"Προηγμένα Συστήματα Υπολογιστών & Επικοινωνιών"
Διαπανεπιστημιακό Διατμηματικό Πρόγραμμα Μεταπτυχιακών Σπουδών

• Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
• Ιατρικής • Ψυχολογίας • Μουσικών Σπουδών • Δημοσιογραφίας και ΜΜΕ • Γενικό
• Μηχανικών Η/Υ, Τηλεπικοινωνιών και Δικτύων του Πανεπιστημίου Θεσσαλίας
• Λογιστικής και Χρηματοοικονομικής του Πανεπιστημίου Μακεδονίας

ΑΡΙΣΤΟΤΕΛΕΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΟΝΙΚΗΣ
ΠΑΝΕΠΙΣΤΗΜΙΟ ΜΑΚΕΔΟΝΙΑΣ
ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

Αυτοματοποίηση παραγωγής κώδικα για αρχιτεκτονικές GPU

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Ευάγγελου Σαββίδη

Επιβλέπων: Νικόλαος Πιτσιάνης
Επίκουρος Καθηγητής

Θεσσαλονίκη, Μάιος 2011

Περίληψη

Το αντικείμενο που πραγματεύεται η διπλωματική, είναι η αυτοματοποίηση της σύνθεσης βελτιστοποιημένου κώδικα υψηλού επιπέδου για αρχιτεκτονικές GPU, με χρήση μηχανών παραγωγής κώδικα. Για το σκοπό αυτό υλοποιήθηκε εργαλείο, το οποίο συναθροίζει ορισμένες τεχνολογίες και φτάνει τελικά σε αποδεκτό κώδικα CUDA. Ο κώδικας ο οποίος παράγεται έχει ως σκοπό να λύσει παρόμοια προβλήματα γραμμικής άλγεβρας, όπως ο πολλαπλασιαμός και η συνέλιξη πινάκων και γι'αυτό είναι απαραίτητο να έχει όσο το δυνατόν μικρότερο χρόνο εκτέλεσης.

Η μηχανή παραγωγής κώδικα που χρησιμοποιήθηκε είναι η JBURG, η οποία είναι κατασκευασμένη με Java. Πιο συγκεκριμένα, χρησιμοποιήσαμε τη γεννήτρια γεννητριών κώδικα, για να παράξουμε τη BURM μηχανή, χωρίς να κρατήσουμε όμως αυτούσια τη λειτουργία της. Αυτό γιατί η συγκεκριμένη μηχανή ασχολείται με την παραγωγή bytecode, ο οποίος είναι κώδικας χαμηλού επιπέδου για πλατφόρμες java. Προσθέσαμε σε αυτό το λογισμικό κλάσεις, που παράγουν κώδικα CUDA.

Επιπλέον, επειδή η απόδοση ενός προγράμματος γραμμένο για μια αρχιτεκτονική αυτού του τύπου (GPU), επηρεάζεται άμεσα από τον τρόπο προσπέλασης της μνήμης και από τον αριθμό των βρόχων, δόθηκε ιδιαίτερη έμφαση σε αυτά τα σημεία με σκοπό την επίτευξη ταχύτερου κώδικα. Χρησιμοποιήθηκαν διάφοροι μετασχηματισμοί βρόχων, - που εντάσσονται στην ευρύτερη κατηγορία της βελτιστοποίησης μεταγλωττιστών (όπως αναφέρεται στο κεφάλαιο 2) - , ώστε να εκμεταλλευτούμε στο έπακρο τα διάφορα χαρακτηριστικά κάθε είδους μνήμης, όπως ταχύτητα διαμεταγωγής, χώρος αποθήκευσης και ταχύτητα προσπέλασης.

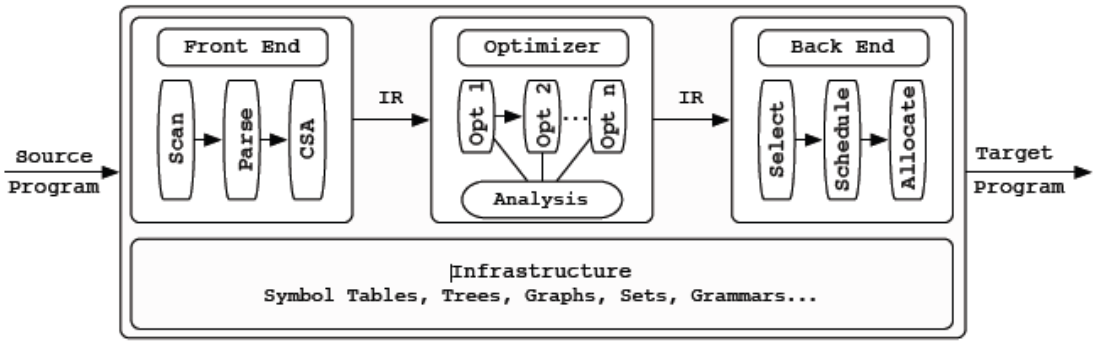
Όλα τα πειράματα εκτελέστηκαν στην πλατφόρμα Tesla C1060 της Nvidia, που είναι κατασκευασμένη αποκλειστικά για την εκτέλεση προγραμμάτων που απαιτούν μαζική επεξεργαστική ισχύ.

Πίνακας Περιεχομένων

1 Μεταγλωττιστές (Compilers)

1.1 Επισκόπηση της μεταγλώττισης

Για τον τελικό χρήστη, ένας μεταγλωττιστής θα πρέπει να συμπεριφέρεται σαν ένα μαύρο κουτί που παίρνει σαν είσοδο ένα πρόγραμμα γραμμένο σε μια συγκεκριμένη γλώσσα προγραμματισμού, και παράγει ένα μεταφρασμένο πρόγραμμα σαν έξοδο [7]. Αυτή η διαδικασία μπορεί να περιλαμβάνει μετατροπή του προγράμματος από μια γλώσσα υψηλού επιπέδου σε μια άλλη γλώσσα επίσης υψηλού επιπέδου. Η παρακάτω εικόνα δείχνει μια τυπική δομή που συναντάται στους περισσότερους μεταγλωττιστές.



Εικόνα 1.1.1 Η δομή ενός τυπικού μεταγλωττιστή [7]

- Το front end ενός compiler, είναι υπεύθυνο για την κατανόηση του συντακτικού και της σημασιολογίας του πηγαίου προγράμματος. Κατά τη φάση αυτή της μεταγλώττισης, επαληθεύεται η λεξικογραφική και γραμματική δομή της εισόδου. Η εννοιολογική ανάλυση (Context Sensitive Analysis CSA) στατιστικά επαληθεύει την εγκυρότητα της σημασιολογίας του πηγαίου προγράμματος (type inference, scope analysis). Αυτή η φάση επίσης βοηθάει τον προγραμματιστή με μηνύματα σφαλμάτων κατά την ανάπτυξη του προγράμματος.

- Το ενδιάμεσο τμήμα αποτελείται από μια ακολουθία μετασχηματισμών βελτιστοποίησης. Μέθοδοι βελτιστοποίησης, όπως αναδίπλωση σταθερών, strictness analysis, εξάλειψη περιττών δηλώσεων(redundancy elimination),

βαθμιαία βελτιστοποίηση (scalar optimization) και δυναμική μείωση (strength reduction) χρησιμοποιούνται.

- Τελικά το back end παίρνει το βελτιστοποιημένο πηγαίο πρόγραμμα και παράγει το πρόγραμμα – έξοδο. Ένα πρόγραμμα – έξοδος (target program) μπορεί να είναι οτιδήποτε ανάμεσα σε μια γλώσσα υψηλού επιπέδου και μια γλώσσα μηχανής χαμηλού επιπέδου [7].

Η δομή που φαίνεται στην παραπάνω εικόνα, αναδεικνύει τη σημαντικότητα επιλογής αποδοτικών δομών δεδομένων και εργαλείων αυτοματοποίησης, καθώς αυτές οι επιλογές επηρεάζουν σε μεγάλο βαθμό την απόδοση, τη χρήση των πόρων, και την πολυπλοκότητα ενός μεταγλωττιστή(compiler).

1.2 Ενδιάμεση αναπαράσταση (Intermediate Representation)

Οι περισσότεροι μεταγλωττιστές μεταφράζουν το πηγαίο πρόγραμμα αρχικά σε μια μορφή ενδιάμεσης αναπαράστασης και στη συνέχεια σε κώδικα γλώσσας μηχανής. Μια ενδιάμεση αναπαράσταση, είναι μια εκδοχή του αρχικού πηγαίου κώδικα, ανεξάρτητη από γλώσσα μηχανής και αρχικής υλοποίησης [24]. Παρόλο που αυτή η μετατροπή του αρχικού κώδικα εισάγει ένα επιπλέον βήμα στην επεξεργασία, η χρήση της ενδιάμεσης αναπαράστασης παρέχει πλεονεκτήματα λόγω του ότι είναι αφηρημένη, του πιο ξεκάθαρου διαχωρισμού ανάμεσα στο μπροστά και πίσω άκρο του μεταγλωττιστή, και επιπλέον παρέχει τη δυνατότητα για retargeting/cross compilation (επανασχεδιασμό/δια-μεταγλώττιση). Πολύ σημαντικό είναι και το γεγονός ότι οι μεταγλωττιστές μπορούν να υποστηρίξουν προηγμένες τεχνικές βελτιστοποιήσεων, και οι βελτιστοποιήσεις αυτές εφαρμόζονται στις ενδιάμεσες αναπαραστάσεις.

Υπάρχουν πολλές ενδιάμεσες αναπαραστάσεις σε χρήση (αρκετοί υποστηρίζουν ότι υπάρχει μια ξεχωριστού είδους ενδιάμεση αναπαράσταση για κάθε έναν μεταγλωττιστή), αλλά περισσότερο μοιάζουν παρά διαφέρουν. Οι

ενδιάμεσες αναπαραστάσεις συνήθως κατηγοριοποιούνται σύμφωνα με το ποια θέση κατέχουν ανάμεσα σε μια γλώσσα υψηλού επιπέδου και μια γλώσσα μηχανής. Οι IR (Intermediate Representations) που είναι κοντά σε γλώσσες υψηλού επιπέδου καλούνται IR υψηλού επιπέδου και αντίστοιχα αυτές πιο κοντά στη γλώσσα μηχανής IR χαμηλού επιπέδου. Για παράδειγμα μια υψηλού επιπέδου IR θα μπορούσε να διατηρεί στοιχεία όπως δείκτες πινάκων ή προσπελάσεις πεδίων, ενώ μια χαμηλού επιπέδου μετατρέπει τέτοιου είδους εκφράσεις σε ακριβείς διευθύνσεις μνήμης και περιθώρια [24]. Ένα τέτοιο παράδειγμα είναι το παρακάτω, όπου αναπαρίσταται η προσπέλαση ενός δισδιάστατου πίνακα:

Original	**High IR**	**Mid IR**	**Low IR**
float a[10][20]; a[i][j+2];	t1 = a[i, j+2]	t1 = j + 2 t2 = i * 20 t3 = t1 + t2 t4 = 4 * t3 t5 = addr a t6 = t5 + t4 t7 = *t6	r1 = [fp - 4] r2 = [r1 + 2] r3 = [fp - 8] r4 = r3 * 20 r5 = r4 + r2 r6 = 4 * r5 r7 = fp – 216 f1 = [r7 + r6]

Αυτό που μπορούμε να παρατηρήσουμε είναι το γεγονός ότι το IR χαμηλού επιπέδου έχει διαφορετικές πληροφορίες από αυτές που έχει το υψηλού επιπέδου IR. Σε αυτό το σημείο εγείρονται ερωτήματα όπως: Τι πληροφορίες έχει ένα υψηλού επιπέδου IR που δεν έχει ένα χαμηλού επιπέδου IR; Τι πληροφορίες έχει ένα χαμηλού επιπέδου IR που δεν τις έχει ένα υψηλού επιπέδου; Τι είδους βελτιστοποιήσεις θα μπορούσαν να πραγματοποιηθούν στο ένα και τι είδους στο άλλο; Τα IR υψηλού επιπέδου συνήθως διατηρούν πληροφορίες όπως εκφράσεις δομής βρόχων και υπό συνθήκης. Τείνουν να αντικατοπτρίζουν την πηγαία γλώσσα που μεταγλωττίζουν περισσότερο από ότι τα χαμηλού επιπέδου IR.

Τα IR μέσου επιπέδου συνήθως επιχειρούν να είναι ανεξάρτητα και από την πηγαία γλώσσα, αλλά και από την παραγόμενη. Τα χαμηλού επιπέδου IR όμως, τείνουν να αντικατοπτρίζουν πολύ την γλώσσα μηχανής και ως εκ τούτου, είναι εξαρτημένα από τη μηχανή. Μερικές φορές ένας μεταγλωττιστής μπορεί να ξεκινήσει με ένα υψηλού επιπέδου IR να εκτελέσει κάποιες βελτιστοποιήσεις, να το μετατρέψει στη συνέχεια σε ένα χαμηλότερου επιπέδου IR, να ξανακάνει μερικές

βελτιστοποιήσεις και ακολουθώντας αυτό τον κυκλικό μοτίβο, να καταλήξει σταδιακά στην παραγωγή του τελικού κώδικα.

1.2.1 Δέντρα Αφηρημένου Συντακτικού (Abstract Syntax Trees)

Ένα parse tree είναι ένα παράδειγμα μιας πολύ υψηλού επιπέδου ενδιάμεσης αναπαράστασης. Συνήθως υπάρχει η δυνατότητα να ανακατασκευαστεί πλήρως ο πηγαίος κώδικας από ένα τέτοιο δέντρο, καθώς περιέχει όλη την πληροφορία για το πηγαίο πρόγραμμα (parsed program).

Μια δενδρική αναπαράσταση που χρησιμοποιείται σαν ένα IR δεν είναι ακριβώς το δέντρο συμπαγούς συντακτικού (literal parse tree) καθώς οι ενδιάμεσοι κόμβοι μπορεί να έχουν καταρρεύσει ή άλλοι κόμβοι μπορεί να έχουν απαλλαχθεί από πληροφορίες, αλλά είναι επαρκές για να γίνει η σημασιολογική επεξεργασία και η παραγωγή κώδικα.

Ένα τέτοιο δέντρο συνήθως καλείται *Δέντρο Αφηρημένου Συντακτικού* (*Abstract Syntax Tree*), ενώ το αντίθετο, ένα δέντρο δηλαδή που παρέχει μια 1:1 αντιστοίχιση χωρίς κόμβους που έχουν καταρρεύσει και με όλα τα τερματικά στα φύλλα, καλείται δέντρο συμπαγούς συντακτικού (*Concrete Syntax Tree*) [24]. Κάθε κόμβος αναπαριστά ένα κομμάτι της δομής του προγράμματος και έχει αναφορές στα υποδέντρα – παιδιά του, - ή καμία αναφορά αν πρόκειται για κόμβο φύλλο - και πιθανότατα έχει και αναφορές προς τον κόμβο γονέα του.

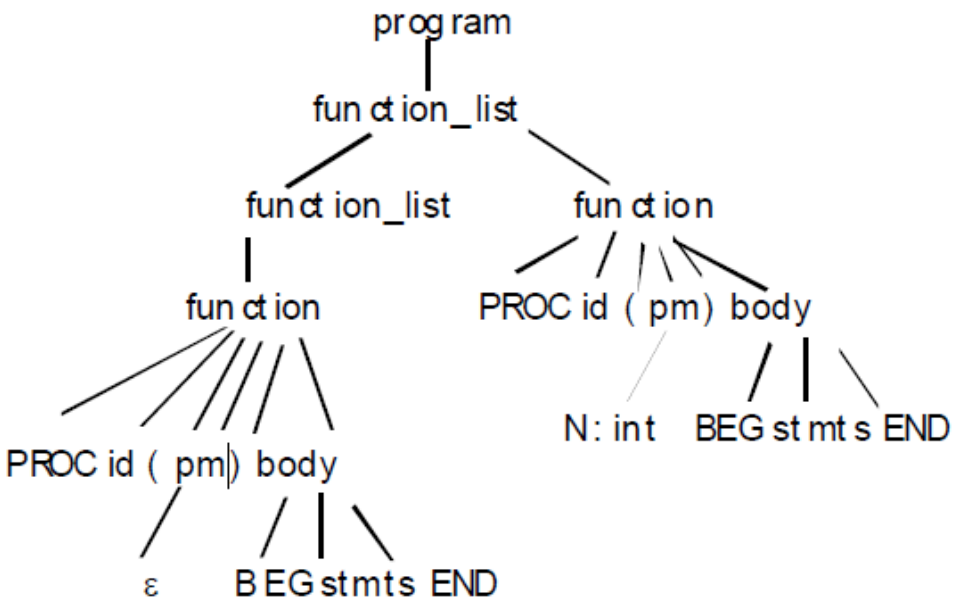
Ένα πηγαίο πρόγραμμα θα μπορούσε να είναι το παρακάτω:

```
PROCEDURE main()
BEGIN
     statement...
END

PROCEDURE factorial(n:INTEGER)
BEGIN
     statement...
END
```

Με τη γραμματική της γλώσσας προγραμματισμού να είναι η εξής:

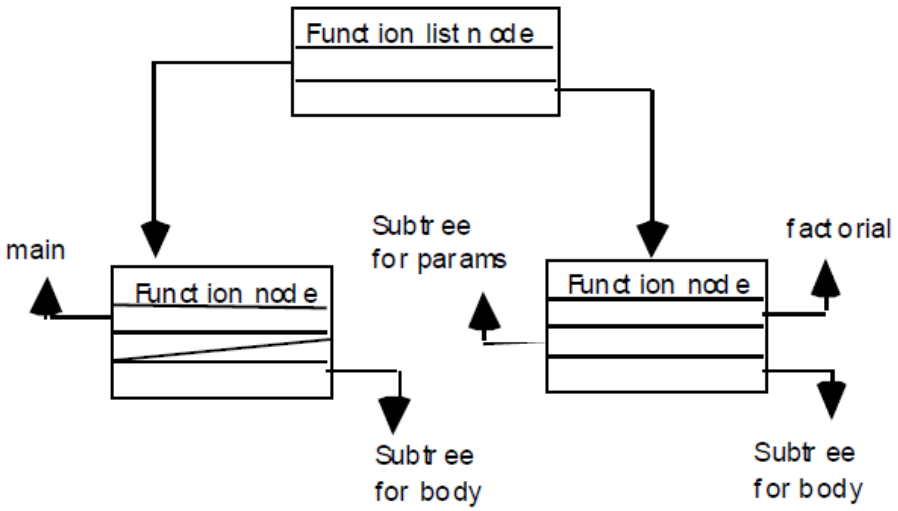
```
program -> function_list
function_list -> function_list function | function
function -> PROCEDURE ident ( params ) body
params -> …
```

Οπότε το δέντρο συμπαγούς συντακτικού (literal parse tree ή concrete syntax tree) για το προηγούμενο παράδειγμα είναι όπως παρακάτω:



Εικόνα 1.2.1.1 Δέντρο συμπαγούς συντακτικού (concrete syntax tree) [24]

Τελικά ένα δέντρο αφηρημένου συντακτικού θα είναι:



Εικόνα 1.2.1.2 Δέντρο αφηρημένου συντακτικού (abstract syntax tree) [24]

Μπορούμε να παρατηρήσουμε ότι μερικά κομμάτια όπως κάποιοι κόμβοι γονείς και λέξεις κλειδιά δε χρειάζονται πλέον σε αυτή την αναπαράσταση.

1.3 Από τη γλώσσα εισόδου στην ενδιάμεση αναπαράσταση

Για να παραχθεί κώδικας μηχανής από μια γλώσσα υψηλού επιπέδου, κάθε τμήμα ενός μεταγλωττιστή διαχειρίζεται και πιθανώς τροποποιεί μια ενδιάμεση αναπαράσταση του πηγαίου προγράμματος (Intermediate Representation – IR).
Το μπροστά άκρο (front end) αντιστοιχίζει μια υψηλού επιπέδου γλώσσα σε μια ενδιάμεση αναπαράσταση (IR) σαρώνοντας τα μέρη της εισόδου, και στη συνέχεια αναλύοντάς τα με σκοπό να αναγνωρίσει συντακτικό γλώσσας. Αυτός ο μετασχηματισμός επίσης καταγράφει πληροφορίες για αργότερη χρήση κατά τη διάρκεια της εννοιολογικής ανάλυσης.

Εκτεταμένη μελέτη και έρευνα στις περιοχές της σάρωσης και ανάλυσης, έχουν οδηγήσει στη δημιουργία εργαλείων που αυτοματοποιούν τις περισσότερες ενέργειες ενός σαρωτή και αναλυτή για ένα μεγάλο εύρος γραμματικής. Έτσι ο δημιουργός ενός μεταγλωττιστή, χρειάζεται μόνο να προσδιορίσει το συντακτικό με μια γλώσσα τεκμηρίωσης, και να παρέχει σημασιολογικές ενέργειες που εκτελούνται κατά την αναγνώριση συντακτικών φράσεων [7].

1.4 Μετασχηματισμοί της ενδιάμεσης αναπαράστασης

Κατά τη φάση των βελτιστοποιήσεων του μεταγλωττιστή, μια δομή ενδιάμεσης αναπαράστασης που έχει παραχθεί από το front end δέχεται μια ακολουθία από μετασχηματισμούς, και έτσι παράγεται μια βελτιστοποιημένη δομή ενδιάμεσης αναπαράστασης. *Βελτιστοποιημένη δομή* σημαίνει ότι η δομή αυτή θα έχει ως αποτέλεσμα την παραγωγή ταχύτερου κώδικα, πιο συμπυκνωμένου ο οποίος θα διαχειρίζεται πιο αποδοτικά τους πόρους. Ένας βελτιστοποιητής (optimizer) μπορεί να κάνει πολλά περάσματα μιας ενδιάμεσης δομής,

μετασχηματίζοντάς τη σε μια πιο αποδοτική και συγκεντρώνοντας αρκετές πληροφορίες, ώστε να εκτελεί συγκεκριμένους τύπους βελτιστοποιήσεων [7].

1.5 Από την ενδιάμεση αναπαράσταση στον κώδικα μηχανής (target code)

Τελικά το πίσω τμήμα (back end) του μεταγλωττιστή παράγει εξειδικευμένο κώδικα μηχανής, από μια βελτιστοποιημένη δομή ενδιάμεσης αναπαράστασης. Η παραγωγή συγκεκριμένης γλώσσας μηχανής, συνεπάγεται ότι επιλέγονται οι σωστές εντολές που υλοποιούν τις λειτουργίες που περιγράφονται στην ενδιάμεση αναπαράσταση, επιλέγοντας σειρά εκτέλεσης καθώς και αποφασίζοντας ποιες τιμές θα βρίσκονται σε καταχωρητές (registers) και ποιες στη μνήμη [7].

1.6 Σύνοψη

Ένας τυπικός μεταγλωττιστής αποτελείται από ένα front end που αναγνωρίζει τη γλώσσα εισόδου, ένα ενδιάμεσο τμήμα που βελτιστοποιεί και ένα back end τμήμα παραγωγής κώδικα. Η ανάπτυξη ενός σωστού μεταγλωττιστή-βελτιστοποιητή είναι μια πολύπλοκη διαδικασία αλλά οι πιο εξέχουσες διαδικασίες της μεταγλώττισης όπως η σάρωση και η ανάλυση μπορούν να αυτοματοποιηθούν. Σκοπός δικός μας είναι η αυτοματοποίηση της βελτιστοποίησης της ενδιάμεσης δομής και η παραγωγή κώδικα υψηλού επιπέδου, στο back end ενός μεταγλωττιστή.

2 Βελτιστοποιήσεις μεταγλωττιστών

Οι εκπληκτικές βελτιώσεις στις ταχύτητες των υπολογιστών τις τελευταίες τρεις δεκαετίες είναι αποτέλεσμα δύο φαινομένων. Πρώτα είναι η αξιοσημείωτη πρόοδος της τεχνολογίας με την οποία κατασκευάζονται οι μηχανές, όπου έχει αναπτυχθεί με τέτοιο ρυθμό, όπως προβλέφτηκε από τον νόμο του Moore. Ωστόσο η τεχνολογία από μόνη της δεν είναι αρκετή. Ο παραλληλισμός από τη μεριά του

είναι αυτός που συνδυασμένος με την τεχνολογία μπορεί να δώσει εξαιρετικές αποδόσεις [1].

