ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Παραλληλοποίηση υπολογισμών και δημιουργίας πινάκων για το επιστημονικό πρόγραμμα «Sirene»

Ιωάννης Δ.Δίπλας

Επιβλέπων **Ιωάννης Κοτρώνης,** Αναπληρωτής Καθηγητής

ΑΘΗΝΑ

ΟΚΤΩΒΡΙΟΣ 2015

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Παραλληλοποίηση υπολογισμών και δημιουργίας πινάκων για το επιστημονικό πρόγραμμα «Sirene»

Ιωάννης Δ. Δίπλας

Α.Μ.:1115200800207

ΕΠΙΒΛΕΠΟΝΤΕΣ: **Ιωάννης Κοτρώνης,** Αναπληρωτής Καθηγητής

ΠΕΡΙΛΗΨΗ

Τα τελευταία χρόνια έχει υπάρξει μια ραγδαία ανάπτυξη των επιστημών στην οποία έχει παίξει καθοριστικό ρόλο η βοήθεια των υπολογιστών. Με την πάροδο του χρόνου η ανάγκη για ταχύτερους και περισσότερους υπολογισμούς έχει αυξηθεί δραματικά, αναγκάζοντας την επιστήμη των υπολογιστών να αναπτύσσεται ραγδαία τόσο σε διάθεση πόρων όσο και στον τρόπο αξιοποίησης αυτών. Μια προσέγγιση για την αξιοποίηση αυτή των πόρων είναι και ο παράλληλος προγραμματισμός ο οποίος προσπαθεί να εκμεταλλευτεί πλήρως τους διαθέσιμους πόρους ενός μηχανήματος ή ακόμα και ενός συνόλου μηχανημάτων.

Η παρούσα πτυχιακή αναπτύχθηκε στο Τμήμα Πληροφορικής και Τηλεπικοινωνιών και αφορά την επιτάχυνση των διαδικασιών, που είναι υπεύθυνες για την δημιουργία πινάκων και για τους υπολογισμούς πάνω σε αυτούς, του ερευνητικού προγράμματος SIRENE που χρησιμοποιείται απο το Εθνικό Κέντρο Έρευνας Φυσικών Επιστημών “Δημόκριτος”. Για την εκτέλεση των διαδικασιών επιτάχυνσης χρησιμοποιήθηκε η αρχιτεκτονική CUDA, μια αρχιτεκτονική η οποία αναπτύχθηκε απο την NVIDIA και παρέχει, μέσω των κατάλληλων επεκτάσεων, εργαλεία για την υλοποίηση προγραμμάτων σε περιβάλλοντα παράλληλου προγραμματισμού. Η τεχνολογία CUDA, της οποίας το όνομα είναι ακρωνύμιο του “Compute Unified Device Architecture”, αξιοποιεί τις δυνατότητες των καρτών γραφικών, μέσα απο τις πολλαπλούς πυρήνες που έχουν στην διάθεσή τους, οι οποίες είναι εξαιρετικά εργαλειά για παράλληλους υπολογισμούς πράξεων, μιας και δημιουργήθηκαν για τον κλάδο των γραφικών.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ:Παράλληλος Προγραμματισμός

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Παράλληλος προγραμματισμός, CUDA, Sirene, Υπολογισμοί πινάκων, OpenMP

ABSTRACT

In recent years, due to enormous human performance in science, there has been a proportionate technology development. The rapid growth in correspondence with the increasing demand of computing resources at lower cost have turned the scientific community in search of developing new techniques to increase the performance of computing systems. The result of this shift is the creating of parallel programming and parallel systems in general.

This present Thesis has been conceptualized and written in the department of Informatics and Telecommunications of National and Kapodistrian University of Athens. It involves the acceleration of processes that are responsible for the creation of arrays and for calculations based on said arrays, all related to the SIRENE research program which is used from the National Center of Research of Natural Sciences "DEMOKRITOS". In order to run the acceleration processes we use the CUDA framework, a framework developed by NVIDIA. CUDA provides, via use of suitable extensions, tools for the implementation of applications in Parallel Computing environments. CUDA -which stands for "Compute Unified Device Architecture", capitalizes on the capabilities of graphics chips- by of the multiple cores to their disposal, which are excellent tools for Parallel Computing instances, since they were created for the graphics industry.

SUBJECTAREA: Parallel programming

KEYWORDS: Parallel programming, CUDA, Sirene, Calculations on arrays, OpenMP

ΠΕΡΙΕΧΟΜΕΝΑ

ΚΑΤΑΛΟΓΟΣ ΣΧΗΜΑΤΩΝ

ΚΑΤΑΛΟΓΟΣ ΕΙΚΟΝΩΝ

ΠΡΟΛΟΓΟΣ

Η μελέτη που ακολουθεί εντάσσεται στα πλαίσια της πτυχιακής του Προγράμματος Προπτυχιακών Σπουδών του Τμήματος Πληροφορικής και Τηλεπικοινωνιών του Εθνικού και Καποδιστριακού Πανεπιστημίου Αθηνών(ΕΚΠΑ).

Η ιδέα της ενασχόλησης μου με την βελτιστοποίηση της διαδικασίας υπολογισμού και δημιουργίας των πινάκων, που χρησιμοποιεί το επιστημονικό πρόγραμμα SIRENE, προέκυψε κατόπιν συζήτησής μου με τον αναπληρωτή καθηγητή κ.Κοτρώνη τον οποίο και ευχαριστώ πολύ για την εμπιστοσύνη που μου έδειξε, καθώς πρόκειται για μια μελέτη που είναι μέρος μιας ευρύτερης εργασίας και προσπάθειας επιτάχυνσης του προγράμματος SIRENE με σκοπό την βοήθεια της επιστημονικής κοινότητας του Εθνικού Κέντρου Έρευνας Φυσικών Επιστημών «Δημόκριτος» στις έρευνες και τα πειράματα που χρησιμοποιούν το παραπάνω εργαλείο.

1. ΕΙΣΑΓΩΓΗ

1.1 Το ευρύτερο πρόβλημα

Το πρόβλημα που καλούμαστε να μελετήσουμε έχει να κάνει με την επιτάχυνση της εκτέλεσης του ερευνητικού προγράμματος SIRENE μέσα απο την χρήση τεχνικών παράλληλου προγραμματισμού μοιράζοντας τον φόρτο εργασίας τόσο σε κάρτες γραφικών όσο και σε ομάδες, clusters, υπολογιστών.

Πρόκειται για ενα ευέλικτο πρόγραμμα προσομοίωσης ανιχνευτή για τηλεσκόπια νετρίνων με φωτο-πολλαπλασιαστές ως φωτο-αισθητήρες. Το πρόγραμμα SIRENE έχει σχεδιαστεί κυρίως για να χρησιμοποιηθεί συνδυαστικά με τον μελλοντικό ανιχνευτή, μεγέθους κυβικού χιλιομέτρου, KM3NeT. Έχει δοκιμαστεί επίσης με το επιστημονικό πρόγραμμα ANTARES μιας και η φύση του προγράμματος είναι τέτοια που να του επιτρέπει την απο κοινού χρήση με άλλα τηλεσκόπια νετρίνων. Το SIRENE αποτελείται από μια βιβλιοθήκη συναρτήσεων για την γλώσσα C++, όπου επιτρέπει την χρήση πολλών διαφορετικών γεωμετριών για ένα τηλεσκόπιο νετρίνων, καθώς και διάφορες παραμετροποιήσεις και χαρακτηριστικά των φωτο-πολλαπλασιαστών οι οποίοι υπάρχουν μέσα στην μορφολογία του γενικότερου τηλεσκοπίου.

Λόγω του όγκου των δεδομένων και του μεγάλου αριθμού απο υπολογισμούς μια εκτέλεση, του παραπάνω προγράμματος, ικανή να παράξει δεδομένα που θα αντιστοιχούσαν σε μια εξάλεπτη προσομοίωση, θα μπορούσε να διαρκέσει ακόμα και 1 μήνα. Όπως, εύκολα, γίνεται αντιληπτό μια τέτοια διαδικασία επιβραδύνει σημαντικά την ερεύνα στον τομέα της προσομοίωσης των τηλεσκοπίων για νετρίνα. Έτσι λοιπόν, μέσα απο την χρήση του παράλληλου προγραματισμού, το πρόγραμμα πρέπει να αναλυθεί και να τροποποιηθεί έτσι ώστε τα επιμέρους κομμάτια του να μπορούν τα τρέξουν ανεξάρτητα σε ξεχωριστές υπολογιστικές μονάδες, είτε αυτές λέγονται κάρτες γραφικών είτε ομάδες υπολογιστών(clusters). Με ποιον τρόπο, λοιπόν, και σε ποια σημεία της προσομοίωσης μπορούμε να επέμβουμε έτσι ώστε να έχουμε αυτονομία, κάτι που οδηγεί στην γενικότερη επιτάχυνση, χωρίς φυσικά να έχουμε αλλοίωση των αποτελεσμάτων;

1.2 Υλοποίηση SIRENE

Η υλοποίηση του SIRENE χρησιμοποιεί πολλαπλές δομές επανάληψης και αυτό διότι εξετάζει πολλές διαφορετικές περιπτώσεις σωματιδίων κατα πόσο πληρούν ορισμένα κριτήρια που χρειάζονται έτσι ώστε να συμπεριληφθούν στα στατιστικά που παράγονται στο τέλος. Έτσι λοιπόν το πρόγραμμα έχει τις παρακάτω δομές επανάληψης:

1.Events, for loop

Κάθε event αποτελείται απο έναν αριθμό απο σωματίδια τα οποία μέσα στο πρόγραμμα SIRENE αναφέρονται ώς “tracks”.

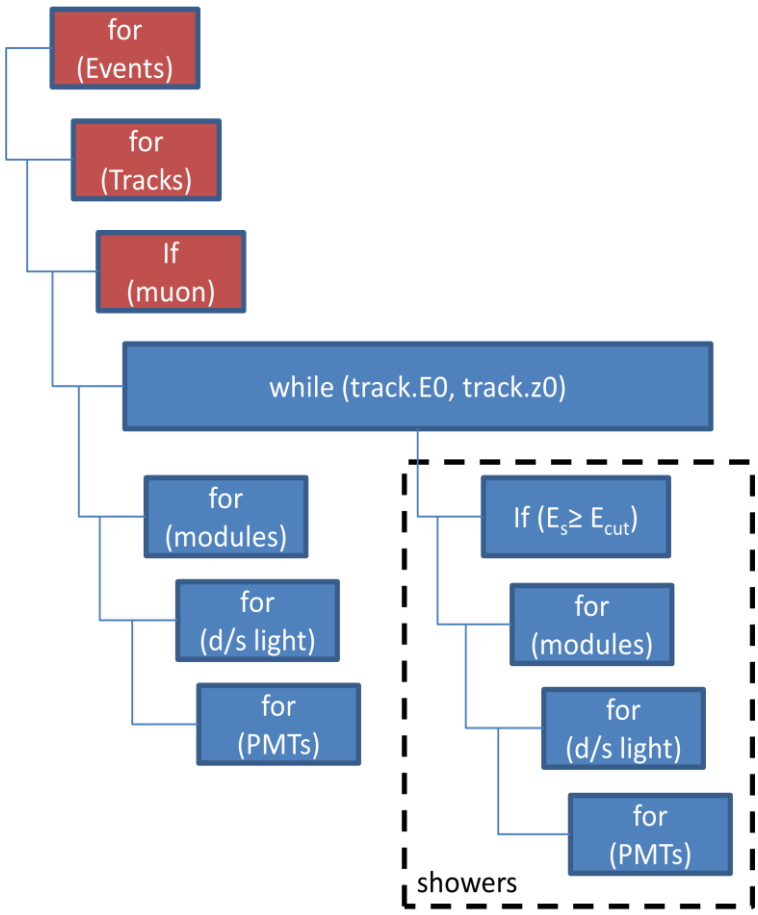
2. Tracks ,for loop

Σε κάθε track γίνεται η απεικόνιση της θέσης και της τροχιάς (track) ενός μιονίου καθώς και το κατά πόσο αυτό θα δημιουργήσει φωτόνια που θα συναντήσουν κάποιον

φωτοπολλαπλασιαστή PMT στην πορεία τους. Σε αυτό το σημείο το πρόγραμμα δημιουργεί ενα σημαντικό αριθμό πινάκων που βοηθάει στην απεικόνιση της τροχιάς του μιονίου. Επίσης στο σημείο αυτό γίνεται ο υπολογισμός της περιστροφής της τροχιάς του σωματιδίου (track) στο άξονα z.

Σημείωση: Οι παραπάνω επαναλήψεις είναι εμφωλευμένες η μία μέσα στην άλλη

Σχηματικά:



Σχήμα 1 : Οπτική αναπαράσταση των επαναλήψεων του επιστημονικού προγράμματος SIRENE (με κόκκινα τα σημεία που εστιάζουμε)

1.3 Που εστιάζουμε εμείς

Όπως αναφέραμε παραπάνω η επανάληψη των tracks περιέχει τη δημιουργία ορισμένων πινάκων για την απεικόνιση της τροχιάς ενός σωματιδίου μέσα στον κύλινδρο με τους φωτοανιχνευτές καθώς και τον υπολογισμό της περιστροφής της τροχιάς ενός σωματιδίου στον άξονα z. Οι πίνακες αυτοί ονομάζονται πίνακες στροφής είναι μεγέθους 3x3 και απαρτίζονται απο μεταβλητές κινούμενης υποδιαστολής, double point, επομένως το μέγεθος ενός πίνακα είναι 72 bytes(9 στοιχεία x 8 bytes μέγεθος). Αν στο παραπάνω μέγεθος προσθέσουμε και εκείνο των 3 μεταβλητών που

αντιπροσωπεύουν την μετατόπιση στους άξονες x,y και z, τότε το συνολικό μέγεθος της πληροφορίας για ένα σωματίδιο ανέρχεται στα 96 bytes.

Ο έλεγχος για την πορεία των σωματιδίων έχει να κάνει με τις διαφορετικές γεωμετρίες του ανιχνευτή νετρίνων. Η γεωμετρία εξαρτάται από τη θέση και το είδος οπτικών μονάδων (optical modules OMs) που περιέχουν τους φωτοανιχνευτές(PMTs). Ο αριθμός των PMTs συνήθως κυμαίνεται στις 117.000 γεωμετρίες. Κάνοντας τους υπολογισμούς βλέπουμε πως το σύνολο του χώρου που χρειαζόμαστε για να αποθηκεύσουμε τους πίνακες για κάθε σωματίδιο είναι περίπου 8GB μνήμης.
Η ρουτίνα των παραπάνω υπολογισμών, λοιπόν είναι η εξής:

1. Ανάγνωση δυο γωνιών φ,θ για κάθε track

2. Υπολογισμός των ημιτόνων και των συνημιτόνων για κάθε μια απο αυτές τις γωνίες

3. Πολλαπλασιασμοί μεταξύ των αποτελεσμάτων του βήματος 2 για κάθε ένα απο τα 9 στοιχεία του πίνακα

4. Χρήση των στοιχείων του πίνακα για υπολογισμό της περιστροφής

Η ρουτίνα αυτή καταναλώνει σημαντικό μέρος του συνολικού χρόνου εκτέλεσης του προγράμματος SIRENE και είναι το βασικό αντικείμενο της παρούσας πτυχιακής. Μέσα απο την χρήση του framework “CUDA” της Nvidia προσπαθούμε να παραλληλοποιήσουμε την ανωτέρω διαδικασία χρησιμοποιώντας τις κάρτες γραφικών της Nvidia, οι οποίες άλλωστε έχουν δημιουργηθεί για την απεικόνιση των γραφικών σε μια οθόνη, κατ’ επέκταση το βασικό τους προτέρημα είναι η διαχείριση πινάκων καθώς και οι υπολογισμοί πάνω σε γωνίες(ημίτονα συνημίτονα κ.α).

1.4 Υλοποίηση σε GPU

Η υλοποίηση της διαδικασίας, όπως αυτή περιγράφηκε παραπάνω, χωρίζεται σε τέσσερα βασικά στάδια:

α) Την ανάγνωση των γωνιών
β) Την μεταφορά των δεδομένων στην GPU
γ) Τους υπολογισμούς των γωνιών και των πινάκων
δ) Την μεταφορά των δεδομένων πίσω στην CPU

1.4.1 Ανάγνωση των γωνιών

Στο συγκεκριμένο στάδιο γίνεται η ανάγνωση των γωνιών φ, θ απο το αρχείο, ενδεικτικά να αναφέρουμε οτι μιλάμε για ένα μέγεθος της τάξεως των 117.000 ζευγών. Η ανάγνωση αυτή θα μπορούσε να επιταχυνθεί μέσα απο την χρήση άλλων frameworks,

όπως εκείνο του MPI καθώς και του OpenMP, καθώς οι κάρτες γραφικών δεν έχουν άμεση πρόσβαση στην είσοδο που μπορεί να δώσει ένας χρήστη(Standar Input), οπότε αναγκαστηκά τα δεδομένα πρέπει να διαβαστούν πρώτα με την βοήθεια της CPU και στην συνέχεια να μεταφερθούν στην κάρτα γραφικών(GPU) για περαιτέρω επεξεργασία. Αυτό το στάδιο δεν επιδέχεται βελτίωσης, σε επίπεδο κάρτας γραφικών.

1.4.2 Μεταφορά δεδομένων απο και προς στην GPU

Πρόκειται για ενα στάδιο το οποίο μπορεί να αποσπάσει έως και το 80% του χρόνου εκτέλεσης ενός προγράμματος CUDA, μιας και οι διαδικασίες μεταφοράς των δεδομένων, τόσο εντός όσο και εκτός μιας κάρτας γραφικών, είναι οι πιο χρονοβόρες και αποτελούν ένα σημαντικό πεδίο επιστημονικής μελέτης για την βελτίωση των διαδικασιών παραλληλοποίησης. Στο συγκεκριμένο στάδιο δεν μπορούμε να επέμβουμε με άμεσο τρόπο, δεν μπορούμε δηλαδή να βελτιώσουμε τον χρόνο τον οποίο χρειάζεται ένα πρόγραμμα για να μεταφέρει τα δεδομένα του στην GPU, αυτό όμως που μπορούμε, και κάνουμε, είναι να διαχειριζόμαστε την μεταφορά αυτή κατα έναν τρόπο τέτοιο ώστε να μειώσουμε τον χρόνο που μένει αδρανές ένα σύστημα περιμένοντας να πάρει, ή να δώσει, δεδομένα.

