Υποδομή λειτουργιών υποβίβασης στον παραλληλοποιητικό μεταφραστή OMPi για κάρτες γραφικών CUDA

Γεώργιος Ευάγγελος Θεοδώρου

Διπλωματική Εργασία

Επιβλέπων: Βασίλειος Δημακόπουλος

Ιωάννινα, Ιούνιος 2022

Τμήμα Μηχ. Η/Υ & Πληροφορικής
Πανεπιστήμιο Ιωαννίνων

Department of Computer Science & Engineering
University of Ioannina

Περίληψη

Στην εργασία αυτή, παρουσιάζεται η ανάπτυξη μηχανισμού υποβίβασης (reduction) στο παραλληλοποιητικό μεταφραστή OMPi, στοχεύοντας επιταχυντές γραφικής επεξεργασίας CUDA. Αρχικά παρουσιάζεται η προγραμματιστική διεπαφή OpenMP η οποία αποτελεί το δημοφιλέστερο μοντέλο παράλληλου προγραμματισμού για συστήματα κοινόχρηστης μνήμης. Στην συνέχεια παρουσιάζεται η αρχιτεκτονική CUDA και η διεπαφή χρόνου εκτέλεσης CUDA Runtime που αφορά σε κάρτες γραφικής επεξεργασίας NVIDIA και στον τρόπο προγραμματισμού τους.
Δίνονται και αναλύονται πολλαπλοί αλγόριθμοι υποβίβασης για όλους τους υποστηριζόμενους τύπους δεδομένων. Επιπλέον, περιγράφουμε πως αυτοί προσαρμόστηκαν έτσι ώστε να ενσωματωθούν στο παραλληλοποιητικό μεταφραστή OMPi και να παρέχουν πλήρεις λειτουργίες OpenMP reduction στη συσκευή cudadev που απευθύνεται σε GPUs τύπου CUDA. Στόχος της ανάπτυξης αυτής είναι να δημιουργηθεί μια υλοποίηση η οποία εκμεταλλεύεται πλήρως τις δυνατότητες παραλληλοποίησης που προσφέρουν οι κάρτες γραφικής επεξεργασίας.

Abstract

The subject of this thesis is the development of a reduction mechanism in the parallel compiler OMPi targeting CUDA graphics processing accelerators. First, the OpenMP programming interface is presented, which is the most popular parallel programming model for shared memory systems. Then the CUDA architecture and the CUDA Runtime interface, which defines a way to program to NVIDIA graphics cards, are presented. Multiple reduction algorithms are presented and analyzed for all supported data types. In addition, we describe how these were adapted to be integrated within the OMPi parallel compiler and provide full OpenMP reduction functionality to the cudadev device targeting CUDA GPUs. The goal of this development is to create an effitient implementation that takes full advantage of the parallelization capabilities offered by graphics processing cards.

Πίνακας Περιεχομένων

Κατάλογος Πινάκων

Κατάλογος Σχημάτων

Κεφάλαιο 1.
Εισαγωγή

1.1 Εξέλιξη των Η/Υ

Ο πρώτος ηλεκτρονικός υπολογιστής έκανε την εμφάνιση του την δεκαετία του 1940 με το όνομα ENIAC. Ο υπολογιστής αυτός καταλάμβανε μεγάλο χώρο, ήταν θορυβώδης, αποτελούταν από εκατοντάδες λυχνίες που καίγονταν συχνά και κατανάλωνε πολλή ενέργεια. Ήταν όμως ένα μεγάλο βήμα προς τους σημερινούς υπολογιστές.

Έκτοτε ο άνθρωπος έχει δαπανήσει πολλούς πόρους για την βελτιστοποίηση αυτού του εγχειρήματος. Είτε πρόκειται για το μέγεθος, είτε για την κατανάλωση ενέργειας και σημαντικότερα την χωρητικότητα μνήμης και την επεξεργαστική ισχύ ένας απλός σημερινός υπολογιστής είναι τάξεις μεγεθών καλύτερος από τον ENIAC.

Παρατηρώντας κάποιος αυτή την δραματική εξέλιξη των υπολογιστών, θα μπορούσε να ισχυριστεί ότι σε μερικά χρόνια η ταχύτητα των υπολογιστών θα είναι τέτοια ώστε θα μπορούσαν να επιλύσουν οποιοδήποτε υπολογίσιμο πρόβλημα σε σχεδόν μηδενικό χρόνο. Όμως ο ισχυρισμός αυτός φαίνεται να διαψεύδεται αφού έχουμε καταλήξει στο συμπέρασμα ότι η αύξηση της ταχύτητας των επεξεργαστών δεν είναι δυνατόν να συνεχιστεί επ' αορίστου.

Φυσικοί νόμοι, όπως η ταχύτητα των ηλεκτρονίων που δεν μπορεί να ξεπεράσει την ταχύτητα του φωτός, καθώς και περιορισμοί στην πυκνότητα των transistors ανά μονάδα επιφάνειας θέτουν ένα άνω όριο στην αύξηση της ταχύτητας.

Για να ξεπεράσουμε όμως αυτούς τους φυσικούς περιορισμούς, έρχεται στο προσκήνιο το παράλληλο μοντέλο υπολογισμού και η παράλληλη επεξεργασία που αποτελούν εξέλιξη του κλασικού μοντέλου προγραμματισμού/αρχιτεκτονικής von Neumann ή αλλιώς σειριακό.

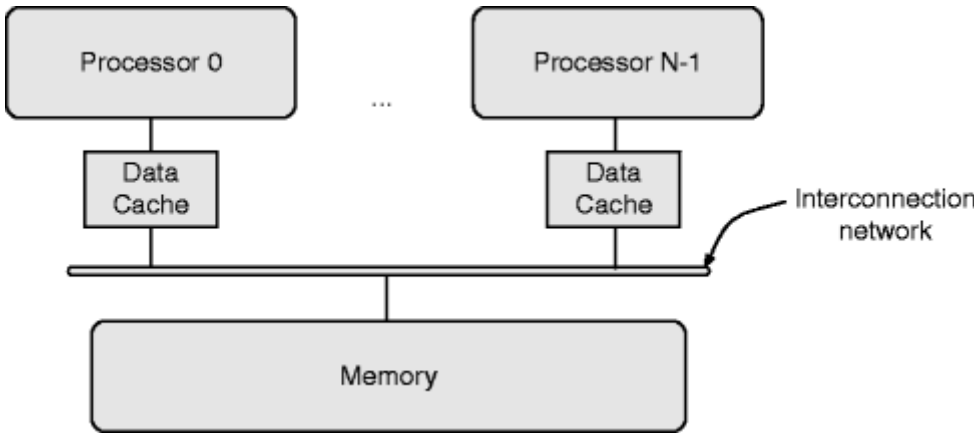
1.2 Παράλληλα Συστήματα

Ως παράλληλο σύστημα μπορεί να οριστεί ένα σύνολο από πολλές επεξεργαστικές μονάδες (CPUs) που συνεργάζονται για την επίλυση ενός προβλήματος. Συγκεκριμένα ένα παράλληλο σύστημα διαιρεί το αρχικό πρόβλημα σε μικρότερα υποπροβλήματα, τα επιλύει ταυτόχρονα, και συγκεντρώνει τα αποτελέσματα, εξάγοντας το τελικό.

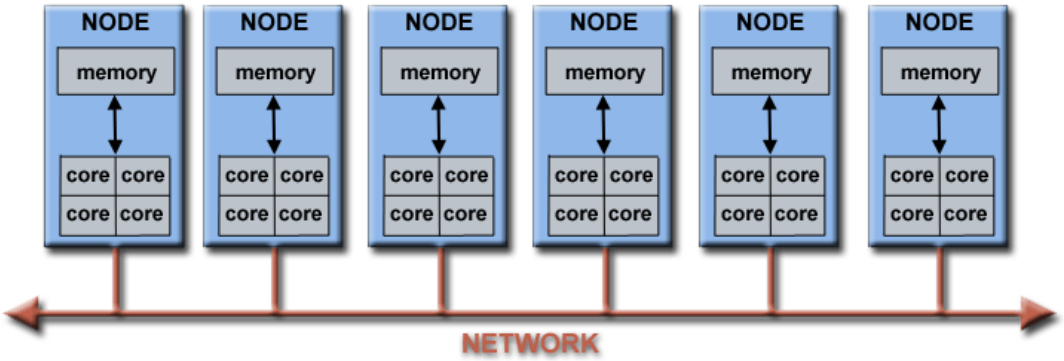
1.2.1 Αρχιτεκτονικές Παράλληλων Συστημάτων

Ένα παράλληλο σύστημα μπορεί να αποτελείται είτε από πολλούς πυρήνες σε ένα chip είτε από πολλούς διαφορετικούς υπολογιστές.

Στην πρώτη περίπτωση αναφερόμαστε στα παράλληλα συστήματα κοινόχρηστης μνήμης. Οι διαφορετικοί επεξεργαστές μοιράζονται κάποιο κοινό χώρο διευθύνσεων μνήμης και επικοινωνούν τροποποιώντας μεταβλητές. Η μνήμη οργανώνεται σε κεντρική θέση, όπως στα συστήματα SMP (symmetric multiprocessors), όπου κάθε επεξεργαστής συνδέεται σε ένα δίαυλο μέσω του οποίου έχει πρόσβαση σε αυτή, όπως απεικονίζεται στο σχήμα 1.1. Όλοι οι επεξεργαστές έχουν τον ίδιο χρόνο προσπέλασης στην κύρια μνήμη και για αυτόν τον λόγο αναφέρονται ως UMA (Uniform Memory Access). Επίσης κάθε επεξεργαστής διαθέτει και τοπική cache μνήμη με αποτέλεσμα να δημιουργούνται προβλήματα με την συνέπειά της. Εναλλακτικά υπάρχει και το μοντέλο NUMA (Non-Uniform Memory Access) όπου κάθε επεξεργαστής είναι συνδεδεμένος σε ιδιωτική μνήμη. Αυτά τα κομμάτια μνήμης όμως συνδυάζονται ώστε να προκύψει ο κοινός χώρος διευθύνσεων. Σε αντίθεση με το μοντέλο UMA, ο χρόνος προσπέλασης της μνήμης διαφέρει για κάθε επεξεργαστή καθώς εξαρτάται από την τοποθέτησή του. Οι πυρήνες ομαδοποιούνται σε κόμβους (nodes) στους οποίους κάθε ένας έχει τον δικό του ελεγκτή μνήμης.



Σχήμα 1.1: Μοντέλο Κοινής Μνήμης



Σχήμα 1.2: Μοντέλο Κατανεμημένης Μνήμης

Η άλλη κατηγορία είναι τα συστήματα κατανεμημένης μνήμης. Τα συστήματα αυτά αποτελούνται από συστάδες διαφορετικών υπολογιστών (clusters) που διαθέτουν δικιά τους μνήμη και συνδέονται με δίκτυο διασύνδεσης για την ανταλλαγή μηνυμάτων. Η διασύνδεση αυτή φαίνεται στο σχήμα 1.2 Τα πλεονεκτήματα αυτής της αρχιτεκτονικής είναι ότι κάθε μονάδα μπορεί να εκμεταλλευτεί όλο το διαθέσιμο εύρος ζώνης της τοπικής μνήμης. Επίσης λόγω της απουσίας διαύλου δεν υπάρχει περιορισμός στο μέγεθος του συστήματος αφού το δίκτυο διασύνδεσης υποστηρίζει μεγάλο αριθμό μονάδων. Ακόμα δεν εμφανίζονται προβλήματα συνέπειας cache καθώς κάθε μονάδα είναι υπεύθυνη για την συνέπεια της δικιάς της μνήμης. Παρά όλα αυτά τα πλεονεκτήματα το μεγάλο μειονέκτημα αυτής της αρχιτεκτονικής είναι ο χρόνος που δαπανάται για τις επικοινωνίες. Όταν ένας επεξεργαστής χρειάζεται δεδομένα από έναν άλλο θα πρέπει να γίνει ανταλλαγή μηνυμάτων, διαδικασία που απαιτεί χρόνο για την δημιουργία και αποστολή του μηνύματος καθώς και την λήψη του, που απαιτεί διακοπή της λειτουργίας του δέκτη για την επεξεργασία του μηνύματος.

1.2.2 Προγραμματιστικά Μοντέλα

Ο προγραμματισμός συστημάτων κοινής μνήμης ακολουθεί το μοντέλο κοινού χώρου διευθύνσεων, με χρήση διεργασιών ή νημάτων. Για τον προγραμματισμό τέτοιων συστημάτων έχουν κατασκευαστεί διάφορες προγραμματιστικές διεπαφές, κάθε μια με τα δικά της χαρακτηριστικά. Η πιο διαδεδομένη είναι τα νήματα POSIX (Pthreads). Ο προγραμματιστής μπορεί με κλήσεις της βιβλιοθήκης να διατηρήσει της συνέπεια των δεδομένων και να συγχρονίσει την εκτέλεση των νημάτων.

Λόγω της δυσκολίας προγραμματισμού με νήματα POSIX αναπτύχθηκε το πρότυπο OpenMP. Το OpenMP είναι μια διεπαφή που περιλαμβάνει ένα σετ από οδηγίες προς τον μεταφραστή που επηρεάζουν τη συμπεριφορά του χρόνου εκτέλεσης. Στόχος είναι ο χρήστης να μπορεί να αναπτύξει εύκολα παράλληλες εφαρμογές, "κρύβοντας" περίπλοκες διαδικασίες που με τον συντονισμό των νημάτων, μέσω των υλοποιήσεων της διεπαφής για μεταφραστές που υποστηρίζουν τις γλώσσες C, C++ και Fortrant. Το OpenMP θα περιγραφεί σε βάθος σε επόμενο κεφάλαιο.

Σε ένα κατανεμημένο σύστημα, παράλληλες διεργασίες ανταλλάσουν δεδομένα μέσω μηνυμάτων η μία στην άλλη. Οι επικοινωνίες αυτές μπορεί να είναι ασύγχρονες, δηλαδή όταν ένα μήνυμα μπορεί να σταλεί πριν ο αποδέκτης να είναι έτοιμος ή σύγχρονες, δηλαδή όταν ο αποδέκτης πρέπει να είναι έτοιμος. Το πιο διαδεδομένο πρότυπο είναι το MPI, το οποίο προσφέρει έτοιμες ρουτίνες για την αποστολή και λήψη μηνυμάτων και άλλων λειτουργιών, χωρίς να αναγκάζει τον προγραμματιστή να γνωρίζει τα πάντα για την αρχιτεκτονική και τη διασύνδεση των υπολογιστών.

1.3 Επιταχυντές και GPUs

Οι επεξεργαστές των σημερινών παράλληλων υπολογιστών συστημάτων, παρότι μπορούν να χρησιμοποιηθούν για την επίλυση όλων των υπολογιστικών προβλημάτων, παρατηρούμε ότι δεν είναι όσο αποδοτικοί θα θέλαμε σε συγκεκριμένα είδη υπολογισμών. Για τον λόγο αυτό αναπτύχθηκαν υπολογιστικές συσκευές ο οποίες φιλοξενούνται σε ένα σύστημα και επιταχύνουν ορισμένους τύπους υπολογισμών. Οι συσκευές αυτές ονομάζονται επιταχυντές και υλοποιούνται σε ξεχωριστό υλικό και είναι άμεσα προσπελάσιμες από την κεντρική μονάδα επεξεργασίας, μέσω διαύλων υψηλής ταχύτητας. Το πιο σύνηθες είδος επιταχυντή είναι οι μονάδες επεξεργασίας γραφικών (GPU). Οι μονάδες επεξεργασίας γραφικών είναι αυτόνομες συσκευές που διαθέτουν μεγάλο αριθμό από ανίσχυρους πυρήνες, οι οποίοι εκτελούν ορισμένες απλές πράξεις με μεγάλη ταχύτητα. Παραδοσιακά οι πυρήνες αυτοί αυτοί χρησιμοποιούνταν για την παραγωγή γραφικών και την απεικόνισή τους στην οθόνη. Με το πέρασμα του χρόνου οι προγραμματιστές διαπίστωσαν ότι οι πυρήνες αυτοί μπορούν να χρησιμοποιηθούν για την επιτάχυνση υπολογισμών γενικής φύσεως. Η οργάνωση και το πλήθος των πυρήνων μιας μονάδας επεξεργασίας γραφικών τους καθιστά τους πλέον ιδανικούς για την εκτέλεση πράξεων σε πίνακες.

1.3.1 Προγραμματισμός των GPUs

Για τον προγραμματισμό των μονάδων γραφικής επεξεργασίας, διατίθενται διάφορα προγραμματιστικά μοντέλα που στοχεύουν σε συγκεκριμένη αρχιτεκτονική. Παραδείγματα τέτοιων μοντέλων είναι η CUDA (Compute Unified Device Architecture)[1] και η OpenCL [2] (Open Computing Language). Συγκεκριμένα, ο προγραμματιστής συγγράφει εφαρμογές στη γλώσσα προγραμματισμού C/C++, οι οποίες αποτελούνται από τμήματα κώδικα που αφορούν τον κύριο επεξεργαστή και τμήματα που αφορούν τη μονάδα γραφικής επεξεργασίας. Ο κώδικας που αφορά τη μονάδα GPU δομείται σε συναρτήσεις οι οποίες ονομάζονται υπολογιστικοί πυρήνες (compute kernels) και περιλαμβάνουν τους ωφέλιμους υπολογισμούς. Ο κώδικας που αφορά τον κύριο επεξεργαστή, είναι υπεύθυνος για την προετοιμασία της μονάδας GPU ώστε να εκτελέσει τους υπολογιστικούς πυρήνες. Συγκεκριμένα, η προετοιμασία αφορά στην εκκίνηση και αρχικοποίηση της μονάδας GPU, τη μεταφορά δεδομένων από/προς αυτή, την εκτέλεση του υπολογιστικού πυρήνα (kernel) καθώς επίσης και τον τερματισμό της. Η ανάπτυξη όμως εφαρμογών αυτού του τύπου μπορεί να γίνει εξαιρετικά δύσκολη καθώς ο προγραμματιστής πρέπει να χειριστεί πολύ προσεκτικά τη διαδικασία μεταφοράς δεδομένων από και προς την μονάδα GPU και να ελέγξει τον τρόπο με τον οποίο θα εκτελεστούν οι υπολογισμοί. Προκειμένου να μειωθεί η δυσκολία του προγραμματισμού των GPUs το πρότυπο OpenMP [3] από την έκδοση 4.5 υποστηρίζει εκτέλεση σε σύνοδες συσκευές (devices) του συστήματος. Η λειτουργία αυτή δίνει την δυνατότητα χρήσης των μονάδων GPU για την επιτάχυνση υπολογισμών μέσα από μία απλούστερη διεπαφή.

1.4 Αντικείμενο της Διπλωματικής

Ο κύριος στόχος της διπλωματικής εργασίας αυτής είναι η μελέτη και υλοποίηση αποδοτικών αλγορίθμων υποβίβαση (reduction) για κάρτες γραφικής επεξεργασίας που στηρίζονται στην αρχιτεκτονική CUDA και η αντίστοιχη επέκταση των δυνατοτήτων του παραλληλοποιητικού μεταφραστή OMPi και πιο συγκεκριμένα, η ενσωμάτωση λειτουργικότητας υποβίβασης για σύνοδες συσκευές OpenMP τύπου CUDA.

Ο OMPi, είναι ένας μεταφραστής source-to-source, ο οποίος δέχεται ως είσοδο ένα σειριακό πρόγραμμα, το οποίο περιέχει εντολές του μοντέλου OpenMP και παράγει ένα πρόγραμμα σε γλώσσα C, παραλληλοποιημένο αποδοτικά με νήματα και έτοιμο να μεταφραστεί από τον εκάστοτε μεταφραστή του συστήματος.

Το reduction είναι η λειτουργία συγκέντρωσης υποαποτελεσμάτων από τα νήματα που εκτελούνται και η σύμπτυξή τους σε ένα τελικό αποτέλεσμα σύμφωνα με τον τελεστή υποβίβασης που δίνει ο χρήστης.

1.5 Δομή του Κειμένου

Συνοπτικά, τα κεφάλαια της εργασίας αυτής, παρουσιάζονται ως εξής:

- Κεφάλαιο 2: Ανάλυση της δομής των συσκευών γραφικής επεξεργασίας και συγκεκριμένα όσων βασίζονται στο μοντέλο CUDA.
- Κεφάλαιο 3: Παρουσίαση του προγραμματιστικού μοντέλου OpenMP. Αναφορά στην υποστήριξη των devices και περιγραφή του μεταφραστή OMPi και του CUDA device.
- Κεφάλαιο 4: Περιγραφή του σχεδιασμού και της υλοποίησης του μηχανισμού υποβίβασης στο CUDA device.
- Κεφάλαιο 5: Εκτέλεση και πειραμάτων και σύγκριση αποτελεσμάτων με άλλους μεταφραστές.
- Κεφάλαιο 6: Σύνοψη της διπλωματικής εργασίας, συμπεράσματα και αναφορά σε μελλοντικές βελτιστοποιήσεις του μεταφραστή OMPi.

Κεφάλαιο 2.
Συσκευές Γραφικής Επεξεργασίας και CUDA

2.1 Εισαγωγή

Μια από τις πιο σημαντικές εξελίξεις στο υλικό των υπολογιστικών συστημάτων ήταν η μετάβαση των συσκευών γραφικής επεξεργασίας, από συσκευές με συγκεκριμένες λειτουργίες γραφικού σκοπού σε συσκευές ικανές να εκτελέσουν και υπολογισμούς γενικού σκοπού.
Το μεγάλο ερευνητικό ενδιαφέρον που βλέπουμε σήμερα για τις GPU προέρχεται από το γεγονός ότι προσφέρουν πολύ καλύτερες αποδόσεις σε σχέση με τα παραδοσιακά συστήματα που βασίζονται σε CPU. Σε αντίθεση με τις παραδοσιακές CPU, οι οποίες έχουν σχεδιαστεί για να εξάγουν όσο το δυνατόν περισσότερο παραλληλισμό και απόδοση από σειριακά προγράμματα, οι GPU έχουν σχεδιαστεί για να εκτελούν αποτελεσματικά εγγενή παράλληλα προγράμματα. Συγκεκριμένα, οι GPU υπερέχουν στην περίπτωση προγραμμάτων που έχουν εγγενή παραλληλισμό σε επίπεδο δεδομένων, όπως για παράδειγμα η εκτέλεση υπολογισμών στα στοιχεία ενός πίνακα ή ενός διανύσματος.

2.2 Προγραμματισμός Συσκευών Γραφικής Επεξεργασίας

Για τον προγραμματισμό των μονάδων γραφικής επεξεργασίας, διατίθενται διάφορα προγραμματιστικά μοντέλα που στοχεύουν σε συγκεκριμένη αρχιτεκτονική. Παραδείγματα τέτοιων μοντέλων είναι η

CUDA [1] (Compute Unified Device Architecture) και η OpenCL [2] (Open Computing Language). Συγκεκριμένα, ο προγραμματιστής συγγράφει εφαρμογές στη γλώσσα προγραμματισμού C/C++, οι οποίες αποτελούνται από τμήματα κώδικα που αφορούν τον κύριο επεξεργαστή και τμήματα που αφορούν την μονάδα γραφικής επεξεργασίας. Ο κώδικας που αφορά τη μονάδα GPU δομείται σε συναρτήσεις οι οποίες ονομάζονται υπολογιστικοί πυρήνες (compute kernels) και περιλαμβάνουν τους ωφέλιμους υπολογισμούς. Ο κώδικας που αφορά τον κύριο επεξεργαστή, είναι υπεύθυνος για την προετοιμασία της μονάδας GPU ώστε να εκτελέσει τους υπολογιστικούς πυρήνες. Συγκεκριμένα, η προετοιμασία αφορά στην εκκίνηση και αρχικοποίηση της μονάδας GPU, τη μεταφορά δεδομένων από/προς αυτή, την εκτέλεση του υπολογιστικού πυρήνα (kernel) καθώς επίσης και τον τερματισμό της.

Στην παρούσα εργασία ασχολούμαστε μόνο με συσκευές γραφικής επεξεργασίας τύπου CUDA, οι οποίες αποτελούν και τη δημοφιλέστερη και πιο ανεπτυγμένη πλατφόρμα προγραμματισμού GPUs. Παρ' όλα αυτά οι αλγόριθμοι και οι τεχνικές που χρησιμοποιούμε μπορούν εύκολα, με τις κατάλληλες τροποποιήσεις να εφαρμοστούν και σε συσκευές που χρησιμοποιούν το μοντέλο OpenCL.

