Υποστήριξη OpenMP σε συστήματα Android

Αριστείδης Παναγιώτου

Διπλωματική Εργασία

Επιβλέπων: Βασίλειος Δημακόπουλος

Ιωάννινα, Ιούνιος 2025

ΤΜΗΜΑ ΜΗΧ. Η/Υ & ΠΛΗΡΟΦΟΡΙΚΗΣ
ΠΑΝΕΠΙΣΤΗΜΙΟ ΙΩΑΝΝΙΝΩΝ

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
UNIVERSITY OF IOANNINA

Περίληψη

Σχεδόν όλα τα υπολογιστικά συστήματα στις μέρες μας, ανεξαρτήτως της ισχύς τους, είναι παράλληλα συστήματα. Ο προγραμματισμός στα συστήματα αυτά παρουσιάζει περισσότερες δυσκολίες από ότι στα σειριακά. Για αυτό χρειάστηκαν να δημιουργηθούν νέα προγραμματιστικά μοντέλα, όπως το δημοφιλές OpenMP, τα οποία εκμεταλλεύονται τις δυνατότητες τους και πετυχαίνουν καλύτερες επιδόσεις. Οι κινητές συσκευές είναι φορητά παράλληλα συστήματα καθώς αποτελούνται από πολυπύρηνους επεξεργαστές και μπορούν να προγραμματιστούν με όμοιο τρόπο. Στον χώρο των κινητών συσκευών το μεγαλύτερο τμήμα της αγοράς χρησιμοποιεί ως λειτουργικό σύστημα το Android.

Η παρούσα διπλωματική εργασία εστιάζει στην υποστήριξη και στον τρόπο χρήσης του προτύπου OpenMP σε συσκευές Android. Αρχικά γίνεται μια ιστορική έρευνα αναφορικά με το επίπεδο υποστήριξης παράλληλου προγραμματισμού μέσω OpenMP σε όλες τις εκδόσεις του Android NDK όπου διαπιστώνεται ότι η υποστήριξη ήρθε καθυστερημένα και μάλλον αποσπασματικά. Στη συνέχεια παρουσιάζεται ο τρόπος με τον οποίο ο προγραμματιστής μπορεί να κάνει χρήση του μοντέλου για να αναπτύξει μια παράλληλη εφαρμογή, η οποία θα χρησιμοποιεί βιβλιοθήκες γραμμένες σε C. Παράλληλα γίνεται και μια αναφορά στην OpenCL για εκτέλεση κώδικα γενικού σκοπού στην κάρτα γραφικών του κινητού. Το τελευταίο μέρος της διπλωματικής εργασίας αφορά στον μεταφραστή OMPi και στην παραμετροποίησή του προκειμένου να μπορεί να χρησιμοποιηθεί ως μεταφραστής για το Android. Σε αυτό το μέρος, περιγράφουμε πως μπορεί κανείς να παράγει αρχεία που υποστηρίζονται από την αρχιτεκτονική των συσκευών Android.

Λέξεις Κλειδιά: Παράλληλος προγραμματισμός, Συστήματα Android, OpenMP, OpenCL, OMPi

Abstract

Almost all computing systems nowadays, regardless of their power, are parallel systems. Programming in these systems presents more difficulties than in serial systems. This is why new programming models, such as the popular OpenMP, had to be invented, that exploit their potential and achieve better performance. Mobile devices are portable parallel systems as they consist of multi-core processors and can be programmed in a similar way. In the realm of mobile devices, the majority of the market uses Android as its operating system.

This thesis focuses on the support and use of the OpenMP API on Android devices. It begins with a historical investigation into the level of support for parallel programming via OpenMP across all Android versions, revealing that support arrived late with inconsistencies. Next, it presents the way a developer can utilize the API to develop a parallel application while making a reference to OpenCL for general purpose code execution on the device's GPU. The final part of the thesis deals with the OMPi compiler and its configuration so that it can be used as a compiler for Android. In this section, we describe how one can produce files that are supported by the architecture of Android devices.

Keywords: Parallel programming, Android Systems, OpenMP, OpenCL, OMPi

Περιεχόμενα

Κατάλογος Σχημάτων

Κατάλογος Πινάκων

Κεφάλαιο 1. Εισαγωγή

1.1 Εξέλιξη των Η/Υ

Ως γνωστόν, τα τελευταία 80 χρόνια ο κλάδος των Η/Υ είναι ένας από τους ταχύτερα αναπτυσσόμενους. Από τον ENIAC την δεκαετία του ’40, στον πρώτο προσωπικό υπολογιστή της IBM την δεκαετία του ’80 και στα σημερινά κινητά τηλέφωνα (smartphones), είναι εμφανές ότι έχουν γίνει τεράστια και πολύ εντυπωσιακά τεχνολογικά άλματα. Ένα από αυτά τα άλματα που μας επιτρέπουν να έχουμε πρόσβαση σε σημαντική επεξεργαστική ισχύ οπουδήποτε είναι η ανάπτυξη των παράλληλων υπολογιστών.

Ας πάρουμε τα πράγματα από την αρχή. Η οργάνωση των σύγχρονων ηλεκτρονικών υπολογιστών βασίζεται στο μοντέλο που διατυπώθηκε από τον John von Neumann το 1945. Σύμφωνα με αυτό το μοντέλο, υπάρχει μια μονάδα επεξεργασίας η οποία εκτελεί εντολές που έχουν ληφθεί από το τμήμα της μνήμης και έπειτα αποθηκεύει το αποτέλεσμα τους σε αυτό. Στην πιο απλή μορφή αυτού του μοντέλου, η εκτέλεση των εντολών γίνεται σειριακά, δηλαδή η εκτέλεση της επομένης εντολής θα ξεκινήσει αφού ολοκληρωθεί η εκτέλεση της προηγούμενης.

Ένας από του τρόπους για να βελτιωθούν οι επιδόσεις των επεξεργαστών είναι να αυξάνεται ο αριθμός των τρανζίστορ ανά μονάδα επιφάνειας, μειώνοντας το μέγεθος εκάστου, και ταυτόχρονα να αυξάνεται η συχνότητα ρολογιού. Μάλιστα, ο Αμερικάνος επιστήμονας και συνιδρυτής της Intel, Gordon Moore, παρατήρησε ότι κάθε 2 χρόνια διπλασιάζεται ο αριθμός των τρανζίστορ. Η παρατήρηση αυτή ιστορικά έχει μείνει σαν ο νόμος του Moore.

Το πρόβλημα που δημιουργήθηκε στο «κυνήγι» για καλύτερες επιδόσεις με αυτή την προσέγγιση είναι το εξής. Όσο αυξάνεται η συχνότητα, αυξάνεται και η απαιτούμενη ισχύς που απαιτείται. Παράλληλα, λόγω του μικρότερου μεγέθους του κάθε τρανζίστορ, μέρος του ρεύματος που τροφοδοτείται σε αυτό απλά διαρρέει, αυξάνοντας περεταίρω

την καταναλισκόμενη ισχύ. Με την σειρά της, η επιπλέον ισχύς φέρνει και υψηλότερες θερμοκρασίες και κατά συνέπεια οι επιδόσεις δεν καταφέρνουν να αυξηθούν όσο θα περιμέναμε.

Η ιδέα για την λύση σε αυτό το πρόβλημα είναι απλή και υλοποιήθηκε στις αρχές τις δεκαετίας του 2000. Αντί να υπάρχει ένας επεξεργαστής με έναν τόσο ισχυρό και απαιτητικό πυρήνα, να υπάρχουν πολλαπλοί πυρήνες (ενδεχομένως όχι τόσο ισχυροί) σε έναν επεξεργαστή οι οποίοι θα εκτελούν εντολές παράλληλα. Προκειμένου, όμως, να εκμεταλλευτούμε την ύπαρξη πολλαπλών πυρήνων χρειαζόμαστε τα κατάλληλα μοντέλα προγραμματισμού αντί των κλασσικών που εκτελούν τις εντολές σειριακά.

1.2 Παράλληλα Συστήματα

Παράλληλο σύστημα είναι ένα σύστημα που διαθέτει παραπάνω από μία επεξεργαστικές μονάδες, οι οποίες επικοινωνούν μεταξύ τους για να λύσουν κάποιο πρόβλημα που τους δίνεται. Στόχος είναι να εκμεταλλευτούμε την ύπαρξη αυτών των μονάδων, διαιρώντας το πρόβλημα σε μικρότερα και αναθέτοντας τα υποπροβλήματα σε καθεμία από αυτές ώστε να εκτελεσθούν παράλληλα. Έτσι, καταφέρνουμε να γλιτώνουμε χρόνο στην εκτέλεση ενός προγράμματος σε σχέση με ένα σειριακό σύστημα το οποίο θα έπρεπε να εκτελέσει την κάθε εντολή ξεχωριστά.

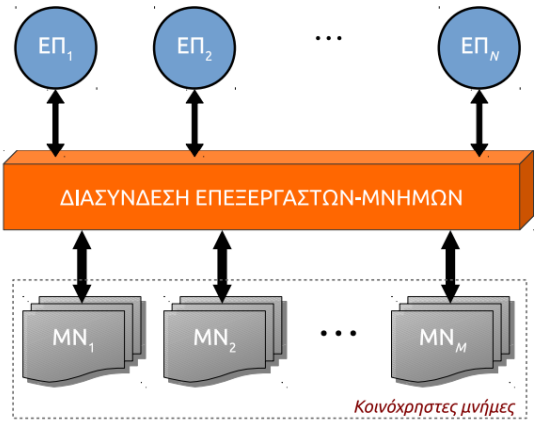
Εδώ αξίζει να σημειώσουμε ότι δεν μπορούν να παραλληλοποιηθούν όλες οι εντολές ενός προγράμματος. Υπάρχουν συγκεκριμένα κομμάτια στον κώδικα που μας δίνουν την δυνατότητα αυτή. Χαρακτηριστικό παράδειγμα κώδικα που ενδείκνυται η παραλληλοποίηση είναι ο βρόγχος for. Ανάλογα την περίπτωση, μπορούμε να θέσουμε προς εκτέλεση την κάθε επανάληψη ή μέρος των επαναλήψεων σε διαφορετική επεξεργαστική μονάδα. Ακόμα, όταν προγραμματίζει κανείς παράλληλα ενδέχεται να τα έχει κάνει όλα σωστά αλλά εντέλει να μην δει κάποια βελτίωση στην ταχύτητα του προγράμματος του ή ακόμη χειρότερα να παρατηρήσει αύξηση του χρόνου επεξεργασίας. Αυτό μπορεί να συμβεί επειδή τα προγραμματιστικά μοντέλα που διαθέτουμε προσθέτουν μια επιπλέον καθυστέρηση (overhead) προκειμένου να εξασφαλισθεί η ορθότητα των αποτελεσμάτων του προγράμματος.

Τη σημερινή εποχή θα βρει κανείς παράλληλους υπολογιστές παντού. Οι επεξεργαστές όλων των καινούργιων κινητών τηλεφώνων και υπολογιστών αποτελούνται από πολλαπλούς πυρήνες. Ακόμα, υπάρχουν οι υπολογιστές συστάδες (compute clusters) και οι υπερυπολογιστές οι οποίοι αποτελούνται από πολλούς πολυπύρηνους επεξεργαστές. Επιπλέον, οι κάρτες γραφικών (GPUs) είναι και αυτές

συσκευές που εκτελούν εντολές παράλληλα καθώς αποτελούνται από μικρούς πυρήνες (CUDA cores) που εκτελούν συγκεκριμένες πράξεις. Συνεπώς, γίνεται εμφανές ότι η ανάγκη για παράλληλη επεξεργασία χρειάζεται παντού πια. Τα παράλληλα συστήματα μπορούν να χωριστούν σε 2 κατηγορίες βάση της οργάνωσης της μνήμης τους, τις οποίες και θα δούμε στα παρακάτω υποκεφάλαια.

