Ατομική Διπλωματική Εργασία

PREPROCESSOR ΚΑΙ BENCHMARKS ΓΙΑ ΠΛΑΤΦΟΡΜΑ TFLUX-SOFT

Μάριος Νικολαΐδης

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Μάιος 2008

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

PREPROCESSOR ΚΑΙ BENCHMARKS ΓΙΑ ΠΛΑΤΦΟΡΜΑ TFLUX-SOFT

Μάριος Νικολαΐδης

Επιβλέπων Καθηγητής
Dr. Pedro Trancoso

Η Ατομική Διπλωματική Εργασία υποβλήθηκε προς μερική εκπλήρωση των απαιτήσεων απόκτησης του πτυχίου Πληροφορικής του Τμήματος Πληροφορικής του Πανεπιστημίου Κύπρου

Μάιος 2008

Ευχαριστίες

Θερμές ευχαριστίες προς τον επιβλέπων καθηγητή Δρ. Pedro Trancoso, ο οποίος, με την υποστήριξή και καθοδήγησή του, συνέβαλε τα μέγιστα στην εκπόνηση αυτής της τόσο ενδιαφέρων διπλωματικής εργασίας. Επίσης, θερμές ευχαριστίες προς τον υποψήφιο διδάκτορα Κυριάκο Σταύρου, καθώς και το συμφοιτητή μου, Δήμο Παύλου, για την άψογη συνεργασία καθ' όλη τη διάρκεια της έρευνας.

Επίσης, για τα ποσοτικά και ποιοτικά αποτελέσματα της ερευνητικής αυτής εργασίας θέλουμε να αναγνωρίσουμε τη σημαντική συνεισφορά του Υποψήφιου Διδάκτορα Κυριάκου Σταύρου.

Περίληψη

Λόγω της συνεχούς αύξησης στην κατανάλωση ενέργειας και στην υπερθέρμανση των επεξεργαστών, η σημερινή τάση στην κατασκευή νέων επεξεργαστών είναι η αύξηση του αριθμού των επεξεργαστών που τοποθετούνται πάνω σε ένα chip. Ένα μοντέλο υπολογισμού που προτάθηκε για εκμετάλλευση αυτής της υπολογιστικής δύναμης είναι το Dataflow, το οποίο εκθέτει το μέγιστο βαθμό παραλληλισμού στον επεξεργαστή, η υλοποίηση του όμως σε επίπεδο εντολής αποδείχτηκε αδύνατη. Το Data-Driven Multithreading (DDM) ξεπερνά τους περιορισμούς αυτούς με την εφαρμογή Dataflow scheduling σε σειρές εντολών, ενώ το scheduling επιτυγχάνεται με τη χρήση του Thread Synchronization Unit (TSU).

Η πλατφόρμα TFlux είναι μια γενική πλατφόρμα για την εκτέλεση προγραμμάτων με τη χρήση του DDM μοντέλου. Είναι ανεξάρτητη από την αρχιτεκτονική του επεξεργαστή και το Λειτουργικό Σύστημα. Η πλατφόρμα TFlux-Soft είναι μια υλοποίηση της πλατφόρμας TFlux, με το TSU υλοποιημένο σε software. Για την άμεση χρησιμοποίησή της όμως ήταν αναγκαίος ένας εύκολος και αυτοματοποιημένος τρόπος προγραμματισμού της πλατφόρμας. Για αυτό το σκοπό επεκτάθηκε ένας Preprocessor ο οποίος υπήρχε ήδη για μια άλλη υλοποίηση της πλατφόρμας TFlux, την TFlux-Hard, ούτως ώστε να υποστηρίζει και παραγωγή κώδικα για την υλοποίηση TFlux-Soft. Έτσι, η μεταφορά και ανάπτυξη εφαρμογών για την πλατφόρμα έγινε πολύ πιο εύκολη για τον προγραμματιστή.

Τέλος, υπήρχε η ανάγκη για αξιολόγηση της πλατφόρμας TFlux-Soft. Έτσι σχηματίστηκε η σουίτα αξιολόγησης της πλατφόρμας, που αποτελείται από έξι διαφορετικά benchmarks, τα οποία παραλληλοποιήθηκαν και μεταφέρθηκαν στην πλατφόρμα TFlux-Soft με τη χρήση του Preprocessor. Ακολούθως, η πειραματική μελέτη έγινε με τη χρήση του *Simics full-system Simulator* σε έναν επεξεργαστή με 28-cores και σε πραγματικό σύστημα με δύο Intel Core2 QuadCore επεξεργαστές. Τα αποτελέσματα δείχνουν ότι επιτυγχάνεται σχεδόν γραμμική επιτάχυνση φτάνοντας το 26.37 speedup με χρήση 27 TFlux Kernels στο *Simics* και 5.88 στο πραγματικό σύστημα. Επομένως, η πλατφόρμα TFlux-Soft μπορεί να χρησιμοποιηθεί άμεσα στους πολυπύρηνους επεξεργαστές της αγοράς.

Περιεχόμενα

Κεφάλαιο 1 Εισαγωγή

1.1 Παράλληλη Επεξεργασία και Πολυεπεξεργαστές

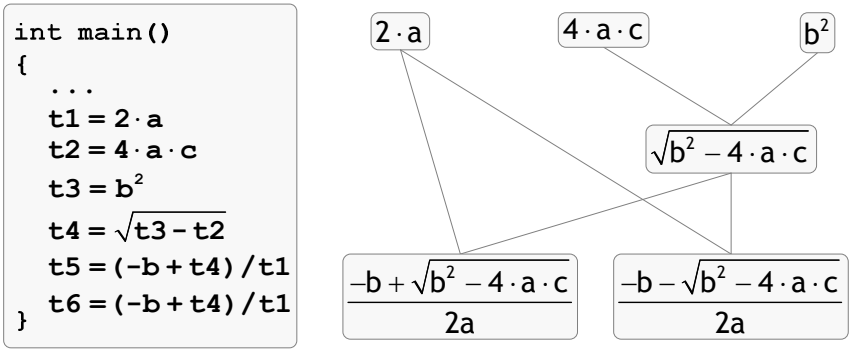
Η σημερινή τάση στην κατασκευή νέων επεξεργαστών είναι η αύξηση του αριθμού των επεξεργαστών που τοποθετούνται πάνω σε ένα chip παρά η αύξηση της συχνότητας και της πολυπλοκότητας ενός επεξεργαστή. Αυτό οφείλεται κυρίως στο πρόβλημα της κατανάλωσης ενέργειας από τον επεξεργαστή, αλλά και στην υπερθέρμανση του επεξεργαστή από την αύξηση στη συχνότητά του. Τεχνικές και εισηγήσεις όπως pipeline [10], instruction level parallelism (ILP) [24], caches [23], prefetching [1 και speculation [13] έδωσαν ότι μπορούσαν να δώσουν οπότε, συνεχίζοντας με τον ίδιο τρόπο, δεν υπάρχει χώρος για άλλη βελτίωση. Η εκμετάλλευση αυτής της υπολογιστικής ισχύος μέσω throughput (δηλαδή η παράλληλη εκτέλεση αριθμού διαφορετικών προγραμμάτων ή processes) είναι εύκολη, αλλά το πιο σημαντικό, και πιο δύσκολο, είναι η εκμετάλλευσή της για την εκτέλεση ενός μόνο προγράμματος, παραλληλοποιώντας το.

1.2 Data-Driven Multithreading

Το Data-Driven Multithreading (DDM) [20] είναι ένα παράλληλο μοντέλο εκτέλεσης το οποίο προέρχεται από το Dataflow [1] μοντέλο υπολογισμού. Η υιοθέτηση των Dataflow τεχνικών έχει το πλεονέκτημα του ότι εκθέτει το μέγιστο δυνατό διαθέσιμο παραλληλισμό, αλλά έχει τα μειονεκτήματα ότι απαιτεί ειδικές γλώσσες προγραμματισμού και αδυνατεί να χειριστεί πολύπλοκες δομές δεδομένων. Το DDM ακολουθεί το παράδειγμα εκτέλεσης του Dataflow, δηλαδή οι εντολές δρομολογούνται για εκτέλεση μόνο όταν τα δεδομένα εισόδου τους έχουν παραχθεί.

Παρόλο που η εφαρμογή dataflow scheduling σε επίπεδο εντολών είναι γνωστό ότι απαιτεί τεράστιους πόρους για το υλικό [5], το DDM εφαρμόζει αυτή την πολιτική στο επίπεδο ακολουθιών από εντολές, οι οποίες ονομάζονται Data-Driven Threads (DDThrs), λύνοντας έτσι το πρόβλημα. Σε επίπεδο εντολών, οι εντολές που ανήκουν σε ένα DDThr εκτελούνται με control-flow τρόπο, εκμεταλλευόμενες όλες τις βελτιστοποιήσεις που προσφέρονται από τον επεξεργαστή στον οποίο εκτελούνται. Στο επίπεδο των Threads, τα DDThrs δρομολογούνται για εκτέλεση με Data-Driven τρόπο.

Πέρα από τη δυνατότητα βελτίωσης της απόδοσης του προγράμματος, το μοντέλο DDM προσφέρει επίσης ένα προγραμματιστικό πλεονέκτημα αφού, για τα περισσότερα προγράμματα, είναι σχετικά εύκολη η αναγνώριση του κώδικα του κάθε DDThr, καθώς επίσης και των εξαρτήσεων δεδομένων μεταξύ τους (Σχήμα 1.1). Ο κώδικας των DDThrs και οι εξαρτήσεις δεδομένων είναι οι μόνες απαιτήσεις για το μοντέλο DDM, το οποίο ακολούθως χειρίζεται τις μεταφορές δεδομένων και επιβάλλει τον απαιτούμενο συγχρονισμό έμμεσα από μόνο του.



Σχήμα 1.1: Απεικόνιση προγράμματος με DDM μοντέλο

Αξίζει επίσης να σημειωθεί ότι, παρόλο ότι υπάρχουν δύο υλοποιήσεις του μοντέλου, το D^2NOW [19] και το TFlux-Hard [16, και οι δύο αυτές υλοποιήσεις αντιμετωπίζουν το πρόβλημα του ότι χρησιμοποιούν επιπλέον ειδικό hardware και έτσι δεν μπορούν να χρησιμοποιηθούν απευθείας.

1.3 Στόχος Ατομικής Διπλωματικής Εργασίας

Στόχος της Ατομικής Διπλωματικής Εργασίας είναι η παροχή υποστήριξης για την ανάπτυξη εφαρμογών για την πλατφόρμα TFlux-Soft, μέσω ενός Preprocessor, η μεταφορά και μελέτη διαφόρων προγραμμάτων-benchmarks στην πλατφόρμα TFlux-Soft με τη βοήθεια του Preprocessor και η αξιολόγηση των πιο πάνω. Το TFlux-Soft είναι μια υλοποίηση σε επίπεδο software της πλατφόρμας παράλληλης επεξεργασίας TFlux για συνηθισμένους πολυεπεξεργαστές [14, οι οποίοι κυκλοφορούν σήμερα ευρέως στην αγορά, η οποία βασίζεται στο Data-Driven Multithreading μοντέλο. Ο Preprocessor είναι μια εφαρμογή η οποία διευκολύνει το γράψιμο και τη μεταφορά προγραμμάτων για εκτέλεση στην πλατφόρμα TFlux. Συγκεκριμένα, λαμβάνει σαν είσοδο ένα πρόγραμμα C με κάποια επιπλέον *pragmas*, τα οποία περιγράφουν τον παραλληλισμό, τα DDThrs και τις εξαρτήσεις μεταξύ τους καθώς και διευκολύνουν κάποιες συνηθισμένες περιπτώσεις πράξεων, και παράγει σαν έξοδο ένα πρόγραμμα C το οποίο μπορεί να εκτελεστεί στην πλατφόρμα TFlux (στην παρούσα του κατάσταση, υποστηρίζει μόνο την υλοποίηση TFlux-Hard της πλατφόρμας).

Πιο συγκεκριμένα, αρχικά θα γίνει μελέτη για την εξεύρεση της εσωτερικής δομής που πρέπει να έχουν τα προγράμματα για εκτέλεση στην πλατφόρμα TFlux-Soft. Έπειτα, θα επεκταθεί ένας ήδη υπάρχων Preprocessor για να υποστηρίζει και την υλοποίηση TFlux-Soft. Τέλος, θα μεταφερθεί ένας αριθμός benchmarks στην πλατφόρμα TFlux με τη χρήση του Preprocessor στο TFlux-Soft για την αξιολόγηση της συγκεκριμένης υλοποίησης της πλατφόρμας.

Η εργασία αυτή είναι οργανωμένη ως ακολούθως. Στο Κεφάλαιο 2, συζητούμε για σχετική δουλειά που έγινε στον ίδιο τομέα. Στο Κεφάλαιο 3, παρουσιάζουμε μία επισκόπηση του Data-Driven Multithreading μοντέλου εκτέλεσης ενώ στο Κεφάλαιο 4 παρουσιάζουμε την πλατφόρμα TFlux και στο Κεφάλαιο 5 την υλοποίησή της σε επίπεδο λογισμικού μόνο, το TFlux-Soft. Στο Κεφάλαιο 6 παραθέτουμε τη μελέτη που έγινε για την κατάλληλη μορφή των προγραμμάτων για TFlux-Soft, ενώ στο Κεφάλαιο 7 παρουσιάζουμε τη διαδικασία ανάπτυξης, τα κύρια μέρη και τη χρήση του Preprocessor για TFlux-Soft. Ακολούθως, στο Κεφάλαιο 8 παρουσιάζουμε τα

benchmarks που μεταφέρθηκαν στην πλατφόρμα TFlux-Soft, καθώς και τη διαδικασία μεταφοράς τους και στο Κεφάλαιο 9 αναλύουμε τα αποτελέσματα από την αξιολόγηση της πλατφόρμας. Τέλος, στο Κεφάλαιο 10, αναλύουμε και παραθέτουμε τα συμπεράσματά μας από αυτή την εργασία και συζητούμε για τη μελλοντική δουλειά που μπορεί να γίνει.

Κεφάλαιο 2 Σχετική Δουλειά

Το γεγονός της ανάπτυξης των πολυπύρηνων επεξεργαστών οδήγησε στην εκτεταμένη μελέτη τρόπων εκμετάλλευσης του παραλληλισμού των προγραμμάτων. Αρκετές από αυτές τις μελέτες επηρεάστηκαν από το μοντέλο dataflow. Πιο κάτω αναφέρονται μερικές από αυτές.

Το EARTH (Efficient Architecture for Running Threads) [11] είναι ένα μοντέλο αρχιτεκτονικής το οποίο είναι αρκετά επηρεασμένο από το dataflow μοντέλο. Στόχος του είναι να εκμεταλλευτεί το fine-grained multithreading στους πολύ-πυρήνες της αγοράς. Χρησιμοποιεί δύο τύπους fine-grained threads, την παράλληλη συνάρτηση και τα fibers, που είναι μικρά κομμάτια κώδικα μέσα στο σώμα της συνάρτησης. Κάθε fiber είναι atomic και μεταξύ τους υπάρχουν σχέσεις τύπου producer/consumer. Ένα fiber ενεργοποιείται όταν λάβει όλα τα input signals και εκτελείται μόλις υπάρχει διαθέσιμη μονάδα επεξεργασίας. Μόλις ολοκληρωθεί η εκτέλεσή του στέλνονται σήματα σε όλα τα consumer fibers του για να ανανεώσουν το sync slot τους. Για το συγχρονισμό χρειάζεται ένα Synchronization Unit που είναι υπεύθυνο για τη διατήρηση των πληροφοριών για τα fibers.

Η EDGE [4] είναι μια Αρχιτεκτονική Συνόλου Εντολών (Instruction Set Architecture – ISA) βασισμένη επίσης στο dataflow μοντέλο. Οι εντολές δεν κωδικοποιούν τους source operands όπως στις αρχιτεκτονικές RISC και CISC, αλλά καθορίζουν το που πρέπει να δρομολογηθούν τα αποτελέσματα. Η γενική ιδέα είναι ότι το dataflow graph, που καθορίζεται από τον μεταγλωττιστή, φορτώνεται στον επεξεργαστή ώστε να μπορεί να αποφύγει την ανάλυση για τις εξαρτήσεις μεταξύ των εντολών και τη μετονομασία των registers, διαδικασίες που καταναλώνουν αρκετή ενέργεια και χρόνο εκτέλεσης. Η αρχιτεκτονική TRIPS [3] είναι η πρώτη υλοποίηση της EDGE ISA. Το πρόγραμμα χωρίζεται σε hyper-blocks, οι εντολές των οποίων εκτελούνται με βάση το dataflow μοντέλο. Η εκτέλεση των hyper-blocks γίνεται σειριακά. Αυτή η προσέγγιση απαιτεί αλλαγές στον επεξεργαστή για την υλοποίηση της EDGE, σε

αντίθεση με την πλατφόρμα TFlux-Soft η οποία δεν απαιτεί καμιά αλλαγή στο υλικό.

Το Thread Level Speculation [15], το οποίο είναι υλοποιημένο από το Stanford's Hydra CMP [8], επιτρέπει στον προγραμματιστή να χωρίσει ένα σειριακό πρόγραμμα σε ανεξάρτητα κομμάτια κώδικα (Threads) χωρίς να χρειάζεται στατικά να οριστούν οι εξαρτήσεις δεδομένων μεταξύ τους. Το hardware επιχειρεί να εκτελέσει τα Threads παράλληλα χρησιμοποιώντας τις προσβάσεις στη μνήμη για να εντοπίσει εξαρτήσεις δεδομένων. Σε περίπτωση λάθους υπόθεσης, ενεργοποιούνται διαδικασίες για να το διορθώσουν. Το Hydra CMP χρειάζεται επιπλέον hardware το οποίο υπολογίζεται στο μέγεθος δύο L1 caches, καθιστώντας έτσι αδύνατη την άμεση υιοθέτησή του για χρήση στους πολύ-πύρηνους επεξεργαστές της αγοράς.

Το Data-Driven Network of Workstations (D^2NOW) [19] παρουσιάστηκε σαν απόδειξη ότι το Data-Driven Multithreading (DDM) μοντέλο μπορεί να υλοποιηθεί χρησιμοποιώντας ήδη υπάρχοντα προϊόντα. Η κύρια συνέπεια του πιο πάνω είναι ότι το D^2NOW μπορεί να επωφεληθεί από την, ανά πάσα στιγμή, πιο σύγχρονη τεχνολογία επεξεργαστών. Στο D^2NOW, το scheduling των Threads γίνεται με τη βοήθεια ενός μικρού κομματιού hardware.

Το TFlux-hard [16 είναι μια υλοποίηση του Data-Driven Multithreading για τους πολύ-πυρήνες της αγοράς. Για τη λειτουργία του χρειάζεται μια μονάδα που είναι υπεύθυνη για το συγχρονισμό των Data-Driven Threads, το Thread Synchronization Unit (TSU) [18]. Στην περίπτωση του TFlux-hard, το TSU είναι υλοποιημένο σε hardware και συνδέεται στον επεξεργαστή ως ένα memory-mapped device. Ο επεξεργαστής ελέγχει το TSU στέλνοντας σήματα μέσω ενός δικτύου. Κατά την εκτέλεση ενός DDM προγράμματος, εγγράφεται στο TSU ο γράφος συγχρονισμού του προγράμματος, δηλαδή οι producer/consumer σχέσεις μεταξύ των Threads. Το TSU είναι υπεύθυνο για τη διαχείριση αυτών των πληροφοριών και την ενημέρωση του εκάστοτε επεξεργαστή για το πιο Thread μπορεί να εκτελεστεί.

Αν και όλες οι πιο πάνω δουλειές πρότειναν και υλοποίησαν μοντέλα και πλατφόρμες παράλληλης επεξεργασίας προγραμμάτων, οι πλείστες από αυτές

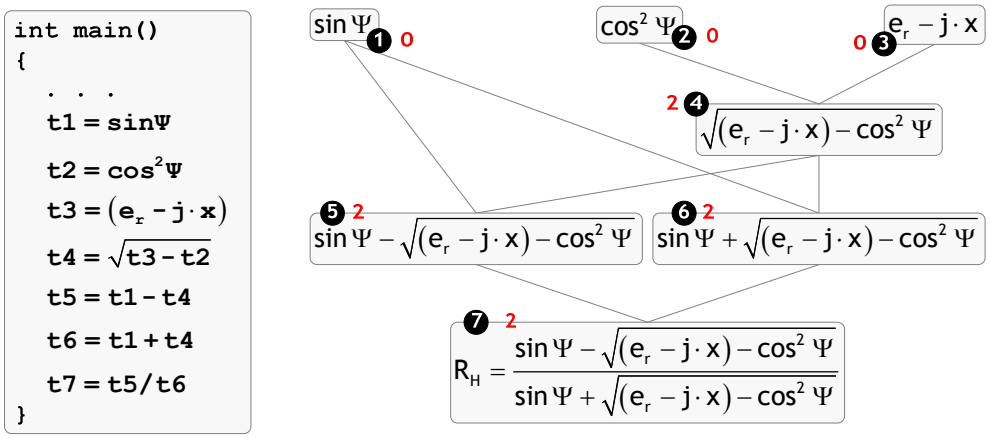
απαιτούν τη χρήση εξειδικευμένου επιπλέον hardware για να μπορέσουν να λειτουργήσουν σε κανονικούς υφιστάμενους επεξεργαστές ή ακόμα και επανασχεδιασμό της αρχιτεκτονικής ενός επεξεργαστή. Σε αντίθεση με αυτά, η πλατφόρμα TFlux-Soft, λόγω της υλοποίησης του ***TSU*** σε επίπεδο λογισμικού, προσφέρει όλα τα πλεονεκτήματα της υλοποίησης *TFlux* όσον αφορά το προγραμματιστικό μοντέλο, αφαιρετικότητα και επιδόσεις, χωρίς όμως την ανάγκη επιπλέον hardware για τη χρήση της. Αυτό επιτρέπει την άμεση χρησιμοποίησή της σε συνηθισμένους πολύ-επεξεργαστές που κυκλοφορούν στην αγορά σήμερα.

Κεφάλαιο 3 Data-Driven Multithreading

3.1 Περιγραφή μοντέλου

Το Data-Driven Multithreading είναι ένα μοντέλο εκτέλεσης το οποίο επιτυγχάνει αποδοτική παράλληλη εκτέλεση μέσω dataflow scheduling.

Τα προγράμματα DDM αποτελούνται από μη-αλληλεπικαλυπτόμενα κομμάτια κώδικα τα οποία ονομάζονται Data-Driven Threads (DDThrs) και μπορούν να έχουν οποιοδήποτε μέγεθος, όπως βλέπουμε στο σχήμα 3.1-(α). Μεταξύ των DDThrs υπάρχουν σχέσεις producer/consumer. Οι εξαρτήσεις μεταξύ των DDThrs σε ένα πρόγραμμα DDM εκφράζονται από το Synchronization Graph, οι κόμβοι του οποίου αντιστοιχούν στα DDThrs του προγράμματος, ενώ οι ακμές στις εξαρτήσεις δεδομένων μεταξύ τους. Για παράδειγμα, το Synchronization Graph του προγράμματος του σχήματος 3.1-(α), φαίνεται στο σχήμα 3.1-(β). Εδώ παρατηρούμε ότι το DDThr 4 είναι consumer των DDThrs 2 και 3 και producer για τα DDThrs 5 και 6.



Σχήμα 3.1: Παράδειγμα εκτέλεσης προγράμματος με DDM

Το scheduling ενός DDThr για εκτέλεση γίνεται δυναμικά με data-driven τρόπο, δηλαδή μόνο όταν όλοι οι producers του έχουν ολοκληρώσει την εκτέλεσή τους. Οι εντολές μέσα σε ένα DDThr εκτελούνται από τον επεξεργαστή με control-flow τρόπο και οποιεσδήποτε βελτιστοποιήσεις, είτε από τον επεξεργαστή δυναμικά κατά την εκτέλεση είτε στατικά από το μεταγλωττιστή, μπορούν να χρησιμοποιηθούν.

Το scheduling των Data-Driven Threads επιτυγχάνεται με τη βοήθεια μίας ειδικευμένης μονάδας, του Thread Synchronization Unit (TSU) [18]. Πριν από την εκτέλεση οποιουδήποτε DDThr, ο γράφος συγχρονισμού (Synchronization Graph) του προγράμματος φορτώνεται στο TSU, όπως φαίνεται από το σχήμα 3.1-(γ). Συγκεκριμένα, για κάθε DDThr, το TSU διατηρεί μια λίστα με τους consumers του και το Ready Count του, μια τιμή που δείχνει τον αριθμό των παραγωγών του DDThrs. Όταν ένα DDThr ολοκληρώσει την εκτέλεσή του ειδοποιεί το TSU, το οποίο με τη σειρά του μειώνει τις τιμές των Ready Count όλων των consumers του. Όταν η τιμή του Ready Count ενός DDThr φτάσει το μηδέν, το DDThr είναι έτοιμο για εκτέλεση. Μόλις ένας επεξεργαστής γίνει διαθέσιμος, η εκτέλεση αυτού του DDThr θα ξεκινήσει. Για να βρει ένας επεξεργαστής το επόμενο DDThr προς εκτέλεση, "ρωτά" το TSU, το οποίο "απαντά" με το προσδιοριστικό ενός από τα έτοιμα για εκτέλεση DDThrs. Στην περίπτωση που δεν υπάρχει κανένα DDThr έτοιμο για εκτέλεση, το TSU θα αναγκάσει τον επεξεργαστή να περιμένει.

