Αριστοτέλειο Πανεπιστήμιο Θεσσαλονίκης
Πολυτεχνική Σχολή
Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Ηλεκτρονικής

ΑΥΤΟΜΑΤΗ ΠΑΡΑΓΩΓΗ ΚΩΔΙΚΑ ΓΙΑ ΕΠΙΤΑΧΥΝΣΗ ΥΠΟΛΟΓΙΣΜΩΝ ΣΤΕΝΣΙΛ ΣΕ ΚΑΡΤΕΣ ΓΡΑΦΙΚΩΝ ΜΕ JULIA

Διπλωματική Εργασία

Κωνσταντίνος Χατζηαντωνίου

Επιβλέπων: Νικόλαος Πιτσιάνης
Καθηγητής Α.Π.Θ.

2 Δεκεμβρίου 2020

Περίληψη

Η Julia είναι μια δυναμική γλώσσα, υψηλού επιπέδου που επικεντρώνεται σε αριθμητικούς υπολογισμούς. Η υψηλή της απόδοση, την κάνει ιδανικό υποψήφιο για επιστημονικές εφαρμογές που απαιτούν ταχύτητα, ενώ η δυναμικότητά της διευκολύνει την ανάλυση και προτοτυποποίηση ιδεών. Με την υποστήριξη για κάρτες γραφικών, μπαίνει πιο δυναμικά στον κόσμο των υπολογισμών υψηλής απόδοσης (High Performance Computing. Οι υπολογισμοί στένσιλ αποτελούν επαναληπτικές διαδικασίες όπου ανανεώνουν τα δεδομένα, συναρτήσει των γειτονικών τους στοιχείων. Εφαρμόζονται σε ένα μεγάλο αριθμό από πεδία, όπως μηχανική ρευστών, επεξεργασία εικόνας, εφαρμογές βιοϊατρικής και επίλυση μερικών διαφορικών εξισώσεων. Λόγω του πλήθους των εφαρμογών, αλλά και της ποικιλίας των επιστημονικών πεδίων τους, αποτελεί επιτακτική ανάγκη όχι μόνο η επιτάχυνση αλλά και η αυτόματη παραγωγή κώδικα στένσιλ. Στην παρούσα διπλωματική, η Julia λύνει και τα 2 προβλήματα. Με το υψηλό της επίπεδο και τη δυναμικότητά της δημιουργείται ένα framework, εύχρηστο για έρευνα κάθε κλάδου, ενώ με την υψηλή της απόδοση, επιτυγχάνονται σημαντικές επιταχύνσεις. Επιπλέον, υλοποιούνται δύο νέοι τρόποι επιτάχυνσης στένσιλ, μια διαφορετική ομαδοποίηση βημάτων χρόνου και υπολογισμός στένσιλ στο πεδίο του χρόνου. Τέλος, παραθέτονται αναλυτικά αποτελέσματα που επιβεβαιώνουν την αποτελεσματικότητα των μεθόδων, την υψηλή απόδοση της και την ποιότητα κώδικα που επιφέρει η χρήση της Julia.

Ευχαριστίες

Ευχαριστώ τον καθηγητή μου, κύριο Νίκο Πιτσιάνη για το ιδιαίτερα ενδιαφέρον θέμα , την πολύτιμη βοήθειά του και την καθοδήγησή του στην εκπόνηση της διπλωματικής εργασίας μου. Ευχαριστώ επίσης τους γονείς μου για την στήριξή τους κατα τη διάρκεια της διπλωματικής αλλά και των σπουδών μου. Ειδικά, ευχαριστώ τη μητέρα μου για τη φιλολογική επιμέλεια των κειμένων της εργασίας μου.

Περιεχόμενα

Κατάλογος Σχημάτων

Τμήματα Κώδικα

Κεφάλαιο 1

Εισαγωγή

1.1 Εισαγωγή στις κάρτες γραφικών

Η αξιόποιηση των παράλληλων και διανεμημένων συστημάτων είναι απαραίτητη στην σύγχρονη εποχή, τόσο για εμπορικά προϊόντα όσο και στην έρευνα. Πολλά προβλήματα που ενδιαφέρουν μηχανικούς και επιστήμονες απαιτούν υπερβολικά πολύ χρόνο σε κλασικούς επεξεργαστές. Η εμπορική διάθεση πολυπύρηνων επεξεργαστών και καρτών γραφικών βοηθάει στην αύξηση του ρυθμού της έρευνας, στην επίλυση προβλήματων που δεν λύνονται σε αποδεκτό χρόνο από μονοπύρηνους επεξεργαστές και στην παροχή υψηλής ταχύτητας προϊόντων λογισμικού από οργανισμούς.

Οι κάρτες γραφικών αποτελούν ένα μέρος του εξοπλισμού που χρησιμοποιούνται για υπολογισμούς υψηλής απόδοσης. Μεταλλάχθηκαν από υλικό πλήρως εξειδικεύμένο στη δημιουργία γραφικών, σε γενικού σκοπού υπολογιστικές μονάδες. Επιστήμονες και μηχανικοί έχουν δημιουργήσει νέους αλγορίθμους οι οποίοι βασίζονται στην αρχιτεκτονική των καρτών γραφικών και έχουν αναλύσει πως γίνεται η πιο αποδοτική χρήση των πόρων τους. Οι κάρτες γραφικών χρησιμοποιούνται πλέον σε διάφορες εφαραμογές, όπως υπολογισμοί στένσιλ, προβλήματα γραμμικής άλγεβρας, προσομοιώσεις δυναμικής ρευστών, νευρωνικά δίκτυα, μηχανική όραση και επεξεργασία σήματος.

Οι δύο πιο γνωστές πλατφόρμες για τον προγραμματισμό των καρτών γραφικών είναι η CUDA και η OpenCL. Η OpenCL είναι γενικότερου σκοπού πλατφόρμα και δεν εξειδικέυεται μόνο σε κάρτες γραφικών. Η CUDA έχει δημιουργηθεί από την NVIDIA για της δικιές της κάρτες γραφικών και έχει περισσότερες δυνατότητες από την OpenCL όσον αφόρα την αξιοποίηση συγκεκριμένων πόρων και θεωρείται πιο ώριμη [3].Οι βασικές γλώσσες που χρησιμοποιούνται για τον προγραμματισμό καρτών γραφικών είναι οι πλέον γνωστές C/C++, ενώ υπάρχει υποστήριξη και για Fortran

1.2 Εισαγωγή στην Julia

Η Julia είναι μια υψηλού επιπέδου γλώσσα που δημιουργήθηκε το 2012 και είναι εξειδικευμένη σε αριθμητικούς υπολογισμούς . Η Julia καταρρίπτει το στερεότυπο πως οι υψηλού επιπέδου γλώσσες είναι αργές και ότι η τελική υλοποίηση θα πρέπει να γραφτεί σε χαμηλότερου επιπέδου γλώσσα από οτι στο πρωτότυπο. Η ταχύτητα της σε εργασίες αριθμητικών υπολογισμών είναι ισάξια της C/C++

Το 2018 η Julia άρχισε να υποστηρίζει προγραμματισμό για κάρτες γραφικών της NVIDIA. Η υψηλού επιπέδου γλώσσα επιτρέπει νέες και δυναμικές τεχνικές στον προγραμματισμό καρτών γραφικών, αυξάνει την παραγωγικότητα των προγραμματιστών ενώ διατηρεί την απόδοση στα ίδια επίπεδα με αυτά που προσφέρουν τα εργαλεία της NVIDIA. Καθώς η υποστήριξη είναι πρόσφατη και σε πρώιμο στάδιο, πέρα από κάποιες βασικές μετρήσεις [;] που συγκρίνουν την ταχύτητα της Julia με C σε κάρτες γραφικών, δεν υπάρχουν μελέτες για το πώς επιτυγχάνεται αυτή η ταχύτητα σε προγράμματα ή πόσο παραγωγική είναι σε έρευνα.

1.3 Κίνητρο και στόχος της διπλωματικής εργασίας

Παρόλο που οι C/C++ φέρνουν τον προγραμματιστή πιο κοντά στο υλικό και στην εκμετάλλευση των πόρων του, η χαμηλού επιπέδου φύση τους αποτελεί εμπόδιο. Ως γλώσσες χαμηλού επιπέδου, χρειάζονται πολλές γραμμές κώδικα για να εκτελεστούν απλές εργασίες, είναι δύσκολο να γίνει αποσφαλμάτωση και συντήρηση και είναι περίπλοκο να γίνει ενσωμάτωση σε ή με άλλο πρόγραμμα. Οι χαμηλού επιπέδου γλώσσες δε βοηθούν στην παραγωγικότητα των προγραμματιστών και αυξανουν το κόστος και τον χρόνο δημιουργίας προγραμμάτων, καθώς απαιτούν περισσότερο κόπο και αυξάνουν το νοητικό φορτίο.

Ο στόχος της παρούσας διπλωματικής είναι να εδραιώσει τη θέση της Θυλια ως καλή ή και καλύτερη εναλλακτική για προγραμματισμό καρτών γραφικών σε CUDA. Για το σκοπό αυτό χρειάζεται να μελετηθεί η απόδοσή της, τόσο σε επίπεδο κώδικα στοχευμένου μόνο για κάρτες γραφικών, όσο και σε επίπεδο ολοκληρωμένης εφαρμογής. Επιπροσθέτως με τη μελέτη της ταχύτητας, χρειάζεται να γίνει ανάλυση της ποιότητας κώδικα που μπορεί να γράψει κάποιος προγραμματιστής και να γίνει σύγκριση με την ανάλογη ποιότητα σε C.

1.4 Οργάνωση Κειμένου

Στη συνέχεια της διπλωματικής εργασίας, που ξεκινάει με το Κεφάλαιο 2, παρουσιάζονται οι βασικές έννοιες της CUDA και της Julia που είναι απαραίτητες για την ανάλυση και τα συμπεράσματα που ακολουθούν στα επόμενα κεφάλαια. Στο Κεφάλαιο 3 γίνονται δύο μελέτες σε αυτοτελή προβλήματα για τη μελέτη της Julia σε επίπεδο κώδικα μόνο για κάρτα γραφικών. Στο Κεφάλαιο 4 παρουσιάζονται οι υπολογισμοί στένσιλ σε κάρτες γραφικών και προηγούμενες έρευνες που χρησιμοποιούνται σαν βάση για το επόμενο κεφάλαιο. Στο Κεφάλαιο 5 εξηγείται η υλοποίηση μιας πλατφόρμας για αυτόματη παραγωγή κώδικα στένσιλ για κάρτες κάρτες γραφικών. Στο κεφάλαιο 6 παρουσιάζονται τα αποτελέσματα, τόσο για ταχύτητα όσο και για ποιότητα κώδικα ενώ στο Κεφάλαιο 7 γίνεται εξαγωγή συμπερασμάτων και συζήτηση για προβλήματα που αντιμετωπίστηκαν και για περαιτέρω έρευνα.

Κεφάλαιο 2

Βασικές Έννοιες

2.1 CUDA και κάρτες γραφικών NVIDIA

Η CUDA (Compute Unified Device Architecture) αποτελεί μία πλατφόρμα παράλληλης επεξεργασίας, που επιτρεπεί τον προγραμματισμό καρτών γραφικών που υποστηρίζουν αυτό το μοντέλο χωρίς να χρειάζεται γνώση αλγορίθμων γραφικής. Σε αντίθεση με άλλα προγραμματιστικά περιβάλλοντα, όπως OpenGL, DirectX, [4] δεν απαιτεί από τον προγραμματιστή την εκμάθηση καινούριας διεπαφής ή βιβλιοθήκης, αφού έχει σχεδιαστεί να δουλεύει με υπάρχουσες γλώσσες προγραμματισμού (C/C++.

2.1.1 Μοντέλο Προγραμματισμού

Το μοντέλο επεξεργασίας που ακολουθείται είναι το **SIMD** Single Instruction Multiple Data. Δηλαδή, όλα τα νήματα εκτελούν τις ίδιες εντολές πάνω σε διαφορετικά δεδομένα.

Η CUDA επεκτείνει τις γλώσσες προγραμματισμού που υποστηρίζει, δίνοντας τη δυνατότητα στον προγραμματιστή να γράψει συναρτήσεις στην ίδια τη γλώσσα, η οποίες θα εκτελεστούν N φορές παράλληλα σε N νήματα της CUDA. Κάθε νήμα που εκτελεί την συνάρτηση-πυρήνα έχει μια ενσωματωμένη μεταβλητή που περιέχει την ταυτότητα του νήματος. Η μεταβλητή αυτή είναι προσβάσιμη μέσα από τον κώδικα της συνάρτησης-πυρήνα.

Η ταυτότητα των νημάτων μπορεί να είναι μονοδιάστατη, δισδιάστατη ή τρισδιάστατη ανάλογα με τη μορφή των δεδομένων. Επιπλέον, τα νήματα οργανώνονται σε μπλοκ και τα μπλοκ σε πλέγματα. Η ταυτότητα των μπλοκ και των πλεγμάτων βρίσκεται επίσης σε ενσωματωμένες μεταβλητές και ακολουθεί την ίδια δεικτιοδότηση σχετικά με τις διαστάσεις. Ο προγραμματιστής ορίζει τον αριθμό τους πριν την εκτέλεση της συνάρτησης-πυρήνα. Ο συνολικός αριθμός των νημάτων σε μια εκτέλεση συνάρτησης πυρήνα υπολογίζεται μέσω της σχέσης:

$$\#\ \text{νημάτων ανά μπλοκ} \times \#\ \text{μπλοκ ανά πλέγμα} \times \#\ \text{πλεγμάτων}$$

Συνήθως ο συνολικός αριθμός των νημάτων εξαρτάται από το μέγεθος των δεδομένων και μπορεί να ξεπερνάει κατά πολύ τον αριθμό νημάτων που μπορεί να εκτελέσει πραγματικά παράλληλα η κάρτα γραφικών.

2.1.2 Ιεραρχία Μνήμης

Τα νήματα CUDA έχουν πρόσβαση σε πολλούς χώρους μνήμης. Κάθε νήμα έχει τη δική του ιδιωτική τοπική μνήμη, που περιλαμβάνει καταχωρητές και ιδιωτικό χώρο στην καθολική μνήμη. Κάθε μπλοκ έχει τη δική του διαμοιραζόμενη μνήμη, στην οποία έχουν πρόσβαση όλα τα νήματα που ανήκουν στο μπλοκ. Τέλος, όλα τα νήματα, ανεξάρτητα από το μπλοκ και το πλέγμα που ανήκουν, έχουν πρόσβαση στην καθολική μνήμη.

Η τοπική και η διαμοιραζόμενη μνήμη μένουν ¨ζωντανές¨ μόνο κατά την εκτέλεση των νημάτων ή των μπλοκ που ανήκουν. Η καθολική μνήμη, διατηρεί τα δεδομένα, ακόμα και σε εκτελέσεις διαφορετικών συναρτήσεων πυρήνων.

Κάθε τύπος μνήμης, έχοντας διαφορετικά πλεονεκτήματα και περιορισμούς, προορίζεται για διαφορετική χρήση. Ακόμα, υπάρχουν 3 επίπεδα κρυφής μνήμης, που όμως δεν προγραμματίζονται απευθείας.

2.1.3 Αρχιτεκτονική Υλικού

Η αρχιτεκτονική των καρτών γραφικών της NVIDIA βασίζεται σε ένα πλέγμα από ειδικούς πολυ-νηματικους επεξεργαστές, που ονομάζονται πολυεπεξεργαστές συνεχόμενης ροής (Streaming Multiprocessors. Όταν ένα πρόγραμμα CUDA καλείται από τον επεξεργαστή του υπολογιστή, τα μπλοκ νημάτων αριθμούνται και διανέμονται στους πολυεπεξεργαστές ανάλογα με τη διαθέσιμη χωρητικότητα. Τα νήματα ενός μπλοκ εκτελούνται ταυτόχρονα σε έναν πολυ-επεξεργαστή, ενώ διαφορετικά μπλοκ του πλέγματος εκτελούνται σε διαφορετικούς πολυπεξεργαστές. Στην πραγματικότητα, ο αριθμός των νημάτων και των μπλοκ που μπορούν να εκτελεστούν παράλληλα είναι περιορισμένος. Όταν ένα μπλοκ ολοκληρώσει την εκτέλεσή του, ένα νέο ξεκινά την εκτέλεση στη θέση του.

Σε χαμηλότερο επίπεδο, ο πολυπεξεργαστής δημιουργεί, διαχειρίζεται και εκτελεί μια ομάδα από 32 νήματα, που ονομάζονται σμήνος (warp). Όταν εκτελείται ένα μπλοκ, τα νηματά του διαχωρίζονται σε σμήνη και κάποια είναι ενεργά (δηλαδή εκτελούνται) ένω άλλα ανενεργά περιμένοντας να βρεθούν διαθέσιμοι πόροι. Σε ένα σμήνος που είναι ενεργό, τα νηματά του εκτελούν πάντα κοινές εντολές, με τη μέγιστη αποδοτικότητα όταν δεν υπάρχουν διαφοροποιήσεις σε βρόγχος επανάληψης ή συνθήκες ελέγχου.

2.1.4 Βελτίωση Απόδοσης

Οι βασικές τεχνικές βελτιστοποίησης για την αύξηση της απόδοσης περιγράφονται από 3 κανόνες.

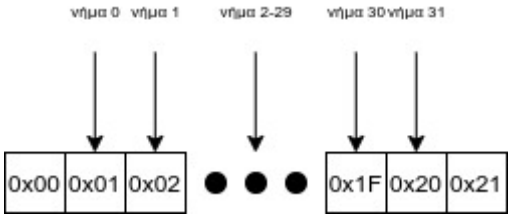
- Μεγιστοποίηση του αριθμού νημάτων που εκτελούνται πραγματικά παράλληλα.
- Βελτιστοποίηση της χρήσης μνήμης για μέγιστη ρυθμαπόδοση της.
- Βελτιστοποίηση σε επίπεδο εντολών για μέγιστη ρυθμαπόδοσή τους.

Ο πρώτος κανόνας στοχεύει στη μέγιστη εκμετάλλευση των πολυεπεξεργαστών. Κάθε πολυπεξεργαστής μπορεί να εκτελέσει περιορισμένο αριθμό μπλοκ παράλληλα. Επομένως, αν κάθε μπλοκ έχει λίγα νήματα, δεν αξιοποιείται πλήρως η επεξεργαστική

ισχύς. Το ίδιο ισχύει και για μικρό αριθμό μπλοκ που διαμοιράζονται στους πολυπεξεργαστές. Παρόμοια κατάσταση ανεκμετάλλευτης επεξεργαστικής ισχύς εμφανίζεται όταν ζητούνται υπερβολικά πολλοί πόροι (κυρίως σε επίπεδο μνήμης). Για παράδειγμα, αν ένα μπλοκ ζητήσει μεγάλο τμήμα της περιορισμένης διαμοιραζόμενης μνήμης που έχει ένας πολυεπεξεργαστής, δε γίνεται να εκτελεστεί παράλληλα ο μέγιστος δυνατός αριθμός μπλοκ.

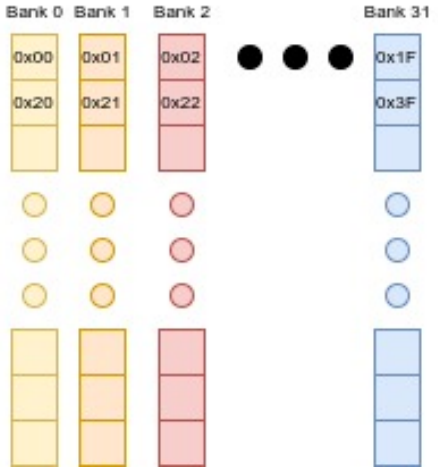
Όπως αναφέρθηκε προηγουμένως, κάθε μνήμη προορίζεται για διαφορετική χρήση και έχει τους δικούς της περιορισμούς.

Η καθολική μνήμη είναι η πιο αργή από τις μνήμες και πολλές φορές αποτελεί περιοριστικό παράγοντα στην επιτάχυνση.Η πρόσβαση στην μνήμη αυτή από τα νήματα πρέπει να έχει συγκεκριμένη μορφή για να αξιοποιηθεί πλήρως η ταχύτητα μεταφοράς δεδομένων που προσφέρει η κάρτα. Ένα σμήνος νημάτων πρέπει να κάνει ομαδοποιημένη (ϑαλεσεδ Αςςεσς) δηλαδή να προσπελαύνει συνεχόμενα τμήματα μνήμης. Έτσι τα δεδομένα διαβάζονται ή γράφονται παράλληλα, σε αντίθεση με προσπέλαση μη συνεχόμενων δεδομένων όπου γίνεται σειριακά στα διαφορετικά νήματα.



Σχήμα 2.1: Ομαδοποιημένη πρόσβαση στην καθολική μνήμη

Η διαμοιραζόμενη μνήμη παίζει τον ρόλο κρυφής μνήμης, ελεγχόμενη όμως πλήρως από τον προγραμματιστή. Σε αυτήν αποθηκεύονται τα δεδομένα της καθολικής μνήμης, τα οποία μπορεί να χρησιμοποιηθούν σε περισσότερα από ένα νήματα. Ομοίως, σε αυτήν αποθηκεύονται τα αποτελέσματα, ώστε να εγγραφούν με ομαδοποιημένη προσπέλαση πίσω στην καθολική μνήμη. Για να επιτευχθεί η μέγιστη ταχύτητα χρειάζεται να μην υπάρχουν bank conflicts, δηλαδή να μη ζητούν πρόσβαση περισσότερα από ένα νήματα στην ίδια τράπεζα μνήμης. Ο τρόπος που χωρίζεται η μεριζόμενη μνήμη σε τράπεζες φαίνεται στο παρακάτω σχήμα.



Σχήμα 2.2: Οπτικοποίηση τραπεζών μνήμης στην μεριζόμενη μνήμη

Τέλος, οι καταχωρητές χρησιμοποιούνται για αποθήκευση δεδομένων τοπικά σε κάθε νήμα και για την εκτέλεση εντολών. Κάθε νήμα έχει τον δικό του χώρο καταχωρητών. Ο προγραμματιστής δε μπορεί να ορίσει ακριβώς τον αριθμό καταχωρητών που χρησιμοποιεί κάθε νήμα, αλλά μπορεί να μειώσει τη χρήση τους αποθηκεύοντας δεδομένα στην μεριζόμενη μνήμη.

2.1.5 Επιπλέον χαρακτηριστικά

Συγχρονισμός Το σμήνος νημάτων είναι η μόνη οντότητα που έχει έμφυτη τη συγχρονισμένη εκτέλεση εντολών. Επειδή γενικά τα νήματα εκτελούνται ταυτόχρονα αλλά όχι πάντα παράλληλα απαιτούνται ειδικές εντολές για τον συγχρονισμό τους, όταν υπάρχει πρόσβαση σε κοινούς πόρους, όπως η διαμοιραζόμενη μνήμη. Μόνο τα νήματα το ίδιου μπλοκ μπορούν να συγχρονιστούν, χρησιμοποιώντας την εντολή __ΣΨΝΣΤΗΡΕΑΔΣ(). Η εντολή αυτή χρησιμοποιείται κυρίως για να είμαστε σίγουροι ότι έχουν διαβαστεί όλα τα κοινά δεδομένα στη διαμοιραζόμενη μνήμη, πριν προσπαθήσει κάποιο νήμα να τα χρησιμοποιήσει.

Ο επόμενος δυνατός συγχρονισμός, είναι σε επίπεδο μπλοκ και επιτυγχάνεται με τη χρησή διαφορετικών κλήσεων συναρτήσεων-πυρήνων και την εντολή __ΣΥΔΑΔΕΙΣΕΣΨΝΣΗΡΟΝΙΖΕ() από τον επεξεργαστή του υπολογιστή.

Ομογενοποιημένη Διευθυνσηοδότηση Μνήμης Ενα πολύ σημαντικό χαρακτηριστικό, που μπορεί να εφαρμοστεί σε 64bit υπολογιστές, είναι η χρήση ενός ιδεατού χώρου διευθύνσεων μνήμης, τόσο για τις κάρτες γραφικών, όσο και για τον υπολογιστή. Ο προγραμματιστής ορίζει μόνο το μέγεθος και τον τύπο της μνήμης που θέλει, και η πλατφόρμα CUDA, ανάλογα με τη χρήση της μνήμης, καταλαβαίνει αν βρίσκεται στην κάρτα, στον επεξεργαστή ή και στα δύο και κάνει αυτόματα τις μεταφορές δεδομένων. Ακόμα πιο σημαντικό, είναι ότι διευκολύνει την διαχείρηση μνήμης σε σύστημα με πολλές κάρτες γραφικών, αφού με την ίδια εντολή μεταφέρει δεδομένα από επεξεργαστή σε κάρτα, το αντίστροφο, και από κάρτα σε άλλη κάρτα.