1.2.1 Συστήματα Κοινόχρηστης Μνήμης

Στην κατηγορία των συστημάτων κοινόχρηστης μνήμης ανήκουν και οι σύγχρονοι πολυπύρηνοι επεξεργαστές που βρίσκουμε στα κινητά και στους εμπορικούς υπολογιστές, μεταξύ άλλων. Τα συστήματα αυτά απαρτίζονται από ένα πλήθος επεξεργαστικών μονάδων, πυρήνων εντός του ίδιου ολοκληρωμένου κυκλώματος στην περίπτωση ενός κινητού, οι οποίοι προσπελαύνουν μια κοινόχρηστη μνήμη. Η μνήμη αυτή δεν είναι απαραίτητο ότι θα αποτελείται μόνο από ένα τμήμα αλλά μπορεί να είναι και μια συλλογή ομοιών μνημών, οι οποίες θα παρέχουν ένα κοινό χώρο διευθύνσεων στις επεξεργαστικές μονάδες.

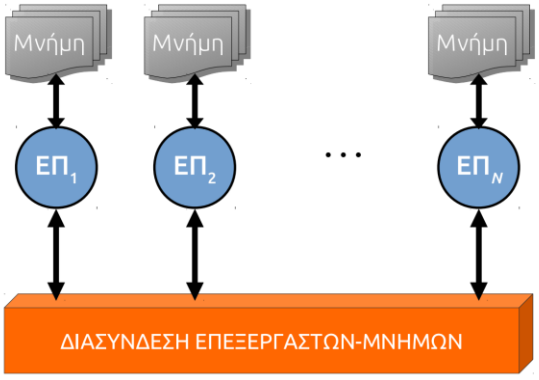


Σχήμα 1.1 Οργάνωση συστήματος με κοινόχρηστες μνήμες

Η επικοινωνία μεταξύ της μνήμης και των επεξεργαστών επιτυγχάνεται μέσω ενός δικτύου διασύνδεσης. Το δίκτυο διασύνδεσης στην απλούστερη μορφή του είναι ένας απλός δίαυλος (bus). Ένας περιορισμός που υπάρχει στην διασύνδεση μέσω διαύλου είναι ότι μόνο ένα ζεύγος επεξεργαστής-μνήμη μπορεί να τον καταλαμβάνει την φορά για να πραγματοποιηθεί η επικοινωνία τους. Συνεπώς, ο δίαυλος είναι χρήσιμος μόνο σε μικρό αριθμό επεξεργαστών. Σε συστήματα μεγαλύτερης κλίμακας την λύση σε αυτό το πρόβλημα φέρνουν τα δίκτυα διακοπτών τα οποία συνδέουν κάθε επεξεργαστή με κάθε μνήμη (και ανάποδα) με την ύπαρξη συγκεκριμένων διαδρομών για κάθε ένωση. Παραδείγματα τέτοιων δικτύων είναι ο διασταυρωτικός διακόπτης (crossbar switch) και το Δίκτυο Δέλτα [1].

1.2.2 Συστήματα Κατανεμημένης Μνήμης

Στην κατηγορία των συστημάτων κατανεμημένης μνήμης συναντούμε συστήματα αρκετά μεγαλύτερης κλίμακας όπως αυτά των υπολογιστικών συστάδων (compute clusters). Τα συστήματα αυτά αποτελούνται από τις επεξεργαστικές μονάδες, που ονομάζονται κόμβοι και από ένα δίκτυο διασύνδεσης. Κάθε κόμβος είναι ένας ανεξάρτητος υπολογιστής, δηλαδή διαθέτει έναν επεξεργαστή και μνήμη τοπικά στην οποία έχει άμεση πρόσβαση. Μέσω του δικτύου διασύνδεσης ένας κόμβος μπορεί να έχει πρόσβαση στην τοπική μνήμη οποιουδήποτε άλλου μόνο με την ανταλλαγή των κατάλληλων μηνυμάτων. Το δίκτυο διασύνδεσης μπορεί να είναι ένα απλό Ethernet δίκτυο, αλλά επειδή θέλουμε αρκετά χαμηλή καθυστέρηση και υψηλές ταχύτητες μεταφοράς δεδομένων μεταξύ των κόμβων, προτιμάται η χρήση πιο προηγμένων δικτύων όπως τα InfiniBand [1].



Σχήμα 1.2 Οργάνωση συστήματος κατανεμημένης μνήμης

1.3 Προγραμματισμός παράλληλων συστημάτων

Είναι γεγονός ότι η ανάπτυξη υλικού που να υποστηρίζει παράλληλη επεξεργασία είχε πιο γρήγορο ρυθμό από την ανάπτυξη κατάλληλων προγραμματιστικών τεχνικών που μας επιτρέπουν να το αξιοποιήσουμε πλήρως. Ακόμη και σήμερα, υπάρχουν παραδείγματα γνωστών προγραμμάτων που τρέχουν μόνο σε ένα πυρήνα ενός επεξεργαστή. Όμως, δεν είναι τυχαίο που επικρατεί αυτή η κατάσταση, καθώς πρέπει να ληφθεί υπόψιν η αρχιτεκτονική του υπολογιστή που δουλεύουμε. Ενώ, για να έχουμε τις επιθυμητές βελτιώσεις και την σωστή εκτέλεση του κώδικα, ο παράλληλος προγραμματισμός παρουσιάζει διάφορες δυσκολίες σε σχέση με τον κλασσικό σειριακό. Υπάρχουν τρία βασικά μοντέλα που μας επιτρέπουν να προγραμματίσουμε μια εφαρμογή παράλληλα.

Το πρώτο είναι το μοντέλο παραλληλισμού δεδομένων (data-parallel model) στο οποίο σε μια δεδομένη χρονική στιγμή όλοι οι επεξεργαστές εκτελούν την ίδια εντολή, ενδεχομένως όχι με τα ίδια δεδομένα. Το πρόβλημα με αυτό το μοντέλο είναι η αυστηρά συγχρονισμένη εκτέλεση των εντολών που μπορεί να μην είναι ο βέλτιστος τρόπος εκτέλεσης για όλα τα συστήματα. Το συγκεκριμένο μοντέλο εφαρμόζεται κυρίως στον προγραμματισμό καρτών γραφικών (GPUs) οι οποίες αποτελούνται από πάρα πολλούς και μικρούς πυρήνες, ικανοί να υποστηρίξουν ένα περιορισμένο αριθμό εντολών.

Το μοντέλο κοινόχρηστου χώρου διευθύνσεων (shared address space model) είναι το δεύτερο προγραμματιστικό μοντέλο που υπάρχει και χρησιμοποιείται στα συστήματα κοινόχρηστης μνήμης. Σύμφωνα με αυτό το μοντέλο κάθε πρόγραμμα αποτελείται από διεργασίες οι οποίες προσπελαύνουν την κοινόχρηστη μνήμη ώστε να αλλάξουν τιμές στις μεταβλητές του προγράμματος. Εδώ δημιουργείται το πρόβλημα της πιθανότητας για ταυτόχρονη πρόσβαση σε μία μεταβλητή από διαφορετικές διεργασίες. Μέσω της βιβλιοθήκης των POSIX threads (pthreads), μπορεί κανείς να δημιουργήσει και να συγχρονίσει νήματα (πιο απλές διεργασίες) με τις προσφερόμενες προγραμματιστικές δομές όπως σημαφόρους (semaphores) ή κλειδαρίες (mutexes) ώστε οι αλλαγές στην μεταβλητή να γίνουν με την σωστή σειρά. Όλα αυτά όμως, αφήνονται στην ευθύνη του προγραμματιστή κάτι που κάνει τον προγραμματισμό αρκετά απαιτητικό. Για αυτό έχουν αναπτυχθεί εργαλεία όπως το OpenMP API (Open Multi-Processing Application Programming Interface) τα οποία κάνουν την δουλεία του προγραμματιστή αρκετά πιο εύκολη απλοποιώντας τον κώδικα. Περισσότερα για το OpenMP θα ειπωθούν στο κεφάλαιο 2.

Το τελευταίο μοντέλο είναι αυτό της μεταβίβασης μηνυμάτων (message-passing model) και χρησιμοποιείται στα συστήματα κατανεμημένης μνήμης. Εδώ, το πρόγραμμα αποτελείται από μια συλλογή ανεξάρτητων διεργασιών δίχως να έχουν κάποια κοινή μεταβλητή. Στο πρότυπο μεταβίβασης μηνυμάτων MPI (Message Passing Interface) υπάρχουν δομές που επιτρέπουν την επικοινωνία μεταξύ των διεργασιών μέσω μηνυμάτων για τις γλώσσες C, C++ και Fortran [1].

1.4 Αντικείμενο της Διπλωματικής Εργασίας

Στο πλαίσιο αυτής της διπλωματικής εργασίας θέσαμε τρεις στόχους γύρω από το θέμα του παράλληλου προγραμματισμού σε συστήματα Android χρησιμοποιώντας το πρότυπο του OpenMP. Ο πρώτος είναι να κάνουμε μια ιστορική αναδρομή στην υποστήριξη του προγραμματισμού C/C++ και του προτύπου στο Android και έπειτα να

δούμε τι υποστηρίζεται σήμερα. Ο δεύτερος είναι να παρουσιάσουμε τον τρόπο που πρέπει να εργαστεί κανείς ώστε να αναπτύξει μια εφαρμογή Android η οποία θα τρέχει παράλληλα. Αυτό θα το πετύχουμε μέσω του προτύπου OpenMP σε γλώσσα C και επιπλέον, θα καταφέρουμε να τρέξουμε κώδικα OpenCL στην GPU του κινητού. Ο τρίτος και τελευταίος στόχος είναι να τροποποιήσουμε τον OMPi, έναν εξατομικευμένο μεταφραστή πηγαίου σε πηγαίο κώδικα (source to source compiler), ώστε να μπορεί να μεταφράσει κώδικα για συσκευές Android. Ο OMPi θα μας δώσει την δυνατότητα μετατροπής C κώδικα με οδηγίες OpenMP σε κώδικα C με κλήσεις νημάτων, γεγονός που απλοποιεί την μεταγλώττιση και διευρύνει την συμβατότητα. Παράλληλα, χάριν αυτής της δυνατότητας του OMPi, επιδιώκουμε να προσθέσουμε υποστήριξη για περισσότερες λειτουργίες.

1.5 Διάρθρωση της Διπλωματικής Εργασίας

Η παρούσα διπλωματική εργασία έχει την εξής οργάνωση.

- Κεφάλαιο 2: Παρουσίαση του προγραμματιστικού προτύπου OpenMP, της λειτουργίας και σύνταξης κάποιων από των δομών που το απαρτίζουν.
- Κεφάλαιο 3: Γενικά στοιχεία για τις συσκευές Android και τον προγραμματισμό τους. Ιστορική αναδρομή στον προγραμματισμό αυτών με C/C++ κώδικα.
- Κεφάλαιο 4: Επεξήγηση του τρόπου εργασίας για την ανάπτυξη εφαρμογών Android που χρησιμοποιών OpenMP και/ή OpenCL. Αναφορά στην δημιουργία βιβλιοθήκης Android.
- Κεφάλαιο 5: Εισαγωγή στον μεταφραστή OMPi. Παραμετροποίηση του ώστε να δημιουργεί βιβλιοθήκες Android. Ένταξη αυτών των βιβλιοθηκών σε κάποια εφαρμογή.
- Κεφάλαιο 6: Παρουσίαση παραδειγμάτων λειτουργίας και σχολιασμός.
- Κεφάλαιο 7: Σύνοψη της διπλωματικής εργασίας και αναφορά σε προσθήκη νέας λειτουργίας.

Κεφάλαιο 2. Το πρότυπο OpenMP

2.1 Εισαγωγή στο OpenMP

Όπως αναφέραμε και στο υποκεφάλαιο 1.3 το OpenMP είναι μια διεπαφή προγραμματισμού εφαρμογών, η οποία επιτρέπει στον προγραμματιστή να προγραμματίσει μια εφαρμογή παράλληλα σε συστήματα κοινόχρηστης μνήμης και υποστηρίζεται στις γλώσσες C, C++ και Fortran. Αποτελείται από οδηγίες προς τον μεταφραστή, ρουτίνες βιβλιοθήκης και μεταβλητές περιβάλλοντος που επηρεάζουν την εκτέλεση του προγράμματος κατά τον χρόνο εκτέλεσης.