Για να γίνει δυνατή η εκτέλεση προγραμμάτων με αυθαίρετα μεγάλους γράφους συγχρονισμού, χωρίς να απαιτείται και εξίσου μεγάλο TSU, τα DDM προγράμματα μπορούν να χωριστούν σε DDM Blocks. Κάθε DDM Block περιέχει ένα υποσύνολο των DDThrs του αρχικού προγράμματος και το μέγιστο μέγεθός του καθορίζεται από το μέγεθος του TSU. Πέρα από τα DDThrs του προγράμματος, κάθε DDM Block περιέχει ακόμα 2 DDThrs, το Inlet και Outlet DDThr. Ο σκοπός του πρώτου είναι να εγγράψει στο TSU όλα τα δεδομένα των DDThrs που ανήκουν στο αντίστοιχο DDM Block, ενώ ο σκοπός του δεύτερου είναι να αποδεσμεύσει τους δεσμευμένους πόρους. Όταν όλα τα DDThrs ενός DDM Block ολοκληρώσουν την εκτέλεσή τους, το Inlet DDThr του επόμενο block φορτώνεται στο TSU, ούτως ώστε να μπορεί να συνεχίσει η εκτέλεση σε εκείνο το Block.

3.2 Hardware Υλοποιήσεις

Η πρώτη υλοποίηση του μοντέλου DDM, το D^2NOW [19], χρησιμοποιήθηκε για να αποδείξει την ικανότητα του μοντέλου να επιτύχει ψηλή απόδοση και επεκτασιμότητα (scalability) χωρίς να απαιτούνται οποιεσδήποτε τροποποιήσεις στο CPU. Το D^2NOW χρησιμοποιούσε dedicated μηχανές συνδεδεμένες σαν ένα Δίκτυο από Σταθμούς Εργασίας (Network of Workstations). Για να μπορούν να υποστηρίξουν το μοντέλο DDM, σε αυτές οι μηχανές προστέθηκε μία επιπλέον μονάδα υλικού, το TSU. Οι μηχανές αυτές εκτελούσαν το πρόγραμμα σε αφιερωμένη κατάσταση, δηλαδή χωρίς υποστήριξη για πολυπρογραμματισμό, αφού δεν έτρεχαν κάποιο Λειτουργικό Σύστημα. Επιπρόσθετα, στο D^2NOW, ο προγραμματισμός γινόταν σε χαμηλό επίπεδο καθώς στον κώδικα έπρεπε να προστεθούν συγκεκριμένες εντολές assembly. Παρόλο που το D^2NOW δεν είχε κάποιο περιορισμό ως προς την εκτέλεσή του σε διαφορετική αρχιτεκτονική, το σύστημα υλοποιήθηκε για μηχανές x86.

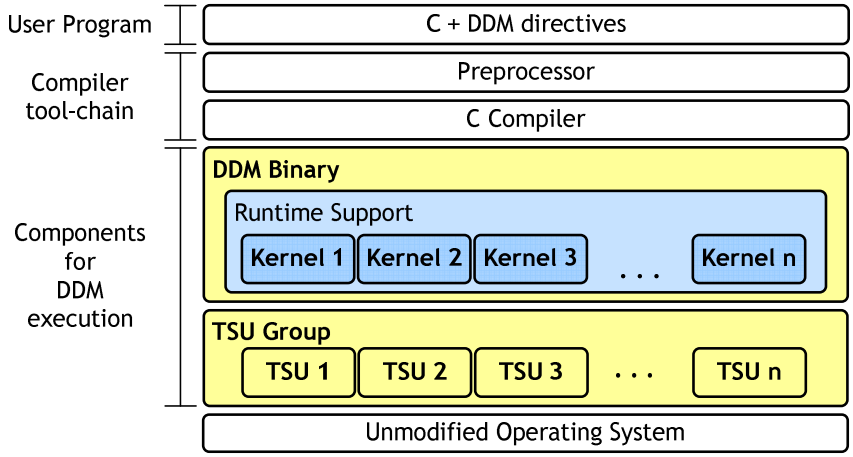
Η επόμενη υλοποίηση του DDM και εξέλιξη του D^2NOW ήταν το TFlux-Hard. Αυτή η υλοποίηση έγινε στο επίπεδο του χρήστη και έτσι επέτρεπε στην εφαρμογή να τρέχει πάνω από ένα οποιοδήποτε Λειτουργικό Σύστημα. Αυτό επέτρεπε στο σύστημα να εκτελεί τόσο DDM εφαρμογές όσο και συνηθισμένες εφαρμογές. Επιπλέον, η αναγνώριση του κώδικα των DDThrs και των εξαρτήσεων μεταξύ τους γίνεται με την προσθήκη ειδικά σχεδιασμένων directives στον κώδικα ψηλού επιπέδου (C). Με τη χρήση του Preprocessor [22], αυτά τα directives μεταφράζονται σε κατάλληλες κλήσεις συναρτήσεων, επιτρέποντας έτσι την εύκολη μεταφορά εφαρμογών. Τέλος, αφού όλη η δουλειά του Preprocessor γίνεται σε ψηλό επίπεδο, χρησιμοποιούνται συνηθισμένοι μεταγλωττιστές για τη δημιουργία του εκτελέσιμου κώδικα και έτσι είναι δυνατή η παραγωγή εκτελέσιμων για οποιαδήποτε αρχιτεκτονική. Παρόλο της καλής προοπτικής του TFlux-Hard για ψηλή απόδοση, επεκτασιμότητα και εύκολο προγραμματισμό, η ανάγκη χρησιμοποίησης ενός ειδικού κομματιού hardware εξακολουθεί να είναι δεσμευτική ως προς τις δυνατότητες του συστήματος και δεν μπορεί να χρησιμοποιηθεί σε υπάρχοντα συστήματα.

Κεφάλαιο 4 Φορητή πλατφόρμα TFlux

Το TFlux [16 δεν αναφέρεται σε κάποια υλοποίηση σε συγκεκριμένη μηχανή αλλά σε μια συλλογή από αφηρημένες οντότητες, οι οποίες επιτρέπουν την εκτέλεση προγραμμάτων χρησιμοποιώντας το μοντέλο DDM πάνω σε διάφορα υπολογιστικά συστήματα πολυεπεξεργασίας, ανεξάρτητα από τις λεπτομέρειες του υλικού ή τον τύπο του λειτουργικού συστήματος. Δηλαδή το TFlux παρέχει το επίπεδο virtualization για την εκτέλεση των TFlux προγραμμάτων κάτω από διαφορετικά συστήματα πολυεπεξεργασίας κοινής μνήμης.

Το σχήμα 4.1 παρουσιάζει το στρωματοποιημένο σχεδιασμό του συστήματος TFlux. Συγκεκριμένα, το πιο ψηλό επίπεδο, το οποίο χρησιμοποιούν οι προγραμματιστές για την ανάπτυξη προγραμμάτων, αφαιρεί όλες τις λεπτομέρειες τις μηχανής που βρίσκεται από κάτω, τόσο σε επίπεδο υλικού όσο και σε λογισμικό. Τα TFlux προγράμματα αναπτύσσονται χρησιμοποιώντας ANSI-C μαζί με κάποια DDM directives [22]. Τα DDM directives χρησιμοποιούνται για να εκφράσουν τον κώδικα των DDThrs, καθώς και τις μεταξύ τους εξαρτήσεις. Στη συνέχεια το πρόγραμμα περνά από τη σειρά εργαλείων μεταγλώττισης του TFlux και έτσι δημιουργείται το εκτελέσιμο αρχείο. Το πρώτο στάδιο αυτής της σειράς είναι ο TFlux Preprocessor, ο οποίος μεταφράζει το C πρόγραμμα με directives σε ένα αντίστοιχο παράλληλο C πρόγραμμα. Το δεύτερο στάδιο είναι η μεταγλώττιση του παραγόμενου

παράλληλου C προγράμματος με ένα συνηθισμένο μεταγλωττιστή (gcc) και η παραγωγή ενός παράλληλου εκτελέσιμου αρχείου. Αυτό το εκτελέσιμο καλεί τις λειτουργίες του “TFlux Runtime Support”, το οποίο επιτρέπει την εκτέλεση με βάση το DDM. Το Runtime Support τρέχει πάνω από ένα Λειτουργικό Σύστημα τύπου Unix, το οποίο δε χρειάζεται να τροποποιηθεί κατά οποιοδήποτε τρόπο, και κρύβει όλες τις λεπτομέρειες σχετικά με το DDM, όπως τη συγκεκριμένη υλοποίηση του TSU. Μια από τις πρωταρχικές ευθύνες του Runtime Support είναι η δυναμική εγγραφή στα TSUs των μεταδεδομένων (meta-data) των DDThrs και η κλήση όλων των TSU λειτουργιών που είναι απαραίτητες για την εκτέλεση.



Σχήμα 4.1: Ο στρωματοποιημένος σχεδιασμός του συστήματος TFlux

Στα υποκεφάλαια που ακολουθούν θα περιγραφούν τα κύρια συστατικά και χαρακτηριστικά της πλατφόρμας TFlux.

4.1 Runtime Support

Το virtualization που προσφέρει το TFlux οφείλεται κυρίως στο Runtime Support του. Το Runtime Support τρέχει πάνω από ένα Λειτουργικό Σύστημα, το οποίο δεν έχει τροποποιηθεί κατά οποιοδήποτε τρόπο, και επιτρέπει την εκτέλεση τόσο TFlux όσο και συμβατικών προγραμμάτων.

Για να είναι εφικτή η χρήση ενός συνηθισμένου Λειτουργικού Συστήματος για DDM εκτέλεση, το runtime support πρέπει να πληρεί δύο σημαντικές προϋποθέσεις. Πρώτα, όταν μια εφαρμογή εκτελείται παράλληλα είτε σε shared-memory είτε σε

distributed-memory πολυεπεξεργαστή, το runtime support πρέπει να παρέχει έναν τρόπο για τα διάφορα DDThrs της TFlux εφαρμογής να έχουν πρόσβαση στις κοινές μεταβλητές που αφορούν τις εξαρτήσεις μεταξύ τους. Δεύτερο, το runtime support πρέπει να παρέχει έναν αποδοτικό τρόπο επικοινωνίας μεταξύ της εφαρμογής και του TSU.

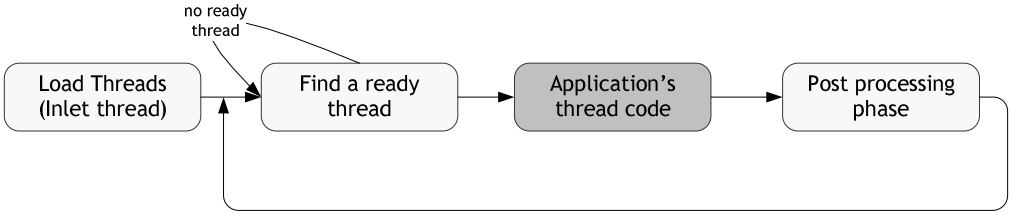
Για να γίνουν τα πιο πάνω δυνατά, σχεδιάσαμε μία απλή οντότητα στο επίπεδο του χρήστη, ο TFlux Kernel [16, το οποίο περιγράφεται στο επόμενο μέρος. Σημαντικό επίσης είναι το γεγονός ότι το runtime support για μια TFlux εφαρμογή ενσωματώνεται στην εφαρμογή κατά τη μεταγλώττιση. Συνεπώς, κάθε TFlux εκτελέσιμο περιέχει όλα όσα χρειάζεται για την εκτέλεσή του και δεν έχει ανάγκη για κανένα επιπλέον λογισμικό.

4.1.1 Kernel

Το runtime support ξεκινά την εκτέλεσή του με τη δημιουργία ***n*** kernels, όπου ***n*** είναι ο μέγιστος αριθμός DDThrs που μπορούν να εκτελεστούν παράλληλα στη μηχανή.

Το σχήμα 4.2 παρουσιάζει τη λειτουργία του kernel. Η πρώτη του ενέργεια είναι να μεταφέρει την εκτέλεση στη διεύθυνση της πρώτης εντολής του Inlet DDThr του πρώτου Block. Στο τέλος αυτού του Inlet DDThr, όπως και οποιουδήποτε άλλου DDThr, ο έλεγχος μεταφέρεται και πάλι στον κώδικα του kernel και πιο συγκεκριμένα στο βρόγχο επανάληψης FindReadyThread. Σε αυτό το σημείο ο kernel ζητά από το TSU το επόμενο DDThr το οποίο είναι έτοιμο για εκτέλεση. Όταν τελικά η κλήση στο TSU επιστρέψει τη διεύθυνση του επόμενου DDThr, η εκτέλεση μεταφέρεται εκεί. Όταν ένα DDThr ξεκινήσει την εκτέλεσή του, δε διακόπτεται από το kernel και, όταν ολοκληρωθεί, τότε ειδοποιεί το TSU και μεταφέρει την εκτέλεση πίσω στο kernel. Τότε το TSU θα εκτελέσει την Post-Processing φάση για το συγκεκριμένο DDThr, κατά την οποία οι τιμές των Ready Count των consumers του θα μειωθούν. Το επόμενο βήμα για τον kernel είναι να μεταφέρει την εκτέλεση στο επόμενο έτοιμο DDThr. Τέλος, όταν όλοι οι kernels ολοκληρώσουν την εκτέλεση

όλων των DDThrs που τους ανατέθηκαν, τελειώνει και η εκτέλεση του DDM προγράμματος.



Σχήμα 4.2: Εκτέλεση ενός Kernel με σκιασμένο τον κώδικα της εφαρμογής

Η μετάβαση από τον κώδικα του kernel στον κώδικα της εφαρμογής, και αντίστροφα, γίνεται με ελάχιστο κόστος και είναι εντελώς διαφανής (transparent) προς το Λειτουργικό Σύστημα. Αυτό επιτυγχάνεται με το να έχουμε τον κώδικα του kernel και των κώδικα των DDThrs της εφαρμογής στην ίδια συνάρτηση του κώδικα του προγράμματος. Αυτές οι μεταβάσεις μπορεί να είναι συχνές, αναλόγως και της εφαρμογής, και έτσι η όσο το δυνατό πιο αποδοτική λειτουργία τους συνεισφέρει σημαντικά στην ελαχιστοποίηση του overhead κατά την εκτέλεση.

4.2 Thread Synchronization Unit

Το Thread Synchronization Unit (TSU) είναι η μονάδα που είναι υπεύθυνη για τη διατήρηση των πληροφοριών που παρέχονται από το Γράφο Συγχρονισμού του TFlux προγράμματος που εκτελείται, καθώς και τη δυναμική ενημέρωσή τους, μέσω διαφόρων συναρτήσεων.

Σε κάθε TFlux kernel αντιστοιχεί το δικό του προσωπικό TSU, στο οποίο εγγράφονται όλες οι πληροφορίες για τα DDThrs τα οποία θα εκτελέσει ο συγκεκριμένος TFlux kernel. Επιπλέον, είναι απαραίτητη η ύπαρξη κάποιων δομών κοινών για όλα τα TSUs ούτως ώστε να είναι δυνατή η επικοινωνία μεταξύ τους, κυρίως για σκοπούς ενημέρωσης των consumer DDThrs μετά την ολοκλήρωση της εκτέλεσης ενός από τα producer DDThrs τους.

Στην προηγούμενη υλοποίηση του DDM, το D^2NOW, κάθε επεξεργαστής έπρεπε να έχει το δικό του προσωπικό TSU, καθώς οι μονάδες εκτέλεσής του ήταν

ανεξάρτητες μηχανές. Στο TFlux, τα TSUs είναι ομαδοποιημένα σε μία μονάδα που ονομάζεται TSU Group. Οι πόροι του TSU Group είναι χωρισμένοι σε δύο κατηγορίες: αυτοί που εξυπηρετούν τον επεξεργαστή στον οποίο αντιστοιχεί ένα TSU και αυτοί που είναι κοινοί για όλους τους επεξεργαστές. Το TSU Group έχει δύο σημαντικά πλεονεκτήματα σε σύγκριση με τη λύση των ξεχωριστών TSUs.

Το πρώτο αφορά την επικοινωνία μεταξύ των TSUs, η οποία είναι συχνή κατά την εκτέλεση ενός DDM προγράμματος. Στο TSU Group, αυτή η επικοινωνία γίνεται εσωτερικά χωρίς την ανάμιξη καμιάς άλλης μονάδας.

Το δεύτερο πλεονέκτημα είναι το ότι γίνεται δυνατή η προσφορά της λειτουργικότητας του TSU στο επίπεδο του λογισμικού χρησιμοποιώντας μόνο μία υπολογιστική οντότητα η οποία μιμείται τις λειτουργίες. Ως συνέπεια, ένας πολυεπεξεργαστής μπορεί να προσφέρει τη λειτουργικότητα του TSU σε επίπεδο λογισμικού (software) αφιερώνοντας μόνο έναν επεξεργαστή.

4.3 Σειρά εργαλείων μεταγλώττισης

Για την ανάπτυξη ενός προγράμματος για DDM ο προγραμματιστής χρησιμοποιεί ANSI-C μαζί με *DDM pragma directives* [22], με τα οποία ορίζει τον κώδικα των DDThrs και τις εξαρτήσεις μεταξύ τους. Ο Data-Driven Multithreading C Preprocessor (DDMCPP) [22] παίρνει σαν είσοδο ένα πρόγραμμα με C κώδικα και τα καθορισμένα *DDM pragma directives* και παράγει σαν έξοδο ένα πρόγραμμα C το οποίο περιέχει όλο τον κώδικα του runtime support, καθώς και τις απαραίτητες TFlux κλήσεις στις συναρτήσεις αλληλεπίδρασης με το TSU προκειμένου να μπορεί το πρόγραμμα να εκτελεστεί πάνω στην αντίστοιχη υλοποίηση της αρχιτεκτονικής TFlux. Αυτό το εργαλείο είναι νοητά χωρισμένο σε δύο ενότητες, την front-end και την back-end.

Το DDMCPP front-end είναι ένας parser ο οποίος είναι ανεξάρτητος της υλοποίησης του DDM και ο σκοπός του είναι να διαβάσει τα *pragma directive*s και μετά να περάσει τις πληροφορίες στο back-end για να παραχθεί ο κώδικας για τη συγκεκριμένη αρχιτεκτονική. Το back-end είναι χτισμένο σαν οι πράξεις μιας

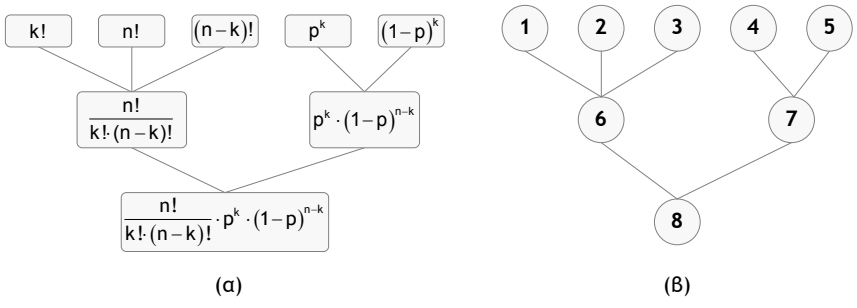
γραμματικής για όλα τα DDM directives και εξαρτάται από τη συγκεκριμένη υλοποίηση του DDM. Ο σκοπός του είναι η δημιουργία του κώδικα που είναι απαραίτητος για το TFlux runtime support όπως τον κώδικα των kernels και τις λειτουργίες φόρτωσης δεδομένων στο TSU, μεταξύ άλλων.

4.4 Βασικά συστατικά εκτέλεσης

4.4.1 Threads

Το βασικό συστατικό ενός προγράμματος TFlux είναι το DDThr. Κάθε DDThr αντιστοιχεί σε ένα μέρος του στατικού κώδικα και μπορεί να έχει οποιοδήποτε μέγεθος. Αυτός ο κώδικας μπορεί να περιέχει οποιοδήποτε τύπο προγραμματιστικής δομής όπως κλήση σε συνάρτηση, βρόγχους και λειτουργίες ελέγχου ροής. Τα DDThrs εκτελούνται παράλληλα μεταξύ τους εκτός και αν υπάρχει κάποια εξάρτηση από ένα σε άλλο. Οι εξαρτήσεις μπορούν να εκφραστούν μέσω του TFlux Preprocessor με τη χρήση των κατάλληλων *pragma directives*. Μέσα σε κάθε DDThr οι εντολές εκτελούνται σε control-flow με τον επεξεργαστή ή και το μεταγλωττιστή να μπορούν να εφαρμόσουν οποιαδήποτε βελτίωση (όπως π.χ. out-of-order execution).

Ένα παράδειγμα του τρόπου λειτουργίας του DDThr παρουσιάζεται στο σχήμα 4.3-(α) που αφορά τις απαραίτητες πράξεις για τον υπολογισμό της διωνυμικής πιθανότητας για ***n*** πειράματα με ***k*** επιτυχίες και το σχήμα 4.3-(β) το Synchronization Graph του αντίστοιχου προγράμματος TFlux. Ο κώδικας του κάθε DDThr είναι αυτός που περιέχεται στον αντίστοιχο κόμβο του σχήματος 4.3-(α) (π.χ. ο κώδικας του DDThr 1 υπολογίζει το k!).



Σχήμα 4.3: Υπολογισμός της διωνυμικής πιθανότητας χρησιμοποιώντας DDThrs

Πέρα από τον κώδικά του, κάθε DDThr χαρακτηρίζεται από το *Thread Template* του και το σύνολο των consumers του. Το *Thread Template* είναι ένα μοναδικό προσδιοριστικό για το κάθε DDThr. Οι consumers του DDThr ***X*** είναι τα DDThrs τα οποία δεν μπορούν να ξεκινήσουν την εκτέλεσή τους πριν το DDThr ***X*** ολοκληρώσει τη δική του εκτέλεση. Ένα DDThr μπορεί να ξεκινήσει την εκτέλεσή του μόνο όταν όλοι οι producers του ολοκληρώσουν την εκτέλεσή τους. Ο αριθμός των producers ενός DDThr είναι η τιμή του Ready Count αυτού του DDThr. Για παράδειγμα, αναφερόμενοι στο Σχήμα 4.3, το DDThr 6 μπορεί να ξεκινήσει την εκτέλεσή του μόνο αφού τα DDThrs 1, 2 και 3 ολοκληρώσουν τη δική τους. Τα DDThrs τα οποία δεν έχουν καμία εξάρτηση μεταξύ τους μπορούν να εκτελεστούν παράλληλα. Ο πίνακας 4.1 συνοψίζει τα χαρακτηριστικά των DDThrs αυτού του συγκεκριμένου παραδείγματος.

Πίνακας 4.1: Τα DDThrs του παραδείγματος του σχήματος 4.3

Thread Template	Ready Count	Consumers	Producers
1	0	6	-
2	0	6	-
3	0	6	-
4	0	7	-
5	0	7	-
6	3	8	1, 2, 3
7	2	8	4, 5
8	2	-	6, 7

4.4.2 Βρόγχοι Επανάληψης στην πλατφόρμα TFlux

Ένα συστατικό το οποίο παρουσιάζεται συχνά σε μεγάλο αριθμό επιστημονικών εφαρμογών είναι αυτό του παράλληλου βρόγχου επανάληψης, δηλαδή ενός βρόγχου του οποίου όλες οι επαναλήψεις μπορούν να εκτελεστούν παράλληλα. Η πλατφόρμα TFlux προσφέρει ειδική λειτουργικότητα για την εκτέλεση τέτοιων βρόγχων, ακόμα και αν έχουν πολύ μεγάλο αριθμό επαναλήψεων, χρησιμοποιώντας DDThrs με πολλαπλά instances. Αυτό γίνεται κατορθωτό με τη χρήση του Thread Id του κάθε DDThr, το οποίο αποτελεί μέρος του πεδίου Thread Template. Κάθε instance ενός τέτοιου DDThr είναι υπεύθυνο για την εκτέλεση μιας διαφορετικής επανάληψης του βρόγχου. Αυτοί οι βρόγχοι εκτελούνται με την ανάθεση

διαφορετικού υποσυνόλου των επαναλήψεων σε κάθε kernel, δηλαδή μόνο ένας TFlux kernel εκτελεί κάθε επανάληψη του βρόγχου.

