Στοχεύοντας επιταχυντές γραφικών βασισμένους σε CUDA μέσω OpenMP

Η Μεταπτυχιακή Διπλωματική Εργασία

υποβάλλεται στην ορισθείσα
από την Συνέλευση
του Τμήματος Μηχανικών Η/Υ και Πληροφορικής
Εξεταστική Επιτροπή

από τον

Ηλία Κασμερίδη

ως μέρος των υποχρεώσεων για την απόκτηση του

ΔΙΠΛΩΜΑΤΟΣ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ
ΣΤΗ ΜΗΧΑΝΙΚΗ ΔΕΔΟΜΕΝΩΝ ΚΑΙ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

ΜΕ ΕΙΔΙΚΕΥΣΗ
ΣΤΑ ΠΡΟΗΓΜΕΝΑ ΥΠΟΛΟΓΙΣΤΙΚΑ ΣΥΣΤΗΜΑΤΑ

Πανεπιστήμιο Ιωαννίνων

Ιανουάριος 2021

Εξεταστική Επιτροπή:

- **Βασίλειος Δημακόπουλος**, Αναπλ. Καθηγητής, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πανεπιστήμιο Ιωαννίνων (Επιβλέπων)
- **Γεώργιος Μανής**, Αναπλ. Καθηγητής, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πανεπιστήμιο Ιωαννίνων
- **Ιωάννης Φούντος**, Καθηγητής, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πανεπιστήμιο Ιωαννίνων

Αφιέρωση

Στη Φωτεινή

Ευχαριστιες

Ευχαριστώ τον επιβλέποντα καθηγητή μου, κ. Βασίλειο Δημακόπουλο, για την καθοδήγησή μου καθ' όλη τη διάρκεια εκπόνησης της παρούσας εργασίας και τη γενικότερη στήριξη που μου προσέφερε, από την ένταξή μου στην ομάδα παράλληλης επεξεργασίας του Τμήματος Μηχανικών Η/Υ και Πληροφορικής του Πανεπιστημίου Ιωαννίνων, έως και σήμερα. Επιπλέον, ευχαριστώ τα υπόλοιπα μέλη της ομάδας, για την άψογη συνεργασία και ιδιαίτερα τον Γεώργιο Ζάχο για την πολύτιμη βοήθεια του.

Περιεχομενα

Κατάλογος Σχημάτων

Κατάλογος Πινάκων

Περιληψη

Ηλίας Κασμερίδης, Δ.Μ.Σ. στη Μηχανική Δεδομένων και Υπολογιστικών Συστημάτων, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πολυτεχνική Σχολή, Πανεπιστήμιο Ιωαννίνων, Ιανουάριος 2021.
Στοχεύοντας επιταχυντές γραφικών βασισμένους σε CUDA μέσω OpenMP.
Επιβλέπων: Βασίλειος Δημακόπουλος, Αναπληρωτής Καθηγητής.

Τα ετερογενή συστήματα τείνουν να γίνουν τα συνηθέστερα στον κόσμο της παράλληλης επεξεργασίας, με τους επιταχυντές γραφικών (GPUs) να αποτελούν πλέον το δεύτερο βασικό συστατικό τους, μετά τους κύριους επεξεργαστές. Με την εξέλιξη των εξειδικευμένων μονάδες γραφικής επεξεργασίας σε γενικότερου σκοπού, δόθηκε η δυνατότητα χρήσης του γραφικού υλικού από απλούς προγραμματιστές εφαρμογών, μέσω κατάλληλων διεπαφών. Η CUDA είναι ένα παράδειγμα διεπαφής και πλατφόρμας προγραμματισμού υψηλών επιδόσεων, η οποία επιτρέπει στους προγραμματιστές να χρησιμοποιούν τους πυρήνες μιας συσκευής γραφικής επεξεργασίας για γενικούς παράλληλους υπολογισμούς. Όμως, η συγγραφή ενός προγράμματος που κάνει χρήση CUDA είναι μια δύσκολη διαδικασία, καθώς η ετερογένεια του υλικού μεταφράζεται και σε προγραμματιστική ετερογένεια· απαιτούνται δύο ειδών κώδικες, ένας για τους κύριους επεξεργαστές και ένας για τους επιταχυντές γραφικών που υπάρχουν στο σύστημα. Για την αντιμετώπιση προκλήσεων αυτής της φύσης, προγραμματιστικά πρότυπα όπως το OpenMP έχουν εισάγει νέα σύνολα οδηγιών ώστε να γίνεται απρόσκοπτα η συγγραφή ενιαίου κώδικα και η αυτόματη φόρτωση και εκτέλεση τμημάτων του σε συνοδές υπολογιστικές συσκευές. Στην παρούσα εργασία περιγράφεται μία υλοποίηση των οδηγιών αυτών στο πλαίσιο του παραλληλοποιητικού μεταφραστή OMPi, στοχεύοντας επιταχυντές που βασίζονται στο μοντέλο της CUDA. Για την υποστήριξη των οδηγιών, εμπλουτίστηκε το τμήμα μεταγλώττισης του μεταφραστή, το οποίο περιλαμβάνει τη συντακτική ανάλυση των οδηγιών OpenMP και τον μετασχηματισμό τους σε κώδικα C. Επιπλέον, εισή-

χθησαν βοηθητικές λειτουργίες που αφορούν το αφηρημένο συντακτικό δέντρο του OMPi, μια δενδρική απεικόνιση της συντακτικής δομής του πηγαίου κώδικα χρήστη. Οι λειτουργίες αυτές καθιστούν εύκολη την μελλοντική εισαγωγή νέων οδηγιών ή επέκταση των ήδη υπάρχοντων. Τέλος, δημιουργήθηκε η βιβλιοθήκη υποστήριξης εκτέλεσης CUDADEV, η οποία αποτελεί τη γέφυρα επικοινωνίας μεταξύ του κύριου συστήματος και των επιταχυντών αυτής της κλάσης. Για την CUDADEV, υλοποιήθηκε τόσο η διεπαφή hostpart, η οποία αποτελεί τον πυρήνα της επικοινωνίας κυρίου συστήματος-συσκευής, όσο και το τμήμα devpart, ένα επίπεδο υποστήριξης λειτουργικότητας OpenMP στο εσωτερικό του κώδικα που εκτελείται στη συσκευή. Το τμήμα hostpart της συσκευής CUDADEV, χρησιμοποιεί στο εσωτερικό του τη προγραμματιστική διεπαφή CUDA Driver, για τον χειρισμό μιας μονάδας γραφικής επεξεργασίας CUDA.

Μέσω της εκτέλεσης δοκιμαστικών εφαρμογών σε δύο διαφορετικά συστήματα που διαθέτουν δύο εντελώς διαφορετικές μονάδες γραφικής επεξεργασίας CUDA, επιβεβαιώνεται η ορθότητα των νέων υλοποιήσεων, όπως επίσης και αξιολογούνται οι επιδόσεις τους. Συνεπάγεται ότι η νέα λειτουργικότητα μπορεί να επιτύχει έως και τρεις φορές καλύτερες επιδόσεις, συγκριτικά με την παραλληλοποίηση στον κύριο επεξεργαστή μέσω της αντίστοιχης οδηγίας OpenMP, ενώ για σειριακά προγράμματα, η επιτάχυνση δύναται να φτάσει τις 170 φορές.

Extended Abstract

Ilias Kasmeridis, M.Sc. in Data and Computer Systems Engineering, Department of Computer Science and Engineering, School of Engineering, University of Ioannina, Greece, January 2021.
Targetting CUDA-based graphics accelerators through OpenMP.
Advisor: Vassilios Dimakopoulos, Associate Professor.

Heterogeneous systems tend to become a common place in the world of parallel processing systems, with graphics accelerators (GPUs) now being their second key component, after main processors. With the evolution of graphics processing units into general purpose devices, developers were given the opportunity to use their hardware programmatically, through suitable interfaces. CUDA is an example of a high-performance programming interface and platform that allows developers to use the cores of a graphics processing unit for general parallel computing. However, writing a program that utilizes the CUDA functionality is a difficult process, as hardware heterogeneity translates into programming heterogeneity; two types of code are required, the first one is executed by the main processor and is responsible for any type of device handling, while the second one pertains to the graphics accelerators in the system. To tackle challenges of this nature, high-performance programming standards such as OpenMP have introduced new sets of directives, allowing programmers to write unified code and automatically offload and execute parts of it on computing devices. This work describes an implementation of these directives in the context of the OMPi comiler, targeting accelerators based on the CUDA model. To support the directives, we extended OMPi compiler functionality, which typically includes the analysis of the OpenMP instructions and their transformation to C code. In addition, we introduced some auxiliary features related to the abstract syntax tree, a tree representation of the user source code syntax. These features enable the introduction new directives or the extension of existing ones in the future. Finally, this work presents

CUDADEV, a runtime library which enables communication between the host system and graphics accelerators of this class.. CUDADEV consists of the hostpart interface, the core of the main processor-device communication, as well as the devpart library that brings OpenMP functionality support within the code running on the device. The hostpart section of the library, utilizes the CUDA Driver API in order to handle the targetted CUDA device.

By running different benchmarking applications on two different systems with two completely different CUDA GPUs, we verified the corectness of the new implementations, as well as evaluated their performance. As far as the results are concerned, the new functionality can achieve up to three times better performance than the main processor parallelism through the corresponding OpenMP directive, while for serial programs, the speedup can reach 170 times.

Κεφαλαιο 1

Εισαγωγη

1.1 Εξέλιξη των Η/Υ

Στη σημερινή εποχή, η εξέλιξη της τεχνολογίας συνδέεται άμεσα με την ανάπτυξη των ηλεκτρονικών υπολογιστών. Η εξέλιξη ενός ηλεκτρονικού υπολογιστή είναι ένα σύνθετο πρόβλημα βελτιστοποίησης κάθε χαρακτηριστικού του - της ταχύτητας υπολογισμών του, της χωρητικότητας μνήμης του και της κατανάλωσης ισχύος του. Ο κύριος επεξεργαστής αποτελεί το βασικό συστατικό ενός υπολογιστικού συστήματος και οι επιδόσεις του επηρεάζουν άμεσα με τις συνολικές επιδόσεις του συστήματος. Παραδοσιακά, τις τελευταίες δεκαετίες, η βελτίωση της τεχνολογίας των επεξεργαστών πραγματοποιείται με τη μείωση των διαστάσεων του τσιπ τους, τον διπλασιασμό των τρανζίστορ που χρησιμοποιούνται σε αυτό και την αύξηση της συχνότητας λειτουργίας τους. Με το πέρασμα των χρόνων αποδείχτηκε ότι η αύξηση στη συχνότητα λειτουργίας των τρανζίστορ γίνεται ολοένα και πιο δύσκολη, λόγω της μεγάλης κατανάλωσης ισχύος που προκύπτει. Για να αντιμετωπίσουν αυτό το

πρόβλημα και, συγχρόνως, να διατηρήσουν τις υψηλές επιδόσεις, οι κατασκευαστές αναγκάστηκαν να ρίξουν τη συχνότητα των τρανζίστορ και να μεταβούν στη χρήση πολλαπλών πυρήνων εντός του επεξεργαστή, καταλήγωντας στο μοντέλο του πολυπύρηνου επεξεργαστή (multicore processor).

Οι πολυπύρηνοι επεξεργαστές αξιοποιήθηκαν από την παράλληλη επεξεργασία, το μοντέλο της οποίας βασίζεται στην ταυτόχρονη εκτέλεση υπολογισμών από τους πυρήνες των επεξεργαστών. Την σημερινή εποχή, οι παράλληλοι υπολογιστές είναι ουσιαστικά το μοναδικό είδος υπολογιστών που συναντάται.

1.2 Παράλληλα Υπολογιστικά Συστήματα

Γενικότερος στόχος της παράλληλης επεξεργασίας είναι η επίτευξη του ταυτοχρονισμού (concurrency), με την έννοια της ταυτόχρονης εκτέλεση υπολογισμών και χειρισμών δεδομένων από μία ή περισσότερες οντότητες που εκτελούν κοινό πρόγραμμα και συνεργάζονται για τη λύση ενός προβλήματος. Υλοποίηση του μοντέλου της παράλληλης επεξεργασίας αποτελούν τα παράλληλα συστήματα, τα οποία διαθέτουν ένα σύνολο επεξεργαστικών μονάδων που επικοινωνούν μεταξύ τους σε πραγματικό χρόνο, ώστε να υπολογίσουν ένα αποτέλεσμα ταχύτερα.

Οι βασικές κατηγορίες των παράλληλων συστημάτων είναι τα συστήματα κοινόχρηστης μνήμης (shared memory) και κατανεμημένης μνήμης (distributed memory). Η τεχνική προγραμματισμού ενός παράλληλου συστήματος εξαρτάται άμεσα από την αρχιτεκτονική του, διότι αυτή χαρακτηρίζει τον τρόπο προσπέλασης των δεδομένων από τις διεργασίες που εκτελούνται μια χρονική στιγμή στο σύστημα. Από εδώ και στο εξής, χωρίς περιορισμό της γενικότητας, θα αναφερόμαστε στον όρο «επεξεργαστική μονάδα» ως «επεξεργαστής».

1.2.1 Αρχιτεκτονική Κοινόχρηστης Μνήμης

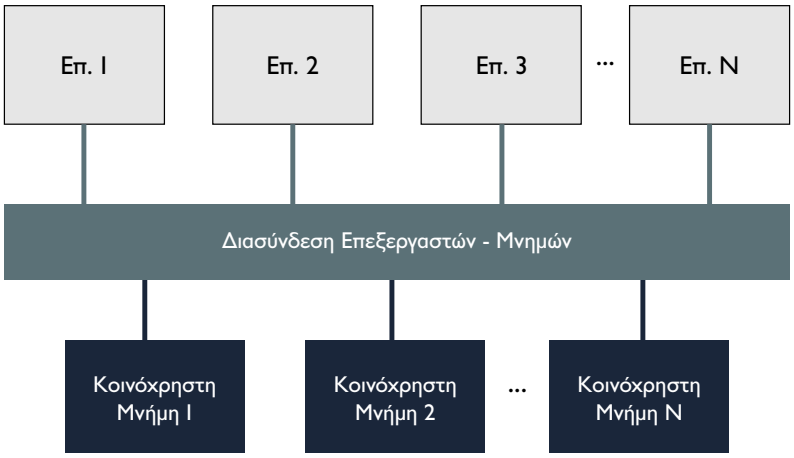
Τα συστήματα κοινόχρηστης μνήμης τυπικά αποτελούνται από μνήμες, οι οποίες οργανώνονται έτσι ώστε ο συνολικός χώρος διευθύνσεων να είναι ενιαίος και κοινόχρηστος. Ο χώρος διευθύνσεων είναι άμεσα προσπελάσιμος και κοινός για όλους τους επεξεργαστές του συστήματος 1.1. Με την ανάγνωση και την τροποποίηση των δεδομένων στη κοινόχρηστη μνήμη, επιτυγχάνεται η επικοινωνία των διάφορων επεξεργαστών του συστήματος και, εν τέλει, ο συγχρονισμός τους. Η πρόσβαση των

επεξεργαστών στη κοινόχρηστη μνήμη γίνεται με τη χρήση υλικού που ονομάζεται ως *μέσο διασύνδεσης*. Στα παράλληλα υπολογιστικά συστήματα μικρής κλίμακας, ως μέσο διασύνδεσης συνήθως χρησιμοποιείται ένας δίαυλος (bus).

Ενώ η αρχιτεκτονική αυτή μοιάζει αρκετά απλή και ιδανική, στην πραγματικότητα είναι αποτελεσματική μόνο για μικρό πλήθος επεξεργαστών. Η αύξηση του πλήθους τους συνεπάγεται μείωση των επιδόσεων, λόγω της φύσης του διαύλου. Συγκεκριμένα, ο δίαυλος δεν υποστηρίζει την παράλληλη χρήση του από τους επεξεργαστές, συνεπώς οι επικοινωνίες πραγματοποιούνται σειριακά. Επιπλέον, για την υποστήριξη μεγάλου πλήθος επεξεργαστών, απαιτείται ένα μεγάλο πλήθος καλωδίων για τη διασύνδεσή τους. Η αύξηση στο μέγεθος της καλωδίωσης δυνητικά προσφέρει ταχύτητα, συνεπάγεται όμως δυσκολία στη κατασκευή και για τον λόγο αυτό δεν προτιμάται.

Ο προγραμματισμός των συστημάτων κοινόχρηστης μνήμης βασίζεται στην χρήση οντοτήτων εκτέλεσης (διεργασιών ή νημάτων), τα οποία έχουν πρόσβαση σε ένα κοινόχρηστο χώρο διευθύνσεων. Οι οντότητες εκτέλεσης διαμοιράζονται ομοιόμορφα στους επεξεργαστές του συστήματος, ώστε να επιτευχθεί η παράλληλη εκτέλεση. Στην προκειμένη περίπτωση, τα νήματα είναι η πιο δημοφιλής επιλογή, διότι αφενός έχουν πολύ χαμηλότερες απαιτήσεις σε χρόνο δημιουργίας και σε χώρο, αφετέρου διαθέτουν εγγενώς κοινόχρηστες μεταβλητές. Αντιθέτως, οι διεργασίες διαθέτουν ιδιωτικό χώρο διευθύνσεων και συνεπώς απαιτούνται περαιτέρω κλήσεις συστήματος για την μεταξύ τους επικοινωνία, γεγονός που δημιουργεί επιπλέον ανεπιθύμητο προγραμματιστικό φόρτο. Επιπλέον, ονομάζουμε τα νήματα ως "ελαφρές" οντότητες, διότι κληρονομούν μόνο τον μετρητή προγράμματος και τους πόρους της διεργασίας-γονέα. Παρ' όλα αυτά, ο συντονισμός τους είναι μια αρκετά δύσκολη και χειροκίνητη διαδικασία. Υπάρχουν πολλές βιβλιοθήκες προγραμματισμού νημάτων που παρέχουν τις απαιτούμενες λειτουργικότητες, με πιο διαδεδομένη τη βιβλιοθήκη των POSIX threads.

Τη λύση στο πρόβλημα του χειρισμού νημάτων δίνουν τα πρότυπα παράλληλου προγραμματισμού. Ένα παράδειγμα προγραμματιστικού μοντέλου αποτελεί το πρότυπο OpenMP [1], το οποίο αφορά τα συστήματα κοινόχρηστης μνήμης. Στόχος του OpenMP είναι η παραγωγή ομοιόμορφου πολυνηματικού κώδικα, με τη χρήση συγκεκριμένων οδηγιών. Ο προγραμματιστής τροποποιεί το ήδη υπάρχον σειριακό πρόγραμμα και σημαίνει ορισμένες περιοχές του κώδικα προς παραλληλοποίηση, παραθέτοντας ορισμένες εντολές προς τον προεπεξεργαστή (preprocessor) του με-



Σχήμα 1.1: Το μοντέλο της κοινόχρηστης μνήμης

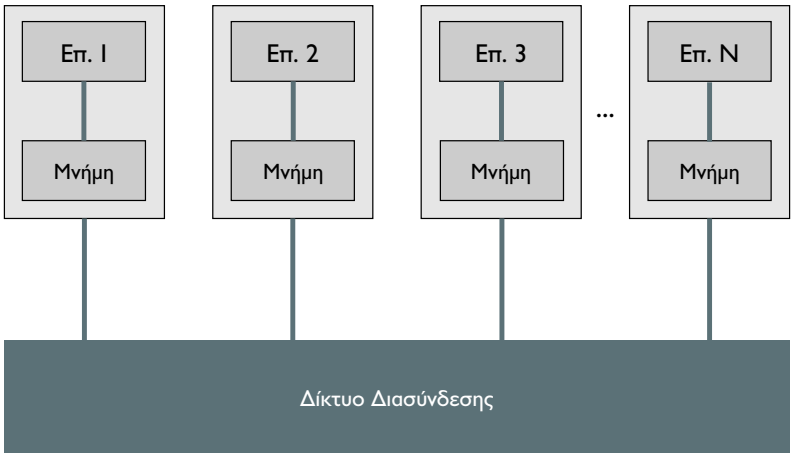
ταφραστή. Εκτενής αναφορά στις λειτουργίες και τις δομές του OpenMP, γίνεται στο κεφάλαιο 2.

1.2.2 Αρχιτεκτονική Κατανεμημένης Μνήμης

Στα συστήματα με αρχιτεκτονική κατανεμημένης μνήμης, κάθε επεξεργαστής διαθέτει μια ιδιωτική μνήμη. Εξ' ορισμού, ο κάθε επεξεργαστής έχει άμεση πρόσβαση στην ιδιωτική του μνήμη. Επιπλέον, για την επικοινωνία των επεξεργαστών χρησιμοποιείται ένα δίκτυο διασύνδεσης. Το δίκτυο αυτό επιτρέπει σε κάθε επεξεργαστή να έχει έμμεση πρόσβαση και στα δεδομένα της ιδιωτικής μνήμης όλων των υπόλοιπων. Για παράδειγμα, έστω ότι ο επεξεργαστής 1 (Επ. 1) διαθέτει μια μεταβλητή X στην ιδιωτική του μνήμη, την οποία θέλει να προσπελάσει ο επεξεργαστής 2 (Επ. 2). Ο Επ. 2 θα πρέπει να αποστείλει ένα μήνυμα μέσω του δικτύου διασύνδεσης, το οποίο θα λάβει ο Επ. 1 και, με την σειρά του, θα αποστείλει μια απάντηση που περιέχει την τιμή της μεταβλητής X.

Όλες οι προσπελάσεις που συμβαίνουν στο δίκτυο διασύνδεσης ενός συστήματος κατανεμημένης μνήμης, είναι ικανές να εισάγουν σημαντικό ανεπιθύμητο φόρτο κατά την εκτέλεση του προγράμματος, ειδικότερα στην περίπτωση όπου ένας επεξεργαστής προσπελάζει διαρκώς δεδομένα που ανήκουν σε άλλους επεξεργαστές. Συνολικά, ο φόρτος αυτός επηρεάζει άμεσα την απόδοση ενός κατανεμημένου συστήματος, η οποία και μειώνεται δραστικά σε τέτοιου είδους περιπτώσεις.

Για τον προγραμματισμό ενός συστήματος κατανεμημένης μνήμης, χρησιμοποιείται ευρέως το μοντέλο MPI [2]. Το MPI ορίζει μια διεπαφή, η οποία επιτρέπει σε επεξεργαστές που συνδέονται με ένα τέτοιο δίκτυο διασύνδεσης, να επικοινωνούν



Σχήμα 1.2: Το μοντέλο της κατανεμημένης μνήμης

μεταξύ τους μέσω μηνυμάτων. Μέσω των επικοινωνιών, οι επεξεργαστές ανταλλάσσουν δεδομένα, ώστε να επιτευχθεί ορθός υπολογισμός των αποτελεσμάτων του προγράμματος. Η διεπαφή του MPI διαθέτει ένα σύνολο κλήσεων τύπου send/receive, για την αποστολή και παραλαβή δεδομένων από/προς άλλους επεξεργαστές. Εκτός από την επικοινωνία δύο επεξεργαστών προσφέρονται και πιο σύνθετες επικοινωνίες, οι λεγόμενες συλλογικές, όπου συμμετέχουν πολλοί επεγεργαστές. Για παράδειγμα, η λειτουργία της εκπομπής (broadcasting) επιτρέπει σε μια διεργασία ενός επεξεργαστή να αποστείλει το ίδιο μήνυμα σε όλες τις υπόλοιπες διεργασίες άλλων επεξεργαστών.

1.3 Επιταχυντές και GPUs

Οι επιστημονικοί υπολογισμοί από την φύση τους χαρακτηρίζονται από ποικιλομορφία. Διαφορετικά προβλήματα απαιτούν διαφορετικά είδη πράξεων, σε διαφορετικούς τύπους δεδομένων. Οι επεξεργαστές των σημερινών παράλληλων υπολογιστικών συστημάτων, παρότι μπορούν χρησιμοποιηθούν δυνητικά για την επίλυση όλων των υπολογιστικών προβλημάτων, συχνά δεν είναι όσο αποδοτικοί θα επιθυμούσαμε, σε συγκεκριμένα είδη υπολογισμών. Ο λόγος αυτός στάθηκε αφορμή για την ανάπτυξη συνοδών υπολογιστικών συσκευών, οι οποίες φιλοξενούνται σε ένα σύστημα και χρησιμοποιούνται για την επιτάχυνση ορισμένων τύπων υπολογισμών. Από εδώ και στο εξής, θα αναφερόμαστε στο σύστημα που φιλοξενεί τις συσκευές, ως *host*.

Οι συνοδές αυτές συσκευές υλοποιούνται σε ξεχωριστό υλικό και είναι άμεσα προσπελάσιμες από την κεντρική μονάδα επεξεργασίας, συνήθως μέσω διαύλων υψηλής χωρητικότητας και ταχύτητας. Στην γενικότερη περίπτωσ, ονομάζονται επιταχυντές (accelerators). Ειδικότερα, ένα υποσύνολο των επιταχυντών είναι οι συνεπεξεργαστές (coprocessors), οι οποίοι λειτουργούν συμπληρωματικά με έναν κεντρικό επεξεργαστή, με την έννοια ότι ένα μέρος των εντολών του επεξεργαστή μοιράζεται σε αυτούς.

Το πιο διαδεδομένο είδος επιταχυντή είναι οι μονάδες γραφικής επεξεργασίας (GPU). Οι μονάδες γραφικής επεξεργασίας είναι αυτόνομες συσκευές οι οποίες διαθέτουν συνήθως μεγάλο πλήθος σχετικά ανίσχυρων πυρήνων, οι οποίοι όμως είναι ταχύτατοι στην εκτέλεση ορισμένων απλών πράξεων. Παραδοσιακά, οι πυρήνες αυτοί χρησιμοποιούταν για την παραγωγή γραφικών και την ανανέωσή τους στην οθόνη. Με το πέρασμα του χρόνου, κατασκευαστές και προγραμματιστές συνειδητοποίησαν ότι είναι εφικτή η διάθεση των πυρήνων αυτών για υπολογισμούς γενικής φύσεως. Οι μονάδες γραφικής επεξεργασίας έγιναν γενικότερου σκοπού, με την έννοια ότι πλέον μπορούν να χρησιμοποιηθούν για επιτάχυνση μεγαλύτερου εύρους υπολογισμών. Η οργάνωση και το πλήθος των μικροπυρήνων μιας μονάδας γραφικής επεξεργασίας, τους καθιστά ιδανικούς για την εκτέλεση πράξεων σε δεδομένα που βασίζονται 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) [3] και η OpenCL (Open Computing Language) [4]. 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 μετα-

φορά δεδομένων από/προς αυτή, την εκτέλεση του υπολογιστικού πυρήνα (kernel) καθώς επίσης και τον τερματισμό της.

Η διαδικασία ανάπτυξης εφαρμογών αυτού του τύπου χαρακτηρίζεται από ετερογένεια, με την έννοια ότι απαιτεί την ξεχωριστή συγγραφή του κώδικα που θα προοριστεί στους πυρήνες γενικού σκοπού της μονάδας GPU και του προγράμματος που θα εκτελεστεί στον επεξεργαστή. Επιπλέον, απαιτεί ένα συγκεκριμένο επίπεδο ικανότητας από τον προγραμματιστή, διότι ο χειρισμός των δεδομένων και των υπολογισμών πρέπει να γίνεται ιδιαίτερα προσεκτικά. Έχοντας επίγνωση των δύο αυτών χαρακτηριστικών, παρουσιάζεται η ευκαιρία στόχευσης τους μέσα από το πρότυπο OpenMP. Το OpenMP από την φύση του προσφέρει ένα φορητό και κλιμακώσιμο μοντέλο για παράλληλο προγραμματισμό, με ευέλικτες διεπαφές προς τον προγραμματιστή και ομοιομορφία στον τελικό πηγαίο κώδικα. Επιπλέον διαθέτει ένα σύνολο οδηγιών που διευκολύνουν την εκτέλεση πολυνηματικού κώδικα σε επιταχυντές γραφικών.

Ξεκινώντας από την έκδοση 4.5 του προτύπου OpenMP, ένα πρόγραμμα χρήστη μπορεί να περιλαμβάνει τμήματα κώδικα τα οποία μπορούν να προοριστούν για εκτέλεση στις διάφορες συνοδές συσκευές (devices) ενός συστήματος. Η λειτουργία αυτή καθιστά δυνατή την χρήση των μονάδων GPU για επιτάχυνση ορισμένων υπολογισμών. Το πρότυπο υποστηρίζει τόσο την μεταφορά δεδομένων και αποτελεσμάτων προς/από μία συσκευή, όσο και την μεταφορά των εκτελέσιμων προγραμμάτων που θα παράξουν τα αποτελέσματα. Συνεπώς, ο φόρτος χειρισμού της συσκευής από τον προγραμματιστή, έχει τη δυνατότητα να αποκρυφθεί και να μεταφερθεί σε μία συσκευή βιβλιοθήκης, επιτρέποντας έτσι στον προγραμματιστή να επικεντρωθεί μόνο στον ωφέλιμο κώδικα των υπολογιστικών πυρήνων.

Η προτεινόμενη ιδέα έχει ήδη υλοποιηθεί από κάποιους δημοφιλείς μεταφραστές, όπως ο gcc-7 [5] και ο LLVM-11.0 [6]. Παρ' όλα αυτά, η λειτουργικότητα που υπολείπεται στους μεταφραστές αυτούς αφορά το μοντέλο CUDA και συγκεκριμένα την αυτόματη παραγωγή κώδικα που κάνει χρήση της διεπαφής CUDA, από τον κώδικα χρήστη. Το γεγονός αυτό στάθηκε η αφορμή για την ανάπτυξη της ιδέας της παρούσας εργασίας.

1.4 Αντικείμενο της Εργασίας

Το βασικό αντικείμενο της εργασίας είναι η υποστήριξη παράλληλης εκτέλεσης κώδικα χρήστη σε μονάδες γραφικής επεξεργασίας που υποστηρίζουν το μοντέλο CUDA, μέσα από το προγραμματιστικό πρότυπο OpenMP. Οι μονάδες GPU αυτής της κλάσης, αποτέλεσαν το ενδιαφέρον της παρούσας εργασίας διότι υποστηρίζουν εγγενώς την εκτέλεση γενικών υπολογισμών στους πυρήνες τους. Ως βάση της παρούσας εργασίας χρησιμοποιήθηκε ο ακαδημαϊκού σκοπού μεταφραστής OMPi [7], ο οποίος προσφέρει μια ελαφριά υποδομή για την υποστήριξη του προτύπου OpenMP. Ο OMPi βασίζεται στον μετασχηματισμό ενός πηγαίου κώδικα γλώσσας προγραμματισμού C σε έναν βελτιστοποιημένο πολυνηματικό κώδικα C. Κύριο συστατικό του OMPi είναι βιβλιοθήκες συσκευών, οι οποίες παρέχουν όλες τις απαραίτητες συναρτήσεις για την μεταφορά πηγαίου κώδικα και δεδομένων προς/από τις συνοδές συσκευές ενός συστήματος. Συνεπώς, για την στόχευση των επιταχυντών γραφικών βασισμένων σε CUDA, μέσω του OpenMP, η υποδομή του OMPi έπρεπε να τροποποιηθεί ώστε να υποστηρίζει την νέα βιβλιοθήκη που θα αφορά το συγκεκριμένο είδος επιταχυντών. Αναλυτικότερα, για τους σκοπούς της εργασίας:

1. Εμπλουτίστηκαν ορισμένες διαδικασίες του OMPi, που αφορούν την συντακτική ανάλυση του πηγαίου κώδικα, για την διευκόλυνση των προσπαθειών της εργασίας, αλλά και οποιωνδήποτε μελλοντικών προσπαθειών

2. Υλοποιήθηκε μια σειρά από οδηγίες του OpenMP, στον μεταφραστή OMPi, οι οποίες στοχεύουν επιταχυντές γραφικών

3. Δημιουργήθηκε μια νέα βιβλιοθήκη συσκευής του OMPi, η CUDADEV, η οποία καθιστά δυνατή τη φόρτωση κώδικα προς εκτέλεση μέσω του OpenMP

1.5 Σχετική Εργασία

Υποστήριξη εκτέλεσης κώδικα σε μονάδες γραφικής επεξεργασίας, υφίσταται και εκτός του προτύπου OpenMP, χρησιμοποιώντας είτε διαφορετικά πρότυπα όπως το OpenACC [8], είτε πειραματικές προγραμματιστικές διεπαφές που δημιουργήθηκαν αποκλειστικά για τον σκοπό αυτό.

