφές τα σημεία Α και Β. Συνεπώς η κοινή εφικτή περιοχή που προκύπτει δεν έχει κάποιο «φράγμα», ώστε να περιοριστούν οι εφικτές λύσεις και οι βέλτιστες ή βέλτιστη λύση στα όρια ενός πολύγωνου σχήματος. Σε αυτή τη περίπτωση η περιοχή της εφικτής περιοχής είναι από τις δύο κορυφές της έως το άπειρο. Στο παράδειγμα αυτό, η τιμή της αντικειμενικής συνάρτησης για τις δύο κορυφές είναι:

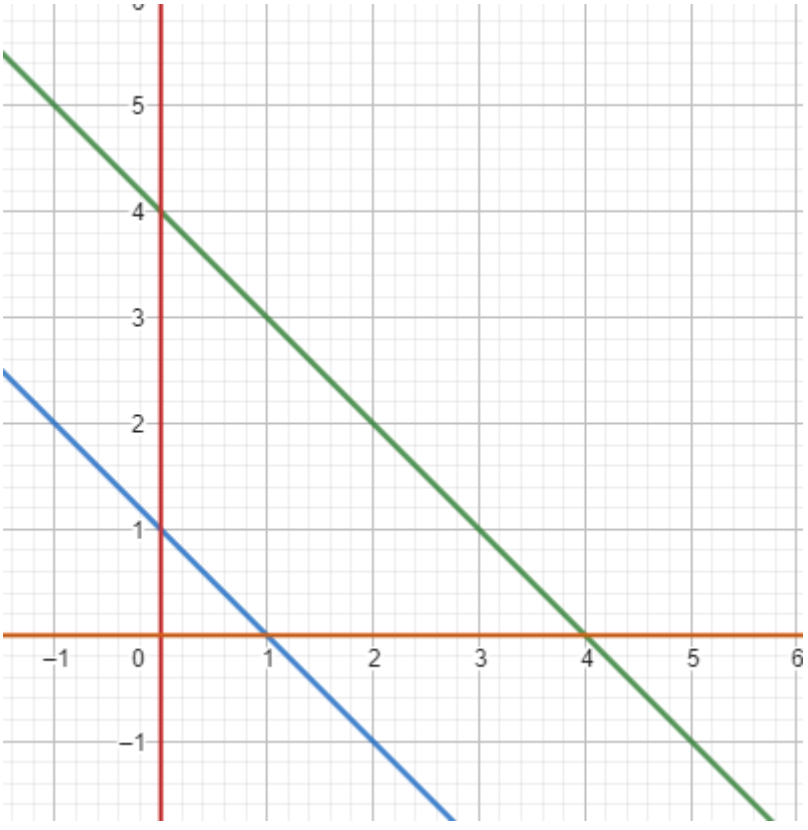
$Z_A = 3x + 2y \Rightarrow Z = 3 * 2 + 2 * 1 \Rightarrow Z=8$ *και* $Z_B = 3x + 2y \Rightarrow Z = 3 * 0 + 2 * 3 \Rightarrow Z=6$

Έτσι, η βέλτιστη τιμή για την μεγιστοποίηση της αντικειμενικής συνάρτησης είναι το άπειρο, ενώ εάν το πρόβλημα ζητούσε την ελαχιστοποίηση της Z, τότε η μικρότερη τιμής της είναι Z=6 από τις τιμές της κορυφής Β.

- Καμία Λύση:

Έστω αντικειμενική συνάρτηση $Z = 3x + 2y$ με περιορισμούς μη-αρνητικότητας $x, y \geq 0$ και περιορισμούς δομής $x + y \geq 4$ και $x + y \leq 1$, όπου ζητείται η μεγιστοποίηση της Z.

Επαναλαμβάνοντας τη διαδικασία που έχει αναφερθεί για την γραφική επίλυση του προβλήματος προκύπτει το παρακάτω σχήμα:



Σχήμα 2.7 : Κανένα κοινό σημείο

Από τα δεδομένα του προβλήματος, προκύπτουν οι ευθείες των περιορισμών. Παρατηρείται ότι, με βάση τους περιορισμούς, οι ευθείες δεν έχουν κανένα κοινό σημείο επαφής, άρα ούτε υπάρχει και εφικτή περιοχή από σχηματισμό πολυγώνου με κορυφές για εύρεση βέλτιστης λύσης. Συνεπώς, σε αυτό το πρόβλημα είναι αδύνατο να βρεθεί εφικτή λύση.

2.4 Κανονική Μορφή ΓΠ

Μέχρι και την προηγούμενη ενότητα, έγινε αναφορά σε προβλήματα γραμμικού προγραμματισμού στην γενική όμως μορφή τους, δηλαδή ένα πρόβλημα ΓΠ μπορεί να έχει ως στόχο την μεγιστοποίηση ή την ελαχιστοποίηση της αντικειμενικής συνάρτησης και οι τύποι των περιορισμών θα είναι ανισώσεις μεγαλύτερου ίσου ή μικρότερου ίσου ανάλογα με την αντικειμενική συνάρτηση ή εξισώσεις. Για την επίλυση όμως προβλημάτων με την μέθοδο Simplex, τα προβλήματα ΓΠ πρέπει να είναι στην κανονική τους μορφή. Ένα πρόβλημα είναι στην κανονική μορφή όταν η αντικειμενική συνάρτηση μεγιστοποιείται, οι μεταβλητές απόφασης είναι μη-αρνητικές και οι περιορισμοί είναι εξισώσεις.

Οπότε όλα τα προβλήματα μπορούν να μετασχηματιστούν σε προβλήματα κανονικής μορφής για να επιλυθούν με τη μέθοδο Simplex. Τα προβλήματα κανονικής μορφής έχουν το εξής πρότυπο.

$$maxf(x) = c_1x_1 + c_2x_2 + c_3x_3 + \cdots + c_nx_n,$$

με γραμμικούς περιορισμούς:

$$a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n =, b_1$$
$$a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2$$
$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$
$$a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n = b_m$$
$$x_1 \geq 0, x_2 \geq 0, \ldots, x_n \geq 0$$

Ο μετασχηματισμός οποιασδήποτε μορφής γραμμικού προγραμματισμού σε κανονική, γίνεται με την χρήση των μεταβλητών περιθωρίου (slack variable, s_i) που σημαίνουν την διαφορά μιας διαθέσιμης ποσότητας σε σχέση με αυτή που χρησιμοποιείται, και την αλλαγή της αντικειμενικής συνάρτησης με από ελαχιστοποίηση σε μεγιστοποίηση τον εξής τρόπο:

$$\max f(x) = cx \equiv -\min \mathrm{f(x)} = -cx$$

$$\sum_j a_{ij}x_j \geq b_i \equiv \sum_j a_{ij}x_j - s_i = b_i,\ s_i \geq 0$$

$$\sum_j a_{ij}x_j \leq b_i \equiv \sum_j a_{ij}x_j + s_i = b_i,\ s_i \geq 0$$

Πιο συνοπτικά:

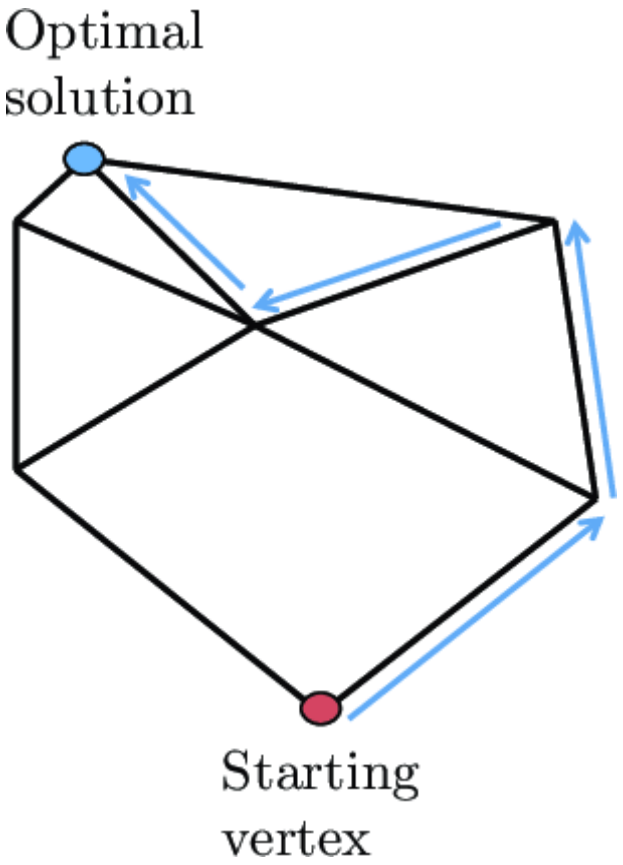
$$\max f(x) = cx$$
$$Ax = b$$
$$x \geq 0$$ [6].

2.5 Μέθοδος Simplex

Η μέθοδος Simplex, όπως έχει ήδη αναφερθεί, είναι μία από τις πιο γνωστές γνωστή και αποτελεσματικές μεθόδους για την βελτιστοποίηση προβλημάτων Γραμμικού Προγραμματισμού και αναπτύχθηκε από τον George Bernard Dantzig στα τέλη της δεκαετίας του 40'. Είναι μια επαναληπτική μέθοδος για την εύρεση άριστης λύσης. Εάν υπάρχουν λύσεις σε ένα πρόβλημα του ΓΠ, τότε αυτές βρίσκονται στις κορυφές της εφικτής περιοχής. Με την εφαρμογή της Simplex, διασχίζεται η εφικτή περιοχή, με επαναλαμβανόμενα βήματα αυξάνοντας κάθε φορά την τιμή της αντικειμενικής συνάρτησης με σκοπό να βρει το βέλτιστο αποτέλεσμα [6]. Η μέθοδος Simplex, είναι χρήσιμη για προβλήματα μεγάλης κλίμακας, με πολλές μεταβλητές απόφασης. Προηγουμένως έγινε αναφορά στην επίλυση προβλημάτων ΓΠ με γραφική μέθοδο, όμως αυτή είναι δυνατό να εφαρμοστεί μόνο, όταν οι μεταβλητές απόφασης είναι δύο. Στην πραγματικότητα τα προβλήματα είναι αρκετά πιο πολύπλοκα με πολλές μεταβλητές απόφασης, καθιστώντας έτσι την εφαρμογή της μεθόδου Simplex, επιτακτική.

2.5.1 Περιγραφή μεθόδου Simplex

Η μέθοδος Simplex με έναν έξυπνο τρόπο βρίσκει μέσω ελέγχου ενός μικρού υποσυνόλου από τις βασικές λύσεις, την βέλτιστη. Αρχικά, γίνεται έλεγχος για την εύρεση της βασικής εφικτής λύσης, όπου η Simplex, μετακινείται από κορυφή σε κορυφή ενός πολυέδρου (κάθε κορυφή είναι μία βασική εφικτή λύση) με σκοπό την αύξηση της τιμής της αντικειμενικής συνάρτησης, μέχρι να βρεθεί η βέλτιστη τιμή (σχήμα 2.8).



Σχήμα 2.8 : Γραφική αναπαράσταση ενός πολυέδρου

Με την μέθοδο Simplex αναπαρίσταται η τυποποιημένη μορφή της αντικειμενικής συνάρτησης και των περιορισμών με την μορφή πινάκων και αναζητείται η βέλτιστη λύση. Η αναπαράσταση ενός προβλήματος με την τυποποιημένη του μορφή, ονομάζεται ταμπλό Simplex (tableau) και στην πραγματικότητα είναι ένας επαυξημένος πίνακας που περιέχει τους συντελεστές των περιορισμών και τους συντελεστές της αντικειμενικής συνάρτησης στην τελευταία γραμμή του, ενώ στην τελευταία στήλη του περιέχονται οι τιμές από τις σταθερές της αντικειμενικής συνάρτησης.

Σε κάθε βήμα της μεθόδου Simplex, μία συγκεκριμένη εφικτή λύση αξιολογείται και οι πληροφορίες για τις λύσεις αναπαρίστανται στο ταμπλό. Στόχος είναι η επίτευξη της βέλτιστης λύσης, όμως ως πρώτο βήμα, είναι η εύρεση της βασικής εφικτής λύσης. Βασική εφικτή λύση υφίσταται όταν μία στήλη του ταμπλό Simplex είναι σε βασική μορφή, δηλαδή τα στοιχεία της αποτελούνται από μηδενικά στοιχεία τα οποία ονομάζονται μη-βασικές μεταβλητές και από στοιχεία που είναι ίσα με 1 όπου ονομάζονται βασικές μεταβλητές (Πίνακας 2.1). Έτσι η βασική εφικτή λύση είναι αυτή όπου η βασική μεταβλητή αντιστοιχίζεται με την τιμή της τελευταίας στήλης με τις σταθερές τιμές. Η βέλτιστη λύση προκύπτει, όταν στην τελευταία στήλη υπάρχουν μη αρνητικά στοιχεία (Πίνακας 2.5).

Στο επόμενο βήμα, γίνονται μετατροπές των μεταβλητών από μη-βασικές σε βασικές μέσω διαφόρων ενεργειών και ουσιαστικά με αυτές τις μετατροπές η simplex αναζητεί από κορυφή σε κορυφή του πολυέδρου την βέλτιστη λύση, αυξάνοντας έτσι την τιμή της αντικειμενικής συνάρτησης μέχρι να βρεθεί η βέλτιστη τιμή. Οι ενέργειες που απαιτούνται γίνονται με μία διαδικασία που ονομάζεται οδήγηση. Η διαδικασία της οδήγησης περιλαμβάνει μια σειρά από βήματα όπου:

Αρχικά, γίνεται έλεγχος του μεγαλύτερου αρνητικού αριθμού στη τελευταία γραμμή του πίνακα, αριστερά της στήλης των σταθερών τιμών της Z. Αν υπάρχει και δεύτερος ίδιος αριθμός, θα επιλεχτεί τυχαία ένας από τους δύο. Αφού βρεθεί, τότε η στήλη που βρίσκεται αυτό το στοιχείο, ονομάζεται στήλη οδηγός και η μεταβλητή που αντιστοιχείται, εισερχόμενη μεταβλητή. Κατά τη διαδικασία αυτή,αφού επιλέχθηκε η στήλη οδηγός, ουσιαστικά από τις τροποποιήσεις που θα γίνουν μέσω πράξεων, η στήλη οδηγός θα μετατραπεί και από μη-βασική σε βασική στήλη (διαδικασία της εύρεσης της βασικήςεφικτής λύσης). Η μεταβλητή, που μέσω πράξεων θα γίνει η στήλη της από βασική σε μη-βασική, θα ονομάζεται εξερχόμενη μεταβλητή. Στη συνέχεια θα γίνει διαίρεση των στοιχείων της στήλης οδηγού (εκτός του στοιχείου της τελευταίας γραμμής), με αυτά της τελευταίας στήλης με τις σταθερές τιμές της
Z. Ο μικρότερος αριθμός που θα προκύψει από τις διαιρέσεις θα ορίσει την γραμμή του ως γραμμή οδηγό (εάν υπάρχουν δυο ίσα αποτελέσματα, τότε επιλέγεται ένα τυχαία). Αν τα στοιχεία της στήλης οδηγού (πλην του τελευταίου στοιχείου), είναι αρνητικά, τότε η διαδικασία τερματίζεται και δεν υπάρχει λύση. Το στοιχείο οδηγός, είναι η τομή της στήλης οδηγού με την γραμμή οδηγός.

Στη συνέχεια, με την διαδικασία της οδήγησης, γίνονται αριθμητικές πράξεις, ώστε να μετατραπεί το στοιχείο οδηγός στον αριθμό 1 και τα υπόλοιπα στοιχεία της στήλης πλην του τελευταίου σε 0. Η διαδικασία της οδήγησης επαναλαμβάνεται από την αρχή, μέχρι να προκύψουν μη αρνητικά στοιχεία στη τελευταία γραμμή του πίνακα.

2.5.2 Παράδειγμα επίλυσης προβλήματος ΓΠ με τη μέθοδο Simplex

Η αντικειμενική συνάρτηση $P = 3x + 4y + z$ ζητείται να μεγιστοποιηθεί με τους περιορισμούς

$$3x + 10y + 5z \leq 120$$

$$5x + 2y + 8z \leq 6$$

$$8x + 10y + 3z \leq 105$$

$$x, y \geq 0$$

Αρχικά, πρέπει να μετατραπεί το πρόβλημα στην κανονική του μορφή, ώστε να εφαρμοσθεί η μέθοδος Simplex. Το παράδειγμα όπως φαίνεται από τα δεδομένα, έχει 3 μεταβλητές απόφασης και έτσι δεν μπορεί να επιλυθεί γραφικά. Συνεπώς το πρώτο βήμα για την επίλυση με την μέθοδο Simplex είναι να χρησιμοποιηθούν οι μεταβλητές περιθωρίου (slack variables) σε κάθε έναν από τους περιορισμούς, ώστε να μετατραπούν από ανισώσεις σε εξισώσεις.

$$3x + 10y + 5z + s_1 = 120$$

$$5x + 2y + 8z + s_2 = 6$$

$$8x + 10y + 3z + s_3 = 105$$

Στη συνέχεια θα φέρουμε την αντικειμενική συνάρτηση και τους περιορισμούς στη τυποποιημένη τους μορφή, ώστε να χρησιμοποιηθούν μετά με την μορφή πίνακα simplex.

$$-3x - 4y - z + P = 0$$

x	y	z	$\boldsymbol{s_1}$	$\boldsymbol{s_2}$	$\boldsymbol{s_3}$	P	B
3	10	5	1	0	0	0	120
5	2	8	0	1	0	0	6
8	10	3	0	0	1	0	105
-3	-4	-1	0	0	0	1	0

Πίνακας 2.1 : Αρχικός Πίνακας Simplex

Στον αρχικό πίνακα παρουσιάζονται οι συντελεστές των μεταβλητών των εξισώσεων με την σειρά που έχουν γραφτεί και στη τελευταία γραμμή βρίσκονται οι συντελεστές των μεταβλητών της αντικειμενικής συνάρτησης. Η βέλτιστη λύση θα βρεθεί όταν στη τελευταία γραμμή του πίνακα οι τιμές είναι μη αρνητικές.

Στη συνέχεια ακολουθεί μία σειρά από ενέργειες που θα μας οδηγήσουν στην βέλτιστη λύση:

1ο Βήμα: Εντοπισμός του στοιχείο οδηγού. Για την εύρεση του σημείου οδηγού πρέπει πρώτα να βρεθεί η στήλη οδηγός και στη συνέχεια η γραμμή οδηγός. Αρχικά, για τη στήλη οδηγό, βρίσκουμε στην τελευταία γραμμή τον ελάχιστο αρνητικό αριθμό. Στο παράδειγμα, είναι το -4 και έτσι η στήλη οδηγός είναι η δεύτερη. Η μεταβλητή που αντιστοιχεί στην στήλη οδηγό, ονομάζεται εισερχόμενη μεταβλητή.

x	y	z	s_1	s_2	s_3	P	B
3	10	5	1	0	0	0	120
5	2	8	0	1	0	0	6
8	10	3	0	0	1	0	105
-3	-4	-1	0	0	0	1	0

Πίνακας 2.2 : Μέγιστος αρνητικός αριθμός

2ο Βήμα: Εντοπισμός της γραμμής οδηγού. Για την εύρεση της γραμμής οδηγού θα διαιρεθούν τα στοιχεία της δεύτερης στήλης εκτός της τελευταίας γραμμής, με τους αντίστοιχους αριθμούς της στήλης με τις σταθερές (B). Το αποτέλεσμα με τον μικρότερο αριθμό θα αποτελέσει το στοιχείο οδηγό.

Πιο συγκεκριμένα οι πράξεις είναι:

$$\frac{120}{10} = 12$$

$$\frac{6}{2} = 3$$

$$\frac{105}{10} = 10.$$

Άρα το στοιχείο οδηγός είναι το 2.

x	y	z	s_1	s_2	s_3	P	B
3	10	5	1	0	0	0	120
5	2	8	0	1	0	0	6
8	10	3	0	0	1	0	105
-3	-4	-1	0	0	0	1	0

Πίνακας 2.3 : Στοιχείο οδηγός

3ο Βήμα: Αφού βρέθηκε το στοιχείο οδηγός, τότε ο σκοπός στη συνέχεια είναι να γίνουν αριθμητικές πράξεις ώστε να μετατραπεί το στοιχείο οδηγός στον αριθμό 1 και οι υπόλοιποι αριθμοί της στήλης στον αριθμό 0. Η διαδικασία αυτή, ονομάζεται οδήγηση. Συνεπώς, διαιρείται η δεύτερη γραμμή δια 2.

x	y	Z	s_1	s_2	s_3	P	B
3	10	5	1	0	0	0	120
5/2	1	4	0	**1/2**	0	0	3
8	10	3	0	0	1	0	105
-3	-4	-1	0	0	0	1	0

Πίνακας 2.4 : Μετατροπή στοιχείου οδηγού σε μονάδα

Για τον μηδενισμό των υπόλοιπων στοιχείων της δεύτερης στήλης, ακολουθούν οι παρακάτω πράξεις: Για συντομία, οι σειρές θα αναγράφονται ως R_1 για την πρώτη , R_2 για τη δεύτερη κοκ.

$$R_1 - 10 * R_2$$

$$R_3 - 10 * R_2$$

$$R_4 + 4 * R_2$$

Έτσι ο πίνακας, έχει την εξής μορφή:

x	y	z	s_1	s_2	s_3	P	B
-22	0	-35	1	-5	0	0	90
5/2	**1**	4	0	**1/2**	0	0	3
-17	0	-37	0	-5	1	0	75
7	0	15	0	2	0	1	12

Πίνακας 2.5 : Τελική μορφή πίνακα Simplex

Η διαδικασία, εύρεσης βέλτιστης λύσης έφτασε στο τέλος της, καθώς η τελευταία γραμμή με τους συντελεστές της αντικειμενικής συνάρτησης έχει μη αρνητικές τιμές. Εάν δεν ήταν όλα τα στοιχεία της τελευταίας στήλης μη αρνητικά, η διαδικασία θα επαναλαμβανόταν από την αρχή και αυτό θα συνέβαινε μέχρι να εμφανιστούν μη αρνητικά στοιχεία στη τελευταία γραμμή του πίνακα.

Πιο συγκεκριμένα, η βέλτιστη λύση που έχει το πρόβλημα, βρίσκεται στις τιμές του πίνακα όπου οι υπάρχουν οι βασικές στήλες του περιέχουν μόνο το 0 και το 1 ως στοιχεία του πίνακα, δηλαδή τις βασικές και τις μη-βασικές μεταβλητές. Βασική στήλη είναι η $2^η$, η $6^η$ και η $7^η$ Όπου βρίσκεται το 1, τότε η αντίστοιχη μεταβλητή της στήλης του 1 παίρνει την τιμή που έχει στην αντίστοιχη θέση του B. Δηλαδή:

Αυτό συμβαίνει μόνο στις στήλες y, s_1, s_3 και P. Οι υπόλοιπες μεταβλητές θα έχουν την τιμή 0.

Άρα οι τιμές της βέλτιστης λύσης που θα έχουν είναι :

y = 3 x = 0

$s_1 = 90$ z = 0

$s_3 = 75$ $s_2 = 0$

Η αντικειμενική συνάρτηση θα έχει τιμή, P = 12.

2.5.3 Άλλοι Αλγόριθμοι Γραμμικού Προγραμματισμού

Εκτός από τη μέθοδο Simplex, η οποία παραμένει η πιο γνωστή και ευρέως χρησιμοποιούμενη μέθοδος για την επίλυση γραμμικών προβλημάτων, υπάρχουν και άλλες αλγοριθμικές προσεγγίσεις που έχουν αναπτυχθεί για την επίλυση προβλημάτων γραμμικού προγραμματισμού. Δύο από τις σημαντικότερες μεθόδους που αξίζει να αναφερθούν είναι η μέθοδος Εσωτερικών Σημείων και η Αναθεωρημένη Μέθοδος Simplex.

Μέθοδος Εσωτερικών Σημείων

Η μέθοδος Εσωτερικών Σημείων, εισήχθη το 1979 από τον Khachian και αποτέλεσε τον πρώτο αλγόριθμο πολυωνυμικού χρόνου για τη λύση προβλημάτων γραμμικού προγραμματισμού. Σε αντίθεση με τη μέθοδο Simplex, η οποία εργάζεται με τους κορυφές του πολυεδρικού χώρου εφικτών λύσεων, η μέθοδος Εσωτερικών Σημείων εξετάζει σημεία εντός του χώρου αυτού και προχωρά προς την κατεύθυνση της βελτίωσης της αντικειμενικής συνάρτησης. Παρά την αρχική του θεωρητική επιτυχία, η απόδοση του ελλειψοειδούς αλγορίθμου του Khachian ήταν περιορισμένη για πρακτικά προβλήματα. Ωστόσο, το 1984, ο Karmakar παρουσίασε μια βελτιωμένη εκδοχή της μεθόδου Εσωτερικών Σημείων, που έχει αποδειχθεί ιδιαίτερα αποτελεσματική για μεγάλης κλίμακας προβλήματα, ξεπερνώντας τη μέθοδο Simplex σε τέτοιες περιπτώσεις. Οι σύγχρονοι αλγόριθμοι Εσωτερικών Σημείων μπορούν να χειριστούν προβλήματα με χιλιάδες μεταβλητές και περιορισμούς πιο αποτελεσματικά σε σχέση με τη μέθοδο Simplex για μεγάλα συστήματα.

Αναθεωρημένη Μέθοδος Simplex

Η Αναθεωρημένη Μέθοδος Simplex είναι μια βελτιωμένη εκδοχή της κλασικής μεθόδου Simplex, η οποία αναπτύχθηκε για να μειώσει την υπολογιστική πολυπλοκότητα και να ενισχύσει την αποδοτικότητα της επίλυσης μεγάλων προβλημάτων. Η Αναθεωρημένη Μέθοδος Simplex εστιάζει στην επίλυση των προβλημάτων γραμμικού προγραμματισμού μέσω της αναθεώρησης της λύσης μόνο των δραστικών μεταβλητών, ενώ η κλασική μέθοδος εξετάζει ολόκληρο το χώρο των μεταβλητών. Αυτό επιτρέπει την αποτελεσματικότερη διαχείριση της υπολογιστικής προσπάθειας και την επίλυση προβλημάτων με λιγότερους υπολογιστικούς πόρους. Παρόλο που η Αναθεωρημένη Μέθοδος Simplex δεν προσφέρει τα ίδια πλεονεκτήματα για μεγάλα προβλήματα σε σύγκριση με τη μέθοδο Εσωτερικών Σημείων, παραμένει ισχυρή για προβλήματα μέσου μεγέθους και είναι εύκολη στην εφαρμογή και τη διαχείριση αρχικών σημείων.

2.6 Περιγραφή Αναθεωρημένης μεθόδου Simplex

Η αναθεωρημένη μέθοδος Simplex χρειάζεται συγκεκριμένες δομές δεδομένων για να υλοποιηθεί. Υλοποιεί τον ίδιο αλγόριθμο σε όρους γραμμικής άλγεβρας. Η μέθοδος θα παρουσιαστεί παρακάτω με τη μορφή πινάκων.