Για το πρόβλημά μας επικεντρωθήκαμε στην εύρεση παραλληλισμού για πολλαπλούς ασύγχρονους επεξεργαστές, με διαμοιραζόμενη και καθολική μνήμη. Ο παραλληλισμός αυτού του τύπου ονομάζεται Coarse-grained και απαιτεί προσοχή σε διάφορα σημεία. Σε τέτοιες αρχιτεκτονικές ανατίθενται πολλά νήματα εκτέλεσης στους επεξεργαστές που εκτελούνται παράλληλα για ένα χρονικό διάστημα, με περιστασιακό συγχρονισμό. Το κλειδί στην επίτευξη υψηλής απόδοσης σε τέτοια συστήματα είναι να βρεθεί ένα 'πακέτο' παραλληλισμού με τέτοια τμηματοποίηση ώστε να ανταπεξέρχεται στις καθυστερήσεις (overhead) από τις αρχικοποιήσεις (initiation) και τους συγχρονισμούς της παράλληλης εκτέλεσης. Έτσι οι προσπάθειες επικεντρώνονται στην εύρεση παράλληλων βρόχων με επαρκή ποσοστά υπολογισμών. Αυτό συνήθως σημαίνει παραλληλισμό των εξωτερικών βρόχων και όχι των εσωτερικών, ή παραλληλισμό βρόχων που περιλαμβάνουν κλήσεις υπορουτινών, με το τελευταίο βέβαια να είναι εκτός της εμβέλειας του θέματος που πραγματευόμαστε.

Η παραγωγή παράλληλου κώδικα για αρχιτεκτονικές coarse-grained (GPU) περιλαμβάνει τη διαχείριση πολλών λεπτών θεμάτων. Ένα από τα πιο δύσκολα είναι η ελαχιστοποίηση των καθυστερήσεων των επικοινωνιών και του συγχρονισμού, καθώς και το ισοζύγισμα του φόρτου εργασίας ανάμεσα σε όλους τους παράλληλους επεξεργαστές. Από τη μια μεριά η απόλυτη ελάχιστη καθυστέρηση πετυχαίνεται όταν ολόκληρο το πρόγραμμα εκτελείται σε έναν επεξεργαστή – καθώς δεν υπάρχει παραλληλισμός και έτσι δεν υπάρχουν επικοινωνίες μεταξύ των επεξεργαστών και συγχρονισμός. Φυσικά η κατανομή του φόρτου εργασίας και η ταχύτητα είναι μη βέλτιστα. Από την άλλη μεριά, η καλύτερη κατανομή του φόρτου εργασίας γίνεται όταν το αρχικό πρόγραμμα αποσυντίθεται στα μικρότερα δυνατά παράλληλα στοιχεία, με τα στοιχεία αυτά να κατανέμονται σε ανενεργούς επεξεργαστές. Έχοντας πολύ μικρά παράλληλα στοιχεία του αρχικού προγράμματος, κανένας επεξεργαστής δεν φορτώνεται με μεγάλα ποσά εργασίας ενώ κάποιοι άλλοι είναι ανενεργοί και περιμένουν. Ωστόσο το κόστος του συγχρονισμού και των επικοινωνιών μεγιστοποιείται σε αυτή την περίπτωση και αυτό το κόστος σχεδόν πάντα ξεπερνάει τα οφέλη που προκύπτουν από την άψογη κατανομή του φόρτου

εργασίας. Κάπου ανάμεσα στις δυο πλευρές βρίσκεται η πιο αποτελεσματική παράλληλη αποσύνθεση, και ένας μεταγλωττιστής επιφορτισμένος με αυτό το έργο, αντιμετωπίζει την πρόκληση της εύρεσης της χρυσής αυτής τομής.

Στη συνέχεια παρουσιάζονται συνοπτικά ορισμένοι μετασχηματισμοί που ευνοούν την καλύτερη διαχείριση του υλικού μιας αρχιτεκτονικής GPU.

2.1 Loop Interchange

Εξαιτίας της ικανότητάς του να επηρεάζει δραστικά τη σειρά εκτέλεσης μεγάλου αριθμού εκφράσεων το Loop interchange είναι ένας πολύτιμος μετασχηματισμός [1]. Ο παραλληλισμός παρουσιάζει ένα διαφορετικό σύνολο προβλημάτων για την ανταλλαγή βρόχων (και για τους μετασχηματισμούς γενικότερα). Συγκεκριμένα η ανάγκη για ισορροπία ανάμεσα στον παραλληλισμό και τα κόστη επικοινωνίας και συγχρονισμού, απαιτεί οι παράλληλες περιοχές να είναι επαρκώς τμηματικές για να ξεπεράσουν αυτά τα κόστη. Με άλλα λόγια μια επιτυχής παράλληλη αποσύνθεση, πρέπει να παράγει λογικό φόρτο εργασίας, αλλά με τα κόστη των επικοινωνιών και του συγχρονισμού μειωμένα επαρκώς ώστε να διατηρούνται τα οφέλη της παράλληλης επεξεργασίας. Καθώς οι βρόχοι γενικά εκτελούνται αρκετές φορές για να παρέχουν κατανομή του φόρτου, η εφαρμογή της προηγούμενης αρχής, έγκειται στην παραλληλοποίηση των εξωτερικών βρόχων. Η επίπτωση αυτής της αρχής όσον αφορά την ανταλλαγή βρόχων, είναι ότι τα loops που δεν έχουν εξαρτήσεις πρέπει να μεταφέρονται στην πιο έξω δυνατή θέση, όσο βέβαια αυτή η μεταφορά δε δημιουργεί εξαρτήσεις. Το ακόλουθο παράδειγμα δείχνει μια τέτοια ανταλλαγή βρόχων:

```
DO I = 1, N
     DO J = 1, M
          A(I+1,J) = A(I,J) + B(I,J)
     ENDDO
ENDDO
```

Το εξωτερικό loop (I) μεταφέρει μια εξάρτηση επανάληψης, ενώ το εσωτερικό (J) δεν έχει εξαρτήσεις. Για να πετύχουμε παραλληλισμό coarse-grained αυτή η θέση των βρόχων είναι προβληματική, επειδή μόνο το εσωτερικό loop μπορεί να

παραλληλοποιηθεί, κάτι που θα απαιτούσε N συγχρονισμούς μπάρας (barrier synchronization) κατά το χρόνο εκτέλεσης, έναν για κάθε επανάληψη του σειριακού εξωτερικού loop. Από την άλλη μεριά, εάν το παράλληλο loop μεταφερθεί στην εξωτερική θέση (όπου θα εξακολουθεί να μην έχει εξαρτήσεις):

```
PARALLEL DO J = 1, M
     DO I = 1, N
          A(I+1,J) = A(I,J) + B(I,J)
     ENDDO
END PARALLEL DO
```

μόνο ένας συγχρονισμός θα απαιτείται κατά τη διάρκεια της εκτέλεσης του εμφωλευμένου loop. Φυσικά δεν είναι πάντα δυνατό να μεταφέρουμε ένα παράλληλο loop προς τα έξω και αυτό να εξακολουθεί να μην έχει εξαρτήσεις. Εάν για παράδειγμα το παραπάνω παράδειγμα αλλάξει λίγο

```
DO I = 1, N
     DO J = 1, M
          A(I+1,J+1) = A(I,J) + B(I,J)
     ENDDO
ENDDO
```

Τότε η ανταλλαγή δεν είναι αποτελεσματική. Η εξάρτηση δε θα εξαλειφθεί με την μεταφορά του εσωτερικού loop στην εξωτερική θέση και θα συνεχίσει να υφίσταται. Είναι δεδομένο ότι στις περισσότερες των περιπτώσεων χρειάζεται η βοήθεια και άλλων μετασχηματισμών.

2.2 Strip Mining

Πολλοί βρόχοι ελεύθεροι εξαρτήσεων των οποίων οι επαναλήψεις μπορούν να εκτελεστούν παράλληλα μπορεί να μην εκτελούνται επαρκώς παράλληλα, αν δεν προγραμματιστούν σωστά. Πολύ συχνά μπορεί να μην υπάρχει αρκετή δουλειά σε μια απλή επανάληψη του βρόχου, ώστε να δικαιολογεί τον προγραμματισμό της σαν μια παράλληλη οντότητα. Σε μια τέτοια περίπτωση, η ομαδοποίηση των επαναλήψεων σε σύνολα καθένα από τα οποία είναι μια προγραμματίσιμη μονάδα, μπορεί να παρέχει πιο αποδοτική χρήση του παραλληλισμού. Ένας από τους βασικότερους μετασχηματισμούς για να το πετύχουμε αυτό είναι το Strip Mining,

που μετατρέπει τον διαθέσιμο παραλληλισμό σε μορφή πιο κατάλληλη για το υλικό (hardware) [1]. Το ακόλουθο απλό loop επιδεικνύει τα πλεονεκτήματα αυτού του μετασχηματισμού για παραλληλισμό.

```
DO I = 1, N
     A(I) = A(I) + B(I)
ENDDO
```

Αν υπάρχουν ακριβώς P επεξεργαστές διαθέσιμοι να εκτελέσουν το loop, τότε η καλύτερη κατανομή φόρτου και η ελαχιστοποίηση των συγχρονισμών πετυχαίνεται ως εξής:

```
k = CEIL(N/P)
PARALLEL DO I = 1, N, k
     DO i = I, MIN(I+k-1,N)
          A(i) = A(i) + B(i)
     ENDDO
END PARALLEL DO
```

Σε βρόχους όπως ο παραπάνω, όπου κάθε επανάληψη ξεκάθαρα απαιτεί το ίδιο ποσό υπολογισμών, η εύρεση κατάλληλου διαμοιρασμού είναι εύκολη όταν είναι γνωστός ο αριθμός των επαναλήψεων του βρόχου και ο αριθμός των διαθέσιμων επεξεργαστών. Όταν αυτές οι παράμετροι δεν είναι γνωστές μέχρι την ώρα της εκτέλεσης το strip mining εκτελείται από ειδικό υλικό (hardware).

Σε περιπτώσεις όπου ο χρόνος εκτέλεσης ποικίλλει ανάμεσα στις επαναλήψεις, η εύρεση μιας κατάλληλης κατανομής φόρτου εργασίας είναι πολύ πιο δύσκολη. Αντί να κάνουμε strip mining για παραλληλισμό σε υποπολλαπλάσια του αριθμού των επεξεργαστών, συνήθως είναι προτιμότερο να διαλέγουμε μικρότερο μέγεθος block. Έτσι οι επεξεργαστές που κάνουν λιγότερη δουλειά, μπορούν να αναλάβουν κάποια επιπλέον, όσο οι βαρύτερα φορτωμένοι επεξεργαστές ακόμα δουλεύουν.

2.3 Blocking – Loop tiling

Το Strip mining είναι ένας χρήσιμος μετασχηματισμός, εάν το σώμα του βρόχου εκτελεί υπολογισμούς γραμμικά κατά μήκος ενός πίνακα. Εάν όμως οι

υπολογισμοί προσπελάζουν έναν πίνακα με έναν τρόπο του τύπου γραμμές – στήλες (row – column wise accesses), καθώς οι προσπελάσεις ανά γραμμή (row-wise accesses) είναι ορθογωνικές σε σχέση με τη διάσχιση (orthogonal to the strip) θα απαιτούν διαφορετικού μεγέθους προσωρινή μνήμη κάθε φορά. Σταδιακά, θα υπάρξει σύγκρουση ανάμεσα στην γραμμή της προσωρινής μνήμης που χρησιμοποιείται και η απόδοση θα πέσει σημαντικά [1].

Το Loop tiling διορθώνει αυτό το πρόβλημα, καθώς εκτελεί strip mining σε πολλαπλές διαστάσεις πίνακα, περιορίζοντας το σύνολο εργασίας έτσι ώστε να χωράει στο μήκος της προσωρινής μνήμης (για διάσχιση στηλών) , αλλά και στον αριθμό των γραμμών της μνήμης (για διάσχιση γραμμών), χωρίζοντας τον πίνακα σε τμήματα – blocks μεγέθους της προσωρινής μνήμης. Το ακόλουθο παράδειγμα δείχνει αυτό ακριβώς για το πρόβλημα της αναστροφής πινάκων:

```
DO I = 1, N
     DO J = 1, N
          A(I,J) = B(J,I)
     ENDDO
ENDDO
```

Με την εφαρμογή Strip mining και στα δύο loops (δηλαδή tiling), το σύνολο δεδομένων εργασίας, περιορίζεται στο μέγεθος της προσωρινής μνήμης (cache memory): 64 γραμμές από 64 στοιχεία η κάθε μία σε αυτή την περίπτωση.

```
DO TI = 1, N, 64
     DO TJ = 1, N, 64
          DO I = TI, MIN(TI+63, N)
               DO J = TJ, MIN(TJ+63, N)
                    A(I,J) = B(J,I)
               ENDDO
          ENDDO
     ENDDO
ENDDO
```

2.4 Loop Unrolling

Το loop unrolling είναι ένας μετασχηματισμός που παραδοσιακά στοχεύει στη μείωση του χρόνου εκτέλεσης ενός βρόχου, μειώνοντας τον αριθμό των υπολογισμών που απαιτούνται για τον έλεγχο της συνθήκης εξόδου του [19]. Για να

γίνει unroll σε ένα βρόχο, πρέπει να δημιουργηθούν f συνεχόμενα στιγμιότυπα επαναλήψεων του βρόχου στο σώμα του βρόχου, και αυξηθεί το βήμα του με τον ίδιο παράγοντα. Ο παράγοντας f ονομάζεται παράγοντας unrolling [1]. Εκτός από τη μείωση του χρόνου εκτέλεσης του βρόχου κατά παράγοντα f, το unrolling επιτυγχάνει και επαναχρησιμοποίηση καθώς πανομοιότυπες και συνεχόμενες τιμές εμφανίζονται πολλές φορές στο 'ξετυλιγμένο' (unrolled) σώμα του βρόχου.

Τα ακόλουθα τμήματα κώδικα, δείχνουν ένα loop στην αρχική του μορφή πριν εφαρμοστεί το unrolling και μετά την εφαρμογή του μετασχηματισμού με παράγοντα unrolling 4.

```
DO I = 0, N
     A(I) = B(I) + C(I)
ENDDO
```

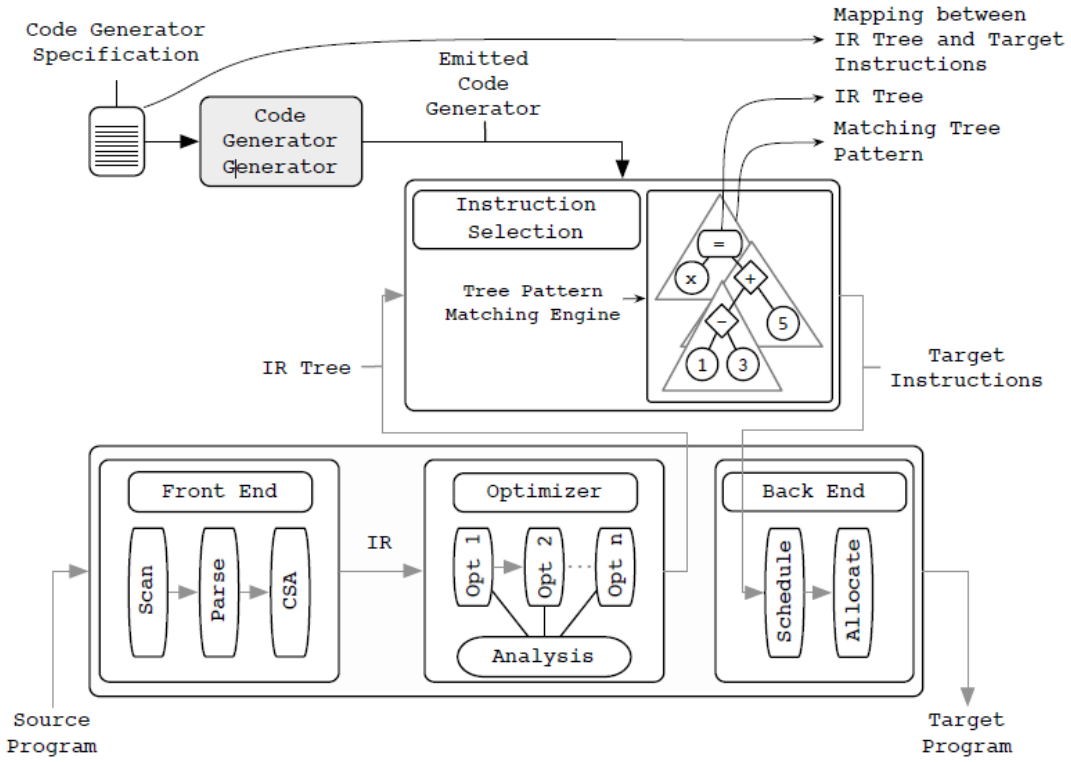
```
DO I = 0, N, 4
     A(I) = B(I) +C(I);
     A(I+1) = B(I+1)+C(I+1)
     A(I+2) = B(I+2)+C(I+2)
     A(I+3) = B(I+2)+C(I+2)
ENDDO
```

3 Γεννήτριες γεννητριών κώδικα (Code generator generators)

3.1 Γενικά

Η πρόοδος στην κατασκευή των μεταγλωττιστών, έχει δείξει ότι πολλές φάσεις μπορούν να αυτοματοποιηθούν κατά τη διάρκεια της διαδικασίας του σχεδιασμού και της υλοποίησης ενός μεταγλωττιστή. Η μεγάλη έρευνα που έχει διεξαχθεί για τις εξέχουσες περιοχές της λεξικογραφικής και συντακτικής ανάλυσης, έχει οδηγήσει στη δημιουργία εργαλείων που παράγουν λεξικογραφικούς και συντακτικούς αναλυτές με αυτόματο τρόπο, δεδομένης μιας συνοπτικής τεκμηρίωσης.

Οι τρέχουσες προσπάθειες επικεντρώνονται στην αυτοματοποίηση της παραγωγής γεννητριών κώδικα από προδιαγραφές, οι οποίες αντιστοιχίζουν μια ενδιάμεση δενδρική δομή σε σύνολο εντολών μηχανής [7]. Η παρακάτω εικόνα δείχνει πως παράγεται μια γεννήτρια κώδικα από μια τέτοια τεκμηρίωση, και πως συνδέεται με το back end ενός μεταγλωττιστή και εκτελεί επιλογή βέλτιστων εντολών, καλύπτοντας μια ενδιάμεση δενδρική δομή με πρότυπα δέντρου που δηλώνουν εντολές μηχανής. Κάθε πρότυπο δέντρου συσχετίζεται με ένα κόστος το οποίο συνεισφέρει στο συνολικό κόστος του παραγόμενου κώδικα. Ο βέλτιστος κώδικας επιτυγχάνεται με την επιλογή προτύπων με τέτοιο τρόπο ώστε τα συνολικά κόστη να γίνονται ελάχιστα.



Εικόνα 3.1.1 Αυτόματη παραγωγή του back end ενός μεταγλωττιστή

Γίνεται έρευνα στην περιοχή αυτοματοποίησης της κατασκευής γεννητριών κώδικα και παρόλο που πολλές ιδέες και προσεγγίσεις έχουν εξελιχθεί, οι γλώσσες γραμματικής τεκμηρίωσης και οι γεννήτριες γεννητριών κώδικα, δεν είναι τόσο εξελιγμένες όσο οι λεξικογραφικοί και γραμματικοί αναλυτές [7].

3.2 Ιστορική αναδρομή

3.2.1 Graham και Glanville - 1980

Μια από τις πρώτες μεθόδους παραγωγής γεννητριών κώδικα που αποδείχτηκε χρήσιμη, είναι η μέθοδος Graham-Glanville (1978). Σε αυτή τη μέθοδο, σειριακά πρότυπα δέντρων αναγνωρίζονται με τη χρήση ενός bottom-up shift-reduce parser. Η LR γραμματική παράγεται αυτόματα, από το σύνολο εντολών της μηχανής. Το μεγαλύτερο πρόβλημα σε αυτή την προσέγγιση, είναι ότι οι γραμματικές για αυτούς τους code-generators, είναι εξαιρετικά διφορούμενες [16].

3.2.2 Davidson και Fraser - 1984

Μια άλλη μέθοδος για επιλογή κώδικα βασισμένη στη βελτιστοποίηση κώδικα αναπτύχθηκε. Αυτή η μέθοδος δεν προσπαθεί να επιλέξει καλό κώδικα απευθείας από το δέντρο. Πρώτα αναπτύσσει τοπικά το δέντρο σε απλές εντολές μηχανής και στη συνέχεια ο παραγόμενος κώδικας βελτιστοποιείται αυξητικά με τη χρήση ενός συνόλου (declarative) βελτιστοποιήσεων. Αυτή η μέθοδος χρησιμοποιείται ακόμα και σήμερα από τον GCC (GNU Compiler Collection) [16].

3.2.3 TWIG - Aho, Ganapathi και Tjiang - 1989

Η προσέγγιση TWIG Aho et al. (1989) εισήγε τις αρχές ταύτισης προτύπων δέντρου και δυναμικού προγραμματισμού (tree pattern matching and dynamic programming). Η προσέγγιση TWIG χρησιμοποιεί ταυτοποίηση προτύπων βασισμένη σε πίνακα, που είναι πιο πολύπλοκη από την απευθείας παραγωγή κώδικα [16].

3.2.4 BURG - Fraser, Henry και Proebsting - 1992

Οι BURG Fraser et al (1992b) γεννήτριες γεννητριών-κώδικα βασίζονται στη θεωρία BURS (Bottom-Up Rewrite System) Nymeyer και Katoen (1997), έτσι ώστε να μετακινηθεί ο δυναμικός προγραμματισμός κατά το χρόνο μεταγλώττισης. Ο πίνακας παραγωγής των BURS είναι πιο πολύπλοκος, αλλά παράγουν βέλτιστο κώδικα σε σταθερό χρόνο για κάθε κόμβο. Το βασικό μειονέκτημά τους είναι ότι τα

κόστη πρέπει να είναι σταθερές, καθότι τα συστήματα που μεταφέρουν το δυναμικό προγραμματισμό στο compile time επιτρέπουν στα κόστη να περιλαμβάνουν αυθαίρετους υπολογισμούς. Η γεννήτρια είναι πιο πολύπλοκη, αλλά πολύ αποδοτική. Σε αυτή την κατηγορία εντάσσονται οι γεννήτριες Iburg, Jburg κ.α

3.3 Μηχανές BURG

Πολλές γεννήτριες γεννητριών κώδικα χρησιμοποιούν τεχνικές ταύτισης προτύπων δέντρου (tree pattern matching) και δυναμικό προγραμματισμό. Δέχονται πρότυπα δενδρικών δομών δεδομένων, και σχετικά κόστη μαζί με σημασιολογικές ενέργειες (semantic actions), οι οποίες για παράδειγμα, κατανέμουν καταχωρητές και παράγουν κώδικα. Παράγουν μηχανές ταύτισης δέντρων, που κάνουν δύο περάσματα σε κάθε ένα αντικείμενο-δέντρο. Το πρώτο πέρασμα είναι από κάτω προς τα πάνω (bottom up) και βρίσκει ένα σύνολο προτύπων που επικαλύπτουν το δέντρο με το ελάχιστο κόστος. Το δεύτερο πέρασμα εκτελεί τις σημασιολογικές ενέργειες που συσχετίζονται με τα πρότυπα ελάχιστου κόστους στους κόμβους που ταυτίστηκαν. Γεννήτριες γεννητριών κώδικα που βασίζονται σε αυτό το μοντέλο είναι οι BEG, TWIG, και BURG [8].

Οι BEG ταυτιστές (matchers) καθρεφτίζουν τα δενδρικά πρότυπα με τον ίδιο τρόπο που οι αναδρομικής – κατάβασης parsers, καθρεφτίζουν τη γραμματική εισόδου τους. Χρησιμοποιούν δυναμικό προγραμματισμό κατά τη φάση της μεταγλώττισης για να προσδιορίσουν μια ελάχιστου κόστους ταύτιση. Οι TWIG matchers χρησιμοποιούν μια παραλλαγή ταύτισης string βασισμένη σε πίνακα (table driven), η οποία προσδιορίζει όλα τα πιθανά ταιριάσματα την ίδια στιγμή. Αυτός ο αλγόριθμος είναι ασυμπτωτικά καλύτερος, από την προσέγγιση της δοκιμής κάθε πιθανής ταύτισης μιας κάθε φορά, αλλά το overhead είναι υψηλότερο. Όπως και οι BEG matchers, έτσι και οι TWIG χρησιμοποιούν δυναμικό προγραμματισμό κατά τη φάση της μεταγλώττισης για να προσδιορίσουν ένα ελάχιστου κόστους ταίριασμα [8].