Το 1997 ήρθε στη δημοσιότητα η πρώτη έκδοση του OpenMP για την γλώσσα Fortran, ενώ την επόμενη χρονιά για τις γλώσσες C και C++. Οι δύο ανεξάρτητες εκδόσεις συνέχισαν να λαμβάνουν ενημερώσεις ξεχωριστά μέχρι που ενοποιήθηκαν το 2005 με την έκδοση 2.5 του προτύπου OpenMP. Μέχρι εκείνη την στιγμή το OpenMP ήταν αρκετά καλό στο να παραλληλοποιεί συχνούς βρόγχους με γνωστό αριθμό επαναλήψεων (for loops). Παρόλα αυτά, υπάρχουν εφαρμογές που μπορεί να αποτελούνται είτε από αναδρομικές κλήσεις είτε από υπολογισμούς που μπορεί να προκύψουν κατά την διάρκεια της εκτέλεσης. Προκείμενου να αντιμετωπιστεί αυτός ο περιορισμός, το OpenMP με την έκδοση 3.0 το 2008 εισήγαγε την δομή των tasks, τα οποία είναι ένα κομμάτι κώδικα που μπορεί να εκτελεστεί από οποιοδήποτε νήμα, οποιαδήποτε στιγμή.

Οι εκδόσεις 4.0 και 4.5 του προτύπου το 2013 και 2015 αντίστοιχα, πρόσθεσαν νέες ιδιότητες όπως το thread affinity καθώς και υποστήριξη για φόρτωση (offloading) και εκτέλεση κώδικα σε επιταχυντές και κάρτες γραφικών μέσω των οδηγιών target. Οι εκδόσεις μετέπειτα έφεραν όλο και λιγότερες προσθήκες αλλά κυρίως διάφορες βελτιώσεις. Η τελευταία έκδοση είναι η 6.0 που ανακοινώθηκε το 2024.

2.2 Προγραμματιστικό Μοντέλο του OpenMP

Το OpenMP βασίζεται στο μοντέλο fork-join με χρήση νημάτων για να πετύχει την παραλληλία σε συστήματα κοινόχρηστης μνήμης. Υπάρχει ένα αρχικό νήμα (master thread) το οποίο εκτελεί τον κώδικα σειριακά μέχρι να συναντήσει μία παράλληλη περιοχή. Τη στιγμή εκείνη δημιουργεί όσα νήματα ζητηθούν και αφού εκτελέσουν την παράλληλη περιοχή, ο έλεγχος επιστρέφει στο αρχικό νήμα και συνεχίζεται η εκτέλεση

σειριακά. Ο αριθμός των νημάτων που θα δημιουργηθούν μπορεί να καθοριστεί είτε στατικά μέσω μιας μεταβλητής περιβάλλοντος είτε δυναμικά από τον προγραμματιστή. Η παραπάνω διαδικασία θα επαναληφθεί για όσες παράλληλες περιοχές έχουν οριστεί στον κώδικα.

2.3 Οδηγίες OpenMP

Ένα μεγάλο πλεονέκτημα του OpenMP είναι η δυνατότητα ο αρχικός σειριακός κώδικας ενός προγράμματος να παραμένει σχεδόν αναλλοίωτος. Αυτό πετυχαίνετε με την χρήση οδηγιών (directives) τα οποία όχι μόνο πιάνουν ελάχιστες γραμμές αλλά και μπορούν να αγνοηθούν από έναν κλασσικό σειριακό μεταφραστή ώστε το πρόγραμμα να λειτουργεί ακριβώς όπως σχεδιάστηκε. Στα επόμενα υποκεφάλαια θα δούμε κάποιες από τις πιο συχνές οδηγίες του προτύπου για τις γλώσσες C και C++. Τέλος, για να πραγματοποιηθεί η μετάφραση ενός προγράμματος παράλληλα με την χρήση OpenMP πρέπει να περαστεί στον μεταφραστή το κατάλληλο όρισμα. Για τους γνωστούς GCC και Clang είναι το -fopenmp όπως φαίνεται παρακάτω:

```
gcc -fopenmp my_program.c
```

2.3.1 Σύνταξη Οδηγιών

Το συντακτικό που ακολουθούν όλες οι οδηγίες είναι το εξής:

```
#pragma omp construct [clause [clause] …] <new-line>
```

Όλες οι οδηγίες αρχίζουν υποχρεωτικά με το *#pragma omp* και έπειτα ακολουθεί κάποιο construct όπου είναι το όνομα της οδηγίας. Προαιρετικά, μπορεί να υπάρχουν μετά φράσεις οδηγιών οι οποίες καθορίζουν τις συνθήκες υπό τις οποίες θα εκτελεστεί η οδηγία. Εάν δεν έχει τοποθετηθεί καμία φράση , τότε οι συνθήκες αυτές καθορίζονται στον χρόνο εκτέλεσης του προγράμματος.

2.3.2 Η οδηγία parallel

Ενδεχομένως η πιο σημαντική οδηγία του προτύπου OpenMP είναι η οδηγία parallel:

```
#pragma omp parallel [clause [clause] …] <new-line>
<structured-block>
```

Με την χρήση της ορίζεται μια παράλληλη περιοχή, δηλαδή ένα δομημένο τμήμα κώδικα (εγκλεισμένο σε άγκιστρα) το οποίο θα εκτελεσθεί από πολλά νήματα που θα δημιουργήσει το αρχικό (master thread). Ο αριθμός των νημάτων μπορεί να καθοριστεί είτε από κάποια φράση είτε από το περιβάλλον εκτέλεσης. Προκειμένου να βεβαιωθούμε ότι στο τέλος της παράλληλης περιοχής θα έχουν τελειώσει την εκτέλεση τους όλα τα νήματα, υπονοείται ότι υπάρχει ένα φράγμα (barrier) που αποτρέπει κάποιο νήμα να συνεχίσει παρακάτω. Έτσι, μόνο όταν ολοκληρωθεί η εκτέλεση όλων, ο έλεγχος θα επιστρέψει στο αρχικό νήμα το οποίο θα καταστρέψει τα υπόλοιπα και θα συνεχίσει την εκτέλεση σειριακά, ακριβώς όπως περιγράφεται από το μοντέλο fork-join που είδαμε νωρίτερα.

2.3.3 Οδηγίες διαμοιρασμού εργασίας

Εντός μιας παράλληλης περιοχής, δίνεται η δυνατότητα στον προγραμματιστή να διαμοιράσει τον φόρτο εργασίας ανάμεσα στα νήματα. Αυτό επιτυγχάνεται με τις οδηγίες for, single και sections ενώ στο τέλος αυτών υπονοείται barrier για τον συγχρονισμό των νημάτων.

- **for**: Η οδηγία αυτή συναντάται πάντα πριν ένα βρόγχο for και είναι υπεύθυνη για την κατανομή των επαναλήψεων του βρόγχου στα νήματα. Αξίζει να σημειωθεί ότι μπορεί να συνδυασθεί με την εντολή parallel για διευκόλυνση του προγραμματιστή. Η σύνταξη της και με τους 2 τρόπους φαίνεται παρακάτω.

```
#pragma omp for [clause [clause] …] <new-line>
<for-loop>
```

```
#pragma omp parallel for [clause [clause] …] <new-line>
<for-loop>
```

- **single/master:** Το πρώτο νήμα που θα συναντήσει την οδηγία single θα εκτελέσει το τμήμα του κώδικα που ορίζεται από κάτω. Τα υπόλοιπα νήματα απλά το αγνοούν και περιμένουν μέχρι να ολοκληρωθεί η εκτέλεση στο φράγμα που υπονοείται. Μια παραλλαγή της οδηγίας single είναι η master. Σε αυτήν το τμήμα κώδικα που ακολουθεί, εκτελείται μόνο από το αρχικό νήμα και δεν υπονοείται φράγμα στο τέλος. Οι εντολές αυτές συντάσσονται ως εξής:

```
#pragma omp [single | master] [clause [clause] …] <new-line>
<structured-block>
```

- **sections:** Αυτή η οδηγία είναι χρήσιμη όταν υπάρχει η ανάγκη για να παραλληλοποιήσουμε εργασίες που είναι άσχετες μεταξύ τους. Εντός του τμήματος κώδικα που την ακολουθεί υπάρχουν δηλώσεις της οδηγίας **section** που ορίζουν μια εργασία στο δομημένο τμήμα κώδικα που υπάρχει αμέσως μετά. Η σύνταξη της είναι η εξής:

```
#pragma omp sections [clause [clause] …] <new-line>
{
    #pragma omp section
    <structured-block>
    …
}
```

2.3.4 Φράσεις Οδηγιών

Οι φράσεις των οδηγιών τοποθετούνται μετά το όνομα της οδηγίας και μπορούν να είναι πάνω από μία στο πλήθος, όπως φαίνεται στην γενική μορφή που δώσαμε στο υποκεφάλαιο 2.3.1. Χρησιμοποιούνται για να καθορίσουν τον τρόπο εκτέλεσης της οδηγίας και κάποιες από αυτές είναι οι παρακάτω.

- **num_threads(**πλήθος νημάτων**):** Χρησιμοποιείται για να ορίσει τον αριθμό των νημάτων που θα δημιουργηθούν και θα εκτελέσουν μια περιοχή παραλληλίας.
- **nowait:** Επιτρέπει στα νήματα να αγνοήσουν κάποιο φράγμα που ενδέχεται να υπάρχει κατά το τέλος της εκτέλεσης με αποτέλεσμα να μην συγχρονίζονται.
- **reduction(**πράξη : λίστα μεταβλητών**):** Η φράση αυτή διευκολύνει αρκετά τον προγραμματιστή καθώς αναλαμβάνει να δημιουργήσει τοπικά αντίγραφα μιας κοινόχρηστης μεταβλητής στα νήματα χωρίζοντας έτσι τον υπολογισμό σε μικρότερα τμήματα για να παραλληλοποιηθεί. Με την ολοκλήρωση της πράξης αποθηκεύει το αποτέλεσμα στην αντίστοιχη κοινόχρηστη μεταβλητή του αρχικού νήματος. Αν θέλουμε να εφαρμοστεί η πράξη σε περισσότερες από μια μεταβλητή, αρκεί απλά να τις χωρίσουμε με κόμμα ενώ οι διαθέσιμες πράξεις είναι οι +, -, *, &, |, ^, && και ||.
- **schedule(**τύπος [, μέγεθος κόκκου]**):** Χρησιμοποιούμε την φράση schedule σε συνδυασμό με την οδηγία for για να ορίσουμε τον τρόπο που θα διαμοιραστούν

οι επαναλήψεις ενός βρόχου for. Επιπλέον, προαιρετικά, ορίζουμε το μέγεθος του κόκκου παραλληλίας (granularity), δηλαδή το πλήθος των επαναλήψεων που θα διαμοιράζεται σε κάθε νήμα τη φορά. Ο τύπος μπορεί να είναι ένας εκ των παρακάτω.

- **static:** Οι επαναλήψεις διασπώνται στατικά σε τμήματα μεγέθους όσο και ο κόκκος παραλληλίας και ανατίθενται κυκλικά (round-robin) στα νήματα. Αν δεν έχει καθοριστεί το μέγεθος του κόκκου τότε το σύνολο των επαναλήψεων χωρίζεται σε τόσα ισομεγέθη τμήματα όσα και το πλήθος των νημάτων.
- **dynamic:** Η διάσπαση των επαναλήψεων γίνεται όμοια με την στατική μόνο που ο διαμοιρασμός των τμημάτων γίνεται δυναμικά σε κάθε νήμα. Δηλαδή κάθε νήμα θα πάρει το επόμενο τμήμα όταν είναι διαθέσιμο και το ζητήσει. Αν το μέγεθος του κόκκου δεν είναι ορισμένο, τότε το μέγεθος των τμημάτων είναι ίσο με 1.
- **guided:** Όμοια με το dynamic αλλά η διαφορά είναι ότι το επόμενο τμήμα που θα δοθεί προς εκτέλεση είναι μειωμένο εκθετικά από το προηγούμενο. Η μείωση γίνεται βάση του πηλίκου υπολειπόμενων επαναλήψεων προς το πλήθος των νημάτων αλλά όχι μικρότερου του μεγέθους κόκκου εκτός ίσως από το τελευταίο τμήμα.
- **runtime:** Στον συγκεκριμένο τύπο διαμοιρασμού, ο τρόπος χρονοδρομολόγησης ορίζεται από την τιμή της μεταβλητής περιβάλλοντος `OMP_SCHEDULE`.