Αρχικά, απαιτείται ένας πίνακας B (Basis Matrix), ο οποίος περιέχει τις στήλες του αρχικού πίνακα A που αντιστοιχούν στις βασικές μεταβλητές X_B της τρέχουσας λύσης. Οι μη βασικές μεταβλητές είναι αυτές που δεν βρίσκονται στη βάση. Περιλαμβάνεται επίσης το διάνυσμα κόστους (C_B), δηλαδή οι συντελεστές της αντικειμενικής συνάρτησης που αντιστοιχούν στις βασικές μεταβλητές. Η αντίστροφη του πίνακα B^{-1} χρησιμοποιείται για την ενημέρωση της λύσης και του κόστους.

Η διαδικασία υλοποίησης του αλγορίθμου ξεκινά με την εύρεση μιας εφικτής λύσης. Ορίζονται οι αρχικές βασικές και μη βασικές μεταβλητές και υπολογίζεται ο αντίστροφος πίνακας B^{-1}. Στη συνέχεια, συνέχεια, υπολογίζεται το μειωμένο κόστος $\mathrm{C_N} - \mathrm{C_B^T} * \mathrm{B^{-1}} * \mathrm{A_N}$, όπου A_N είναι οι στήλες του πίνακα A που αντιστοιχούν στις μη βασικές μεταβλητές.
Αν όλα τα μειωμένα κόστη είναι μη αρνητικά, τότε έχει βρεθεί η βέλτιστη λύση και ο αλγόριθμος τερματίζεται. Για την επιλογή της εισερχόμενης μεταβλητής, επιλέγεται αυτή με το πιο αρνητικό μειωμένο κόστος, ώστε η αντικειμενική συνάρτηση να αυξηθεί στο μέγιστο βαθμό. Στη συνέχεια, υπολογίζεται η στήλη του πίνακα $B^{-1}\ A_j$, όπου A_j είναι η στήλη του πίνακα A που αντιστοιχεί στην εισερχόμενη μεταβλητή.

Για τον υπολογισμό της εξερχόμενης μεταβλητής, υπολογίζεται ο λόγος $\frac{X_{B_i}}{(B^{-1}A_j)}$ για όλες τις θετικές

τιμές του $B^{-1}\ A_j$ και επιλέγεται ο ελάχιστος λόγος. Αυτή είναι η κλασική διαδικασία του κανόνα της αναλογίας (ratio test).

Τέλος, ενημερώνεται ο βασικός πίνακας αντικαθιστώντας τη στήλη της εξερχόμενης μεταβλητής με τη στήλη της εισερχόμενης μεταβλητής και επαναυπολογίζεται ο B^{-1} για την τελική μορφή του βασικού πίνακα. Η διαδικασία αυτή επαναλαμβάνεται μέχρι να μην υπάρχουν αρνητικά μειωμένα κόστη.

2.6.1 Αλγόριθμος Αναθεωρημένου Simplex

Input: Matrix $\mathbf{A}$, vectors $\mathbf{b}$ and $\mathbf{c}$. Problem in canonical augmented form.
Output: Optimal solution or unbounded problem message

```
1  /* Initialize data (Range assignements with Matlab-like notation). */
2  B[m][m] ← Initialize(A);
3  c_B[m] ← c[n − m : n − 1];
4  x_B[m] ← 0;
5  Optimum ← ⊥;
6  while !Optimum do
7      /* Determine the entering variable */
8      Index p ← {j|c̃_j == min_t (c_B B^-1 A_t − c_t)};
9      x_B ← B^-1 b;
10     if c̃_p ≥ 0 then
11         Optimum ← ⊤;
12         break;
13     /* Determine the leaving variable */
14     α ← B^-1 A_p;
15     Index q ← {j|θ_j == min_t (x_Bt / α_t, α_t > 0)};
16     if α ≤ 0 then
17         exit("Problem unbounded");
18         break;
19     /*Update the basis */
20     B ← UpdateBasis(A, p, q);
21     /* Update basis cost */
22     c_Bq ← c_p;
23     /* Update basis solution */
24     x_B ← B^-1 b;
25 if Optimum then
26     exit(x_B, z ← x_B c_B);
```

Σχήμα 2.9 : Αλγόριθμος revised simplex [6]

Κεφάλαιο 3ο: GPU & CUDA

3.1 Εισαγωγή

Ο στόχος αυτής της διπλωματικής εργασίας, είναι, αφού παρουσιάσει και περιγράψει την μέθοδο Simplex, να προσπαθήσει να επιταχύνει τον αλγόριθμο με την εκτέλεση του μέσω κάρτας γραφικών με τη βοήθεια πακέτων της Python και να παρουσιάσει τα αποτελέσματα αλλά και τα συμπεράσματα αυτής της διαδικασίας. Η διαδικασία αυτή βέβαια, περιλαμβάνει βήματα τα οποία δεν είναι τόσο συνηθισμένα, για παράδειγμα, υπάρχουν έννοιες και όροι που είναι ανάγκη να παρουσιαστούν και να αναλυθούν, ώστε ο αναγνώστης όταν έρθει σε επαφή με την εκτέλεση του αλγορίθμου να κατανοεί έννοιές όπως ο παράλληλος προγραμματισμός, η εκτέλεση ενός προγράμματος μέσω GPU (Graphics Processing Unit) και όχι CPU (Central Processing Unit) όπως συνηθίζεται κ.α. που θα αναλυθούν στη συνέχεια. Αρχικά θα γίνει αναφορά στην χρησιμότητα και την εξέλιξη των καρτών γραφικών σε τομείς που δεν έχουν αμιγώς σχέση με γραφικά, αλλά περισσότερο με υπολογισμούς.

Η ιστορία των καρτών γραφικών αρχίζει το 1981 και έκτοτε έχουν εξελιχθεί σε τέτοιο βαθμό, όπου συμβάλλουν σε τομείς όπως η ψυχαγωγία με την συνεισφορά τους σε video games, στον εργασιακό τομέα με συνεισφορά σε επαγγέλματα όπως η γραφιστική αλλά και στο ερευνητικό κομμάτι, καθώς πολλοί επιστήμονες χρησιμοποιούν πλέον κάρτες γραφικών για να βελτιστοποιήσουν τις εφαρμογές τους, ενώ αρχικά αναπτύχθηκαν για σχεδίαση με την βοήθεια υπολογιστή (CAD) και για προσομοιώσεις πτήσεων [6]. Είναι σημαντικό να αναφερθεί, ότι οι κάρτες γραφικών δεν χρησιμοποιούνται αποκλειστικά για εφαρμογές που έχουν σχέση με γραφικά. Αυτή η χρήση της κάρτας γραφικών, ονομάζεται GPGPU (General-purpose computing on graphics processing units) δηλαδή γενικού σκοπού χρήση της GPU και αφορά συνήθως επιστημονικές και ερευνητικές εφαρμογές. Με τον όρο γενικού σκοπού όμως δεν εννοείται, ότι μία GPU, είναι κατάλληλη να υποστηρίξει οποιασδήποτε μορφής εφαρμογή αλλά ούτε και σχεδιάζονται με σκοπό να αποτελέσουν μια εναλλακτική ενός γενικού σκοπού επεξεργαστή. Μια εφαρμογή για να εκμεταλλευτεί πλήρως τα πλεονεκτήματα μίας κάρτας γραφικών, πρέπει να έχει κάποια δεδομένα χαρακτηριστικά, όπως οι υψηλές υπολογιστικές απαιτήσεις, μεγάλο βαθμό παραλληλισμού, όπου η διεκπεραίωσή (throughput), δηλαδή η επιτυχής προσπέλαση μεγάλου όγκου δεδομένων ταυτόχρονα είναι πιο σημαντική από την ταχύτητα εκτέλεσης διεργασιών (latency) [6].

Για την πλήρη κατανόηση αυτών των όρων, είναι σημαντικό να αναλυθούν στις επόμενες υποενότητες, μερικές έννοιες που αφορούν τις κάρτες γραφικών, τους επεξεργαστές, την αξία της χρήσης μίας κάρτας γραφικών για προγραμματισμό εφαρμογών, ενώ θα γίνει αναφορά και στον προγραμματισμό γενικού σκοπού με GPU.

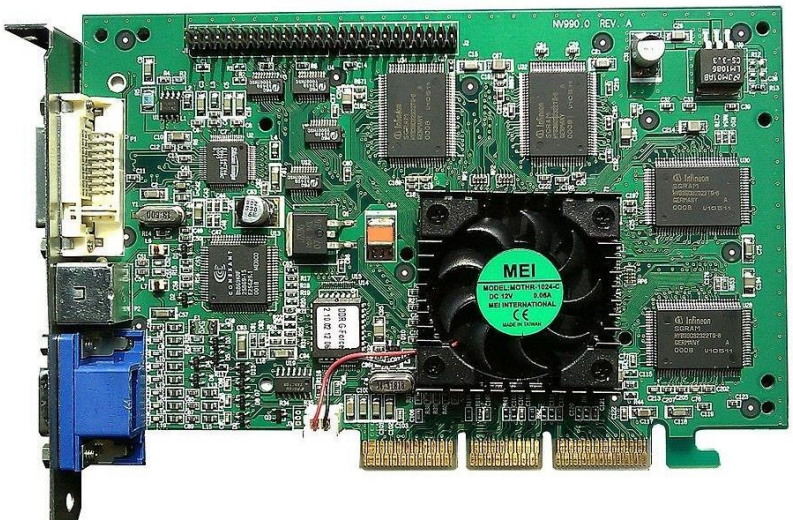
3.2 Εξέλιξη Επεξεργαστών

Οι πρώτοι επεξεργαστές (CPU) και πιο συγκεκριμένα μικροεπεξεργαστές παρατηρούνται εδώ και τουλάχιστον 50 χρόνια. Τη δεκαετία του 1970-80 δημιουργήθηκαν οι πρώτοι μικροεπεξεργαστές και οι πρώτοι προσωπικοί υπολογιστές είχαν ταχύτητες μεγέθους 1 MHz. Στη συνέχεια η ταχύτητα αυξήθηκε σε τιμές από 1 GHz έως 4 GHz, ενώ σήμερα οι ταχύτητες είναι της τάξεως των GFLOPS, δηλαδή σε 1 εκατομμύριο πράξεις κινητής υποδιαστολής ανά δευτερόλεπτο. Η ανάγκη για αύξηση των ταχυτήτων οδηγήθηκε από τις αυξήσεις των επιδόσεων που επιζητούσαν οι χρήστες. Αρχικά ο τρόπος με τον οποίο αυξανόταν η ταχύτητα του επεξεργαστή, γινόταν με την αύξηση της συχνότητας του ρολογιού του. Οι επιδόσεις των μονοπύρηνων τότε επεξεργαστών αυξανόντουσαν και ο νόμος του Moore με το πέρασμα των χρόνων επιβεβαιωνόταν. Ο νόμος ονομάζεται έτσι από έναν εκ των ιδρυτών της εταιρείας μικροεπεξεργαστών Intel, τον Gordon Moore. Το 1965 δήλωσε ότι ο αριθμός των τρανζίστορ ενός πυκνού ολοκληρωμένου κυκλώματος θα διπλασιάζεται ανά δύο χρόνια. Αυτός ο νόμος ίσχυε για πάνω από 50 χρόνια και έφτασε περίπου στο τέλος του καθώς αυξανόταν η θερμοκρασία αλλά και η κατανάλωση ενέργειας των επεξεργαστών. Η Intel είχε προβλέψει το 2003 ότι το τέλος του θα έλθει το διάστημα 2013 με 2018. Έτσι ο παράλληλος υπολογισμός, όπου σήμερα κυριαρχεί, αντικαθιστά πλήρως τον νόμο του Moore. Ο παραλληλισμός δεν είναι κάτι καινούργιο καθώς παρατηρείται από τη δεκαετία του 1950 και αργότερα τις δεκαετίες του 60 και του 70 υπήρχαν οι λεγόμενοι Super Computers («Υπερ-Υπολογιστές»), όπου χρησιμοποιούσαν πολλαπλούς επεξεργαστές και παράλληλο λογισμικό για την επίτευξη μεγαλύτερων ταχυτήτων [10]. Η εξέλιξη των επεξεργαστών και ο παράλληλος προγραμματισμός- υπολογισμός είναι έννοιες που συμβαδίζουν. Με το τέλος του νόμου του Moore, η βιομηχανία παραγωγής υπολογιστικών συστημάτων, άλλαξε ριζικά το 2005 όταν η Intel, προσπέρασε την IBM και την Sun Microsystems, ανακοινώνοντας ότι οι υψηλής απόδοσης επεξεργαστές της θα βασίζονται σε πολλαπλούς πυρήνες [10]. Οι μονοπύρηνοι επεξεργαστές αντικαταστάθηκαν με περισσότερους πυρήνες. Οι πολλοί πυρήνες αυξάνουν ουσιαστικά τις επιδόσεις του συστήματος καθώς ο κάθε πυρήνας αποτελεί έναν ξεχωριστό στοιχείο επεξεργαστή και ο κάθε ένας από αυτούς είναι σαν ένας CPU όπως στους παλιούς μονοπύρηνους υπολογιστές [10]. Οι κοινοί επεξεργαστές σήμερα έχουν από 4 έως και 16 πυρήνες ενώ αναμένεται ο αριθμός να αυξηθεί κατακόρυφα. Οι πολυπύρηνοι επεξεργαστές χρησιμοποιούνται για διάφορες εφαρμογές, όπως γενικού σκοπού, σε δίκτυα αλλά και σε ενσωματωμένα συστήματα καθώς και σε επιστημονικές εφαρμογές.

Με την εξέλιξη αυτή της αρχιτεκτονικής των επεξεργαστών, επηρεάστηκε και η κοινότητα των προγραμματιστών όπου έπρεπε να αλλάξουν τον τρόπο που έγραφαν τον κώδικα τους που εκτελούταν σειριακά και έτσι δεν μπορούσαν να εκμεταλλευτούν τους πολλούς πυρήνες καθώς η σειριακή μορφή του προγράμματος θα εκτελούνταν σε έναν πυρήνα [9]. Από την σειριακή τους μορφή, έπρεπε να γραφτούν με παράλληλο τρόπο, καθώς η ύπαρξη πολλών νημάτων συμμετείχε στην εκτέλεση μιας διεργασίας και έτσι επιταχυνόταν το πρόγραμμα [9].

3.3 GPU

Η απόδοση γραφικών είναι μια διαδικασία, η οποία είναι αρκετά έντονη για τον υπολογιστή και πολλές φορές ανάλογα με την χρήση που γίνεται, όπως για παράδειγμα, η επεξεργασία εικόνας-ήχου, ή βίντεο-παιχνιδιών, μπορεί να επιβαρύνουν αρκετά την απόδοση του υπολογιστή, αν υπάρχει μόνο ο CPU στον υπολογιστή. Η δημιουργία των καρτών γραφικών, καλύπτει ουσιαστικά τέτοιου είδους ανάγκες, όπου οι απαιτήσεις είναι μεγάλες και προϋποθέτουν την ικανότητα για πολλές πράξεις κινητής υποδιαστολής (floating-point), ώστε να δημιουργηθούν τα όποια τρισδιάστατα γραφικά (3D) ή και σχεδίαση δισδιάστατων εικόνων (2D) [10]. Ουσιαστικά ο όρος γραφικών, αποδόθηκε από την Nvidia, το 1999, όπου κυκλοφόρησε στην αγορά η πρώτη κάρτα γραφικών, το μοντέλο GeForce 256 [10]. Σήμερα οι κάρτες γραφικών εμπεριέχονται σε υπολογιστές, κινητά και tablets.



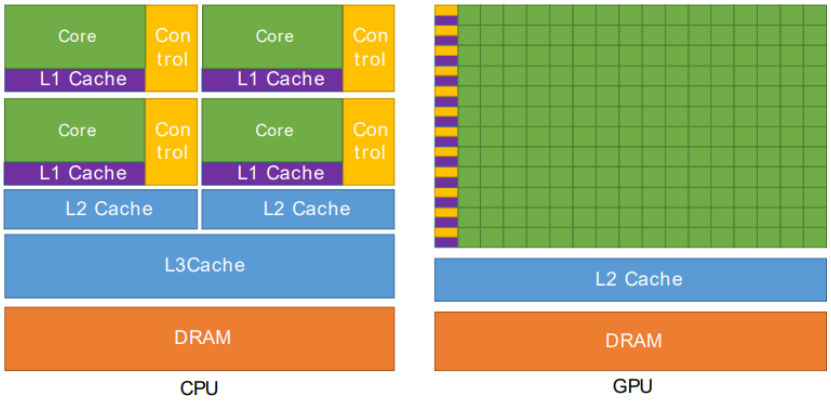
Σχήμα 3.1: Η πρώτη κάρτα γραφικών «NVIDIA GeForce 256»

Οι επεξεργαστές των καρτών γραφικών, είναι πολυπύρηνοι, υψηλού βαθμού παραλληλισμού και περιέχουν πολλές χιλιάδες νήματα που τρέχουν ταυτόχρονα για την εκτέλεση μιας διεργασίας [6]. Όπως αναφέρθηκε και προηγουμένως, οι κάρτες γραφικών δεν χρησιμοποιούνται αποκλειστικά για γραφικά, αλλά και για προγραμματισμό επιστημονικών εφαρμογών, καθώς πολλοί επιστήμονες από διάφορους κλάδους ξεκίνησαν στα τέλη της δεκαετίας του 1990 να τις χρησιμοποιούν. Ήταν η απαρχή για τον λεγόμενο σήμερα προγραμματισμό γενικού σκοπού με χρήση καρτών γραφικών (GPGPU).

3.3.1 Λειτουργία CPU & GPU

Η διαφορά στις δυνατότητες μεταξύ των επεξεργαστών CPU και GPU, βρίσκεται στον σκοπό για τον οποίο σχεδιάζονται. Ο επεξεργαστής CPU είναι σχεδιασμένος για να εκτελεί σειριακά προγράμματα, όπου τα νήματα να εκτελούνται είτε σειριακά το καθένα, είτε μερικές δεκάδες παράλληλα. Ο επεξεργαστής της κάρτας γραφικών, είναι σχεδιασμένος για να εκτελεί χιλιάδες νήματα παράλληλα όπου με αυτόν τον τρόπο κερδίζει σε ταχύτητα μεταφοράς δεδομένων σε σχέση με την σειριακή εκτέλεση των νημάτων.

Η κάρτα γραφικών προσφέρει πολύ καλύτερη και μεγαλύτερη μεταφορά δεδομένων αλλά και καλύτερο εύρος ζώνης από τον επεξεργαστή CPU στην ίδια τιμή και ικανότητα ισχύος. Αρκετές εφαρμογές εκμεταλλεύονται αυτές τις υψηλότερες δυνατότητες για να τρέχουν στην κάρτα γραφικών, παρά στον επεξεργαστή. Στον επεξεργαστή GPU, τα περισσότερα τρανζίστορ αφιερώνονται στο data processing, δηλαδή στους υπολογισμούς με δεδομένα (πράξεις κινητής υποδιαστολής) και έτσι ωφελούνται περισσότερο οι υψηλού βαθμού παραλληλισμού πράξεις. Ο επεξεργαστής GPU, μπορεί να «κρύψει» τις καθυστερήσεις στις προσπελάσεις μνήμης εκτελώντας ταχύτερα πράξεις, αντί να βασίζεται σε μεγάλες μνήμες cache, δηλαδή δεδομένα προσωρινής αποθήκευσης και περίπλοκους ελέγχους ροής για να μειωθούν οι καθυστερήσεις σε προσπελάσεις μνήμης, όπως γίνεται στην λειτουργία ενός επεξεργαστή CPU. Ο έλεγχος ροής και η μνήμη cache, αντιθέτως, δεν συνεισφέρουν στην ταχύτητα εκτέλεσης πράξεων [9].

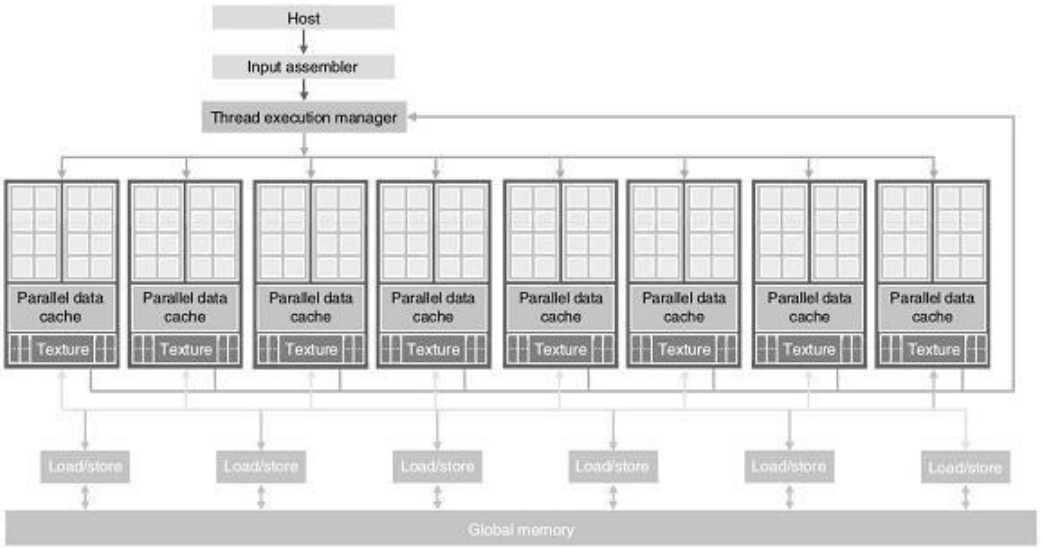


Σχήμα 3.2: Περισσότερα τρανζίστορ της GPU για data processing

Γενικά, μια εφαρμογή έχει χαρακτηριστικά και παραλληλισμού αλλά και σειριακά χαρακτηριστικά, όπως και τα συστήματα είναι σχεδιασμένα με μία μίξη από CPU και GPU ώστε να μεγιστοποιούν την συνολική απόδοση. Στην περίπτωση όμως που, προγράμματα έχουν χαρακτηριστικά όπως ο μεγάλός όγκος δεδομένων και υψηλού βαθμού παραλληλισμού και εκτελεστεί πάνω στον επεξεργαστή CPU, τότε δεν θα μπορέσει το πρόγραμμα να εκμεταλλευτεί τις ταχύτητες της GPU καθώς θα εκτελεστεί στην σειριακή λογική του CPU. Ένα πρόγραμμα σειριακής λογικής θα ήταν κατάλληλο να εκτελεστεί πάνω στον CPU, εάν έχει χαρακτηριστικά έντονων ελέγχων ροής. Μια εφαρμογή για να εκμεταλλευτεί πλήρως τις δυνατότητες αλλά και τις σχεδιαστικές λειτουργίες των CPU και GPU, είναι ανάγκη να εκτελεστεί παράλληλα, δηλαδή το μέρος με τις μεγάλες απαιτήσεις να υπολογιστεί στον επεξεργαστή της κάρτας γραφικών και το σειριακό μέρος στον επεξεργαστή της CPU. Μία από τις πλατφόρμες που υποστηρίζουν το συγκεκριμένο μοντέλο προγραμματισμού είναι το CUDA, το οποίο θα αναλυθεί παρακάτω.

3.3.2 Αρχιτεκτονική CUDA GPU

Μία κάρτα γραφικών η οποία υποστηρίζει το μοντέλο CUDA, έχει σχεδιαστεί ως εξής. Στην παρακάτω εικόνα υπάρχουν 16 πολυεπεξεργαστές συνεχούς ροής (Streaming Multiprocessor) ή SM. Ο κάθε SM έχει 8 επεξεργαστές (Streaming Processor) ή SP, στο σύνολο δηλαδή 128 SP. Κάθε ένας από τους SP έχει από μία μονάδα πολλαπλασιασμού (Multiply Unit) και μία μονάδα πρόσθεσης (Addition Unit). Κάθε SP έχει πολλά νήματα και μπορεί να εκτελέσει χιλιάδες νήματα ανά υπολογισμό. Γενικά ένας πολυεπεξεργαστής συνεχούς ροής μπορεί να διαχειρισθεί, να δημιουργεί και να εκτελεί ταυτόχρονα νήματα σε μηδενικό χρόνο από την κατάσταση ετοιμότητας στην κατάσταση εκτέλεσης του αποστολέα (dispatcher) [6].



Σχήμα 3.3: GPU CUDA Αρχιτεκτονική

3.3.3 Μνήμες GPU

Σε μία GPU υπάρχουν δύο είδη μνήμης. Η μνήμη που βρίσκεται πάνω στο τσιπ του CPU και η μνήμη της συσκευής (DRAM) της GPU. Συχνά η πρώτη αναφέρεται ως on-chip ή host μνήμη, ενώ η δεύτερη ως device μνήμη. Αυτό συμβαίνει ώστε να μην υπάρχουν καθυστερήσεις σε κάθε προσπέλαση στην device μνήμη και έτσι το CUDA έχει σχεδιαστεί ώστε να παρέχει και την on-chip μνήμη και έτσι να μην χρειάζεται κάθε φορά να γίνεται προσπέλαση μόνο στην device μνήμη.

Η device μνήμη είναι μία DRAM όπου είναι πιο αργή σε σχέση με την on-chip μνήμη. Τα σημαντικότερα διαστήματα της μνήμης της βρίσκονται στην global, local, constant και texture περιοχή. Πιο συγκεκριμένα τα διαστήματα global και local μνήμης είναι read-write only, ενώ η constant και texture περιοχές στη μνήμη είναι μόνο read-only και υποστηρίζονται από cache μνήμη

3.4 Παράλληλος Προγραμματισμός

Όπως αναφέρθηκε και προηγουμένως, οι προγραμματιστές για να εκμεταλλευτούν την παράλληλη αρχιτεκτονική, είναι ανάγκη να γράφουν τον κώδικα για τις εφαρμογές τους, ο οποίος πρέπει να είναι διαχειρίσιμος και να εκτελείται ταυτόχρονα σε πολλούς πυρήνες. Ο παράλληλος προγραμματισμός χρησιμοποιείται αρκετά και ειδικά στον τομέα του HPC (High Performance Computing), αλλά τα τελευταία χρόνια είναι αρκετά δημοφιλής και χρησιμοποιείται σε διάφορους τομείς όπως στην μηχανική μάθηση αλλά και σε διάφορες επιστημονικές εφαρμογές.

Οι αλγόριθμοι γράφονται από τους προγραμματιστές σε κατάλληλα διαμορφωμένα λογισμικά, όπου με την βοήθεια διαφόρων πακέτων και βιβλιοθηκών, καθιστούν την υλοποίηση του προγράμματος ευκολότερη και περισσότερο αποδοτική [11]. Τα παράλληλα προγράμματα είναι δυσκολότερο να γραφτούν αλλά και να διορθωθούν (debug), από τα σειριακά προγράμματα. Οι δυσκολίες στο debugging βρίσκονται κυρίως σε προβλήματα όπως του race condition, δηλαδή όταν το σύστημα προσπαθεί να εκτελέσει δύο ή περισσότερα τμήματα κώδικα ή ακόμα και διάφορες άλλες λειτουργίες, την ίδια στιγμή. Επίσης, η επικοινωνία αλλά και ο συγχρονισμός διαφορετικών διεργασιών είναι ζητήματα που είναι πιθανό να επηρεάσουν την απόδοση του προγράμματος [10]. Ένας αλγόριθμος για να εκμεταλλευτεί τις δυνατότητες της παράλληλης αρχιτεκτονικής του συστήματος, με την βοήθεια και ενός λογισμικού, είναι σημαντικό να επιτευχθούν δύο προ απαιτούμενα:

- Πρώτον, η αποδοτική συνεργασία μεταξύ των επεξεργαστών

- Δεύτερον, η βέλτιστη διαχείριση των διαθέσιμων πόρων [11].