3.4 Ταύτιση προτύπων δέντρου και δυναμικός προγραμματισμός

3.4.1 Δυναμικός προγραμματισμός σε μια μηχανή BURM

Ένα από τα δύσκολα προβλήματα που αντιμετωπίζει μια γεννήτρια κώδικα, είναι ότι υπάρχουν πολλές επιλογές που μπορεί να είναι σωστές. Για παράδειγμα, εάν η γεννήτρια κώδικα συναντήσει έναν κόμβο PLUS(identifier, integer_literal), δεν γνωρίζει αν πρόκειται για ακέραια πρόσθεση ή κινητής υποδιαστολής. Εάν η γεννήτρια δουλεύει με τρόπο από πάνω προς τα κάτω, τότε θα μπορούσε να χρησιμοποιήσει μια από τις ακόλουθες μεθόδους [11]:

- Εάν η έκφραση είναι της μορφής d = x + 1, (όπου το d είναι δεκαδικός και το x ακέραιος) τότε αυτή η ανάθεση αποτελεί έναν κόμβο που 'γνωρίζει' ότι αυτή είναι μια πρόσθεση κινητής υποδιαστολής. Η πληροφορία για αυτό 'ρέει' προς τα κάτω στο δέντρο, από τον κόμβο της ανάθεσης, μέχρι τη μετατροπή της έκφρασης στα δεξιά, σε αριθμό κινητής υποδιαστολής. Η πληροφορία δεν είναι απαραίτητο να ταξιδέψει μακριά επειδή οι γλώσσες προγραμματισμού είναι σχεδιασμένες έχοντας υπόψη τους αυτό το θέμα: το x + 1 μπορεί να προχωρήσει σαν ακέραια πρόσθεση, και να μετατραπεί σε δεκαδικό λίγο πριν ανατεθεί.

- Εάν πάλι η έκφραση είναι x + 1.0, τότε είναι ο κομβος της σταθεράς κινητής υποδιαστολής που 'γνωρίζει' ότι η έκφραση είναι κινητής υποδιαστολής: η πληροφορία αυτή ταξιδεύει προς τα πάνω στο δέντρο.

Ο δυναμικός προγραμματισμός είναι μια τεχνική επίλυσης προβλημάτων που εφαρμόζεται με μεγάλη επιτυχία σε τέτοιου είδους προβλήματα. Ας θεωρήσουμε καταρχήν τη γενική προσέγγιση 'Διαίρει και Βασίλευε' στην επίλυση προβλημάτων. Αυτή είναι μια από πάνω προς τα κάτω τεχνική που ξεκινάει από το αρχικό πρόβλημα σαν ολότητα (whole), και προχωράει αναδρομικά διαιρώντας το αρχικό πρόβλημα σε ανεξάρτητα υποπροβλήματα. Το μειονέκτημα αυτής της μεθόδου, είναι ότι η αναδρομική επίλυση των υποπροβλημάτων, μπορεί να οδηγήσει σε εκτέλεση των ίδιων υπολογισμών αρκετές φορές, επειδή αρκετά υποπροβλήματα

μπορεί να είναι πανομοιότυπα. Ο δυναμικός προγραμματισμός από την πλευρά του, υποδιαιρεί ένα πρόβλημα (για παράδειγμα η παραγωγή κώδικα από μια δομή ενδιάμεσης αναπαράστασης), σε έναν αριθμό από φάσεις που μπορούν να επιλυθούν ανεξάρτητα, και επιπλέον τα ενδιάμεσα αποτελέσματα απομνημονεύονται, με αποτέλεσμα να μην ξανα-υπολογίζονται [3]. Οι κόμβοι ενός δέντρου σχηματίζουν μια φυσική ιεραρχία που μπορεί αμέσως να προσαρμοστεί σε φάσεις (για παράδειγμα το αυτόνομο πρόβλημα που μπορεί να λυθεί στο επίπεδο του κόμβου PLUS) [11]. Οι πιθανές λύσεις μιας φάσης λέγονται καταστάσεις, και σε κάθε ζευγάρι φάσης/κατάστασης ανατίθεται ένα κόστος: αν υπάρχουν πολλαπλές πιθανές λύσεις, τότε η λύση με το μικρότερο κόστος επικρατεί.

Οι γεννήτριες κώδικα τύπου JBurg χρησιμοποιούν ένα κοινό σύνολο καταστάσεων που αντιστοιχεί στον τύπο του τελεστέου που βρίσκεται σε μια στοίβα κατά το χρόνο εκτέλεσης, ή σε μια τιμή σε έναν καταχωρητή, ή σε μια τρέχουσα έκφραση. Για παράδειγμα στο πρότυπο τύπου JBurg

int PLUS(int , constant_int)

οι καταστάσεις είναι int και constant_int.

3.4.2 Υλοποίηση της γεννήτριας

Μια γεννήτρια γεννήτριας κώδικα, είναι ένα συστατικό λογισμικού που παίρνει σαν είσοδο ένα αρχείο προδιαγραφών (μια BURG γραμματική δέντρου στην περίπτωση του JBURG), και παράγει μια ενότητα λογισμικού που συνδέεται με τον μεταγλωττιστή [16]. Αυτή η ενότητα (module) λογισμικού, διαβάζει ένα δέντρο ενδιάμεσης αναπαράστασης και παράγει κώδικα μηχανής σαν αποτέλεσμα. Μια γεννήτρια αυτού του τύπου πραγματοποιεί δύο σαρώσεις – περάσματα της ενδιάμεσης δενδρικής δομής. Το πρώτο πέρασμα είναι η διαδικασία label και αναθέτει ένα κόστος που αντιστοιχεί στο βέλτιστο πρότυπο εντολής, σε κάθε έναν κόμβο, με μια από κάτω προς τα πάνω σάρωση. Το δεύτερο πέρασμα είναι η διαδικασία reduce, και εκτελεί την ενέργεια που συσχετίζεται με το βέλτιστο πρότυπο εντολής που επιλέχθηκε για κάθε κόμβο, με μια από πάνω προς τα κάτω σάρωση. Το δεύτερο πέρασμα είναι γενικό και δεν εξαρτάται από τη γραμματική. Όπως φαίνεται και στο παραπάνω παράδειγμα γραμματικής, αρκετά μη τερματικά

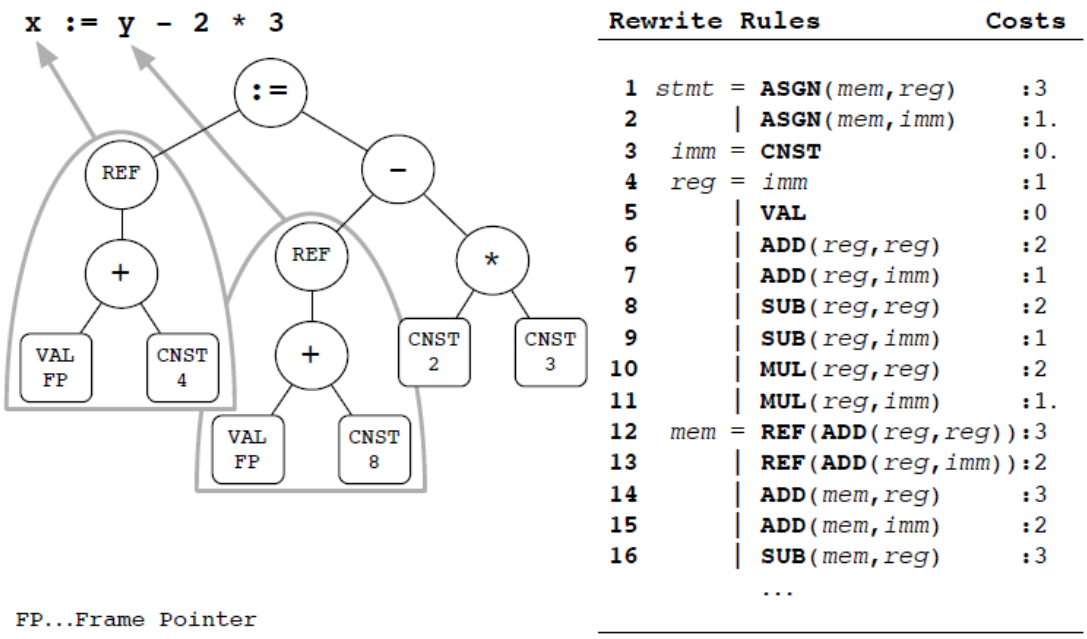
σύμβολα χρησιμοποιούνται για να προσδιορίσουν πως τα πρότυπα εντολών ταιριάζουν μαζί.

3.4.3 Η συνάρτηση label

Η συνάρτηση label είναι αυτή που κάνει το πέρασμα κατά το οποίο ανατίθενται ετικέτες στο δέντρο. Αυτή η συνάρτηση παίρνει έναν κόμβο της ενδιάμεσης δενδρικής δομής σαν παράμετρο, και ο στόχος της είναι να αναθέσει σε αυτό τον κόμβο ένα κόστος που βασίζεται στο πρότυπο της βέλτιστης εντολής. Για την αντιμετώπιση της πολυπλοκότητας που προκύπτει από τη χρήση αρκετών μη τερματικών συμβόλων στη BURG γραμματική, το βέλτιστο κόστος του κόμβου, πρέπει να διατηρείται για κάθε ένα μη τερματικό σύμβολο. Έτσι εισάγεται ένας πίνακας από κόστη στη δενδρική δομή δεδομένων που χρησιμοποιείται για τους κόμβους. Επιπλέον τα επιλεγμένα πρότυπα εντολών απομνημονεύονται και αυτά, έτσι ώστε το πέρασμα της μείωσης (reduce) να εκτελεί την κατάλληλη ενέργεια. Αυτό οδηγεί στην εισαγωγή ακόμα ενός πίνακα στο δένδρο, ο οποίος κρατάει για κάθε ένα μη τερματικό σύμβολο αυτή την ενέργεια που συσχετίζεται με το πρότυπο της βέλτιστης εντολής [16].

3.4.4 Υπολογίζοντας τη βέλτιστη ακολουθία εντολών

Έστω ότι έχουμε την ενδιάμεση αναπαράσταση για την έκφραση ανάθεσης x = y – 2 * 3. Τα υποδέντρα τα οποία υπολογίζουν τις θέσεις των μεταβλητών είναι επισημασμένα και δείχνουν στην αντίστοιχη μεταβλητή. Ένας πίνακας παρουσιάζει όλους τους κανόνες και τα κόστη τους για δεδομένη δομή ενδιάμεσης αναπαράστασης (δεξί τμήμα της παρακάτω εικόνας). Ένας κανόνας αντικατάστασης αποτελείται από ένα μη τερματικό σύμβολο που προκύπτει από ένα πρότυπο δέντρου (μη τερματικά υποδηλώνουν οι καταστάσεις immediate, register και memory στο παράδειγμα αυτό) [7].

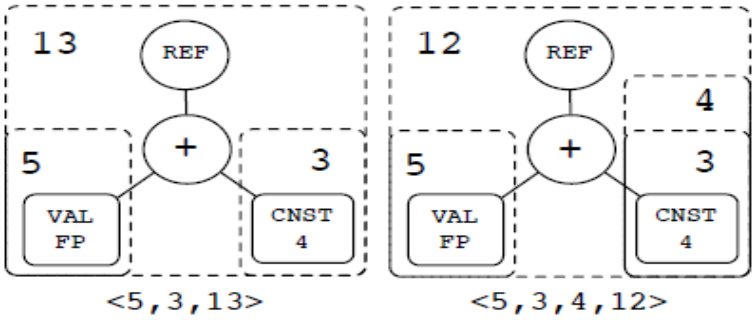


Εικόνα 3.4.4.1 Δενδρική δομή ενδιάμεσης αναπαράστασης για μια εντολή ανάθεσης με τους αντίστοιχους κανόνες αντικατάστασης

Το υποδέντρο REG(ADD(VAL FP, CNST 4)) που υπολογίζει τη διεύθυνση της μεταβλητής x στην παραπάνω εικόνα, είναι αυτό που μας ενδιαφέρει. Κοιτάζοντας τους κανόνες βλέπουμε ότι μπορούμε να καλύψουμε τον κόμβο VAL FP με τον κανόνα 5, δηλ. reg = VAL. Στη συνέχεια μπορούμε να εφαρμόσουμε τον κανόνα αντικατάστασης 3 για να καλύψουμε τον κόμβο CNST 4. Σαν συνέπεια της εφαρμογής των κανόνων αυτών ο κανόνας 13 reg = REF(ADD(reg, imm)), ταυτίζεται με το παραγόμενο υποδέντρο.

Με μια πιο προσεκτική ματιά όμως μπορούμε να δούμε ότι αυτή δεν είναι η μόνη ακολουθία από κανόνες που μπορούν να εφαρμοστούν. Στην παρακάτω εικόνα φαίνεται ότι υπάρχει και άλλη μια ακολουθία κανόνων που καλύπτει το υποδέντρο. Η πρώτη είναι η <5, 3, 13> που έχει κόστος δύο, και η δεύτερη <5, 3, 4, 12> που έχει κόστος τέσσερα. Μια γραμματική ενός πραγματικού μεταγλωττιστή συνήθως περιλαμβάνει πολύ περισσότερες πιθανές ακολουθίες κανόνων. Έτσι χρειάζεται ένας επαρκής αλγόριθμος που να ανακαλύπτει όλες τις πιθανές βέλτιστες ακολουθίες κανόνων. Η λύση αυτού του προβλήματος βασίζεται στο δυναμικό προγραμματισμό. Ο δυναμικός προγραμματισμός διαιρεί αναδρομικά ολόκληρο το πρόβλημα σε έναν αριθμό από υποπροβλήματα που μπορούν να

επιλυθούν ανεξάρτητα. Οι λύσεις των υποπροβλημάτων κατασκευάζονται αυξητικά από μικρότερα υποπροβλήματα και αποθηκεύονται για την αποφυγή των επανυπολογισμών [7].

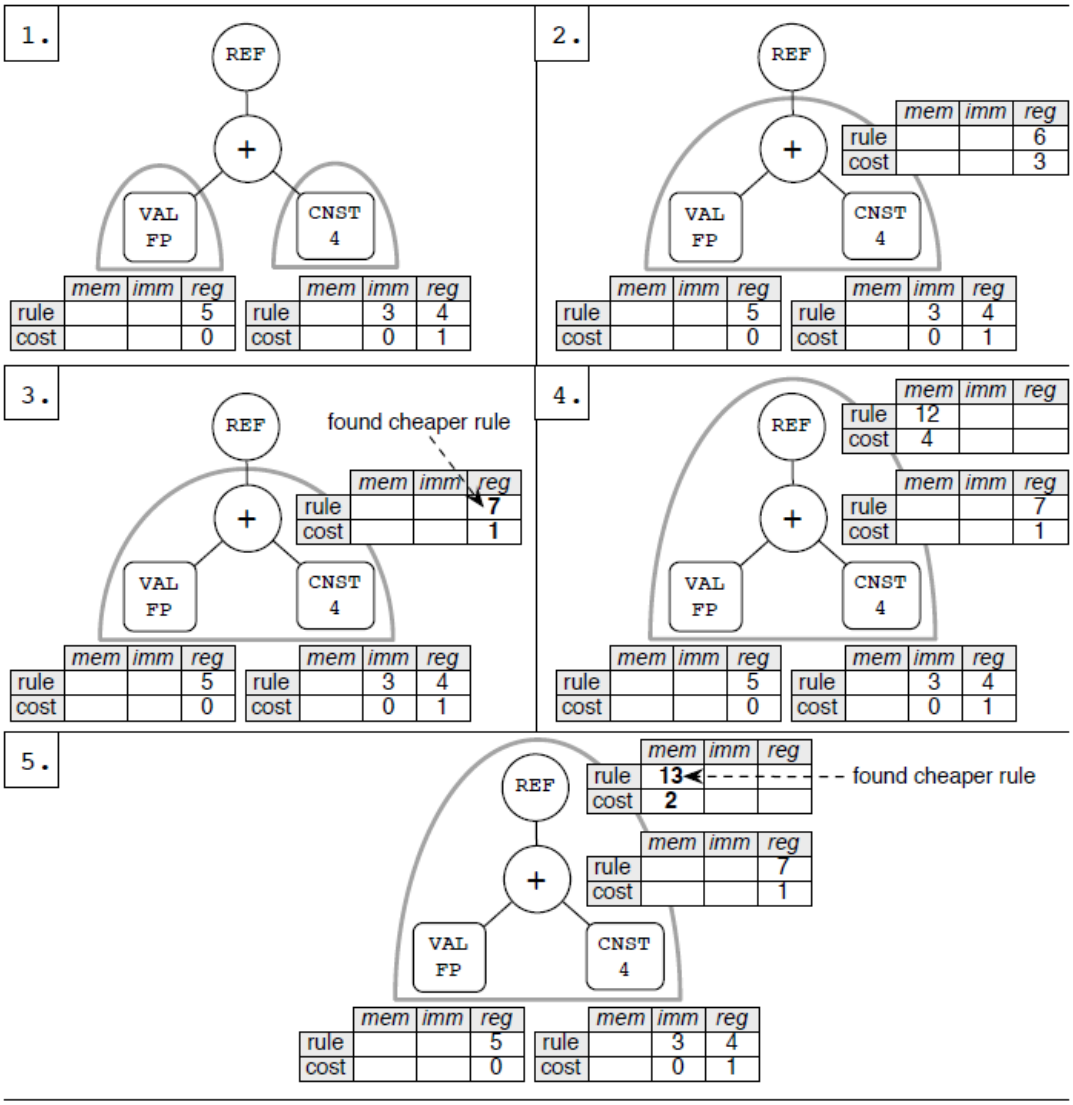


Εικόνα 3.4.4.2 Πιθανά ταιριάσματα κανόνων αντικατάστασης για ένα τμήμα ενδιάμεσης αναπαράστασης

Στην περίπτωσή μας το πρόβλημα της παραγωγής μιας βέλτιστης ακολουθίας κανόνων για μια δενδρική ενδιάμεση αναπαράσταση T, διαιρείται σε υποπροβλήματα εύρεσης βέλτιστης ακολουθίας κανόνων για υποδέντρα του T. Ένας κανόνας γράφεται στον κόμβο ρίζα του κάθε υποδέντρου αν και μόνο αν δεν υπάρχει άλλος φθηνότερος κανόνας που να παράγει το ίδιο μη τερματικό.

Η παρακάτω εικόνα δείχνει πως ένα IR δέντρο καλύπτεται με βέλτιστο τρόπο από δενδρικά πρότυπα με τη χρήση δυναμικού προγραμματισμού. Η διαδικασία label δουλεύει από κάτω προς τα πάνω (bottom-up) [7]. Στο πρώτο βήμα ανατίθενται ετικέτες στους κόμβους VAL και CNST. Ο κανόνας 5 ταιριάζει στον κομβο VAL και εγγράφεται σε έναν πίνακα που κωδικοποιεί όλα τα συστατικά ενός κανόνα, δηλ. τον αριθμό, το κόστος του, και το μη τερματικό σύμβολο που παράγει. Το δενδρικό πρότυπο 3 ταιριάζει στον κόμβο CNST και εγγράφεται και αυτό. Να σημειώσουμε ότι εφόσον ο κόμβος CNST παράγει ένα imm μη τερματικό σύμβολο, τώρα ταιριάζει επίσης και ο κανόνας 4. Με αυτό τον τρόπο ένα πρότυπο που ταιριάζει μπορει να προκαλέσει μια αλυσίδα από επακόλουθα ταιριάσματα. Στο επόμενο βήμα ανατίθεται ετικέτα στον κόμβο ADD και ο πρώτος κανόνας που ταιριάζει είναι ο 6. Απαιτεί οι κόμβοι – παιδιά του ADD, να παράγουν μη τερματικά reg.

Τα κόστη από τα απαιτούμενα μη τερματικά που προέκυψαν από τα παιδιά του κόμβου ADD (κανόνας 6), αλλά και το κόστος του ίδιου του κόμβου (κανόνας 6), συμβάλλουν στο καθολικό κόστος που προήλθε από τον κανόνα 6 εφαρμοζόμενο στον κόμβο ADD και έτσι αυτό το κόστος εγγράφεται. Στο τρίτο βήμα όμως, φαίνεται ότι ένας άλλος κανόνας ο 7, επίσης ταιριάζει στο ADD. Ο 7 τώρα αντικαθιστά τον 6 στον πίνακα για το ADD καθώς το συνολικό του κόστος είναι μικρότερο από αυτό του 6. Με άλλα λόγια ο κανόνας 7 είναι προτιμότερος του 6. Καθώς δεν υπάρχει άλλος κανόνας που να ταιριάζει στο ADD, μπορούμε να προχωρήσουμε στον κόμβο REF. Η διαδικασία επιλογής ενός βέλτιστου δενδρικού προτύπου για τον REF είναι ίδια με την προηγούμενη που περιγράφηκε.



Εικόνα 3.4.4.3 Bottom-Up βέλτιστη επιλογή εντολών με χρήση δυναμικού προγραμματισμού

3.4.5 Παράγοντας τη βέλτιστη ακολουθία εντολών

Καθώς το πρώτο από κάτω προς τα πάνω (bottom-up) πέρασμα της ενδιάμεσης αναπαράστασης έχει ολοκληρωθεί, παράγοντας μια ακολουθία βέλτιστων κανόνων σαν αποτελέσματα, παράγονται σημασιολογικές ενέργειες που συσχετίζονται με τους κανόνες [7]. Επειδή το δέντρο διασχίζεται από πάνω προς τα κάτω κατά τη διάρκεια αυτής της φάσης, σημασιολογικές ενέργειες που δηλώνονται sn, μπορεί να συμπεριληφθούν οπουδήποτε μέσα στους κανόνες όπως φαίνεται παρακάτω:

```
goal = s1 assign s2 (4.1)
assign = s1 ASGN s2 (s3 reg s4, s5 reg s6) (4.2)
reg = s1 REF s2 (s3 ADD s4(s5 reg s6, s7 reg s8)s9)s10 (4.3)
```

Μόλις ένα υποδέντρο ταυτιστεί με ένα πρότυπο, η επεξεργασία συνεχίζεται ως ακολούθως σύμφωνα με τον τύπο του κανόνα αντικατάστασης:

1. Βασικός κανόνας: Οι κανόνες 4.2 και 4.3 είναι βασικοί. Οι βασικοί κανόνες αποτελούνται από τελεστές (τερματικά) και τελεστέους (μη τερματικά). Πρώτα παράγονται οι σημασιολογικές ενέργειες πριν και μετά τον τελεστή. Στην περίπτωσή μας αυτό θα ήταν το s1 για τον τελεστή REF. Η παρένθεση, δηλώνει ότι ένας τελεστής έχει παιδιά κόμβους (τελεστέους), όπως στην περίπτωση των REF, ASGN, και ADD. Πριν την αναδρομική κατάβαση για την επεξεργασία ενός κόμβου-παιδιού τερματικού ή μή, ορίζονται οι σημασιολογικές ενέργειες πριν γίνει η επεξεργασία των αντίστοιχων τερματικών ή μη τερματικών. Μετά την επιστροφή από την αναδρομή που ήταν υπεύθυνη για την επεξεργασία του δεύτερου τελεστή (reg) του κόμβου ADD, εκτελείται η σημασιολογική ενέργεια s8.
2. Αλυσιδωτός κανόνας: Ο κανόνας 4.1 αναπαριστά έναν αλυσιδωτό κανόνα. Ένας αλυσιδωτός κανόνας είναι ένας κανόνας του οποίου το πρότυπο, είναι ένα μη τερματικό (τελεστέος). Ξανά η σημασιολογική ενέργεια s1 στον κανόνα 4.1 εκτελείται πριν την επεξεργασία του μη τερματικού assign που

ακολουθείται από την εκτέλεση της σημασιολογικής ενέργειας που ακολουθεί - s2.

3.5 Γλώσσα περιγραφής παραγωγής κώδικα (Code Generation Description Language)

3.5.1 Γενική δομή

Δομικά στοιχεία μιας γλώσσας περιγραφής για την παραγωγή κώδικα είναι τα παρακάτω [7]:

- Τερματικά, που αναφέρονται επίσης και ως τελεστές και γράφονται με κεφαλαία γράμματα
- Μη τερματικά, που αναφέρονται επίσης και ως τελεστέοι και γράφονται με πεζά γράμματα
- Δενδρικά πρότυπα που ορίζουν πιθανές αντικαταστάσεις ενός μη τερματικού συμβόλου και διαχωρίζονται με τον χαρακτήρα διασωλήνωσης ΄|’.
- Σημασιολογικές ενέργειες που βρίσκονται ανάμεσα στα (. και .), δηλώνοντας ένα τμήμα κώδικα που γράφεται στη γλώσσα παραγωγής της γεννήτριας κώδικα.
- Κόστη τα οποία προσδιορίζονται μετά το colon στο τέλος κάθε δενδρικού προτύπου. Τα κόστη μπορεί να είναι είτε σταθερές, ή αυθαίρετες εκφράσεις γραμμένες στη γλώσσα παραγωγής (target language).
- Ο συνδυασμός ενός δενδρικού προτύπου, του κόστους του και των μη τερματικών στα οποία οδηγεί, ονομάζεται *κανόνας αντικατάστασης.*

```
7     rules -- rewrite rules
8     stmts = stmt [ stmts ] : 0 . -- start rule
9     stmt = ASGN ( reg ,NUM) ( . . . . . ) : 1
10          | ASGN ( reg , reg ) ( . . . . . ) : 2 .
11    reg = NUM ( . . . . . ) : 1
12          | ADD ( reg , reg ) ( . . . . . ) : 2
13          | SUB ( reg , reg ) ( . . . . . ) : 2 .
14    end
```

3.5.2 Κανόνες αντικατάστασης

Ένας κανόνας αντικατάστασης, ορίζει ένα ή περισσότερα δενδρικά πρότυπα μιας ενδιάμεσης αναπαράστασης [7]. Αποτελείται από δύο πλευρές, οι οποίες διαχωρίζονται από το σύμβολο '='. Η δεξιά πλευρά περιλαμβάνει δενδρικά πρότυπα σε πλήρως παρενθεσιοποιημένη μορφή, και η αριστερή πλευρά ορίζει ένα μη τερματικό, που αναπαριστά το υποδέντρο που ταυτίστηκε με τη δεξιά πλευρά. Τα μη τερματικά μπορούν με τη σειρά τους να χρησιμοποιηθούν σαν τελεστέοι σε άλλα δενδρικά πρότυπα και συνήθως αναπαριστούν κλάσεις αποθήκευσης, ή μορφές διευθυνσιοδότησης, που παρέχονται από την εκάστοτε αρχιτεκτονική. Ο πρώτος κανόνας αντικατάστασης που ορίζεται στο τμήμα των κανόνων είναι ο κανόνας έναρξης. Πρέπει να ταυτιστεί με τον κόμβο ρίζα ενός δέντρου – ενδιάμεσης αναπαράστασης. Όλοι οι επακόλουθοι κανόνες αντικατάστασης μπορούν να οριστούν με οποιαδήποτε σειρά. Ο κάθε κανόνας προσδιορίζεται έτσι ώστε να ταυτίζεται με τουλάχιστον ένα δενδρικό πρότυπο. Τα δενδρικά πρότυπα μπορούν να δίνονται σε οποιαδήποτε σειρά και το κάθε ένα πρέπει να έχει ένα κόστος.