2.3.5 Φράσεις διαμοιρασμού δεδομένων

Οι φράσεις διαμοιρασμού δεδομένων χρησιμοποιούνται για να δηλώσουν τον τρόπο διαμοιρασμού μίας ή περισσότερων μεταβλητών μεταξύ των νημάτων και είναι οι παρακάτω:

- **shared(**λίστα μεταβλητών**):** Ορίζει τα αντικείμενα της λίστας μεταβλητών ως κοινόχρηστα.
- **private(**λίστα μεταβλητών**):** Οι μεταβλητές της λίστας είναι ιδιωτικές καθώς δημιουργείται από ένα αντίγραφο για κάθε νήμα.
- **firstprivate(**λίστα μεταβλητών**):** Όμοια με την φράση private απλά γίνεται αρχικοποίηση των μεταβλητών με την τιμή των αντίστοιχων που βρίσκονται εκτός της παράλληλης περιοχής.

- **lastprivate(**λίστα μεταβλητών**):** Το lastprivate λειτουργεί όπως και το private, με την μόνη διαφορά ότι στο τέλος της περιοχής παραλληλίας οι τιμές των αντικείμενων της λίστας μεταβλητών ενημερώνονται.
- **default(**λίστα μεταβλητών**):** Καθορίζει την προκαθορισμένη πολιτική διαμοιρασμού με διαθέσιμες επιλογές να είναι οι shared, firstprivate, private, none. Με το none όσες δεν ορίζονται ρητά, προκαλούν συντακτικό λάθος κατά τη μετάφραση αναγκάζοντας τον προγραμματιστή να ορίσει ρητά τα πάντα.

2.3.6 Ρουτίνες βιβλιοθήκης χρόνου εκτέλεσης

Μέσω του αρχείου omp.h, το OpenMP προσφέρει ορισμένες συναρτήσεις που μπορεί να χρησιμοποιήσει ο προγραμματιστής. Μερικές από αυτές είναι οι εξής:

- **int omp_get_num_threads(**void**):** Επιστρέφει το πλήθος των νημάτων που χρησιμοποιούνται στην τρέχουσα ομάδα.
- **void omp_set_num_threads(int** num_threads**):** Ορίζει το πλήθος των νημάτων που θα χρησιμοποιηθούν σε επερχόμενες παράλληλες περιοχές. Εξαίρεση αποτελούν οι παράλληλες περιοχές που χρησιμοποιούν τη φράση num_threads() η οποία υπερισχύει.
- **int omp_get_thread_num(**void**):** Επιστρέφει το αριθμητικό αναγνωριστικό του τρέχοντος νήματος της ομάδας.
- **int omp_get_num_procs(**void**):** Επιστρέφει το πλήθος των διαθέσιμων πυρήνων που βλέπει το σύστημα.
- **double omp_get_wtime(**void**):** Την στιγμή που καλείται, επιστρέφει το χρόνο (wall clock) σε δευτερόλεπτα. Χρήσιμη όταν θέλουμε να χρονομετρήσουμε πόση ώρα διαρκεί ένας υπολογισμός, καλώντας 2 φορές, μία πριν και μια μετά τον υπολογισμό.

2.4 Συσκευές OpenMP (Devices)

Όπως έχουμε δει μέχρι τώρα, ένα πρόγραμμα περικλείεται από μια υπονοούμενη περιοχή παραλληλίας, γονέα των υπόλοιπων περιοχών, στην οποία συμμετέχει μόνο το κύριο νήμα. Αναφερόμαστε στο κύριο σύστημα το οποίο εκτελεί την περιοχή και η κύρια επεξεργαστική της μονάδα (host device) μπορεί ανά πάσα στιγμή να αποστείλει δεδομένα και κώδικα σε μια συσκευή, με τη χρήση κατάλληλων οδηγιών OpenMP, ώστε να επιταχύνει την εκτέλεση του προγράμματος. Ειδικότερα, ο κώδικας που δίνεται σε μία

συσκευή προς εκτέλεση μπορεί και ο ίδιος να περιέχει παραλληλισμό εκφρασμένο μέσω οδηγιών OpenMP, επομένως μπορεί να γίνει παράλληλη εκτέλεση και εντός της συσκευής, εφόσον υποστηρίζεται.

Οι συσκευές ενδέχεται να προτιμούνται έναντι ενός συστήματος γενικής χρήσης λόγω της ταχύτητάς τους σε συγκεκριμένες πράξεις ή λόγω άλλων ιδιαιτεροτήτων τους. Ένα παράδειγμα συσκευής είναι μια κάρτα γραφικών, η οποία έχει την ιδιότητα να εκτελεί πράξεις κινητής υποδιαστολής γρήγορα και μαζικά, ή ένα εικονικό μηχάνημα το οποίο έχουμε παραμετροποιήσει με τα επιθυμητά χαρακτηριστικά. Στην εργασία αυτή μας ενδιαφέρει η κάρτα γραφικών των συσκευών Android.

2.4.1 Η οδηγία target

Η οδηγία αυτή μας επιτρέπει να εκτελέσουμε κώδικα σε κάποια συσκευή. Οι εντολές του προγράμματος που βρίσκονται μέσα στο μπλοκ κώδικα που την ακολουθεί εκτελούνται στην συσκευή και όχι στον κεντρικό επεξεργαστή. Έχει την εξής σύνταξη:

```
#pragma omp target [clause [clause] …] <new-line>
<structured-block>
```

Πιο αναλυτικά, αυτό που συμβαίνει είναι ότι το κύριο νήμα, όταν συναντά την οδηγία target, καλείται να αποστείλει το δομημένο τμήμα κώδικα στη συσκευή. Παράλληλα, δημιουργείται για ολόκληρη τη συσκευή ένα περιβάλλον δεδομένων (device data environment), το οποίο περιλαμβάνει όλα τα απαραίτητα δεδομένα που κληρονομήθηκαν από το κύριο πρόγραμμα. Το ενιαίο σύνολο του δομημένου τμήματος και του περιβάλλοντος δεδομένων το ονομάζουμε ως πυρήνα (kernel). Εντός της συσκευής, τα νήματα της ενεργοποιούνται και όλα εκτός του κυρίου νήματος βρίσκονται σε αναμονή. Η συμπεριφορά του κυρίου νήματος της συσκευής ταυτίζεται με αυτή του κυρίου νήματος στο host device. Δηλαδή δημιουργεί και χειρίζεται ομάδες νημάτων, στις οποίες συμμετέχει, ενώ επιπλέον εκτελεί κανονικά σειριακό κώδικα και οδηγίες OpenMP. Συνεπώς, όπως και στο host, τα υπόλοιπα νήματα της συσκευής ενεργοποιούνται όταν συναντάται μια περιοχή παραλληλίας εντός της οδηγίας target.

2.4.2 Η οδηγία target data

Ο προγραμματιστής συχνά χρειάζεται να εισάγει διαδοχικές περιοχές target στο πρόγραμμά του. Όταν αυτές χρησιμοποιούν κοινές μεταβλητές, είναι προτιμότερο να τις

αποστείλουμε στην συσκευή μια φορά, παραλείποντας, προσωρινά, την εκτέλεση κώδικα. Έτσι, ορίζοντας περιοχές target data στο εσωτερικό του προγράμματός μας, μπορούμε να ελαττώσουμε σημαντικά τον αριθμό των αναγνώσεων/εγγραφών μνήμης που απαιτούνται για την επανειλημμένη δημιουργία περιβάλλοντος δεδομένων. Το κύριο νήμα, όταν συναντά την οδηγία target data, δημιουργεί ένα νέο περιβάλλον δεδομένων στη συσκευή. Αυτό θα κληρονομηθεί από όλα τα περιβάλλοντα δεδομένων των περιοχών target που είναι τοποθετημένα στο εσωτερικό του δομημένου τμήματος. Η σύνταξή της οδηγίας φαίνεται παρακάτω:

```
#pragma omp target data [clause [clause] …] <new-line>
<structured-block>
```

2.4.3 Φράσεις οδηγιών target/target data

Εντός των περιοχών target μπορούν να χρησιμοποιηθούν φράσεις που καθορίζουν διάφορες λειτουργίες. Κάποιες μας επιτρέπουν να επιλέξουμε ποια συσκευή θα χρησιμοποιηθεί, τον τρόπο και χρόνο που θα ενημερωθούν τα δεδομένα του περιβάλλοντος της συσκευής, καθώς επίσης και σε ποια κατεύθυνση (προς/από το κύριο πρόγραμμα). Μερικές από αυτές τις φράσεις είναι:

- **device(**έκφραση ακεραίων**):** Ορίζει την συσκευή που θα γίνει η αποστολή δεδομένων ή/και κώδικα (target/target data).
- **map(**[τύπος :] λίστα μεταβλητών**)**. Η φράση map χρησιμοποιείται για να ορίσει τον τρόπο με τον οποίο θα γίνει η αντιστοίχιση/μεταφορά των μεταβλητών μεταξύ host και device. Οι βασικότεροι τύποι που χρησιμοποιούνται στην πράξη είναι οι παρακάτω:
  - **to:** Η τιμή της κάθε μεταβλητής της λίστας ανατίθεται από το host σύστημα στην μνήμη της συσκευής πριν την εκτέλεση του τμήματος κώδικα.
  - **from:** Η τιμή της κάθε μεταβλητής της λίστας αντιγράφεται από την μνήμη της συσκευής πίσω στην μνήμη του host συστήματος μετά την εκτέλεση του τμήματος κώδικα.
  - **tofrom:** Συνδυασμός των δυο παραπάνω.
  - **alloc:** Δημιουργείται μια αντιστοιχία δεδομένων ανάμεσα σε βασικό σύστημα και συσκευή, δίχως να πραγματοποιείται κάποια μεταφορά δεδομένων.

Κεφάλαιο 3. Συστήματα Android

3.1 Εισαγωγή

Το Android είναι σήμερα το πιο διαδεδομένο λειτουργικό σύστημα για φορητές συσκευές παγκοσμίως, χρησιμοποιούμενο σε εκατομμύρια smartphones, tablets, smartwatches και άλλες έξυπνες συσκευές. Η επιτυχία του οφείλεται τόσο στην ανοιχτή φύση του κώδικα (open-source) όσο και στη συνεχή του εξέλιξη, η οποία ανταποκρίνεται στις διαρκώς μεταβαλλόμενες ανάγκες της αγοράς και των χρηστών.

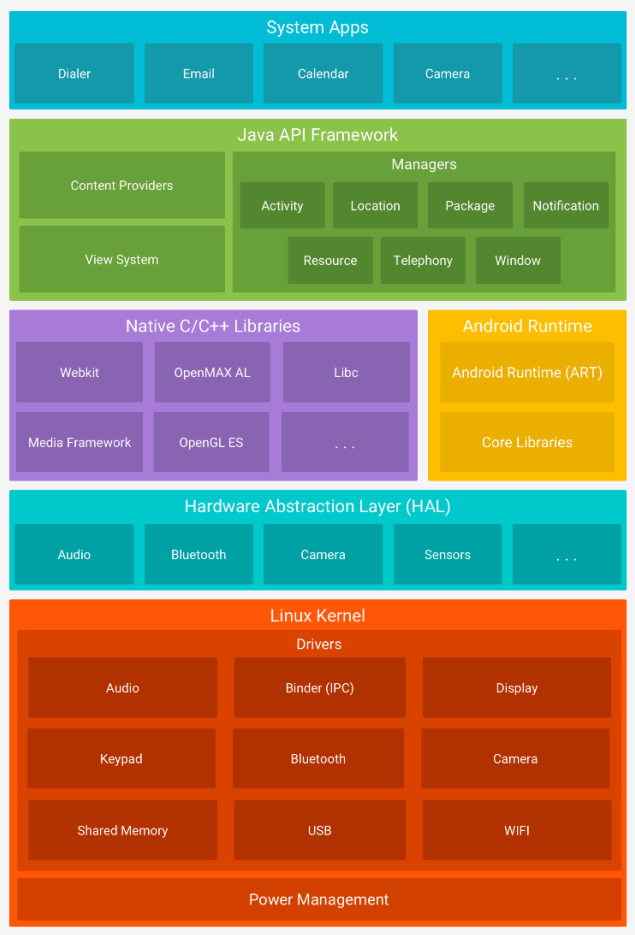
Η αρχιτεκτονική ενός τυπικού συστήματος Android βασίζεται σε διαδοχικά επίπεδα λογισμικού που συνεργάζονται και επικοινωνούν μεταξύ τους. Στον πυρήνα του συστήματος βρίσκεται ο Linux Kernel, ο οποίος προσφέρει βασικές υπηρεσίες όπως διαχείριση μνήμης και διεργασιών, ασφάλεια συστήματος και διασύνδεση με το υλικό (drivers). Ο πυρήνας αυτός έχει υποστεί ειδικές προσαρμογές ώστε να καλύπτει τις απαιτήσεις κινητών συσκευών, όπως την αποδοτικότερη κατανάλωση ενέργειας και τη διαχείριση ασύρματων επικοινωνιών.