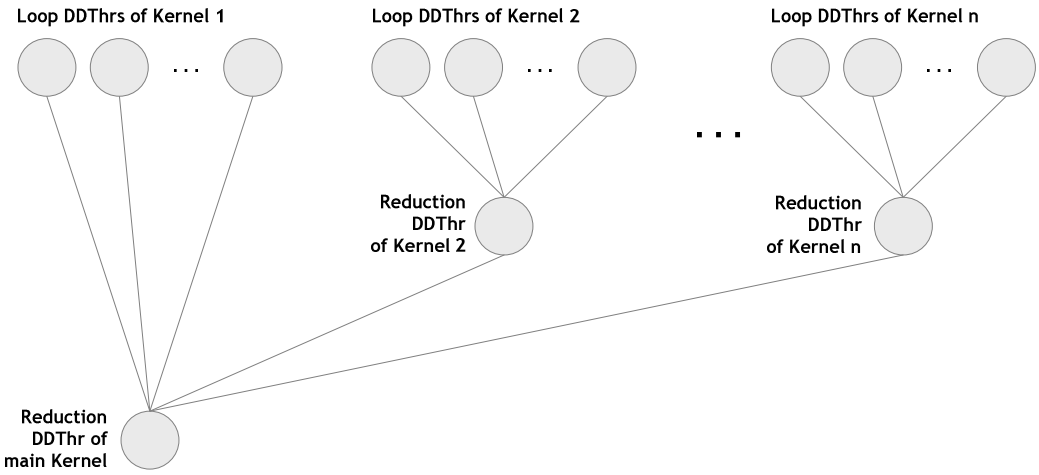
Τα TFlux loops μπορούν να εξαρτώνται είτε από απλά DDThrs είτε από άλλα TFlux loops. Συγκεκριμένα, όταν ένα TFlux loop εξαρτάται από ένα απλό DDThr ***X***, καμία επανάληψη του Loop DDThr δεν μπορεί να ξεκινήσει την εκτέλεσή της πριν το DDThr ***X*** ολοκληρώσει τη δική του εκτέλεση. Όσο για την αντίθετη περίπτωση όπου ένα απλό DDThr ***X*** εξαρτάται από ένα Loop DDThr ***Y***, τότε το ***X*** μπορεί να ξεκινήσει την εκτέλεσή του μόνο αφού όλες οι επαναλήψεις του Loop DDThr ***Y*** ολοκληρώσουν την εκτέλεσή τους. Τέλος, όταν ένα TFlux Loop ***X*** εξαρτάται από ένα άλλο TFlux Loop ***Y***, καμία επανάληψη του ***X*** δεν μπορεί να ξεκινήσει την εκτέλεσή της προτού ολοκληρώσουν την εκτέλεσή τους όλες οι επαναλήψεις του Y.

4.4.2.1 Reduction Loops

Τα Reduction Loops είναι μια ειδική κατηγορία των TFlux Loops. Η λειτουργία τους είναι να διενεργούν πράξεις reduction οι οποίες μπορούν να εκτελεστούν παράλληλα. Συγκεκριμένα, μετά το τέλος όλων των επαναλήψεων ενός Loop DDThr ενός TFlux kernel, εκτελείται ένα άλλο Reduction DDThr, το οποίο είναι υπεύθυνο για τη διενέργεια της πράξης reduction για το συγκεκριμένο TFlux kernel και το συγκεκριμένο TFlux loop. Τέλος, ένας από τους TFlux kernels είναι υπεύθυνος για τη διενέργεια του συνολικού reduction, ολοκληρωθεί η εκτέλεση όλων των επαναλήψεων του TFlux loop, χρησιμοποιώντας τις τιμές από τα μερικά reductions από τους υπόλοιπους TFlux kernels. Ένα παράδειγμα ενός τέτοιου βρόγχου φαίνεται στα σχήματα 4.4 και 4.5. Τέλος, αξίζει να σημειωθεί ότι οι εξαρτήσεις για τα Reduction Loops έχουν τις ίδιες ιδιότητες με τα κανονικά TFlux Loops.

```
sum = 0;
for (i = 0; i < n; i++)
{
      sum = sum + A[i];
}
```

Σχήμα 4.4: Παράδειγμα Reduction Loop



Σχήμα 4.5: Ο γράφος του πιο πάνω Reduction Loop

4.4.3 Blocks

Για να καταστεί δυνατή η εκτέλεση των DDThrs σύμφωνα με το μοντέλο DDM, τα μεταδεδομένα τους πρέπει να εγγραφούν στο “Thread Synchronization Unit” (TSU). Καθώς η χωρητικότητα αυτής της μονάδας είναι περιορισμένη, για να είναι δυνατή η εκτέλεση προγραμμάτων με αυθαίρετα μεγάλο μέγεθος γράφου συγχρονισμού, είναι απαραίτητη η δυναμική εγγραφή και διαγραφή αυτών των δεδομένων στο TSU. Αυτή η δυναμική διαχείριση των μεταδεδομένων επιτυγχάνεται με τη χρήση των DDM Blocks.

Ένα DDM Block είναι μια οντότητα η οποία περιέχει έναν αριθμό από DDThrs. Πιο συγκεκριμένα, όλα τα DDThrs μιας εφαρμογής πρέπει να βρίσκονται σε κάποια DDM Block. Ο μέγιστος αριθμός από DDThrs τα οποία μπορούν να μπουν σε ένα DDM Block καθορίζεται από τη χωρητικότητα του TSU. Κάθε πρόγραμμα TFlux αποτελείται από τουλάχιστον ένα DDM Block, ενώ ο αριθμός των DDM Blocks που μπορούν να υπάρχουν σε ένα πρόγραμμα εν περιορίζεται από κάποια μέγιστη τιμή.

Εκτός από τα DDThrs που περιέχουν τον αρχικό κώδικα της εφαρμογής, κάθε DDM Block περιέχει δύο επιπρόσθετα DDThrs για κάθε TFlux kernel, το *Inlet* και το *Outlet* DDThr του κάθε kernel. Το Inlet DDThr είναι υπεύθυνο για την εγγραφή στο TSU όλων των μεταδεδομένων όλων των DDThrs που θα εκτελεστούν από το συγκεκριμένο TFlux kernel και τα οποία ανήκουν στο συγκεκριμένο DDM Block. Όσο αφορά το Outlet DDThr, είναι υπεύθυνο για την απελευθέρωση αυτών των

δεσμευμένων πόρων για να δημιουργήσει χώρο για το επόμενο DDM Block, στην περίπτωση που υπάρχει επόμενο block.

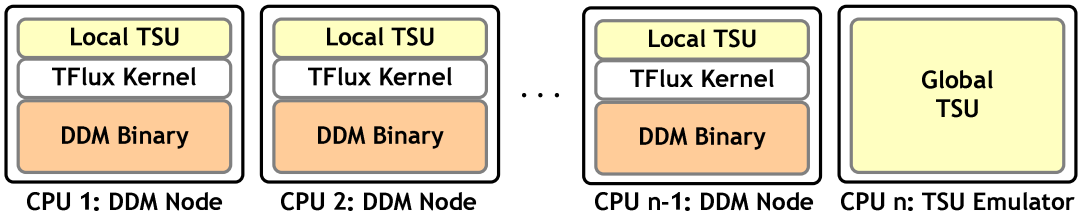
Ενώ τα DDThrs μέσα σε ένα DDM Block εκτελούνται με ένα Data-Driven τρόπο, τα διαφορετικά DDM Blocks ενός προγράμματος εκτελούνται σειριακά, δηλαδή η εκτέλεση ενός DDM Block ξεκινά μόνο αφού όλα τα DDThrs των προηγούμενων DDM Blocks ολοκληρώσουν την εκτέλεσή τους. Επίσης, είναι δυνατή η ύπαρξη κώδικα μεταξύ δύο διαδοχικών DDM Blocks. Αυτός ο κώδικας θα εκτελεστεί μετά την ολοκλήρωση του πρώτου DDM Block και η εκτέλεση του δεύτερου DDM Block θα ξεκινήσει μετά την ολοκλήρωση της εκτέλεσης αυτού του τμήματος κώδικα.

Κεφάλαιο 5 Παρουσίαση υλοποίησης TFlux-Soft

5.1 TFlux-Soft

Η υλοποίηση TFlux-Soft στοχεύει σε συνηθισμένους πολύ-πυρήνες με κοινή μνήμη, και επομένως ένα πρωτόκολλο cache coherency για την ιεραρχία μνήμης που περιέχεται στο chip. Ένα παράδειγμα τέτοιου πολυεπεξεργαστή είναι ο πρόσφατος Intel QuadCore [12]. Εδώ αξίζει να σημειώσουμε ότι, παρόλο που η υλοποίηση στοχεύει σε συστήματα πολύ-πυρήνων, δεδομένων των αναγκών, το TFlux-Soft μπορεί επίσης να εκτελεστεί σε έναν πολυεπεξεργαστή με κοινή μνήμη.

Στην περίπτωση του TFlux-Soft, το TSU είναι υλοποιημένο σαν μία οντότητα λογισμικού (software) η οποία εκτελεί τον κώδικά της σε έναν από τους πυρήνες του επεξεργαστή πολύ-πυρήνων (multicore processor) (Σχήμα 5.1). Αφού αυτή η οντότητα λογισμικού αντιγράφει τις λειτουργικότητες αυτών που υλοποιήθηκαν σε υλικό (hardware) στα D^2NOW και TFlux-Hard, ονομάζεται TSU Emulator.



Σχήμα 5.1: Παράδειγμα λειτουργίας του TFlux-Soft σε έναν επεξεργαστή πολύ-πυρήνων

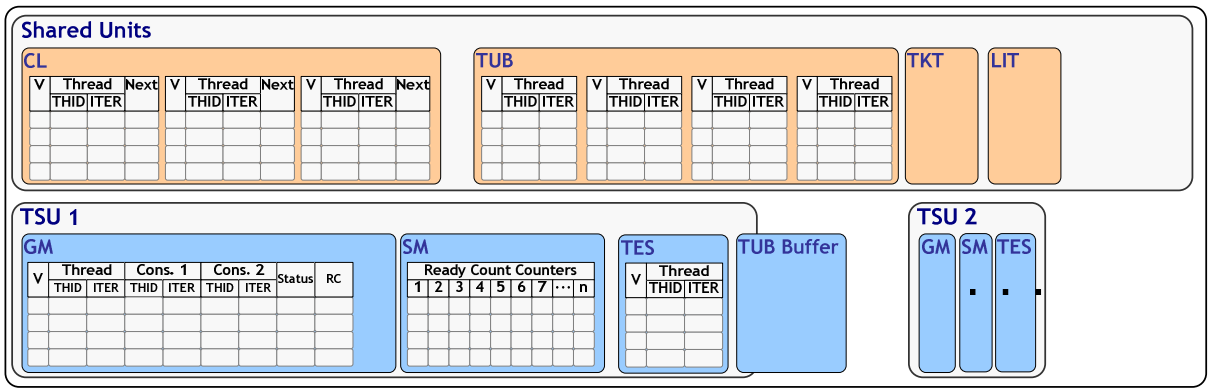
5.2 Εκτέλεση στην πλατφόρμα TFlux-Soft

Ένα TFlux πρόγραμμα ξεκινά την εκτέλεσή του δημιουργώντας *n* TFlux kernels. Ένας TFlux kernel είναι ένα POSIX Thread [6] και για σκοπούς βελτιστοποίησης της επίδοσης κάθε kernel εκτελείται από διαφορετικό CPU. Μετά τη δημιουργία τους, οι kernels ξεκινούν την εκτέλεση μιας συνάρτησης η οποία περιέχει τα DDThrs της εφαρμογής. Οι πρώτες εντολές αυτής της συνάρτησης εγγράφουν στο TSU το Inlet DDThr του πρώτου DDM Block του συγκεκριμένου kernel το οποίο έχει Ready Count ίσο με μηδέν. Ακολούθως, ο kernel "ζητά" από TSU που του αντιστοιχεί το επόμενο DDThr που είναι έτοιμο προς εκτέλεση και σε αυτό το σημείο το TSU "απαντά" με το Inlet DDThr του πρώτου Block. Με την εκτέλεση αυτού του DDThr, οι πληροφορίες των DDThrs του πρώτου Block θα φορτωθούν στο TSU και αυτό με τη σειρά του θα επιτρέψει να ξεκινήσει η εκτέλεση των DDThrs του Block. Ένας TFlux kernel ολοκληρώνει την εκτέλεσή του και τερματίζει μόνο αφού έχει εκτελέσει το Outlet DDThr του τελευταίου Block, δηλαδή αφού έχουν εκτελεστεί όλα τα DDThrs που είχαν ανατεθεί στο συγκεκριμένο kernel. Τέλος, το πρόγραμμα τερματίζει όταν έχουν τερματίσει όλα τα TFlux kernels που δημιουργήθηκαν στην αρχή.

Στο TFlux-Soft, η λειτουργικότητα του TSU παρέχεται στο επίπεδο του λογισμικού, μερικώς από τους TFlux kernels και μερικώς από μια λογισμική οντότητα που ονομάζεται "TSU Emulator". Αυτή η οντότητα είναι το κυρίως thread της εφαρμογής (δηλαδή η main), η οποία, μετά από τη δημιουργία των TFlux kernels, αρχίζει να εκτελεί τον κώδικα του TSU Emulation. Αυτός ο κώδικας τερματίζει μόνο αφού τερματίσουν όλοι οι TFlux kernels και έτσι τερματίζει και η εφαρμογή.

5.2.1 Το Thread Synchronization Unit (TSU)

Το Thread Synchronization Unit (TSU) είναι το βασικό συστατικό που χρησιμοποιεί το TFlux-Soft ούτως ώστε να επιτύχει data-driven scheduling. Κάθε TFlux kernel έχει το δικό του προσωπικό TSU. Όλα αυτά τα TSUs μαζί με κάποιες άλλες μονάδες οι οποίες είναι κοινές για όλους τους kernels ονομάζονται "TSU Group" (Σχήμα 5.2)



Σχήμα 5.2: Το Thread Synchronization Unit (TSU)

Η μονάδα Graph Memory (GM) αποθηκεύει τις *στατικές* πληροφορίες για τα DDThrs. Για κάθε DDThr, αυτές οι πληροφορίες αφορούν το Thread Template (Thread Id και Iteration Id) του και τα Thread Templates των consumers του. Εάν ένα DDThr έχει μέχρι και δύο (0, 1 ή 2) consumers, τότε τα Thread Templates τους αποθηκεύονται στο GM, ενώ σε διαφορετική περίπτωση, οι consumers αποθηκεύονται σε μια ξεχωριστή μονάδα, το Consumer List (CL), το οποίο είναι κοινό για όλους τους TFlux kernels. Επιπρόσθετα, το GM κρατά το πεδίο *Status* όπου αφορά την παρούσα κατάσταση ενός DDThr και σε άλλο πεδίο την τιμή του Ready Count του.

Για DDThrs με πολλαπλά instances, δηλαδή DDThrs που εκτελούν βρόγχους επανάληψης, οι τιμές των Ready Count για ένα αριθμό από instances βρίσκονται αποθηκευμένες στο Synchronization Memory (SM). Για αυτή την περίπτωση με τα πολλαπλά instances, το πεδίο του Ready Count (RC) στο GM εξυπηρετεί σα δείκτης στην αντίστοιχη θέση του SM όπου βρίσκονται αποθηκευμένα τα Ready Counts για το συγκεκριμένο DDThr.

Το Thread Execution Stack (TES) κρατά τα DDThrs τα οποία είναι έτοιμα για εκτέλεση, δηλαδή έχουν Ready Count ίσο με μηδέν για το συγκεκριμένο TFlux kernel. Το DDThr το οποίο εκτελείται από το TFlux kernel σε κάποια δεδομένη στιγμή είναι αυτό που βρίσκεται στην κεφαλή (top) της μονάδας TES.

Κάθε φορά που ένα DDThr ολοκληρώνει την εκτέλεσή του, χρειάζεται να μειώσει τις τιμές των Ready Count για τα DDThrs που είναι consumers του. Για αυτό το σκοπό,

ο TFlux kernel του γράφει τα Thread Templates των consumers του σε μια κοινή μονάδα που ονομάζεται Threads-to-Update Buffer (TUB).

5.2.2 Βασικές λειτουργίες του TSU

Για να καταστεί δυνατή η εκτέλεση σύμφωνα με το μοντέλο DDM το TSU πρέπει να παρέχει πέντε βασικές λειτουργίες:

(a) Τη λειτουργία "Thread Load" η οποία επιτρέπει σε κάποιο TFlux kernel να εγγράψει στο TSU του τα μετα-δεδομένα όλων των DDThrs που είναι υπεύθυνο για την εκτέλεσή τους

(b) Τη λειτουργία "Find Next Thread" η οποία χρησιμοποιείται από τον kernel για να βρει το επόμενο DDThr το οποίο είναι έτοιμο για εκτέλεση

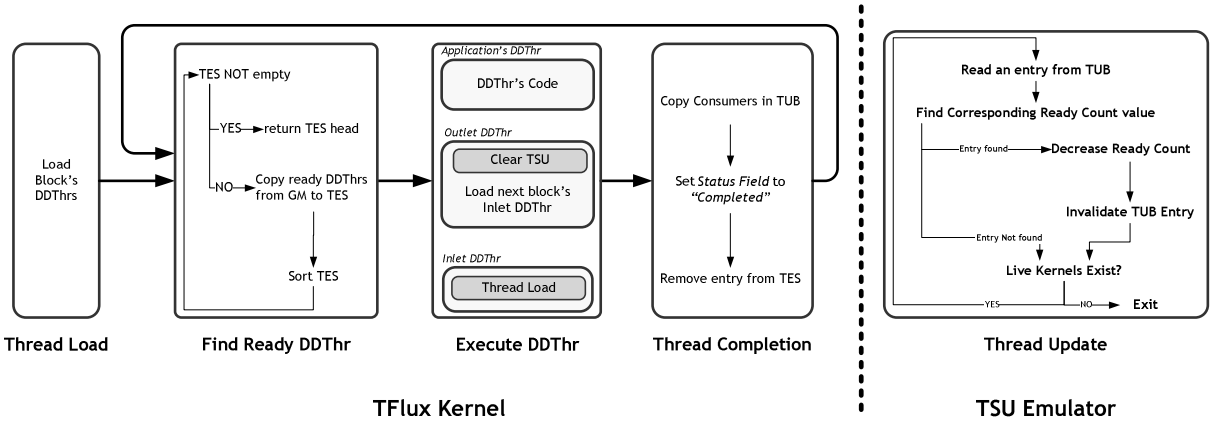
(c) Τη λειτουργία "Thread Completion" με την οποία ειδοποιείται το TSU κάθε φορά που ένα DDThr ολοκληρώνει την εκτέλεσή του

(d) Τη λειτουργία "Thread Update" με την οποία μειώνονται οι τιμές των Ready Counts των consumers ενός DDThr όταν ολοκληρώσει την εκτέλεσή του

(e) Τη λειτουργία "Clear TSU" η οποία απελευθερώνει τους δεσμευμένους πόρους από τις διάφορες μονάδες των TSUs και γενικότερα του TSU Group

Το σχήμα 5.3 παρουσιάζει τη σειρά εκτέλεσης των πιο πάνω βασικών λειτουργιών. Οι TFlux kernels ξεκινούν με το να εγγράψουν στα TSUs τους μέσω της λειτουργίας Thread Load. Ακολούθως, με τη χρήση της λειτουργίας Find Ready Thread ξεκινούν να εκτελούν ένα DDThr που είναι έτοιμο για εκτέλεση. Μόλις αυτό το DDThr ολοκληρώσει την εκτέλεσή του, εκτελείται η λειτουργία Thread Completion και αμέσως μετά η Find Ready Thread και πάλι για να συνεχίσει η εκτέλεση του προγράμματος με το επόμενο έτοιμο για εκτέλεση DDThr. Η λειτουργία Clear TSU πραγματοποιείται κατά την εκτέλεση του Outlet DDThr από το TFlux kernel. Για να γίνει δυνατή η συνέχιση της εκτέλεσης του προγράμματος από το ένα Block στο επόμενο, μετά από τη λειτουργία Clear TSU, τα Outlet DDThrs φορτώνουν στο αντίστοιχο TSU τους το Inlet DDThr του επόμενου Block. Στη συνέχεια, τα Inlet DDThrs με τη σειρά τους φορτώνουν τα DDThrs του Block τους στα TSUs μέσω της λειτουργίας Thread Load. Το Outlet DDThr του τελευταίου Block, αντί να εκτελέσει τη λειτουργία Clear TSU, αναγκάζει τον TFlux kernel του να τερματίσει.

Την ίδια στιγμή με τα πιο πάνω, ο TSU Emulator, ο οποίος είναι το κυρίως thread του προγράμματος, εκτελεί επανειλημμένα τη λειτουργία Thread Update και τερματίζει μόνο αφού όλοι οι TFlux kernels ολοκληρώσουν την εκτέλεσή τους και τερματίσουν.



Σχήμα 5.3: Οι βασικές λειτουργίες του TSU

Στις παραγράφους που ακολουθούν παρουσιάζεται η υλοποίηση των πιο πάνω βασικών λειτουργιών του TFlux-Soft, καθώς και το interface των αντίστοιχων υλοποιημένων συναρτήσεων.

5.2.2.1 Thread Load

Η λειτουργία Thread Load πραγματοποιείται από το TFlux kernel κατά την εκτέλεση ενός Inlet DDThr. Αυτή η λειτουργία φορτώνει τα μετα-δεδομένα (meta-data) των DDThrs στις δομές GM και SM. Εάν ένα DDThr έχει περισσότερους από δύο consumers, τότε τα Thread Templates τους γράφονται στο Consumer List (CL). Τέλος, το πεδίο Status των φορτωμένων DDThrs ορίζεται στην τιμή “Waiting” αφού δεν έχουν ακόμα κριθεί έτοιμα για εκτέλεση.

Η λειτουργικότητα αυτή παρέχεται από τη συνάρτηση *loadThread*, η οποία δέχεται σαν παραμέτρους το Thread Template του DDThr που θα φορτωθεί, το αρχικό Ready Count του, τον αριθμό των instances του (για τα Loop DDThrs) και το πλήθος και τα Thread Templates των consumers του.

5.2.2.2 Thread Completion

Όταν ένα DDThr ολοκληρώσει την εκτέλεσή του, ο kernel του καλεί τη λειτουργία Thread Completion η οποία αποτελείται από τρία στάδια. Αρχικά, ο kernel εισάγει τα Thread Templates των consumers του DDThr που μόλις τερμάτισε στο Thread-to-Update Buffer, ούτως ώστε να μειωθούν οι τιμές των Ready Counts τους. Έπειτα, αφαιρεί το DDThr από το TES και τελικά ανανεώνει το πεδίο Status του DDThr στο GM σε "Completed".

Η λειτουργικότητα αυτή παρέχεται από τρεις διαφορετικές συναρτήσεις, ανάλογα του τύπου του DDThr το οποίο ολοκλήρωσε την εκτέλεσή του (κανονικό ή Loop DDThr) και, αν πρόκειται για Loop DDThr, ανάλογα με το αν είναι το τελευταίο iteration του συγκεκριμένου Thread Template που θα εκτελεστεί ή αν πρέπει να κάνει *recycle* σε κάποιο επόμενο iteration του ιδίου DDThr. Το *recycle* είναι μία τεχνική που χρησιμοποιείται για την εκτέλεση Loop DDThrs με μεγάλο πλήθος επαναλήψεων. Πιο συγκεκριμένα, για να μη χρειάζεται να δεσμεύουμε χώρο στα TSUs για όλες τις επαναλήψεις ενός Loop DDThrs, αρχικά δεσμεύουμε μόνο μία καταχώρηση σε κάθε TSU με ένα μικρό πλήθος από instances (Ενδεικτικά χρησιμοποιούσαμε 32 instances για κάθε entry). Στη συνέχεια, μόλις ολοκληρώσει την εκτέλεσή του ένα instance του DDThr και υπάρχουν σε αυτό τον TFlux kernel και άλλα iterations που δεν έχουν γραφτεί ακόμα στο TSU, εγγράφει στο TSU ένα επόμενο iteration (και, αν χρειάζεται, εγγράφει και νέο DDThr στο TSU).

Για τα κανονικά DDThrs, χρησιμοποιείται η συνάρτηση *threadCompletedExecution*. Για τα Loop DDThrs, πριν κληθεί κάποια συνάρτηση, ελέγχεται αν πρέπει να κάνουν recycle σε κάποιο επόμενο iteration του ιδίου DDThr. Αν ναι, τότε χρησιμοποιείται η συνάρτηση *threadRecycled*, η οποία παίρνει σαν παραμέτρους το επόμενο iteration του DDThr το οποίο πρέπει να εκτελεστεί και το αρχικό Ready Count το οποίο θα έχει το καινούριο instance του Loop DDThr. Σε διαφορετική περίπτωση, δηλαδή αν το Loop DDThr δε θα κάνει recycle, χρησιμοποιείται η τελευταία συνάρτηση αυτής της λειτουργίας, η *forcedRcDecrease*.

5.2.2.3 Thread Update

Αυτή η λειτουργία, η οποία εκτελείται από τον TSU Emulator, μειώνει τις τιμές των Ready Counts των consumers των DDThrs που ολοκλήρωσαν την εκτέλεσή τους και εισήχθησαν στο TUB από τους TFlux kernels κατά τη λειτουργία Thread Completion. Ο Emulator προσπελαύνει το TUB και, για κάθε έγκυρη εγγραφή που συναντά, εντοπίζει την τιμή Ready Count του αντίστοιχου DDThr και τη μειώνει.

Η λειτουργικότητα αυτή παρέχεται από τη συνάρτηση *emulateTSU*, η οποία δε χρειάζεται παραμέτρους.

5.2.2.4 Find Ready Thread

Αυτή η λειτουργία εκτελείται από τον TFlux kernel και στόχος της είναι να βρεθεί το επόμενο DDThr που είναι έτοιμο για εκτέλεση. Στην κοινή περίπτωση, όπου δηλαδή το TES δεν είναι άδειο, η λειτουργία τερματίζει επιστρέφοντας το DDThr που βρίσκεται στην κεφαλή (top) του TES. Σε διαφορετική περίπτωση, όπου δηλαδή το TES είναι άδειο, ο TFlux kernel προσπελαύνει πρώτα το GM του, αντιγράφει τα Thread Templates όλων των έτοιμων για εκτέλεση DDThrs στο TES και θέτει το πεδίο Status τους σε “In TES”. Αν το GM δεν περιέχει καθόλου έτοιμα για εκτέλεση DDThrs, τότε η διαδικασία επαναλαμβάνεται μέχρι να βρεθεί ένα τέτοιο DDThr.