Τα δενδρικά πρότυπα επίσης μπορούν να ταιριάζουν με συνδεδεμένες αλυσίδες υποδένδρων όπως φαίνεται στο παραπάνω τμήμα τεκμηρίωσης στη γραμμή 8. Το παράδειγμα δείχνει πως μια αλυσίδα από κόμβους stmt, μπορεί να ταυτιστεί από τον κανόνα stmts = stmt[stmts], όπου οι αγκύλες δηλώνουν ότι τα μη τερματικά σύμβολα stmts είναι προαιρετικά.

3.5.3 Κόστη δενδρικών προτύπων

Τα κόστη υπολογίζονται κατά το πρώτο από κάτω προς τα πάνω πέρασμα της ενδιάμεσης αναπαράστασης και κάθε δενδρικό πρότυπο πρέπει να συσχετιστεί με το κόστος του. Τα κόστη δεν είναι απαραίτητο να είναι πάντα σταθερές. Είναι δυνατό να οριστούν κόστη ως αυθαίρετες εκφράσεις γραμμένες στη γλώσσα της αρχιτεκτονικής που μας ενδιαφέρει, μέσα στα σύμβολα (. και .) [7]. Τέτοιες εκφράσεις πρέπει να οδηγούν σε ακέραιους αριθμούς. Επιπλέον μόνο μεταβλητές και συναρτήσεις που ορίζονται στο τμήμα των δηλώσεων και παραγωγής κώδικα

της τεκμηρίωσης, – όπως ο κόμβος ρίζα του σχετικού δενδρικού προτύπου –, μπορούν να είναι μέσα στην εμβέλεια τέτοιων εκφράσεων.

3.5.4 Σημασιολογικές ενέργειες

Μια σημασιολογική ενέργεια ορίζεται ανάμεσα στα (. και .). Αποτελεί ένα τμήμα κώδικα γραμμένο σε γλώσσα της αρχιτεκτονικής που μας ενδιαφέρει. Οι σημασιολογικές ενέργειες εκτελούνται από τη γεννήτρια κώδικα που έχει παραχθεί στο σημείο που έχει προσδιοριστεί για αυτές, κατά το δεύτερο από πάνω προς τα κάτω πέρασμα της ενδιάμεσης αναπαράστασης [7]. Οι κόμβοι που ταυτίζονται με δενδρικά πρότυπα, είναι προσβάσιμοι μέσω συνδέσεων στις σημασιολογικές ενέργειες. Μια σύνδεση μπορεί να οριστεί για κάθε ένα τερματικό ή μή σε ένα δενδρικό πρότυπο, και η εμβέλειά της εκτείνεται από τη δήλωσή της μέχρι το τέλος του δενδρικού προτύπου. Ο κώδικας μέσα σε κάθε σημασιολογική ενέργεια μπορεί επίσης να προσπελάσει μεταβλητές και μεθόδους που ορίζονται στο τμήμα δηλώσεων της τεκμηρίωσης. Στο παρακάτω τμήμα κώδικα φαίνεται ένα παράδειγμα τεκμηρίωσης γεννήτριας κώδικα, που δείχνει πως χτίζεται μια λίστα από εντολές μηχανής για συνδεδεμένες εκφράσεις υποδένδρων.

```
1     generator -- import statements
2           ( . import java.util.LinkedList;
3           import java.util.List;
4           import code.codegen.Code; . )
5     declarations -- general declarations
6           ( . List <String > code = new LinkedList(); . )
7     operators
8           NUM( . E_CNST . ) , ASGN( . E_ASGN . ) , ADD( .
            E_ADD . ) ,
9           SUB( . E_SUB . ) , MUL( . E_MUL . )
10    rules -- rewrite rules
11          stmts = stmt [ stmts ] : 0 . -- start rule
12          stmt = ASGN ( reg ,NUM) ( . . . . . ) : 1
13                | ASGN ( reg , reg ) ( . . . . . ) : 2 .
14          reg = NUM a ( . a.result = Code.getReg();
15                code.add("loadI "+
16                      a.val +","+
17                      a.result); . ) : 1
18                | ADD a ( reg b , reg c )
19                      ( . a.result = Code.getReg();
20                            code.add("add "+
21                            b.result +","+
```

```
22                          c.result +","+
23                          a.result);
24                    Code.freeReg(b.result);
25                    Code.freeReg(c.result); . ) : 2
26              | SUB a ( reg c , reg c ) ( . . . . . ) : 2
27              | MUL a ( reg b , reg c ) ( . . . . . ) : 4 .
28   end
```

3.5.5 Χαρακτηριστικά

Ο κανόνας αντικατάστασης reg = ADD(reg, reg) παράγει ένα μη τερματικό reg, ταυτίζοντας δυαδικά υποδένδρα που έχουν έναν κόμβο – ρίζα ADD ο οποίος αναμένει οι κόμβοι παιδιά του να παράξουν μη τερματικά reg. Δεν είναι ακόμα ξεκάθαρο όμως πως μη τερματικά μπορούν να παράγουν τιμές και πως να τις προσπελάσουν όπως δηλώνεται στις σημασιολογικές ενέργειες [7].

Η πλήρης τυποποίηση της γραμματικής μιας γεννήτριας, δίνει λύση στο παραπάνω πρόβλημα. Τοποθετούνται σημειώσεις σε μη τερματικά με τα χαρακτηριστικά που δηλώνουν τις τιμές που παράγουν. Έτσι είναι δυνατό να κωδικοποιηθούν άμεσα πληροφορίες σχετικά με τις τιμές που παράγονται από μη τερματικά μέσα στην τεκμηρίωση της γραμματικής. Ένα τέτοιο παράδειγμα είναι το παρακάτω:

```
reg<.out Reg r0.> = ADD (reg<.out Reg r1.>, reg<.out Reg r2.>)
```

Γίνεται διαχωρισμός ανάμεσα σε επίσημα χαρακτηριστικά που ορίζονται στη δήλωση των μη τερματικών στο αριστερό τμήμα του δενδρικού προτύπου, και στα πραγματικά χαρακτηριστικά που ορίζονται με την εμφάνιση των μη τερματικών μέσα στα δενδρικά πρότυπα. Καθώς η εμβέλεια ενός τυπικού χαρακτηριστικού βρίσκεται πάνω από όλα τα δενδρικά πρότυπα που ορίζουν μη τερματικά, η εμβέλεια ενός πραγματικού χαρακτηριστικού βρίσκεται από το σημείο της δήλωσής του μέχρι το τέλος του προτύπου. Έτσι ο προηγούμενος κανόνας αντικατάστασης ορίζει ένα μη τερματικό reg που παράγει μια τιμή τύπου Reg που αποθηκεύεται στη μεταβλητή r0, ορίζοντας το επίσημο χαρακτηριστικό <.out Reg r0.>. Μέχρι τώρα

αναφέρθηκε η δυνατότητα να οριστούν χαρακτηριστικά *εξόδου* από μη τερματικά, αλλά είναι επίσης δυνατό να οριστούν και χαρακτηριστικά *εισόδου* [7].

Το παρακάτω παράδειγμα εκτελεί τις ίδιες ενέργειες με την προηγούμενη παράθεση, αλλά το κάνει χρησιμοποιώντας χαρακτηριστικά εξόδου. Επίσης δείχνει πως ορίζονται τα χαρακτηριστικά εξόδου, και πως προσπελάζονται από τις σημασιολογικές ενέργειες.

```
1     generator -- import statements
2           ( . import java.util.LinkedList;
3           import java.util.List;
4           import code.codegen.Code;
5           import code.codegen.Reg; . )
6     declarations -- general declarations
7           ( . List <String > code = new LinkedList(); . )
8     operators
9           NUM( . E_CNST . ) , ASGN( . E_ASGN . ) , ADD( .
            E_ADD . ) ,
10          SUB( . E_SUB . ) , MUL( . E_MUL . )
11    rules -- rewrite rules
12          stmts = stmt [ stmts ] : 0 . -- start rule
13          stmt = ASGN ( reg<.out Reg b.>,NUM) ( . . . . . ) :1
14                | ASGN(reg<.out Reg b.>, reg )( . . . . . ):2.
15          reg<.out Reg r.>
16                ( . r = Code.getReg(); . ) -- result register
17                = NUM a
18                ( . code.add("loadI "+ a.val +","+ r); . ) : 1
19                | ADD ( reg<.out Reg b.>, reg<.out Reg c.>)
20                      ( . code.add("add "+ r +","+ b +","+ c);
21                      Code.freeReg(b); Code.freeReg(c); . ) : 2
22
23
24    end
```

Παρακάτω δίνεται ένα κομμάτι γραμματικής τεκμηρίωσης της γεννήτριας γεννητριών κώδικα JBURG, που περιλαμβάνει τα χαρακτηριστικά που είδαμε στις παραπάνω ενότητες:

```
1  constant_int = PLUS( constant_int i1, constant_int i2 ):0
2  {
3    return new Integer ( i1.intValue() + i2.intValue() );
4  }
5
6  integer_value = PLUS(integer_value r1, integer_value r2): 1
7  {
8    r1.append(r2);
```

```
9     r1.append ( new IADD() );
10
11    return r1;
12  }
13
14  integer_value = MINUS(integer_value r1, integer_value r2):1
15  {
16    r1.append(r2);
17    r1.append ( new ISUB() );
18    return r1;
19  }
```

Οι παραπάνω τρεις κανόνες αντικατάστασης, ορίζουν τι συμβαίνει όταν η μηχανή συναντήσει τους κόμβους PLUS και MINUS και μάλιστα για τον PLUS ορίζονται δύο κανόνες, όπου ο ένας έχει κόστος 0 και ο άλλος κόστος 1. Το `integer_value` στο αριστερό μέρος των κανόνων όπου εμφανίζεται είναι τερματικό σύμβολο, ενώ το `constant_int` όχι. Μέσα στις παρενθέσεις ορίζεται επίσης, τι κόμβους παιδιά αναμένει ο κανόνας να έχουν οι κόμβοι PLUS και MINUS αντίστοιχα: ο κόμβος PLUS μπορεί να βρεθεί να έχει κόμβους – παιδιά τύπου απλής σταθεράς (constant_int) ή κόμβους τύπου τάξης ακέραιου αριθμού (Integer). Οι σημασιολογικές ενέργειες ορίζονται μέσα στις αγκύλες {} και θα εκτελεστούν από τη μηχανή κατά τη δεύτερη αναδρομική σάρωση του δέντρου.

4 GP – GPUs

4.1 Γενικά

Η μονάδα επεξεργασίας γραφικών (Graphics Processing Unit – GPU) στις σημερινές κάρτες γραφικών, έχει εξελιχθεί σε έναν εξαιρετικά δυνατό και ευέλικτο επεξεργαστή. Οι GPUs προσφέρουν ανώτερο εύρος μνήμης και υπολογιστική ισχύ κάνοντας έτσι ενδιαφέρουσα τη χρήση τους και πέρα από το πεδίο των γραφικών υπολογιστών [26]. Ο όρος GPGPU προέρχεται από τους όρους ‘General Purpose Computation on GPUs’ (Υπολογισμοί Γενικού Σκοπού σε GPU). Έρευνα έχει δείξει ότι η εκμετάλλευση μιας GPU μπορεί να επιταχύνει την επίλυση προβλημάτων που δεν

έχουν να κάνουν με γραφικά, πολύ περισσότερο από μια CPU, και αρκετές γλώσσες έχουν αναπτυχθεί, ώστε να κάνουν προσβάσιμη αυτή την υπολογιστική δύναμη.

Σημαντικά εμπόδια υπάρχουν ωστόσο για τον προγραμματιστή που θέλει να χρησιμοποιήσει τη δύναμη μιας GPU. Αυτά τα ολοκληρωμένα κυκλώματα έχουν σχεδιαστεί για την ανάπτυξη παιχνιδιών, το προγραμματιστικό μοντέλο είναι ασυνήθιστο, και οι αρχιτεκτονικές είναι σε μεγάλο βαθμό άγνωστες [26].

Η CUDA της NVIDIA είναι ένα νέο σχετικά σύστημα γενικού σκοπού υπολογιστικής σε GPU. Η CUDA είναι βασισμένη σε ένα νέο προγραμματιστικό API, το οποίο είναι εξολοκλήρου διαχωρισμένο από τον οδηγό των γραφικών. Χρησιμοποιεί τη γλώσσα C με επεκτάσεις (extensions) και εκθέτει νέα χαρακτηριστικά του υλικού που δεν είναι διαθέσιμα στο OpenGL ή στο Direct3D. Το πιο σημαντικό από αυτά τα νέα χαρακτηριστικά είναι η διαμοιραζόμενη μνήμη, η οποία μπορεί να βελτιώσει θεαματικά την απόδοση εφαρμογών που βασίζονται στο εύρος μνήμης, ένα αυθαίρετο μοντέλο ανάκτησης/αποθήκευσης από/στη μνήμη, κάνοντας έτσι δυνατή την υλοποίηση αλγορίθμων που πρωτύτερα ήταν αδύνατη η ανάπτυξή τους σε GPU. Θα αναφερθούμε στην CUDA αναλυτικότερα στο επόμενο κεφάλαιο.

4.2 Εξέλιξη των GPUs

Οι πρώτες GPUs εμφανίστηκαν το 1981 από την IBM και αποτελούνταν από 16KB μνήμης και ένα RAMDAC [25]. Απουσίαζε οποιαδήποτε δυνατότητα επεξεργασίας και rasterization των pixels (που έπρεπε να γίνουν και πάλι στην CPU) και χρησίμευε ως ένας ενδιάμεσος buffer πριν την απεικόνιση στην οθόνη. Οι GPUs με δυνατότητες επεξεργασίας εμφανίστηκαν το 1984, ενσωματώνοντας σε ένα ολοκληρωμένο ρουτίνες επεξεργασίας και απεικόνισης/rasterization, περιορίζονταν όμως μόνο σε δισδιάστατα γραφικά. Η απαίτηση για 3D παιχνίδια και εφαρμογές (CAD κ.ά.) οδήγησε στην εξέλιξη των πρώτων σύγχρονων GPUs στης αρχές του 1990, που έδιναν την δυνατότητα επιτάχυνσης των απαιτητικών τρισδιάστατων γραφικών με ξεχωριστές κάρτες γραφικών από τις οποίες απουσίαζαν οι δισδιάστατες δυνατότητες. Μέχρι τα τέλη της δεκαετίας, οι 2D και 3D δυνατότητες επεξεργασίας και επιτάχυνσης των γραφικών συγχωνεύθηκαν και άρχισαν να προστίθενται στο

ίδιο ολοκληρωμένο περισσότερες του ενός pipelines επεξεργασίας (texture pipelining, texture filtering), προάγγελοι των σύγχρονων πολυπύρηνων σχεδιάσεων. Το 2001 (Nvidia Geforce3) εμφανίστηκαν οι πρώτες κάρτες που επέτρεπαν τον προγραμματισμό των vertex και pixel shaders, υλοποιώντας ουσιαστικά μία παράλληλη αρχιτεκτονική όπου κάθε vertex και pixel επεξεργαζόταν ανεξάρτητα από τα υπόλοιπα, ανοίγοντας έτσι το δρόμο για τις GPUs γενικού σκοπού (GPGPUs - General Purpose GPUs) [25].

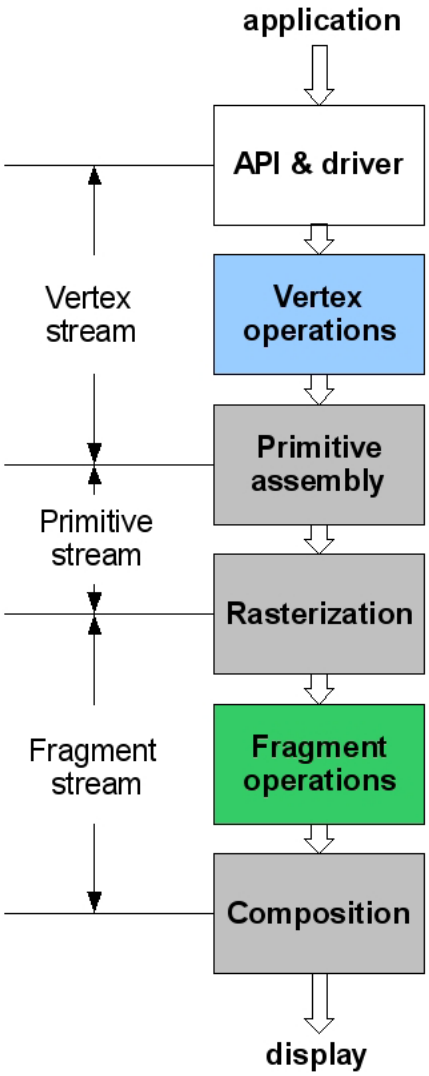
4.3 Αρχιτεκτονική των GPUs

Τα στάδια επεξεργασίας γραφικών τα οποία υλοποιεί μία σύγχρονη GPU είναι:

- Vertex operations: δημιουργούνται τα vertices (ακμές των αντικειμένων) και υπολογίζονται τα χρώματα και οι φωτισμοί (pixel shading).
- Σχηματισμός τριγώνων από τα vertices
- Rasterization: Υπολογίζεται ποια τρίγωνα είναι ορατά από την γωνία θέασης του παρατηρητή. Κάθε τρίγωνο που είναι ορατό δημιουργεί ένα ”fragment” σε κάθε εικονοστοιχείο της οθόνης που καλύπτει. Το τελικό χρώμα του εικονοστοιχείου υπολογίζεται από τον συνδυασμό όλων των fragments του συγκεκριμένου εικονοστοιχείου.
- Fragment operations: Στα fragments προστίθενται και πληροφορίες από το περιβάλλον, όπως αντανακλάσεις και εφαρμόζονται διάφορες υφές (textures) ώστε να υπολογιστεί το τελικό χρώμα. Υπολογιστικά είναι το πλέον απαιτητικό μέρος και υψηλά παραλληλίσιμο.
- Composition: Σχηματίζεται η τελική δισδιάστατη εικόνα από τα fragments, με ένα χρώμα ανά pixel.

Η μετάβαση στις προγραμματιζόμενες μονάδες vertex και fragment επέτρεψε τον ορισμό από τον προγραμματιστή, για παράδειγμα, του αλγορίθμου που ελέγχει το χρώμα σε κάθε vertex ανάλογα με τις ιδιότητες του και τον φωτισμό του

περιβάλλοντος, σε αντίθεση με την προηγούμενη αρχιτεκτονική όπου ο προγραμματιστής μπορούσε να επηρεάσει μόνο την θέση του vertex και τις πηγές φωτισμού. Η συνεχής αύξηση των δυνατοτήτων προγραμματισμού αυτών



Εικόνα 4.3.1 Pipeline επεξεργασίας γραφικών [25]

των σταδίων οδήγησε στα σημερινά ενοποιημένα μοντέλα σκίασης (Unified Shader Model), όπου τα vertex και fragment shaders έχουν ενοποιηθεί και αποτελούν μία και μοναδική πλήρως προγραμματιζόμενη μονάδα επεξεργασίας. Αυτό οδήγησε και στην λύση ενός κύριου προβλήματος των καρτών επεξεργασίας γραφικών σχετικά με τη δυνατότητα σωστής κατανομής του φόρτου (ανάλογα με τις απαιτήσεις της κάθε εφαρμογής) στα διαφορετικά τμήματα της GPU και του προβλήματος της σχετικής σχεδίασης των ολοκληρωμένων [25].

Οι GPUs διαφέρουν από την αρχιτεκτονική των CPUs καθώς ανταποκρίνονται σε διαφορετικές απαιτήσεις: μεγάλο πλήθος αριθμητικών πράξεων με εγγενή δυνατότητα παραλληλισμού, με έμφαση στην αυξημένη ρυθμαπόδοση (throughput), δηλαδή την ικανότητα υπολογισμών στην μονάδα του χρόνου, παρά στο χρόνο υστέρησης (latency), δηλαδή τον χρόνο από την είσοδο μέχρι την έξοδο των τελικά επεξεργασμένων δεδομένων. Αυτή η κατεύθυνση προέκυψε από την χαμηλή ικανότητα του ματιού να επεξεργαστεί ταχέως μεταβαλλόμενες εικόνες και από τα χιλιάδες έως και εκατομμύρια εικονοστοιχεία που απαιτούνται για να συνθέσουν μία ψηφιακή εικόνα. Μία τυπική εφαρμογή γραφικών λοιπόν χρειάζεται να εφαρμόσει μία σειρά διαδοχικών υπολογισμών (graphical pipeline), όπως τα vertex και fragment operations, σε ένα μεγάλο πλήθος δεδομένων εισόδου. Για να εκτελεστεί αυτό το έργο, μία CPU θα έπαιρνε μία ομάδα στοιχείων των δεδομένων εισόδου και θα εκτελούσε πρώτα τους υπολογισμούς για τα vertices, μετά τους υπολογισμούς για τα fragments κλπ, με όλους τους υπολογισμούς να γίνονται σειριακά για το κάθε σετ δεδομένων σε πολύ μικρό χρόνο (χαμηλό latency για το κάθε σετ δεδομένων). Το graphical pipeline δηλαδή διαιρείται στον χρόνο, με ένα στάδιο να εκτελείται σειριακά κάθε φορά.

Αντίθετα, οι GPUs θυσιάζουν το χαμηλό latency και την λειτουργικότητα ενός επεξεργαστή γενικού σκοπού, για να εκμεταλλευθούν δύο άλλα χαρακτηριστικά: την δυνατότητα για παράλληλη επεξεργασία πολλών δεδομένων (data parallelism) και την δυνατότητα παραλληλοποίησης του graphical pipeline, με όλα τα στάδια του να επεξεργάζονται ταυτόχρονα διαφορετικά σετ δεδομένων (task parallelism), αν και το τελευταίο χαρακτηριστικό τα τελευταία χρόνια έχει εξαλειφθεί με την έλευση των Unified Shader Models στις σύγχρονες GPGPUs.

Είσοδος λοιπόν στην GPU είναι ένα σετ γεωμετρικών αντικειμένων, τυπικά τρίγωνα σε έναν τρισδιάστατο χώρο, τα οποία μετά από μια σειρά αλγορίθμων επεξεργασίας απεικονίζονται σε μία δισδιάστατη εικόνα. Το graphical pipeline διαιρείται στον χώρο και η GPU επεξεργάζεται ταυτόχρονα διαφορετικά στάδιά του, με το τμήμα του επεξεργαστή που εκτελεί το κάθε στάδιο να εξάγει τα δεδομένα του στο τμήμα του επεξεργαστή που εκτελεί το επόμενο τμήμα του pipeline. Με αυτόν τον τρόπο τα τμήματα του επεξεργαστή που αναλαμβάνουν ένα συγκεκριμένο κομμάτι του pipeline μπορεί να υλοποιηθούν κατάλληλα σε επίπεδο

υλικού ώστε να είναι αποδοτικά για το είδος του υπολογισμού που αναλαμβάνουν. Επίσης, πέρα από την ταυτόχρονη εκτέλεση των διαφορετικών σταδίων, το κάθε στάδιο μπορεί να δεχθεί πολλαπλά δεδομένα εισόδου και να εφαρμόσει τον ίδιο υπολογισμό ταυτόχρονα, εκμεταλλευόμενο ότι οι υπολογισμοί είναι οι ίδιοι. Αυτό το χαρακτηριστικό σημαίνει ότι οι GPUs είναι μια υλοποίηση τύπου SIMD (Single Input-Multiple Data) όπου η παραλληλοποίηση επιτυγχάνεται με την ταυτόχρονη εκτέλεση των ίδιων πράξεων σε πολλαπλά στοιχεία. Με την εισαγωγή των προγραμματιζόμενων σταδίων των pixel και vertex shaders που περιγράψαμε παραπάνω, το ειδικευμένο υλικό των σταδίων αντικαταστάθηκε από προγραμματιζόμενες μονάδες χωρίς όμως να αλλάξει η οργάνωση και η λογική του pipeline.