Μία θεωρητική αξιολόγηση της απόδοσης του προγράμματος, γίνεται με τον νόμο του Άμνταλ (Amdahl's law). Η μέγιστη θεωρητική ταχύτητα ενός προγράμματος ισούται με :

$$\frac{1}{(1-P)+\frac{P}{N}}$$

Εάν το P, αφορά την ποσότητα του προγράμματος που μπορεί να εκτελεστεί παράλληλα και το 1-P αφορά την ποσότητα του προγράμματος που δεν εκτελείται παράλληλα, τότε η μέγιστη θεωρητική ταχύτητα χρησιμοποιώντας N επεξεργαστές, ισούται με τον παραπάνω τύπο.

Η αξιολόγηση ενός αλγορίθμου που εκτελείται παράλληλα, είναι χρήσιμο να γίνεται χρησιμοποιώντας τον νόμο του Amdahl, όσον αφορά ορισμένες δυνατότητες του προγράμματος, αλλά σε πραγματικές εφαρμογές, πρακτικά δεν είναι δυνατό να υπάρξει ολοκληρωμένη εικόνα. Η απόδοση ενός προγράμματος μπορεί να αξιολογηθεί από παράγοντες όπως το εύρος ζώνης δικτύου, σε περιπτώσεις κατανεμημένης μνήμης αλλά και το φορτίο επεξεργαστών σε περιβάλλοντα πολλών χρηστών [11].

Στην επόμενη ενότητα, θα γίνει εκτενής αναφορά στο λογισμικό παράλληλου προγραμματισμού CUDA.

3.5 CUDA: Πλατφόρμα παράλληλου υπολογισμού & Προγραμματιστικό μοντέλο

Τον Νοέμβριο του 2006, η NVIDIA παρουσίασε το CUDA, μια πλατφόρμα παράλληλου υπολογισμού γενικού σκοπού και προγραμματιστικό μοντέλο, όπου εκμεταλλεύεται την παράλληλη υπολογιστική δύναμη των καρτών γραφικών της NVIDIA, ώστε να επιλύσει αρκετά υπολογιστικά προβλήματα με αποδοτικότερο τρόπο από έναν επεξεργαστή CPU.

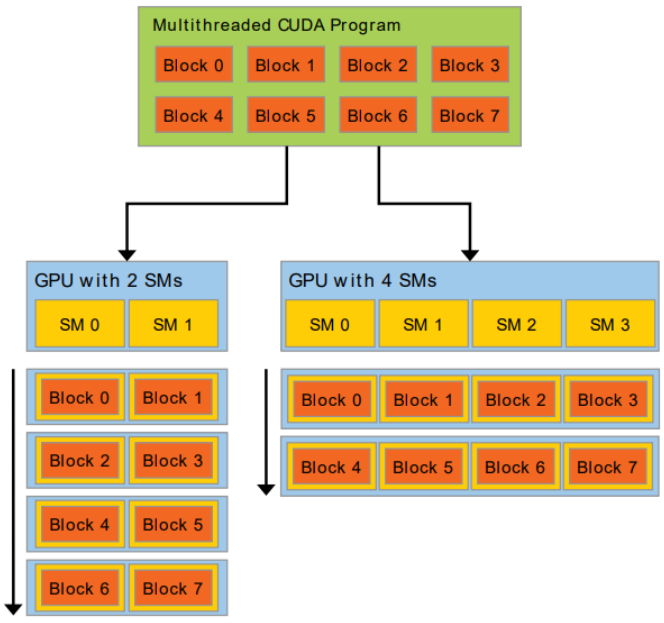
Το CUDA παρέχει ένα προγραμματιστικό περιβάλλον όπου μπορούν οι προγραμματιστές να χρησιμοποιήσουν την C++ ως γλώσσα υψηλού επιπέδου προγραμματισμού. Επίσης, μπορούν να χρησιμοποιηθούν και άλλες γλώσσες προγραμματισμού εφαρμογών και βιβλιοθήκες τους, όπως η C, Fortran, Java, Python κτλ.

Με την έλευση των πολυπύρηνων επεξεργαστών και των πολλών πυρήνων επεξεργαστών σε κάρτες γραφικών, οι επεξεργαστές ενός μέσου χρήστη θεωρούνται παράλληλα συστήματα. Το CUDA υποστηρίζει τους προγραμματιστές ώστε να υλοποιηθούν εφαρμογές όπου εκμεταλλεύονται στο έπακρο τους πολλούς πυρήνες του επεξεργαστή, αλλά και 3D εφαρμογές που θα εκμεταλλεύονται τους πυρήνες του επεξεργαστή της κάρτας γραφικών.

Τα κύρια χαρακτηριστικά που το διέπουν είναι, η ιεραρχία των νημάτων σε groups (ομάδες), οι κοινές μνήμες (shared memories), ο συγχρονισμός των νημάτων (synchronization barrier) και ο προγραμματιστής πρέπει να τα λαμβάνει όλα αυτά υπόψιν. Αυτά τα χαρακτηριστικά, οδηγούν τον προγραμματιστή να διαχωρίζει τα προβλήματα σε κατά προσέγγιση επίλυση σε ξεχωριστά blocks (κομμάτια) νημάτων και το κάθε υπό-πρόβλημα σε μικρότερα μέρη, μπορεί να επιλυθεί παράλληλα από όλα τα νήματα ανάμεσα στο block.

Αυτή η αποσύνθεση του προβλήματος, διατηρεί την εκφραστικότητα της γλώσσας που χρησιμοποιείται, καθώς επιτρέπει στα νήματα να συνεργαστούν όταν επιλύουν το κάθε υποπρόβλημα και ταυτόχρονα επιτρέπει την αυτόματη επεκτασιμότητα. Κάθε block νημάτων μπορεί να διαχειρίζεται από οποιοδήποτε διαθέσιμο πολυεπεξεργαστή της κάρτας γραφικών, με οποιαδήποτε σειρά εκτέλεσης, είτε παράλληλα είτε σειριακά, ώστε το μεταγλωττισμένο CUDA πρόγραμμα να εκτελεστεί σε όσους διαθέσιμους πολυεπεξεργαστές της κάρτας γραφικών του συστήματος, όπως φαίνεται και στο σχήμα 3.3.

Το επεκτάσιμο αυτό προγραμματιστικό μοντέλο μπορεί να αξιοποιηθεί από διάφορων ειδών καρτών γραφικών της αγοράς. Από κάρτες που χρησιμοποιούνται για υψηλές αποδόσεις και gaming όπως το μοντέλα GeForce αλλά και πιο επαγγελματικής χρήσης όπως τα μοντέλα Quadro και Hopper [17].



Σχήμα 3.4 : Αυτόματη επεκτασιμότητα

3.5.1 Kernels

Στο CUDA, ο προγραμματιστής μπορεί να ορίσει συναρτήσεις που ονομάζονται kernels (πυρήνας) και όταν κληθούν θα εκτελεστούν τόσες φορές, όσο το πλήθος των νημάτων του επεξεργαστή της κάρτας γραφικών. Το μοντέλο εκτέλεσης υπολογισμών, ονομάζεται Single Instruction Multiple Thread (SIMT), δηλαδή, αρκετά νήματα εκτελούν την ίδια ροή σε διαφορετικά δεδομένα.

Το kernel ουσιαστικά αποτελεί συνάρτηση και δηλώνεται χρησιμοποιώντας την λέξη (keyword) «_global_» αλλά και δηλώνοντας τον αριθμό των νημάτων που θα εκτελέσουν τον κώδικα της συνάρτησης kernel. Το κάθε νήμα έχει ένα μοναδικό αναγνωριστικό «ID» την μεταβλητή «threadIdx» και βρίσκεται σε ένα συγκεκριμένο block αποτελούμενο από νήματα, με αναγνωριστικό «blockIdx». Το σύνολο των blocks ανήκει ονομάζεται grid. Στο παρακάτω παράδειγμα γίνεται πρόσθεση 2 διανυσμάτων (vectors) όπου κάθε νήμα αναλαμβάνει στοιχεία του πίνακα και στη συνέχεια καλείται και εκτελείται η συνάρτηση [17].

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

Σχήμα 3.5: Πρόσθεση δύο διανυσμάτων A,B και αποθήκευση στο διάνυσμα C.

3.5.2 Ιεραρχία νημάτων

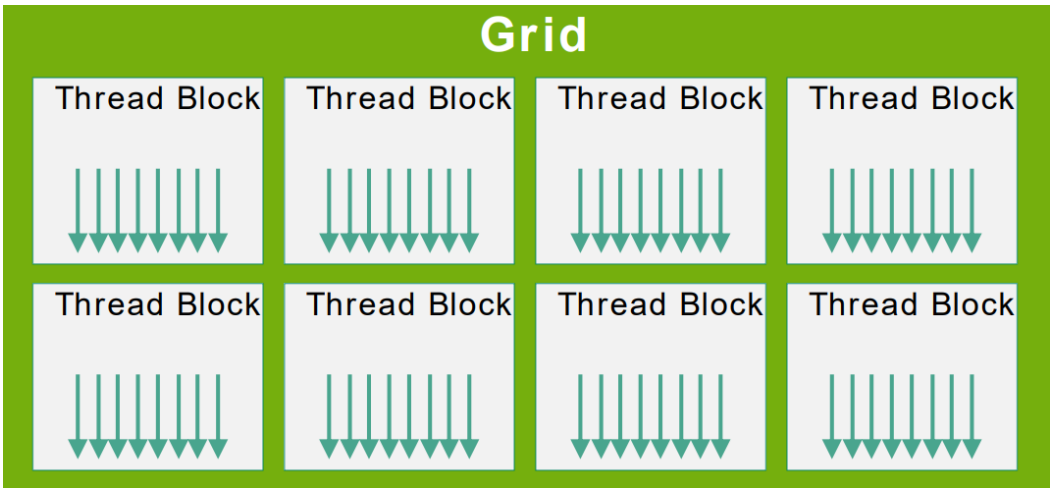
Τα νήματα οργανώνονται μέχρι και σε τριών διαστάσεων blocks, τα οποία ονομάζονται thread blocks. Το «threadIdx» είναι ένα διάνυσμα που ορίζει τον δείκτη των νημάτων ανάλογα με τις διαστάσεις των blocks, δηλαδή οι τιμές που παίρνει είναι είτε μονοδιάστατος, είτε δισδιάστατος είτε τρισδιάστατος.

Ο δείκτης ενός νήματος και το αναγνωριστικό ID του σχετίζονται με τον εξής τρόπο. Για παράδειγμα :

- Για μονοδιάστατο block, το νήμα έχει την ίδια τιμή με το block
- Για δισδιάστατο block (D_x, D_y), το νήμα (x,y) έχει ID το x + yD_x
- Για τρισδιάστατο block (D_x, D_y, D_z), το νήμα (x,y,z) έχει ID το x + yD_x + zD_xD_y

Το πλήθος των νημάτων που μπορεί να υποστηρίξει ένα block, έχει ένα συγκεκριμένο όριο. Όλα τα νήματα ενός block βρίσκονται στον ίδιο πυρήνα του SM (Streaming Multiprocessor) επεξεργαστή συνεχούς ροής και μοιράζονται την ίδια μνήμη. Τα block νημάτων των σημερινών GPU, υποστηρίζουν έως και 1024 νήματα. Ένα kernel είναι δυνατό να εκτελείται από πολλά block νημάτων ίσου μεγέθους, έτσι ώστε ο συνολικός αριθμός των νημάτων να είναι ίσος με τον αριθμό των νημάτων ανά block, επί των αριθμό των blocks [17].

Τα blocks οργανώνονται σε grids ανάλογα με τις διαστάσεις τους, όπως φαίνεται στο παρακάτω σχήμα (3.5)



Σχήμα 3.6: Grid από block νημάτων

Ο αριθμός των νημάτων ανά block και ο αριθμός των block ανά grid, δηλώνονται στους χαρακτήρες <<<…>>> και μπορεί να είναι τύπου int3 ή dim3. Τα δισδιάστατα blocks ή grids προσδιορίζονται στο παρακάτω παράδειγμα.

Κάθε block μέσα στο grid μπορεί να αναγνωρισθεί από το μονοδιάστατο ή δισδιάστατο ή τρισδιάστατο μοναδικό δείκτη που είναι προσβάσιμος στο kernel με την μεταβλητή blockIdx. Η διάσταση του block των νημάτων είναι προσβάσιμη στο kernel μέσω της μεταβλητής blockDim μεταβλητής [17].

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

Σχήμα 3.7: Διαχείριση πολλαπλών block

Ένα block είναι μεγέθους 256 νημάτων (16x16). Το grid, έχει σχεδιαστεί με αρκετά blocks ώστε ένα νήμα να αναλαμβάνει ένα στοιχείο-πίνακα. Για ευκολία, στο συγκεκριμένο παράδειγμα, ο αριθμός των νημάτων ανά grid για κάθε διάσταση είναι ίσος με τον αριθμό των νημάτων ανά block στην ίδια διάσταση.

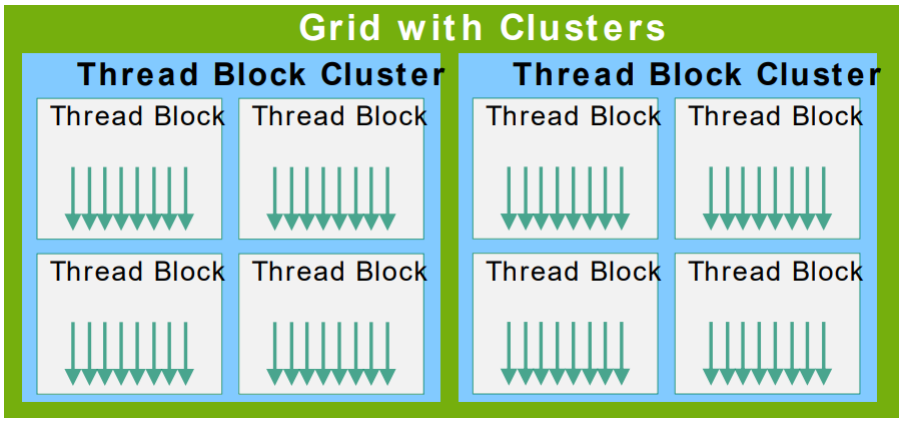
Είναι προαπαιτούμενο τα blocks νημάτων να λειτουργούν ανεξάρτητα. Να μπορούν να εκτελεστούν είτε παράλληλα είτε σειριακά με οποιαδήποτε σειρά οποιωνδήποτε πυρήνων όπως φαίνεται στο σχήμα 3.3, επιτρέποντας στους προγραμματιστές να γράψουν κώδικα, ο οποίος είναι ικανός να τρέχει σε διαφορετικούς πυρήνες, όπου ο αριθμός τους μπορεί να αυξηθεί κατά τη διάρκεια εκτέλεσης του κώδικα.

Τα νήματα μέσα στο block, μπορούν να συνεργαστούν με το να μοιράζονται δεδομένα μέσω της κοινής μνήμης αλλά και από τον συγχρονισμό της εκτέλεσης τους, συντονίζονται οι προσπελάσεις μνήμης. Πιο συγκεκριμένα, με τη μέθοδο «_syncthreads()», μπορούν να προσδιοριστούν τα σημεία συγχρονισμού, όπου λειτουργεί ως φραγμός για τα νήματα μέσα στο block και βρίσκονται σε αναμονήώσπου να επιτραπεί σε κάποιο να εκτελεστεί [17].

Συστάδες block νημάτων

Οι συστάδες των block νημάτων αποτελούν ένα προαιρετικό επίπεδο ιεραρχίας. Όπως και στα block νημάτων, οι συστάδες προγραμματίζονται με οποιαδήποτε σειρά, σε οποιονδήποτε αριθμό πυρήνων. Παρομοίως με τα blocks νημάτων, οι συστάδες οργανώνονται από μία έως τρεις διαστάσεις όπως φαίνεται και στο σχήμα 3.7. Στο CUDA, ο αριθμός των blocks σε μία συστάδα μπορεί να ορισθεί από τον χρήστη και ο μέγιστος αριθμός των block μπορεί να είναι μέχρι και οκτώ. Έχοντας μία συστάδα μεγαλύτερου μεγέθους των 8 block είναι αρχιτεκτονικά συγκεκριμένο και μπορεί να χρησιμοποιηθεί με το query cudaOccupancyMaxPotentialClusterSize API. Ουσιαστικά το API (Application Programming Interface) είναι μια διεπαφή διαχείρισης της κάρτας γραφικών και πιο συγκεκριμένα για την οργάνωση των νημάτων [9]. Μέσω διαφόρων συναρτήσεων που περιέχονται στο πακέτο της CUDA, γίνεται η διαχείριση της κάρτας γραφικών, ενώ είναι ανάγκη να σημειωθεί ότι υπάρχουν δύο κατηγορίες διαχείρισης. Αρχικά το CUDA Driver API, που αποτελεί μία low-level (χαμηλού επιπέδου) διεπαφή, όπου τα καθήκοντα της αφορούν περισσότερο τον έλεγχο της κάρτας γραφικών και διαχειρίζεται δυσκολότερα καθώς η λειτουργία των kernels αλλά και η εκτέλεση και ο ορισμός συναρτήσεων είναι περίπλοκες διαδικασίες. Η δεύτερη κατηγορία είναι το CUDA Runtime API, που αποτελεί high-level

διεπαφή (υψηλού επιπέδου) και διαχειρίζεται ευκολότερα σε σχέση με το Driver API, διότι αφορά διαδικασίες όπως η διασφάλιση ατέρμονης εκτέλεσης του προγράμματος (runtime), διαχείριση περιεχομένου αλλά και των modules.



Σχήμα 3.8: Grid από συστάδες.

Μία συστάδα από blocks μπορεί να ενεργοποιηθεί σε ένα kernel, είτε χρησιμοποιώντας το attribute του kernel που θα μεταγλωττιστεί: «_cluster_dims_(X,Y,Z)», είτε χρησιμοποιώντας το API του CUDA, cudaLaunchKernelEx. Παρακάτω δύο εικόνες με τους δύο διαφορετικούς τρόπους ενεργοποίησης συστάδων σε ένα kernel [17].

```
// Kernel definition
// Compile time cluster size 2 in X-dimension and 1 in Y and Z dimension
__global__ void __cluster_dims__(2, 1, 1) cluster_kernel(float *input, float*
 output)
{

}

int main()
{
    float *input, *output;
    // Kernel invocation with compile time cluster size
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);

    // The grid dimension is not affected by cluster launch, and is still enumerated
    // using number of blocks.
    // The grid dimension must be a multiple of cluster size.
    cluster_kernel<<<numBlocks, threadsPerBlock>>>(input, output);
}
```

Σχήμα 3.9: Ενεργοποίηση cluster, με το χαρακτηριστικό << , >>

```
// Kernel definition
// No compile time attribute attached to the kernel
__global__ void cluster_kernel(float *input, float* output)
{

}

int main()
{
    float *input, *output;
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    cluster_kernel<<<numBlocks, threadsPerBlock>>>();
    // Kernel invocation with runtime cluster size
    {
        cudaLaunchConfig_t config = {0};
        // The grid dimension is not affected by cluster launch, and is still
 enumerated
        // using number of blocks.
        // The grid dimension should be a multiple of cluster size.
        config.gridDim = numBlocks;
        config.blockDim = threadsPerBlock;

        cudaLaunchAttribute attribute[1];
        attribute[0].id = cudaLaunchAttributeClusterDimension;
        attribute[0].val.clusterDim.x = 2; // Cluster size in X-dimension
        attribute[0].val.clusterDim.y = 1;
        attribute[0].val.clusterDim.z = 1;
        config.attrs = attribute;
        config.numAttrs = 1;

        cudaLaunchKernelEx(&config, cluster_kernel, input, output);
    }
}
```

Σχήμα 3.10: Ενεργοποίηση cluster, με χρήση του cudaLaunchKernelEx API

3.5.3 Ιεραρχία μνήμης

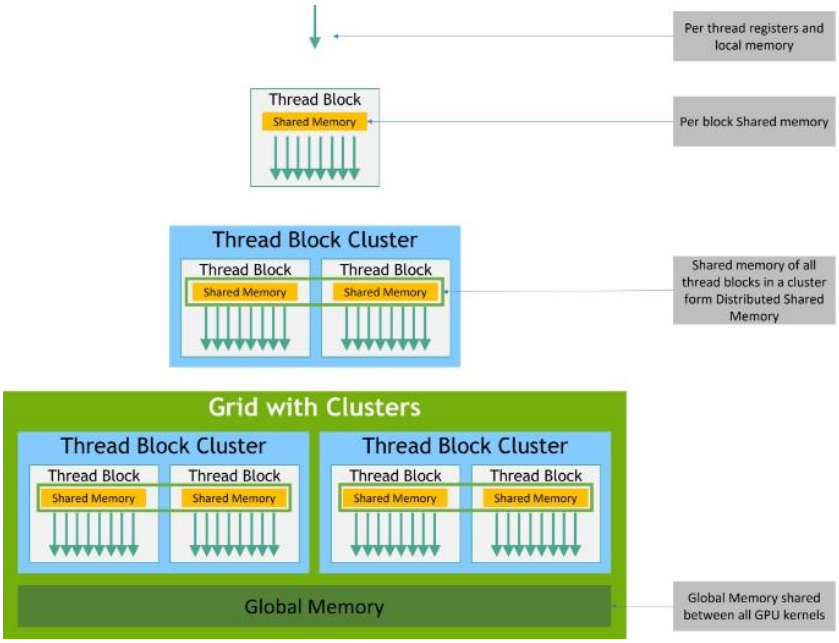
Τα νήματα του CUDA κατά την εκτέλεση τους, έχουν πρόσβαση στα δεδομένα από διάφορες μνήμες του συστήματος, όπως φαίνεται και στο παρακάτω σχήμα (3.10). Κάθε νήμα έχει την δική του ιδιωτική μνήμη. Κάθε block νημάτων έχει κοινή μνήμη, ορατή σε όλα τα νήματα του block με την ίδια διάρκεια ζωής όσο έχει και το block. Τα block νημάτων σε μία συστάδα από blocks έχουν δυνατότητες όπως το διάβασμα, εγγραφή και ατομικές λειτουργίες στην κοινή μνήμη των υπόλοιπων blocks της συστάδας, ενώ όλα τα νήματα έχουν πρόσβαση στην ίδια global μνήμη. Η χρήση της global μνήμης αφορά την αποθήκευση δεδομένων.

Επιπλέον υπάρχουν άλλες δύο read-only μνήμες (δυνατότητα διαβάσματος) που έχουν πρόσβαση τα νήματα. Η global, η constant και η texture μνήμες έχουν βελτιστοποιηθεί για συγκεκριμένη χρήση. Η constant μνήμη διαφέρει από την global ως προς την αποθήκευση σταθερών τιμών. Η texture μνήμη προσφέρει διαφορετικό τρόπο διευθυνσιοδότησης, αλλά και διαχείρισης δεδομένων γιασυγκεκριμένους τύπους δεδομένων. Ως προς την χρήση της, γίνεται αποθήκευση και με μεγαλύτερη ταχύτητα στην ανάγνωση δεδομένων όπως πχ στις εικόνες.

Δύο κατηγορίες μνήμης που σχετίζονται μεταξύ τους είναι οι καταχωρητές (registers) και η τοπική μνήμη (local memory). Στην πρώτη κατηγορία όλα τα νήματα έχουν πρόσβαση στους καταχωρητές για αποθήκευση τοπικών μεταβλητών και ενδιάμεσων αποτελεσμάτων. Στη δεύτερη κατηγορία η τοπική μνήμη βρίσκει χρήση όταν οι καταχωρητές έχουν εξαντληθεί, ενώ είναι καταλληλότερη για την αποθήκευση των δεδομένων που διατηρούνται όταν εκτελείται ένας κύκλος.

Τέλος η global, η constant και η texture μνήμες χρησιμοποιούνται καθ’ όλη τη διάρκεια μίας εφαρμογής που εκτελείται στο kernel.

Για την βέλτιστη απόδοση όταν εκτελούνται παράλληλα προγράμματα στο πακέτο CUDA, είναι αναγκαίο να γίνεται σωστή διαχείριση των μνημών [17].



Σχήμα 3.11: Ιεραρχία Μνήμης

3.5.4 Ετερογενής προγραμματισμός

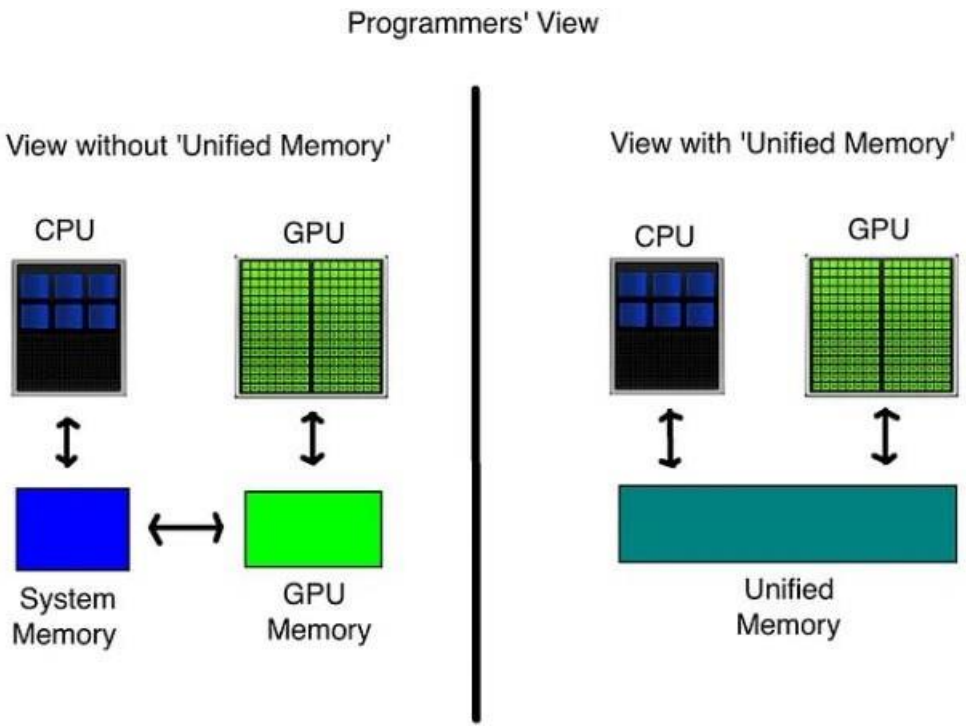
Στο προγραμματιστικό μοντέλο του CUDA, τα νήματα εκτελούνται σε διαφορετική συσκευή (device) που λειτουργεί ως ένας επιπλέον επεξεργαστής εκτός του host που τρέχει ένα πρόγραμμα. Ως host γίνεται αναφορά στην μνήμη της κεντρικής μονάδας και εκτελείται το κύριο πρόγραμμα. Για παράδειγμα, όταν ένα kernel εκτελείται στην GPU, το υπόλοιπο πρόγραμμα μίας συγκεκριμένης γλώσσας που έχει επιλεγεί από τον χρήστη (πχ C++), εκτελείται στον CPU επεξεργαστή.