Η λειτουργικότητα αυτή παρέχεται από τη συνάρτηση *currentlyExecutedThread*, η οποία επίσης δε χρειάζεται παραμέτρους.

5.2.2.5 Clear TSU

Αυτή η λειτουργία εκτελείται από το TFlux kernel κατά την εκτέλεση του Outlet DDThr και το αποτέλεσμά της είναι η απελευθέρωση των δεσμευμένων πόρων από τα TSUs και το TSU Group γενικά μέχρι την εκτέλεση του συγκεκριμένου Block.

Η λειτουργικότητα αυτή παρέχεται από τη συνάρτηση *clearTSU*.

Κεφάλαιο 6 Preprocessor και προγράμματα για TFlux-Soft

Ο Data-Driven Multithreading C Pre-Processor (DDMCPP) [22] είναι ένα εργαλείο της *Σειράς Μεταγλώττισης* της πλατφόρμας TFlux, το οποίο παίρνει σαν είσοδο ένα κανόνικο πρόγραμμα C μαζί με κάποια *DDM pragma directives* και παράγει ένα άλλο πρόγραμμα C που περιέχει όλο τον κώδικα του *runtime support* και τις κλήσεις στις συναρτήσεις της πλατφόρμας TFlux, ούτως ώστε να είναι δυνατή η εκτέλεση του προγράμματος πάνω σε μια αρχιτεκτονική της πλατφόρμας TFlux. Αυτό το εργαλείο είναι λογικά χωρισμένο σε δύο μέρη, το *front-end* και το *back-end*.

Το πλεονέκτημα του Preprocessor είναι ότι ο χρήστης-προγραμματιστής μπορεί να εκφράσει τον παραλληλισμό μίας εφαρμογής χρησιμοποιώντας ένα σύνολο από *απλά directives*, επιταχύνοντας και διευκολύνοντας έτσι τη διαδικασία παραλληλοποίησης εφαρμογών για την πλατφόρμα TFlux. Επιπρόσθετα, με τη χρησιμοποίηση του DDMCPP, είναι δυνατό να κατανοήσουμε καλύτερα τις εργασίες που ένας πλήρως αυτοματοποιημένος μεταγλωττιστής θα μπορούσε να είναι υπεύθυνος. Έτσι, θα ήταν δυνατό να χρησιμοποιήσουμε τον Preprocessor στο μέλλον σαν ένα εργαλείο που θα βοηθούσε στην ανάπτυξη ενός τέτοιου μεταγλωττιστή.

Η χρήση του Preprocessor επιτρέπει στον χρήστη-προγραμματιστή να επικεντρωθεί στον παραλληλισμό. Όλες οι άλλες λεπτομέρειες, όπως η μεταφορά δεδομένων ή ο συγχρονισμός, χειρίζονται αυτόματα, είτε από τον Preprocessor, είτε από την ίδια την πλατφόρμα TFlux.

Το γεγονός ότι στο μοντέλο DDM οι εξαρτήσεις δεδομένων είναι έμφυτες στο μοντέλο επιτρέπει στα εκτελέσιμα της πλατφόρμας TFlux να είναι πιο αποδοτικά, αφού δε χρειάζεται να χειρίζονται θέματα συγχρονισμού. Επίσης, τα χαρακτηριστικά του μοντέλου DDM επιτρέπουν στον προγραμματιστή να εκφράσει τον παραλληλισμό με έναν πιο φυσικό τρόπο σε σύγκριση με άλλα παρόμοια μοντέλα, όπως το OpenMP ή το MPI.

6.1 Data-Driven Multithreading C Pragma directives

Ο πρωταρχικός στόχος του DDM Preprocessor είναι να κρύψει τις λεπτομέρειες του συστήματος *runtime support* από τον προγραμματιστή. Έτσι, για την ανάπτυξη μίας παράλληλης εφαρμογής, ο προγραμματιστής θα πρέπει μόνο να περιγράψει τα παράλληλα κομμάτια του προγράμματος και δε χρειάζεται να ανησυχεί για εργασίες που θα γίνονται κατά την εκτέλεση του προγράμματος, όπως τη λειτουργία του TSU, αλλά και την ίδια την υλοποίηση της πλατφόρμας TFlux.

6.1.1 DDM Block

```
#pragma ddm block B_ID
```
(α)

```
#pragma ddm endblock
```
(β)

Σχήμα 6.1: Παράδειγμα καθορισμού ενός *DDM Block* με *DDM pragma directive*

Όπως εξηγήσαμε σε προηγούμενα κεφάλαια, όλα τα DDThrs ενός προγράμματος TFlux πρέπει να βρίσκονται μέσα σε κάποιο *DDM Block*. Έτσι, είναι απαραίτητο ένα *pragma directive* για τη δήλωση των ορίων και της δομής αυτών των blocks. Το *DDM pragma* directive που βλέπουμε στο Σχήμα 6.1-(α) καθορίζει την αρχή του *DDM Block B_ID*, όπου το ***B_ID*** μπορεί να έχει οποιαδήποτε ακέραια θετική τιμή. Αντίστοιχα, το *directive* που βλέπουμε στο Σχήμα 6.1-(β), καθορίζει το τέλος του τελευταίου "ενεργού" *DDM Block*.

6.1.2 DDM Thread

```
#pragma ddm thread T_ID kernel K_ID depends(T_List)
```
(α)

```
#pragma ddm endthread
```
(β)

Σχήμα 6.2: Παράδειγμα καθορισμού ενός απλού DDThr με το ανάλογο *DDM directive*

Με τη χρήση του *DDM pragma directive* που βλέπουμε στο Σχήμα 6.2-(α), ορίζουμε το DDThr με ταυτότητα ***T_ID***, η οποία είναι ένας μοναδικός ακέραιος αριθμός μεταξύ ***1*** και ***MAX_THREADS***. Το ***K_ID*** αντιπροσωπεύει την ταυτότητα του TFlux Kernel ο οποίος θα αναλάβει την εκτέλεση του συγκεκριμένου DDThr και είναι μία ακέραια τιμή μεταξύ ***1*** και του αριθμού των TFlux Kernels (ο οποίος, όπως θα δούμε αργότερα, ορίζεται επίσης με ένα *DDM pragma directive*).

Το όρισμα *depends()* χρησιμοποιείται για να ορίσουμε τις εξαρτήσεις δεδομένων μεταξύ DDThrs και το ***T_List*** περιέχει όλα τα DDThrs (είτε απλά είτε Loop DDThr) τα οποία είναι producers αυτού του DDThr. Αν δεν υπάρχουν καθόλου producers, τότε δε χρειάζεται να γραφτεί αυτό το όρισμα του *DDM* directive. Τέλος, το *DDM pragma directive* που βλέπουμε στο Σχήμα 6.2-(β) χρησιμοποιείται για να καθοριστεί το τέλος του τελευταίου "ενεργού" DDThr.

6.1.3 TFlux Loop

```
#pragma ddm for thread T_ID depends(T_List)
      for (i = LOWER_BOUND; i < UPPER_BOUND; i++)
            LOOP BODY
#pragma ddm endfor
```

Σχήμα 6.3: Παράδειγμα δήλωσης *TFlux Loop DDThr* με *DDM directive*

Με τη χρήση των *DDM pragma directives* όπως βλέπουμε στο Σχήμα 6.3, γίνεται δυνατός ο καθορισμός ενός βρόγχου επανάληψης ο οποίος θα εκτελεστεί παράλληλα από όλους του TFlux kernels. Οι επαναλήψεις αυτού του TFlux Loop θεωρούνται ότι είναι ανεξάρτητες μεταξύ τους. Το ***T_ID*** είναι και πάλι το *Thread Id* του συγκεκριμένου Loop DDThr.

Το *TFlux Loop pragma directive* επίσης παρέχει στον προγραμματιστή δύο επιπλέον επιλογές, το ***unroll*** και το ***reduction***, οι οποίες μπορούν να χρησιμοποιηθούν και μαζί στο ίδιο TFlux Loop DDThr.

```
#pragma ddm for thread T_ID unroll UNROLL_FACTOR
```

Σχήμα 6.4: Χρήση επιλογής *unroll* σε ένα TFlux Loop DDThr

Στο Σχήμα 6.4 βλέπουμε τον τρόπο χρήσης της επιλογής *unroll*, με την οποία γίνεται αυτόματο *unroll* του Loop body με βάση το ***UNROLL_FACTOR*** (που είναι ένας ακέραιος θετικός αριθμός πολλαπλάσιο του ***2***), δηλαδή για παράδειγμα, αν το ***UNROLL_FACTOR*** είναι ίσο με 32, τότε κάθε *instance* του TFlux Loop DDThr θα περιέχει 32 επαναλήψεις του αρχικού βρόγχου επανάληψης αντί για μία μόνο. Αυτό είναι ιδιαίτερα χρήσιμο για περιπτώσεις βρόγχων επαναλήψεως που έχουν μεγάλο πλήθος επαναλήψεων, αλλά μικρό μέγεθος κώδικα στο loop body.

```
#pragma ddm for thread T_ID reduction REDCUTION_CALCULATION
```

Σχήμα 6.5: Χρήση επιλογής *reduction* σε ένα TFlux Loop DDThr

Στο Σχήμα 6.5 βλέπουμε τον τρόπο χρήσης της επιλογής *reduction*, η οποία είναι απαραίτητη σε περιπτώσεις Loop DDThrs που χρειάζονται μία πράξη reduction στο τέλος τους και η οποία μπορεί να παραλληλοποιηθεί (π.χ. άθροισμα). Το ***REDUCTION_CALCULATION*** είναι η περιγραφή της πράξης reduction η οποία επιθυμεί ο προγραμματιστής να εκτελεστεί στο τέλος της εκτέλεσης όλων των επαναλήψεων του TFlux Loop DDThr. Αυτή η πράξη μπορεί να είναι από μία απλή πρόσθεση μέχρι και κλήση μίας συνάρτησης (η οποία δηλώνεται και υλοποιείται από τον προγραμματιστή) για τη διενέργεια κάποιας πιο περίπλοκης πράξης reduction. Παραδείγματα των δύο αυτών περιπτώσεων φαίνονται στο Σχήμα 6.6 και Σχήμα 6.7 αντίστοιχα. Σημειώνεται ότι για το παράδειγμα του σχήματος

```
#pragma ddm for thread 1 reduction sum + double totalSum
```

Σχήμα 6.6: Παράδειγμα της επιλογής ***reduction*** με απλή πράξη (πρόσθεση) σε ένα *DDM Loop pragma directive*

```
#pragma ddm for thread 1 reduction foo(max, min, int localMax, int localMin)
```

Σχήμα 6.7: Παράδειγμα της επιλογής ***reduction*** με κλήση σε συνάρτηση σε ένα *DDM Loop pragma directive*

6.1.4 Διαχείριση Συστήματος και Μεταβλητών

6.1.4.1 Καθορισμός αριθμού TFlux Kernels

```
#pragma ddm kernel 8
```

Σχήμα 6.8: Παράδειγμα καθορισμού του αριθμού των TFlux Kernels σε ***8***

Στο Σχήμα 6.8 φαίνεται ένα παράδειγμα χρήσης του *DDM pragma directive* με το οποίο καθορίζεται από τον προγραμματιστή (σε οποιοδήποτε σημείο του C κώδικα) ο αριθμός των TFlux Kernels που επιθυμεί να χρησιμοποιηθούν κατά την εκτέλεση του προγράμματος. Αν και λειτουργικά, δεν υπάρχει περιορισμός για αυτό τον αριθμό, για λόγους απόδοσης, συνήθως τίθεται κατά ***2*** μικρότερος από τον αριθμό υπολογιστικών πόρων που υπάρχουν στη μηχανή που αναμένεται να εκτελεστεί το πρόγραμμα TFlux. Αυτό γίνεται ούτως ώστε να αφήνεται ένας υπολογιστικός πόρος

για την εκτέλεση της ***main*** συνάρτησης και ακόμα ένας υπολογιστικός πόρος για το Λειτουργικό Σύστημα.

6.1.4.2 Δήλωση ιδιωτικών (private) μεταβλητών

```
#pragma ddm private var int      temp1
#pragma ddm private var double   temp2 10
#pragma ddm private var myStruct temp3 20 10
```

(α)

```
int      temp1;
double   temp2[10];
myStruct temp3[20][10];
```

(β)

Σχήμα 6.9: Παραδείγματα δηλώσεων ιδιωτικών μεταβλητών. (α): Δομή και χρήση κατάλληλων *DDM pragma directives*. (β): Αντίστοιχες δηλώσεις μεταβλητών σε C κώδικα

Στο Σχήμα 6.9-(α) φαίνεται η δομή και η χρήση κατάλληλων *DDM pragma directives* για την ιδιωτικοποίηση κάποιων μεταβλητών από τον προγραμματιστή. Στις τρεις περιπτώσεις ιδιωτικοποίησης που παρουσιάζονται βλέπουμε ότι είναι δυνατή η δήλωση όποιου τύπου μεταβλητής επιθυμεί ο προγραμματιστής, συμπεριλαμβανομένων και δομών που έχει κατασκευάσει ο ίδιος ή από κάποια βιβλιοθήκη. Επίσης, είναι δυνατή η ιδιωτικοποίηση πινάκων με στοιχεία, όπως βλέπουμε από τα παραδείγματα μονοδιάστατου και δυσδιάστατου πίνακα. Στο Σχήμα 6.9-(β) βλέπουμε τις αντίστοιχες δηλώσεις μεταβλητών σε C κώδικα, οι οποίες θα προέλθουν από τη χρήση αυτών των συγκεκριμένων *DDM pragma directives*. Αυτές οι δηλώσεις θα βρίσκονται στον τελικό C κώδικα που θα παραχθεί από τον DDMCPP Preprocessor σε κάθε TFlux Kernel.

6.1.4.3 Καθορισμός αρχής προγράμματος

```
#pragma ddm startprogram
```

Σχήμα 6.10: Παράδειγμα ορισμού αρχής του προγράμματος με τη χρήση κατάλληλου *DDM pragma directive*

Στο Σχήμα 6.10 παρουσιάζεται το *DDM pragma directive* με το οποίο ο προγραμματιστής μπορεί να δηλώσει την αρχή του προγράμματός του. Αυτό πρέπει να γίνεται στη ***main*** συνάρτηση του προγράμματος TFlux (πριν να επεξεργαστεί από τον Preprocessor) και στο σημείο μετά από τις δηλώσεις των μεταβλητών του

προγράμματος. Σκοπός αυτού του *DDM pragma directive* είναι να μεταφερθούν όλες οι δηλώσεις μεταβλητών που στο αρχικό πρόγραμμα βρίσκονται στη ***main*** συνάρτηση πριν από αυτό το *DDM directive* σε *global*, για να έχουν πρόσβαση σε αυτές οι *TFlux Kernels*, λόγω της φύσεως και του τρόπου λειτουργίας των *POSIX Threads*.

6.1.4.4 Ανάθεση ταυτότητας kernel σε μεταβλητή

```
#pragma ddm kernelid <var>
```

Σχήμα 6.11: Ανάθεση της ταυτότητας ενός *TFlux Kernel* σε μεταβλητή

Στο Σχήμα 6.11 φαίνεται το *DDM pragma directive* με το οποίο ο προγραμματιστής μπορεί να αναθέσει την ταυτότητα του *TFlux Kernel* ο οποίος εκτελεί το συγκεκριμένο κομμάτι κώδικα σε μία δεδομένη στιγμή σε μία μεταβλητή τύπου ακεραίου. Αυτό είναι πολύ χρήσιμο σε περιπτώσεις προγραμμάτων όπου χρειάζεται να εκτελείται διαφορετική εργασία ή διαφορετικά μέρη μίας εργασίας από διαφορετικούς *TFlux Kernels*.

6.2 Περιγραφή Preprocessor

Ο *Data-Driven Multithreading C Preprocessor (DDMCPP)* είναι ένα εργαλείο το οποίο δέχεται σαν είσοδο ένα πρόγραμμα με απλό κώδικα C μαζί με κάποια από τα *DDM pragma directives* (όπως παρουσιάστηκαν στο προηγούμενο μέρος αυτού του κεφαλαίου) και παράγει ένα πρόγραμμα C το οποίο περιέχει όλες τις απαραίτητες κλήσεις σε συναρτήσεις του TSU ούτως ώστε να μπορεί να εκτελεστεί σε κάποια από τις υλοποιήσεις της πλατφόρμας TFlux. Επιπρόσθετα, στο παραγόμενο C κώδικα, εμπεριέχεται και ο κώδικας του συστήματος *Runtime Support*. Αυτό το εργαλείο είναι χωρίζεται λογικά σε δύο μέρη, το *front-end* και το *back-end*, τα οποία περιγράφονται σε μεγαλύτερη λεπτομέρεια στη συνέχεια.

6.2.1 Front-End

Το *front-end* του *DDMCPP* είναι ένα εργαλείο *parser* βασισμένο στα εργαλεία *flex* και *bison*. Ο *parser* αναγνωρίζει τα *DDM pragma directives* που παρουσιάστηκαν σε προηγούμενο μέρος αυτού του κεφαλαίου. Το *front-end* είναι ανεξάρτητο από την αρχιτεκτονική, δηλαδή καθήκον του είναι η αναγνώριση των *DDM pragma directives* και η μεταφορά των πληροφοριών στο *back-end* κομμάτι του Preprocessor ούτως ώστε να παραχθεί ο κώδικας για τη συγκεκριμένη αρχιτεκτονική που επιθυμεί ο προγραμματιστής.

6.2.2 Back-End

Το *back-end* υλοποιήθηκε ως οι ενέργειες της γραμματικής του *bison* για τα *DDM pragma directives* και εξαρτάται άμεσα από την αρχιτεκτονική στην οποία επιθυμεί ο προγραμματιστής να εκτελέσει την εφαρμογή. Η εργασία του *back-end* είναι η δημιουργία του κώδικα που είναι απαραίτητος για το σύστημα *Runtime Support* της συγκεκριμένης αρχιτεκτονικής της πλατφόρμας TFlux, όπως τον κώδικα των *TFlux Kernels*, το βρόγχο του *Thread Select*, τις λειτουργίες εγγραφής δεδομένων στα *TSUs* και άλλες εργασίες. Σημειώνεται ότι ο καθορισμός της αρχιτεκτονικής γίνεται από τον προγραμματιστή μέσω μιας παραμέτρου που δίνει στον Preprocessor τη στιγμή της εκτέλεσής του.

6.2.3 Χρήση

```
ddmcpp –t soft –o <OUTPUT FILE> –i <INPUT FILE>
```

Σχήμα 6.12: Χρήση DDMCPP Preprocessor

Η εντολή για χρήση του Preprocessor φαίνεται στο Σχήμα 6.12. Ο Preprocessor παράγει τον τελικό C κώδικα (<***OUTPUT FILE***>) με βάση το <***INPUT FILE***>, που περιέχει C κώδικα μαζί με *DDM pragma directives*. Ο παραγόμενος κώδικας θα είναι για την υλοποίηση TFlux-Soft της πλατφόρμας TFlux (λόγω του ορίσματος ***-t soft***), ενώ για την παραγωγή κώδικα για την ίδια εφαρμογή (δηλαδή αρχείο εισόδου)

αλλά για κάποια άλλη υλοποίηση της πλατφόρμας χρειάζεται απλά η αλλαγή του ορίσματος *-**t soft*** (π.χ. για την υλοποίηση TFlux-Hard σε *-**t hard***).

6.3 Δομές Δεδομένων

Ο Preprocessor, κατά τη λειτουργία προ-επεξεργασίας ενός κώδικα εισόδου και την ανάλυσή του, χρησιμοποιεί διάφορες δομές δεδομένων για την αποθήκευση απαραίτητων πληροφοριών για την παραγωγή του τελικού C κώδικα. Αυτές οι πληροφορίες αφορούν τα DDThrs, τα DDM Blocks, μεταβλητές του προγράμματος, TFlux Loops και άλλα στοιχεία. Στο πρώτο πέρασμα από το αρχικό πρόγραμμα εισόδου, αυτές οι δομές δεδομένων δημιουργούνται και εγγράφονται σε αυτές όλες οι απαραίτητες πληροφορίες και χαρακτηριστικά του προγράμματος, με βάση τα *DDM pragma directives* που ανιχνεύονται στον κώδικα. Στη συνέχεια, κατά το δεύτερο πέρασμα από τον αρχικό κώδικα εισόδου, οι πληροφορίες που υπάρχουν πλέον σε αυτές τις δομές δεδομένων χρησιμοποιούνται για την παραγωγή του τελικού προγράμματος. Πιο κάτω θα μελετήσουμε με κάποια λεπτομέρεια τις βασικές δομές δεδομένων που χρησιμοποιούνται και τα χαρακτηριστικά τους.

6.3.1 DDM Blocks

Η καταγραφή των DDM Blocks από τα οποία αποτελείται ένα πρόγραμμα TFlux γίνεται μέσω μίας λίστας που περιέχει όλα τα DDM Blocks του προγράμματος, καθώς και αναγκαίες πληροφορίες για το καθένα, όπως την ταυτότητα του και δείχτες προς το προηγούμενο και επόμενο DDM Block (για τη σωστή δημιουργία και λειτουργία των *Inlet* και *Outlet DDThrs* κάθε DDM Block). Η δομή των πληροφοριών για κάθε DDM Block φαίνεται στο Σχήμα 6.13.

```
struct ddmBlock
{
      int blockId;
      struct ddmBlock* next;
      struct ddmBlock* prev;
};
```

Σχήμα 6.13: Δομή δεδομένων για αποθήκευση των DDM Blocks

6.3.2 DDThrs

Η καταγραφή και αποθήκευση των DDThrs ενός προγράμματος TFlux από τον Preprocessor είναι απαραίτητη. Για αυτό τον σκοπό, κατά την ανάλυση (πρώτο πέρασμα) του προγράμματος, δημιουργείται μία λίστα με όλα τα DDThrs του προγράμματος (απλά, Loop και Reduction), καθώς και διάφορες αναγκαίες πληροφορίες για το κάθε DDThrs, όπως τον τύπο του (Inlet, Outlet, απλό, Loop ή Reduction), το DDM Block στο οποίο ανήκει και τον TFlux Kernel ο οποίος είναι υπεύθυνος για την εκτέλεσή του (αν υπάρχει), το Ready Count του και τους producers και consumers του. Η δομή των πληροφοριών αυτών φαίνεται στο Σχήμα 6.14. Κάποια από τα στοιχεία που φαίνονται στο σχήμα ισχύουν μόνο για μερικά DDThrs, ανάλογα με τον τύπο ή τα χαρακτηριστικά τους (π.χ. το ***consumerList*** και το ***dependsOnList*** ισχύουν μόνο για DDThrs που έχουν consumers ή producers αντίστοιχα, ενώ το ***threadsToLoadList*** ισχύει μόνο για *Inlet DDThrs* και το ***loop*** ισχύει μόνο για *Loop DDThrs*).

```
struct ddmThread
{
      struct threadTemplate threadTemplate;
      int threadType;
      int blockId;
      int kernelId;
      int readyCount;
      struct thread_TCI* consumerList;
      struct thread_T* dependsOnList;

      struct thread_TCI* threadsToLoadList;
      struct loopTemplate loop;
};
```

Σχήμα 6.14: Δομή δεδομένων για αποθήκευση των DDThrs

6.3.3 Loops

Για τα Loop DDThrs είναι απαραίτητη η αποθήκευση περισσότερων πληροφοριών, όπως το πλήθος των επαναλήψεων, τον παράγοντα του ***unroll*** (αν υφίσταται), το κατώτατο και ανώτατο όριο, το όνομα της μεταβλητής ελέγχου του βρόγχου (για σκοπούς αντικατάστασής του με το *Iteration Id* του συγκεκριμένου DDThr μέσα στο

σώμα του βρόγχου) και αν το Loop DDThr χρειάζεται μία πράξη *reduction* στο τέλος του και, αν ναι, τις μεταβλητές που εμπλέκονται και την πράξη (ή την κλήση σε συνάρτηση). Στο Σχήμα 6.15 φαίνεται καλύτερα η δομή των επιπλέον πληροφοριών για τα Loops. Σημειώνεται εδώ ότι το κάθε Loop DDThr υπάρχει και στη δομή δεδομένων για τα DDThrs που είδαμε στο προηγούμενο μέρος και υπάρχει δείχτης στις επιπλέον πληροφορίες του που παρουσιάστηκαν εδώ.

```
struct loopTemplate
{
      int   unrollFactor;
      int   iterationsForLoop;
      int   lowerBound;
      int   upperBound;
      char* controlVariable;

      char   reductionRule;
      char*  reductionFunction;
      struct dataList* reductionLocalVarList;
      struct dataList* reductionGlobalVarList;
};
```

Σχήμα 6.15: Δομή επιπλέον πληροφοριών για Loop DDThrs

6.4 Διαδικασία παραγωγής κώδικα για TFlux-Soft

Ένα C πρόγραμμα, για να εκτελεστεί στην πλατφόρμα TFlux-Soft, πρέπει να μπορεί να επικοινωνεί με το runtime support της πλατφόρμας με επιτυχία και με όσο το δυνατό πιο αποδοτικό τρόπο, ούτως ώστε να μειώνεται τυχόν overhead της παραλληλοποίησής του και γενικότερα της μεταφοράς του στην πλατφόρμα. Για να καταστεί αυτό δυνατό, αυτό το πρόγραμμα πρέπει να τηρεί κάποιους κανόνες, όπως για παράδειγμα να γίνονται include οι κατάλληλες βιβλιοθήκες, να δηλώνονται κάποιες απαραίτητες δομές δεδομένων και μεταβλητές και γενικά να έχει κάποια μορφή που να επιτρέπει την επικοινωνία με το TSU, το οποίο συμπεριφέρεται σα μια βιβλιοθήκη παρέχοντας τις συναρτήσεις και λειτουργίες που είδαμε σε προηγούμενο κεφάλαιο.