Μια προσπάθεια πειραματικού προγραμματιστικού πλαισίου είναι το OMPCUDA [9], το οποίο βασίζεται στον Omni OpenMP Compiler [10]. Η εργασία αυτή των

Satoshi Ohshima και Shoichi Hirasawa επικεντρώθηκε κυρίως στην παραλληλοποίηση επαναληπτικών βρόχων (for loops) και την μεταφορά μεταβλητών προς και από την συσκευή, χρησιμοποιώντας την υποδομή του Omni.

Άλλες προσπάθειες όπως η [11], των Seyong Lee, Seung-Jai Min και Rudolf Eigenmann, προτείνουν ένα προγραμματιστικό πλαίσιο για μεταφραστές, το οποίο μετασχηματίζει αυτόματα οδηγίες παραλληλοποίησης βρόχου *parallel for* του προτύπου OpenMOP, σε κώδικα CUDA. Οι συγγραφείς χρησιμοποίησαν τις εφαρμογές JACOBI και SPMUL, καθώς επίσης και τα μετροπρογράμματα NAS OpenMP Parallel Benchmarks [12], για την εφαρμογή του προτεινόμενου προγραμματιστικού πλαισίου. Παρομοίως, η εργασία [13] των Jaillet και Krajecki, στοχεύει επίσης στην μετάφραση της οδηγίας *parallel for* σε κώδικα CUDA, επεκτείνοντας τον μεταφραστή OMPi. Ο παραγόμενος κώδικας και στις δύο εργασίες, είναι έτοιμος προς μετάφραση από τον *nvcc*, τον σχετικό μεταγλωττιστή της NVIDIA.

Η καινοτομία της παρούσας εργασίας, συγκριτικά με τις προαναφερθείσες, είναι ότι αφορά τις περιοχές target, οι οποίες εισήχθησαν στην έκδοση 4.5 του προτύπου OpenMP και όχι την επέκταση της ήδη υπάρχουσας οδηγίας *parallel for* για την φόρτωση κώδικα στη συσκευή. Επιπλέον, η εργασία δεν περιορίζεται μόνο στη παραλληλοποίηση βρόχων, μιας και δεν έχει καμία γνώση για την υποκείμενη εφαρμογή που πρόκειται να μεταφραστεί. Ο κώδικας του χρήστη που προορίζεται για εκτέλεση στη συσκευή CUDA, παρ' όλο που πρέπει να ακολουθεί κάποιες συμβάσεις του προτύπου OpenMP, είναι άγνωστος και, συνεπώς, ο μετασχηματισμός του πρέπει να είναι γενικεύσιμος.

1.6 Δομή της Εργασίας

Η δομή της παρούσας εργασίας είναι η εξής:

- Κεφάλαιο 1: Εισαγωγή στις αρχιτεκτονικές παράλληλων συστημάτων και περιγραφή σχετικών εργασιών
- Κεφάλαιο 2: Περιγραφή του προτύπου OpenMP, σχετικών οδηγιών του, καθώς και του όρου των Συσκευών
- Κεφάλαιο 3: Ανάλυση της δομής των συσκευών γραφικής επεξεργασίας και συγκεκριμένα όσων βασίζονται στο μοντέλο CUDA

- Κεφάλαιο 4: Περιγραφή της δομής του μεταφραστή OMPi και λεπτομερής ανάλυση της υποδομής λειτουργικότητας για τις συσκευές
- Κεφάλαιο 5: Παρουσίαση της νέας συσκευής CUDADEV που υλοποιήθηκε στον μεταφραστή OMPi
- Κεφάλαιο 6: Παράθεση των αποτελεσμάτων όλων των πειραμάτων και αξιολόγηση των επιδόσεων της προτεινόμενης υποδομής
- Κεφάλαιο 7: Σύνοψη της εργασίας και αναφορά σε μελλοντικές βελτιστοποιήσεις και συμπληρωματικές υποδομές του μεταφραστή OMPi

Κεφαλαιο 2

OpenMP και Συσκευες

2.1 Εισαγωγή

Με τη ραγδαία ανάπτυξη των παράλληλων υπολογιστικών συστημάτων, προέκυψαν νέες απαιτήσεις που αφορούν τις τεχνικές προγραμματισμού μιας πολυνηματικής εφαρμογής. Ειδικότερα, στα συστήματα κοινόχρηστης μνήμης, με την αύξηση του αριθμού των επεξεργαστών, δημιουργήθηκαν προβλήματα όπως οι καθυστερήσεις στην προσπέλαση μνήμης. Για να αποφύγει τέτοιας φύσεως ζητήματα, ο προγραμματιστής καλείται να χρησιμοποιήσει πιο εξειδικευμένες μεθόδους, για τον συντονισμό των διαφόρων οντοτήτων και προσπελάσεων δεδομένων μέσα στο πρόγραμμα.

Τη λύση στη δυσκολία προγραμματισμού με νήματα, δίνει το προγραμματιστικό πρότυπο OpenMP, με τον ορισμό μιας διεπαφής η οποία αυτοματοποιεί τις περισσότερες διαδικασίες και τεχνικές παράλληλου προγραμματισμού, που αρχικά εφαρμόζονταν χειροκίνητα από τον προγραμματιστή. Ο χειρισμός των προσπελάσεων στα δεδομένα, είναι μία διαδικασία η οποία αφενός αφήνεται στον προγραμ-

ματιστή, αφετέρου πραγματοποιείται σαφώς πιο οργανωμένα χάρη στις βοηθητικές λειτουργίες του OpenMP. Συγκεκριμένα, το OpenMP αποτελείται από τρία κύρια συστατικά, τα οποία εκτίθονται άμεσα στον εκάστοτε προγραμματιστή:

- Ένα σύνολο οδηγιών προς τον μεταφραστή, τα λεγόμενα *pragmas ή directives*, τα οποία ορίζουν τον τρόπο με τον οποίο ο μεταφραστής θα χειριστεί ένα τμήμα κώδικα, ή ένα σύνολο δεδομένων.

- Μια συλλογή ρουτινών βιβλιοθήκης, τις οποίες μπορεί να χρησιμοποιήσει ο προγραμματιστής για να καθορίσει διάφορες παραμέτρους εκτέλεσης, να διευκολύνει τον συγχρονισμό των νημάτων ή να κάνει χρήση άλλων εργαλείων όπως η χρονομέτρηση των επιδόσεων. Ένα σημαντικό παράδειγμα παραμέτρου εκτέλεσης είναι το πλήθος των νημάτων που πρόκειται να εκτελέσει ένα τμήμα κώδικα, ενώ ένα παράδειγμα συγχρονισμού τους είναι η χρήση *κλειδαριών* (locks) με κλήσεις προς τις αντίστοιχες ρουτίνες.

- Ένα πλήθος μεταβλητών περιβάλλοντος, οι οποίες καθορίζουν εκ των προτέρων ορισμένες παραμέτρους εκτέλεσης.

Απώτερος σκοπός του OpenMP είναι η κλιμακωτή συγγραφή παράλληλων προγραμμάτων, μέσω της παραλληλοποίησης ορισμένων τμημάτων κώδικα μιας σειριακής εφαρμογής, χωρίς να τροποποιηθεί επ' ουδενί η λογική της. Επιπλέον, η φύση του OpenMP βασίζεται στην έννοια της *φορητότητας* (portability), μιας και η λογική χειρισμού των νημάτων και των δεδομένων που προσπελάζουν, καλείται να υλοποιηθεί από τον εκάστοτε μεταφραστή και όχι από τον ίδιο τον προγραμματιστή.

Το πρότυπο OpenMP είναι το πιο δημοφιλές για τον προγραμματισμό παράλληλων εφαρμογών. Παρ' όλα αυτά, ενώ διευκολύνει τον προγραμματιστή κρύβοντας λεπτομέρειες για τα νήματα, δεν του επιτρέπει να εφαρμόσει ορισμένες τεχνικές, οι οποίες θα μπορούσαν να εφαρμοστούν αν ο ίδιος χειριζόταν τα νήματα. Παρ'όλα αυτα, όπως θα δούμε παρακάτω, το OpenMP πρσφέρει πάρα πολλές και ισχυρές προγραμματιστικές δομές οι οποίες διευκολύνουν τον προγραμματιστή στη συγγραφή του παράλληλου προγράμματος και στην άντληση επιδόσεων από το παράλληλο σύστημα που θα εκτελέσει την εφαρμογή.

2.2 Προγραμματιστικό Μοντέλο

Το OpenMP κάνει χρήση του λεγόμενου μοντέλου προγραμματισμού *fork-join*. Ειδικότερα, το πρόγραμμα εκτελείται αρχικά από το *κύριο νήμα* (master thread), το οποίο είναι και μοναδικό. Έπειτα, όταν συναντηθεί μια *περιοχή παραλληλίας* (parallel region), θα δημιουργηθεί (fork) μία ομάδα νημάτων, στην οποία θα συμμετέχει και το κύριο νήμα. Η ομάδα νημάτων θα εκτελέσει παράλληλα το τμήμα κώδικα που περικλείεται από την οδηγία παραλληλοποίησης. Με το πέρας της περιοχής παραλληλίας, η ομάδα νημάτων θα καταστραφεί (join) και η εκτέλεση του υπόλοιπου θα συνεχιστεί ξανά από το κύριο νήμα. Η διαδικασία αυτή θα επαναληφθεί για κάθε περιοχή παραλληλίας του προγράμματος.

2.3 Οδηγίες OpenMP για C/C++

Στην παρούσα υποενότητα γίνεται μια αναφορά στις σημαντικότερες λειτουργίες OpenMP που προσφέρονται για τις γλώσσες προγραμματισμού C και C++. Αρχικά, περιγράφεται ο πλήρης τρόπος σύνταξης μιας οδηγίας OpenMP. Έπειτα, περιγράφεται λεπτομερώς η δημιουργία μιας *περιοχής παραλληλίας* (parallel region), με την αντίστοιχη οδηγία της. Μιας και αναπόσπαστο κομμάτι της παραλληλοποίησης είναι ο *διαμοιρασμός εργασίας* (worksharing) σε μια ομάδα νημάτων, εξηγούνται οι διάφορες οδηγίες που αφορούν την τεχνική αυτή. Τέλος, παρατίθονται οι οδηγίες *συγχρονισμού* (synchronization) των νημάτων, καθώς και ένα σημαντικό υποσύνολο των *φράσεων* (clauses) που παραμετροποιούν την εκτέλεση περιοχών OpenMP.

2.3.1 Σύνταξη

Οι οδηγίες του OpenMP για τις γλώσσες προγραμματισμού C/C++ καθορίζονται με την χρήση της οδηγίας προεπεξεργαστή `pragma`. Η σύνταξη μιας οδηγίας OpenMP είναι η εξής:

#pragma omp *όνομα οδηγίας* [*φράση*[[,] *φράση*] ...] *νέα-γραμμή*

Ισχύουν οι παρακάτω κανόνες όσον αφορά την σύνταξη της οδηγίας:

- Το όνομα της οδηγίας είναι υποχρεωτικό μετά από την χρήση ενός **#pragma omp**.

- Τα **#pragma omp** κάνουν διάκριση πεζών-κεφαλαιών, όσον αφορά το όνομα της οδηγίας.

- Μετά την οδηγία τοποθετούνται, προαιρετικά, οι φράσεις οδηγιών (χωρίς περιορισμό στην σειρά δήλωσής τους), οποίες ορίζουν τις συνθήκες υπό τις οποίες θα εκτελεστεί η οδηγία. Εάν δεν έχει τοποθετηθεί καμία οδηγία, τότε οι συνθήκες αυτές καθορίζονται στον χρόνο εκτέλεσης του προγράμματος.

- Η σύνταξη μιας οδηγίας OpenMP απαιτεί την ύπαρξη νέας γραμμής (new-line) στο τέλος της.

2.3.2 Περιοχές Παραλληλίας (Parallel Regions)

Ως περιοχή παραλληλίας ονομάζεται το τμήμα κώδικα που πρόκειται να εκτελεστεί παράλληλα από μία ομάδα νημάτων. Η δήλωση της περιοχής γίνεται μέσω της οδηγίας `parallel`:

#pragma omp *parallel* [*φράση*[[,] *φράση*] ...] *νέα-γραμμή*
δομημένο τμήμα

Ως δομημένο τμήμα ονομάζουμε το τμήμα κώδικα που πρόκειται να παραλληλοποιηθεί, το οποίο είτε περικλείεται σε άγκιστρα, είτε αποτελείται από μία μοναδική εντολή, χωρίς άγκιστρα. Όταν, κατά την εκτέλεση, το κύριο νήμα συναντήσει μια περιοχή παραλληλίας, τότε δημιουργεί μία ομάδα νημάτων, στην οποία, ως γονέας, συμμετέχει και το ίδιο. Η ομάδα νημάτων εκτελεί το τμήμα κώδικα και κάθε νήμα τερματίζει την εκτέλεση σε χρόνο διαφορετικό από τα υπόλοιπα, λόγω φύσεως του υλικού του επεξεργαστή. Συνεπώς, στο τέλος κάθε περιοχής παραλληλίας, το OpenMP υπονοεί την χρήση ενός *φράγματος* (barrier), το οποίο συγχρονίζει τα νήματα, αναγκάζοντας τα να αναμένουν έως ότου η εκτέλεση τερματιστεί από όλα τα νήματα της ομάδας. Κάτα την ολοκλήρωση της εκτέλεσης, το φράγμα προσπερνάται και η ομάδα καταστρέφεται, ενώ το κύριο νήμα συνεχίζει κανονικά την εκτέλεση του υπόλοιπου προγράμματος.

Ο χρήστης, όπως προαναφέρθηκε, έχει την δυνατότητα να ορίσει το μέγεθος του τμήματος της ομάδας νημάτων που θα εκτελέσει την περιοχή παραλληλίας. Οι βασικές (ισοδύναμες) μέθοδοι που χρησιμοποιούνται για τον καθορισμό του μεγέθους της ομάδας είναι οι εξής:

- Κλήση της συνάρτησης `omp_set_num_threads()` εκτός της περιοχής παραλληλίας.

```c
#include <stdio.h>
#include "omp.h"
int main()
{
    omp_set_num_threads(16);
    #pragma omp parallel
    {
        int thrid = omp_get_thread_num();
        printf("I am thread %d.\n", thrid);
    }
}
```

Σχήμα 2.1: Παράδειγμα περιοχής παραλληλίας OpenMP

Η `omp_set_num_threads()` δέχεται ως όρισμα το πλήθος των νημάτων που θα απαρτίσουν την ομάδα που πρόκειται να δημιουργηθεί.

- Εκχώρηση τιμής στην μεταβλητή περιβάλλοντος `OMP_NUM_THREADS`, πριν την εκτέλεση του παράλληλου προγράμματος.
- Τοποθέτηση της φράσης `num_threads()` μετά από την οδηγία parallel με όρισμα το πλήθος των επιθυμητών νημάτων.

Στο παράδειγμα 2.1, το κύριο νήμα πρώτα θα εκτελέσει την εντολή ρουτίνας βιβλιοθήκης `omp_set_num_threads()` με όρισμα `16`, συνεπώς σε κάθε περιοχή παραλληλίας από το σημείο αυτό και έπειτα, θα δημιουργείται μια ομάδα 16 νημάτων. Έπειτα, το κύριο νήμα συναντά την οδηγία `parallel` και δημιουργείται η παράλληλη ομάδα. Κάθε νήμα θα εκτελέσει τον κώδικα και θα βρει το αναγνωριστικό του, το οποίο θα είναι μια τιμή που ανήκει στο διάστημα `[0, 15]`. Τέλος, κάθε νήμα θα τυπώσει το αναγνωριστικό του. Αξίζει να σημειωθεί ότι η σειρά των μηνυμάτων που θα τυπωθούν, διαφέρει ανά εκτέλεση του προγράμματος.

2.3.3 Διαμοιρασμός Εργασίας (Worksharing Regions)

Ο διαμοιρασμός της εργασίας στα νήματα μίας ομάδας είναι εξίσου σημαντική λειτουργία του OpenMP. Ο κώδικας μιας περιοχής παραλληλίας απαιτεί αρκετές τροποποιήσεις ώστε να μπορεί να διαμοιραστεί, για παράδειγμα, ο φόρτος ενός

επαναληπτικού βρόχου στο εσωτερικό της. Για τον σκοπό αυτό, το OpenMP εισήγαγε τις περιοχές διαμοιρασμού εργασίας, οι οποίες δίνουν στον προγραμματιστή τη δυνατότητα, μέσω ενός συνόλου οδηγιών, να διαμοιράσει αποτελεσματικά τον φόρτο υπολογισμού σε μια παράλληλη ομάδα. Οι περιοχές διαμοιρασμού εργασίας μπορούν να χρησιμοποιηθούν και εκτός περιοχών παραλληλίας, αλλά αποκτούν νόημα όταν βρίσκονται στο εσωτερικό αυτών.

Στο τέλος των περιοχών διαμοιρασμού εργασίας υπονοείται ένα φράγμα, 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. Υπάρχουν τρεις τύποι περιοχών διαμοιρασμού εργασίας και ορίζονται, αντίστοιχα, με τις εξής οδηγίες:

- **Οδηγία for**. Αφορά τον καταμερισμό του φόρτου ενός επαναληπτικού βρόχου for. Αμέσως μετά την δήλωση της συγκεκριμένης οδηγίας, ακολουθεί ο βρόχος που πρόκειται να παραλληλοποιηθεί. Ο τρόπος με τον οποίο διαμοιράζεται ο φόρτος του βρόχου περιγράφεται αναλυτικά στην συνέχεια του κεφαλαίου. Ο τρόπος σύνταξης της οδηγίας είναι ο εξής:

  #pragma omp *for* [*φράση*[[,] *φράση*] ...] *νέα-γραμμή*

  `for` (*αρχικοποίηση*; *συνθήκη*; *βήμα*) {

  ...

  }

- **Οδηγία sections**. Η συγκεκριμένη οδηγία χρησιμοποιείται για την παραλληλοποίηση εργασιών οι οποίες δεν σχετίζονται μεταξύ τους, από τα νήματα εντός μιας περιοχής παραλληλίας. Στο εσωτερικό της οδηγίας τοποθετούνται διαδοχικές δηλώσεις της οδηγίας `#pragma omp section`, συνοδευόμενες από δομημένα τμήματα κώδικα. Η δομή της δήλωσης της οδηγίας είναι η εξής:

  #pragma omp *sections* [*φράση*[[,] *φράση*] ...] *νέα-γραμμή*

  {

  #pragma omp *section*

  δομημένο τμήμα

  ...

  }

- **Οδηγία single**. Η οδηγία αυτή επιτρέπει σε ένα δομημένο τμήμα να εκτελεστεί μόνο από το νήμα το οποίο την συναντά πρώτο. Τα υπόλοιπα νήματα, με τη σειρά τους, θα προσπεράσουν την οδηγία. Η συμπεριφορά των νημάτων, από προεπιλογή, είναι να αναμένουν έως ότου το πρώτο ολοκληρώσει την εκτέλεση του κώδικα, για λόγους συγχρονισμού. Παραλλαγή της οδηγίας `single` αποτελεί η οδηγία `master`, η οποία απαιτεί η εκτέλεση του δομημένου τμήματος κώδικα να γίνει από το κύριο νήμα. Επιπλέον, στην περίπτωση αυτής της οδηγίας δεν υπονοείται φράγμα συγχρονισμού των νημάτων. Η σύνταξη των δύο προαναφερθέντων οδηγιών είναι η εξής:

  #pragma omp *single* | *master* [*φράση*[[,] *φράση*] ...] *νέα-γραμμή*

  δομημένο τμήμα

2.3.4 Συγχρονισμός Νημάτων

Κρίσιμο για την ορθή λειτουργία ενός παράλληλου προγράμματος είναι και ο συγχρονισμός των νημάτων στο εσωτερικό μιας ομάδας. Για να επιτευχθεί αυτό, το OpenMP διαθέτει κάποιες οδηγίες, οι οποίες εξασφαλίζουν τη συνέπεια κοινόχρηστων δεδομένων. Αυτές οι οδηγίες είναι οι εξής:

- **Οδηγία atomic**. Η οδηγία `atomic` εξασφαλίζει την ατομική εκτέλεση μιας πράξης από τα νήματα, με την έννοια ότι η τροποποίηση στα κοινόχρηστα δεδομένα που επηρεάζονται από την πράξη, θα εκτελεστεί αδιαίρετα και αδιάκοπα, από ένα νήμα τη φορά.

- **Οδηγία barrier**. Η συγκεκριμένη οδηγία ορίζει ένα φράγμα συγχρονισμού εκτέλεσης, το οποίο, όταν συναντάται από τα νήματα της ομάδας, τα αναγκάζει να αναστείλουν την εκτέλεσή τους και να αναμένουν έως ότου καταφθάσουν και όλα τα υπόλοιπα νήματα. Εσωτερικά, το φράγμα είναι μια κλήση σε μία συνάρτηση η οποία θέτει σε αποκλεισμό τα νήματα.

- **Οδηγία critical**. Η οδηγία αυτή ορίζει κάποιες περιοχές κώδικα ως κρίσιμες, με την έννοια ότι αυτές πρέπει να εκτελεστούν από ένα νήμα τη φορά. Τα υπόλοιπα νήματα αναμένουν έως ότου το νήμα που επιλέχθηκε ολοκληρώσει την εκτέλεση της περιοχής, το οποίο είναι και το πρώτο που συνάντησε τη συγκεκριμένη οδηγία. Θεωρείται ότι η συγκεκριμένη οδηγία είναι επέκταση της οδηγίας `atomic`.

- **Οδηγία flush**. Η οδηγία flush χρησιμοποιείται για να συγχρονίσει τα νήματα της ομάδας και να εξασφαλίσει συνέπεια της κοινόχρηστης μνήμης των νημάτων, με την έννοια ότι δεν εκκρεμούν εγγραφές σε αυτήν. Αναφερόμαστε στην διαδικασία αυτή ως write back.

- **Οδηγία ordered**. Η οδηγία ordered χρησιμοποιείται για την σειριοποίηση της εκτέλεσης ενός τμήματος κώδικα εντός μιας παράλληλης περιοχής. Συνήθως τοποθετείται στο εσωτερικό ενός βρόχου επανάληψης, όταν χρειάζεται να τηρηθεί συγκεκριμένη σειρά στην εκτέλεση των επαναλήψεων.

2.3.5 Φράσεις Οδηγιών

Οι φράσεις οδηγιών εισάγονται ακριβώς μετά το όνομα μιας οδηγίας για την παραμετροποίηση του τρόπου εκτέλεσής της. Συνήθως, μεταβάλλουν την τιμή ενός μεγέθους, όπως για παράδειγμα ο αριθμός των νημάτων που θα εκτελέσουν μια περιοχή παραλληλίας, καθώς και ρυθμίζουν τη συμπεριφορά των νημάτων πριν, μετά ή κατά τη διάρκεια της εκτέλεσής τους. Ορισμένες σημαντικές φράσεις του προτύπου OpenMP είναι οι εξής:

- **Φράση if(συνθήκη)**. Όταν τοποθετείται η συγκεκριμένη φράση σε μία οδηγία, τότε αρχικά, ελέγχεται η τιμή αληθείας της συνθήκης. Αν αυτή είναι αληθής, η εκτέλεση της οδηγίας θα πραγματοποιηθεί κανονικά, διαφορετικά ολόκληρη η οδηγία θα αγνοηθεί.

- **Φράση shared(λίστα μεταβλητών)**. Η φράση shared δέχεται μια σειρά από μεταβλητές χωρισμένες με κόμμα, οι οποίες χαρακτηρίζονται ως κοινόχρηστες μεταξύ των νημάτων της ομάδας. Τα νήματα της ομάδας κληρονομούν τις μεταβλητές αυτές από το κύριο νήμα, συνεπώς πρέπει να προϋπάρχει η δήλωσή τους, εκτός της παράλληλης περιοχής. Τα νήματα χρησιμοποιούν μία κοινή διεύθυνση ανά μεταβλητή, για την προσπέλαση ενός δεδομένου.

- **Φράση private(λίστα μεταβλητών)**. Η φράση private χρησιμοποιείται για να ορίσει ως ιδιωτικά τα αντικείμενα της λίστας μεταβλητών, για κάθε νήμα της ομάδας παραλληλίας. Στην πράξη, οι μεταβλητές της λίστας δηλώνονται εκ νέου στην ιδιωτική μνήμη κάθε νήματος και έχουν ισχύ μέχρι το πέρας της περιοχής παραλληλίας.

- **Φράση firstprivate(λίστα μεταβλητών)**. Οι μεταβλητές που χαρακτηρίζονται ως firstprivate γίνονται ιδιωτικές για κάθε νήμα και αρχικοποιούνται στην τιμή που είχε η καθεμία πριν ξεκινήσει η περιοχή παραλληλίας. Πρακτικά, η φράση firstprivate λειτουργεί ακριβώς όπως και η private, με τη μόνη διαφορά να έγκειται στην αρχικοποίηση των μεταβλητών.

- **Φράση lastprivate(λίστα μεταβλητών)**. Η lastprivate λειτουργεί όπως και η private, με την μόνη διαφορά ότι στο τέλος της περιοχής παραλληλίας οι τιμές των αντικείμενα της λίστας μεταβλητών ενημερώνονται.

- **Φράση nowait**. Όταν τοποθετείται η φράση `nowait` στην δήλωση μιας οδηγίας, τότε υπονοείται ότι τα νήματα της ομάδας δεν θα αναμείνουν σε κάποιο φράγμα κατά το τέλος της εκτέλεσης και, συνεπώς, δεν θα συγχρονιστούν.

- **Φράση num_threads(πλήθος νημάτων)**. Η φράση `num_threads` χρησιμοποιείται για να ορίσει το πλήθος των νημάτων που θα εκτελέσουν μια περιοχή παραληλλίας και θα απαρτίσουν μια ομάδα.

- **Φράση schedule(τύπος[, μέγεθος κόκκου])**. Η φράση `schedule`, σε συνδυασμό με την οδηγία `for` ορίζει τον τρόπο με τον οποίο θα διαμοιραστεί ο φόρτος ενός βρόχου επανάληψης, ανάλογα με τον τύπο της χρονοδρομολόγησης. Επιπλέον, προαιρετικά ορίζουμε το μέγεθος του κόκκου παραλληλίας, δηλαδή το πλήθος των επαναλήψεων που θα διαμοιράζεται σε κάθε νήμα τη φορά.

  Ο τύπος μπορεί να είναι ένας εκ των τέσσερις παρακάτω:

  - **static**. Ορίζει στατική χρονοδρομολόγηση για τον επαναληπτικό βρόχο. Ο βρόχος διασπάται σε τμήματα μεγέθους όσου και ο κόκκος παραλληλίας και κάθε τμήμα διαμοιράζεται κυκλικά στα νήματα της ομάδας. Όταν δεν έχει οριστεί μέγεθος κόκκου, τότε ο βρόχος διασπάται σε τόσα τμήματα, όσα και τα νήματα της ομάδας.
  - **dynamic**. Ορίζει δυναμική χρονοδρομολόγηση για τον επαναληπτικό βρόχο, κατά την οποία ο βρόχος διασπάται με τρόπο όμοιο με αυτόν της στατικής χρονοδρομολόγησης, με την μόνη διαφορά ότι ο διαμοιρασμός των τμημάτων γίνεται δυναμικά σε κάθε νήμα. Αν το μέγεθος του κόκκου δεν είναι ορισμένο, τότε το μέγεθος των τμημάτων είναι ίσο με 1.

– **guided**. Στην χρονοδρομολόγηση τύπου `guided`, οι επαναλήψεις διαμοιράζονται κυκλικά στα νήματα, αρχικά σε τμήματα μεγέθους όσο και αυτό του κόκκου παραλληλλίας. Όταν ολοκληρώνεται μια ανάθεση επαναλήψεων σε ένα νήμα, γίνεται εκθετική μείωση στο μέγεθος του επόμενου τμήματος και ανατίθεται δυναμικά σε ένα νήμα. Η διαδικασία επαναλαμβάνεται έως ότου το μέγεθος γίνει ίσο με 1, ή ο βρόχος τερματιστεί.

– **runtime**. Στον συγκεκριμένο τύπο διαμοιρασμού, ο τρόπος χρονοδρομολόγησης ορίζεται από την τιμή της μεταβλητής περιβάλλοντος `OMP_SCHEDULE`, η οποία περιγράφεται στη συνέχεια.

- **Φράση reduction(πράξη : λίστα μεταβλητών)**. Με τη χρήση της φράσης `reduction`, επιτυγχάνεται αφαίρεση στα αντικείμενα της λίστας μεταβλητών, μέσω της πράξης που έχει δοθεί ως όρισμα. Ουσιαστικά, στο πέρας της περιοχής παραλληλίας, δημιουργείται ένα τοπικό αντίγραφο κάθε μεταβλητής της λίστας για το κάθε νήμα. Έπειτα, για κάθε μεταβλητή, πραγματοποιείται η πράξη (+, -, *, /) μεταξύ των αντίστοιχων αντιγράφων των νημάτων. Το αποτέλεσμα της πράξης αποθηκεύεται στην αντίστοιχη μεταβλητή του κυρίου νήματος και, έτσι, εξασφαλίζεται συνέπεια των δεδομένων.

2.4 Συσκευές OpenMP (OpenMP Devices)

Στο OpenMP, ως συσκευή ορίζουμε οποιοδήποτε υπολογιστικό σύστημα μπορεί να χρησιμοποιηθεί για την εκτέλεση κώδικα. Ο κώδικας αυτός μπορεί με τη σειρά του να περιλαμβάνει περαιτέρω οδηγίες και φράσεις του προτύπου OpenMP.

Ο κύριος επεξεργαστής μπορεί να θεωρηθεί ως μια συσκευή. Το πρότυπο OpenMP ονομάζει τη συσκευή αυτή ως *host device*. Η παραδοχή αυτή διευκολύνει την κατανόηση της δομής του OpenMP.

Η υλοποίηση μιας συσκευής OpenMP, όπως ορίζονται προγραμματιστικά από το πρότυπο, πραγματοποιείται με συγκεκριμένο κώδικα, κατά τον οποίο λαμβάνονται υπ'όψη οι ιδιότητες και οι περιορισμοί της, για να επιτευχθεί αποδοτική εκτέλεση. Είναι μια εργασία που αφορά τον προγραμματισμό μεταγλωττιστών. Συνήθως, ο συνολικός κώδικας υλοποιείται ως βιβλιοθήκη. Η βιβλιοθήκη πρέπει να περιλαμβάνει κατ'ελάχιστον μια στοιχειώδη διεπαφή με το σύστημα host, το οποίο παραδοσιακά είναι ο κύριος επεξεργαστής, καθώς επίσης και την υλοποίηση του συνόλου των

ρουτινών της συσκευής που της επιτρέπουν να υποστηρίζει, έως έναν βαθμό, το πρότυπο OpenMP, στο εσωτερικό του υπολογιστικού πυρήνα που αυτή εκτελεί.