Επιπλέον το μοντέλο CUDA, προϋποθέτει ότι και ο host αλλά και η συσκευή διατηρούν την δικό τους ξεχωριστό μέρος μνήμης στην DRAM (δυναμική μνήμη) και αυτές οι δύο μνήμες αναφέρονται ως host μνήμη (host memory) και μνήμη συσκευής (device memory).

Έπειτα, ένα πρόγραμμα διαχειρίζεται μέρη της global, constant και texture μνήμης, όπου είναι διαθέσιμα στα kernels μέσω του CUDA runtime. Επίσης περιλαμβάνεται η κατανομή της device memory αλλά και η από-κατανομή, όπως και η μεταφορά δεδομένων μεταξύ της host και της device μνήμης.

Υπάρχει και η Unified Memory (ενοποιημένη μνήμη), παρέχει μία διαχειρίσιμη μνήμη (managed memory) όπου συνδέει τα μέρη (διαστήματα) των host και device μνημών. Η managed memory είναι προσβάσιμη από όλες τις CPU και GPU που υπάρχουν στο υπολογιστικό σύστημα και αποτελεί μία μνήμη με συνοχή αλλά και κοινή διεύθυνση.

Αυτή η δυνατότητα, προσφέρει την υπερ-εγγραφή δεδομένων της device memory και μπορεί να απλοποιήσει τις διεργασίες της μνήμης με το να κατανέμει τα δεδομένα ταυτόχρονα και στην device και στην host μνήμη [17].



Σχήμα 3.12: Ενοποιημένη μνήμη

3.5.5 NVIDIA GPU Αρχιτεκτονική

Η αρχιτεκτονική μιας NVidia GPU είναι σχεδιασμένη στη βάση ενός κλιμακωτού πίνακα που αποτελείται από πολυνηματικούς επεξεργαστές συνεχούς ροής (Streaming multiprocessors) ή (SMs). Όταν ένα πρόγραμμα του CUDA, που τρέχει στον host της CPU, καλεί το grid ενός kernel, τα blocks του grid είναι μετρημένα και κατανεμημένα στους πολύ-επεξεργαστές με διαθέσιμη χωρητικότητα εκτέλεσης. Τα νήματα ενός μπλοκ ή και παραπάνω, εκτελούνται ταυτόχρονα σε έναν πολυεπεξεργαστή. Όταν τερματιστεί η εκτέλεση ενός μπλοκ για μία συγκεκριμένη λειτουργία, τότε νέα μπλοκ νημάτων εκτελούνται στους κενούς πλέον πολυεπεξεργαστές.

Ένας πολυεπεξεργαστής είναι σχεδιασμένος να εκτελεί εκατοντάδες νήματα ταυτόχρονα. Για την διαχείριση ενός μεγάλου αριθμού νημάτων, χρησιμοποιείται η αρχιτεκτονική SIMT (Single-Instruction, Multiple-Thread), όπου θα αναλυθεί παρακάτω. Η αρχιτεκτονική SIMT πρόκειται ουσιαστικά για μία οδηγία που την εκτελούν παράλληλα πολλά νήματα [17].

3.5.6 Αρχιτεκτονική SIMT

Ένας πολυεπεξεργαστής μπορεί να δημιουργήσει, να διαχειριστεί, να χρονοπρογραμματιστεί και να εκτελεί νήματα σε ομάδες (group) των 32 παράλληλων νημάτων, που ονομάζονται στημόνια (warps). Το κάθε νήμα που αποτελεί μαζί με τα υπόλοιπα ένα στημόνιο, ξεκινούν στην ίδια διεύθυνση προγράμματος, αλλά έχουν ξεχωριστά το καθένα τη δική του διεύθυνση οδηγίας (instruction address) και κατάσταση μετρητή (counter) και καταχωρητή (register). Με αυτόν τον τρόπο, τα νήματα μπορούν να ξεκινούν και να εκτελούνται ανεξάρτητα.

Όταν ένας πολυεπεξεργαστής εκτελεί ένα ή και περισσότερα μπλοκ νημάτων, τότε τα ξεχωρίζει σε στημόνια και το κάθε ένα από αυτά χρονόπρογραμματίζεται, ώστε να εκτελεστεί. Ο τρόπος με τον οποίο διαχωρίζονται τα μπλοκ σε στημόνια είναι συγκεκριμένος. Το κάθε στημόνιο περιέχει συνεχώς αυξανόμενα ID νημάτων με το πρώτο στημόνιο να περιέχει το νήμα 0. Η διαδικασία περιγράφεται αναλυτικότερα παραπάνω στην υποενότητα της ιεραρχίας νημάτων.

Ένα στημόνιο εκτελεί μια συγκεκριμένη οδηγία κάθε φορά και η μέγιστη αποδοτικότητα επιτυγχάνεται όταν και τα 32 νήματα σε ένα στημόνιο, έχουν την ίδια διαδρομή (path) εκτέλεσης. Τα νήματα ενός στημονιού μπορεί να έχουν απόκλιση μεταξύ τους εξαιτίας μιας διακλάδωσης, δηλαδή μίας συνθήκης που είναι εξαρτώμενη από δεδομένα που πρέπει να εκτελεστούν με διαφορετική σειρά.

Παρόλα αυτά τα νήματα μπορούν να εκτελεστούν ξεχωριστά, αλλά όταν συμβαίνει αυτό, τότε αυξάνεται η πιθανότητα μείωσης της απόδοσης. Ο λόγος είναι ότι, όταν υπάρχει διαφωνία μεταξύ την σειράς εκτέλεσης των νημάτων, ο χρονοπρογραμματιστής (scheduler), είναι ανάγκη να τα επαναφέρει σε μία σειρά εκτέλεσης. Ενώ όταν όλες οι ξεχωριστές διαδρομές έχουν χρησιμοποιηθεί, τότε τα νήματα συγκλίνουν στην ίδια διαδρομή εκτέλεσης [6].

Κεφάλαιο 4ο: Βιβλιογραφική Έρευνα υλοποίησης αλγορίθμου με επιτάχυνση GPU

4.1 Εισαγωγή

Σε αυτό το κεφάλαιο θα γίνει αναφορά σε μία εργασία [6] όπου έχει γίνει υλοποίηση εφαρμογής επίλυσης αντίστοιχου προβλήματος και πιο συγκεκριμένα υλοποιείται επιτάχυνση του αλγορίθμου Simplex με τη γλώσσα C++ και εντολές CUDA (παράλληλη εκδοχή) αλλά γίνεται και σειριακή υλοποίηση με τη γλώσσα C, ώστε ο συγγραφέας να συγκρίνει την σειριακή με την παράλληλη έκδοση και να αξιολογήσει τα αποτελέσματα και να διαπιστώσει αν όντως υπάρχουν κάποια οφέλη σε αυτό το εγχείρημα.

Θα εξετάσουμε τις μεθόδους που χρησιμοποιήθηκαν, τις βελτιώσεις στην απόδοση που επιτεύχθηκαν, καθώς και τις προκλήσεις και τα εμπόδια που αντιμετωπίστηκαν κατά την ανάπτυξη αυτών των υλοποιήσεων.

Ειδικότερα, θα επικεντρωθούμε στην υλοποίηση της αναθεωρημένης μεθόδου Simplex, η οποία είναι ιδιαίτερα κατάλληλη για παραλληλισμό λόγω των υπολογιστικών της απαιτήσεων και της δυνατότητας εκτέλεσης αλγεβρικών και διαδικασιών μείωσης σε παράλληλο περιβάλλον. Θα αναλυθεί η μετάβαση από τη σειριακή στην παράλληλη έκδοση του αλγορίθμου, τη χρήση της βιβλιοθήκης CUBLAS για αλγεβρικές ρουτίνες και την εφαρμογή CUDA kernels για μη αλγεβρικές ρουτίνες.

Η βιβλιογραφική έρευνα σε αυτό το κεφάλαιο θα παρέχει μια σαφή εικόνα του τρέχοντος επιπέδου έρευνας και ανάπτυξης στον τομέα αυτόν, καθιστώντας δυνατή την κατανόηση των πλεονεκτημάτων και των περιορισμών της χρήσης GPU για την επίλυση προβλημάτων γραμμικού προγραμματισμού.

Η συγκεκριμένη έρευνα, αφορά μια ερευνητική εργασία [6], από την οποία η παρούσα διπλωματική εργασία έχει βασιστεί σε έναν βαθμό και με την έννοια του περιεχομένου αλλά και της υλοποίησης δομικά. Πιο συγκεκριμένα αφορά την έρευνα που είχε κάνει ο Daniel Giuseppe Spampinato στο μάθημα των Πολύπλοκων Υπολογιστικών Συστημάτων, στο τμήμα Πληροφορικής του Νορβηγικού Πανεπιστημίου Επιστημών και τεχνολογιών.

4.1.1 Περιγραφή στόχων εργασίας

Η συγκεκριμένη εργασία [6] έχει παρόμοιο σκοπό με την παρούσα διπλωματική. Ο Daniel Spampinato στην εργασία που θα αναφερθεί στη συνέχεια προσπαθεί να υλοποιήσει έναν κώδικα ο οποίος χρησιμοποιεί την αναθεωρημένη μέθοδο Simplex, αρχικά σε σειριακή εκδοχή και έπειτα με παράλληλο τρόπο, ώστε να γίνει η σύγκριση στις μεταξύ τους διαφορές και τελικά να βγουν χρήσιμα συμπεράσματα.
Αρχικά είναι ανάγκη να αναφερθεί ότι το συγκεκριμένο εγχείρημα είναι εκ φύσεως πολύ δύσκολο και συμφώνα με έρευνες των Jung και Green [12]-[13], τα προβλήματα γραμμικού προγραμματισμού είναι πιο δύσκολο να αξιοποιήσουν την κάρτα γραφικών σε σχέση με άλλου είδους προβλήματα. Όμως ο Daniel Spampinato κυρίως για λόγους ερευνητικούς θέλησε να πειραματιστεί και να βγάλει το οποιοδήποτε χρήσιμο αποτέλεσμα.

4.2 Επιλογή μεθόδου Simplex

Όπως αναφέρθηκε και προηγουμένως η μέθοδος που επέλεξε είναι η αναθεωρημένη μέθοδος Simplex. Η επιλογή της σχετίζεται με την απλότητα υλοποίησης της καθώς χρησιμοποιεί κυρίως βασικές αλγεβρικές πράξεις και χρήση πινάκων όπου είναι πιο εύκολο να επεξεργαστούν παράλληλα και να επωφεληθούν από την αρχιτεκτονική των GPU, σε αντίθεση με την κλασική μέθοδο Simplex όπου είναι απαραίτητη η χρήση των διανυσμάτων αλλά και ειδικά σχεδιασμένων δομών δεδομένων.
Σύμφωνα με τον συγγραφέα, τα κύρια στοιχεία που πρέπει να αναλογιστεί κάποιος για να επιλέξει έναν αλγόριθμο με σκοπό την υψηλή απόδοση σε παράλληλη υλοποίηση, είναι τα εξής:

- Ρυθμός Διεκπεραίωσης (Throughput): Αναφέρεται στις εργασίες που εκτελούνται στο σύστημα σε ένα συγκεκριμένο χρόνο (πχ: πλήθος πράξεων ανά δευτερόλεπτο εκτέλεσης από την GPU).

- Καθυστέρηση (Latency) : Χρόνος που απαιτείται για την εκτέλεση μιας μεμονωμένης πράξης

Συμπερασματικά, δίνεται προτεραιότητα στην συνολική απόδοση του συστήματος και όχι στην ταχύτητα εκτέλεσης μεμονωμένων πράξεων διότι οι κάρτες γραφικών χάρις στις υψηλές τους δυνατότητες επωφελούνται από την παράλληλη επεξεργασία και πετυχαίνουν υψηλούς ρυθμούς διεκπεραίωσης που είναι πολύ σημαντικότερο κυρίως σε προβλήματα μεγάλης κλίμακας.
Επιπρόσθετα, τονίζει πως και η επιλογή του αναθεωρημένου Simplex δεν είναι μια λύση χωρίς δυσκολίες διότι, προβλήματα όπως τα σφάλματα στρογγυλοποίησης και η απώλεια σημαντικών ψηφίων είναι αναπόφευκτα.

4.3 Στρατηγική Υλοποίησης

Η υλοποίηση του αλγορίθμου θα γίνει με δύο εκδοχές. Η πρώτη εκδοχή αφορά την σειριακή εκδοχή του αλγορίθμου που έχει γραφτεί με την γλώσσα C, ενώ η δεύτερη είναι η παράλληλη εκδοχή γραμμένη σε γλώσσα C++ και περιλαμβάνει εντολές CUDA.

Οι δύο υλοποιήσεις χρησιμοποιούν δομές δεδομένων όπως οι πίνακές και οι πράξεις γίνονται μέσω ρουτινών οι οποίες διαχωρίζονται σε δύο κατηγορίες, τις αλγεβρικές και τις μη-αλγεβρικές. Οι αλγεβρικές ρουτίνες περιλαμβάνουν όλες τις πράξεις που είναι δυνατόν να γίνουν μεταξύ πινάκων, όπως η πρόσθεση, αφαίρεση, πολλαπλασιασμός και διαίρεση, ενώ οι μη αλγεβρικές ρουτίνες περιλαμβάνουν τις επαναληπτικές δομές δεδομένων (πχ for, while) και τις διάφορες συνθήκες ελέγχου(πχ. If-else).

Στην σειριακή υλοποίηση οι πίνακες με τα δεδομένα είναι αποθηκευμένοι στην κεντρική μνήμη του υπολογιστή, ενώ στην παράλληλη υλοποίηση οι πίνακες είναι αποθηκευμένοι στην μνήμη της κάρτα γραφικών του υπολογιστή. Τέλος, οι αλγεβρικές ρουτίνες στη σειριακή έκδοση υλοποιούνται με το πακέτο BLAS (Basic Linear Algebra Subrpograms), ενώ στη παράλληλη με το πακέτο cuBLAS. Το πακέτο BLAS είναι μία βιβλιοθήκη που περιέχει ρουτίνες με βασικές πράξεις της γραμμικής άλγεβρας, όπου χρησιμοποιούνται για την επίλυση μαθηματικών προβλημάτων και το πακέτο cuBLAS είναι η GPU-επιταχυνόμενη εκδοχή του BLAS, αναπτυγμένη από την NVIDIA για την αρχιτεκτονική CUDA.

Όσον αφορά τις μη-αλγεβρικές ρουτίνες στον σειριακό κώδικα υλοποιούνται με τις γνωστές εντολές επανάληψης και συνθηκών ελέγχου, ενώ στον παράλληλο κώδικα υλοποιούνται είτε με τις γνωστές εντολές επανάληψης και συνθηκών ελέγχου είτε με εντολές CUDA. Πιο συγκεκριμένα το CUDA παρέχει ένα σύνολο εντολών και συναρτήσεων που επιτρέπουν την παράλληλη εκτέλεση κώδικα στην GPU. Αυτές οι εντολές χρησιμοποιούνται για τον προγραμματισμό των kernels, την κατανομή και

διαχείριση της μνήμης της GPU, και τον συγχρονισμό των threads. Τα kernels υλοποιούνται κυρίως με πίνακες, οι οποίες διαχωρίζονται σε υποπίνακες που αντιστοιχούν σε δισδιάστατα μπλοκ.

4.3.1 Διαμόρφωση μη αλγεβρικών ρουτινών: Υπολογισμός Εισερχόμενης Μεταβλητής

Η **εισερχόμενη μεταβλητή** είναι η μεταβλητή που εισάγεται στη βάση του Simplex αλγορίθμου κατά τη διάρκεια ενός βήματος. Η επιλογή αυτής της μεταβλητής βασίζεται στο κριτήριο του ποια μεταβλητή μπορεί να αυξήσει την τιμή της αντικειμενικής συνάρτησης, βελτιώνοντας έτσι τη λύση. Η μεταβλητή αυτή είναι η πιο «υποσχόμενη» όσον αφορά τη βελτίωση της τρέχουσας λύσης και συνήθως είναι αυτή που αντιστοιχεί στο μεγαλύτερο αρνητικό στοιχείο της γραμμής κόστους.

Στη σειριακή έκδοση, για τον υπολογισμό της εισερχόμενης μεταβλητής, υπολογίζεται το γινόμενο δύο πινάκων, η μετακίνηση ενός πίνακα και γίνεται η εύρεση του ελαχίστου. Στο παρακάτω σχήμα η αναζήτηση του ελαχίστου περιλαμβάνεται στην μέθοδο `entering_index()`.

```
// y = cb*Binv
cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
    1, m, m, 1, cb, m, Binv, m, 0, y, m);
memcpy(&yb[1], y, m*sizeof(float));

// e = [1 y]*[-c ; A]
cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
    1, n, m+1, 1, yb, n, D, n, 0, e, n);
ev = entering_index(e, n);
```

Σχήμα 4.1: Υπολογισμός εισερχόμενης μεταβλητής

Για την εύρεση της εισερχόμενης μεταβλητής στον παράλληλο κώδικα, οι τιμές υπολογίζονται με κλήσεις συναρτήσεων του cuBLAS, έπειτα πραγματοποιείται η εύρεση του πιο αρνητικού αριθμού με τη χρήση ενός reduction kernel σε CUDA. Πιο συγκεκριμένα αυτή η διαδικασία περιλαμβάνει την παράλληλη μείωση (parallel reduction) μέχρι να παραμείνει μια μόνο τιμή, που είναι το ελάχιστο. Τέλος , το αποτέλεσμα μεταφέρεται στην global μνήμη.

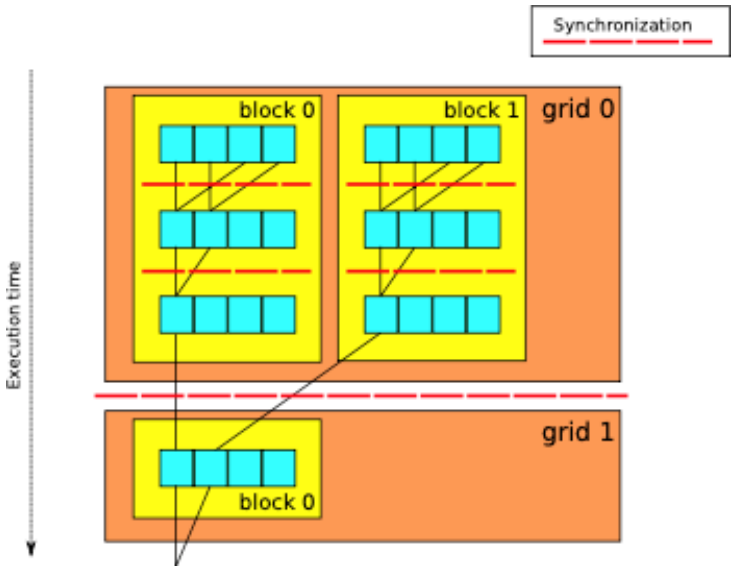
```
//Each block loads its elements into shared mem,
//padding if not multiple of BS
__shared__ float sf[BS];
sf[tid] = (j<n) ? f[j] : FLT_MAX;
__syncthreads();
...
for(int s=blockDim.x/2; s>0; s>>=1)
{
  if(tid < s) sf[tid] = sf[tid] > sf[tid+s]
    ? sf[tid+s] : sf[tid];
  __syncthreads();
}

if(tid == 0) min[blockIdx.x] = sf[0];
```

Σχήμα 4.2: Μετακίνηση πίνακα στην shared μνήμη και η διαδικασία παράλληλης μείωσης για την εύρεση του πιο αρνητικού αριθμού

4.3.2 Διαμόρφωση μη αλγεβρικών ρουτινών: Υπολογισμός Εξερχόμενης Μεταβλητής

Η **εξερχόμενη μεταβλητή** είναι η μεταβλητή που αφαιρείται από τη βάση του Simplex αλγορίθμου για να διατηρηθεί η εφικτότητα της λύσης. Επιλέγεται με βάση τον κανόνα του μικρότερου λόγου(minimum ratio test), όπου συγκρίνεται πόσο πολύ μπορεί να αυξηθεί η εισερχόμενη μεταβλητή πριν κάποια από τις μεταβλητές της βάσης γίνει μηδενική ή αρνητική. Αυτή η μεταβλητή πρέπει να αντικατασταθεί για να αποφευχθεί η παραβίαση των περιορισμών.

Για την εύρεση της εξερχόμενης μεταβλητής απαιτείται η εξαγωγή μιας συγκεκριμένης στήλης από έναν πίνακα και η μετακίνηση της σε έναν νέο πίνακα. Ο νέος πίνακας από το πρώτο βήμα πολλαπλασιάζεται με έναν διαφορετικό πίνακα. Στη συνέχεια υπολογίζεται η διαίρεση των δύο πινάκων, δηλαδή κάθε στοιχείο του πρώτου διαιρείται με το πρώτο στοιχείο του δευτέρου. Τέλος το αποτέλεσμα της εξερχόμενης μεταβλητής είναι το πιο αρνητικό στοιχείο που έχει προκύψει από όλη τη διαδικασία.



Σχήμα 4.3: Ρουτίνα εύρεσης ελαχίστου (reduction)

4.3.3 Διαμόρφωση μη αλγεβρικών ρουτινών: Υπολογισμός Αντίστροφου Πίνακα B^{-1}

Η διαδικασία αντίστροφής του πίνακα δεν είναι πάντα απαραίτητη και μπορεί να αποφευχθεί. Στην πραγματικότητα, ο πίνακας B δεν χρησιμοποιείται άμεσα στις πράξεις και στους υπολογισμούς. Αντί να πραγματοποιούμε την πλήρη αντίστροφή του πίνακα, προτιμάται η χρήση μιας μεθόδου που υπολογίζει την αντίστροφο πίνακα χωρίς να πραγματοποιεί την άμεση αντίστροφή.

Η τεχνική που χρησιμοποιείται βασίζεται στη δημιουργία ενός πίνακα E (διαστάσεων m×m) που εξαρτάται από τις εισερχόμενες και εξερχόμενες μεταβλητές. Ο πίνακας E πολλαπλασιάζεται με τον αντίστροφο πίνακα B^{-1} για να επιτευχθεί το επιθυμητό αποτέλεσμα.

Για την q-στη στήλη του πίνακα E, εφαρμόζεται ο εξής τύπος για τον υπολογισμό της: $E_{iq} = \frac{a_q}{a_i} - a_q$, όπου a_q και a_i είναι οι συντελεστές που σχετίζονται με τις εισερχόμενες και εξερχόμενες μεταβλητές αντίστοιχα.

4.4 Περιβάλλον πειράματος

Το πείραμα πραγματοποιήθηκε σε ετερογενές υπολογιστικό σύστημα. Αποτελούταν από έναν 64-bit επεξεργαστή Intel Core2 Quad (Q9550) 2.83 GHz, με 12 MB cache και ρυθμό λειτουργίας 1333 MHz και από μία κάρτα γραφικών NVIDIA GeForce GTX 280. Ο επεξεργαστής και η κάρτα γραφικών επικοινωνούν μέσω PCI-Express 2.0. Η κάρτα γραφικών είναι συνδεδεμένη σε θύρα 16 λωρίδων (x16) δηλαδή η κάρτα γραφικών μπορεί να χρησιμοποιεί έως και 16 λωρίδες PCIe, επιτρέποντας μέγιστο ρυθμό μεταφοράς δεδομένων μέχρι και 8 GB/s (0.5 GB/s ανά λωρίδα × 16 λωρίδες). Το λειτουργικό σύστημα του υπολογιστή είναι το Ubuntu 8.04 με Linux kernel 2.6.24-22. Η CPU υλοποίηση μεταγλωττίστηκε χρησιμοποιώντας gcc έκδοση 4.2.4. Η BLAS βιβλιοθήκη βελτιστοποιήθηκε από την έκδοση 3.6.0 ATLAS. Η GPU έκδοση υλοποιήθηκε με το CUDA 2.0.

4.5 Μεθοδολογία – Αντικειμενικοί στόχοι πειράματος

Στο πείραμα εκτελέστηκαν 1000 διαφορετικά προβλήματα με τυχαίες τιμές. Η κλίμακα των προβλημάτων αυξανόταν σταδιακά. Το μεγαλύτερο πρόβλημα σε κλίμακα ήταν της τάξης των 2000x2000 (μεταβλητές, περιορισμοί) πίνακα. Σε όλες τις περιπτώσεις προβλημάτων, κάθε πείραμα είχε τον ίδιο αριθμό μεταβλητών και περιορισμών.

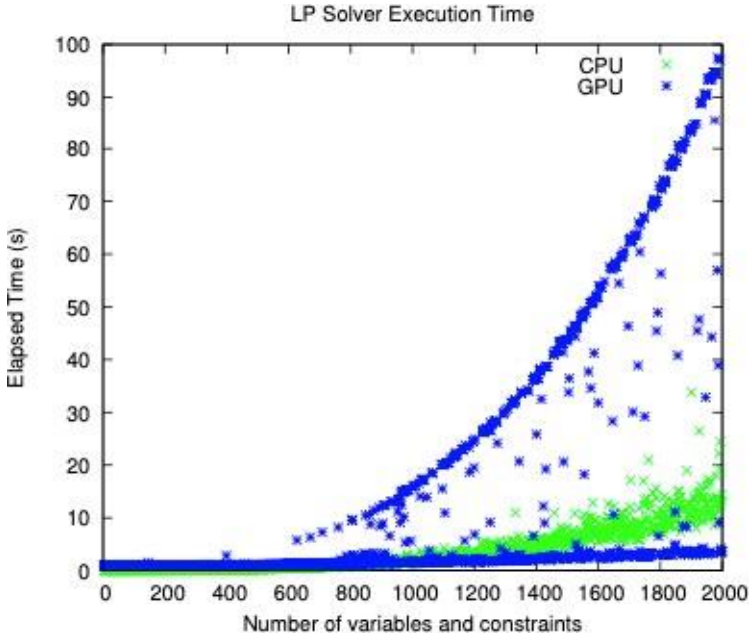
Όσον αφορά την ανάλυση απόδοσης του αλγορίθμου, ένα αντικειμενικό κριτήριο είναι η χρονική διάρκεια και η επιτάχυνση του. Συγκεκριμένα η απόδοση του χρόνου μετρήθηκε σε νανοδευτερόλεπτα (n_s) και όχι με χιλιοστά του δευτερολέπτου (m_s) για μεγαλύτερη ακρίβεια ως προς την σύγκριση της σειριακής έκδοσης με την παράλληλη. Σχετικά με την επιτάχυνση, είναι ο λόγος των χρόνων εκτέλεσης του σειριακού κώδικα και του παράλληλου και υπολογίζει τις φορές που εκτελείται ταχύτερα η παράλληλη υλοποίηση του κώδικα από αυτή του σειριακού κώδικα.