Στόχος μας ήταν να καταστούν όλα τα πιο πάνω δυνατά χωρίς ο χρήστης-προγραμματιστής να πρέπει να ξέρει οποιαδήποτε λεπτομέρεια για την υλοποίηση

και απλά να γράφει το πρόγραμμα που επιθυμεί σε απλή γλώσσα C με την προσθήκη *DDM pragma directives* (Υποκεφάλαιο 6.1)για την παραλληλοποίηση του προγράμματος. Έτσι, για να επιτευχθεί αυτός ο στόχος, μελετήθηκαν κάποιες απλές εφαρμογές που εκτελέστηκαν στην υλοποίηση TFlux-Soft για να βρεθούν τα απαραίτητα στοιχεία που πρέπει να περιέχουν τα C προγράμματα, καθώς και η μορφή και η ροή του κώδικα και. Στη συνέχεια, τροποποιήθηκε ο *DDMCPP Preprocessor* ούτως ώστε, με βάση τα ίδια *DDM pragma directives*, να παράγει κώδικα με βάση αυτή τη μορφή και κανόνες για να μπορεί αργότερα ο κώδικας να μεταγλωττιστεί (χρησιμοποιώντας ένα συνηθισμένο μεταγλωττιστή όπως ***gcc***) και να εκτελεστεί στην υλοποίηση TFlux-Soft της πλατφόρμας. Πολύ σημαντικό είναι ότι ο αρχικός κώδικας που γράφει ο προγραμματιστής (C + *DDM directives)* δεν περιέχει τίποτα δεσμευτικό για τη συγκεκριμένη υλοποίηση της πλατφόρμας προς την οποία απευθύνεται και ο προγραμματιστής θα μπορεί, με ακριβώς τον ίδιο κώδικα, να παράξει τελικό κώδικα για όποια συγκεκριμένη υλοποίηση της πλατφόρμας επιθυμεί, αλλάζοντας μόνο μια επιλογή στην εντολή προ-επεξεργασίας του Preprocessor (Υποκεφάλαιο 6.2.3).

Στη συνέχεια θα παρουσιάσουμε τα βασικά στοιχεία της λειτουργίας του Preprocessor για την παραγωγή κώδικα για την υλοποίηση TFlux-Soft, κάνοντας αναφορά στα *DDM pragma directives* που θα πρέπει να χρησιμοποιήσει ο προγραμματιστής ανάλογα με τη συγκεκριμένη εφαρμογή που επιθυμεί να παραλληλοποιήσει, καθώς επίσης και στον αντίστοιχο παραγόμενο τελικό κώδικα μετά την επεξεργασία από τον Preprocessor. Σημειώνεται ότι ο Preprocessor διενεργεί δύο περάσματα στον αρχικό κώδικα. Κατά το πρώτο, κτίζει το Γράφο Συγχρονισμού του προγράμματος, χρησιμοποιώντας τις δομές δεδομένων που είδαμε προηγουμένως (Υποκεφάλαιο 6.3), ενώ στο δεύτερο πέρασμα, χρησιμοποιώντας το Γράφο Συγχρονισμού και τις πληροφορίες που συλλέχθηκαν προηγουμένως, παράγει τον τελικό κώδικα εξόδου για τη συγκεκριμένη υλοποίηση της πλατφόρμας που επιθυμεί ο χρήστης. Στην παρακάτω περιγραφή θα επικεντρωθούμε κυρίως στο δεύτερο πέρασμα, αφού σε αυτό γίνεται η παραγωγή του κώδικα για την πλατφόρμα TFlux-Soft.

6.4.1 Αρχικά στοιχεία και global μεταβλητές

Αρχικά, ο Preprocessor συμπεριλαμβάνει στον τελικό κώδικα κάποια απαραίτητα *include* βιβλιοθηκών και περιγράφει κάποιες δομές δεδομένων που χρησιμοποιούνται από το *TFlux Runtime Support*. Ακολούθως, είναι απαραίτητο να οριστούν κάποιες σταθερές για τα μεγέθη των δομών του ***TSU*** και την εκτέλεση του προγράμματος. Επίσης εδώ ορίζεται από τον Preprocessor το πλήθος των TFlux Kernels που επιθυμεί ο προγραμματιστής να δημιουργηθούν κατά την εκτέλεση της εφαρμογής. Αυτό γίνεται μέσω του κατάλληλου *DDM pragma directive*, όπως φαίνεται στο Σχήμα 6.16-(α), το οποίο μπορεί να τοποθετηθεί σε οποιοδήποτε σημείο του αρχικού κώδικα, ενώ ο παραγόμενος κώδικας από τον Preprocessor φαίνεται στο Σχήμα 6.16-(β).

```
#pragma ddm kernel <TFLUX_KERNELS_NUM>
```
(α)

```
#define KERNEL_NUM <TFLUX_KERNELS_NUM>
```
(β)

Σχήμα 6.16: (α): *DDM directive* για καθορισμό αριθμό *TFlux Kernels* από τον προγραμματιστή. (β): Παραγόμενος C κώδικας από τον Preprocessor

Στη συνέχεια, δηλώνονται από τον Preprocessor στον τελικό κώδικα κάποιες global μεταβλητές, οι οποίες είναι απαραίτητες για τη λειτουργία του προγράμματος. Στο Σχήμα 6.17 παρουσιάζονται δύο από αυτές τις μεταβλητές, ένας ακέραιος ο οποίος κρατάει τον αριθμό των live TFlux kernels (δηλαδή που δεν ολοκλήρωσαν την εκτέλεσή τους ακόμα) για το σκοπό αναγνώρισης του τέλους του προγράμματος (όταν δηλαδή τελειώσουν όλοι οι TFlux kernels) και μία μεταβλητή τύπου pthread_mutex_t για πραγματοποίηση αμοιβαίου αποκλεισμού κατά την αλλοίωση του αριθμού των live kernels από τους ίδιους τους kernels. Ο αμοιβαίος αποκλεισμός για την αλλοίωση της συγκεκριμένης μεταβλητής είναι απαραίτητος, αφού μπορεί δύο ή περισσότεροι TFlux kernels να ολοκληρώσουν την εκτέλεσή τους ταυτόχρονα και να επιχειρήσουν να αλλοιώσουν τη μεταβλητή.

```
int             __ddmVar_numberOfLiveKernels = 0;
pthread_mutex_t __ddmVar_mutexNumberOfLiveKernels;
```

Σχήμα 6.17: Global variables για χρήση του TSU

Η δήλωση των συγκεκριμένων μεταβλητών ως global είναι απαραίτητη, για το λόγο ότι χρησιμοποιούνται και από τη main συνάρτηση αλλά και από τους TFlux kernels, οι οποίοι, λόγω της φύσεως των POSIX Threads, τρέχουν σε διαφορετική συνάρτηση.

Ακολούθως, ο Preprocessor μεταφέρει τις δηλώσεις των μεταβλητών της ***main*** συνάρτησης του αρχικού σε global μεταβλητές στον τελικό κώδικα. Αυτό γίνεται για να έχουν πρόσβαση οι TFlux kernel σε αυτές, αφού θα τρέχουν τον κώδικα τους σε διαφορετική συνάρτηση από τη ***main*** του τελικού κώδικα λόγω του τρόπου λειτουργίας των *POSIX Threads*. Για την αναγνώριση του τέλους των δηλώσεων μεταβλητών της ***main***, ο προγραμματιστής είναι υπεύθυνος να τοποθετήσει το κατάλληλο *DDM pragma directive* όπως φαίνεται στο Σχήμα 6.18 μετά από τις δηλώσεις των μεταβλητών και πριν από οποιοδήποτε άλλο κώδικα της ***main*** συνάρτησης.

```
int main()
{
      int var1;
      double var2;
      #pragma ddm startprogram
      var1 = 0;
```

(α)

```
int var1;
double var2;

int main()
{
      var1 = 0;
```

(β)

Σχήμα 6.18: (α): Χρήση *DDM pragma directive startprogram*. (β): Αντίστοιχος τελικός κώδικας

Στο σημείο αυτό ο προγραμματιστής μπορεί επίσης να δηλώσει ποιες από τις μεταβλητές του προγράμματος θα πρέπει να γίνουν ιδιωτικές στον τελικό κώδικα, μέσω χρήσης κατάλληλων *DDM pragma directives*, όπως φαίνεται στο Σχήμα 6.19. Αυτό είναι απαραίτητο για τις μεταβλητές στις οποίες υπάρχει πιθανότητα να γίνεται ταυτόχρονη πρόσβαση από πολλαπλούς TFlux kernels, λόγω του ότι τα *POSIX Threads* δουλεύουν σε *shared-memory* περιβάλλον. Ο Preprocessor, κατά την ανάλυση των συγκεκριμένων *DDM directives* για ιδιωτικοποίηση, αποθηκεύει τις πληροφορίες για τις μεταβλητές σε μία κατάλληλη δομή δεδομένων. *Εδώ* σημειώνεται ότι εξακολουθεί να είναι απαραίτητη η δήλωση από τον

προγραμματιστή των μεταβλητών αυτών, η ιδιωτικοποίησή τους είναι απλά μια ένδειξη στον Preprocessor ότι θα πρέπει αυτές οι μεταβλητές να επαναδηλωθούν στον κώδικα του κάθε TFlux kernel για να είναι ιδιωτικές. Επίσης, στις περιπτώσεις TFlux Loops, δεν είναι απαραίτητη η ιδιωτικοποίηση της μεταβλητής ελέγχου (control variable) του εξωτερικού (παράλληλου) βρόγχου διότι αυτή η μεταβλητή θα αντικατασταθεί, όπως θα δούμε αργότερα, στο σώμα του βρόγχου με το Iteration Id του κάθε DDThr που εκτελεί εκείνο το TFlux loop, αλλά για σκοπούς καθαρότητας, είναι καλό να γίνεται.

```
#pragma ddm private var int  var1
#pragma ddm private var myType var2
```

Σχήμα 6.19: Παράδειγμα καθορισμού μεταβλητών για ιδιωτικοποίηση

6.4.2 Main συνάρτηση

Στη συνέχεια, δηλώνεται η ***main*** συνάρτηση του τελικού κώδικα από τον Preprocessor και αντιγράφεται σε αυτή ο υπόλοιπος κώδικας της αρχικής ***main*** συνάρτησης μέχρι τη συνάντηση του πρώτου *DDM Block* που έχει δηλώσει ο προγραμματιστής με το αντίστοιχο *DDM pragma directive*. Σε αυτό το σημείο, ο Preprocessor τοποθετεί στον τελικό κώδικα κατάλληλες εντολές και κλήσεις συναρτήσεων για τη δημιουργία και αρχικοποίηση του ***TSUGroup*** και των δομών του, κώδικα για τη δημιουργία των TFlux Kernels (Σχήμα 6.20), κώδικα για το βρόγχο επανάληψης που θα εκτελεί η ***main*** συνάρτηση προσομοιώνοντας τη λειτουργία του ***TSU*** μέσω κλήσης στην κατάλληλη συνάρτηση του ***TSUGroup*** (Σχήμα 6.21) και τέλος κλείνει τη ***main*** συνάρτηση του τελικού κώδικα για να τελειώσει το πρόγραμμα. Αυτό θα γίνει μόνο όταν όλοι οι TFlux Kernels ολοκληρώσουν την εκτέλεσή τους και τερματίσουν, μειώνοντας έτσι τον αριθμό των *live TFlux Kernels* στο μηδέν.

```
pthread_mutex_init(&__ddmVar_mutexNumberOfLiveKernels, NULL);
initializeTSUGroup(<PARAMETERS>);

// C R E A T E   T H E   D D M   K E R N E L S              //
for(__ddmVar_cv01 = 0; __ddmVar_cv01 < KERNEL_NUM; __ddmVar_cv01++)
{
      pthread_mutex_lock(&__ddmVar_mutexNumberOfLiveKernels);
      __ddmVar_numberOfLiveKernels++;
      pthread_mutex_unlock(&__ddmVar_mutexNumberOfLiveKernels);
      __ddmVar_ownTSU[__ddmVar_cv01] = __ddmVar_cv01;
      pthread_create( &__ddmVar_threads[__ddmVar_cv01], NULL, (void
            *)&ddm_code, (void *)&(__ddmVar_ownTSU[__ddmVar_cv01]));
}
```

Σχήμα 6.20: Τελικός κώδικας δημιουργημένος από τον Preprocessor για αρχικοποίηση του ***TSUGroup*** και τη δημιουργία των *TFlux Kernels*

```
            while (__ddmVar_numberOfLiveKernels)
                  emulateTSU(__ddmVar_cv01);

            pthread_exit(NULL);
      }
```

Σχήμα 6.21: Τελικός κώδικας για προσομοίωση της λειτουργίας του ***TSU*** από τη ***main*** συνάρτηση και τέλος προγράμματος

6.4.3 Συνάρτηση των TFlux Kernels (ddm_code)

Μετά το τέλος της ***main*** συνάρτησης, όπως είδαμε πιο πάνω, ο Preprocessor δημιουργεί στον τελικό κώδικα εξόδου τον κώδικα της συνάρτησης ***ddm_code()***, όπου εκτελούνται οι TFlux kernels και ο κώδικας της εφαρμογής (δηλαδή τα DDThrs). Εδώ, αρχικά μπαίνει ο κώδικας για τη δήλωση των ιδιωτικών μεταβλητών, οι οποίες ανιχνεύθηκαν από τον Preprocessor όπως είδαμε στο προηγούμενο μέρος. Η ιδιωτικοποίησή τους γίνεται με τον τρόπο που φαίνεται στο Σχήμα 6.22, ούτως ώστε να μεταφερθεί και οποιαδήποτε τυχόν αρχικοποίηση είχαν από την εκτέλεση της ***main***.

```
void *ddm_code(short int *__ddmVar_ownTSU)
{
      int temp01 = sum;
      int sum = temp01;
```

Σχήμα 6.22: Απόσπασμα τελικού κώδικα που παράγεται από τον Preprocessor για τη συνάρτηση ***ddm_code*** και παράδειγμα ιδιωτικοποίησης μίας μεταβλητής

Ακολούθως, ο Preprocessor, στο πρώτο πέρασμά του, αναλύει τα *DDM pragma directives* για τον καθορισμό όλων των *DDM Blocks* και *DDThrs* (απλά, Loop και Reduction) του προγράμματος από τον προγραμματιστή (Σχήμα 6.23). Με βάση αυτά κτίζει το Γράφο Συγχρονισμού του προγράμματος αποθηκεύοντας τις πληροφορίες στις δομές που παρουσιάστηκαν σε προηγούμενο μέρος. Ταυτόχρονα, για κάθε *DDM Block*, ο Preprocessor δημιουργεί ακόμα δύο *DDThrs*, το *Inlet* και το *Outlet*. Έτσι, κατά το δεύτερο πέρασμα του Preprocessor από τον αρχικό κώδικα και σε αυτό το σημείο τοποθετεί κώδικα στο τελικό αρχείο εξόδου για την εγγραφή στο κάθε ***TSU*** του *Inlet DDthr* του πρώτου *DDM Block* του, ούτως ώστε να είναι το πρώτο *DDThr* που θα εκτελεστεί από τον κάθε TFlux kernel και να μπορεί να συνεχίσει η εκτέλεση με τα υπόλοιπα DDThrs.

```
#pragma ddm block 1
      #pragma ddm thread 1 kernel 1
            CODE FOR DDTHR 1
      #pragma end thread

      #pragma ddm for thread 2 depends(1)
            for (…)
      #pragma ddm endfor

      #pragma ddm for thread 3 reduction sum + double totalSum depends(2)
#pragma ddm endblock
```

Σχήμα 6.23: Παράδειγμα καθορισμού *DDM Block* και *DDThrs* διαφόρων τύπων από τον προγραμματιστή χρησιμοποιώντας κατάλληλα *DDM pragma directives*

Στη συνέχεια, για κάθε DDM Block που συναντάται στο δεύτερο πέρασμα, ο Preprocessor δημιουργεί ένα κομμάτι κώδικα, το ***THREAD_SELECT***, όπου ο κάθε TFlux kernel "ζητά" από το ***TSU*** του την ταυτότητα του επόμενου DDThr για εκτέλεση και, μέσω ενός *switch statement* (Σχήμα 6.24) μεταβαίνει στον κώδικα του συγκεκριμένου DDThr.

```
THREAD_SELECT_1:
      __ddmVar_currentThread = currentlyExecutedThread(*__ddmVar_ownTSU);
      __ddmVar_threadUnderExecution_THID= FIND_THREAD_ID(__ddmVar_currentThread);
      switch(__ddmVar_threadUnderExecution_THID)
      {
            //INLET THREADS
            case    11:
                  goto INLET_THREAD_11_BLOCK_1;
            //OUTLET THREADS
            case    12:
                  goto OUTLET_THREAD_12_BLOCK_1;
            //EXECUTION THREADS
            case     2:
                  goto EXECUTION_THREAD_2_BLOCK_1;
            //LOOP THREADS
            case     1:
                  goto LOOP_THREAD_1_BLOCK_1;
            //REDUCTION THREADS
            case     3:
                  goto REDUCTION_THREAD_3_BLOCK_1;
            //If no thread is ready for execution try again
            case     0:
                  goto THREAD_SELECT_1;
      }
```

Σχήμα 6.24: Παράδειγμα κώδικα ***THREAD_SELECT*** για το *DDM Block **1*** μίας εφαρμογής

Ο κώδικας κάθε DDThr του συγκεκριμένου *DDM Block* τοποθετείται και αυτός στο τελικό αρχείο κώδικα εξόδου από τον Preprocessor με το ανάλογο *label* μπροστά ανάλογα με τον τύπο του κάθε DDThr, δηλαδή από το *DDM pragma directive* που επιλέγει ο προγραμματιστής για το κάθε DDThr, όπως φαίνεται στο Σχήμα 6.25 για ένα Inlet DDThr, Σχήμα 6.26 για ένα απλό DDThr, Σχήμα 6.27 για ένα Loop DDThr, Σχήμα 6.28 για ένα Reduction DDThr και Σχήμα 6.29 για ένα Oulet DDThr. Ο κώδικας του Inlet DDThr προστίθεται στον τελικό κώδικα στην αρχή κάθε νέου DDM Block, ενώ ο κώδικας του Outlet DDThr στο τέλος κάθε DDM Block. Τέλος, το Outlet DDThr του τελευταίου DDM Block του προγράμματος τερματίζει τον συγκεκριμένο TFlux kernel μέσω της κλήσης ***pthread_exit()***.

```
BLOCK_1:
      INLET_THREAD_11_BLOCK_1:
            LOAD ALL DDTHRS OF BLOCK 1
            threadCompletedExecution();
            goto THREAD_SELECT_1;
```

Σχήμα 6.25: Παράδειγμα τελικού κώδικα για ένα *Inlet DDThr*, όπως παράγεται από τον Preprocessor

```
#pragma ddm thread 1 kernel 1
      CODE FOR DDTHR 1
#pragma ddm endthread
```

(α)

```
EXECUTION_THREAD_1_BLOCK_1:
      CODE FOR DDTHR 1
      threadCompletedExecution();
      goto THREAD_SELECT_1;
```

(β)

Σχήμα 6.26: (α): Αρχικός κώδικας για ένα απλό DDThr, όπως γράφεται από τον προγραμματιστή. (β): Τελικός κώδικας για το ίδιο DDThr, όπως παράγεται από τον Preprocessor

```
#pragma ddm for thread 2
      for (i = 0; i < MAX; i++)
            A[i] = i;
#pragma ddm endfor
```

(α)

```
LOOP_THREAD_2_BLOCK_1:
      A[ITERATION_ID] = ITERATION_ID;
      threadCompletedExecution();
      goto THREAD_SELECT_1;
```

(β)

Σχήμα 6.27: (α): Αρχικός κώδικας για ένα Loop DDThr. (β): Τελικός κώδικας για το ίδιο Loop DDThr

```
#pragma ddm for thread 3 reduction sum + double totalSum
      for (i = 0; i < MAX; i++)
            sum += i;
#pragma ddm endfor
```

(α)

```
LOOP_THREAD_3_BLOCK_1:
      localsum += ITERATION_ID;
      threadCompletedExecution();
      goto THREAD_SELECT_1;

REDUCTION_THREAD_5_BLOCK_1:
      kernel2_totalSum = localsum;
      threadCompletedExecution();
      goto THREAD_SELECT_1;

REDUCTION_THREAD_4_BLOCK_1:
      totalSum = localsum;
      totalSum += kernel2_totalSum;
      threadCompletedExecution();
      goto THREAD_SELECT_1;
```

(β)

Σχήμα 6.28: (α): Αρχικός κώδικας για ένα Reduction Loop DDThr. (β): Τελικός κώδικας για το ίδιο Reduction Loop DDThr

```
OUTLET_THREAD_26_BLOCK_1:
      threadCompletedExecution();
      //This is the Outlet Thread of the last block
      //DDM Kernel completed
      pthread_mutex_lock(&__ddmVar_mutexNumberOfLiveKernels);
      __ddmVar_numberOfLiveKernels--;
      pthread_mutex_unlock(&__ddmVar_mutexNumberOfLiveKernels);
      pthread_exit(NULL);
```

Σχήμα 6.29: Παράδειγμα τελικού κώδικα για ένα *Outlet DDThr* του τελευταίου *DDM Block* του προγράμματος, όπως παράγεται από τον Preprocessor

Κεφάλαιο 7 Benchmarks

7.1 Στόχος

Για την αξιολόγηση του συστήματος TFlux, χρησιμοποιήσαμε ένα σύνολο από Benchmarks, τα οποία ονομάσαμε “Σουίτα Αξιολόγησης του TFlux”. Για να εξασφαλίσουμε ότι αυτή η σουίτα θα είναι αρκετή για μια καλή και γενική αξιολόγηση της υλοποίησής μας, πρέπει να περιέχει μία μεγάλη γκάμα από benchmarks με όσο το δυνατό διαφορετικά χαρακτηριστικά. Σε αυτό το σημείο θα παρουσιάσουμε τις απαιτήσεις που ορίσαμε για τη σουίτα αξιολόγησης, καθώς επίσης και τα χαρακτηριστικά στα οποία οι εφαρμογές μας θα πρέπει να διαφέρουν.

Καταρχάς, όλα τα benchmarks της σουίτας αξιολόγησης του TFlux πρέπει να έχουν πρακτικό ενδιαφέρον σε πραγματικά προβλήματα με μη-τετριμμένες απαιτήσεις εκτέλεσης. Για να καλύψουμε αυτό το στόχο, όλες οι εφαρμογές μας είτε προέρχονται από ευρέως αναγνωρισμένες σουίτες από benchmarks, είτε είναι κομμάτια κώδικα που αντιστοιχούν σε ρουτίνες οι οποίες χρησιμοποιούνται ευρέως σε επιστημονικές εφαρμογές.

Το κατεξοχήν στοιχείο το οποίο ξεχωρίζει το TFlux από άλλες πλατφόρμες παράλληλης εκτέλεσης είναι ο τρόπος με τον οποίο χειρίζεται τις εξαρτήσεις μεταξύ κομματιών κώδικα. Έτσι, το κύριο συστατικό το οποίο πρέπει να διαφοροποιεί τα benchmarks είναι η πολυπλοκότητα των γράφων συγχρονισμού τους.

Επίσης, ένα πολύ σημαντικό θέμα σχεδιασμού των παράλληλων αρχιτεκτονικών είναι η μείωση των overheads κατά την εκτέλεση λόγω της παραλληλοποίησης. Για να μπορέσουν να μελετηθούν αυτά τα overheads για το TFlux, η σουίτα αξιολόγησης περιέχει εφαρμογές με σημαντικές διαφορές στο granularity των παράλληλων τμημάτων τους. Επιπρόσθετα, κάθε εφαρμογή μελετήθηκε με τρία διαφορετικά μεγέθη προβλήματος εισόδου.

Η τελευταία παράμετρος της σουίτας αξιολόγησης αφορά την επεκτασιμότητα (scalability) του συστήματος. Συγκεκριμένα, είναι επιθυμητό να επιβεβαιώνουμε την καλή απόδοση που επιτυγχάνει το σύστημα, όχι μόνο με μικρό αριθμό επεξεργαστών αλλά και για συστήματα με μεγάλο αριθμό πυρήνων. Για αυτό το σκοπό, η σουίτα περιέχει εφαρμογές οι οποίες μπορούν να επεκταθούν και για τέτοια συστήματα.

Όσον αφορά τα συγκεκριμένα συστατικά του TFlux, ο αριθμός των δυναμικών DDThrs για τα διάφορα benchmarks κινείται από 7 μέχρι και 2^{21}, ενώ ο αριθμός των στατικών DDThrs από 1 μέχρι και περισσότερα από 200. Οι εφαρμογές χρησιμοποιούν όλα τα βασικά συστατικά εκτέλεσης που αναλύσαμε σε προηγούμενα κεφάλαια, δηλαδή απλά DDThrs, TFlux Loops και TFlux Loops με Reduction.