1.4.3 Υπολογισμοί γωνιών και πινάκων

Το στάδιο αυτό είναι εκείνο στο οποίο δίνουμε την μεγαλύτερη βάση, μιας και μπορούμε να επέμβουμε άμεσα σε πράγματα όπως η σειρά εκτέλεσης των εντολών, η διαχείριση της πρόσβασης των επιμέρους νημάτων της κάρτας στα δεδομένα κ.α. Πρόκειται για το σημείο στο οποίο επικεντρωνόμαστε περίσσοτερο προσφέροντας διάφορες υλοποιήσεις και παρατηρώντας τη σταδιακή βελτίωση των χρόνων, ανάλογα με τους αλγόριθμους που χρησιμοποιούμε, και εξάγοντας συμπεράσματα σχετικά με το αν αξίζει η παραλληλοποίηση μιας τέτοιας διαδικάσιας, κάτω απο ποιες συνθήκες, τι όγκο δεδομένων, με την χρήση συνδυασμού απο frameworks ή οχι κ.α.

Οι υλοποιήσεις της παραπάνω διαδικασίας έχουν να κάνουν με την σειριακή εκτέλεση και απεικόνιση των αποτελεσμάτων (χρησιμοποιώντας μόνο τον επεξεργαστή ενός υπολογιστή), με την παράλληλη εκτέλεση των εντολών με την χρήση της κάρτας γραφικών που μπορεί να μας παρέχει μέχρι και 1000 μικροεπεξεργαστές και τέλος με τον συνδυασμό των δυο παραπάνω προσεγγίσεων για την δημιουργία μιας τρίτης, εκεινής που συνδυάζει την εκκίνηση πολλαπλών πυρήνων στην κάρτα γραφικών με την βοήθεια του κεντρικού πυρήνα του υπολογιστή.

1.5 Υλοποίηση σε CPU

Στην υλοποίηση αυτή χρησιμοποιούμε μόνο τον επεξεργαστή ενός υπολογιστή για την εκτέλεση της διαδικασίας, πράγμα που σημαίνει πως αναλαμβάνει, τελείως σειριακά, το διάβασμα του αρχείου, την δέσμευση του απαραίτητου χώρου για τους πίνακες, τους υπολογισμούς που χρειάζονται και την απεικόνιση των αποτελεσμάτω

1.6 Υλοποίηση σε GPU

Χρησιμοποιώντας την κάρτα γραφικών επιταχύνουμε τους υπολογισμούς σε επίπεδο κάρτας γραφικών χρησιμοποιώντας τα νήματα που προσφέρουν οι μικροεπεξεργαστές της κάρτας για να αναλάβουν κάθε ένα απο τα στοιχεία του συνολικού πίνακα που θέλουμε να υπολογίσουμε καθώς και τον υπολογισμό της περιστροφής της τροχιάς του σωματιδίου στους άξονες x,y και z. Έτσι λοιπόν, επιγραμματικά, οι προσεγγίσεις που ακολουθούμε είναι οι εξής:

1.6.1 Όλοι οι υπολογισμοί απο ένα thread

Πρόκειται για τον αλγόριθμο ο οποίος παίρνει τον πίνακα με τα φ,θ και και αναθέτει ένα ζεύγος ανά thread, πράγμα που σημαίνει πως το thread είναι υπεύθυνο να υπολογίσει τα cos(φ), cos(θ), sin(φ), sin(θ) και στην συνέχεια υπολογίζει ολόκληρο τον πίνακα 3χ3(9 στοιχείων).

1.6.2 Ξεχωριστοί υπολογισμοί

Πρόκειται για την υλοποίηση κατα την οποία αναθέτουμε σε ένα thread/ζεύγος να υπολογίσει τα cos(φ), cos(θ), sin(φ), sin(θ) και αφού υπολογιστούν όλα τότε αναθέτουμε σε καινούριο group απο threads να υπολογίσει τον πίνακα(1 thread/πίνακα)

1.6.3 Ξεχωριστά threads για κάθε γωνία

Η υλοποίηση αυτή είναι μια παραδοχή της παραπάνω προσέγγισης(“Ξεχωριστοί υπολογισμοί”) κατα την οποία χρησιμοποιούμε 4 thread/ζεύγος , 1 για κάθε cos ή sin. Και στην συνέχεια καλούμε εκ νέου ενα kernel το οποίο υπολογίζει τον πίνακα (1 thread/στοιχείο του πίνακα)

Οι παραπάνω προσεγγίσεις αφορούν μόνο την διαδικασία υπολογισμού ενός πίνακα και όχι εκείνη του υπολογισμού της περιστροφής της τροχιάς του σωματιδίου στον άξονα x,y και z. Οι υλοποιήσεις που περιλαμβάνουν και την περιστροφή χωρίζονται σε δυο βασικες κατηγορίες, εκεινή του υπολογισμού των 3 αξόνων απο το ίδιο thread που έκανε και τον υπολογισμό του πίνακα και εκείνης που αφότου τελειώσουν όλοι οι υπολογισμοί των πινάκων εκκινεί εκ νέου πυρήνες για τον υπολογισμό των αξόνων αναθέτοντας σε κάθε thread ξεχωριστά τον υπολογισμό ενός σετ αξόνων(x,y,z).

2. ΣΧΕΔΙΑΣΜΟΣ

2.1 Εισαγωγή

Σε αυτό το κεφάλαιο εξετάζουμε την δυναμική των διαφορετικών υλοποιήσεων για τον υπολογισμό μόνο του μέρους των πινάκων, καθώς και για το πως ανταποκρίνονται στο μέγεθος των δεδομένων που παίρνουν ώς είσοδο. Για να μπορούμε να έχουμε μια πιο ξεκάθαρη εικόνα των αποτελεσμάτων χρησιμοποιούμε μέγεθος δεδομένων που ξεπερνάει εκείνο που χρειάζεται το πρόγραμμα SIRENE, φτάνοντας έως και 1000% φορές πάνω.Το μέγεθος των εισόδων του προγράμματος που εξετάζουμε σε αυτό το κεφάλαιο είναι της τάξεως των 100.000 - 10.000.000 ζευγών γωνιών. Πριν ξεκινήσουμε όμως την ανάλυση των διαφορετικών αλγορίθμων που χρησιμοποιήθηκαν, ας δούμε κάποια βασικά συστατικά του προγράμματός μας για να καταλάβουμε ποια μέρη προσπαθούμε να παραλληλοποιήσουμε, καθώς και για την γενικότερη διαδικασία των υπολογισμών.

2.2 Σύνθεση του προγράμματος

Θα ορίσουμε ορισμένα βασικά χαρακτηριστικά του προγράμματός μας προκειμένου να δώσουμε μια ξεκάθαρη εικόνα για τα σημεία στα οποία θα γίνει η παραλληλοποίηση.

Έτσι λοιπόν το πρόγραμμά που εκτελεί την ρουτίνα των υπολογισμών έχει ως εξής.

Αρχικά υπάρχει ένα struct το οποίο περιλαμβάνει τις πληροφορίες που χρειαζόμαστε για κάθε ζεύγος των γωνιών φ,θ.
Έτσι λοιπόν για κάθε γωνία φ και θ που διαβάζουμε τα αποθηκεύουμε σε ένα struct στο οποίο struct τελικά θα συμπεριληφθεί και ο πίνακας, των 9 στοιχείων, που θα προκύψει απο τους πολλαπλασιασμούς μεταξύ των ημιτόνων και των συνημιτόνων των 2 αυτών γωνιών.Επειδή κάθε αρχείο περιέχει μεγάλο πλήθος(100.000-10.000.000) απο τέτοια ζεύγη θα δημιουργήσουμε έναν πίνακα απο structs, έναν πίνακα τόσων στοιχείων όσα και τα ζεύγη των γωνιών.

```
struct f_theta{
        double theta;
        double f;
        double calculations[9];
        double x;
        double y;
        double z;
};
```

Εικόνα 1 : Η δομή στην οποία αποθηκεύουμε όλη την πληροφορία ενός ζεύγους γωνιών

Μετά απο την κατάλληλη αρχικοποίηση και δέσμευση μνήμης ακολουθεί το διάβασμα όλων των ζευγών απο το αρχείο και η αποθήκευσή τους στον πίνακα απο structs που μόλις αναφέραμε. Να σημειώσουμε, όπως είπαμε και στην αρχή, πως επειδή η κάρτα γραφικών δεν προσφέρει υλικό για είσοδο και έξοδο απευθείας από και προς τον

χρήστη, το διάβασμα του αρχείου πρέπει να γίνει απαραίτητα απο την κεντρική μονάδα επεξεργασίας(CPU). Έτσι λοιπόν με την χρήση της συνάρτησης fscanf διαβάζουμε το αρχείο και έχουμε σαν αποτέλεσμα το ακόλουθο κομμάτι κώδικα.

```
while(fscanf(fp,"%lf ",&read_number_F)==1 && fscanf(fp,"%lf\n",&read_number_Theta)==1){
        arr[i].theta=read_number_Theta;
        arr[i].f = read_number_F;
```

Εικόνα 2 : Η διαδικασία ανάγνωσης του αρχείου

Στην συνέχεια δεσμεύουμε την απαραίτητη μνήμη στην κάρτα γραφικών στην οποία θα μεταφέρουμε τα δεδομένα που μόλις διαβάσαμε. Σκοπός αυτής της δέσμευσης είναι να μπορούν να τα επεξεργαστούν τα διάφορα νήματα που θα αναλάβουν την εκτέλεση της διαδικασίας των υπολογισμών. Εκείνα με την σειρά τους να αποθηκεύσουν τα αποτελέσματα με σκοπό την ανάκτηση τους απο την CPU για να μπορούν να χρησιμοποιηθούν στα υπόλοιπα μέρη του γενικότερου προγράμματος SIRENE. Για την διαδικασία της δέσμευσης χρησιμοποιούμε την συνάρτηση της CUDA cudaMalloc καθώς και την cudaMemset για να αρχικοποιήσουμε όλον τον πίνακα πριν μεταφέρουμε τα δεδομένα, με απώτερο σκοπό την αποφυγή σφάλματος σε περίπτωση που χρησιμοποιηθεί μη-αρχικοποιημένη μεταβλητή στην διαδικασία υπολογισμού.

```
cudaMalloc((void**)&device_test.theta,sizeof(double)*lines);
cudaMalloc((void**)&device_test.f,sizeof(double)*lines);
cudaMalloc((void**)&device_test.calculations,sizeof(double)*9*lines);
cudaMalloc((void**)&device_test.x,sizeof(double)*lines);
cudaMalloc((void**)&device_test.y,sizeof(double)*lines);
cudaMalloc((void**)&device_test.z,sizeof(double)*lines);

cudaMemset(device_test.theta,0.0,sizeof(double)*lines);
cudaMemset(device_test.f,0.0,sizeof(double)*lines);
cudaMemset(device_test.calculations,0.0,sizeof(double)*9*lines);
cudaMemset(device_test.x,0.0,sizeof(double)*lines);
cudaMemset(device_test.y,0.0,sizeof(double)*lines);
cudaMemset(device_test.z,0.0,sizeof(double)*lines);
```

Εικόνα 3 : Δέσμευση της απαιτούμενης μνήμης στην κάρτα γραφικών και αρχικοποίηση αυτής

Πρίν τελικά εκκινήσουμε τους πυρήνες/νήματα της κάρτας γραφικών μένει να μεταφέρουμε τα δεδομένα που μόλις διαβάσαμε στις θέσεις μνήμης που δεσμεύσαμε για την περαιτέρω επεξεργασία και υπολογισμούς απο τα νήματα. Για τον λόγο αυτό λοιπόν θα χρησιμοποιήσουμε την συνάρτηση cudaMemcpy για να μεταφέρουμε τα δεδομένα απο την CPU(host) στην κάρτα γραφικών(device).

```
cudaMemcpy(device_test.theta,arr.theta,sizeof(double)*lines,cudaMemcpyHostToDevice);
cudaMemcpy(device_test.f,arr.f,sizeof(double)*lines,cudaMemcpyHostToDevice);
cudaMemcpy(device_test.calculations,arr.calculations,sizeof(double)*lines*9,cudaMemcpyHostToDevice);
cudaMemcpy(device_test.x,arr.x,sizeof(double)*lines,cudaMemcpyHostToDevice);
cudaMemcpy(device_test.y,arr.y,sizeof(double)*lines,cudaMemcpyHostToDevice);
cudaMemcpy(device_test.z,arr.z,sizeof(double)*lines,cudaMemcpyHostToDevice);
```

Εικόνα 4 : Μεταφορά των δεδομένων απο την κεντρική μονάδα επεξεργασίας στην κάρτα γραφικών

Στην συνέχεια καλούμε τις συναρτήσεις με τους διαφορετικούς τρόπους υλοποίησης για να δούμε ποιος είναι εκείνος που προσφέρει την μεγαλύτερη επιτάχυνση. Οι διαφορετικές αυτές συναρτήσεις αναλύονται παρακάτω.

```
device<<<lines/1024,1024>>>(device_test);
```

Εικόνα 5 : Κλήση της συνάρτησης «device» στο περιβάλλον CUDA

Πριν τελικά ελευθερώσουμε τον όποιο χώρο δεσμεύσαμε, τόσο στην κεντρική μονάδα επεξεργασίας(CPU) όσο και στην κάρτα γραφικών(GPU) μεταφέρουμε τα δεδομένα, που αντιπροσωπεύουν τα αποτελέσματα των υπολογισμών, πίσω στην CPU έτσι ώστε να είναι προσβάσιμα απο το υπόλοιπο μέρος του προγράμματος, όπου αυτό τα χρειάζεται. Όπως και παραπάνω, για την διαδικασία αυτή χρησιμοποιήθηκε πάλι η cudaMemcpy.

```
cudaMemcpy(arr.theta,device_test.theta,sizeof(double)*lines,cudaMemcpyDeviceToHost);
cudaMemcpy(arr.f,device_test.f,sizeof(double)*lines,cudaMemcpyDeviceToHost);
cudaMemcpy(arr.calculations,device_test.calculations,sizeof(double)*9*lines,cudaMemcpyDeviceToHost);
cudaMemcpy(arr.x,device_test.x,sizeof(double)*lines,cudaMemcpyDeviceToHost);
```

Εικόνα 6 : Μεταφορά των δεδομένων απο την κάρτα γραφικών στην κεντρική μονάδα επεξεργασίας

Τέλος χρησιμοποιούμε τη συνάρτηση της CUDA cudaFree για να αποδεσμεύσουμε την μνήμη που χρησιμοποιήσαμε στην κάρτα γραφικών, καθώς και την free, σε επίπεδο κεντρικής μονάδας επεξεργασίας, για την αποδέσμευση της μνήμης που χρειάστηκε για το διάβασμα και την αποθήκευση των δεδομένων.

```
cudaFree(device_test.theta);
cudaFree(device_test.f);
cudaFree(device_test.calculations);
cudaFree(device_test.x);
cudaFree(device_test.y);
cudaFree(device_test.z);
```

Εικόνα 7 : Αποδέσμευση μνήμης στην κάρτα γραφικών

2.3 Οι επιμέρους συναρτήσεις των υπολογισμών

Αφού ορίσαμε τα βασικά χαρακτηριστικά του προγράμματός μας προχωράμε στην ανάλυση των συναρτήσεων που εκτελούνται στην κάρτα γραφικών και είναι υπεύθυνες για την διαδικασία των υπολογισμών. Να σημειώσουμε πως στο σημείο αυτό θα μιλήσουμε μόνο για τους υπολογισμούς που αφορούν τον υπολογισμό του πίνακα των 9 στοιχείων, καθώς και τον υπολογισμό των τριγωνομετρικών συναρτήσεων. Αργότερα θα έρθουμε να προσθέσουμε και την διαδικασία εύρεσης της περιστροφής του σωματιδίου στους 3 άξονες x,y και z.

2.3.1 Όλοι οι υπολογισμοί απο ένα thread

Όπως αναφέρθηκε και στο προηγούμενο κεφάλαιο: “Πρόκειται για τον αλγόριθμο ο οποίος παίρνει τον πίνακα με τα φ,θ και και αναθέτει ένα ζεύγος ανά thread, πράγμα που σημαίνει πως το thread είναι υπεύθυνο να υπολογίσει τα cos(φ),cos(θ),sin(φ),sin(θ) και στην συνέχεια υπολογίζει ολόκληρο τον πίνακα 3χ3(9 στοιχείων).”

Έτσι λοιπόν εξετάζουμε την απόδοση και την επιτάχυνση του γενικότερου προγράμματος αναθέτοντας σε ένα thread όλους τους υπολογισμούς που αφορούν ένα ζεύγος γωνιών. Με τον όρο “υπολογισμοί” αναφερόμαστε :

- στην χρήση των συναρτήσεων υπολογισμού των τριγωνομετρικών συναρτήσεων cos και sin για κάθε ζεύγος γωνιών
- στην χρήση των παραπάνω αποτελεσμάτων για τον υπολογισμό ενός μονοδιάστατου πίνακα 9 θέσεων όπου κάθε θέση αντιστοιχεί σε έναν πολλαπλασιασμό.