Επιτάχυνση: T_s/T_p

4.6 Αποτελέσματα

Συνολικά τα αποτελέσματα είναι εμφανή ως προς την υπεροχή της GPU, έναντι του CPU, ειδικότερα όταν οι περιορισμοί ξεπερνούν τους 900. Όταν οι περιορισμοί είναι λιγότεροι από 900 η εκτέλεση σε GPU δεν είναι σταθερά αποδοτικότερη καθώς υπάρχουν μετρήσεις όπου η εκτέλεση σε CPU αποδίδει καλύτερα. Ένας από τους λόγους που αποδίδει καλύτερα η εκτέλεση σε CPU είναι τα επίπεδα ανοχής, όπου στην CPU εκτέλεση γίνεται χρήση ανοχής $\mathrm{e} = 10^{-4}$, ενώ στην GPU εκτέλεση $\mathrm{e} = 7 * 10^{-5}$. Με την ανοχή ορίζεται το επιτρεπόμενο σφάλμα για τις τιμές που υπολογίζονται. Αν και γίνεται χρήση μικρότερης ανοχής στην GPU, εκτέλεση δεν επιτυγχάνει την ίδια ακρίβεια με την CPU εξαιτίας των σφαλμάτων στρογγυλοποίησης και της διαφορετικής φύσης των υπολογισμών σε παράλληλο περιβάλλον. Αναλυτικότερα, οι κάρτες γραφικών υπολογίζουν πράξεις με απλή ακρίβεια (single precision) σε αντίθεση με τον επεξεργαστή που υπολογίζει σε διπλή ακρίβεια (στις σύγχρονες κάρτες γραφικών δεν ισχύει αυτό πλέον). Συνεπώς η εκτέλεση σε κάρτα γραφικών έχει μεγαλύτερο σφάλμα στρογγυλοποίησης. Ένας ακόμη παράγοντας είναι ότι εκ φύσεως η παράλληλη εκτέλεση προκαλεί μικρές αποκλίσεις επειδή δεν υπάρχει ακριβής συγχρονισμός των υπολογισμών ή ακόμη και συσσώρευση μικρών σφαλμάτων στρογγυλοποίησης, σε αντίθεση με την τους CPU όπου η εκτέλεση είναι ακριβέστερη και ελεγχόμενη εξαιτίας της σειριακής φύσης. Σε πρώτη φάση, για μέχρι και 1000 περιορισμούς η σειριακή εκτέλεση αποδίδει λίγο καλύτερα καθώς όπως φαίνεται και στο παρακάτω διάγραμμα, έχει μία σταθερή διαφορά 0.9 δευτερολέπτων. Αυτή η διαφορά οφείλεται στο γεγονός ότι για να εκτελεστεί η παράλληλη εκδοχή, απαιτείται χρόνος κατανομής, αποδέσμευσης και τέλος μεταφοράς των δεδομένων μεταξύ της κεντρικής μνήμης του υπολογιστικού συστήματος στην κάρτα γραφικών για να εκτελεστεί ο αλγόριθμος.

Μετρήθηκαν τρεις φάσεις του αλγορίθμου όπου υπολογίζει κρίσιμες διαδικασίες για την επίλυση του προβλήματος όπως, ο χρόνος και η επιτάχυνση για την εισερχόμενη μεταβλητή, την εξερχόμενη μεταβλητή και την ενημέρωση της αντιστροφής της βάσης.



Σχήμα 4.4: Σύγκριση απόδοσης σειριακής-παράλληλης υλοποίησης

4.7 Χρόνος Αναζήτησης Εισερχόμενης Μεταβλητής

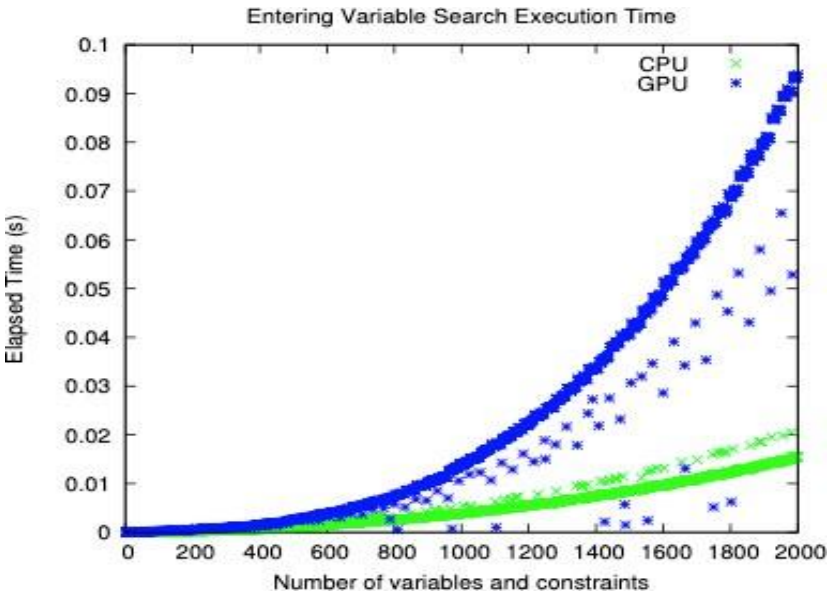
Όπως φαίνεται και στο διάγραμμα, η αναζήτηση για την εισερχόμενη μεταβλητή διήρκησε περισσότερο στην GPU εκτέλεση για προβλήματα με λιγότερους από 1000 περιορισμούς. Ο συγκεκριμένος υπολογισμός περιλαμβάνει τόσο αλγεβρικές πράξεις που υλοποιούνται μέσω των BLAS ρουτινών όσο και μη αλγεβρικές πράξεις. Πιο συγκεκριμένα, οι cuBLAS ρουτίνες συνεισφέρουν μόλις κατά 0.004% στον συνολικό χρόνο εκτέλεσης της GPU, επομένως δεν ευθύνονται για τη χρονική διαφορά (1s) εις βάρος της παράλληλης εκτέλεσης. Η κύρια αιτία της καθυστέρησης σχετίζεται με τη διαχείριση μνήμης, και πιο συγκεκριμένα με την κατανομή, αποδέσμευση και μεταφορά δεδομένων από την CPU στην GPU.

Επιπλέον, η καθυστέρηση που προκαλείται από την αναζήτηση της εισερχόμενης μεταβλητής δεν συνεισφέρει σημαντικά στη συνολική καθυστέρηση, καθώς αποτελεί μόνο ένα πολύ μικρό ποσοστό (0.5%) του συνολικού χρόνου εκτέλεσης.

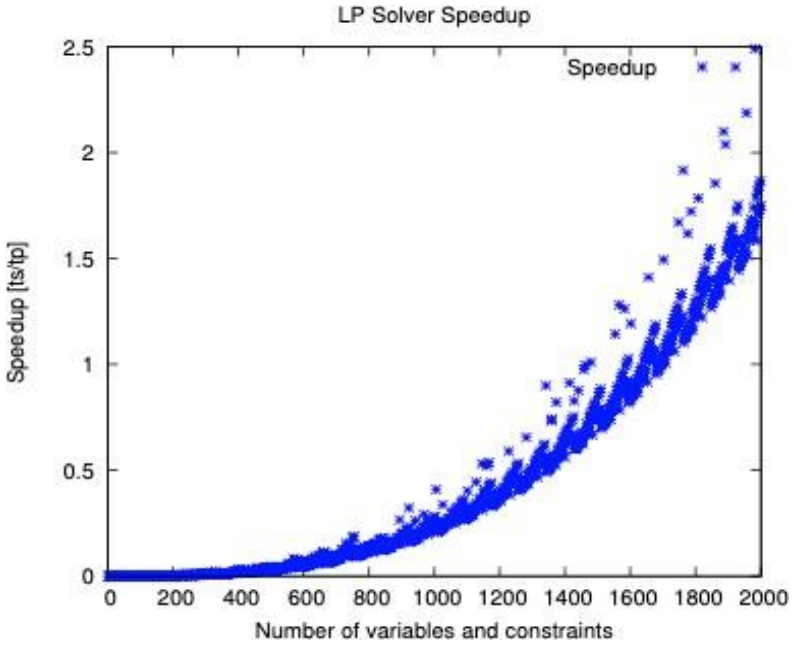
4.8 Χρόνοι Επιτάχυνσης

Η επιτάχυνση διακρίνεται στην GPU σε μεγαλύτερης κλίμακας προβλήματα και αυξάνεται πολύ γρηγορότερα για ακόμη μεγαλύτερα δεδομένα. Αρχικά, παρατηρείται μια μικρή αύξηση με έναν παράγοντα *0.5 όταν οι περιορισμοί έχουν φτάσει στους 1400. Έπειτα η επιτάχυνση αυξάνεται γρηγορότερα και έχει τιμές από 2 έως και 2,5 φορές γρηγορότερη, ειδικά όταν οι περιορισμοί είναι πλέον στους 2000. Τελικά η παράλληλη εκτέλεση ξεπερνά την σειριακή για προβλήματα μεγέθους από 1600-1800 περιορισμούς.

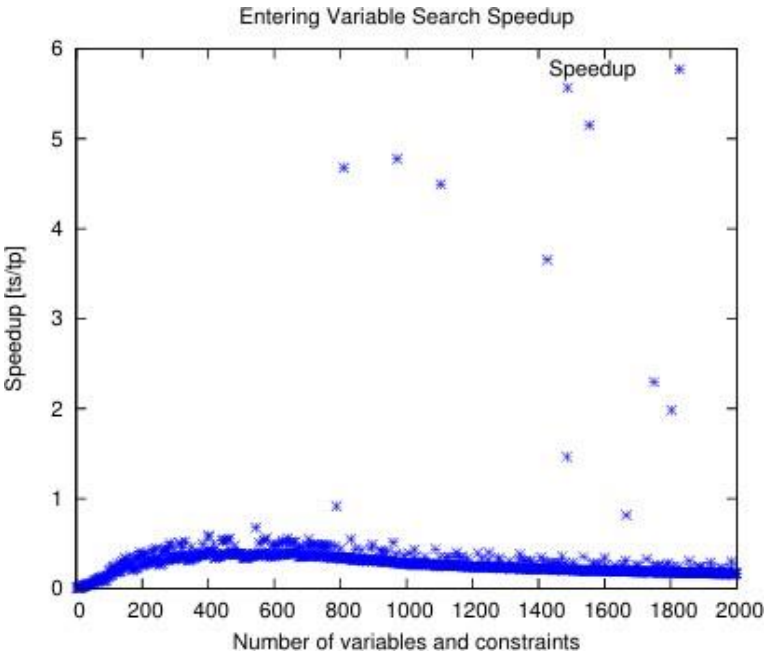
Όσον αφορά την τοπική επιτάχυνση της εισερχόμενης μεταβλητής παρατηρήθηκε ότι έχει μεγάλες αυξομειώσεις και επιτάχυνση έως και 5 φορές για συγκεκριμένες κατηγορίες προβλημάτων.



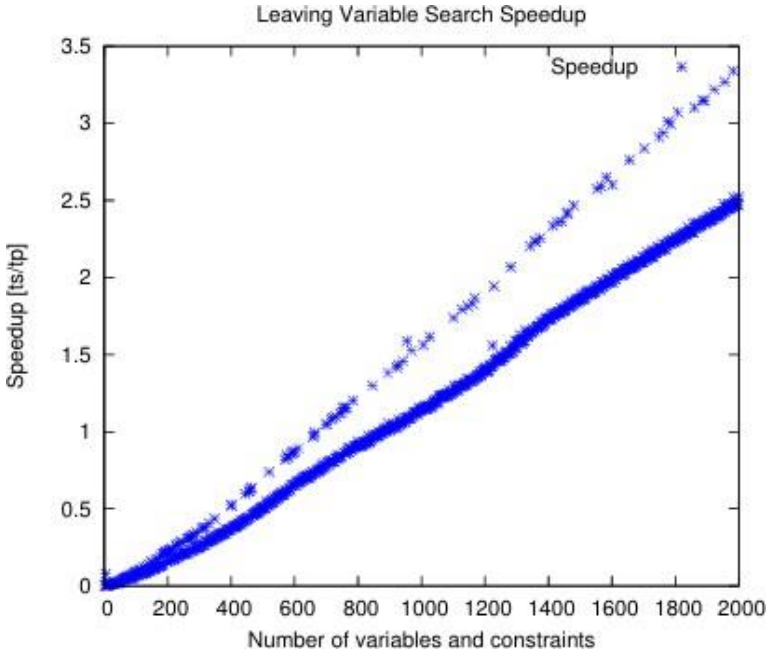
Σχήμα 4.5: Χρόνος Αναζήτησης Εισερχόμενης Μεταβλητής



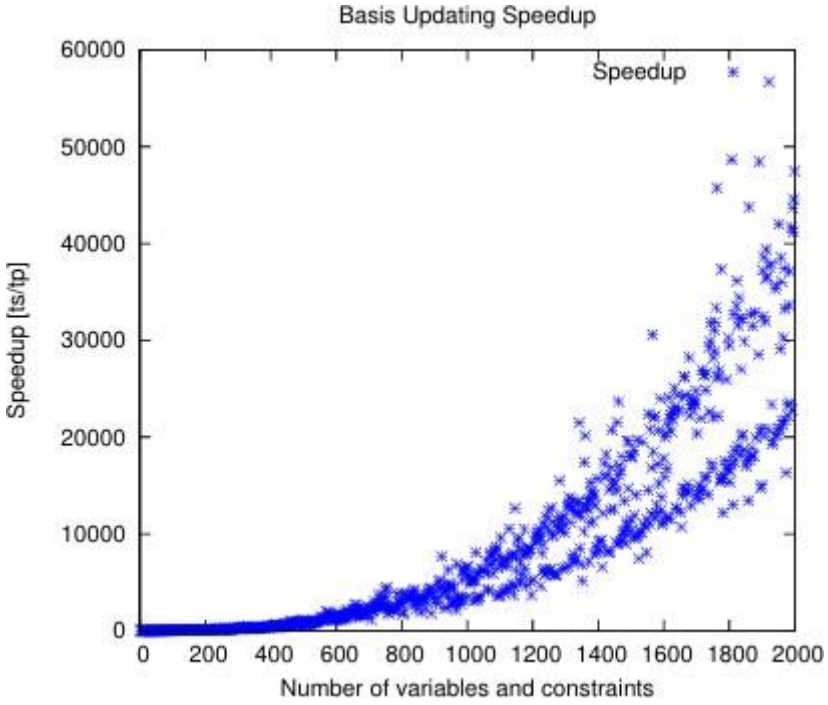
Σχήμα 4.6: Συνολική Επιτάχυνση



Σχήμα 4.7: Τοπική επιτάχυνση εύρεσης εισερχόμενης μεταβλητής



Σχήμα 4.8: Τοπική επιτάχυνση εύρεσης εξερχόμενης μεταβλητής



Σχήμα 4.9: Τοπική επιτάχυνση αντιστροφής βάσης

4.9 Συμπεράσματα

Με βάση τα όσα προηγήθηκαν, προκύπτει ότι η απόδοση ενός αλγορίθμου και εν προκειμένω της υλοποίησης του αναθεωρημένου Simplex, δεν εξαρτάται μόνο από την παράλληλη εκτέλεση του αλλά από πολλούς παράγοντες τους οποίους θα αναφέρουμε.

Αρχικά είναι πολύ σημαντικό να αναφερθεί ξανά πως η GPU σχεδιάστηκε κυρίως για την επεξεργασία γραφικών, μια διαδικασία που είναι αρκετά πολύπλοκη και απαιτεί επαναλαμβανόμενους και παράλληλους υπολογισμούς. Με την έλευση του GPGPU οι επιστήμονες και οι ερευνητές ξεκίνησαν να υλοποιούν τα προγράμματα τους στην κάρτα γραφικών, προβλήματα κυρίως μεγάλης κλίμακας. Είναι σημαντικό όμως να γίνει κατανοητό πως δεν αρκεί απλώς η μεταφορά του κώδικα που εκτελέστηκε στην CPU σειριακά, στην κάρτα γραφικών για να εκτελεστεί αλλά η εξαρχής κατασκευή του κώδικα με γνώμονα την εκμετάλλευση των αρχιτεκτονικών ιδιαιτεροτήτων της εκάστοτε κάρτας γραφικών. Σε περίπτωση που δεν ικανοποιείται αυτή η συνθήκη είναι σύνηθες τα αποτελέσματα να μην είναι αποδοτικά και ικανοποιητικά. Επιπρόσθετα, είναι ανάγκη ο αλγόριθμος να είναι κατασκευασμένος έτσι, ώστε να αποφευχθούν οι εξαρτήσεις δεδομένων (Data dependencies), διότι δημιουργείται ένα «bottleneck» στη ροή των προγραμμάτων και δεν καθίσταται εύκολο να εκτελεστούν πολλές εντολές ταυτόχρονα με συνέπεια να καθυστερεί συνολικά η εφαρμογή.

Σχετικά με τα αποτελέσματα του πειράματος, προκύπτουν συμπεράσματα που αφορούν το χρονικό της υλοποίησης του κώδικα, της απασφαλμάτωσης του, την απόδοση του κώδικα αλλά και την επικοινωνία μεταξύ του επεξεργαστή και της κάρτας γραφικών.

Αρχικά ο συγγραφέας αναφέρει πως για την συγγραφή του CUDA κώδικα απαιτήθηκε παρόμοιος χρόνος με αυτόν του σειριακού κώδικα και πως συνολικά για την κατανόησή και την υλοποίηση των παραλληλισμένων αλγορίθμων απαιτείται μεγάλη προσοχή και ανέφερε πως είναι μία χρονοβόρα διαδικασία. Στη συνέχεια για το κομμάτι της αποσφαλμάτωσης, ανέφερε πως είναι ιδιαίτερα απαιτητική, καθώς η υποστήριξη εργαλείων του CUDA είναι περιορισμένη αλλά κυρίως για την αποσφαλμάτωση απαιτείται η μεταφορά των δεδομένων από την κάρτα στον επεξεργαστή για να ανιχνευθούν τα όποια λάθη υπάρχουν. Για την επίτευξη αυτής της διαδικασίας όμως είναι αναγκαία η κατασκευή ειδικών δομών δεδομένων που θα μεταφέρουν τα δεδομένα και θα ανιχνεύουν τα σφάλματα.

Όσον αφορά το κομμάτι της απόδοσης, όπως αναφέρθηκε προηγουμένως και επισημαίνει και ο συγγραφέας, η απόδοση του εκτελεσμένου αλγορίθμου στην κάρτα γραφικών είναι αυξημένη από 2 έως και 2.5 φορές σε σχέση με την σειριακή εκτέλεση του αλγορίθμου, για προβλήματα όμως μεγάλης κλίμακας με χιλιάδες μεταβλητές και περιορισμούς. Συνολικά, με βεβαιότητα αναφέρει πως υπήρχε μία κλιμακωτή αύξηση της απόδοσης, αναλογικά με την αύξηση του μεγέθους του προβλήματος και συνεπώς όσο μικρότερο είναι το πρόβλημα τόσο λιγότερο απέδιδε η παράλληλη έκδοση στην GPU. Παρά την μείωση της απόδοσης σε μικρότερης κλίμακας προβλήματα, ο συγγραφέας αναφέρει πως ένα σημαντικό ζήτημα ήταν ότι η GPU κατά τη διαδικασία της εκφόρτωσης εργασιών από τον επεξεργαστή είχε θετικό πρόσημο, ειδικά για εφαρμογές πολυνηματικές διότι συνολικά γινόταν καλύτερος καταμερισμός εργασίας και το υπολογιστικό σύστημα είχε βελτιωμένη απόδοση.

Ένας ακόμη κρίσιμος παράγοντας για την αποδοτική επιτάχυνση μέσω GPU, ανέφερε πως είναι η επικοινωνία του επεξεργαστή με την κάρτα γραφικών καθώς μία μη αποδοτική μεταφορά δεδομένων μεταξύ τους δημιουργεί τα λεγόμενα bottlenecks και έτσι μειώνεται η απόδοση της συνολικής εφαρμογής.

Όπως αναφέρθηκε προηγουμένως η ακρίβεια των υπολογισμών είναι καθοριστική και για την απόδοση του αλγορίθμου καθώς σε πράξεις με μικρή ακρίβεια και μεγάλο αριθμό σφαλμάτων, επηρεαζόταν η απόδοση της παράλληλης εφαρμογής.

Συμπερασματικά, όπως εύκολα διακρίνεται η απόδοση ενός παράλληλου αλγορίθμου εξαρτάται από πολλούς παράγοντες και αποτελεί μια δύσκολη διαδικασία κατά την όποια ο οποιασδήποτε είναι ανάγκη να βελτιώνει τμήματα του αλγορίθμου και να λαμβάνει υπόψιν όχι μόνο κατασκευαστικά ζητήματα του αλγορίθμου αλλά και την θεωρία στην λειτουργία των υπολογιστικών συστημάτων και συσκευών του επεξεργαστή της κάρτας γραφικών και μνήμης RAM.

Κεφάλαιο 5ο: Επιτάχυνση αλγορίθμου Simplex με χρήση της βιβλιοθήκης CuPy

5.1 Εισαγωγή

Σε αυτό το κεφάλαιο, θα παρουσιαστεί η υλοποίηση του αλγορίθμου Simplex, ο οποίος θα εκτελεστεί τόσο σε σειριακή μορφή, χρησιμοποιώντας τις βιβλιοθήκες NumPy και SciPy της Python, όσο και σε παράλληλη μορφή, αξιοποιώντας την επιτάχυνση μέσω GPU με τη βιβλιοθήκη CuPy. Στη συνέχεια, θα γίνει συγκριτική ανάλυση της απόδοσης του αλγορίθμου σε διάφορα στιγμιότυπα προβλημάτων μικρής, μεσαίας και μεγάλης κλίμακας, με στόχο την αξιολόγηση της επιτάχυνσης που επιτυγχάνεται μέσω της χρήσης GPU.

Αρχικά, είναι σημαντικό να επισημανθεί ότι, από τη φύση τους, τα προβλήματα γραμμικού προγραμματισμού είναι δύσκολο να παραλληλοποιηθούν, και συχνά-κυρίως σε προβλήματα μικρής και μεσαίας κλίμακας- δεν αποδίδουν εντυπωσιακά αποτελέσματα χρονικής επιτάχυνσης. Αυτό οφείλεται και στο γεγονός πως οι αλγόριθμοι αυτοί περιλαμβάνουν σύνθετες πράξεις μεταξύ πινάκων και διανυσμάτων, καθώς και επαναληπτικές διαδικασίες, οι οποίες δεν προσαρμόζονται πάντα εύκολα σε παράλληλες υπολογιστικές αρχιτεκτονικές. Στο πλαίσιο αυτό, θα αναλυθούν οι παράγοντες που επηρεάζουν την απόδοση της παράλληλης επεξεργασίας, είτε θετικά είτε αρνητικά.

Επιπλέον, θα γίνει λεπτομερής ανάλυση των εργαλείων που χρησιμοποιήθηκαν για την υλοποίηση του αλγορίθμου, με έμφαση στις βιβλιοθήκες NumPy, SciPy και CuPy. Θα εξεταστεί η λειτουργία κάθε βιβλιοθήκης, τι περιλαμβάνει, καθώς και οι διαφορές τους, τόσο σε επίπεδο απόδοσης όσο και σε επίπεδο ευκολίας χρήσης. Τέλος, θα παρουσιαστεί μια συγκριτική αξιολόγηση της απόδοσης του αλγορίθμου στις διαφορετικές υλοποιήσεις (NumPy, SciPy, CuPy), εξετάζοντας πώς αυτές ανταποκρίνονται σε προβλήματα διαφορετικής κλίμακας.

5.2 NumPy & SciPy

Το NumPy και το SciPy είναι δύο από τις πιο γνωστές και θεμελιώδεις βιβλιοθήκες για υπολογισμούς επιστημονικών δεδομένων στην γλώσσα Python. Σχετικά με την βιβλιοθήκη NumPy, προσφέρει την δυνατότητα χρήσης πολυδιάστατων πινάκων αλλά και βασικές λειτουργίες της γραμμικής άλγεβρας, όπου την καθιστούν την πλέον απαραίτητη για υπολογισμούς αλλά και υλοποίησης αλγορίθμων γραμμικού προγραμματισμού και για αρκετούς αλγορίθμους βελτιστοποίησης. Όσον αφορά τη βιβλιοθήκη SciPy, είναι βασισμένη στην NumPy και προσφέρει επιπλέον εξειδικευμένες μαθηματικές συναρτήσεις.

Ένα πλεονέκτημα της βιβλιοθήκης SciPy έναντι της NumPy είναι ότι περιλαμβάνει έτοιμες υλοποιήσεις αλγορίθμων, όπως αυτή της μεθόδου Simplex. Με την εντολή `scipy.optimize.linprog`, μπορεί κάποιος να επιλύσει ένα πρόβλημα γραμμικού προγραμματισμού με τη μέθοδο Simplex. Με αυτόν τον τρόπο η επίλυση ενός τέτοιου είδους προβλήματος είναι πολύ πιο απλή από το να γραφτεί ο αλγόριθμος από την αρχή, όπως θα γινόταν με την χρήση της βιβλιοθήκης NumPy. Επιπλέον, τα αποτελέσματα της επίλυσης του προβλήματος με την βιβλιοθήκη SciPy θα είναι περισσότερο ακριβή με την διασφάλιση της Python από το να κατασκευαστεί ο αλγόριθμος εξ ολοκλήρου από τον χρήστη [14-15].

```python
import numpy as np

#Δημιουργία δύο πινάκων και εκτέλεση πράξεων
A = np.array([[1. 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])

#Array multiplication
C = np.dot(A, B)
print(C)
```

Σχήμα 5.1: Πολλαπλασιασμός δύο πινάκων με χρήση NumPy

```python
from scipy.optimize import linprog

# Συντελεστές της αντικειμενικής συνάρτησης
c = [-1, -2]

# Συντελεστές των περιορισμών
A = [[2, 1], [1, 1]]
b = [20, 16]

# Εύρεση της βέλτιστης λύσης με Simplex
result = linprog(c, A_ub=A, b_ub=b, method='simplex')

print(result)
```

Σχήμα 5.2: Υλοποίηση αλγορίθμου Simplex με χρήση SciPy

5.3 CuPy

Η **CuPy** είναι μια βιβλιοθήκη ανοιχτού κώδικα για την επιτάχυνση των υπολογισμών σε GPU χρησιμοποιώντας τη γλώσσα προγραμματισμού Python. Βασισμένη στιςβιβλιοθήκες **NumPy** και **SciPy**, η CuPy αξιοποιεί την υπολογιστική ισχύ των καρτών γραφικών (GPU)μέσω του **CUDA Toolkit**, περιλαμβάνοντας βιβλιοθήκες όπως cuBLAS, cuRAND, cuSOLVER, cuSPARSE, cuFFT, cuDNN και NCCL. Αυτή η αξιοποίηση της GPU επιτρέπει στην CuPy να επιταχύνει σημαντικά τις μαθηματικές πράξεις και άλλες λειτουργίες αλγορίθμων μέσω παράλληλης εκτέλεσης [16].

Υψηλή Απόδοση με Χρήση GPU

Η CuPy σχεδιάστηκε για να προσφέρει σημαντική αύξηση ταχύτητας στις πράξεις που εκτελούνται σε GPU, χρησιμοποιώντας ένα ευρύ φάσμα βιβλιοθηκών του CUDA Toolkit. Έχει αποδειχθεί ότι μπορεί να αυξήσει την ταχύτητα εκτέλεσης ορισμένων πράξεων πάνω από 100 φορές σε σύγκριση με την NumPy, ιδιαίτερα σε εφαρμογές που επωφελούνται από την παράλληλη επεξεργασία της GPU. Αυτό καθιστά την CuPy ιδανική για μεγάλους όγκους δεδομένων και επαναληπτικές πράξεις [16].