2.5 Αποστολή Κώδικα και Δεδομένων (Offloading/Mapping)

Ο κύριος επεξεργαστής μπορεί ανά πάσα στιγμή να χρησιμοποιήσει τις συνοδές συσκευές του συστήματος, για να αποστείλει δεδομένα ή κώδικα προς εκτέλεση, ώστε να επιταχύνει ορισμένους υπολογισμούς. Γενικότερα, οι συσκευές προτιμούνται έναντι ενός συστήματος γενικής χρήσης λόγω της ταχύτητάς τους σε συγκεκριμένες πράξεις ή λόγω άλλων ιδιαιτεροτήτων τους. Παραδείγματα συσκευών είναι μια κάρτα γραφικών, η οποία έχει την ιδιότητα να εκτελεί πράξεις κινητής υποδιαστολής γρήγορα και μαζικά, ένας συνεπεξεργαστής, ο οποίος λειτουργεί σε συνδυασμό με τον κύριο επεξεργαστή.

Η αποστολή ενός τμήματος κώδικα προς μια συνοδή συσκευή, γίνεται με την χρήση της βιβλιοθήκης συσκευής που αναφέρθηκε στην προηγούμενη ενότητα. Η βιβλιοθήκη αυτή, θα πρέπει να διαθέτει ένα σύντολο συναρτήσεων που αναλαμβάνουν την αρχικοποίηση/τερματισμό της συσκευής, την εγγραφή και ανάγνωση δεδομένων προς/από τη μνήμη της, καθώς και τη μεταφορά εκτελέσιμου κώδικα.

Η διαδικασία μεταφοράς κώδικα σε μία συσκευή ονομάζεται *φόρτωση* (offloading) και περιλαμβάνει, συν τοις άλλοις, την παραμετροποίηση του εκτελέσιμου κώδικα προτού αυτός φορτωθεί. Παραδείγματα παραμετροποίησης είναι ο ορισμός του αριθμού των νημάτων που θα εκτελέσουν τον κώδικα και η προετοιμασία των ορισμάτων του.

Όμως, η εκτέλεση ενός τμήματος κώδικα σε μία συσκευή δεν θα είχε κανένα νόημα, χωρίς την ύπαρξη του μηχανισμού *αντιστοίχισης δεδομένων* (data mapping), ο οποίος περιλαμβάνει τις διαδικασίες μεταφοράς δεδομένων προς και από την συσκευή. Η κατεύθυνση *προς* χρησιμοποιείται για τη μεταφορά δεδομένων που αναπαριστούν την είσοδο στο τμήμα κώδικα, ενώ η κατεύθυνση *από* για τη μεταφορά αποτελεσμάτων από τη συσκευή προς τον κύριο επεξεργαστή. Η διαδικασία ονομάζεται αντιστοίχιση διότι:

- Τα δεδομένα που μεταφέρονται αφορούν μεταβλητές, οι οποίες έχουν κοινή ονομασία ανάμεσα στον κύριο επεξεργαστή και τη συσκευή

- Επιλύει οποιαδήποτε ζητήματα προκύπτουν σχετικά με τις διευθύνσεις των μεταβλητών και την προσπέλασή τους από τη συσκευή, όπως για παράδειγμα εάν γίνεται χρήση κοινόχρηστων μεταβλητών ή δεικτών.

2.5.1 Περιοχές Φόρτωσης (Target Regions)

Στο OpenMP, οι περιοχές οι οποίες ενεργοποιούν τις παραπάνω διαδικασίες, ονομάζονται *περιοχές φόρτωσης* (target regions). Η βασική μέθοδος αποστολής κώδικα και δεδομένων σε μια συσκευή είναι η χρήση της οδηγίας `#pragma omp target`. Η σύνταξη της οδηγίας είναι η εξής:

#pragma omp *target* [*φράση*[[,] *φράση*] ...] *νέα-γραμμή*
δομημένο τμήμα

Όταν το κύριο νήμα συναντήσει μια περιοχή `target`, τότε δημιουργείται για τη συσκευή ένα *περιβάλλον δεδομένων* (data environment), το οποίο περιλαμβάνει όλα τα δεδομένα που θα μεταφερθούν από/προς τη συσκευή προς/από το κύριο πρόγραμμα. Το περιβάλλον δεδομένων, μαζί με το δομημένο τμήμα κώδικα που φορτώνεται στη συσκευή, ονομάζεται ως *υπολογιστικός πυρήνας* (kernel). Για ευκολία θα το ονομάζουμε απλά πυρήνα, όπου δεν υπάρχει σύγχυση με τους πυρήνες (cores) από τους οποίους αποτελείται ένας επεξεργαστής.

Όταν εκτελεστεί η περιοχή `target` στην συσκευή, αρχικά θα δημιουργηθεί ένα κύριο νήμα, το λεγόμενο `κύριο νήμα συσκευής`. Το κύριο νήμα θα εκτελέσει σειριακά τον κώδικα, εκτός και αν συναντηθούν οδηγίες που δημιουργούν περιοχές παραλληλίας (`parallel`). Συνεπώς, η συμπεριφορά του κυρίου νήματος συσκευής είναι πανομοιότυπη με αυτή του κυρίου νήματος του κύριου επεξεργαστή.

Αναφορά στις φράσεις που μπορούν να συνοδεύσουν μια οδηγία `target`, γίνεται στην ενότητα 2.5.3, ενώ ολοκληρωμένο παράδειγμα κώδικα δίνεται στο σχήμα 2.2.

2.5.2 Περιοχές Φόρτωσης Δεδομένων (Target Data Regions)

Υπάρχει η δυνατότητα δημιουργίας μόνο ενός περιβάλλοντος δεδομένων για τη συσκευή, χωρίς την φόρτωση κώδικα. Η διαδικασία αυτή προσφέρεται μέσω της οδηγίας `target data`, η οποία συντάσσεται ως εξής:

```
#include "omp.h"
int main()
{
    int i, x[16];
    #pragma omp target map(tofrom:x) private(i)
    {
        for (i = 0; i < 16; i++)
            x[i] = i;
    }
}
```

Σχήμα 2.2: Παράδειγμα περιοχής φόρτωσης target OpenMP

#pragma omp *target data* [*φράση*[[,] *φράση*] ...] *νέα-γραμμή*
δομημένο τμήμα

Αξίζει να σημειωθεί ότι στη συγκεκριμένη οδηγία η λέξη-κλειδί `data` δεν αποτελεί φράση της οδηγίας `target`.

Σκοπός των περιοχών `target data` είναι η ελάττωση του αριθμού των μεταφορών δεδομένων, με τη δημιουργία ενός μοναδικού περιβάλλοντος δεδομένων, αντί πολλών, όπως συμβαίνει σε περιπτώσεις επαναλαμβανόμενων περιοχών `target`.

Από την έκδοση 4.5 του προτύπου OpenMP, δίνεται η δυνατότητα άμεσης αποστολής/λήψης δεδομένων προς και από το περιβάλλον δεδομένων της συσκευής, με τις οδηγίες `target enter data` για την εισαγωγή δεδομένων και `target exit data` για τη εξαγωγή τους και αποθήκευσή τους στη μνήμη του κύριου επεξεργαστή. Η σύνταξη των οδηγιών αυτών είναι η εξής:

#pragma omp *target enter data* [*φράση*[[,] *φράση*] ...] *νέα-γραμμή*

#pragma omp *target exit data* [*φράση*[[,] *φράση*] ...] *νέα-γραμμή*

Παρατηρούμε ότι οι οδηγίες αυτές είναι αυτοδύναμες, με την έννοια ότι υπολείπονται δομημένου τμήματος στο εσωτερικό τους.

2.5.3 Φράσεις Οδηγιών target/target data

Οι φράσεις των οδηγιών `target` και `target data` αποτελούν αναπόσπαστο κομμάτι της χρήσης τους. Μέσω των συγκεκριμένων φράσεων, ο προγραμματιστής μπορεί να καθορίσει τον τρόπο με τον οποίο τα δεδομένα εισέρχονται και εξέρχονται του περιβάλλοντος δεδομένων της συσκευής. Καθορίζεται η κατεύθυνσή τους, η οποία

είναι είτε προς ή από το κύριο πρόγραμμα, πότε αυτά τα θα ενημερωθούν και με ποιον τρόπο. Οι πιο σημαντικές φράσεις που σχετίζονται με περιοχές φόρτωσης κώδικα είναι οι εξής:

- **device(έκφραση ακεραίων)**. Ορίζει τη συσκευή στην οποία θα πραγματοποιηθεί η φόρτωση κώδικα και δεδομένων. Η τελική τιμή της έκφρασης ακεραίων θα πρέπει να ισούται με το αναγνωριστικό της συσκευής.

- **map([τύπος:] λίστα μεταβλητών)**. Καθορίζει τον τρόπο με τον οποίον θα γίνει η αντιστοίχιση των δεδομένων του κύριου επεξεργαστή με τα δεδομένα της συσκευής. Ο τύπος αντιστοίχισης μπορεί να λάβει μία εκ των τεσσάρων παρακάτω τιμών:

  - **to**: Με τη χρήση του τύπου `to`, οι μεταβλητές που ανήκουν στη λίστα μεταβλητών απλώς αντιγράφονται από τη μνήμη του κύριου επεξεργαστή στο περιβάλλον δεδομένων της συσκευής. Οποιαδήποτε τροποποίησή τους στο εσωτερικό της περιοχής `target` δεν θα επηρεάσει τις μεταβλητές του επεξεργαστή.

  - **from**: Οι μεταβλητές που αντιστοιχίζονται με τον τύπο `from`, έχουν ακαθόριστη αρχική τιμή στο περιβάλλον δεδομένων της συσκευής, αλλά με το πέρας της περιοχής `target` γίνεται αντιγραφή των τιμών τους στη μνήμη του κύριου επεξεργαστή. Συνήθως, ως `from` αντιστοιχίζονται μεταβλητές που αναπαριστούν αποτελέσματα.

  - **tofrom** (προεπιλογή): Από προεπιλογή, εάν δεν καθοριστεί τύπος αντιστοίχισης, οι μεταβλητές χαρακτηρίζονται ως `tofrom`, ένας τύπος που αποτελεί συνδυασμό των προηγούμενων δύο.

  - **alloc**: Με τον τύπο alloc, η αρχική τιμή του κάθε νέου αντικειμένου της λίστας μεταβλητών είναι ακαθόριστη.

2.6 Ομάδες Νημάτων (Teams)

Μια σημαντική λειτουργικότητα που ορίζεται στο πρότυπο OpenMP είναι η μαζική δημιουργία ομάδων νημάτων, μέσω της οδηγίας *#pragma omp teams*. Στο εσωτερικό μιας οδηγίας ομάδων βρίσκεται μια περιοχή κώδικα. Όταν το κύριο νήμα συναντήσει

μια περιοχή ομάδων, τότε δημιουργείται μια σειρά από ομάδες, οι οποίες αρχικά αποτελούνται μόνο από το αρχικό τους νήμα, το οποίο καλείται και νήμα-αρχηγός. Κάθε αρχικό νήμα θα εκτελέσει την περιοχή που περικλείεται από την οδηγία. Τα υπόλοιπα νήματα-μέλη μιας ομάδας μπορούν να θεωρηθούν ως απενεργοποιημένα, αν και η κατάστασή τους καθορίζεται από την εκάστοτε υλοποίηση. Σημειώνεται ότι το νήμα που συνάντησε την οδηγία teams, θα αποτελέσει το νήμα-αρχηγό μίας από τις δημιουργηθείσες ομάδες.

Η σύνταξη της οδηγίας `teams` είναι η εξής:

#pragma omp *teams* [*φράση*[[,] *φράση*] ...] *νέα-γραμμή*
δομημένο τμήμα

Το δομημένο τμήμα αφορά την περιοχή που θα εκτελεστεί από τα κύρια νήματα των ομάδων. Νήμα στις ομάδες δίνουν περαιτέρω εντολές που μπορούν να χρησιμοποιηθούν μέσα στην περιοχή ομάδων, όπως για παράδειγμα η οδηγία κατανομής *#pragma omp distribute*. Η οδηγία αυτή λειτουργεί όπως η εντολή κατανομής εργασίας *for*, με τη διαφορά ότι προορίζεται για χρήση σε συνδυασμό με την οδηγία ομάδων, όπως ακριβώς η *for* συνδυάζεται με την οδηγία *parallel*.

Οι οδηγίες αυτές αρχικά αναπτύχθηκαν για αποκλειστική χρήση με συσκευές. Ο όρος της ομάδας μπορεί να αντιστοιχιστεί άμεσα με ένα τμήμα λογικής οργάνωσης των πυρήνων, σε μια μονάδα γραφικής επεξεργασίας. Στην έκδοση 5.0, όμως, του προτύπου, οι οδηγίες αυτές γενικεύθηκαν, ώστε να μπορούν να χρησιμοποιηθούν και από τον κύριο επεξεργαστή.

Οι σημαντικότερες φράσεις που μπορούν να συνοδεύσουν την οδηγία teams είναι οι:

- **num_teams(N)**. Η φράση `num_teams` καθορίζει το πλήθος των ομάδων που θα δημιουργηθούν, όταν συναντηθεί η οδηγία.
- **thread_limit(N)**. Η φράση `thread_limit` καθορίζει ένα άνω όριο του πλήθους των νημάτων που θα αποτελούν την κάθε ομάδα που πρόκειται να δημιουργηθεί.

2.6.1 Κατανομή Εργασίας σε Ομάδες (Distribute)

Όπως προαναφέρθηκε, μια σημαντική λειτουργία του OpenMP είναι η κατανομή της εργασίας ανά τις ομάδες που δημιουργούνται μέσω της οδηγίας *teams*. Η οδηγία κατανομής *distribute* χρησιμοποιείται σε συνδυασμό με έναν επαναληπτικό βρόχο, του

οποίου οι επαναλήψεις διαμοιράζονται στα κύρια νήματα των ομάδων. Η χρονοδρομολόγηση, όσον αφορά τον διαμοιρασμό της εργασίας, είναι πάντοτε στατική.

Ενώ αποκτά περισσότερο νόημα η χρήση της οδηγίας *distribute* εντός μιας περιοχής ομάδας, μπορεί να χρησιμοποιηθεί και αυτόνομα, όπως ακριβώς η οδηγία *for*. Ωστόσο, η χρήση αυτή δεν συναντάται συχνά. Η οδηγία συντάσσεται ως εξής:

#pragma omp *distribute* [*φράση*[[,] *φράση*] ...] *νέα-γραμμή*
`for` (*αρχικοποίηση*; *συνθήκη*; *βήμα*) `{`
...
`}`

Οι σημαντικότερες φράσεις που μπορούν να συνοδεύσουν την οδηγία `distribute` είναι οι:

- **firstprivate(λίστα μεταβλητών)**.
- **lastprivate(λίστα μεταβλητών)**.
- **private(λίστα μεταβλητών)**.
- **dist_schedule(static[, μέγεθος κόκκου])**. Ακολουθώντας παρόμοια λογική με την χρονοδρομολόγηση στην οδηγία `for`, η φράση αυτή χρησιμοποιείται μόνο για να καθορίσει το μέγεθος του κόκκου παραλληλίας, μιας και ο μοναδικός τύπος χρονοδρομολόγησης είναι ο στατικός.
- **collapse(N)**. Η φράση αυτή χρησιμοποιείται για την σύμπτυξη `N` εμφωλευμένων επαναληπτικών βρόχων, σε έναν μοναδικό.

Σημειώνεται ότι η συγκεκριμένη οδηγία, μπορεί να χρησιμοποιηθεί με την οδηγία *teams* και στη *συνδυασμένη* (combined) της μορφή, ως *#pragma omp teams distribute*. Στην περίπτωση αυτή, στο εσωτερικό της οδηγίας, περιλαμβάνεται μόνο ο βρόχος προς διαμοιρασμό.

2.6.2 Κατανομή Εργασίας σε Ομάδες με Παραλληλοποίηση (Distribute Parallel Worksharing)

Το πρότυπο OpenMP επιτρέπει τον περαιτέρω διαμοιρασμό του φόρτου ενός συνόλου επαναλήψεων που έχει προηγουμένως κατανεμηθεί μέσω της οδηγίας `distribute` στα αρχικά νήματα των ομάδων. Μέσω της οδηγίας `#pragma omp distribute parallel`

for, στο εσωτερικό μιας περιοχής ομάδων, τα νήματα-αρχηγοί, έχουν τη δυνατότητα επιπλέον να δημιουργούν περιοχές παραλληλίας που στο εσωτερικό τους περιλαμβάνουν τον βρόχο προς διαμοιρασμό. Η διαδικασία μπορεί να θεωρηθεί ότι πραγματοποιείται με τα εξής νοητά βήματα:

1. Το συνολικό πλήθος επαναλήψεων του βρόχου κατανέμεται στα κύρια νήματα των ομάδων, με τρόπο που ορίζει η οδηγία *distribute*

2. Τα κύρια νήματα των ομάδων, που πλέον διαθέτουν ένα τμήμα των επαναλήψεων, ενεργοποιούν τα υπόλοιπα νήματα της ομάδας, με αποτέλεσμα το μέγεθος της ομάδας να αυξάνεται από ένα νήμα σε περισσότερα

3. Το τμήμα επαναλήψεων κάθε ομάδας διαμοιράζεται περαιτέρω στα νήματα που την αποτελούν, με τον τρόπο που ορίζει η οδηγία *for*, επιτυγχάνοντας τελικά δύο επίπεδα κατανομής του αρχικού βρόχου

Η σύνταξη της οδηγίας *distribute parallel for* είναι η εξής:

#pragma omp *distribute parallel for* [*φράση*[[,] *φράση*] ...] *νέα-γραμμή*
for (*αρχικοποίηση*; *συνθήκη*; *βήμα*) {
...
}

Οι αποδεκτές φράσεις της οδηγίας είναι ένας συνδυασμός των πιθανών φράσεων της οδηγίας *distribute* και της οδηγίας *parallel for*. Σημειώνεται ότι η συγκεκριμένη οδηγία, μπορεί να χρησιμοποιηθεί με την οδηγία *teams* και στη συνδυασμένη της μορφή, ως *#pragma omp teams distribute parallel for*. Στην περίπτωση αυτή, όπως και προηγουμένως, περιλαμβάνεται μόνο ο βρόχος προς διαμοιρασμό στο εσωτερικό της οδηγίας.

```
#include "omp.h"
int main() {
    int x[16];
    #pragma omp target
    {
        #pragma omp teams num_teams(16)
        {
            int my_id = omp_get_team_num();
            x[my_id] = my_id;
        }
    }
}
```

Σχήμα 2.3: Παράδειγμα περιοχής teams μέσα σε περιοχή target στο OpenMP

Κεφαλαιο 3

Συσκευεσ Γραφικησ Επεξεργασιασ και CUDA

3.1 Εισαγωγή

Οι μονάδες γραφικής επεξεργασίας (GPUs) θεωρούνται πλέον από τα πιο κρίσιμα συστατικά ενός υπολογιστικού συστήματος. Στα πρώτα τους βήματα, με τη μορφή των λεγόμενων καρτών γραφικών, η διαδικασία που εκτελούσαν ήταν η λήψη δεδομένων δυαδικής μορφής από τον κεντρικό επεξεργαστή και, η μετατροπή τους σε εικόνες, οι οποίες εμφανίζονται στην οθόνη. Τυπικά, τα δεδομένα αφορούν τα χρώματα που πρέπει να απεικονιστούν σε κάθε εικονοστοιχείο μιας οθόνης.

Όπως και ο κύριος επεξεργαστής του συστήματος, έτσι και οι μονάδες γραφικής επεξεργασίας διαθέτουν τη δική τους μνήμη. Η μνήμη αυτή χρησιμοποιείται για την επικοινωνία τους με τις επεξεργαστικές μονάδες, με τις οποίες και συνδέεται μέσω ενός διαύλου υψηλής ταχύτητας. Σκοπός της μνήμης μιας μονάδας γραφικής επεξεργασίας είναι η κατάλληλη επεξεργασία και προετοιμασία των εικόνων που είναι έτοιμες προς εμφάνιση, αλλά και οποιασδήποτε άλλης πληροφορίας. Επιπλέον,

ΚΜΕ	GPU
Λίγοι ισχυροί πυρήνες	Χιλιάδες πιο αδύναμοι πυρήνες
Χαμηλή καθυστέρηση	Υψηλή ρυθμαπόδοση
Ιδανικές για σειριακή επεξεργασία	Ιδανικές για παράλληλη επεξεργασία
Δεκάδες υπολογισμοί τη φορά	Χιλιάδες υπολογισμοί τη φορά
Μεγάλη κατανάλωση μνήμης	Μικρή κατανάλωση μνήμης

Πίνακας 3.1: Σύγκριση κεντρικών μονάδων επεξεργασίας (ΚΜΕ) με τις μονάδες GPU.

κύριο χαρακτηριστικό της είναι η δυνατότητά της να δέχεται τόσο αναγνώσεις όσο και εγγραφές την ίδια χρονική στιγμή.

Φυσικά, η προετοιμασία των εικονοστοιχείων για προβολή σε μια οθόνη είναι μια εντελώς δυναμική διαδικασία, η οποία απαιτεί σύνθετες μαθηματικές πράξεις και γεωμετρικούς υπολογισμούς. Έως έναν βαθμό, η διαδικασία αυτή θα μπορούσε να πραγματοποιηθεί από την κύρια επεξεργαστική μονάδα του συστήματος, παρ' όλα αυτά η φύση της δεν της επιτρέπει να πετύχει τις απαραίτητες επιδόσεις. Για αυτόν τον σκοπό, το σύστημα μπορεί να εκμεταλλευτεί τις μονάδες GPU, οι οποίες βάσει του σχεδιασμού τους, είναι κατάλληλες για εκτέλεση χιλιάδων αφενός απλών πράξεων σε μια χρονική στιγμή, αφετέρου σε μεγάλη κλίμακα λόγω του πλήθους των πυρήνων που διαθέτουν. Αντιθέτως, μια επεξεργαστική μονάδα αποτελείται από δεκάδες πυρήνες, ικανούς να εκτελέσουν δεκάδες πράξεις ταυτόχρονα. Στον πίνακα 3.1 παρουσιάζονται οι κύριες διαφορές ενός επεξεργαστή με μια μονάδα γραφικής επεξεργασίας.

3.2 Ανασκόπηση

Μέσα σε μόνο μερικές δεκαετίες, οι μονάδες GPU εξελίχθηκαν από μονοπύρηνα συστήματα τα οποία διέθεταν συγκεκριμένες λειτουργίες γραφικού σκοπού, σε συστήματα χιλιάδων προγραμματίσιμων πυρήνων. Η ιστορία των σύγχρονων μονάδων GPU ξεκινάει το 1995, με την εμφάνιση του πρώτου υλικού τρισδιάστατης απεικόνισης, το οποίο υιοθετήθηκε από τους προσωπικούς Η/Υ. Προηγουμένως, η βιομηχανία των μονάδων GPU βασιζόταν κυρίως στη διδιάστατη απεικόνιση, με μια αρχιτεκτο-

νική που δεν θυμίζει τόσο τη σημερινή.

Η πρώτη αρχιτεκτονική των μονάδων GPU βασιζόταν στο μοντέλο της διοχέτευσης και περιλάμβανε το στάδιο μετατροπής των δεδομένων του επεξεργαστή σε εικόνες (rasterization), καθώς και μια προσωρινή μνήμη καρέ (frame buffer), η οποία χρησιμοποιούταν για την αποθήκευση των παραγόμενων εικόνων (καρέ). Στη συνέχεια, εισήχθησαν περαιτέρω στάδια που μέχρι τότε υλοποιούνταν από τον κύριο επεξεργαστή του συστήματος. Τα στάδια αυτά ήταν οι μετασχηματισμοί κορυφών (vertex transformations) και η μετατροπή των εντολών σχεδιασμού κορυφών σε εντολές που έχουν νόημα για την μονάδα GPU.

Με το πέρασμα των χρόνων και με την απήχηση των μονάδων GPU, οι κατασκευαστές τους ξεκίνησαν να εισάγουν νέες δυνατότητες, οι οποίες αφορούν:

- Την παραλληλοποίηση της γραφικής διοχέτευσης, η οποία έγινε δυνατή με την εισαγωγή πολλαπλών πυρήνων στις μονάδες GPU, επιτρέποντας την απεικόνιση σύνθετων διαδραστικών τρισδιάστατων σκηνών
- Την διάθεση των πυρήνων προς προγραμματισμό από το χρήστη, με την εισαγωγή διεπαφών στις ήδη υπάρχουσες γλώσσες προγραμματισμού

Σήμερα, οι μονάδες GPU έχουν μετονομαστεί σε *γενικού σκοπού* (general purpose GPUs - GPGPUs), διότι εκτός από γραφική απεικόνιση στην οθόνη, μπορούν να εκτελούν και γενικούς υπολογισμούς, μέσω του είδους των πυρήνων που διαθέτουν. Η μετάβαση από πυρήνες συγκεκριμένης λειτουργικότητας σε πυρήνες που έχουν τη δυνατότητα να συντονίζονται και να εκτελούν παράλληλα χιλιάδες υπολογισμούς σε μια χρονική στιγμή, είναι η αιτία προτίμησης των μονάδων GPU έναντι των κύριων επεξεργαστών, σε υπολογισμούς που αφορούν δεδομένα ορισμένης μορφής.

Μπορούμε να κατηγοριοποιήσουμε τις μονάδες GPU σε δύο κατηγορίες, ανάλογα με το προγραμματιστικό μοντέλο που χρησιμοποιείται για τον γενικό προγραμματισμό τους. Στην πρώτη κατηγορία ανήκουν οι μονάδες που χρησιμοποιούν την OpenCL και στη δεύτερη αυτές που βασίζονται στο μοντέλο CUDA. Τα δύο αυτά μοντέλα επιτρέπουν τον άμεσο προγραμματισμό των συσκευών που τα υποστηρίζουν, μέσω των αντίστοιχων προγραμματιστικών διεπαφών. Το μοντέλο της OpenCL υποστηρίζεται σχεδόν από όλους τους κατασκευαστές καρτών γραφικών, ενώ το CUDA αποκλειστικά από την NVIDIA. Επιπλέον, η χρήση της OpenCL δεν περιορίζεται σε μονάδες GPU, αλλά υποστηρίζει γενικότερα ένα μεγάλο πλήθος αρχιτεκτονικών, όπως η ARM. Οι κατασκευαστές που επιθυμούν να υποστηρίξουν

την OpenCL στα συστήματά τους, έχουν τη δυνατότητα να προχωρήσουν σε υλοποίησή της. Πάντως, συγκρίνοντας τα δύο μοντέλα, μπορεί κανείς να παρατηρήσει ότι υπάρχουν πολλές ομοιότητες στη λογική με την οποία χειρίζεται το καθένα τα υποκείμενα συστήματα που τα υποστηρίζουν.

3.3 Το Μοντέλο CUDA

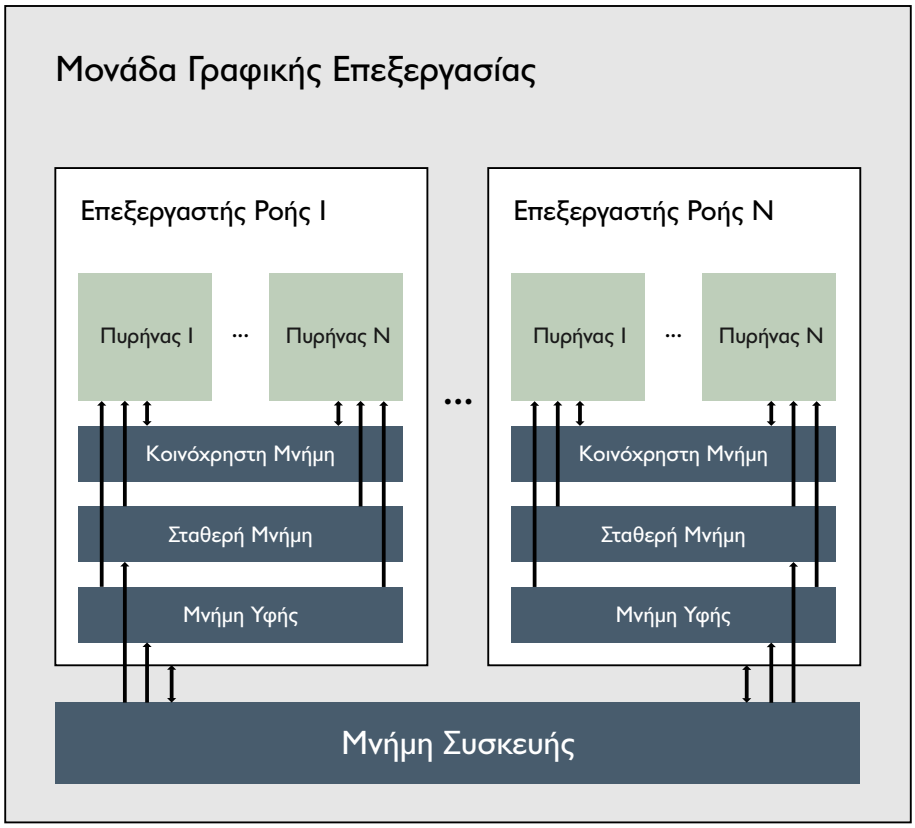
Ο όρος CUDA χρησιμοποιείται συχνά για την αναφορά σε δύο διαφορετικές έννοιες. Η πρώτη, είναι η πλατφόρμα παράλληλου υπολογισμού, η οποία ενσωματώνεται με τη μορφή υλικολογισμικού στις μονάδες γραφικής επεξεργασίας NVIDIA και επιτρέπει τη χρήση των *πυρήνων* (cores) γενικού σκοπού που αυτές διαθέτουν. Η δεύτερη έννοια, αφορά την προγραμματιστική διεπαφή, η οποία αποτελείται από το τμήμα του host και το τμήμα του κώδικα συσκευής και επιτρέπει σε μια εφαρμογή να χρησιμοποιήσει την πλατφόρμα. Σημειώνεται ότι από εδώ και στο εξής, θα χρησιμοποιούμε τους όρους "υπολογιστικός πυρήνας" και "kernel" για να αναφερόμαστε στον κώδικα που εκτελείται σε μια μονάδα GPU. Οι όροι αυτοί θα χρησιμοποιούνται ως ισοδύναμοι.

Στις επόμενες ενότητες περιγράφεται η αρχιτεκτονική της πλατφόρμας CUDA και γενικότερα των μονάδων γραφικής επεξεργασίας που την υποστηρίζουν. Έπειτα, γίνεται αναφορά στην προγραμματιστική διεπαφή της CUDA και παρατίθενται παραδείγματα κώδικα που τη χρησιμοποιούν.

3.3.1 Η Αρχιτεκτονική

Η πλατφόρμα CUDA είναι ένα επίπεδο υλικολογισμικού το οποίο πρακτικά δίνει άμεση πρόσβαση στο σύνολο εικονικών εντολών GPU και στα παράλληλα υπολογιστικά στοιχεία μιας μονάδας γραφικής επεξεργασίας, για την εκτέλεση των kernel. Ως πυρήνα, ονομάζουμε τον ωφέλιμο κώδικα τον οποίο εκτελείται από ένα σύνολο νημάτων της μονάδας GPU.