Αφού κληθεί η συνάρτηση, για κάθε ένα απο τα threads που θα χρησιμοποιήσουμε, υπολογίζουμε τον μοναδικό του αριθμό (id) το οποίο το χαρακτηρίζει και ανάλογα με τον αριθμό αυτό προσπελαύνει το αντίστοιχο μέρος του πίνακα. Έτσι λοιπόν το πρώτο thread θα επεξεργαστεί τα στοιχεία του πρώτου στοιχείου του πίνακα, το δεύτερο του δεύτερου στοιχείου κ.ο.κ.

```
int index=threadIdx.x + blockIdx.x * blockDim.x;
```

Εικόνα 8 : Υπολογισμός του μοναδικού αριθμού(id) που αντιστοιχεί σε κάθε thread

Στην συνέχεια, με την χρήση της βιβλιοθήκης math.h που περιλαμβάνει μαθηματικές συναρτήσεις, υπολογίζουμε τις τριγωνομετρικές συναρτήσεις cos και sin για κάθε μια απο τις γωνίες φ και θ. Έτσι λοιπόν προκύπτει το εξής κομμάτι κώδικα που αφορά τους 4 υπολογισμούς.

```
ct=cos(Theta);
st=sin(Theta);
cp=cos(F);
sp=sin(F);
```

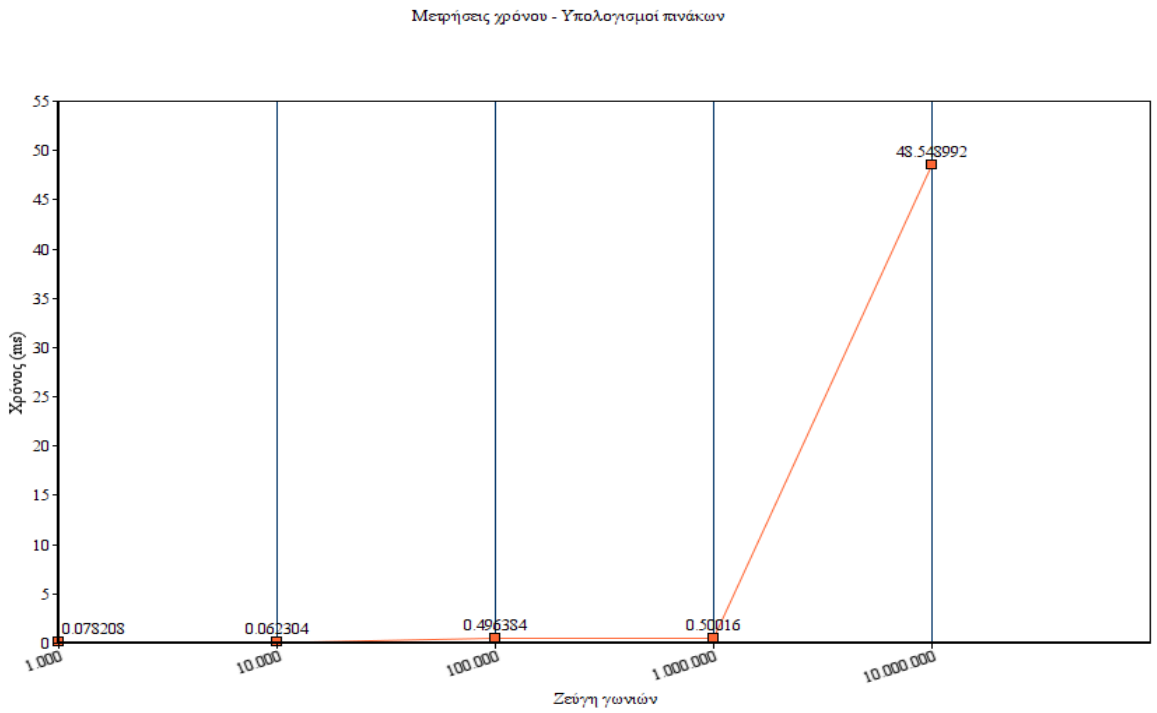
Εικόνα 9 : Εκτέλεση των τεσσάρων τριγωνομετρικών συναρτήσεων

Τέλος, πριν επιστρέψουμε στην κυρίως ρουτίνα, χρησιμοποιούμε αυτές τις 4 μεταβλητές(ct,st,cp,sp) για να δημιουργήσουμε τον πίνακα των 9 στοιχείων. Παρακάτω βλέπουμε την διαδικασία προσπέλασης και υπολογισμού των 9 στοιχείων του πίνακα απο ένα thread.

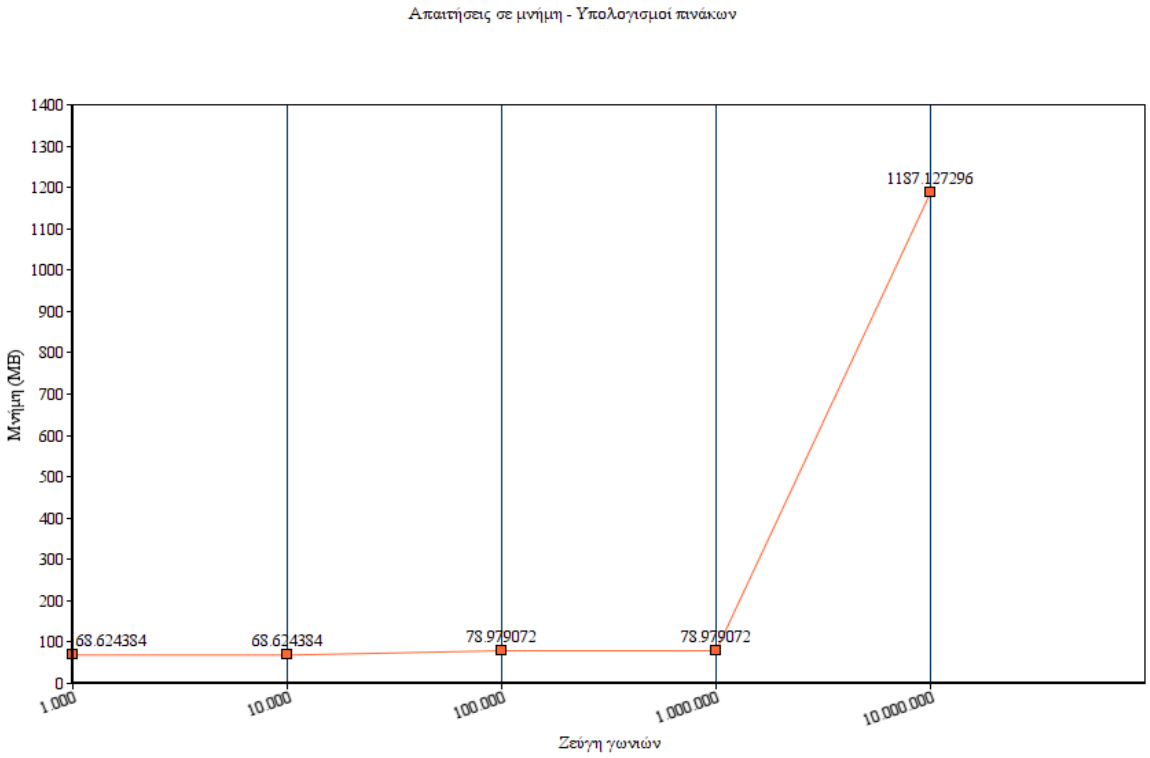
```
array.calculations[index*9+0]=ct*cp;
array.calculations[index*9+1]=ct*sp;
array.calculations[index*9+2]=-st;
array.calculations[index*9+3]=-sp;
array.calculations[index*9+4]=cp;
array.calculations[index*9+5]=0;
array.calculations[index*9+6]=st*cp;
array.calculations[index*9+7]=st*sp;
array.calculations[index*9+8]=ct;
```

Εικόνα 10 : Διαδικασία υπολογισμού του πίνακα 9 στοιχείων

Το βασικό χαρακτηριστικό αυτής της υλοποίησης είναι η διεκπεραίωση της διαδικασίας, για ένα στοιχείο του πίνακα, απο ένα thread και εκεί είναι η βασική διαφορά μεταξύ των υλοποιήσεων όπως θα δούμε και παρακάτω.



Σχήμα 2 : Απαιτήσεις σε χρόνο(ms) για την εκτέλεση της διαδικασίας



Σχήμα 3: Αποτελέσματα μετρήσεων με βάση τις απαιτήσεις σε μνήμη

2.3.2 Ξεχωριστοί υπολογισμοί

“Πρόκειται για την υλοποίηση κατα την οποία αναθέτουμε σε ένα thread/ζεύγος να υπολογίσει τα cos(φ), cos(θ), sin(φ), sin(θ) και αφού υπολογιστούν όλα τότε αναθέτουμε σε καινούριο group απο threads να υπολογίσει τον πίνακα (1 thread/πίνακα)”

Σε αυτήν την προσέγγιση υπάρχουν 2 συναρτήσεις τις οποίες χρησιμοποιούμε για την ολοκλήρωση της διαδικασίας. Η μια συνάρτηση είναι εκείνη η οποία θα υπολογίσει όλες τις τριγωνομετρικές συναρτήσεις και θα τις αποθηκεύσει σε ένα προσωρινό πίνακα, ενώ η δεύτερη είναι εκείνη που θα προσπελάσει τον προσωρινό αυτό πίνακα και θα χρησιμοποιήσει τα δεδομένα του για να εξάγει τελικώς τον πίνακα των 9 διαστάσεων.

Έτσι λοιπόν έχουμε τα εξής κομμάτια κώδικα:

Αρχικά και οι 2 συναρτήσεις πρέπει να κάνουν τις απαραίτητες πράξεις για να υπολογίσουν τον μοναδικό αριθμό (id) του κάθε thread.

```
int index=threadIdx.x + blockIdx.x * blockDim.x;
```

Εικόνα 11 : Υπολογισμός του μοναδικού αριθμού(id) που αντιστοιχεί σε κάθε thread

Στην συνέχεια η πρώτη συνάρτηση, εκείνη που είναι υπεύθυνη για τις τριγωνομετρικές συναρτήσεις, θα περιλαμβάνει το παρακάτω κομμάτι κώδικα.

```
cos_sin_Array[(index*4)+0]=cos(Theta);
cos_sin_Array[(index*4)+1]=sin(Theta);
cos_sin_Array[(index*4)+2]=cos(F);
cos_sin_Array[(index*4)+3]=sin(F);
```

Εικόνα 12 : Υπολογισμός των 4 τριγωνομετρικών συναρτήσεων και αποθήκευση αυτών στον προσωρινό πίνακα"cos_sin_Array"

Κάθε thread δηλαδή είναι υπεύθυνο για τον υπολογισμό του ημιτόνου και του συνημιτόνου ενός ζεύγους γωνιών φ και θ. Βλέπουμε πως σε αυτήν την υλοποίηση χρησιμοποιούμε έναν επιπλέον πίνακα με όνομα “cos_sin_Array” ο οποίος είναι υπεύθυνος για την προσωρινή αποθήκευση των τριγωνομετρικών αποτελεσμάτων καθώς και για να “περάσει” τα δεδομένα αυτά απο την μία συνάρτηση στην άλλη. Αυτό λοιπόν προυποθέτει την δέσμευση επιπλέον μνήμης, τόσο σε επίπεδο CPU όσο και σε GPU χρησιμοποιώντας τις συναρτήσεις cudaMalloc και cudaMemset.

```
cudaMalloc((void**)&cos_sin_Array,sizeof(double)*4*lines);
```

Εικόνα 13 : Δέσμευση μνήμης στην κάρτα γραφικών

```
cudaMemset(cos_sin_Array,0.0,sizeof(double)*4*lines);
```

Εικόνα 14 : Αρχικοποίηση μνήμης στην κάρτα γραφικών

Μετα το τέλος της πρώτης συνάρτησης σειρά έχει η συνάρτηση που είναι υπεύθυνη για να υπολογίσει των πίνακα των 9 στοιχείων. Έτσι λοιπόν αφού ανακτήσουμε πρώτα τα δεδομένα που χρειαζόμαστε απο τον προσωρινό πίνακα των τριγωνομετρικών αποτελεσμάτων στην συνέχεια κάνουμε πολλαπλασιασμούς μεταξύ αυτών απο όπου και προκύπτουν τα τελικά αποτελέσματά μας.

```
ct=cos_sin_Array[(index*4)+0];
st=cos_sin_Array[(index*4)+1];
cp=cos_sin_Array[(index*4)+2];
sp=cos_sin_Array[(index*4)+3];
```

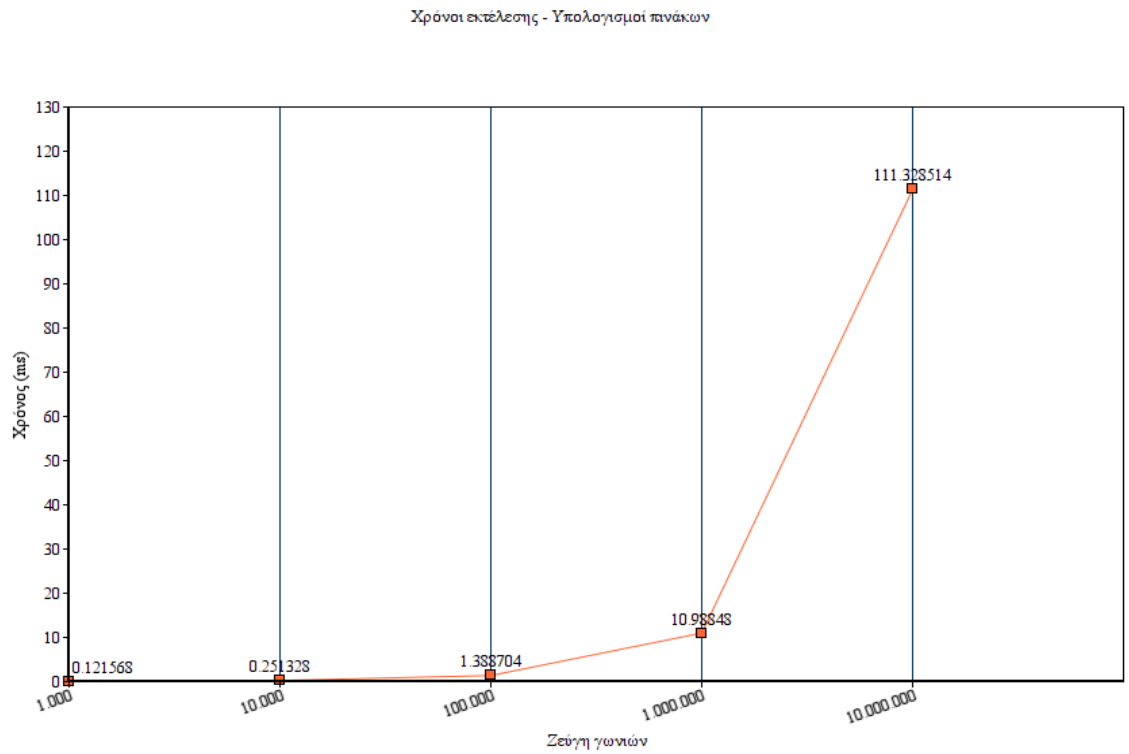
Εικόνα 15 : Ανάκτηση των τριγωνομετρικών αποτελεσμάτων απο τον προσωρινό πίνακα "cos_sin_Array"

Ανάκτηση αποτελεσμάτων απο τον προσωρινό πίνακα “cos_sin_Array”.

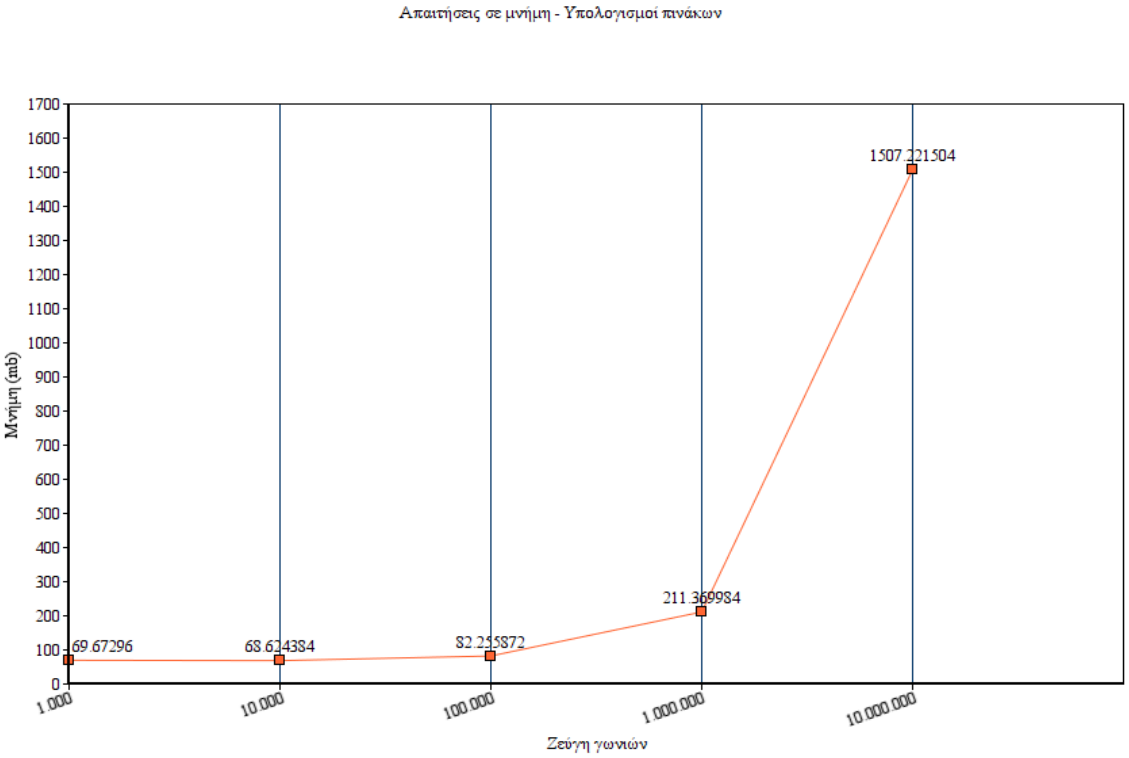
```
array.calculations[index*9+0]=ct*cp;
array.calculations[index*9+1]=ct*sp;
array.calculations[index*9+2]=-st;
array.calculations[index*9+3]=-sp;
array.calculations[index*9+4]=cp;
array.calculations[index*9+5]=0;
array.calculations[index*9+6]=st*cp;
array.calculations[index*9+7]=st*sp;
array.calculations[index*9+8]=ct;
```

Εικόνα 16 : Υπολογισμός των 9 στοιχείων του τελικού πίνακα

Όπως τονίσαμε και στην πρώτη περίπτωση οι βασικές διαφορές μεταξύ των υλοποιήσεων είναι η διάσπαση των βασικών χαρακτηριστικών της διαδικασίας υπολογισμού σε μια ή περισσότερες συναρτήσεις, οι οποίες παίρνουν την σειρά τους, η μια μετα την άλλη, στην κάρτα γραφικών.