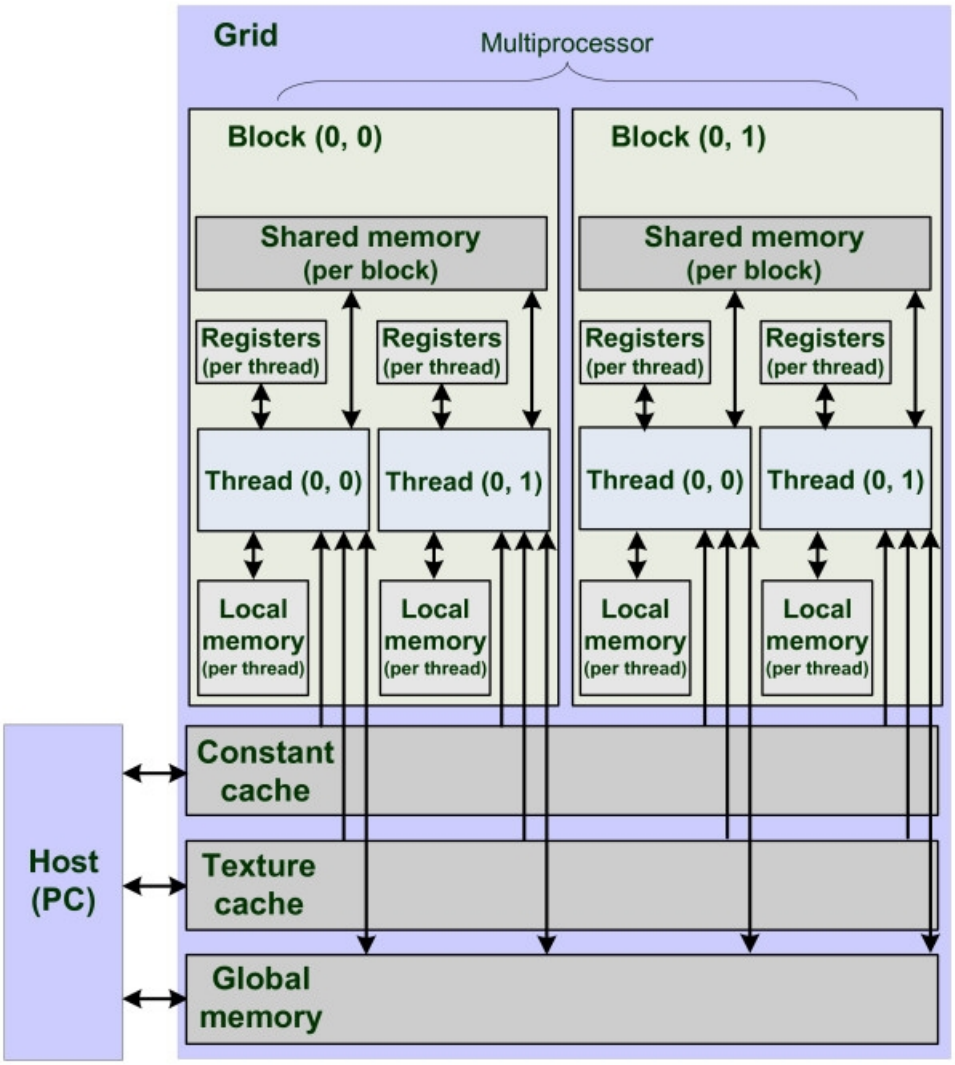
2.3 Το Μοντέλο CUDA

Με τον όρο CUDA συχνά αναφερόμαστε δύο έννοιες. Η πρώτη είναι η πλατφόρμα παράλληλου υπολογισμού που υποστηρίζουν οι μονάδες γραφικής επεξεργασίας της NVIDIA και επιτρέπει την χρήση των πυρήνων που διαθέτουν για την εκτέλεση υπολογισμών γενικού σκοπού. Η δεύτερη έννοια αναφέρεται στην προγραμματιστική διεπαφή, η οποία αποτελείται από το τμήμα κώδικα του host και το τμήμα κώδικα συσκευής και επιτρέπει σε ένα πρόγραμμα να χρησιμοποιήσει την πλατφόρμα.

2.3.1 Η Αρχιτεκτονική

Η πλατφόρμα CUDA διαθέτει δική της οργάνωση συστήματος και μνήμης (σχήμα 2.1 που πάρθηκε από το CUDA toolkit[1]) που διαφέρει αρκετά από αυτή των κλασικών επεξεργαστών.

Κάθε μονάδα γραφικής επεξεργασίας NVIDIA αποτελείται από πολυνηματικούς επεξεργαστές ροής (Streaming Multiprocessors). Το εσωτερικό των επεξεργαστών ροής αποτελείται από πυρήνες (cores). Κάθε πυρήνας εκτελεί το δικό του νήμα (thread), το οποίο αποτελεί την βασική οντότητα εκτέλεσης.



Σχήμα 2.1: Η οργάνωση μιας μονάδας γραφικής επεξεργασίας NVIDIA

Στην CUDA οι παραπάνω όροι αντιστοιχίζονται με νέους διότι στο φυσικό επίπεδο κάθε μονάδα γραφικής επεξεργασίας διαφέρει από τις υπόλοιπες. Έτσι λοιπόν είναι απαραίτητη η θέσπιση μίας κοινής ονοματολογίας, που καλύπτει όλες τις μονάδες γραφικής επεξεργασίας και προσφέρει ομοιομορφία.

Έτσι κατά την διεπαφή ολόκληρη η μονάδα GPU ονομάζεται πλέγμα (grid). Το πλέγμα χωρίζεται σε περαιτέρω τμήματα (blocks) τα οποία ουσιαστικά αντιστοιχίζονται στους επεξεργαστές ροής. Τέλος κάθε τμήμα περιλαμβάνει μία ομάδα από νήματα και κάθε νήμα εκτελείται από έναν φυσικό πυρήνα της μονάδα γραφικής επεξεργασίας.

Όσον αφορά τα είδη μνήμης, αυτά διατηρούν την ονομασία τους. Κάθε επεξεργαστής ροής διαθέτει την δική του ιδιωτική μνήμη η οποία δομείται σε τρία επίπεδα και είναι προσβάσιμη από τους πυρήνες του κάθε block.

- **Κοινόχρηστη μνήμη (Shared memory)**: Το πρώτο επίπεδο μνήμης είναι η κοινόχρηστη μνήμη, η οποία χρησιμοποιείται για αποθήκευση δεδομένων τα οποία θέλουμε να είναι προσβάσιμα από όλους τους πυρήνες ενός επεξεργαστή ροής. Μέσω της κοινόχρηστης μνήμης, οι πυρήνες ενός επεξεργαστή ροής μπορούν να επικοινωνούν και να συγχρονίζουν την εκτέλεσή τους.
- **Σταθερή μνήμη (Constant memory)**: Το δεύτερο επίπεδο μνήμης είναι η σταθερή μνήμη, η οποία χρησιμοποιείται για δεδομένα που δεν πρόκειται να μεταβληθούν. Ο τύπος μνήμης αυτός δίνει την δυνατότητα μόνο για ανάγνωση των δεδομένων.
- **Μνήμη υφής (Texture memory)**: Το τρίτο επίπεδο μνήμης είναι η μνήμη υφής, μια ποσότητα μνήμης μόνο για ανάγνωση, η οποία μπορεί να βελτιώσει σημαντικά τις επιδόσεις του προγράμματος και να μειώσει το κόστος προσπελάσεων μνήμης, όταν οι αναγνώσεις μνήμης επαναλαμβάνονται με κάποιο μοτίβο.

Τέλος υπάρχει και καθολική **μνήμη (global memory)** η οποία είναι προσβάσιμη από όλους τους επεξεργαστές ροής είναι πιο αργή συγκριτικά με τους άλλους τύπους μνήμης και χρησιμοποιείται κατά κύριο λόγω για την μεταφορά δεδομένων από και προς την μονάδα γραφικής επεξεργασίας.

2.3.2 Η Διεπαφή Χρόνου Εκτέλεσης CUDA Runtime

Η προγραμματιστική διεπαφή CUDA runtime είναι μια επέκταση της γλώσσας προγραμματισμού C, που επιτρέπει στους προγραμματιστές να εκμεταλλευτούν τα ετερογενή υπολογιστικά συστήματα που αποτελούνται τόσο από επεξεργαστικές μονάδες όσο και από παράλληλες μονάδες GPU. Ονομάζουμε τη διεπαφή αυτή CUDA C ή απλά CUDA. Κάθε μονάδα GPU υποστηρίζει στον δικό της βαθμό τη λειτουργικότητα που προσφέρεται από το μοντέλο CUDA. Ο βαθμός υποστήριξης εξαρτάται από την υπολογιστική έκδοση (compute capability) της κάθε μονάδας. Νεώτερες μονάδες GPU διαθέτουν υψηλότερη υπολογιστική έκδοση και υποστηρίζουν πιο εξειδικευμένες λειτουργίες, ενώ παλαιότερες μονάδες GPU υποστηρίζουν βασικότερες λειτουργίες.

Ένα πρόγραμμα σε CUDA C αποτελείται από ένα συνδυασμό σειριακών και παράλληλων εκτελέσεων. Οι σειριακοί υπολογισμοί εκτελούνται στον host, ενώ οι παράλληλοι στα νήματα της μονάδας GPU,

```
1    __global__ void MatrixAdd( int** A, int** B, int** C)
2    {
3      int i = threadIdx.x;
4      int j = blockIdx.x;
5
6      C[i][j] = A[i][j] + B[i][j];
7    }
8
9    int main()
10   {
11     int i, j;
12     int numberOfThreads = 1024;
13     int numberOfBlocks = 8;
14
15     int **h_A, **h_B, **h_C, **d_A, **d_B, **d_C;
16
17     for(i=0; i<numberOfThreads; i++)
18             for(j=0; j<numberOfBlocks; j++)
19             {
20                     h_A[i][j] = rand()%10;
21                     h_B[i][j] = rand()%10;
22
23             }
24     /* allocate device memory for d_A, d_B, d_C */
25     /* copy h_A, h_B to d_A, d_B */
26     MatrixAdd<<numberOfBlocks,numberOfThreads>>(d_A, d_B, d_c);
27     /* copy d_C to h_C */
28     return 0;
29   }
```

Σχήμα 2.2: Παράδειγμα προγράμματος CUDA C

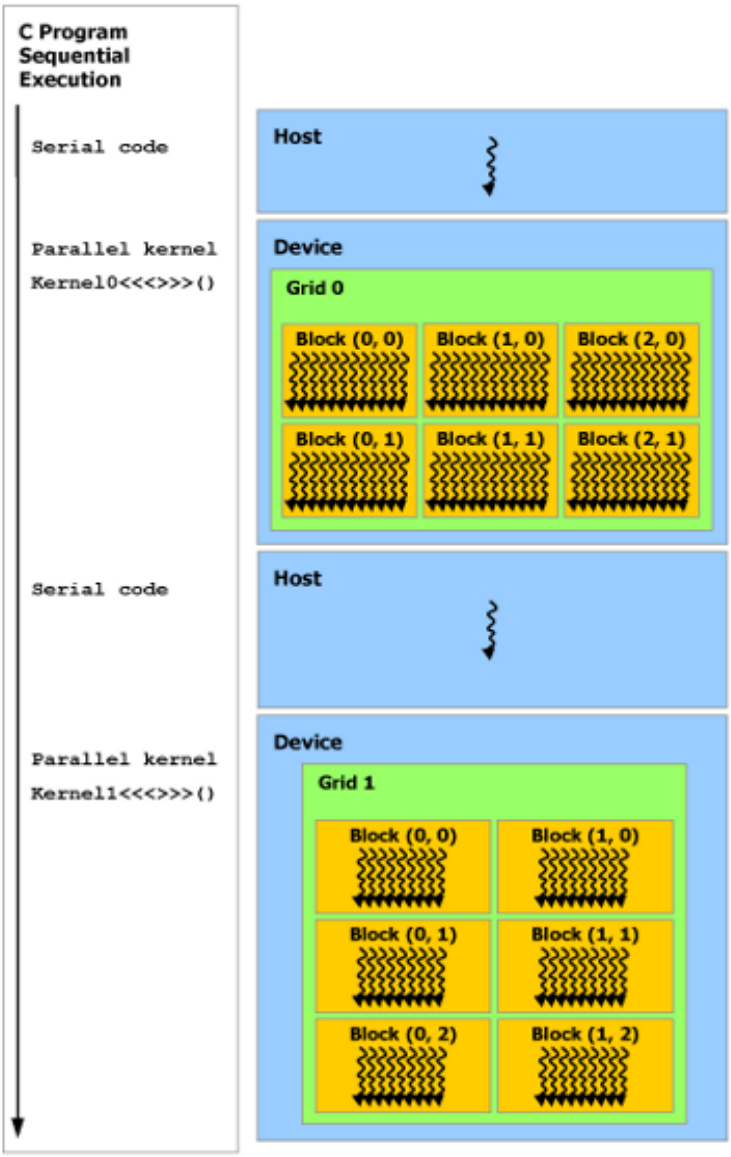
με την μορφή kernels. Ο κώδικας πυρήνα φορτώνεται στη μονάδα GPU μέσω ειδικών κλήσεων χρόνου εκτέλεσης, που ορίζονται από τη διεπαφή. Ο προγραμματιστής συγγράφει έναν κώδικα πυρήνα σαν να προορίζεται για εκτέλεση από ένα νήμα. Συνεπώς, πρέπει να είναι γενικευμένος αρκετά ώστε να μπορεί να εκτελεστεί ορθά από όλα τα νήματα. Η διαφοροποίηση των νημάτων γίνεται με ειδικά αναγνωριστικά, τα οποία είναι προσβάσιμα από τον κώδικα κατά τη διάρκεια της εκτέλεσης.

Ένα παράδειγμα προγράμματος CUDA C παρατίθεται στο σχήμα 2.2, το οποίο υλοποιεί την πρόσθεση πινάκων. Η εκτέλεση ξεκινά από την κλήση της συνάρτησης main όπου οι πίνακες αρχικοποιούνται σειριακά από το κύριο νήμα του host. Έπειτα δεσμεύεται μνήμη στην GPU για την αποθήκευση κάθε πίνακα και αντιγράφονται στην GPU οι πίνακες d_A και

d_B μέσω ειδικών κλήσεων του Runtime API. Αμέσως μετά στην γραμμή 26 καλείται το kernel MatrixAdd. Στο σημείο αυτό, γίνονται όλες οι απαραίτητες διαδικασίες ώστε να φορτωθεί η συνάρτηση MatrixAdd στη μονάδα GPU. Μέσα στα τριπλά σύμβολα ανισότητας, ο προγραμματιστής έχει τη δυνατότητα να ορίσει το πλήθος των blocks, καθώς και το πλήθος των νημάτων που θα συμμετέχουν στην εκτέλεση. Στη συγκεκριμένη περίπτωση, θα δημιουργηθούν 8 blocks που αποτελούνται από 1024 νήματα.
Μιας και το πλήθος των νημάτων ισούται με το μέγεθος του πίνακα, κάθε νήμα θα υπολογίσει μόνο ένα επιμέρους άθροισμα της πρόσθεσης διανυσμάτων. Το τελικό αποτέλεσμα, θα είναι ένας πίνακας C , για τον Κάθε νήμα, όταν εκτελέσει τον κώδικα της MatrixAdd, αρχικά θα βρει το αναγνωριστικό του (ID), με τη χρήση της δομής threadIdx καθώς και το αναγνωριστικό του block στο οποίο ανήκει με την χρήση της δομής blockIdx. οποίο ισχύει C= A+B. Ο πίνακας αυτός βρίσκεται στη διεύθυνση d_C και συνεπώς θα πρέπει να αντιγραφεί στη μνήμη του host, κάτι που συμβαίνει μέσω ειδικών κλήσεων μνήμης, στη γραμμή 27. Για λόγους απλότητας του παραδείγματος, ο κώδικας μεταφορών και
δέσμευσης μνήμης έχει παραληφθεί και έχει αντικατασταθεί με τα σχετικά σχόλια (γραμμές 24, 25 και 27). Ο προσδιοριστής (specifier) __global__ συμβολίζει ότι το kernel MatrixAdd θα κληθεί από τον host, ώστε να
εκτελεστεί στην μονάδα GPU. Ακόμα υπάρχουν οι προσδιοριστές __device__ και __host__ , οι οποίοι συμβολίζουν ότι μια συνάρτηση μπορεί να κληθεί μόνο από κώδικα συσκευής για εκτέλεση στη συσκευή και μόνο από κώδικα host, για εκτέλεση στον host, αντίστοιχα.

Τα προγράμματα αυτά περιλαμβάνουν μεγάλες περιοχές σειριακού κώδικα, με ενδιάμεσες παράλληλες εκτελέσεις στη μονάδα GPU (σχήμα 2.3 που πάρθηκε από το CUDA toolkit[1]). Το μοτίβο αυτό επαναλαμβάνεται όσες φορές κρίνει απαραίτητο ο προγραμματιστής, για την επιτάχυνση των υπολογισμών.

Τέλος αξίζει να σημειωθεί ότι αυτός ο τρόπος προγραμματισμού μπορεί αποθαρρύνει έναν νέο προγραμματιστή να εκμεταλλευτεί τις επιδόσεις της αρχιτεκτονικής CUDA λόγω της δυσκολίας του ετερογενούς μοντέλου που προτείνει CUDA και της αποστήθισης των ιδιαιτεροτήτων και των συμβάσεων της γλώσσας προγραμματισμού CUDA C.



Σχήμα 2.3: Ροή εκτέλεσης προγράμματος σε CUDA C

Κεφάλαιο 3.
OpenMP και OMPi

3.1 Εισαγωγή στο OpenMP

Το OpenMP [3] (Open Multi-Proccesing) είναι μια διεπαφή που υποστηρίζει παράλληλο προγραμματισμό κοινόχρηστης μνήμης στις γλώσσες προγραμματισμού C, C++ και Fortrant. Αποτελείται από οδηγίες προς τον μεταφραστή, ρουτίνες βιβλιοθήκης και μεταβλητές περιβάλλοντος που επηρεάζουν την εκτέλεση του προγράμματος. Χρησιμοποιεί ένα φορητό, κλιμακώσιμο μοντέλο το οποίο δίνει στους προγραμματιστές μια απλή και ευέλικτη πλατφόρμα για την ανάπτυξη παράλληλων εφαρμογών.

Η πρώτη έκδοση δημοσιεύτηκε τον Οκτώβρη του 1997 για Fortrant και τον επόμενο χρόνο για C και C++. Μέχρι το 2005 το OpenMP όριζε απλούς τρόπους παραλληλοποίησης συχνών βρόγχων στους οποίους το πλήθος των επαναλήψεων είναι γνωστό εκ των προτέρων. Αυτός ο περιορισμός οδήγησε στην υποστήριξη του παραλληλισμού με tasks. Έτσι το 2008 έγινε διαθέσιμη η έκδοση 3.0 που εισήγαγε δυνατότητες πέρα από την παραλληλοποίηση βασικών βρόχων

Η έκδοση 4.0 που παρουσιάστηκε το 2013 έφερε υποστήριξη για επιταχυντές και νέες ιδιότητες όπως το thread affinity. Η έκδοση 5.0 που παρουσιάστηκε το 2018 εισήγαγε υποστήριξη για συστήματα πολυεπίπεδης μνήμης και υποστήριξη για τις τελευταίες εκδόσεις των γλωσσών προγραμματισμού
C, C++ και Fortrant. Η τρέχουσα έκδοση είναι η 5.2 και παρουσιάστηκε τον
Νοέμβριο του 2021.

3.2 Προγραμματισμός με OpenMP

Το OpenMP, όπως αναφέρθηκε, αφορά την παραλληλία με χρήση νημάτων σε συστήματα κοινής μνήμης και στηρίζεται στο μοντέλο fork-join. Το αρχικό νήμα (initial thread) εκτελεί τον κώδικα σειριακά μέχρι να συναντήσει μια παράλληλη περιοχή. Τότε δημιουργείται μία ομάδα νημάτων, στην οποία συμμετέχει και το αρχικό νήμα ως αρχηγός (master), η οποία εκτελεί παράλληλα την περιοχή και όταν ολοκληρώσει επιστρέφει τον έλεγχο στο αρχικό νήμα για να συνεχίσει τη σειριακή εκτέλεση. Η διαδικασία αυτή μπορεί να επαναληφθεί πολλές φορές ανάλογα με το πλήθος των παράλληλων περιοχών που έχει ορίσει ο προγραμματιστής στην εφαρμογή του.

3.2.1 Οδηγίες OpenMp για C/C++

Μια οδηγία στο OpenMP συντάσσεται ως εξής (σχήμα 3.1):

- Ξεκινά υποχρεωτικά με την φράση #pragma omp
- Ακολουθεί το όνομα της οδηγίας (directive name). Η οδηγία αυτή μπορεί να αφορά την δημιουργία παράλληλης ομάδας, την επίτευξη συγχρονισμού, εκτέλεση κώδικα σε συσκευή και πολλά άλλα.
- Προαιρετικά, μπορεί να υπάρχουν φράσεις οδηγιών (clause) οι οποίες ορίζουν τις συνθήκες υπό τις οποίες θα εκτελεστεί η οδηγία. Εάν δεν έχει τοποθετηθεί καμία φράση , τότε οι συνθήκες αυτές καθορίζονται στον χρόνο εκτέλεσης του προγράμματος.

```
#pragma omp directive-name [clause[ [,] clause]...] new-line
```

Σχήμα 3.1: Γενική σύνταξη οδηγίας OpenMP

3.2.2 Περιοχές Παραλληλίας

Η πιο σημαντική οδηγία στο OpenMP είναι η parallel, η οποία είναι υπεύθυνη για τη δημιουργία των νημάτων που θα εκτελέσουν μία παράλληλη περιοχή. Έτσι όταν το κύριο νήμα συναντήσει μια οδηγία parallel, δημιουργεί τα υπόλοιπα και τους αναθέτει την εργασία που εκείνα θα πρέπει να εκτελέσουν παράλληλα. Λόγω της απρόβλεπτης συμπεριφοράς στην εκτέλεση των νημάτων το OpenMP υπονοεί έναν barrier στο τέλος της παράλληλης περιοχής για συγχρονισμό των νημάτων. Μετά το τέλος της παράλληλης περιοχής τα νήματα καταστρέφονται και μένει μόνο το αρχικό νήμα.

3.2.3 Οδηγίες Διαμοιρασμού Εργασίας

Ο προγραμματιστής μπορεί, εντός μίας παράλληλης περιοχής, να επιλέξει των τρόπο με τον οποίο θα διαμοιραστεί η εργασία μεταξύ των νημάτων μέσω των εξής εντολών:

- **for :** Αφορά την κατανομή των επαναλήψεων ενός βρόγχου for στα νήματα
  sections : Αφορά στον καταμερισμό των τμημάτων που δηλώνονται, ώστε αυτά να διαμοιραστούν σε διαφορετικά νήματα.
- **single/master :** Το πρώτο νήμα που συναντήσει την εντολή single ή το νήμα αρχηγός στην περίπτωση της εντολής master εκτελεί το τμήμα κώδικα που ακολουθεί ενώ τα υπόλοιπα το προσπερνούν.

```
#include <omp.h>

int main()
{
        int a[512], b[512], c[512];
        int size = 512;

        int i;

        #pragma omp parallel for
                for(i=0; i<size; i++)
                        c[i] = a[i] + b[i];
        return 0;
}
```

Σχήμα 3.2: Παράδειγμα χρήσης οδηγίας parallel for

Στο τέλος των περιοχών διαμοιρασμού εργασίας υπονοείται ένα φράγμα (barrier), ακριβώς όπως και στην περίπτωση των παράλληλων περιοχών, με σκοπό τον συγχρονισμό των νημάτων. Το φράγμα αυτό μπορεί να παραλειφθεί με την τοποθέτηση της φράσης nowait στην οδηγία. Στο σχήμα 3.2 παρατίθεται παράδειγμα χρήσης των οδηγιών parallel for. Στο παράδειγμα αυτό ορίζονται τρεις πίνακες A, B και C μεγέθους 512 στοιχείων και με την χρήση της οδηγίας **parallel for** παραλληλοποιείται ο βρόχος που εκτελεί την πρόσθεση των i-οστών στοιχείων των πινάκων A και B, τοποθετώντας το αποτέλεσμα στην i-οστή θέση του πίνακα C.

3.2.4 Οδηγίες Συγχρονισμού

Οι οδηγίες συγχρονισμού χρησιμοποιούνται για το συγχρονισμό των νημάτων κατά την παράλληλη εκτέλεση ώστε να εξασφαλίζεται η συνέπεια των δεδομένων και η σωστή λειτουργία του προγράμματος. Οι βασικότερες είναι:

- **atomic :** Χρησιμοποιείται για αδιαίρετη πράξη σε μια μεταβλητή.Εξασφαλίζει ότι στη μεταβλητή που πρόκειται να τροποποιηθεί δεν παρεμβάλλεται κάποιο άλλο νήμα.
- **barrier :** Ορίζει ένα φράγμα συγχρονισμού των νημάτων στο σημείο που τοποθετείται. Τα νήματα που καταφτάνουν στο σημείο αυτό περιμένουν τα υπόλοιπα της ομάδας, ώστε να συνεχιστεί η εκτέλεση με τις τιμές των κοινόχρηστων μεταβλητών ίδιες για όλα τα νήματα.
- **critical :** Παρόμοια με την οδηγία atomic αλλά για τμήμα κώδικα που πρέπει να εκτελεστεί από ένα νήμα τη φορά. Τα νήματα που φτάνουν στην κρίσιμη περιοχή περιμένουν την εκτέλεση από το νήμα που βρίσκεται ήδη σε αυτήν.