Πάνω από τον πυρήνα, το επίπεδο Hardware Abstraction Layer (HAL) λειτουργεί ως γέφυρα μεταξύ του λογισμικού και των διάφορων εξαρτημάτων του υλικού, όπως οι κάμερες, οι αισθητήρες, το GPS και οι ασύρματες επικοινωνίες. Το HAL επιτρέπει στο Android να αλληλεπιδρά με διαφορετικά κομμάτια hardware με έναν ομοιογενή τρόπο, χωρίς το λογισμικό να πρέπει να γνωρίζει τις ιδιαιτερότητες κάθε υλοποίησης.

Ακολουθεί το επίπεδο των Libraries και του Android Runtime (ART). Σε αυτό το σημείο βρίσκονται κρίσιμες βιβλιοθήκες γραμμένες σε C/C++ και Java, οι οποίες προσφέρουν λειτουργίες όπως την απόδοση γραφικών (OpenGL), την περιήγηση στο διαδίκτυο (WebKit) και τη διαχείριση βάσεων δεδομένων (SQLite). Παράλληλα, το Android Runtime περιέχει το Dalvik Virtual Machine (DVM) το οποίο εκτελεί τις εφαρμογές που είναι γραμμένες σε γλώσσες όπως η Java ή η Kotlin.

Το Application Framework παρέχει ένα πλούσιο σύνολο από APIs και υπηρεσίες, τα οποία χρησιμοποιούνται στην ανάπτυξη των εφαρμογών για την διαχείριση των ειδοποιήσεων, των πόρων και την «ζωή» της εφαρμογής στο παρασκήνιο. Στην κορυφή της αρχιτεκτονικής βρίσκονται οι ίδιες οι εφαρμογές, είτε αυτές που έρχονται προεγκατεστημένες από το σύστημα, όπως η εφαρμογή τηλεφώνου ή μηνυμάτων, είτε

εκείνες που εγκαθιστά ο χρήστης μέσω των ηλεκτρονικών καταστημάτων (πχ Google Play Store) [2].



Σχήμα 3.1 Η Αρχιτεκτονική του Android [2]

Όσον αφορά το υλικό των συσκευών Android, οι περισσότεροι κατασκευαστές επιλέγουν επεξεργαστές βασισμένους σε αρχιτεκτονική ARM, όπως τα SoC (System on Chip) της Qualcomm (Snapdragon), της MediaTek, της Samsung (Exynos) και της Google (Tensor). Τα SoC αυτά συνδυάζουν σε ένα ενιαίο κύκλωμα τον κεντρικό επεξεργαστή (CPU), τον επεξεργαστή γραφικών (GPU), το μόντεμ επικοινωνιών και άλλους επιμέρους επεξεργαστές, όπως NPU (Neural Processing Unit) για επιτάχυνση τεχνητής νοημοσύνης. Η μνήμη RAM και οι αποθηκευτικοί χώροι βασίζονται σε τεχνολογίες όπως LPDDR5 και UFS 3.1 ή 4.0, αντίστοιχα, προσφέροντας υψηλές ταχύτητες και ενεργειακή αποδοτικότητα.

3.2 Προγραμματισμός στο Android

Ο προγραμματισμός στο Android έχει εξελιχθεί σημαντικά τα τελευταία χρόνια, προσφέροντας στους προγραμματιστές ένα ισχυρό και ευέλικτο οικοσύστημα εργαλείων και τεχνολογιών. Ο προγραμματισμός εφαρμογών ως επι το πλείστον γίνεται σε Java και Kotlin. Το προτιμητέο περιβάλλον ανάπτυξης εφαρμογών Android είναι το Android Studio, μια ολοκληρωμένη πλατφόρμα που βασίζεται στο IntelliJ IDEA και προσφέρει εργαλεία για συγγραφή κώδικα, σχεδίαση διεπαφών χρήστη, debugging και ανάλυση απόδοσης εφαρμογών. Το Android Studio υποστηρίζει τόσο ανάπτυξη σε υψηλού επιπέδου γλώσσες όπως Java και Kotlin, όσο και υλοποιήσεις σε χαμηλότερο επίπεδο με τη χρήση κώδικα C και C++.

Στην περίπτωση όμως που υπάρχει απαίτηση για απευθείας πρόσβαση σε χαρακτηριστικά του hardware ή για αυξημένες επιδόσεις (π.χ. σε μια εφαρμογή επεξεργασίας εικόνας), το Android παρέχει το NDK (Native Development Kit). Το NDK επιτρέπει τη συγγραφή τμημάτων της εφαρμογής σε C ή C++, δίνοντας τη δυνατότητα αξιοποίησης βελτιστοποιημένου κώδικα και της χρήσης υπαρχουσών native (C/C++) βιβλιοθηκών. Γίνεται εμφανές ότι η όποια δυνατότητα χρήσης OpenMP παρέχεται μόνο σε επίπεδο NDK.

Η δημιουργία και διαχείριση του native κώδικα γίνεται με τη βοήθεια εργαλείων όπως το ndk-build, το οποίο βασίζεται στο GNU Make και παρέχει έναν σχετικά απλό τρόπο οργάνωσης και μεταγλώττισης C/C++ αρχείων. Ωστόσο, τα τελευταία χρόνια προτιμάται η χρήση του CMake, ενός σύγχρονου και πιο ευέλικτου εργαλείου που προσφέρει καλύτερη ενσωμάτωση με το Android Studio και επιλογών μεταγλώττισης μέσω ενός απλού αρχείου κειμένου, το CMakeLists.txt [3].

Σημαντικό ρόλο στη διαδικασία ανάπτυξης παίζει και το Gradle, το βασικό build system του Android Studio. Μέσω του Gradle, διαχειρίζεται κανείς όλα τα στάδια παραγωγής της εφαρμογής: από τη μεταγλώττιση του κώδικα και την ένωση των διαφόρων πόρων, μέχρι την υπογραφή και τη δημιουργία του τελικού πακέτου APK. Το Gradle προσφέρει ευελιξία στην ανάπτυξη εφαρμογών που περιλαμβάνουν τόσο Java/Kotlin όσο και native κώδικα, ενώ μέσω αρχείων όπως το build.gradle μπορεί να καθοριστούν διάφοροι παράμετροι όπως η χρήση του NDK, η χρήση των εργαλείων CMake ή ndk-build, η ελάχιστη έκδοση Android για την οποία θα φτιάξουμε την εφαρμογή και πολλές άλλες.

3.3 Ιστορικά στοιχεία

Το Android NDK εμφανίστηκε για πρώτη φορά τον Ιούνιο του 2009 με την έκδοση r1. Από τότε, με κάθε επόμενη έκδοση, έχει δεχθεί πάρα πολλές προσθήκες και αλλαγές, είτε σημαντικές είτε κάποιες μικρότερες όπως bug fixes. Σε αυτήν την ενότητα θα δούμε κάποιες από αυτές που σχετίζονται με την εργασία.

Μέχρι τον Νοέμβριο του 2012 ο μόνος μεταγλωττιστής που υπήρχε στο NDK ήταν ο GCC, ώσπου ήρθε η έκδοση r8c που πρόσθεσε τον Clang. Η πρώτη φορά που προστέθηκε υποστήριξη για το OpenMP ήταν με την έκδοση r9b τον Οκτώβριο του επόμενου έτους. Η έκδοση r13b τον Οκτώβριο του 2016 έφερε δυο σημαντικές αλλαγές. Η πρώτη είναι ότι το NDK πια υποστηρίζει και το CMake σαν build system native κώδικα το οποίο όπως αναφέραμε και πριν θα γίνει το προτιμότερο αντί του ndk-build. Η δεύτερη είναι η αφαίρεση της υποστήριξης του GCC με αποτέλεσμα ο Clang να τον αντικαταστεί, κάτι που ισχύει μέχρι και σήμερα. Μάλιστα από τον Σεπτέμβριο του 2018 αφαιρέθηκε πλήρως με την έκδοση r18b. Επιπλέον, από τον Μάρτιο του 2021 (έκδοση r22) ο LLVM LLD και ο LLVM-ar έγιναν οι προεπιλεγμένοι linker και archiver αντίστοιχα, σηματοδοτώντας έτσι την πλήρη αλλαγή στα εργαλεία του LLVM [4] [5].

Στον παρακάτω πίνακα φαίνονται συνοπτικά οι συμβατές εκδόσεις GCC, Clang και OpenMP για κάθε έκδοση του NDK. Κάθε έκδοση του NDK είναι συμβατή με ένα μεγάλο εύρος εκδόσεων Android. Για παράδειγμα η έκδοση r27c είναι συμβατή με τις εκδόσεις Android 5 – Android 15. Γενικά, συνίσταται η χρήση της τελευταίας έκδοσης NDK ανεξάρτητα από την έκδοση Android (εφόσον υποστηρίζεται). Τέλος, αξίζει να σημειωθεί ότι καμία έκδοση του Clang, ως τώρα, δεν υποστηρίζει το offloading του OpenMP σε κάρτες γραφικών συσκευών Android, συνεπώς προκειμένου να εκτελέσουμε κώδικα στην κάρτα γραφικών χρειάζεται συγγραφή κώδικα OpenCL.[6] Περισσότερα παρακάτω στο υποκεφάλαιο 4.2. Στην εργασία αυτή χρησιμοποιήσαμε την έκδοση r27c.

NDK version	GCC version	Clang version	OpenMP version
r9d	4.8	3.4	3.1 (GCC μόνο)
r10e – r13b	4.9	3.6 – 3.8.256229	4.0 (GCC μόνο)
r14b – r17c	4.9	3.8.275480 – 6.0.2	3.1(Clang), 4.0(GCC)
r18b – r21e	–	7.0.2 – 9.0.9	3.1
r22b – r26d	–	11.0.5 – 17.0.2	5.0
r27c	–	18.0.3	5.1

Πίνακας 1 Συμβατές εκδόσεις GCC, Clang και OpenMP για κάθε έκδοση του NDK

Κεφάλαιο 4. Ανάπτυξη εφαρμογών

Στο παρόν κεφάλαιο θα παρουσιάσουμε πως μπορεί κανείς να εκμεταλλευτεί τους πολλαπλούς πυρήνες του επεξεργαστή μιας κινητής συσκευής μέσω του OpenMP αλλά και πως μπορεί να χρησιμοποιήσει την μονάδα γραφικών του (GPU) για την επιτάχυνση υπολογισμών μέσω της OpenCL.

Το NDK είναι υπεύθυνο για την δημιουργία μιας δυναμικής βιβλιοθήκης που περιέχει τις ρουτίνες native κώδικα. Ο προγραμματιστής ορίζει μια Java μέθοδο με το keyword “native” και φορτώνει την δυναμική βιβλιοθήκη με την κλήση `System.loadLibrary("native-lib")`. Έπειτα, μέσω του Java Native Interface (JNI) δίνεται η δυνατότητα κλήσης και εκτέλεσης αυτών των ρουτινών καθώς και η αυτόματη μετατροπή μεταξύ των διαφορετικών τύπων δεδομένων από Java σε C/C++ και το ανάποδο.

Η ανάπτυξη του κώδικα θα γίνει στο Android Studio IDE σε λειτουργικό σύστημα Windows 10/11. Στο πλαίσιο αυτής της εργασίας χρησιμοποιήθηκε η έκδοση 2024.1.2 του Android Studio και οι εφαρμογές δοκιμάστηκαν σε συσκευή που τρέχει Android 14 με ABI το arm64-v8a και Adreno 650 GPU.