Το αποτέλεσμα είναι μία μακρά feed-forward graphical pipeline με πολλά ειδικευμένα στάδια, όπου κάθε υπολογισμός μπορεί να χρειαστεί χιλιάδες κύκλους μηχανής για να εκτελεστεί (υψηλό latency), αλλά λόγω του task και data parallelism οι υπολογισμοί εκτελούνται σε ένα πλήθος στοιχείων επιτυγχάνοντας υψηλή ρυθμαπόδοση (throughput). Στη CPU αντίθετα, κάθε πράξη χρειάζεται λίγους κύκλους ρολογιού, αλλά μόνο ένα στοιχείο ή μία ομάδα στοιχείων θα έχει επεξεργαστεί.

4.3.1 Η αρχιτεκτονική SIMT

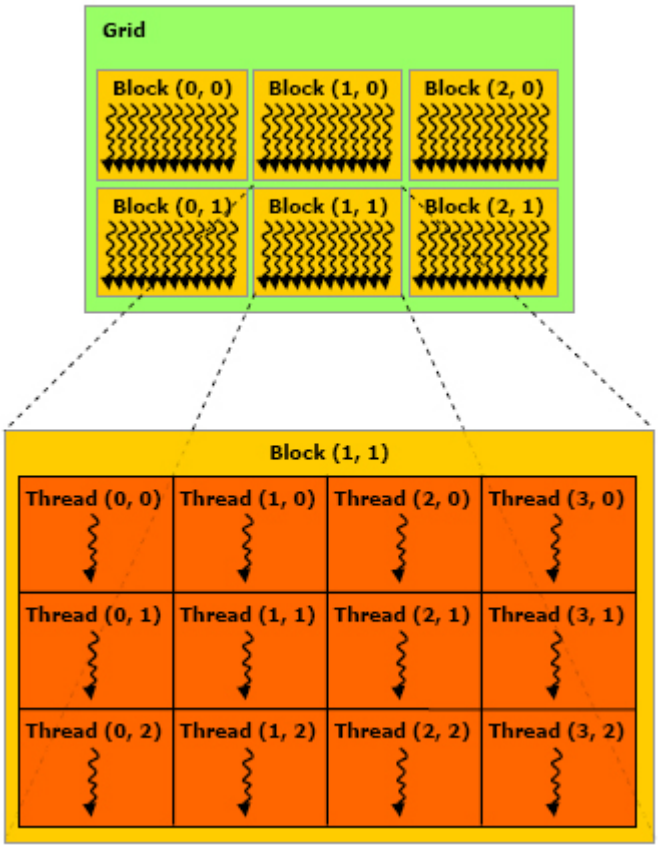
Την καρδιά αυτής της αρχιτεκτονικής αποτελούν οι πολυεπεξεργαστές. Η nVIDIA αποκαλεί τους πολυεπεξεργαστές της μονάδας επεξεργασίας Streaming Multiprocessors (SMs). Ο κάθε πολυεπεξεργαστής (SM) αποτελείται από 8 βαθμωτούς πυρήνες (SP - Scalar Processor) μονής ακρίβειας MAD (Multiply-and-Add), έναν πυρήνα διπλής ακριβείας MAD, δύο ειδικές μονάδες για χειρισμό transcendentals όπως πράξεις με ημίτονα (SFU - Special Function Unit), μία μονάδα που αναλαμβάνει την οργάνωση των threads (instruction unit) και τέλος 16KB on-chip κοινής μνήμης για όλα τα νήματα του block (shared memory). Το overhead για την δημιουργία, τον προγραμματισμό και την εκτέλεση των νημάτων είναι πρακτικά μηδενικό, επιτρέποντας την δημιουργία block με μεγάλο αριθμό από threads. Επίσης η εντολή συγχρονισμού των threads ενός block __syncthreads() απαιτεί μόνο μία εντολή μηχανής. Αυτά τα δύο χαρακτηριστικά επιτρέπουν την υψηλή

διακριτοποίηση της παράλληλης διαδικασίας με την δημιουργία πληθώρας ανεξάρτητων νημάτων, αντιστοιχώντας ακόμα και ένα στοιχείο των δεδομένων προς επεξεργασία σε κάθε thread (πχ. ένα εικονοστοιχείο μιας εικόνας) [25].

Η nVIDIA ονομάζει αυτόν τον τρόπο επεξεργασίας SIMT (Single Instruction – Multiple Threaded). Ο κάθε πολύεπεξεργαστής δημιουργεί, διαχειρίζεται, προγραμματίζει και εκτελεί τα νήματα σε ομάδες των 32, οι οποίες καλούνται warps. Τα νήματα που αποτελούν ένα warp αρχίζουν την εκτέλεσή τους από την ίδια διεύθυνση μνήμης, αλλά το καθένα έχει το δικό του ξεχωριστό μετρητή διεύθυνσης εκτέλεσης και τους δικούς τους καταχωρητές και έτσι μπορούν να αποκλίνουν κατά την εκτέλεσή τους και να εκτελούνται ανεξάρτητα [17].

Όταν ανατίθενται σε έναν πολυεπεξεργαστή ένα ή περισσότερα thread blocks για εκτέλεση, αυτός τα διαιρεί σε warps τα οποία προγραμματίζονται από τον warp scheduler (προγραμματιστής warp), για εκτέλεση. Ο τρόπος με τον οποίο ένα block διαμερίζεται σε warps, είναι πάντα ο ίδιος, οπότε κάθε warp περιλαμβάνει νήματα συνεχόμενων και αυξανόμενων thread IDs, με το πρώτο warp να περιλαμβάνει το thread 0.

Ένα warp εκτελεί μια κοινή εντολή τη φορά και κατά συνέπεια, η μέγιστη αποδοτικότητα επιτυγχάνεται όταν όλα τα νήματα ενός warp ακολουθούν παρόμοια πορεία εκτέλεσης και δεν αποκλίνουν. Εάν τα νήματα ενός warp αποκλίνουν ακολουθώντας διαφορετικά κομμάτια κώδικα μετά από εντολή διακλάδωσης, τότε το warp εκτελεί κάθε διακλάδωση ξεχωριστά, απενεργοποιώντας threads που δεν είναι σε αυτό το μονοπάτι. Όταν όλα τα διαφορετικά μονοπάτια εκτελεστούν, τα νήματα συγκλίνουν ξανά στο ίδιο μονοπάτι εκτέλεσης. Διαφορετικά warps εκτελούνται ξεχωριστά, ανεξαρτήτως κοινού ή αποκλίνοντος μονοπατιού εκτέλεσης.

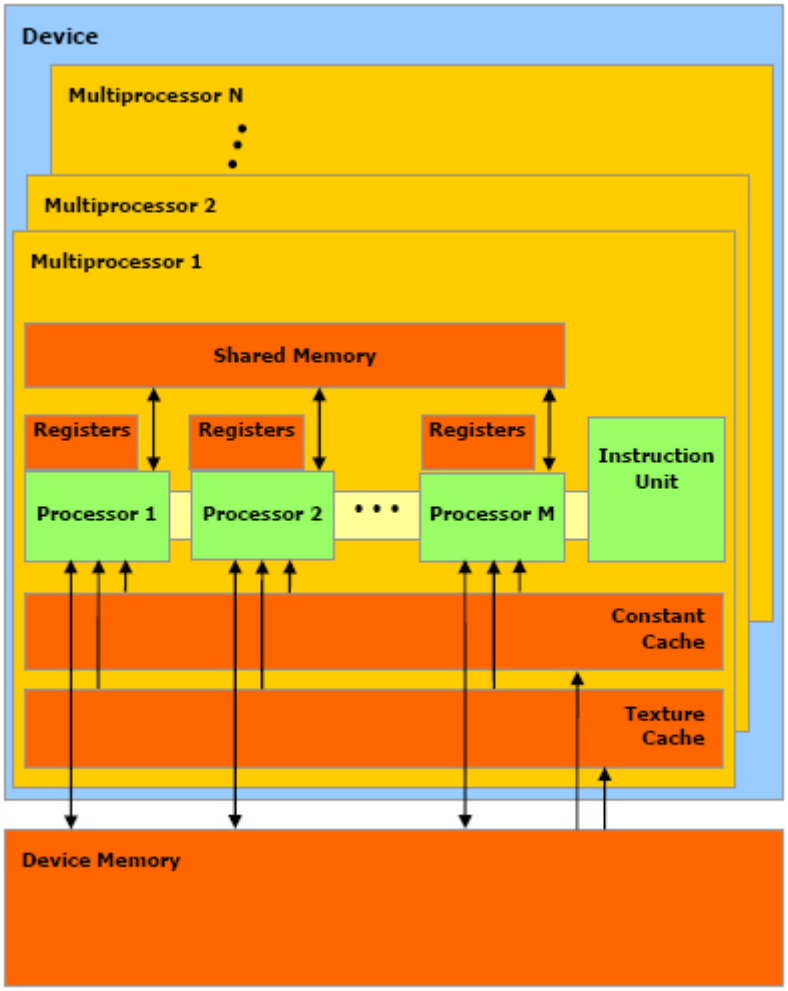


Εικόνα 4.3.1.1 Διάταξη Grid και Blocks

Ένα κατάλληλα γραμμένο πρόγραμμα σε CUDA όταν εκτελείται στη GPU δημιουργεί μια πληθώρα από νήματα (threads), καθένα από τα οποία ανήκει σε ένα block του grid. Τα blocks χαρακτηρίζονται από έναν αύξων αριθμό και διανέμονται στους SMs, με το κάθε block να εκτελείται εξολοκλήρου σε έναν SM παράλληλα και ανεξάρτητα από τα υπόλοιπα. Σε κάθε SM είναι έτοιμα προς επεξεργασία παραπάνω του ενός block (πρέπει να δίνεται μέριμνα ώστε ο αριθμός των block να υπερκαλύπτει τον αριθμό των SMs της κάθε κάρτας για βέλτιστη απόδοση), έτσι ώστε μόλις ένα block ολοκληρώσει τους υπολογισμούς του να αντικατασταθεί άμεσα από ένα άλλο.

Σε αντίθεση με τις συνήθεις αρχιτεκτονικές SIMD που απαιτούν το μέγεθος του διανύσματος παράλληλης επεξεργασίας (vector length) για να μεταγλωττιστούν (compile) για μία συγκεκριμένη αρχιτεκτονική, στην SIMT το μέγεθος του warp δεν απαιτείται, οπότε και ένα πρόγραμμα γραμμένο σε CUDA θα εκτελεστεί σε οποιαδήποτε GPU, με τους SM να αναλαμβάνουν τον διαχωρισμό και τον προγραμματισμό των threads. Πρακτικά ο προγραμματιστής ορίζει το μήκος του

vector length με τον ορισμό του μεγέθους του block. Επιπρόσθετα, τα νήματα κάθε warp στην αρχιτεκτονική SIMT ακολουθούν ανεξάρτητες μεταξύ τους αλληλουχίες εντολών, αντίθετα με τις διανυσματικές αρχιτεκτονικές (vector machines) SIMD όπου ο προγραμματιστής πρέπει να προβλέψει και να χειριστεί τις παρεκκλίσεις από την κοινή ακολουθία εντολών.



Εικόνα 4.3.1.2 Αρχιτεκτονική SIMT

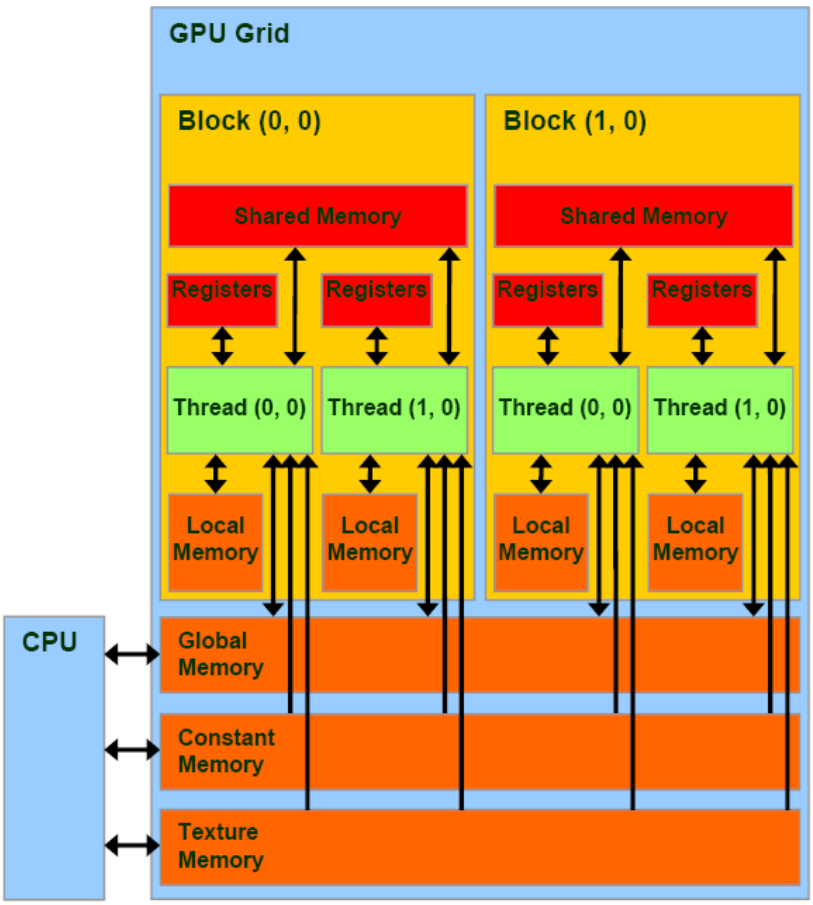
Σε κάθε χρόνο εκτέλεσης εντολής ένας πολυεπεξεργαστής επιλέγει ένα warp που είναι έτοιμο προς εκτέλεση και εκτελεί την εντολή ταυτόχρονα για όλα τα νήματα του warp. Απόρροια αυτού είναι ότι τα block πρέπει να περιέχουν πάντα περισσότερο από 32 νήματα, με δοκιμές να δείχνουν ότι 64 είναι αρκετά για να υπάρχει συνέχεια μία πληθώρα στοιχείων έτοιμων για επεξεργασία χωρίς να εξαντλείται η on-chip μνήμη από τα πολλά warps που είναι έτοιμα προς εκτέλεση. Η ρυθμαπόδοση σε έναν SM που είναι πλήρως κατειλημμένος από νήματα (32)

προκύπτει για μονή ακρίβεια, transcendental και διπλή ακρίβεια 32/8, 32/2 και 32 κύκλοι ρολογιού ανά εντολή αντίστοιχα.

Η μνήμη που υπάρχει πάνω (on-chip) σε κάθε SM είναι

- Ένα set 32μπιτων καταχωρητών (registers).
- Μία κοινή μνήμη (shared memory) για όλους τους πυρήνες (SPs).
- Μία μνήμη μόνο ανάγνωσης constant cache για όλους τους SPs που επιταχύνει τις αναγνώσεις από τον χώρο constant memory της κύριας μνήμης της κάρτας γραφικών.
- Μία μνήμη μόνο ανάγνωσης texture cache για όλους τους SPs που επιταχύνει τις αναγνώσεις από τον χώρο texture memory της κύριας μνήμης της κάρτας γραφικών.

Η κύρια μνήμη της κάρτας γραφικών χαρακτηρίζεται ως local ή global μνήμη και η προσπέλαση σε αυτήν δεν γίνεται διαμέσου κάποιας μνήμης cache.



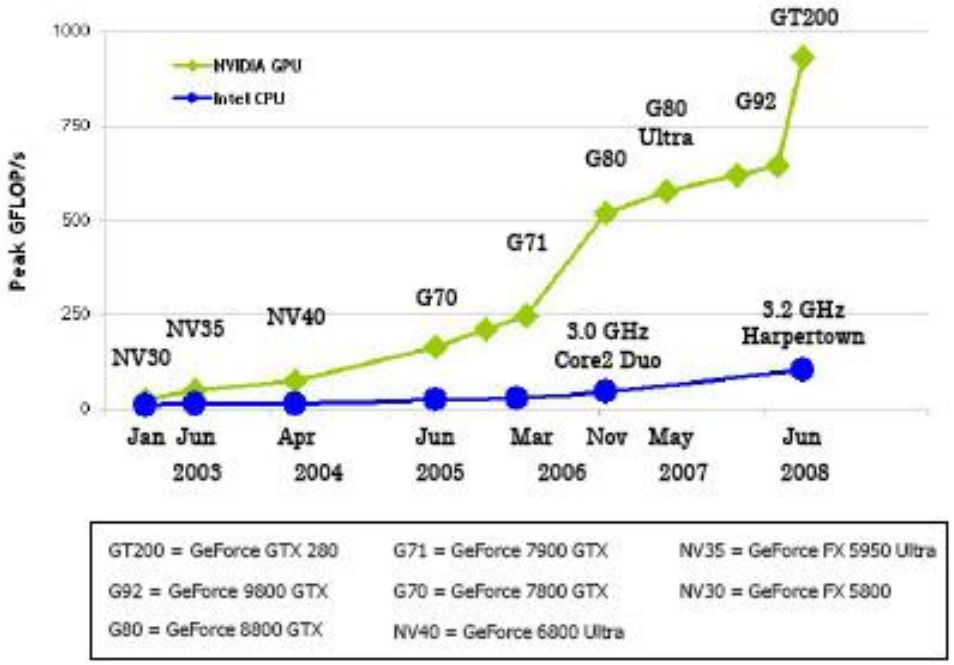
Εικόνα 4.3.1.3 Ιεραρχία της μνήμης

5 Βασικά χαρακτηριστικά της CUDA

5.1 Εισαγωγή

Τα τελευταία χρόνια η κατασκευαστική εξέλιξη των ολοκληρωμένων των επεξεργαστών μέσω της συνεχούς αύξησης των ρολογιών χρονισμού τους έχει αμβλυνθεί, εξαιτίας κυρίως των θερμικών περιορισμών και της αντίστοιχης αυξημένης κατανάλωσης ισχύος, όπως επίσης και λόγω της δυσκολίας περαιτέρω σμίκρυνσης των ολοκληρωμένων με τις υπάρχουσες μεθόδους επεξεργασίας (αυξάνοντας το ρολόι χρονισμού τα ηλεκτρόνια προλαβαίνουν να καλύψουν μικρότερη απόσταση και άρα οι πύλες των ολοκληρωμένων πρέπει να βρίσκονται

όλο και πιο κοντά). Αν και ο νόμος του Moore που προβλέπει τον ανά 18μηνο διπλασιασμό της επεξεργαστικής ισχύος εξακολουθεί να ισχύει, οι κατασκευαστές των ολοκληρωμένων έχουν μετατοπίσει τις προσπάθειες τους στην ολοένα και μεγαλύτερη παραλληλοποίηση της επεξεργαστικής ικανότητας των επεξεργαστών τους. Οι επεξεργαστές των προσωπικών υπολογιστών ήδη την τελευταία πενταετία έχουν καταστεί πολυπύρηνοι, με τις σχεδιάσεις να στοχεύουν την ενσωμάτωση δεκάδων πυρήνων στα αμέσως επόμενα χρόνια [25]. Η τάση αυτή ακολουθείται πλέον και από τα ολοκληρωμένα που χρησιμοποιούνται σε μικρότερες υπολογιστικές μονάδες (mobile phones, PDAs, tablet PCs), όπου και εκεί οι θερμικές απαιτήσεις και η κατανάλωση ισχύος είναι ιδιαίτερα πιεστικές. Σε αυτό το υπό διαμόρφωση πεδίο, οι κάρτες γραφικών βρίσκονται στον φυσικό τους χώρο, καθώς η δυνατότητα μαζικής παράλληλης επεξεργασίας με την χρήση πολλών όμοιων πυρήνων ήταν το βασικό τους επεξεργαστικό χαρακτηριστικό, όπως υπαγορευόταν από την ανάγκη εφαρμογής των ίδιων μαθηματικών μετασχηματισμών για την δημιουργία των χιλιάδων εικονοστοιχείων που απαιτούνται σε δισδιάστατες και πολύ περισσότερο σε τρισδιάστατες εφαρμογές.



Εικόνα 5.1.1 Πράξεις κινητής υποδιαστολής ανά δευτερόλεπτο και εύρος μνήμης για CPU και GPU [17]

Η εξέλιξη στις κάρτες γραφικών τα τελευταία χρόνια έχει κυρίαρχα στραφεί στην ενσωμάτωση ολοένα και περισσότερων επεξεργαστικών πυρήνων (processing cores), με τις σύγχρονες σχεδιάσεις να διαθέτουν μέχρι και 500, ενώ οι τάσεις δείχνουν σχεδόν διπλασιασμό της επεξεργαστικής τους ισχύος ανά 2 χρόνια. Αυτή η τεράστια παράλληλη επεξεργαστική ισχύς έχει δημιουργήσει την ανάγκη για κατάλληλα προγραμματιστικά εργαλεία και μεθόδους που θα διευκολύνουν την συγγραφή κώδικα κατάλληλου για παραλληλοποίηση.

Η μετατροπή ενός κώδικα γραμμένου με την κλασσική σειριακή προσέγγιση σε κατάλληλα παράλληλο κώδικα δεν έχει καταστεί εφικτό να γίνεται αυτόματα σε αποδοτικό βαθμό με χρήση τετριμμένων εργαλείων, και λόγω και των αλλαγών στο υλικό που περιγράφηκαν παραπάνω χαρακτηρίζεται πλέον ως η κυρίαρχη πρόκληση της βιομηχανίας λογισμικού [25]. Επιπλέον, οι διαφορές κάθε πλατφόρμας παραλληλισμού (CPUs, GPUs, clusters, vector machines κλπ) απαιτούν και διαφορετική υλοποίηση του παράλληλου κώδικα. Η λύση της nVIDIA σε αυτό το πρόβλημα είναι η CUDA (Compute Unified Device Architecture), μια πλατφόρμα λογισμικού που στόχο έχει την βελτιστοποίηση της συγγραφής κώδικα στις κάρτες γραφικών της εταιρείας. Η CUDA αντικαθιστά τις πλατφόρμες που χρησιμοποιούνται για την συγγραφή τρισδιάστατων εφαρμογών (openGL, Direct3D κ.ά.), που έκαναν την συγγραφή κώδικα για γενικού τύπου επεξεργαστικές εφαρμογές δύσκολο έως αδύνατο, με μερικές απλές προσθήκες στην γλώσσα C. Ο προγραμματιστής μπορεί να αναμίξει ”κλασσικό” κώδικα C που τρέχει στην CPU με κώδικα CUDA που τρέχει παράλληλα στον πυρήνα με τη χρήση κάποιων επιπλέον βιβλιοθηκών, δομών και abstractions που μέσω του API (Application Programming Interface) κρύβουν τις λεπτομέρειες του χαμηλού επιπέδου υλικού. Έτσι ο προγραμματιστής της εφαρμογής μπορεί εύκολα να γράψει παράλληλο κώδικα, δημιουργώντας συναρτήσεις kernels που εκτελούνται ασύγχρονα στην GPU και χρησιμοποιώντας έτοιμες βιβλιοθήκες συναρτήσεων της CUDA.

5.2 Το προγραμματιστικό μοντέλο της CUDA

Στον πυρήνα της CUDA υπάρχουν τρεις σημαντικές έννοιες: η ιεραρχία των νημάτων, η ιεραρχία της μνήμης και ο συγχρονισμός των πυρήνων. Ο χειρισμός αυτών γίνεται μέσω κατάλληλων κλήσεων σε συναρτήσεις του API της CUDA, ανεξάρτητα από το είδος του επεξεργαστή που χρησιμοποιείται κάθε φορά. Έτσι το εκτελέσιμο ενός προγράμματος μπορεί να εκτελεστεί με οποιονδήποτε αριθμό πυρήνων, και μόνο οι οδηγοί της κάρτας γραφικών χρειάζεται να ξέρουν τον αριθμό τους και τον συγκεκριμένο τύπο της κάρτας.

5.2.1 Η ιεραρχία των thread

Η CUDA επεκτείνει τη γλώσσα C δίνοντας τη δυνατότητα στον προγραμματιστή να χειριστεί τους πυρήνες της GPU με το να γράψει ειδικές συναρτήσεις C, που ονομάζονται kernels. Ο κάθε kernel προσδιορίζει τον κώδικα που τρέχει το κάθε thread (σε έναν πυρήνα) της GPU, και όλα τα thread που δημιουργούνται τρέχουν παράλληλα τον ίδιο αυτό κώδικα. Ο kernel καλείται δίνοντας του δύο ορίσματα, με την παρακάτω εντολή:

kernel<<<dimGrid, dimBlock>>>(... parameter list ...);

όπου με dimGrid ορίζεται ένα τρισδιάστατο διάνυσμα για τον αριθμό των blocks στο grid και με dimBlock ένα δισδιάστατο διάνυσμα για τον αριθμό των threads σε κάθε block. Κάθε block περιέχει dimBlock αριθμό από threads, και άρα ο συνολικός αριθμός των threads που δημιουργούνται είναι dimGrid x dimBlock.