Σχήμα 4 : Απαιτήσεις σε χρόνο(ms) για την εκτέλεση της διαδικασίας



Σχήμα 5 : Απαιτήσεις σε μνήμη(mb) για την εκτέλεση της διαδικασίας

2.3.3 Ξεχωριστά threads για κάθε γωνία

“Η υλοποίηση αυτή είναι μια παραδοχή της παραπάνω προσέγγισης(“Ξεχωριστοί υπολογισμοί”) κατα την οποία χρησιμοποιούμε 4 thread/ζεύγος , 1 για κάθε cos ή sin. Και στην συνέχεια καλούμε εκ νέου ενα kernel το οποίο υπολογίζει τον πίνακα(1 thread/στοιχείο του πίνακα)”

Όπως εύκολα μπορεί να καταλάβει κάποιος, η υλοποίηση αυτή είναι μια παραλλαγή της παραπάνω, που λέει πως ενα thread θα υπολογίσει όλα τα ημίτονα και τα συνημίτονα των γωνιών φ και θ, θα τα αποθηκεύσει σε έναν προσωρινό πίνακα και στην συνέχεια θα αναθέσουμε σε καινούρια συνάρτηση το διάβασμα του προσωρινού αυτού πίνακα και της εξαγωγής των 9 τελικών υπολογισμών που χρειαζόμαστε. Στην συγκεκριμένη λοιπόν περίπτωση θα επαναλάμβάνουμε ακριβώς την ίδια διαδικασία με την διαφορά οτι για κάθε 1 ημίτονο ή συνημίτονο, είτε της γωνίας φ είτε της γωνίας θ, τον υπολογισμό θα τον κάνει ξεχωριστό thread. Αυτό πρακτικά σημαίνει ότι το cos(φ) θα υπολογιστεί απο ένα thread, το cos(θ) απο άλλο κ.ο.κ. Όταν τελειώσει ο υπολογισμός όλων των τριγωνομετρικών συναρτήσεων τότε θα περάσουμε στην διαδικασία υπολογισμού του πίνακα των 9 στοιχείων, όπου και θα έχουμε τον υπολογισμό του 1ου στοιχείου του πίνακα απο ένα thread, τον υπολογισμό του 2ου απο ένα άλλο κ.ο.κ.

Έτσι λοιπόν έχουμε το παρακάτω κομμάτι κώδικα :

Αρχικά και οι 2 συναρτήσεις πρέπει να κάνουν τις απαραίτητες πράξεις για να υπολογίσουν τον μοναδικό αριθμό(id) του κάθε thread.

```
int index=threadIdx.x + blockIdx.x * blockDim.x;
```

Εικόνα 17 : Υπολογισμός του μοναδικού αριθμού(id) που αντιστοιχεί σε κάθε thread

Στην συνέχεια ορίζουμε σε ποια γωνία αντιστοιχεί στο συγκεκριμένο thread

```
Theta = array[index/4].theta;
F = array[index/4].f;
```

Εικόνα 18 : Ανάκτηση της πληροφορίας για τις γωνίες "φ" και "θ" απο τον πίνακα

Και συνεχίζουμε υπολογίζοντας την μια απο τις τέσσερις τριγωνομετρικές συναρτήσεις, ανάλογα με τον μοναδικό αριθμό(id) του thread.

```
switch(index%4){
   case 0:
       cos_sin_Array[((index/4)*4)+0]=cos(Theta);
       break;
   case 1:
       cos_sin_Array[((index/4)*4)+1]=sin(Theta);
       break;
   case 2:
       cos_sin_Array[((index/4)*4)+2]=cos(F);
       break;
   case 3:
       cos_sin_Array[((index/4)*4)+3]=sin(F);
       break;
}
```

Εικόνα 19 : Switch – case για την επιλογή που αντιστοιχεί στον μοναδικό αριθμό(id) του κάθε thread

Στο τέλος της συνάρτησης που εκτελεί τον παραπάνω κώδικα θα έχουμε έναν πίνακα (cos_sin_Array) στον οποίο έχουν υπολογιστεί οι 4 συναρτήσεις για κάθε ζεύγος γωνιών φ και θ. Υπενθυμίζουμε ότι στην συγκεκριμένη υλοποίηση κάθε ένα thread αναλαμβάνει και τον υπολογισμό μιας απο αυτές τις 4 συναρτήσεις για ένα ζεύγος γωνιών, εν αντιθέσει με την προηγούμενη υλοποίηση όπου ένα thread εκτελούσε και τις 4 συναρτήσεις για κάθε ζεύγος.

Αφού ολοκληρωθεί ο υπολογισμός του πίνακα, λοιπόν, προχωράμε στην κλήση νέας συνάρτησης η οποία είναι υπεύθυνη για τον υπολογισμό των 9 στοιχείων του τελικού πίνακα.
Στην συνάρτηση αυτή κάθε thread θα υπολογίζει και ένα απο τα εννέα στοιχεία του πίνακα για κάθε ζεύγος γωνιών φ και θ.

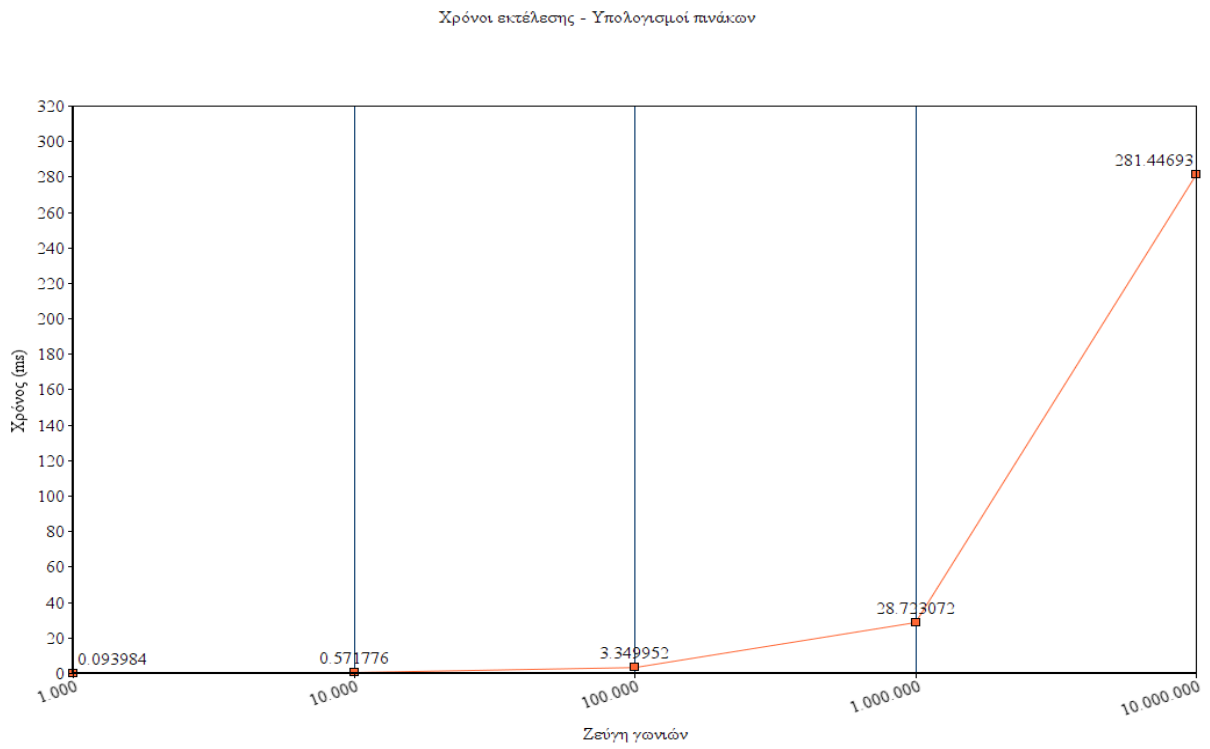
```
ct=cos_sin_Array[((index/9)*4)+0];
st=cos_sin_Array[((index/9)*4)+1];
cp=cos_sin_Array[((index/9)*4)+2];
sp=cos_sin_Array[((index/9)*4)+3];
```

Εικόνα 20 : Ανάκτηση της πληροφορίας για τα τριγωνομετρικά αποτελέσματα

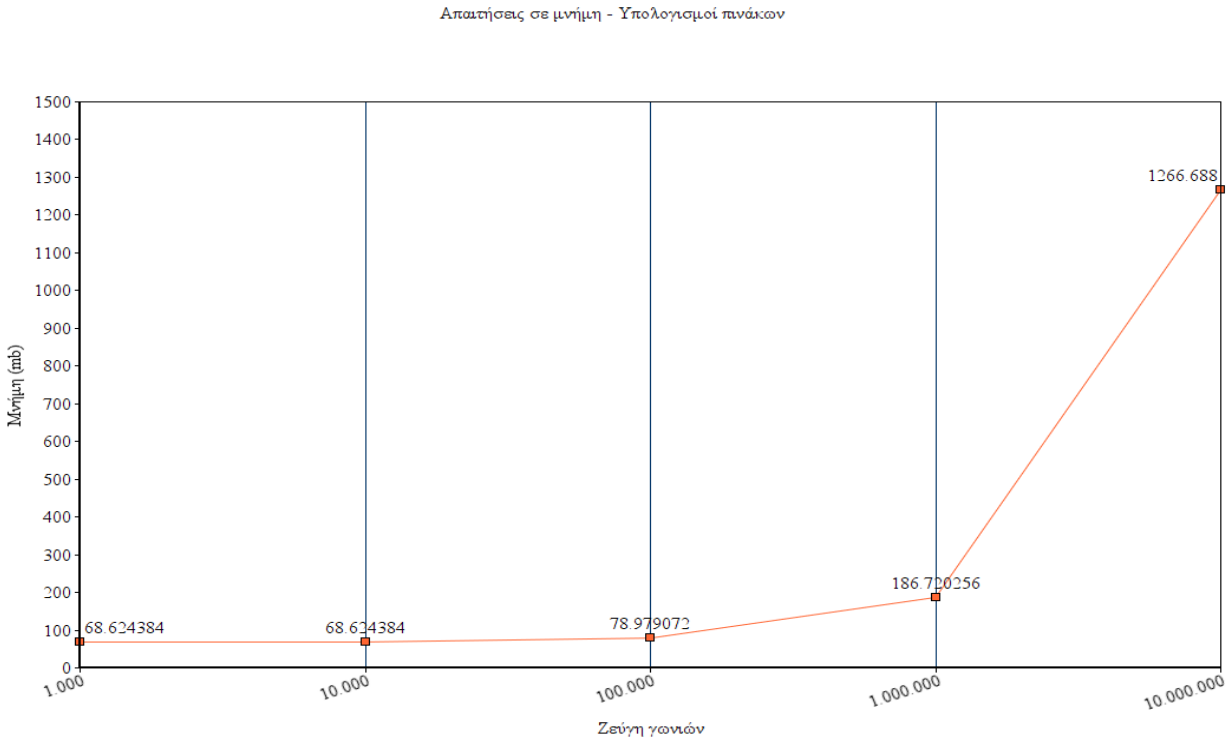
Και αφού ανακτήσουμε απο τον πίνακα τον αριθμό που μας ενδιαφέρει τότε εκτελούμε τον πολλαπλασιασμό ώς εξής:

```
switch(index%9){
    case 0:
      array[index/9].calculations[0]=ct*cp;
      break;
    case 1:
      array[index/9].calculations[1]=ct*sp;
      break;
    case 2:
      array[index/9].calculations[2]=-st;
      break;
    case 3:
      array[index/9].calculations[3]=-sp;
      break;
    case 4:
      array[index/9].calculations[4]=cp;
      break;
    case 5:
      array[index/9].calculations[5]=0;
      break;
    case 6:
      array[index/9].calculations[6]=st*cp;
      break;
    case 7:
      array[index/9].calculations[7]=st*sp;
      break;
    case 8:
      array[index/9].calculations[8]=ct;
      break;
}
```

Εικόνα 21 : Υπολογισμός του τελικού πίνακα με βάση τον μοναδικό αριθμό(id) του κάθε thread

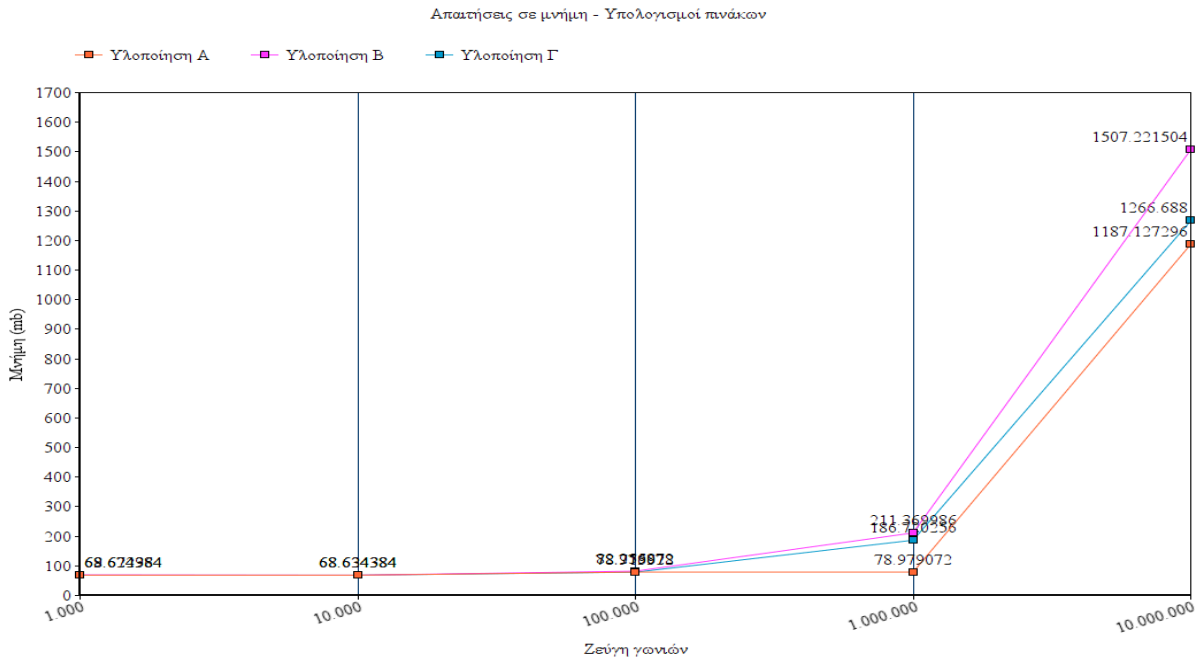


Σχήμα 6 : Απαιτήσεις σε χρόνο(ms) για την εκτέλεση της διαδικασίας

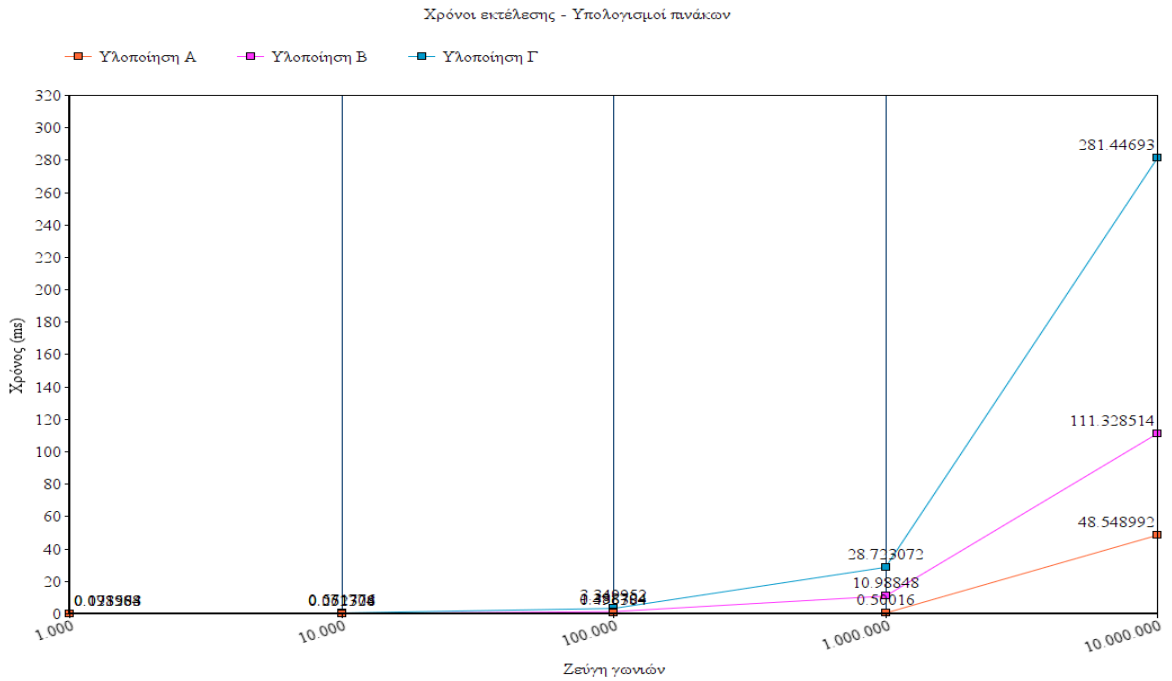


Σχήμα 7 : Απαιτήσεις σε μνήμη(mb) για την εκτέλεση της διαδικασίας

Παρακάτω παραθέτουμε συγκεντρωτικά τους χρόνους εκτέλεσης καθώς και τις απαιτήσεις σε μνήμη των παραπάνω εκτελέσεων.