Τέλος, για το θέμα της μεταφοράς των benchmarks στο TFlux, όπως είδαμε σε προηγούμενο κεφάλαιο, ο Preprocessor για την υλοποίηση TFlux της πλατφόρμας TFlux επεκτάθηκε με βάση και χρησιμοποιώντας τα ίδια *DDM pragma directives* με τα οποία είχε προηγουμένως αναπτυχθεί για την υλοποίηση TFlux-Hard. Αυτό καθιστά εύκολη την παραλληλοποίηση μίας εφαρμογής για την πλατφόρμα TFlux-Soft. Επίσης, με αυτό τον τρόπο μπορούμε να μεταφέρουμε benchmarks τα οποία

είναι ήδη γραμμένα για την υλοποίηση TFlux-Hard σε TFlux-Soft, χωρίς καμία αλλαγή του αλγορίθμου ή του κώδικα.

7.2 Tradeoffs από την παραλληλοποίηση

Σε αυτό το μέρος θα παρουσιάσουμε τα κύρια tradeoffs που σχετίζονται με τη μεταφορά μιας εφαρμογής στο TFlux και αφορούν το granularity των DDThrs. Αυτό το granularity αφορά τόσο τον στατικό κώδικα των DDThrs όσο και τον αριθμό των δυναμικών εντολών που εκτελούν, που φυσικά μπορεί να σχετίζονται.

Όταν ένα πρόγραμμα θα μεταφερθεί στην πλατφόρμα TFlux, ο προγραμματιστής πρέπει να εντοπίσει τα DDThrs, μία διαδικασία η οποία επιτελείται στο επίπεδο του πηγαίου κώδικα. Το πλήθος των στατικών εντολών που θα έχει ένα DDThr καθορίζει το "στατικό granularity" του, ενώ το πλήθος των δυναμικών εντολών το "δυναμικό granularity" του.

Το στατικό granularity των DDThrs έχει περισσότερο να κάνει με το ποσοστό παραλληλισμού που θα εκτίθεται στο hardware. Συγκεκριμένα, με στατικά πιο fine-grain DDThrs είναι δυνατό να εκφράσουμε περισσότερο παραλληλισμό σε μια εφαρμογή, δίνοντας της έτσι τη δυνατότητα εκμετάλλευσης περισσότερης ταυτοχρονίας. Καθώς τα DDThrs γίνονται πιο fine-grained, περισσότερα DDThrs θα απαιτούνται για να καλύψουν τον κώδικα της εφαρμογής. Τα μειονεκτήματα της αύξησης του αριθμού των στατικών DDThrs είναι δύο. Το πρώτο είναι ότι θα απαιτείται περισσότερος χρόνος για τη φόρτωση των μεταδεδομένων όλων αυτών των DDThrs στο TSU για εκτέλεση. Το δεύτερο είναι ότι, αν ο αριθμός των στατικών DDThrs αυξηθεί πέρα από τη χωρητικότητα του TSU, θα είναι αναγκαίο να χωρίσουμε το πρόγραμμα σε πολλαπλά TFlux Blocks. Κάθε επιπλέον TFlux Block επηρεάζει αρνητικά την απόδοση, λόγω του ότι μειώνει τον παραλληλισμό του προγράμματός (αφού κάθε νέο block ξεκινά την εκτέλεσή του μόνο αφού ολοκληρωθεί η εκτέλεση όλων των DDThrs του προηγούμενου block για όλους τους TFlux kernels). Επιπρόσθετα, κάθε block προκαλεί και το overhead ενός επιπλέον Inlet και Outlet DDThr για κάθε TFlux kernel.

Καθώς το στατικό μέγεθος των DDThrs αυξάνεται, αναμένεται ότι ο αριθμός των εντολών που ένα DDThr εκτελεί (δηλαδή το δυναμικό μέγεθος του DDThr) επίσης θα αυξάνεται. Μια άλλη τεχνική για την αύξηση του δυναμικού μεγέθους ενός DDThr είναι το loop unrolling. Αυτή η δεύτερη τεχνική χρησιμοποιείται μόνο στα TFlux loops και, αν και δεν επηρεάζει τον αριθμό των στατικών DDThs, εντούτοις μειώνει τον αριθμό των δυναμικών τους εκτελέσεων. Ως αποτέλεσμα, αυξάνοντας το δυναμικό μέγεθος των DDThrs μειώνει το overhead του χειρισμού, δηλαδή το πλήθος των λειτουργιών Thread Completion και Find Ready Thread που επιτελούνται.

Από την άλλη όμως, καθώς τα DDThrs γίνονται δυναμικά μεγαλύτερα σε μέγεθος, αυτό οδηγεί σε άλλες επιπλοκές. Αρχικά, αν το δυναμικό μέγεθος των DDThrs αυξηθεί πέρα από ένα σημείο (το οποίο είναι διαφορετικό για κάθε εφαρμογή), ο παραλληλισμός θα μειωθεί σε τέτοιο επίπεδο ώστε δε θα υπάρχουν αρκετά παράλληλα κομμάτια κώδικα για να κρατούν απασχολημένες όλες τις υπολογιστικές μονάδες. Αυτό ισχύει ακόμη περισσότερο για συστήματα με μεγάλο αριθμό πυρήνων. Επίσης, μεγάλα κομμάτια κώδικα είναι πιθανόν να έχουν διαφορετικό χρόνο εκτέλεσης, ειδικά αν περιλαμβάνουν μεγάλο αριθμό προσβάσεων στη μνήμη, η καθυστέρηση των οποίων συχνά είναι απρόβλεπτη. Η διαφορά στο χρόνο εκτέλεσης πιθανό να δημιουργήσει προβλήματα ισορροπίας φόρτου εργασίας (load balancing issues) και μείωση στην απόδοση. Τέλος, η χρησιμοποίηση υπερβολικού unroll στα TFlux loops μπορεί να οδηγήσει σε φτωχή απόδοση του Instruction Cache λόγω code explosion.

Από τα πιο πάνω οδηγούμαστε στο συμπέρασμα ότι τα DDThrs πρέπει να είναι αρκετά μεγάλα για να υπερκαλύπτουν το overhead της παραλληλοποίησης, αλλά ταυτόχρονα αρκετά μικρά για να εκθέτουν όσο το δυνατόν περισσότερο από το διαθέσιμο παραλληλισμό στο υλικό, να αποφεύγουν το υπερβολικό πλήθος από DDM Blocks και να μειώνουν τις ανισορροπίες στο φόρτο εργασίας τους.

Για να μελετήσουμε αυτό το tradeoff, όλα τα benchmarks στη σουίτα αξιολόγησης του TFlux μελετήθηκαν για τρία διαφορετικά μεγέθη προβλήματος εισόδου, μικρό, μεσαίο και μεγάλο. Ταυτόχρονα, η μελέτη περιλαμβάνει εκδόσεις των benchmarks

όπου τα TFlux loops τους έχουν γίνει unroll με διαφορετικούς παράγοντες, όπου αυτό ήταν δυνατό. Η χρήση διαφορετικών παραμέτρων (όπως ***loop unroll***) για το κάθε benchmark επίσης μας επιτρέπει να μελετήσουμε τα διάφορα tradeoffs για την κάθε υλοποίηση της πλατφόρμας TFlux (TFlux-Soft και TFlux-Hard)

7.3 Η σουίτα αξιολόγησης του TFlux

Αυτό το μέρος παρουσιάζει τα benchmarks τα οποία επιλέχθηκαν για την αξιολόγηση του TFlux-Soft στη σειρά η οποία αντιστοιχεί στην πολυπλοκότητα των γράφων συγχρονισμού τους. Για τα benchmarks τα οποία δεν προέρχονται από κάποια σουίτα, περιλαμβάνονται επίσης πληροφορίες για τη λειτουργία και τον παραλληλισμό του αλγορίθμου.

7.3.1 Πολλαπλασιασμός Πινάκων (MMULT)

Το MMULT πολλαπλασιάζει δύο δυσδιάστατους πίνακες (C = A · B). Η *βασική εργασία (basic task)* αυτού του αλγορίθμου είναι ο υπολογισμός ενός στοιχείου του τελικού πίνακα (C) και μία τέτοια *βασική εργασία* επιτελείται για κάθε στοιχείο του τελικού πίνακα. Για να βρούμε την τιμή του στοιχείου $C_{i,j}$ η βασική εργασία υπολογίζει το εσωτερικό γινόμενο της γραμμής ***i*** του πίνακα ***A*** με τη στήλη ***j*** του πίνακα ***B***. Ως αποτέλεσμα, κάθε εκτέλεση της βασικής εργασίας διαβάζει μια γραμμή από τον πίνακα ***A***, μια στήλη από τον πίνακα ***B*** και αλλάζει το στοιχείο $C_{i,j}$ το οποίο, για κάθε εκτέλεση μίας διαφορετικής βασικής εργασίας, βρίσκεται σε διαφορετική θέση μνήμης (Σχήμα 7.1). Έτσι συμπεραίνουμε ότι όλες οι εκτελέσεις της *βασικής εργασίας* είναι ανεξάρτητες η μία από την άλλη και μπορούν να τρέξουν παράλληλα και έτσι οδηγούμαστε στο χαρακτηρισμό του MMULT σαν μία “embarrassingly parallel” εφαρμογή.

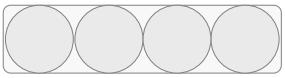
Το MMULT δεν ανήκει σε κάποια συγκεκριμένη σουίτα από benchmarks. Παρόλα αυτά, η πράξη που εκτελεί είναι πολύ διαδεδομένη σε ένα μεγάλο αριθμό από επιστημονικές εφαρμογές και έτσι δικαιολογείται η ύπαρξή του στη σουίτα αξιολόγησης του TFlux.

$$\begin{bmatrix} C_{0,0} & C_{0,1} & C_{0,2} & \dots & C_{0,k} \\ C_{1,0} & C_{1,1} & C_{1,2} & \dots & C_{1,k} \\ C_{2,0} & C_{2,1} & C_{2,2} & \dots & C_{2,k} \\ \dots & & & & \\ C_{n,0} & C_{n,1} & C_{n,2} & \dots & C_{n,k} \end{bmatrix} = \begin{bmatrix} A_{0,0} & A_{0,1} & A_{0,2} & \dots & A_{0,n} \\ A_{1,0} & A_{1,1} & A_{1,2} & \dots & A_{1,n} \\ A_{2,0} & A_{2,1} & A_{2,2} & \dots & A_{2,n} \\ \dots & & & & \\ A_{m,0} & A_{m,1} & A_{m,2} & \dots & A_{m,n} \end{bmatrix} \mathbf{X} \begin{bmatrix} B_{0,0} & B_{0,1} & B_{0,2} & \dots & B_{0,k} \\ B_{1,0} & B_{1,1} & B_{1,2} & \dots & B_{1,k} \\ B_{2,0} & B_{2,1} & B_{2,2} & \dots & B_{2,k} \\ \dots & & & & \\ B_{n,0} & B_{n,1} & B_{n,2} & \dots & B_{n,k} \end{bmatrix}$$

Σχήμα 7.1: Λειτουργία του αλγορίθμου πολλαπλασιασμού δύο δυσδιάστατων πινάκων

7.3.1.1 Διαδικασία μεταφοράς στην πλατφόρμα TFlux

Η παράλληλη έκδοση για TFlux του MMULT αποτελείται από ένα TFlux loop (Σχήμα 7.2-(α)), κάθε DDThr του οποίου υπολογίζει ένα στοιχείο του τελικού πίνακα εξόδου. Όπως φαίνεται στο Σχήμα 7.2-(β), ο γράφος συγχρονισμού του MMULT μπορεί να εκφραστεί μόνο με ένα *DDM loop pragma directive*.



(α)

```
#pragma ddm for thread 1
      for (i = 0; i < ARRAY_SIZE; i++)
      {
            BASIC OPERATION
      }
#pragma ddm endfor
```

(β)

Σχήμα 7.2: (α) Ο γράφος συγχρονισμού του MMULT για 4 kernels. (β) Παραλληλοποιημένο MMULT χρησιμοποιώντας *DDM pragma directives*

7.3.1.2 Δυναμική συμπεριφορά

Όπως εξηγήσαμε πιο πάνω, κάθε DDThr του παράλληλου βρόγχου επανάληψης του προγράμματός υπολογίζει ένα στοιχείο του τελικού πίνακα εξόδου και έτσι, ο αριθμός των DDThrs είναι ίσος με το πλήθος των στοιχείων εκείνου του πίνακα. Για το MMULT, το μέγεθος του προβλήματος είναι το μέγεθος των τετραγωνικών πινάκων που πολλαπλασιάζονται.

Το MMULT χρησιμοποιεί ένα μεγάλο όγκο δεδομένων. Στον πρώτο πίνακα A, η πρόσβαση στα στοιχεία του γίνεται κατά γραμμή ενώ, στο δεύτερο πίνακα B, γίνεται κατά στήλη. Ως αποτέλεσμα, οι προσβάσεις στα στοιχεία του δευτέρου πίνακα αναμένεται να οδηγούν σε πολύ μεγαλύτερο miss rate στο data cache. Η χρησιμοποίηση τεχνικών *blocking* πιθανόν να έλυνε το πιο πάνω πρόβλημα, ομώς

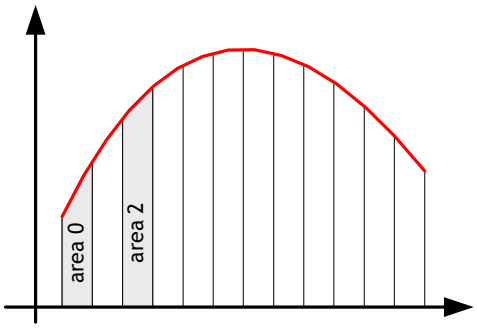
αυτό δεν έγινε για να υπάρχει στη σουίτα αξιολόγησης και μία αντιπροσωπευτική εφαρμογή με αυτά τα χαρακτηριστικά δυναμικής συμπεριφοράς.

7.3.2 Τραπεζοειδής μέθοδος ολοκλήρωσης (TRAPEZ)

Το benchmark TRAPEZ υπολογίζει το ολοκλήρωμα μιας συνάρτησης για ένα δεδομένο διάστημα. Για να γίνει αυτός ο υπολογισμός παράλληλα, το διάστημα ολοκλήρωσης χωρίζεται σε υποδιαστήματα και η τραπεζοειδής μέθοδος εφαρμόζεται στο κάθε υποδιάστημα (Σχήμα 7.3). Αυτή η πράξη αποτελεί τη βασική πράξη του παράλληλου αλγορίθμου. Στο τέλος όλων των βασικών πράξεων, το άθροισμα όλων των μερικών αποτελεσμάτων είναι η τιμή του ολοκληρώματος σύμφωνα με τη συγκεκριμένη μέθοδο.

Κάθε κλήση της βασικής πράξης μπορεί να εκτελεστεί παράλληλα, δεδομένου ότι κάθε επεξεργαστής κρατά μία ιδιωτική μεταβλητή για το άθροισμα των μερικών αποτελεσμάτων ολοκλήρωσης που υπολόγισε ο συγκεκριμένος επεξεργαστής. Το τελικό βήμα του παράλληλου αλγορίθμου είναι ο υπολογισμός του αθροίσματος όλων των ιδιωτικών μεταβλητών, δηλαδή μια πράξη μείωσης (reduction operation) ούτως ώστε να υπολογίσουμε τη συνολική τιμή του ολοκληρώματος.

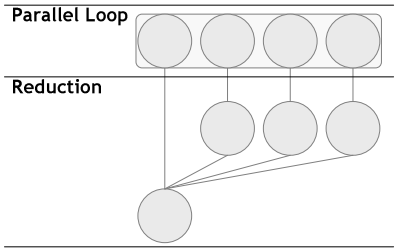
Παρόμοια με το MMULT που είδαμε προηγουμένως, ούτε το TRAPEZ ανήκει σε κάποια συγκεκριμένη σουίτα από benchmarks. Ο υπολογισμός που εκτελεί όμως χρησιμοποιείται ευρέως σε πολλές και σημαντικές επιστημονικές εφαρμογές.



Σχήμα 7.3: Λειτουργία της παράλληλης έκδοσης της Τραπεζοειδούς Μεθόδου για την ολοκλήρωση

7.3.2.1 Διαδικασία μεταφοράς στην πλατφόρμα TFlux

Η παράλληλη έκδοση του TRAPEZ για TFlux αποτελείται από ένα TFlux Reduction Loop (Σχήμα 7.4-(α)), στο οποίο κάθε DDThr εκτελεί τη βασική πράξη ολοκλήρωσης σε διαφορετικό υποδιάστημα. Όπως φαίνεται στο Σχήμα 7.4-(β), ο γράφος συγχρονισμού του TRAPEZ μπορεί να περιγραφεί με ένα μόνο *DDM Reduction Loop pragma directive*. Αξίζει να σημειωθεί ότι με τη χρήση αυτού του *pragma directive*, η δήλωση των ιδιωτικών μεταβλητών για τα μερικά αθροίσματα κάθε TFlux kernel, όπως και ο κώδικας που πραγματοποιεί το reduction στο τέλος γίνονται αυτόματα από τον Preprocessor.



(α)

```
#pragma ddm for thread 1 reduction area + totalArea
      for (i = 0; i < NUMBER_OF_SUBINTERVALS; i++)
      {
            area += BASIC OPERATION
      }
#pragma ddm endfor
```

(β)

Σχήμα 7.4: (α) Ο Γράφος Συγχρονισμού του TRAPEZ για 4 TFlux kernels. (β) Παράλληλο TRAPEZ χρησιμοποιώντας *DDM pragma directives*

7.3.2.2 Δυναμική συμπεριφορά

Κάθε DDThr του TFlux loop του παράλληλου προγράμματος εκτελεί τη βασική πράξη πάνω σε ένα υποδιάστημα και έτσι, το πλήθος των DDThrs του προγράμματος θα είναι ίσος με τον αριθμό των υποδιαστημάτων συν το πλήθος των DDThrs που εκτελούν την πράξη Reduction στο τέλος, δηλαδή ένα DDThr για κάθε TFlux kernel. Το μέγεθος εισόδου για το TRAPEZ ορίζει αυτό το πλήθος των υποδιαστημάτων.

Όσον αφορά την πρόσβαση στα δεδομένα, το TRAPEZ χρησιμοποιεί μόνο κάποιες μεταβλητές και έτσι, αυτή η εφαρμογή αναμένεται να έχει ελάχιστο αριθμό από misses κατά τις προσβάσεις στη μνήμη, ακόμα και για caches με πολύ μικρό μέγεθος.

7.3.3 Susan – Smoothing (SUSAN)

Το benchmark Susan είναι ένα πρόγραμμα επεξεργασίας εικόνας. Το πρόγραμμα Susan – Smoothing που χρησιμοποιήσαμε σα μέρος της σουίτας προγραμμάτων για αξιολόγηση του TFlux είναι ένα κομμάτι του Susan το οποίο ασχολείται με την λείανση (smoothing) μιας εικόνας που δίνεται σαν είσοδος στο πρόγραμμα. Η εικόνα αναπαρίσταται σαν ένας δυσδιάστατος πίνακας από pixels. Ο αρχικός αλγόριθμος του Susan – Smoothing χωρίζεται σε δύο βήματα. Αρχικά γίνεται η δημιουργία και αρχικοποίηση μίας μάσκα, η οποία είναι ένας δυσδιάστατος πίνακας. Στη συνέχεια, πραγματοποιείται η λείανση κάθε pixel της εικόνας, χρησιμοποιώντας κάθε φορά τη μάσκα που δημιουργήθηκε και αρχικοποιήθηκε στο πρώτο βήμα. Στο πρώτο βήμα, η πράξη αρχικοποίησης ενός στοιχείου του πίνακα της μάσκας αποτελεί τη βασική πράξη του πρώτου βήματος, ενώ, για το δεύτερο και πιο ουσιαστικό, αλλά και πιο χρονοβόρο, βήμα η πράξη λείανσης που πραγματοποιείται σε ένα pixel της εικόνας είναι η δική του βασική πράξη.

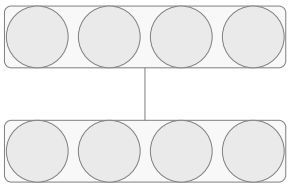
Και στα δύο βήματα του αλγορίθμου, κάθε κλήση των δύο βασικών πράξεων μπορεί να εκτελεστεί παράλληλα, δεδομένου ότι η κάθε επανάληψη της κάθε βασικής πράξης αλλοιώνει μόνο το “δικό” της στοιχείο στο δυσδιάστατό της πίνακα, δηλαδή τον πίνακα μάσκας για την πρώτη βασική πράξη και τον πίνακα της εικόνας για τη δεύτερη. Επίσης όλοι οι βοηθητικοί υπολογισμοί που πραγματοποιούνται στις δύο βασικές πράξεις είναι εντελώς ανεξάρτητοι μεταξύ των διαφορετικών επαναλήψεων.

Τέλος, σε αντίθεση με τα δύο προηγούμενα benchmarks που αναλύσαμε, τα οποία δεν προέρχονταν από κάποια συγκεκριμένη σουίτα, το Susan ανήκει στη σουίτα benchmarks “MiBench” [7].

7.3.3.1 Διαδικασία μεταφοράς στην πλατφόρμα TFlux

Η τελική έκδοση του παράλληλου αλγορίθμου του SUSAN για την πλατφόρμα TFlux αποτελείται από δύο TFlux Loops (Σχήμα 7.5-(α)). Στο πρώτο TFlux Loop, κάθε DDThr εκτελεί τη βασική πράξη αρχικοποίησης της μάσκας, ενώ στο δεύτερο TFlux

Loop, κάθε DDThr εκτελεί τη βασική πράξη λείανσης του pixel της εικόνας στο οποίο αντιστοιχεί. Όπως βλέπουμε από το Σχήμα 7.5-(β), ο γράφος συγχρονισμού του SUSAN μπορεί να εκφραστεί με τη χρήση μόνο δύο *DDM Loop pragma directives*.



```
#pragma ddm for thread 1
      for (i = 0; i < MASK_SIZE; i++)
      {
            MASK INITIALIZATION BASIC OPERATION
      }
#pragma ddm endfor

#pragma ddm for thread 2 depends (1)
      for (i = 0; i < IMAGE_SIZE; i++)
      {
            IMAGE SMOOTHING BASIC OPERATION
      }
#pragma ddm endfor
```

(α) (β)

Σχήμα 7.5: (α) Ο Γράφος Συγχρονισμού του SUSAN για 4 TFlux kernels. (β) Παράλληλο SUSAN χρησιμοποιώντας *DDM pragma directives*

Επίσης, στον αρχικό κώδικα της εφαρμογής, κάθε βρόγχος επανάληψης των δύο βημάτων του TFlux κώδικα αποτελείται από δύο φωλιασμένους βρόγχους, ο εξωτερικός για τη συντεταγμένη Y του πίνακα (είτε της μάσκας είτε της εικόνας), δηλαδή τη γραμμή, και ο εσωτερικός για τη συντεταγμένη X, δηλαδή τη στήλη. Στην παράλληλη έκδοση του αλγορίθμου αυτοί οι δύο φωλιασμένοι βρόγχοι συγχωνεύθηκαν σε ένα βρόγχο (Σχήμα 7.6), ούτως ώστε να αυξηθεί το πλήθος των DDThrs που μπορούν να εκτελεστούν παράλληλα ανά πάσα στιγμή και έτσι να εξαχθεί περισσότερος παραλληλισμός από την εφαρμογή.

```
for (i = 0; i < MASK_SIZE_Y; i++)
{
      for (j = 0; j < MASK_SIZE_X; j++)
      {
            MASK INITIALIZATION BASIC OPERATION
      }
}

for (i = 0; i < IMAGE_SIZE_Y; i++)
{
      for (j = 0; j < IMAGE_SIZE_X; j++)
      {
            IMAGE SMOOTHING BASIC OPERATION
      }
}
```

(α)

```
for (i = 0; i < (MASK_SIZE_Y x MASK_SIZE_X); i++)
{
      MASK INITIALIZATION BASIC OPERATION
}

for (i = 0; i < (IMAGE_SIZE_Y x IMAGE_SIZE_X); i++)
{
      IMAGE SMOOTHING BASIC OPERATION
}
```

(β)

Σχήμα 7.6: (α) Αρχικός κώδικας δύο βημάτων του SUSAN. (β) Η τελική έκδοση του κώδικα

Τέλος, ήταν απαραίτητη η ιδιωτικοποίηση κάποιων μεταβλητών. Συγκεκριμένα, κάποιες μεταβλητές οι οποίες χρησιμοποιούνταν μέσα στις δύο βασικές πράξεις για την προσωρινή αποθήκευση ενδιάμεσων υπολογισμών, καθώς επίσης και δύο μεταβλητές ελέγχου για δύο εσωτερικούς φωλιασμένους βρόγχους που υπάρχουν μέσα στον κώδικα της βασικής πράξης του δευτέρου βήματος, μετατράπηκαν από κοινές σε ιδιωτικές, με τη χρήση ενός *DDM pragma directive* για κάθε μεταβλητή ούτως ώστε κάθε TFlux kernel να έχει το δικό του αντίγραφο και να μην έχουμε προβλήματα συγχρονισμού.

7.3.3.2 Δυναμική συμπεριφορά

Κατά την εκτέλεση του πρώτου βήματος του αλγορίθμου, δηλαδή της αρχικοποίησης της μάσκας με βάση την οποία θα γίνει αργότερα η λείανση της εικόνας, αφού το κάθε DDThr αναλαμβάνει να αρχικοποίησει ένα στοιχείο του πίνακα της μάσκας, το πλήθος των DDThrs είναι ίσο με το μέγεθος της μάσκας, το μέγεθος της οποίας είναι ανεξάρτητο του μεγέθους της εικόνας.