Ο δείκτης ενός thread και το ID του, σχετίζονται μεταξύ τους με έναν άμεσο τρόπο: Για ένα μονοδιάστατο block, είναι τα ίδια, για ένα δισδιάστατο block μεγέθους (Dx, Dy), το thread ID ενός thread με δείκτη (x, y), είναι (x + y Dx). Για ένα τρισδιάστατο block μεγέθους (Dx, Dy, Dz) το ID ενός thread με δείκτη (x, y, z) είναι (x + y Dx + z Dx Dy).

Η κλήση ενός πυρήνα εισάγει ένα σχετικό μικρό overhead (3-7 μs), ενώ η δημιουργία και η εκτέλεση των threads δεν εισάγουν κάποιο παραπάνω κόστος [20]. Τα νήματα ενός block μπορούν να συγχρονίζονται μεταξύ τους με την εντολή

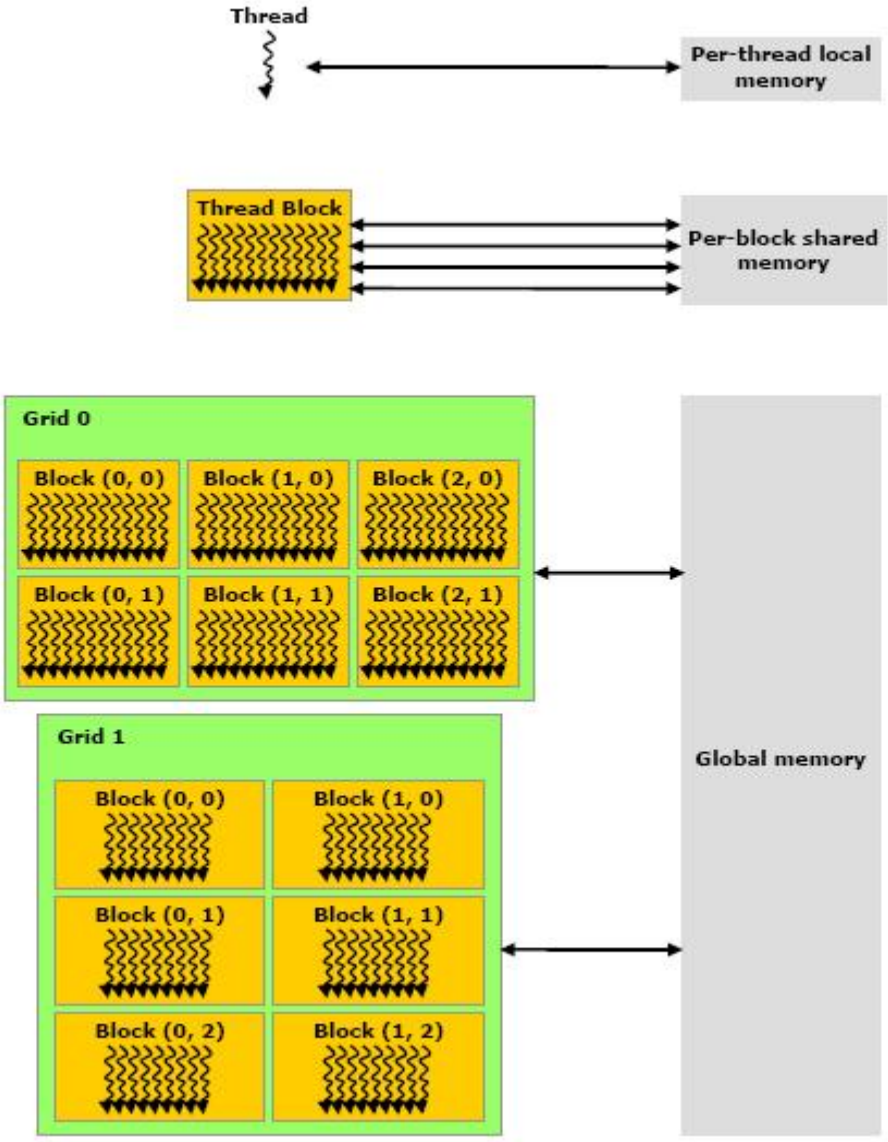
__syncthreads(), όπως επίσης και να ανταλλάσσουν δεδομένα μεταξύ τους μέσω της κοινής γρήγορης μνήμης shared memory που διαθέτει ο κάθε πυρήνας. Αντίθετα, threads διαφορετικών blocks είναι δυνατό να συγχρονίζονται μεταξύ τους μόνο μέσω των λειτουργιών ατομικής μνήμης (atomic memory operations) πάνω στην κοινή καθολική μνήμη, που όμως κοστίζουν πολύ, και το προγραμματιστικό μοντέλο της εκάστοτε εφαρμογής πρέπει να τα αντιμετωπίζει ως ανεξάρτητες, παράλληλες υπολογιστικές μονάδες. Η χρονική σειρά εκτέλεσης των blocks καθορίζεται αποκλειστικά από τις μονάδες ελέγχου του επεξεργαστή και δεν είναι δυνατόν να προσδιοριστεί από τον προγραμματιστή. Με αυτόν τον τρόπο δίνεται η δυνατότητα υλοποίησης παράλληλου κώδικα τόσο χαμηλής διακριτότητας, με την χρήση πολλών παράλληλων blocks, όσο και κώδικα υψηλότερης διακριτότητας μέσω των threads του κάθε block.

5.2.2 Η ιεραρχία της μνήμης

Στην CUDA η μνήμη του υπολογιστή (RAM) αναφέρεται ως host memory, ενώ η μνήμη της κάρτας γραφικών ως device memory. Ο κώδικας που βρίσκεται στο εσωτερικό ενός kernel μπορεί να επεξεργαστεί δεδομένα που βρίσκονται μόνο στην device memory, ενώ αντίθετα κώδικας έξω από τον kernel δεν μπορεί να συνεργαστεί απευθείας με αυτήν. Για αυτόν τον λόγο υπάρχουν συναρτήσεις δέσμευσης, αποδέσμευσης, αντιγραφής και μεταφοράς δεδομένων ανάμεσα στις host και device memory.

Τα threads έχουν πρόσβαση σε δεδομένα από διάφορους χώρους μνήμης της κάρτας γραφικών. Σε κάθε thread αντιστοιχεί ένα σύνολο από καταχωρητές (registers) προσβάσιμους μόνο από αυτό. Το συνολικό μέγεθος αυτού του register file για κάθε πυρήνα είναι 32-64KB και αποτελεί την πιο γρήγορη μνήμη που είναι διαθέσιμη. Επίσης όλα τα threads ενός block έχουν πρόσβαση σε έναν κοινό χώρο μνήμης μεγέθους 16KB ανά πυρήνα, με διάρκεια ζωής ίδια με κείνη του block. Και οι δύο αυτοί χώροι μνήμης βρίσκονται on-chip και άρα επιτυγχάνουν πολύ υψηλούς ρυθμούς διαμεταγωγής, αλλά εάν δεν υπάρχει ανάγκη για επικοινωνία δεδομένων ανάμεσα στα threads πρέπει να προτιμάται η χρήση του register file καθώς είναι γρηγορότερο [6]. Τέλος υπάρχει η global ή device memory της κάρτας γραφικών που χρησιμοποιείται για την μεταφορά και αποθήκευση δεδομένων από και προς

την μνήμη host, ενώ δύο ειδικοί χώροι μνήμης της είναι οι constant και texture memory που είναι βελτιστοποιημένες για κάποιες ειδικές χρήσεις (data filtering κλπ) [17].



Εικόνα 5.2.2.1 Κοινοί χώροι μνήμης. [17]

5.2.3 Η δομή της CUDA

Όπως είδαμε νωρίτερα, τα threads ενός block εκτελούνται σε ομάδες των 32 που ονομάζονται warps, με το κάθε thread να αντιστοιχίζεται σε έναν SP. Το μοντέλο αυτό της Nvidia (SIMT - Single-Instruction Multiple-Thread), διαφέρει από το SIMD (Single-Instruction Multiple-Data) στο ότι το δεύτερο, απαιτεί από τον

προγραμματιστή να δηλώσει τον αριθμό των παράλληλα εκτελούμενων threads (για παράδειγμα η SSE της Intel). Τα warps ενός block εκτελούνται ανεξάρτητα το ένα από το άλλο και για μέγιστη ταχύτητα πρέπει να αποφεύγεται η απόκλιση στον κώδικα ανάμεσα στα threads ενός warp. Σε περίπτωση κώδικα που αποκλίνει μέσω μιας συνθήκης, τα συγκεκριμένα threads εκτελούνται σειριακά επιπλέον της παράλληλης εκτέλεσης των υπολοίπων.

Ο μέγιστος αριθμός threads και warps ανά επεξεργαστικό πυρήνα είναι 1024 και 32 αντίστοιχα, αλλά ο αριθμός των ενεργών blocks ανά πυρήνα καθορίζεται από το πόσοι registers και πόση local memory χρειάζονται για έναν kernel, αφού οι συγκεκριμένοι χώροι μνήμης μοιράζονται ανάμεσα στα threads όλων των blocks. Αξιοσημείωτο είναι ότι η βελτιστοποίηση της απόδοσης δεν απαιτεί έναν μεγάλο αριθμό από blocks και threads, αλλά απαιτεί λιγότερα threads και αύξηση της παράλληλης δουλειάς ανα thread. Επιπλέον λιγότερα threads, συνεπάγεται μεγαλύτερος αριθμός από registers ανά thread και άρα λιγότερη πρόσβαση στη διαμοιραζόμενη μνήμη που είναι μεν γρήγορη, αλλά όχι όσο οι καταχωρητές [4]. Επιπλέον βελτιστοποίηση της απόδοσης του κώδικα μπορεί να επιτευχθεί φροντίζοντας οι προσβάσεις στην καθολική (global) μνήμη να είναι κατάλληλα στοιχισμένες. Πολλοί αλγόριθμοι γραμμικής άλγεβρας περιορίζονται από τη μέγιστη διαμεταγωγή της μνήμης (memory-bound algorithms) και όχι από την επεξεργαστική ισχύ (compute-bound). Χρησιμοποιώντας κατάλληλο padding στους πίνακες που αποθηκεύουμε για την αποφυγή διενέξεων (memory access conflicts) και με την έξυπνη χρήση της διαμοιραζόμενης μνήμης (shared memory) ως ενός ενδιάμεσου buffer κατά τη διάρκεια της εκτέλεσης των υπολογισμών του kernel, τα κόστη που σχετίζονται με τις μεταφορές από τη μνήμη στους πυρήνες και το αντίστροφο περιορίζονται αρκετά. Είναι γενικά προτιμότερο να επαναχρησιμοποιούνται δεδομένα που βρίσκονται στη διαμοιραζόμενη μνήμη, παρά να γράφονται και να ανακαλούνται δεδομένα από την καθολική μνήμη [4].

5.3 Το προγραμματιστικό περιβάλλον της CUDA

Τα βασικά στοιχεία του προγραμματιστικού μοντέλου της CUDA είναι:

- Αρχικά στο CPU κομμάτι του κώδικα αντιγράφουμε τα δεδομένα προς επεξεργασία από την host memory στην device memory. Χαρακτηριστική συνάρτηση της CUDA για δέσμευση μνήμης στην device memory είναι η cudaMalloc() και αντιγραφής (και προς τις δύο κατευθύνσεις) η cudaMemcpy().

  cudaMalloc(void **devPtr, site_t size)

  cudaMemcpy(void *dst,const void *src, size_t count, enum cudaMemcoyKind kind)

  Επίσης ορίζουμε τις μεταβλητές dimGrid και dimBlock τύπου dim3 που ορίζουν την οργάνωση του μοντέλου παραλληλοποίησης που υλοποιείται. Λόγω της πολύ υψηλής δυνατότητας παραλληλοποίησης συνήθως σε κάθε thread αντιστοιχίζεται η επεξεργασία ενός και μόνο στοιχείου (πχ εικονοστοιχείο μιας εικόνας).

- Στο εσωτερικό του kernel γίνονται οι παράλληλοι υπολογισμοί από το κάθε thread και χρησιμοποιείται η shared memory για γρήγορη εγγραφή και ανάγνωση. Το τελικό αποτέλεσμα μεταφέρεται πίσω στην global memory.

- Μεταφορά των αποτελεσμάτων στην host memory.

5.3.1 Περιορισμοί

Για μια πληρέστερη κατανόηση της CUDA, είναι σημαντικό να αναφέρουμε και τους περιορισμούς της με μερικές αναφορές στις μελλοντικές προοπτικές εξέλιξής της.

- Κατάλληλοι για μεταφορά στην CUDA είναι αλγόριθμοι εξαιρετικά απαιτητικοί σε επεξεργαστική ισχύ (computationally intensive) με καλές δυνατότητες παραλληλοποίησης που δεν απαιτούν συνεχείς επικοινωνίες ανάμεσα σε όλα τα threads (αν και σε μελλοντικές εκδόσεις της CUDA προβλέπεται η επικοινωνία ανάμεσα και στα διαφορετικά blocks).

- Η δυνατότητα κλήσης δύο kernel που να τρέχουν παράλληλα αναμένεται να υλοποιηθεί στην επόμενη έκδοση της CUDA. Αυτό θα επιτρέψει την κλήση πολλών μικρότερων (σε δεδομένα που πρόκειται να επεξεργαστούν) kernels, κάτι που σήμερα είναι ασύμφορο με την σειριακή τους εκτέλεση.
- Δεν υποστηρίζεται η αναδρομική (recursive) κλήση των kernels εξαιτίας των απαγορευτικών ποσοτήτων μνήμης που θα συνεπάγονταν οι στοίβες όλων των threads. Αυτό περιορίζει την δυνατότητα υλοποίησης κάποιων αλγορίθμων, όπως ο quicksort.
- Η απαίτηση μεταφοράς των δεδομένων από την device στη host memory με τα τεράστια κόστη που αυτή συνεπάγεται σε σύγκριση με τον χρόνο επεξεργασίας αυτών στην GPU, καθιστά μη αποδοτική τη χρήση της CUDA για μικρού μεγέθους προβλήματα.
- Η δυνατότητα για υπολογισμούς διπλής ακρίβειας (double precision) υποστηρίζεται μόνο από την τελευταία έκδοσή της και με κάθε πυρήνα να διαθέτει μόνο μία MAD pipeline διπλής ακρίβειας (και αυτό το χαρακτηριατικό πρόκειται να αλλάξει στις επόμενες εκδόσεις) [25].

6 Η μηχανή

Το πρόβλημα προς επίλυση, είναι η αυτόματη παραγωγή βελτιστοποιημένου-γρήγορου κώδικα για αρχιτεκτονικές GPU με χρήση των μηχανών BURG. Ο κώδικας αυτός αποτελεί τη λύση παρόμοιων προβλημάτων γραμμικής άλγεβρας, όπως ο πολλαπλασιασμός και η συνέλιξη πινάκων. Για μικρά μεγέθη προβλημάτων, η CPU από μόνη της επαρκεί για να εκτελέσει αυτές τις πράξεις, όταν όμως το πρόβλημα μεγαλώνει περιλαμβάνοντας αριθμούς πράξεων της τάξης εκατομμυρίων, τότε είναι αναγκαία η αποδοτική χρησιμοποίηση των πόρων που μπορεί να προσφέρει μια αρχιτεκτονική GPU.

Πιο συγκεκριμένα, δεδομένης μιας σειριακής έκδοσης κώδικα που λύνει ένα πρόβλημα γραμμικής άλγεβρας, παράγεται ένας παραλληλοποιημένος κώδικας που

αποτελεί τη μετασχηματισμένη έκδοση του σειριακού προκατόχου του, και αποτελεί τη λύση του ίδιου προβλήματος.

Η μηχανή που κατασκευάστηκε για το σκοπό αυτό λειτουργεί ως εξής:

Σαν front end λειτουργεί ο parser CustomParser.java, ο οποίος σαρώνει ένα αρχείο πηγαίου κώδικα, και δημιουργεί μια δενδρική δομή ενδιάμεσης αναπαράστασης, που περιλαμβάνει όλους τους βρόχους του πηγαίου αρχείου, μαζί με τα statements που βρίσκονται εκεί. Η τάξη CustomParser βρίσκεται στο Παράρτημα, παρακάτω όμως φαίνεται η μέθοδος Parse που εκκινεί τη διαδικασία με κλήση της μεθόδου Compile().

```
public TL1INode Parse() {

 TL1INode tree = null;
 try {
        tree = Compile();
        tree = FormatTree(tree);
 } catch (ParseException e) {
    System.err.println("CustomParser.java:"+ e.getMessage());
 } catch (Exception ee) {
   System.err.println("CustomParser.java: "+ee.getMessage());
 }
 return tree;
}
```

Στο δεύτερο βήμα της διαδικασίας, η δενδρική δομή που δημιουργήθηκε στο προηγούμενο βήμα, δίνεται σαν παράμετρος στην τάξη BurmComplete.java. Η τάξη αυτή είναι η BURM μηχανή, που κάνει tree pattern matching και έχει δημιουργηθεί από τη γεννήτρια γεννητριών κώδικα JBURG. Στο specification το οποίο χρησιμοποιήθηκε από τη μηχανή JBURG για τη γένεση της μηχανής burm περιλήφθηκαν οι απαραίτητοι κανόνες ταύτισης δενδρικών προτύπων που περιλαμβάνουν βρόχους στους κόμβους τους, έτσι ώστε η BurmComplete.java, να είναι ικανή να επεξεργαστεί το δένδρο ενδιάμεσης αναπαράστασης με τους βρόχους, που δημιουργήθηκε από τον parser. Παρακάτω φαίνεται ο κανόνας του αρχείου τεκμηρίωσης (specification), που κάνει ταύτιση σε δενδρικό κόμβο με βρόχους:

```
statement = FOR(FOR(statement s1, statement s2), statement s3) :
getCudaCost(){
  //semantic actions
}
```

και το αντίστοιχο κομμάτι κώδικα ελέγχου που δημιουργείται μέσα στη μηχανή BurmComplete.java, με σκοπό να κάνει το pattern matching

```
if (node.getArity() == 1 && node.getOperator() == FOR &&
node.getNthChild(0).getOperator() == FOR)
```

Διακρίνουμε στον παραπάνω κανόνα αντικατάστασης του specification, ότι το κόστος ενός κόμβου του δέντρου, τύπου FOR, είναι δυναμικά υπολογιζόμενο, από τη μέθοδο getCudaCost(), η οποία πυροδοτεί την έναρξη της διαδικασίας παραγωγής κώδικα cuda και εκτέλεσής του για την ανάκτηση του αντίστοιχου κόστους:

```
private double getCudaCost(jburg.compiler.tl1.ir.TL1INode p) {

     double cost = Double.MAX_VALUE;
     if(transformGeneric.treeDepth(p)>=2&&
transformGeneric.treeInNormalForm(p)) {
          cost = 0.0;

          try {
                this.cGen.dumpCode(p);

                NumberFormat formatter = new DecimalFormat("##");

                double response = this.cGen.Execute();
                cost = response;
            } catch (NumberFormatException e) {
                return Double.MAX_VALUE;
            } catch (ShellException ee) {
                //returning max value, means that the current
experiment is being ignored, due to
                //a very bad processing time
                return Double.MAX_VALUE;
            }
```

```
        }
        return cost;
    }
```

Στο επόμενο βήμα της διαδικασίας βρίσκεται ο IRTransformer.java, η τάξη που παραλαμβάνει τη δενδρική δομή, και εφαρμόζει μετασχηματισμούς (βρίσκονται στο παράρτημα οι κώδικες των μετασχηματισμών) για να πετύχει το ιδανικότερο blocking του προβλήματος, δηλαδή το δέντρο που θα πετύχει την δημιουργία του ταχύτερου κώδικα CUDA. Η τάξη αυτή συνεργάζεται στενά με τις τάξεις CudaGen.java και Emitter.java, οι οποίες είναι υπεύθυνες για τη σάρωση του δέντρου, την παραγωγή cuda κώδικα και την εκτέλεσή του, ώστε να μετρηθεί ο χρόνος εκτέλεσης. Η μέθοδος της τάξης Emitter η οποία παράγει τα cuda statements, βρίσκεται στο Παράρτημα. Το παρακάτω τμήμα κώδικα, είναι η δημιουργία του cuda kernel από το μετασχηματισμένο δέντρο, και βρίσκεται στην τάξη CudaGen.java

```
public void GenerateKernel(TL1INode node) {
    .
    .
    .
    .
    .

    String kern = GenKernelHeader(INITIAL_TREE_DEPTH) + " {";

    //declare thread and block id variables
    kern+=UnpackCode("\t", emitter.GenThreadBlockDeclarations());

    //create sum declarations
    kern += GenSumDeclarations((iiLoop == null) ? iLoop : iiLoop,
(jjLoop == null) ? jLoop : jjLoop);
    kern+=UnpackCode("\t", emitter.GenVariableDefinitions());

    //create shared memory array declarations
    kern+=UnpackCode("\t", emitter.GenCacheArrayDeclarations());

    //main kernel generation
    kern += "\n" + Kernel("\t", N3Loop);
```

```
    kern += UnpackCode("\t", emitter.GenAssignSums());
    kern += "\n}\n";

    kernel = kern;
  }
```

και η αναδρομική Kernel που δημιουργεί τους εμφωλευμένους βρόχους του cuda kernel,

```
private String Kernel(String indentation, TL1INode n3node) {
        String kern = "";

  if (n3node != null && n3node.getObjectContent() instanceof For)   {
    //print itself
    For forr = (For) n3node.getObjectContent();
    kern += "\n" + indentation + forr.toString() + "{";

    //print its children
    kern += Kernel(indentation + "\t", n3node.getNthChild(1));
    kern+="\n"+Kernel(indentation + "\t", n3node.getNthChild(0));

    kern += "\n" + indentation + "}";

} else if (n3node != null && n3node.getObjectContent() instanceof
StatementList) {
   StatementList stmts=(StatementList) n3node.getObjectContent();
    kern += UnpackCode(indentation, stmts);
  }
   return kern;
}
```

αλλά και η UnpackCode που δέχεται σαν παράμετρο μια λίστα από statements και επιστρέφει ένα String με τις εντολές

```
  private String UnpackCode(String indentation, StatementList list) {

    String result = "";
    ArrayList arrlist = list.getList();
```

```
    for (int i = 0; i < arrlist.size(); i++) {
        Assign assign = (Assign) arrlist.get(i);
        result+="\n"+indentation+      assign.toString().replaceAll("#",
"");
    }
    return result;
}
```

Μετά απο μια διαδικασία μετασχηματισμού του αρχικού δέντρου, δημιουργίας κώδικα και εκτέλεσής του, που μπορεί να διαρκέσει από λίγα λεπτά μέχρι ώρες, η μηχανή καταλήγει στο βέλτιστο δέντρο, το οποίο και σώζεται. Το δέντρο αυτό αποτελεί το παραλληλοποιημένο, βελτιστοποιημένο ισοδύναμο του αρχικού σειριακού κώδικα που αναγνώστηκε από το πηγαίο αρχείο.

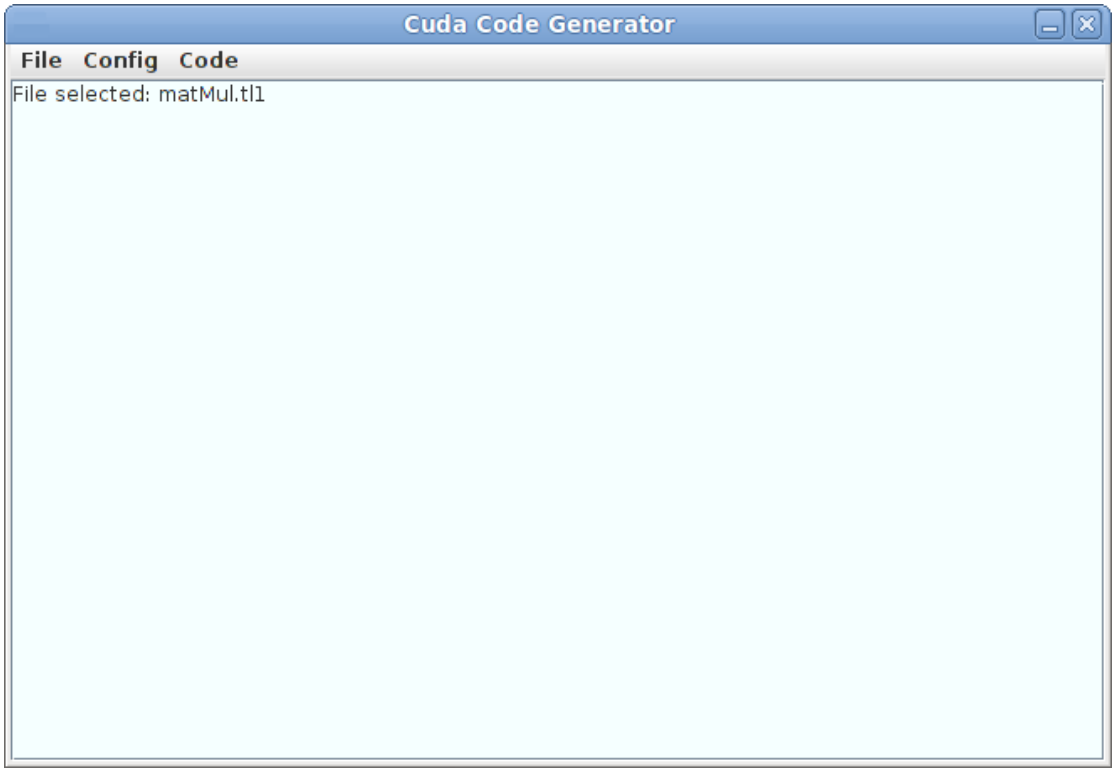
Συνοπτικά λοιπόν το σύστημα παίρνει σαν είσοδο ένα πρόγραμμα σε γλώσσα υψηλού επιπέδου, και παράγει το παράλληλο ισοδύναμό του σε CUDA, επίσης γλώσσα υψηλού επιπέδου.