Σχήμα 8 : Απαιτήσεις σε μνήμη(mb) για την διαδικασία υπολογισμού πινάκων



Σχήμα 9 : Απαιτήσεις σε μνήμη(mb) για την εκτέλεση της διαδικασίας

Είναι εμφανές, μέσα απο τις μετρήσεις, πως η υλοποίηση με την καλύτερη κλιμάκωση είναι η πρώτη, εκείνη δηλαδή η οποία αναθέτει σε ενα thread την υλοποίηση ολόκληρης της διαδικασίας, τον υπολογισμό των τριγωνομετρικών συναρτήσεων δηλαδή καθώς και τον υπολογισμό των 9 στοιχείων απο τα οποία αποτελείται ο πίνακας. Αυτό μπορεί να συμβαίνει διότι, όπως είπαμε και στην αρχή του κειμένου, το μεγαλύτερο ποσοστό του χρόνου στην εκτέλεση ενός προγράμματος που εκμεταλλεύεται την κάρτα γραφικών, το κατέχει η δέσμευση μνήμης καθώς και η μεταφορά των στοιχείων απο και προς την GPU. Σε όλες τις περιπτώσεις, πλην της πρώτης, έχουμε μεγαλύτερη δέσμευση μνήμης και μεταφορά δεδομένων διότι χρησιμοποιούμε παραπάνω πίνακες με σκοπό να χωρίσουμε την όλη διαδικασία σε 2 μικρότερες υποκατηγορίες, εκείνες των τριγωνομετρικών υπολογισμών και εκείνες τους υπολογισμού του πίνακα. Στην προσπάθεια μας να υποδιαιρέσουμε το πρόβλημα σε μικρότερα συναντάμε ως εμπόδιο την μεταφορά και την δέσμευση μνήμης.

Έτσι λοιπόν επιλέξαμε την πρώτη ως ιδανικότερη, εκ των τριών, υλοποιήση και σύμφωνα με αυτήν θα προχωρήσουμε στην περαιτέρω βελτιστοποίησή της.

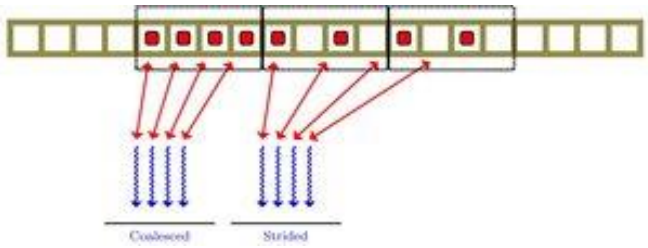
Οι βελτιώσεις, μέσα απο την καταγραφή χρόνων και απαιτήσεων σε μνήμη, που παρουσιάζονται στα παρακάτω κεφάλαια και παραγράφους αφορούν την πρώτη υλοποίηση, την οποία και επιλέξαμε ώς βέλτιστη, ανάμεσα στις τρείς.

2.4 Συνένωση - Coalescing

2.4.1 Τί είναι η συνένωση(coalescing);

Ένας απο τους πρώτους και σημαντικότερους σταθμούς στην προσπάθεια βελτιστοποίησης, της υλοποίησης που επιλέξαμε, είναι η συνένωση(coalescing). Η συνένωση είναι μια διαδεδομένη μέθοδος για την διαχείρηση της μνήμης την οποία προσπελαύνουν τα νήματα όταν πρόκειται να χρησιμοποιήσουν δεδομένα που βρίσκονται σε αυτήν.

Η μνήμη RAM, που χρησιμοποιεί ένα πρόγραμμα, χωρίζεται σε μεγάλα μπλοκ στα οποία αποθηκεύονται τα δεδομένα που χρειάζεται ένα πρόγραμμα κατα την εκτέλεσή του. Λόγω των πολλών προγραμμάτων, ή της υψηλής απαίτησης σε μνήμη ενός προγράμματος, τα δεδομένα που περιέχει η μνήμη δεν είναι πάντα τα ίδια, και αυτό διότι εναλλάσονται για να μπορούν να εξυπηρετηθούν πολλαπλά προγράμματα σε λιγότερο χρόνο. Για παράδειγμα εάν έχουμε μνήμη συνολικού μεγέθους 1 GB και έχουμε 7 προγράμματα που τρέχουν τα οποία χρειάζονται απο 200 MB μνήμης τότε βλέπουμε πως η μνήμη δεν αρκεί, μιας και συνολικά θέλουμε 7x200MB = 1.4GB. Για να μπορέσει η μνήμη να διαχειριστει τις απαιτήσεις των προγραμμάτων αρχίζει και εναλλάσει, ανάλογα με το πιο πρόγραμμα έχει σειρά να εκτελεστεί στον πυρήνα, τα ζητούμενα μπλοκς εντός και εκτός RAM γράφοντάς τα στον δίσκο και ανακτώντας τα όταν χρειαστεί. Όπως εύκολα μπορεί κάποιος να καταλάβει η διαδικασία της εναλλαγής των διάφορων δεδομένων εντός και εκτός της μνήμης είναι μια διαδικασία η οποία μπορεί να αυξήσει κατα πολύ τον χρόνο εκτέλεσης ενός προγράμματος.



Σχήμα 10 : Σχηματική απεικόνιση της σημασίας της συνένωσης

Το σχήμα 10 μας βοηθάει να καταλάβουμε τη διαφορά μεταξύ των δυο διαφορετικών προσεγγίσεων προσπέλασης της μνήμης. Έτσι λοιπόν αριστερά βλέπουμε ένα κομμάτι μνήμης το οποίο προσπελαύνεται απο τα threads με τρόπο συνενωμένο(coalesced). Τέσσερα threads διαβάζουν τέσσερις διαδοχικές θέσεις μνήμης, εκμεταλλεύονται δηλαδή ολόκληρο το μπλοκ μνήμης που έχουν στην διάθεσή τους, ενώ αντίθετα βλέπουμε πως αν δεν γίνει σωστη διαχείριση του τρόπου προσπέλασης τοτε μπορεί να έχουμε περιπτώσεις όπως εκεινή που βρίσκεται ακριβώς δίπλα, κατά την οποία τέσσερα threads διαβάζουν δεδομένα απο 2 μπλοκ μνήμης, καθυστερόντας έτσι τη γενικότερη εκτέλεση του προγράμματος.

2.4.1.1 Χρήση μεθόδων συνένωσης για την βελτιστοποίηση στον υπολογισμό των πινάκων

Φέρνοντας το πρόβλημα στην δικιά μας περίπτωση βλέπουμε πως λόγω των υψηλών απαιτήσεων σε μνήμη ίσως χρειαστεί να εφαρμόσουμε και εμείς μεθόδους για την συνένωση μιας και μπορεί ένα μπλοκ απο threads να προσπελαύνει τα δεδομένα, τον πίνακα των γωνιών δηλαδή, με τέτοιο τρόπο ώστε η μνήμη RAM να χρειαστεί χρόνο για να φέρει όλα τα απαιτούμενα μπλοκς απο τον σκληρό δίσκο.

Έτσι λοιπόν προσπαθούμε να δομήσουμε το διάβασμα των δεδομένων απο την μνήμη με έναν τρόπο τέτοιο ώστε να αποφύγουμε καθυστερήσεις που οφείλονται στην εναλλαγή των δεδομένων εντός και εκτός μνήμης RAM.

Το βασικότερο μέρος του προγράμματός μας, στο οποίο μπορεί να υπάρξει βελτίωση είναι εκείνο της αποθήκευσης των δεδομένων που διαβάζονται απο το αρχείο σε σχετικούς πίνακες. Αρχικά η δομή του προγράμματος χρησιμοποιεί ενα struct, μια ομάδα δηλαδή απο μεταβλητές τις οποίες χρησιμοποιεί για να αποθηκεύσει ότι πληροφορία διαβάζει και υπολογίζει, δηλαδή από το διάβασμα των γωνιών φ και θ μέχρι και την τελική μορφή του πίνακα εννέα στοιχείων για κάθε ζεύγος.

```
struct f_theta{
        double theta;
        double f;
        double calculations[9];
        double x;
        double y;
        double z;
};
```

Εικόνα 22 : Η δομή στην οποία αποθηκεύουμε όλη την πληροφορία ενός ζεύγους γωνιών

Στην παραπάνω εικόνα βλέπουμε την δομή τους struct που κρατάει τις πληροφορίες για καθε ενα ζεύγος γωνιών φ και θ.

Με την σειρά, όπως αυτές εμφανίζονται στην φωτογραφία, έχουμε τις εξής μεταβλητές:

- theta : Ο αριθμός που αντιστοιχεί στην γωνία “θ” ενός ζεύγους
- f : Ο αριθμός που αντιστοιχεί στην γωνία “φ” ενός ζεύγους
- calculations[9] : Ο πίνακας των τελικών υπολογισμών
- x : Η περιστροφή στον άξονα χ
- y : Η περιστροφή στον άξονα y
- z : Η περιστροφή στον άξονα z

Για να μπορέσουμε να κρατήσουμε τις πληροφορίες που χρειαζόμαστε για όλα τα ζεύγη των γωνιών χρησιμοποιούμε ένα πίνακα απο structs, δηλαδή έναν πίνακα τόσων θέσεων όσο ~~για~~ και το πλήθος των ζευγών που διαβάζουμε απο το αρχείο. Προφανώς, κάθε στοιχείο του πίνακα περιέχει πληροφορίες που αντιστοιχούν στην δομή της παραπάνω εικόνας.

Έτσι λοιπόν, όταν αναθέτουμε στα διαφορετικά threads της κάρτας γραφικών το διάβασμα των στοιχείων από τον πίνακα (υπενθυμίζουμε πως ένα thread αναλαμβάνει να εκτελέσει τους υπολογισμούς των στοιχείων μιας θέσης του πίνακα) ζητάμε από την μνήμη RAM της κάρτας των γραφικών να μας φέρει τα δεδομένα ενός struct, παρόλο που εμείς σκοπεύουμε να χρησιμοποιήσουμε ενα μέρος αυτών και όχι όλα. Για την διαδικασία, δηλαδή, χρειαζόμαστε μόνο τις μεταβλητές f, theta, calculations[9] και όχι τις x, y, z. Φέρνουμε επομένως μνήμη την οποία δεν σκοπεύουμε να την χρησιμοποιήσουμε. Επειδή δεν μπορούμε να δώσουμε εντολή στην κάρτα γραφικών για το τι να φέρει και τι όχι θα πρέπει να αλλάξουμε την δομή του προγράμματός μας.

2.4.2 Πίνακες απο struct ή struct απο πίνακες

Η μέχρι τώρα υλοποίηση έχει ως βασική δομή δεδομένων έναν πίνακα απο structs. Αυτό όπως είδαμε και παραπάνω μπορεί να μας μειώνει σημαντικά τους χρόνους προσπέλασης στα δεδομένα γι αυτό θα αλλάξουμε την δομή αυτή σε ένα struct απο πίνακες.

```
struct f_theta{

        double *theta;
        double *f;
        double *calculations;
        double *x;
        double *y;
        double *z;

};
```

Εικόνα 23 : Η νέα δομή που θα αποθηκεύσουμε τις πληροφορίες για το κάθε ζεύγος γωνιών

Αυτή τη φορά τα δεδομένα θα τα αποθηκεύσουμε σε ένα μόνο struct, το οποίο βέβαια θα περιλαμβάνει έναν πίνακα, τόσων θέσεων όσες και το πλήθος των ζευγών φ και θ, για τις γωνίες “φ”, έναν πίνακα για τις γωνίες “θ” κ.ο.κ. Με αυτόν τον τρόπο όταν ένα thread ζητάει να έχει πρόσβαση σε ορισμένα δεδομένα τότε η μνήμη φροντίζει να φέρει όλους τους πίνακες που αφορούν τα “φ”,”θ” και “calculations” εξαλείφοντας έτσι το πρόβλημα της μη-χρήσης όλων των διαθέσιμων θέσεων μνήμης απο όλα τα threads.

Έτσι λοιπόν απο την παρακάτω εικόνα του πίνακα απο structs

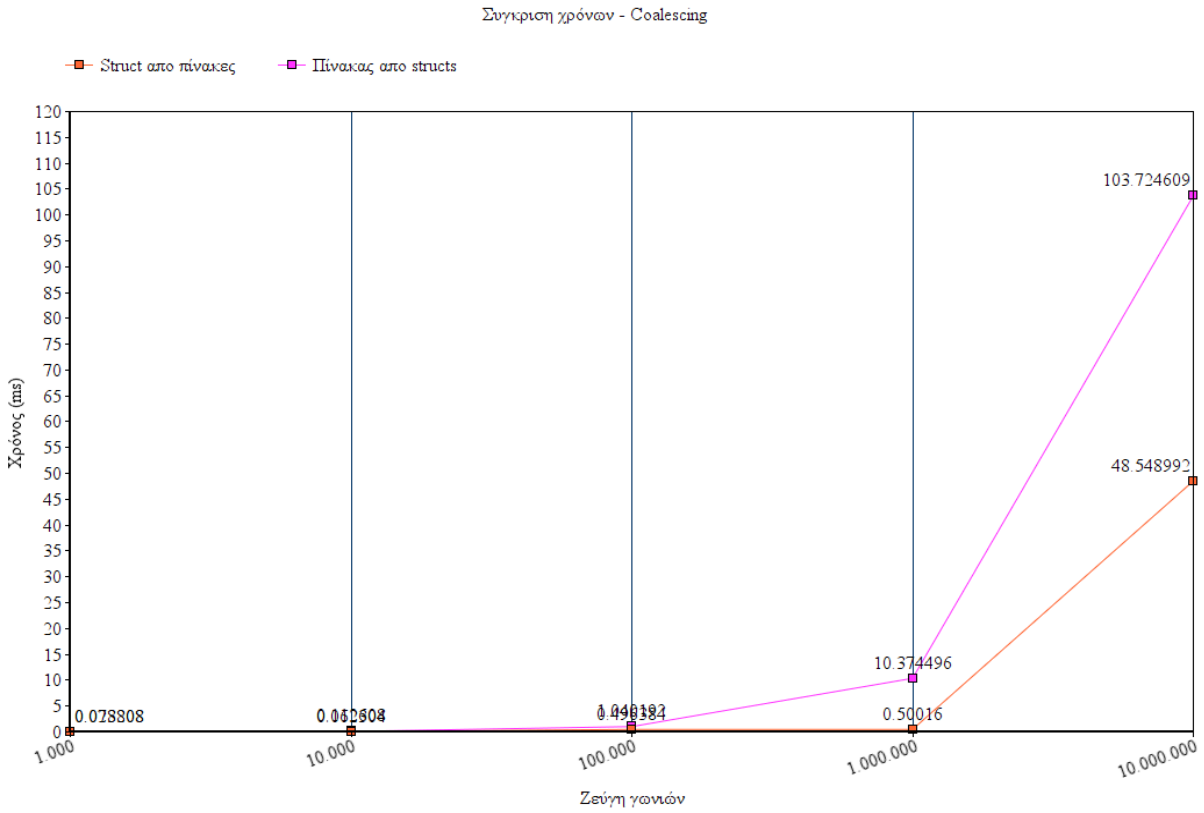
```
array[index].calculations[0]=ct*cp;
array[index].calculations[1]=ct*sp;
array[index].calculations[2]=-st;
array[index].calculations[3]=-sp;
array[index].calculations[4]=cp;
array[index].calculations[5]=0;
array[index].calculations[6]=st*cp;
array[index].calculations[7]=st*sp;
array[index].calculations[8]=ct;
```

Εικόνα 24 : Ο υπολογισμός του πίνακα με την παλιά μέθοδο προσπέλασης

μεταφερόμαστε στην λογική του struct απο πίνακες

```
array.calculations[index*9+0]=ct*cp;
array.calculations[index*9+1]=ct*sp;
array.calculations[index*9+2]=-st;
array.calculations[index*9+3]=-sp;
array.calculations[index*9+4]=cp;
array.calculations[index*9+5]=0;
array.calculations[index*9+6]=st*cp;
array.calculations[index*9+7]=st*sp;
array.calculations[index*9+8]=ct;
```

Εικόνα 25 : Ο υπολογισμός του πίνακα με την νέα μέθοδο προσπέλασης



Σχήμα 11 : Σύγκριση χρόνων μεταξύ των δυο παραδοχών της δομής δεδομένων του προγράμματος

Σύγκριση απαιτήσεων σε μνήμη δεν παρουσιάζεται καθώς είναι αρκετά ξεκάθαρο πως δεν αλλάζουμε τίποτα παρα μόνο τον τρόπο με τον οποίο προσπελαύνουμε στην μνήμη, και όχι το μέγεθος που χρειαζόμαστε για τα δεδομένα μας.

3. ΣΥΓΚΡΙΣΕΙΣ

3.1 Εισαγωγή

Σε αυτό το κεφάλαιο θα εξετάσουμε την ανταπόκριση του αλγόριθμού μας σε πραγματικές εισόδους του προγράμματος SIRENE και πως μπορούμε να βελτιστοποιήσουμε την απόδοση. Πρόκειται για το κεφάλαιο στο οποιο θα παντρέψουμε τις διαφορετικές τεχνικές και βιβλιοθήκες που υπάρχουν τόσο σε επίπεδο κεντρικής μονάδας επεξεργασίας όσο και σε επίπεδο κάρτας γραφικών. Προχωρώντας ένα βήμα παραπέρα θα μιλήσουμε και για τα μέρη που υπάρχουν πριν και μετά την διαδικασία των υπολογισμών και πως αυτά επηρεάζουν τις μετρήσεις και τον σχεδιασμό που κάναμε στο παραπάνω κεφάλαιο.