3.2.5 Tasks

Με την χρήση της οδηγίας task ο προγραμματιστής μπορεί να ορίσει μία περιοχή η οποία μπορεί να εκτελεστεί από οποιοδήποτε νήμα, σε οποιαδήποτε χρονική στιγμή. Συγκεκριμένα ο κώδικας που βρίσκεται μέσα στην οδηγία αυτή εκτελείται όταν είναι διαθέσιμος κάποιος πυρήνας ή μπορεί να εκτελεστεί ακόμα και παράλληλα και με κάποια άλλη περιοχή. Αυτός ο τρόπος προγραμματισμού ταιριάζει σε εφαρμογές όπου η κατανομή φόρτου ανάμεσα στα νήματα είναι άνιση. Τέλος δίνεται η δυνατότητα εναλλαγής των tasks ανάμεσα σε νήματα, δηλαδή κάποιο νήμα να ξεκινά την εκτέλεση και σε κάποιο σημείο να την αναλαμβάνει κάποιο άλλο.

3.2.6 Devices

Η ανάπτυξη εφαρμογών για μονάδες γραφικής επεξεργασίας μπορεί να γίνει εξαιρετικά δύσκολη καθώς ο προγραμματιστής με πρέπει να χειριστεί πολύ προσεκτικά την διαδικασία μεταφοράς δεδομένων από και προς την μονάδα GPU και να ελέγξει τον τρόπο με τον οποίο θα εκτελεστούν οι υπολογισμοί. Προκειμένου να μειωθεί η δυσκολία του προγραμματισμού των GPUs το πρότυπο OpenMP από την έκδοση 4.0 υποστηρίζει εκτέλεση σε σύνοδες συσκευές (devices) του συστήματος. Τέτοιες συσκευές αποτελούν οι συνεπεξεργαστές, οι επιταχυντές και οι κάρτες γραφικών. Η CPU παίζει τον ρόλο του host και η συσκευή αναφέρεται ως device. Επειδή οι συσκευές αποτελούν το κεντρικό θέμα στην εργασία αυτή, θα παρουσιάσουμε με μεγαλύτερη λεπτομέρεια τα σχετικά χαρακτηριστικά του OpenMP στις ενότητες που ακοκουθούν.

3.2.6.1 Η Οδηγία Target

Η οδηγία αυτή μας επιτρέπει να εκτελέσουμε κώδικα σε συσκευή. Οι εντολές του προγράμματος που βρίσκονται μέσα στο μπλοκ κώδικα που την ακολουθεί εκτελούνται στην συσκευή και όχι στον κεντρικό επεξεργαστή.
Η οδηγία target μας επιτρέπει την μεταφορά δεδομένων μεταξύ host και device με τους εξής τρόπους:

- Με την φράση map(to : <variable_list>) η τιμή μιας ή περισσοτέρων μεταβλητών αντιγράφονται από το βασικό σύστημα στην μνήμη στης συσκευής πριν την εκτέλεση του τμήματος κώδικα.
- Με την φράση map(from : <variable_list>) η τιμή μιας ή περισσοτέρων μεταβλητών αντιγράφονται από την μνήμη της συσκευής πίσω στην μνήμη του κύριου συστήματος μετά την εκτέλεση του τμήματος κώδικα.
- Με την φράση map(tofrom : <variable_list>) επιτυγχάνουμε το συνδυασμό των δύο παραπάνω οδηγιών.
- Με την φράση map(alloc: <variable list>) δημιουργείται μια αντιστοιχία δεδομένων ανάμεσα σε βασικό σύστημα και συσκευή, αλλά δεν πραγματοποιείται καμία μεταφορά δεδομένων.

Στο σχήμα 3.3 δίνεται παράδειγμα χρήσης της οδηγίας target, ορίζονται τρεις πίνακες A, B και C μεγέθους 512 στοιχείων οι οποίοι με χρήση της φράσης map to αντιγράφονται στην μνήμη της συσκευής οποία αναλαμβάνει την πρόσθεσή τους. Το αποτελέσματα αποθηκεύεται στον

```
#include <omp.h>

int main()
{
        int a[512], b[512], c[512];
        int size = 512;

        int i;

        #pragma omp target parallel for map(to: a[0:size], b[0:size], size) \
          map(from: c[0:size]
                for(i=0; i<size; i++)
                        c[i] = a[i] + b[i];
        return 0;
}
```

Σχήμα 3.3: Παράδειγμα χρήσης οδηγίας target

πίνακα C ο οποίος αντιγράφεται πίσω στην μνήμη του κύριου συστήματος με την φράση map from.

3.2.6.2 Η Οδηγία Target Data

Με την οδηγία target data ορίζουμε μία περιοχή μετακίνησης δεδομένων από και προς τη συσκευή. Κώδικας μπορεί να εκτελεστεί αν υπάρχει οδηγία target φωλιασμένα. Στη διάθεσή του θα έχει και τις μεταβλητές που ορίστηκαν πριν με την οδηγία target data , οι οποίες θα είναι έγκυρες και διαθέσιμες για όλο το μπλοκ κώδικα της οδηγίας. Ο λόγος ύπαρξης αυτής της οδηγίας είναι να βοηθήσει στη μείωση του όγκου των δεδομένων που μεταφέρονται από και προς τη συσκευή όταν υπάρχουν διαδοχικές οδηγίες target που χρησιμοποιούν τα ίδια δεδομένα.

Στο σχήμα 3.4 παρουσιάζεται σύντομο παράδειγμα χρήσης της οδηγίας target data. Στο παράδειγμα αυτό ορίζεται ο πίνακας data. Όπως φαίνεται στις γραμμές 10 και 14 ο πίνακας αυτός χρησιμοποιείται από δύο παράλληλες περιοχές οπότε με την χρήση της οδηγίας target data αυτός θα μεταφερθεί μόνο μία φορά στην μνήμη της συσκευής.

```
1  #include <omp.h>
2
3  int main() {
4        int data[1000];
5
6        #pragma omp target data map(tofrom: data)
7        {
8                #pragma omp target parallel for
9                        for (int i = 0; i < N; i++)
10                               data[i] *= 2;
11
12               #pragma omp target parallel for
13                       for (int i = 0; i < N; i++)
14                               data[i] += 5;
15       }
16       return 0;
17 }
```

Σχήμα 3.4: Παράδειγμα χρήσης οδηγίας taget data

3.2.6.3 Η Οδηγία Declare Target

Η οδηγία declare target δίνει στον προγραμματιστή την δυνατότητα να χρησιμοποιήσει καθολικές μεταβλητές και να καλέσει συναρτήσεις μέσα σε κώδικα που εκτελείται σε συσκευή. Αυτό συμβαίνει δηλώνοντας τις μεταβλητές και τα πρωτότυπα των συναρτήσεων ανάμεσα στις οδηγίες #pragma omp declare target και #pragma omp end declare target.

Στο σχήμα 3.5 παρουσιάζεται παράδειγμα προγράμματος που χρησιμοποιεί την εντολή declare target για να δηλώσει την καθολική μεταβλητή someVariable και το πρωτότυπο της συνάρτησης foo έτσι ώστε να χρησιμοποιηθούν από την συσκευή.

```
# pragma omp declare target
int foo(int x, int y )
int someVariable = 5;
#pragma omp declare end
```

Σχήμα 3.5: Παράδειγμα χρήσης οδηγίας declare target

3.2.6.4 Η Οδηγία Target Update

Γενικά η μεταφορά δεδομένων μεταξύ host και device πραγματοποιείται μόνο στην αρχή και το τέλος του μπλοκ κώδικα που ακολουθεί την οδηγία target. Όμως με την οδηγία target update μπορούμε να συγχρονίσουμε δεδομένα μεταξύ host και device οποιαδήποτε στιγμή το επιθυμούμε.

Ακολουθεί στο σχήμα 3.6 παράδειγμα για την χρήση της οδηγίας target update. Αρχικά η οδηγία target data (γραμμή 5) αντιγράφει τους πίνακες v1 και v2 στην μνήμη της συσκευής. Ο host συναντά την πρώτη περιοχή target και περιμένει την ολοκλήρωσή της. Μετά την ολοκλήρωση της πρώτης περιοχής target ο host εκχωρεί νέες τιμές στους πίνακες v1 και v2 (γραμμή 11). Μετά η συσκευή με χρήση της οδηγίας target update (γραμμή 12) ενημερώνονται οι τιμές των πινάκων τις νέες που εκχώρησε ο host. Στην δεύτερη περιοχή target η συσκευή χρησιμοποιεί τις ενημερωμένες τιμές των v1 και v2.

```
1   void vec_mult(float *p, float *v1, float *v2, int N)
2   {
3       int i;
4       init(v1, v2, N);
5       #pragma omp target data map(to: v1[:N], v2[:N]) map(from: p[0:N])
6       {
7               #pragma omp target
8               #pragma omp parallel for
9                       for (i=0; i<N; i++)
10                              p[i] = v1[i] * v2[i];
11              init_again(v1, v2, N);
12              #pragma omp target update to(v1[:N], v2[:N])
13              #pragma omp target
14              #pragma omp parallel for
15                      for (i=0; i<N; i++)
16                              p[i] = p[i] + (v1[i] * v2[i]);
17      }
18      output(p, N);
19  }
```

Σχήμα 3.6: Παράδειγμα χρήσης οδηγίας target update

3.3 Reductions στο OpenMP

Οι παράλληλες εργασίες συχνά παράγουν αποτελέσματα τα οποία πρέπει να αθροιστούν ή να συνενωθούν με κάποιο άλλο τρόπο. Προκειμένου να ξεπεραστεί το πρόβλημα της συνέπειας των δεδομένων που προκύπτει από το ανταγωνισμό των νημάτων να επέμβουν στη μεταβλητή που συνενώνει τα δεδομένα το πρότυπο OpenMP χρησιμοποιεί τη λειτουργία reduction.

Η λειτουργία αυτή ενεργοποιείται με φράση σε κατάλληλες οδηγίες και συντάσσεται ως εξής:

reduction(<operator> : <variable_list>)

όπου operator είναι ο τελεστής υποβίβασης ο οποίος για αριθμητικές μεταβλητές μπορεί να είναι:

- **+** : Για την εύρεση αθροίσματος τιμών
- ***** : Για την εύρεση γινόμενου
- **-** : Για την εύρεση διαφοράς τιμών
- **max** : Για την εύρεση της μέγιστης τιμής
- **min** : Για την εύρεση της ελάχιστης τιμής

και για λογικές μεταβλητές μπορεί να είναι:

- **&** : Για την εύρεση του δυαδικού ΚΑΙ (AND) όλων των τιμών
- **&&** : Για την εύρεση του λογικού ΚΑΙ (AND) όλων των τιμών
- **|** : Για την εύρεση του δυαδικού Η (OR) όλων των τιμών
- **||** : Για την εύρεση του λογικού Η (OR) όλων των τιμών
- **^** : Για την εύρεση του δυαδικού αποκλειστικού Η (XOR) όλων των τιμών

και variable_list είναι η μεταβλητή ή οι μεταβλητές υποβίβασης, δηλαδή η μεταβλητή ή οι μεταβλητές που συγκεντρώνεται το τελικό αποτέλεσμα και μπορούν να είναι απλές μεταβλητές οποιουδήποτε τύπου ή πίνακας οποιουδήποτε τύπου. Στη δεύτερη περίπτωση, η υποβίβαση γίνεται σε κάθε στοιχείο του πίνακα ξεχωριστά.

Όταν το OpenMP συναντήσει μία οδηγία με τη φράση reduction εκτελεί τα εξής βήματα:

- Δημιουργεί μια τοπική μεταβλητή υποβίβασης σε κάθε νήμα και αναγνωρίζει τον τελεστή υποβίβασης
- Κάθε νήμα εκτελεί το κώδικά του αποθηκεύοντας τιμές στην τοπική του μεταβλητή

```
#include <omp.h>
#define N 10000000

double pi = 0.0, w = 1.0/N;

int main() {
        int i;

        pragma omp target parallel for map(to: w) reduction(+: pi)
                for (i=0; i<N; i++)
                        pi += 4*w / (1 + (i*0.5)*(i*0.5)*w*w);
        return 0;
}
```

Σχήμα 3.7: Παράδειγμα χρήσης φράσης reduction

- Στο τέλος της περιοχής της οδηγίας οι τιμές των τοπικών μεταβλητών υποβίβασης συνενώνονται στην καθολική μεταβλητή χρησιμοποιώντας τον τελεστή υποβίβασης.

Στο σχήμα 3.7 δίνεται ενδεικτικό πρόγραμμα που χρησιμοποιεί την λειτουργία reduction. Το συγκεκριμένο πρόγραμμα υπολογίζει σε συσκευή τη σταθερά π = 3,14... μέσω μιας απλής αριθμητική ολοκλήρωσης, κάνοντας χρήστη της οδηγίας parallel for και της φράσης reduction(+:pi). Θα δημιουργηθεί μία τοπική μεταβλητή pi για το κάθε νήμα και στο τέλος θα αθροιστούν στην μεταβλητή pi του πυρήνα, η οποία με το τέλος της οδηγίας target θα επιστραφεί στο κυρίως πρόγραμμα θέτοντας της τιμή της καθολικής μεταβλητής pi.

3.4 Περιγραφή του Μεταφραστή OMPi και της Συσκευής cudadev

Ο OMPi είναι ένας παραλληλοποιητικός μεταφραστής ανοιχτού κώδικα για προγράμματα που έχουν γραφτεί σε γλώσσα C και υποστηρίζει το πρότυπο OpenMP. Αποτελείται από δύο τμήματα:

- Το μεταφραστή (compiler) που δέχεται ως είσοδο προγράμματα γραμμένα σε γλώσσα προγραμματισμού C με εντολές OpenMP. Ο συντακτικός αναλυτής διατρέχει το πρόγραμμα και παράγει το συντακτικό δέντρο, το οποίο στη συνέχεια επεξεργάζεται κατάλληλα ώστε να αφαιρέσει τις οδηγίες OpenMP. Παράγει ως έξοδο προγράμματα σε C όπου οι οδηγίες OpenMP έχουν αντικατασταθεί

με κλήσεις συναρτήσεων της βιβλιοθήκης χρόνου εκτέλεσης. Τέλος, ο τοπικός μεταφραστής C που είναι εγκατεστημένος στο σύστημα αναλαμβάνει τη δημιουργία του τελικού εκτελέσιμου από τον κώδικα C.

- Τη βιβλιοθήκη χρόνου εκτέλεσης που παρέχει συναρτήσεις για την δημιουργία υπολογιστικών οντοτήτων (όπως νήματα και διεργασίες) που αναλαμβάνουν την εκτέλεση του κώδικα που έχει παραλληλοποιηθεί, συναρτήσεις που διαχειρίζονται κλειδαριές και άλλες βοηθητικές δυνατότητες, καθώς και συναρτήσεις που διαχειρίζονται την επικοινωνία με συσκευές και την εκτέλεση τμημάτων κώδικα σε αυτές.
- Στην περίπτωση μετάφρασης ενός προγράμματος με περιοχές target κάθε περιοχή εξάγεται σε ένα δικό της ξεχωριστό αρχείο, το οποίο ονομάζεται kernel file. Στην συνέχεια το κάθε kernel file μεταφράζεται από τον μεταφραστή της εκάστοτε συσκευής, όπου και παράγονται ενδιάμεσα αρχεία ένα για κάθε kernel. Τέλος ο μεταφραστής C που είναι εγκατεστημένος στο σύστημα είναι υπεύθυνος να μεταγλωττίσει το κομμάτι του host καθώς και τα ενδιάμεσα αρχεία έτσι ώστε να παραχθεί το τελικό εκτελέσιμο αρχείο.

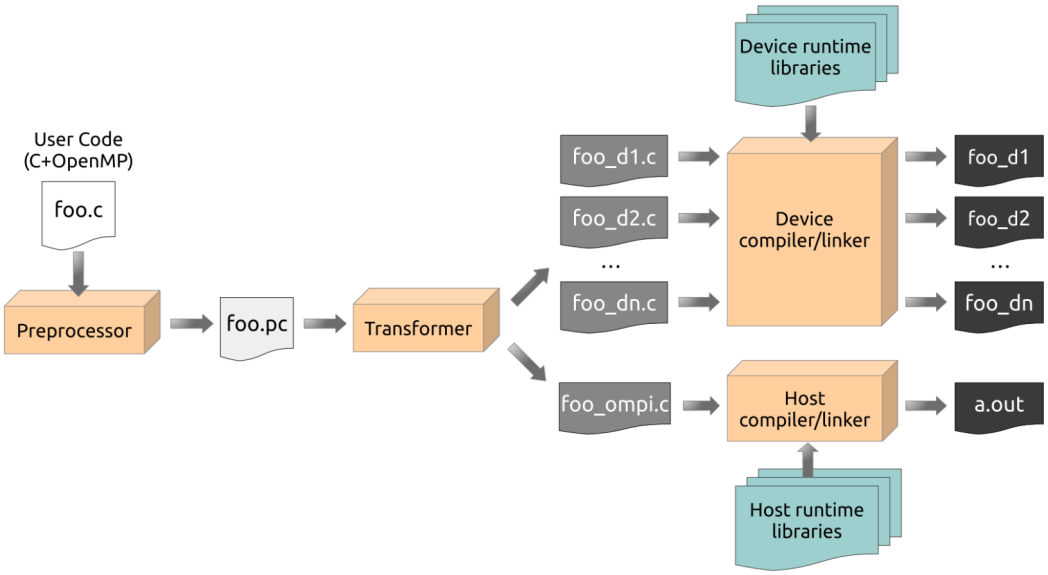
Ο OMPi υποστηρίζει μέχρι στιγμής την συσκευή epiphany3 [4] η οποία στοχεύει στον συνεπεξεργαστή της πλατφόρμας Parallella-16, ενός μίνι-υπολογιστή που αποτελείται από έναν διπύρηνο ARM επεξεργαστή και τον δεκαεξαπύρηνο συνεπεξεργαστή Epithany-III. Επίσης υποστηρίζεται η συσκευή mpinode [5] η οποία προσομοιώνει τους κόμβους ενός cluster ως συσκευές του OpenMP. Ακόμα υλοποιείται συσκευή η οποία υποστηρίζει CUDA, η οποία παρουσιάζεται αναλυτικά παρακάτω. Τέλος ο OMPi βρίσκεται σε διαδικασία επέκτασης των δυνατοτήτων του ώστε να υποστηρίζει κάρτες γραφικής επεξεργασίας τύπου OpenCL.

Στο σχήμα 3.8 παρουσιάζεται η διαδικασία μετάφρασης ενός προγράμματος στον μεταφραστή OMPi.

3.4.1 Cudadev: η Συσκευή CUDA

Η συσκευή CUDA, βασίζεται στην προγραμματιστική διεπαφή CUDA Driver. Το CUDA Driver API [6] αποτελεί τη χαμηλότερου επιπέδου μορφή του CUDA Runtime API, προσφέροντας καλύτερο έλεγχο στην διαχείριση μίας συσκευής CUDA.

Η συσκευή CUDA αποτελείται απο τα τμήματα hostpart και devpart. Η διεπαφή hostpart είναι υπεύθυνη για επικοινωνία του κύριου συστήματος (host) με την μονάδα γραφικής επεξεργασίας ενώ η



Σχήμα 3.8: Η διαδικασία μετάφρασης στον OMPi

βιβλιοθήκη συσκευής devpart περιλαμβάνει το σύνολο των ρουτινών OpenMP, οι οποίες καλούνται από τον κώδικα χρήστη που πρόκειται να φορτωθεί στην συσκευή.

Συγκεκριμένα το hostpart υλοποιεί τις εξής λειτουργίες:

- **Αρχικοποίηση και τερματισμός συσκευής.** Η διεπαφή hostpart ορίζει συναρτήσεις που ανακαλύπτουν όλες τις εγκατεστημένες συσκευές CUDA στο σύστημα, για τις οποίες αποθηκεύει σημαντικές πληροφορίες όπως η έκδοση CUDA, οι προδιαγραφές της συσκευής και αρχικοποιούν το CUDA context, το οποίο αποτελεί την εσωτερική κατάσταση (internal state) της συσκευής. Επίσης μετά την χρήση της συσκευής υλοποιούνται συναρτήσεις που τερματίζουν λειτουργία της συσκευής σωστά.
- **Δέσμευση και αποδέσμευση μνήμης.** Ορίζονται συναρτήσεις οι οποίες πραγματοποιούν δέσμευση και αποδέσμευση μνήμης της συσκευής από τον host.
- **Ανάγνωση και εγγραφή** της μνήμης για την μεταφορά δεδομένων από και προς την συσκευή.
- **Δημιουργία Kernel.** Το κύριο σύστημα έχει τη δυνατότητα να δημιουργήσει έναν kernel και να τον παραμετροποιήσει ανάλογα με τις φράσεις που έχουν χρησιμοποιηθεί στην οδηγία target από τον χρήστη.

  Ο πηγαίος κώδικας του αρχείου περιλαμβάνει τη συνάρτηση πυρήνα, καθώς και ορισμένα αρχεία επικεφαλίδων (header files), για συναρτήσεις και δομές OpenMP που μπορούν να χρησιμοποιηθούν

στο εσωτερικό του. Επίσης για να γλιτώσουμε σημαντικές χρονικές επιβαρύνσεις από την φόρτωση του kernel στην συσκευή κάθε kernel με την πρώτη εμφάνισή του αποθηκεύεται σε έναν μηχανισμό προσωρινής αποθήκευσης kernels έτοιμων-για-φόρτωση (kernel cache).

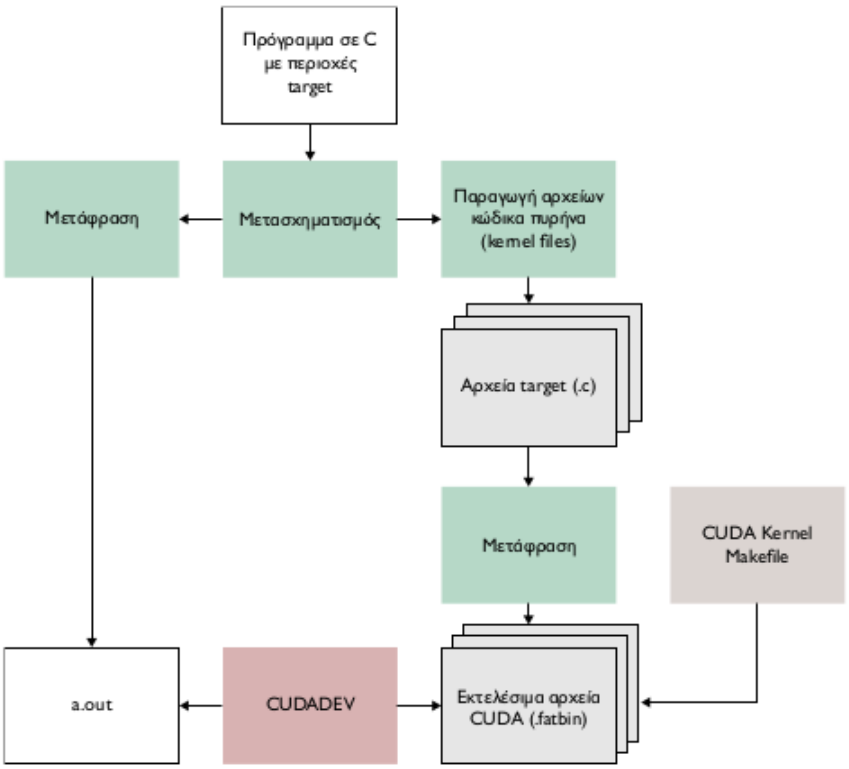
- **Αποθήκευση Παραμέτρων Kernel.** Φορτώνονται στην συσκευή όλα τα δεδομένα που θα χρησιμοποιηθούν από τον kernel καθώς και οι διευθύνσεις τους.
- **Εκτέλεση Kernel.** Αρχικοποιούνται οι παράμετροι εκτέλεσης του kernel όπως ο αριθμός των ομάδων που θα εκτελεστούν καθώς και ο αριθμός νημάτων ανά ομάδα. Ο μεταφραστής υπολογίζει τον βέλτιστο αριθμό των blocks που θα δημιουργηθούν και το πλήθος των νημάτων που θα τα
απαρτίσουν. Η διαδικασία αυτή λαμβάνει χώρα για να αποφευχθεί η δημιουργία μονοδιάστατων πλεγμάτων και block. Με το πέρας της εκτέλεσης, τα νήματα CUDA συγχρονίζονται και οι δομές για τα ορίσματα του kernel αποδεσμεύονται.