Η CUDA διαθέτει δική της οργάνωση συστήματος και μνήμης. Για να κατανοηθεί πλήρως, σημαντική είναι πρώτα η φυσική οργάνωση των συσκευών επεξεργασίας. Κάθε μονάδα γραφικής επεξεργασίας NVIDIA αποτελείται από έναν αριθμό πολυνηματικών *επεξεργαστών ροής* (Streaming Multiprocessors). Οι πολυεπεξεργαστές, σαν έννοια, μπορούν να αντιστοιχιστούν στον επεξεργαστή ενός κύριου συστήματος.

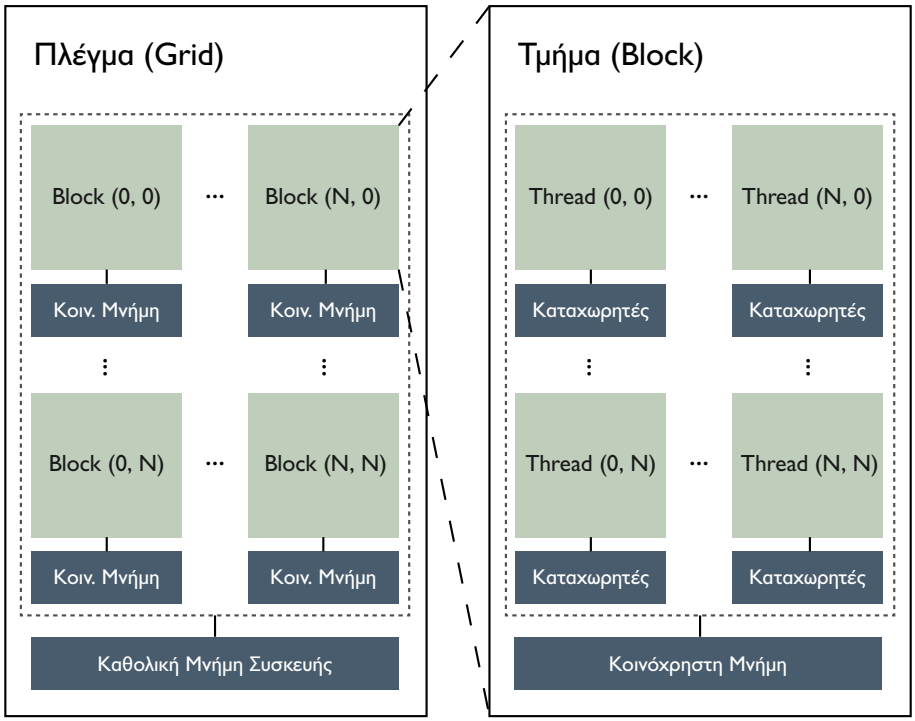


Σχήμα 3.1: Η οργάνωση μιας μονάδας γραφικής επεξεργασίας NVIDIA

Στο εσωτερικό των επεξεργαστών ροής, συναντούνται οι *πυρήνες* (cores), ακριβώς όπως και σε έναν κύριο επεξεργαστή. Κάθε πυρήνας εκτελεί το δικό του *νήμα* (thread), το οποίο αποτελεί τη βασική οντότητα εκτέλεσης. Το σχήμα 3.1 απεικονίζει την φυσική οργάνωση μιας μονάδας γραφικής επεξεργασίας NVIDIA.

Στην CUDA, κατά βάση όλοι οι προαναφερθέντες όροι αντιστοιχίζονται με νέους. Αυτό συμβαίνει διότι κάθε μονάδα GPU, στο φυσικό της επίπεδο, συνήθως διαφέρει από τις υπόλοιπες. Για αυτόν τον σκοπό, απαραίτητη είναι η θέσπιση ενός λογικού επιπέδου, που καλύπτει όλες τις διαφορές ανά τις μονάδες GPU και προσφέρει ομοιομορφία. Συγκεκριμένα, κατά τη διεπαφή, ολόκληρη η μονάδα GPU ονομάζεται ως *πλέγμα* (grid). Το πλέγμα οργανώνεται περαιτέρω σε *τμήματα* (blocks), τα οποία ουσιαστικά είναι οι επεξεργαστές ροής. Κάθε τμήμα, περιλαμβάνει μια ομάδα από νήματα και κάθε νήμα εκτελείται από έναν φυσικό πυρήνα της μονάδας GPU. Τα είδη των μνημών κρατούν την ονομασία τους. Το σχήμα 3.2 απεικονίζει την αρχιτεκτονική της πλατφόρμας CUDA.

Κάθε επεξεργαστής ροής διαθέτει τη δική του ιδιωτική μνήμη, η οποία είναι άμεσα προσβάσιμη από τους πυρήνες και δομείται σε τρία επίπεδα:



Σχήμα 3.2: Η οργάνωση της πλατφόρμας επιπέδου λογισμικού CUDA

1. **Κοινόχρηστη μνήμη (Shared memory)**. Το πρώτο επίπεδο μνήμης είναι η κοινόχρηστη μνήμη, η οποία χρησιμοποιείται για αποθήκευση δεδομένων κοινού τύπου, όπως μεταβλητές του kernel. Μέσω της κοινόχρηστης μνήμης, οι πυρήνες ενός επεξεργαστή ροή μπορούν να επικοινωνούν και να συγχρονίζουν την εκτέλεσή τους.

2. **Σταθερή μνήμη (Constant memory)**. Το δεύτερο επίπεδο μνήμης είναι η σταθερή μνήμη, η οποία χρησιμοποιείται για δεδομένα που δεν πρόκειται να μεταβληθούν. Σημειώνεται ότι ο τύπος μνήμης αυτός τίθεται μόνο για ανάγνωση.

3. **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, η ίδια η διεπαφή CUDA αποτελείται από τη διεπαφή χρόνου εκτέλεσης (runtime) και τη διεπαφή οδηγού (driver). Η πρώτη προσφέρεται στους προγραμματιστές για άμεση χρήση μιας μονάδας CUDA μέσα από εφαρμογές, αποτελώντας υψηλού επιπέδου διεπαφή. Η διεπαφή driver είναι χαμηλού, προσφέρει όμως καλύτερο προγραμματιστικό έλεγχο της μονάδας CUDA. Στη συνέχεια παρουσιάζεται η

διεπαφή runtime.

3.3.2 Η Διεπαφή Χρόνου Εκτέλεσης CUDA Runtime

Η προγραμματιστική διεπαφή CUDA runtime είναι μια επέκταση στη δημοφιλή γλώσσα προγραμματισμού C, με νέες λέξεις κλειδιά και δυνατότητες προς τους προγραμματιστές, οι οποίες τους επιτρέπουν να εκμεταλλευτούν τα ετερογενή υπολογιστικά συστήματα που αποτελούνται τόσο από επεξεργαστικές μονάδες όσο και από παράλληλες μονάδες GPU. Ονομάζουμε τη διεπαφή αυτή ως CUDA C [14]. Οι προγραμματιστές συχνά αναφέρονται στη CUDA C απλά ως CUDA. Κάθε μονάδα GPU υποστηρίζει στον δικό της βαθμό τη λειτουργικότητα που προσφέρεται από το μοντέλο CUDA. Ο βαθμός υποστήριξης εξαρτάται από την *υπολογιστικής έκδοσης* (compute capability) της κάθε μονάδας. Νεότερες μονάδες GPU διαθέτουν υψηλότερη υπολογιστική έκδοση και υποστηρίζουν νεότερες λειτουργίες, ενώ παλαιότερες μονάδες GPU υποστηρίζουν βασικότερες λειτουργίες.

Ένα πρόγραμμα σε CUDA C είναι ένας συνδυασμός σειριακών και παράλληλων εκτελέσεων. Κατά κανόνα, οι σειριακοί υπολογισμοί εκτελούνται στον host, ενώ οι παράλληλοι στα νήματα της μονάδας GPU, 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 3.3, το οποίο υλοποιεί την πρόσθεση διανυσμάτων. Η εκτέλεση ξεκινά από την κλήση της συνάρτησης `main` και τα διανύσματα αρχικοποιούνται σειριακά από το κύριο νήμα του host. Έπειτα, κάθε διάνυσμα πρέπει να αντιγραφεί στη GPU μέσω ειδικών κλήσεων του Runtime API. Στον πίνακα `d_C` θα αποθηκευτεί το αποτέλεσμα της πρόσθεσης, συνεπώς δεν χρειάζεται ακόμη κάποια ενέργεια για αυτόν. Στη γραμμή 14, συναντάται μια κλήση στον kernel `vecAdd`. Στο σημείο αυτό, γίνονται όλες οι απαραίτητες διαδικασίες ώστε να φορτωθεί η συνάρτηση `vecAdd` 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 στο

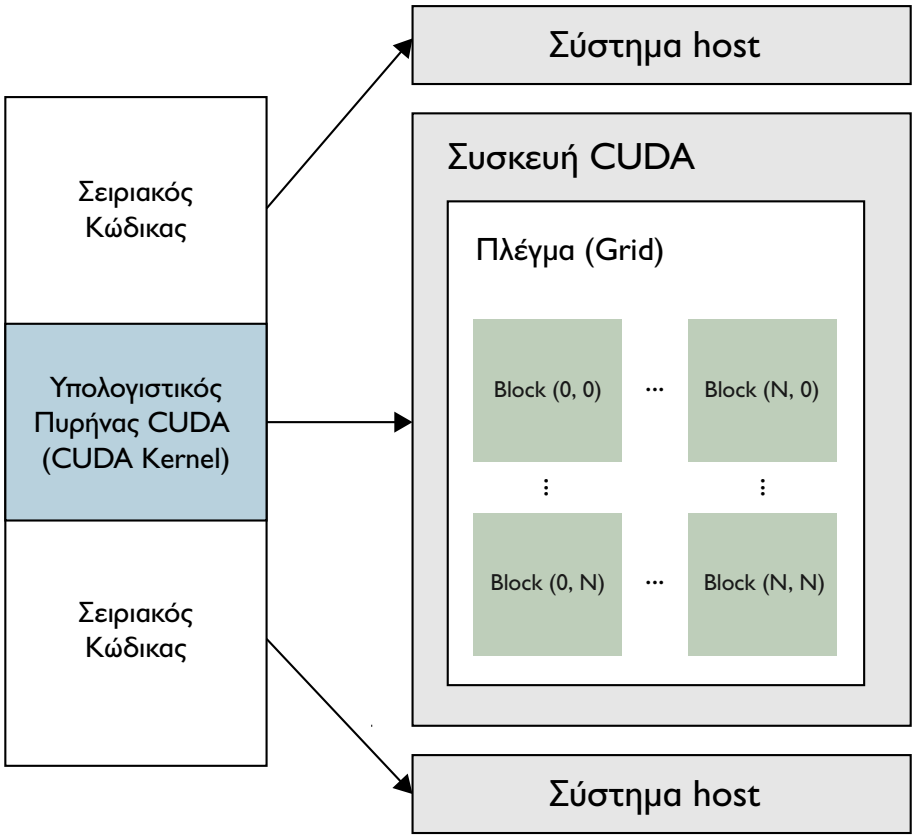
```
1  #define N 1024
2  __global__ void vecAdd(float* A, float* B, float* C)
3  {
4      int i = threadIdx.x;
5      C[i] = A[i] + B[i];
6  }
7  int main()
8  {
9      int x;
10     float *h_A, *h_B, *h_C, *d_A, *d_B, *d_C;
11     for (x = 0; x < N; x++)
12         h_A[x] = h_B[x] = x;
13     /* copy h_A, h_B to d_A, d_B */
14     vecAdd<<<1, N>>>(d_A, d_B, d_C);
15     /* copy d_C to h_B */
16 }
```

Σχήμα 3.3: Παράδειγμα προγράμματος CUDA C

καθένα. Στη συγκεκριμένη περίπτωση, θα δημιουργηθεί ένα block που αποτελείται από `N` (1024) νήματα. Κάθε νήμα, όταν εκτελέσει τον κώδικα της `vecAdd`, αρχικά θα βρει το αναγνωριστικό του (ID), με τη χρήση της δομής `threadIdx`. 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 ένα διάνυσμα `C`, για το οποίο ισχύει `C = A + B`. Το διάνυσμα βρίσκεται στη διεύθυνση `d_C` και συνεπώς θα πρέπει να αντιγραφεί στη μνήμη του host, κάτι που συμβαίνει μέσω ειδικών κλήσεων μνήμης, στη γραμμή 15. Χάρην αναγνωσιμότητας του παραδείγματος, ο κώδικας μεταφορών μνήμης έχει παραληφθεί και έχει αντικατασταθεί με τα σχετικά σχόλια (γραμμές 13 και 15).

Η λέξη-κλειδί, ή αλλιώς *προσδιοριστής* (specifier) `__global__` συμβολίζει ότι ο kernel `vecAdd` θα κληθεί από τον host, ώστε να εκτελεστεί στη μονάδα GPU. Άλλα είδη προσδιοριστών είναι τα `__device__` και `__host__`, τα οποία 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.

Το σχήμα 3.4 απεικονίζει τη βασική δομή ενός προγράμματος σε CUDA C. Τα σχετικά προγράμματα περιλαμβάνουν μεγάλες περιοχές σειριακού κώδικα, με ενδιάμεσες παράλληλες εκτελέσεις στη μονάδα GPU. Το μοτίβο αυτό μπορεί να επα-



Σχήμα 3.4: Ετερογενής προγραμματισμός με CUDA C

ναλαμβάνεται όσες φορές κρίνει απαραίτητο ο προγραμματιστής, για την επιτάχυνση των υπολογισμών.

3.3.3 Κριτική

Η σύνταξη ενός προγράμματος σε CUDA C, μπορεί φαινομενικά να μοιάζει με μια απλή διαδικασία, ειδικότερα στο σχήμα 3.3. Παρ' όλα αυτά, όταν η φύση ενός προβλήματος απαιτεί πολλά και διαφορετικά είδη υπολογισμών και, συνεπώς, διαφορετικούς κώδικες πυρήνα στην υλοποίηση μιας λύσης, ο βαθμός της ετερογένειας στη συγγραφή του προγράμματος αυξάνει δραματικά. Ο προγραμματιστής καλείται να προνοήσει για τα διάφορα είδη προσδιοριστών που συνοδεύουν τις συναρτήσεις πυρήνα, για τον διαχωρισμό του κώδικα host από τον κώδικα συσκευής, καθώς επίσης και για την διαφοροποίηση του κώδικα των νημάτων. Επιπλέον, καλείται να αποστηθίσει τις ιδιαιτερότητες και τις συμβάσεις της γλώσσας προγραμματισμού CUDA C.

Οι δυσκολίες αυτές που προκαλούνται από την ετερογενή φύση της διεπαφής

CUDA C, είναι αρκετές ώστε να αποθαρρύνουν έναν νέο προγραμματιστή να εκμεταλλευτεί τις επιδόσεις της αρχιτεκτονικής CUDA, ή, διαφορετικά, να μειώσουν σημαντικά την παραγωγικότητα ενός ήδη εξοικιωμένου με το πρότυπο προγραμματιστή. Ως λύση, στη παρούσα εργασία, προτείνεται η αυτόματη παραγωγή κώδικα CUDA C, μέσω ενός ήδη πετυχημένου προτύπου για παράλληλο προγραμματισμό και συγκεκριμένα του OpenMP.

3.4 OpenMP και CUDA

Η γενική ιδέα στην οποία βασίζεται η συγκεκριμένη εργασία, είναι η χρήση του προτύπου OpenMP για στόχευση μονάδων GPU που βασίζονται στο μοντέλο CUDA. Η έννοια των συσκευών OpenMP, με την προτεινόμενη λειτουργικότητα από το πρότυπο, μπορεί άμεσα να συμπεριλάβει τις μονάδες GPU. Μέχρι στιγμής, ορισμένοι μεταφραστές και εργασίες έχουν υλοποιήσει λύσεις που αφορούν την αυτόματη παραγωγή κώδικα CUDA μέσω του προτύπου OpenMP. Οι υλοποιήσεις αυτές παρουσιάζονται στη συνέχεια.

3.4.1 Υποστήριξη από Μεταφραστές

Διάφοροι μεταφραστές, εμπορικοί και μη, έχουν συμπεριλάβει, σε διαφορετικά επίπεδα ο καθένας, κάποια υποδομή για παραγωγή κώδικα CUDA μέσω του προτύπου OpenMP. Συγκεκριμένα, ο μεταφραστής gcc έκδοσης 7, επιτρέπει την παραγωγή κώδικα NVPTX [15], για οδηγίες OpenMP. Η NVPTX είναι η μια εικονική μηχανή παράλληλης εκτέλεσης χαμηλού επιπέδου της NVIDIA, που αφορά την αρχιτεκτονική CUDA. Μέσω ενός συνόλου εργαλειών, τα λεγόμενα nvptx-tools, ο μεταφραστής gcc έχει τη δυνατότητα να φορτώσει τον κώδικα PTX ως μια συσκευή CUDA. Κάποιος μπορεί να σκεφτεί τον κώδικα PTX με ένα είδος γλώσσας assembly. Ο οδηγός (driver) της μονάδας γραφικής επεξεργασίας αναλαμβάνει την μετάφραση του σε δυαδικό κώδικα, ο οποίος και τελικά εκτελείται από τους πυρήνες της συσκευής.

Ένα άλλο παράδειγμα μεταφραστή ο οποίος υποστηρίζει τη φόρτωση κώδικα C μέσω του προτύπου OpenMP σε συσκευές τύπου CUDA, είναι ο LLVM. Ο LLVM υποστηρίζει την παραγωγή κώδικα από περιοχές target, με δύο τρόπους. Ο πρώτος αποκαλείται SPMD (Single Program, Multiple Data) και χρησιμοποιεί ένα απλοποιημένο σύνολο λειτουργιών υποστήριξης εκτέλεσης, οι οποίες αυξάνουν τις επιδόσεις

των προγραμμάτων, αφαιρώντας όμως συγκεκριμένες δυνατότητες του OpenMP. Ο δεύτερος τρόπος, αποκαλείται non-SPMD και αποτελεί μια πιο γενική μορφή φόρτωσης, υποστηρίζοντας όλες τις διαθέσιμες λειτουργίες OpenMP. Αξίζει να σημειωθεί, ότι όπως και στην περίπτωση του μεταφραστή gcc, ο LLVM παράγει κώδικα NVPTX και όχι κώδικα που κάνει χρήση της διεπαφής CUDA Driver.

Τέλος, ο μεταφραστής XL της IBM [16], υποστηρίζει πλήρως την έκδοση 4.5 του προτύπου OpenMP, το οποίο συμπεριλαμβάνει τις οδηγίες που πραγματεύεται η παρούσα εργασία. Μέσω εργαλείων της NVIDIA, ο XL έχει τη δυνατότητα μέσω των οδηγιών να φορτώσει κώδικα από περιοχές OpenMP target σε συσκευές CUDA. Ο τρόπος υποστήριξης, παρ' όλα αυτά, παραμένει κρυφός λόγω της εμπορικής φύσης του μεταφραστή.

3.4.2 Ερευνητικές Εργασίες

OpenMP to GPGPU (Lee, Min & Eigenmann)

Στο παρελθόν, έχουν γίνει αρκετές ερευνητικές προσπάθειες που επικεντρώνονται στην χρήση συσκευών CUDA μέσω OpenMP. Στην εργασία [11], οι Lee, Min και Eigenmann αφοσιώθηκαν στον βασικό μετασχηματισμό ενός προγράμματος OpenMP σε ένα πρόγραμμα που βασίζεται σε εντολές CUDA. Η διαδικασία του μετασχηματισμού που προτείνεται, αποτελείται από τρία βήματα:

1. Μετάφραση οδηγιών OpenMP σε κώδικα που χρησιμοποιεί τη διεπαφή CUDA και η αναγνώριση περιοχών kernel μέσα στο πρόγραμμα

2. Εξαγωγή των περιοχών target σε υπορουτίνες και ο μετασχηματισμός του σε CUDA kernels

3. Ανάλυση των κοινόχρηστων δεδομένων τα οποία πρόκειται να προσπελαστούν από τη μονάδα γραφικής επεξεργασίας και εισαγωγή απαραίτητων εντολών CUDA για την εγγραφή και ανάγνωση τους

Το πρώτο βήμα της εργασίας είναι μια διαδικασία η οποία αφορά την μετάφραση των εννοιών που ορίζονται στο πρότυπο OpenMP, σε έννοιες και τεχνικές που ορίζονται από το μοντέλο προγραμματισμού CUDA. Για παράδειγμα, η εντολή *#pragma omp barrier*, παρουσιάζεται ως ένα κομβικό σημείο, το οποίο σηματοδοτεί τον

διαχωρισμό ενός kernel σε δύο υποπεριοχές κώδικα. Οι υποπεριοχές αυτές μετασχηματίζονται περαιτέρω ως kernels. Το δεύτερο βήμα, αφορά την αναγνώριση των περιοχών kernel, οι οποίοι παράγονται κατά το προηγούμενο βήμα. Ο αλγόριθμος που ακολουθείται, αφορά στην ανάλυση ενός συνόλου παράλληλων περιοχών που ανήκουν στο πρόγραμμα χρήστη και η εξαγωγή ενός συνόλου από kernels συσκευής από αυτές. Αφού παραχθούν οι σχετικές περιοχές kernels, τότε αυτοί αναλύονται προκειμένου να εντοπιστούν όλα τα κοινόχρηστα και ιδιωτικά δεδομένα που χρησιμοποιούνται στο εσωτερικό τους. Τα κοινόχρηστα δεδομένα ενός kernel, καταλήγουν στην κοινόχρηστη μνήμη των πυρήνων της μονάδας GPU και είναι προσβάσιμα από κάθε νήμα, ενώ τα ιδιωτικά δεδομένα αποθηκεύονται στις τράπεζες καταχωρητών των νημάτων.

OMPCUDA (Ohshima & Hirasawa)

Η εργασία OMPCUDA [9], των Ohshima και Hirasawa, βασίζεται στην επέκταση ενός ήδη υπάρχοντος μεταφραστή OpenMP, τον Omni. Ο Omni δεν υποστηρίζει τις τελευταίες εκδόσεις του προτύπου OpenMP, αλλά διαθέτει κάποιες λειτουργίες οι οποίες απέβησαν χρήσιμες για την υποστήριξη της διαδικασίας φόρτωσης κώδικα σε συσκευές CUDA. Το πρόγραμμα χρήστη το οποίο περιλαμβάνει οδηγίες OpenMP, αρχικά μετασχηματίζεται σε μια μορφή ενδιάμεσου κώδικα και έπειτα σε κώδικα C. Ο κώδικας C μεταφράζεται μέσω ενός δεύτερου μεταγλωττιστή για να παραχθεί το τελικό εκτελέσιμο. Η εργασία προτείνει την εισαγωγή του εργαλείου OMPCUDA στο στάδιο της μετατροπής ενδιάμεσου κώδικα, ώστε να παράγονται αρχεία με κώδικα συσκευής, τα οποία μεταφράζονται με τη χρήση του nvcc compiler. Το τελικό εκτελέσιμο πλέον συνοδεύουν τα εκτελέσιμα συσκευής τύπου .CUBIN (CUDA binary), τα οποία φορτώνονται στο περιβάλλον της συσκευής CUDA, μέσω των κατάλληλων βιβλιοθηκών. Σημειώνεται ότι η προσπάθεια αυτή αφορά μόνο την παραλληλοποίηση βρόχων for και όχι τη φόρτωση γενικού κώδικα χρήστη μέσω περιοχών target.

Source-to-Source Code Translator (Jaillet & Krajecki)

Στην εργασία [13], οι Jaillet και Krajecki, έχοντας ως βάση τον μεταφραστή OMPi, προτείνουν τον μετασχηματισμό μιας οδηγίας παραλληλοποίησης βρόχου (*#pragma omp parallel for*) σε έναν kernel της CUDA. Η τότε έκδοση του προτύπου OpenMP στην οποία βασίστηκε η εργασία, υπολειπόταν της λειτουργικότητας target. Για την

υποστήριξη του μετασχηματισμού, αρχικά τροποποιήθηκε το τμήμα του λεκτικού και συντακτικού αναλυτή του OMPi, ώστε να υποστηρίζει τις λέξεις-κλειδιά του προτύπου CUDA, οι οποίες αποτελούν επέκταση της γλώσσας C. Έπειτα, η εργασία προτείνει τον μετασχηματισμό της οδηγίας (*#pragma omp parallel for*) σαν μια οδηγία *#pragma omp parallel* που στο εσωτερικό της περιλαμβάνει μια οδηγία *#pragma omp for*. Σημειώνεται ότι η τεχνική αυτή υλοποιούταν ήδη από τον OMPi compiler. Σε επόμενο στάδιο, οι μεταβλητές που εμφανίζονται με τις φράσεις `shared`, `private`, `firstprivate` και `lastprivate`, αντιστοιχίζονται με τον κατάλληλο τρόπο στο περιβάλλον δεδομένων της συσκευής CUDA, μέσω συναρτήσεων που αφορούν τον χειρισμό μνήμης. Τέλος, ο βρόχος στο εσωτερικό της οδηγίας (*#pragma omp parallel for*) μετασχηματίζεται ελαφρώς και εξάγεται σε μια συνάρτηση kernel. Η εργασία χρησιμοποιεί τη διεπαφή CUDA runtime, η οποία θεωρείται υψηλότερου επιπέδου και δεν προσφέρει το ίδιο επίπεδο προγραμματιστικού ελέγχου μιας συσκευής CUDA, συγκριτικά με τη διεπαφή CUDA driver της παρούσας εργασίας.

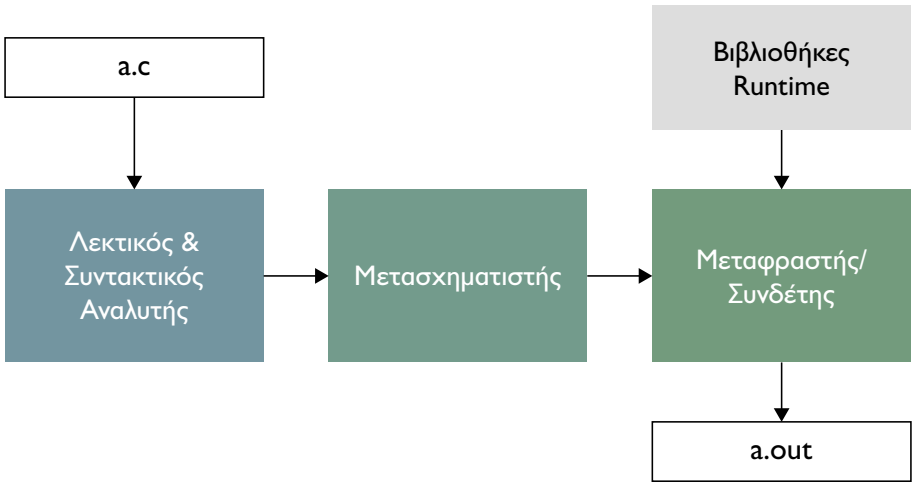
Κεφαλαιο 4

OMPI και Υποδομη Λειτουργικοτητας για Συσκευες

4.1 Δομή του OMPi

Ο μεταφραστής OMPi είναι ένας παραλληλοποιητικός μεταφραστής γλώσσας C για το πρότυπο OpenMP. Η λογική του είναι διαχωρισμένη σε δύο βασικά τμήματα, αυτό του μεταγλωττιστή (compiler) και αυτό της υποστήριξης εκτέλεσης (runtime). Το τμήμα του μεταγλωττιστή χρησιμοποιείται για την μετατροπή πηγαίου κώδικα, που κάνει χρήση οδηγιών OpenMP, σε πολυνηματικό κώδικα C. Στον παραγόμενο κώδικα όλες οι οδηγίες αντικαθιστώνται από κλήσεις σε ρουτίνες του τμήματος υποστήριξης εκτέλεσης.

Ο πολυνηματικός κώδικας μπορεί να χρησιμοποιεί διαφορετικό είδος νημάτων για παράλληλη εκτέλεση. Αξίζει να σημειωθεί ότι οι οντότητες εκτέλεσης θεωρούνται



Σχήμα 4.1: Η διαδικασία της μετάφρασης στον OMP

αφηρημένες, μέχρι τη στιγμή που το σύστημα υποστήριξης εκτέλεσης θα καθορίσει το είδος τους, μέσω των προσφερόμενων βιβλιοθηκών υποστήριξης εκτέλεσης.

4.2 Μετασχηματισμός Πηγαίου σε Πηγαίο Κώδικα (Source-to-source Translation)

Το αρχικό βήμα της μεταγλώττισης περιλαμβάνει την διαδικασία της λεκτικής και συντακτικής ανάλυσης, η οποία είναι κοινή μεταξύ όλων των μεταγλωττιστών. Αρχικά η διαδικασία της λεκτικής ανάλυσης μετατρέπει μια ακολουθία από χαρακτήρες σε μια ακολουθία από λεκτικές μονάδες και στη συνέχεια η συντακτική ανάλυση δημιουργεί ένα συντακτικό δέντρο από τις μονάδες αυτές. Αφού παραχθεί το συντακτικό δέντρο, τότε αυτό διασχίζεται μία φορά ώστε να αντικατασταθεί κάθε οδηγία OpenMP με τις αντίστοιχες κλήσεις του συστήματος υποστήριξης εκτέλεσης. Το αποτέλεσμα των αντικαταστάσεων είναι ένας πηγαίος κώδικας σε C, ο οποίος βασίζεται μόνο σε βιβλιοθήκες υποστήριξης εκτέλεσης του μεταφραστή.

Το τελευταίο βήμα της διαδικασίας της μεταγλώττισης είναι η μετάφραση του παραγόμενου πηγαίου κώδικα C και η σύνδεση των απαραίτητων βιβλιοθηκών υποστήριξης εκτέλεσης με αυτόν, ώστε να παραχθεί το εκτελέσιμο αρχείο. Το τελευταίο αυτό βήμα, επιτυγχάνεται με τη χρήση ενός *μεταφραστή συστήματος* (host compiler). Η ολοκληρωμένη διαδικασία απεικονίζεται στο σχήμα 4.1.

4.3 Το Σύστημα Υποστήριξης Εκτέλεσης OMPi Runtime

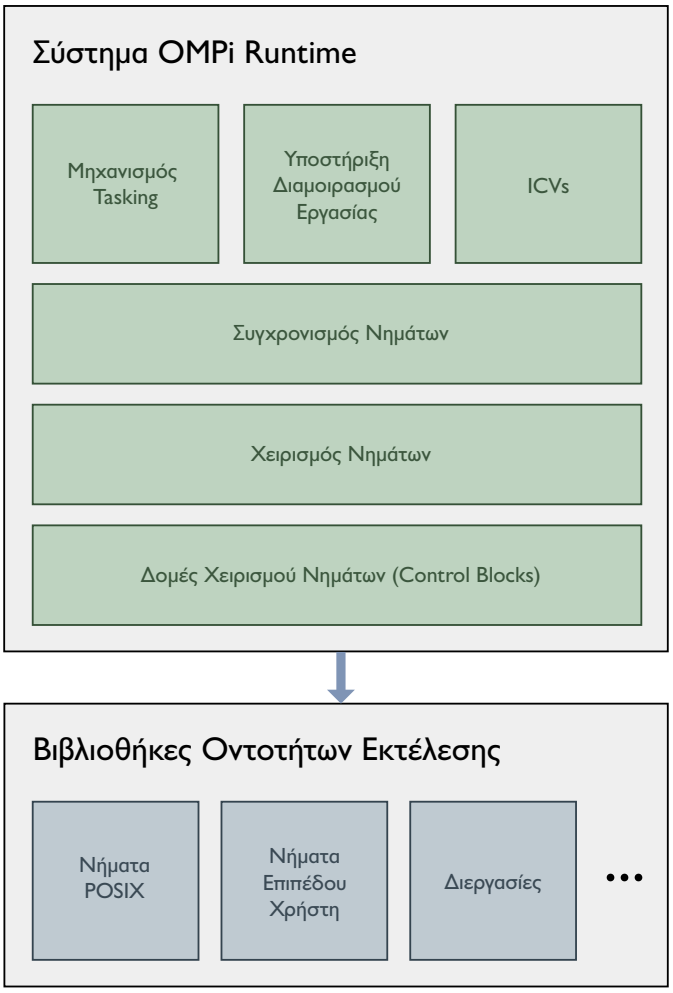
Το σύστημα υποστήριξης εκτέλεσης του OMPi είναι περαιτέρω χωρισμένο στο τμήμα *host*, το οποίο αφορά το κύριο σύστημα, καθώς και το τμήμα *devices* που αφορά τον κώδικα που εκτελείται σε συνοδές συσκευές. Η ενότητα host περιλαμβάνει το συντονισμό των νημάτων εκτέλεσης, την υλοποίηση των οδηγιών OpenMP με τις φράσεις τους, τον χειρισμό μεταβλητών περιβάλλοντος, καθώς επίσης και τις ρουτίνες βιβλιοθήκης υποστήριξης εκτέλεσης όπως ορίζονται στις προδιαγραφές του προτύπου. Επιπλέον, είναι υπεύθυνη για την κλήση όλων των συναρτήσεων που βρίσκονται στο εσωτερικό μιας βιβλιοθήκης συσκευής και αφορούν βασικές λειτουργίες της όπως η αρχικοποίηση της, ο τερματισμός της, η διαχείριση της μνήμης της και η φόρτωση κώδικα σε αυτή.