Συμβατότητα με NumPy και SciPy

Ένα από τα κύρια πλεονεκτήματα της CuPy είναι η υψηλή της συμβατότητα με τις βιβλιοθήκες **NumPy** και **SciPy**. Ο κώδικας που έχει γραφτεί για αυτές τις βιβλιοθήκες μπορεί συχνά ναεκτελείται με την CuPy με ελάχιστες ή καθόλου αλλαγές. Η μόνη απαίτηση είναι η αντικατάσταση τωνεισαγωγών της NumPy (`import numpy as np`) με τις αντίστοιχες της CuPy (`import cupy as cp`). Επιπλέον, για τις λειτουργίες της SciPy, υπάρχει υποστήριξη μέσω του `cupyx.scipy`. Η CuPy διατηρεί την ευχρηστία της NumPy, υποστηρίζοντας διάφορες μεθόδους, ευρετήρια, τύπους δεδομένων και broadcasting [16].

Προϋποθέσεις και Εγκατάσταση

Για την εκτέλεση της CuPy, είναι απαραίτητο το σύστημα να διαθέτει GPU που υποστηρίζει CUDA, καθώς και το **CUDA Toolkit** εγκατεστημένο στο σύστημα. Το CUDA Toolkit παρέχει τα απαραίτητα εργαλεία και βιβλιοθήκες για την εκτέλεση υπολογισμών στη GPU [16].

Πλεονεκτήματα και Περιορισμοί

Η CuPy είναι ιδανική για εφαρμογές που απαιτούν μεγάλους όγκους δεδομένων και επαναληπτικές πράξεις που μπορούν να επωφεληθούν από την παράλληλη επεξεργασία της GPU. Αν και προσφέρει σημαντικές επιταχύνσεις, η σωστή διαχείριση της μνήμης της GPU και η ελαχιστοποίηση των μεταφορών δεδομένων μεταξύ CPU και GPU είναι κρίσιμες για την αποφυγή μειώσεων της απόδοσης. Ειδικότερα, η CuPy αναλαμβάνει τη διαχείριση της μνήμης και των μεταφορών δεδομένων, αλλά δεν παρέχει το ίδιο επίπεδο ελέγχου και ευελιξίας που προσφέρουν οι αποκλειστικοί CUDA kernels [16].

```python
import cupy as cp

# Δημιουργία δύο πινάκων και εκτέλεση πράξεων στη GPU
A = cp.array([[1, 2], [3, 4]])
B = cp.array([[5, 6], [7, 8]])

# Πολλαπλασιασμός πινάκων στη GPU
C = cp.dot(A, B)

# Μεταφορά του αποτελέσματος στη CPU για εκτύπωση
print(C.get())
```

.Σχήμα 5.3: Πολλαπλασιασμός δύο πινάκων με χρήση CuPy

5.4 Περιβάλλον πειράματος

Το πείραμα διεξήχθη σε ένα ετερογενές σύστημα CPU/GPU. Η CPU που χρησιμοποιήθηκε είναι η AMD Ryzen 7 5800H, ένας 64-bit επεξεργαστής με βασική ταχύτητα ρολογιού 3.2 GHz και 16 MB L3 cache.
Το σύστημα διαθέτει 16 GB φυσικής μνήμης, με 3.29 GB διαθέσιμα κατά τη διάρκεια του πειράματος. Η συνολική εικονική μνήμη είναι 29 GB, με 12 GB διαθέσιμα. Η GPU που χρησιμοποιήθηκε είναι η NVIDIA GeForce RTX 3050 Laptop GPU με 4 GB αποκλειστικής μνήμης.
Το κεντρικό σύστημα και η GPU επικοινωνούν μέσω του PCIe 4.0 x8. Δεδομένου ότι κάθε lane του PCIe 4.0 έχει bandwidth 2 GB/s, το σύστημα μπορεί να μεταφέρει δεδομένα με ταχύτητες έως 16 GB/s.
Το λειτουργικό σύστημα είναι Windows 11, και η ανάπτυξη του λογισμικού για την GPU έγινε με χρήση της CUDA 12.4.

5.5 Ανάλυση βασικών στοιχείων του κώδικα

Η υλοποίηση του αλγορίθμου Simplex γίνεται με τη χρήση των βιβλιοθηκών NumPy και CuPy, για CPU και GPU αντίστοιχα. Η μελέτη αυτή εστιάζει στη σύγκριση της εκτέλεσης του αλγορίθμου Simplex σε δύο διαφορετικά περιβάλλοντα υπολογισμού: CPU και GPU. Ειδικότερα, συγκρίνονται οι βιβλιοθήκες NumPy, που είναι σχεδιασμένη για εκτέλεση σε επεξεργαστές (CPU), και CuPy, που είναι σχεδιασμένη για εκτέλεση σε κάρτες γραφικών (GPU). Στην παρούσα ανάλυση, θα εξεταστεί η υλοποίηση του αλγορίθμου Simplex χρησιμοποιώντας τις δύο αυτές βιβλιοθήκες. Η ανάλυση εστιάζει στις εξής πτυχές:

- **Εξέταση Δομών Δεδομένων**: Πιο συγκεκριμένα, γίνεται περιγραφή των δομών δεδομένων που χρησιμοποιούνται, όπως τα διανύσματα, οι πίνακες και οι πίνακες ειδικών δεικτών, συμπεριλαμβανομένων των ακριβών διαστάσεων και τύπων δεδομένων.
- **Περιγραφή Μεθόδων Αναλογιών**: Αναλύονται οι μέθοδοι `compute_ratios_numpy` και `compute_ratios_cupy` που χρησιμοποιούν τις βιβλιοθήκες NumPy και CuPy αντίστοιχα για τον υπολογισμό των αναλογιών σε CPU και GPU.
- **Περιγραφή Μεθόδων Εκτέλεσης**: Αναλύεται η μέθοδος εκτέλεσης Simplex που εκτελείται και για NumPy, `SimplexLargeScale_numpy` και για CuPy, `SimplexLargeScale_cupy`.
- **Διαχείριση και μεταφορά δεδομένων**: Εξετάζεται ο τρόπος αποθήκευσης και διαχείρισης δεδομένων για CPU και GPU, καθώς και οι διαδικασίες μεταφοράς δεδομένων από CPU σε GPU.
- **Σύγκριση Εκτέλεσης**: Αξιολογείται η εκτέλεση και των δύο εκδόσεων του κώδικα για δεδομένα μικρής και μεγάλης κλίμακας, ώστε να υπάρχει ασφαλής αξιολόγηση.

5.5.1 Δομές Δεδομένων

Για την υλοποίηση της μεθόδου Simplex στον κώδικα χρησιμοποιήθηκαν διάφορες δομές δεδομένων, με σκοπό την αποθήκευση και επεξεργασία των δεδομένων. Πιο συγκεκριμένα χρησιμοποιήθηκαν οι εξής:

Διανύσματα (Vectors): Χρησιμοποιούνται για την αποθήκευση των τιμών των μεταβλητών, των τιμών των συντελεστών της αντικειμενικής συνάρτησης και των λόγων κατά την εκτέλεση του αλγορίθμου, επιτρέποντας γρήγορες αριθμητικές πράξεις. Τα διανύσματα επιτρέπουν γρήγορες αριθμητικές πράξεις. Για παράδειγμα, το διάνυσμα των συντελεστών της αντικειμενικής συνάρτησης μπορεί να έχει διαστάσεις n×1, όπου n είναι ο αριθμός των μεταβλητών.

```
cbT = c[nonbasicSize:]
cnT = c[:nonbasicSize]
```

Σχήμα 5.4:Δημιουργία διάνυσματος από τον πίνακα *c*

Πίνακες (Matrices): Χρησιμοποιούνται για την αναπαράσταση των περιορισμών και των συντελεστών της αντικειμενικής συνάρτησης. Πιο συγκεκριμένα, οι πίνακες είναι δύο διαστάσεων και περιλαμβάνουν τους συντελεστές των περιορισμών. Οι πίνακες έχουν διαστάσεις m×n , όπου m είναι ο αριθμός των περιορισμών και n είναι ο αριθμός των μεταβλητών.

```
A = np.random.rand(n_constraints, n_variables)
b = np.random.rand(n_constraints)
c = np.random.rand(n_variables)
```

Σχήμα 5.5: Δημιουργία τυχαίου διανυσματικού πίνακα A,b,c με διαστάσεις n_constraints x n_variables

Πίνακες Δεικτών (Index Arrays): Χρησιμοποιούνται για την παρακολούθηση και τη διαχείριση των βασικών και μη βασικών μεταβλητών. Αυτοί οι πίνακες επιτρέπουν την ανανέωση της βάσης κατά την εκτέλεση του αλγορίθμου. Περιλαμβάνουν δείκτες που αντιστοιχούν στις θέσεις των βασικών και μη βασικών μεταβλητών στις πίνακες δεδομένων.

```
cindx = np.arange(len(c))
```

Σχήμα 5.6: Δημιουργία διανύσματος δεικτών cindx για την παρακολούθηση των θέσεων των στοιχείων του πίνακα κόστους *c*.

5.5.2 Μέθοδοι Αναλογιών

Οι μέθοδοι αναλογιών είναι κρίσιμες για την επιλογή της εξερχόμενης μεταβλητής κατά την εκτέλεση της μεθόδου Simplex. Οι αναλογίες υπολογίζονται για να εντοπιστεί ποιος περιορισμός θα καταστεί ενεργός πρώτος, καθώς αυξάνεται η τιμή της εισερχόμενης μεταβλητής, οδηγώντας στην έξοδο μιας μεταβλητής από τη βάση. Αυτό σημαίνει ότι, καθώς η εισερχόμενη μεταβλητή αυξάνεται, η τιμή της μπορεί να επηρεάσει διάφορους περιορισμούς. Κάθε περιορισμός μπορεί να επιβάλει έναν ορισμένο ανώτατο όριο στην τιμή της εισερχόμενης μεταβλητής πριν αρχίσει να παραβιάζεται. Οι αναλογίες υπολογίζονται ως το πηλίκο του διαθέσιμου «χώρου» του περιορισμού προς την αλλαγή που προκαλείται από την αύξηση της εισερχόμενης μεταβλητής. Η μικρότερη έγκυρη αναλογία δείχνει ποιος περιορισμός θα φτάσει πρώτος στο όριό του, καθορίζοντας έτσι ποια μεταβλητή θα πρέπει να εξέλθει από τη βάση, ώστε να παραμείνει η λύση εντός των επιτρεπτών περιορισμών. Με αυτή τη διαδικασία, μπορούμε να διασφαλίσουμε ότι η λύση παραμένει εφικτή κατά την πορεία του αλγορίθμου Simplex.

Μέθοδος `compute_ratios_numpy`: Υλοποιείται χρησιμοποιώντας τη βιβλιοθήκη NumPy και υπολογίζει τις αναλογίες για τον εντοπισμό της εξερχόμενης μεταβλητής κατά την εκτέλεση του αλγορίθμου Simplex σε περιβάλλον CPU. Στη μέθοδο, ο όρος `Binv @ N` αντιπροσωπεύει τον πολλαπλασιασμό του αντίστροφου του βασικού πίνακα με τον πίνακα των μη βασικών μεταβλητών. Το διάνυσμα ratios περιλαμβάνει τις αναλογίες που υπολογίζονται για κάθε μη βασική μεταβλητή, και η μέθοδος επιστρέφει το δείκτη της μεταβλητής με την ελάχιστη έγκυρη αναλογία, η οποία καθορίζει ποια μεταβλητή θα εξέλθει από τη βάση.

```
def compute_ratios_numpy(bHat, Ahat):
    ratios = np.zeros_like(bHat)
    for i in range(bHat.size):
        Bval = bHat[i]
        Aval = Ahat[i]
        if Aval > 0:
            ratios[i] = Bval / Aval
        else:
            ratios[i] = np.inf
    return ratios
```

Σχήμα 5.7: Μέθοδος *compute_rations_numpy*

Μέθοδος `compute_ratios_cupy`: Είναι αντίστοιχη με την `compute_ratios_numpy`, αλλά εκτελείται με τη βιβλιοθήκη CuPy για εκτέλεση στην GPU, εκμεταλλευόμενη την παράλληλη επεξεργασία της GPU για ταχύτερο υπολογισμό των αναλογιών. Οι αναλογίες υπολογίζονται με τον ίδιο τρόπο, αλλά η επιτάχυνση της επεξεργασίας επιτυγχάνεται μέσω της GPU.

```
def compute_ratios_cupy(bHat, Ahat):
    ratios = cp.zeros_like(bHat)
    for i in range(bHat.size):
        Bval = bHat[i]
        Aval = Ahat[i]
        if Aval > 0:
            ratios[i] = Bval / Aval
        else:
            ratios[i] = cp.inf
    return ratios
```

Σχήμα 5.8: Μέθοδος *compute_ratios_cupy*

5.5.3 Δομή Αλγορίμου Simplex

Η διαδικασία του κώδικα ξεκινά με την εισαγωγή των δεδομένων, όπου ο πίνακας A αντιπροσωπεύει τους περιορισμούς του προβλήματος, το διάνυσμα b περιέχει τις τιμές των περιορισμών και το διάνυσμα c τους συντελεστές της αντικειμενικής συνάρτησης. Ο αλγόριθμος χωρίζει τις μεταβλητές σε βασικές και μη βασικές, με τον πίνακα B να αντιπροσωπεύει τις στήλες του A που αντιστοιχούν στις βασικές μεταβλητές, και τον πίνακα N να αντιστοιχεί στις μη βασικές. Κατά την εκτέλεση κάθε επανάληψης, υπολογίζεται η αντίστροφη του πίνακα βάσης B^{-1}, απαραίτητη για τον υπολογισμό των νέων τιμών των βασικών μεταβλητών. Στην υλοποίηση με **NumPy**, η αντιστροφή γίνεται στη CPU με τη μέθοδο `np.linalg.inv(B)`, ενώ στην υλοποίηση με **CuPy**, η διαδικασία εκτελείται στη GPU με τη μέθοδο `cp.linalg.inv(B)`.

Μετά την αντιστροφή του πίνακα, υπολογίζονται οι τιμές των βασικών μεταβλητών $\hat{b} = B^{-1} * b$ καθώς και οι τιμές $\widehat{c_n} = c_n - y^T * N$, όπου $y^T = C_B^T * B^{-1}{}_B$Οι τιμές $\widehat{c_n}$ καθορίζουν αν έχει βρεθεί η βέλτιστη λύση. Αν όλες οι τιμές είναι μη αρνητικές, η διαδικασία τερματίζεται, καθώς η λύση είναι βέλτιστη. Αν υπάρχουν αρνητικές τιμές, επιλέγεται η μεταβλητή με το μεγαλύτερο αρνητικό $\widehat{c_n}$ ως η εισερχόμενη μεταβλητή, και γίνεται υπολογισμός των αναλογιών των βασικών μεταβλητών για να καθοριστεί ποια μεταβλητή θα βγει από τη βάση. Η διαδικασία αυτή γίνεται επαναληπτικά, με ενημέρωση των πινάκων B και N, μέχρις ότου όλες τιμές του $\widehat{c_n}$ να γίνουν μη αρνητικές, οπότε και ο αλγόριθμος επιστρέφει τη βέλτιστη λύση.

5.5.4 Μέθοδοι Εκτέλεσης

Μέθοδος *`SimplexLargeScale_numpy`*: Εκτελεί τον αλγόριθμο Simplex χρησιμοποιώντας τη βιβλιοθήκη NumPy. Η μέθοδος αυτή διαχειρίζεται δεδομένα, εκτελεί επαναληπτικά τον αλγόριθμο και υπολογίζει τη βέλτιστη λύση. Στον κώδικα, οι παράμετροι *A, b*, και *c* είναι τα δεδομένα εισόδου, που αντιπροσωπεύουν τον πίνακα περιορισμών, το διάνυσμα περιορισμών και το διάνυσμα συντελεστών της αντικειμενικής συνάρτησης αντίστοιχα. Οι πίνακες *B* και *N* αντιπροσωπεύουν τις βασικές και μη βασικές μεταβλητές. Η μέθοδος εκτελεί επαναληπτικά βήματα, ενημερώνοντας τις βασικές μεταβλητές έως ότου βρεθεί η βέλτιστη λύση. Κατά την εκτέλεση του Simplex, υπολογίζονται οι τιμές `cnHat`, οι οποίες αντιπροσωπεύουν τη διαφορά κόστους μεταξύ των βασικών και μη βασικών μεταβλητών. Αυτές οι τιμές είναι κρίσιμες για τον καθορισμό της βέλτιστης λύσης, καθώς αν όλες οι τιμές `cnHat` είναι μη αρνητικές, η διαδικασία τερματίζει με επιτυχία

Μέθοδος *`SimplexLargeScale_cupy`*: Η μέθοδος *`SimplexLargeScale_cupy`* εκτελεί τον αλγόριθμο Simplex χρησιμοποιώντας τη βιβλιοθήκη CuPy, η οποία είναι σχεδιασμένη για εκτέλεση σε GPU. Αυτή η μέθοδος είναι παρόμοια με τη *`SimplexLargeScale_numpy`*, αλλά εκμεταλλεύεται την παράλληλη επεξεργασία της GPU για ταχύτερη εκτέλεση. Η GPU επιτρέπει την ταχύτερη επεξεργασία μεγάλων όγκων δεδομένων, βελτιώνοντας την αποδοτικότητα του αλγορίθμου σε μεγάλες κλίμακες.

5.5.5 Διαχείριση και μεταφορά δεδομένων

Η διαχείριση και μεταφορά δεδομένων είναι κρίσιμα σημεία στην εκτέλεση του αλγορίθμου Simplex, ειδικά όταν η εκτέλεση γίνεται σε διαφορετικά υπολογιστικά περιβάλλοντα όπως η CPU και η GPU. Η σωστή διαχείριση των δεδομένων είναι απαραίτητη για την επίτευξη αποδοτικότητας και την ελαχιστοποίηση των καθυστερήσεων που μπορεί να προκύψουν κατά τη μεταφορά δεδομένων μεταξύ διαφορετικών υπολογιστικών μονάδων.

Διαχείριση Δεδομένων σε CPU: Στη βιβλιοθήκη NumPy, τα δεδομένα διαχειρίζονται ως πίνακες και διανύσματα που αποθηκεύονται στη μνήμη RAM της CPU. Οι υπολογισμοί γίνονται απευθείας στη CPU, και η διαχείριση της μνήμης γίνεται από τη βιβλιοθήκη NumPy, η οποία παρέχει αποδοτικές λειτουργίες για μαθηματικούς υπολογισμούς.

```
# Αρχικοποίηση δεδομένων σε NumPy
A = np.random.rand(m, n)  # Πίνακας περιορισμών
b = np.random.rand(m)      # Διάνυσμα περιορισμών
c = np.random.rand(n)      # Διάνυσμα συντελεστών αντικειμενικής συνάρτησης

# Διαχείριση δεδομένων
B = np.eye(m)  # Βασικός πίνακας
N = A  # Μη βασικός πίνακας
Binv = np.linalg.inv(B)  # Αντίστροφος του βασικού πίνακα
```

Σχήμα 5.9: Διαχείριση δεδομένων

Διαχείριση Δεδομένων σε GPU με τη χρήση CuPy: Στη βιβλιοθήκη CuPy, τα δεδομένα αποθηκεύονται και διαχειρίζονται στη μνήμη της GPU, γεγονός που επιτρέπει ταχύτερη επεξεργασία μέσω παράλληλων υπολογισμών. Η μεταφορά δεδομένων από τη CPU στη GPU και αντίστροφα γίνεται μέσω της CuPy, η οποία παρέχει λειτουργίες για τη μετατροπή NumPy arrays σε CuPy arrays και αντίστροφα.

```
# Μεταφορά δεδομένων στη GPU με CuPy
A_gpu = cp.array(A)  # Μεταφορά πίνακα περιορισμών στη GPU
b_gpu = cp.array(b)  # Μεταφορά διανύσματος περιορισμών στη GPU
c_gpu = cp.array(c)  # Μεταφορά διανύσματος συντελεστών αντικειμενικής συνάρτησης στη GPU

# Διαχείριση δεδομένων στη GPU
B_gpu = cp.eye(m)  # Βασικός πίνακας στη GPU
N_gpu = A_gpu  # Μη βασικός πίνακας στη GPU
Binv_gpu = cp.linalg.inv(B_gpu)  # Αντίστροφος του βασικού πίνακα στη GPU
```

Σχήμα 5.10: Μεταφορά δεδομένων στη GPU

Μεταφορά Δεδομένων: Κατά τη μεταφορά δεδομένων από τη CPU στη GPU, χρησιμοποιούνται οι συναρτήσεις `cupy.asarray()` και `cupy.get_array_module()` για την ακριβή μετατροπή και αποθήκευση των δεδομένων στη GPU. Η διαδικασία μεταφοράς πρέπει να γίνεται προσεκτικά, καθώς η αποδοτικότητα της μεταφοράς μπορεί να επηρεάσει την συνολική απόδοση της εκτέλεσης του αλγορίθμου. Η διαδικασία μεταφοράς μπορεί να περιλαμβάνει τη δημιουργία αντιγράφων των δεδομένων στην GPU και τη διαχείριση τους κατά τη διάρκεια των υπολογισμών.

5.5.6 Σύγκριση εκτελέσεων

Η σύγκριση της εκτέλεσης του αλγορίθμου Simplex σε CPU και GPU είναι απαραίτητη για την αξιολόγηση της αποδοτικότητας των δύο αυτών υπολογιστικών περιβαλλόντων, ιδιαίτερα όταν εκτελούνται εργασίες μεγάλης κλίμακας.

NumPy: Η εκτέλεση του Simplex σε CPU μέσω της βιβλιοθήκης NumPy εκμεταλλεύεται την ευελιξία και τη σταθερότητα που προσφέρει η CPU, ειδικά για προβλήματα μικρής και μεσαίας κλίμακας. Η NumPy είναι βελτιστοποιημένη για πράξεις σε πίνακες και διανύσματα, προσφέροντας σταθερές επιδόσεις. Στο παράδειγμα, ο χρόνος εκτέλεσης του Simplex σε CPU μετράται με ακρίβεια, παρέχοντας πληροφορίες για την αποδοτικότητα της μεθόδου σε αυτό το περιβάλλον.

```
start_time_np = time.time()
result_np = SimplexLargeScale_numpy(A_large, b_large, c_large)
end_time_np = time.time()
execution_time_np = end_time_np - start_time_np
print(f"Χρόνος εκτέλεσης σε CPU (NumPy): {execution_time_np:.4f} δευτερόλεπτα")
```

Σχήμα 5.11: Χρονομέτρηση NumPy εκτέλεσης

CuPy: Η εκτέλεση του Simplex σε GPU μέσω της βιβλιοθήκης CuPy εκμεταλλεύεται την υψηλή υπολογιστική ισχύ και την παράλληλη επεξεργασία της GPU. Αυτή η προσέγγιση είναι ιδιαίτερα αποδοτική για προβλήματα μεγάλης κλίμακας, όπου οι καθυστερήσεις στη μεταφορά δεδομένων αντισταθμίζονται από την ταχύτερη εκτέλεση των πράξεων. Στο παράδειγμα, ο χρόνος εκτέλεσης του Simplex σε GPU καταγράφεται και συγκρίνεται με τον αντίστοιχο χρόνο της CPU, αναδεικνύοντας την απόδοση της GPU σε περιβάλλοντα μεγάλης κλίμακας.

```
start_time_cupy = time.time()
result_cupy = SimplexLargeScale_cupy(A_large, b_large, c_large)
end_time_cupy = time.time()
execution_time_cupy = end_time_cupy - start_time_cupy
print(f"Χρόνος εκτέλεσης σε GPU (CuPy): {execution_time_cupy:.4f} δευτερόλεπτα")
```

Σχήμα 5.12: Χρονομέτρηση εκτέλεσης CuPy

5.6 Ανάλυση Αποτελεσμάτων

Η υλοποίηση του αλγορίθμου Simplex που περιγράφεται στην παρούσα μελέτη εφαρμόζεται απευθείας με δεδομένα που είναι ήδη σε κανονική μορφή, δηλαδή με περιορισμούς που είναι γραμμικές εξισώσεις και όχι ανισώσεις. Ο κώδικας δεν περιλαμβάνει διαδικασία μετατροπής ανισώσεων σε εξισώσεις (π.χ., δεν προσθέτει μεταβλητές slack για τη μετατροπή των ανισώσεων σε εξισώσεις).

Για να διασφαλιστεί η σωστή εκτέλεση του αλγορίθμου, τα δεδομένα εισόδου πρέπει να είναι σε κανονική μορφή πριν από την εφαρμογή του Simplex. Εάν τα δεδομένα δεν είναι ήδη σε αυτή τη μορφή, απαιτείται αρχική μετατροπή για να προετοιμαστούν κατάλληλα.

Δεδομένα Εισόδου: Τα δεδομένα εισόδου περιλαμβάνουν τον πίνακα περιορισμών `A`, το διάνυσμα περιορισμών `b`, και το διάνυσμα συντελεστών της αντικειμενικής συνάρτησης `c`. Αυτά τα δεδομένα είναι κρίσιμα για την εκτέλεση του αλγορίθμου και τη σωστή αξιολόγηση της λύσης.
Πειραματική Αξιολόγηση: Για την πειραματική αξιολόγηση της υλοποίησης, έχουν δημιουργηθεί περισσότερα από 1000 προβλήματα γραμμικού προγραμματισμού (LP) με τυχαίες τιμές. Τα προβλήματα αναπτύσσονται με προοδευτική αύξηση του μεγέθους τους, με το μεγαλύτερο πρόβλημα να περιλαμβάνει πίνακα περιορισμών διαστάσεων 9000 x 9000. Η δημιουργία αυτών των προβλημάτων αποσκοπεί στη δοκιμή της ανθεκτικότητας και της αποδοτικότητας του αλγορίθμου σε διάφορες κλίμακες.
Υλοποίηση και Σύγκριση: Ο αλγόριθμος Simplex υλοποιείται τόσο σε περιβάλλον CPU όσο και GPU, χρησιμοποιώντας τις βιβλιοθήκες NumPy και CuPy αντίστοιχα. Η αξιολόγηση της απόδοσης επικεντρώνεται στην σύγκριση των χρόνων εκτέλεσης και της αποδοτικότητας μεταξύ των δύο περιβαλλόντων.
Διαδικασία Παρακολούθησης και Ανάλυσης: Η διαδικασία εκτέλεσης του αλγορίθμου παρακολουθείται προσεκτικά για διαφορετικά μεγέθη προβλημάτων. Η ανάλυση των αποτελεσμάτων περιλαμβάνει:

- **Χρόνος Εκτέλεσης:** Μέτρηση του χρόνου που απαιτείται για την εκτέλεση του αλγορίθμου σε διάφορα μεγέθη προβλημάτων.
- **Απόδοση και Εξοικονόμηση Πόρων:** Σύγκριση της αποδοτικότητας της υλοποίησης σε CPU και GPU, με εστίαση στη χρήση πόρων όπως η μνήμη και η επεξεργαστική ισχύς.
- **Επίδραση Κλίμακας:** Αξιολόγηση της επίδρασης της κλίμακας του προβλήματος στην απόδοση, αναλύοντας εάν ο χρόνος εκτέλεσης και η αποδοτικότητα κλιμακώνονται γραμμικά ή εκθετικά με την αύξηση του μεγέθους των δεδομένων.

Τα αποτελέσματα της ανάλυσης παρέχουν χρήσιμες πληροφορίες για την αξιολόγηση της αποτελεσματικότητας της υλοποίησης του αλγορίθμου Simplex σε διάφορα υπολογιστικά περιβάλλοντα, καθώς και για την κατανόηση της επίδρασης της κλίμακας στο χρόνο εκτέλεσης και την αποδοτικότητα.