4.1 Ανάπτυξη εφαρμογών με OpenMP

Προκειμένου να ενσωματώσουμε το OpenMP στην εφαρμογή μας, τα τρία βασικά βήματα που ακολουθούμε είναι τα ακόλουθα.

- Δημιουργία ενός Native C++ project στο Android Studio.
- Από την στιγμή που εργαζόμαστε σε γλώσσα C θα χρειαστεί η μετονομασία του αρχείου native-lib.cpp σε native-lib.c. Εντός αυτού του αρχείου, τοποθετούμε τον C κώδικα που θέλουμε. Η συνάρτηση που θα κληθεί από την Java είναι:

```
Java_com_example_[project name]_MainActivity_stringFromJNI(JNIEnv*
env, jobject thiz)
```

Η οποία θα έχει μετονομαστεί αυτόματα από το UI του Android Studio σε:

```
MainActivity.stringFromJNI(JNIEnv* env, jobject thiz)
```

Ακόμα, αυτή η συνάρτηση πρέπει να επιστρέφει δεδομένα σε κάποιον τύπο που αναγνωρίζει η Java. Έτσι, στην περίπτωση που θέλουμε να επιστρέψουμε κάποιο Java string τότε κάνουμε την παρακάτω κλήση όπου buffer είναι η μεταβλητή που περιέχει κάποιο C string.

```
return (*env)->NewStringUTF(env, buffer);
```

- Σε αυτό σημείο χρειάζεται να δηλώσουμε την χρήση του OpenMP runtime στον κώδικα μέσω του CMake. Στο αρχείο CMakeLists.txt προσθέτουμε τις ακόλουθες κλήσεις στην περιοχή `add_library` και `target_include_directories` αντίστοιχα.

```
find_package(OpenMP REQUIRED)
και
OpenMP::OpenMP_C
```

Από την στιγμή που αναπτύσσουμε την εφαρμογή μας για μια συσκευή με ABI το arm64-v8a τότε θέτουμε στο αρχείο build.gradle.kts στην περιοχή `defaultconfig` τον εξής περιορισμό.

```
ndk {
    abiFilters += "arm64-v8a"
}
```

Τέλος, αν στην εφαρμογή μας χρειάζονται και άλλα αρχεία τα οποία περιέχουν ρουτίνες σε C που θέλουμε να καλέσουμε τότε αρκεί να τα προσθέσουμε στην περιοχή *`add_library`* στο CMake. Όμοια με πριν, πρέπει να υπάρχουν στον Java κώδικα μέθοδοι με το keyword “native” και στην C θα κληθεί η συνάρτηση που περιέχει το όνομα της Java μεθόδου στο signature του JNI.

4.2 Το πρότυπο της OpenCL

4.2.1 Εισαγωγή στην OpenCL

Η OpenCL (Open Computing Language) αποτελεί ένα ανοιχτό πρότυπο για τον παράλληλο προγραμματισμό ετερογενών υπολογιστικών συστημάτων. Αναπτύχθηκε αρχικά από την Apple και πλέον διαχειρίζεται από τον οργανισμό Khronos Group, στον

οποίο ανήκουν επίσης άλλα δημοφιλή πρότυπα όπως το OpenGL και το Vulkan, τα οποία επίσης υποστηρίζονται στο Android. Στόχος της OpenCL είναι να παρέχει ένα ενιαίο και φορητό προγραμματιστικό μοντέλο που να επιτρέπει την αξιοποίηση διαφορετικών τύπων επεξεργαστών, όπως CPUs, GPUs, επιταχυντές (accelerators), ψηφιακούς επεξεργαστές σήματος (DSPs) και FPGAs, για την επιτάχυνση εφαρμογών με έντονο υπολογιστικό φορτίο.

Η OpenCL είναι μια γλώσσα προγραμματισμού βασισμένη στη C99, γνωστή ως OpenCL C, η οποία επεκτείνεται με ειδικές δομές για παράλληλο υπολογισμό. Η αρχιτεκτονική της OpenCL στηρίζεται σε τέσσερις βασικές έννοιες: την πλατφόρμα (platform), τη συσκευή (device), τον πυρήνα (kernel) και την ουρά εντολών (command queue). Η πλατφόρμα αντιπροσωπεύει το υπολογιστικό περιβάλλον ενός κατασκευαστή, ενώ η συσκευή είναι κάθε υπολογιστική μονάδα, όπως μια CPU ή GPU. Ο πυρήνας είναι η συνάρτηση που εκτελείται παράλληλα σε πολλές μονάδες, και η ουρά εντολών αποτελεί το μηχανισμό με τον οποίο ο host επεξεργαστής (συνήθως η CPU) δρομολογεί εντολές προς τις συσκευές[7].

Ένα από τα σημαντικά πλεονεκτήματα της OpenCL είναι η φορητότητα: ο ίδιος κώδικας μπορεί, με ελάχιστες ή και καθόλου τροποποιήσεις, να εκτελεστεί σε διαφορετικές συσκευές και λειτουργικά συστήματα. Επιπλέον, η OpenCL παρέχει χαμηλού επιπέδου έλεγχο της διαχείρισης μνήμης και των πόρων, επιτρέποντας στον προγραμματιστή να βελτιστοποιήσει την απόδοση της εφαρμογής του. Η κλιμάκωση από μικρά ενσωματωμένα συστήματα έως ισχυρούς υπολογιστικούς κόμβους με πολλές GPUs είναι ένα ακόμα βασικό χαρακτηριστικό του προτύπου.

Ένα τυπικό πρόγραμμα OpenCL περιλαμβάνει τα εξής βήματα: εντοπισμό των διαθέσιμων πλατφορμών και συσκευών, δημιουργία πλαισίου εκτέλεσης (context) και ουρών εντολών, μεταφορά των δεδομένων από το host προς τη συσκευή, μεταγλώττιση και εκτέλεση του πυρήνα, και τέλος επιστροφή των αποτελεσμάτων πίσω στο host. Αν και η διαδικασία είναι πιο περίπλοκη σε σχέση με άλλες υψηλού επιπέδου λύσεις, όπως το offload του OpenMP, προσφέρει αυξημένη ευελιξία και απόδοση.

Στην OpenCL, τα work-items, work-groups και τα compute units είναι βασικές έννοιες που σχετίζονται με την παραλληλία και την εκτέλεση κώδικα στο hardware. Ένα work-item είναι η μικρότερη μονάδα εκτέλεσης και αντιπροσωπεύει μια ανεξάρτητη διεργασία του kernel που εκτελείται σε ένα σημείο των δεδομένων. Πολλά work-items οργανώνονται σε work-groups, τα οποία μπορούν να συνεργάζονται μέσω της κοινής τοπικής μνήμης και συγχρονισμού. Τέλος, τα compute units είναι οι φυσικές μονάδες

επεξεργασίας της συσκευής (όπως οι πυρήνες σε μια GPU ή CPU) και κάθε compute unit μπορεί να εκτελεί ένα ή περισσότερα work-groups παράλληλα.

Η OpenCL βρίσκει εφαρμογή σε ένα ευρύ φάσμα πεδίων, όπως επεξεργασία εικόνας και βίντεο, υπολογιστική βιολογία, μηχανική μάθηση, φυσική προσομοίωση και κρυπτογραφία. Ειδικά στον χώρο των φορητών συσκευών και του Android, η OpenCL μπορεί να χρησιμοποιηθεί για την αξιοποίηση των ενσωματωμένων GPUs για γενικού σκοπού υπολογισμούς. Αυτό δίνει τη δυνατότητα για σημαντική επιτάχυνση εφαρμογών που απαιτούν υψηλή υπολογιστική ισχύ, ξεπερνώντας τους περιορισμούς της CPU.

4.2.2 Ανάπτυξη εφαρμογών με OpenCL

Η συγγραφή κώδικα OpenCL απαιτεί την χρήση ενός ακόμη εργαλείου (Adreno OpenCL SDK Core 2.1) το οποίο παρέχει η Qualcomm μέσα από την ιστοσελίδα της [8]. Όπως και στο προηγούμενο κεφάλαιο χρειάζεται να φτιαχτεί ένα αρχείο C με τον ίδιο τρόπο, το οποίο θα περιέχει τον κώδικα OpenCL. Έπειτα, πρέπει να γίνουν οι εξής παραμετροποιήσεις.

- Όμοια με το OpenMP, χρειάζεται να δηλωθεί η χρήση της OpenCL στο CMake, μέσω του αρχείου CMakeLists.txt. Προσθέτουμε τις ακόλουθες κλήσεις στις περιοχές `add_library` και `target_include_directories` αντίστοιχα.

```
find_package(OpenCL REQUIRED)
και
OpenCL::OpenCL
```

- Το gradle πρέπει να ενημερώσει το CMake ότι για την μεταγλώττιση των αρχείων OpenCL χρειάζεται μια βιβλιοθήκη και κάποια header files που βρίσκονται στο σημείο που εγκαταστάθηκε το SDK που δίνει η Qualcomm. Επίσης, πρέπει να δηλωθεί ότι η εφαρμογή μας κατά την εκτέλεση της θα χρησιμοποιεί την βιβλιοθήκη OpenCL που βρίσκεται προεγκατεστημένη στην κινητή συσκευή για αυτό και δεν συμπεριλαμβάνεται η βιβλιοθήκη του SDK στο APK. Έτσι, εντός του αρχείου build.gradle.kts γίνονται οι παρακάτω προσθήκες. Αρχικά στην περιοχή `defaultConfig` και έπειτα στην περιοχή `android`

```
externalNativeBuild {
            cmake {
                arguments += listOf(
                    "-DOpenCL_LIBRARY=full path to libOpenCL.so",
                    "-DOpenCL_INCLUDE_DIR=full path to CL folder
which contains the necessary header files"A
                    "-DANDROID_ABI=arm64-v8a")
            }
        }
```

```
packaging{
        jniLibs.excludes += "lib/arm64-v8a/libOpenCL.so"
    }
```

- Τέλος, εντός του αρχείου AndroidManifest.xml ορίζεται η εξάρτηση της εφαρμογής από την βιβλιοθήκη της OpenCL.

```
<uses-native-library
            android:name="libOpenCL.so"
            android:required="true" />
```

4.3 Σύνδεση βιβλιοθήκης

Το Android Studio μας δίνει την δυνατότητα να δημιουργήσουμε μια βιβλιοθήκη Android. Μια βιβλιοθήκη Android είναι ένα επαναχρησιμοποιήσιμο αρχείο που περιέχει κοινόχρηστο κώδικα σε Java αλλά και C, πόρους, αρχεία και native βιβλιοθήκες (.so) τα οποία μπορούν να χρησιμοποιηθούν σε οποιαδήποτε εφαρμογή Android θέλουμε. Σε αντίθεση με μια εφαρμογή που μεταγλωττίζεται σε APK, μια βιβλιοθήκη μεταγλωττίζεται σε AAR (Android Archive). Το μόνο που χρειάζεται είναι να δημιουργήσουμε ένα module βιβλιοθήκης μέσα από τις επιλογές του Android Studio και μετά να προσθέσουμε το aar αρχείο σαν εξάρτηση (dependency) στο project που θέλουμε[9].