Η βιβλιοθήκη συσκευής περιλαμβάνει τις εξής λειτουργίες, για την υποστήριξη του χρόνου εκτέλεσης του kernel:

- **Υλοποίηση οδηγιών διαμοιρασμού εργασίας** όπως την κατανομή των επαναλήψεων ενός βρόγχου for και τον καταμερισμό των τμημάτων (sections).
- **Υλοποίηση μηχανισμών συγχρονισμού** των νημάτων όπως κλειδαριές, atomic operations και μηχανισμούς υλοποίησης συγχρονισμού κρίσιμης περιοχής (critical section).
- **Λήψη τμήματος βρόχου (chunk).** Υλοποιεί τη διαδικασία λήψης ενός τμήματος βρόχου, ο οποίος διαμοιράζεται μέσω της οδηγίας #pragma omp distribute.
- **Αναγνωριστικά (ID).** Η λειτουργικότητα περιλαμβάνει τις συναρτήσεις που αφορούν την εύρεση του πλήθους των νημάτων και ομάδων, καθώς και την ταυτοποίηση των ομάδων και νημάτων.
- **Εργαλεία Μεταγλώττισης Κώδικα Συσκευής.** Η βιβλιοθήκη περιλαμβάνει δύο βασικά συστατικά, τα οποία είναι υπεύθυνα για την μετάφραση των αρχείων συσκευών. Συγκεκριμένα, το πρώτο συστατικό αποτελεί ένα εργαλείο που ακολουθεί την διαδικασία του μεταγλωττιστή nvcc για τη μετάφραση κώδικα CUDA C ενώ το δεύτερο συστατικό είναι
το αρχείο Kernel Makefile, το οποίο χρησιμοποιεί το προαναφερθέν εργαλείο για να μεταγλωττίσει τον εξαγόμενο κώδικα συσκευής target. Συγκεκριμένα το Kernel Makefile είναι υπεύθυνο για την μετάφραση όλων των kernel files σε εκτελέσιμα για τη μονάδα GPU.

Το αποτέλεσμα είναι ένα σύνολο από εκτελέσιμα αρχεία .fatbin. Τέλος, στο πρόγραμμα a.out, έχει αντικατασταθεί κάθε περιοχή target από κώδικα ο οποίος αναλαμβάνει την φόρτωση ενός αρχείου .fatbin στη συσκευή CUDA.

Η βιβλιοθήκη συσκευής του cudadev υλοποιεί το μεγαλύτερο τμήμα των λειτουργιών του OpenMP υποστηρίζοντας τους kernels που εκτελούνται στη συσκευή. Όμως, δεν υπάρχει ακόμα υποστήριξη των λειτουργιών υποβίβασης, κάτι που το εισάγουμε για πρώτη φορά στην εργασία αυτή.



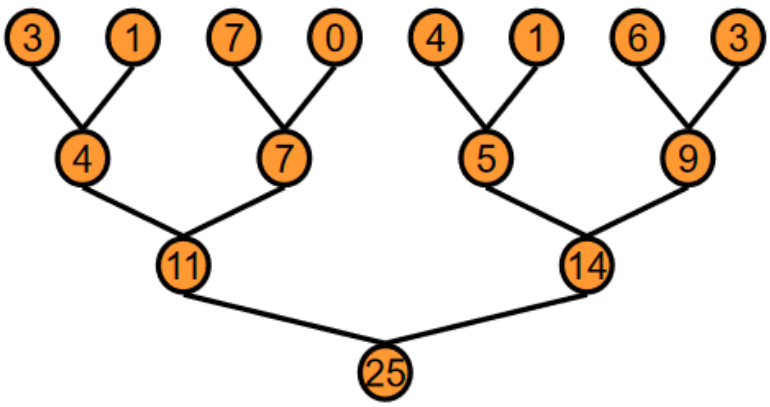
Σχήμα 3.9 : Η Διαδικασία μεταγλώττισης ενός προγράμματος για την συσκευή CUDA

Κεφάλαιο 4.
Σχεδιασμός και Υλοποίηση Reductions στο CUDA Device

4.1 Αλγόριθμοι για Reductions

Η παράλληλη υποβίβαση αναφέρεται σε αλγόριθμους που συνδυάζουν μια σειρά στοιχείων και παράγουν μια συνολική τιμή ως αποτέλεσμα. Τα προβλήματα που είναι κατάλληλα για τέτοιους αλγόριθμους είναι εκείνα που περιλαμβάνουν τελεστές που έχουν προσεταιριστικές ιδιότητες.

Η βασική ιδέα υλοποίησης τέτοιων αλγορίθμων[7] σε επιταχυντές γραφικών CUDA είναι η χρήση πολλών μπλοκ νημάτων για την υποβίβαση ενός μικρού μέρους του τελικού αποτελέσματος. Μέσα σε κάθε μπλοκ νημάτων υλοποιείται αλγόριθμος υποβίβασης με χρήση δενδρικών δομών (σχήμα 4.1) και κοινόχρηστη μνήμη για την επίτευξη επικοινωνίας μεταξύ νημάτων του ίδιου μπλοκ και συγχρονισμού μεταξύ τους.



Σχήμα 4.1: Δενδρική δομή υποβίβασης

Αν και ο βασικός αλγόριθμος φαίνεται απλός, η δυσκολία έγκειται στην εφαρμογή του με αποδοτικό τρόπο. Προκειμένου να σχεδιάσουμε έναν αποδοτικό αλγόριθμο θα πρέπει να κατανοήσουμε τα χαρακτηριστικά απόδοσης (performance characteristics) της CUDA. Στην περίπτωση του προβλήματος της παράλληλης υποβίβασης ο προγραμματιστής θα πρέπει να χρησιμοποιήσει κατάλληλες τεχνικές έτσι ώστε να αποφύγει τα εξής φαινόμενα:

- **Brach divergence**: Το φαινόμενο αυτό εμφανίζεται όταν νήματα που εκτελούνται από τον ίδιο πυρήνα CUDA ακολουθούν διαφορετικά μονοπάτια σε μία συνθήκη ελέγχου. Έτσι μερικά νήματα πρέπει να περιμένουν, παραμένοντας αδρανή, μέχρι την ολοκλήρωση των υπόλοιπων νημάτων. Αυτή η κατάσταση αδράνειας στην οποία επέρχονται τα νήματα έχει ως αποτέλεσμα την μείωση της απόδοσης της μονάδας γραφικής επεξεργασίας CUDA.
- **Memory bank conflict**: Για την επίτευξη υψηλού εύρους ζώνης μνήμης για ταυτόχρονες προσβάσεις, η κοινόχρηστη μνήμη χωρίζεται σε μονάδες μνήμης (banks) ίδιου μεγέθους, οι οποίες μπορούν να προσπελασθούν ταυτόχρονα. Έτσι στην περίπτωση που πολλά νήματα προσπαθήσουν να προσπελάσουν δεδομένα που βρίσκονται σε διαφορετικά banks οι μεταφορές δεδομένων εξυπηρετούνται παράλληλα. Αντίθετα στην περίπτωση που πολλά νήματα προσπαθήσουν να προσπελάσουν δεδομένα που βρίσκονται στο ίδιο bank η πρόσβαση στα δεδομένα θα γίνει σειριακά, με αποτέλεσμα να μειώνεται το εύρος ζώνης της μνήμης.
- **Inactive threads**: Ο αριθμός των νημάτων που μένουν αδρανή μετά από κάθε επανάληψη ενός βρόγχου.

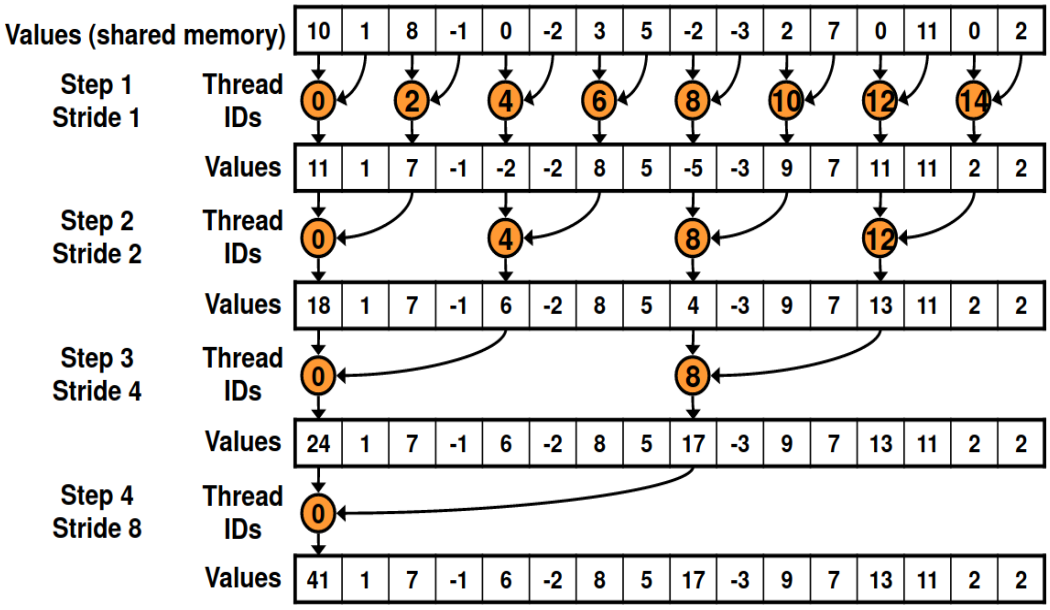
Αλγόριθμος #1: Interleaved Addressing

Στο σχήμα 4.2 παρουσιάζεται ο αλγόριθμος Interleaved Addressing και στο σχήμα 4.3 (που πάρθηκε από την εργασία "Optimizing parallel reduction in CUDA." του Mark Harris [7]) παρουσιάζεται παράδειγμα εκτέλεσης του αλγορίθμου Interleaved Addressing.

Στην πρώτη υλοποίηση του αλγόριθμου εμφανίζεται το φαινόμενο του memory bank conflict αφού, λόγω του τρόπου διάσχισης των θέσεων της κοινόχρηστης μνήμης παρατηρούμε ότι πολλά νήματα προσπαθούν να προσπελάσουν την ίδια θέση μνήμης. Επίσης η εντολή **if(tid % (2*s) == 0)** περιέχει τον τελεστή (%) ο οποίος είναι μη αποδοτικός και επίσης εμφανίζεται το φαινόμενο branch divergence. Στην CUDA γενικά ο κάθε streaming multiprocessor εξυπηρετεί 32 νήματα την φορά (το λεγόμενο «warp»), τα οποία έχουν ακολουθιακά αναγνωριστικά νήματος.

```
__global__ void reduce(int *g_idata, int *g_odata){
	extern __shared__ int sdata[];

	//each thread loads one element from global to shared memory
	unsigned int tid = threadIdx.x;
	unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
	sdata[tid] = g_idata[i];
	__syncthreads();

	//do reduction in shared memory

	for(unsigned int s=1; s<blockDim.x; s *=2) {
		if(tid % (2*s) == 0) {
			sdata[tid] += sdata[tid + s];
		}
		__syncthreads();
	}

	//write results for this block to global memory
	if (tid ==0)
		g_odata[blockIdx.x] = s_data[0];
}
```

Σχήμα 4.2: Αλγόριθμος Interleaved Addresing

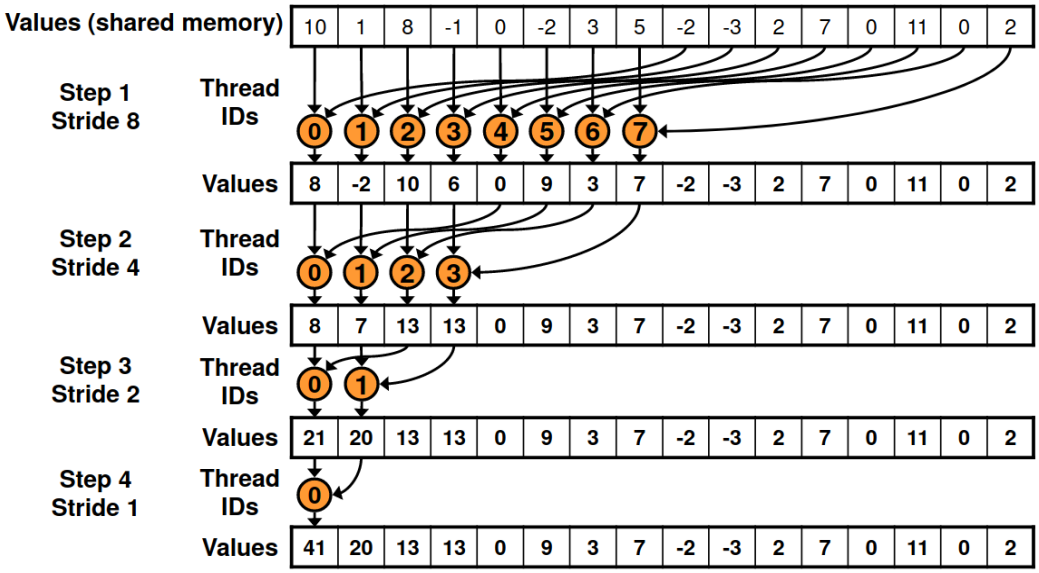


Σχήμα 4.3: Εκτέλεση Αλγόριθμου Interleaved Addressing.

Στον αλγόριθμο αυτό μπορούμε να παρατηρήσουμε ότι τα νήματα με αναγνωριστικό 0-31 δεν θα ακολουθήσουν το ίδιο μονοπάτι στην συνθήκη ελέγχου **if(tid % (2*s) == 0)** άρα εύκολα φαίνεται ότι προκαλείται branch divergence. Η υλοποίηση αυτή αποτελεί την βασική υλοποίηση και έχει την χαμηλότερη αποδοτικότητα ως προς τον χρόνο εκτέλεσης και την αξιοποίηση εύρους ζώνης της μνήμης.

Αλγόριθμος #2: Sequential Addressing with reversed loop and thread ID-based indexing

Στο σχήμα 4.4 (που πάρθηκε από την εργασία "Optimizing parallel reduction in CUDA." του Mark Harris [7]) παρουσιάζεται παράδειγμα εκτέλεσης του αλγορίθμου Sequential Addressing.



Σχήμα 4.4: Εκτέλεση Αλγόριθμου Sequential Addressing.

```
for (unsigned int s=1; s<blockDim.x; s *= 2) {
        if (tid % (2*s) == 0) {
                sdata[tid] += sdata[tid +s];
        }
        __syncthreads();
}
```

Σχήμα 4.5: Divergent Branch in Inner Loop

```
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
        if (tid < s) {
                sdata[tid] += sdata[tid +s];
        }
        __syncthreads();
}
```

Σχήμα 4.6: Reversed Loop with ThreadID-based Indexing

Αλλάζοντας τον τρόπο πρόσβασης της κοινόχρηστης μνήμης σε ακολουθιακές θέσεις εξαλείφεται το πρόβλημα του memory bank conflict κερδίζοντας έτσι καλύτερη αξιοποίηση της ταχύτητας της κοινόχρηστης μνήμης. Ακόμα η αντικατάσταση του βρόχου που φαίνεται στο Σχήμα 4.5, με καινούργιο όπου απαλείφεται ο τελεστής (%) (Σχήμα 4.6) και χρησιμοποιώντας indexing για τον πίνακα με βάση το αναγνωριστικό κάθε νήματος κερδίζουμε απόδοση όχι μόνο από τη χρήση πιο αποδοτικού τελεστή αλλά και από την εξάλειψη του φαινομένου του branch divergence.

Συγκριτικά με τον πρώτο αλγόριθμο, αυτός πετυχαίνει συνολικό speedup 4.68 φορές μεγαλύτερο και 4.6 φορές καλύτερη αξιοποίηση του εύρους ζώνης της κοινόχρηστης μνήμης. Παρά τις βελτιώσεις στη ταχύτητα που προσφέρει αυτός ο αλγόριθμος παρουσιάζεται το πρόβλημα των inactive threads καθώς μπορούμε να δούμε ότι κατά τη πρώτη επανάληψη του βρόγχου μόνο τα μισά νήματα είναι ενεργά.

Αλγόριθμος #3: Sequential Addresing with First Operation During Load

Προκειμένου να πετύχουμε τη πλήρη αξιοποίηση όλων των νημάτων του μπλοκ θα αλλάξουμε τον τρόπο με τον οποίο αποθηκεύονται τα δεδομένα στη κοινόχρηστη μνήμη.

Στο παρακάτω σχήμα (σχήμα 4.7) φαίνεται ότι στην αρχική υλοποίηση κάθε νήμα απλά αποθηκεύει στη κοινόχρηστη μνήμη το κάθε στοιχείο που λαμβάνει από τη καθολική μνήμη.

```
//each thread loads one element from global memory
  to shared memory
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
sdata[tid] = g_idata[i];
__syncthreads();
```

Σχήμα 4.7: Global Loading

```
 // perform first level of reduction
// reading from global memory, writing to shared memory
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
sdata[tid] = g_idata[i + blockDim.x];
__syncthreads();
```

Σχήμα 4.8: First Operation During Global Load

Στη καινούργια υλοποίηση (σχήμα 4.8) εκτός από τη μεταφορά του στοιχείου στη κοινόχρηστη μνήμη κάθε νήμα αναλαμβάνει και τη πρώτη επανάληψη της διαδικασίας υποβίβασης αξιοποιώντας έτσι όλα τα διαθέσιμα νήματα του μπλοκ. Ο αλγόριθμος αυτός πετυχαίνει speedup 8.34 φορές μεγαλύτερο συγκριτικά με τον αλγόριθμο 1 και αξιοποίηση του εύρους ζώνης μνήμης 8.3 φορές μεγαλύτερο.

```
Template <unsigned int blockSize>
__device__ void warpReduce(volatile int *sdata, int tid) {
        if (blocksize >= 64) sdata[tid] += sdata[tid + 32];
        if (blocksize >= 32) sdata[tid] += sdata[tid + 16];
        if (blocksize >= 16) sdata[tid] += sdata[tid + 8];
        if (blocksize >= 8) sdata[tid] += sdata[tid + 4];
        if (blocksize >= 4) sdata[tid] += sdata[tid + 2];
        if (blocksize >= 2) sdata[tid] += sdata[tid + 1];
}
```

```
    if (blockSize >= 512) {
      if (tid < 256 )
              sdata[tid] += sdata[tid + 256];
      __syncthreads();
    }
    if (blockSize >= 256) {
      if (tid < 128 )
              sdata[tid] += sdata[tid + 128];
      __syncthreads();
    }
    if (blockSize >= 128) {
      if (tid < 64 )
              sdata[tid] += sdata[tid + 64];
      __syncthreads();
    }
    if (tis <32)
      wrapReduce<blocksize>(sdata, tid);
```

Σχήμα 4.9: Completely Unrolled Warps

Αλγόριθμος #4: Completely Unrolled Warps

Παρά τη μεγάλη αποδοτικότητα του αλγόριθμου 3 παρατηρούμε ότι η υλοποίηση μπορεί να βελτιωθεί περαιτέρω. Γνωρίζοντας ότι η λειτουργία υποβίβασης έχει μικρή αριθμητική πολυπλοκότητα θα εστιάσουμε τις προσπάθειες βελτιστοποίησης του αλγόριθμου στην αποφυγή βρόγχων και περιττών αριθμητικών πράξεων. Έτσι εισάγουμε τη τεχνική του warp unrolling (σχήμα 4.9).

Με τη χρήση της τεχνικής αυτής γλυτώνουμε περιττούς βρόγχους, ελέγχους και εντολές συγχρονισμού πετυχαίνοντας έτσι μια βέλτιστη λύση στο πρόβλημα των παράλληλων υποβιβάσεων. Ο αλγόριθμος αυτός πετυχαίνει speedup 21.16 φορές μεγαλύτερο του αρχικού και έχει 21 φορές καλύτερη αξιοποίηση του εύρους ζώνης της κοινόχρηστης μνήμης.

Η σύγκριση των παραπάνω αλγορίθμων [7] γίνεται σε σύστημα που διαθέτει τη κάρτα γραφικής επεξεργασίας NVIDIA G8, όλα τα kernel εκτελούνται με 128 νήματα και ο πίνακας τα στοιχεία του οποίου υποβιβάζονται είναι τεσσάρων εκατομμυρίων στοιχείων.

	Time	Bandwidth	Cumulative Speedup
Αλγόριθμος #1: Interleaved Addressing	8.054 ms	2.083 GB/s	
Αλγόριθμος #2: Sequential Addressing with reversed loop and threadID-based indexing	1.722ms	9.741 GB/s	4.68x
Αλγόριθμος #3: Sequential Addresing with First Operation During Load	0.965ms	17.377 GB/s	8.34x
Αλγόριθμος #4: Completely Unrolled Warps	0.381ms	43.996 GB/s	21.16x

Πίνακας 4.1: Σύγκριση επιδόσεων αλγόριθμων για υποβίβαση 4 εκατομμυρίων στοιχείων

4.2 Υλοποίηση στον OMPi

Για την ενσωμάτωση της λειτουργίας υποβιβάσεων στον παραλληλοποιητικό μεταφραστή OMPi αρχικά προστέθηκε το αρχείο που την υλοποιεί στον κατάλογο της διεπαφής devpart του CUDA device. Έπειτα έγιναν οι κατάλληλες αλλαγές στα αρχεία MakeFile έτσι ώστε να μπορεί ο OMPi να αναγνωρίσει τη προσθήκη της καινούργιας λειτουργίας.

Εντός του αρχείου που υλοποιεί τη λειτουργία υποβίβασης αρχικά υλοποιήθηκαν οι εξής βοηθητικές συναρτήσεις:

- **int get_blockDim()** : Επιστρέφει τον αριθμό των νημάτων ενός μπλοκ
- **int get_LocalIdx()** : Επιστρέφει το αναγνωριστικό (ID) ενός νήματος μέσα στο μπλοκ
- **int get_blockId()** : Επιστρέφει το αναγνωριστικό (ID) του κάθε μπλοκ
- **int get_gridDim()** : Επιστρέφει το μέγεθος του πλέγματος, δηλαδή το πλήθος των μπλοκ που βρίσκονται μέσα του
- **int get_prevPow2(int blockDim)** : Επιστρέφει τον αμέσως μικρότερο αριθμό που είναι δύναμη του 2

Για λόγους συμβατότητας χρησιμοποιήθηκε το ίδιο στυλ διεπαφής, δηλαδή το ίδιο πρωτότυπο της συνάρτησης που καλεί τη λειτουργία υποβίβασης. Συγκεκριμένα με τη χρήση μακροεντολών (C macros) ορίζεται ένα πρότυπο (template) που προβλέπει όλες τις περιπτώσεις που μπορεί να κληθεί η συνάρτηση όπως διαφορετικοί τύποι δεδομένων και διαφορετικοί τελεστές υποβίβασης.

Ο κυρίως αλγόριθμος υποβίβασης χωρίζεται σε τρία μέρη και μέρος του παρουσιάζεται στο σχήμα 4.10

- Αρχικά κάθε νήμα λαμβάνει ένα στοιχείο από τη καθολική μνήμη. Αν το στοιχείο αυτό είναι πίνακας εφαρμόζει στα στοιχεία του τον τελεστή υποβίβασης και αποθηκεύει το αποτέλεσμα στη κοινόχρηστη μνήμη. Αν το στοιχείο που θα λάβει είναι απλή μεταβλητή απλά την αποθηκεύει στη κοινόχρηστη μνήμη.
- Αμέσως μετά υποβιβάζουμε σε επίπεδο μπλοκ. Για τη διαδικασία αυτή χρησιμοποιείται μια παραλλαγή του αλγόριθμου Sequential Addressing with reversed loop and threadID-based που παρουσιάστηκε παραπάνω. Αφού ολοκληρωθεί η διαδικασία αυτή το υποαποτέλεσμα αποθηκεύεται στη πρώτη θέση της κοινόχρηστης μνήμης.

- Τέλος για να λάβουμε το τελικό αποτέλεσμα της διαδικασίας θα πρέπει να υποβιβάσουμε σε επίπεδο πλέγματος. Αρχικά ορίζουμε ως αρχηγό του κάθε μπλοκ το νήμα με αναγνωριστικό 0. Κάθε τέτοιο νήμα αναλαμβάνει το υποαποτέλεσμα από τη κοινόχρηστη μνήμη του μπλοκ του και το συνδυάζει με τα τα υπόλοιπα υποαποτελέσματα των άλλων μπλοκ. Μετά το πέρας αυτής της διαδικασίας το τελικό αποτέλεσμα αποθηκεύεται πίσω στη καθολική μνήμη. Στη περίπτωση που έχουμε μόνο ένα μπλοκ διαθέσιμο το νήμα αρχηγός απλά μεταφέρει το υποαποτέλεσμα το μπλοκ του πίσω στη καθολική μνήμη.

```
1  for(j=0; j<nelems; j++)
2  {
3       int remainder = get_blockDim();
4       partialRes[tid] = local_d[j];
5       __syncthreads();
6       while(!(remainder==1 || remainder==0))
7       {\
8       prev_pow2=get_prevPow2(remainder);
9       for(i=prev_pow2/2; i>0; i>>=1)
10      {
11              if(tid<i)
12              {
13                      partialRes[tid] = partialRes[tid] + partialRes[tid+i];
14              }
15              __syncthreads();
16       }
17
18      if(tid == prev_pow2 && remainder-prev_pow2!=0)
19      {
20              partialRes[0] = partialRes[0] + partialRes[tid];
21      }
22      __syncthreads();
23      if(tid > prev_pow2 && remainder-prev_pow2!=0)
24      {
25              partialRes[tid-prev_pow2] = partialRes[tid];
26      }
27      __syncthreads();
28      remainder = remainder- prev_pow2;
29      }
30      __syncthreads();
31 }
```

Σχήμα 4.10: Μέρος του αλγορίθμου υποβίβασης

Ο αλγόριθμος Sequential Addressing with reversed loop and threadID-based Indexing δεν είναι ο πιο αποδοτικός αλγόριθμος υποβίβασης όπως φαίνεται και από τα στοιχεία που παρουσιάζονται στο πίνακα 4.1. Όμως είναι ο πιο αποδοτικός αλγόριθμος που μπορεί ταυτόχρονα να προσαρμοστεί και να επεκταθεί έτσι ώστε να λειτουργεί σε ένα περιβάλλον μεταφραστή όπου καλούμαστε να υποστηρίζουμε ένα μεγάλο πλήθος από διαφορετικές περιπτώσεις της λειτουργίας υποβίβασης. Στη περίπτωση που επιλέγαμε τον πιο αποδοτικό αλγόριθμο Completely Unrolled Warps αυτός δεν θα μπορούσε να υλοποιηθεί για οποιαδήποτε αριθμό νημάτων γιατί όπως φαίνεται και από το σχήμα 4.9 εξαλείφεται ο βρόχος που διατρέχει τα αναγνωστικά των νημάτων όπως γίνεται στον Sequential Addressing with reversed loop and threadID-based Indexing (σχήμα 4.6) και αντικαθίσταται με στατικές προσπελάσεις συγκεκριμένων θέσεων μνήμης.