5.6.1 Επίδοση CuPy έναντι NumPy

Η σύγκριση της απόδοσης του αλγορίθμου Simplex, όπως υλοποιείται σε NumPy και CuPy, ανέδειξε σημαντικές διαφορές στην ταχύτητα εκτέλεσης καθώς αυξάνονταν τα μεγέθη των δεδομένων. Τα αποτελέσματα που προκύπτουν από τις μετρήσεις χρόνου εκτέλεσης και την αντιστροφή της βάσης παρουσιάζονται παρακάτω. Από αυτό το σημείο και έπειτα, η αναφορά στη λέξη δεδομένα θα εννοεί τη κλίμακα του προβλήματος, δηλαδή τον αριθμό μεταβλητών και περιορισμών.

Σχήμα 5.13: Επιτάχυνση αλγορίθμου με χρήση CuPy

Τα δεδομένα δείχνουν ότι η CuPy, η οποία αξιοποιεί την επεξεργασία GPU, αποδείχθηκε σαφώς ταχύτερη από τη NumPy, η οποία εκτελεί σε CPU, ιδιαίτερα για μεγαλύτερα μεγέθη δεδομένων. Ο παρακάτω πίνακας παρουσιάζει την αναλογία ταχύτητας (speedup) της CuPy σε σχέση με τη NumPy για διάφορα μεγέθη δεδομένων. Συγκεκριμένα, τα αποτελέσματα του κώδικα για τη μέση επιτάχυνση του αλγορίθμου δείχνουν ότι υπάρχει ισορροπία για μεταβλητές έως και 2000. Αυτό συμβαίνει διότι για μικρής κλίμακας δεδομένα, ο επεξεργαστής αποδίδει καλύτερα ενώ η κάρτα γραφικών απαιτεί μεταφορά δεδομένων χωρίς να αξιοποιεί πλήρως την υπολογιστική της ισχύ. Επίσης, η μεταφορά δεδομένων από την CPU στην GPU μπορεί να επηρεάζει την απόδοση σε μικρότερα μεγέθη δεδομένων, καθώς η επιβάρυνση από αυτή τη μεταφορά δεν δικαιολογεί πάντα τα οφέλη της GPU.

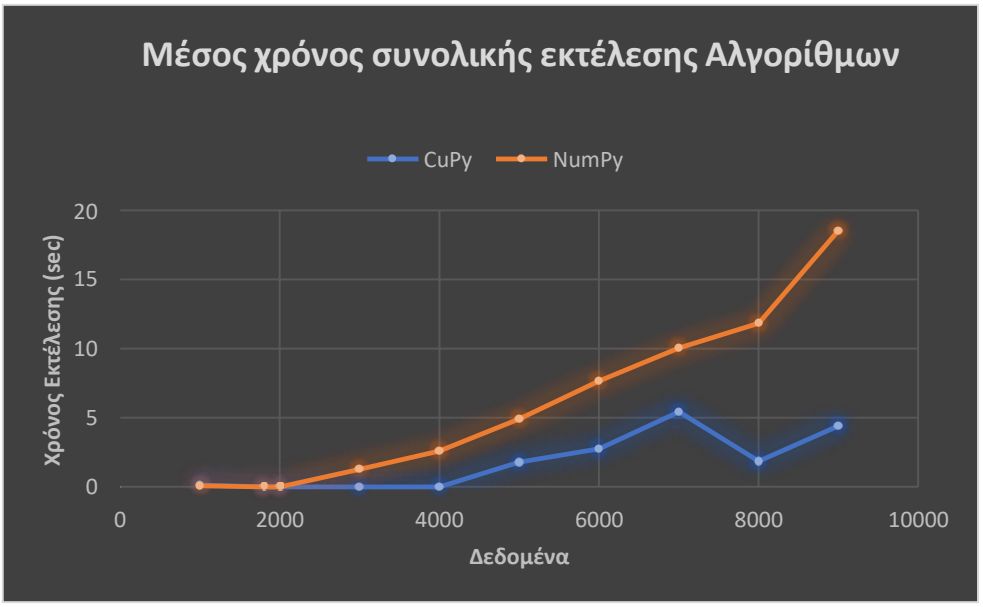
Από τα 2000 έως και τα 6000 δεδομένα, παρατηρείται αύξηση στην επιτάχυνση της CuPy στην εκτέλεση του κώδικα, αν και η επιτάχυνση παρουσιάζει αστάθεια. Σε μεγέθη δεδομένων γύρω από 5000 μεταβλητές, η επιτάχυνση μειώνεται, πιθανώς λόγω σφαλμάτων στρογγυλοποίησης και αριθμητικής ακρίβειας. Τα σφάλματα στρογγυλοποίησης προκύπτουν όταν οι υπολογισμοί με περιορισμένη ακρίβεια αριθμών επηρεάζουν την ποιότητα των αποτελεσμάτων, ενώ η αριθμητική ακρίβεια επηρεάζει την ποιότητα των υπολογισμών. Επιπλέον, η μεταφορά μεγάλων όγκων δεδομένων από την CPU στην GPU μπορεί να είναι χρονοβόρα και να επηρεάζει την συνολική απόδοση.

Από τα 6000 έως τα 10000 δεδομένα, η αναλογία επιτάχυνσης παραμένει ασυνεχής, αλλά η επιτάχυνση αρχίζει να αυξάνεται σημαντικά με την αύξηση των δεδομένων. Αυτό μπορεί να οφείλεται στη βελτίωση της αξιοποίησης των πόρων της GPU καθώς η κλίμακα των δεδομένων μεγαλώνει, επιτρέποντας καλύτερη εκμετάλλευση των επικαλύψεων εργασιών και των υπολογιστικών δυνατοτήτων της GPU. Ταυτόχρονα, τα θέματα μνήμης και οι επικαλύψεις εργασιών «παίζουν» σημαντικό ρόλο στην επίδοση, καθώς η GPU διαχειρίζεται καλύτερα τα δεδομένα όταν η κλίμακα τους είναι αρκετά μεγάλη για να εκμεταλλευτεί πλήρως τους παράλληλους υπολογισμούς.

Εν κατακλείδι, ενώ η CuPy δείχνει σαφή βελτίωση στην απόδοση για μεγάλες κλίμακες δεδομένων, η αστάθεια που παρατηρείται μπορεί να οφείλεται σε μια συνδυασμένη επίδραση σφαλμάτων στρογγυλοποίησης, περιορισμένης αριθμητικής ακρίβειας, αναποτελεσματικής διαχείρισης της μνήμης και μεταφοράς δεδομένων.

5.6.2 Μέσος χρόνος εκτέλεσης

Αναφορικά με τη χρονική διαφορά μεταξύ των δύο εκτελέσεων, παρατηρείται μια ισορροπία για 2000 δεδομένα, όπου οι μέσοι χρόνοι εκτέλεσης δείχνουν αρχικά υπεροχή της NumPy έκδοσης σε δεδομένα της τάξης των 200. Συγκεκριμένα, ο μέσος χρόνος εκτέλεσης του προγράμματος με NumPy είναι 0,002 δευτερόλεπτα, ενώ με CuPy απαιτείται περισσότερος χρόνος κατά 0,0675 δευτερόλεπτα. Μέχρι τα 1800 δεδομένα, η NumPy έκδοση χρειάζεται λιγότερο χρόνο για εκτέλεση, με παρόμοιες διαφορές όπως με τα 200 δεδομένα. Όταν ο αριθμός των δεδομένων υπερβαίνει τα 2000, η CuPy έκδοση του κώδικα απαιτεί σημαντικά λιγότερο χρόνο εκτέλεσης. Ειδικότερα, για 4000 δεδομένα, η διαφορά στον χρόνο εκτέλεσης είναι περίπου 2,2 δευτερόλεπτα, σημειώνοντας την πρώτη σημαντική διαφορά στην εκτέλεση των δύο εκδοχών του αλγορίθμου Simplex. Από αυτό το σημείο μέχρι το μέγιστο σημείο που εξετάστηκε, οι διαφορές μεταξύ των δύο εκδοχών αυξάνονται, με τον χρόνο διαφοράς να φτάνει μέχρι 5 δευτερόλεπτα (5,5 δευτερόλεπτα μέσος χρόνος εκτέλεσης για CuPy και 10,5 δευτερόλεπτα για NumPy). Για δεδομένα έως 10000, οι διαφορές φτάνουν έως και 14 δευτερόλεπτα μέσου χρόνου εκτέλεσης του κώδικα, όπως φαίνεται στο παρακάτω σχήμα.



Σχήμα 5.14: Μέσος χρόνος συνολικής εκτέλεσης NumPy & CuPy

Χρόνος Αντιστροφής Πίνακα

Στην παρούσα ενότητα αναλύεται ο χρόνος που απαιτείται για την αντιστροφή του πίνακα, όπως υπολογίζεται μέσω των μεθόδων NumPy και CuPy. Η διαδικασία αυτή είναι κρίσιμη για την απόδοση των αλγορίθμων βελτιστοποίησης, όπως ο αλγόριθμος Simplex για μεγάλης κλίμακας προβλήματα.

Πολυπλοκότητα Αντιστροφής Πίνακα

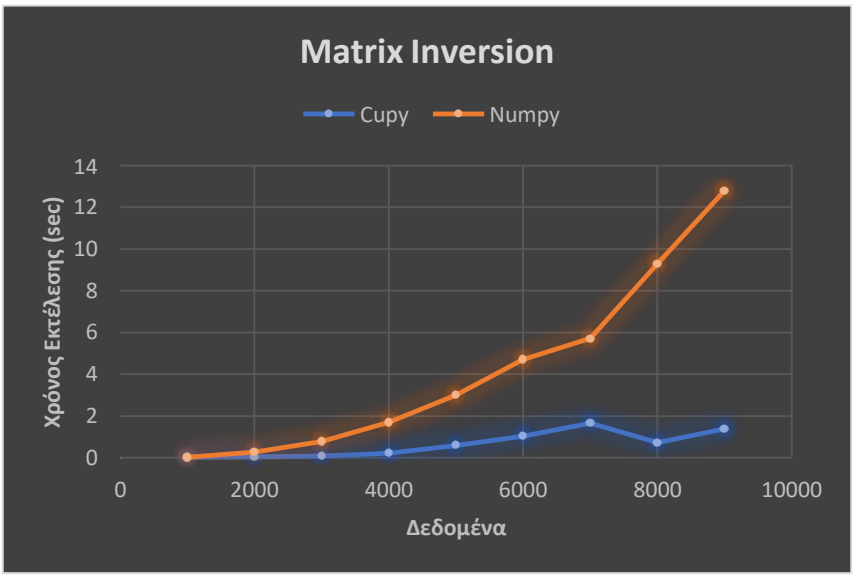
Η διαδικασία αντιστροφής ενός τετραγωνικού πίνακα n×n γενικά έχει πολυπλοκότητα της τάξης $O(n^3)$. Αυτό σημαίνει ότι ο χρόνος υπολογισμού της αντιστροφής αυξάνεται κυβικά με την αύξηση της διάστασης του πίνακα. Συγκεκριμένα, αλγόριθμοι όπως η ανάλυση LU Decomposition ή η μέθοδος Gauss-Jordan, που χρησιμοποιούνται για την αντιστροφή πίνακα, απαιτούν υπολογισμούς που είναι αναλογικοί με τον κύβο της διάστασης του πίνακα [6].

Υπολογισμός Χρόνου Αντιστροφής Πίνακα με NumPy: Στον κώδικα που αναλύεται, η αντιστροφή του πίνακα πραγματοποιείται μέσω της συνάρτησης `np.linalg.inv()` της βιβλιοθήκης NumPy, η οποία υπολογίζει τον αντίστροφο ενός πίνακα στην CPU. Ο χρόνος υπολογισμού της αντιστροφής καταγράφηκε για διαφορετικά μεγέθη πινάκων (1000x1000 έως 9000x9000) και η μέση τιμή του χρόνου καταγράφηκε για κάθε μέγεθος. Τα αποτελέσματα δείχνουν ότι ο χρόνος που απαιτείται για την αντιστροφή του πίνακα αυξάνεται εκθετικά με την αύξηση της διάστασης του πίνακα. Αυτό είναι αναμενόμενο λόγω της πολυπλοκότητας $\boldsymbol{O(n^3)}$. Ειδικότερα, για πίνακες διαστάσεων 1000x1000 έως 9000x9000, η χρονική αύξηση ακολουθεί μια αναμενόμενη αναλογία, με μεγαλύτερους πίνακες να απαιτούν σημαντικά περισσότερη χρονική διάρκεια για την αντιστροφή.

Υπολογισμός Χρόνου Αντιστροφής Πίνακα με CuPy: Για την CuPy, η διαδικασία αντιστροφής εκτελείται μέσω της συνάρτησης `cp.linalg.inv()` στην GPU. Η CuPy αξιοποιεί την παράλληλη επεξεργασία των πυρήνων της GPU, πράγμα που ενδέχεται να προσφέρει σημαντική βελτίωση στην απόδοση, ειδικά για μεγάλους πίνακες. Ο χρόνος υπολογισμού της αντιστροφής περιλαμβάνει επίσης την μεταφορά δεδομένων μεταξύ CPU και GPU, καθώς και τη διαχείριση της μνήμης GPU. Τα αποτελέσματα δείχνουν ότι η χρήση της GPU μπορεί να μειώσει σημαντικά τον χρόνο αντιστροφής, σε σύγκριση με την CPU. Ωστόσο, οι χρόνοι μεταφοράς δεδομένων και διαχείρισης της μνήμης GPU έχουν επίσης επίδραση στον συνολικό χρόνο εκτέλεσης. Παρόλα αυτά, η συνολική ταχύτητα της CuPy στην αντιστροφή των μεγάλων πινάκων είναι συνήθως καλύτερη από αυτήν της NumPy λόγω της ικανότητας της GPU να εκτελεί παράλληλους υπολογισμούς.

Συγκριτική Ανάλυση: Η σύγκριση των χρόνων εκτέλεσης μεταξύ NumPy και CuPy δείχνει ότι η CuPy συνήθως επιτυγχάνει ταχύτερους χρόνους για μεγάλους πίνακες. Ειδικότερα, η CuPy επιτυγχάνει σημαντική βελτίωση στις μεγαλύτερες διαστάσεις πινάκων (π.χ., 8000 x 8000 και 9000 x 9000), όπου η χρήση της GPU είναι πιο αποτελεσματική. Η ταχύτητα της CuPy μπορεί να είναι έως και 4 φορές μεγαλύτερη σε σύγκριση με την NumPy, λόγω της παράλληλης επεξεργασίας που προσφέρει η GPU.

Εντούτοις, είναι σημαντικό να σημειωθεί ότι η αρχική μεταφορά δεδομένων στη GPU και η διαχείριση της μνήμης επηρεάζουν τον συνολικό χρόνο εκτέλεσης. Ειδικά για μικρότερους πίνακες, το υπερβολικό «overhead» από τη μεταφορά δεδομένων μπορεί να μειώσει τα οφέλη από τη χρήση της GPU.



Σχήμα 5.15: Μέσος χρόνος εκτέλεσης αντιστροφής βάσης σε Numpy & CuPy

5.6.3 Σύγκριση εκτέλεσης με SciPy

Η βιβλιοθήκη SciPy, η οποία είναι ένα ισχυρό εργαλείο για επιστημονικούς υπολογισμούς στην Python, περιλαμβάνει την υλοποίηση του αλγορίθμου Simplex μέσω της μεθόδου HiGHS. Η HiGHS είναι μια σύγχρονη και βελτιστοποιημένη υλοποίηση για τη λύση γραμμικών προγραμμάτων (LP), σχεδιασμένη για να προσφέρει υψηλή απόδοση και ακρίβεια. Το πακέτο SciPy ενσωματώνει την HiGHS μέσω του module `scipy.optimize`, παρέχοντας μια έτοιμη και αποδοτική λύση για την επίλυση προβλημάτων γραμμικού προγραμματισμού με το αλγόριθμο Simplex.

Αντιθέτως, οι υλοποιήσεις Simplex με NumPy και CuPy χρησιμοποιούν προσαρμοσμένους αλγόριθμους Simplex που αναπτύχθηκαν για συγκεκριμένες ανάγκες του έργου. Ο κώδικας που χρησιμοποιεί NumPy και CuPy περιλαμβάνει μια δική του υλοποίηση του αλγορίθμου Simplex, η οποία βασίζεται σε αντιστροφή πίνακα και υπολογισμούς που είναι χαρακτηριστικοί της συγκεκριμένης υλοποίησης, χωρίς τη χρήση των προηγμένων βελτιστοποιήσεων και αλγορίθμων που ενσωματώνονται στη SciPy.

Η σύγκριση της προσαρμοσμένης υλοποίησης Simplex με τις βιβλιοθήκες NumPy και CuPy με την ολοκληρωμένη υλοποίηση του SciPy είναι κρίσιμη για τους εξής λόγους:

- **Αξιολόγηση Απόδοσης**: Η HiGHS του SciPy έχει σχεδιαστεί για βελτιστοποιημένη απόδοση και παρέχει μια αξιόπιστη βάση σύγκρισης για τις προσαρμοσμένες υλοποιήσεις Simplex, επισημαίνοντας αν οι custom υλοποιήσεις μπορούν να προσφέρουν συγκρίσιμα ή καλύτερα αποτελέσματα, ειδικά σε μεγάλα ή πολύπλοκα προβλήματα.
- **Ευελιξία και Προσαρμογή**: Οι προσαρμοσμένες υλοποιήσεις επιτρέπουν ειδικές τροποποιήσεις που μπορεί να είναι απαραίτητες για συγκεκριμένα προβλήματα ή περιβάλλοντα. Εξετάζοντας αυτή τη σύγκριση, μπορούμε να κατανοήσουμε αν η ευελιξία των προσαρμοσμένων υλοποιήσεων υπερτερούν την ήδη βελτιστοποιημένη επίλυση της HiGHS μέσω της SciPy.
- **Χρήση Πόρων**: Οι υλοποιήσεις με CuPy χρησιμοποιούν την GPU για επιτάχυνση, ενώ η HiGHS είναι βελτιστοποιημένη για CPU. Η σύγκριση αναδεικνύει αν η GPU μπορεί να υπερκεράσει τα πλεονεκτήματα της βελτιστοποιημένης λύσης της HiGHS.

Σχήμα 5.16: Μέσος Συνολικός Χρόνος εκτέλεσης NumPy-CuPy και SciPy

Η σύγκριση των χρόνων εκτέλεσης μεταξύ των υλοποιήσεων NumPy, CuPy, και SciPy για διάφορα μεγέθη δεδομένων αποκαλύπτει ενδιαφέροντα ευρήματα:

- Για 200 δεδομένα, η SciPy παρουσίασε τον υψηλότερο χρόνο εκτέλεσης (0,05 δευτερόλεπτα), ενώ η CuPy ήταν η ταχύτερη (0,01 δευτερόλεπτα), με την NumPy να ακολουθεί (0,016 δευτερόλεπτα).
- Στα 400 δεδομένα, η SciPy συνεχίζει να έχει τον μεγαλύτερο χρόνο εκτέλεσης (0,25 δευτερόλεπτα), με την CuPy και την NumPy να έχουν κοντινούς χρόνους (0,019 και 0,01 δευτερόλεπτα αντίστοιχα).
- Στα 600 δεδομένα, η διαφορά μεγαλώνει, με τη SciPy να απαιτεί 2,15 δευτερόλεπτα, την CuPy 0,03 δευτερόλεπτα, και την NumPy 0,06 δευτερόλεπτα.
- Για 800 δεδομένα, η SciPy συνεχίζει να επιβραδύνεται (1,8 δευτερόλεπτα), ενώ η CuPy παραμένει η ταχύτερη (0,01 δευτερόλεπτα) και η NumPy έχει 0,03 δευτερόλεπτα.
- Στα 1000 δεδομένα, οι χρόνοι εκτέλεσης της SciPy (2,35 δευτερόλεπτα) ξεπερνούν κατά πολύ αυτούς της CuPy (0,43 δευτερόλεπτα) και της NumPy (0,27 δευτερόλεπτα).

Αυτά τα αποτελέσματα υποδεικνύουν ότι, αν και η SciPy με τη μέθοδο HiGHS είναι εξαιρετικά βελτιστοποιημένη για την ακρίβεια και την επεξεργαστική ικανότητα, η CuPy αποδεικνύεται πιο αποδοτική για τα δεδομένα που εξετάστηκαν, ειδικά όταν τα δεδομένα είναι σε μικρές έως μεσαίες κλίμακες. Η χρήση της GPU με την CuPy παρέχει σημαντική βελτίωση της ταχύτητας, ενδεχομένως λόγω της καλύτερης αξιοποίησης της παράλληλης επεξεργασίας. Από την άλλη πλευρά, η SciPy φαίνεται να αντιμετωπίζει μεγαλύτερες προκλήσεις στην επεξεργασία μεγάλων δεδομένων, πιθανόν λόγω του συνόλου χαρακτηριστικών της που επικεντρώνονται στην ακρίβεια και στην επεκτασιμότητα του αλγορίθμου, αντί να επικεντρώνεται αποκλειστικά στην ταχύτητα.

Αυτή η σύγκριση αναδεικνύει την αξία της επιλογής του εργαλείου ανάλογα με την κλίμακα και την απαιτούμενη ταχύτητα επεξεργασίας. Η CuPy είναι ιδιαίτερα αποδοτική για μεγαλύτερες κλίμακες δεδομένων λόγω της GPU, ενώ η SciPy, με την ολοκληρωμένη υλοποίηση της μεθόδου HiGHS, μπορεί να είναι προτιμότερη για εφαρμογές όπου η ακρίβεια και η επεκτασιμότητα είναι κρίσιμες, ακόμα και αν αυτό σημαίνει μεγαλύτερος χρόνος εκτέλεσης.

Κεφάλαιο 6ο: Συμπεράσματα & Μελλοντική Εργασία

6.1 Συμπεράσματα

Η μελέτη αυτή επικεντρώθηκε στη σύγκριση της απόδοσης του αλγορίθμου Simplex όταν υλοποιείται με χρήση των βιβλιοθηκών NumPy, SciPy, και CuPy σε περιβάλλοντα CPU και GPU. Μέσω αυτής της ανάλυσης, καταγράφηκαν τα παρακάτω σημαντικά συμπεράσματα:

Απόδοση και Αποδοτικότητα: Η εκτέλεση του Simplex σε CPU με τη χρήση των NumPy και SciPy προσφέρει σταθερές και ακριβείς επιδόσεις, ιδιαίτερα για προβλήματα μικρής και μεσαίας κλίμακας. Ωστόσο, όταν τα δεδομένα αυξάνονται σε κλίμακα, η παράλληλη επεξεργασία μέσω GPU με τη χρήση της βιβλιοθήκης CuPy απέδειξε ότι μπορεί να προσφέρει σημαντικές βελτιώσεις στην ταχύτητα εκτέλεσης.

Συμβατότητα και Ευχρηστία: Η CuPy προσφέρει ένα σημαντικό πλεονέκτημα λόγω της υψηλής συμβατότητάς της με τη NumPy, επιτρέποντας στους προγραμματιστές να μεταφέρουν τον κώδικα από CPU σε GPU με ελάχιστες τροποποιήσεις. Αυτό την καθιστά ιδιαίτερα ελκυστική για την επιτάχυνση υφιστάμενων έργων χωρίς την ανάγκη εκτεταμένων αλλαγών στον κώδικα.

Προβλήματα Παράλληλης Επεξεργασίας: Παρά τα πλεονεκτήματα της χρήσης GPU, η φύση των προβλημάτων γραμμικού προγραμματισμού δεν επιτρέπει πάντα την επίτευξη του μέγιστου δυνατού οφέλους από την παράλληλη επεξεργασία. Οι διαδικασίες μεταφοράς δεδομένων μεταξύ CPU και GPU, καθώς και η διαχείριση της μνήμης της GPU, μπορούν να αποτελέσουν σημαντικά σημεία συμφόρησης που μειώνουν τα οφέλη της επιτάχυνσης. Επιπλέον, η γενική απόδοση της GPU μπορεί να επηρεαστεί από παράγοντες όπως η γενική αρχιτεκτονική της κάρτας, η διαθέσιμη μνήμη, και η αποδοτικότητα του κώδικα σε σχέση με τους πόρους της GPU.

Διαχείριση Πόρων από τη CuPy: Η CuPy, ως ένα υψηλού επιπέδου πακέτο, διαχειρίζεται αυτόματα διάφορες πτυχές της GPU, όπως τα threads, τα blocks και η μνήμη. Αυτό καθιστά τη χρήση της πιο απλή και φιλική προς τον χρήστη, ειδικά για προγραμματιστές που δεν έχουν εξειδικευμένες γνώσεις στον προγραμματισμό GPU. Ωστόσο, αυτή η αυτοματοποιημένη διαχείριση σημαίνει ότι οι χρήστες δεν έχουν τον πλήρη έλεγχο για τη βελτιστοποίηση των πόρων της GPU. Σε περιπτώσεις όπου απαιτείται μέγιστη απόδοση, η χειροκίνητη διαχείριση των πόρων μέσω CUDA kernels μπορεί να προσφέρει καλύτερα αποτελέσματα, αν και αυτός ο τρόπος απαιτεί βαθύτερες γνώσεις σχετικά με την αρχιτεκτονική των καρτών γραφικών και είναι πιο περίπλοκος στη συγγραφή σε σύγκριση με τον κώδικα που παρέχει η CuPy.

Ακρίβεια και Αξιοπιστία: Η χρήση της SciPy για την υλοποίηση του Simplex εγγυάται υψηλή ακρίβεια και αξιοπιστία, κάτι που την καθιστά ιδανική για προβλήματα όπου η ακρίβεια των αποτελεσμάτων είναι κρίσιμη.

6.2 Μελλοντική Εργασία

Η παρούσα εργασία αφήνει αρκετές ανοιχτές κατευθύνσεις για περαιτέρω διερεύνηση και βελτίωση. Μερικές από τις κύριες περιοχές που θα μπορούσαν να αποτελέσουν αντικείμενο μελλοντικής εργασίας περιλαμβάνουν:

Βελτιστοποίηση Μεταφοράς Δεδομένων: Εξετάζοντας τρόπους βελτιστοποίησης της μεταφοράς δεδομένων μεταξύ CPU και GPU, θα μπορούσε να μειωθεί το κόστος αυτών των λειτουργιών, επιτρέποντας μεγαλύτερη επιτάχυνση των αλγορίθμων που βασίζονται σε παράλληλη επεξεργασία.

Περαιτέρω Βελτιώσεις Αλγορίθμων: Η ανάπτυξη και υλοποίηση βελτιωμένων παραλλαγών του Simplex, ειδικά προσαρμοσμένων για παράλληλη εκτέλεση σε GPU, θα μπορούσε να οδηγήσει σε ακόμη μεγαλύτερες επιταχύνσεις και αποδοτικότητα για μεγάλης κλίμακας προβλήματα.

Διερεύνηση Άλλων Αλγορίθμων: Παρόλο που ο Simplex είναι ευρέως χρησιμοποιούμενος, η διερεύνηση άλλων αλγορίθμων γραμμικού προγραμματισμού, όπως ο αλγόριθμος εσωτερικών σημείων (Interior Point Method), θα μπορούσε να αποκαλύψει πρόσθετα οφέλη από τη χρήση της CuPy και της GPU.