Όσον αφορά το δεύτερο και πιο ουσιαστικό μέρος του αλγορίθμου, δηλαδή την πράξη λείανσης της εικόνας, αφού το κάθε DDThr αναλαμβάνει να επιτελέσει τη βασική πράξη λείανσης πάνω σε ένα pixel της εικόνας, ο αριθμός των DDThrs του προγράμματος σε αυτό το βήμα είναι ίσος με το πλήθος των pixels της εικόνας, άρα και με το μέγεθος της εικόνας εισόδου.

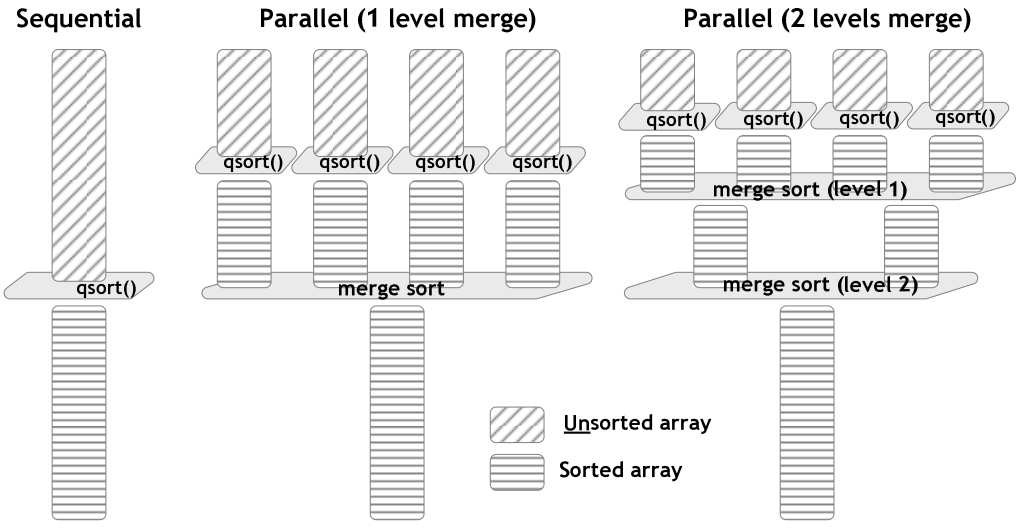
Τέλος, όσο για την πρόσβαση δεδομένων, το SUSAN, ιδιαίτερα κατά την εκτέλεση του δεύτερου βήματος του αλγορίθμου, ενεργεί πάνω σε ένα μεγάλο όγκο δεδομένων και χρησιμοποιεί και αρκετές προσωρινές μεταβλητές και πίνακες μεταβλητών για την επιτέλεση των βοηθητικών πράξεων για κάθε βασική πράξη. Παρόλα αυτά, οι προσβάσεις στους πίνακες γίνονται με κανονικό τρόπο, πράγμα που ευνοεί την καλή συμπεριφορά του cache. Το μόνο σημείο που πιθανόν να προκαλεί κάποια περισσότερα data misses από τα άλλα είναι το ότι στο τέλος της κάθε βασικής πράξης αρχικοποίησης ή λείανσης, το κάθε DDThr γράφει το αποτέλεσμα των πράξεων του σε έναν κοινό πίνακα (πίνακας μάσκας και εικόνα εξόδου αντίστοιχα). Αν και το κάθε DDThr γράφει σε διαφορετικό στοιχείο του

πίνακα και έτσι είναι ανεξάρτητες οι πράξεις, αυτό θα προκαλεί συνεχή μεταφορά μέρους ή μερών των πινάκων από το cache ενός επεξεργαστή σε αυτό άλλου επεξεργαστή.

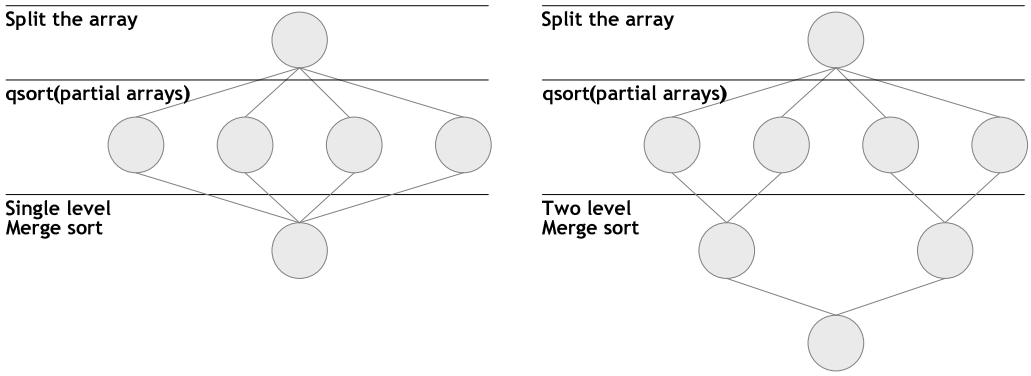
7.3.4 Ταξινόμηση χρησιμοποιώντας τον αλγόριθμο qSort (SORT)

Η ιδέα πίσω από το SORT benchmark, το οποίο προέρχεται από τη σουίτα MiBench [7], είναι να ταξινομήσει έναν πίνακα αριθμών χρησιμοποιώντας τη συνάρτηση ταξινόμησης ως ένα "μαύρο κουτί". Η αρχική έκδοση αυτού του benchmark ταξινομεί ολόκληρο τον πίνακα χρησιμοποιώντας τη συνάρτηση qsort του συστήματος, αλλά θα μπορούσε να χρησιμοποιηθεί οποιαδήποτε άλλη συνάρτηση ή υλοποίηση αλγορίθμου ταξινόμησης.

Για τη διενέργεια αυτής της λειτουργίας παράλληλα, ο αλγόριθμος τροποποιήθηκε σε μία διαδικασία δύο βημάτων (Σχήμα 7.7). Πρώτα ο πίνακας εισόδου χωρίζεται σε αριθμό κομματιών ίσο με τον αριθμό των επεξεργαστών. Κάθε επεξεργαστής ταξινομεί τον υποπίνακα που του ανατέθηκε χρησιμοποιώντας τη συνάρτηση συστήματος qsort, η οποία δεν έχει τροποποιηθεί με κανένα τρόπο. Στο δεύτερο βήμα, οι ταξινομημένοι υποπίνακες συγχωνεύονται στον τελικό ταξινομημένο πίνακα. Σημειώνουμε επίσης ότι η πράξη συγχώνευσης μπορεί να υλοποιηθεί σε πολλαπλά επίπεδα (Σχήμα 7.8).



Σχήμα 7.7: Η λειτουργία της παράλληλης έκδοσης ταξινόμησης



Σχήμα 7.8: Ο Γράφος Συγχρονισμού του SORT για τέσσερεις TFlux Kernels

Η παραλληλοποίηση της ταξινόμησης με αυτό τον τρόπο είναι επωφελής λόγω των χαρακτηριστικών χρόνου των δύο διαδικασιών, ταξινομώντας έναν πίνακα μεγέθους ***n*** χρησιμοποιώντας qsort ($time_{qsort(n)}$) και συγχωνεύοντας ***k*** πλήθος πινάκων με ***n/k*** στοιχεία ($time^{k}_{merge(n/k)}$). Συγκεκριμένα, το $time_{qsort(n)}$ είναι μικρότερο από το $time_{qsort(n)} + time^{k}_{merge(n/k)}$.

7.3.4.1 Διαδικασία μεταφοράς στην πλατφόρμα TFlux

Όπως μπορούμε να δούμε από το γράφο συγχρονισμού του benchmark (Σχήμα 7.8) αποτελείται μόνο από ανεξάρτητα DDThrs χωρισμένα σε δύο ομάδες. Τα DDThrs της πρώτης ομάδας εκτελούν τη διαδικασία ταξινόμησης πάνω στον υποπίνακα που τους ανατίθεται, ενώ αυτά της δεύτερης ομάδας εκτελούν τη συγχώνευση των υποπινάκων στον τελικό ταξινομημένο πίνακα.

Καθώς όλα τα DDThrs της πρώτης ομάδας εκτελούν την ίδια πράξη, αυτό μπορεί να εκφραστεί με τη χρήση του *DDM thread kernel all pragma directive*, το οποίο δηλώνει ένα DDThr το οποίο θα εκτελεστεί από όλους του TFlux Kernels (Σχήμα 7.9). Καθώς το κάθε DDThr λειτουργεί πάνω σε διαφορετικό υποσύνολο στοιχείων του αρχικού πίνακα, κατάλληλοι υπολογισμοί συμπεριλήφθηκαν στο DDThr ούτως ώστε να μπορεί να εντοπίσει τον υποπίνακα που του αντιστοιχεί. Η πράξη συγχώνευσης εκτελείται από τα DDThrs της δεύτερης ομάδας. Κάθε τέτοιο DDThr ορίζεται να εξαρτάται από τα αντίστοιχα DDThrs της πρώτης ομάδας.

Για τον παραλληλισμό της εφαρμογής ήταν απαραίτητο να συμπεριλάβουμε διαδικασίες για το διαχωρισμό του αρχικού πίνακα καθώς και για τη συγχώνευση. Επιπρόσθετα, ένας προσωρινός πίνακας χρειάζεται να χρησιμοποιηθεί για την αποθήκευση των ταξινομημένων υποπινάκων μετά την εκτέλεση της ταξινόμησης από τα DDThrs της πρώτης ομάδας.

```
#pragma ddm thread 1 kernel all
      myL = findLowerBoundaryMyArray();
      myU = findUpperBoundaryMyArray();
      qSort(initialArray, myL, myU-myL);
#pragma ddm endthread

#pragma ddm thread 2 depends (1)
      outputArray = merge();
#pragma ddm endthread
```

(α)

```
#pragma ddm thread 1 kernel all
      myL = findLowerBoundaryMyArray();
      myU = findUpperBoundaryMyArray();
      qSort(initialArray, myL, myU-myL);
#pragma ddm endthread

#pragma ddm thread 2 depends (1/0,...)
      tempArray1 = merge(...);
#pragma ddm endthread

#pragma ddm thread 3 depends (…,1/k)
      tempArray2 = merge(…);
#pragma ddm endthread

#pragma ddm thread 4 depends (2,3)
      outputArray = merge();
#pragma ddm endthread
```

(β)

Σχήμα 7.9: Ο κώδικας του παραλληλοποιημένου SORT benchmark με τη χρήση DDThrs. (α) Με ένα επίπεδο συγχώνευσης. (β) Με δύο επίπεδα συγχώνευσης

7.3.4.2 Δυναμική συμπεριφορά

Ο αριθμός των DDThrs σε αυτό το benchmark είναι μικρός και ανεξάρτητος από το μέγεθος του προβλήματος. Συγκεκριμένα, η φάση ταξινόμησης έχει τόσα DDThrs όσος και ο αριθμός των TFlux Kernels, ενώ για τη φάση της συγχώνευσης, ο αριθμός των DDThrs είναι ακόμα μικρότερος (ένα DDThr για συγχώνευση ενός επιπέδου και ***1 + k/2*** για συγχώνευση δύο επιπέδων όπου ***k*** είναι ο αριθμός των TFlux kernels). Για το SORT benchmark, το μέγεθος του προβλήματος αντιπροσωπεύει το πλήθος των στοιχείων του πίνακα εισόδου που θα ταξινομηθούν.

Οι προσβάσεις μνήμης των DDThrs που εκτελούν την ταξινόμηση καθορίζονται από την υλοποίηση της συνάρτησης συστήματος qsort. Όσο για τα DDThrs που εκτελούν

τη συγχώνευση, πραγματοποιούν προσβάσεις σε διαφορετικούς υποπίνακες αλλά σε διαδοχικά στοιχεία και έτσι εκμεταλλεύονται ένα μεγάλο βαθμό data locality.

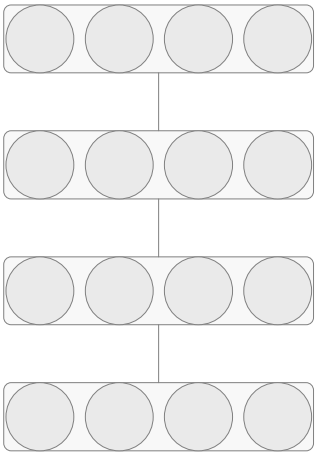
7.3.5 Αλγόριθμος Runge-Kutta (RK)

Το Runge-Kutta (RK) είναι μια αριθμητική μέθοδος για επίλυση προβλημάτων αρχικής τιμής σε κανονικές διαφορικές εξισώσεις. Η υλοποίηση που χρησιμοποιήσαμε σαν benchmark μας είναι ο τύπος Runge-Kutta τέταρτης τάξης, η οποία είναι μια διαδικασία τεσσάρων βημάτων. Κάθε βήμα αποτελείται από ένα παράλληλο βρόγχο επανάληψης ο οποίος υπολογίζει μια τιμή απαραίτητη για όλες τις επαναλήψεις των επόμενων βρόγχων. Έτσι, για να μπορέσει μια επανάληψη ενός βρόγχου να ξεκινήσει την εκτέλεσή της θα πρέπει όλες οι επαναλήψεις του προηγούμενου βρόγχου να ολοκληρώσουν τη δική τους εκτέλεση.

Παρόμοια με το MMULT και το TRAPEZ, έτσι και το RK δεν ανήκει σε κάποια συγκεκριμένη σουίτα benchmarks, αλλά είναι μια πολύ διαδεδομένη και χρησιμοποιημένη μέθοδος σε πολλές επιστημονικές εφαρμογές.

7.3.5.1 Διαδικασία μεταφοράς στην πλατφόρμα TFlux

Η παραλληλοποίηση του RK ήταν σχετικά εύκολη αφού η βάση του αποτελείται από τέσσερεις βρόγχους επανάληψης, οι οποίοι παραλληλοποιήθηκαν με τη χρήση *DDM for pragma directives* (Σχήμα 7.10-(β)). Όπως εξηγήσαμε προηγουμένως, λόγω της φύσης του αλγορίθμου, το barrier μεταξύ των παραλληλοποιημένων βρόγχων δεν μπορεί να εξαληφθεί.



```
#pragma ddm for thread 1
      for (i = 0; i < n; i++)
      {
            STEP_1
      }
#pragma ddm endfor

#pragma ddm for thread 2 depends (1)
      for (i = 0; i < n; i++)
      {
            STEP_2
      }
#pragma ddm endfor

#pragma ddm for thread 3 depends (2)
      for (i = 0; i < n; i++)
      {
            STEP_3
      }
#pragma ddm endfor

#pragma ddm for thread 4 depends (3)
      for (i = 0; i < n; i++)
      {
            STEP_4
      }
#pragma ddm endfor
```

(α) (β)

Σχήμα 7.10: (α) Ο Γράφος Συγχρονισμού του benchmark RK. (β) Η παράλληλη έκδοση του κώδικα των 4 βρόγχων του benchmark RK

7.3.5.2 Δυναμική συμπεριφορά

Ο αλγόριθμος ενεργεί πάνω σε ένα σύνολο από πίνακες, ένα δυσδιάστατο ο οποίος χρησιμοποιείται για να περιγράφει την είσοδο, ένα μονοδιάστατο για την έξοδο και ακόμα ένα μονοδιάστατο για κάθε ένα βήμα της μεθόδου RK. Έτσι, το benchmark προκαλεί από μέτριο μέχρι υψηλό φόρτο εργασίας στο cache.

Το πλήθος των DDThrs του RK εξαρτάται από το μέγεθος της εισόδου του προβλήματος. Πιο συγκεκριμένα, κάθε ένας από τους τέσσερεις βρόγχους επανάληψης έχει ένα DDThr για κάθε στοιχείο εισόδου, το οποίο καταλήγει σε

συνολικό πλήθος από DDThrs ίσο με τέσσερεις φορές το μέγεθος του προβλήματος. Για το RK, το μέγεθος του προβλήματος αφορά το πλήθος των στοιχείων σε κάθε διάσταση του πίνακα ο οποίος περιγράφει τη συνάρτηση εισόδου.

7.3.6 Fast Fourier Transformation (FFT)

Το benchmark FFT υπολογίζει το Διακριτό Μετασχηματισμό Fourier (Discrete Fourier Transformation – DFT), ο οποίος χρησιμοποιείται ευρέως σε διάφορα πεδία όπως στη Ψηφιακή Επεξεργασία Σημάτων και Ηλεκτρομαγνητισμό. Η υλοποίηση του FFT την οποία χρησιμοποιήσαμε σα μέρος της σουίτας των benchmarks μας περιέχει τον υπολογιστικό πυρήνα μιας τρισδιάστατης φασματικής μεθόδου βασισμένη στο FFT. Αυτό το benchmark προέρχεται από τη σουίτα NAS Parallel Benchmarks [2] και λειτουργεί πάνω σε έναν τρισδιάστατο πίνακα από μιγαδικούς αριθμούς. Η έκδοση στην οποία βασίστηκε το TFlux FFT ήταν μια υλοποίηση του NAS FFT [9] με OpenMP.

Παρόλο που μεταφέρθηκε ολόκληρο το benchmark στην πλατφόρμα TFlux, η ανάλυσή μας επικεντρώθηκε πάνω στη συνάρτηση *fft*, η οποία είναι το κομμάτι της εφαρμογής που υλοποιεί τους χρήσιμους υπολογισμούς του αλγορίθμου. Αυτή η συνάρτηση καλεί τρεις άλλες συναρτήσεις (*cffts1*, *cffts2* και *cffts3*), καθεμιά από τις οποίες εφαρμόζει το μονοδιάστατο FFT μετασχηματισμό σε μια από τις τρεις διαστάσεις του τρισδιάστατου πίνακα εισόδου. Κάθε μια από αυτές τις συναρτήσεις καλεί περαιτέρω τη συνάρτηση *cfftz*, η οποία εκτελεί μια παραποίηση του Stockham FFT.

7.3.6.1 Διαδικασία μεταφοράς στην πλατφόρμα TFlux

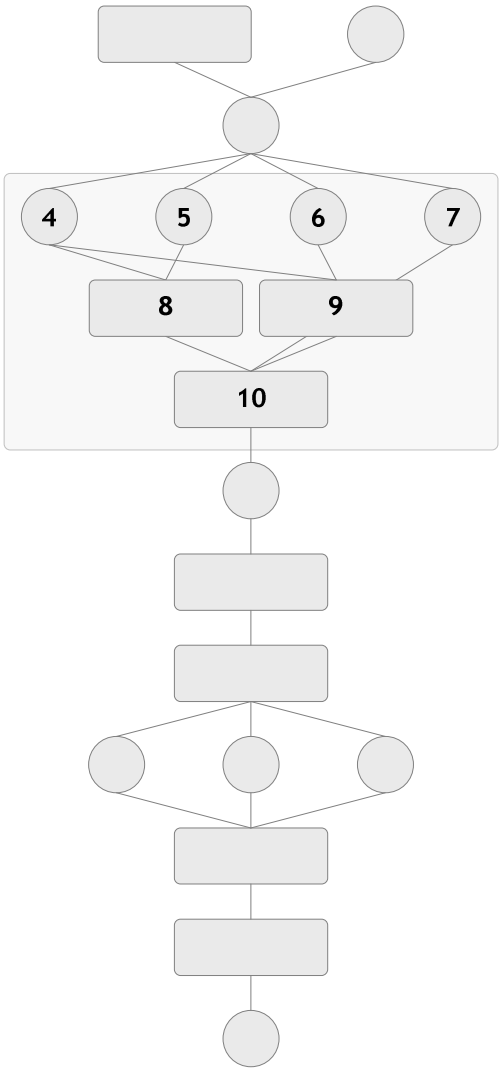
Για τη μεταφορά του FFT, οι συναρτήσεις *cffts1*, *cffts2* και *cffts3* έγιναν inlined στο σώμα της καλούσας συνάρτησης. Το μεγαλύτερο μέρος του χρόνου εκτέλεσης αυτών των συναρτήσεων αναλώνεται στην κλήση της συνάρτησης *cfftz*. Η συνάρτηση *cfftz* καλείται πολλαπλές φορές κατά την εκτέλεση των συναρτήσεων που αναφέρθηκαν προηγουμένως και κάθε τέτοια κλήση είναι ανεξάρτητο. Ως επακόλουθο, η παραλληλοποίηση έγινε σε αυτό το σημείο, δηλαδή στην κλήση της συνάρτησης *cfftz*.

Στο Σχήμα 7.11-(α) φαίνεται ο Γράφος Συγχρονισμού για ολόκληρο το benchmark FFT και η συνάρτηση *fft* παρουσιάζεται σκιασμένη. Τα DDThrs 4, 5, 6 και 7 πραγματοποιούν κάποιες αρχικοποιήσεις καθώς και κάποιους αρχικούς υπολογισμούς, οι οποίοι είναι αναγκαίοι για την εκτέλεση των TFlux loops που ακολουθούν. Όπως φαίνεται από το Γράφο Συγχρονισμού, οι κλήσεις στη συνάρτηση *cfftz* που αντιστοιχούν στις συναρτήσεις *cffts1* (TFlux loop 8) και *cffts2* (TFlux loop 9) μπορούν να εκτελεστούν παράλληλα, ενώ οι κλήσεις στη συνάρτηση *cfftz* από τη συνάρτηση *cffts3* (TFlux loop 10) εξαρτούνται από αυτές των *cffts1* και *cffts2*. Ο κώδικας DDM ο οποίος αντιστοιχεί στη συνάρτηση *fft* φαίνεται στο Σχήμα 7.11-(β).

7.3.6.2 Δυναμική συμπεριφορά

Το FFT λειτουργεί πάνω σε τρισδιάστατους πίνακας μιγαδικών αριθμών, το οποίο προκαλεί αυξημένο αριθμό προσβάσεων στη μνήμη. Όμως, το pattern των προσβάσεων είναι κανονικό και αυτό οδηγεί στην αναμονή μέτριων μέχρι και χαμηλών miss rates. Το benchmark έχει ένα μέτριο πλήθος από DDThrs, το οποίο εξαρτάται από το μέγεθος του προβλήματος. Πιο συγκεκριμένα, εκτελεί έντεκα απλά DDThrs και οκτώ TFlux Loops, με το πλήθος των επαναλήψεων για αυτά τα loops να βρίσκεται στην τάξη του O(input^2). Όσο αφορά τη συνάρτηση fft, αυτή αποτελείται από τέσσερα DDThrs και τρία TFlux Loops. Το barrier μεταξύ των loops σε αυτή τη συνάρτηση δεν μπορεί να αφαιρεθεί λόγω της φύσεως του αλγορίθμου.

Τέλος, αφού οι επαναλήψεις των TFlux Loops αυτής της εφαρμογής αποτελούνται από κώδικα με πολλές εντολές, το overhead της παραλληλοποίησης αναμένεται ότι, σε μεγάλο βαθμό, θα εξαλειφθεί.



(α)

```
#pragma ddm thread 4 kernel 1 depends (3)
      Initializations
#pragma ddm endthread

#pragma ddm thread 5 kernel 2 depends (3)
      Initializations
#pragma ddm endthread

#pragma ddm thread 6 kernel 3 depends (3)
      Initializations
#pragma ddm endthread

#pragma ddm thread 7 kernel 4 depends (3)
      Initializations
#pragma ddm endthread

#pragma ddm for thread 8 depends (4,5)
      cffts1
#pragma ddm endfor

#pragma ddm for thread 9 depends (4,6,7)
      cffts1
#pragma ddm endfor

#pragma ddm for thread 10 depends (8,9)
      cffts1
#pragma ddm endfor
```

(β)

Σχήμα 7.11: (α) Ο Γράφος Συγχρονισμού του benchmark FFT. Η συνάρτηση FFT παρουσιάζεται σκιασμένη. (β) Το μέρος του κώδικα του FFT το οποίο αντιστοιχεί στη συνάρτηση *fft*, η οποία παραλληλοποιήθηκε χρησιμοποιώντας *DDM pragma directives*.

7.3.7 Σύνοψη

Τα χαρακτηριστικά των benchmarks της σουίτας αξιολόγησης της πλατφόρμας TFlux παρουσιάζονται στον Πίνακα 7.1.

Πίνακας 7.1: Σύνοψη χαρακτηριστικών και συμπεριφοράς των benchmarks

Characteristic	MMULT	TRAPEZ	SORT	SUSAN	RK	FFT
Synchronization Graph (# Static DDThrs)						
TFlux Loops	1	1	-	2	4	3
Single DDThrs	-	K	K + 2	-	-	4
Porting to TFlux						
# Pragma Directives	1	1	2	2	4	7
Origination of the benchmark						
Benchmark Suite	kernel	kernel	MiBench	MiBench	kernel	NAS
Input Sizes						
Small	64x64	2^{17}	10K	256x288	1024	32x32x32
Medium	128x128	2^{19}	20K	512x576	2048	64x64x64
Large	256x256	2^{21}	50K	1024x576	4096	128x128x128
Number of executed DDThrs (# Dynamic DDThrs)						
Small	2^{12}	2^{17} + K	K + 1	256x288 + 225	2^{12}	3 X 2^{5} + 4
Medium	2^{14}	2^{19} + K	K + 1	512x576 + 225	2^{13}	3 X 2^{6} + 4
Large	2^{16}	2^{21} + K	K + 1	1024x576 + 225	2^{14}	3 X 2^{7} + 4
Dynamic Data Behavior						
Data Usage	High	High	Low	High	Medium	High
Access Pattern	Irregular	-	Regular	Regular	Regular	Regular

Κεφάλαιο 8 Αξιολόγηση

8.1 Επεξήγηση διαδικασίας αξιολόγησης

Η αξιολόγηση της πλατφόρμας έγινε σε δύο μέρη. Στο πρώτο μέρος η πλατφόρμα αξιολογήθηκε σε ένα σύστημα με τη χρήση ενός *full-system* Simulator και στο δεύτερο μέρος σε ένα πραγματικό σύστημα. Η χρήση του Simulator ήταν απαραίτητη για τη μελέτη των δυνατοτήτων της πλατφόρμας σε επεξεργαστή με περισσότερους πυρήνες από αυτούς που υπάρχουν στην αγορά. Επιπλέον, ο Simulator παρείχε ένα πιο ελεγχόμενο περιβάλλον αφού η κατάσταση του Λειτουργικού Συστήματος ήταν σε όλα τα πειράματα η ίδια. Τέλος, τα πειράματα σε πραγματικό σύστημα έγιναν σαν απόδειξη ότι όντως η πλατφόρμα μπορεί να χρησιμοποιηθεί άμεσα στους επεξεργαστές της αγοράς, χωρίς καμία αλλαγή ούτε στο υλικό αλλά ούτε και στο Λειτουργικό Σύστημα.