2.2 Julia

Η Julia είναι μια υψηλού επιπέδου, υψηλής απόδοσης και δυναμική γλώσσα προγραμματισμού που στοχεύει στη λύση του ¨προβλήματος των δύο γλωσσών¨ που διέπει την έρευνα και τους τεχνικούς υπολογισμούς. Οι προγραμματιστές προτιμούν υψηλού επιπέδου γλώσσες για ανάλυση δεδομένων και ανάπτυξη αλγορίθμων καθώς είναι βολικές και αυξάνουν την παραγωγικότητα. Αντίθετα, για αποδοτικότητα έντονες υπολογιστικά διεργασίες προτιμούνται χαμηλού επιπέδου, όπως C/C++ και Fortran.

2.2.1 Μεταπρογραμματισμός και Ενδοσκόπηση

Ο μετα-προγραμματισμός είναι το σημανικότερο κληροδότημα της LISP [5] στην Julia. Η Julia αναπαριστά τον κώδικα της σαν δομή δεδομένων της ίδιας της γλώσσας, η οποία μπορεί να δημιουργηθεί και να τροποποιηθεί. Έτσι, είναι δυνατόν να γίνει περίπλοκη παράγωγη κώδικα χωρίς επιπλέον βήματα μεταγλώττισης. Ακόμα, οι μακρο-εντολές λειτουργούν σε επίπεδο συντακτικού δέντρου, όπως και στην LISP, κάνοντας πιο εύκολη την ενσωμάτωση βελτιστοποιήσεων και αυξάνοντας την εκφραστικότητα. Σε αντίθεση, οι

μακρο-εντολές στην C έχουν περιορισμένες δυνατότητες καθώς λειτουργούν σε επίπεδο αλφαριθμητικών, πριν γίνει συντακτική ανάλυση του κώδικα. Από τα παραπάνω χαρακτηριστικά, εξάγεται ότι η Julia υποστηρίζει και την ενδοσκόπηση κώδικα, που είναι πολύ χρήσιμη στρατηγική για τον προγραμματισμό και επιτρέπει την μετατροπή του εκτελέσιμου κώδικα κατά την εκτέλεση του προγράμματος.

2.2.2 LLVM και JIT

Ένα από τα χαρακτηριστικά ¨δυναμισμού¨ που ενσωμάτωσαν οι δημιουργοί της Julia σε αυτή, είναι η δυνατότητα ο χρήστης να τρέξει κώδικα στο στάδιο το φορτώματος και στο στάδιο της υλοποίησης, απαλλάσσοντας τον από το χειροκίνητο ¨χτίσιμο¨ του κώδικα που αποσπά την προσοχή. Για να πετύχει αυτόν τον στόχο, η Julia χρησιμοποιεί το μοντέλο μεταγλώττισης JIT και την πλατφόρμα LLVM.

Στο δυναμικό μοντέλο μεταγλώττισης JIT (Just in Time υπάρχει μια συνεχής διαδικασία μεταγλώττισης του προγράμματος καθώς εκτελείται. Με κάθε εκτέλεση γίνεται ανίχνευση σημείων του κώδικα που μπορούν να βελτιστοποιηθούν και γίνεται επαναμεταγλώττιση.

Το LLVM είναι μια πλατφόρμα μεταγλώττισης, που μπορεί να σε ενσωματωθεί σε οποιαδήποτε γλώσσα και δίνει πληροφορίες για τη μεταγλώττιση του προγράμματος καθόλη τη διάρκεια ζωής του (εκτέλεση, αναμονή, μεταγλώττιση).

Με αυτόν τον τρόπο, ο προγραμματιστής ασχολείται μόνο με τη σύνταξη κώδικα, μπορεί πολύ εύκολα να υλοποιήσει εφαρμογή που παράγει αυτόματα κώδικα και να εφαρμοστούν βελτιστοποιήσεις από την ίδια τη γλώσσα σε μη-καλά ορισμένους τύπους δεδομένων.

2.2.3 CUDA.jl

Η βιβλιοθήκη CUDA.jl παρέχει δύο πολύ βασικές λειτουργίες για τον προγραμματισμό καρτών γραφικών. Αρχικά, όλες οι εντολές της CUDA σε C, που χρησιμοποιούνται για τον έλεγχο του υλικού των καρτών γραφικών, έχουν καλυφθεί από κώδικα Julia για να είναι εύκολα προσβάσιμες από τον προγραμματιστή. Η δεύτερη λειτουργία, είναι η μετατροπή κώδικα Julia σε κώδικα μηχανής κάρτας γραφικών. Αυτό επιτυγχάνεται ενσωματώνοντας τον μεταγλωττιστή της NVIDIA (NVTX) και χρησιμοποιώντας την πλατφόρμα LLVM, ρυθμισμένη διαφορετικά. Έτσι ο προγραμματιστής μπορεί αν δημιουργήσει δυναμικές συναρτήσεις, οι οποίες για παράδειγμα μπορεί να τροποποιούν τον κώδικά τους και να τον εκτελούν απρόσκοπτα σε κάρτες γραφικών.

Η πλατφόρμα CUDA ενισχύεται με τη βιβλιοθήκη CuArrays, που είναι ενσωματωμένη στην βιβλιοθήκη CUDA.jl. Έχει δημιουργηθεί ένας νέος τύπος πίνακα, που καλύπτει πολλές από της λειτουργίες της CUDA για διαχείριση μνήμης των καρτών γραφικών. Με τον νέο τύπο πίνακα, φαίνεται σαν να υπάρχει έμφυτη υποστήριξη από τη γλώσσα για διαχείριση μνήμης σε κάρτες γραφικών, καθώς ο τρόπος χρήσης τους μοιάζει με αυτόν των κανονικών πινάκων.

Κεφάλαιο 3

Μελέτες Περιπτώσεων

3.1 Εισαγωγή

Το πρώτο βήμα για να εδραιωθεί η Julia σαν καλή ή και καλύτερη επιλογή για προγραμματισμό σε CUDA, είναι να συγκριθεί με τη C ως προς την ταχύτητα και την ποιότητα κώδικα. Παρόλο που ήδη υπάρχουν μετρήσεις που πραγματοποιούν αυτήν τη σύγκριση [ροντινια], το να γίνουν υλοποιήσεις από το μηδέν βοηθάει στην απόκτηση βαθύτερης γνώσης, εντοπισμό λεπτομερειών και πιο εύκολη αλλαγή του κώδικα για εκτενέτερη έρευνα.

3.2 Μελέτη 1 - Κοντινότερος Γείτονας σε πλέγμα

3.2.1 Περιγραφή Προβλήματος

Η εύρεση του κοντινότερου γείτονα σε πλέγμα (Grid KNN) είναι μια παραλλαγή του κανονικού αλγορίθμου εύρεσης κοντινότερων γειτόνων για τρισδιάστατα σημεία. Τα δεδομένα, τόσο εισόδου όσο και ερωτημάτων (queries) ταξινομούνται σε κύβους ίσου μεγέθους δημιουργώντας ένα πλέγμα. Η αναζήτηση του κοντινότερου γείτονα για ένα ερώτημα γίνεται πρώτα στον κύβο που ανήκει και έπειτα στους γειτονικούς κύβους.

3.2.2 Περιγραφή Υλοποίησης

Υλοποιήθηκε μόνο η περίπτωση που αναζητάμε τον έναν κοντινότερο γείτονα. Υπάρχουν δύο προσεγγίσεις: είτε να γίνει η αναζήτηση σε όλους τους γειτονικούς κύβους, χωρίς κάποιον έλεγχο (απλή εκδοχή), είτε να ελέγχεται αν κάποιος γείτονας είναι πιο κοντά από τις πλευρές των κύβων για να αποφευχθούν οι περιττές αναζητήσεις (εκδοχή με έλεγχο).

Για τις γλώσσες, οι υλοποιήσεις είναι πανομοιότυπες όσον αφορά την αλγοριθμική προσέγγιση, με τη διαφορά να βρίσκεται στο συντακτικό. Η Julia, στην απλή εκδοχή της, έχει τρεις διαφορετικές συντακτικές προσεγγίσεις

- "jl_simple Ίδια με την υλοποίηση σε C. Οι τύποι των μεταβλητών δηλώνονται και η μόνη διαφορά είναι ότι χρησιμοποιείται τρισδιάστατη διευθυνσιοδότηση στους πίνακες.

- jl_view Η διαφόρα με τα παραπάνω είναι ότι χρησιμοποιείται η μακροεντολή @view για να μη χρησιμοποιείται μετατόπιση διεύθυνσης σε όλο τον κώδικα.
- jl_no_types Ίδια με την παραπάνω απλά χωρίς να δηλώνονται οι τύποι των μεταβλητών.

Για την εκδοχή με έλεγχο ακολουθείται ο ίδιος τρόπος ονομασίας.

3.2.3 Ποιότητα Κώδικα

Δήλωση τύπων μεταβλητών Η Julia δεν απαιτεί να δηλώνουμε τον τύπο των μεταβλητών. Αυτό έχει ως αποτέλεσμα λιγότερο ¨γεμάτο¨ κώδικα.

Τμήμα Κώδικα 3.1: C παράδειγμα με δήλωση τύπων

```
int tid = threadIdx.x + threadIdx.y*blockDim.x;
int stride = blockDim.x*blockDim.y;
int start_points = intgr_points_per_block[p_bid];
int start_queries = intgr_queries_per_block[q_bid];
```

Τμήμα Κώδικα 3.2: Julia παράδειγμα χωρίς δήλωση τύπων

```
tid = threadIdx().x + (threadIdx().y-1)*blockDim().x
stride = (blockDim().x)*(blockDim().y)
startPoints = IntPointsperblock[p_bid]
startQueries = IntQueriesperblock[q_bid]
```

Θα περιμέναμε ότι είναι πολύ σημαντικό για προγραμματισμό σε κάρτες γραφικών να δηλώνονται οι τύποι. 64-bit αριθμητικές εντολές μπορεί να είναι έως και 8 φορές πιο αργές από τις αντίστοιχες 32-bit.

Προσπέλαση στοιχείων πίνακα Η Julia μπορεί να χρησιμοποιήσει πολυδιάστατη διευθυνσιοδότηση σε πίνακες της κάρτας γραφικών, τόσο στους καθολικούς global όσο και στους στατικούς και δυναμικούς μεριζόμενους (shared). Αντίθετα η C χρησιμοποιεί μονοδιάστατη γραμμική διευθυνσιοδότηση σε όλους τους πίνακες, εκτός από τους στατικούς μεριζόμενους πίνακες, όπου μπορεί και πολυδιάστατη.

Χρησιμοποιώντας πολυδιάστατη διευθυνσιοδότηση, ο κώδικας είναι πιο ευανάγνωστος και λιγότερο ευπαθής σε λογικά λάθη.

Τμήμα Κώδικα 3.3: C παράδειγμα προσπέλασης πίνακα

```
int q_index = q + tid + start_queries;
if(tid + q < total_queries){
    for(int d = 0; d < dimensions; d++){
        sh_queries[tid + d*stride]
                = queries[q_index + d*num_of_queries];
    }
    distance = distsances[q_index];
    neighbour = neighbours[q_index];
}
```

καπρτιον

Τμήμα Κώδικα 3.4: Julia παράδειγμα προσπέλασης πίνακα

```
qIndex = startQueries + q + tid
if tid + q <= totalQueries
    for d = 1:dimensions
       @inbounds SharedQueries[d,tid] = Queries[qIndex, d]
    end
    @inbounds dist = Distances[qIndex]
    @inbounds nb = Neighbours[qIndex]
end
```

Δείκτης σε γραμμή ή στήλη Η πολυδιάστατη διευθυνσιοδότηση στην Julia συμπληρώνεται από την μακροεντολή @view, που επιτρέπει τον ορισμό ενός τμήματος πίνακα. Το ανάλογο στην C είναι η χρήση δείκτη (pointer) για τον ορισμό της έναρξης μιας γραμμής ή μιας στήλης ενός πίνακα. Οι δείκτες περιορίζονται στον ορισμό μονοδιάστατου τμήματος ενός πίνακα σε αντίθεση με τη μακροεντολή που μπορεί να ορίσει οποιονδήποτε υποπίνακα.

Τμήμα Κώδικα 3.5: Julia παράδειγμα με μακροεντολή @view. Μόνο ο αριθμός του νήματος (thread id) και η μεταβλητή της επανάληψης χρησιμοποιούνται για την προσπέλαση πίνακα.

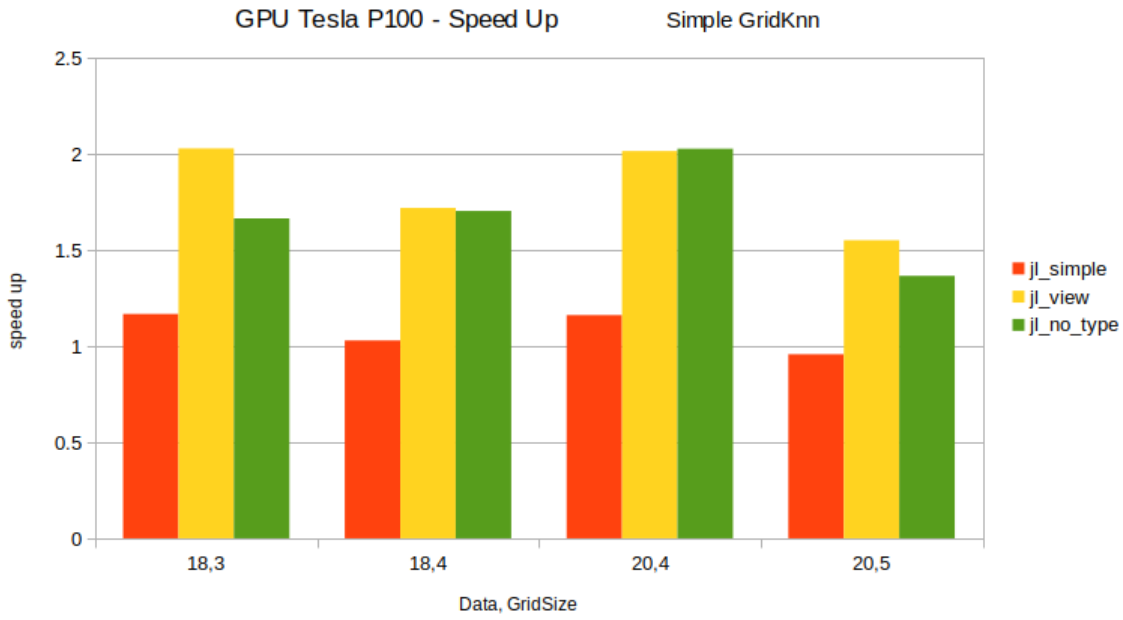
```
# Defining the views
@inbounds Queries = @view devQueries[
        (startQueries+1):(startQueries+totalQueries), :]
@inbounds query = @view SharedQueries[:, tid]
@inbounds Distances = @view devDistances[
        (startQueries+1):(startQueries+totalQueries)]
@inbounds Neighbours = @view devNeighbours[
        (startQueries+1):(startQueries+totalQueries)]

...
# Reading Queries from Global Memory
 if tid + q <= totalQueries
    for d = 1:dimensions
        @inbounds  query[d] = Queries[tid+q, d]
    end
    @inbounds dist = Distances[tid+q]
    @inbounds nb = Neighbours[tid+q]
end
```

Συνηθισμένη πρακτική για κώδικα σε κάρτες γραφικών, είναι ο διαχωρισμός των δεδομένων στα μπλοκ της κάρτας. Επομένως, αντί να χρησιμοποιούμε γραμμική διευθυνσιοδότηση και απόκλιση σε κάθε προσπέλαση πίνακα, μπορούμε να ορίσουμε τον υποπίνακα του κάθε μπλοκ με την μακροεντολή @view. Έτσι, γίνεται πιο κατανοητή η λογική του προγράμματος και δε χρειάζεται να σκεφτόμαστε κάθε φορά την περίπλοκη γραμμική διευθυνσιοδότηση.

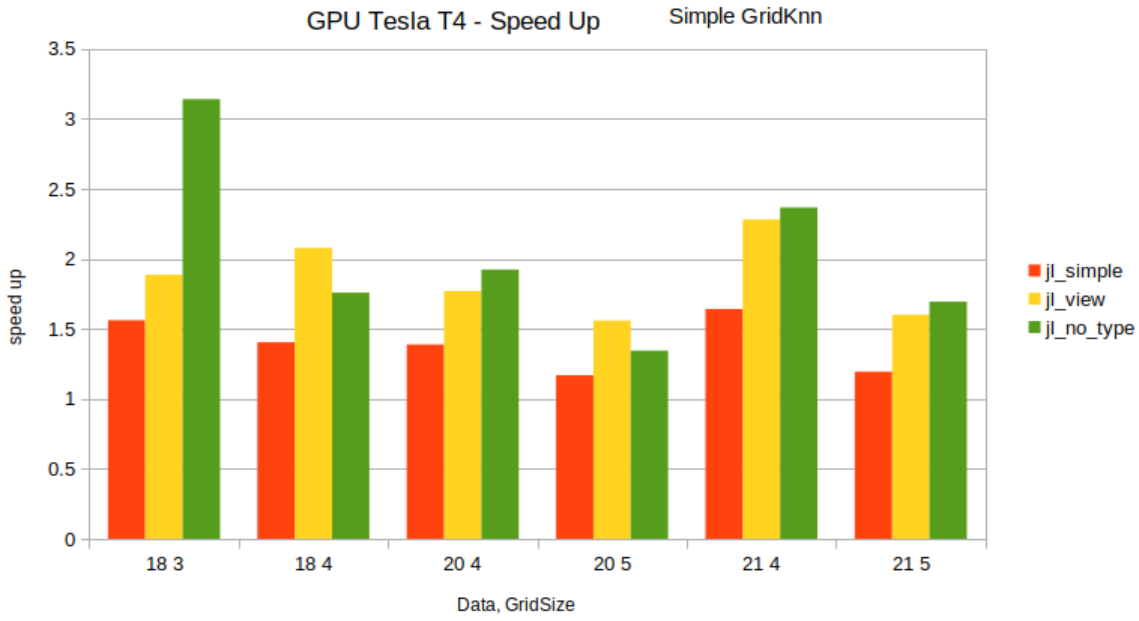
3.2.4 Επίδοση και Μετρικές

Ο χρόνος εκτέλεσης κάθε υλοποίησης μετρήθηκε στις κάρτες γραφικών NVIDIA Tesla P100 και T4 για 2^{18} τρισδιάστατα σημεία με 2^3 και 2^4 μέγεθος πλέγματος, και για 2^20 τρισδιάστατα σημεία με 2^4 και 2^5 μέγεθος πλέγματος

Η επιτάχυνση στα παρακάτω γραφήματα ορίζεται ως $\frac{\text{χρόνος της C}}{\text{χρόνος της Julia}}$ για δεδομένο μέγεθος προβλήματος.



Σχήμα 3.1: Σύγκριση Julia με C για simple GridKnn

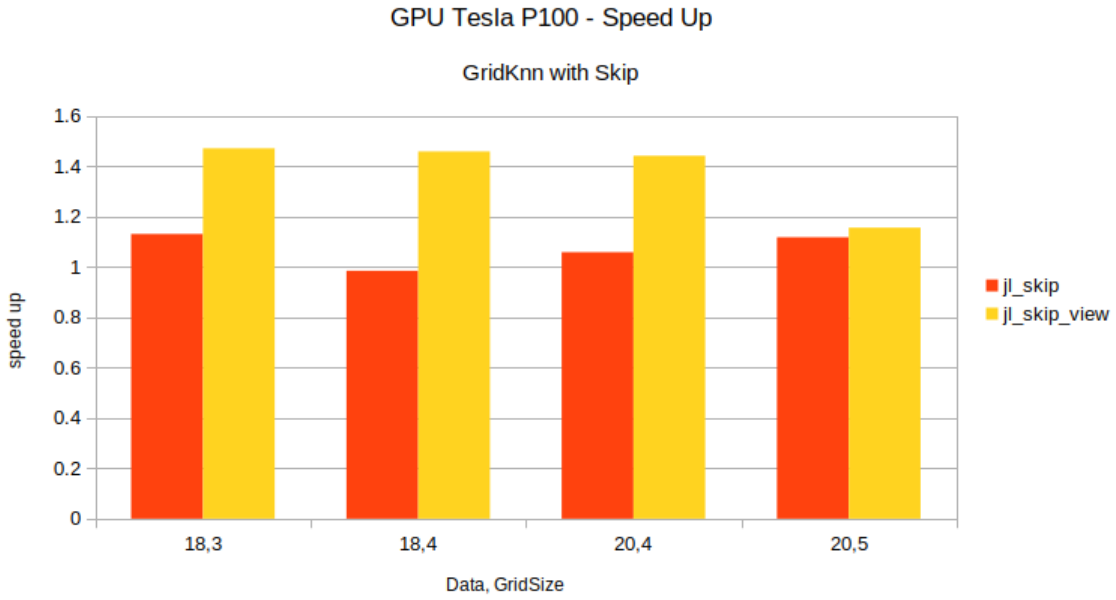


Σχήμα 3.2: Σύγκριση Julia με C για simple GridKnn

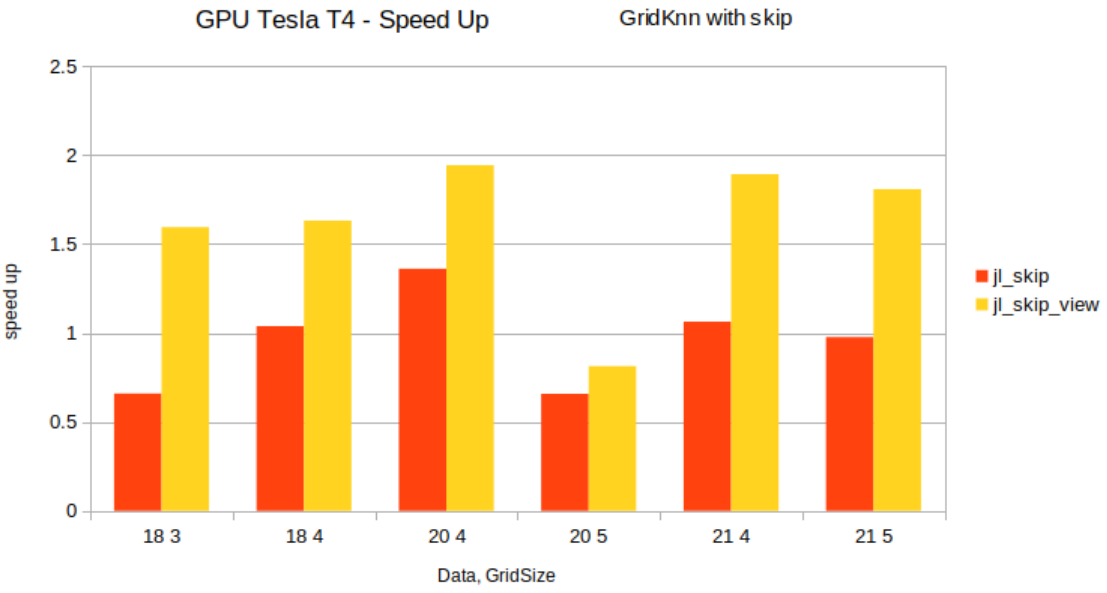
Σε πρώτη ματιά, το πρόγραμμα σε C είναι πάντα πιο αργό από τουλάχιστον ένα πρόγραμμα σε Julia. Τα περισσότερα από τα προγράμματα σε Julia είναι περίπου δύο φορές

πιο γρήγορα, ενώ φτάνει έως και 3 φορές πιο γρήγορα.

Ομοίως στην εκδοχή με έλεγχο, η Julia είναι πιο γρήγορο, αλλά το χάσμα είναι μικρότερο.



Σχήμα 3.3: Σύγκριση Julia με C για GridKnn with Skip



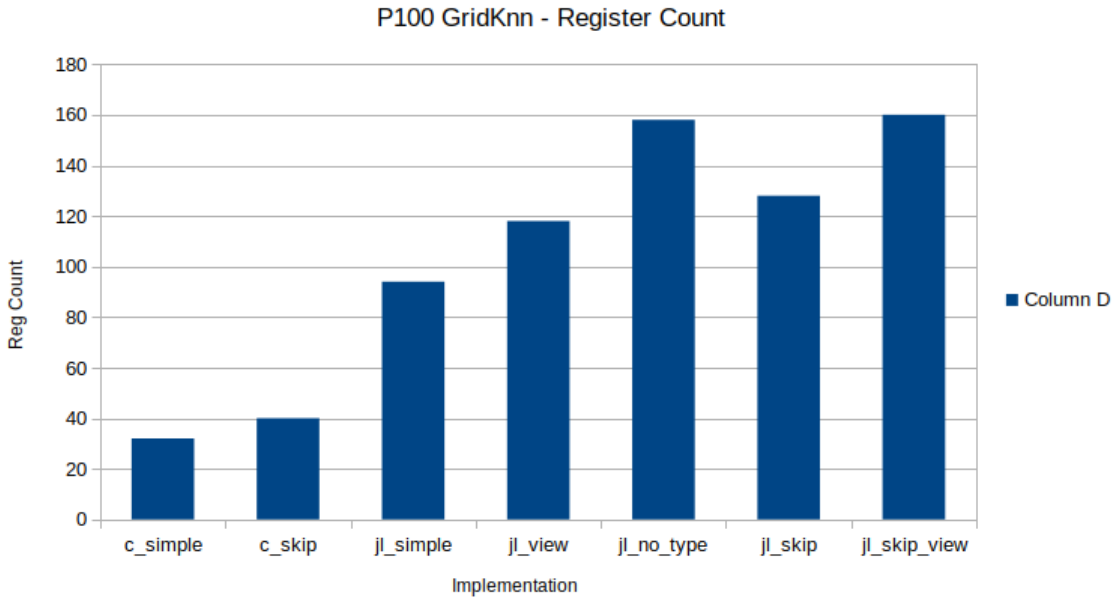
Σχήμα 3.4: Σύγκριση Julia με C για GridKnn with Skip

Είναι ενδιαφέρον πως μία δυναμική, υψηλού επιπέδου γλώσσα μπορεί να είναι σημαντικά πιο γρήγορη από μια στατική, χαμηλού επιπέδου, σε ένα αντικείμενο που σχετίζεται άμεσα με το hardware.Ακόμα πιο ενδιαφέρον είναι, πώς δυο φαινομενικά

ίδιοι κώδικες παράγουν διαφορετική PTX γλώσσα μηχανής όπως φαίνεται από την επίδοση, αλλά και από τις μετρικές παρακάτω.