Το συντακτικό που χρησιμοποιήθηκε για την περιγραφή, αποθήκευση, επαναδημιουργία και γενικότερη διαχείριση των εντολών της cuda, για λόγους απλότητας αποτελείται από τρεις τάξεις: For.java, Assign.java και StatementList.java. Με αυτό τον τρόπο γίνεται ξεκάθαρα ο διαχωρισμός ανάμεσα στους βρόχους και στα υπόλοιπα statements, κάτι που διευκολύνει την όλη διαδικασία.

Η τάξη For αναπαριστά αντικείμενα βρόχων κρατώντας χρήσιμα χαρακτηριστικά όπως ο δείκτης ενός βρόχου, τα άνω και κάτω όρια των επαναλήψεων, το βήμα μεταβολής του μετρητή κλπ. Η Assign είναι εξειδικευμένη τάξη της Statement.java και κρατάει ένα οποιοδήποτε statement. Η τάξη αυτή λειτουργεί κυρίως σαν αποθήκη, καθώς το κάθε statement αναπαρίσταται ως ένα String χωρίς κάποια ιδιαίτερα χαρακτηριστικά. Τέλος η StatementList.java, είναι η τάξη που ομαδοποιεί όλα τα statements που είναι τύπου Assign, σε μια λίστα. Έτσι γίνεται πολύ εύκολη η ανάθεση αρκετών statements σε ένα βρόχο του δέντρου, καθώς και η μετακίνησή τους σε άλλο κόμβο, ή η διαγραφή τους.

Τέλος δημιουργήθηκε και μια γραφική διεπαφή χρήστη (GUI – Graphical User Interface), η οποία δίνει τη δυνατότητα στο χρήστη, να εκκινήσει τη διαδικασία παραγωγής κώδικα που περιγράφηκε, όπως επίσης και να ρυθμίσει και να

παραμετροποιήσει τη μηχανή. Στο κενό μέρος του παραθύρου, εμφανίζεται η πρόοδος της διαδικασίας παραγωγής κώδικα, καθώς και άλλα ενημερωτικά μηνύματα. Στην παρακάτω εικόνα φαίνεται το αρχικό παράθυρο της εφαρμογής:



Εικόνα 6.1 Γραφική διεπαφή χρήστη της μηχανής

Όλες οι λειτουργίες παρέχονται από το μενού στο πάνω μέρος του παραθύρου.

- Από το μενού *File*, ο χρήστης μπορεί να επιλέξει το αρχείο πηγαίου κώδικα που επιθυμεί ή να τερματίσει την εφαρμογή με τις επιλογές *Open File* ή *Exit* αντίστοιχα.
- Στο μενού *Config* παρέχονται επιλογές παραμετροποίησης των μετασχηματισμών που θα εκτελεστούν, όπως το αν θα γίνει blocking με strip-mining ή tiling καθώς και τα άνω και κάτω όρια των μεγεθών των blocks που θα χρησιμοποιήσει η μηχανή. Αντίστοιχες είναι και οι επιλογές που παρέχονται για το unrolling.
- Το μενού *Code* είναι αυτό όπου ο χρήστης μπορεί να ξεκινήσει τη διαδικασία παραγωγής κώδικα με την επιλογή *Generate*, ή να τη σταματήσει με την επιλογή *Stop Process.*

Τα πειράματα που εκτελέστηκαν περιλαμβάνουν τον πολλαπλασιασμό και τη συνέλιξη πινάκων δύο και τριών διαστάσεων, ενώ την πλατφόρμα εκτέλεσης των πειραμάτων, αποτέλεσε η κάρτα Tesla C1060 που αποτελεί πρόταση της Nvidia αποκλειστικά για εκτέλεση προβλημάτων που απαιτούν μεγάλη επεξεργαστική ισχύ.

6.1 Πολλαπλασιαμός πινάκων

Στον πολλαπλασιασμό πινάκων, έγινε προσπάθεια προσέγγισης των παραδειγμάτων για μετασχηματισμούς που προτείνει ο M. Wolfe στα άρθρα του 'Programming GPUs Today' και 'Optimizing GPU Kernels', για την επίτευξη ταχύτερου κώδικα CUDA. Αποκλίναμε όμως από τα πρότυπα των παραπάνω άρθρων με σκοπό να πετύχουμε καλύτερη απόδοση και έγινε σύγκριση με τους παραγόμενους κώδικες. Στα πειράματα που εκτελέστηκαν ώστε να βρεθεί ο ταχύτερος πολλαπλασιαμός, εφαρμόστηκαν διάφοροι τύποι βελτιστοποιήσεων βρόχων, με σκοπό να αλλάξουν τα πρότυπα πρόσβασης στην καθολική μνήμη, να αναδείξουν τη χρήση της διαμοιραζόμενης μνήμης, - καθώς διαπιστώθηκε και στην πράξη ότι οι επικοινωνίες με την καθολική μνήμη είναι ιδιαίτερα αργές -, αλλά και τη χρήση του αρχείου των καταχωρητών, μιας δομής εξαιρετικά ταχείας πρόσβασης. Αξιοσημείωτο είναι το γεγονός, ότι βέλτιστα αποτελέσματα δεν επιτεύχθηκαν όσο αυξανόταν η χρησιμοποίηση των threads. Αντιθέτως οι χρόνοι βελτιώνονταν, όταν με χρήση του loop unrolling μειωνόταν ο αριθμός τους και αυξάνονταν οι πράξεις ανά thread.

Η απόκλιση όμως από τις βελτιστοποιήσεις του M. Wolfe, έγινε με την εξολοκλήρου αφαίρεση της πρόσβασης στην καθολική μνήμη, από τον εσωτερικό βρόχο που εκτελεί τις πράξεις (προσθέσεις και πολλαπλασιασμούς), με χρήση του μετασχηματισμού strip-mining. Η αύξηση της ταχύτητας είναι θεαματική συγκρινόμενη με τον κώδικα που προτείνει ο M. Wolfe. Στους παρακάτω πίνακες εμφανίζονται συγκεντρωτικά οι χρόνοι εκτέλεσης και τα Gflops του βέλτιστου κώδικα, των πειραμάτων που εκτελέστηκαν για μεταβλητά μεγέθη πινάκων, συγκρινόμενοι με τους χρόνους που πετυχαίνουν οι αντίστοιχοι κώδικες του M.Wolfe.

Μέγεθος πίνακα / block	64	512	1088	2496	5312	8000
64x4	0,0000	0,0011	0,0093	0,1100	1,0552	3,6008
	0,0000	0,0018	0,0160	0,1899	1,8228	6,2139
128x4	0,0000	0,0010	0,0094	0,1082	1,0281	3,4955
	0,0000	0,0018	0,0166	0,1899	1,7861	6,0784
256x4	0,0000	0,0010	0,0109	0,1093	1,0252	3,5275
	0,0001	0,0018	0,0195	0,1957	1,8181	6,2385

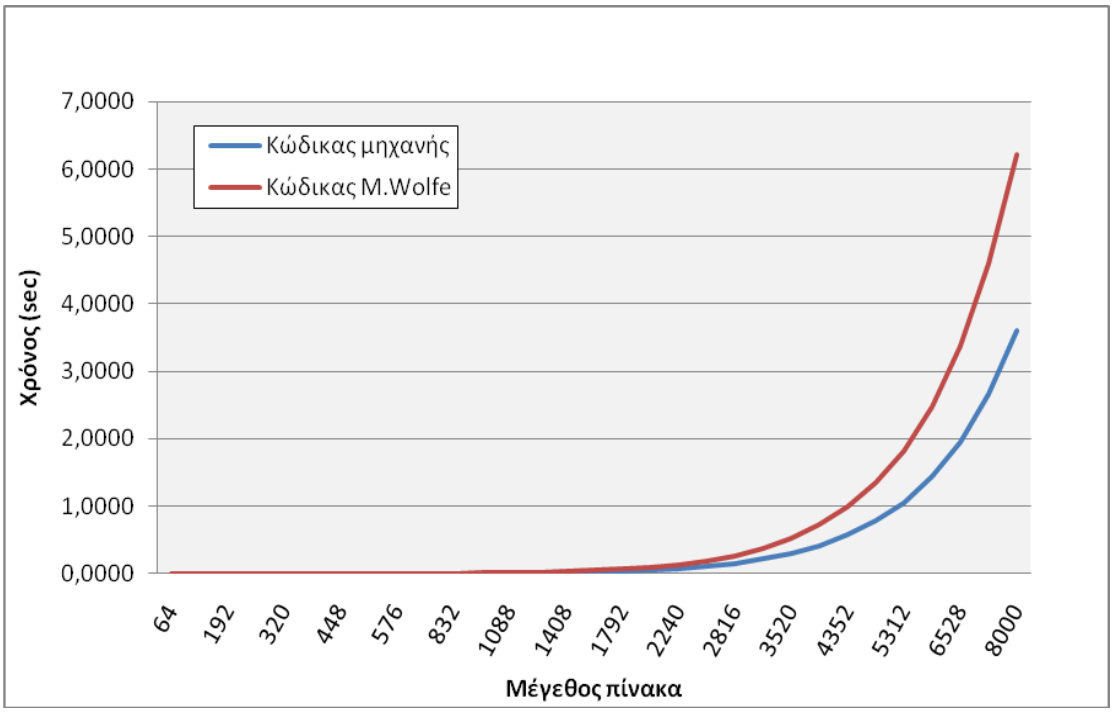
Πίνακας 6.1.1 Χρόνοι εκτέλεσης πολ/σμού για μεταβλητά μεγέθη πινάκων και blocks. Κώδικας μηχανής και M.Wolfe αντίστοιχα

Μέγεθος πίνακα / block	64	512	1088	2496	5312	8000
64x4	24,58	254,71	276,68	282,62	284,11	284,38
	17,68	146,88	161,35	163,77	164,46	164,79
128x4	25,28	265,07	274,04	287,46	291,58	292,95
	17,61	149,17	154,82	163,76	167,84	168,47
256x4	12,74	265,44	235,73	284,48	292,41	290,29
	9,15	151,2	132,1	158,92	164,88	164,14

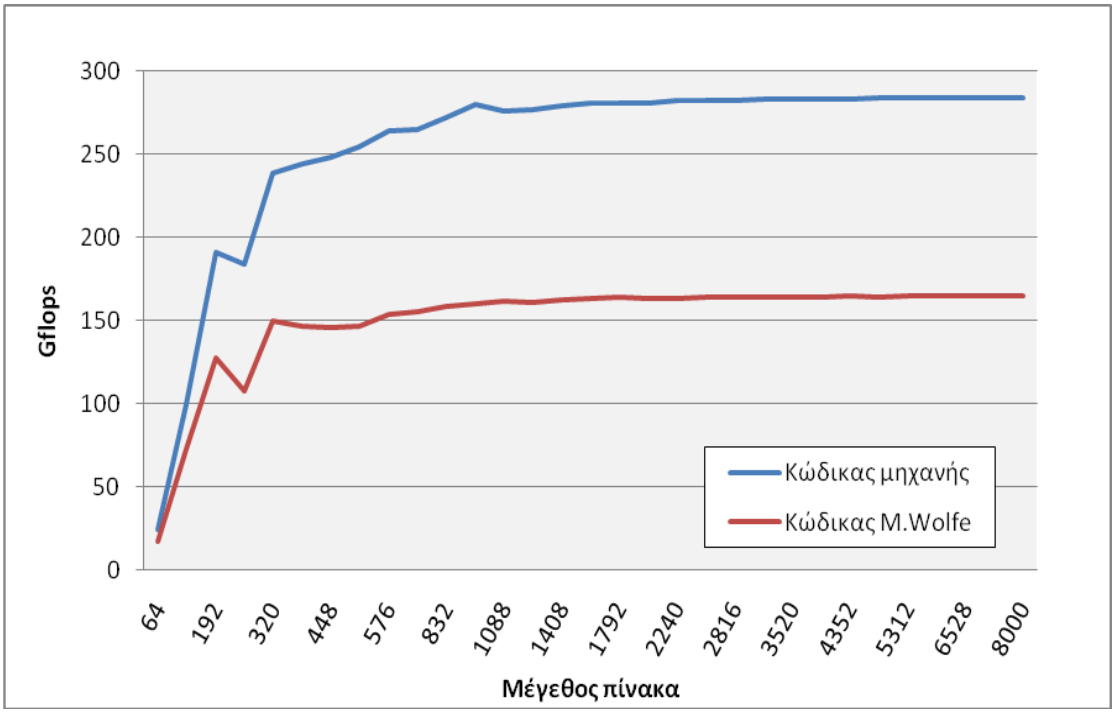
Πίνακας 6.1.2 Gflops που πετυχαίνουν οι κώδικες μηχανής και M.Wolfe για μεταβλητά μεγέθη πινάκων και blocks

Ακολουθούν οπτικοποιημένες οι παραπάνω μετρήσεις στα αντίστοιχα διαγράμματα:

- Με strip-mining και unrolling δημιουργήθηκαν μεγέθη block από 32x1 έως 64x4, με το βέλτιστο αποτέλεσμα να επιτυγχάνεται με block μεγέθους 64x4, με unrolling 2 φορές στο SIMT loop που αφορά στα threads στον άξονα X, unrolling 4 φορές στο SIMT loop για τα threads του άξονα Y, και unrolling 2 φορές στο εσωτερικό loop. Η απόδοσή του μετρήθηκε για μεγέθη πινάκων από 64x64 έως 8000x8000:



Εικόνα 6.1.1 Χρόνοι εκτέλεσης για μεταβλητά μεγέθη πινάκων, blocks 64x4

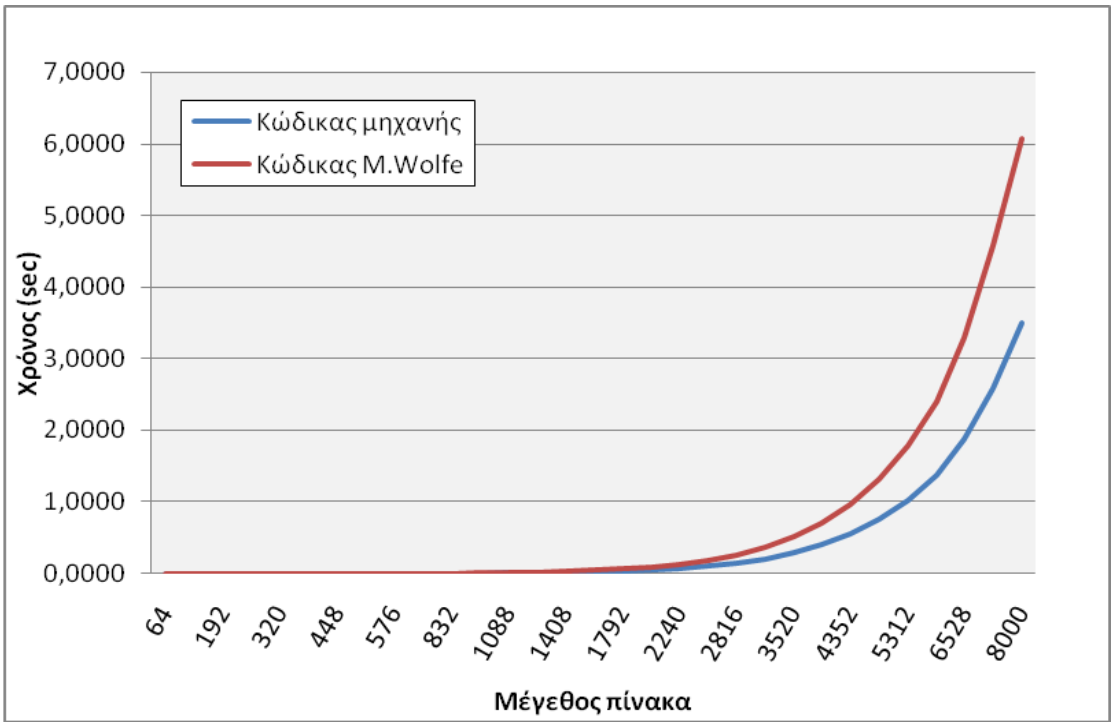


Εικόνα 6.1.2 Gflops που επιτυγχάνουν οι δύο διαφορετικοί κώδικες, blocks 64x4

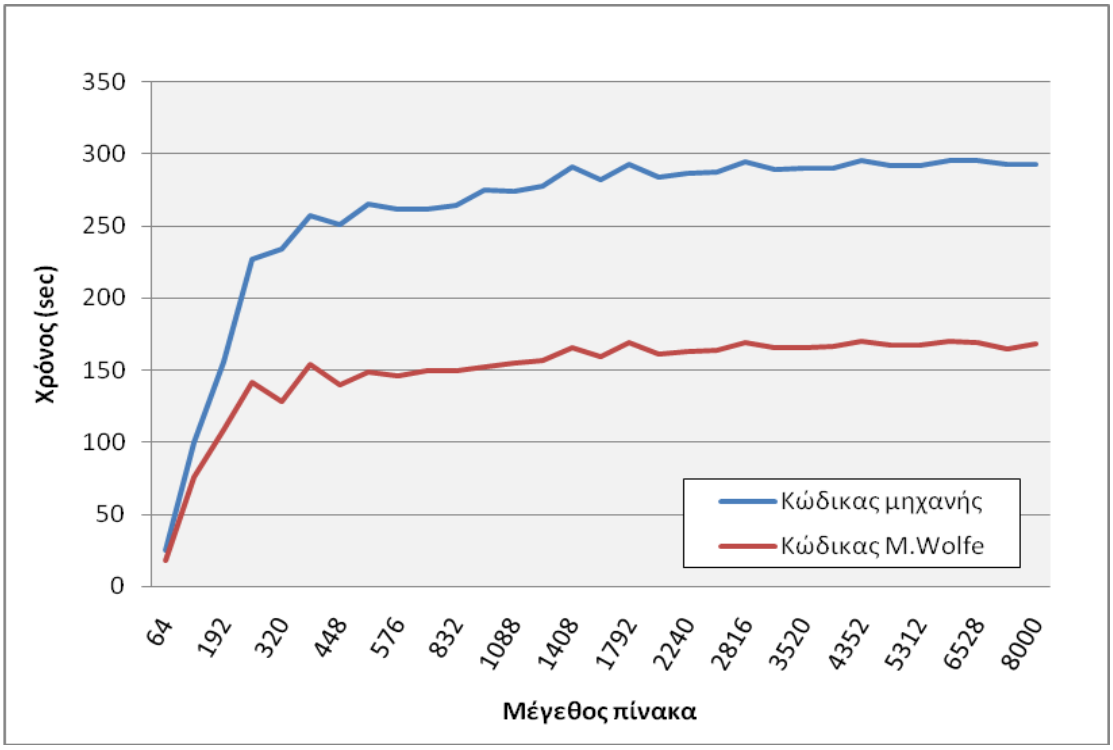
Μπορούμε να δούμε ότι ο κώδικας του M.Wolfe υστερεί σε απόδοση, λόγω της εντολής πρόσβασης στην καθολική μνήμη, μέσα στο εσωτερικό loop που εκτελεί τις

πράξεις. Ο παραγόμενος κώδικας της μηχανής φτάνει στα 284 Gflops ενώ ο κώδικας του M.Wolfe τα 164.

- Όπως και προηγουμένως, με strip-mining και unrolling δημιουργήθηκαν μεγέθη block από 32x1 έως 128x4, με το βέλτιστο αποτέλεσμα να επιτυγχάνεται με block μεγέθους 128x4, με unrolling στα SIMT loops, αλλά και στο εσωτερικό loop:



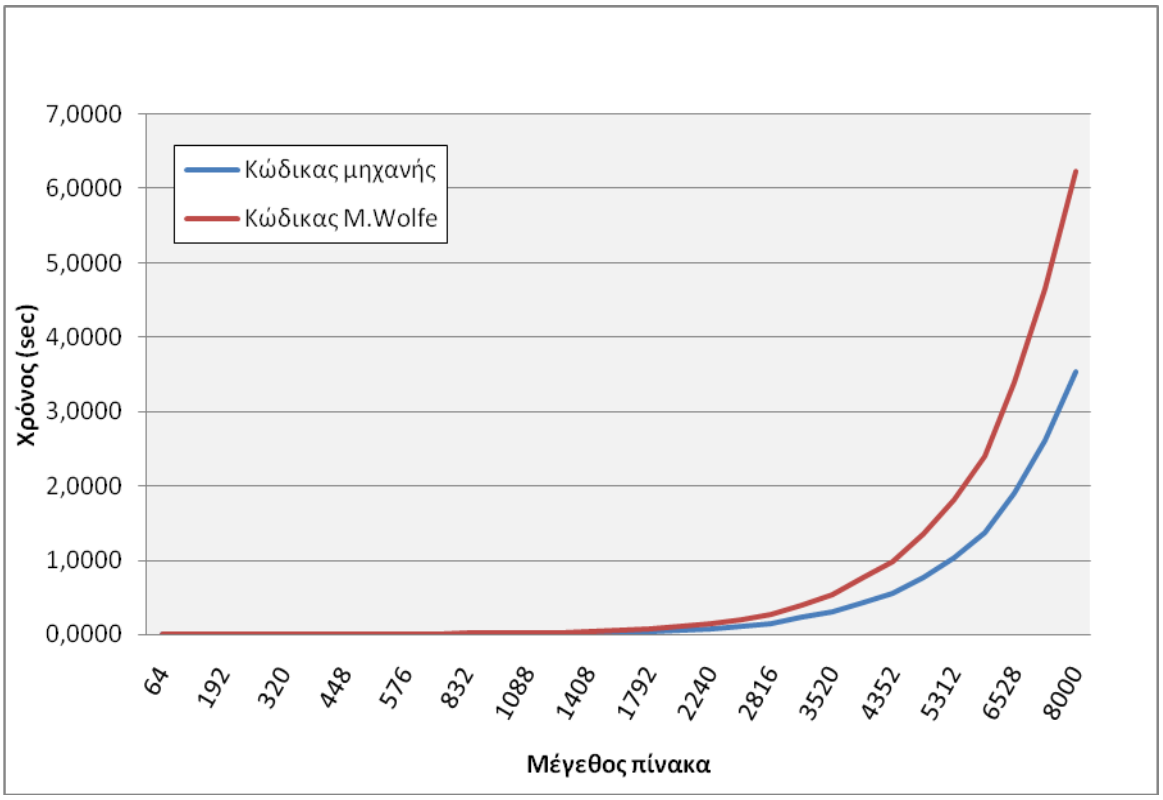
Εικόνα 6.1.3 Χρόνοι εκτέλεσης για μεταβλητά μεγέθη πινάκων, blocks 128x4



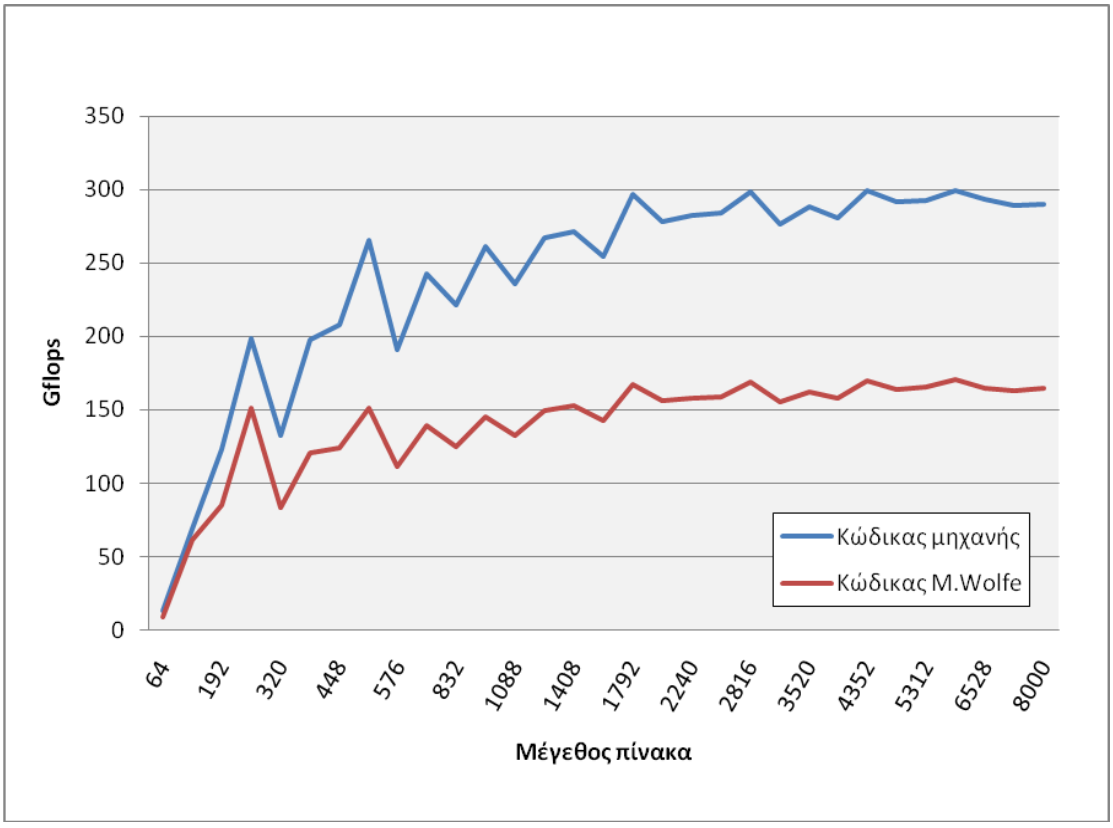
Εικόνα 6.1.4 Gflops που επιτυγχάνουν οι διαφορετικοί κώδικες, blocks 128x4

Παρά το ότι η απόδοση του κώδικα του M.Wolfe σε αυτή την περίπτωση βελτιώνεται φτάνοντας τα 170 Gflops, εντούτοις υστερεί σε σχέση με τον κώδικα μηχανής, καθώς ο δεύτερος φτάνει στα 292 Gflops.