Οι οντότητες εκτέλεσης συντονίζονται από το σύστημα υποστήριξης εκτέλεσης (ORT), το οποίο στηρίζεται σε αυτές, χωρίς, όμως, να είναι υπεύθυνο για την δημιουργία τους. Συγκεκριμένα, η παραγωγή οντοτήτων εκτέλεσης είναι μια διαδικασία που υλοποιείται σε ξεχωριστές βιβλιοθήκες, τις λεγόμενες *Βιβλιοθήκες Οντοτήτων Εκτέλεσης* (Execution Entity Libraries – EELIBS). Η αρχιτεκτονική αυτή του συστήματος υποστήριξης εκτέλεσης του OMPi είναι μοναδική και modular. Επιτρέπει τη διαφανή χρήση διαφορετικών βιλιοθηκών οντοτήτων, ανάλογα με τις ανάγκες του προγραμματιστή, οι οποίες μπορεί, για παράδειγμα, να βασίζονται σε διεργασίες ή νήματα. Το σχήμα 4.2 διευκολύνει στην κατανόηση της δομής του OMPi.

Η ενότητα devices περιλαμβάνει τις βιβλιοθήκες υποστήριξης εκτέλεσης για τις συσκευές, οι οποίες χωρίζονται στα τμήματα *devpart* και *hostpart*. Το τμήμα hostpart, είναι μια διεπαφή μέσω της οποίας πραγματοποιείται το δεύτερο άκρο επικοινωνίας μεταξύ του κύριου συστήματος και των συσκευών. Το τμήμα devpart προσφέρει την υποδομή για την υποστήριξη οδηγιών και ρουτίνων OpenMP στο εσωτερικό του κώδικα kernel, που φορτώνεται στη συσκευή. Το επίπεδο υποστήριξης εξαρτάται από την εκάστοτε βιβλιοθήκη.

4.3.1 Περιοχές Παραλληλίας στον OMPi

Κατά τη διαδικασία του μετασχηματισμού, οι οδηγίες *#pragma omp parallel* του αρχικού πηγαίου κώδικα μετατρέπονται σε κλήσεις `ort_execute_parallel()`. Η βιβλιοθήκη από την οποία θα κληθεί η ρουτίνα ορίζεται από το σημείο εμφάνισης της οδηγίας. Για παράδειγμα, αν αυτή εμφανίζεται μέσα σε μια περιοχή *target* η οποία



Σχήμα 4.2: Η δομή του OMPi runtime για το σύστημα host

αφορά τη συσκευή A, τότε η βιβλιοθήκη που θα χρησιμοποιηθεί θα είναι αυτή της υποστήριξης εκτέλεσης της A. Το σχήμα 4.3 αποτελεί το αποτέλεσμα του μετασχηματισμού του κώδικα στο σχήμα 2.1, ο οποίος περιέχει μία περιοχή παραλληλίας. Στο συγκεκριμένο παράδειγμα η εκτέλεση λαμβάνει χώρα μόνο στην μεριά του host.

Κατά την εκτέλεση του προγράμματος, η συνάρτηση `main` καλεί τη συνάρτηση `__original_main`, η οποία περιέχει το πρόγραμμα του χρήστη, με μετασχηματισμούς στα σημεία που χρησιμοποιείται οδηγία OpenMP. Παρατηρούμε ότι η οδηγία *#pragma omp parallel* έχει αντικατασταθεί από κλήση στη συνάρτηση `ort_execute_parallel()`, με πρώτο όρισμα την `_thrFunc0_()`. Στον OMPi, η `_thrFunc0_()` είναι η *συνάρτηση νήματος* (thread function), η οποία θα εκτελεστεί παράλληλα από τα νήματα της παράλληλης ομάδας. Στο εσωτερικό της, έχει μετακομίσει ο κώδικας χρήστη, που βρισκόταν εντός της οδηγίας *parallel*. Μιας και η περιοχή παραλληλίας δεν βρισκόταν αρχικά μέσα σε κάποια οδηγία *target*, ο κώδικας θα εκτελεστεί από τα νήματα

```
1  int __original_main(int _argc_ignored, char ** _argv_ignored)
2  {
3    omp_set_num_threads(16);
4    /* (l6) #pragma omp parallel
5   */
6    {
7      ort_execute_parallel(_thrFunc0_, (void *) 0, -1, 0, 0);
8    }
9  }
10
11 static void * _thrFunc0_(void * __arg)
12 {
13   {
14     int thrid = omp_get_thread_num();
15     printf("I am thread %d.\n", thrid);
16   }
17   CANCEL_parallel_6 :
18     ort_taskwait(2);
19   return ((void *) 0);
20 }
```

Σχήμα 4.3: Παράδειγμα μετασχηματισμού περιοχής παραλληλίας στον OMPi

του κύριου συστήματος. Σημειώνεται, επίσης, ότι στο τέλος της συνάρτησης είναι αναγκαία η κλήση στη συνάρτηση `ort_taskwait`, η οποία στο εσωτερικό της χρησιμοποιεί ένα φράγμα συγχρονισμού των νημάτων. Αναφερθήκαμε σε αυτό το φράγμα, στο κεφάλαιο 2, ως *υπονοούμενο φράγμα*.

4.3.2 Υποστήριξη Συσκευών στον OMPi

Μια από τις πιο σημαντικές λειτουργικότητες του OMPi, είναι η υποστήριξη της φόρτωσης δεδομένων και κώδικα σε συσκευές του συστήματος, μέσω των περιοχών target, όπως ορίζονται στο πρότυπο OpenMP. Η φόρτωση αυτή πραγματοποιείται με τη χρήση της διεπαφής hostpart, που αναφέρθηκε προηγουμένως. Η hostpart είναι καλά ορισμένη και διαθέτει όλες τις λειτουργίες για την υποστήριξη περιοχών target, οι οποίες αφορούν την αρχικοποίηση και τερματισμό της συσκευής, την πρόσβαση στη μνήμη της και τη φόρτωση κώδικα. Η διεπαφή περιγράφεται στη συνέχεια λεπτομερώς.

Η πρώτη ενέργεια που πραγματοποιείται όταν συναντάται μια περιοχή target, είναι ο μετασχηματισμός του κώδικα που βρίσκεται στο εσωτερικό της, για την αντικατάσταση περαιτέρω οδηγιών OpenMP, από κώδικα C του μεταφραστή. Αυτή η διαδικασία μετασχηματισμού, σε γενικές γραμμές δεν διαφέρει από αυτή που ακολουθείται γενικότερα από τον OMPi, σε ένα πρόγραμμα που εκτελείται στο σύστημα host. Η μόνη διαφορά έγκειται στο ότι με το πέρας του μετασχηματισμού μιας περιοχής target, ο κώδικας της εξάγεται σε ξεχωριστό αρχείο.

Η εξαγωγή των περιοχών target του προγράμματος σε ξεχωριστό αρχείο, εξασφαλίζει ότι για κάθε περιοχή θα παραχθεί ένας πηγαίος κώδικας με όλες τις απαραίτητες κλήσεις προς τη βιβλιοθήκη συσκευής. Αυτός ο πηγαίος κώδικας, είναι δομημένος με τέτοιο τρόπο ώστε να μπορεί να μεταγλωττιστεί από κάποιον άλλο μεταφραστή που ορίζει η εκάστοτε συσκευή και, τελικά, να παραχθεί ο εκτελέσιμος πυρήνας (kernel) που θα φορτωθεί σε αυτή.

Ενώ υπάρχουν και συσκευές με ίδιο χώρο διευθύνσεων με τον host, στη γενική περίπτωση οι χώροι διευθύνσεων είναι διαφορετικοί. Επομένως απαιτείται αντιστοίχιση από τον έναν χώρο στον άλλο. Επειδή ο τελικός χώρος αποθήκευσης στη συσκευή μπορεί να μην είναι εκ των προτέρων γνωστός, ο OMPi χρησιμοποιεί κάποιες προσωρινές διευθύνσεις που τις αποκαλεί ενδιάμεσες.

Το παράδειγμα του σχήματος 4.4 αφορά το μετασχηματισμό του προγράμμα-

τος 2.2 και αποτελεί το ξεχωριστό αρχείο που παράγεται από τη περιοχή target. Ο κώδικας του χρήστη εξάγεται στη συνάρτηση πυρήνα `_kernelFunc0_`, της οποίας το αναγνωριστικό αυξάνει ανά περιοχή target. Για τα δεδομένα που πρέπει να αντιστοιχιστούν μέσω της διαδικασίας mapping στο περιβάλλον δεδομένων της συσκευής, όπως ο πίνακας x, λαμβάνει χώρα η εξής διαδικασία:

1. Οι διευθύνσεις των δεδομένων του συστήματος host αρχικά μετατρέπονται σε ενδιάμεσες διευθύνσεις. Ο OMPi στο εσωτερικό του, φυλάει όλες αυτές τις ενδιάμεσες διευθύνσεις για μελλοντικές αναγνώσεις και εγγραφές.

2. Στον εσωτερικό του κώδικα συσκευής, οι ενδιάμεσες διευθύνσεις δεδομένων μετατρέπονται σε διευθύνσεις συσκευής, ώστε να γίνει δυνατή η χρήση τους από τον kernel.

Το βήμα 2 είναι εμφανές στις γραμμές 3-6 και 11. Η δομή `_dev_data` περιλαμβάνει αρχικά τις ενδιάμεσες διευθύνσεις, οι οποίες έχουν περάσει από το σύστημα host σαν όρισμα στη συνάρτηση `_kernelFunc0_`. Στη γραμμή 11 παρατηρούμε ότι η συσκευή, πλέον, μετατρέπει την ενδιάμεση διεύθυνση του πίνακα x, σε δική της διεύθυνση.

Ένα ερώτημα που γεννάται αφορά τον τρόπο με τον οποίο το σύστημα host μετατρέπει τις διευθύνσεις του σε ενδιάμεσες και, με ποιον τρόπο μπορεί και ανακτά τα τροποποιημένα δεδομένα από τη συσκευή, για παράδειγμα στις περιπτώσεις όπου υπάρχει αντιστοίχιση *tofrom*. Ο ολοκληρωμένος κώδικας αφήνεται προς μελέτη στο παράρτημα Α.3.

Ο τρόπος με τον οποίο θα μεταγλωττιστεί ένας πηγαίος κώδικας πυρήνα καθορίζεται από ένα ειδικό Makefile που πραγματοποιεί την σύνδεση με μια συγκεκριμένη βιβλιοθήκη συσκευής. Είναι απαραίτητο το πλήθος των Makefiles να ισούται με το πλήθος των εγκατεστημένων βιβλιοθηκών συσκευής. Συνολικά, αν το πλήθος των περιοχών ειναι X και το πλήθος των συσκευών Y, παράγονται $X*Y$ εκτελέσιμα αρχεία.

4.4 Υποδομή Μεταγλωττιστή για Υποστήριξη CUDA

Όπως αναφέρθηκε προηγουμένως, ο μεταφραστής OMPi διαθέτει υποδομή για αρκετές από τις προδιαγραφόμενες λειτουργίες των συσκευών OpenMP. Παρ' όλα

```
1  static void * _kernelFunc0_(void * __arg)
2  {
3      struct __dev_struct {
4          int (* x) [ 16];
5          unsigned long _x_offset;
6      } * _dev_data = (struct __dev_struct *) __arg;
7
8      int i;
9      int (*x) [16] = (int (*) [16]) (devpart_med2dev_addr(_dev_data->x,
10         sizeof(*(_dev_data->x))) - _dev_data->_x_offset);
11
12     /* (l10) #pragma omp target map(tofrom: x) private(i) */
13     {
14         for (i = 0; i < 16; i++)
15             (*x)[i] = i;
16     }
17     return ((void *) 0);
18 }
```

Σχήμα 4.4: Παράδειγμα μετασχηματισμού περιοχής target στον OMPi

αυτά, η λειτουργικότητα που περιγράφηκε στο κεφάλαιο 3 και υποστηρίζεται σε κάποιο βαθμό από τους μοντέρνους μεταγλωττιστές, υπολειπόταν στον OMPi. Συνεπώς, πρώτο βήμα της παρούσας εργασίας, ήταν η υλοποίηση μιας σειράς από οδηγίες και βοηθητικές λειτουργίες στο τμήμα του μεταγλωττιστή. Οι νέες οδηγίες που υποστηρίχθηκαν είναι οι εξής:

- **#pragma omp distribute**. Η οδηγία *distribute* περιγράφηκε στην ενότητα 2.6.1.
- **#pragma omp distribute parallel for**. Η οδηγία *distribute parallel for* περιγράφηκε στην ενότητα 2.6.2.
- **#pragma omp target teams**. Η οδηγία *target teams* αποτελεί την συνδυασμένη (combined) εκδοχή της οδηγίας *target*, η οποία περιέχει στο εσωτερικό της μια μοναδική οδηγία *teams*.
- **#pragma omp target teams distribute**. Η οδηγία *target teams distribute* αποτελεί την συνδυασμένη (combined) εκδοχή της οδηγίας *target*, η οποία περιέχει στο εσωτερικό της μια μοναδική οδηγία *teams distribute*, η οποία δημιουργεί ομάδες νημάτων και διαμοιράζει την εργασία ενός βρόχου στους αρχηγούς της κάθε ομάδας.
- **#pragma omp target teams distribute parallel for**. Η οδηγία *target teams distribute parallel for* αποτελεί την συνδυασμένη (combined) εκδοχή της οδηγίας *target*, η οποία περιέχει στο εσωτερικό της μια μοναδική οδηγία *teams distribute parallel for*. Η συνδυασμένη αυτή οδηγία δημιουργεί ομάδες νημάτων, αρχικά διαμοιράζει την εργασία ενός βρόχου στους αρχηγούς της κάθε ομάδας, με τον τρόπο που ορίζει η οδηγία *distribute*. Κάθε αρχηγός δημιουργεί τη δική του παράλληλη ομάδα και διαμοιράζει το κομμάτι που ανέλαβε στα μέλη αυτής, με τον τρόπο που ορίζει η οδηγία *parallel for*.
- **#pragma omp target parallel**. Η οδηγία *target parallel* αποτελεί την συνδυασμένη (combined) εκδοχή της οδηγίας *target*, η οποία περιέχει στο εσωτερικό της μια μοναδική οδηγία *parallel*. Χρησιμοποιείται για την απευθείας δημιουργία μιας παράλληλης ομάδας, σε μια συνοδή συσκευή.
- **#pragma omp target parallel for**. Η οδηγία *target parallel for* αποτελεί την συνδυασμένη (combined) εκδοχή της οδηγίας *target*, η οποία περιέχει στο εσωτερικό της μια μοναδική οδηγία *parallel for*. Χρησιμοποιείται για την απευθείας

δημιουργία μιας παράλληλης ομάδας και τον διαμοιρασμό εργασίας ενός βρόχου σε αυτή, σε μια συνοδή συσκευή.

4.4.1 Υποστήριξη Γραμματικής

Το τμήμα μεταγλωττιστή του OMPi βασίζεται στο εργαλείο Bison, για την συντακτική ανάλυση των οδηγιών OpenMP που χρησιμοποιούνται στο πρόγραμμα χρήστη. Το Bison λειτουργεί με αρχεία παραμετροποίησης, μέσα στα οποία συνυπάρχουν οι κανόνες γραμματικής μιας γλώσσας προγραμματισμού. Στην περίπτωση του OMPi, όπου η τελική μετάφραση ενός προγράμματος γίνεται από τον μεταφραστή συστήματος, η γραμματική αποτελεί επέκταση της γλώσσας προγραμματισμού C, για τις οδηγίες OpenMP. Για την υποστήριξη των οδηγιών που αφορούν τις ομάδες OpenMP, απαραίτητη ήταν, σε πρώτο στάδιο, η εισαγωγή νέων κανόνων γραμματικής στο αρχείο παραμετροποίησης του Bison. Ένα παράδειγμα γραμματικής αφήνεται στην ενότητα Α για περαιτέρω μελέτη.

4.4.2 Επέκταση του Αφηρημένου Συντακτικού Δέντρου (Abstract Syntax Tree)

Μετά την υποστήριξη στο επίπεδο συντακτικής ανάλυσης, απαραίτητο είναι το στάδιο μετασχηματισμού των οδηγιών σε κώδικα C. Η λειτουργία αυτή του OMPi πραγματοποιείται μέσω του *αφηρημένου συντακτικού δέντρου* (abstract syntax tree). Το συντακτικό δέντρο είναι μια δενδρική απεικόνιση της συντακτικής δομής του πηγαίου κώδικα χρήστη. Η κατασκευή του γίνεται μέσω της συντακτικής ανάλυσης του προγράμματος. Θεωρείται αφηρημένο διότι δεν αναπαριστά κάθε λεπτομέρεια που εμφανίζεται στην πραγματική σύνταξη του προγράμματος, αλλά μόνο τα δομικά στοιχεία της. Το σχήμα 4.7 αναπαριστά το αφηρημένο συντακτικό δέντρο του κώδικα 2.2.

Στο αφηρημένο συντακτικό δέντρο, κάθε οδηγία OpenMP αναπαριστάται ως ένας κόμβος τύπου *OpenMP Statement* (δήλωση OpenMP), με όλες τις απαραίτητες πληροφορίες, όπως ο τύπος της οδηγίας και το τμήμα κώδικα που βρίσκεται στο εσωτερικό της. Ο μετασχηματισμός των οδηγιών, στη βάση του πραγματοποιείται με το μετασχηματισμό τέτοιου είδους κόμβων. Αρχικά, το δέντρο διασχίζεται πλήρως και όλοι οι κόμβοι που αντιστοιχούν σε ένα OpenMP statement, αντικαθίστανται από ένα νέο υποδέντρο, το οποίο αποτελείται από νέους κόμβους-statements. Οι

```
#include "omp.h"
int main() {
    int x[16];
    #pragma omp target parallel map(tofrom: x) num_threads(16)
    {
        int my_id = omp_get_thread_num();
        x[my_id] = my_id;
    }
}
```

Σχήμα 4.5: Παράδειγμα συνδυασμένης οδηγίας στον OMPi

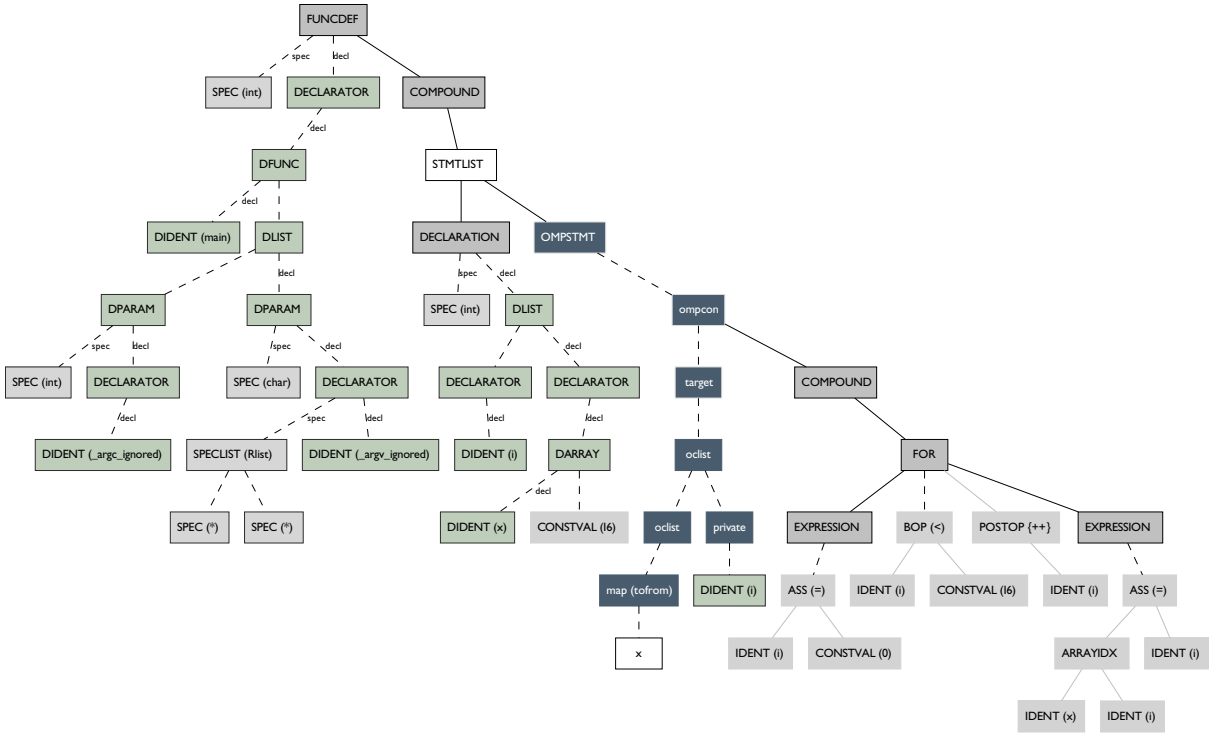
```
#include "omp.h"
int main() {
    int x[16];
    #pragma omp target map(tofrom:x) {
        #pragma omp parallel num_threads(16) {
            int my_id = omp_get_thread_num();
            x[my_id] = my_id;
        }
    }
}
```

Σχήμα 4.6: Παράδειγμα αρχικού μετασχηματισμού συνδυασμένης οδηγίας στον OMPi

νέοι κόμβοι μπορούν επιπλέον να αποτελούν και οι ίδιοι OpenMP statements, ή όπως συνηθίζεται, statements της γλώσσας C. Τέλος, για την διευκόλυνση των προγραμματιστών και την καλύτερη οργάνωση του μεταφραστή, κάθε μετασχηματισμός OpenMP statement συμβαίνει στο εσωτερικό μιας συνάρτησης που δημιουργήθηκε για αυτό το σκοπό και περιέχει το πρόθεμα *xform* (transform) στο όνομά της.

Για την υποστήριξη των νέων οδηγιών, τροποποιήθηκε ο πηγαίος κώδικας της διάσχισης του συντακτικού δέντρου. Συγκεκριμένα, εισήχθησαν οι νέοι τύποι OpenMP statements, για να είναι δυνατή η διάκρισή τους από τα υπόλοιπα. Επιπλέον, υλοποιήθηκαν νέες συναρτήσεις χειρισμού των οδηγιών. Η λογική που ακολουθήθηκε για τον χειρισμό των οδηγιών είναι η εξής:

- Εάν πρόκειται για combined οδηγία η οποία αποτελεί τον συνδυασμό δύο



Σχήμα 4.7: Παράδειγμα αφηρημένου συντακτικού δέντρου

μεμονωμένων οδηγιών, όπως για παράδειγμα η οδηγία *distribute parallel for*, τότε γίνεται ο διαχωρισμός των δύο επιμέρους οδηγιών με τον εξής τρόπο (σχήματα 4.5 & 4.6):

1. Οι φράσεις που συνοδεύουν την combined οδηγία, μοιράζονται στις οδηγίες με τέτοιο τρόπο ώστε η κάθε οδηγία να λαμβάνει τις φράσεις που υποστηρίζονται από αυτή. Εάν μια φράση υποστηρίζεται και από τις δύο οδηγίες, τότε ακολουθούνται οι περιορισμοί του προτύπου OpenMP και είτε δίνεται και στις δύο οδηγίες, είτε μόνο στην πρώτη.
2. Ο τύπος της οδηγίας αλλάζει σε αυτόν της πρώτης (εδώ: *distribute*)
3. Το εσωτερικό της αρχικής οδηγίας (σώμα) περικλείεται από ένα νέο OpenMP statement, που έχει ως τύπο αυτόν της δεύτερης οδηγίας (εδώ: *parallel for*)
4. Τροποποιούνται οι πληροφορίες που αφορούν την γραμμή της οδηγίας στο πρόγραμμα, για την παραγωγή σωστών μηνυμάτων σφάλματος
5. Η νέα οδηγία δίνεται προς μετασχηματισμό, με την πιο εσωτερική να μετασχηματίζεται πρώτη και την πιο εξωτερική να μετασχηματίζεται τελευταία

- Εάν πρόκειται για μεμονωμένη οδηγία, όπως για παράδειγμα η οδηγία *distribute*, τότε πρώτα γίνεται μετασχηματισμός στο εσωτερικό της. Το τμήμα κώδικα που περιλαμβάνεται στην οδηγία, μπορεί να περιέχει περαιτέρω OpenMP statements, τα οποία δημιουργούνται από περιπτώσεις όπως η προηγούμενη. Όταν ολοκληρωθεί ο μετασχηματισμός του εσωτερικού κώδικα και αυτός πλέον αποτελείται μόνο από απλά C statements, τότε καλείται η αντίστοιχη συνάρτηση *xform*, η οποία τροποποιεί τον κόμβο του συντακτικού δέντρου.

Στη συνέχεια περιγράφονται οι τεχνικές μετασχηματισμού για καθεμία από τις νεοσύστατες οδηγίες.

4.5 Μετασχηματισμοί Νέων Οδηγιών

4.5.1 Οδηγία Distribute

Για τον μετασχηματισμό της οδηγίας *distribute*, χρησιμοποιήθηκε λογική παρόμοια με αυτή του χειρισμού μιας οδηγίας *for*. Τα βήματα μετασχηματισμού είναι τα εξής:

1. Ο βρόχος στο εσωτερικό της οδηγίας αναλύεται, ώστε να αποθηκευτεί όλη η πληροφορία που αφορά την αρχική και τελική τιμή του μετρητή, το βήμα του, καθώς και τον τελεστή που χρησιμοποιείται για την μεταβολή του βήματος

2. Χειρίζονται οι μεταβλητές που δηλώθηκαν ως firstprivate, lastprivate ή private, με τις αντίστοιχες φράσεις στην οδηγία

3. Παράγονται οι αρχικές δηλώσεις των ειδικών βοηθητικών μεταβλητών *dist_niters_*, *dist_fiter_* και *dist_liter_*, στις οποίες αποθηκεύεται το συνολικό πλήθος επαναλήψεων του βρόχου, λαμβάνοντας υπ' όψην το βήμα του βρόχου και την αρχική/τελική τιμή του μετρητή

4. Γίνεται έλεγχος ύπαρξης της φράσης *dist_schedule*:

   (α) Στην περίπτωση που διαπιστωθεί η χρήση της με συγκεκριμένο μέγεθος κόκκου (chunk size), τότε παράγεται ένας νέος βρόχος ο οποίος διαμοιράζει με στατική χρονοδρολόγηση τις επαναλήψεις του αρχικού βρόχου ανά τους αρχηγούς των ομάδων, βάσει του αναγνωριστικού τους. Εάν η ομάδα δεν υπάρχει, δηλαδή δεν βρισκόμαστε σε περιοχή ομάδων teams,

```
int main() {
    int x[16];
    #pragma omp distribute
        for (i = 0; i < 16; i++)
            x[i] = i;
}
```

Σχήμα 4.8: Παράδειγμα περιοχής distribute στον OMPi (χωρίς chunk size)

τότε η τιμή αυτή ισούται με το 0 και ουσιαστικά η χρονοδρομολόγηση αφορά ένα μόνο νήμα (σχήματα A.1 & A.2 του παραρτήματος A)

(β) Σε οποιαδήποτε άλλη περίπτωση, τότε αντί ενός νέου βρόχου, χρησιμοποιείται η συνάρτηση `ort_get_distribute_chunk()`, η οποία επιστρέφει στατικά ένα τμήμα του βρόχου επαναλήψεων. Σημειώνεται ότι σε αυτή τη περίπτωση κάθε νήμα-αρχηγός μιας ομάδας θα λάβει μόνο ένα κομμάτι, τη στιγμή που στην προηγούμενη περίπτωση ο διαμοιρασμός συνέβαινε επαναληπτικά (σχήματα 4.8 & 4.9)

5. Γίνεται έλεγχος ύπαρξης της φράσης *collapse*. Εάν διαπιστωθεί η χρήση της, τότε ελέγχεται η τιμή N, η οποία αφορά το πλήθος των εμφωλευμένων επαναληπτικών βρόχων προς συγχώνευση. Οι εμφωλευμένοι βρόχοι εξετάζονται και άμεσα διαπιστώνεται εάν το πλήθος τους ισούται τουλάχιστον με το πλήθος N. Στην περίπτωση όπου η ισότητα ισχύει, στο εσωτερικό του παραγόμενου κώδικα του προηγούμενου βήματος, εισάγεται μια συγχωνευμένη μορφή των N βρόχων, στην οποία ο αριθμός επαναλήψεων και οι τιμές των είναι ακριβώς ίδιες, με τη μόνη διαφορά να έγκειται στην γραμμικότητα του νέου βρόχου.

Με το πέρας των βημάτων, έχουμε την ολοκληρωμένη μορφή του μετασχηματισμού και ο ίδιος αντικαθιστά τον αρχικό κώδικα. Η ενέργεια αυτή λαμβάνει χώρα σε κάθε υλοποιημένη οδηγία, συνεπώς θα θεωρείται αυτονόητη από εδώ και στο εξής.

4.5.2 Οδηγία Target Teams

Ο μετασχηματισμός της οδηγίας *target teams* βασίζεται στον μετασχηματισμό τύπου *target*. Συγκεκριμένα, η ιδέα αφορά στην υλοποίηση της οδηγίας αυτής ως μια οδηγία

```
1  int __original_main(int _argc_ignored, char ** _argv_ignored)
2  {
3      int i;
4
5      unsigned long dist_niters_ = 0, dist_iter_ = 0, dist_fiter_,
6          dist_liter_ = 0;
7
8      dist_niters_ = (long) (((16) >= (0)) ? ((16) - (0)) : 0);
9      if (ort_get_distribute_chunk(dist_niters_, &dist_fiter_, &dist_liter_))
10     {
11         for (dist_iter_ = dist_fiter_,
12             i = (0) + dist_fiter_ * 1;
13             dist_iter_ < dist_liter_;
14             dist_iter_++, i += 1)
15
16         x[i] = i;
17     }
18 }
```

Σχήμα 4.9: Παράδειγμα μετασχηματισμού περιοχής distribute στον OMPi (χωρίς chunk size)

target, η οποία παραμετροποιείται ανάλογα με τις φράσεις που αφορούν τις ομάδες. Στις συσκευές όπου δεν είναι δυνατή η δυναμική δημιουργία ομάδων στο εσωτερικό του kernel, η συνδυασμένη αυτή οδηγία αποβαίνει χρήσιμη διότι επιτρέπει στον μεταφραστή να γνωρίζει στατικά και εκ των προτέρων ότι:

- Συμβαίνει μόνο μία δημιουργία ομάδων, χωρίς να προηγείται ή να έπειται άλλος κώδικας
- Το πλήθος των ομάδων είναι προκαθορισμένο, καθώς επίσης και άλλες πληροφορίες όπως το όριο των νημάτων ανά ομάδα (thread limit)

Με το σκεπτικό αυτό, τροποποιήθηκε κατάλληλα το τμήμα υποστήριξης εκτέλεσης του OMPi, ώστε να χρησιμοποιεί τις παραπάνω πληροφορίες κατά τη φόρτωση ενός kernel στη συσκευή.