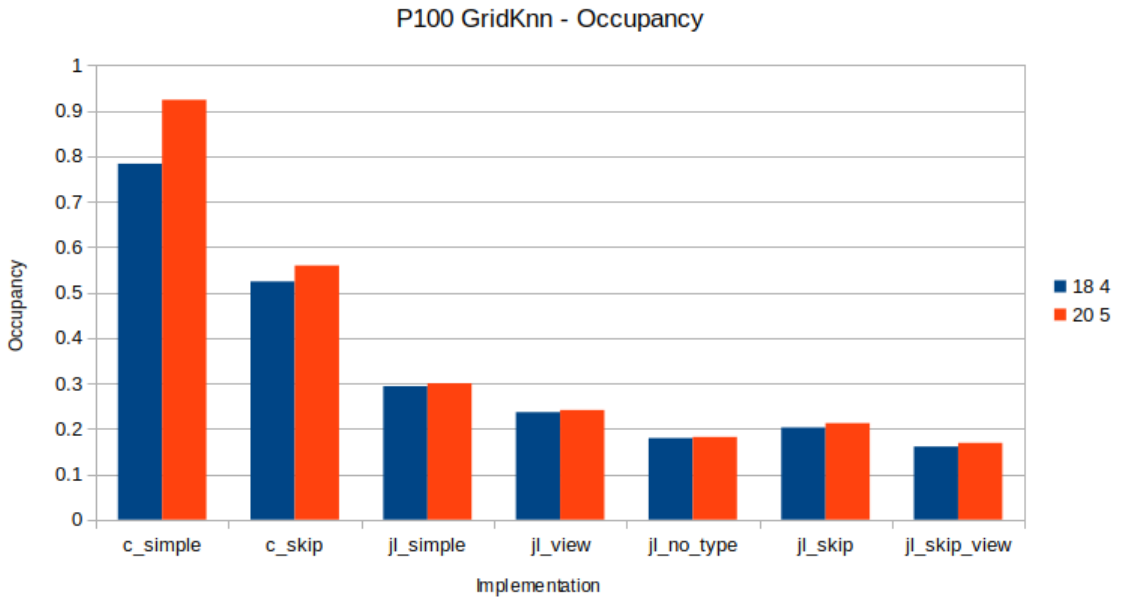
Οι 3 βασικές μετρικές, που γενικά χρησιμοποιούνται σαν γνώμονας για βελτίωση των προγραμμάτων, έχουν τις ίδιες τιμές ανεξάρτητα από τη γλώσσα που χρησιμοποιήθηκε. Οι μετρικές αυτές είναι *Global Memory Read Efficiency*, *Global Memory Write Efficiency*, *Share Memory Efficiency*. Οι μετρικές αυτές σχετίζονται με τον τρόπο που γίνεται η προσπέλαση σε πίνακες. Για την καθολική (global) μνήμη, η ανάγνωση και αποθήκευση δεδομένων πρέπει να γίνεται ομαδικά (coalesced) από τα νήματα, ενώ στη μεριζόμενη (shared) μνήμη πρέπει να αποφεύγονται τα bank conflicts. Ήταν αναμενόμενο αυτές οι μετρικές να είναι ίδιες στις δύο γλώσσες.

Η χρήση διαφορετικού αριθμού καταχωρητών είναι η πρώτη ένδειξη ότι ο παραγόμενος κώδικας μηχανής είναι διαφορετικός.



Σχήμα 3.5: Χρήση καταχωρητών για GridKNN.

Ο οδηγός προγραμματισμού CUDA σε C αναφέρει ότι όσο λιγότερους καταχωρητές χρησιμοποιεί ένας kernel τόσο περισσότερα νήματα και ομάδες νημάτων μπορούν να τρέχουν ταυτόχρονα σε μια κάρτα γραφικών, βελτιώνοντας την απόδοση. Το πρώτο κομμάτι του ισχυρισμού ισχύει στην περίπτωσή μας, το δεύτερο όμως για την βελτίωση απόδοσης όχι.

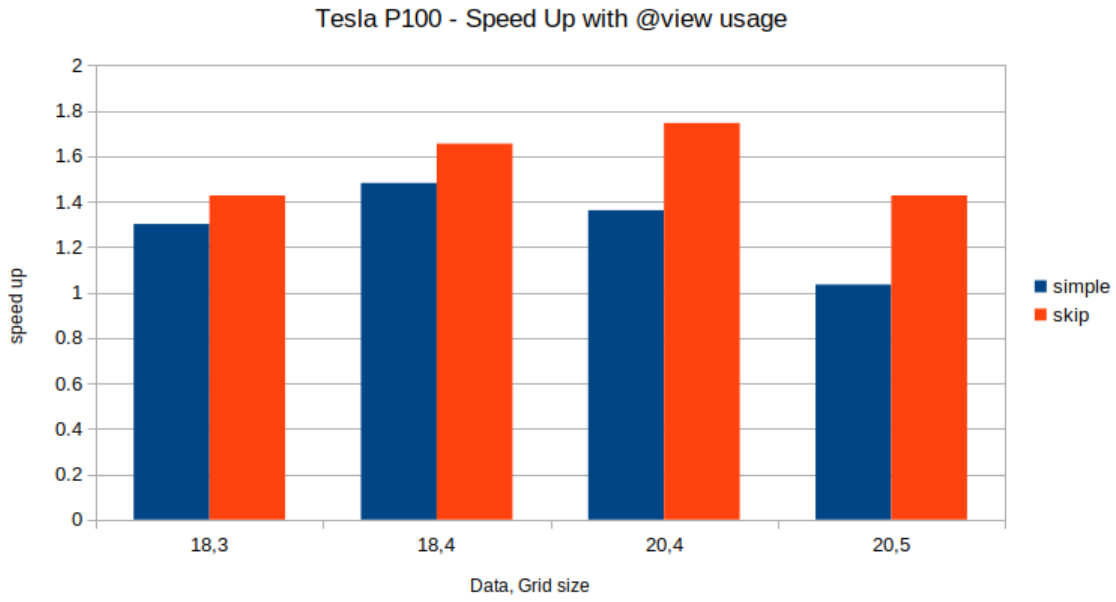


Σχήμα 3.6: Αξιοποίηση πόρων κάρτας γραφικών.

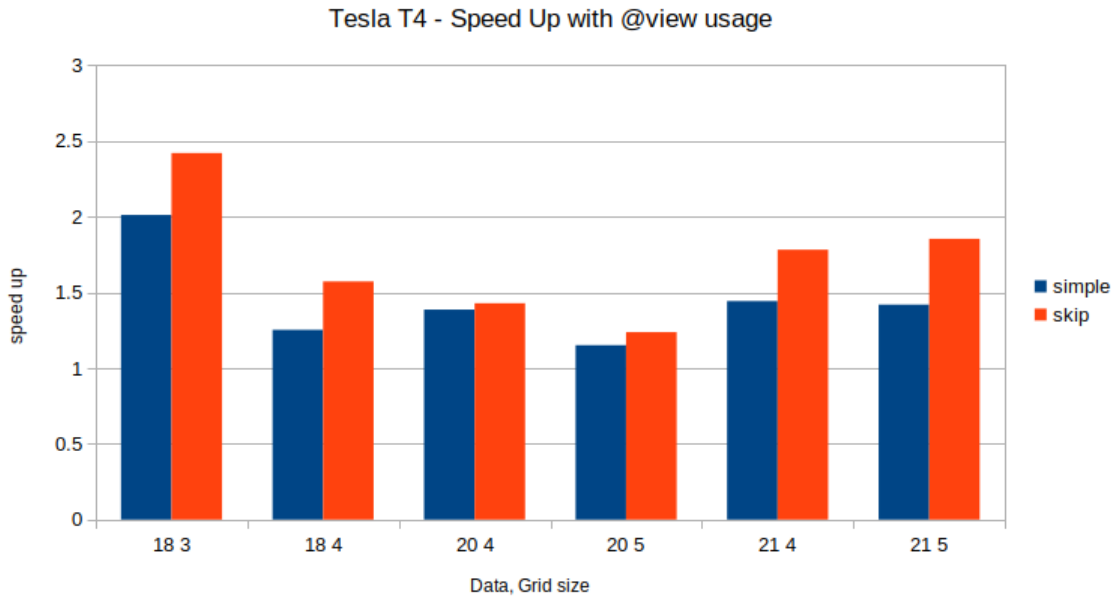
Μια ακόμα σημαντική διαφορά σε μετρικές, είναι η διαφορά στον ρυθμό εγγραφής στην L2 cache. Η L2 cache χρησιμοποιείται σαν μεσάζοντας σε διάβασμα και γράψιμο στην καθολική μνήμη. Ενώ η υλοποίηση σε C γράφει με ρυθμό εκατοντάδων MBps, η υλοποίηση σε Julia φτάνει την τάξη μεγέθους των GBps.

Παρόλο που το χάσμα στις μετρικές είναι αξιοσημείωτο, δεν παρέχει πολλή πληροφορία για τον τρόπο με τον οποίο διαφέρουν οι παραγόμενοι κώδικες μηχανής.

Τέλος, αξιοσημείωτη είναι και η επιρροή της μακροεντολής @view στην απόδοση των προγραμμάτων σε Julia.



Σχήμα 3.7: Julia - Επιτάχυνση με τη χρήση @view.



Σχήμα 3.8: Julia - Επιτάχυνση με τη χρήση @view.

Και στις δύο περιπτώσεις η χρήση του @view βελτιώνει αισθητά την απόδοση.

3.3 Μελέτη 2 - Αριθμός Τριγώνων σε Γράφο

Ο αριθμός των τριγώνων σε έναν συμμετρικό, μη κατευθυνόμενο γράφο, υπολογίζεται χρησιμοποιώντας τον πίνακα γειτνίασης A και τον τύπο $n = \frac{1}{6}\sum_{ij}(A \cdot A) \odot A$. Αξιοποιώντας τις ιδιότητες συμμετρίας για τον πίνακα A, ο τύπος μετατρέπεται σε $n = \sum_{ij}(U^T \cdot U) \odot U$, όπου U είναι το άνω τριγωνικό τμήμα του πίνακα γειτνίασης. Ο τελικός τύπος απλουστεύει το πρόβλημα σε σύγκριση των γραμμών και στήλων του U, που ορίζονται

από τα μη μηδενικά στοιχεία του ίδιου του U.

3.3.1 Περιγραφή Υλοποίησης

Υπάρχουν τρεις διαφορετικές εκδοχές, η κάθε μία με τα δικά της μειονεκτήματα:

- "row_per_th" Κάθε νήμα αναλαμβάνει μία γραμμή και ελέγχει κάθε στοιχείο της με όλες τις στήλες. (Μειονεκτήματα: Μεγάλη χρήση μεριζόμενης μνήμης, απόκλιση νημάτων, χαμηλή αξιοποίηση πόρων)
- "elem_per_th" Όλες οι γραμμές αποθηκεύονται στην μεριζόμενη μνήμη. Κάθε νήμα συγκρίνει ένα στοιχείο κάθε φορά με όλες τις γραμμές. (Μειονέκτημα: περισσότεροι υπολογισμοί, μεγάλη χρήση μεριζόμενης μνήμης)
- "elem_per_th_limit" Παρόμοιο με παραπάνω, αλλά αποθηκεύεται τμήμα των γραμμών κάθε φορά, για να είναι εφικτός ο υπολογισμός τριγώνων σε αραιούς γράφους με αρκέτα μη μηδενικά στοιχεία σε κάποιες γραμμές. (Μειονέκτημα: περισσότεροι υπολογισμοί)

Κάθε εκδοχή γράφτηκε σε Julia δύο φορές, με και χωρίς δήλωση τύπων.

3.3.2 Ποιότητα Κώδικα

Δήλωση Τύπων Σε αυτό το πρόβλημα, η μη δήλωση τύπων στην Julia οδηγεί σε χαμηλή απόδοση. Όπως φαίνεται από τα επόμενα τμήματα κώδικα, οι δύο γλώσσες κινούνται στο ίδιο επίπεδο απλότητας.

Τμήμα Κώδικα 3.6: C παράδειγμα με δήλωση τύπων.

```
// ------- Read all the columns -------
for(int i = 0; i < len; i++){
    int col = sh_row[i];
    start_row = csr_rows[col];
    end_row = col==(rows-1)? nnz : csr_rows[col+1];
    int templen = end_row - start_row;
    if(tid == 0)
        sh_len[i] = templen;
    if(tid < templen){
        sh_cols[i*stride + tid] = col_indx[start_row + tid];
    }
}
```

] Read all the other rows for i::Int32 = 1:len if threadIdx().x == 1 @inbounds col::Int32 = $sh_row[i]$@$inbounds row_start = csr_rows[col]$@$inbounds row_end = (col == rows)?nnz+Int32(1) : csr_rows[col+1] end row_start = shfl_sync(0xffffffff, row_start, 1) row_end = shfl_sync(0xffffffff, row_end, 1) templen :: Int32 = row_end − row_start (tid == 1)(sh_len[i] = templen) if tid <= templen @inbounds sh_cols[i, tid] = col_indx[row_start+tid−1] end end$

Προσπέλαση Πίνακα

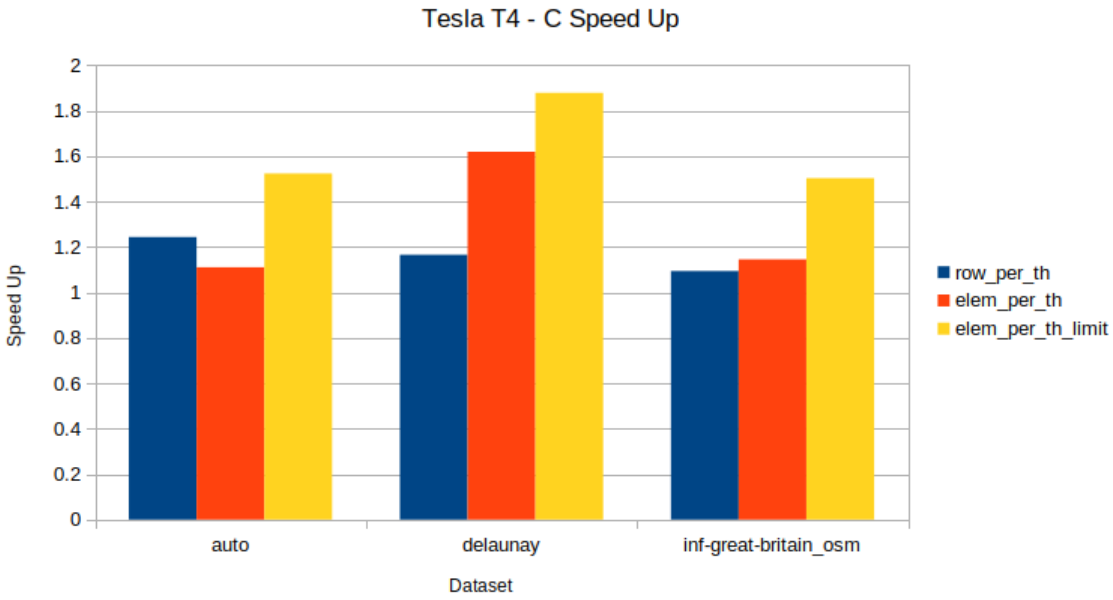
Ο αραιός πίνακας γειτνίασης αποθηκεύεται σε μορφή CSR (Compressed Sparse Rows). Έτσι κάθε πίνακας είναι μονοδιάστατος και η Julia δε μπορεί να αξιοποιήσει την πολύ

χρήσιμη πολυδιάστατη διευθυνσιοδότηση.

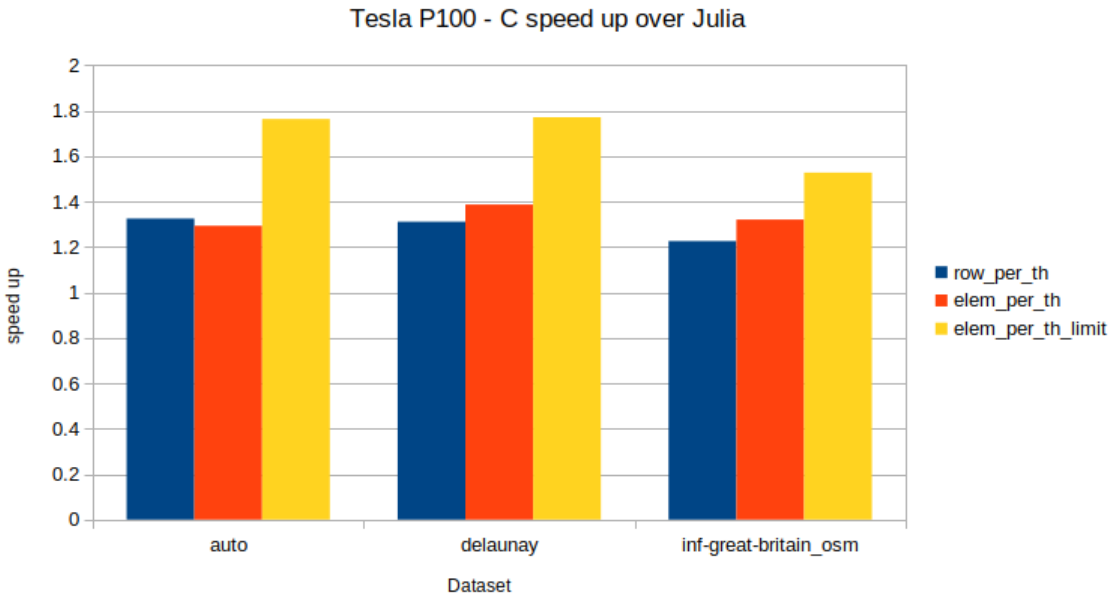
Επιπλέον, η C μπορεί μέσω των δεικτών να ορίσει ένα τμήμα του πίνακα. Αυτή τη φορά η Julia δεν επωφελείται, σε σχέση με τη C, απο τη χρήση της μακροεντολής @view.

3.3.3 Απόδοση και Μετρικές

Όλες οι υλοποίησεις εκτελέστηκαν στην Tesla P100 κάρτα γραφικών. Ακόμα, οι υλοποιήσεις με δήλωση τύπων εκτελέστηκαν και στην Tesla T4. Τρία σετ δεδομένων χρησιμοποιήθηκαν, με διαφορετικό μέγεθος, μέσο και μέγιστο αριθμό μη μηδενικών στοιχείων ανά γραμμή.



Σχήμα 3.9: C - Επιτάχυνση με δήλωση τύπων.

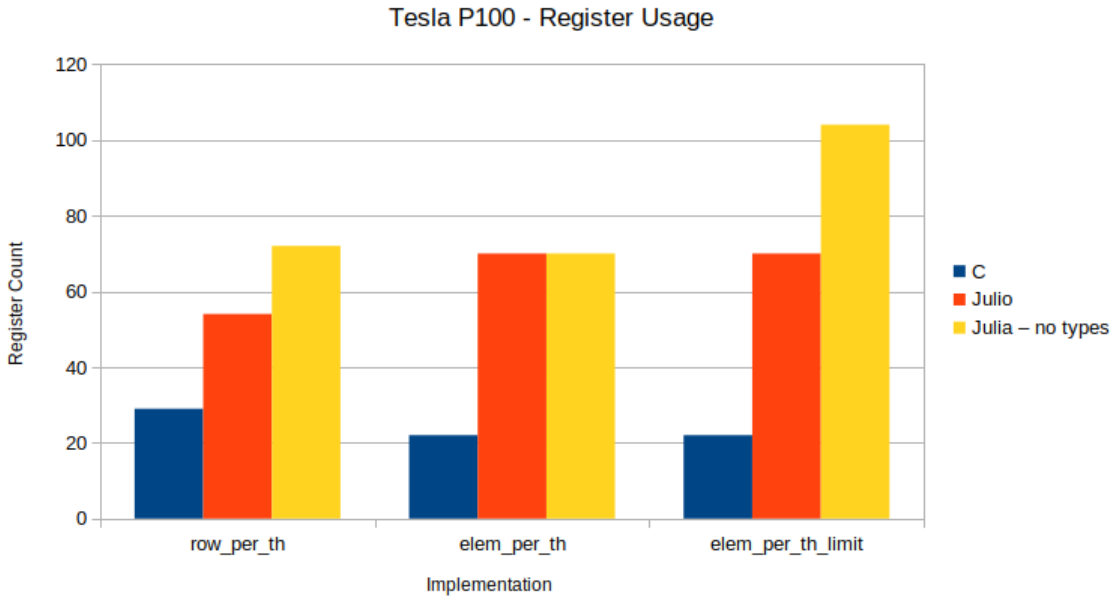


Σχήμα 3.10: C - Επιτάχυνση με δήλωση τύπων.

Η C, σε κάθε εκδοχή της, είναι πιο γρήγορη από τη Julia. Η επιτάχυνση κυμαίνεται κοντά στο 1.3x. Η χειρότερη απόδοση της Julia βρίσκεται στην 3η εκδοχή, όπου είναι 1.8x πιο αργή.

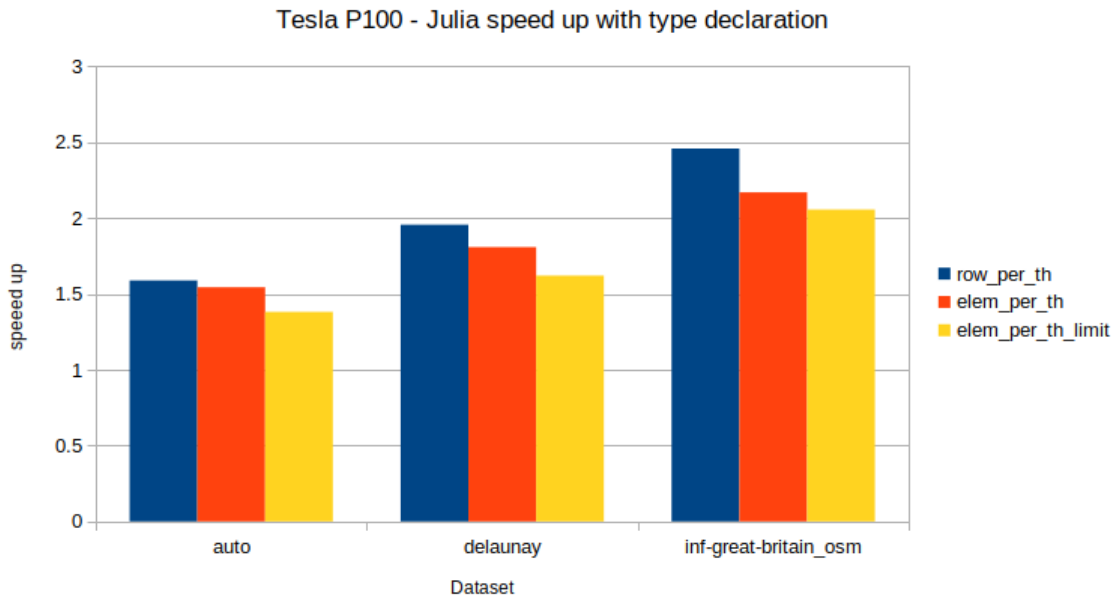
Για ακόμα μια φορά, οι τρεις βασικές μετρικές (*Global Memory Read Efficiency*, *Global Memory Write Efficiency*, Share Memory Efficiency) έχουν ίδιες τιμές για κάθε εκδοχή.

Αντίστοιχα με πριν, η Julia χρησιμποποιεί περισσότερους καταχωρητές.



Σχήμα 3.11: Χρήση καταχωρητών.

Όπως αναφέρθηκε και προηγουμένως, η μη δήλωση τύπων μειώνει την απόδοση.



Σχήμα 3.12: Julia - Επιτάχυνση με δήλωση τυπων.