Για την αξιολόγηση και με τις δύο προαναφερθείσες μεθόδους χρησιμοποιήθηκαν όλα τα benchmarks που παρουσιάστηκαν στο προηγούμενο κεφάλαιο (Κεφάλαιο 7 – Υποκεφάλαιο 7.3). Κάθε benchmark εξετάστηκε με τρία διαφορετικά μεγέθη εισόδου, και επιπλέον, όπου ήταν δυνατό, τα loops των προγραμμάτων έγιναν unroll από 1 μέχρι 64 φορές, ώστε να μελετηθεί το runtime overhead που προστίθεται από τη χρήση της πλατφόρμας.

Σε όλα τα αποτελέσματα που παρουσιάζονται σαν Speedup, δηλαδή πόσες φορές πιο γρήγορη είναι η παράλληλη εκτέλεση από την αντίστοιχη σειριακή, η βελτίωση υπολογίστηκε με τη σύγκριση του καλύτερου χρόνου εκτέλεσης του σειριακού προγράμματος με τον καλύτερο χρόνο εκτέλεσης του παράλληλου, ανάμεσα στα διάφορα unrolls, όπου αυτά υπήρχαν.

8.2 Πειραματική διαρρύθμιση

8.2.1 Πειράματα με χρήση *full-system* Simulator

Η αξιολόγηση έγινε με τη χρήση του *Simics full-system Simulator* [21]. Η προσομοιωμένη μηχανή είναι βασισμένη σε ένα σύστημα με 28 πυρήνες Sparc64 επεξεργαστή, 32KB IL1 – 32KB DL1 – 2MB DL2 cache και Linux kernel 2.6.13-1.1603sp13smp. Όλες οι μετρήσεις έγιναν με τη χρήση hardware performance counters για περισσότερη ακρίβεια.

Συνοπτικά τα πειράματα που έγιναν για αξιολόγηση της επίδοσης με τη χρήση της πλατφόρμας φαίνονται στον πίνακα 8.1. Κάθε benchmark εξετάστηκε με τρία διαφορετικά μεγέθη εισόδου ούτως ώστε να εξεταστεί το πόσο επηρεάζει την επίδοση το overhead της πλατφόρμας.

Πίνακας 8.1: Σύνολο πειραμάτων στον *Simics* για αξιολόγηση επίδοσης

Benchmark	**Input Size**			**Unroll**
	Small	**Medium**	**Large**	
MMULT	64 X 64	128 X 128	256 X 256	1 – 64
TRAPEZ	2^{17}	2^{19}	2^{21}	1 – 64
QSORT	10K	20K	50K	-
SUSAN	256 X 288	512 X 576	1024 X 576	1 – 64
RK	1024	2048	4096	1 – 64
FFT	32 X 32 X 32	64 X 64 X 64	128 X 128 X 128	-

8.2.2 Πειράματα σε πραγματικό σύστημα

Για την αξιολόγηση σε πραγματικό σύστημα χρησιμοποιήθηκε ένας server IBM x3650 με 2 επεξεργαστές Xeon E5320 Core2 QuadCore επεξεργαστές. Κάθε core έχει 32KB, 64B line size, 8-way set associative L1 data και instruction cache. Κάθε QuadCore έχει 4MB, 64B line size, 16-way set associative unified L2 cache. Επιπλέον, η μηχανή είναι εξοπλισμένη με 18GB DDR2 RAM με συχνότητα 333MHz.

Στο πραγματικό σύστημα έγινε αξιολόγηση της βελτίωσης του χρόνου εκτέλεσης. Οι μετρήσεις πραγματοποιήθηκαν με τη χρήση της συνάρτησης συστήματος *gettimeofday()*. Το σύνολο των πειραμάτων φαίνεται στον πίνακα 8.2.

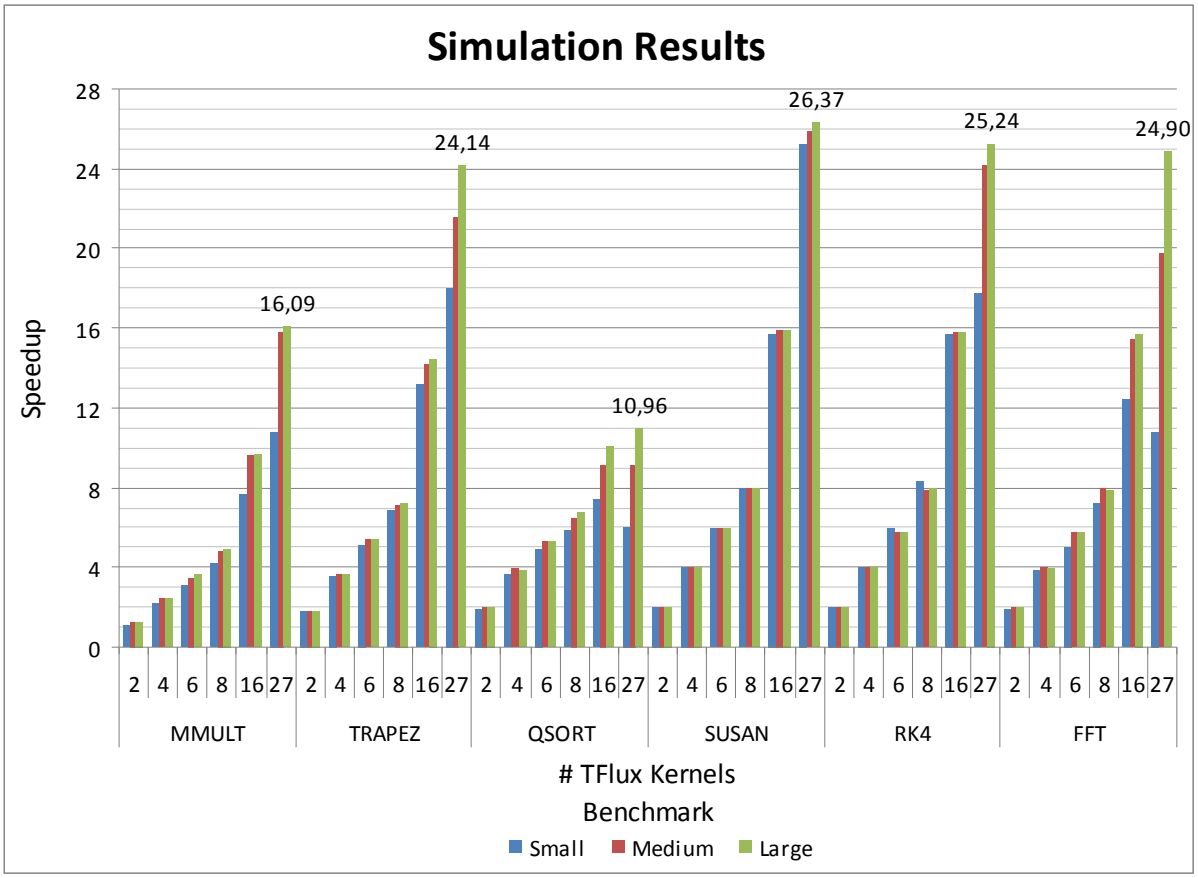
Πίνακας 8.2: Σύνολο πειραμάτων σε πραγματικό σύστημα για αξιολόγηση επίδοσης

Benchmark	**Input Size**			**Unroll**
	Small	**Medium**	**Large**	
MMULT	64 X 64	128 X 128	256 X 256	1 – 64
TRAPEZ	2^{17}	2^{19}	2^{21}	1 – 64
QSORT	10K	20K	50K	-
SUSAN	256 X 288	512 X 576	1024 X 576	1 – 64
RK	1024	2048	4096	1 – 64
FFT	32 X 32 X 32	64 X 64 X 64	128 X 128 X 128	-

8.3 Παρουσίαση και ανάλυση αποτελεσμάτων

8.3.1 Αποτελέσματα πειραμάτων στον Simics full-system Simulator

Τα αποτελέσματα των πειραμάτων στον *Simics full-system* Simulator φαίνονται στο Σχήμα 8.1. Για όλα τα benchmarks παρατηρείται ότι το speedup αυξάνεται ανάλογα με τον αριθμό των TFlux Kernels που χρησιμοποιούνται και ανάλογα με το μέγεθος του προβλήματος.



Σχήμα 8.1: Αποτελέσματα πειραμάτων στον Simulator. Το speedup υπολογίστηκε με βάση το χρόνο εκτέλεσης του σειριακού προγράμματος

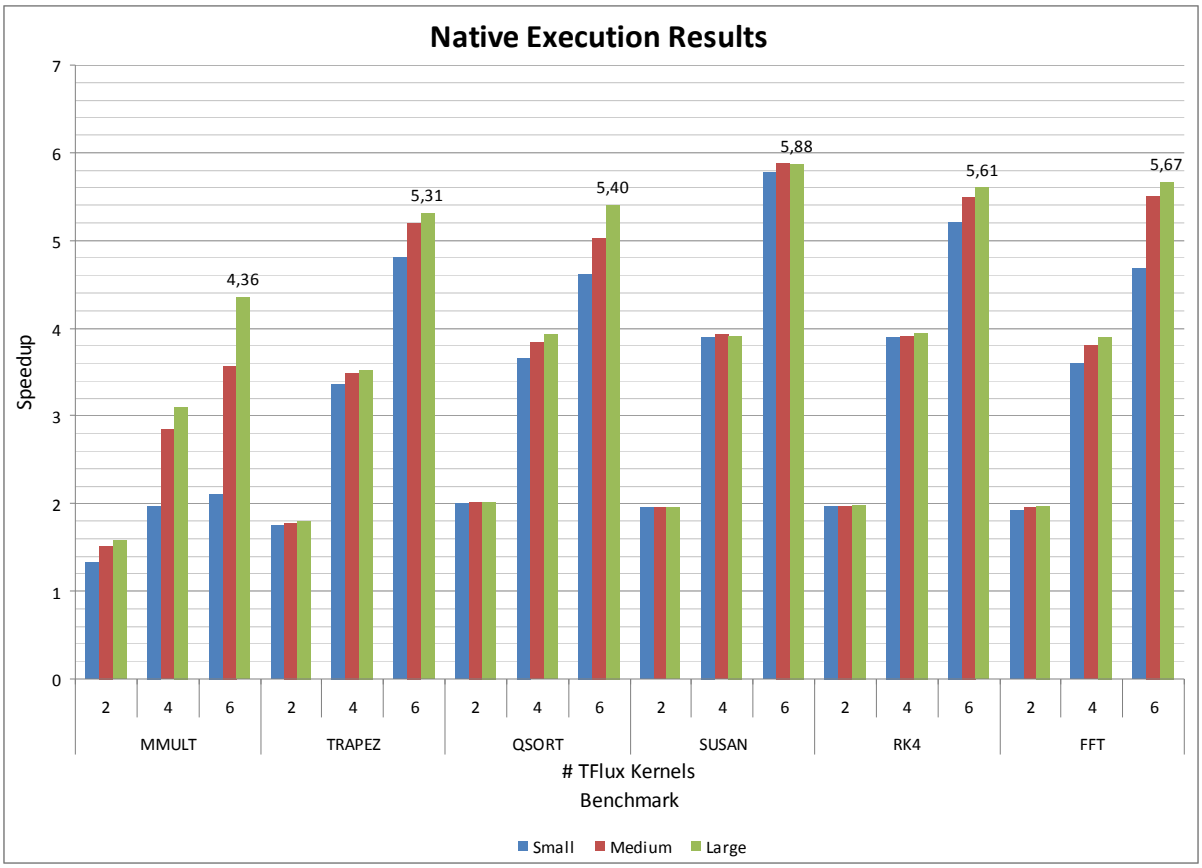
Στα benchmarks *TRAPEZ*, *SUSAN*, *RK* και *FFT* παρατηρείται γραμμική αύξηση στο speedup, πετυχαίνοντας το ιδανικό speedup ανάλογα με τον αριθμό των TFlux Kernels για το μεγάλο μέγεθος εισόδου. Με μικρότερα μεγέθη εισόδου, φαίνεται ότι το overhead της πλατφόρμας δεν αφομοιώνεται από τον παράλληλο υπολογισμό. Συγκεκριμένα, καθώς αυξάνονται οι TFlux Kernels μειώνεται το μέγεθος του υπολογισμού που ανατίθεται στον καθένα, συνεπώς γίνεται πιο έντονο το overhead της πλατφόρμας.

Όσον αφορά τα benchmark *MMULT* και *QSORT* δεν επιτυγχάνεται το ιδανικό speedup. Το speedup αυξάνεται ανάλογα με τον αριθμό των TFlux Kernels και το μέγεθος εισόδου. Στην περίπτωση του *QSORT*, ο λόγος για τον οποίο δεν επιτυγχάνεται το ιδανικό speedup είναι το γεγονός ότι ο παραλληλισμός είναι περιορισμένος από τον σταθερό αριθμό DDThr που εκτελούνται και είναι εντελώς ανεξάρτητος από το μέγεθος εισόδου. Στην περίπτωση του *MMULT* η όχι και τόσο

καλή επίδοση οφείλεται στο μέγεθος της μνήμης που χρησιμοποιείται από την εφαρμογή. Η μνήμη αυξάνεται ανάλογα με το μέγεθος εισόδου, αφού αυξάνεται το μέγεθος των πινάκων των οποίων θα υπολογιστεί το εξωτερικό γινόμενο. Λόγω του μεγάλου όγκου δεδομένων και της μη κανονικής σειράς πρόσβασης, γίνονται πολλά misses στην προσπάθεια ανάγνωσης των δεδομένων από την L1 cache. Επιπλέον, λόγω της υλοποίησης της πλατφόρμας σε επίπεδο λογισμικού μόνο, το cache επιβαρύνεται ακόμα περισσότερο από τα δεδομένα του TSU που χρησιμοποιεί ο εκάστοτε TFlux Kernel.

8.3.2 Αποτελέσματα πειραμάτων σε πραγματικό σύστημα

Τα αποτελέσματα των πειραμάτων σε πραγματικό σύστημα φαίνονται στο Σχήμα 8.2. Για όλα τα benchmarks παρατηρείται ότι το speedup αυξάνεται ανάλογα με τον αριθμό των TFlux Kernels και ανάλογα με το μέγεθος του προβλήματος.



Σχήμα 8.2: Αποτελέσματα πειραμάτων σε πραγματικό σύστημα. Το speedup υπολογίστηκε με βάση το χρόνο εκτέλεσης του σειριακού προγράμματος

Για όλα τα benchmarks εκτός από το MMULT, παρατηρείται γραμμική αύξηση στο speedup, πλησιάζοντας κάθε φορά το ιδανικό speedup. Ο κύριος λόγος για τον οποίο το MMULT δεν έχει το ίδιο καλά αποτελέσματα με τα υπόλοιπα benchmarks είναι ο μεγάλος όγκος δεδομένων που χρησιμοποιείται. Η μη κανονικές προσβάσεις σε αυτά έχουν σαν αποτέλεσμα μεγάλο αριθμό misses στην L1 cache.

Τα αποτελέσματα στο πραγματικό σύστημα είναι συνεπή με τα αποτελέσματα των πειραμάτων στον Simulator. Επομένως μπορούμε να συμπεράνουμε ότι η επίδοση της πλατφόρμα είναι ανεξάρτητη από την αρχιτεκτονική στην οποία χρησιμοποιείται.

Κεφάλαιο 9 Συμπεράσματα – Μελλοντική Δουλειά

9.1 Συμπεράσματα

Σε αυτή την εργασία παρουσιάστηκε ένας Preprocessor για την πλατφόρμα TFlux-Soft καθώς και η διαδικασία δημιουργίας μίας σουίτας αξιολόγησης για την ίδια πλατφόρμα μέσω της μεταφοράς διαφόρων γνωστών και διαδεδομένων benchmarks. Επίσης, τα benchmarks αυτά χρησιμοποιήθηκαν για την αξιολόγηση της πλατφόρμας μέσω πειραμάτων, τόσο σε ένα *full-system Simulator* όσο και σε ένα πραγματικό σύστημα.

Η παρουσίαση και ανάλυση της κατασκευής και λειτουργίας του Preprocessor, αλλά και η περιγραφή της μεταφοράς διαφόρων εφαρμογών στην πλατφόρμα TFlux-Soft έδειξαν ότι ο προγραμματιστής διευκολύνεται σε πολύ μεγάλο βαθμό κατά τη μεταφορά ή ανάπτυξη μίας εφαρμογής για τη συγκεκριμένη πλατφόρμα. Τα *DDM pragma directives* τα οποία ο προγραμματιστής προσθέτει στον κώδικα του επιτρέπουν να εκφράσει με εύκολο αλλά και πολύ αποδοτικό τρόπο τον παραλληλισμό ενός προγράμματος, αφού το μόνο που έχει να κάνει είναι να εντοπίσει και να δηλώσει τις εξαρτήσεις μεταξύ των διαφόρων τμημάτων κώδικα (DDThrs) και την υπόλοιπη δουλειά για την παραλληλοποίηση την επιτελεί αυτόματα ο Preprocessor. Επίσης, με αυτό τον τρόπο, ο προγραμματιστής, αντί να πρέπει να εντοπίσει και να εκφράσει όλα τα παράλληλα τμήματα και όλο τον παραλληλισμό μίας εφαρμογής, στην ουσία εντοπίζει μόνο τα μέρη ενός προγράμματος τα οποία δεν μπορούν να τρέξουν παράλληλα και τα υπόλοιπα αφήνονται από την πλατφόρμα να εκτελεστούν παράλληλα.

Τα πιο πάνω έχουν ως αποτέλεσμα ευκολότερο προγραμματισμό της πλατφόρμας, αλλά και μεγαλύτερο περιθώριο για υψηλή επίδοση, αφού ο παραλληλισμός του κάθε προγράμματος δεσμεύεται μόνο από τις πραγματικές εξαρτήσεις δεδομένων και τους διαθέσιμους πόρους και όχι από οποιοδήποτε άλλο παράγοντα.

Τέλος, η πλατφόρμα αξιολογήθηκε χρησιμοποιώντας τη σουίτα αξιολόγησης από benchmarks, τόσο σε ένα *full-system Simulator* αλλά και σε ένα πραγματικό σύστημα, με πάρα πολύ καλά αλλά και συνεπή αποτελέσματα. Αυτό αποδεικνύει από τη μία ότι η πλατφόρμα έχει καλό scalability, αφού δοκιμάστηκε σε προσομοιωμένες μηχανές με μεγάλο αριθμό πυρήνων με σχεδόν ιδανικά αποτελέσματα σε όλες τις περιπτώσεις, αλλά ταυτόχρονα αποδεικνύει ότι η πλατφόρμα μπορεί να χρησιμοποιηθεί άμεσα σε πραγματικά συστήματα πολύ-πυρήνων που κυκλοφορούν σήμερα στην αγορά και πάλι με πάρα πολύ καλά αποτελέσματα και εκμεταλλευόμενη ανά πάσα στιγμή τις τελευταίες εξελίξεις της τεχνολογίας. Επίσης, από τα αποτελέσματα της αξιολόγησης φαίνεται ότι η απόδοση της πλατφόρμας δεν επηρεάζεται από την αρχιτεκτονική πάνω από την οποία εκτελείται.

9.2 Μελλοντική Δουλειά

Στο μέλλον, θα πρέπει να μεταφερθούν και άλλες εφαρμογές και benchmarks στην πλατφόρμα TFlux-Soft για καλύτερη και πιο ολοκληρωμένη αξιολόγησή της, καθώς και για μελέτη της συμπεριφοράς της για προγράμματα με διαφορετικούς Γράφους Συγχρονισμού από αυτούς που μελετήθηκαν. Παράλληλα και για να γίνει αυτό κατορθωτό, πολύ πιθανόν να χρειαστούν αλλαγές και προσθήκες στον κώδικα και στη λειτουργία του Preprocessor, ούτως ώστε να μπορεί να χειρίζεται εφαρμογές με πιο πολύπλοκους Γράφους Συγχρονισμού.

Τέλος, η πλατφόρμα TFlux-Soft, μαζί με τη σειρά εργαλείων της και τον Preprocessor θα γίνουν διαθέσιμα στο κοινό μέσω του διαδικτύου, ώστε να μπορούν όλοι οι ενδιαφερόμενοι να τη χρησιμοποιήσουν.

Βιβλιογραφία

[1] Arvind and Gostelow. *"The U-Interpreter"*, Computer, 1982

[2] Bailey, D. H. Barszcz, E. Barton, J. T. Browning, D. S. Carter, R. L. Dagnum, D. Fatoohi, R. A. Frederickson, P. O. Lasinski, T. A. Schreiber, R. S. Simon, H. D. Venkatakrishman, V. Weetarunga, S. K., *"The NAS Parallel Benchmarks"*, The International Journal of Supercomputer Applications 5 (1991) 63-73

[3] D. B. et al. "*Scaling to the End of Silicon with EDGE Architectures*", Computer, 2004

[4] D. Burger , S. W. Keckler , K. S. McKinley , M. Dahlin , L. K. John , C. Lin , C. R. Moore , J. Burrill , R. G. McDonald , W. Yoder , the TRIPS Team, *"Scaling to the End of Silicon with EDGE Architectures"*, Computer, v.37 n.7, p.44-55, July 2004

[5] D. E. Culler, K. E. Schauser and T. von Eicken, *"Two Fundamental Limits on Dataflow Multiprocessing"*, Proceedings of the IFIP WG10.3, 1993

[6] David R. Butenhof. "*Programming with POSIX Threads*", Addison-Wesley, ISBN 0-201-63392-2

[7] Guthaus, M. R. Ringenberg, J. S. Ernst, D. Austin, T. M. Mudge, T. Brown, R. B., *"MiBench: A free, commercially representative embedded benchmark suite"*, Proceedings of the 4th IEEE International Workshop on Workload Characterization (WWC-4), Washington, DC, USA, IEEE Computer Society (2001) 3-14

[8] Hammond e*t al. "The Stanford Hydra CMP"*, IEEE Micro, 2000

[9] Haoqiang Jin, M. F. Yan, J.: *"The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance"*, Technical Report NAS Technical Report NAS-99-011, NASA Armes Research Center – NAS System Division (1999)

[10] Hartstein and Puzak. *"The optimum pipeline depth for a microprocessor"*, Proceedings of the 29th annual international Symposium on Computer Architecture, 2002

[11] Herbert H.J. Hum et al. *"A design study of the EARTH multiprocessor"*, Proceedings of Parallel Architectures and Compilation Techniques, 1995

[12] Intel, *"Intel Core2 Duo Processor Overview"*, www.intel.com/products/processor/core2duo/index.htm, 2008

[13] Jimenez. *"Piecewise Linear Branch Prediction"*, ACM SIGARCH Computer Architecture News, Volume 33 , Issue 2, Pages 382 – 393, 2005

[14 K. Chang et al. "*The case for a single-chip multiprocessor"*, Proceedings of ASPLOS-VII, 1996

[15] K. Olukotun, L. Hammond, and M. Willey. "*Improving the performance of speculatively parallel applications on the hydra CMP*", Proceedings of ICS, pages 21-30, 1999

[16] K. Stavrou et al. "*TFlux: A Portable Platform for Data-Driven Multithreading on Commodity Multicore Systems*", Proceedings of ICPP, 2008

[17] Oren. *"A survey on prefetching techniques"*, ACM Computing Surveys (CSUR), Volume 32 , Issue 2, Pages 174 – 199, 2000

[18] P. Evripidou. "*Thread Synchronization Unit (TSU): A building block for High Performance Computers"*. Proceedings of the ISHPC, 1997

[19] P. Evripidou and C. Kyriacou, *"Data driven network of workstations (D^2NOW)"*, J. UCS, 2000

[20] P. Evripidou and J. Gaudiot. *"A Decoupled graph/computation Data-Driven architecture with variable resolution actors"*, Proceedings of ICPP, 1990

[21] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt and B. Werner. *"Simics: A Full System Simulation Platform"*. IEEE Computer, 35(2):50-58, February 2002

[22] P. Trancoso, K. Stavrou and P. Evripidou. *"DDMCPP: The data-driven multithreading C Pre-Processor"*, pages 32-39, February 2007

[23] Smith. *"Sequential program prefetching in memory hierarchies."*, Computer, 1978

[24] Smith et al. *"The microarchitecture of superscalar processors"*, Proceedings of the IEEE Volume 83, Issue 12, Dec 1995 Pages 1609 – 1624