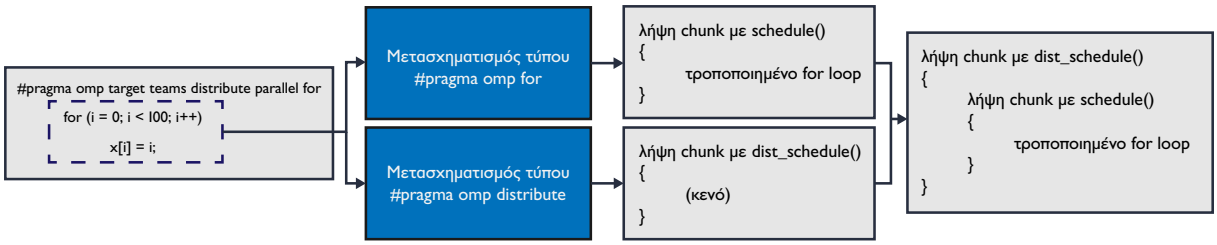
4.5.3 Οδηγία Target Teams Distribute

Σημαντική βάση για την φόρτωση κώδικα σε επιταχυντές γραφικών αποτέλεσε και η υλοποίηση της οδηγίας *target teams distribute*. Συγκεκριμένα, ο μετασχηματισμός βασίζεται σε αυτόν της οδηγίας target teams, με τη μόνη διαφορά ότι ο εσωτερικός βρόχος μετασχηματίζεται βάσει των βημάτων που ακολουθούνται στον μετασχηματισμό της οδηγίας distribute. Όπως και προηγουμένως, ο μεταφραστής έχει γνώση όλων των χαρακτηριστικών των ομάδων που θα δημιουργηθούν στο εσωτερικό ενός πυρήνα, συνεπώς έχει τη δυνατότητα να παράγει στατικό κώδικα, που λειτουργεί σε συσκευές δίχως δυνατότητα υποστήριξης δυναμικού παραλληλισμού.

4.5.4 Οδηγία Target Teams Distribute Parallel For

Συνδυάζοντας όλες τις προηγούμενες υλοποιήσεις μετασχηματισμών, υποστηρίχθηκε η οδηγία *target teams distribute parallel for*. Συγκεκριμένα, η οδηγία διαχωρίζεται με την λογική που περιγράφηκε στην ενότητα 4.5, στις οδηγίες *target teams* και *distribute parallel for*. Αρχικά μετασχηματίζεται το σώμα της οδηγίας, σαν αυτό να άνηκε στην οδηγία *distribute parallel for* και έπειτα ο παραγόμενος κώδικας φορτώνεται στη συσκευή με τη λειτουργικότητα που ορίζει η οδηγία *target teams*.

Για την υποστήριξη της οδηγίας *distribute parallel for*, προχωρήσαμε σε επέκταση της αρχικής οδηγίας *distribute*:



Σχήμα 4.10: Μετασχηματισμός distribute parallel for στο εσωτερικό ενός target teams

1. Ο κόμβος συντακτικού δέντρου που αφορά το εσωτερικό της οδηγίας, αντιγράφεται σε έναν νέο, στον οποίο εκχωρούνται όλες οι φράσεις που χρησιμοποιούνται με την *target teams distribute parallel for*

2. Ο νέος κόμβος μετασχηματίζεται ακριβώς όπως θα μετασχηματιζόταν ένας κόμβος τύπου OpenMP *for*, με ορισμένες διαφοροποιήσεις οι οποίες είναι απαραίτητες για την συνεργασία με την οδηγία *distribute*, που αφορούν κυρίως τη μετονομασία μεταβλητών ελέγχου και τροποποίηση κομβικών σημείων για το χειρισμό της φράσης *collapse*

3. Ο παραγόμενος κώδικας μετασχηματίζεται περαιτέρω, σαν να βρισκόταν στο εσωτερικό μιας οδηγίας *distribute*

Το σχήμα 4.10 απεικονίζει τα στάδια μετασχηματισμού. Ο κώδικας του δεύτερου βήματος που τοποθετείται στο εσωτερικό της οδηγίας διαμοιρασμού, εκτελείται από κάθε νήμα που ανήκει σε μια ομάδα. Στην πραγματικότητα, δεν υπάρχει διαχωρισμός αρχηγού νήματος και νήματος-μέλους. Κάθε νήμα εκτελεί τον ίδιο παραγόμενο κώδικα. Αρχικά, το νήμα λαμβάνει το τμήμα του βρόχου επανάληψης με τον τρόπο που ορίζει η οδηγία *distribute*, για τους αρχηγούς των ομάδων και έπειτα από το τμήμα αυτό λαμβάνει ένα υποτμήμα με τον τρόπο που ορίζει η οδηγία *parallel for*. Η απόφαση για την υλοποίηση αυτή λήφθηκε λαμβάνοντας υπ' όψην τις συσκευές που δεν υποστηρίζουν δυναμικό παραλληλισμό και άρα δυναμική δημιουργία ομάδων και περιοχών παραλληλίας.

4.5.5 Οδηγίες Target Parallel & Target Parallel for

Η οδηγία *target parallel* βασίζεται στο μετασχηματισμό τύπου *target*, με τη μόνη διαφορά να έγκειται στην αποθήκευση του πλήθους των νημάτων της περιοχής παραλληλίας (num_threads), η οποία θα εκτελεστεί στη συσκευή. Η πληροφορία για

το πλήθος νημάτων περνιέται κατά τη φόρτωση του kernel στη βιβλιοθήκη της συσκευής.

Με επέκταση της οδηγίας *target parallel*, στην εργασία αυτή υλοποιήθηκε επιπλέον η οδηγία *target parallel for*, η οποία χρησιμοποιείται για την απευθείας παραλληλοποίηση με διαμοιρασμό της εργασίας του βρόχου σε μία συσκευή. Η προσθήκη, σε σύγκριση με τον μετασχηματισμό της οδηγίας *target parallel*, αφορά στον αρχικό μετασχηματισμό του βρόχου με τον τρόπο που ορίζει η οδηγία *for*.

4.6 Υλοποίηση Βοηθητικών Λειτουργιών

Εκτός από την υποστήριξη σε επίπεδο συντακτικής ανάλυσης και αφηρημένου συντακτικού δέντρου, στην παρούσα εργασία υλοποιήθηκαν διάφορες βοηθητικές λειτουργίες, που διευκολύνουν την μελλοντική εισαγωγή οδηγιών από έναν προγραμματιστή του OMPi.

4.6.1 Η Υποδομή ClauseRules

Στο πρότυπο OpenMP, οι συνδυασμένες οδηγίες συχνά υπόκεινται σε περιορισμούς, όσον αφορά τη χρήση των φράσεων που επιτρέπεται να τις συνοδεύσουν. Συνήθως, μια οδηγία OpenMP που αποτελεί συνδυασμένη μορφή δύο άλλων, θα επιτρέψει τη χρήση όλων των φράσεων που υποστηρίζονται από τις εκάστοτε οδηγίες, με εξαίρεση ορισμένες φράσεις που μπορούν να εμφανιστούν μία φορά, ή καθόλου. Στο σημείο της συντακτικής ανάλυσης, ο μεταφραστής OMPi δεν διαθέτει αρκετές πληροφορίες ώστε να αποφασίσει εάν θα επιτραπεί η χρήση φράσεων που υπόκεινται σε περιορισμούς. Έτσι, η εργασία της επικύρωσης φράσεων, αφήνεται στο σημείο της διάσχισης του πλήρους συντακτικού δέντρου του προγράμματος, όπου έχει αποθηκευτεί όλη η πληροφορία για τον πηγαίο κώδικα του χρήστη.

Επιπλέον, ενω μέχρι την έκδοση 3.1 του OpenMP υπήρχαν 2 μόλις συνδυασμένες οδηγίες, από την έκδοση 4.0 και μετά αυξήθηκε σημαντικά το πλήθος των συνδυασμένων οδηγιών και των περιπτώσεων που έπρεπε κανείς να εξετάσει. Προκειμένου να διευκολυνθεί και να συστηματοποιηθεί ο έλεγχος φράσεων, σχεδιάστηκε ο μηχανισμός *ClauseRules*. Ο μηχανισμός *ClauseRules* αποτελείται από μία μικρή μεταγλώσσα παραμετροποίησης, μέσω της οποίας ο προγραμματιστής του OMPi έχει τη δυνατότητα να ορίσει τις υποστηριζόμενες φράσεις για όλες τις δυνατές (και

μελλοντικές) οδηγίες του OpenMP, που υποστηρίζονται από τον OMPi.

Η σύνταξη της μεταγλώσσας *ClauseRules* είναι αρκετά απλή και γίνεται μέσω μακροεντολών προεπεξεργαστή και συμβολοσειρών. Αρχικά, ο προγραμματιστής ορίζει μια ομάδα κανόνων με κώδικα C, η οποία περιέχει κανόνες για παρόμοιες οδηγίες. Κατ' ελάχιστον, πρέπει να υπάρχει μία ομάδα κανόνων, ενώ μπορούν να ορισθούν όσες κρίνεται απαραίτητο. Στην ομάδα κανόνων εισάγονται ζεύγη κλειδιού-τιμής, με το κλειδί να αναπαριστά μια οδηγία και, την τιμή, η συμβολοσειρά με τους κανόνες φράσεων για την οδηγία-κλειδί, οι οποίοι συγγράφονται ακολουθώντας την εξής σύνταξη:

φράση [(*τροποποιητής*[, *τροποποιητής*...])][: *αριθμός* | *] [| *νέος κανόνας*...]

Με τη παραπάνω σύνταξη, ο προγραμματιστής έχει τη δυνατότητα να ορίσει το πλήθος φορών που μπορεί να εμφανιστεί μία φράση σε μια οδηγία, το οποίο μπορεί να είναι συγκεκριμένος αριθμός ή άπειρο, καθώς επίσης και τους τροποποιητές (modifiers) που επιτρέπεται να οριστούν στο εσωτερικό της φράσης.

Η υποδομή *ClauseRules* αντικαθιστά την έως τώρα ενσωματωμένη λογική επικύρωσης με μία κλήση σε συνάρτηση η οποία απαντά αν μια φράση υποστηρίζεται από μία οδηγία, υπ' όψην των περιορισμών. Η απάντηση δίνεται σε χρόνο O(1), διότι οι κανόνες αναλύονται και αποθηκεύονται κατά την μεταγλώττιση του προγράμματος σε έναν διδιάστατο πίνακα, στον οποίο φυλάγεται ο βαθμός υποστήριξης κάθε δυνατής φράσης για κάθε δυνατή οδηγία.

4.6.2 Η Συνένωση Εμφωλευμένων Οδηγιών OpenMP

Τα προγράμματα που κάνουν χρήση του προτύπου OpenMP, συχνά αποτελούνται από εμφωλευμένες οδηγίες, για τις οποίες το πρότυπο ήδη υποστηρίζει μια συνδυασμένη μορφή. Οι συνδυασμένες οδηγίες, διευκολύνουν τον μετασχηματισμό κώδικα που προορίζεται για συσκευές, διότι όλη η πληροφορία για τις φράσεις είναι γνωστή εκ των προτέρων. Επιπλέον, ορισμένες συσκευές διαθέτουν μοντέλα εκτέλεσης που δεν επιτρέπουν στο εσωτερικό τμήμα κώδικα μιας οδηγίας να συνυπάρχει δεύτερη οδηγία μαζί με άλλα τμήματα κώδικα C.

Προκειμένου να μην αποτρέψουμε το μετασχηματισμό όταν το δομημένο τμήμα κώδικα μίας οδηγίας OpenMP περιλαμβάνει μόνο **μία**, δεύτερη, οδηγία OpenMP και τίποτε άλλο, στην εργασία αυτή προτείνεται η ιδέα της *συνένωσης* (merge), δηλαδή η μετατροπή των εμφωλευμένων οδηγιών σε μία ισοδύναμη συνδυασμένη οδηγία. Η

συνένωση είναι μία διαδικασία η οποία λαμβάνει χώρα στο σημείο μετασχηματισμού μιας εντολής. Συγκεκριμένα, για την οδηγία προς μετασχηματισμό, ελέγχεται το εσωτερικό της, και εάν αυτό αποτελείται από οδηγία OpenMP που περικλείεται από άγκιστρα, ή αποτελείται άμεσα από μία οδηγία OpenMP, τότε:

1. Αποφασίζεται μέσω μίας νεοσύστατης υποδομής, της *MergeRules*, ο τύπος της συνδυασμένης οδηγίας που θα προκύψει από την συνένωση της εξωτερικής με την εσωτερική οδηγία. Η *MergeRules* λειτουργεί παρόμοια με την υποδομή *ClauseRules*· ο προγραμματιστής του OMPi εισάγει εγγραφές, στις οποίες ορίζεται το αποτέλεσμα της συνένωσης δύο οδηγιών.

2. Παράγεται ένα OpenMP statement με τον τύπο που αποφασίστηκε στο προηγούμενο βήμα

3. Αντιγράφονται οι επιμέρους φράσεις των δύο οδηγιών στη νέα συνδυασμένη μορφή, τηρώντας τους περιορισμούς της δομής *ClauseRules*

4. Το σώμα της εσωτερικής οδηγίας αντιγράφεται στο σώμα της νέας οδηγίας

5. Η συνδυασμένη οδηγία μετασχηματίζεται με τον τρόπο που παρουσιάστηκε στην ενότητα 4.4.2. Σημειώνεται ότι ο μεταφραστής OMPi γνωρίζει και δεν ακολουθεί την διαδικασία διαχωρισμού της συνδυασμένης μορφής, όταν έχει προηγηθεί η διαδικασία της συνένωσης.

Κεφαλαιο 5

Η Συσκευη CUDADEV

5.1 Εισαγωγή

Μέχρι πρότινος, ο μεταφραστής OMPi είχε τη δυνατότητα να επικοινωνεί με άλλου είδους συσκευές, όπως ο επιταχυντής Epiphany [17], υπολειπόταν όμως μιας διεπαφής η οποία θα του επέτρεπε να πραγματοποιήσει τη φόρτωση κώδικα και δεδομένων σε μονάδες GPU. Συνεπώς, δεύτερο και εξίσου σημαντικό τμήμα των υλοποιήσεων της παρούσας εργασίας, είναι η κατασκευή της διεπαφής αυτής στο εσωτερικό του OMPi runtime. Ονομάσαμε τη διεπαφή αυτή *CUDADEV* (CUDA Device). Η CUDADEV αποτελείται, όπως οι υπόλοιπες συσκευές, από τα τμήματα *hostpart* και *devpart*, τα οποία αναφέρθηκαν στην ενότητα 4.3.2. Για την καλύτερη κατανόηση της δομής της, αρχικά παρουσιάζεται η οργάνωση της διεπαφής hostpart και έπειτα ο τρόπος υλοποίησής της για την συσκευή CUDADEV. Έπειτα, επεξηγούνται οι λειτουργίες που προσφέρονται από τη βιβλιοθήκη συσκευής devpart και η συνολική διαδικασία μεταγλώττισης στον μεταφραστή OMPi, όταν χρησιμοποιείται η παρούσα συσκευή.

5.2 Η Διεπαφή hostpart

Όπως περιγράψαμε στην ενότητα 4.3, στον OMPi, η επικοινωνία του κύριου συστήματος (host) με τις συσκευές πραγματοποιείται με τη χρήση της διεπαφής hostpart, η οποία υλοποιείται με διαφορετικό τρόπο, ανά συσκευή OpenMP. Η συσκευή host, όταν συναντήσει οδηγία target, κάνει χρήση της διεπαφής για να προετοιμάσει τη συσκευή, αρχικοποιώντας την και ορίζοντας κοινόχρηστες δομές που διευκολύνουν την επικοινωνία. Τελικά, αποστέλλει κώδικα ή/και δεδομένα, τα οποία αποτελούν το αρχείο πυρήνα και τερματίζει τη συσκευή στο πέρας της περιοχής target. Η υλοποίηση της hostpart για τη συσκευή CUDADEV, βασίζεται στη προγραμματιστική διεπαφή CUDA Driver [18]. Το CUDA Driver API αποτελεί τη χαμηλότερου επιπέδου μορφή του CUDA Runtime API, προσφέροντας καλύτερο έλεγχο στη διαχείριση μιας συσκευής CUDA. Οι επόμενες ενότητες περιγράφουν τις λειτουργίες της διεπαφής hostpart και τον τρόπο που συνεργάζεται με τη διεπαφή CUDA Driver.

5.2.1 Αρχικοποίηση και Τερματισμός Συσκευής

Για την αρχικοποίηση της συσκευής και την προετοιμασία της προς χρήση από το κύριο σύστημα, η διεπαφή hostpart ορίζει την συνάρτηση `hm_initialize`. Η `hm_initialize` αρχικά ανακαλύπτει όλες τις εγκατεστημένες συσκευές CUDA στο σύστημα, για τις οποίες αποθηκεύει σημαντικές πληροφορίες όπως η έκδοση CUDA, οι προδιαγραφές της συσκευής, το όνομά της κ.ο.κ. Επιπλέον, αρχικοποιούνται στην τιμή 0 διάφοροι μετρητές που αφορούν το πλήθος των δεσμεύσεων μνήμης και των φορτώσεων από το κύριο σύστημα. Τέλος, πραγματοποιείται η αρχικοποίηση του *CUDA context*, το οποίο αποτελεί την εσωτερική κατάσταση (internal state) της συσκευής CUDA. Ένα CUDA context είναι απαραίτητο προκειμένου να ξεκινήσει οποιουδήποτε είδους συναλλαγή με τη συσκευή CUDA. Σημειώνεται ότι μιας και περισσότερα από ένα νήματα host μπορούν να χρησιμοποιήσουν τη συσκευή CUDADEV, για παράδειγμα σε περιοχές παραλληλίας που αποτελούνται από περιοχές target, κάθε νήμα θα πρέπει να διαθέτει το δικό του *CUDA context*.

Ο τερματισμός της συσκευής πραγματοποιείται μέσω μιας κλήσης του κύριου συστήματος στην εντολή `hm_finalize`. Σε αυτό το σημείο, η κατάσταση της συσκευής CUDADEV καταστρέφεται ολοκληρωτικά και η συσκευή τερματίζει τη λειτουργία της. Προκειμένου να ξαναχρησιμοποιηθεί ξανά μια συσκευή CUDADEV για την οποία έχει κληθεί η `hm_finalize`, είναι απαραίτητο να αρχικοποιηθεί εκ νέου.

5.2.2 Διαχείριση Μνήμης Συσκευής

Η διαχείριση της μνήμης της συσκευής CUDADEV αποτελείται από ένα σύνολο έξι βασικών λειτουργιών:

- **Δέσμευση (allocation)**. Η δέσμευση μνήμης στη συσκευή από το κύριο σύστημα πραγματοποιείται με τη συνάρτηση `hm_dev_alloc()`. Η `hm_dev_alloc()` δέχεται ως όρισμα τον αριθμό των bytes που πρόκειται να δεσμευθούν στη συσκευή. Η συνάρτηση αναλαμβάνει να δημιουργήσει και να επιστρέψει έναν *δείκτη συσκευής CUDA* (CUDA Device Pointer), ο οποίος θα δείχνει στον νέο χώρο μνήμης. Η δέσμευση μνήμης πραγματοποιείται με κλήση στη συνάρτηση `cuMemAlloc()` του CUDA Driver API.

- **Αποδέσμευση (deallocation)**. Η αποδέσμευση μνήμης υλοποιείται από τη συνάρτηση `hm_dev_free()`, η οποία δέχεται ως όρισμα έναν δείκτη CUDA, ο οποίος έχει επιστραφεί προηγουμένως από την συνάρτηση `hm_dev_alloc()`. Η αποδέσμευση μνήμης πραγματοποιείται με κλήση στη συνάρτηση `cuMemFree()` του CUDA Driver API.

- **Ανάγνωση (reading)**. Η ανάγνωση μνήμης της συσκευής από το κύριο σύστημα είναι δυνατή μέσω της συνάρτησης `hm_fromdev()`. Στο εσωτερικό της συνάρτησης, πραγματοποιείται αντιγραφή ενός χώρου μνήμης συσκευής, μεγέθους ορισμένων bytes, στο χώρο μνήμης του κυρίου συστήματος. Συγκεκριμένα, η ανάγνωση πραγματοποιείται με κλήση στη συνάρτηση `cuMemcpyDtoH()` του CUDA Driver API.

- **Εγγραφή (writing)**. Η εγγραφή μνήμης στης συσκευή από το κύριο σύστημα είναι δυνατή μέσω της συνάρτησης `hm_todev()`, η οποία αναλαμβάνει να αντιγράψει έναν χώρο μνήμης συγκεκριμένου μεγέθους, στο χώρο μνήμης της συσκευής. Η εγγραφή στη μνήμη πραγματοποιείται με κλήση στη συνάρτηση `cuMemcpyHtoD()` του CUDA Driver API.

5.3 Φόρτωση Κώδικα στη Συσκευή

Βασική λειτουργία της διεπαφής hostpart είναι η φόρτωση κώδικα από το κύριο σύστημα στη συσκευή. Η ενέργεια αυτή πραγματοποιείται μέσω της συνάρτησης

`hm_offload()`, η οποία και αποτελεί τον κορμό της λειτουργικότητας της οδηγίας target. Μέσα από την `hm_offload()`, 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. Τέτοιες παράμετροι αφορούν το πλήθος των νημάτων που θα πρέπει να δημιουργηθούν στη συσκευή, για την εκτέλεση του kernel. Τέλος, για τα δεδομένα που θα χρησιμοποιηθούν από τον kernel, οι δομές `dev_data` και `decl_data` περιλαμβάνουν δείκτες, οι οποίοι έχουν επιστραφεί προηγουμένως από τη συνάρτηση `hm_dev_alloc()`. Οι δύο δομές δίνονται ως ορίσματα στην `hm_offload()`. Η πρώτη περιλαμβάνει τα δεδομένα που χρησιμοποιούνται με τη διαδικασία mapping, στην περιοχή target, ενώ η δεύτερη τα δεδομένα που δηλώθηκαν μέσω της οδηγίας *#pragma omp declare target*. Τα τρία στάδια της φόρτωσης κώδικα περιγράφονται στις επόμενες υποενότητες.

5.3.1 Δημιουργία Kernel

Το πρώτο στάδιο είναι η δημιουργία του kernel που θα φορτωθεί, από τον εκτελέσιμο κώδικα συσκευής. Ο πηγαίος κώδικας του αρχείου περιλαμβάνει τη συνάρτηση πυρήνα (σχήμα 4.4), καθώς και ορισμένα αρχεία επικεφαλίδων (header files), για συναρτήσεις και δομές OpenMP που μπορούν να χρησιμοποιηθούν στο εσωτερικό του πυρήνα. Η εκτελέσιμη μορφή του kernel, έχει παραχθεί κατά τη μεταγλώττιση του προγράμματος και το κύριο σύστημα φροντίζει να κάνει γνωστή την πλήρη διαδρομή του στη συσκευή, μαζί με το όνομα της συνάρτησης kernel.

Το CUDA Driver API από τη φύση του διαθέτει ένα σύνολο συναρτήσεων, οι οποίες έχουν τη δυνατότητα να φορτώσουν στη μνήμη της συσκευής έναν εκτελέσιμο kernel (`cuModuleLoad()`), καθώς επίσης και να εντοπίσουν σε αυτό τη συνάρτηση προς εκτέλεση (`cuModuleGetFunction()`). Ο OMPi αξιοποιεί τις συναρτήσεις αυτές στο εσωτερικό της δημιουργίας kernel. Παρ' όλα αυτά, η συνολική διαδικασία είναι εξαιρετικά χρονοβόρα. Για παράδειγμα, η φόρτωση ενός κενού kernel κοστίζει έως και 80msec, σε ορισμένες μονάδες GPU.

Ορισμένα προγράμματα χρήστη διαθέτουν περιοχές target οι οποίες μπορεί να επαναλαμβάνονται στην έκταση του κώδικα. Ένα τέτοιο παράδειγμα είναι η εμφώλευση περιοχής target στο εσωτερικό μιας περιοχής παραλληλίας. Σε αυτό το σενάριο, ο kernel, παρ' ό,τι σταθερός, πρέπει διαρκώς να εντοπίζεται από το CUDA Driver API και να φορτώνεται στη συσκευή. Οι κλήσεις στις συναρτήσεις `cuModuleLoad`

και `cuModuleGetFunction()`, προσθέτουν σημαντικές χρονικές επιβαρύνσεις στο πρόγραμμα, όταν καλούνται επαναλαμβανόμενα. Για τον σκοπό αυτό, σχεδιάσαμε έναν μηχανισμό προσωρινής αποθήκευσης kernels έτοιμων-για-φόρτωση (kernel cache). Με την πρώτη εμφάνισή του ο kernel θα περάσει από όλα τα χρονοβόρα στάδια της διαδικασίας που προβλέπει το CUDA Driver API. Πριν όμως δοθεί για φόρτωση, η τελική του μορφή αποθηκεύεται στην kernel cache που δημιουργήσαμε. Από εκεί και ύστερα, σε κάθε επόμενη εμφάνισή του, αυτός θα ανακτηθεί από τη κρυφή μνήμη, μαζί με το αρχείο προέλευσής του και την ονομασία του και οι κλήσεις προς το CUDA Driver API θα παρακαμφθούν, γλιτώνοντας σημαντικές πλεονάζουσες επιβαρύνσεις.

5.3.2 Αποθήκευση Παραμέτρων Kernel

Το δεύτερο στάδιο φόρτωσης αποτελείται από την αποθήκευση όλων των παραμέτρων που θα χρησιμοποιηθούν από τον kernel σε έναν πίνακα. Τυπικά, ο OMPi, εκτός από τα ορίσματα `dev_data` και `decl_data`, περνά τις διευθύνσεις των δεδομένων, με χρήση μεταβλητών ορισμάτων στη συνάρτηση `hm_offload`. Εκτός από τα ίδια τα δεδομένα, περνιέται και το offset τους στον χώρο διευθύνσεων. Το offset προσαυξάνεται ανάλογα με το μέγεθος ενός δεδομένου σε bytes. Σημειώνεται ότι το στάδιο της αποθήκευσης παραμέτρων, συμβαίνει κάθε φορά που πρόκειται να εκτελεστεί ένας kernel, αποθηκευμένος ή μη στη κρυφή μνήμη.

5.3.3 Εκτέλεση Kernel

Τελικό στάδιο για τη φόρτωση κώδικα στη συσκευή αποτελεί το στάδιο της εκτέλεσης του kernel. Στο στάδιο αυτό, αρχικά οριστικοποιείται ο αριθμός των ομάδων που θα εκτελέσουν τον kernel (`num_teams`), καθώς και ο αριθμός των νημάτων ανά ομάδα, ο οποίος φράσσεται από τη παράμετρο `thread_limit`. Οι περιοχές ομάδων υλοποιούνται με blocks από νήματα. Οι πληροφορίες `num_teams`, `thread_limit` αλλά και `num_threads`, όλες προερχόμενες από τις ομώνυμες φράσεις, είναι γνωστές σε αυτό το σημείο. Ο μεταφραστής τις χρησιμοποιεί για να υπολογίσει 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. Τέλος, η εκτέλεση πραγματοποιείται με βάση όλες τις υπολογισμένες παραμέτρους και όλα τα απαραίτητα δεδομένα, μέσω της

συνάρτησης `cuLaunchKernel()`. Με το πέρας της εκτέλεσης, τα νήματα CUDA συγχρονίζονται και οι δομές για τα ορίσματα του kernel αποδεσμεύονται.

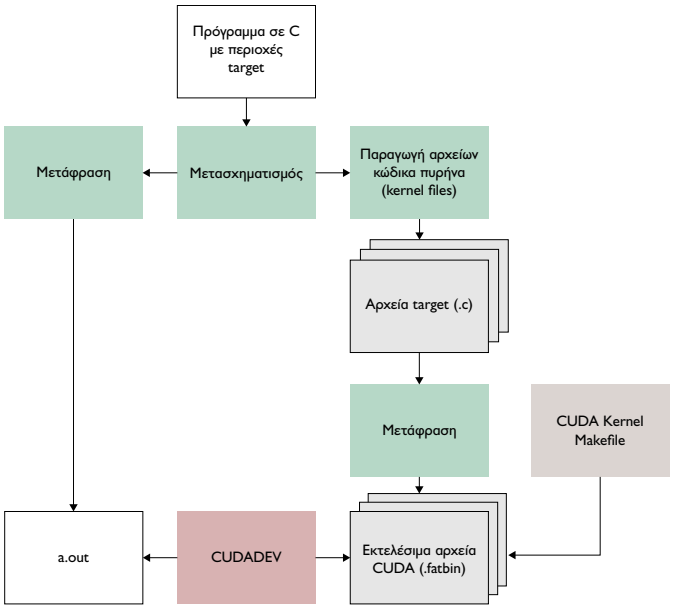
5.4 Η Βιβλιοθήκη Συσκευής devpart

Στην CUDADEV, η βιβλιοθήκη συσκευής περιλαμβάνει το σύνολο των ρουτινών OpenMP, οι οποίες δυνητικά καλούνται από τον κώδικα χρήστη που πρόκειται να φορτωθεί στη συσκευή. Προς το παρόν, η συσκευή περιλαμβάνει τις εξης λειτουργίες, για την υποστήριξη του χρόνου εκτέλεσης του προγράμματος:

- **Λήψη τμήματος βρόχου (chunk)**. Η βιβλιοθήκη συσκευής της CUDADEV υποστηρίζει τη συνάρτηση `ort_get_distribute_chunk()`, η οποία υλοποιεί τη διαδικασία λήψης ενός τμήματος βρόχου, ο οποίος διαμοιράζεται μέσω της οδηγίας *#pragma omp distribute*. Επιπλέον, παρέχεται η υλοποίηση της συνάρτησης `ort_get_static_default_chunk()`, με παρόμοια λειτουργικότητα, όταν πραγματοποιείται διαμοιρασμός ενός βρόχου με την οδηγία *#pragma omp for*. Αυτές είναι και οι μόνες δύο λειτουργίες OpenMP που μπορούν να υπάρχουν σε έναν kernel προς το παρόν.

- **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, στο εσωτερικό του κώδικα που φορτώνεται στη συσκευή μέσω των οδηγιών που υλοποιήθηκαν.