4.3 Θέματα που Προέκυψαν κατά την Υλοποίηση

Το βασικό πρόβλημα που προέκυψε κατά την υλοποίηση είναι η μετατροπή του αλγόριθμου Sequential Addressing with reversed loop and threadID-based indexing έτσι ώστε να λειτουργεί στο περιβάλλον του μεταφραστή. Ο περιορισμός αυτού του αλγόριθμου είναι ότι ο αριθμός των νημάτων πρέπει να είναι δύναμη του 2 και θέλουμε ο χρήστης να μπορεί να χρησιμοποιήσει τη λειτουργία υποβίβασης για οποιοδήποτε αριθμό νημάτων. Για να ξεπεραστεί αυτός ο περιορισμός ακολουθήθηκαν τα εξής βήματα:

- Αρχικά υπολογίζεται ο αμέσως μικρότερος αριθμός από το πλήθος των νημάτων που είναι δύναμη του 2.
- Μετά εφαρμόζεται ο αλγόριθμος στα στοιχεία της κοινόχρηστης μνήμης με θέση μικρότερη του αριθμού που υπολογίστηκε.
- Τα στοιχεία που βρίσκονται σε μεγαλύτερη θέση μνήμης μετατοπίζονται σε αριστερότερες θέσεις μνήμης και υπολογίζεται το υπόλοιπο των στοιχείων που δεν έχει επισκεφτεί ο αλγόριθμος.
- Η διαδικασία αυτή συνεχίζεται μέχρι ο αλγόριθμος να έχει επισκεφτεί όλα τα στοιχεία της κοινόχρηστης μνήμης.

Ένα ακόμα πρόβλημα που προέκυψε είναι ο συγχρονισμός μεταξύ των μπλοκ του πλέγματος. Στο τρίτο μέρος της διαδικασίας υποβίβασης όταν χρειάζεται να συγκεντρωθούν τα υποαποτέλεσμα δημιουργείται μία κρίσιμη περιοχή αφού όλα τα νήματα τροποποιούν την ίδια μεταβλητή. Αυτό δεν θα αποτελούσε κανονικά πρόβλημα αλλά η CUDA δεν προσφέρει έμμεσα κάποιον μηχανισμό συγχρονισμού νημάτων που ανήκουν σε

```
__device__ int global_lock = 0;
int needlock = 1;

do {
        if (0 == atomicCAS(&global_lock, 0, 1));
                {
                        /* critical section */
                        needlock = 0;
                }
}
while (needlock == 1);
__threadfence();
atomicExch(&global_lock, 0);
```

Σχήμα 4.11: Υλοποίηση Κλειδαριών

διαφορετικά μπλοκ. Για να αντιμετωπιστεί αυτό το πρόβλημα υλοποιήθηκε ένας πολύ βασικός μηχανισμός κλειδαριών. Ο μηχανισμός αυτός βασίζεται στη χρήση των συναρτήσεων **atomicCAS** και **atomicExch** που προσφέρει το CUDA API.

Οι συναρτήσεις αυτές ουσιαστικά επιτρέπουν μόνο σε ένα νήμα να τροποποιήσει μια καθολική μεταβλητή. Η υλοποίηση των κλειδαριών με τη χρήση αυτών των συναρτήσεων φαίνεται στο σχήμα 4.11.

Τέλος κατά τη διαδικασία αξιολόγησης των επιδόσεων του μηχανισμού υποβίβασης παρατηρήθηκαν χαμηλές επιδόσεις στη περίπτωση εκτέλεσης προγραμμάτων με μεγάλο αριθμό μπλοκ. Αυτό οφείλεται στο γεγονός ότι οι συναρτήσεις **atomicCAS** και **atomicExch** εισάγουν καθυστερήσεις, καθώς έχουν μεγάλη πολυπλοκότητα. Ένας τρόπος μείωσης αυτών των καθυστερήσεων είναι ο συγχρονισμός των μπλοκ με τη χρήση του μηχανισμού των **Cooperative Groups**[8]. Το προγραμματιστικό μοντέλο των Cooperative Groups περιγράφει μοτίβα συγχρονισμού τόσο εντός όσο και μεταξύ των μπλοκ νημάτων CUDA. Παρέχει μία διεπαφή στη συσκευή CUDA για τον ορισμό, τη κατάτμηση και το συγχρονισμό ομάδων νημάτων. Παρέχει επίσης διεπαφή από τη πλευρά του host για την εκκίνηση πλεγμάτων των οποίων όλα τα νήματα είναι εγγυημένα ότι εκτελούνται ταυτόχρονα για να επιτευχθεί συγχρονισμός μεταξύ των μπλοκ νημάτων. Αυτές οι λειτουργίες επιτρέπουν νέους τρόπους συνεργατικού παραλληλισμού εντός του CUDA, συμπεριλαμβανομένου του παραλληλισμού παραγωγού-καταναλωτή και του καθολικού συγχρονισμού σε ολόκληρο το πλέγμα νημάτων ή ακόμα και σε πολλαπλές GPU.

Έτσι με βάση τις λειτουργίες που προσφέρει το προγραμματιστικό μοντέλο των Cooperative Groups ο μηχανισμός συγχρονισμού των μπλοκ τροποποιήθηκε ως εξής:

- Αρχικά ορίζεται ένα Cooperative Group που περιλαμβάνει όλα τα νήματα του πλέγματος
- Μετά την εύρεση των υποαποτελεσμάτων από το κάθε μπλοκ, αυτά γράφονται στη καθολική μνήμη της συσκευής και με τη χρήση της συνάρτησης **cg::sync(grid)**, η οποία υλοποιεί ένα barrier για όλα τα νήματα του πλέγματος, βεβαιώνεται ότι έχουμε το σωστό υποαποτέλεσμα από κάθε μπλοκ.
- Τέλος το μπλοκ με αναγνωριστικό 0 αντιγράφει στη κοινόχρηστη μνήμη του τα υποαποτελέσματα και με τη χρήση του τροποποιημένου αλγορίθμου Sequential Addressing with reversed loop and threadID-based indexing που περιγράφηκε παραπάνω, υπολογίζει το τελικό αποτέλεσμα.

Κεφάλαιο 5.
Αξιολόγηση

5.1 Εισαγωγή

Για τον έλεγχο της υποδομής λειτουργιών υποβίβασης πραγματοποιήθηκαν τόσο πειράματα ορθότητας όσο και πειράματα επιδόσεων. Στα πρώτα ελέγχεται η ορθή λειτουργία της υποδομής για διάφορους τελεστές υποβίβασης και τύπους δεδομένων. Στα δεύτερα, εξετάζονται οι επιδόσεις που επιτυγχάνονται συγκριτικά με άλλους μεταφραστές που υποστηρίζουν τη συγκεκριμένη λειτουργία. Η αξιολόγηση της υποδομής πραγματοποιήθηκε μέσω της εκτέλεσης πειραμάτων σε δύο μονάδες γραφικής επεξεργασίας, NVIDIA Tesla P40 και την NVIDIA GeForce GT730. Οι δύο αυτές μονάδες γραφικής επεξεργασίας επιλέχθηκαν διότι αποτελούν δύο αντιδιαμετρικές περιπτώσεις καθώς η πρώτη αποτελεί GPU υψηλών επιδόσεων και η δεύτερη αποτελεί GPU που μπορεί να βρεθεί στον μέσο προσωπικό υπολογιστή, Οι προδιαγραφές των συσκευών παρουσιάζονται στον πίνακα 5.1. Τα κύρια συστήματα στα οποία βρίσκονται εγκατεστημένες οι προαναφερθείσες μονάδες GPU είναι τα συστήματα Parallax και Paragon, αντίστοιχα. Το σύστημα Parallax διαθέτει δύο επεξεργαστές Intel Xeon Gold, με συνολική χωρητικότητα μνήμης 64 GiB. Το σύστημα Paragon διαθέτει δύο κύριους επεξεργαστές της σειράς Opteron, με χωρητικότητα μνήμης 16 GiB. Τα δύο συστήματα έχουν εγκατεστημένο το λειτουργικό σύστημα CentOS 8.

	NVIDIA Tesla P40	NVIDIA GeForce GT730
# Πολυπεξεργαστών	30	2
# Πυρήνων/Πολυεπεξεργαστή	128	192
Συνολικός # Πυρήνων	3840	384
Μέγιστος # Threads/Block	1024	1024
Μνήμη	24GB	2GB
CUDA compute capability	6.1	3.5

Πίνακας 5.1: Τεχνικά χαρακτηριστικά μονάδας γραφικής επεξεργασίας NVIDIA Tesla P40 και NVIDIA GeForce GT730

5.2 Αξιολόγηση Ορθότητας

Για την επαλήθευση της ορθότητας της υλοποίησης έγιναν εξαντλητικές δοκιμές με δικά μας δοκιμαστικά προγράμματα τα οποία ελέγχουν όλους τους τύπους λειτουργιών υποβίβασης και όλους τους δυνατούς τύπους δεδομένων. Όλες ήταν επιτυχημένες. Επιπλέον, επιλέχθηκαν προγράμματα ελέγχου από τη συλλογή SOLLVE (Scaling OpenMP Via LLVM for Exascale Performance and Portability)[9]. Συγκεκριμένα:

- **test_target_teams_distribute_reduction_add.c:** Αυτό το δοκιμαστικό πρόγραμμα ελέγχει αν το σειριακό άθροισμα όλων των στοιχείων δύο πινάκων ισούται με το αντίστοιχο που υπολογίζεται στη συσκευή γραφικής επεξεργασίας με χρήση του του τελεστή υποβίβασης **(+)**.
- **test_target_teams_distribute_reduction_and.c:** Αυτό το δοκιμαστικό πρόγραμμα ελέγχει αν το σειριακό δυαδικό ΚΑΙ όλων των στοιχείων ενός πίνακα ισούται με το αντίστοιχο που υπολογίζεται στη συσκευή γραφικής επεξεργασίας με χρήση του του τελεστή υποβίβασης **(&&)**.
- **test_target_teams_distribute_reduction_bitand.c:** Αυτό το δοκιμαστικό πρόγραμμα ελέγχει ελέγχει αν το σειριακό λογικό ΚΑΙ όλων των στοιχείων ενός πίνακα ισούται με το αντίστοιχο που υπολογίζεται στη συσκευή γραφικής επεξεργασίας με χρήση του του τελεστή υποβίβασης **(&)**.
- **test_target_teams_distribute_reduction_bitor.c:** Αυτό το δοκιμαστικό πρόγραμμα ελέγχει αν το σειριακό δυαδικό Η όλων των στοιχείων ενός πίνακα ισούται με το αντίστοιχο που υπολογίζεται στη συσκευή γραφικής επεξεργασίας με χρήση του του τελεστή υποβίβασης **(|)**.
- **test_target_teams_distribute_reduction_bitxor.c:** Αυτό το δοκιμαστικό πρόγραμμα ελέγχει αν το σειριακό δυαδικό αποκλειστικό Η όλων

των στοιχείων ενός πίνακα ισούται με το αντίστοιχο που υπολογίζεται στη συσκευή γραφικής επεξεργασίας με χρήση του του τελεστή υποβίβασης **(^)**.

- **test_target_teams_distribute_reduction_multiply.c:** Το δοκιμαστικό αυτό πρόγραμμα ελέγχει αν το σειριακό γινόμενο όλων των στοιχείων ενός πίνακα ισούται με το αντίστοιχο που υπολογίζεται στη συσκευή γραφικής επεξεργασίας με χρήση του του τελεστή υποβίβασης **(*)**.
- **test_target_teams_distribute_reduction_or.c:** Αυτό το δοκιμαστικό πρόγραμμα ελέγχει αν το σειριακό λογικό Η όλων των στοιχείων ενός πίνακα ισούται με το αντίστοιχο που υπολογίζεται στη συσκευή γραφικής επεξεργασίας με χρήση του του τελεστή υποβίβασης **(||)**.
- **test_target_teams_distribute_reduction_subtract.c:** Το δοκιμαστικό αυτό πρόγραμμα ελέγχει αν η σειριακή διαφορά όλων των στοιχείων δύο πινάκων ισούται με το αντίστοιχο που υπολογίζεται στη συσκευή γραφικής επεξεργασίας με χρήση του του τελεστή υποβίβασης **(-)**.
- **test_target_teams_distribute_min.c:** Αυτό το δοκιμαστικό πρόγραμμα ελέγχει αν το σειριακό ελάχιστο από τα αθροίσματα των στοιχείων δύο πινάκων ισούται με το αντίστοιχο που υπολογίζεται στη συσκευή γραφικής επεξεργασίας με χρήση του του τελεστή υποβίβασης **(min)**.

Τα αποτελέσματα των πειραμάτων παρουσιάζονται στον πίνακα 5.2

	Compilation	Execution
test_target_teams_distribute_reduction_add.c	Passed	Passed
test_target_teams_distribute_reduction_and.c	Passed	Passed
test_target_teams_distribute_reduction_bitand.c	Passed	Passed
test_target_teams_distribute_reduction_bitor.c	Passed	Passed
test_target_teams_distribute_reduction_bitxor.c	Passed	Passed
test_target_teams_distribute_reduction_multiply.c	Passed	Passed
test_target_teams_distribute_reduction_or.c	Passed	Passed
test_target_teams_distribute_reduction_subtract.c	Passed	Passed
test_target_teams_distribute_min.c	Passed	Passed

Πίνακας 5.2: Αποτελέσματα εκτέλεσης δοκιμαστικών προγραμμάτων

5.3 Αξιολόγηση Επιδόσεων

Η αξιολόγηση των επιδόσεων της λειτουργίας υποβίβασης αποτελείται από τρία μέρη. Το πρώτο μέρος αφορά την εύρεση του αθροίσματος των στοιχείων πίνακα ακέραιων αριθμών (integers), το δεύτερο αφορά την υποβίβαση πίνακα και το τρίτο αφορά εφαρμογή η οποία εφαρμόζει intensity normalization σε φωτογραφία. Κάθε μια από τις παραπάνω περιπτώσεις έχει εκτελεσθεί για διάφορους αριθμούς νημάτων και μπλοκ.
Στη χρονομέτρηση των δύο πρώτων περιπτώσεων δεν προσμετράται ο χρόνος μεταφοράς από και προς τη μνήμη της συσκευής γραφικής επεξεργασίας ενώ στη τρίτη περίπτωση μετράμε όλον τον χρόνο εκτέλεσης του kernel που εκτελείται στην μονάδα γραφικής επεξεργασίας.

Στις μετρήσεις που εκτελούνται στο σύστημα Parallax συγκρίνονται, η υλοποίηση με κλειδαριές (πεδίο OMPi), η υλοποίηση με Cooperative Groups (πεδίο OMPi CG) καθώς και η έκδοση 13 του μεταφραστή LLVM (πεδίο LLVM13). Στο σύστημα Paragon συγκρίνεται η υλοποίηση με κλειδαριές (πεδίο OMPi) με την έκδοση 15 του μεταφραστή LLVM (πεδίο LLVM15). Στο σύστημα αυτό δεν παρουσιάζεται η υλοποίηση με Cooperative Groups καθώς δεν υποστηρίζονται από τη μονάδα γραφικής επεξεργασίας NVIDIA GeForce GT730.

5.3.1 Άθροιση στοιχείων πίνακα μεγέθους δύο εκατομμυρίων

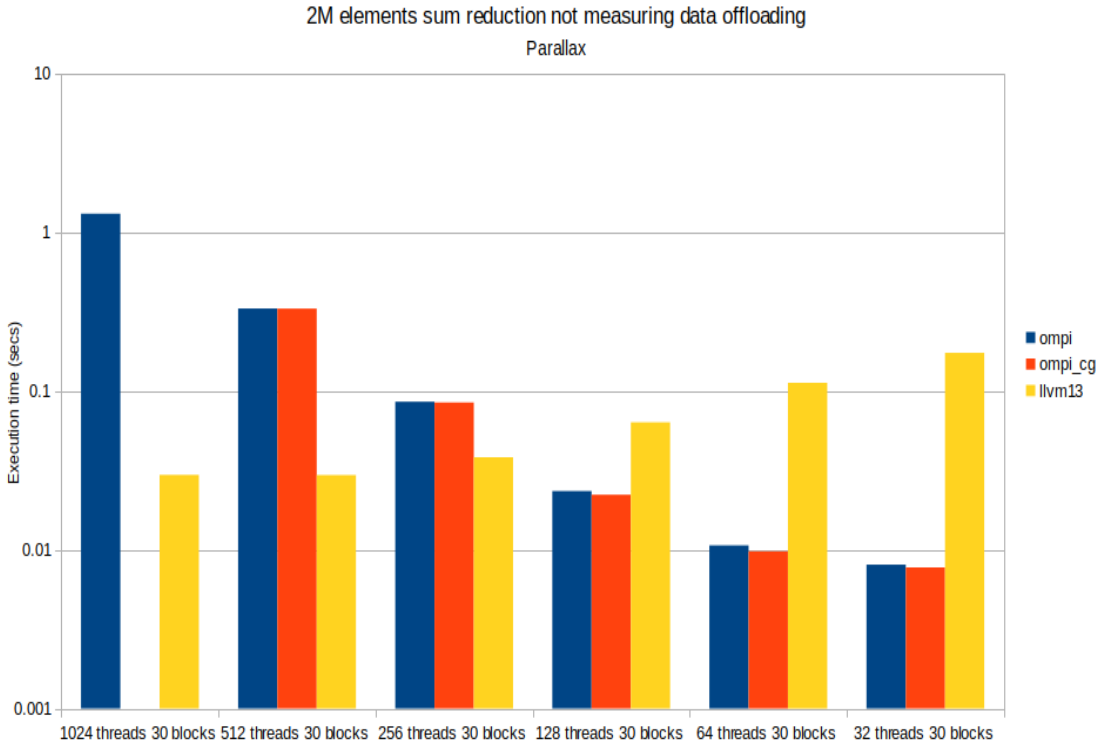
Το δοκιμαστικό πρόγραμμα για την άθροιση των στοιχείων ενός πίνακα αποτελεί μια απλή εφαρμογή στην οποία ορίζεται ένας πίνακας ακεραίων μεγέθους 2 εκατομμυρίων στοιχείων. Ο πίνακας αυτός αντιγράφεται στη καθολική μνήμη της γραφικής μονάδας επεξεργασίας η οποία υπολογίζει το άθροισμα με χρήση του τελεστή υποβίβασης **(+)**.
Η εφαρμογή είναι γραμμένη χρησιμοποιώντας τις εντολές του προτύπου OpenMP και τμήμα της δίνεται στο σχήμα 5.1

```
int parallel_sum = 0;

#pragma omp target map(to: a[0:N])
        printf("> ");
start = omp_get_wtime();
#pragma omp target teams distribute parallel for \
   num_threads(512) num_teams(10) reduction(+: parallel_sum)
        for (i=0; i<N; i++)
                parallel_sum += a[i];
finish = omp_get_wtime();
```

Σχήμα 5.1: Τμήμα εφαρμογής άθροισης στοιχειών πίνακα

Οι εκτελέσεις του πειράματος δεν παρουσίασαν μεγάλη διακύμανση στους χρόνους και επομένως οι χρόνοι που παρουσιάζονται στα παρακάτω σχήματα προέκυψαν από τον μέσο όρο 5 εκτελέσεων της κάθε περίπτωσης.

Στο Σχήμα 5.2 παρουσιάζεται το γράφημα των αποτελεσμάτων μέτρησης άθροισης στοιχείων πίνακα με 30 blocks και για διαφορετικούς αριθμούς νημάτων στο σύστημα Parallax. Ο άξονας x αναπαριστά τις διαφορετικές περιπτώσεις εκτέλεσης ενώ ο άξονας y αναπαριστά τον χρόνο εκτέλεσης σε δευτερόλεπτα. Ο άξονας y είναι σε λογαριθμική κλίμακα για καλύτερη αναπαράσταση των δεδομένων.

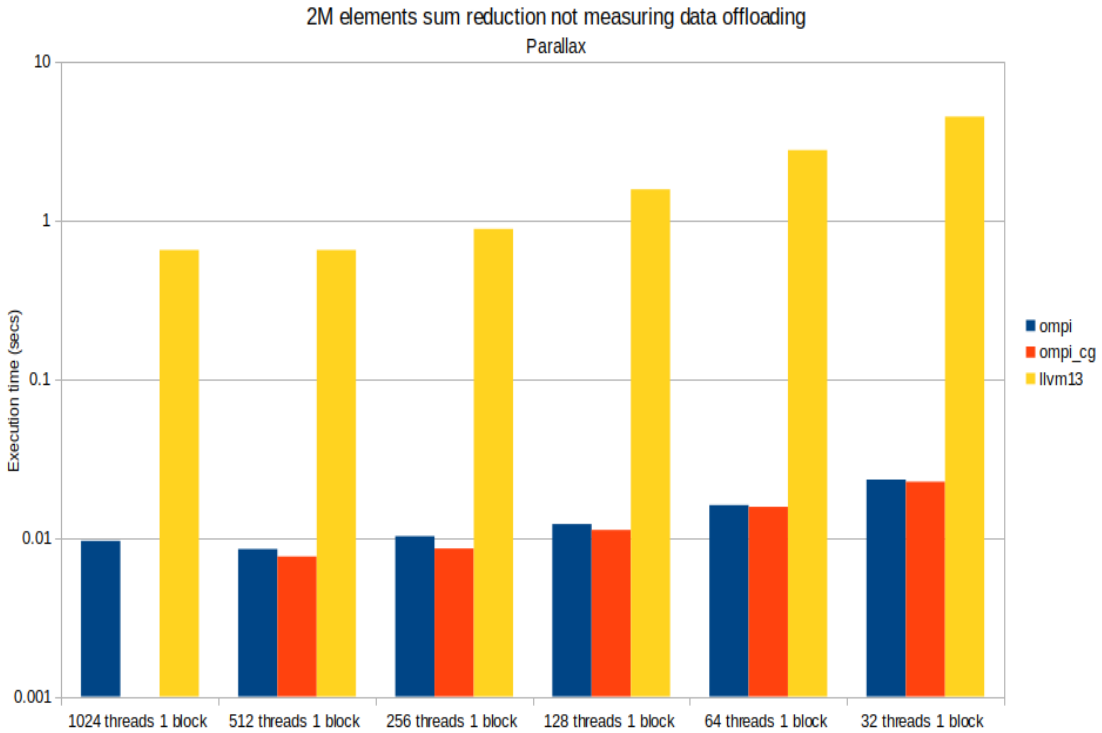


Σχήμα 5.2: Γράφημα αποτελεσμάτων μέτρησης άθροισης στοιχείων πίνακα με 30 blocks στο σύστημα Parallax

Από το παραπάνω γράφημα εξάγονται τα εξής συμπεράσματα:

- Και οι δύο εκδόσεις του OMPi έχουν παρόμοιους χρόνους εκτέλεσης σε όλες τις περιπτώσεις
- Ο OMPi στην έκδοση με τα cooperative groups έχει τις καλύτερες επιδόσεις για αριθμό νημάτων μικρότερο ίσο του 128
- Ο LLVM έχει καλύτερες επιδόσεις για αριθμό νημάτων μεγαλύτερο ίσο του 256
- Ο OMPi στην έκδοση με τα cooperative groups δεν παράγει αποτελέσματα για αριθμό νημάτων ίσο με 1024. Αυτό συμβαίνει διότι στη περίπτωση αυτή παραβιάζονται οι συνθήκες του occupancy. Το occupancy είναι ο λόγος των ενεργών warps προς τον συνολικό αριθμό warps που υποστηρίζονται σε έναν πολυεπεξεργαστή της GPU. Κάθε πολυεπεξεργαστής στη συσκευή έχει ένα σύνολο καταχωρητών που είναι διαθέσιμοι για χρήση από νήματα προγράμματος CUDA. Αυτοί οι καταχωρητές είναι ένας κοινόχρηστος πόρος που κατανέμεται μεταξύ των μπλοκ νημάτων που εκτελούνται σε έναν πολυεπεξεργαστή. Όταν ένα πρόγραμμα CUDA χρησιμοποιεί περισσότερους καταχωρητές από αυτούς που παρέχονται, η εκτέλεσή του αποτυγχάνει.