Παρατηρούμε πως η δήλωση τύπων, έχει ως αποτέλεσμα τουλάχιστον ×1.5 επιτάχυνση. Αυτό το αποτέλεσμα έρχεται σε σύγκρουση με τις μετρήσεις από το πρόβλημα GridKNN, όπου είχαμε καλύτερες ταχύτητες όταν δε δηλώνουμε τύπους.

Τέλος, ο ρυθμός εγγραφής στην L2 cache είναι 12 φορές μεγαλύτερος στην Julia, οπως και στο προηγούμενο πρόβλημα.

3.4 Σχόλια

Οι μελέτες δεν κατέληξαν σε μια συγκεκριμένη ¨συνταγή¨ για να γράφουμε τον πιο γρήγορο και αποδοτικό κώδικα σε Julia για κάρτες γραφικών. Στην πρώτη περίπτωση, η δήλωση τύπου αύξησε την ταχύτητα, ενώ στη δεύτερη συνέβη το αντίθετο. Επίσης στο πρόβλημα GridKNN η Julia ήταν πιο γρήγορη, στο πρόβλημα μέτρησης τριγώνων ήταν σημαντικά πιο αργή.

Επίσης, αξιοσημείωτη είναι η συνεισφορά της μακροεντολής @view τόσο στην ποιότητα κώδικα, όσο και στην επιτάχυνση. Είναι περίεργο πώς ένα χαρακτηριστικό γλώσσας υψηλού επιπέδου βελτιώνει την απόδοση σε μια διαδικασία χαμηλού επιπέδου.

Μια πιθανή περιοχή για περισσότερη έρευνα είναι ο λόγος για τον οποίο ο μεταγλωττιστής NVTX της Julia παράγει διαφορετικό κώδικα μηχανής από τον μεταγλωττιστή nvcc για C. Μια τέτοια έρευνα θα μπορούσε να εξηγήσει γιατί η Julia χρησιμοποιεί περισσότερους καταχωρητές, πώς τους αξιοποιεί και πώς ο προγραμματιστής μπορεί να το εκμεταλλευτεί.

Όσον αφορά τον υπόλοιπο κώδικα, εκτός δηλαδή από την συνάρτηση-πυρήνα, η Julia φαίνεται να χρειάζεται λιγότερες γραμμές κώδικα για την επικοινωνία και μεταφορά δεδομένων στην κάρτα γραφικών. Επιπλέον, είναι πολύ πιο εύκολη η προδιεργασία

δεδομένων και η επαλήθευση του αποτελέσματος, λόγω του συντακτικού υψηλού επιπέδου της Julia, χωρίς να μειώνεται η συνολική απόδοση.

Κεφάλαιο 4

Υπολογισμοί Στένσιλ

4.1 Ορισμός του Στένσιλ

Οι κώδικες στένσιλ είναι μια τάξη επαναληπτικών διαδικασιών και χρησιμοποιούνται σε πληθώρα εφαρμογών. Περιλαμβάνουν μια σειρά περασμάτων (βήματα-χρόνου) πάνω από έναν πίνακα δεδομένων. Σε κάθε χρόνο, ανανεώνονται οι τίμες των σημείων σύμφωνα με μια συνάρτηση των γειτονικών τους σημείων. Το γεωμετρικό σχήμα αυτής της γειτονιάς αποτελεί το στένσιλ.

Οι υπολογισμοί στένσιλ χρησιμοποιούνται στην προσωμοίωση ενός συστήματος για πολλαπλά βήματα χρόνου ή για να διαπιστωθεί αν ένα σύστημα συγκλίνει σε μια κατάσταση σε βάθος χρόνου. Ακόμα, τα στένσιλ χρησιμοποιούνται σε προβλήματα που λύνονται από μεθόδους πεπερασμένων διαφορών, για την αριθμητική επίλυση μερικών διαφορικών εξισώσεων, στα κυτταρικά αυτόματα (cellular automata), στην επεξεργασία εικόνων και σε άλλα επιστημονικά πεδία.

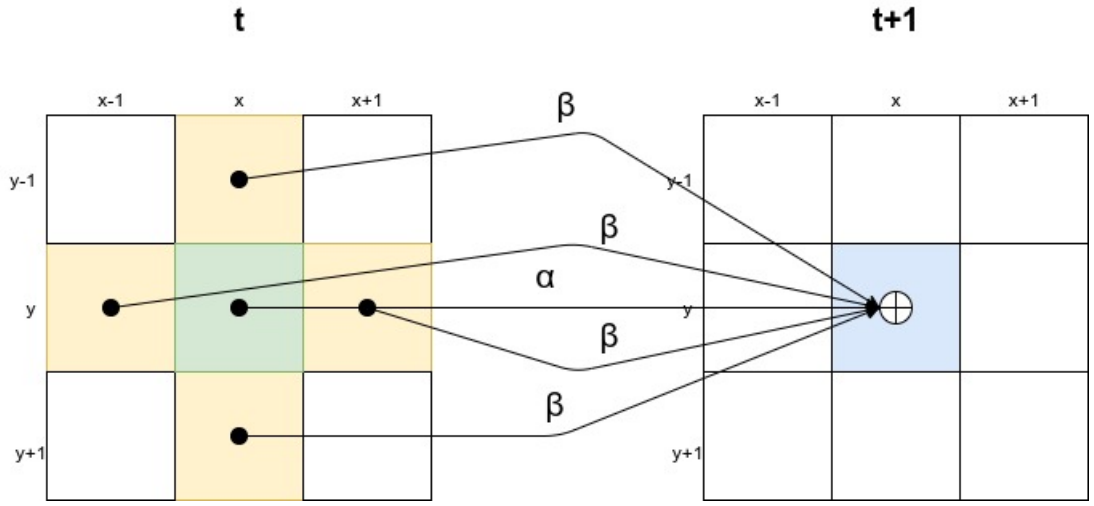
Στην απλή τους μορφή, το στένσιλ εφαρμόζονται σε μόνο ένα κανονικό πλέγμα, ενώ οι εκδοχές που εφαρμόζονται σε μη-δομημένα ή πολλαπλά πλέγματα είναι αρκετά πιο πολύπλοκες.

Ο χρόνος υπολογισμού και η χρήση μνήμης για τον υπολογισμό στένσιλ αυξάνεται γραμμικά με τον όγκο δεδομένων. Επομένως, παράλληλες υλοποιήσεις για τον υπολογισμό στένσιλ έχουν πολύ μεγάλη σημασία για τους ερευνητές [6].

Ένα χαρακτηριστικό παράδειγμα εφαρμογής στένσιλ, είναι η τρισδιάστατη διακριτή Λαπλασιανή,

$$D^{t+1}[x,y,z] = aD^t[x,y,z] + \beta(D^t[x-1,y,z] + D^t[x+1,y,z] + \\ D^t[x,y-1,z] + D^t[x,y+1,z] + D^t[x,y,z-1] + D^t[x,y,z+1])$$

που είναι ανάλογη με τον συνεχή Λαπλασιανό τελεστή: $\nabla = \Delta^2 = (\frac{\partial^2}{\partial^2 x} + \frac{\partial^2}{\partial^2 y} + \frac{\partial^2}{\partial^2 z})$.



Σχήμα 4.1: Οπτικοποίηση στένσιλ για ένα στοιχείο και βήμα χρόνου για την δισδιάστατη διακριτή Λαπλασιανή.

4.2 Στένσιλ σε κάρτες γραφικών

Οι υπολογισμοί στένσιλ είναι απο τη φύση τους παράλληλο πρόβλημα. Η τιμή κάθε στοιχείου στο επόμενο βήμα χρόνου μπορεί να υπολογιστεί ανεξάρτητα από τα υπόλοιπα στοιχεία. Αυτό μπορεί πολύ εύκολα να το εκμεταλλευτεί η κάρτα γραφικών για επιτάχυνση. Παρόλα αυτά, ο έμφυτος παραλληλισμός δεν οδηγεί στην αποδοτικότερη λύση. [6]. Οι υπολογισμοί στένσιλ εφαρμόζονται σε δεδομένα που ξεπερνούν σε μέγεθος τις διαθέσιμες κρυφές μνήμες, ενώ η επαναχρησιμοποίηση δεδομένων περιορίζεται από το μικρό αριθμό σημείων που περιλαμβάνει το σχήμα του στένσιλ. Έτσι, στα πολυπύρηνα συστήματα, η ταχύτητα υπολογισμού περιορίζεται από την ρυθμαπόδοση της μνήμης. [2]. Η αναδιάταξη των υπολογισμών για την πλήρη αξιοποίηση της ιεραρχίας μνήμης μαζί με άλλες τεχνικές βελτιστοποίησης αποτελούν το αντικείμενο έρευνας εδώ και πολλά χρόνια.

4.3 Βελτιστοποίηση

4.3.1 Μοτίβο Προσπέλασης Μνήμης

[7] Η βελτιστοποίηση της χρήσης μνήμης είναι το πιο σημαντικό κομμάτι για την αύξηση της επιτάχυνσης και απαραίτητη για τους υπολογισμούς στένσιλ που περιορίζονται από τη ρυθμαπόδοση της μνήμης. Η ανάγνωση και εγγραφή δεδομένων στις κάρτες γραφικών είναι από τις πιο αργές διαδικασίες και χρειάζονται συγκεκριμένα μοτίβα προσπέλασης για την πλήρη αξιοποίηση της ρυθμαπόδοσης.

Για τα δισδιάστατα στένσιλ ο τρόπος προσπέλασης μνήμης είναι προφανής. Κάθε ομάδα νημάτων διαβάζει έναν τετράγωνο από τα δεδομένα στην μεριζόμενη μνήμη. Έτσι, αξιοποιεί τις ομαδικές (coalesced) προσπελάσεις νημάτων και μπορεί να επαναχρησιμοποιήσει δεδομένα. Έπειτα κάθε νήμα μπορεί να διαβάσει τα δεδομένα που χρειάζεται από τη μεριζόμενη μνήμη χωρίς να υπάρχει διαμάχη στις τράπεζες μνήμης, αφού τα δεδομένα που διαβάζονται είναι συνεχόμενα σε μια γραμμή.

Για τα τρισδιάστατα στένσιλ, η διαδικασία είναι πιο περίπλοκη. Ακολουθώντας την ίδια λογική με τις δύο διαστάσεις, ένας κύβος από τα δεδομένα μεταφέρεται στην μεριζόμενη μνήμη. Οι ομαδικές προσπελάσεις αξιοποιούνται και οι διαμάχες στις τράπεζες μνήμης αποφεύγονται. Παρόλα αυτά, η προσέγγιση αυτή χρησιμοποιεί υπερβολικά πολλή μνήμη που οδηγεί σε μειωμένη εκμετάλλευση των πόρων της κάρτας γραφικών ή ακόμα και σε αδυναμία εκτέλεσης αν η μνήμη που απαιτείται ξεπερνάει τη διαθεσιμή. Ο Micikevicous [8] παραθέτει μια υλοποίηση για κάρτες γραφικών στην οποία διαβάζει ένα τετράγωνο μνήμης στη μεριζόμενη μνήμη και χρησιμοποιεί τους καταχωρητές για την τρίτη διάσταση, εξοικονομώντας μεριζόμενη μνήμη. Ο κώδικας της υλοποίησης επικεντρώνεται μόνο σε τρισδιάστατες πεπερασμένες διαφορές και δεν μπορεί να εφαρμοστεί για άλλα σχήματα στένσιλ. Ο Mo κ.α. [9] παρουσιάζουν μια τροποποιημένη υλοποίηση στην οποία χρησιμοποιούν τους καταχωρητές όχι για την ανάγνωση δεδομένων, αλλά για την αποθήκευση μερικών αποτελεσμάτων. Πέρα από την αποδοτικότερη χρήση μνήμης, η υλοποίηση αυτή μπορεί να εφαρμοστεί σε οποιοδήποτε σχήμα στένσιλ.

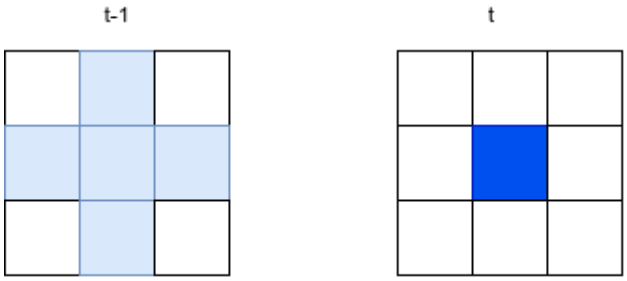
4.3.2 Αλλαγή Μεγέθους Πίνακα

[7] Οι εντολές ελέγχου ροής έχουν σημαντική επιρροή στην ταχύτητα εκτέλεσης. Όταν υπάρχει διακλάδωση, δηλαδή κάθε νήμα δεν ακολουθεί το ίδιο ¨μονοπάτι¨, τα διαφορετικά μονοπάτια εκτελούνται σειριακά και όχι παράλληλα. Στους υπολογισμούς στένσιλ, διακλάδωση εντολών ελέγχου εμφανίζεται όταν γίνεται έλεγχος αν κάποιο στοιχείο βρίσκεται εντός των ορίων του πίνακα. Για να αποφευχθεί αυτό το πρόβλημα, αυξάνουμε το μέγεθος του πίνακα, γεμίζοντας τις άκρες με μηδενικά, ώστε να είμαστε απολύτως σίγουροι ότι δε θα υπάρχει προσπάθεια πρόσβασης σε στοιχείο εκτός των ορίων. Επιπροσθέτως, μπορούμε να αυξήσουμε κι άλλο το μέγεθος του πίνακα, ώστε να είναι πολλαπλάσιο του 32 (warp size) και να μην γίνεται έλεγχος στα άκρα για αδρανή νήματα.

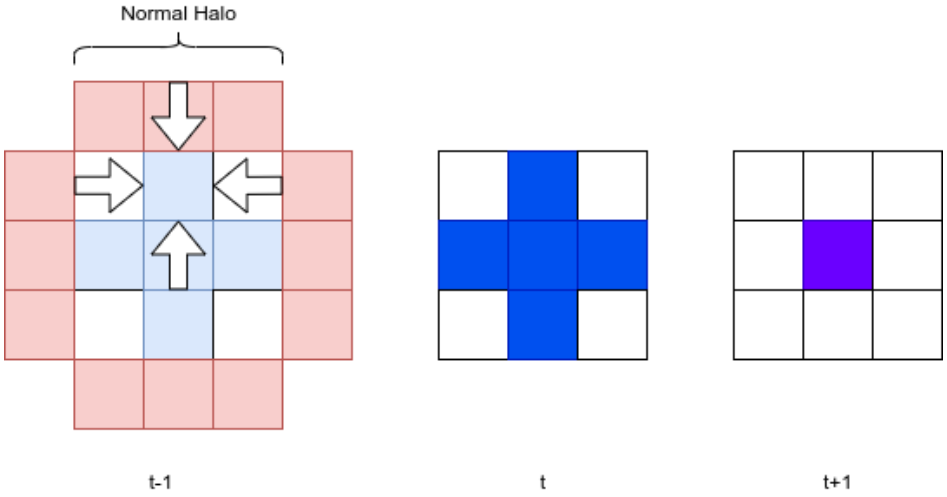
4.3.3 Ομαδοποίηση στον χρόνο

Η ομαδοποίηση στον χρόνο αποτελεί μια μέθοδο βελτιστοποίησης που στοχεύει στην επαναχρησιμοποίηση των δεδομένων που έχουν ήδη διαβαστεί σε κάποια γρήγορη μνήμη. Στην απλή της μορφή, η συνάρτηση για τον υπολογισμό στένσιλ υπολογίζει μόνο ένα βήμα στον χρόνο και καλείται όσες φορές χρειάζεται. Αυτό σημαίνει ότι σε κάθε κλήση τα δεδομένα διαβάζονται και αποθηκεύονται στην καθολική μνήμη, το οποίο όπως έχει αναφερθεί, είναι αργό. Ομαδοποιώντας περισσότερα βήματα χρόνου σε μία κλήση μειώνονται οι προσπελάσεις στην καθολική μνήμη.

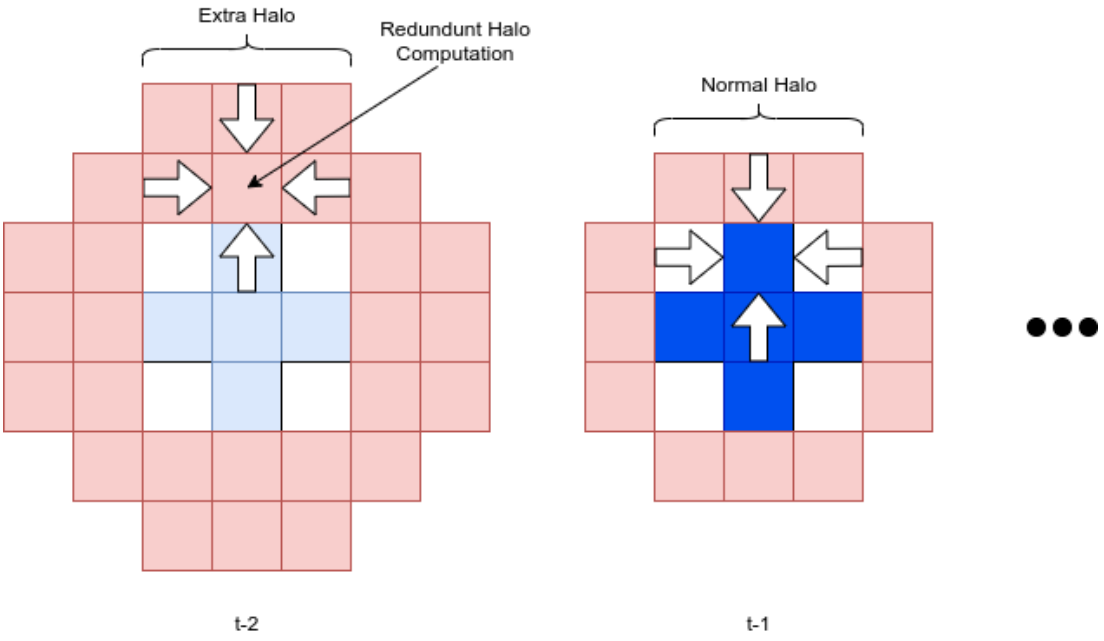
Η ομαδοποίηση στον χρόνο έχει εφαρμοστεί επιτυχώς για μονοδιάστατα και δισδιάστατα στένσιλ, σε διάφορες αρχιτεκτονικές, μονοπύρηνα [10] και πολυπύρηνα [11] συστήματα και κάρτες γραφικών [2], φτάνοντας μέχρι και 5x επιτάχυνση. Συγκεκριμένα για τις κάρτες γραφικών, η μέθοδος αυτή υλοποιείται εφαρμόζοντας το στένσιλ περισσότερες από μία φορές στα δεδομένα που έχουν διαβαστεί στην μεριζόμενη μνήμη. Αυτό απαιτεί να διαβαστούν περισσότερα δεδομένα στην μνήμη και να γίνουν περιττές πράξεις για την ανανέωση των τιμών των επιπλέον στοιχείων.



Σχήμα 4.2: Ένα βήμα χρόνου για το κεντρικό στοιχείο. Δε χρειάζονται επιπλέον στοιχεία.



Σχήμα 4.3: Δύο βήματα χρόνου για το κεντρικό στοιχείο.Χρειάζονται επιπλέον στοιχεία, το οποία όμως έχουν ήδη διαβαστεί στο κανονικό στεφάνι(halo).



Σχήμα 4.4: Δύο βήματα χρόνου για το κεντρικό στοιχείο. Χρειάζονται επιπλέον στοιχεία για το στεφάνι και απαιτούνται περιττές πράξεις για την ανανέωσή του.

Στο σχήμα 4.2 παρουσιάζεται η περίπτωση για ένα βήμα χρόνου στο κεντρικό στοιχείο. Τα δεδομένα που χρειάζονται βρίσκονται στο βασικό τετράγωνο. Στο σχήμα 4.3 για δύο βήματα χρόνου τα γαλάζια στοιχεία χρειάζεται να ανανεωθούν για να υπολογιστεί η τιμή του κεντρικού σημείου για δύο βήματα χρόνου. Για την ανανέωση των γαλάζιων στοιχείων χρειάζεται να διαβαστεί ένα στεφάνι γύρω από το τετράγωνο των δεδομένων. Στο σχήμα 4.4 για τρία βήματα χρόνου, πρέπει να ανανεωθούν οι τιμές του στεφανιού και έτσι πρέπει να αυξηθεί το μέγεθός του.

Είναι σημαντικό να αναφερθεί, ότι τα στοιχεία του στεφανιού είναι μέρος του βασικού τετραγώνου μιας άλλης ομάδας νημάτων και για αυτό θεωρούνται περιττές πράξεις οι ανανεώσεις του στεφανιού.

Επίσης, για μια ομάδα νημάτων, όσο πιο πολλά βήματα χρόνου ομαδοποιούνται, τόσο πιο μικρό γίνεται το βασικό τετράγωνο για να μη υπάρχουν εντολές ελέγχου που εμποδίζουν την παραλληλοποίηση εντολών.

Είναι πολύ σημαντικό να ισορροπιστούν τα παραπάνω μειονεκτήματα με την μείωση προσπέλασης καθολικής μνήμης για να βρεθεί ο κατάλληλος αριθμός βημάτων χρόνου που ομαδοποιείται. Τέτοια έρευνα σε κάρτες γραφικών και σε διαφορετικά σχήματα στένσιλ έχουν πραγματοποιήσει ο Meng κ.α. [1]

Στις τρεις διαστάσεις, η χρήση μεριζόμενης μνήμης και οι περιττές πράξεις αυξάνονται πολύ γρηγορότερα με τον αριθμό βημάτων χρόνου. Ο Holewinski κ.α., όπως και ο Meng κ.α. δεν κατάφεραν καμία επιτάχυνση με ομαδοποίηση στον χρόνο για τρισδιάστατα στένσιλ. Ο Nguyen κ.α. πέτυχαν 1.8x επιτάχυνση, αλλά η μέθοδός τους δεν είναι κατάλληλη για αυτόματη παραγωγή υψήλης ταχύτητας κώδικα για κάρτες γραφικών.

4.3.4 Αυτόματη Ρύθμιση

Η αυτόματη ρύθμιση παραμέτρων για συναρτήσεις-πυρήνες κάρτας γραφικών έχει εφαρμοστεί με επιτυχία σε διάφορες περιπτώσεις, όπως σε αραιά και πυκνά προβλήματα γραμμικής άλγεβρας και επεξεργασίας σημάτων. Οι προγραμματιστές που γράφουν προγράμματα σε CUDA πρέπει να προσδιορίσουν τον αριθμό των νημάτων ανά ομάδα, το πλήθος των ομάδων, καθώς και παραμέτρους από μεθόδους βελτιστοποίησης που βρίσκονται στο εσωτερικό της συνάρτησης-πυρήνα. Οι προγραμματιστές μπορεί να μην έχουν την διαίσθηση που απαιτείται για να ορίσουν αυτές τις παραμέτρους, οι οποίες έχουν περίπλοκες και απρόβλεπτες συσχετίσεις μεταξύ τους [12]. Επομένως η αυτόματη ρύθμιση των παραμέτρων είναι αναγκαία διαδικασία για να βρεθεί η καλύτερη επιλογή μέσα από τον εκτενή και περίπλοκο χώρο των ρυθμίσεων.

Για τους υπολογισμούς στένσιλ έχουν προταθεί αρκέτα frameworks [6] [12] [13] [14] τα οποία υπολογίζουν τις παραμέτρους εκτέλεσης (αριθμό νημάτων και ομάδων) καθώς και αριθμό βημάτων χρόνου για τη ομαδοποίηση στο χρόνο, πόσο χρειάζεται να ξεδιπλωθούν οι βρόγχοι επανάληψης κ.α. Ενδεικτικά, τεχνικές που χρησιμοποιούν είναι παραμετροποίηση του χώρου των ρυθμίσεων, μηχανική μάθηση ή και ευριστικούς κανόνες. Μία αξιοσημείωτη υλοποίηση, που διαφέρει από τις προηγούμενες, δημιούργησε ο Lim κ.α. [12] ,όπου αναλύει στατικά τον παραγόμενο κώδικα μηχανής PTX, εκτιμάει διάφορες μετρικές και ορίζει τις παραμέτρους χωρίς το μεγάλο κόστος εκτέλεσης της συνάρτησης πυρήνα.