- **Εργαλεία Μεταγλώττισης Κώδικα Συσκευής**. Εκτός από την υλοποίηση ρουτινών του OpenMP και βοηθητικών συναρτήσεων, η βιβλιοθήκη περιλαμβάνει δύο βασικά συστατικά, τα οποία είναι υπεύθυνα για την μετάφραση των αρχείων συσκευών που παράγονται κατά τον μετασχηματισμό μιας οδηγίας target (Ενότητα 4.3.2). Συγκεκριμένα, το πρώτο συστατικό αποτελεί ένα εργαλείο που ακολουθεί την διαδικασία του μεταγλωττιστή `nvcc` για τη μετάφραση κώδικα CUDA C, με ορισμένες τροποποιήσεις. Οι τροποποιήσεις αφορούν στην αφαίρεση ορισμένων βιβλιοθηκών που δεν χρησιμοποιούνται από τον μεταφραστή OMPi, κατά τη διαδικασία μετάφρασης. Το δεύτερο συστατικό είναι το αρχείο Kernel Makefile, το οποίο χρησιμοποιεί το προαναφερθέν εργαλείο για να μεταγλωττίσει τον εξαγόμενο κώδικα συσκευής target.



Σχήμα 5.1: Η πλήρης διαδικασία μεταγλώττισης ενός προγράμματος OpenMP για τη συσκευή CUDADEV

Στο σχήμα 5.1 απεικονίζεται η διαδικασία μετάφρασης ενός προγράμματος με περιοχές target που στοχεύουν συσκευές CUDA. Αρχικά, το πρόγραμμα περνάει από το στάδιο του μετασχηματισμού, όπου ο κώδικας OpenMP αντικαθιστάται από κώδικα C. Στη συνέχεια παράγεται ένα αρχείο που αφορά μόνο την εκτέλεση στον host, για το οποίο και παράγεται το βασικό εκτελέσιμο (a.out). Επιπλέον, κάθε περιοχή target εξάγεται σε ένα δικό της ξεχωριστό αρχείο, το οποίο ονομάζεται kernel file. Κάθε kernel file πρέπει να μεταφραστεί. Στο σημείο αυτό εισέρχεται ο ρόλος του kernel Makefile, ενός αρχείου που είναι υπεύθυνο να μεταφράσει όλα τα 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.

Κεφαλαιο 6

Αξιολογηση

6.1 Εισαγωγή

Η αξιολόγηση της υποδομής που υλοποιήθηκε στα πλαίσια της παρούσας εργασίας, πραγματοποιήθηκε μέσω πειραμάτων σε δύο μονάδες γραφικής επεξεργασίας, την NVIDIA Tesla P40 και την NVIDIA GeForce GT 730. Οι δύο μονάδες επιλέχθηκαν διότι αποτελούν δύο αντιδιαμετρικές περιπτώσεις, με την πρώτη να διαθέτει υψηλή υπολογιστική έκδοση CUDA (CUDA compute capability 6.1) και σαφώς καλύτερες προδιαγραφές, συγκριτικά με τη δεύτερη, η οποία διαθέτει την ελάχιστη των υπολογιστικών εκδόσεων (CUDA compute capability 3.5). Οι προδιαγραφές των συσκευών παρουσιάζονται στον πίνακα 6.1. Τα κύρια συστήματα στα οποία βρίσκονται εγκατεστημένες οι προαναφερθείσες μονάδες GPU είναι τα συστήματα Parallax και Paragon, αντίστοιχα. Το σύστημα Parallax είναι ένας εξυπηρετητής EMC PowerEdge που διαθέτει δύο κύριους επεξεργαστές της σειράς Xeon Gold, με 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.

	P40	**GT 730**
# Πολυεπεξεργαστών	30	2
# Πυρήνων/Πολυεπεξεργαστή	128	192
Συνολικό # Πυρήνων	3840	384
Μέγιστο # Threads/Block	1024	1024
Μνήμη	24GB	2GB

Πίνακας 6.1: Σύγκριση συσκευών γραφικής επεξεργασίας.

Τα πειράματα που πραγματοποιήθηκαν κατηγοριοποιούνται σε πειράματα ορθότητας και πειράματα επιδόσεων. Η πρώτη κατηγορία αφορά στην επιβεβαίωση της ορθής λειτουργίας όλων των υλοποιήσεων, ενώ η δεύτερη κατηγορία μελετά τις επιδόσεις που επιτυγχάνονται με χρήση της ολοκληρωμένης υποδομής της εργασίας, για τη φόρτωση κώδικα στις δύο μονάδες GPU.

6.2 Αξιολόγηση Ορθότητας

Για την επαλήθευση της ορθότητας των υλοποιήσεων, επιλέχθηκε η χρήση δύο συλλογών δοκιμαστικών προγραμμάτων, οι οποίες αφορούν τη λειτουργικότητα των οδηγιών τύπου target. Η πρώτη συλλογή αποτελείται από ένα σύνολο εσωτερικών προγραμμάτων του μεταφραστή OMPi και καλύπτουν ένα μεγάλο αριθμό σεναρίων και περιπτώσεων οδηγιών. Η δεύτερη συλλογή, ονομάζεται OMPVV (OpenMP Validation and Verification) [19] και αποτελεί τμήμα του έργου SOLLVE (Scaling OpenMP Via LLVM for Exascale Performance and Portability). Ενδεικτικά, τα δοκιμαστικά προγράμματα δοκιμάζουν τις εξής λειτουργικότητες που σχετιζόνται με την οδηγία target:

- **Περιβάλλοντα Δεδομένων (Data)**. Ένα υποσύνολο των δοκιμών σχετίζεται με την ορθή δημιουργία περιβάλλοντος δεδομένων στη συσκευή μέσω της οδηγίας *target data*, χρησιμοποιώντας όλες τις δυνατές φράσεις που μπορούν να την συνοδεύσουν.

- **Τμήματα Πινάκων (Array Sections)**. Το είδος αυτό δοκιμαστικών προγραμμάτων αφορά την ορθή αντιστοίχιση τμημάτων πινάκων στο περιβάλλον δεδομένων της συσκευής, την τροποποίηση τους καθώς επίσης και τη μεταφορά

στο κύριο σύστημα.

- **Δηλώσεις (Declare Target)**. Οι δοκιμές αυτές αφορούν τη σωστή δήλωση καθολικών δεδομένων στη συσκευή, με χρήση της οδηγίας *declare target*.

- **Εισαγωγή/Εξαγωγή Δεδομένων (Enter/Exit Data)**. Μέσω δοκιμών των οδηγιών *target enter data* και *target exit data*, δοκιμάζεται η ορθή αποστολή και λήψη δεδομένων προς και από το περιβάλλον της συσκευής.

- **Δείκτες**. Δοκιμάζεται η αντιστοίχιση δεικτών στο περιβάλλον της συσκευής, η τροποποίηση των δεδομένων που βρίσκονται στην δεικτούμενη διεύθυνση, καθώς επίσης και η ορθότητα τους όταν αυτά αντιστοιχιστούν με την φράση `from` από το κύριο σύστημα.

- **Ενημέρωση (Update)**. Με το συγκεκριμένο είδος δοκιμαστικών προγραμμάτων, ελέγχεται η ορθή ενημέρωση του περιβάλλοντος δεδομένων της συσκευής, έπειτα από τροποποίηση μιας μεταβλητής στο κύριο σύστημα, η οποία έχει προηγουμένως αντιστοιχιστεί.

- **Ομάδες (Teams)**. Η λειτουργικότητα των ομάδων εξετάζεται σε βάθος, με χρήση όλων των σχετικών οδηγιών, όπως *target teams distribute* και *target teams distribute parallel for*. Συγκεκριμένα, ελέγχεται η ορθότητα του διαμοιρασμού ενός επαναληπτικού βρόχου στις ομάδες, καθώς επίσης και ο περαιτέρω διαμοιρασμός στα νήματά της.

- **Περιοχές Παραλληλίας εντός Target**. Με το είδος αυτό των δοκιμαστικών προγραμμάτων, ελέγχεται η ορθή δημιουργία ομάδας παραλληλίας στη συσκευή, μέσω οδηγιών *target parallel*, καθώς επίσης και ο διαμοιρασμός επαναληπτικού βρόχου σε αυτή, μέσω της οδηγίας *target parallel for*.

- **Ρουτίνες OpenMP**. Ένα υποσύνολο δοκιμαστικών προγραμμάτων ελέγχουν ορισμένες ρουτίνες OpenMP που αφορούν τη δυναμική δέσμευση και αντιγραφή δεδομένων στο περιβάλλον της συσκευής.

Τα αποτελέσματα των πειραμάτων ορθότητας, για τα δύο συστήματα, παρουσιάζονται στους πίνακες 6.2 και B.1 (παράρτημα B). Ως `RP` συμβολίζεται η επιτυχία στο επίπεδο εκτέλεσης, η οποία ισοδυναμεί με πλήρη επιτυχία. Από την άλλη, ως

	Parallax	Paragon		Parallax	Paragon
mmm_target.c	RP	RP	test_target_data_use_device_ptr.c	RP	RP
qmcpack_target_math.c	RP	RP	test_target_enter_data_depend.c	RP	RP
qmcpack_target_static_lib.c	RP	RP	test_target_enter_data_devices.c	RP	RP
reduction_separated_directives.c	CF	CF	test_target_enter_data_global_array.c	RP	RP
offloading_success.c	RP	RP	test_target_enter_data_if.c	RP	RP
test_target_defaultmap.c	RP	RP	test_target_enter_data_malloced_array.c	RP	RP
test_target_depends.c	RP	RP	test_target_enter_data_struct.c	RP	RP
test_target_device.c	RP	RP	test_target_enter_exit_data_devices.c	RP	RP
test_target_if.c	RP	RP	test_target_enter_exit_data_if.c	RP	RP
test_target_is_device_ptr.c	RP	RP	test_target_enter_exit_data_map_global_array.c	RP	RP
test_target_map_array_default.c	RP	RP	test_target_enter_exit_data_map_malloced_array.c	RP	RP
test_target_map_global_arrays.c	RP	RP	test_target_enter_exit_data_map_pointer_translation.c	RP	RP
test_target_map_local_array.c	RP	RP	test_target_enter_exit_data_struct.c	RP	RP
test_target_map_pointer.c	RP	RP	test_target_parallel.c	RP	RP
test_target_map_pointer_no_map_type_modifier.c	RP	RP	test_target_update_depend.c	RP	RP
test_target_map_scalar_no_map_type_modifier.c	RP	RP	test_target_update_devices.c	RP	RP
test_target_map_zero_length_pointer.c	RP	RP	test_target_update_from.c	RP	RP
test_target_private.c	RP	RP	test_target_update_if.c	RP	RP
test_target_firstprivate.c	RP	RP	test_target_update_to.c	RP	RP
test_target_data_if.c	RP	RP	test_target_and_task_nowait.c	RP	RP
test_target_data_map_alloc.c	RP	RP	test_task_target.c	RP	RP
test_target_data_map_devices.c	RP	RP			
test_target_data_map_from.c	RP	RP			
test_target_data_map_pointer_translation.c	RP	RP			
test_target_data_map_to.c	RP	RP			
test_target_data_map_to_from.c	RP	RP			
test_target_data_map_tofrom.c	RP	RP			
test_target_data_pointer_swap.c	RP	RP			

Πίνακας 6.2: Τα αποτελέσματα εκτέλεσης των δοκιμαστικών προγραμμάτων OMPVV (RP = Runtime Pass, CF = Compilation Fail, CP = Compilation Pass)

CF ορίζονται οι περιπτώσεις όπου το δοκιμαστικό πρόγραμμα απέτυχε να μεταφραστεί, συνήθως λόγω μη υποστήριξης συγκεκριμένης οδηγίας ή λειτουργίας στο επίπεδο της μεταγλώττισης. Τέλος, ως CP συμβολίζονται οι περιπτώσεις όπου το δοκιμαστικό πρόγραμμα μεταφράζεται με επιτυχία, αλλά δεν επιτυγχάνεται ορθή εκτέλεση. Περιπτώσεις όπως η τελευταία αφορά οδηγίες οι οποίες αναγνωρίζονται από τον μεταφραστή, αλλά δεν έχουν υλοποιηθεί στο επίπεδο runtime. Παρατηρείται ότι τα μοναδικά δοκιμαστικά προγράμματα που απέτυχαν ήταν αυτά που αφορούν τις κλειδαριές και την λειτουργία reduction. Αυτό συμβαίνει διότι οι αντίστοιχες λειτουργικότητες απουσιάζουν την τρέχουσα στιγμή από τη συσκευή CUDADEV.

6.3 Αξιολόγηση Επιδόσεων

Η αξιολόγηση των επιδόσεων της συσκευής CUDADEV αποτελείται από διάφορα είδη μετροπρογραμμάτων αλλά και εφαρμογών, οι οποίες αξιολογούν διάφορες πτυχές της. Η πρώτη υποενότητα αφορά την μέτρηση της καθυστέρησης εκκίνησης της συσκευής CUDA. Η δεύτερη υποενότητα εξετάζει την επιτάχυνση με την εισαγωγή

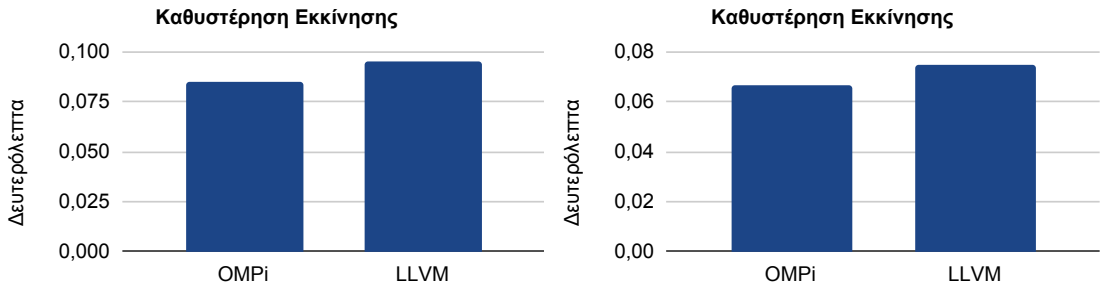
	Επαν. #1	Επαν. #2	Επαν. #3	Επαν. #4	Επαν. #5	Μέση τιμή
Parallax (OMPi)	0,0845	0,0874	0,0831	0,0843	0,0838	**0,0846**
Parallax (LLVM)	0,0964	0,0952	0,0958	0,0946	0,0951	**0,0954**
Paragon (OMPi)	0,0666	0,0665	0,0666	0,0666	0,0662	**0,0665**
Paragon (LLVM)	0,0759	0,0747	0,0746	0,0747	0,0750	**0,0750**

Πίνακας 6.3: Μέτρηση χρονικής καθυστέρησης εκκίνησης συσκευής CUDA (δευτερόλεπτα)

της κρυφής μνήμης για τους kernel, στο επίπεδο φόρτωσης. Η τρίτη υποενότητα εξετάζει τις επιδόσεις της συσκευής CUDADEV σε ορισμένες εφαρμογές.

6.3.1 Καθυστερήσεις Εκκίνησης

Για τη μέτρηση του χρόνου εκκίνησης της συσκευής CUDADEV, εκτελέστηκε ένα απλό πρόγραμμα το οποίο αποτελείται από δύο όμοιες περιοχές target. Το πρόγραμμα χρονομετρά την εκτέλεση και των δύο περιοχών και εξάγει τους χρόνους T1 και T2 αντίστοιχα. Λαμβάνουμε τη χρονική διάρκεια `T1 - T2`, η οποία αφορά καθαρά την αρχικοποίηση της κατάστασης της συσκευής. Οι δοκιμές πραγματοποιήθηκαν και για τα δύο συστήματα, στους μεταφραστές OMPi και LLVM και τα αποτελέσματά τους παρατίθενται στον πίνακα 6.3. Η σύγκριση των αποτελεσμάτων απεικονίζεται στο σχήμα 6.1. Συμπεραίνεται ότι το κόστος εκκίνησης είναι σχεδόν σταθερό, με το σύστημα Parallax και τη μονάδα GPU P40 να διαθέτει μεγαλύτερη επιβάρυνση, συγκριτικά με το σύστημα Paragon. Η διαφορά αυτή προκύπτει λόγω της φύσης της εκκίνησης μιας συσκευής CUDA, η οποία περιλαμβάνει την αντιστοίχιση ολόκληρου του χώρου διευθύνσεων της συσκευής και του κυρίου συστήματος σε έναν μοναδικό εικονικό χώρο διευθύνσεων, καθώς επίσης και η αρχικοποίηση ολόκληρου του runtime του μεταφραστή, διαδικασίες οι οποίες εξαρτώνται από την επίδοση ενός νήματος του κυρίου επεξεργαστή (single-thread performance) του συστήματος. Επιπλέον, παρατηρούμε οτι ο μεταφραστής OMPi επιτυγχάνει συνολικά καλύτερες επιδόσεις για την εκκίνηση της μονάδας GPU, σε σύγκριση με τον μεταφραστή LLVM.



(α) Parallax (P40)

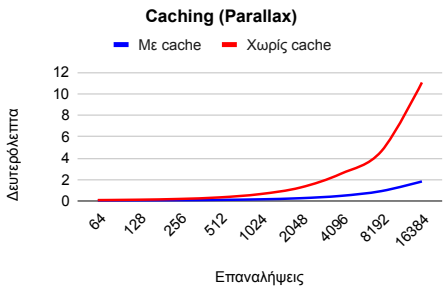
(β) Paragon (GT 730)

Σχήμα 6.1: Σύγκριση χρονικής καθυστέρησης εκκίνησης συσκευής CUDA (δευτερόλεπτα)

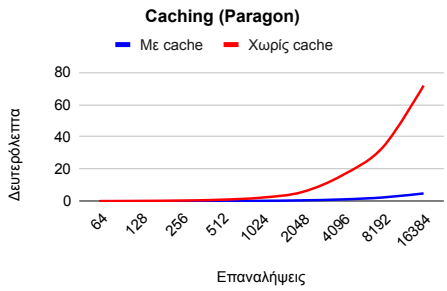
6.3.2 Κρυφή Μνήμη

Για την αξιολόγηση της κρυφής μνήμης που χρησιμοποιείται κατά τη φόρτωση ενός kernel στη συσκευή CUDADEV του μεταφραστή OMPi, χρησιμοποιήθηκε ένα δοκιμαστικό πρόγραμμα που αποτελείται από έναν επαναληπτικό βρόχο, ο οποίος στο εσωτερικό του δημιουργεί τρεις περιοχές target, δύο εκ των οποίων είναι όμοιες. Ο μηχανισμός της κρυφής μνήμης ενεργοποιείται λόγω των επανειλημμένων χρήσεων του ίδιου kernel. Σημειώνεται ότι εάν απουσίαζε ο επαναληπτικός βρόχος και οι όμοιες περιοχές γραφόταν με το χέρι, η κρυφή μνήμη δεν θα ενεργοποιούταν. Τα αποτελέσματα για τη χρήση κρυφής μνήμης παρατίθενται στο παράρτημα Β, στους πίνακες Β.2 και Β.4. Στην περίπτωση όπου δεν υφίσταται κρυφή μνήμη, κάθε μοναδική φόρτωση kernel απαιτεί την εκ νέου εύρεση και δημιουργία του. Τα αποτελέσματα της περίπτωσης αυτής παρατίθενται στον πίνακα Β.3 και Β.5. Μια γραφική απεικόνιση των αποτελεσμάτων και των διαφορών τους ανά σύστημα, παρουσιάζεται στο σχήμα 6.2.

Παρατηρείται ότι στο σύστημα Parallax, χωρίς την χρήση κρυφής μνήμης, οι καθυστερήσεις για μεγάλο πλήθος επαναλήψεων, ξεπερνούν τα 10 δευτερόλεπτα, ενώ για την ίδια περίπτωση, στο σύστημα Paragon, οι διαφορές στις χρονικές επιβαρύνσεις ξεπερνούν τα 70 δευτερόλεπτα.



(α) Parallax (P40)



(β) Paragon (GT 730)

Σχήμα 6.2: Σύγκριση χρόνων εκτέλεσης για τη χρήση ή μη κρυφή μνήμης kernel (δευτερόλεπτα)

6.3.3 Εφαρμογές

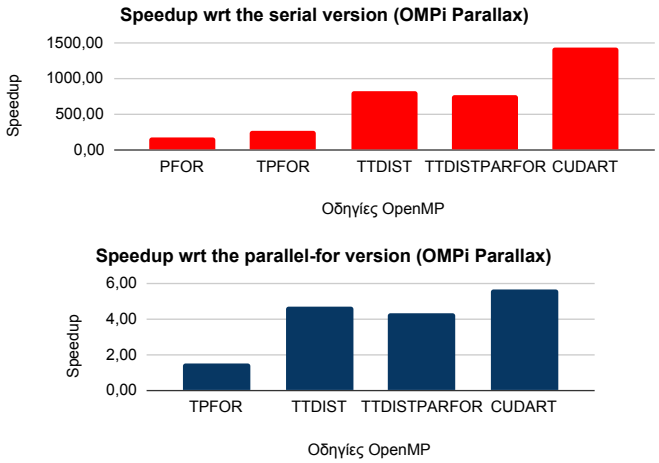
Περιοχές Διαμοιρασμού & Παραλληλίας

Το πρώτο δοκιμαστικό πρόγραμμα που χρησιμοποιήθηκε για τη μέτρηση των επιδόσεων όλων των υλοποιημένων οδηγιών, αποτελεί μια ψευδοεφαρμογή που αποτελείται από επαναληπτικούς βρόχους. Την δουλειά που κάνουν αυτοί οι βρόχοι την υλοποιούμε με 5 διαφορετικούς τρόπους:

1. Σειριακός υπολογισμός
2. Με χρήση της οδηγίας *parallel for*, όπου χρησιμοποιούνται προφανώς οι μόνο πυρήνες των κύριων επεξεργαστών του συστήματος
3. Με χρήση της οδηγίας *target parallel for*
4. Με χρήση της οδηγίας *target teams distribute*
5. Με χρήση της οδηγίας *target teams distribute parallel for*

όπου στις 3 τελευταίες περιπτώσεις ο υπολογιμός γίνεται στην GPU. Τέλος, συγκρίνουμε και με μια 6η εκδοχή της ψευτοεφαρμογής όπου όλη η υλοποίηση έχει γίνει αποκλειστικά με χρήση CUDA runtime. Ένα δείγμα του δοκιμαστικού προγράμματος βρίσκεται στο παράρτημα Α, στο σχήμα Α.4.

Αρχικά, το πρόγραμμα δημιουργεί έναν πίνακα διαστάσεων 1024 x 1024. Ο κώδικας που εκτελείται στο εσωτερικό των οδηγιών, αποτελείται από έναν τριπλά εμφωλευμένο επαναληπτικό βρόχο. Οι δύο πρώτοι επαναληπτικοί βρόχοι αποτελούνται από 1024 επαναλήψεις. Στο εσωτερικό του τρίτου βρόχου εκτελούνται 2000



Σχήμα 6.3: Σύγκριση χρόνων εκτέλεσης της εκτέλεσης στην GPU χρησιμοποιώντας τις υλόποιημένες οδηγίες συγκριτικά με τη σειριακή εκτέλεση και την παράλληλη εκτέλεση στην κύρια CPU (Speedup)

πολλαπλασιασμοί και προσθέσεις σε τοπική μεταβλητή και μετά τον τερματισμό του, η τιμή της μεταβλητής αποθηκεύεται σε μία θέση του πίνακα. Για τις οδηγίες *parallel for* και *target parallel for*, επιλέγεται η χρήση 64 νημάτων. Ο αριθμός αυτός επιλέχθηκε διότι το σύστημα Parallax διαθέτει 64 νήματα. Για τις οδηγίες *target teams distribute* και *target teams distribute parallel for*, επιλέγεται πλήθος ομάδων ίσο με 1024. Το πλήθος αυτό επιλέχθηκε διότι η μονάδα GPU επιτρέπει μέγιστο πλήθος block ίσο με 1024. Τέλος, για την οδηγία *target teams distribute parallel for*, επιλέγεται και το πλήθος νημάτων να ισούται με 1024 και για την εφαρμογή που κάνει χρήση του CUDA runtime, 1024 blocks μεγέθους 1024 νημάτων.

Κάθε μια από τις εκδοχές εκτελείται από 5 φορές, για να ληφθεί μια μέση τιμή του χρόνου εκτέλεσής της. Αφού ληφθούν όλοι οι μέσοι χρόνοι εκτέλεσης των εκδοχών, λαμβάνεται η επιτάχυνση (speedup) σε σχέση με τον σειριακό χρόνο εκτέλεσης και έπειτα σε σχέση με τον χρόνο εκτέλεσης της εκδοχής *parallel for*. Τα αποτελέσματα παρουσιάζονται στις γραφικές παραστάσεις των σχημάτων 6.3 και 6.4, για τις δύο GPU των συστημάτων Parallax και Paragon. Συμβολίζουμε την εκδοχή που κάνει χρήση της οδηγίας parallel for ως `PFOR`, την οδηγία *target parallel for* ως `TPFOR`, την οδηγία *target teams distribute* ως `TTDIST` και την οδηγία *target teams distribute parallel for* ως `TTDISTPFOR`. Η εκδοχή που είναι γραμμένη αποκειστικά σε CUDA runtime συμβολίζεται ως CUDART.

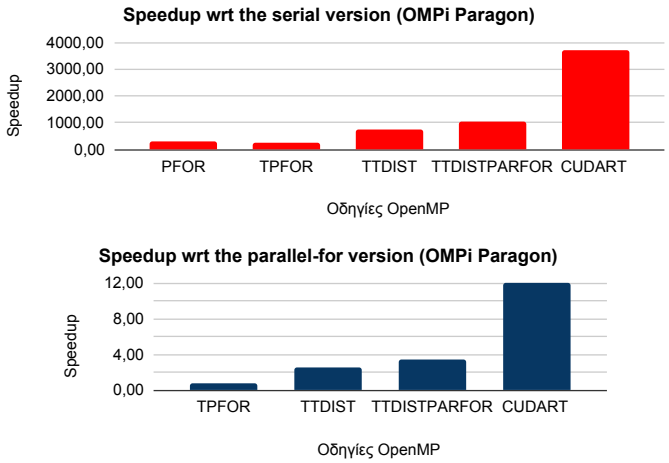
Η επιτάχυνση που επιτυγχάνει η εκδοχή που χρησιμοποιεί την οδηγία *parallel*

for σε σχέση με τη σειριακή μορφή, είναι περίπου 17x για το σύστημα Parallax και 16x για το σύστημα Paragon. Η επιτάχυνση αυτή οφείλεται στην αξιοποίηση πολλών πυρήνων του επεξεργαστή, έναντι του ενός που χρησιμοποιείται στην σειριακή εκτέλεση, για την παραλληλοποίηση του εξωτερικού βρόχου. Στο σύστημα Paragon, η επιτάχυνση είναι μεγαλύτερη διότι ο κύριος επεξεργαστής είναι παλαιότερης γενιάς, έχοντας χειρότερη επίδοση ανά πυρήνα, σε σύγκριση με τον επεξεργαστή του συστήματος Parallax.

Προχωρώντας, παρατηρούμε ότι η εκδοχή *target parallel for* επιτυγχάνει επιτάχυνση σχεδόν 250x στο σύστημα Parallax και περίπου 230x στο σύστημα Paragon, σε σχέση με το σειριακό πρόγραμμα. Η επιτάχυνση οφείλεται στο ότι δημιουργείται ένα μπλοκ 32 νημάτων στο εσωτερικό της μονάδας GPU, στο οποίο διαμοιράζονται οι επαναλήψεις του εξωτερικού βρόχου. Το μπλοκ αυτό έχει τη δυνατότητα να εκτελεί υπολογισμούς αυτής της μορφής με ταχύτατο τρόπο, εξαιτίας της φύσης των πυρήνων της GPU. Για το σύστημα Paragon, παρατηρείται ότι η επιτάχυνση είναι μικρότερη αυτής που επιτυγχάνει η εκδοχή *parallel for*, διότι η μονάδα GPU που χρησιμοποιείται είναι αρκετά βασικού επιπέδου και δεν διαθέτει μεγάλη υπολογιστική ισχύ ανά πυρήνα.

Παρατηρούμε ότι σημαντικά μεγάλες επιταχύνσεις (σχεδόν 800x), επιτυγχάνουν οι οδηγίες *target teams distribute* και *target teams distribute parallel for*, οι οποίες στο εσωτερικό της GPU ένα πλήθος από μπλοκ, ίσο με την πρώτη διάσταση του πίνακα. Στην περίπτωση της οδηγίας *target teams distribute*, κάθε μπλοκ διαθέτει ένα μοναδικό νήμα, ενώ στη περίπτωση της οδηγίας *target teams distribute parallel for*, ο αριθμός των νημάτων ισούται με την δεύτερη διάσταση του πίνακα. Η μικρή υστέρηση που παρατηρείται στην δεύτερη οδηγία, σε σχέση με την πρώτη, οφείλεται στο γεγονός ότι τα νήματα εκτελούν περαιτέρω κώδικα για τον δεύτερο διαμοιρασμό των επαναλήψεων.

Τη μέγιστη των επιταχύνσεων επιτυγχάνει η εκδοχή που κάνει χρήση του CUDA runtime, με τα αποτελέσματα να φτάνουν το 1400x για το σύστημα Parallax και περίπου 3700x για το σύστημα Paragon. Παρ' όλο που οι χρονικές επιταχύνσεις μοιάζουν να διαφέρουν σημαντικά με αυτές των οδηγιών OpenMP, η πραγματική διαφορά στις χρονικές επιδόσεις είναι της τάξεως των λίγων millisecond (ms). Τέλος, στο σύστημα Parallax η επιτάχυνση είναι μικρότερη από αυτή του συστήματος Paragon, διότι το χάσμα επιδόσεων επεξεργαστή και μονάδας GPU είναι μεγαλύτερο στη δεύτερη περίπτωση.



Σχήμα 6.4: Σύγκριση χρόνων εκτέλεσης σειριακής εκδοχής και εκδοχής parallel for με τις υλοποιημένες οδηγίες στο μηχάνημα Paragon (Speedup)

Gaussian Blur

Η δεύτερη εφαρμογή που αναπτύχθηκε αφορά την εφαρμογή θόλωσης Gauss (Gaussian blur) σε έγχρωμες εικόνες. Το πρόγραμμα λαμβάνει ως παραμέτρους την ακτίνα της θόλωσης, καθώς επίσης και την εικόνα σε μορφή bitmap που πρόκειται να θολωθεί. Κατά την εκκίνηση του προγράμματος, διατρέχονται όλα τα εικονοστοιχεία της εικόνας και η χρωματική τριάδα για κάθε εικονοστοιχείο αποθηκεύεται σε τρεις διαφορετικούς μονοδιάστατους πίνακες `R`, `G` και `B`, διάστασης $W*H$. Αφού ολοκληρωθεί η αποθήκευση της εικόνας στη μνήμη, τότε ξεκινά ο αλγόριθμος για τη θόλωση, ο οποίος αποτελείται από έναν βρόχο που διατρέχει όλα τα αποθηκευμένα εικονοστοιχεία και υπολογίζει τις τιμές τους, βάσει της θόλωσης με την δοθείσα ακτίνα. Ο ψευδοκώδικας της εφαρμογής σε C παρατίθεται στο σχήμα A.5 του παραρτήματος A.

Από τη δομή του αλγορίθμου, προκύπτει η ευκαιρία για την παραλληλοποίησή του. Συγκεκριμένα, ο υπολογισμός της νέας τιμής κάθε εικονοστοιχείου μπορεί να γίνει ανεξάρτητα από τα υπόλοιπα στοιχεία (και άρα παράλληλα), επιταχύνοντας έτσι σημαντικά την όλη διαδικασία της θόλωσης. Η συνολική εργασία του επαναληπτικού βρόχου μπορεί να διαμοιραστεί είτε στα νήματα του κύριου επεξεργαστή, μέσω της οδηγίας *parallel for* ή στην μονάδα GPU, με χρήση των νέων οδηγιών που βασίζονται στις περιοχές target.