Όπως είδαμε λοιπόν και στην άρχη, το πρόγραμμα SIRENE, σε ένα μέρος του, χρειάζεται να υπολογίσει ορισμένες γωνίες (πίνακας 9 στοιχείων) που αντιπροσωπεύουν τη μορφολογία ενός φωτοανιχνευτή. Είδαμε επίσης πως οι πίνακες αυτοί είναι όσοι και τα ζεύγη των γωνιών τα οποία διαβάζουν απο ένα αρχείο, μια τάξη μεγέθους των 117.000 ζευγών. Αυτό στο οποίο δεν έχουμε δώσει έμφαση μέχρι στιγμής είναι το πόσες φορές επαναλαμβάνεται η διαδικασία αυτή. Είδαμε στην αρχή πως το πρόγραμμα χωρίζεται σε ορισμένες, σημαντικές, δομές επανάληψης, όπως εκείνη των γεγονότων(events) η οποία περιλαμβάνει μια επανάληψη για τα tracks. Μέσα σε κάθε επανάληψη των tracks διαβάζουμε ένα αρχείο που περιλαμβάνει ζεύγη απο γωνίες και δημιουργούμε ορισμένους πίνακες.

Στο παραπάνω κεφάλαιο ασχοληθήκαμε με την βελτιστοποίηση των αλγορίθμων που αφορούν αυστηρά το διάβασμα απο ένα αρχείο και στη συνέχεια τον υπολογισμό όλων των πινάκων. Αυτό που δεν έχουμε δεί, και θα μελετήσουμε στο κεφάλαιο αυτό, είναι πως αλληλεπιδρά ο αλγόριθμος οταν θα χρειαστεί να τον επαναλάβουμε παραπάνω απο 1 φορές. Το μέγεθος δεδομένων είναι τέτοιο ώστε να έχουμε εμφανές κέρδος απο την χρήση της κάρτας γραφικών; Μπορούμε να συνδυάσουμε την κάρτα γραφικών με την CPU;

3.2 Κάρτα γραφικών(GPU) ή κεντρική μονάδα επεξεργασίας(CPU);

Η κάρτα γραφικών, όπως έχουμε αναφέρει αρκετές φορές, είναι ένα εξαιρετικό εργαλείο για την επιτάχυνση πολλών διεργασίων που μπορεί να κάνει ένα υπολογιστής. Προσφέρει πολλούς μικροπυρήνες οι οποίοι μπορούν να προγραμματιστούν έτσι ώστε να λειτουργούν τελείως αυτόνομα και να μην εξαρτώνται απο τα αποτελέσματα των υπολοίπων. Επίσης έχουμε πεί πως σε ένα πρόγραμμα που χρησιμοποιεί την κάρτα γραφικών το μεγαλύτερο μέρος του χρόνου καταναλώνεται στην μεταφορά δεδομένων απο και προς την κάρτα καθως και την δέσμευση του χώρου που χρειάζονται τα δεδομένα αυτά. Αυτό δημιουργεί απο μόνο του ένα “όριο” στο ποτε συμφέρει η χρήσης της κάρτας γραφικών και αυτό διότι τα δεδομένα μπορεί να είναι είτε πολυ μικρής τάξεως είτε η δομή του προγράμματός μας να χρειάζεται μια συνεχή μεταφορά απο και προς την κάρτα, καταλήγοντας έτσι να μην συμφέρει η χρήση αυτής.

Στη διαδικασία υπολογισμού των πινάκων, για δεδομένα τα οποία ξεπερνάνε κατα πολυ σε πλήθος αυτά που χρειάζεται το SIRENE, είδαμε πως ο αλγόριθμος παίζει εναν πολυ

σημαντικό ρόλο στην επιτάχυνση και γι αυτόν τον λόγο επιλέξαμε, με βάση τις μετρήσεις μας, έναν απο τους 3 ως επικρατέστερο. Αν παρατηρήσουμε και πάλι τα γραφήματα των χρόνων και των απαιτήσεων σε μνήμη θα δούμε ότι για πολυ χαμηλό πλήθος εισόδων(10.000,100.000 κλπ) οι αλγόριθμοι δεν έχουν διαφορές, τουλάχιστον όχι τέτοιες που να μας κάνουν ξεκάθαρο το ποιος αλγόριθμος είναι πιο αποδοτικός. Αυτό ενισχύει την άποψη περι “ορίου” στο ποτε πρέπει να χρησιμοποιούμε μια κάρτα γραφικών και αν τελικά κερδίζουμε σε απόδοση συγκριτικά με το να υλοποιούσαμε τις διαδικασίες σε επίπεδο κεντρικής μονάδας επεξεργασίας.

Αφού λοιπόν προσθέσουμε μια επιπλέον επανάληψη, εκεινή που αντιπροσωπεύει τα tracks, θα παραθέσουμε γραφήματα που αφορούν τους χρόνους εκτέλεσης τόσο σε CPU όσο και σε GPU.

Αρχικά θα ορίσουμε τη διαδικασία των υπολογισμών μέσα σε μια συνάρτηση την οποία θα ονομάσουμε “calculations_routine” και θα περιλαμβάνει το παρακάτω κομμάτι κώδικα

```
FILE *fp;
double read_number_F,read_number_Theta;

f_theta arr;
f_theta device_test;

fp= fopen(filename,"r");

arr.theta = (double*)malloc(sizeof(double)*lines);
arr.f =  (double*)malloc(sizeof(double)*lines);
arr.calculations =  (double*)malloc(sizeof(double)*9*lines);
arr.x =  (double*)malloc(sizeof(double)*lines);
arr.y =  (double*)malloc(sizeof(double)*lines);
arr.z =  (double*)malloc(sizeof(double)*lines);
```

Εικόνα 26 : Αρχικοποίηση και δέσμευση μνήμης για το διάβασμα της εισόδου

```
while(fscanf(fp,"%lf ",&read_number_F)==1 && fscanf(fp,"%lf\n",&read_number_Theta)==1){
        arr.theta[i]=read_number_Theta;
        arr.f[i] = read_number_F;
        arr.x[i]=15.452;
        arr.y[i]=1.326;
        arr.z[i]=92.92;
        i++;

}
```

Εικόνα 27 : Διάβασμα και αρχικοποίηση όλων των δεδομένων απο το αρχείο

```
for(int i=0;i<lines;i++){

        double Theta = arr.theta[i];
        double F = arr.f[i];

        double ct=cos(Theta);
        double st=sin(Theta);
        double cp=cos(F);
        double sp=sin(F);

        arr.calculations[i*9+0]=ct*cp;
        arr.calculations[i*9+1]=ct*sp;
        arr.calculations[i*9+2]=-st;
        arr.calculations[i*9+3]=-sp;
        arr.calculations[i*9+4]=cp;
        arr.calculations[i*9+5]=0;
        arr.calculations[i*9+6]=st*cp;
        arr.calculations[i*9+7]=st*sp;
        arr.calculations[i*9+8]=ct;

}
```

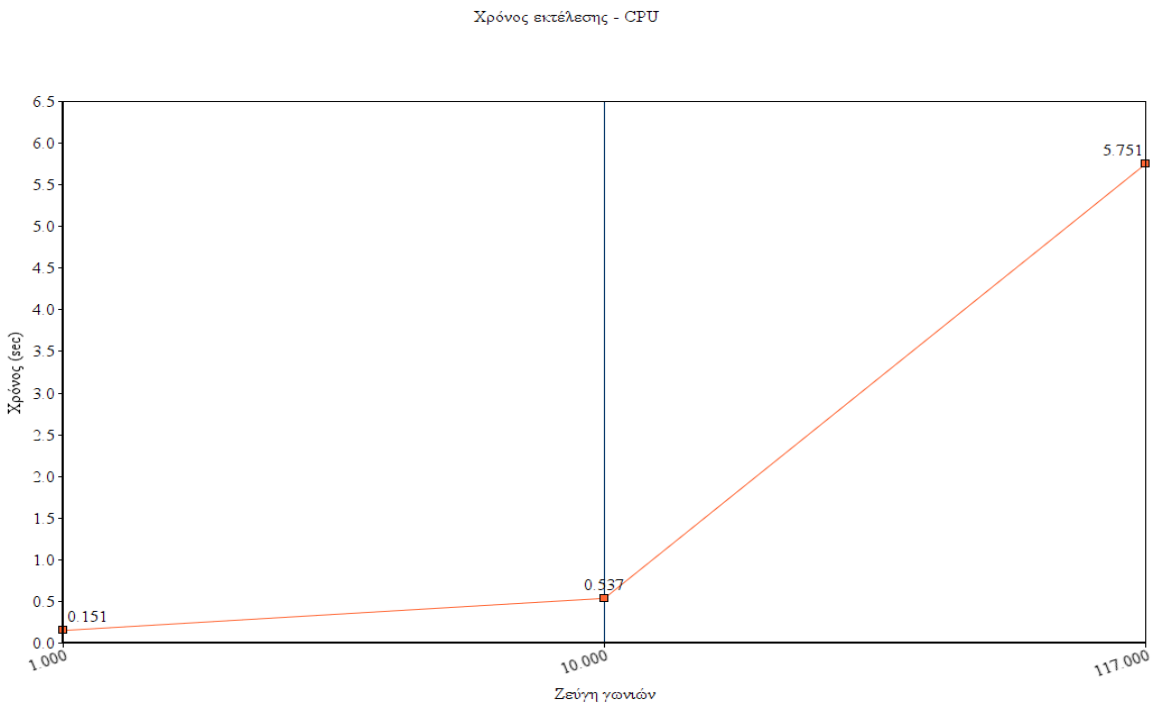
Εικόνα 28 : Τελική εκτέλεση των υπολογισμών

Στην συνέχεια θα ορίσουμε μια επανάληψη “for” για την που αντιπροσωπεύει τη δομή επανάληψης που χρησιμοποιείται για να εκφράσουμε τα διαφορετικά tracks ενός γεγονότος(event)

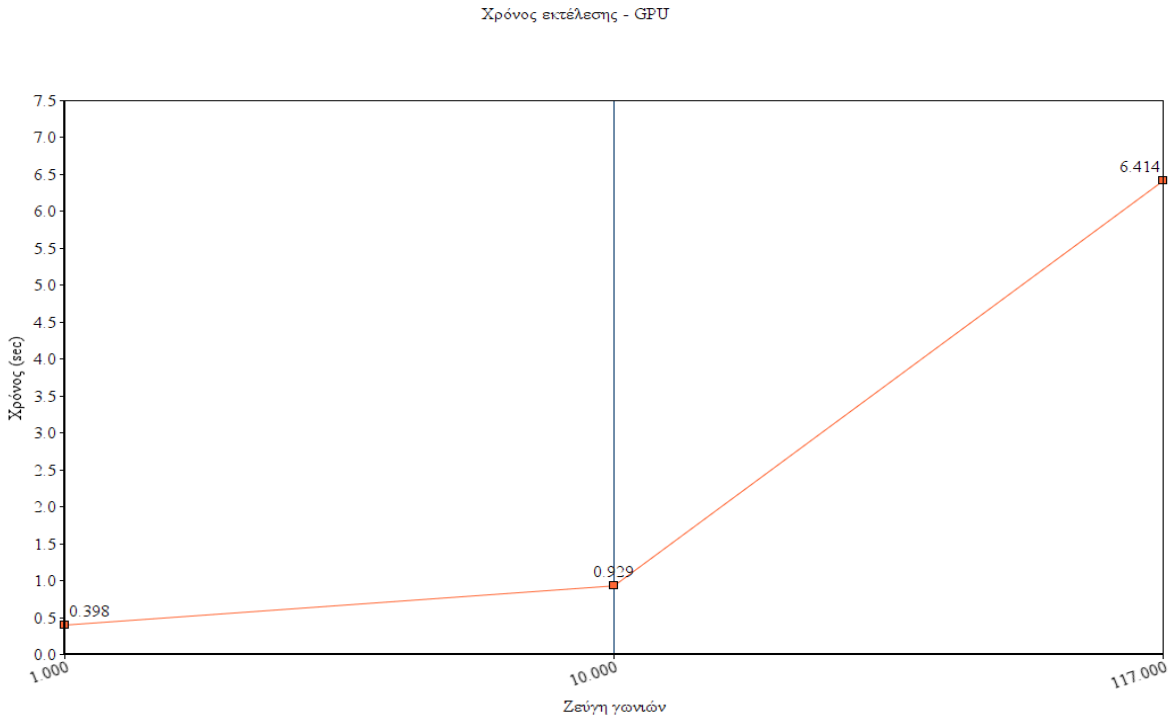
```
for(int i=0;i<100;i++){

        f_theta array;

        array=calculations_routine("input.txt",lines);
}
```

Εικόνα 29 : Η επανάληψη forπου προσωμοιώνει την διαδικασία των tracks

Αφού ορίσαμε το περιεχόμενο του κώδικα θα προχωρήσουμε στις μετρήσεις που αφορούν την διαδικασία της γενικότερης προσομοίωσης σε επίπεδο CPU και σε επίπεδο GPU.



Σχήμα 12 : Χρόνοι εκτέλεσης σε CPU



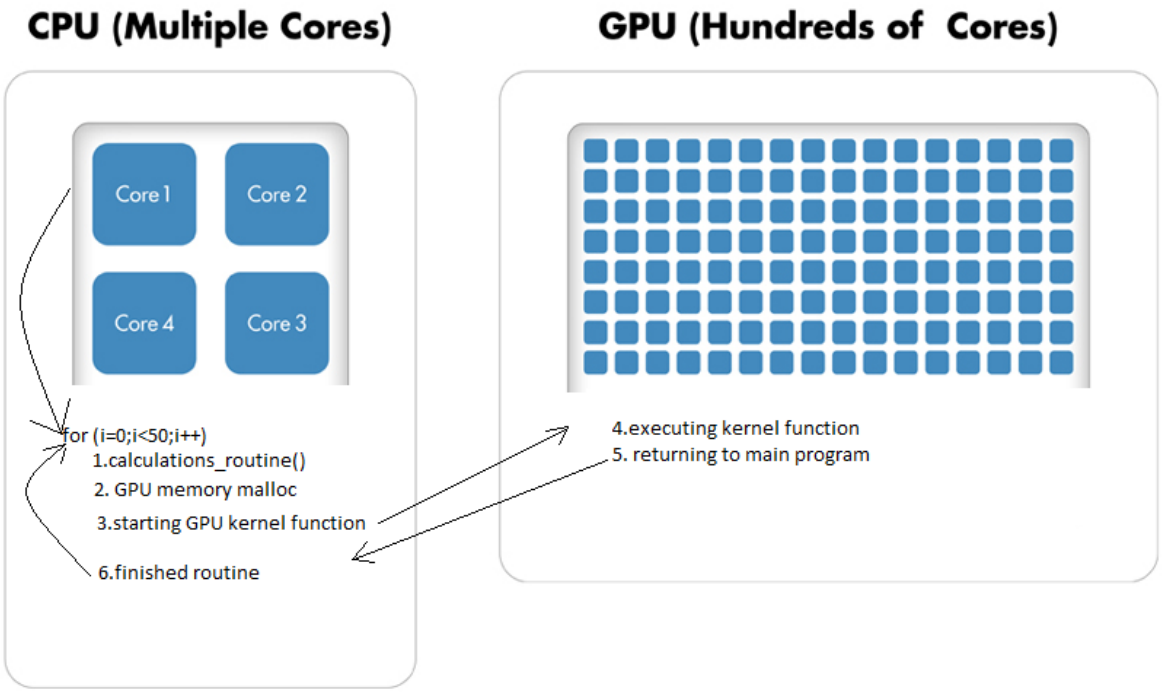
Σχήμα 13 : Χρόνοι εκτέλεσης σε GPU

Όπως βλέπουμε και απο τα σχεδιαγράμματα η χρήση της κάρτας γραφικών(GPU) για τον υπολογισμό των πινάκων όλων των track είναι οριακά χειρότερη απο την χρήση της κεντρικής μονάδας επεξεργασίας(CPU) μιας και για την συνηθισμένη είσοδο του SIRENE(117.000 ζεύγη) η κάρτα γραφικών χρειάζεται περίπου 12% παραπάνω χρόνο για να ολοκληρωθεί, ένα ποσοστό που έχει εκθετική αύξηση ανάλογα με το μέγεθος της εισόδου.

3.3 Συνδυασμός τεχνικών και βιβλιοθηκών CPU & GPU

Στο παραπάνω κεφάλαιο είδαμε τον χρόνο που χρειάζεται για να ολοκληρωθεί η διαδικασία των υπολογισμων για όλα τα tracks τόσο σε επίπεδο κεντρικής μονάδας επεξεργασίας όσο και σε επίπεδο κάρτας γραφικών, οδηγώντας μας σε συμπεράσματα που ενισχύουν την χρήση της CPU για ένα πρόγραμμα το οποίο έχει τόσο μικρή κλίμακα δεδομένων.

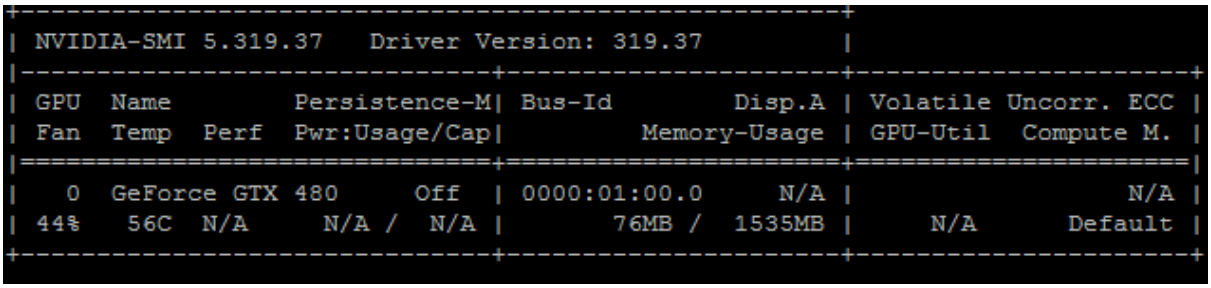
Έχοντας καταλήξει στην βέλτιστη μορφή του προγράμματός μας, σε επίπεδο κάρτας γραφικών, και παρατηρώντας πως ακόμα και αυτή η μορφή δεν μπορεί να υπερισχύσει της κεντρικής μονάδας επεξεργασίας για ένα τόσο “μικρό” πρόγραμμα η επόμενή μας σκέψη είναι ο συνδυασμός τεχνικών για τον διαμοιρασμό των δεδομένων μεταξύ CPU & GPU.
Αρχικά θα εξηγήσουμε τον τρόπο με τον οποίο λειτουργεί μια κάρτα γραφικών όταν καλούμε το πρόγραμμα των υπολογισμών.