Κεφάλαιο 5. Ο Μεταφραστής OMPi

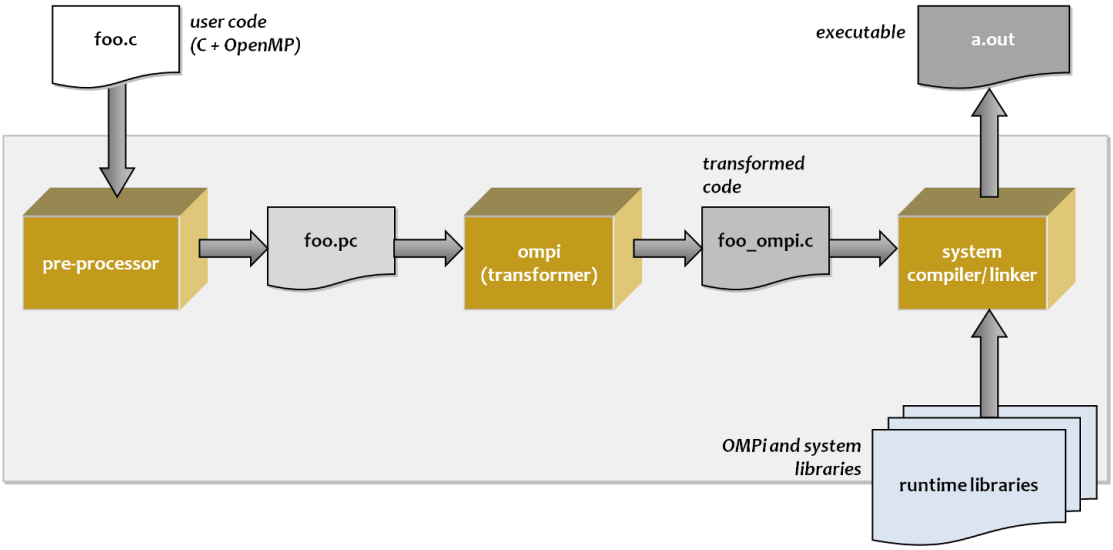
5.1 Η δομή του OMPi

Ο μεταφραστής OMPi είναι ένας παραλληλοποιητικός μεταφραστής γλώσσας C για το πρότυπο OpenMP που αναπτύσσεται από την Ομάδα Παράλληλης Επεξεργασίας του Πανεπιστημίου Ιωαννίνων από το 2001 [10]. Είναι διαχωρισμένος σε δύο βασικά τμήματα, αυτό του μεταγλωττιστή (compiler) και αυτό της υποστήριξης εκτέλεσης (runtime). Το πρώτο, χρησιμοποιείται για την μετατροπή του πηγαίου κώδικα, που περιλαμβάνει οδηγίες OpenMP, σε πολυνηματικό κώδικα C. Το δεύτερο, είναι το σύστημα χρόνου εκτέλεσης το οποίο παρέχει όλες τις απαραίτητες βοηθητικές συναρτήσεις που απαιτούνται σε χρόνο εκτέλεσης. Παράδειγμα τέτοιων συναρτήσεων είναι οι συναρτήσεις που υλοποιούν αμοιβαίο αποκλεισμό και συγχρονισμό όπως κλειδαριές και κλήσεις φραγής, συναρτήσεις δημιουργίας οντοτήτων εκτέλεσης, συναρτήσεις που διαβάζουν τις μεταβλητές περιβάλλοντος και άλλες. Στον παραγόμενο πολυνηματικό κώδικα όλες οι οδηγίες αντικαθιστώνται από κλήσεις σε ρουτίνες του τμήματος υποστήριξης εκτέλεσης και ενδέχεται να χρησιμοποιεί διαφορετικό είδος νημάτων για παράλληλη εκτέλεση. Τέλος, ο OMPi προκειμένου να μεταγλωττιστεί και να εγκατασταθεί σε ένα σύστημα Linux, χρησιμοποιεί το Meson Build System [11].

5.2 Ροή μετάφρασης του OMPi

Αρχικά δίνεται σαν είσοδος στον OMPi κάποιο πρόγραμμα το οποίο περιέχει τον πηγαίο κώδικα σε γλώσσα C που περιλαμβάνει οδηγίες OpenMP. Όπως συμβαίνει με όλα τα προγράμματα C, πρώτα περνάνε από το στάδιο της προεπεξεργασίας. Η έξοδος αυτού του πρώτου σταδίου τροφοδοτείται στο τμήμα του μεταγλωττιστή (compiler) του OMPi, ο οποίος κάνει λεκτική και συντακτική ανάλυση. Από τη συντακτική ανάλυση προκύπτει το αφηρημένο συντακτικό δέντρο (AST- Abstract Syntax Tree) που χρησιμεύει στην αντικατάσταση των οδηγιών OpenMP με κλήσεις συναρτήσεων του συστήματος χρόνου εκτέλεσης, οι οποίες υλοποιούν τις λειτουργίες που καθορίζουν οι αντίστοιχες οδηγίες. Τέλος, ο μετασχηματισμένος κώδικας δίνεται ως είσοδος σε έναν απλό μεταφραστή για γλώσσα C (π.χ. GCC/Clang) και μέσω της σύνδεσης (linking) του με τη βιβλιοθήκη χρόνου

εκτέλεσης του OMPi και τις βιβλιοθήκες του συστήματος, προκύπτει το τελικό εκτελέσιμο αρχείο. Η διαδικασία απεικονίζεται στο σχήμα 5.1.



Σχήμα 5.1 Η διαδικασία της μετάφρασης στον OMPi

5.3 Δημιουργία βιβλιοθήκης με τον OMPi

Ο OMPi από προεπιλογή είναι σχεδιασμένος να μεταφράζει κώδικα που θα χρησιμοποιηθεί σε ηλεκτρονικούς υπολογιστές που τρέχουν κάποια έκδοση του λογισμικού Linux. Εμείς όμως, θέλουμε να τον χρησιμοποιήσουμε σε κινητή συσκευή, διαφορετικής αρχιτεκτονικής και λογισμικού. Για αυτό, χρειάζεται να γίνουν κάποιες τροποποιήσεις πριν καν εγκατασταθεί από το εργαλείο Meson στον υπολογιστή που αναπτύσσεται ο κώδικας (build machine). Το Meson, μέσω των αρχείων .build που βρίσκονται σε όλους τους φακέλους του κώδικα του OMPi, αναλαμβάνει να αναθέσει την μεταγλώττιση στον μεταγλωττιστή του υπολογιστή και έπειτα προχωράει στην εγκατάσταση του OMPi.

- Αρχικά, πρέπει να θέσουμε τις αντίστοιχες μεταβλητές περιβάλλοντος για τον προεπεξεργαστή και τον μετταγλωτιστή που θα περάσει τον κώδικα ο OMPi κατά το τελευταίο στάδιο στην διαδικασία που εξηγήθηκε στο προηγούμενο υποκεφάλαιο. Εντός του NDK βρίσκεται ο Clang για την έκδοση του Android που θέλουμε να τρέχει η εφαρμογή μας και είναι αυτός που θα αναλάβει τις δυο αυτές δουλείες. Οι μεταβλητές που αλλάζουμε είναι οι OMPI_CPP, OMPI_CC στο μονοπάτι που βρίσκεται το εκτελέσιμο του Clang και η μεταβλητή OMPI_CPPFLAGS τίθεται ως "-E".

- Στο κύριο αρχείο meson.build που βρίσκεται στον αρχικό φάκελο του κώδικα του OMPi, αφαιρούμε την υποστήριξη για το thread affinity και concurrency καθώς και την βιβλιοθήκη hwloc, αφού δεν υποστηρίζονται από το Android.

Σε αυτό το σημείο μπορούμε να προχωρήσουμε στην εγκατάσταση του OMPi στο build machine με τις εντολές meson setup και meson install. Παρόλα αυτά, το κομμάτι του OMPi runtime περιέχει μια στατική βιβλιοθήκη, την libort.a, η οποία πρέπει να φορτωθεί μαζί με το APK της εφαρμογής μας στο κινητό. Η βιβλιοθήκη αυτή περιέχει απαραίτητες συναρτήσεις οι οποίες θα κληθούν κατά τον χρόνο εκτέλεσης από τον τροποποιημένο κώδικα που θα παράξει ο OMPi για την εφαρμογής μας. Συνεπώς, θα χρειαστεί να φτιάξουμε 2 scripts, ένα για την μεταγλώττιση και ένα για την εγκατάσταση της βιβλιοθήκης. Στο πρώτο, θα χρειαστεί να μεταγλωττίσουμε όλα τα αρχεία κώδικα που αποτελείται η βιβλιοθήκη σε αρχεία object (.o), χρησιμοποιώντας τον μεταγλωττιστή Clang του NDK. Έπειτα, καλώντας τον archiver του Clang (llvm-ar) σε όλα τα παραπάνω αρχεία object φτιάχνουμε την βιβλιοθήκη libort.a την οποία με το δεύτερο script μπορούμε να την εγκαταστήσουμε στον υπολογιστή μας.

Τώρα, θα χρειαστεί να φτιάξουμε μια δεύτερη βιβλιοθήκη, αυτήν που θα περιέχει τον κώδικα που θα τρέξει η εφαρμογή μας στο κινητό. Ο κώδικας αυτός, κατ' ελάχιστον, πρέπει να αποτελείται από ένα αρχείο header και από ένα αρχείο πηγής (source file) σε γλώσσα C. Εντός του αρχείου header πρέπει να υπάρχει η δήλωση της συνάρτησης `int _ort_init(int *argc, char ***argv, int embedmode, int nmodules, ...)`, ενώ εντός του αρχείου source πρέπει να αρχικοποιηθεί η ίδια ως `_ort_init(0, 0L, 0, 0)`. Στην περίπτωση που ο κώδικας στο αρχείο source χρειάζεται κάποιο αρχείο header όπως το stdio.h ή το string.h, τότε πέρα από την ένταξη αυτών στο αρχείο header, πρέπει να προστεθούν οι παρακάτω οδηγίες οι οποίες απενεργοποιούν ή αντικαθιστούν τους μη υποστηριζόμενους τύπους από τον Clang του NDK.

```
#define __attribute__(x)
#define __signed__ signed
#define _Nullable
#define _Nonnull
#define _Null_unspecified
#define __int128
#define __s128 signed long long
#define __u128 unsigned long long
```

```
#define __asm__(x)
```

Από την στιγμή που έχουμε διαμορφώσει τον OMPi ώστε να δημιουργεί εκτελέσιμα αρχεία που θα τρέχουν σε συσκευές με αρχιτεκτονική Aarch64, τότε μπορούμε να τον χρησιμοποιήσουμε στην μεταγλώττιση των αρχείων source και header που φτιάξαμε νωρίτερα. Όπως νωρίτερα, θα δημιουργήσουμε ένα αρχείο object το οποίο μέσω του llvm-ar θα το κάνουμε στατική βιβλιοθήκη (.a).

Το τελευταίο στάδιο είναι η εισαγωγή των δυο παραπάνω στατικών βιβλιοθηκών και του αρχείου header που φτιάχτηκε νωρίτερα στο CMake στο Android Studio. Έτσι, σε οποιαδήποτε εφαρμογή που περιέχει κώδικα C θα μπορούμε να χρησιμοποιήσουμε τις συναρτήσεις που έχουμε εντός της στατικής βιβλιοθήκης που φτιάξαμε με τον OMPi.

Κεφάλαιο 6. Παραδείγματα Λειτουργίας

Για τον έλεγχο της ορθότητας των τρόπων εργασίας που παρουσιάσαμε στα κεφάλαια 4 και 5, χρειάστηκε να εκτελέσουμε διάφορα προγράμματα σε OpenMP και OpenCL. Όπως αναφέραμε και στο υποκεφάλαιο 3.2 για την μεταγλώττιση των προγραμμάτων χρησιμοποιήσαμε την έκδοση r27c του NDK. Το κινητό στο οποίο εκτελέστηκαν οι εφαρμογές διαθέτει το SoC (System-on-Chip) Snapdragon 870. Ο επεξεργαστής του αποτελείται από οκτώ πυρήνες, έναν πυρήνα με μέγιστη συχνότητα ρολογιού τα 3.2 GHz, τρείς με μέγιστη τα 2.42GHz και άλλους τέσσερις που φτάνουν τα 1.8GHz. Το SoC διαθέτει επίσης την Adreno 650 GPU και στο κινητό υπήρχαν 6GB μνήμης RAM σύνολο.

6.1 Παραδείγματα με OpenMP

Στον παρόν υποκεφάλαιο θα συγκρίνουμε την εκτέλεση προγραμμάτων που περιέχουν οδηγίες OpenMP. Η σύγκριση θα γίνει σε 2 συνθήκες. Η πρώτη αφορά το πλήθος των πυρήνων που θα χρησιμοποιηθούν, ο καθορισμός των οποίων θα γίνει με την φράση `numthreads()`, ενώ η δεύτερη θα συγκρίνει την μεταγλώττιση των προγράμματων αφενός με τον Clang που προσφέρει το NDK και αφετέρου με τον OMPi.