Στο Σχήμα 5.3 παρουσιάζεται το γράφημα των αποτελεσμάτων μέτρησης άθροισης στοιχείων πίνακα με 1 block και για διαφορετικούς αριθμούς νημάτων στο σύστημα Parallax. Ο άξονας x αναπαριστά τις διαφορετικές περιπτώσεις εκτέλεσης ενώ ο άξονας y αναπαριστά τον χρόνο εκτέλεσης σε δευτερόλεπτα. Ο άξονας y είναι σε λογαριθμική κλίμακα για καλύτερη αναπαράσταση των δεδομένων.

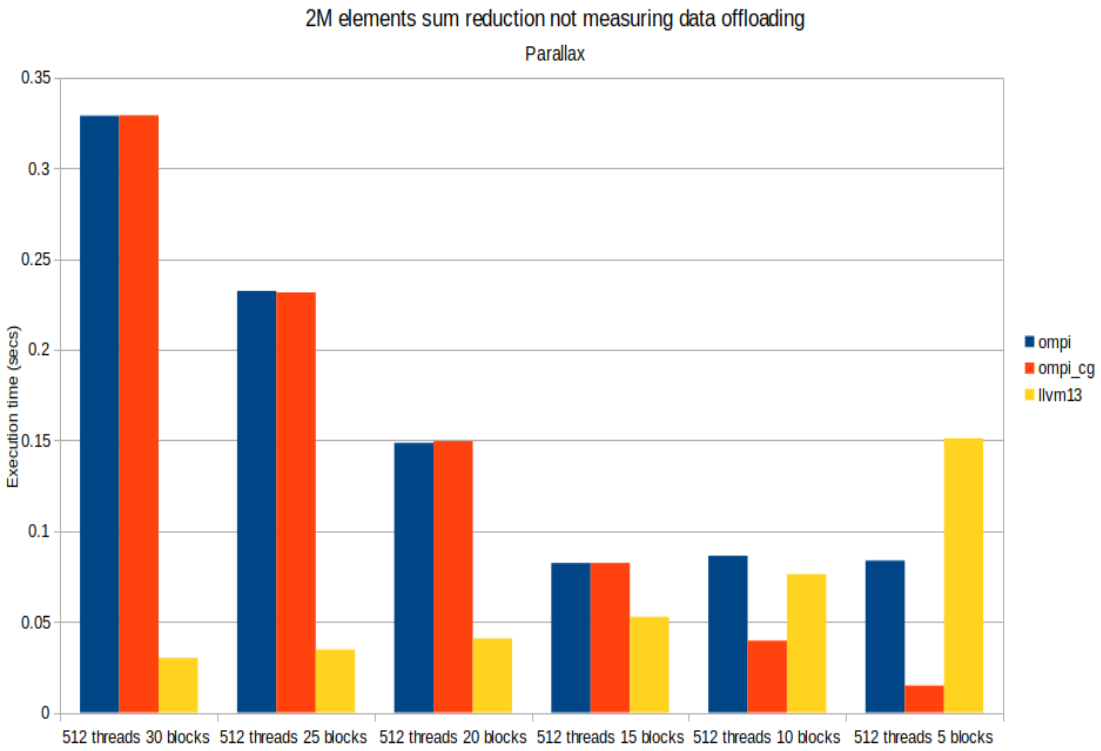


Σχήμα 5.3: Γράφημα αποτελεσμάτων μέτρησης άθροισης στοιχείων πίνακα με 1 block στο σύστημα Parallax

Από το Σχήμα 5.3 εξάγονται τα εξής συμπεράσματα:

- Και οι δύο εκδόσεις του OMPi έχουν παρόμοιους χρόνους εκτέλεσης σε όλες τις περιπτώσεις, με ελαφρά καλύτερες αυτές του ompi_cg και οι οποίοι είναι σημαντικά καμηλότεροι από τον χρόνο που απαιτεί ο LLVM.
- Ο OMPi στην έκδοση με τα cooperative groups έχει τις καλύτερες επιδόσεις για όλες τις περιπτώσεις
- Ο OMPi στην έκδοση με τα cooperative groups δεν παράγει αποτελέσματα για αριθμό νημάτων ίσο με 1024 για λόγους που έχουν αναφερθεί παραπάνω.

Στο Σχήμα 5.4 παρουσιάζεται το γράφημα των αποτελεσμάτων μέτρησης άθροισης στοιχείων πίνακα με 512 νήματα και για διαφορετικούς αριθμούς block στο σύστημα Parallax. Ο άξονας x αναπαριστά τις διαφορετικές περιπτώσεις εκτέλεσης ενώ ο άξονας y αναπαριστά τον χρόνο εκτέλεσης σε δευτερόλεπτα.

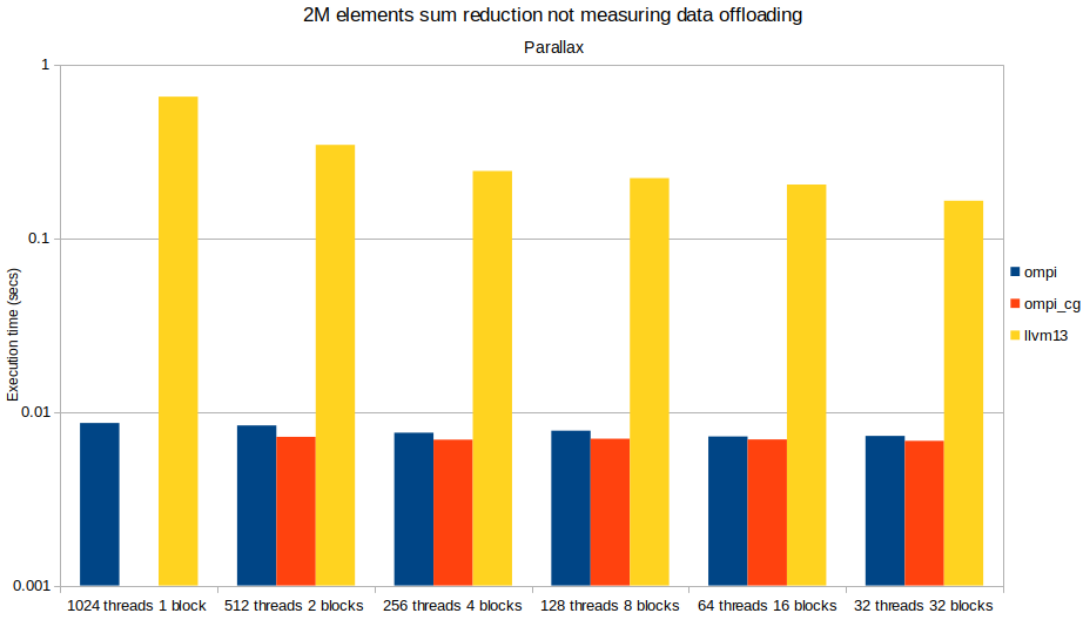


Σχήμα 5.4: Γράφημα αποτελεσμάτων μέτρησης άθροισης στοιχείων πίνακα με 512 νήματα στο σύστημα Parallax

Από το Σχήμα 5.4 εξάγονται τα εξής συμπεράσματα:

- Και οι δύο εκδόσεις του OMPi έχουν παρόμοιους χρόνους εκτέλεσης σε όλες τις περιπτώσεις
- Ο OMPi στην έκδοση με τα cooperative groups έχει τις καλύτερες επιδόσεις για αριθμό block μικρότερο ίσο του 10
- Ο LLVM έχει καλύτερες επιδόσεις για αριθμό block μεγαλύτερο ίσο του 15

Στο Σχήμα 5.5 παρουσιάζεται το γράφημα των αποτελεσμάτων μέτρησης άθροισης στοιχείων πίνακα με σταθερό αριθμό συνολικών νημάτων στο σύστημα Parallax. Ο άξονας x αναπαριστά τις διαφορετικές περιπτώσεις εκτέλεσης ενώ ο άξονας y αναπαριστά τον χρόνο εκτέλεσης σε δευτερόλεπτα. Ο άξονας y είναι σε λογαριθμική κλίμακα για καλύτερη αναπαράσταση των δεδομένων.

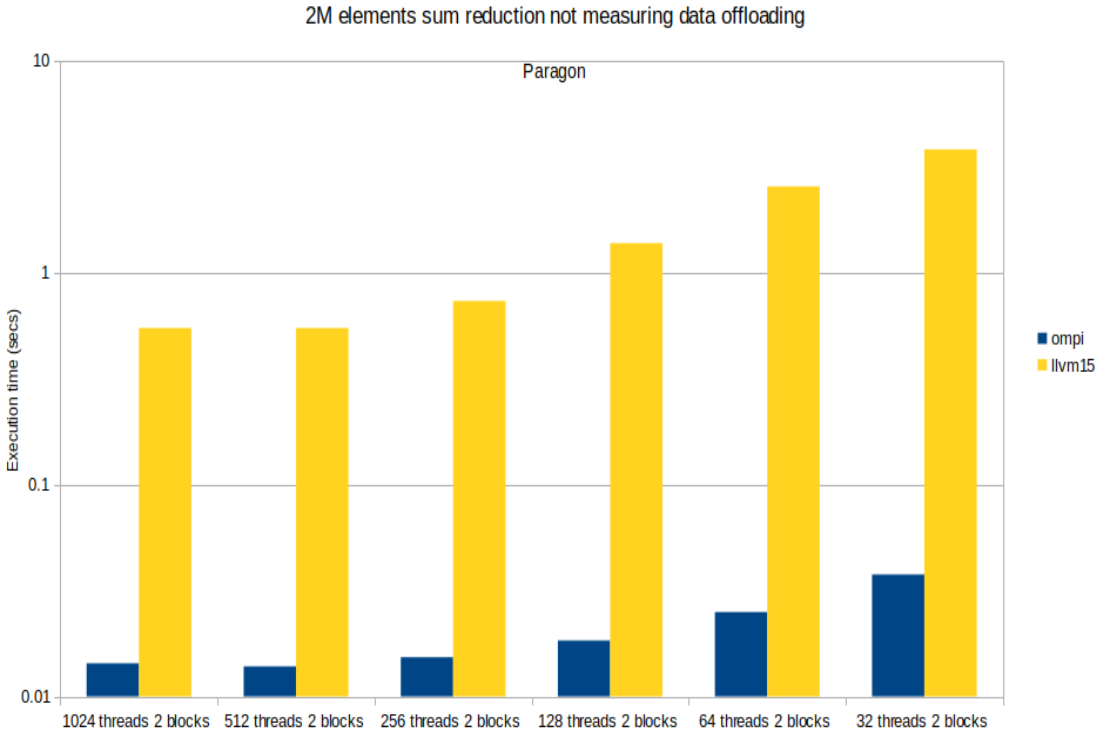


Σχήμα 5.5: Γράφημα αποτελεσμάτων μέτρησης άθροισης στοιχείων πίνακα με σταθερό αριθμό νημάτων στο σύστημα Parallax

Από το Σχήμα 5.5 εξάγονται τα εξής συμπεράσματα:

- Και οι δύο εκδόσεις του OMPi έχουν παρόμοιους χρόνους εκτέλεσης σε όλες τις περιπτώσεις
- Ο OMPi στην έκδοση με τα cooperative groups δεν παράγει αποτελέσματα για αριθμό νημάτων ίσο με 1024 για λόγους που έχουν αναφερθεί παραπάνω.
- Ο OMPi στην έκδιση με τα cooperative groups πετυχαίνει εντυπωσιακά καλύτερους χρόνους εκτέλεσης από τον LLVM σε όλες τις περιπτώσεις

Στο Σχήμα 5.6 παρουσιάζεται το γράφημα των αποτελεσμάτων μέτρησης άθροισης στοιχείων πίνακα με 2 blocks και για διαφορετικούς αριθμούς νημάτων στο σύστημα Paragon. Ο αριθμός των μπλοκ επιλέχθηκε γιατί η μονάδα γραφικής επεξεργασίας GeForce GT730 διαθέτει 2 πολυεπεξεργαστές. Ο άξονας x αναπαριστά τις διαφορετικές περιπτώσεις εκτέλεσης ενώ ο άξονας y αναπαριστά τον χρόνο εκτέλεσης σε δευτερόλεπτα. Ο άξονας y είναι σε λογαριθμική κλίμακα για καλύτερη αναπαράσταση των δεδομένων.

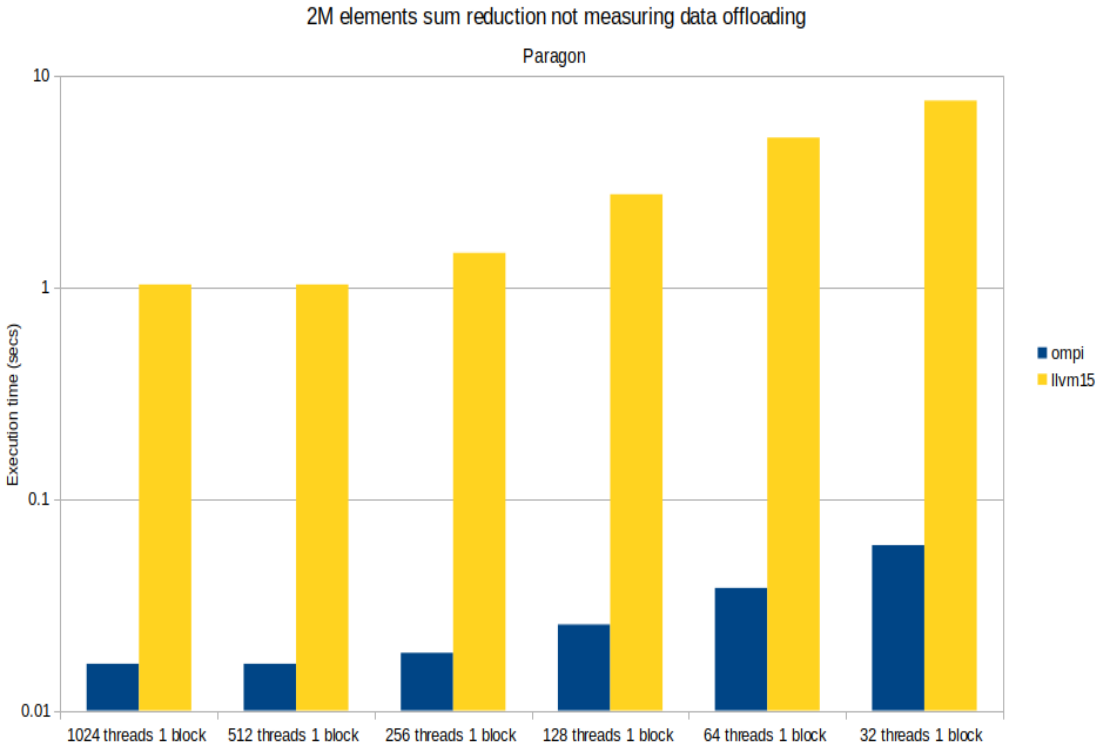


Σχήμα 5.6: Γράφημα αποτελεσμάτων μέτρησης άθροισης στοιχείων πίνακα με 2 blocks στο σύστημα Paragon

Από το Σχήμα 5.6 εξάγονται τα εξής συμπεράσματα:

- Ο OMPi πετυχαίνει εντυπωσιακά καλύτερους χρόνους εκτέλεσης από τον LLVM σε όλες τις περιπτώσεις
- Ο αριθμός των μπλοκ που υποστηρίζεται από τη μονάδα γραφικής επεξεργασίας δεν είναι αρκετά μεγάλος έτσι ώστε να παρατηρήσουμε πλεονέκτημα του LLVM όπως παρατηρήθηκαν στο σύστημα Parallax.

Στο Σχήμα 5.7 παρουσιάζεται το γράφημα των αποτελεσμάτων μέτρησης άθροισης στοιχείων πίνακα με 1 block και για διαφορετικούς αριθμούς νημάτων στο σύστημα Paragon . Ο άξονας x αναπαριστά τις διαφορετικές περιπτώσεις εκτέλεσης ενώ ο άξονας y αναπαριστά τον χρόνο εκτέλεσης σε δευτερόλεπτα. Ο άξονας y είναι σε λογαριθμική κλίμακα για καλύτερη αναπαράσταση των δεδομένων.



Σχήμα 5.7: Γράφημα αποτελεσμάτων μέτρησης άθροισης στοιχείων πίνακα με 1 block στο σύστημα Paragon

Από το Σχήμα 5.6 εξάγεται το εξής συμπέρασμα:

- Ο OMPi πετυχαίνει καλύτερους χρόνους εκτέλεσης από τον LLVM σε όλες τις περιπτώσεις

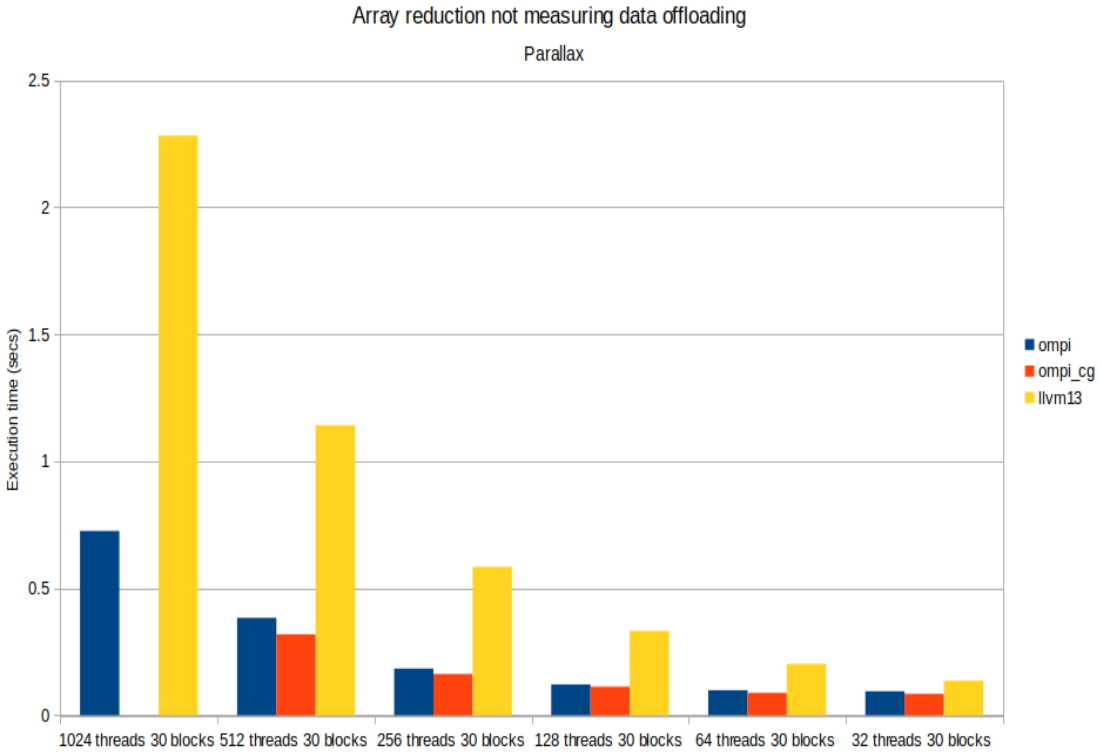
5.3.2 Υποβίβαση πίνακα

Το δοκιμαστικό πρόγραμμα για την υποβίβαση πίνακα αποτελεί μία απλή εφαρμογή στην οποία ορίζεται ένας πινάκας μεγέθους N, όπου N ισούται με αριθμός νημάτων επί πλήθος μπλοκ. Ο πίνακας αυτός υποβιβάζεται με την εντολή **reduction(+: array[0:N])**. Επειδή στη περίπτωση αυτή κάνουμε απλά N υποβιβάσεις χωρίς να σπαταλήσουμε χρόνο σε προηγούμενες πράξεις θα μετρήσουμε σχεδόν καθαρό overhead του μηχανισμού υποβίβαση.
Η εφαρμογή είναι γραμμένη χρησιμοποιώντας τις εντολές του προτύπου OpenMP και τμήμα της δίνεται στο Σχήμα 5.8

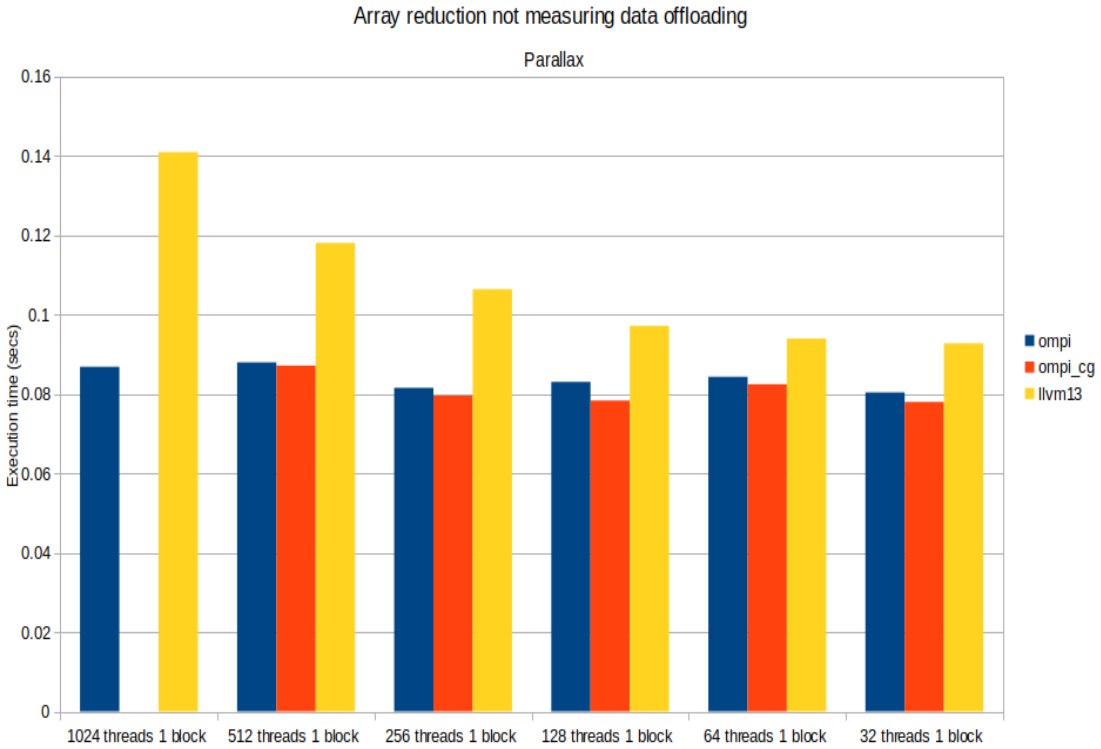
```
start = omp_get_wtime();
      #pragma omp target teams distribute parallel for \
         num_threads(32) num_teams(1) reduction(+:A[0:N])
            for (int i = 0; i < omp_get_num_threads(); i++)
             A[i] = 1;
finish = omp_get_wtime();

printf("Time elapsed = %f secs\n", (finish-start));
```

Σχήμα 5.8: Τμήμα εφαρμογής υποβίβασης πίνακα

Στο Σχήμα 5.9 παρουσιάζεται το γράφημα των αποτελεσμάτων μέτρησης υποβίβασης πίνακα με 30 blocks και για διαφορετικούς αριθμούς νημάτων στο σύστημα Parallax. Ο άξονας x αναπαριστά τις διαφορετικές περιπτώσεις εκτέλεσης ενώ ο άξονας y αναπαριστά τον χρόνο εκτέλεσης σε δευτερόλεπτα. Ο άξονας y είναι σε γραμμική κλίμακα.