Σχήμα 14 : Αναπαράσταση της διαδικασίας εκτέλεσης του αλγόριθμου σε CPUκαι GPU

1. Αρχικά ένας απο τους πυρήνες (CPU) αναλαμβάνει την εκτέλεση μιας επανάληψης. Μέσα στην επανάληψη καλούμε την συνάρτηση “calculations_routine”
2. Η συνάρτηση δεσμεύει τον χώρο που θα χρειαστεί η GPU
3. Καλούμε την συνάρτηση στην κάρτα γραφικών
4. Εκτέλεση του κώδικα της συνάρτησης απο έναν αριθμό απο threads
5. Επιστροφή στην κύρια μονάδα επεξεργασίας
6. Τέλος της επαναληπτικής διαδικασίας, ξεκινάμε απο την αρχή.

Απο την διαδικασία κλήσης της συνάρτησης, που θα εκτελεστεί στους πυρήνες της κάρτας γραφικών, μέχρι την επιστροφή αυτής, η CPU παραμένει σε κατάσταση αναμονής αφήνοντας έτσι ανεκμετάλλευτους πόρους που θα μπορούσαν να χρησιμοποιηθούν για περαιτέρω υπολογισμούς. Επίσης, λόγω του μικρού μεγέθους του προβλήματος δεν εκμεταλλευόμαστε πλήρως της δυνατότητες της κάρτας γραφικών όπως φαίνεται και στο παρακάτω σχήμα:



Εικόνα 30 : Οι απαιτήσεις σε μνήμη για την εκτέλεση του προγράμματος.

Βλέπουμε πως κατα την εκτέλεση του αλγόριθμού μας στην κάρτα γραφικών(για μέγεθος εισόδου 117.000 ζεύγη) καταλαμβάνουμε μόλις 76/1535 MB, δηλαδή μόνο το 4.9% της διαθέσιμης μνήμης!

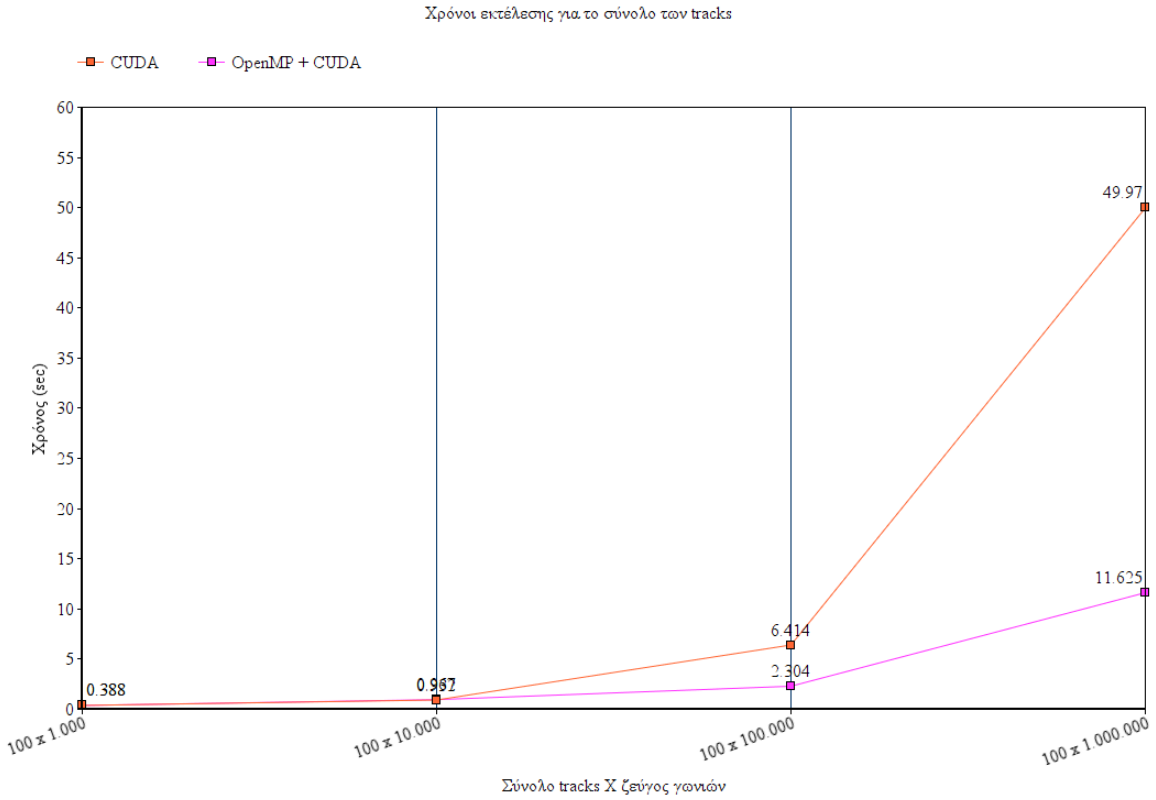
Μια πιθανή βελτίωση που εμφανίζεται στο σημείο αυτό είναι ο συνδυασμός τεχνικών για την παράλληλη εκκίνηση, παραπάνω απο μιας, συναρτήσεων με σκοπό να εκμεταλλευτούμε όλο το εύρος των πόρων και παράλληλα να μην χρειαζεται η CPU να περιμένει την επιστροφή των συναρτήσεων απο την GPU ώστε να ξεκινήσει τους υπολογισμούς για το επόμενο track.

3.3.1 Open MP μαζί με CUDA

Μέχρι τώρα χρησιμοποιούσαμε μόνο την βιβλιοθήκη συναρτήσεων της Nvidia που είναι υπεύθυνη για την εκκίνηση πολλών πυρήνων σε επίπεδο κάρτας γραφικών. Με σκοπό την παράλληλη εκκίνηση πολλαπλών ομάδων απο πυρήνες θα χρησιμοποιήσουμε την βιβλιοθήκη OpenMP(Open Multi-processing). Ποια είναι όμως η λογική πίσω απο αυτή την υλοποίηση;

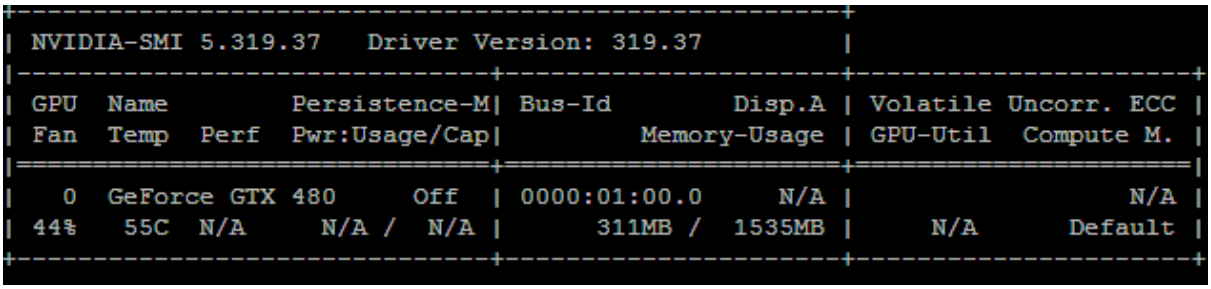
Η OpenMP χρησιμοποιείται για τον παράλληλο προγραμματισμό σε επίπεδο κεντρικής μονάδας επεξεργασίας. Πρόκειται δηλαδή για μια βιβλιοθήκη που δίνει τη δυνατότητα στο προγραμματιστή να εκμεταλλευτεί όλους τους πόρους μιας CPU με σκοπό να μοιράσει τον φόρτο ενος προγράμματος σε όλους τους επεξεργαστές που παρέχει αυτή ή ακόμα και σε όλα τα threads τα οποία έχει διαθέσιμα ένας επεξεργαστής.

Αν ανατρέξουμε λίγο παραπάνω, στην ανάλυση των εργασιών που κάνει κάθε μια εκ των CPU και GPU απο την αρχή μέχρι το τέλος του προγράμματός μας, θα δούμε πως ένα thread απο έναν πυρήνα αναλαμβάνει να εκτελέσει κάθε φορά την επανάληψη for που εμείς έχουμε ορίσει. Χρησιμοποιώντας την βιβλιοθήκη OpenMP αναγκάζουμε την κεντρική μονάδα επεξεργασίας να διαιρέσει την κεντρική δομή επανάληψης for σε μικρότερες. Συγκεκριμένα έχοντας μια for 100 επαναλήψεων και 32 threads στην διάθεση μας, χωρίζουμε σε 32 ισότιμα κομμάτια των ~3 επαναλήψεων και τα δίνουμε στην CPU να τα εκτελέσει παράλληλα. Αυτό στην πράξη σημαίνει πως η κλήση της συνάρτησης υπολογισμού των πινάκων θα κληθεί ταυτόχρονα απο 32 πηγές, καταλαμβάνοντας έτσι πολυ μεγαλύτερο ποσοστό στην κάρτα μνήμης , εκμεταλλευόμενοι πολλούς περισσότερους πόρους.



Σχήμα 15 : Χρόνοι εκτέλεσης συνδυασμού OpenMP μαζί με CUDAγια το σύνολο των tracks(100)

Όπως βλέπουμε και απο το παραπάνω διάγραμμα ο συνδυασμός των βιβλιοθηκών OpenMP και CUDA μας δίνει μια σταθερή επιτάχυνση η οποία είναι της τάξης του ~3x.



Εικόνα 31 : Οι απαιτήσεις σε μνήμη για την εκτέλεση του προγράμματος

Απο την παραπάνω εικόνα μπορούμε να διακρίνουμε το ποσοστό της μνήμης που καταναλώνει αυτην την φορά το πρόγραμμά μας. Έχουμε αυξήσει τις απαιτήσεις απο 74 MB στα 311 MB. Να σημειώσουμε εδώ οτι σε πραγματικό χρόνο η GPU χρησιμοποιεί πολύ παραπάνω μνήμη απο την αυτή που βλέπουμε στην παραπάνω εικόνα, αλλά λόγω της ταχύτητας που εναλλάσονται οι, αυτη την φορά παραπάνω απο μια, συναρτήσεις μέσα στον πυρήνα είναι δύσκολο η μέτρηση να δείξει τον πραγματικό αριθμό των MB που χρησιμοποιούνται.

4. ΠΕΡΙΣΤΡΟΦΗ ΣΤΟΥΣ ΑΞΟΝΕΣ

4.1 Εισαγωγή

Στο κεφάλαιο αυτό θα αναφερθούμε στο μέρος εκείνο των υπολογισμών για τις περιστροφές των σωματιδίων στους άξονες x,y και z. Προκειται για μια ομάδα υπολογισμών που, λόγω της φύσης του προγράμματός, πρέπει να εκτελείται ξεχωριστά απο τα υπόλοιπα μέρη, μιας και τα δεδομένα που παράγονται χρησιμοποιούνται σε δεύτερο χρόνο. Μεσα στο κεφάλαιο αυτό θα αναλύσουμε την φυσική σημασία των υπολογισμών, τις πράξεις που περιλαμβάνει, γιατι δεν μπορούμε να τα υπολογίσουμε απο κοινού με τους πίνακες που είδαμε στα προηγούμενα κεφάλαια, πως επηρεάζουν οι διαφορετικές υλοποιήσεις την απόδοση του συγκεκριμένου σημείου.

4.2 Η φυσική σημασία

Όπως έχουμε πεί και στην αρχή, το συγκεκριμένο κομμάτι που μελετάμε προσομοιώνει την ύπαρξη και την οριοθέτηση των φωτοανιχνευτών του τηλεσκοπίου νετρίνων μέσα στον χώρο, με απώτερο σκόπο την μελέτη του κατα πόσο ένα σωματίδιο, σε ευθεία τροχιά, θα συναντήσει έναν απο τους παραπάνω φωτοανιχνευτές στην πορεία του. Γι' αυτόν τον λόγο δημιουργούμε έναν πίνακα 9 στοιχείων που αποτελεί τον πίνακα στροφής και στην συνέχεια τις συνενταγμένες x,y και z οι οποίες αναφέρονται στην θέση ενός φωτοπολλαπλασιαστή μέσα στο γενικότερο χώρο του τηλεσκοπίου των νετρίνων.

4.3 Οι πράξεις που περιλαμβάνει

Έχοντας μια γενικότερη εικόνα της φυσικής σημασίας των περιστροφών μπορούμε να προχωρήσουμε στον ορισμό των πράξεων που αφορούν το σημείο αυτό. Έτσι λοιπόν έχουμε 3 βασικούς πολλαπλασιασμούς.

```
array[index].x = array[index].calculations[0]*array[index].x + array[index].calculations[1]*array[index].y+
array[index].calculations[2]*array[index].z;
```

Εικόνα 32 : Υπολογισμός του X, παλιά υλοποίηση

```
array[index].y = array[index].calculations[3]*array[index].x + array[index].calculations[4]*array[index].y+
array[index].calculations[5]*array[index].z;
```

Εικόνα 33 : Υπολογισμός του y, παλιά υλοποίηση

```
array[index].z = array[index].calculations[6]*array[index].x + array[index].calculations[7]*array[index].y+
array[index].calculations[8]*array[index].z;
```

Εικόνα 34 : Υπολογισμός z, παλιά υλοποίηση

Όπως βλέπουμε και στις παραπάνω εικόνες οι υπολογισμοί της περιστροφής στους 3 άξονες περιλαμβάνουν την χρήση του πίνακα των 9 στοιχειών που έχουμε ήδη υπολογίσει καθώς και τις προηγούμενες τιμές των x,y και z. Με άλλα λόγια βλέπουμε πως η περιστροφή εξαρτάται απο την προηγούμενη θέση του φωτοπολλαπλασιαστή στον χώρο καθώς και απο τις νέες θέσεις που έχουν οι ανιχνευτές του φωτοπολλαπλασιαστή μέσα σε αυτόν. Οι νέες θέσεις προκύπτουν από τον πολλαπλασιασμό του πίνακα στροφής με τις (παλιές) θέσεις στο αρχικό σύστημα συντεταγμένων (πριν την περιστροφή).

Σε προηγούμενα κεφάλαια είχαμε ορίσει την έννοια της συνένωσης(coalescing) και πως αυτή, μέσα απο την διαχείριση της πρόσβασης στην μνήμη του υπολογιστή, επηρεάζει την γενικότερη απόδοση του προγράμματός μας. Όπως είναι φυσικό το κομμάτι των περιστροφών δεν αποτελεί την εξαίρεση στον κανόνα αυτόν, έτσι λοιπόν παρακάτω θα παραθέσουμε τις εικόνες που δείχνουν το κομμάτι του κώδικα, τροποποιημένο κατα τέτοιο τρόπο ώστε να επιτυγχάνεται η τεχνική της συνένωσης.

```
array.x[index] = array.calculations[index*9+0]*array.x[index] + array.calculations[index*9+1]*array.y[index]+
array.calculations[index*9+2]*array.z[index];
```

Εικόνα 35 : Υπολογισμός x, νέα υλοποίηση

```
array.y[index] = array.calculations[index*9+3]*array.x[index] + array.calculations[index*9+4]*array.y[index]+
array.calculations[index*9+5]*array.z[index];
```

Εικόνα 36 : Υπολογισμός y, νέα υλοποίηση

```
array.z[index] = array.calculations[index*9+6]*array.x[index] + array.calculations[index*9+7]*array.y[index]+
array.calculations[index*9+8]*array.z[index];
```

Εικόνα 37 : Υπολογισμός z, νέα υλοποίηση

Γενικότερα βλέπουμε πως ο υπολογισμός των περιστροφών περιλαμβάνει 3 πολλαπλασιασμούς που αναφέρονται στην χρήση του πίνακα των 9 στοιχείων που έχει υπολογιστεί στο προηγούμενο βήμα και στις αρχικές τιμές των x,y και z.

Έτσι λοιπόν έχουμε :

$$x = πίνακαςΥπολογισμών[0] * x + πίνακαςΥπολογισμών[1] * y + πίνακαςΥπολογισμών[2] * z$$

$$y = πίνακαςΥπολογισμών[3] * x + πίνακαςΥπολογισμών[4] * y + πίνακαςΥπολογισμών[5] * z$$

$$z = πίνακαςΥπολογισμών[6] * x + πίνακαςΥπολογισμών[7] * y + πίνακαςΥπολογισμών[8] * z$$

4.4 Η ανεξάρτητη κλήση της συνάρτησης περιστροφών

Οι περιστροφές είναι μέρος μιας μεμονωμένης συνάρτησης, η κλήση της οποίας γίνεται σε χρόνο ουδέτερο, συγκριτικά με εκείνον των αρχικών υπολογισμών. Έτσι λοιπόν έχουμε δημιουργήσει μια νεα συνάρτηση η οποία περιέχει τα εξής κομμάτια κώδικα:

Την πράξη για τον υπολογισμό του μοναδικού αριθμού(id) για το κάθε thread

```
int index = threadIdx.x + blockIdx.x * blockDim.x;
```

Εικόνα 38: Διαδικασία υπολογισμού του μοναδικού αριθμού(id) για κάθε thread

Και τους τρεις υπολογισμούς των αξόνων περιστροφής

```
 array.x[index] = array.calculations[index*9+0]*array.x[index] + array.calculations[index*9+1]*array.y[index]+
array.calculations[index*9+2]*array.z[index];
 array.y[index] = array.calculations[index*9+3]*array.x[index] + array.calculations[index*9+4]*array.y[index]+
array.calculations[index*9+5]*array.z[index];
 array.z[index] = array.calculations[index*9+6]*array.x[index] + array.calculations[index*9+7]*array.y[index]+
```

Εικόνα 39: Ο κώδικας των τριών συναρτήσεων υπολογισμού περιστροφής

Τέλος η συνάρτηση με όνομα “rotations_routine” θα κληθεί με την εξής διαδικασία

```
rotations_routine(array,lines);
```

Εικόνα 40 : Η κλήση της συνάρτησης που είναι υπεύθυνη για τις περιστροφές

Η δομή του προγράμματος SIRENE είναι τέτοια που να μην μας επιτρέπει την εφαρμογή των υπολογισμών, για την περιστροφή, την ίδια στιγμή με εκείνες του πίνακας των 9 στοιχείων, μιας και τα αποτελέσματα μπορεί να χρησιμοποιηθούν σε δεύτερο χρόνο. Έτσι λοιπόν υποχρεωτικά διαχωρίζουμε το σύνολο των υπολογισμών σε δυο συναρτήσεις. Εκείνη που αφορά τους υπολογισμούς που αναλύθηκαν στα προηγούμενα κεφάλαια(“calculations_routine”) και σε εκείνη που περιλαμβάνει τις πράξεις περιστροφής “rotations_routine”. Για τις ανάγκες της παρούσας πτυχιακής και της προσομοίωσης της όλης διαδικασίας οι συναρτήσεις θα κληθούν η μια μετα την άλλη και τα δεδομένα θα αποθηκευτούν στη μνήμη για περαιτέρω πρόσβαση και επεξεργασία απο τα υπόλοιπα μέρη του προγράμματος. Εναλλακτικά θα μπορούσαμε

να χρησιμοποιήσουμε δυο τελείως αυτόνομα, μεταξύ τους, προγράμματα για τους 2 τύπους υπολογισμών.