Η πρώτη εφαρμογή που θα δείξουμε αποτελείται από ένα πρόγραμμα που κάνει υπολογισμούς πάνω σε πρώτους αριθμούς. Πιο συγκεκριμένα, μετράει το πλήθος των πρώτων αριθμών και εμφανίζει τον τελευταίο μέχρι ένα άνω όριο (10000000 στην προκειμένη περίπτωση). Η δεύτερη εφαρμογή, εκτελεί κατάλληλες πράξεις ώστε να υπολογίσει το π με μεγάλη ακρίβεια. Οι υπολογισμοί γίνονται με δυο τρόπους και χρονομετρούνται σε δευτερόλεπτα, μια φορά σειριακά και άλλη μια παράλληλα μέσω OpenMP. Η μόνη διαφορά, μεταξύ των ρουτινών που πραγματοποιούν τους υπολογισμούς, είναι ότι στην μία έχει προστεθεί η αντίστοιχη οδηγία OpenMP που παραλληλοποιεί τους βρόγχους for. Τα αποτελέσματα των παρακάτω πινάκων είναι οι μέσοι όροι από 5 εκτελέσεις.

Νήματα	Πρώτοι αριθμοί	Υπολογισμός π
0(σειριακό)	5.45	4.26
2	2.99	1.98
4	1.57	1.05
6	1.34	1.43
8	1.18	1.35

Πίνακας 2 Εκτελέσεις με διαφορετικό πλήθος νημάτων

Βλέποντας τους χρόνους του Πίνακα 2, θα παρατηρήσει κανείς ότι με την αύξηση του πλήθους των νημάτων ο χρόνος εκτέλεσης δεν παρουσιάζει την αναμενόμενη συμπεριφορά. Όπως φαίνεται ενδέχεται να υπάρχει είτε μικρή μείωση, δηλαδή όχι ανάλογη της αύξησης του πλήθους νημάτων, είτε ακόμα και αύξηση του χρόνου εκτέλεσης. Αυτό οφείλεται στην ιδιαίτερη αρχιτεκτονική του επεξεργαστή, ο οποίος διαθέτει πυρήνες ανόμοιους μεταξύ τους. Στις κινητές συσκευές προτεραιότητα δίνεται στην εξοικονόμηση ενέργειας, οπότε οι κατασκευαστές επιλέγουν αυτήν την «περίεργη» διαμόρφωση ώστε να πετύχουν αυτόν τον στόχο. Παρόλα αυτά, το θετικό και κύριο πόρισμα είναι ότι ανεξάρτητα από την επιλογή του προγραμματιστή για τον αριθμό των νημάτων που θα χρησιμοποιήσει, θα δει εμφανής μείωση του χρόνου εκτέλεσης. Έτσι, γίνεται εμφανής πρακτικά η αξία στην χρήση του παράλληλου προγραμματισμού και του OpenMP.

Μεταφραστής	Πρώτοι αριθμοί	Υπολογισμός π
Clang	1.57	1.05
OMPi	1.57	1.34
Σειριακή εκτέλεση	5.45	4.26

Πίνακας 3 Εκτελέσεις με διαφορετικούς μεταφραστές

Για την εξαγωγή των παραπάνω χρόνων χρειάζεται να αναφέρουμε ότι οι οδηγίες του OpenMP στα δυο προγράμματα έκαναν χρήση τεσσάρων νημάτων. Από κει και πέρα, επιβεβαιώνεται ότι η τροποποίηση, η οποία περιγράφεται στο υποκεφάλαιο 5.3, που κάναμε στον OMPi ώστε να μεταφράζει προγράμματα για την αρχιτεκτονική των συσκευών Android, είναι επιτυχής. Ενδέχεται βέβαια να προσθέσει λίγο χρόνο στην εκτέλεση σε σχέση με τον Clang, όπως φαίνεται με τον υπολογισμό του π, αλλά βάση και υπόλοιπων προγραμμάτων που δοκιμάσαμε, αυτό δεν είναι ο κανόνας.

6.2 Εκτέλεση στην GPU

Η εκτέλεση κώδικα μέσω των οδηγιών target του OpenMP δεν ήταν δυνατή. Ούτε υποστηρίζεται από τον Clang όπως είδαμε στον υποκεφάλαιο 3.3, αλλά ούτε, μέσω του OMPi, καταφέραμε να μεταφράζεται η οδηγία αυτή επιτυχώς για εκτέλεση σε συσκευές Android. Συνεπώς, προκειμένου να εκτελεστεί παράλληλος κώδικας στην GPU του κινητού χρησιμοποιήσαμε την OpenCL.

Το πρώτο πρόγραμμα που φτιάξαμε είναι ένα που εκτυπώνει στον χρήστη χαρακτηριστικά της κάρτας γραφικών του κινητού του. Τα χαρακτηριστικά αυτά είναι το όνομα της συσκευής, η έκδοση OpenCL που υποστηρίζει, το μέγιστο πλήθος work-items ανά work-group, το πλήθος των compute units και τα μεγέθη της συνολικής και της τοπικής μνήμης ανά work-group. Αυτές οι πληροφορίες είναι πολύ χρήσιμες στον προγραμματιστή ώστε να μπορέσει να διαμοιράσει τον φόρτο αποδοτικά για την κάρτα γραφικών που διαθέτει, κάτι που κάναμε και εμείς για το επόμενο πρόγραμμα.

Το δεύτερο πρόγραμμα πραγματοποιεί διανυσματική πρόσθεση, προσθέτοντας στοιχείο προς στοιχείο δύο διανύσματα ακέραιων αριθμών (A και B) και αποθηκεύει το αποτέλεσμα σε ένα τρίτο διάνυσμα (C). Ο πυρήνας (kernel) της OpenCL που υλοποιεί την πρόσθεση φορτώνεται και εκτελείται στη GPU. Τα δεδομένα μεταφέρονται από τη κύρια μνήμη του κινητού (CPU) στη μνήμη της κάρτας γραφικών, η πράξη εκτελείται εκεί, και τα αποτελέσματα επιστρέφονται πίσω. Για τον υπολογισμό του αθροίσματος διανυσμάτων που περιέχουν 100 εκατομμύρια στοιχεία, είχαμε σύνολο 100 εκατομμύρια work-items τα οποία διαμοιράστηκαν σε 390625 work-groups των 256 work-items έκαστο. Στον πίνακα 4 φαίνεται η ταχύτητα της OpenCL σε δευτερόλεπτα που χρειάζεται αυτό το πρόγραμμα για να εκτελεστεί σε σχέση με σειριακή και παράλληλη εκτέλεση με OpenMP.

Τρόπος εκτέλεσης	Διανυσματική πρόσθεση
Σειριακά	0.51
OpenMP	0.15
OpenCL	0.03

Πίνακας 4 Χρόνοι με διαφορετικό τρόπο εκτέλεσης

Κεφάλαιο 7. Επίλογος

7.1 Ανακεφαλαίωση

Συνοψίζοντας, στην παρούσα διπλωματική εργασία θέσαμε τρείς στόχους πάνω σε ένα ζήτημα, το οποίο στην πράξη φάνηκε ότι είναι αρκετά εξειδικευμένο. Ο πρώτος ήταν να γίνει μια ιστορική αναδρομή στην υποστήριξη του προτύπου OpenMP σε συστήματα Android όπως έγινε στο υποκεφάλαιο 3.3. Εκεί καταγράψαμε κάποιες από τις αλλαγές που έφερναν οι εκδόσεις του Android NDK όπως την έκδοση του OpenMP, τους υποστηριζόμενους compilers και πολλές άλλες. Ο επόμενος στόχος ήταν να δείξουμε στον αναγνώστη, ενδεχομένως προγραμματιστή, την προετοιμασία που πρέπει να κάνει ώστε να μπορεί να αναπτύξει μια εφαρμογή η οποία τρέχει παράλληλα. Έτσι, στο 4ο κεφάλαιο τον καταφέραμε όχι μόνο με την χρήση του OpenMP, αλλά και της OpenCL χάριν στο εργαλείο της Qualcomm, αξιοποιώντας έτσι την κάρτα γραφικών της κινητής συσκευής για υπολογισμούς γενικού σκοπού (General Purpose GPUs). Ο τελευταίος στόχος ήταν να τροποποιήσουμε τον OMPi ώστε να μπορεί πια να αξιοποιηθεί και σε συσκευές Android, παράγοντας αρχεία συμβατά με την αρχιτεκτονική των συσκευών αυτών. Στο κεφάλαιο 5 αποδείξαμε πώς γίνεται αυτό με τις κατάλληλες παραμετροποιήσεις, συμβάλλοντας έτσι στην διεύρυνση των ικανοτήτων του OMPi ως ένας μεταφραστής πηγαίου σε πηγαίο κώδικα.

7.2 Μελλοντική εργασία

Είναι προφανές ότι, διαβάζοντας αυτήν την εργασία κάποιος προγραμματιστής Android συσκευών, καταλαβαίνει ότι υπάρχουν αμέτρητες εφαρμογές που μπορεί να φτιάξει χρησιμοποιώντας τα εργαλεία που παρουσιάζονται. Παρόλα αυτά, ένα κομμάτι που δεν καταφέραμε να λύσουμε σε αυτή την εργασία έχει να κάνει με την οδηγία target του OpenMP. Η οδηγία αυτή, αναλαμβάνει την εκτέλεση ενός μπλοκ κώδικα σε κάποια άλλη συσκευή η οποία μπορεί να είναι μια κάρτα γραφικών, ένας επιταχυντής κλπ. Στο κομμάτι του προγραμματισμού για γενικού σκοπού υπολογισμούς σε κάρτες γραφικών, η αξία της είναι πολύ σημαντική καθώς η συγγραφή OpenCL χρησιμοποιεί ένα προγραμματιστικό μοντέλου χαμηλού επιπέδου καθιστώντας την αρκετά πολύπλοκη. Ακολουθώντας, την διαδικασία που δείξαμε στο υποκεφάλαιο 4.1 και δοκιμάζοντας κάποια οδηγία που να περιέχει την target μέσα, δεν είδαμε το επιθυμητό αποτέλεσμα

αφού ο Clang του NDK δεν το υποστηρίζει σε συσκευές Android. Μία δεύτερη σκέψη ήταν να δημιουργήσουμε με τον OMPi μια βιβλιοθήκη που θα περιέχει μέσα τον τροποποιημένο κώδικα, δίχως οδηγίες OpenMP, αλλά ούτε αυτή δούλεψε. Είναι πολύ πιθανό ότι με την κατάλληλη τροποποίηση ο OMPi μπορεί να καταφέρει κάτι τέτοιο. Συνεπώς, η επίτευξη αυτής της λειτουργίας θα συμβάλλει ακόμα περισσότερο στην ευελιξία και διευκόλυνση του προγραμματιστή, ενώ παράλληλα θα κάνει τον OMPi ακόμα πιο χρήσιμο.

Βιβλιογραφία

[1] V.V. Dimakopoulos. “Parallel Systems and Programming”, March 2017.

[2] Google, “Platform Architecture”, Android Developers, May 2024. [Online]. Available: https://developer.android.com/guide/platform#linux-kernel

[3] Kitware Inc., CMake Software Build System. [Online]. Available: https://cmake.org/

[4] Google, “NDK Revision History”, Android NDK, July 2024. [Online]. Available: https://developer.android.com/ndk/downloads/revision_history

[5] Google, “Android NDK wiki”, April 2025. [Online]. Available: https://github.com/android/ndk/wiki

[6] The Clang Team, “Clang 18.1.0rc documentation”, OpenMP support, March 2024. [Online]. Available: https://releases.llvm.org/18.1.0/tools/clang/docs/OpenMPSupport.html

[7] The Khronos Group Inc., “The OpenCL™ Specification”, Apr 2025. [Online]. Available. https://registry.khronos.org/OpenCL/specs/3.0-unified/html/OpenCL_API.html#_the_opencl_architecture

[8] Qualcomm Adreno OpenCL SDK. [Online]. Available: https://qpm.qualcomm.com/#/main/tools/details/Adreno_OpenCL_SDK

[9] Google, “Create an Android library”, IDE Guides, May 2025. [Online]. Available: https://developer.android.com/studio/projects/android-library

[10] 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.

[11] The Meson Build System. [Online]. Available: https://mesonbuild.com/index.html