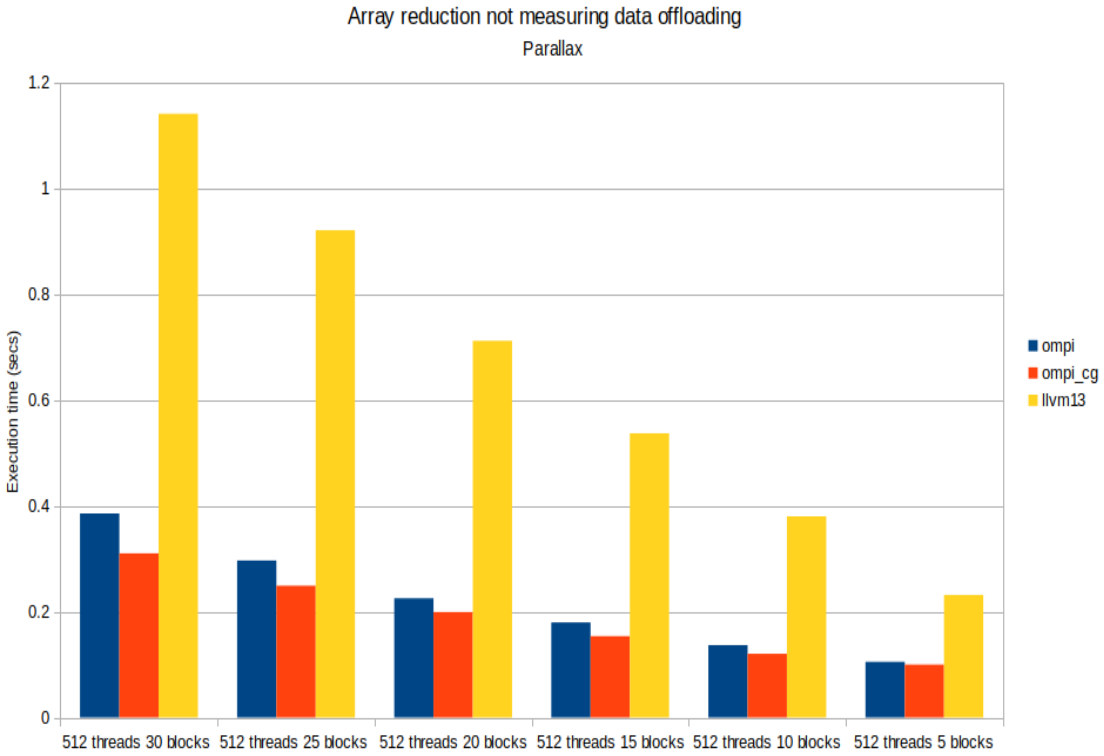
Σχήμα 5.9: Γράφημα αποτελεσμάτων μέτρησης υποβίβασης πίνακα με 30 blocks στο σύστημα Parallax



Σχήμα 5.10: Γράφημα αποτελεσμάτων μέτρησης υποβίβασης πίνακα με 1 block στο σύστημα Parallax

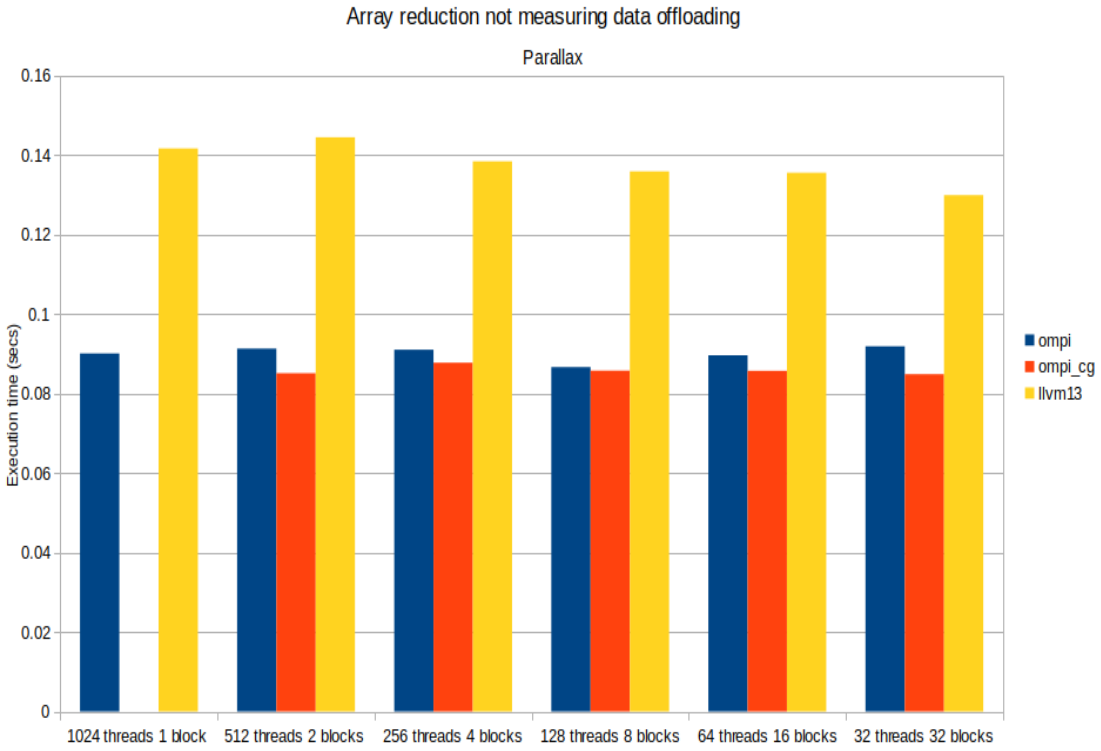
Στο Σχήμα 5.10 παρουσιάζεται το γράφημα των αποτελεσμάτων μέτρησης υποβίβασης πίνακα με 1 block και για διαφορετικούς αριθμούς νημάτων στο σύστημα Parallax. Ο άξονας x αναπαριστά τις διαφορετικές περιπτώσεις εκτέλεσης ενώ ο άξονας y αναπαριστά τον χρόνο εκτέλεσης σε δευτερόλεπτα. Ο άξονας y είναι σε γραμμική κλίμακα.

Στο Σχήμα 5.11 παρουσιάζεται το γράφημα των αποτελεσμάτων μέτρησης υποβίβασης πίνακα με 512 νήματα και για διαφορετικούς αριθμούς block. Ο άξονας x αναπαριστά τις διαφορετικές περιπτώσεις εκτέλεσης ενώ ο άξονας y αναπαριστά τον χρόνο εκτέλεσης σε δευτερόλεπτα. Ο άξονας y είναι σε γραμμική κλίμακα



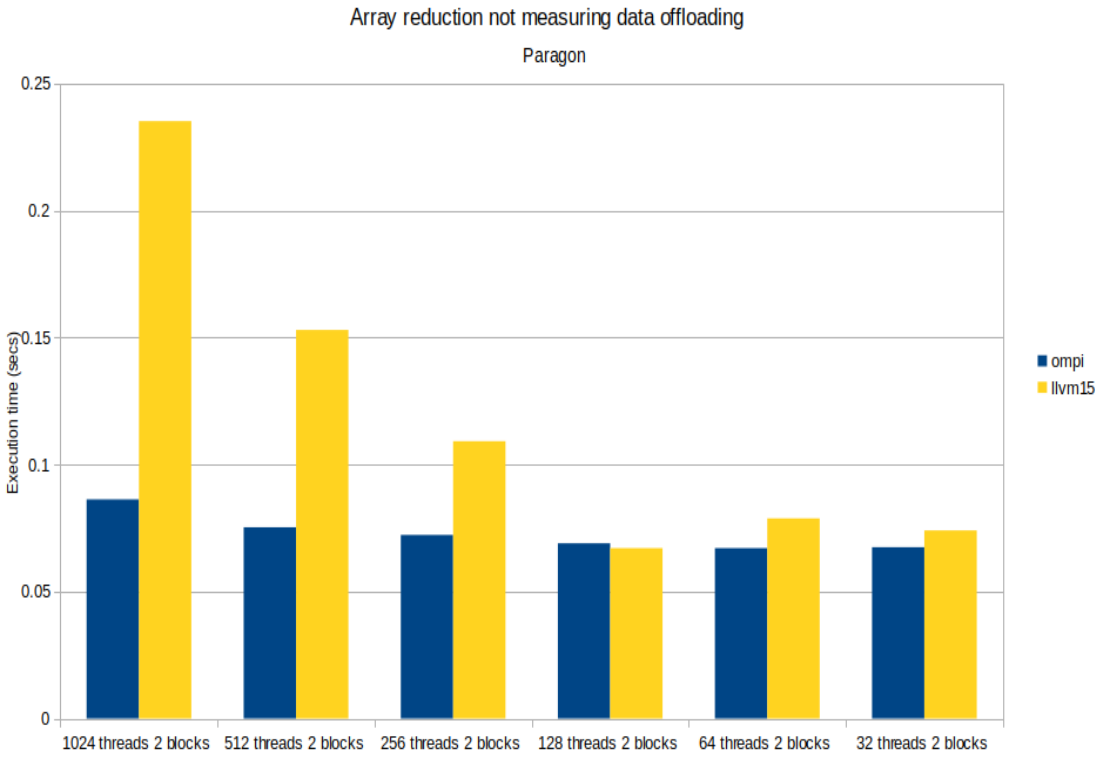
Σχήμα 5.11: Γράφημα αποτελεσμάτων μέτρησης υποβίβασης πίνακα με 512 νήματα στο σύστημα Parallax

Στο Σχήμα 5.12 παρουσιάζεται το γράφημα των αποτελεσμάτων μέτρησης υποβίβασης πίνακα με σταθερό αριθμό συνολικών νημάτων στο σύστημα Parallax. Ο άξονας x αναπαριστά τις διαφορετικές περιπτώσεις εκτέλεσης ενώ ο άξονας y αναπαριστά τον χρόνο εκτέλεσης σε δευτερόλεπτα. Ο άξονας y είναι σε λογαριθμική κλίμακα για καλύτερη αναπαράσταση των δεδομένων.



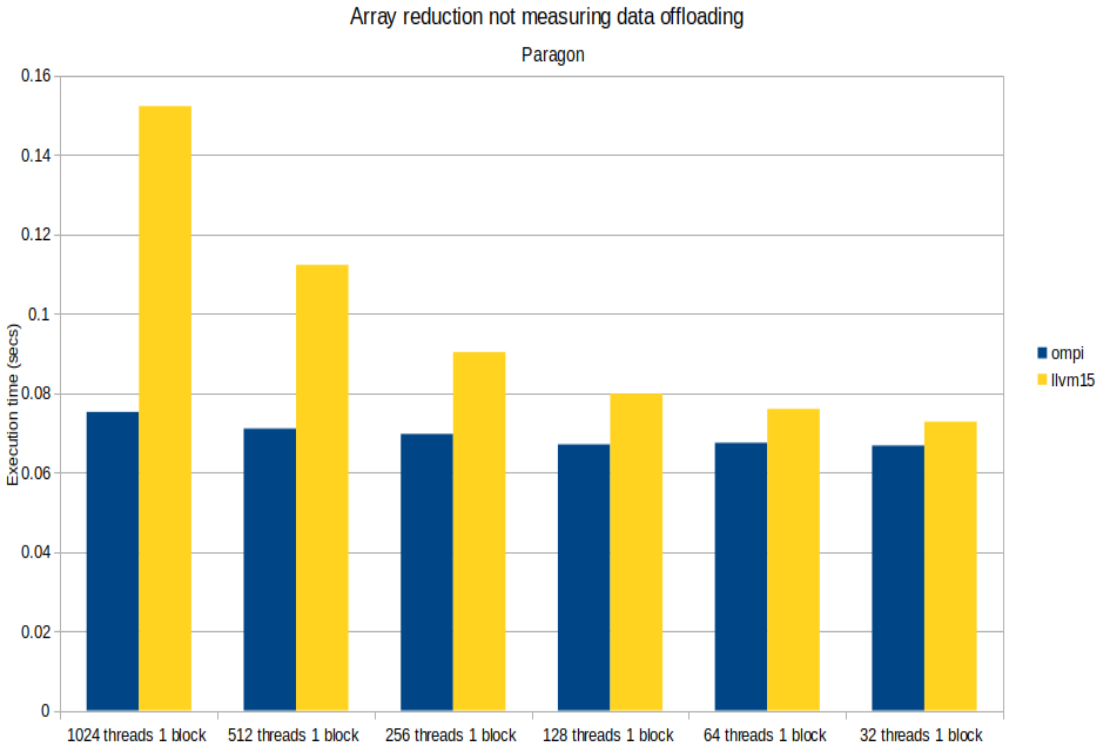
Σχήμα 5.12: Γράφημα αποτελεσμάτων μέτρησης υποβίβασης πίνακα με σταθερό αριθμό νημάτων στο σύστημα Parallax

Στο Σχήμα 5.13 παρουσιάζεται το γράφημα των αποτελεσμάτων μέτρησης υποβίβασης πίνακα με 2 blocks και για διαφορετικούς αριθμούς νημάτων στο σύστημα Paragon. Ο άξονας x αναπαριστά τις διαφορετικές περιπτώσεις εκτέλεσης ενώ ο άξονας y αναπαριστά τον χρόνο εκτέλεσης σε δευτερόλεπτα. Ο άξονας y είναι σε γραμμική κλίμακα.



Σχήμα 5.13: Γράφημα αποτελεσμάτων μέτρησης υποβίβασης πίνακα με 2 blocks στο σύστημα Paragon

Στο Σχήμα 5.14 παρουσιάζεται το γράφημα των αποτελεσμάτων μέτρησης υποβίβασης πίνακα με 1 block και για διαφορετικούς αριθμούς νημάτων στο σύστημα Paragon. Ο άξονας x αναπαριστά τις διαφορετικές περιπτώσεις εκτέλεσης ενώ ο άξονας y αναπαριστά τον χρόνο εκτέλεσης σε δευτερόλεπτα. Ο άξονας y είναι σε γραμμική κλίμακα.



Σχήμα 5.14: Γράφημα αποτελεσμάτων μέτρησης υποβίβασης πίνακα με 1 block στο σύστημα Paragon

Από τα παραπάνω σχήματα (Σχήμα 5.8, 5.9, 5.10, 5.11, 5.12) εξάγονται τα εξής συμπεράσματα:

- Και οι δύο εκδόσεις του OMPi έχουν παρόμοιους χρόνους εκτέλεσης σε όλες τις περιπτώσεις στο σύστημα Parallax
- Ο OMPi στην έκδοση με τα cooperative groups έχει τις καλύτερες επιδόσεις για όλες τις περιπτώσεις στο σύστημα Parallax
- Ο OMPi υπερέχει του LLVM (στην έκδοση 13 και 15) ανεξάρτητα από τον αριθμό των μπλοκ και των νημάτων και στα δύο μηχανήματα
- Ο OMPi στην έκδοση με τα cooperative groups δεν παράγει αποτελέσματα για αριθμό νημάτων ίσο με 1024 για λόγους που έχουν αναφερθεί στο παράδειγμα της άθροισης στοιχείων πίνακα

5.3.3 Εφαρμογή intensity normalization σε φωτογραφία

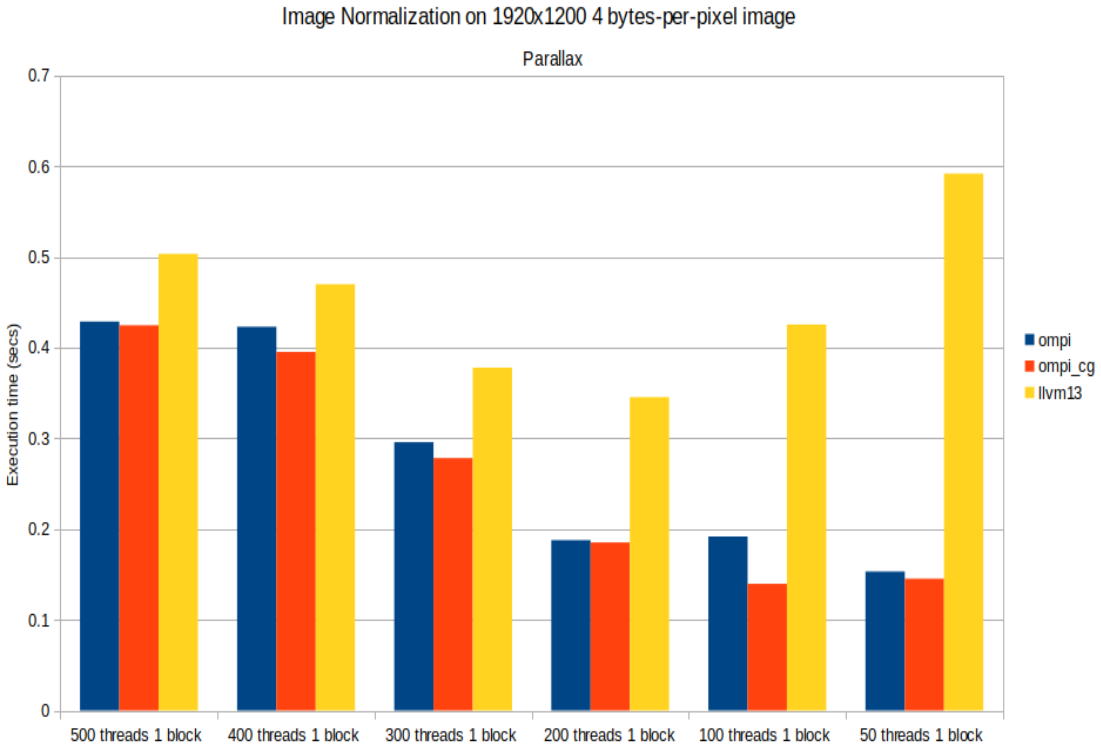
Η κανονικοποίηση (normalization)[10] είναι μια διαδικασία κατά την οποία οι εντάσεις των εικονοστοιχείων μίας εικόνας κλιμακώνονται γραμμικά. Οι εντάσεις των εικονοστοιχείων κανονικοποιούνται με βάση τον μέσο όρο της έντασης όλων των εικονοστοιχείων. Αυτό συνήθως εκτελείται για να φέρει τις εντάσεις σε ένα τυπικό εύρος. Αυτό επιτυγχάνεται με τον υπολογισμό πρώτα του μέσου όρου και της τυπικής απόκλισης των εντάσεων των εικονοστοιχείων και στη συνέχεια κλιμακώνοντάς τα όπου μ και σ είναι ο μέσος όρος και η τυπική απόκλιση των εντάσεων της εικόνας. Η εφαρμογή είναι γραμμένη χρησιμοποιώντας τις εντολές του προτύπου OpenMP και τμήμα της δίνεται στο σχήμα 5.15

```
#pragma omp target map(to: original_photo[0:totalSize]) map(to: n,mean,var) \
  map(from: transformed_photo[0:totalSize])

      #pragma omp parallel for shared(original_photo, n) \
        num_threads(500) reduction(+:mean) private(i) schedule(static)
            for(i=0; i<n; i++)
                  mean+= (float) (original_photo[i]);

      #pragma omp parallel for shared(original_photo, n) \
        num_threads(500) reduction(+:var) private(i, svar) schedule(static)
            for(i=0; i<n; i++)
            {
                  svar = (float) (original_photo[i]) - mean;
                  var += svar*svar;
            }
      var /= (float) n;
      std = sqrtf(var);

      #pragma omp parallel for shared(original_photo, transformed_photo, n) \
        num_threads(500) private(i) firstprivate(mean, std) schedule(static)
            for(i=0; i<n; i++)
                  transformed_photo[i] = (original_photo[i] – mean)/std;
```

Σχήμα 5.15: Τμήμα εφαρμογής intensity normalization σε φωτογραφία

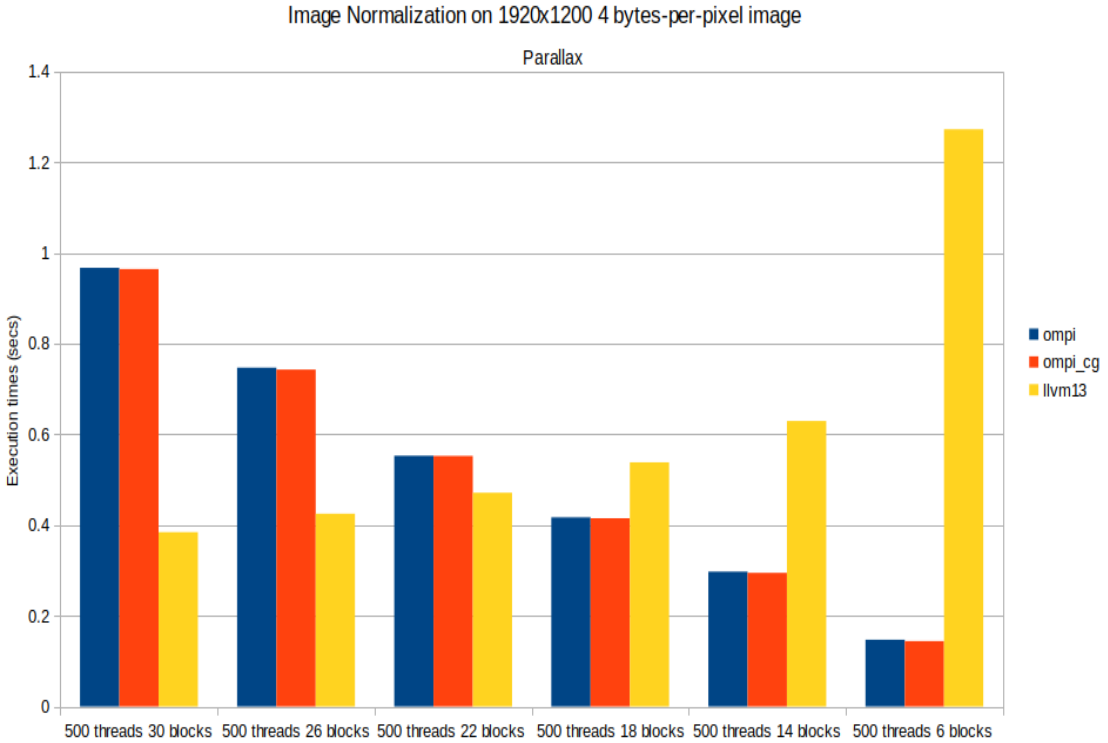
Οι εκτελέσεις του πειράματος δεν παρουσίασαν μεγάλη διακύμανση στους χρόνους και επομένως οι χρόνοι που παρουσιάζονται στα παρακάτω σχήματα προέκυψαν από τον μέσο όρο 5 εκτελέσεων της κάθε περίπτωσης.

Στο Σχήμα 5.16 παρουσιάζεται το γράφημα των αποτελεσμάτων μέτρησης εφαρμογής image normalization με 1 block και για διαφορετικούς αριθμούς νημάτων στο σύστημα Parallax. Ο άξονας x αναπαριστά τις διαφορετικές περιπτώσεις εκτέλεσης ενώ ο άξονας y αναπαριστά τον χρόνο εκτέλεσης σε δευτερόλεπτα. Ο άξονας y είναι σε γραμμική κλίμακα.



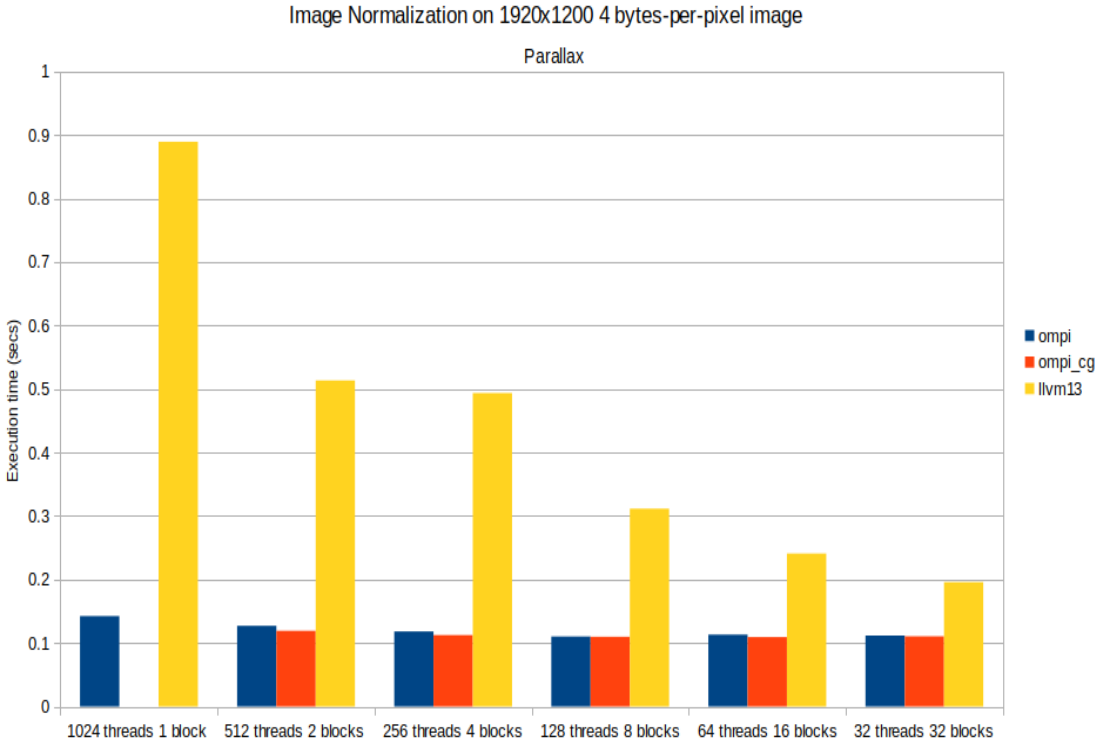
Σχήμα 5.16: Γράφημα αποτελεσμάτων μέτρησης εφαρμογής image normalization σε εικόνα μεγέθους 1920x1200 με 4 bytes-per-pixel με 1 block στο σύστημα Parallax

Στο Σχήμα 5.17 παρουσιάζεται το γράφημα των αποτελεσμάτων μέτρησης εφαρμογής image normalization με 500 νήματα και για διαφορετικούς αριθμούς block στο σύστημα Parallax. Ο άξονας x αναπαριστά τις διαφορετικές περιπτώσεις εκτέλεσης ενώ ο άξονας y αναπαριστά τον χρόνο εκτέλεσης σε δευτερόλεπτα. Ο άξονας y είναι σε γραμμική κλίμακα.