4.3.5 Αυτόματη Παραγωγή Κώδικα

Η αυτόματη παραγωγή παράλληλου κώδικα αποτελεί ένα πολύ σημαντικο πεδίο έρευνας και έχει ως στόχο να κρύψει από τους ερευνητές άλλων πεδίων τη δύσκολη διαδικασία επιτάχυνσης προγραμμάτων με τεχνικές παραλληλοποίησης. Frameworks όπως LibGeoDecomp [15], PATUS [16] και του Holewinksi [2] παράγουν αποδοτικό παράλληλο κώδικα, για διάφορες αρχιτεκτονικές, συμπεριλαμβανομένου καρτών γραφικών, ενσωματώνοντας τεχνικές βελτιστοποίησης εσωτερικά στις συναρτήσεις-πυρήνες με μεγάλη επιτυχία. Ο χρήστης δίνει σαν είσοδο την περίγραφη του προβλήματος/στένσιλ σε μια εξειδικευμένη γλώσσα προγραμματισμού, η οποία μετά αναλύεται συντακτικά, παράγεται ο παράλληλος κώδικας και εκτελείται με τα δεδομένα που έχουν δοθεί. Μια σημαντική παρατήρηση είναι πως τα παραπάνω frameworks έχουν περίπλοκη δομή καθώς χρησιμοποιούν εξωτερικές βιβλιοθήκες/frameworks, γραμμένα σε διαφορετικές γλώσσες, για την ανάλυση και παραγωγή κώδικα.

Κεφάλαιο 5

Framework για παραγωγή κώδικα στένσιλ και εκτέλεση σε πολλαπλές κάρτες γραφικών

5.1 Κίνητρο

Οι υπολογισμοί στένσιλ χρησιμοποιούνται σε ένα μεγάλο εύρος επιστημονικών πεδίων. Η σύνταξη μιας συνάρτησης πυρήνα από την αρχή ή η τροποποίηση κάποιας υπάρχουσας υλοποίησης δεν είναι αποδοτικές λύσεις και μπορεί εύκολα να γίνουν λάθη.

Η απόκρυψη της διαδικασίας σύνταξης και επιτάχυνσης των προγραμμάτων είναι πολύ σημαντική για την αποδοτικότητα των ερευνητών, καθώς μειώνει την πιθανότητα ανθρώπινου λάθους, τον χρόνο ανάπτυξης και εκτέλεσης.

Όπως αναφέρθηκε στο προηγούμενο κεφάλαιο, τα υπάρχοντα frameworks για αυτόματη παραγώγη κώδικα έχουν περίπλοκη δομή με εξωτερικές βιβλιοθήκες και είναι γραμμένα σε διαφορετικές γλώσσες. Επιπλέον, η διεπαφή με τον χρήστη σε κάποια είναι γραμμένη σε C++, μια χαμηλού επιπέδου γλώσσα που περιορίζει την εκφραστικότητα στον ορισμό του προβλήματος και χρειάζεται περισσότερες γραμμές τυπικού κώδικα.

5.2 Σκοπός και Περιγραφή του framework

Η βασική λειτουργία του framework είναι η αυτόματη παραγωγή κώδικα για υπολογισμό στένσιλ σε κάρτες γραφικών. Επιπλέον, ενσωματώνει στη συνάρτηση-πυρήνα τεχνικές βελτιστοποίησης σύμφωνα με παραμέτρους που ορίζει ο χρήστης και μπορεί να εκτελέσει υπολογισμούς στένσιλ σε μία ή περισσότερες κάρτες γραφικών ή ακόμα και μέσω FFT. Πέρα από την έρευνα, στην επιτάχυνση υπολογισμών στένσιλ σε κάρτες γραφικών, το framework αυτό δημιουργήθηκε με σκοπό την ανάδειξη της Julia ως κατάλληλης γλώσσας για την υλοποίηση ενός ολοκληρωμένου προγράμματος, χωρίς την ανάγκη για εξωτερικές βιβλιοθήκες και με εξίσου καλή επιτάχυνση. Συγκεκριμένα, οι στόχοι για την Julia είναι να δείξουμε ότι μπορεί:

- να δημιουργήσει μια απλή και κατανοητή διεπαφή για τον χρήστη χωρίς να α-

παιτεί πολύ κώδικα από αυτόν.

- εύκολα να αναλύσει συντακτικά κώδικα που έχει δώσει σαν είσοδο ο χρήστης.
- να παράγει αυτόματα αποδοτικό κώδικα και να εφαρμόζει μεθόδους βελτιστοποίησης.
- να έχει υψηλή απόδοση στις εκτελέσεις σε μία κάρτα γραφικών.
- να εκμεταλλευτεί εύκολα περισσότερες από μια κάρτες γραφικών.

Όσον αφορά το κομμάτι της έρευνας για την επιτάχυνση των υπολογισμών στένσιλ, παρουσιάζονται δύο καινούριες τεχνικές:

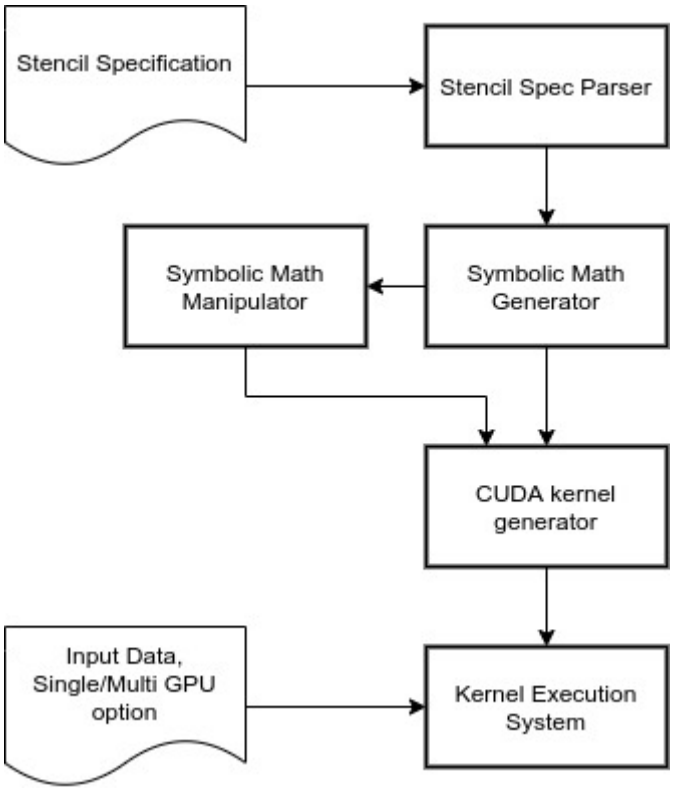
- ομαδοποίηση στον χρόνο για μη ολιστικές υλοποιήσεις για στένσιλ
- εκτέλεση πολλών βημάτων χρόνου με FFT

Πέρα από τις μετρήσεις και τα συμπεράσματα για την επιτάχυνση των υπολογισμών στένσιλ, είναι σημαντικό να μελετηθεί η συνεισφορά της Julia στην ερευνητική διαδικασία. Δηλάδη, αν δίνει ευελεξία στον ερευνητή να υλοποιήσει ή να τροποποιήσει λειτουργίες του προγράμματός του και να μετατρέψει εύκολα τις ιδέες του σε πρωτότυπο.

5.3 Δομή του framework

Το framework αποτελείται από 4 δομικά στοιχεία. Ο *συντακτικός αναλυτής*, ο οποίος δέχεται σαν είσοδο κώδικα Julia (expression) και τον τροποποιεί. Στη συνέχεια το δεύτερο δομικό στοιχείο τροποποιεί αυτόν τον κώδικα και τον μετατρέπει σε συμβολικά μαθηματικά. Σε αυτό το σημείο γίνεται και η μαθηματική έκφραση για την ομαδοποίηση στον χρόνο. Τρίτον, ο *παραγωγός κώδικα* μετατρέπει τα συμβολικά μαθηματικά σε αριθμητικές εντολές και δημιουργεί την συνάρτηση-πυρήνα για την κάρτα γραφικών. Τέλος, το *σύστημα εκτέλεσης* είναι υπεύθυνο για την επικοινωνία με τις κάρτες γραφικών, την κατάλληλη επεξεργασία των δεδομένων και την εκτέλεση της συνάρτησης που έχει παραχθεί.

Όλα τα δομικά στοιχεία είναι γραμμένα σε Julia με τη βοήθεια βιβλιοθηκών γραμμένες στην ίδια γλώσσα. Η χρήση εξωτερικών προγραμμάτων/βιβλιοθηκών ή άλλης γλώσσας δεν ήταν αναγκαία.



Σχήμα 5.1: Γενική εικόνα της αρχιτεκτονικής.

Καθορισμός του στένσιλ και συντακτική ανάλυση Ο χρήστης μπορεί να καθορίσει το στένσιλ με δύο τρόπους. Είτε να δώσει σαν είσοδο ένα τρισδιάστατο πίνακα με συντελεστές, είτε τη μαθηματική έκφραση μέσω κώδικα. Η περίπτωση με τον τρισδιάστατο πίνακα είναι πολύ απλή και δε χρειάζεται συντακτική ανάλυση. Για τη δεύτερη περίπτωση, αξιοποιούνται οι δυνατότητες της Julia σε μετα-προγραμματισμό και ενδοσκόπηση)introspection για τη συντακτική ανάλυση και μετατροπή του κώδικα που δόθηκε σαν είσοδο.

Η μαθηματική περιγραφή του στένσιλ μπορεί να δοθεί από τον χρήστη μέσω κώδικα με εύληπτο και απλό τρόπο:

Τμήμα Κώδικα 5.1: Julia Παράδειγμα ορισμού στένσιλ-αστέρι

```
star_stencil = @def_stencil_expression c[0]D[x,y,z]
           + @sum(i, 1, 4, c[i]*(
           D[x+i,y,z] + D[x,y+i,z] + D[x,y,z+i] +
           D[x-i,y,z] + D[x,y-i,z] + D[x,y,z-i]))
```

Το πρώτο πράγμα που γίνεται όταν καλείται η μακρο-εντολή @def_stencil είναι να εκτελέσει τις εσωτερικές συναρτήσεις μακρο-εντολές και να αντικαταστήσει τις τιμές στην έκφραση που έχει δοθεί σαν όρισμα. Η έκφραση expression στην Julia είναι ένας τύπος δεδομένων που κρατάει κώδικα Julia και δίνει τις δυνατότητες ενδοσκόπησης και μετα-προγραμματισμού. Η έκφραση έχει μορφή δέντρου, οπότε μέσω αναδρομής μπορούμε να φτάσουμε σε δείκτες πινάκων, σταθερούς αριθμούς και να τους τροποποιήσουμε. Η μακρο-εντολή @sum λειτουργεί όπως ο τελεστής , ανοίγοντας την έκφραση και αντικαθιστώντας τους δείκτες με αριθμούς.

Συμβολικά Μαθηματικά Το δομικό στοιχείο συμβολικών μαθηματικών έχει δύο καθήκοντα. Το πρώτο και σημαντικότερο είναι να μετατρέψει τον κώδικα σε συμβολική μαθηματική έκφραση. Αυτό γίνεται πολύ εύκολα εκτελώντας τον τροποποιημένο κώδικα σε καθολική εμβέλεια (global scope). Αυτό σημαίνει πως η Julia θα ψάξει για έναν πίνακα D και έναν πίνακα c, τους οποίους θα έχουμε ήδη ορίσει να περιέχουν σύμβολα και συντελεστές αντίστοιχα.

Η δεύτερη δουλειά, η οποία είναι προαιρετική, είναι η τροποποίηση της συμβολικής έκφρασης για ομαδοποίηση βημάτων χρόνου. Ο τρόπος που γίνεται η ομαδοποίηση περιγράφεται αναλυτικά σε επόμενη παράγραφο. Στόχος είναι η δημιουργία μιας νέας μαθηματικής έκφρασης, η οποία όταν θα εφαρμόζεται μία φορά έχει ως αποτέλεσμα πολλαπλά βήματα χρόνου. Η υλοποίηση περιλαμβάνει αναδρομική αντικατάσταση κάθε συμβόλου της έκφρασης με την ίδια την έκφραση μετατοπισμένη κατάλληλα.

Η βιβλιόθηκη συμβολικών μαθηματικών που χρησιμοποιήθηκε ειναι η Symengine, η οποία είναι ένα ¨περιτύλιγμα¨ (wrapper) σε Julia γύρω από τη βιβλιοθήκη symengine σε C++.

Παραγωγή Κώδικα Αφού αναλυθεί ο κώδικας που έχει δώσει ο χρήστης σαν είσοδο για να περιγράψει το στένσιλ, καθορίζονται οι αριθμητικές πράξεις που χρειάζονται. Σαν πρότυπο για την δημιουργία της συνάρτησης πυρήνα χρησιμοποιήθηκε η τροποποιημένη υλοποίηση της NVIDIA που παρουσιάστηκε στο προηγούμενο κεφάλαιο. Το πρότυπο μπορεί να διασπαστεί στα εξής τμήματα:

- Αρχικοποίηση. Γίνεται η κατανομή μνήμης, ορισμός υποπινάκων και αρχικοποίηση καταχωρητών.
- Βρόγχος επανάληψης κατά z άξονα:
  - Μεταφορά δεδομένων στη μεριζόμενη μνήμη
  - Αριθμητικές πράξεις.
  - Αποθήκευση αποτελεσμάτων στην καθολική μνήμη.

Η βιβλιοθήκη για συμβολικά μαθηματικά είναι πολύ χρήσιμη για το κομμάτι των αριθμητικών πράξεων. Για κάθε σύμβολο που μπορεί να περιλαμβάνεται στην έκφραση αναζητείται υπολογίζεται ο συντελεστής και η σχετική θέση του στον τρισδιάστατο χώρο ως προς το κέντρο του στένσιλ. Σε κάθε σύμβολο που έχει μη μηδενικό συντελεστή αναλογεί μια αριθμητική πράξη. Ο καταχωρητής στον οποίο θα αποθηκευτεί το αποτέλεσμα εξάγεται από τη θέση στον χώρο.

ΑΦού ο κώδικας έχει παραχθεί, δίνεται ως όρισμα στην βιβλιοθήκη SyntaxTree όπου και μετατρέπεται σε συνάρτηση που μπορεί να κληθεί.

Σύστημα Εκτέλεσης Το σύστημα εκτέλεσης είναι υπεύθυνο για την εκτέλεση της συνάρτησης πυρήνα σε μία ή περισσότερες κάρτες γραφικών, ή για τον υπολογισμό στένσιλ μέσω FFT. Για μία κάρτα γραφικών, το σύστημα αλλάζει το μέγεθος του πίνακα ανάλογα με την ομαδοποίηση βημάτων χρόνου, μεταφέρει τα δεδομένα στην κάρτα και καλεί τη συνάρτηση πυρήνα όσες φορές χρειάζεται. Για περισσότερες κάρτες γραφικών, το σύστημα πρέπει επιπλέον να χωρίσει τα δεδομένα, έχοντας επικαλυπτομένα τμήματα ανάλογα με τον αριθμό βημάτων που γίνονται σε κάθε κάρτα πριν επικοινωνήσουν

μεταξύ τους. Για τον υπολογισμό στένσιλ μέσω συνέλιξης στο πεδίο της συχνότητας γίνονται παρόμοιες διαδικασίες με αυτές για τη μία κάρτα γραφικών. Η επιλογή για FFT προτείνεται από το σύστημα και η τελική επιλογή γίνεται από τον χρήστη.

5.4 Βελτιστοποιήσεις

5.4.1 Καθολική και Μεριζόμενη μνήμη

Οι μέθοδοι βελτιστοποίησης που εφαρμόζονται στη συνάρτηση πυρήνα περιγράφονται στην προηγούμενη παράγραφο. Ως πρότυπο χρησιμοποιήθηκε η τροποποιημένη υλοποίηση της NVIDIA από τον tlMo κ.α και επομένως οι ίδιες τεχνικές χρησιμοποιούνται για την καλύτερη αξιοποίηση της μνήμης. Η αποδοτικότητα της καθολικής μνήμης είναι όσο πιο ψηλά γίνεται, ενώ η απόδοση της μεριζόμενης μνήμης είναι 100%.

5.4.2 Ξετύλιγμα βρόγχων επανάληψης

Οι αριθμητικές πράξεις στους υπολογισμούς στένσιλ μπορούν να γίνουν με επαναληπτικό τρόπο κατά μήκος της διαμέτρου, όπως γίνεται με τον τελεστή αθροίσματος στο παράδειγμα της λαπλασιανής. Υπάρχουν στένσιλ που δεν εμφανίζουν περιοδική δομή και επομένως δε μπορούν να υπολογιστούν με τη βοήθεια επαναληπτικού βρόγχου. Αυτός είναι καθοριστικός λόγος για τον οποίο οι επαναληπτικοί βρόγχοι ξετυλίγονται τελείως.

Τμήμα Κώδικα 5.2: .

```
@inbounds temp = tile[txr + -1, tyr + 0]
current += cfarr[1] * temp
@inbounds temp = tile[txr + 0, tyr + -1]
current += cfarr[1] * temp
@inbounds temp = tile[txr + 0, tyr + 0]
current += temp
infront1 += cfarr[1] * temp
behind1 += cfarr[1] * temp
@inbounds temp = tile[txr + 0, tyr + 1]
current += cfarr[1] * temp
@inbounds temp = tile[txr + 1, tyr + 0]
current += cfarr[1] * temp
```

Αυτό το επιθετικό ξετύλιγμα βρόγχων επανάληψης έχει πλεονεκτήματα και μειονεκτήματα [17]. Στα πλεονεκτήματα συγκαταλέγονται μειωμένος αριθμός εντολών, λόγω της αφαίρεσης εντολών ελέγχου στους βρόγχους επανάληψης, αυξημένη δυνατότητα παραλληλισμού σε επίπεδο εντολών (Instruction Level Parallelism) και πιο αποδοτική χρήση μεριζόμενης μνήμης. Στο παραπάνω παράδειγμα, σε τρεις από τις αριθμητικές πράξεις χρησιμοποιείται το ίδιο στοιχείο από τη μεριζόμενη μνήμη. Άμα χρησιμοποιούνταν βρόγχοι επανάληψης, οι τρεις αυτές πράξεις δε θα μπορούσαν να είναι συνεχόμενες και το ίδιο στοιχείο θα διαβάζονταν τρεις φορές. Όσον αφορά τα μειονεκτήματα, αυξάνεται η πίεση στους καταχωρητές, αφού κάθε αποτέλεσμα χρειάζεται ξεχωριστό καταχωρητή, με αποτέλεσμα να μην αξιοποιούνται αποδοτικά οι πόροι της κάρτας γραφικών.

5.4.3 Ομαδοποίηση Βημάτων Χρόνου

Ο τρόπος με τον οποίο προηγούμενες έρευνες υλοποιούν την ομαδοποίηση βημάτων χρόνου, είναι να κάνουν πολλαπλές επαναλήψεις στα δεδομένου που έχουν διαβαστεί σε κρυφή/γρήγορη μνήμη για να υπολογίσουν πολλαπλά βήματα χρόνου. Αυτός ο τρόπος δε μπορεί να εφαρμοστεί αποδοτικά σε τρισδιάστατα στένσιλ και καθόλου σε μη ολιστικές προσεγγίσεις. Η βασική ιδέα, που εφαρμόζεται σε αυτό το framework, είναι να δημιουργηθεί ένα καινούριο στένσιλ, που όταν εφαρμοστεί μία φορά, θα έχει το ίδιο αποτέλεσμα με το να εφαρμοστεί το αρχικό στένσιλ περισσότερες φορές.

Η εξίσωση για το δισδιάστατο στένσιλ-αστέρι είναι:

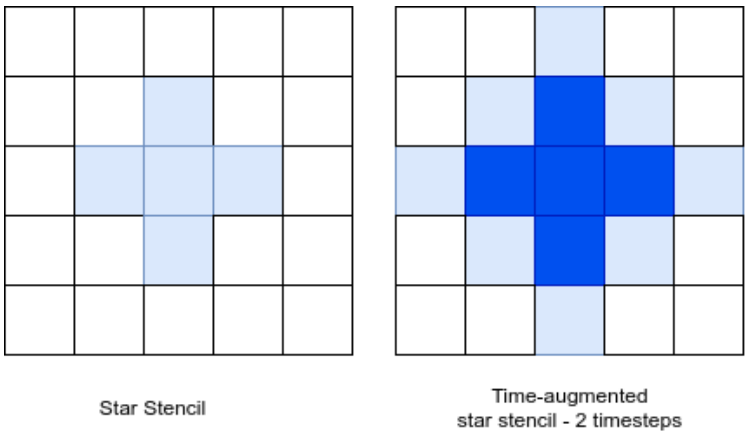
$$D^{t+1}[x,y] = c[0]D^{t}[x,y] + \sum_{i=1}^{n} c[i]D^{t}[x+i,y] + D^{t}[x-i,y] + D^{t}[x,y+i] + D^{t}[x,y-i]) \tag{5.1}$$

Για το επόμενο βήμα χρόνου, t αντικαθίσταται με t+1

$$D^{t+2}[x,y] = c[0]D^{t+1}[x,y] + \sum_{i=1}^{n} c[i]D^{t+1}[x+i,y] + D^{t+1}[x-i,y] + D^{t+1}[x,y+i] + D^{t+1}[x,y-i]) \tag{5.2}$$

Αν αντικαταστήσουμε την 5.1 στην 5.2, βρίσκουμε την τιμή D^{t+2}, χρησιμοποιώντας μόνο τις τιμές D^{t}. Αυτή η διαδικασία μπορεί να εφαρμοστεί m φορές για να βρεθούν οι τιμές D^{t+m}. Το καινούριο στένσιλ θα έχει μέγιστη διάμετρο $m * n$ και οι παραπάνω αριθμητικές πράξεις εξαρτώνται από το σχήμα του στένσιλ.

Τα ίδια ισχύουν και για τα τρισδιάστατα στένσιλ. Με τη μη ολιστική προσέγγιση, η επιρροή της ομαδοποίησης βημάτων χρόνου στη μεριζόμενη μνήμη είναι ίδια με αυτή στα δισδιάστατα στένσιλ, με τη διαφορά ότι χρειάζονται επιπλέον καταχωρητές. Και με αυτήν την προσέγγιση γίνονται περιττοί υπολογισμοί. Οι περιττοί υπολογισμοί δε βρίσκονται στην ανανέωση τιμών του στεφανιού, αλλά σαν επιπλέον πράξεις για κάθε στοιχείο. Ένα δισδιάστατο στένσιλ αστέρι χρειάζεται $2 * 5 = 10$ πράξεις για δύο βήματα χρόνου με επανάληψη και 11 με ομαδοποίηση βημάτων χρόνου. Για τρισδιάστατο αστέρι, από $2 * 9 = 18$ πηγαίνει σε 25 πράξεις.



Σχήμα 5.2: Δισδιάστατο στένσιλ αστέρι με ομαδοποίηση βημάτων χρόνου.

5.5 Πολλαπλές Κάρτες Γραφικών

Για την αξιοποίηση πολλαπλών καρτών γραφικών πρέπει να λυθούν τρία προβλήματα.

- Πώς θα διαχωριστούν τα δεδομένα.
- Πώς θα γίνει η επικοινωνία του ανανεωμένου επικαλυπτόμενου τμήματος.
- Πόσα βήματα χρόνου θα γίνουν σε κάθε κάρτα πριν επικοινωνήσουν μεταξύ τους

Η απάντηση στο πρώτο πρόβλημα είναι να χωριστούν τα δεδομένα κατά τον z άξονα, δηλαδή κατά τον άξονα που μεταβάλλεται αργότερα καθώς κινούμαστε στα δεδομένα στη φυσική μνήμη. Το πλεονέκτημα αυτής της προσέγγισης είναι η απλότητά της και το γεγόνος ότι κάθε τμήμα θα είναι συνεχόμενο στη μνήμη. Αν υπάρχουν διαφορετικές σε απόδοση κάρτες γραφικών, τα δεδομένα θα διασπαστούν αναλόγως.

Η παραπάνω προσέγγιση βοηθάει και στο δεύτερο πρόβλημα, της επικοινωνίας. Τα επικαλυπτόμενα τμήματα θα είναι και αυτά συνεχόμενα στην φυσική μνήμη, και επομένως η μεταφορά τους απο τη μία κάρτα στην άλλη θα είναι εύκολη και γρήγορη, είτε έχουμε απευθείας μεταφόρα μεταξύ των καρτών, είτε μέσω του επεξεργαστή, είτε αν έχουμε διανεμημένο σύστημα. Ο λόγος που χρειάζονται επικαλυπτόμενα τμήματα οφείλεται στο ¨στεφάνι¨ που έχει αναφερθεί προηγουμένως. Στη συγκεκριμένη περίπτωση, χρησιμοποιείται απευθείας επικοινωνία μεταξύ καρτών (peer to peer).

Σχήμα 5.3: Οπτικοποίηση διάσπασης δεδομένων σε πολλαπλές κάρτες γραφικών.

Το τρίτο πρόβλημα σκοπεύει στη μείωση του σταθερού κόστους επικοινωνίας μειώνοντας τον αριθμό τους, πραγματοποιώντας περισσότερα από ένα βήματα χρόνου πριν οι κάρτες γραφικών επικοινωνήσουν. Όσο περισσότερα είναι αυτά τα βήματα χρόνου, αυξάνεται η χρήση μνήμης και ο αριθμός των περιττών πράξεων. Για να βελτιωθεί η απόδοση πρέπει οι περιττοί υπολογισμοί να κοστίζουν λιγότερο από το σταθερό κόστος επικοινωνίας. Επομένως, αυτή η μέθοδος εξαρτάται άμεσα από την απόδοση της κάθε κάρτας γραφικών.