Εφαρμογή σε Πραγματικά Σενάρια: Η εφαρμογή της υλοποίησης του Simplex σε πραγματικά σενάρια, όπως στην ανάλυση δεδομένων μεγάλου όγκου ή σε σύνθετα προβλήματα βελτιστοποίησης, θα μπορούσε να αξιολογήσει την πρακτική αξία και τα οφέλη της χρήσης της CuPy σε πραγματικές συνθήκες.

Ανάπτυξη Υβριδικών Μεθόδων: Η ανάπτυξη υβριδικών μεθόδων που θα συνδυάζουν τα πλεονεκτήματα της CPU και της GPU θα μπορούσε να αποτελέσει μια πολλά υποσχόμενη κατεύθυνση, αξιοποιώντας τις δυνατότητες των δύο αυτών υπολογιστικών μονάδων για βέλτιστα αποτελέσματα.

Εξατομίκευση Διαχείρισης GPU: Εξέταση της βελτιστοποίησης της διαχείρισης των πόρων της GPU μέσω της CuPy ή άλλων βιβλιοθηκών όπως η CUDA Python [18]. Σύμφωνα με το CUDA Python Manual, υπάρχουν αρκετές στρατηγικές που μπορούν να βελτιώσουν την απόδοση χωρίς την ανάγκη πλήρους χειροκίνητης διαχείρισης «CUDA kernels»:

- **Αυτόματη Διαχείριση Μνήμης:** Η CUDA Python παρέχει δυνατότητες αυτόματης διαχείρισης της μνήμης GPU, επιτρέποντας στους χρήστες να αποφύγουν την περίπλοκη χειροκίνητη διαχείριση.
- **Βελτιστοποίηση Εκτέλεσης:** Χρησιμοποιώντας τις ενσωματωμένες στρατηγικές για βελτιστοποίηση μεταφορών δεδομένων και επιλογής τύπων δεδομένων, μπορεί να επιτευχθεί μεγαλύτερη αποδοτικότητα.
- **CUDA Streams:** Η αξιοποίηση των CUDA Streams για την ταυτόχρονη εκτέλεση πολλαπλών εργασιών μπορεί να μειώσει τους χρόνους αναμονής και να βελτιώσει την αποδοτικότητα της GPU.

Αυτές οι στρατηγικές μπορούν να επιτρέψουν την αποτελεσματική χρήση των πόρων της GPU με λιγότερες ανάγκες για περίπλοκο προγραμματισμό, βελτιώνοντας την συνολική απόδοση των εφαρμογών που εκμεταλλεύονται τις δυνατότητες GPU.

Η παρούσα διπλωματική εργασία αποτελεί μια προσωπική ερευνητική προσέγγιση στην αξιοποίηση των σύγχρονων υπολογιστικών πόρων για την επίλυση προβλημάτων βελτιστοποίησης, μέσω της υλοποίησης του αλγορίθμου Simplex. Εντάσσεται στο ευρύτερο πλαίσιο των λύσεων που επιδιώκουν την ανάπτυξη αποδοτικών και ισχυρών μεθόδων, αξιοποιώντας τις δυνατότητες των σύγχρονων υπολογιστικών συστημάτων.

ΒΙΒΛΙΟΓΡΑΦΙΑ

[1] V. Kostoglou, *Operations research and organization of production systems.* Thessaloniki, 2016.

[2] George B. Dantzig, "Linear Programming", *Operations Research* 50(1):42-47, 2002.

[3] Theodosis Dimitrakos, "Linear Programming", *Linear Programming*. University of Aegean. Samos 2010.

[4] Introduction to Operation Research, National Technical University of Athens. Athens, 2006.

[5] Alanoud Alotaibi, Farrukh Nadeem, " *A Review of Applications of Linear Programming to Optimize Agricultural Solutions*", International Journal of Information Engineering and Electronic Business(IJIEEB), Vol.13, No.2, pp. 11-21, 2021. DOI: 10.5815/ijieeb.2021.02.02

[6] Daniele Giuseppe Spampinato, "*Linear Optimization with CUDA*". Department of Computer and Information Science Norwegian University of Science and Technology. Norway, 2009.

[7] I.K. Kounetas, N. Chatzistamoulou, "*Introduction to Operation Reasearch and Linear Programming. Solving problems using R".*

[8] Romanos Alexios Spanos, "*Linear Programming models – Case study in precision agriculture*". Aristotle University of Thessaloniki, 2020

[9] T. Paschalis, "*GPU-based acceleration of Smith-Waterman algorithm for audio effects detection*". University of Piraeus, 2017

[10] N. Kozyris, "*Implementation and evaluation of metaprograms for graphics cards*". National Technical University of Athens, 2012

[11] N. Ploskas "*A parallel implementation of an exterior point algorithm for linear programming problems".* University of Macedonia, Thessaloniki, 2009.

[12] G. Greeff , *The revised simplex algorithm on a GPU,* University of Stellen bosch, Tech. Rep., Feb. 2005.

[13] J. H. Junk and D. P. OLeary, Implementing an interior point method for linear programs on a CPU-GPU system, Electronic Transaction on Numerical Analysis, vol. 28, pp. 174189, 2008.

[14] "SciPy,". [Online]. Available: https://docs.scipy.org/doc/scipy/.

[15] "CuPy,". [Online]. Available: https://docs.cupy.dev/en/stable/.

[16] "NumPy,". [Online]. Available: https://numpy.org/doc/stable/.

[17] "CUDA," . [Online] . Available: https://docs.nvidia.com/cuda/.

[18] "CUDA Python Manual**,"**. [Online]. Available: https://nvidia.github.io/cuda-python/.

ΠΑΡΑΡΤΗΜΑ Α : ΚΩΔΙΚΑΣ ΣΥΓΚΡΙΣΗΣ NUMPY - CUPY

```
import numpy as np
import cupy as cp
import time

# Παράμετροι
n_variables = 3000
n_constraints = 2999
num_iterations = 3  # Αριθμός επαναλήψεων

# Ορισμός seed για τυχαίους αριθμούς
np.random.seed(0)
cp.random.seed(0)

def compute_ratios_numpy(bHat, Ahat):
    ratios = np.zeros_like(bHat)
    for i in range(bHat.size):
        Bval = bHat[i]
        Aval = Ahat[i]
        if Aval > 0:
            ratios[i] = Bval / Aval
        else:
            ratios[i] = np.inf
    return ratios

def SimplexLargeScale_numpy(A, b, c):
    basicSize = A.shape[0]
    nonbasicSize = A.shape[1] - basicSize
    if basicSize == 0 or nonbasicSize == 0:
        raise ValueError("A should have more columns than rows (variables
should be more than constraints).")
    cindx = np.arange(len(c))
    cbT = c[nonbasicSize:]
    cnT = c[:nonbasicSize]
    iteration_count  =  0
    while True:
        iteration_count += 1
        cbIndx = cindx[nonbasicSize:]
        cnIndx = cindx[:nonbasicSize]
        B = A[:, cbIndx]

        # Μέτρηση χρόνου για την αντιστροφή της βάσης
        Binv_start = time.time()
        Binv = np.linalg.inv(B)
        Binv_end = time.time()
        Binv_time = Binv_end - Binv_start

        N = A[:, cnIndx]
        bHat = Binv @ b
        yT = cbT @ Binv
        cnHat = cnT - (yT @ N)
```

```
        # Μέτρηση χρόνου για την εισερχόμενη μεταβλητή
        entering_time_start = time.time()
        cnMinIndx = np.argmin(cnHat)
        entering_time_end = time.time()
        entering_time = entering_time_end - entering_time_start

        if np.all(cnHat >=  0):
            entering_var_index = cindx[cnMinIndx]
            exiting_time_start = time.time()
            exiting_var_index = cbIndx[np.argmin(cnHat)]
            exiting_time_end = time.time()
            exiting_time = exiting_time_end - exiting_time_start

            entering_var_value = c[entering_var_index]
            exiting_var_value = b[exiting_var_index]
            return cbT, cbIndx, cnT, cnIndx, bHat, cnHat,entering_var_value,
exiting_var_index,   True,   iteration_count,   Binv_time,   entering_time,
exiting_time
        indx = int(cindx[cnMinIndx])
        Ahat = Binv @ A[:, indx]
        ratios = compute_ratios_numpy(bHat, Ahat)
        if np.all(ratios == np.inf):
            raise ValueError("No valid pivot element found. The problem may
be degenerate or unbounded.")
        ratioMinIndx = np.argmin(ratios)
        cnT[cnMinIndx],     cbT[ratioMinIndx]     =     cbT[ratioMinIndx],
cnT[cnMinIndx]
        cindx[cnMinIndx],    cindx[ratioMinIndx    +    nonbasicSize]    =
cindx[ratioMinIndx + nonbasicSize], cindx[cnMinIndx]

def compute_ratios_cupy(bHat, Ahat):
    ratios =  cp.zeros_like(bHat)
    for i in range(bHat.size):
        Bval = bHat[i]
        Aval = Ahat[i]
        if Aval > 0:
            ratios[i] = Bval / Aval
        else:
            ratios[i] = cp.inf
    return ratios

def SimplexLargeScale_cupy(A, b, c):
    basicSize = A.shape[0]
    nonbasicSize = A.shape[1] - basicSize
    if basicSize == 0 or nonbasicSize == 0:
        raise ValueError("A should have more columns than rows (variables
should be more than constraints).")
    cindx = cp.arange(len(c))
    cbT = c[nonbasicSize:]
    cnT = c[:nonbasicSize]
    iteration_count = 0

    # Χρήση μνήμης GPU πριν την εκκίνηση
    memory_pool = cp.cuda.MemoryPool(cp.cuda.malloc_managed)
```

```
    memory_before = memory_pool.used_bytes()

    # Μέτρηση χρόνου για μεταφορά δεδομένων στη GPU
    data_transfer_start = time.time()

    # Μετατροπή δεδομένων σε CuPy
    A_gpu = cp.asarray(A)
    b_gpu = cp.asarray(b)
    c_gpu = cp.asarray(c)

    data_transfer_end = time.time()
    data_transfer_time = data_transfer_end - data_transfer_start

    # Μέτρηση χρόνου έναρξης
    start_event = cp.cuda.Event()
    end_event = cp.cuda.Event()
    start_event.record()

    while True:
        iteration_count += 1
        cbIndx = cindx[nonbasicSize:]
        cnIndx = cindx[:nonbasicSize]
        B = A_gpu[:, cbIndx]

        # Μέτρηση χρόνου για την αντιστροφή της βάσης
        Binv_start_event = cp.cuda.Event()
        Binv_end_event = cp.cuda.Event()
        Binv_start_event.record()
        Binv = cp.linalg.inv(B)
        Binv_end_event.record()
        Binv_end_event.synchronize()
        Binv_time       =       cp.cuda.get_elapsed_time(Binv_start_event,
Binv_end_event) / 1000  # Μετατροπή σε δευτερόλεπτα

        N = A_gpu[:, cnIndx]
        bHat = cp.dot(Binv, b_gpu)
        yT = cp.dot(cbT, Binv)
        cnHat = cnT - cp.dot(yT, N)

        # Μέτρηση χρόνου για την εισερχόμενη μεταβλητή
        entering_time_start_event = cp.cuda.Event()
        entering_time_end_event = cp.cuda.Event()
        entering_time_start_event.record()
        cnMinIndx = cp.argmin(cnHat)
        entering_time_end_event.record()
        entering_time_end_event.synchronize()
        entering_time = cp.cuda.get_elapsed_time(entering_time_start_event,
entering_time_end_event) / 1000  # Μετατροπή σε δευτερόλεπτα

        if cp.all(cnHat >= 0):
            end_event.record()
            end_event.synchronize()
            elapsed_time = cp.cuda.get_elapsed_time(start_event, end_event)
/ 1000  # Μετατροπή σε δευτερόλεπτα
```

```
            memory_after = memory_pool.used_bytes()
            memory_used = (memory_after - memory_before) / (1024 * 1024)  #
Μετατροπή σε MB
            entering_var_index = cindx[cnMinIndx].get()

            exiting_time_start_event = cp.cuda.Event()
            exiting_time_end_event = cp.cuda.Event()
            exiting_time_start_event.record()
            exiting_var_index = cbIndx[cp.argmin(cnHat)].get()
            exiting_time_end_event.record()
            exiting_time_end_event.synchronize()
            exiting_time                                                    =
cp.cuda.get_elapsed_time(exiting_time_start_event,  exiting_time_end_event)
/ 1000  # Μετατροπή σε δευτερόλεπτα

            entering_var_value = c_gpu[entering_var_index].get()
            exiting_var_value = b_gpu[exiting_var_index].get()
            return    cbT,    cbIndx,    cnT,    cnIndx,    bHat,    cnHat,
entering_var_value,  exiting_var_index,  True,  iteration_count,  Binv_time,
memory_used, elapsed_time, data_transfer_time, entering_time, exiting_time
        indx = int(cindx[cnMinIndx].get())
        Ahat = cp.dot(Binv, A_gpu[:, indx])
        ratios = compute_ratios_cupy(bHat, Ahat)
        if cp.all(ratios == cp.inf):
            end_event.record()
            end_event.synchronize()
            elapsed_time = cp.cuda.get_elapsed_time(start_event, end_event)
/ 1000  # Μετατροπή σε δευτερόλεπτα
            memory_after = memory_pool.used_bytes()
            memory_used = (memory_after - memory_before) / (1024 * 1024)  #
Μετατροπή σε MB
            raise ValueError("No valid pivot element found. The problem may
be degenerate or unbounded.")
        ratioMinIndx = cp.argmin(ratios)
        cnT[cnMinIndx],     cbT[ratioMinIndx]     =     cbT[ratioMinIndx],
cnT[cnMinIndx]
        cindx[cnMinIndx],    cindx[ratioMinIndx    +    nonbasicSize]    =
cindx[ratioMinIndx + nonbasicSize], cindx[cnMinIndx]

# Συγκέντρωση αποτελεσμάτων
total_entering_var_value_np = 0
total_exiting_var_value_np = 0
total_Binv_time_np = 0
total_execution_time_np = 0
total_entering_time_np = 0
total_exiting_time_np = 0

total_entering_var_value_cp = 0
total_exiting_var_value_cp = 0
total_Binv_time_cp = 0
total_execution_time_cp = 0
total_entering_time_cp = 0
total_exiting_time_cp = 0
```

```
for iteration in range(num_iterations):
    # Δημιουργία τυχαίων δεδομένων με την ίδια seed
    A_large                 =                 np.random.randn(n_constraints,
n_variables).astype(np.float32)
    b_large = np.random.randn(n_constraints).astype(np.float32)
    c_large = np.random.randn(n_variables).astype(np.float32)

    # Μετατροπή δεδομένων σε CuPy
    A_large_cp = cp.asarray(A_large, dtype=cp.float32)
    b_large_cp = cp.asarray(b_large, dtype=cp.float32)
    c_large_cp = cp.asarray(c_large, dtype=cp.float32)

    # Εκτέλεση και μέτρηση χρόνου για NumPy
    start_time_np = time.time()
    result_np = SimplexLargeScale_numpy(A_large, b_large, c_large)
    end_time_np = time.time()
    execution_time_np = end_time_np - start_time_np

    cbT_large_np,    cbIndx_large_np,    cnT_large_np,    cnIndx_large_np,
bHat_large_np, cnHat_large_np, entering_var_value_np, exiting_var_index_np,
is_optimal_large_np,  iteration_count_np,  Binv_time_np,  entering_time_np,
exiting_time_np = result_np

    # Συγκέντρωση αποτελεσμάτων για NumPy
    total_entering_var_value_np += entering_var_value_np
    total_exiting_var_value_np += b_large[exiting_var_index_np]
    total_Binv_time_np += Binv_time_np
    total_execution_time_np += execution_time_np
    total_entering_time_np += entering_time_np
    total_exiting_time_np += exiting_time_np

    # Εκτέλεση και μέτρηση χρόνου για CuPy
    start_time_cp = time.time()
    result_cp = SimplexLargeScale_cupy(A_large_cp, b_large_cp, c_large_cp)
    end_time_cp = time.time()
    execution_time_cp = end_time_cp - start_time_cp

    cbT_large_cp,    cbIndx_large_cp,    cnT_large_cp,    cnIndx_large_cp,
bHat_large_cp, cnHat_large_cp, entering_var_value_cp, exiting_var_index_cp,
is_optimal_large_cp,  iteration_count_cp,  Binv_time_cp,  memory_used_cp,
total_time_cp,  data_transfer_time,  entering_time_cp,  exiting_time_cp  =
result_cp

    # Συγκέντρωση αποτελεσμάτων για CuPy
    total_entering_var_value_cp += entering_var_value_cp
    total_exiting_var_value_cp += b_large_cp[exiting_var_index_cp].get()
    total_Binv_time_cp += Binv_time_cp
    total_execution_time_cp += execution_time_cp
    total_entering_time_cp += entering_time_cp
    total_exiting_time_cp += exiting_time_cp

# Υπολογισμός μέσων τιμών
mean_entering_var_value_np = total_entering_var_value_np / num_iterations
mean_exiting_var_value_np = total_exiting_var_value_np / num_iterations
```

```
mean_Binv_time_np = total_Binv_time_np / num_iterations
mean_execution_time_np = total_execution_time_np / num_iterations
mean_entering_time_np = total_entering_time_np / num_iterations
mean_exiting_time_np = total_exiting_time_np / num_iterations

mean_entering_var_value_cp = total_entering_var_value_cp / num_iterations
mean_exiting_var_value_cp = total_exiting_var_value_cp / num_iterations
mean_Binv_time_cp = total_Binv_time_cp / num_iterations
mean_execution_time_cp = total_execution_time_cp / num_iterations
mean_entering_time_cp = total_entering_time_cp / num_iterations
mean_exiting_time_cp = total_exiting_time_cp / num_iterations

# Εκτύπωση συνολικών αποτελεσμάτων
print(f"Average entering variable value for NumPy:
{mean_entering_var_value_np:.4f}")
print(f"Average exiting variable value for NumPy:
{mean_exiting_var_value_np:.4f}")
print(f"Average time for matrix inversion (NumPy): {mean_Binv_time_np:.4f}
seconds")
print(f"Average total execution time (NumPy): {mean_execution_time_np:.4f}
seconds")
print(f"Average entering time (NumPy): {mean_entering_time_np:.4f} seconds")
print(f"Average exiting time (NumPy): {mean_exiting_time_np:.4f} seconds")
print(f"Average entering variable value for CuPy:
{mean_entering_var_value_cp:.4f}")
print(f"Average exiting variable value for CuPy:
{mean_exiting_var_value_cp:.4f}")
print(f"Average time for matrix inversion (CuPy): {mean_Binv_time_cp:.4f}
seconds")
print(f"Average total execution time (CuPy): {mean_execution_time_cp:.4f}
seconds")
print(f"Average entering time (CuPy): {mean_entering_time_cp:.4f} seconds")
print(f"Average exiting time (CuPy): {mean_exiting_time_cp:.4f} seconds")
print(f"Average speedup time of total execution:
{mean_entering_time_np/mean_entering_time_cp:4f}")
print(f"Average speedup time of total execution:
{mean_exiting_time_np/mean_exiting_time_cp:4f}")
print(f"Average speedup time of total execution:
{mean_execution_time_np/mean_execution_time_cp:4f}")
```

ΠΑΡΑΡΤΗΜΑ Β : ΚΩΔΙΚΑΣ ΣΥΓΚΡΙΣΗΣ SCIPY ΜΕ NUMPY-CUPY

```
import numpy as np
import cupy as cp
import time
from scipy.optimize import linprog

# Παράμετροι
n_variables = 3000
n_constraints = 2999
num_iterations = 1  # Αριθμός επαναλήψεων

# Ορισμός seed για τυχαίους αριθμούς
np.random.seed(0)
cp.random.seed(0)

def compute_ratios(bHat, Ahat, use_cupy=False):
    if use_cupy:
        ratios = cp.zeros_like(bHat)
        for i in range(bHat.size):
            Bval = bHat[i]
            Aval = Ahat[i]
            ratios[i] = Bval / Aval if Aval > 0 else cp.inf
        return ratios
    else:
        ratios = np.zeros_like(bHat)
        for i in range(bHat.size):
            Bval = bHat[i]
            Aval = Ahat[i]
            ratios[i] = Bval / Aval if Aval > 0 else np.inf
        return ratios

def SimplexLargeScale(A, b, c, use_cupy=False):
    if use_cupy:
        A = cp.asarray(A, dtype=cp.float32)
        b = cp.asarray(b, dtype=cp.float32)
        c = cp.asarray(c, dtype=cp.float32)

    basicSize = A.shape[0]
    nonbasicSize = A.shape[1] - basicSize
    if basicSize == 0 or nonbasicSize == 0:
        raise ValueError("A should have more columns than rows (variables
should be more than constraints).")

    cindx = cp.arange(len(c)) if use_cupy else np.arange(len(c))
    cbT = c[nonbasicSize:]
    cnT = c[:nonbasicSize]
    iteration_count = 0

    if use_cupy:
        memory_pool = cp.cuda.MemoryPool(cp.cuda.malloc_managed)
```

```
        memory_before = memory_pool.used_bytes()

    data_transfer_start = time.time()
    if use_cupy:
        A_gpu = cp.asarray(A)
        b_gpu = cp.asarray(b)
        c_gpu = cp.asarray(c)
    data_transfer_end = time.time()
    data_transfer_time = data_transfer_end - data_transfer_start

    start_event = cp.cuda.Event() if use_cupy else None
    end_event = cp.cuda.Event() if use_cupy else None
    if start_event:
        start_event.record()

    while True:
        iteration_count += 1
        cbIndx = cindx[nonbasicSize:]
        cnIndx = cindx[:nonbasicSize]
        B = A[:, cbIndx] if not use_cupy else A_gpu[:, cbIndx]

        # Μέτρηση χρόνου για την αντιστροφή της βάσης
        Binv_start = time.time()
        Binv = np.linalg.inv(B) if not use_cupy else cp.linalg.inv(B)
        Binv_end = time.time()
        Binv_time = Binv_end - Binv_start

        N = A[:, cnIndx] if not use_cupy else A_gpu[:, cnIndx]
        bHat = Binv @ b
        yT = cbT @ Binv
        cnHat = cnT - (yT @ N)

        # Μέτρηση χρόνου για την εισερχόμενη μεταβλητή
        entering_time_start = time.time()
        cnMinIndx = np.argmin(cnHat) if not use_cupy else cp.argmin(cnHat)
        entering_time_end = time.time()
        entering_time = entering_time_end - entering_time_start

        if (np.all(cnHat >= 0)) if not use_cupy else (cp.all(cnHat >= 0)):
            entering_var_index = cindx[cnMinIndx].get() if use_cupy else
cindx[cnMinIndx]
            exiting_time_start = time.time()
            exiting_var_index = cbIndx[np.argmin(cnHat)].get() if use_cupy
else cbIndx[np.argmin(cnHat)]
            exiting_time_end = time.time()
            exiting_time = exiting_time_end - exiting_time_start

            entering_var_value = c[entering_var_index].get() if use_cupy
else c[entering_var_index]
            exiting_var_value = b[exiting_var_index].get() if use_cupy else
b[exiting_var_index]

            if use_cupy:
                end_event.record()
```

```
                end_event.synchronize()
                elapsed_time    =    cp.cuda.get_elapsed_time(start_event,
end_event) / 1000  # Μετατροπή σε δευτερόλεπτα
                memory_after = memory_pool.used_bytes()
                memory_used = (memory_after - memory_before) / (1024 * 1024)
# Μετατροπή σε MB
                print("### CuPy Results ###")
                print(f"CuPy  Total  Execution  Time:  {elapsed_time:.6f}
seconds")
                print(f"CuPy Data Transfer Time: {data_transfer_time:.6f}
seconds")
                print(f"CuPy Memory Used: {memory_used:.6f} MB")
                print(f"CuPy Basis Inversion Time: {Binv_time:.6f} seconds")
                print(f"CuPy  Entering  Variable  Time:  {entering_time:.6f}
seconds")
                print(f"CuPy Exiting Variable Time: {exiting_time:.6f}
seconds")
            else:
                print("### NumPy Results ###")
                print(f"NumPy   Basis   Inversion   Time:   {Binv_time:.6f}
seconds")
                print(f"NumPy Entering Variable Time: {entering_time:.6f}
seconds")
                print(f"NumPy  Exiting  Variable  Time:  {exiting_time:.6f}
seconds")
            return    cbT,    cbIndx,    cnT,    cnIndx,    bHat,    cnHat,

entering_var_value, exiting_var_index, True, iteration_count, Binv_time,
entering_time, exiting_time

        indx    =    int(cindx[cnMinIndx].get())    if    use_cupy    else
int(cindx[cnMinIndx])
        Ahat = Binv @ A[:, indx] if not use_cupy else cp.dot(Binv, A_gpu[:,
indx])
        ratios = compute_ratios(bHat, Ahat, use_cupy)
        if (np.all(ratios == np.inf)) if not use_cupy else (cp.all(ratios ==
cp.inf)):
            raise ValueError("No valid pivot element found. The problem may
be degenerate or unbounded.")
        ratioMinIndx   =   np.argmin(ratios)   if   not   use_cupy   else
cp.argmin(ratios)
        cnT[cnMinIndx],     cbT[ratioMinIndx]     =     cbT[ratioMinIndx],
cnT[cnMinIndx]
        cindx[cnMinIndx],    cindx[ratioMinIndx    +    nonbasicSize]    =
cindx[ratioMinIndx + nonbasicSize], cindx[cnMinIndx]

def SimplexLargeScale_scipy(A, b, c):
    res = linprog(c, A_eq=A, b_eq=b, method='highs')

    Binv_start = time.time()
    Binv_end = time.time()
    Binv_time = Binv_end - Binv_start

    entering_time_start = time.time()
    entering_time_end = time.time()
```

```
    entering_time = entering_time_end - entering_time_start

    exiting_time_start = time.time()
    exiting_time_end = time.time()
    exiting_time = exiting_time_end - exiting_time_start

    print("### SciPy (highs) Results ###")
    print(f"SciPy Basis Inversion Time: {Binv_time:.6f} seconds")
    print(f"SciPy Entering Variable Time: {entering_time:.6f} seconds")
    print(f"SciPy Exiting Variable Time: {exiting_time:.6f} seconds")

    return res

# Δημιουργία μεγάλου συστήματος εξισώσεων
A = np.random.rand(n_constraints, n_variables)
b = np.random.rand(n_constraints)
c = np.random.rand(n_variables)

# Μέτρηση χρόνου για το NumPy
numpy_times = []
for i in range(num_iterations):
    start = time.time()
    SimplexLargeScale(A, b, c, use_cupy=False)
    end = time.time()
    numpy_times.append(end - start)

# Μέτρηση χρόνου για το CuPy
cupy_times = []
for i in range(num_iterations):
    start = time.time()
    SimplexLargeScale(A, b, c, use_cupy=True)
    end = time.time()
    cupy_times.append(end - start)

# Μέτρηση χρόνου για το SciPy
scipy_times = []
for i in range(num_iterations):
    start = time.time()
    SimplexLargeScale_scipy(A, b, c)
    end = time.time()
    scipy_times.append(end - start)

# Εκτύπωση αποτελεσμάτων
print("\nExecution times:")
print(f"NumPy times: {numpy_times}")
print(f"CuPy times: {cupy_times}")
print(f"SciPy (highs) times: {scipy_times}")
```