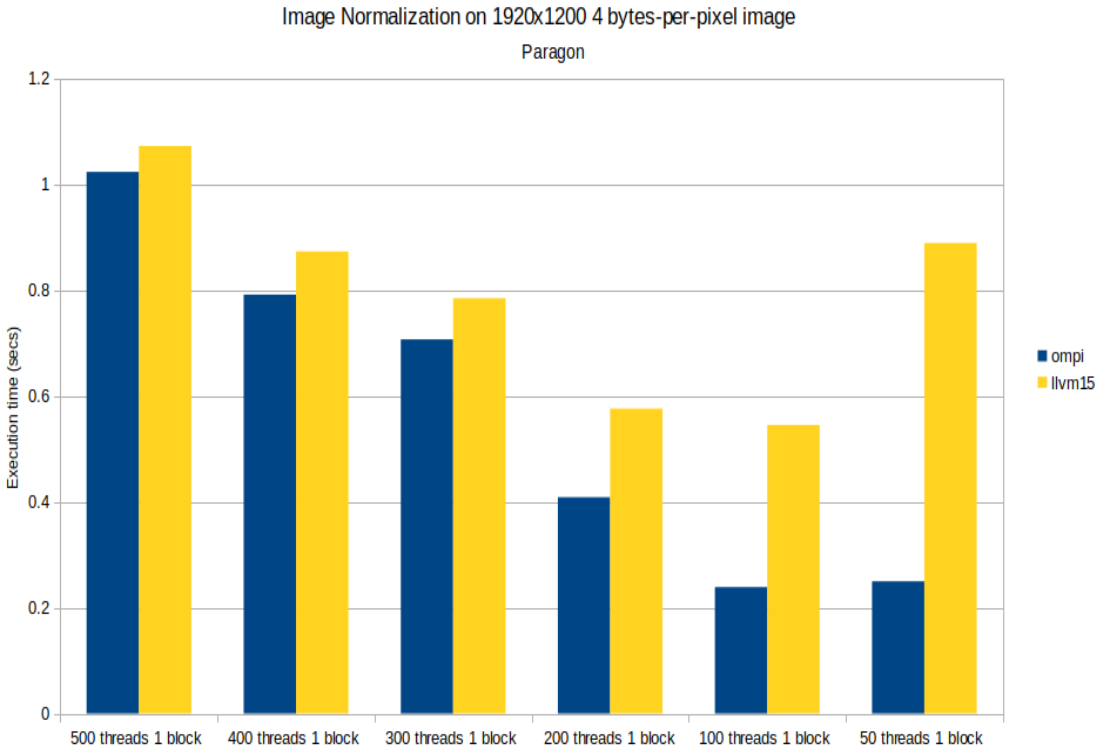
Σχήμα 5.17: Γράφημα αποτελεσμάτων μέτρησης εφαρμογής image normalization σε εικόνα μεγέθους 1920x1200 με 4 bytes-per-pixel με 500 νήματα στο σύστημα Parallax

Στο Σχήμα 5.18 παρουσιάζεται το γράφημα των αποτελεσμάτων μέτρησης εφαρμογής image normalization με σταθερό συνολικό αριθμό νημάτωνστο σύστημα Parallax. Ο άξονας x αναπαριστά τις διαφορετικές περιπτώσεις εκτέλεσης ενώ ο άξονας y αναπαριστά τον χρόνο εκτέλεσης σε δευτερόλεπτα. Ο άξονας y είναι σε γραμμική κλίμακα.



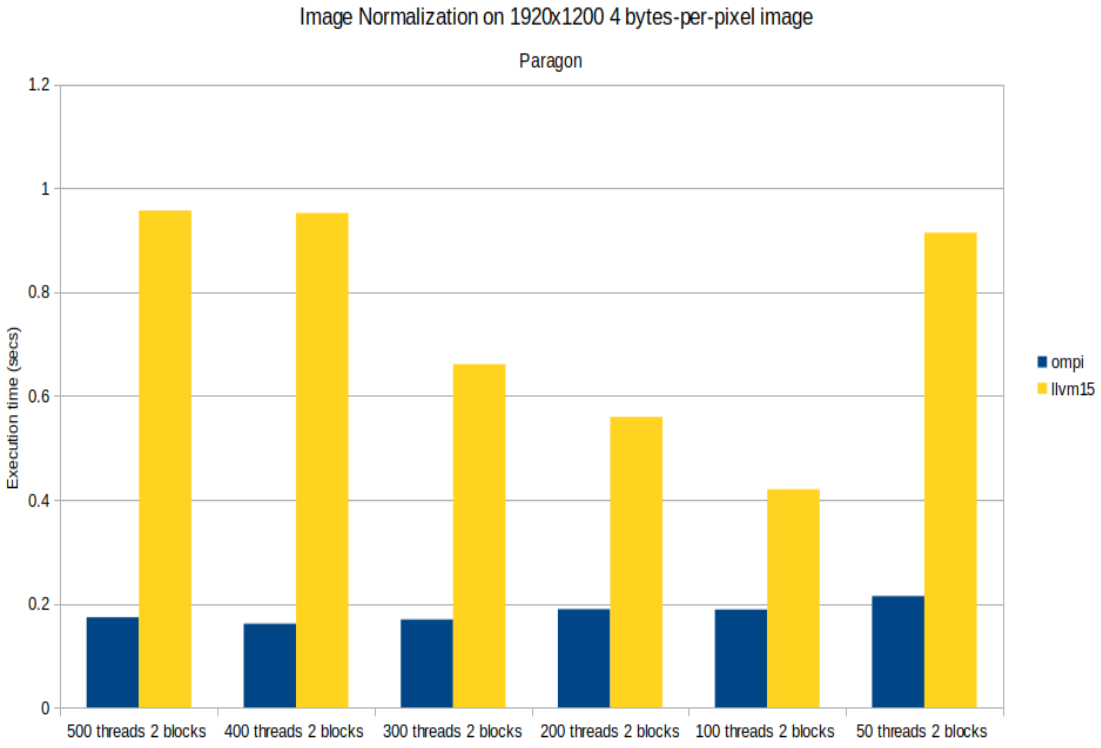
Σχήμα 5.18: Γράφημα αποτελεσμάτων μέτρησης εφαρμογής image normalization σε εικόνα μεγέθους 1920x1200 με 4 bytes-per-pixel με σταθερό αριθμό νημάτων στο σύστημα Parallax

Στο Σχήμα 5.19 παρουσιάζεται το γράφημα των αποτελεσμάτων μέτρησης εφαρμογής image normalization πίνακα με 1 block και για διαφορετικούς αριθμούς νημάτων στο σύστημα Paragon. Ο άξονας x αναπαριστά τις διαφορετικές περιπτώσεις εκτέλεσης ενώ ο άξονας y αναπαριστά τον χρόνο εκτέλεσης σε δευτερόλεπτα. Ο άξονας y είναι σε γραμμική κλίμακα.



Σχήμα 5.19: Γράφημα αποτελεσμάτων μέτρησης εφαρμογής image normalization σε εικόνα μεγέθους 1920x1200 με 4 bytes-per-pixel με 1 block στο σύστημα Paragon

Στο Σχήμα 5.20 παρουσιάζεται το γράφημα των αποτελεσμάτων μέτρησης εφαρμογής image normalization πίνακα με 1 block και για διαφορετικούς αριθμούς νημάτων στο σύστημα Paragon. Ο άξονας x αναπαριστά τις διαφορετικές περιπτώσεις εκτέλεσης ενώ ο άξονας y αναπαριστά τον χρόνο εκτέλεσης σε δευτερόλεπτα. Ο άξονας y είναι σε γραμμική κλίμακα.



Σχήμα 5.20: Γράφημα αποτελεσμάτων μέτρησης εφαρμογής image normalization σε εικόνα μεγέθους 1920x1200 με 4 bytes-per-pixel με 2 block στο σύστημα Paragon

Από τα παραπάνω σχήματα (Σχήμα 5.14, 5.15, 5.16, 5.17) εξάγονται τα εξής συμπεράσματα:

- Οι δύο εκδόσεις του OMPi έχουν παρόμοιους χρόνους εκτέλεσης σε όλες τις περιπτώσεις στο σύστημα Parallax
- Ο OMPi στην έκδοση με τα cooperative groups έχει τις καλύτερες επιδόσεις για όλες τις περιπτώσεις στο σύστημα Parallax
- Ο OMPi υπερέχει του LLVM (στην έκδοση 13 και 15) ανεξάρτητα από τον αριθμό των μπλοκ και των νημάτων και στα δύο μηχανήματα
- Οι αριθμοί των μπλοκ που έχουν επιλεχθεί επιβεβαιώνουν ότι η υλοποίηση στον OMPi λειτουργεί αποδοτικά και για πλήθος νημάτων που δεν είναι δύναμη του δύο.
- Ο OMPi στην έκδοση με τα cooperative groups έχει τις καλύτερες επιδόσεις για αριθμό block μικρότερο ίσο του 18 στο σύστημα Parallax
- Ο LLVM έχει καλύτερες επιδόσεις για αριθμό block μεγαλύτερο ίσο του 22 στο σύστημα Parallax

Κεφάλαιο 6.
Σύνοψη

6.1 Συμπεράσματα

Οι ανάγκες που προκύπτουν για υπολογιστικά συστήματα υψηλών επιδόσεων έχουν φέρει στο προσκήνιο τη χρήση των μονάδων γραφικής επεξεργασίας ως συσκευές γενικού σκοπού. Ενώ μέχρι και τη προηγούμενη δεκαετία η χρήση τους αφορούσε αποκλειστικά τη παραγωγή 3Δ γραφικών πλέον με την ανάπτυξη της πλατφόρμας CUDA, η οποία επιτρέπει των προγραμματισμό των πυρήνων μιας μονάδας γραφικής επεξεργασίας, μέσω μίας διεπαφής (API), μπορούν πλέον να χρησιμοποιηθούν για την επίλυση προβλημάτων γενικού σκοπού. Οι μονάδες γραφικής επεξεργασίας προσφέρουν υψηλότερες επιδόσεις συγκριτικά με τις κύριες μονάδες επεξεργασίας.
Στη παρούσα διπλωματική εργασία υλοποιήθηκε ο μηχανισμός υποβίβασης (reduction) στο παραλληλοποιητικό μεταφραστή OMPi, στοχεύοντας επιταχυντές γραφικής επεξεργασίας CUDA. Η υλοποίηση καλύπτει όλους τους τελεστές υποβίβασης που ορίζει το πρότυπο OpenMP καθώς και όλους τους τύπους δεδομένων που υποστηρίζει η γλώσσα προγραμματισμού C. Επιπλέον, με τη χρήση βελτιστοποιημένων αλγόριθμων μειώθηκαν στο ελάχιστο τα φαινόμενα του brach divergence, των memory bank conflict και των inactive threads τα οποία μειώνουν την απόδοση των προγραμμάτων και εκμεταλλεύονται σε υψηλό βαθμό την υπολογιστική ισχύ που προσφέρουν οι μονάδες γραφικής επεξεργασίας.

Από τα αποτελέσματα των πειραμάτων που πραγματοποιήθηκαν αποδείχθηκε ότι:

- Η υλοποίηση στον μεταφραστή OMPi είναι τάξεις μεγέθους πιο αποδοτική από το μεταφραστή LLVM στη περίπτωση που το πρόγραμμα εκτελείται σε ένα μπλοκ της μονάδας γραφικής επεξεργασίας ανεξάρτητα από τον αριθμό νημάτων του μπλοκ.

- Στη περίπτωση εκτέλεσης της εφαρμογής άθροισης στοιχείων πίνακα για αριθμό μπλοκ≤10 η υλοποίηση του OMPi είναι πιο αποδοτική από τον LLVM 13, ενώ για αριθμό μπλοκ≥15 ο μεταφραστής LLVM 13 είναι τάξεις μεγέθους πιο αποδοτικός από το μεταφραστή OMPi.
- Στην περίπτωση εκτέλεσης της εφαρμογής υποβίβασης πίνακα παρατηρήσαμε ότι ο OMPi είναι πιο αποδοτικός από το το μεταφραστή LLVM σε όλες τις περιπτώσεις που μελετήθηκαν.
- Στη περίπτωση εκτέλεσης της εφαρμογής intensity normalization σε φωτογραφία για αριθμό μπλοκ≤18 η υλοποίηση του OMPi είναι πιο αποδοτική από τον LLVM 13, ενώ για αριθμό μπλοκ≥22 ο μεταφραστής LLVM 13 είναι πιο αποδοτικός από το μεταφραστή OMPi.

Από τις παραπάνω παρατηρήσεις προκύπτει ότι ο τροποποιημένος αλγόριθμος Sequential Addressing with reversed loop and threadID-based που χρησιμοποιεί ο OMPi για υποβίβαση εντός του μπλοκ αποτελεί πολύ αποδοτική υλοποίηση καθώς εκμεταλλεύεται τις δυνατότητες που παρέχει το προγραμματιστικό μοντέλο CUDA.

Επίσης διαπιστώνεται ότι ο μηχανισμός συγχρονισμού μεταξύ των μπλοκ που υλοποιείται στον OMPi δεν είναι τόσο αποδοτικός σε σχέση με αυτόν που υλοποιεί ο LLVM σε συγκεκριμένα πειράματα.

Τέλος από τα αποτελέσματα των μετρήσεων προκύπτει ότι η υλοποίηση στον OMPi με τη χρήση Cooperative Groups δεν προσφέρει κάποια σημαντική βελτίωση στους χρόνους εκτέλεσης

6.2 Μελλοντική Εργασία

Η εργασία αυτή οδήγησε σε εξαιρετικές επιδόσεις, ειδικά εντός ενός block, ακόμα και σε σύγκριση με τον πιο προχωρημένο μεταφραστή OpenMP αυτή τη στιγμή, τον LLVM. Παρόλα αυτά, υπάρχουν περιθώρια για μελλοντικές βελτιώσεις και επεκτάσεις.

- Για τη περαιτέρω εξέλιξη του μηχανισμού υποβίβασης θα πρέπει να μελετηθεί μία υλοποίηση η οποία βελτιώνει την απόδοση του μηχανισμού συγχρονισμού μεταξύ των μπλοκ μίας μονάδας γραφικής επεξεργασίας. Μία τέτοια υλοποίηση θα μπορούσε να περιλαμβάνει μια υβριδική προσέγγιση όπου τα υποαποτέλεσμα που συλλέγονται από το κάθε μπλοκ να υποβιβάζονται τελικά από τη κεντρική μονάδα επεξεργασίας.
- Επίσης θα μπορούσε να υλοποιηθεί ένας έλεγχος επιλογής της κατάλληλης έκδοσης του OMPi, (υλοποίηση με κλειδαριές ή υλοποίηση με cooperative groups) ανάλογα με το τι υποστηρίζει η

μονάδα γραφικής επεξεργασίας του είναι εγκατεστημένη στο εκάστοτε σύστημα.

- Ακόμα θα μπορούσε να εισαχθεί ένας μηχανισμός δυναμικής δέσμευσης της κοινόχρηστης μνήμης που χρησιμοποιείται από τη λειτουργία υποβίβασης έτσι ώστε να μην δεσμεύεται παραπάνω μνήμη από αυτή που απαιτεί ένα συγκεκριμένο πρόγραμμα.
- Οι μηχανισμοί και οι υλοποιήσεις που παρουσιάστηκαν, μπορούν με κατάλληλες τροποποιήσεις, να χρησιμοποιηθούν για να συσκευές γραφικής επεξεργασίας που υποστηρίζουν το πρότυπο OpenCL. Ο OMPi πρόκειται να επεκταθεί ώστε να υποστηρίζει και τέτοιου είδους συσκευές και οι τεχνικές μας θα μπορούσαν να συνεισφέρουν σε αυτή τη κατεύθυνση.

Βιβλιογραφία

[1] Vingelmann, P., and F. H. Fitzek. "Cuda, release: 10.2. 89." 2020, *NVIDIA*. [Online]. Available: https://developer.nvidia.com/cuda-toolkit

[2] Munshi, Aaftab. "The opencl specification." *2009 IEEE Hot Chips 21 Symposium (HCS)*. IEEE, 2009.

[3] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, and J. McDonald, "Parallel programming in OpenMP". Morgan kaufmann, 2001.

[4] S.N. Agathos, A. Papadogiannakis, V.V. Dimakopoulos, "Targeting the Parallella", in *Proc. Euro-Par 2015, 21st Int'l Conference on Parallel and Distributed Computing*, Vienna, Austria, Aug. 2015, pp. 662--674

[5] F. Sitaras, *Αποδοτική εκτέλεση προγραμμάτων OpenMP σε συστάδες Η/Υ*, Ptychion Thesis (in Greek), No. PPG-15D8, Dept. of Computer Science, Univ. of Ioannina, Sept. 2009

[6] NVIDIA: CUDA Driver API. [Online]. Available: https://docs.nvidia.com/cuda/cuda-driver-api/index.html

[7] Harris, Mark. "Optimizing parallel reduction in CUDA." *Nvidia developer technology* 2.4 (2007): 70. [Online]. Available: https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

[8] NVIDIA, Kyrylo Perelygin and Yuan Lin: "Cooperative Groups" October 2017. [Online]. Available: https://on-demand.gputechconf.com/gtc/2017/presentation/s7622-Kyrylo-perelygin-robust-and-scalable-cuda.pdf

[9] Joshua Hoke Davis: "The SOLLVE OMPVV offloading verification and validation test suite handbook" March 2021. [Online]. Available: https://crpl.cis.udel.edu/ompvvsollve/Documentation/_index.files/OMPVV_Handbook_v1.1.pdf

[10] Greg Slabaugh, Richard Boyes, Xiaoyun Yang Multicore "Image Processing with OpenMP" Queen Mary University of London, December 2020. [Online]. Available: http://www.eecs.qmul.ac.uk/~gslabaugh/publications/OpenMP_SPM.pdf

Παράρτημα

Άθροιση στοιχείων πίνακα μεγέθους δύο εκατομμυρίων

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#define N 2000000

int main(int argc, char *argv[])
{
        double start=0.0, finish=0.0;
        int i, a[N];
        for(i=0; i<N; i++)
        a[i]=rand()%10;
        int serial_sum = 0;
        for(i=0; i<N; i++)
                serial_sum+=a[i];
        int parallel_sum=0;

        #pragma omp target map(to:a[0:N])
        printf("> ");

        start = omp_get_wtime();
        #pragma omp target teams distribute parallel for \
          num_threads(512) num_teams(10) reduction(+: parallel_sum)
                                for(i=0; i<N; i++)
                        parallel_sum+=a[i];
        finish = omp_get_wtime();
        printf("Serial sum is %d\n", serial_sum);
        printf("Parallel sum is %d\n", parallel_sum);
        if(serial_sum==parallel_sum)
        printf("Test passed, results match\n");
        else
        printf("Test failed, results don't match\n");

        printf("Time elapsed = %f secs\n", (finish-start));
        return 0;
}
```

Υποβίβαση πίνακα

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#define N 1024*30


int main()
{
        double start=0.0, finish=0.0;
        int A[N] = {0};

        start = omp_get_wtime();
        #pragma omp target teams distribute parallel for \
          num_threads(1024) num_teams(30) reduction(+:A[0:N])
             for (int i = 0; i < omp_get_num_threads(); i++)
              A[i] = 1;
        finish = omp_get_wtime();

        printf("Time elapsed = %f secs\n", (finish-start));

        return 0;
}
```

Εφαρμογή intensity normalization σε φωτογραφία

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>
#define DATA_OFFSET_OFFSET 0x000A
#define WIDTH_OFFSET 0x0012
#define HEIGHT_OFFSET 0x0016
#define BITS_PER_PIXEL_OFFSET 0x001C
#define HEADER_SIZE 14
#define INFO_HEADER_SIZE 40
#define NO_COMPRESION 0
#define MAX_NUMBER_OF_COLORS 0
#define ALL_COLORS_REQUIRED 0

void ReadImage(const char *fileName,unsigned char **pixels, unsigned int *width,
unsigned int *height, unsigned int *bytesPerPixel)
{
        FILE *imageFile = fopen(fileName, "rb");
        unsigned int dataOffset;
        fseek(imageFile, DATA_OFFSET_OFFSET, SEEK_SET);
        fread(&dataOffset, 4, 1, imageFile);
        fseek(imageFile, WIDTH_OFFSET, SEEK_SET);
        fread(width, 4, 1, imageFile);
        fseek(imageFile, HEIGHT_OFFSET, SEEK_SET);
        fread(height, 4, 1, imageFile);
        short  bitsPerPixel;
        fseek(imageFile, BITS_PER_PIXEL_OFFSET, SEEK_SET);
        fread(&bitsPerPixel, 2, 1, imageFile);
        *bytesPerPixel = ((unsigned int)bitsPerPixel) / 8;
        int paddedRowSize = (int)(4 * ceil((float)(*width) / 4.0f))*(*bytesPerPixel);
        int unpaddedRowSize = (*width)*(*bytesPerPixel);
        int totalSize = unpaddedRowSize*(*height);
        *pixels = (unsigned char*)malloc(totalSize);
        int i = 0;

        unsigned char*currentRowPointer = *pixels+((*height-1)*unaddedRowSize);
       for (i = 0; i < *height; i++)
       {
               fseek(imageFile, dataOffset+(i*paddedRowSize), SEEK_SET);
               fread(currentRowPointer, 1, unpaddedRowSize, imageFile);
               currentRowPointer -= unpaddedRowSize;
       }

       fclose(imageFile);
}
```

```
void WriteImage(const char *fileName, unsigned char *pixels, unsigned int width,
unsigned int height,unsigned int bytesPerPixel)
{
	FILE *outputFile = fopen(fileName, "wb");
	const char *BM = "BM";
	fwrite(&BM[0], 1, 1, outputFile);
	fwrite(&BM[1], 1, 1, outputFile);
	int paddedRowSize = (int)(4 * ceil((float)width/4.0f))*bytesPerPixel;
	unsigned int fileSize = paddedRowSize*height + HEADER_SIZE +
	INFO_HEADER_SIZE;
	fwrite(&fileSize, 4, 1, outputFile);
	unsigned int reserved = 0x0000;
	fwrite(&reserved, 4, 1, outputFile);
	unsigned int dataOffset = HEADER_SIZE+INFO_HEADER_SIZE;
	fwrite(&dataOffset, 4, 1, outputFile);

	unsigned int infoHeaderSize = INFO_HEADER_SIZE;
	fwrite(&infoHeaderSize, 4, 1, outputFile);
	fwrite(&width, 4, 1, outputFile);
	fwrite(&height, 4, 1, outputFile);
	short  planes = 1;
	fwrite(&planes, 2, 1, outputFile);
	short  bitsPerPixel = bytesPerPixel * 8;
	fwrite(&bitsPerPixel, 2, 1, outputFile);
	unsigned int compression = NO_COMPRESION;
	fwrite(&compression, 4, 1, outputFile);
	unsigned int imageSize = width*height*bytesPerPixel;
	fwrite(&imageSize, 4, 1, outputFile);
	unsigned int resolutionX = 11811;
	unsigned int resolutionY = 11811;
	fwrite(&resolutionX, 4, 1, outputFile);
	fwrite(&resolutionY, 4, 1, outputFile);
	unsigned int colorsUsed = MAX_NUMBER_OF_COLORS;
	fwrite(&colorsUsed, 4, 1, outputFile);
	unsigned int importantColors = ALL_COLORS_REQUIRED;
	fwrite(&importantColors, 4, 1, outputFile);

	int i = 0;
	int unpaddedRowSize = width*bytesPerPixel;
	for ( i = 0; i < height; i++)
	{
		int pixelOffset = ((height - i) - 1)*unpaddedRowSize;
		fwrite(&pixels[pixelOffset], 1, paddedRowSize, outputFile);
	}
	fclose(outputFile);
}
```

```
int main()
{
        unsigned char *original_photo;
        unsigned char *transformed_photo;
        unsigned int width = 1920;
        unsigned int height = 1200;
        unsigned int bytesPerPixel = 4;
        int totalSize = width*height*bytesPerPixel;
        int n = width*height*bytesPerPixel;
        float mean = 0.0;
        float var = 0.0;
        int i, j;
        float svar;
        float std;
        double start=0.0, finish=0.0;
        transformed_photo = (unsigned char *)malloc(totalSize);

        ReadImage("landscape.bmp", &original_photo, &width,
&height,&bytesPerPixel);
        start = omp_get_wtime();

        #pragma omp target map(to: original_photo[0:totalSize]) \
          map(to: n,mean,var) map(from: transformed_photo[0:totalSize])

                #pragma omp parallel for shared(original_photo, n) \
                  num_threads(500) reduction(+:mean) private(i) schedule(static)
                        for(i=0; i<n; i++)
                                mean+= (float) (original_photo[i]);

                #pragma omp parallel for shared(original_photo, n)  \
                  num_threads(500) reduction(+:var) private(i, svar) \
                  schedule(static)
                        for(i=0; i<n; i++)
                        {
                                svar = (float) (original_photo[i]) - mean;
                                var += svar*svar;
                        }
                var /= (float) n;
                std = sqrtf(var);

                #pragma omp parallel for num_threads(500) private(i)\
                   shared(original_photo, transformed_photo, n) \
                    firstprivate(mean, std) schedule(static)
                        for(i=0; i<n; i++) {
                        transformed_photo[i] = (original_photo[i] - mean)/std;
                        }
```

```
        finish = omp_get_wtime();
        WriteImage("landscape_normalized.bmp", transformed_photo, width,
        height, bytesPerPixel);

        free(original_photo);
        free(transformed_photo);
        printf("Time elapsed = %f secs\n", (finish-start));
        return 0;
}
```