Τελικά η πορεία που ακολουθείται είναι:

1. Διαχωρισμός δεδομένων και μεταφορά στις κάρτες γραφικών
2. Εκτέλεση ενός ή περισσότερων βημάτων χρόνου
3. Ανταλλαγή επικαλυπτόμενου τμήματος δεδομένων σε κάρτες που έχουν γειτονικά μέρη.

5.6 Υπολογισμός στένσιλ με FFT

Όσο αυξάνεται το μέγεθος, ακτίνα και πυκνότητα του στένσιλ, αυξάνονται οι απαιτήσεις σε πόρους και ο χρόνος εκτέλεσης. Μεγάλα στένσιλ χρειάζονται περισσότερους καταχωρητές, περισσότερη μεριζόμενη μνήμη και δε χρησιμοποιούνται αποδοτικά όλοι οι πόροι της κάρτας γραφικών. Για ακόμα μεγαλύτερα μεγέθη (ακτίνα $\geq$ 32 στοιχεία) οι πόροι δεν αρκούν για να εκτελεστεί ο υπολογισμός του στένσιλ. Μια επιλογή θα ήταν η διάσπαση του στένσιλ σε διαφορετικές συναρτήσεις πυρήνα, αλλά μέσα από εκτελέσεις αποδείχθηκε ότι είναι τουλάχιστον 4 φορές πιο αργό. Παρόλου που το στένσιλ διασπάται στη μέση, οι απαιτήσεις σε μνήμη δεν γίνονται μισές και απαιτείται συγχρονισμός διαφορετικών συναρτήσεων-πυρήνα, που γίνεται σε επίπεδο επεξεργαστή.

Μία προσέγγιση που μπορεί να λύσει το πρόβλημα των μεγάλων στένσιλ, είναι ο υπολογισμός του στην συχνότητα μέσω FFT. Ένα απλό στένσιλ είναι παρόμοιο με τη συνέλιξη, με τη διαφορά να βρίσκεται πως σε συνήθεις εφαρμογές στένσιλ εφαρμόζεται πολλές φορές στα δεδομένα σε σχέση με την συνέλιξη. Εφαρμογές σε επεξεργασία ψηφιακής εικόνας και σήματος, όπως υπολογισμός της συσχέτισης ή εφαρμογή φίλτρου Gauss [18].

Για παράδειγμα, ο τύπος της δισδιάστατης συνέλιξης $f(x,y) = \sum_i \sum_j g(i,j) * h[x-i, y-j]$ μετατρέπεται σε περιγραφή στένσιλ αν θεωρήσουμε τη συνάρτηση h τα δεδομένα εισόδου την χρονική στιγμή t, τη συνάρτηση f ως τα δεδομένα εισόδο την χρονική στιγμή $t+1$ και την g ως γειτονιά του στένσιλ.

Το FLCC (Fast Library for Convolution and Correlation [19] είναι μία βιβλιοθήκη που στοχεύει στην επιτάχυνση του υπολογισμού της συνέλιξης και των τοπικών συντελεστών συσχέτισης με τη βοήθεια καρτών γραφικών. Συγκεκριμένα για τη συνέλιξη, παρουσιάζεται μια υλοποίηση που χρησιμοποιεί FFT και αναλύονται τα μεγέθη των δεδομένων για τα οποία συμφέρει ο FFT από την επαναληπτική μέθοδο. Για μεγάλες εικόνες και για πρότυπο με πλάτος μεγαλύτερο από 16 πίξελ, η επιτάχυνση αγγίζει το $\times 10$. Η ίδια προσέγγιση μπορεί να χρσηιμοποιηθεί όχι μόνο για μεγάλα στένσιλ, αλλά και για αυτά που παράγονται από την ομαδοποίηση βημάτων χρόνου.

Η υλοποίηση περιλαμβάνει μεταφορά των δεδομένων και του στένσιλ στο πεδίο της συχνότητας, πολλαπλασιασμός στοιχείο με στοιχείο και επαναφορά του αποτελέσματος στο πεδίο του χρόνου. Όλα τα βήματα αυτά λαμβάνουν μέρος στην κάρτα γραφικών, οπότε η μεταφορά δεδομένων από τον επεξεργαστή στην κάρτα και αντίστροφα είναι το ίδιο με την περίπτωση της επαναληπτικής υλοποίησης σε μία κάρτα γραφικών.

Επιπροσθέτως, αξιοποιείται η ομαδοποίηση βημάτων χρόνου για να υπολογιστούν περισσότερα βήματα χρόνου με μία εφαρμογή της παραπάνω διαδικασίας. Παρόλο που το καινούριο στένσιλ έχει πολλαπλάσια ακτίνα, οι επιπλέον υπολογισμοί που απαιτεί ο FFT δεν αυξάνονται τόσο γρήγορα, καθώς η ακτίνα του στένσιλ απλά προστίθεται στο μέγεθος των δεδομένων εισόδου. Επίσης το καινούριο στένσιλ έχει μεγαλύτερη

πυκνότητα, το οποίο όμως δεν επηρεάζει την ταχύτητα αφού δεν επηρεάζεται από τον αριθμό των μηδενικών η διαδικασία μετατροπής δεδομένων στη συχνότητα ή ο πολλαπλασιασμός στοιχείων.

Η μαθηματική ανάλυση της εφαρμογής συνέλιξης στο πεδίο της συχνότητας είναι γνωστή και περιγράφεται αναλυτικά στη διπλωματικη διατριβή όπου και υλοποιείται αναλυτικά η βιβλιοθήκη FLCC [20].

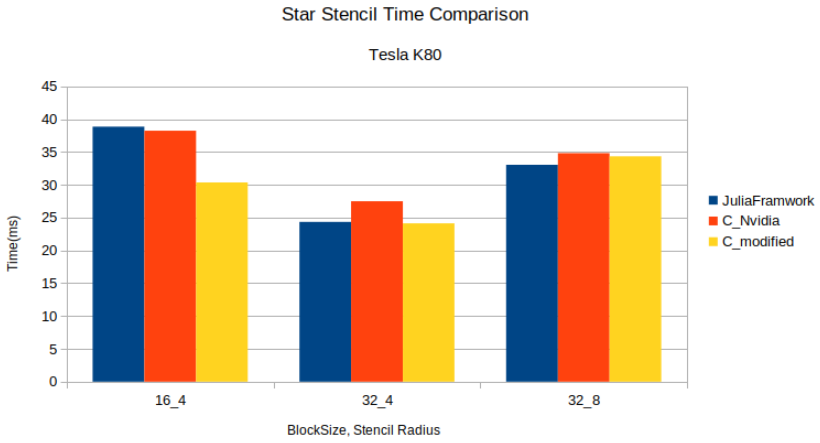
Κεφάλαιο 6

Άποτελέσματα

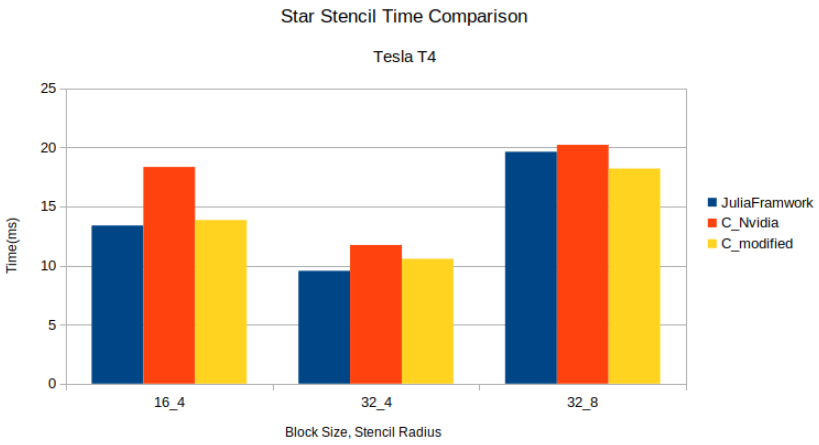
6.1 Αποτελέσματα απόδοσης

Οι μετρήσεις που συλλέχθηκαν για την απόδοση το framework πρέπει να συγκριθούν με υπάρχουσες υλοποιήσεις που είναι αποδεδειγμένα υψηλής απόδοσης. Για αυτόν τον σκοπό, επιλέχθηκε η υλοποίηση της NVIDIA, για την οποία υπάρχει δημοσιευμένος κώδικας στην έρευνα. Επίσης, χρησιμοποιείται και η τροποποιημένη υλοποίηση της NVIDIA, η οποία υλοποιήθηκε από το μηδέν σε C καθώς ο Mo κ.α δεν παρέχουν κώδικα στη δημοσίευσή τους.

Εφόσον η συνάρτηση πυρήνα της NVIDIA για πεπερασμένες διαφορές μπορεί να υπολογίσει μόνο στένσιλ τύπου αστέρι, όλες οι συγκρίσεις έγιναν για αυτόν τον τύπο. Οι μετρήσεις έγιναν σε κάρτες γραφικών Tesla T4 και Tesla K80 για διάφορα μεγέθη ακτίνας στένσιλ και ομάδων νημάτων.



Σχήμα 6.1: Σύγκριση της Julia με τη C για στένσιλ αστέρι.



Σχήμα 6.2: Σύγκριση της Julia με τη C για στένσιλ αστέρι.

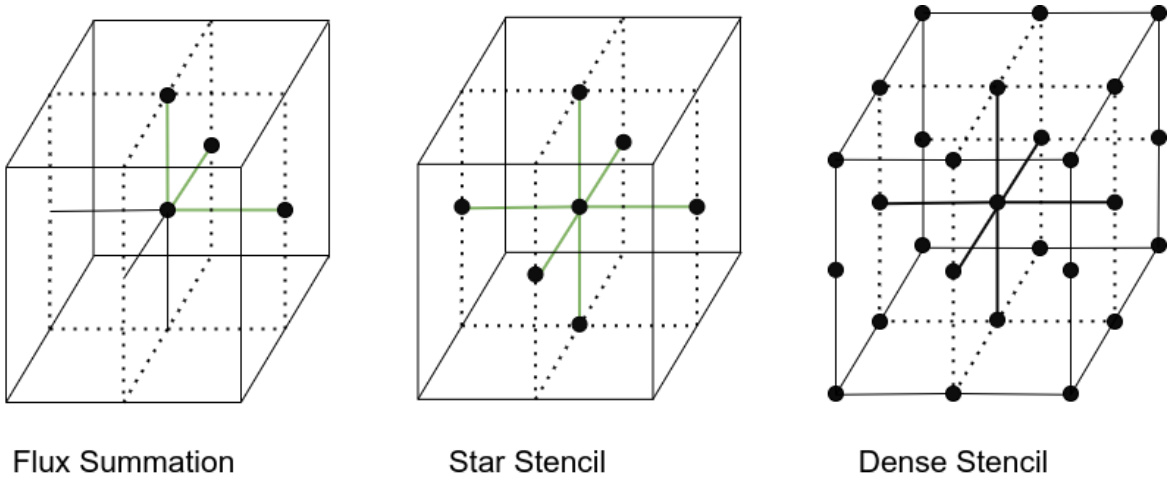
Για ομάδα νημάτων 32×32 η υλοποίηση σε Julia είναι πάντα πιο γρήγορη από την υλοποίηση της NVIDIA και ορισμένες φορές πιο γρήγορη από την τροποποιημένη υλοποίηση γραμμένη σε C. Τα αποτελέσματα δείχνουν ότι επιλογή της συνάρτησης-πυρηνά,που χρησιμοποιήθηκε σαν πρότυπο ήταν σωστή. Ακόμα, φαίνεται ότι η απόδοση της Julia σαν γλώσσα είναι καλή και ότι η παραγωγή κώδικα δουλεύει σωστά.

Όπως και στο 3ο κεφάλαιο, οι μετρικές της κάρτας γραφικών είναι ολόιδιες για ίδιες υλοποιήσεις. Η συνάρτηση πυρήνα που παράγει το framework εκμεταλλεύεται στο 100% τη ρυθμαπόδοση της μεριζόμενης μνήμης και στο μέγιστο δυνατό τη ρυθμαπόδοση της καθολικής.

Έχοντας εδραιώσει τη σωστή λειτουργία και καλή απόδοση του framework για τη βασική περίπτωση, ακολουθεί η αξιολόγηση των μεθόδων βελτιστοποίησης πάνω σε αυτή.

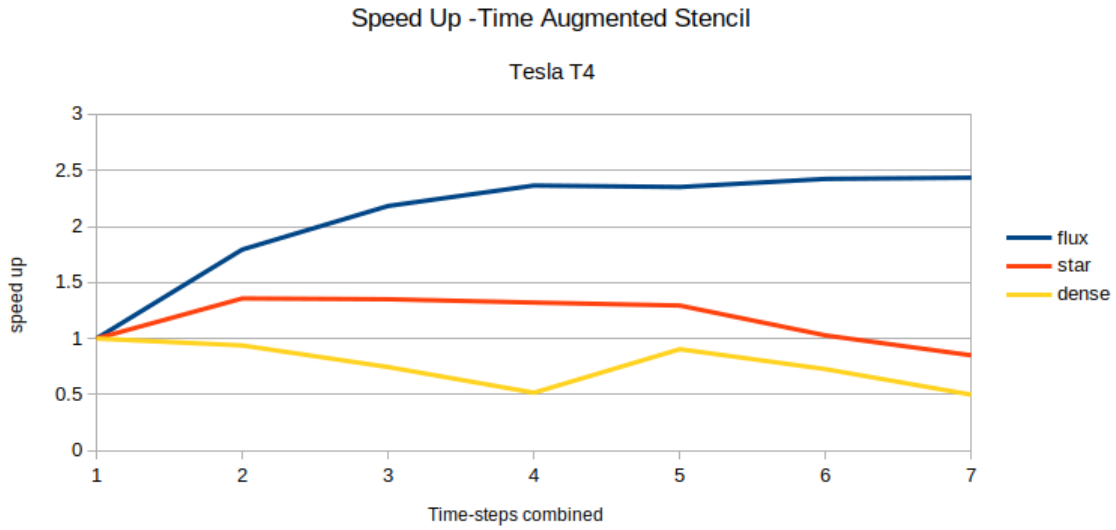
6.1.1 Ομαδοποίηση στον χρόνο

Οι βασικοί παράγοντες που επηρεάζουν την απόδοση αυτής της μεθόδου βελτιστοποίησης είναι η πυκνότητα του στένσιλ, η αρχική του ακτίνα και ο αριθμός τον βημάτων χρόνου που ομαδοποιούνται. Τρεις τύποι στένσιλ επιλέχθηκαν, οι οποίοι έχουν διαφορετική πυκνότητα, άθροισμα ροών(flux summation) [], αστέρι και πυκνό στένσιλ, από μικρότερη σε μεγαλύτερη πυκνότητα.

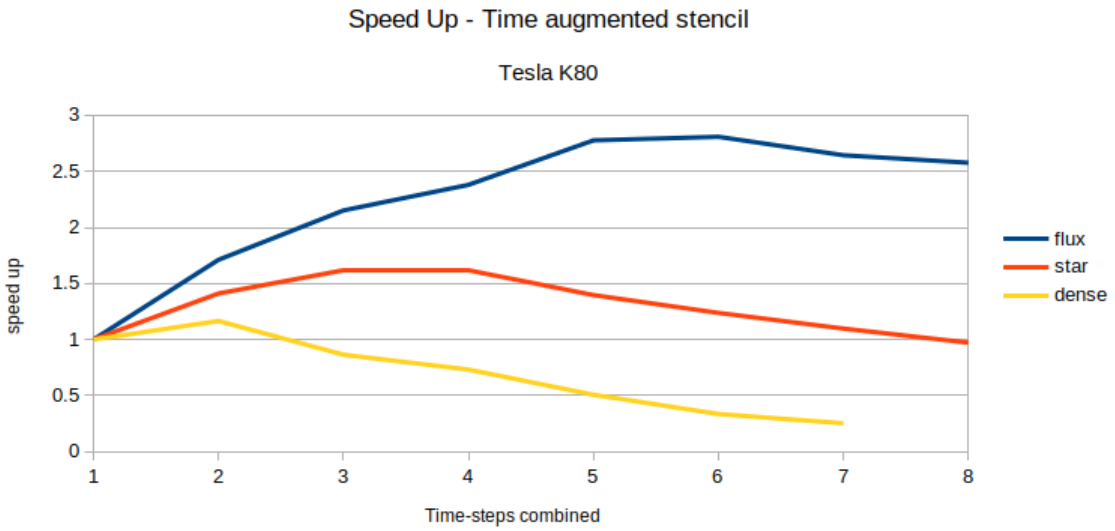


Σχήμα 6.3: 3 τύπου στένσιλ με διαφορετικές πυκνότητες. 1) Άθροισμα ροών 2) Αστέρι 3) Πυκνό

Οι εκτελέσεις για μετρήσεις έγιναν στις κάρτες γραφικών Tesla T4 και Tesla K80. Τα γραφήματα στο σχήμα 6.3 δείχνουν την επιτάχυνση που επιφέρει η ομαδοποίηση βημάτων χρόνου σε σχέση με τον χρόνο που χρειάζεται το αρχικό στένσιλ για τα ίδια βήματα. Σε όλες τις περιπτώσεις τα αρχικά στένσιλ έχουν ακτίνα ένα, όπως και στο σχήμα 6.2.



Σχήμα 6.4: Επιτάχυνση με ομαδοποίηση βημάτων χρόνου.



Σχήμα 6.5: Επιτάχυνση με ομαδοποίηση βημάτων χρόνου.

Από τις μέτρησεις φαίνεται πως το πυκνό στένσιλ δεν επωφελείται από την ομαδοποίηση βημάτων χρόνου. Οι εκτελέσεις με ομαδοποίηση είναι πιο αργές από το αρχικό στένσιλ, εκτός από μία περίπτωση που πετυχαίνει 1.16× επιτάχυνση, για δύο βήματα χρόνου ομαδοποιημένα.

Στο άλλο άκρο, όσον αφορά την πυκνότητα, το στένσιλ αθροίσματος ροών έχει την καλύτερη επιτάχυνση με ομαδοποίηση βημάτων χρόνου σε σχέση με τα άλλα δύο. Για κάθε αριθμό βημάτων χρόνου υπάρχει θετική επιτάχυνση, η οποία αγγίζει το 2.8× και φαίνεται ότι υπάρχει προοπτική για επιτάχυνση και σε περισσότερα από 8 βήματα χρόνου.

Το ενδιαφέρον όμως επικεντρώνεται κυρίως στο στένσιλ αστέρι, που έχει ενδιάμεση πυκνότητα από τα άλλα δύο και χρησιμοποιείται πιο συχνά, όπως στις πεπερασμένες διαφορές και σε κυτταρικά αυτόματα. Η καλύτερη επιτάχυνση είναι 1.65× για την Tesla K80 και 1.45× για την Tesla T4. Μέχρι 5 βήματα χρόνου η επιτάχυνση αυξάνεται, με την καλύτερη να βρίσκεται για 4 βήματα. Μετά από 5 βήματα χρόνου η επιτάχυνση μειώνεται ενώ για 7 είναι αρνητική.

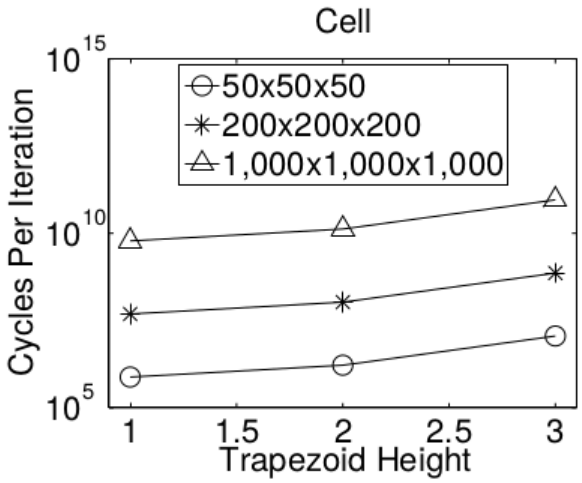
Για στένσιλ αστέρι με ακτίνα 2, η μέγιστη επιτάχυνση είναι 1.15× για 2 βήματα χρόνου και καθόλου για τα υπόλοιπα. Για μεγαλύτερη ακτίνα δεν υπάρχει καθόλου επιτάχυνση καθώς οι πόροι που απαιτούνται για μνήμη είναι πολλοί και δεν επιτρέπουν την αποδοτική χρήση των υπόλοιπων πόρων της κάρτας γραφικών.

Για να αξιολογηθεί το πόσο καλή ή όχι είναι η επιτάχυνση της ομαδοποίησης βημάτων χρόνου που χρησιμοποιεί το framework, θα συγκριθεί με υπάρχουσες έρευνες για ομαδόποιηση βημάτων χρόνου με επαναληπτική προσέγγιση. Ο Meng κ.α. [1] μελετούν την επιρροή του μεγέθους του στεφανιού στην απόδοση, το οποίο αυξάνεται με τον αριθμο βημάτων χρόνου που ομαδοποιούνται, ενώ ο Holewinski κ.α. [2] χρησιμοποιούν τη μέθοδο της ομαδοποίησης στο αυτοματοποιημένο framework που παρουσιάζουν. Οι δύο έρευνες επικεντρώνονται σε δισδιάστατα στένσιλ, αλλά παρουσιάζουν την εφαρμογή της μεθόδου και σε τρισδιάστατα.

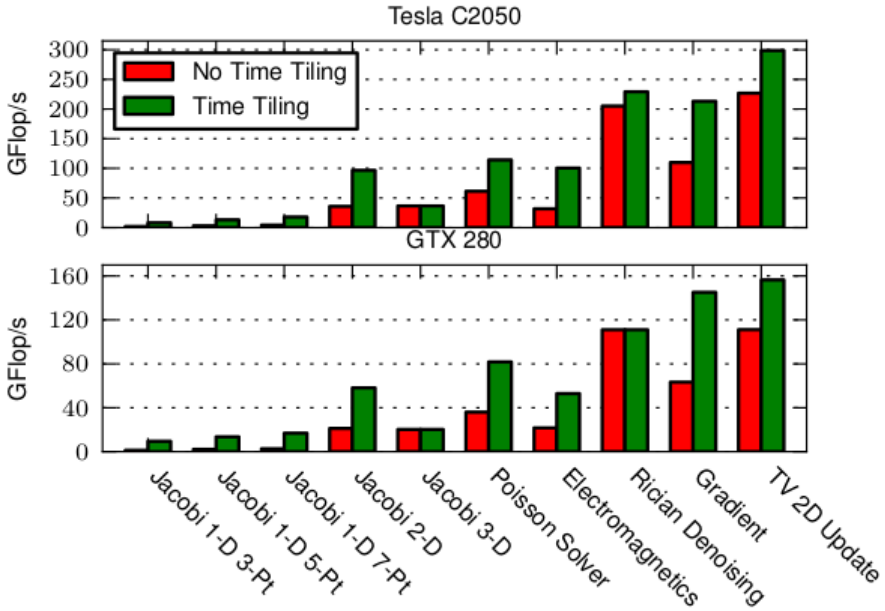
Το τρισδιάστατο στένσιλ που χρησιμοποιεί ο Meng κ.α. προέρχεται από τα κυτταρικά

αυτόματα και χρησιμοποιείται από το Παιχνίδι της Ζωής του Conway. Είναι ολόιδιο με το στένσιλ αστέρι ακτινάς 1, με τη μόνη διαφορά να βρίσκεται πως αντί για προσθέσεις έχει και λογικές πράξεις. Η σύγκριση είναι έγκυρη αν μείνει στην επιτάχυνση και όχι στους απόλυτους χρόνους εκτέλεσης.

Ο Holewinski κ.α. χρησιμοποιούν το τρισδιάστατο στένσιλ αστέρι ακτίνας 1 για τη μέθοδο Jacobi.



Σχήμα 6.6: Meng κ.α. [1] Ομαδοποίηση βημάτων χρόνου για διάφορα μεγέθη δεδομένων εισόδου. Το ύψος το τραπεζίου συμβολίζει τον αριθμό βημάτων χρόνου που ομαδοποιούνται.