4.5 Πως επηρεάζουν οι διαφορετικές υλοποιήσεις το συγκεκριμένο κομμάτι κώδικα

Όπως είπαμε και παραπάνω για τις ανάγκες της παρούσας πτυχιακής οι κλήσεις των δυο συναρτήσεων θα είναι ετερόκλητες, πράγμα που σημαίνει πως οι τακτικές και οι διαφορετικές υλοποιήσεις, όπως αυτές μετρήθηκαν σε προηγούμενα κεφάλαια, επηρεάζουν ακριβώς με τον ίδιο τρόπο και το συγκεκριμένο κομμάτι κώδικα.

Επειδή πρόκειται για μια σειρά υπολογισμών πολύ μικρότερου μεγέθους απο εκείνες του τελικού πίνακα των 9 στοιχείων, δεν υπάρχουν πολλοι τρόποι που να μπορούμε να χρησιμοποιήσουμε για τον τελικό υπολογισμό. Στον υπολογισμό του πίνακα των 9 στοιχείων, προσπαθήσαμε να διαχωρίσουμε τις τριγωνομετρικές συναρτήσεις απο τους εννέα, τελικούς, πολλαπλασιασμούς μεταξύ των τριγωνομετρικών αποτελεσμάτων με απώτερο σκοπό την βελτιστοποίηση του χρόνου εκτέλεσης. Στην περίπτωση αυτή, όμως, δεν υπάρχει κάποια τέτοια δυνατότητα, μιας και μιλάμε για μόλις τρεις πολλαπλασιασμούς. Μια πιθανή υλοποίηση θα ήταν η διάσπαση των πολλαπλασιασμών σε τρεις νέες και αυτόνομες, μεταξύ τους, συναρτήσεις οι οποίες θα εκτελούνταν διαδοχικά η μια μετα την άλλη σε επίπεδο κάρτας γραφικών. Μια υλοποίηση όμως η οποία θα μας έδινε πολυ χειρότερα αποτελέσματα, μιας και όπως δείξαμε και σε προηγούμενα κεφάλαια η χρήση πολλών συναρτήσων, για μικρού μεγέθους δεδομένα,χωρίς ιδιαίτερο λόγο, οδηγεί σε μείωση της απόδοσης καθώς το μεγαλύτερο κομμάτι του χρόνου εκτέλεσης το καταναλώνει η αντιγραφή και η δέσμευση της μνήμης στους πυρήνες της κάρτας γραφικών.

Έτσι λοιπόν η μόνη βελτιστοποίηση σε επίπεδο κώδικα που μπορέσαμε να πραγματοποιήσουμε ήταν εκείνη της συνένωσης(coaelscing) αλλάζοντας την μορφή της δομής δεδομένων στην οποία αποθηκεύουμε την πληροφορία που χρειαζόμαστε.

Παρακάτω βλέπουμε την παλιά και την καινούρια μορφή της δομής δεδομένων που αποθηκεύουμε τις πληροφορίες που χρειαζόμαστε

```
struct f_theta{

        double theta;
        double f;
        double calculations[9];
        double x;
        double y;
        double z;

};
```

Εικόνα 41 : Παλιά μορφή της δομής που αποθηκεύουμε τα δεδομένα μας

```
struct f_theta{

        double *theta;
        double *f;
        double *calculations;
        double *x;
        double *y;
        double *z;

};
```

Εικόνα 42 : Νέα μορφή της δομής που αποθηκεύουμε τα δεδομένα μας

καθώς και τον τρόπο που την προσπελαύνουμε μέσα στην συνάρτηση των υπολογισμών.

```
 array[index].x = array[index].calculations[0]*array[index].x + array[index].calculations[1]*array[index].y+
array[index].calculations[2]*array[index].z;
 array[index].y = array[index].calculations[3]*array[index].x + array[index].calculations[4]*array[index].y+
array[index].calculations[5]*array[index].z;
 array[index].z = array[index].calculations[6]*array[index].x + array[index].calculations[7]*array[index].y+
array[index].calculations[8]*array[index].z;
```

Εικόνα 43 : Παλιά μέθοδος προσπέλασης των δεδομένων

```
 array.x[index] = array.calculations[index*9+0]*array.x[index] + array.calculations[index*9+1]*array.y[index]+
array.calculations[index*9+2]*array.z[index];
 array.y[index] = array.calculations[index*9+3]*array.x[index] + array.calculations[index*9+4]*array.y[index]+
array.calculations[index*9+5]*array.z[index];
 array.z[index] = array.calculations[index*9+6]*array.x[index] + array.calculations[index*9+7]*array.y[index]+
array.calculations[index*9+8]*array.z[index];
```

Εικόνα 44 : Νέα μέθοδος προσπέλασης των δεδομένων

5. ΣΥΜΠΕΡΑΣΜΑΤΑ

Οι κάρτες γραφικών είναι ένα εργαλείο που μπορεί να προσφέρει τρομερή βοήθεια σε προγράμματα υπολογισμών. Ειδικεύονται στις πράξεις μεταξύ πινάκων, και σε τριγωνομετρικές συναρτήσεις, μεγάλης κλίμακας μιας και είναι υπεύθυνες για την απεικόνιση όλων των στοιχείων μιας οθόνης, μιας αναπαράστασης δηλαδή του “κόσμου” των υπολογιστών.

Είδαμε στην περίπτωση του προγράμματος Sirene πως ένα σημαντικό μέρος της εκτέλεσης καταναλώνεται στο κομμάτι του υπολογισμού της θέσης των ανιχνευτών, της απεικόνισης δηλαδή των στοιχείων ενός φωτοπολλαπλασιαστή καθώς και της θέσης αυτού μέσα στον ευρύτερο χώρο του τηλεσκοπίου νετρίνων. Η φύση του προβλήματος μας οδήγησε στην χρήση της αρχιτεκτονικής CUDA της NVidia, μιας και το συγκεκριμένο κομμάτι εκφράζει τις θέσεις που αναφέραμε ώς έναν πίνακα με στοιχεία ορισμένα τριγωνομετρικά αποτελέσματα που προκύπτουν απο δυο γωνίες.

Αυτό που μπορεί να μας προβληματίσει αρχικά είναι αν η τάξη μεγέθους της εισόδου, που καλούμαστε να υλοποιήσουμε, είναι τέτοια ώστε να δικαιολογεί την χρήση μιας κάρτας γραφικών για την επίλυση του προβλήματος. Η απάντηση σε αυτό δεν μπορεί να είναι ξεκάθαρη, μιας και αν δεχτούμε το πρόβλημα των υπολογισμών σα μια μεμονωμένη διαδικασία που θα αναπαραχθεί μια φορα τότε ξεκάθαρα η απάντηση μπορεί να είναι όχι. Είναι όμως αυτή η πραγματική φύση του προβλήματος;

Προς καλύτερη κατανόηση και υπενθύμιση σχετικά με τους χρόνους εκτέλεσης του προγράμματος τόσο σε CPU όσο και σε GPU, δείτε τα σχήματα 12 & 13.

Όπως βλέπουμε και απο τα παραπάνω διαγράμματα η χρήση της κεντρικής μονάδας επεξεργασίας(CPU) σαν βασικό εργαλείο εκτέλεσης των υπολογισμών, για την τάξη μεγέθους που μας αφορά, υπερτερεί της κάρτας γραφικών(GPU). Αυτό βέβαια συμβαίνει όταν προσπαθούμε να δούμε τα αποτελέσματα του αλγορίθμου για μια επανάληψη αυτού. Προχωρώντας ένα βήμα παραπέρα βλέπουμε πως το πρόγραμμα Sirene χρειάζεται να επαναλάβει τον αλγόριθμο αυτόν για παραπάνω απο μια φορές και ανάλογα την περίπτωση που εξετάζει το πρόγραμμα. Στην περίπτωση αυτή θα πρέπει να καλέσουμε την συνάρτηση που έχει πρόσβαση στην κάρτα γραφικών 50 με 100 φορές, περιμένοντας κάθε φορά να ολοκληρωθεί η προηγούμενη επανάληψη για να προχωρήσουμε στην επόμενη πράγμα που μπορεί να δημιουργήσει, προσθετικά, μεγάλη αναμονή. Εκτός της αναμονής, ενισχύοντας έτσι και το ερώτημά μας κατα πόσο η τάξη μεγέθους του προβλήματος είναι ικανοποιητικά μεγάλη για μια κάρτα γραφικών, βλέπουμε μέσα απο τα εργαλεία απεικόνισης του φόρτου της κάρτας πως μια εκτέλεση του αλγόριθμού μας καλύπτει μόλις το 4.9% της συνολικής διαθέσιμης μνήμης της κάρτας.

Η στρατηγική για την επίλυση των προβλημάτων της αναμονής αλλα και της κατανάλωσης μεγαλύτερου εύρους πόρων την λύνουμε μέσα απο τον συνδυασμό των βιβλιοθηκών που αφορούν τον παράλληλο προγραμματισμό σε επίπεδο CPU και GPU. Έτσι λοιπόν χρησιμοποιούμε την βιβλιοθήκη OpenMP με σκοπό να εκκινήσουμε παραπάνω απο 1 φορές, ταυτόχρονα, τον αλγόριθμό μας σε επίπεδο κάρτας γραφικών. Έτσι επιτυγχάνουμε την παράλληλη εκτέλεση των συναρτήσεών μας, μειώνοντας τον γενικότερο χρόνο της αναμονής(χωρίς να σημαίνει πως τον εξαλείφουμε) καθώς και την αύξηση του ποσοστού των χρησιμοποιούμενων πόρων. Ο συνδυασμός των βιβλιοθηκών της CUDA και της OpenMP μας οδηγεί, μέσα απο την λογική και την υλοποίηση που αναφέραμε παραπάνω, σε μια μείωση του χρόνου στο 1/3(Σχήμα 15). Η μείωση αυτή αφορά την σύγκριση μεταξύ της χρήσης αυστηρά και μόνο της βιβλιοθήκης CUDA, εκκινώντας 1 πυρήνα παράλληλων υπολογισμών κάθε φορά, και της υλοποίησης του συνδυασμού OpenMP και CUDA, χρησιμοποιώντας δηλαδή τα νήματα της CPU για να εκκινήσουμε παραπάνω απο έναν πυρήνες παράλληλων υπολογισμών.

Αν συγκεντρωθούμε στο επίπεδο της κάρτας γραφικών υπάρχουν καποια πολύ ουσιαστικά ευρήματα σχετικά με την δομή ενός προγράμματος και πως αυτή μπορεί να επηρεάσει την απόδοση του αλγορίθμου μας. Ένα απο τα βασικότερα είναι εκείνο της συνένωσης(coaelscing) το οποίο αναφέρεται στο πώς επηρεάζει η προσπέλαση των δεδομένων μας τον χρόνο απόκρισης της μνήμης RAM σε αυτά μας τα αιτήματα (Βλέπε κεφάλαιο 1.10.1). Μέσα απο την συνένωση επιτύχαμε επιτάχυνση της τάξεως του ~2X αλλάζοντας μόνο τον τρόπο που προσπελάυνουμε και αποθηκεύομε τα δεδομένα μας στις γενικότερες δομές δεδομένων που χρησιμοποιήσαμε.

Μια εξίσου σημαντική παρατήρηση είναι εκείνη που αφορά την μεταφορά των δεδομένων μέσα και έξω απο την κάρτα γραφικών. Η μεταφορά των δεδομένων είναι μια διεργασία ιδιαίτερα χρονοβόρα και συνήθως καλύπτει το μεγαλύτερο μέρος ενός προγράμματος, κάτι που εξαρτάται πάντα απο τον τρόπο που έχει δομηθεί το πρόγραμμα και τον σκοπό που αυτό εξυπηρετεί. Έτσι λοιπόν στην δικιά μας περίπτωση η μεταφορά των δεδομένων καλύπτει το ~91% του συνολικού χρόνου εκτέλεσης, ένα ποσοστό που ήταν ικανό για να μας αποτρέψει στην χρήση αλγορίθμων που είχαν πολλές μεταφορές εντος και εκτός της κάρτας γραφικών. Χρησιμοποιώντας την εντολή “nvprof” βλέπουμε πως κατανέμεται ο συνολικός χρόνος εκτέλεσης ενός προγράμματος στα επιμέρους κομμάτια αυτού, εκείνα δηλαδή της μεταφοράς των δεδομένων, της εκτέλεσης των συναρτήσεων κ.α

Όπως είπαμε και παραπάνω στην δικιά μας περίπτωση το ~91% του συνολικού χρόνου εκτέλεσης αφορά την μεταφορά των στοιχείων εντός και εκτός κάρτας γραφικών. Πρόκειται για ενα δεδομένο που κατέστησε αποτρεπτική την χρήση μεθόδων που χρησιμοποιούσαν πολλές φορές την συνάρτηση “cudaMemcpy” μιας και επιβραδύναμε τον συνολικό χρόνο εκτέλεσης χωρίς να έχουμε ουσιαστικό κέρδος στην ποσότητα των υπολογισμών που πραγματοποιούσαμε.

Βλέποντας τους χρόνους, όπως αυτοι περιγράφονται στο σχήμα 15, που χρειάζεται το πρόγραμμά μας για να εκτελεστεί μπορούμε να κάνουμε τις εξής παρατηρήσεις, σχετικά με την συνολική εκτέλεση του προγράμματος: Για μια ρεαλιστική εκτέλεση, του προγράμματος SIRENE,η οποία θέλει να υπολογίσει 1.000.000 διαφορετικά events τα

οποία περιέχουν 100 tracks το κάθε ένα απο αυτά θα έχουμε τους παρακάτω, συνολικούς, χρόνους που θα χρειαστούμε για την εκτέλεση. Έχουμε 117.000 ζεύγη γωνιών για κάθε ένα απο τα 100 tracks. Για να κάνουμε των υπολογισμούς και για τα 100 tracks, όπως βλέπουμε και στο σχήμα 15, χρειαζόμαστε 2.3 δευτερόλεπτα(όταν χρησιμοποιούμε τον συνδυασμό των βιβλιοθηκών CUDA μαζί με το OpenMP). Αν τα 2.3 δευτερόλεπτα τα πολλαπλασιάσουμε με τον συνολικό αριθμό απο events, 1.000.000,στο ρεαλιστικό σενάριο που παραθέτουμε, χρειαζόμαστε συνολικά ~27 ημέρες για την ολοκλήρωση του ρεαλιστικού σεναρίου μας απο ένα μηχάνημα CUDA, πράγμα που σημαίνει ότι για την εκτέλεση του παραπάνω σεναρίου θα χρειαζόμαστε 27 μηχανήματα για να μπορέσουμε να ολοκληρώσουμε την διαδικασία μέσα σε μόλις μια ημέρα.

ΠΙΝΑΚΑΣ ΟΡΟΛΟΓΙΑΣ

Ξενόγλωσσος όρος	**Ελληνικός Όρος**
Threads	Νήματα
Coalescing	Συνένωση
Cluster	Ομάδα
Events	Γεγονότα
Tracks	Διαδρομές
For loop	Επανάληψη ΓΙΑ
Double point variable	Μεταβλητές πραγματικών αριθμών
Standard Input	Είσοδος δεδομένων
Frameworks	Βιβλιοθήκες
Cosine	Συνημίτονο
Sine	Ημίτονο
Central Processing Unit	Κεντρική Μονάδα Επεξεργασίας
Graphics Processing Unit	Κάρτες Γραφικών
Host	Μονάδα που ενεργοποιεί την κ. γραφικών
Device	Η συσκευή της κάρτας γραφικών
Id	Αναγνωριστικός αριθμός
Array	Πίνακας
Calculations	Υπολογισμοί
Calculations routine	Διαδικασία υπολογισμών
Rotations Routine	Διαδικασία περιστροφών
Kernel	Πυρήνας

ΣΥΝΤΜΗΣΕΙΣ – ΑΡΚΤΙΚΟΛΕΞΑ – ΑΚΡΩΝΥΜΙΑ

CUDA	Compute Unified Device Architecture
CPU	Central Processing Unit
GPU	Graphical Processing Unit
Cos	Cosine
Sin	Cosine
KM3NeT	Cubic Kilometre Neutrino Telescope
ANTARES	Abyss environmental RESearch
ΕΚΠΑ	Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών
GB	Gigabyte
MB	Megabyte
PMT	Photo-Multiplier Tubes
MPI	Message Passing Interface
OpenMp	Open Multi Processing
RAM	Random Access Memory

ΑΝΑΦΟΡΕΣ

[1] Claudine Colnard Nikhef, National Institue of Nuclear Physics and High Energy Physics Amsterdam, SIRENE, a detector simulation program for Neutrino Telescopes.

[2] C.M.M. Colnard, Faculty of Science, Ultra-high energy neutrino simulations, 2009

[3] M. de Jong, The Jpp-Jtools Package, September 24,2012

[4] M. de Jong, The Jpp-JPhysics Package, September 24,2012

[5] Peter S. Pacheco, An introduction to Parallel Programming, 2011