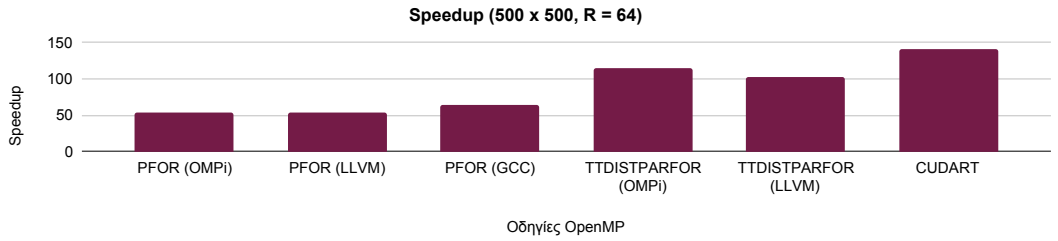
Σκοπός της αξιολόγησης είναι η σύγκριση των επιδόσεων που επιτυγχάνονται με σειριακή εκτέλεση στον κύριο επεξεργαστή και της εκδοχής που χρησιμοποιεί

500 x 500				
	R=1	**R=32**	**R=64**	**R=128**
SERIAL	0,080097	32,826188	128,999644	512,402343
PFOR (OMPi)	0,019979	0,625330	2,414494	9,103194
PFOR (GCC)	0,029814	0,063593	1,991589	7,754831
PFOR (LLVM)	0,035934	0,652496	2,379363	9,001340
TTDISTPFOR (OMPi)	0,092224	0,387227	1,123572	3,980379
TTDISTPFOR (LLVM)	0,096257	0,427303	1,257162	4,503726
CUDART	0,001004	0,269007	0,918387	2,414439

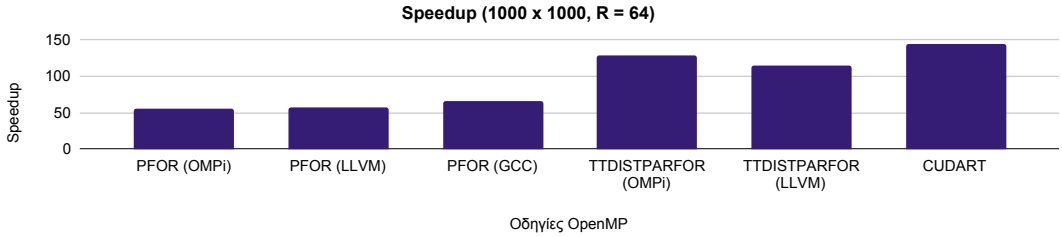
Πίνακας 6.4: Χρονικές επιδόσεις της εφαρμογής Gaussian blur για μέγεθος εικόνας 500 x 500 (δευτερόλεπτα)

αποκλειστικά τη διεπαφή CUDA runtime, με τις επιδόσεις που επιτυγχάνονται παραλληλοποιώντας τον βρόχο που διατρέχει κάθε εικονοστοιχείο της εικόνας-εισόδου. Ως είσοδοι χρησιμοποιήθηκαν τρεις εικόνες, διαστάσεων 2000 x 2000, 1000 x 1000 και 500 x 500, αντίστοιχα. Οι τιμές που επιλέχθηκαν για την ακτίνα θόλωσης είναι οι 1, 32, 64 και 128. Για κάθε εικόνα, εκτελέστηκε η διαδικασία της θόλωσης με τις παραπάνω τιμές και το πρόγραμμα τροποποιήθηκε και δημιουργήθηκαν τρεις διαφορετικές εκδοχές του. Η πρώτη, με την ονομασία `PFOR`, χρησιμοποιεί την οδηγία *parallel for* για παραλληλοποίηση του υπολογισμού των νέων τιμών θόλωσης. Η δεύτερη εκδοχή (`TTDISTPARFOR`) χρησιμοποιεί την οδηγία *target teams distribute parallel for* για την παραλληλοποίηση στη συσκευή CUDA. Η τρίτη (`CUDART`) χρησιμοποιεί εξ' ολοκλήρου τη διεπαφή CUDA runtime. Όλες οι μετρήσεις πραγματοποιήθηκαν με χρήση του μεταφραστή OMPi, στο σύστημα Parallax, με την μονάδα GPU P40. Για τα δύο πρώτα σενάρια, χρησιμοποιήθηκαν επιπλέον οι μεταφραστές gcc και LLVM. Το τρίτο σενάριο αφορά τη χρήση του μεταφραστή nvcc, ο οποίος μεταφράζει ένα πρόγραμμα CUDA C σε εκτελέσιμο. Όλες οι παραλληλοποιήσεις βρόχων με οδηγίες OpenMP κάνουν χρήση στατικής χρονοδρομολόγησης. Τα αποτελέσματα που λάβαμε από τις δύο τροποποιημένες μορφές, συγκρίνονται με την σειριακή μορφή του προγράμματος, η οποία μεταφράζεται με τον μεταφραστή συστήματος gcc και παρατίθενται στους πίνακες 6.4 και 6.5. Οι γραφικές αναπαραστάσεις των αποτελεσμάτων για ορισμένες περιπτώσεις παρατίθενται στο σχήμα 6.5.

Για την οδηγία *target teams distribute parallel for*, χρησιμοποιήθηκαν 500 ομάδες των 500 νημάτων. Η επιλογή αυτή έγινε για την αντιστοίχιση, αρχικά, κάθε νήματος σε ένα εικονοστοιχείο, όταν χρησιμοποιείται ως είσοδος η εικόνα διαστάσων 500



(α) Εικόνα 500x500, R=64



(β) Εικόνα 1000x1000, R=64

Σχήμα 6.5: Σύγκριση χρόνων εκτέλεσης εκδοχών που κάνουν χρήση οδηγιών OpenMP για την εφαρμογή Gaussian blur στο μηχάνημα Parallel (Speedup)

x 500. Τα πλήθη αυτά διατηρήθηκαν και για τις επόμενες εικόνες. Επιπλέον, σημειώνεται ότι για την συγκεκριμένη οδηγία χρησιμοποιήθηκε η φράση `collapse` με παράμετρο 2, ώστε να μπορέσει να διαμοιραστεί με ορθό τρόπο ο εξωτερικός βρόχος. Από τα αποτελέσματα του σχήματος 6.5, αποδεικνύεται ότι η εν λόγω οδηγία έχει τη δυνατότητα να επιτύχει έως και 130 φορές καλύτερες επιδόσεις, σε σχέση με το σειριακό πρόγραμμα, στον μεταφραστή OMPi. Ο μεταφραστής LLVM, με χρήση της ίδιας οδηγίας, επιτυγχάνει επιτάχυνση περίπου 110x. Με χρήση της οδηγίας parallel for, ο OMPi μπορεί να επιτύχει επιτάχυνση που αγγίζει το 50x, σε σχέση με το σειριακό πρόγραμμα. Οι μεταφραστές LLVM και gcc, αγγίζουν επίσης το 50x. Όσον αφορά την οδηγία *target teams distribute parallel for*, η μεγάλη διαφορά στις επιδόσεις οφείλεται στο γεγονός ότι η εργασία για κάθε pixel της εικόνας μπορεί άμεσα να αντιστοιχιστεί σε έναν μικροπυρήνα της μονάδας GPU, των οποίων το πλήθος είναι αρκετά μεγαλύτερο συγκριτικά με το πλήθος των πυρήνων του κύριου επεξεργαστή. Η φύση των υπολογισμών για την θόλωση Gauss είναι πολύ κοντά στο είδος πράξεων που μπορούν να εκτελεστούν ταχύτατα από έναν μικροπυρήνα. Τέλος, η εφαρμογή που κάνει εξ' ολοκλήρου χρήση της διεπαφής CUDA runtime επιτυγχάνει επιδόσεις έως και σχεδόν 150 φορές καλύτερες από το σειριακό πρόγραμμα. Η διαφορά αυτή οφείλεται στο γεγονός ότι οι μεταφραστές OpenMP, προκειμέ-

1000 x 1000				
	R=1	**R=32**	**R=64**	**R=128**
SERIAL	0,302881	131,330476	518,129028	2.010,345017
PFOR (OMPi)	0,027614	2,373087	9,353769	36,148291
PFOR (GCC)	0,030468	2,026861	7,852925	31,547519
PFOR (LLVM)	0,160699	2,590599	9,131944	35,213705
TTDISTPFOR (OMPi)	0,098117	1,143651	4,014489	15,437011
TTDISTPFOR (LLVM)	0,094171	1,278659	4,539923	17,518972
CUDART	0,003689	1,016114	3,576960	11,859930

Πίνακας 6.5: Χρονικές επιδόσεις της εφαρμογής Gaussian blur για μέγεθος εικόνας 1000 x 1000 (δευτερόλεπτα)

νου να επιτρέψουν την συγγραφή γενικευμένου κώδικα, κάνουν χρήση περαιτέρω βιβλιοθηκών που εισάγουν επιπρόσθετες καθυστερήσεις κατά την εκτέλεση.

Κεφαλαιο 7

Συμπερασματα και Μελλοντικη Εργασια

7.1 Συμπεράσματα

Τη σημερινή εποχή, τα ετερογενή υπολογιστικά συστήματα, γίνονται ολοένα και πιο συνηθισμένα στον κόσμο των υψηλών επιδόσεων. Οι μονάδες γραφικής επεξεργασίας (GPUs) αποτελούν το δεύτερο βασικότερο πλέον συστατικό των ετερογενών συστημάτων, μετά τους κύριους επεξεργαστές. Η εξέλιξή τους σε συσκευές γενικού σκοπού, έδωσε τη δυνατότητα σε προγραμματιστές να χρησιμοποιήσουν το υλικό τους, το οποίο μέχρι και την προηγούμενη δεκαετία αφορούσε αποκλειστικά στην παραγωγή 3Δ γραφικών. Μια πλατφόρμα προγραμματισμού υψηλών επιδόσεων είναι η CUDA, η οποία επιτρέπει των προγραμματισμό των πυρήνων μιας μονάδας γραφικής επεξεργασίας, μέσω της ομώνυμης διεπαφής.

Η παρούσα εργασία αποτελεί μια ολοκληρωμένη υλοποίηση υποδομής στον παραλληλοποιητικό μεταφραστή OMPi, για την στόχευση μονάδων γραφικής επεξεργασίας που βασίζονται στο μοντέλο CUDA. Το εύρος της καλύπτει τόσο τη σκοπιά της διαδικασίας μεταγλώττισης του μεταφραστή, όσο και το τμήμα υποστήριξης εκτέλεσης. Κύριοι στόχοι της εργασίας είναι η διευκόλυνση στη συγγραφή παράλληλων προγραμμάτων, τα οποία χρησιμοποιούν συνοδές συσκευές CUDA για την επιτάχυνση υπολογισμών, καθώς επίσης και η διατήρηση των υψηλών επιδόσεων

που προσφέρεται από το αντίστοιχο προγραμματιστικό μοντέλο. Για την επίτευξη του πρώτου στόχου της εργασίας, απαιτήθηκε η υλοποίηση ενός συνόλου οδηγιών OpenMP, ένα πρότυπο που από τη φύση του επιτρέπει τη συγγραφή ενιαίου κώδικα ο οποίος περιλαμβάνει φόρτωση και εκτέλεση τμημάτων του σε συνοδές υπολογιστικές συσκευές. Για την υποστήριξη των συσκευών τύπου CUDA και συνεπώς τη διατήρηση υψηλών επιδόσεων στους υπολογισμούς, δημιουργήθηκε η συσκευή CUDADEV, μια βιβλιοθήκη υποστήριξης εκτέλεσης, η οποία καθιστά δυνατή την επικοινωνια μεταξύ του κυρίου συστήματος και των συσκευών CUDA.

Από τα αποτελέσματα των πειραμάτων που πραγματοποιήθηκαν, αποδείχθηκε ότι:

- Η αντικατάσταση παράλληλων περιοχών που δημιουργούνται μέσω της οδηγίας *parallel for*, με την οδηγία *target teams distribute parallel for*, έχει τη δυνατότητα να επιτύχει υψηλές επιδόσεις, όταν οι υπολογισμοί βασίζονται σε επαναληπτικούς βρόχους

- Η εισαγωγή ενός τύπου κρυφής μνήμης με αντικατάσταση, κατά τη φόρτωση, στο επίπεδο της συσκευής CUDADEV, μειώνει σε σημαντικό βαθμό το χρόνο επαναλαμβανόμενων φορτώσεων των περιοχών target

- Κατά την εκκίνηση μιας συσκευής CUDA και την αρχικοποίηση της κατάστασής της, είναι δυνατόν να υπάρξει μία μικρή καθυστέρηση, διαφορετική ανά συσκευή CUDA, η οποία όμως, θεωρείται σταθερή και αμελητέα σε σχέση με τις επιδόσεις που επιτυγχάνονται

7.2 Μελλοντική Εργασία

Οι υλοποιήσεις της τρέχουσας εργασίας δοκιμάστηκαν στις υπολογιστικές εκδόσεις CUDA 3.5 και 6.1, στις μονάδες γραφικής επεξεργασίας GT 730 και P40. Στοχεύοντας στην υποστήριξη όσο το δυνατόν μεγαλύτερου εύρους εκδόσεων, η συσκευή CUDADEV χρησιμοποιεί τις βασικότερες λειτουργίες της διεπαφής CUDA driver. Για την περαιτέρω εξέλιξη της συσκευής, είναι δυνατή η χρήση πιο εξειδικευμένων λειτουργιών, που αφορούν μονάδες GPU νεότερων υπολογιστικών εκδόσεων, ώστε να εκμεταλλευθούν στο έπακρο τα νέα χαρακτηριστικά τους. Παραδείγματα τέτοιων χαρακτηριστικών είναι η μνήμη χωρίς αντιγραφή (zero-copy memory) και

η ενοποιημένη μνήμη (unified memory), οι οποίες εισήχθησαν σε νεότερες εκδόσεις CUDA. Η επιλογή του επιπέδου υποστήριξης είναι δυνατόν να γίνει κατά την ανακάλυψη της συσκευής CUDA από το κύριο σύστημα. Με αυτόν τον τρόπο, η βιβλιοθήκη CUDADEV έχει την προοπτική να αποκτήσει μια υβριδική μορφή και να αξιοποιεί ορισμένες δυνατότητες συσκευών κατά περίπτωση.

Μια επιπλέον ιδέα που θα μπορούσε να υλοποιηθεί στο πλαίσιο της μελλοντικής εργασίας, είναι η υποστήριξη μεγαλύτερου συνόλου λειτουργιών OpenMP, στο τμήμα devpart της συσκευής CUDADEV. Μία πρόκληση είναι η υποστήριξη κλειδαριών (locks) στο εσωτερικό των περιοχών target, η οποία συναντάται ως βάση πολλών άλλων οδηγιών. Επιπλέον, η υλοποίηση της λειτουργίας υποβίβασης (reduction) για τις παράλληλες περιοχές εντός μιας περιοχής target, μοιάζει πραγματοποιήσιμη και πολλά υποσχόμενη.

Τέλος, μια προτεινόμενη διαδικασία, σε βάθος χρόνου, είναι ο εμπλουτισμός του συνόλου οδηγιών OpenMP που υποστηρίζονται στον μεταφραστή OMPi και αφορούν συγκεκριμένα τους επιταχυντές γραφικών, καθώς επίσης και η επέκταση των ήδη υλοποιημένων από την τρέχουσα εργασία.

Βιβλιογραφια

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

[2] M. P. Forum, "MPI: A Message-Passing Interface Standard," USA, Tech. Rep., 1994.

[3] NVIDIA, P. Vingelmann, and F. H. Fitzek, "CUDA, release: 10.2.89," 2020. [Online]. Available: https://developer.nvidia.com/cuda-toolkit

[4] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Computing in Science Engineering*, vol. 12, no. 3, pp. 66–73, 2010.

[5] R. M. Stallman and G. D. Community, *GCC 7.0 Manual 1/2 (Volume 1)*. London, GBR: Samurai Media Limited, 2016.

[6] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, ser. CGO '04. USA: IEEE Computer Society, 2004, p. 75.

[7] V. V. Dimakopoulos, E. Leontiadis, and G. Tzoumas, "A portable C compiler for OpenMP v.2.0," in *Proc. EWOMP 2003, 5th European Workshop on OpenMP*, Sept. 2003, pp. 5–11.

[8] R. Farber, *Parallel Programming with OpenACC*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2016.

[9] S. Ohshima, S. Hirasawa, and H. Honda, "OMPCUDA : OpenMP Execution Framework for CUDA based on Omni OpenMP compiler," in *Beyond Loop Level*

Parallelism in OpenMP: Accelerators, Tasking and More, 6th Internationan Workshop on OpenMP, IWOMP 2010, vol. 6132, Jun. 2010, pp. 161–173.

[10] K. Kusano, M. Sato, T. Hosomi, and Y. Seo, "The Omni OpenMP Compiler on the Distributed Shared Memory of Cenju-4," in *WOMPAT '01: Proceedings of the International Workshop on OpenMP Applications and Tools: OpenMP Shared Memory Parallel Programming*, Jul. 2001, pp. 20–30.

[11] S. Lee, S.-J. Min, and R. Eigenmann, "OpenMP to GPGPU: A compiler framework for automatic translation and optimization," in *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2009, Raleigh, NC, USA*, vol. 44, Apr. 2009, pp. 101–110.

[12] D. H. Bailey, *NAS Parallel Benchmarks*. Boston, MA: Springer US, 2011, pp. 1254–1259. [Online]. Available: https://doi.org/10.1007/978-0-387-09766-4_133

[13] G. Noaje, C. Jaillet, and M. Krajecki, "Source-to-Source Code Translator: OpenMP C to CUDA," in *13th IEEE International Conference on High Performance Computing & Communication, HPCC 2011, Banff*, Sept. 2011, pp. 512–519.

[14] S. Cook, *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012.

[15] NVIDIA. Parallel Thread Execution ISA Version 7.2. [Online]. Available: https://docs.nvidia.com/cuda/parallel-thread-execution/

[16] IBM. IBM XL CC++ for Linux, V16.1.1. [Online]. Available: https://www.ibm.com/support/knowledgecenter/SSXVZZ_16.1.1/com.ibm.compilers.linux.doc/compiler.pdf

[17] A. Olofsson, T. Nordström, and Z. Ul-Abdin, "Kickstarting high-performance energy-efficient manycore architectures with Epiphany," in *2014 48th Asilomar Conference on Signals, Systems and Computers*, 2014, pp. 1719–1726.

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

[19] J. M. Diaz, S. Pophale, O. Hernandez, D. E. Bernholdt, and S. Chandrasekaran, "Openmp 4.5 validation and verification suite for device offload," in *Evolving*

OpenMP for Evolving Architectures, B. R. de Supinski, P. Valero-Lara, X. Martorell, S. Mateo Bellido, and J. Labarta, Eds. Cham: Springer International Publishing, 2018, pp. 82–95.

Παραρτημα Α

Παραγομενος Κωδικας Μεταφραστη OMPi

Το παράρτημα αυτό περιλαμβάνει όλα τα σχήματα τα οποία δεν τοποθετήθηκαν στο εσωτερικό ενός κεφαλαίου, για λόγους αναγνωσιμότητας.

```
1 int main() {
2     int x[16];
3     #pragma omp distribute dist_schedule(static,2)
4                 for (i = 0; i < N; i++)
5                         x[i] = i;
6 }
```

Σχήμα Α.1: Παράδειγμα περιοχής distribute στον OMPi (με chunk size)

```
1  int __original_main(int _argc_ignored, char ** _argv_ignored)
2  {
3      int i;
4      unsigned long dist_niters_ = 0, dist_iter_ = 0, dist_fiter_, dist_liter_ = 0;
5      int dist_chid_, dist_TN_ = omp_get_num_teams();
6
7      dist_niters_ = (long) (((16) >= (0)) ? ((16) - (0)) : 0);
8      for (dist_chid_ = omp_get_team_num(); ; dist_chid_ += dist_TN_)
9      {
10         dist_fiter_ = dist_chid_ * (2);
11         if (dist_fiter_ >= dist_niters_)
12           break;
13         dist_liter_ = dist_fiter_ + (2);
14         if (dist_liter_ > dist_niters_)
15           dist_liter_ = dist_niters_;
16
17         for (dist_iter_ = dist_fiter_,
18             i = (0) + dist_fiter_ * 1;
19             dist_iter_ < dist_liter_;
20             dist_iter_++, i += 1)
21           x[i] = i;
22     }
23 }
```

Σχήμα Α.2: Παράδειγμα μετασχηματισμού περιοχής distribute στον OMPi (με chunk size)

```
1 int __original_main(int _argc_ignored, char ** _argv_ignored)
2 # 4 "target_graph.c"
3 {
4   int i, x[ 16];
5
6   /* (l10) #pragma omp target map(tofrom: x) private(i)
7  */
8   {
9     int __ompi_devID = (-1);
10    void * __ort_denv = ort_start_target_data(1, __ompi_devID);
11
12    /* map tofrom */
13    ort_map_tdvar(&x, sizeof(x), &x, 0, 1);
14    struct __dev_struct {
15        __attribute__((aligned(8))) int (* x)[ 16];
16        unsigned long _x_offset;
17      } * _dev_data;
18
19    _dev_data = (struct __dev_struct *) ort_devdata_alloc(
20        sizeof(struct __dev_struct), __ompi_devID);
21    /* maptofrom variables */
22      /* moved to device data environment */
23    _dev_data->x = (int (*)[ 16]) ort_host2med_addr(&x, __ompi_devID);
24    _dev_data->_x_offset = 0;
25    ort_offload_kernel(_kernelFunc0_, (void *) _dev_data,
26        (void *) 0, 0, 0, 0, "target_graph_d00", __ompi_devID,
27        0, 0, _dev_data->x, _dev_data->_x_offset, (void *) 0);
28    /* no other operations */
29    ort_devdata_free(_dev_data, __ompi_devID);
30    /* unmap tofrom */
31    ort_unmap_tdvar(&x, 1);
32    ort_end_target_data(__ort_denv);
33  }
34 }
```

Σχήμα A.3: Παράδειγμα χειρισμού δεδομένων περιοχής target στον OMPi, από τον host

```
double start, end, duration;
for (run = 0, sum = 0.0; run < 5; run++)
{
    start = omp_get_wtime();
    #pragma omp target teams distribute parallel for \\
        num_teams(X) num_threads(32) collapse(2)

        for (i = 0; i < X; i++)
            for (j = 0; j < Y; j++)
            {
                float r = 0.0;
                float p = 2.0;
                int z;
                for (z = 0; z < INNER; z++)
                {
                    p *= 2;
                    r += p;
                }
                x[i][j] = r;
            }

    end = omp_get_wtime();
    duration = end - start;
    sum += duration;
}
```

Σχήμα Α.4: Ψευδοκώδικας θόλωσης Gauss σε C

```
void gaussian_blur(int radius)
{
    /* 1. Read image data */
    /* 2. Store image r, g, b values in 3 corresponding
    one-dimensional arrays */
    /* 3. Perform gaussian blur for each image pixel */
    int i, j;
    #pragma omp target teams distribute parallel for \\
      num_threads(500) num_teams(500) private(i,j) \\
      collapse(2) map(tofrom: red[:250000], green[:250000], blue[:250000])
    for (i = 0 ; i < image_height; i++) {
        for (j = 0 ; j < image_width ; j++) {
            double row, col;
            double redSum = 0, greenSum = 0, blueSum = 0, weightSum = 0;
            for (row = i-radius; row <= i + radius; row++) {
                for (col = j-radius; col<= j + radius; col++) {
                    int x = setBoundary(col,0,width-1);
                    int y = setBoundary(row,0,height-1);
                    int tempPos = y * width + x;
                    double square = (col-j)*(col-j)+(row-i)*(row-i);
                    double sigma = radius*radius;
                    double weight = exp(-square / (2*sigma)) / (3.14*2*sigma);
                    redSum += red[tempPos] * weight;
                    greenSum += green[tempPos] * weight;
                    blueSum += blue[tempPos] * weight;
                    weightSum += weight;
                }
            }
            red[i*width+j] = round(redSum/weightSum);
            green[i*width+j] = round(greenSum/weightSum);
            blue[i*width+j] = round(blueSum/weightSum);
            redSum = greenSum = blueSum = weightSum = 0;
        }
    }
    /* 4. Generate image from new r, g, b values */
    /* 5. Save image */
}
```

Σχήμα A.5: Ψευδοκώδικας θόλωσης Gauss σε C

Παραρτημα Β

Αποτελεσματα Αξιολογησης

Το παράρτημα αυτό περιλαμβάνει όλα τους πίνακες αποτελεσμάτων του κεφαλαίου 6 οι οποίοι δεν τοποθετήθηκαν στο εσωτερικό του, για λόγους αναγνωσιμότητας.

	Parallax	Paragon		Parallax	Paragon
7_test_target_map_pointer.c	RP	RP	simple.c	RP	RP
alltargcases.c	RP	RP	simpletest.c	RP	RP
arrsec-entex-1.c	RP	RP	targetdata1.c	RP	RP
arrsec-ptr-1.c	RP	RP	targetmath.c	RP	RP
arrsec-ptr-2.c	RP	RP	targetsimple.c	RP	RP
arrsec-ptr-3.c	RP	RP	targetTest.c	RP	RP
arrsec-zlas-1.c	RP	RP	targetUpdateTest.c	RP	RP
callfunc5.c	RP	RP	targparallel1.c	RP	RP
changingptr.c	RP	RP	targparfor.c	RP	RP
decl45.c	RP	RP	targteams.c	RP	RP
declaretargetTest.c	RP	RP	targteamsdist.c	RP	RP
declare-v45-1.c	RP	RP	targteamsdistparfor.c	RP	RP
declare-v45-2.c	RP	RP	targteamsdistparforCollapse.c	RP	RP
declinj.c	CF	CF	targteamsdistparforsimd1.c	CP	CP
decltarg_locks.c	CF	CF	targteamsdistsimd1.c	CP	CP
devptr1.c	RP	RP	targupdate-depend.c	RP	RP
devptr2.c	RP	RP	timetarg.c	RP	RP
devptr3.c	RP	RP	v45devmem.c	RP	RP
enterexit1.c	RP	RP	v45runtime.c	RP	RP
enterexit2.c	RP	RP			
enterexit3.c	RP	RP			
ll.c	RP	RP			
mapalways.c	RP	RP			
maprulesv45.c	RP	RP			
partarg.c	RP	RP			
simple1.c	RP	RP			
simple2.c	RP	RP			
simple3.c	RP	RP			

Πίνακας Β.1: Τα αποτελέσματα εκτέλεσης των εσωτερικών δοκιμαστικών προγραμμάτων του OMPi (RP = Runtime Pass, CF = Compilation Fail, CP = Compilation Pass)

Με χρήση κρυφής μνήμης (P40)						
N	**#1**	**#2**	**#3**	**#4**	**#5**	**Avg**
64	0,093	0,090	0,092	0,094	0,093	**0,092**
128	0,096	0,100	0,100	0,098	0,100	**0,099**
256	0,112	0,112	0,109	0,108	0,110	**0,110**
512	0,143	0,142	0,134	0,135	0,134	**0,138**
1024	0,189	0,188	0,188	0,186	0,189	**0,188**
2048	0,291	0,295	0,293	0,291	0,294	**0,293**
4096	0,507	0,513	0,514	0,514	0,505	**0,511**
8192	0,955	0,952	0,957	0,956	0,958	**0,956**
16384	1,851	1,848	1,849	1,858	1,856	**1,852**

Πίνακας Β.2: Μέτρηση επιδόσεων φόρτωσης kernel με χρήση κρυφή μνήμης στο σύστημα Parallax (δευτερόλεπτα)

Χωρίς χρήση κρυφής μνήμης (P40)						
N	**#1**	**#2**	**#3**	**#4**	**#5**	**Avg**
64	0,119	0,117	0,118	0,119	0,119	**0,118**
128	0,155	0,152	0,153	0,152	0,154	**0,153**
256	0,221	0,223	0,223	0,223	0,227	**0,223**
512	0,366	0,371	0,367	0,369	0,364	**0,367**
1024	0,680	0,667	0,670	0,673	0,677	**0,673**
2048	1,286	1,288	1,287	1,286	1,283	**1,286**
4096	2,566	2,563	2,561	2,573	2,576	**2,568**
8192	5,258	5,267	2,257	5,264	5,292	**4,668**
16384	11,066	11,078	11,069	11,023	11,077	**11,063**

Πίνακας Β.3: Μέτρηση επιδόσεων φόρτωσης kernel χωρίς χρήση κρυφή μνήμης στο σύστημα Parallax (δευτερόλεπτα)

Με χρήση κρυφής μνήμης (GT 730)						
N	**#1**	**#2**	**#3**	**#4**	**#5**	**Avg**
64	0,083	0,081	0,081	0,081	0,082	**0,082**
128	0,097	0,097	0,097	0,097	0,098	**0,097**
256	0,128	0,129	0,129	0,128	0,128	**0,128**
512	0,194	0,193	0,193	0,192	0,194	**0,193**
1024	0,331	0,332	0,329	0,331	0,330	**0,331**
2048	0,613	0,615	0,619	0,617	0,616	**0,616**
4096	1,205	1,201	1,211	1,212	1,206	**1,207**
8192	2,420	2,422	2,428	2,419	2,426	**2,423**
16384	4,951	4,921	4,929	4,915	4,900	**4,923**

Πίνακας Β.4: Μέτρηση επιδόσεων φόρτωσης kernel με χρήση κρυφή μνήμης στο σύστημα Paragon (δευτερόλεπτα)

Χωρίς χρήση κρυφής μνήμης (GT 730)						
N	**#1**	**#2**	**#3**	**#4**	**#5**	**Avg**
64	0,158	0,158	0,158	0,158	0,157	**0,158**
128	0,262	0,261	0,261	0,261	0,262	**0,261**
256	0,490	0,488	0,489	0,488	0,490	**0,489**
512	1,012	1,012	1,014	1,008	1,014	**1,012**
1024	2,300	2,314	2,313	2,310	2,299	**2,307**
2048	5,740	5,794	5,658	5,775	5,805	**5,754**
4096	16,208	16,162	16,393	16,330	16,235	**16,266**
8192	33,751	33,628	33,687	33,745	33,732	**33,709**
16384	71,015	72,282	71,630	72,240	71,898	**71,813**

Πίνακας Β.5: Μέτρηση επιδόσεων φόρτωσης kernel χωρίς χρήση κρυφή μνήμης στο σύστημα Paragon (δευτερόλεπτα)

Δημοσιευσεις Συγγραφεα

- I.K. Kasmeridis, V.V. Dimakopoulos, "A General-Purpose Mapper Module for Adaptive OpenMP Runtime Support", in Proc. 4th South-East Europe Design Automation, Computer Engineering, Computer Networks and Social Media Conference (SEEDA-CECNSM 2019), Piraeus, Greece, Sept. 2019

Σύντομο Βιογραφικο

Ο Ηλίας Κασμερίδης γεννήθηκε στην Δράμα το 1994. Εισήχθη στο Τμήμα Μηχανικών Η/Υ και Πληροφορικής του Πανεπιστημίου Ιωαννίνων το 2012, από το οποίο και έλαβε το Δίπλωμά του, το 2018. Τη τρέχουσα περίοδο, παρακολουθεί το πρόγραμμα μεταπτυχιακών σπουδών στο ίδιο τμήμα. Είναι μέλος της Ομάδας Παράλληλης Επεξεργασίας από το 2016. Τα ενδιαφέροντά του αφορούν το πεδίο της παράλληλης επεξεργασίας, των επιταχυντών γραφικών και των μεταγλωττιστών.