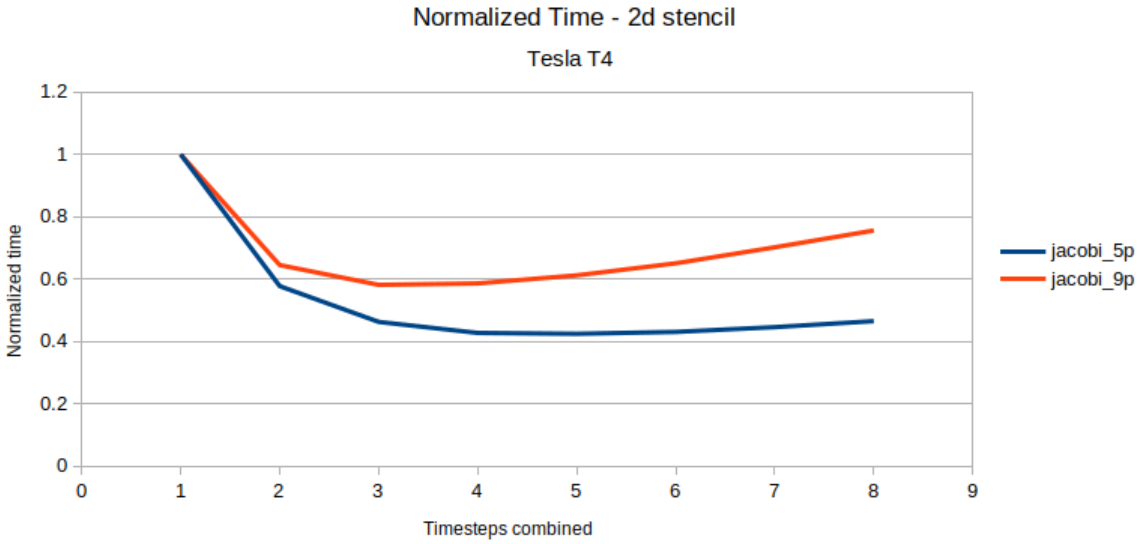


Σχήμα 6.7: Holewinski κ.α. [2] Απόδοση σε διαφορετικές κάρτες γραφικών, με και χωρίς ομαδοποίηση βημάτων χρόνου. Μόνο το Jacobi 3D είναι τρισδιάστατο.

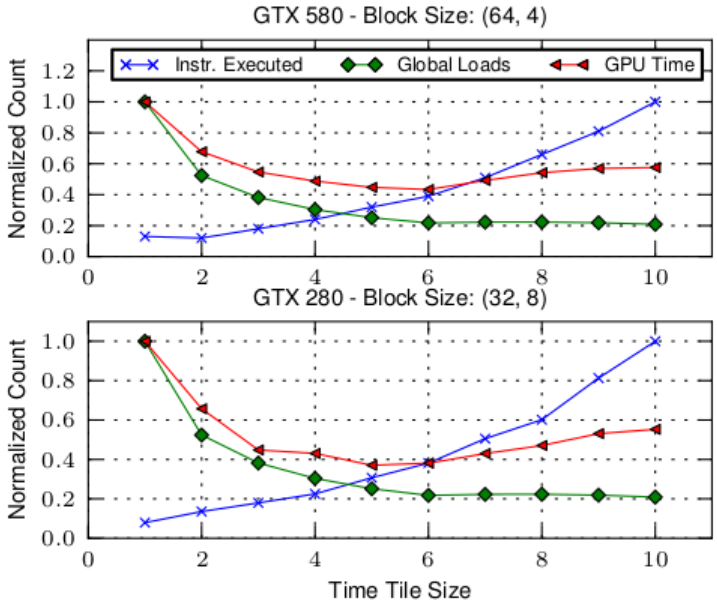
Είναι προφανές πως για τα τρισδιάστατα στένσιλ δεν επιτυγχάνεται καμία επιτάγχυνση. Μάλιστα, για το κυτταρικό αυτόματο η απόδοση μειώνεται αρκετά με την εφαρμογή της

ομαδοποίησης βημάτων χρόνου. Συγκρίνοντας την απόδοση του framework, που έχει διαφορετική προσέγγιση για την ομαδοποίηση βημάτων χρόνου, με τις δύο έρευνες, η 1.5× επιτάχυνση φαίνεται μεγάλη βελτίωση και ενθαρρύνω περαιτέρω έρευνα και για προσαρμογή για μεγαλύτερα στένσιλ.

Παρόλο που το framework δημιουργήθηκε έχοντας υπόψην κυρίως τα τρισδιάστατα στένσιλ, είναι πολύ εύκολο να εκτελεστούν δισδιάστατα στένσιλ και να εφαρμοστεί η ομαδιοποίηση σε αυτά. Στις δύο παραπάνω έρευνες, η προηγούμενη προσέγγιση ομαδοποίησης βημάτων χρόνου είναι επιτυχημένη για δισδιάστατα στένσιλ. Για την σύγκριση στο δισδιάστατο κομμάτι θα χρησιμοποιηθεί το στένσιλ αστερί με ακτίνα 1 (5 σημεία) και 2 (9 σημεία), το οποίο εφαρμόζεται στη δισδιάστατη μέθοδο Jacobi και χρησιμοποιείται στις μετρήσεις του Holewinski κ.α.



Σχήμα 6.8: Ομαδοποίηση βημάτων χρόνου σε στένσιλ αστέρι.



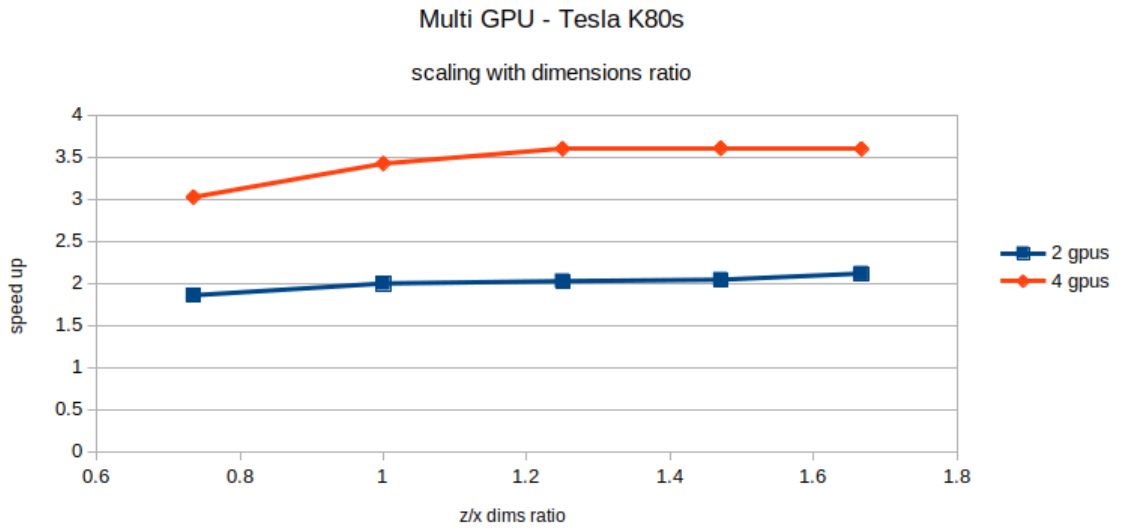
Σχήμα 6.9: Holewinski κ.α. [2] Ομαδοποίηση βημάτων χρόνου για δισδιάστατο Jacobi

Ο Holewinski κ.α καταφέρνουν την καλύτερη επιτάχυνση για 6 βήματα χρόνου, με χρόνο 0.4 επί του αρχικού. Αντίστοιχα, το framework σε Julia έχει την καλύτερη επιτάχυνση στο 5ο βήμα χρόνου, με χρόνο λίγο 0.43 επί του αρχικού. Το γεγονός ότι η καλύτερη επιτάχυνση βρίσκεται για διαφορετικό βήμα χρόνου δεν έχει σημασία για την τελική απόδοση. Η διαφορά σε επιτάχυνση μεταξύ των δύο προσεγγίσεων ομαδοποίησης χρόνου είναι πολύ μικρή και αποδεικνύει την αποτελεσματικότητα της νέας μεθόδου.

6.1.2 Πολλαπλές Κάρτες Γραφικών

Οι εκτελέσεις για την μέτρηση της απόδοσης υπολογισμών στένσιλ σε πολλαπλές κάρτες γραφικών έγιναν σε 4 Tesla K80. Επιπλέον μελετήθηκε αν η εκτέλεση περισσότερων βημάτων χρόνου πριν την ανταλλάγη δεδομένων μεταξύ των καρτών γραφικών βελτιώνει την απόδοση. Αυτό είχε προταθεί από τον Micikevicius [8] και υποψιαζόταν πως θα υπάρχει βελτίωση για μικρά στένσιλ.

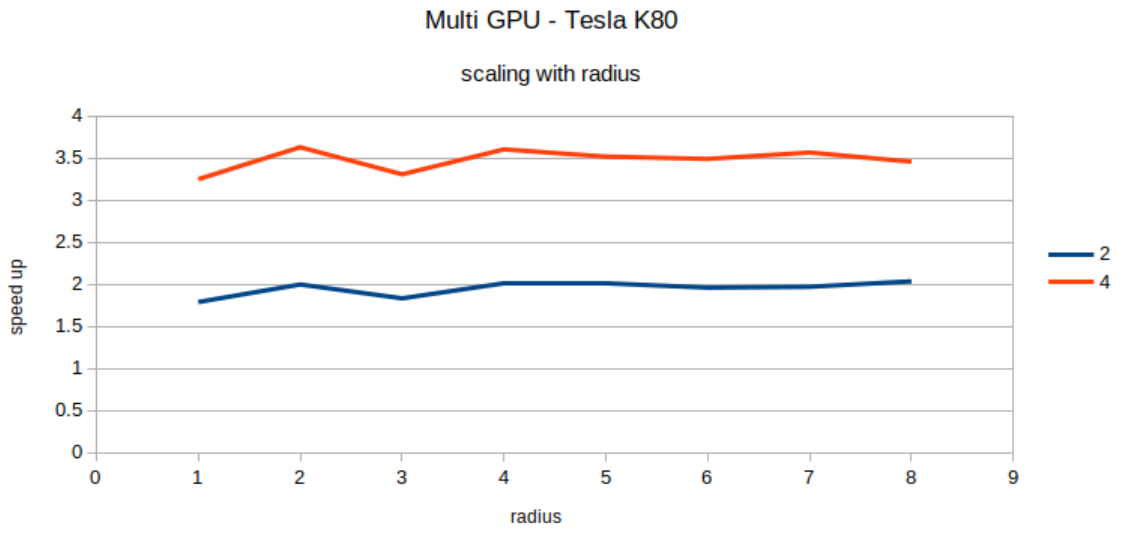
Στο επόμενο γραφημα φαίνεται απόδοση σε πολλαπλές κάρτες γραφικών για διαφορετικά μεγέθη δεδομένων εισόδου με διαφορετική αναλογία στο μήκος των διαστάσεων.



Σχήμα 6.10: Πορεία της επιτάχυνσης για διαφορετική αναλογία στο μήκος των διαστάσεων. Στένσιλ αστέρι με ακτίνα 8

Στο γράφημα παρατηρείται μια ελαφριά τάση για μεγαλύτερη επιτάχυνση όσο μεγαλώνει και το μήκος του z αξόνα σε σχέση με τους x y. Αυτό ήταν αναμενόμενο, καθώς όσο μεγαλύτερα είναι τα μήκη των διαστάσεων x y τόσο μεγαλύτερα τμήματα μνήμης χρειάζεται να μεταφερθούν σε σχέση με τους υπολογισμούς. Βέβαια η διαφορά είναι αρκετά μικρή και δεν αποτελεί προτεραιότητα να αλλάξει ο τρόπος με τον οποίο χωρίζονται τα δεδομένα στις κάρτες γραφικών.

Στη συνέχεια μελετάται η πορεία της επιτάχυνσης για διαφορετικές ακτίνες.



Σχήμα 6.11: Επιτάχυνση για διαφορετικές ακτίνες. Στένσιλ αστέρι

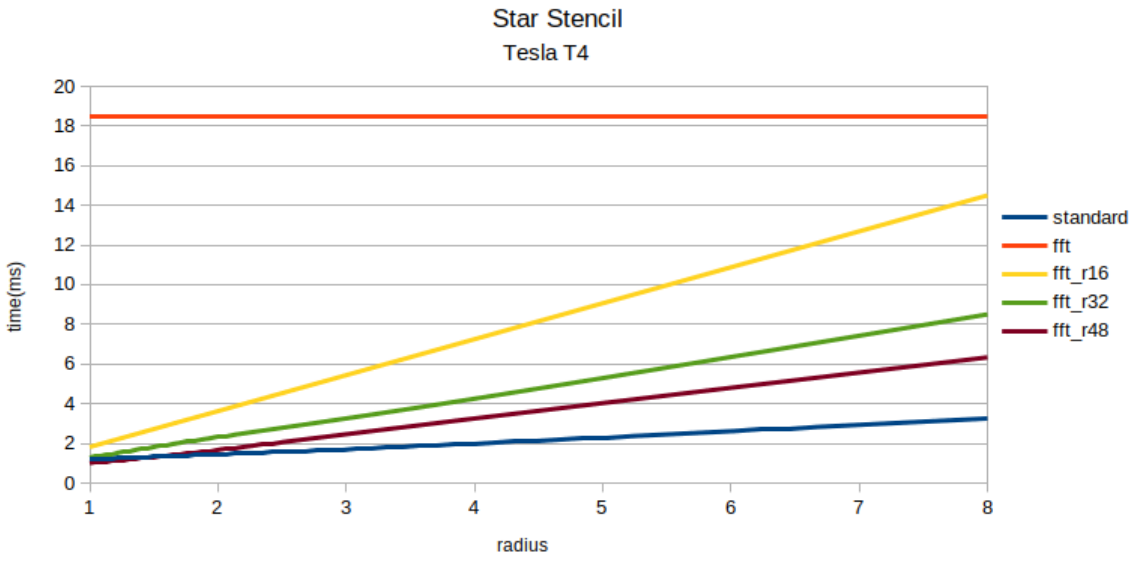
Η πορεία της επιτάχυνσης σε σχέση με την ακτίνα είναι σχετικά σταθερή. Για 2 κάρτες γραφικών η επιτάχυνση είναι σταθερή 2× και για 4 κάρτες είναι ομοίως σταθερή 3.5×.

Η μη-γραμμική αύξηση της επιτάχυνσης σε σχέση με τον αριθμό των καρτών γραφικών οφείλεται στις επικοινωνίες μεταξύ των καρτών όπου υπάρχουν αδρανείς κάρτες που περιμένουν να ολοκληρωθεί η μεταφορά δεδομένων στις άλλες, κάτι το οποίο δεν εμφανίζεται για 2 κάρτες γραφικών.

Η τακτική να πραγματοποιούνται περισσότερα βήματα χρόνου πριν τη μεταφορά δεδομένων μεταξύ των καρτών δεν ήταν αποτελεσματική. Η καλύτερη επιτάχυνση που παρατηρήθηκε ήταν αμελητέα, 1.06× . Για μεγάλες ακτίνες και πολλά βήματα χρόνου δεν υπήρχε καμία βελτίωση ή υπήρχε επιβράδυνση 2×. Όσο περισσότερα βήματα χρόνου γίνονται σε κάθε κάρτα πριν την επικοινωνία, τόσο μεγαλύτερο είναι το επικαλυπτόμενο τμήμα που πρέπει να υπολογιστεί και να μεταφερθεί. Έτσι, παρόλο που το σταθερό κόστος επικοινωνίας μειώθηκε πραγματοποιώντας λιγότερες σε αριθμό επικοινωνίες, ο όγκος των δεδομένων που μεταφέρεται παραμένει ίδιος και αυξάνονται τα δεδομένα στα οποία γίνονται υπολογισμοί. Το σταθερό κόστος επικοινωνίας είναι πολύ μικρό σε σχέση με τους επιπλέον υπολογισμούς, οπότε δεν παρατηρείται κάποια επιτάχυνση με αυτή τη μέθοδο.

6.1.3 Στένσιλ με FFT

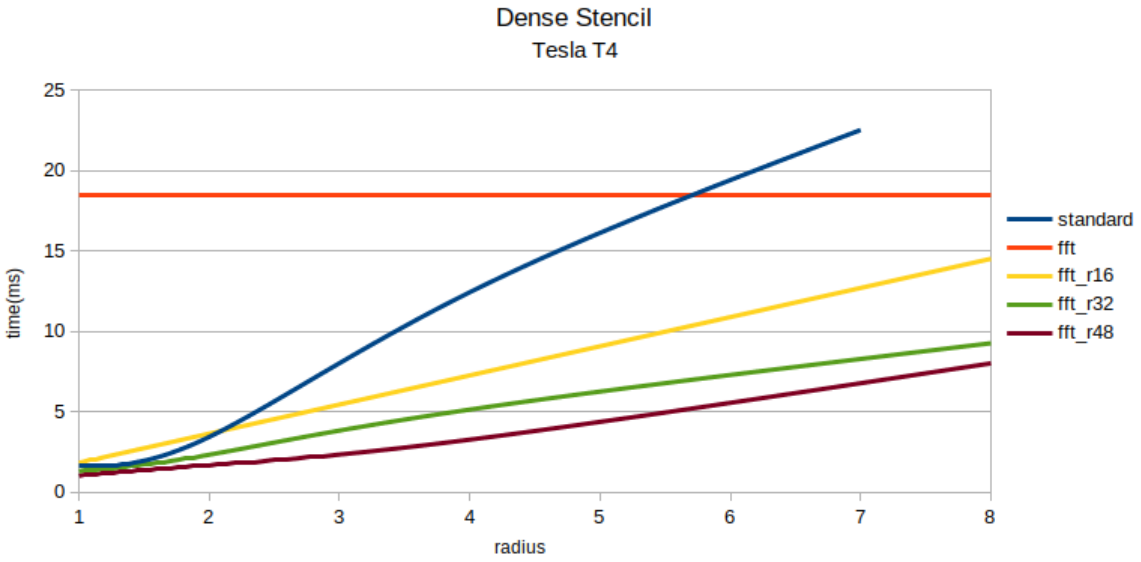
Αρχικά, γίνεται σύγκριση της επαναληπτικής μεθόδου με τον FFT, σε 2 τρισδιάστατα στένσιλ με διαφορετικές πυκνότητες για διαφορετικά μεγέθη. Οι γραμμές στο γράφημα που αναφέρονται στον FFT, έχουν διαφορετική ομαδοποίηση βημάτων χρόνου. Ο αριθμός δίπλα στο fft δηλώνει τη νέα ακτίνα του στένσιλ με ομαδοποίηση. Για παράδειγμα, για ακτίνα 4, το fft_32 εκτελέι 8 βήματα χρόνου με ένα πέρασμα. Όλες οι γραμμές του σχήματος είναι κανονικοποιημένες και αντιστοιχούν στο χρόνο που χρειάζεται ένα βημα.



Σχήμα 6.12: Επιτάχυνση για διαφορετικές ακτίνες. Στένσιλ αστέρι

Το στένσιλ αστέρι δεν παρουσιάζει καμία βελτίωση. Η επαναληπτική μέθοδος είναι αρκετά γρήγορη ακόμα και για μεγάλη ακτίνα καθώς το στένσιλ αυτό είναι πολύ αραιό, με πυκνότητα μόλις 3% των στοιχείων του κύβου που το περικλύει για ακτίνα 4. Μόνο

για ακτίνα 1 και ομαδοποίηση 48 βημάτων χρόνου παρατηρείται μια μικρή επιτάχυνση ×1.18

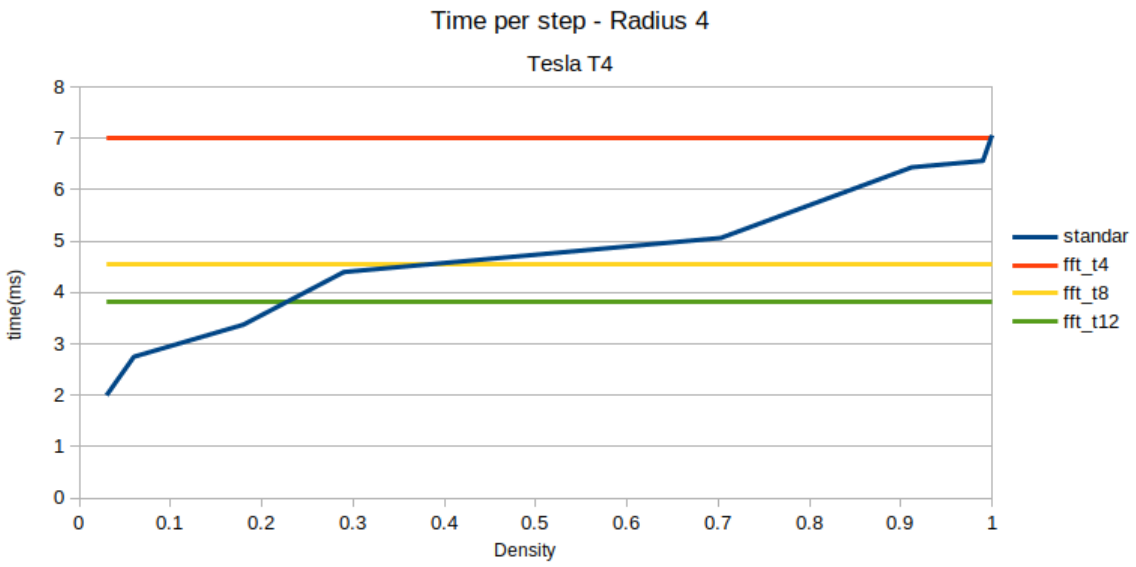


Σχήμα 6.13: Επιτάχυνση για διαφορετικές ακτίνες. Στένσιλ αστέρι

Για το πυκνό στένσιλ τα αποτσελέσματα είναι πολύ καλύτερα. Για κάθε μέγεθος ακτίνας, υπάρχει εκτέλεση του FFT που είναι πιο γρήγορη. Σε αντίθεση με το αστέρι, το πυκνό στένσιλ έχει πυκνότητα 100%.

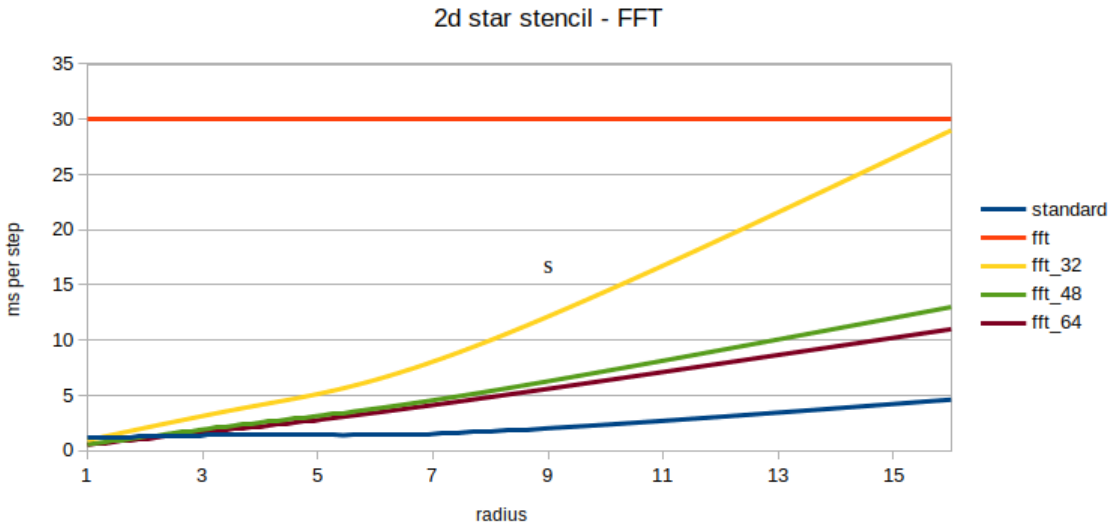
Ένα σημαντικό συμπέρασμα που προκύπτει από τα γραφήματα, είναι πως όσο μεγαλύτερη είναι η ομαδοποίηση βημάτων χρόνου, τόσο βελτιώνεται η απόδοση του FFT για ένα βήμα. Αυτό ήταν αναμενόμενο, καθώς , όπως αναφέρθηκε και στο προηγούμενο κεφάλαιο, ενώ πολλαπλασιάζεται η ακτίνα του στένσιλ , οι επιπλέον υπολογισμοί δεν ακολουθούν την ίδια αυξητική συμπεριφορά και υπολογίζονται πολλαπλάσια βήματα χρόνου.

Για να γίνει αντιληπτό το όριο στο οποίο η μέθοδος στη συχνοτητα είναι πιο γρήγορη από την επαναληπτική, χρειάζεται να γίνει η εφαρμογή τους σε στένσιλ με ενδιάμεσες τιμές πυκνότητας.

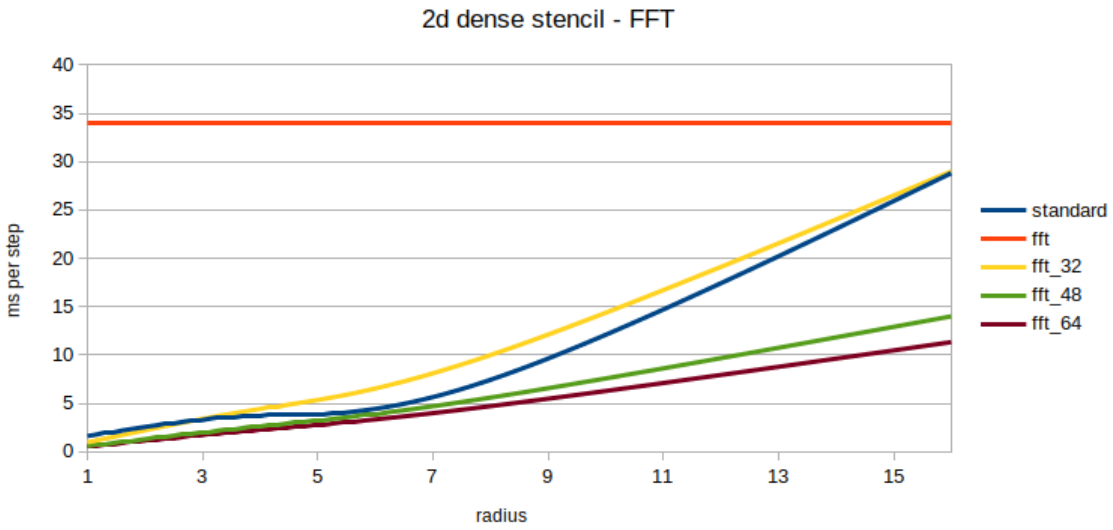


Σχήμα 6.14: Επιτάχυνση για διαφορετικές ακτίνες. Στένσιλ αστέρι

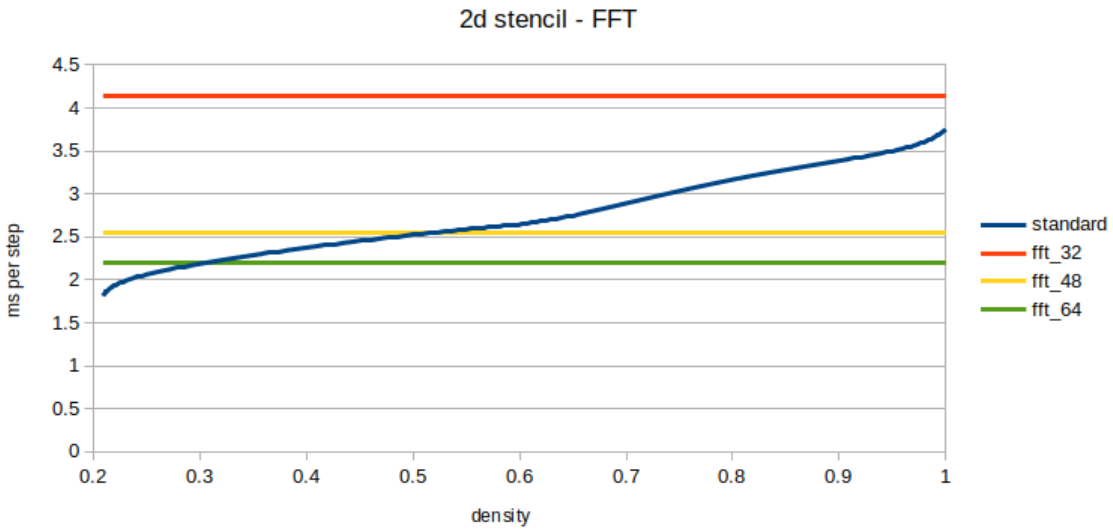
Όπως και στην περίπτωση της ομαδοποίησης βημάτων χρόνου, έτσι και εδώ θια αξιολογηθεί η μέθοδος επιτάχυνσης και για δισδιάστατα στένσιλ.



Σχήμα 6.15: Επιτάχυνση για διαφορετικές ακτίνες. Στένσιλ αστέρι



Σχήμα 6.16: Επιτάχυνση για διαφορετικές ακτίνες. Στένσιλ αστέρι



Σχήμα 6.17: Επιτάχυνση για διαφορετικές ακτίνες. Στένσιλ αστέρι

Ομοίως με το τρισδιάστατο στένσιλ, παρατηρείται επιτάχυνση μόνο για ακτίνα 1. Σε αυτήν την περίπτωση η επιτάχυνση είναι μεγαλύτερη, ×1.9 και ×2.1 για 48 και 64 ομαδοποιημένα βήματα χρόνου. Για διαφορετικές πυκνότητες, φαίνεται η ίδια συμπεριφορά με το τρισδιάστατο στένσιλ, με τη διαφορά πως το σημείο που ο FFT είναι πιο γρήγορος εμφανίζεται για μεγαλύτερη πυκνότητα.

Παρατηρώντας τα αποτελέσματα του FLCC [19] για την συνέλιξη, το σημείο πέρα από το οποίο συμφέρει ο FFT είναι για συνέλιξη με πρότυπο μεγέθους 12ξ12, που αναλογέι σε στένσιλ ακτίνας 6. Στην περίπτωση του πυκνού στένσιλ δε συμφέρει καθόλου ο FFT χωρίς ομαδοποίηση χρόνου. Αυτό σημαίνει δύο πράγματα. Πρώτον υπάρχει χώρος βελτίωσης για την υλοποίηση του FFT, αλλά δεύτερον και πιο σημαντικό, ότι η ομαδοποίηση βημάτων χρόνου είναι μία πολύ καλή και αποτελεσματική μέθοδος

επιτάχυνσης, τόσο για το επαναληπτικό όσο και για το στένσιλ στο πεδίο της συχνότητας.

6.2 Ποιότητα Κώδικα

Στο 3ο κέφαλαιο, μελετώντας δύο περιπτώσεις, έγινε αξιολόγηση της ποιότητας κώδικα των συναρτήσεων πυρήνων, γραμμένες σε Julia. Η εικόνα όμως δεν είναι ολοκληρωμένη καθώς ένα πρόγραμμα στοχευμένο για κάρτες γραφικών, πέρα από τη συνάρτηση πυρήνα, περιλαμβάνει κώδικα για τον έλεγχο και ρύθμιση παραμέτρων της κάρτας γραφικών, προεπεξεργασία και οπτικοποίηση δεδομένων, έλεγχο του αποτελέσματος, διεπαφή με το χρήστη κτλ. Η διαδικασία για τη δημιουργία του framework είχε ως αποτέλεσμα περισσότερη εμβάθυνση και καλύτερη κατανόηση των δυνατοτήτων αυτής της γλώσσας.

6.2.1 Δημιουργία του framework

Οι δυνατότητες μεταπρογραμματισμού και ενδοσκόπησης ήταν σημαντικές, έως και απαραίτητες για την υλοποίηση ενός σχετικά πολυπλόκου προγράμματος, σε περιορισμένο χρόνο, από έναν μόνο προγραμματιστή. Χωρίς να συμπεριληφθούν οι αλγοριθμικές υλοποιήσεις για κάρτες γραφικών, τα πιο πολύπλοκα τμήματα του προγράμματος ήταν η μετατροπή του κώδικα που έδινε σαν είσοδο ο χρήστης σε μαθηματικές πράξεις και αυτόματη παραγωγή κώδικα για κάρτες γραφικών από αυτές τις πράξεις. Το σύνθετο έργο της συντακτικής ανάλυσης κειμένου σε κώδικα και η μετατροπή κειμένου σε εκτελέσιμο κώδικα λύνονται αυτόματα από τη Julia. Έτσι μένει μόνο η υλοποίηση της λογικής.

Λόγω της εκφραστικότητας και ευελιξίας που παρέχει η Julia δε χρειάζεται να χρησιμοποιηθούν εξωτερικά προγράμματα ως συντακτικοί αναλυτές. Ο χρήστης δίνει σαν είσοδο κώδικα Julia, που είναι αυτόματα σε μορφή δεδομένων κατανοητή για το πρόγραμμα.

Αξιοποιώντας, πάλι, τη δυνατότητα ενδοσκόπησης, είναι δυνατόν να γραφτεί εύκολα κώδικας Julia μέσω της Julia. Στο παρακάτω δείγμα κώδικα είναι εμφανές πόσο απλό είναι να δημιουργηθεί κώδικας. Η μετατροπή του κώδικα από δομή δεδομένων σε εκτελέσιμη συνάρτηση γίνεται επίσης εύκολα μέσω βιβλιοθήκης.

Τμήμα Κώδικα 6.1: Julia παράδειγμα παραγωγής κώδικα Julia.

```
#Initialize code structure with 2 instructions
code = quote
    a = 0
    a += D[1]
end
#Add instruction to existing code structure
new_instruction = :(a += D[2])
push!(code, new_instruction)
```

Πολύ σημαντική επίσης, είναι η δυνατότητα να οπτικοποιηθούν τα δεδομένα μέσα στο ίδιο το πρόγραμμα. Το γεγονός ότι δε χρησιμοποιείται άλλη γλώσσα προγραμματισμού, όπως συμβαίνει με τη C, κάνει την αποσφαλμάτωση πιο γρήγορη και εύκολη.

6.2.2 Μία Κάρτα Γραφικών

Τόσο στις δύο προηγούμενες μελέτες, όσο και στο framework στην απλή περίπτωση, υπάρχει έλεγχος μίας κάρτας γραφικών από τον επεξεργαστή. Ο έλεγχος περιλαμβάνει τον ορισμό μνήμης για την κάρτα, την μεταφορά δεδομένων από και προς αυτή και την εκτέλεση της συνάρτησης πυρήνα. Οι χαμηλού επιπέδου [21] [22] συναρτήσεις της CUDA έχουν ¨τυλιχθεί¨ με κώδικα Julia και φαίνεται σαν να υποστηρίζεται έμφυτα από την γλώσσα ο προγραμματισμός μιας κάρτας γραφικών, αλλά και γενικά η εκτέλεση κώδικα σε αυτή.

Τμήμα Κώδικα 6.2: Julia παράδειγμα παραγωγής κώδικα Julia.

```
# Upload data to GPU
device_data = CuArray(data)
# Execute GPU kernel
@cuda threads=32 blocks=(16,16) kernel(device_data)
```

Τμήμα Κώδικα 6.3: Julia παράδειγμα παραγωγής κώδικα Julia.

```
# Upload data to GPU
devA = CuArray(dataA)
devB = CuArray(dataB)
# Array multiplication in GPU
dev_result = devA*devB
# Download result to CPU
res = Array(dev_result)
```

Για την εκτέλεση συνάρτησης πυρήνα για στένσιλ, ήταν απαραίτητο να αλλάξει το μέγεθος του πίνακα και να γεμίσουν με μηδενικά οι καινούριες θέσεις. Στην C, για να υλοποιηθεί αυτό, θα απαιτούνταν η χρήση νέου πίνακα στην RAM και η μετακίνηση των στοιχείων σε αυτόν, που απαιτεί διπλάσια μνήμη. Λόγω του υψηλού επιπέδου της Julia, μπορεί πολύ εύκολα να δημιουργηθεί η περιγραφή του επαυξημένου πίνακα χωρίς να βρίσκεται πραγματικά στη μνήμη, και έπειτα μέσω της περιγραφής να γίνει η μεταφορά δεδομένων στην κάρτα γραφικών. Στο παρακάτω δείγμα κώδικα ορίζεται μια συνάρτηση που δημιουργεί επαυξημένο πίνακα προσθέτοντας *pd* στοιχεία σε 4 από τις 6 πλευρές του πίνακα. Έπειτα, η μετακίνηση των δεδομένων στην κάρτα γίνεται όπως και στα απλά δεδομένα.

Τμήμα Κώδικα 6.4: Julia παράδειγμα παραγωγής κώδικα Julia.

```
function PadData(pd, data)
    x = size(data,1)
    y = size(data,2)
    z = size(data,3)
    padx = 1:(x+2pd)
    pady = 1:(y+2pd)
    padz = 1:z
    datax = (pd+1):(x+pd)
    datay = (pd+1):(y+pd)
    dataz = (1:z)
    return PaddedView(0, data,
```

```
                            (padx,pady,padz),
                            (datax, datay, dataz))

end
padded_data = PadData(data, pd)
dev_arr = CuArray(padded_data)
```

Μια τέτοια προσέγγιση θα μπορούσε να υλοποιηθεί στην C, όμως θα απαιτούσε κόπο,χρόνο και δε θα μπορούσε να καλύψει πολλές περιπτώσεις όπως γίνεται σε μια γλώσσα υψηλού επιπέδου.

6.2.3 Πολλαπλές κάρτες γραφικών

Μέχρι αυτήν τη στιγμή (Δεκέμβριος 2020), υποστηρίζει λίγες δυνατότητες από την εργαλειοθήκη της CUDA για προγραμματισμό σε πολλαπλές κάρτες γραφικών [23]. Η δουλειά για υποστήριξη πολλαπλών καρτών γραφικών είναι σε εξέλιξη. Το γεγονός αυτό έκανε αρκετά πιο δύσκολη την υλοποίηση για πολλαπλές κάρτες γραφικών από το framework.

Πολύ μεγάλο μέρος του χρόνου αναλώθηκε σε απλά πράγματα, όπως η δέσμευση ομογενοποιημένης μνήμης ή η μεταφορά τμήματος πίνακα από μία κάρτα γραφικών σε άλλη. Καμία από τις συναρτήσεις που χρειάστηκαν δεν υπήρχαν στο εγχειρίδιο της γλώσσας. Επίσης, η Julia δεν υποστηρίζει έμφυτα τη χρήση δεικτών μνήμης, η χρήση των συναρτήσεων CUDA σε Julia ήταν διαφορετική από ότι στην C.

Στο επόμενο παράδειγμα κώδικα, φαίνονται τα βήματα που χρειάζονται για να γίνει χρήση της ομογενοποιημένης μνήμης που παρέχει η CUDA. Ένα σημαντικό εμπόδιο, είναι πως αυτό το χαρακτηριστικό δεν υποστηρίζεται άμεσα από τη βιβλιοθήκη CuArrays, που διευκόλυνε σε πολύ μεγάλο βαθμό τη διαδικασία σε μία κάρτα γραφικών. Για να υπάρχει τρσιδιάστατη διευθυνσιοδότηση, όμως, πρέπει τα δεδομένα να δίνονται στην συνάρτηση πυρήνα ως CuArray. Έτσι, αρχικά δημιουργείται ομογενοποιημένος buffer, μετατρέπεται σε δομή δεδομένων που παίζει το ρολο δείκτη και τέλος σε CuArray.

Τμήμα Κώδικα 6.5: Julia παράδειγμα παραγωγής κώδικα Julia.

```
gpu_buffer = Mem.alloc(Mem.Unified,
                       size(data)*sizeof(Float32))
gpu_pointer = convert(CuPtr{Nothing}, gpu_buffer)
gpu_cuarray = unsafe_wrap(CuArray{Float32, 3},
                          gpu_pointer, x, y, z,
                          own=false)
```

Κάθε μια από αυτές τις μεταβλητές χρησιμοποιείται μέσα στον κώδικα, ο buffer για μεταφορά δεδομένων από επεξεργαστή προς κάρτες γραφικων, ο δείκτης για μεταφορά τμήματος μνήμης μεταξύ καρτών γραφικών και CuArray σαν όρισμα στις συναρτήσεις πυρήνα.

Η μεταφορά δεδομένων ανάμεσα σε κάρτες γραφικών είναι σχετικά απλή, μοιάζει με την C, όμως δεν αρμόζει σε κώδικα υψηλού επιπέδου όπως φαίνεται από το επόμενο παράδειγμα:

Τμήμα Κώδικα 6.6: Julia παράδειγμα παραγωγής κώδικα Julia.

```
p1 = gpu_pointers_in[i[1]]
p2 = gpu_pointers_in[i[2]]
offp1 = p1 + dx*dy*(dz-2radius*t_group)*sizeof(Float32)
CUDA.cuMemcpy(p2, offp1, bsize)
```

Αλλά σημαντικά προβλήματα που αντιμετωπίστηκαν ήταν η αποδέσμευση μνήμης και η πραγματική δέσμευση ομογενοποιημένης μνήμης. Η βιβλιοθήκη CuArrays ακολουθεί το μοντέλο της γλώσσας και έχει αυτόματη αποδέσμευση μνήμης για τις κάρτες γραφικών. Επειδή δεν υποστηρίζει ομογενοποιημένη μνήμη, όταν μετατρέπεται ο buffer σε CuArray, η βιβλιοθήκη προσπαθεί να αποδεσμεύσει τη μνήμη, η οποία δεν της ανήκει. Αυτό οδηγεί σε σφάλματα που είναι δύσκολο να εντοπιστεί η πραγματική τους αιτία, καθώς συνέβαιναν σε τυχαίο χρόνο. Όσον αφορά την ομογενοποιημένη μνήμη, η Julia λανθασμένα δεσμεύει όλους τους πίνακες σε μία κάρτα γραφικών, πριν ¨καταλάβει¨ και δεσμεύσει κάθε πίνακα στη σωστή και έτσι, υπάρχει μια μικρή καθυστέρηση. Αυτό το πρόβλημα δεν εντοπίστηκε σε πρόγραμμα στη γλώσσα C.

Κεφάλαιο 7

Παράρτημα

αρχη

Τμήμα Κώδικα 7.1: Julia παράδειγμα παραγώμενου κώδικα για τρισδιάστατο στένσιλ αστέρι με ακτίνα 2.

```
quote
    dimz = size(g_input, 3)
    dimzend = dimz - 2
    bdimx = ((CUDA).blockDim()).x
    bdimy = ((CUDA).blockDim()).y
    bx = (((CUDA).blockIdx()).x - 1) * 32 + 1 + offx
    by = (((CUDA).blockIdx()).y - 1) * 32 + 1 + offy
    tx = ((CUDA).threadIdx()).x
    ty = ((CUDA).threadIdx()).y
    txr = ((CUDA).threadIdx()).x + 2
    tyr = ((CUDA).threadIdx()).y + 2
    tile = (CUDA).@cuDynamicSharedMem(Float32, (36, 36))
    @inbounds l_input = @view(g_input[bx:bx + 35,
                                      by:by + 35, :])
    @inbounds l_output = @view(g_output[bx:bx + 35,
                                        by:by + 35, :])
    behind2 = Float32(0.0)
    behind1 = Float32(0.0)
    current = Float32(0.0)
    infront1 = Float32(0.0)
    infront2 = Float32(0.0)
    cfarr = (StaticArrays).SA_F32[0.25f0, 0.5f0]
    for z = 1 + offz_f:dimz - offz_b
        (CUDA).sync_threads()
        if tx + 0 <= 36 && ty + 0 <= 36
            @inbounds tile[tx + 0, ty + 0] =
                                l_input[tx + 0, ty + 0, z]
        end
        if tx + 0 <= 36 && ty + 32 <= 36
            @inbounds tile[tx + 0, ty + 32] =
```

```
                                l_input[tx + 0, ty + 32, z]
        end
        if tx + 32 <= 36 && ty + 0 <= 36
            @inbounds tile[tx + 32, ty + 0] =
                                l_input[tx + 32, ty + 0, z]
        end
        if tx + 32 <= 36 && ty + 32 <= 36
            @inbounds tile[tx + 32, ty + 32] =
                                l_input[tx + 32, ty + 32, z]
        end
        (CUDA).sync_threads()
        @inbounds temp = tile[txr + -2, tyr + 0]
        current += cfarr[1] * temp
        @inbounds temp = tile[txr + -1, tyr + 0]
        current += cfarr[2] * temp
        @inbounds temp = tile[txr + 0, tyr + -2]
        current += cfarr[1] * temp
        @inbounds temp = tile[txr + 0, tyr + -1]
        current += cfarr[2] * temp
        @inbounds temp = tile[txr + 0, tyr + 0]
        current += temp
        infront2 += cfarr[1] * temp
        infront1 += cfarr[2] * temp
        behind2 += cfarr[1] * temp
        behind1 += cfarr[2] * temp
        @inbounds temp = tile[txr + 0, tyr + 1]
        current += cfarr[2] * temp
        @inbounds temp = tile[txr + 0, tyr + 2]
        current += cfarr[1] * temp
        @inbounds temp = tile[txr + 1, tyr + 0]
        current += cfarr[2] * temp
        @inbounds temp = tile[txr + 2, tyr + 0]
        current += cfarr[1] * temp
        if z > 2
            @inbounds l_output[txr, tyr, z - 2] = behind2
        end
        behind2 = behind1
        behind1 = current
        current = infront1
        infront1 = infront2
        infront2 = Float32(0)
    end
    @inbounds l_output[txr, tyr, (dimz-0) - offz_b] = behind1
    @inbounds l_output[txr, tyr, (dimz-1) - offz_b] = behind2
    return nothing
end
```

Τελος παραρτηματος

Παράρτημα Α΄

Ακρωνύμια και συντομογραφίες

LAN Local Area Network

Βιβλιογραφία

[1] J. Meng and K. Skadron, "A Performance Study for Iterative Stencil Loops on GPUs with Ghost Zone Optimizations," *International Journal of Parallel Programming*, vol. 39, no. 1, pp. 115–142, Feb. 2011. [Online]. Available: http://link.springer.com/10.1007/s10766-010-0142-5

[2] J. Holewinski, L.-N. Pouchet, and P. Sadayappan, "High-performance code generation for stencil computations on GPU architectures," in *Proceedings of the 26th ACM international conference on Supercomputing*, ser. ICS '12. Association for Computing Machinery, pp. 311–320. [Online]. Available: https://doi.org/10.1145/2304576.2304619

[3] M. Misic, Đurđević, and M. Tomasevic, "Evolution and trends in GPU computing," Jan. 2012, pp. 289–294.

[4] P. Zunitch, "CUDA vs. OpenCL vs. OpenGL," Jan. 2018, section: Editing. [Online]. Available: https://www.videomaker.com/article/c15/19313-cuda-vs-opencl-vs-opengl

[5] F. Fleutot and L. Tratt, "Contrasting compile-time meta-programming in Metalua and Converge," p. 10.

[6] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *2008 SC - International Conference for High Performance Computing, Networking, Storage and Analysis*. Austin, TX: IEEE, Nov. 2008, pp. 1–12. [Online]. Available: http://ieeexplore.ieee.org/document/5222004/

[7] "CUDA C Best Practices Guide," p. 85.

[8] P. Micikevicius, "3d finite difference computation on GPUs using CUDA," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units - GPGPU-2*. ACM Press, pp. 79–84. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1513895.1513905

[9] T. Mo and L. Renfa, "A new memory mapping mechanism for gpgpus' stencil computation," *Computing*, vol. 97, pp. 795–812, 11 2014.

[10] M. Frigo and V. Strumpen, "Cache oblivious stencil computations," *Proceedings of the 19th annual international conference on Supercomputing - ICS '05*. [Online]. Available: https://www.academia.edu/19784992/Cache_oblivious_stencil_computations

[11] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey, "3.5-d blocking optimization for stencil computations on modern cpus and gpus," 11 2010, pp. 1–13.

[12] R. V. Lim, B. Norris, and A. D. Malony, "Autotuning GPU Kernels via Static and Predictive Analysis," *arXiv:1701.08547 [cs]*, Jun. 2017, arXiv: 1701.08547. [Online]. Available: http://arxiv.org/abs/1701.08547

[13] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams, "An auto-tuning framework for parallel multicore stencil computations," 04 2010, pp. 1–12.

[14] J. D. Garvey and T. S. Abdelrahman, "A Strategy for Automatic Performance Tuning of Stencil Computations on GPUs," *Scientific Programming*, vol. 2018, pp. 1–24, May 2018. [Online]. Available: https://www.hindawi.com/journals/sp/2018/6093054/

[15] A. Schäfer and D. Fey, "LibGeoDecomp: A Grid-Enabled Library for Geometric Decomposition Codes," Jan. 1970, pp. 285–294.

[16] M. Christen, O. Schenk, and H. Burkhart, "Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures," 05 2011, pp. 676–687.

[17] G. S. Murthy, M. Ravishankar, M. M. Baskaran, and P. Sadayappan, "Optimal loop unrolling for GPGPU programs," in *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. Atlanta, GA, USA: IEEE, 2010, pp. 1–11. [Online]. Available: http://ieeexplore.ieee.org/document/5470423/

[18] R. A. Haddad and A. N. Akansu, "A class of fast Gaussian binomial filters for speech and image processing," *IEEE Transactions on Signal Processing*, vol. 39, no. 3, pp. 723–727, Mar. 1991, conference Name: IEEE Transactions on Signal Processing.

[19] G. Papamakarios, G. Rizos, and N. Pitsianis, "FLCC: A library for fast computation of convolution and local correlation coefficients," 12 2011. [Online]. Available: http://flcc.cs.duke.edu/home

[20] Παπαμακάριος, Γεώργιος και Ρίζος, Γεώργιος, "FLCC:ΜΙΑ ΒΙΒΛΙΟΘΗΚΗ ΓΙΑ ΤΑΧΥ ΥΠΟΛΟΓΙΣΜΟ ΣΥΝΕΛΙΞΗΣ ΚΑΙ ΤΟΠΙΚΩΝ ΣΥΝΤΕΛΕΣΤΩΝ ΣΥΣΧΕΤΙΣΗΣ," Διπλωματική Διατριβή, Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, 2011, p. 159.

[21] T. Besard, V. Churavy, A. Edelman, and B. De Sutter, "Rapid software prototyping for heterogeneous and distributed platforms," *Advances in Engineering Software*, vol. 132, pp. 29–46, 2019.

[22] T. Besard, C. Foket, and B. De Sutter, "Effective extensible programming: Unleashing Julia on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, 2018.

[23] "Multiple GPUs · CUDA.jl." [Online]. Available: https://juliagpu.gitlab.io/CUDA.jl/usage/multigpu/