- Δημιουργία μεγεθών block από 128x1 έως 256x4, με το βέλτιστο αποτέλεσμα να επιτυγχάνεται με block μεγέθους 256x4, με unrolling στα SIMT loops, αλλά και στο εσωτερικό loop:



Εικόνα 6.1.5 Χρόνοι εκτέλεσης για διαφορετικά μεγέθη πινάκων, blocks 256x4



Εικόνα 6.1.6 Gflops που επιτυγχάνουν οι διαφορετικοί κώδικες, blocks 256x4

Παρατηρούμε στις παραπάνω περιπτώσεις τη σημασία της χρήσης λιγότερων threads με loop unrolling, οπότε και την αύξηση του αριθμού των registers ανά thread, αλλά και τη χρήση της διαμοιραζόμενης μνήμης. Με τη χρήση του loop unrolling επιτυγχάνουμε επαναχρησιμοποίηση των δεδομένων που φορτώνονται τοπικά σε κάθε thread, και μείωση της κίνησης στη διαμοιραζόμενη μνήμη. Φυσικά για να πετύχει τα παραπάνω ένας κώδικας, προϋπόθεση είναι να μη βασίζεται στην εξολοκλήρου στην καθολική μνήμη. Αυτό φαίνεται στον κώδικα του M.Wolfe, όπου η απόδοσή του δεν μπορεί να βελτιωθεί σημαντικά παρά τις βελτιστοποιήσεις των βρόχων, γιατί διατηρεί εντολή πρόσβασης στην καθολική μνήμη (`float rb = b[i+WB*(k+ks)];`) μέσα στο εσωτερικό loop. Έτσι στην τελευταία περίπτωση φτάνει τα 170 Gflops, ενώ ο κώδικας μηχανής στα 300 Gflops. Παρακάτω δίνονται τα cuda kernels της μηχανής και του M.Wolfe αντίστοιχα του συγκεκριμένου πειράματος:

kernel μηχανής:

```
__global__ void optKernel(float * a, float * b, float * c, int WA, int
WB, int WC) {
        int x = threadIdx.x;
        int xx = __mul24(blockIdx.x, 256);
        int tx = x + xx;
        int y = threadIdx.y;
        int yy = __mul24(blockIdx.y, 4);
        int ty = y + yy;

        float sum[8];
        __shared__ float bs[256], cs[512];

     for(int k = 0; k < WB; k += 128){
        bs [ x ] = b [ x + xx + WB * k + x ] ;
        bs [ ( x + 128 ) ] = b [ x + ( xx + 128 ) + WB * k + x + 128 ] ;
        cs [ x ] = c [ k + x + WC * yy ] ;
        cs [ ( x + 128 ) ] = c [ k + x + WC * ( yy + 1 ) ] ;
        cs [ ( x + 256 ) ] = c [ k + x + WC * ( yy + 2 ) ] ;
        cs [ ( x + 384 ) ] = c [ k + x + WC * ( yy + 3 ) ] ;

        __syncthreads();
```

```
        for(int kk = k; kk < k + 128; kk += 2){
         sum [ 0 ] += bs[ kk - k ] * cs[ kk - k ] ;
         sum [ 1 ] += bs[ kk - k ] * cs[ kk - k + 128 ] ;
         sum [ 2 ] += bs[ kk - k ] * cs[ kk - k + 256 ] ;
         sum [ 3 ] += bs[ kk - k ] * cs[ kk - k + 384 ] ;
         sum [ 4 ] += bs[ kk - k + 128 ] * cs[ kk - k ] ;
         sum [ 5 ] += bs[ kk - k + 128 ] * cs[ kk - k + 128 ] ;
         sum [ 6 ] += bs[ kk - k + 128 ] * cs[ kk - k + 256 ] ;
         sum [ 7 ] += bs[ kk - k + 128 ] * cs[ kk - k + 384 ] ;
         sum [ 0 ] += bs[ kk + 1 - k ] * cs[ kk + 1 - k ] ;
         sum [ 1 ] += bs[ kk + 1 - k ] * cs[ kk + 1 - k + 128 ] ;
         sum [ 2 ] += bs[ kk + 1 - k ] * cs[ kk + 1 - k + 256 ] ;
         sum [ 3 ] += bs[ kk + 1 - k ] * cs[ kk + 1 - k + 384 ] ;
         sum [ 4 ] += bs[ kk + 1 - k + 128 ] * cs[ kk + 1 - k ] ;
         sum [ 5 ] += bs[ kk + 1 - k + 128 ] * cs[ kk + 1 - k + 128 ] ;
         sum [ 6 ] += bs[ kk + 1 - k + 128 ] * cs[ kk + 1 - k + 256 ] ;
         sum [ 7 ] += bs[ kk + 1 - k + 128 ] * cs[ kk + 1 - k + 384 ] ;
        }
       }
       a [ ( tx ) + WA * ( yy ) ] = sum [ 0 ];
       a [ ( tx ) + WA * ( ( 1 ) + yy ) ] = sum [ 1 ];
       a [ ( tx ) + WA * ( ( 2 ) + yy ) ] = sum [ 2 ];
       a [ ( tx ) + WA * ( ( 3 ) + yy ) ] = sum [ 3 ];
       a [ ( ( tx + 128 ) ) + WA * ( yy ) ] = sum [ 4 ];
       a [ ( ( tx + 128 ) ) + WA * ( ( 1 ) + yy ) ] = sum [ 5 ];
       a [ ( ( tx + 128 ) ) + WA * ( ( 2 ) + yy ) ] = sum [ 6 ];
       a [ ( ( tx + 128 ) ) + WA * ( ( 3 ) + yy ) ] = sum [ 7 ];
}
```

kernel M.Wolfe

```
__global__ void optKernel(float * a, float* b, float* c, int WA, int WB,
int WC){

    int tx = threadIdx.x;
    int i = blockIdx.x*256 + tx;
    int j = blockIdx.y*4;
    __shared__ float cb0[128], cb1[128], cb2[128], cb3[128];
   float sum00 = 0.0, sum01 = 0.0, sum02 = 0.0, sum03 = 0.0;
   float sum10 = 0.0, sum11 = 0.0, sum12 = 0.0, sum13 = 0.0;
```

```
    for(int ks = 0; ks < WB; ks += 128){
          cb0[tx] = c[ks+tx+WC*j];
          cb1[tx] = c[ks+tx+WC*(j+1)];
          cb2[tx] = c[ks+tx+WC*(j+2)];
          cb3[tx] = c[ks+tx+WC*(j+3)];
          __syncthreads();
          for(int k = 0; k < 128; k+=2){
                float rb = b[i+WB*(k+ks)];
                sum00 += rb * cb0[k];
                sum01 += rb * cb1[k];
                sum02 += rb * cb2[k];
                sum03 += rb * cb3[k];
                rb = b[i+WB*(k+ks+1)];
                sum00 += rb * cb0[k+1];
                sum01 += rb * cb1[k+1];
                sum02 += rb * cb2[k+1];
                sum03 += rb * cb3[k+1];
                rb = b[i+128+WB*(k+ks)];
                sum10 += rb * cb0[k];
                sum11 += rb * cb1[k];
                sum12 += rb * cb2[k];
                sum13 += rb * cb3[k];
                rb = b[i+128+WB*(k+ks+1)];
                sum10 += rb * cb0[k+1];
                sum11 += rb * cb1[k+1];
                sum12 += rb * cb2[k+1];
                sum13 += rb * cb3[k+1];
          }
          __syncthreads();
    }
    a[i+WA*j] = sum00;
    a[i+WA*(j+1)] = sum01;
    a[i+WA*(j+2)] = sum02;
    a[i+WA*(j+3)] = sum03;
    a[i+128+WA*j] = sum10;
    a[i+128+WA*(j+1)] = sum11;
    a[i+128+WA*(j+2)] = sum12;
    a[i+128+WA*(j+3)] = sum13;
}
```

6.2 Συνέλιξη πινάκων

Στη συνέλιξη πινάκων έγινε προσπάθεια προσέγγισης του βελτιστοποιημένου κώδικα της βιβλιοθήκης FLCC v1.0 που κάνει συνελίξεις, και αναπτύχθηκε από τους φοιτητές του τμήματος Γεώργιο Παπαμακάριο και Γεώργιο Ρίζο. Ο κώδικας της βιβλιοθήκης FLCC περιλαμβάνει τρεις ομάδες εμφωλευμένων βρόχων από τις οποίες οι δύο πρώτες χρησιμοποιούνται για τη φόρτωση των δεδομένων του πηγαίου πίνακα και του πυρήνα της συνέλιξης στη διαμοιραζόμενη μνήμη, και η τρίτη ομάδα βρόχων εκτελεί τις πράξεις. Η μηχανή παραγωγής κώδικα, αφήνει την ομάδα βρόχων που φορτώνει τον πηγαίο πίνακα στη διαμοιραζόμενη μνήμη ως έχει, και ενσωματώνει τις άλλες δύο σε μία. Αυτή η τροποποίηση ισχύει και για τα δύο είδη συνέλιξης που εξετάστηκαν – δύο και τριών διαστάσεων.

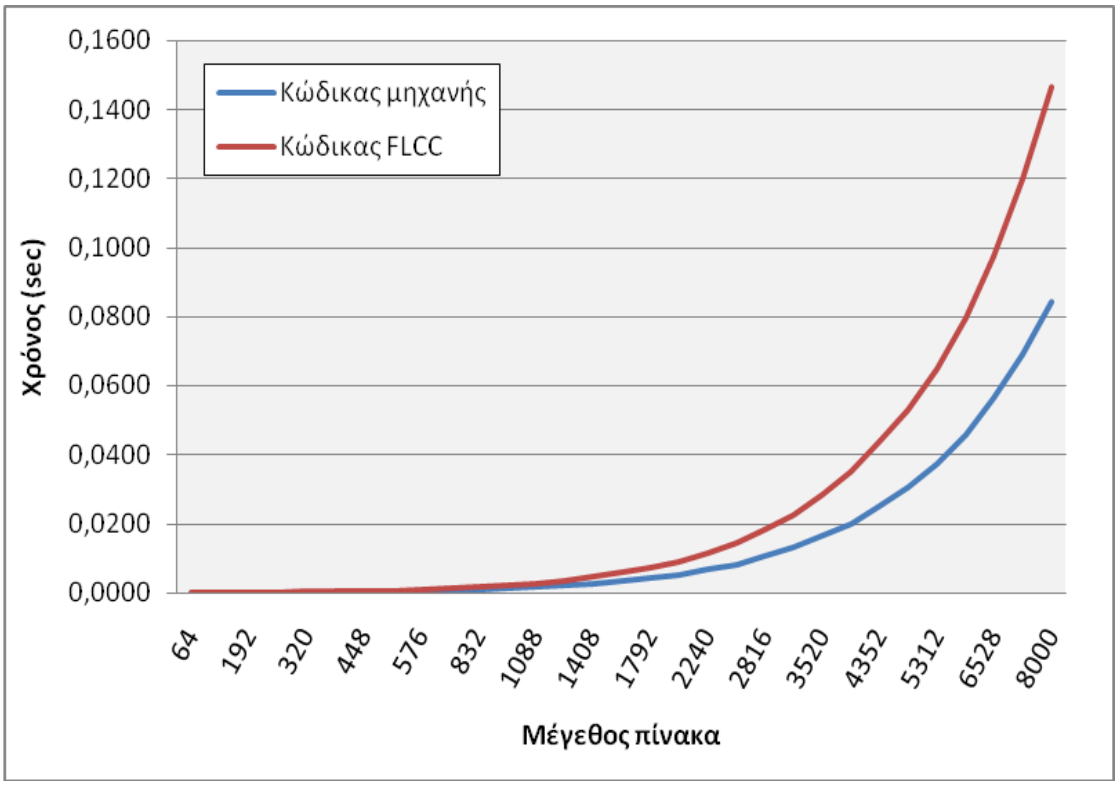
6.2.1 Συνέλιξη δύο διαστάσεων

Για τη συνέλιξη δύο διαστάσεων έγιναν πειράματα ώστε να μετρηθεί η απόδοση του κώδικα μηχανής για πυρήνες μεγέθους από 2x2 έως 32x32. Εδώ παραθέτουμε την απόδοση του κώδικα συγκρινόμενη με αυτή του κώδικα της βιβλιοθήκης FLCC για πυρήνες μεγέθους 8x8 και 16x16 και για μεγέθη πηγαίων πινάκων από 64x64 έως 8000x8000. Παρατίθεται ο πίνακας με τις μετρήσεις χρόνων του κώδικα μηχανής και της βιβλιοθήκης FLCC για μεταβλητά μεγέθη πινάκων, και ακολουθεί η οπτικοποιημένη αναπαράστασή τους.

Μέγεθος πίνακα / block	**64**	**512**	**1088**	**2496**	**5312**	**8000**
8x8	0,0000	0,0004	0,0016	0,0082	0,0371	0,0842
	0,0000	0,0006	0,0028	0,0143	0,0646	0,1464
16x16	0,0001	0,0021	0,0092	0,0472	0,2123	0,4803
	0,0001	0,0016	0,0068	0,0349	0,1570	0,3553

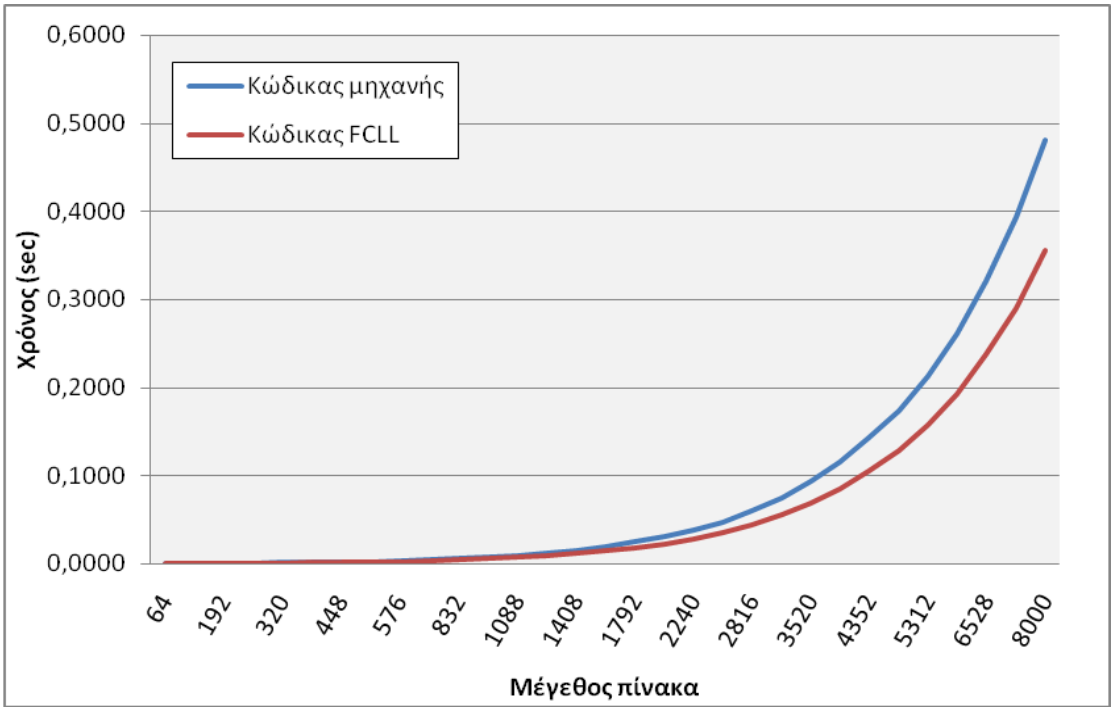
Πίνακας 6.2.1.1 Χρόνοι εκτέλεσης συνέλιξης για μεταβλητό μέγεθος πηγαίου πίνακα και πυρήνα. Κώδικες μηχανής και FLCC v1.0

Όπως φαίνεται στο παρακάτω σχήμα ο κώδικας μηχανής υπερτερεί σε απόδοση σε σχέση με τον κώδικα της βιβλιοθήκης για πυρήνα μεγέθους 8x8:



Εικόνα 6.2.1.1 Χρόνοι εκτέλεσης για μεταβλητά μεγέθη πινάκων και πυρήνα 8x8

Αντίθετα για τη συνέλιξη με μέγεθος πίνακα πυρήνα 16x16, ο κώδικας μηχανής υστερεί σε απόδοση σε σχέση με τον κώδικα της βιβλιοθήκης:



Εικόνα 6.2.1.2 Χρόνοι εκτέλεσης για μεταβλητά μεγέθη πινάκων και πυρήνα 16x16

Παρατίθενται τα cuda kernels της μηχανής και της βιβλιοθήκης FLCC v1.0 των Γ. Παπαμακάριου και Γ. Ρίζου:

Kernel μηχανής

```
__global__ void optKernel(float *a, float *b, float *c, int TH, int TW,
int SW, int BW){

   int x = threadIdx.x;
   int y = threadIdx.y;

   int u0 = __mul24(blockIdx.x, 16);
   int v0 = __mul24(blockIdx.y, 16);

   const int PH = 16 + TH - 1;
   const int PW = 16 + TW - 1;

   __shared__ float bs[961];
   __shared__ float cs[256];

   const int i_idx = u0 + (v0 * BW);

   int p_idx = x + (y * PW);
   int t_idx = (TH * TW) - 1;

   float sum[1];

   for (int j=0; j < PH; j += 16) {
         for (int i=0; i < PW; i += 16) {
```

```
                bs[__mul24((j + y), PW) + i + x] = b[i_idx + __mul24((j +
y), BW) + i + x];
          }
   }

   for (int j = 0; j < TH; j += 16) {
          for (int i = 0; i < TW; i += 16) {

                const int jtwi = __mul24((j + y), TW) + i + x;
                cs[jtwi] = c[jtwi];

                __syncthreads();

                for (int jj=0; jj<i + 16; jj++) {
                      for (int ii=0; ii<i + 16; ii++) {
                            sum[0] += bs[p_idx + ii + __mul24(jj, PW)] *
cs[t_idx - ii - __mul24(jj, TW)];
                      }
                }
          }
   }

   a[(u0+x) + __mul24((v0+y), SW)] = sum[0];
}
```

Kernel FLCC

```
__global__ void optKernel(float *outputDev, float *imageDev, float
*templateDev, int TH, int TW, int SW, int BW){

   const int u0 = __mul24(blockIdx.x, blockDim.x);
   const int v0 = __mul24(blockIdx.y, blockDim.y);

   const int PH = 16 + TH - 1;
   const int PW = 16 + TW - 1;

   const int P_SIZE = PH*PW;
   const int T_SIZE = TH*TW;

   __shared__ float panelSh[961];

   const int i_idx = u0 + __mul24(v0, BW);

   for (int j=threadIdx.y; j < PH; j += 16) {
          for (int i=threadIdx.x; i < PW; i += 16) {

                panelSh[__mul24(j, PW) + i] = imageDev[i_idx + __mul24(j,
BW) + i];
          }
   }

   __shared__ float templateSh[256];
```

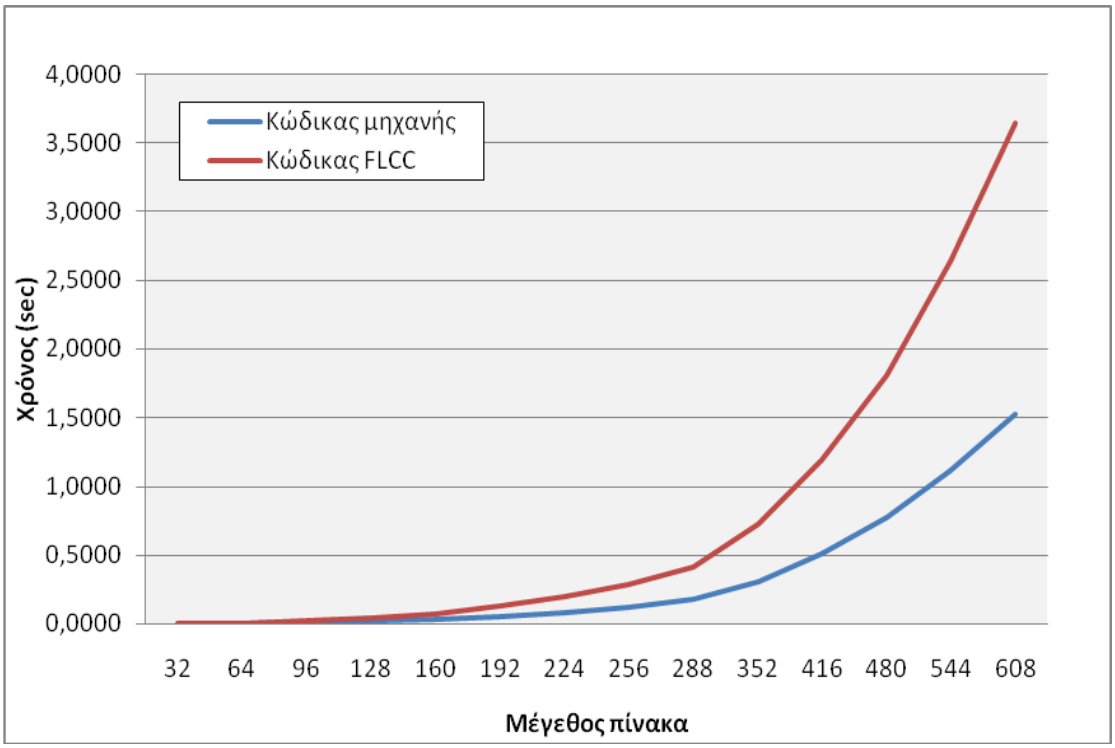
```
   for (int j = threadIdx.y; j < TH; j += 16) {
         for (int i = threadIdx.x; i < TW; i += 16) {

               const int jtwi = __mul24(j, TW) + i;
               templateSh[jtwi] = templateDev[jtwi];
         }
   }

   __syncthreads();

   float conv = 0.0;
   int p_idx = threadIdx.x + __mul24(threadIdx.y, PW);
   int t_idx = T_SIZE - 1;

   for (int j=0; j<TH; j++) {
         for (int i=0; i<TW; i++) {
               conv += panelSh[p_idx + i] * templateSh[t_idx - i];
         }
         p_idx += PW;
         t_idx -= TW;
   }

   outputDev[(u0+threadIdx.x) + __mul24((v0+threadIdx.y), SW)] = conv;
}
```

6.2.2 Συνέλιξη τριών διαστάσεων

Στη συνέλιξη τριών διαστάσεων, τα πειράματα που έγιναν αφορούν μεγέθη πηγαίου πίνακα από 32x32x32 – 608x608x608 και μεγέθη πίνακα πυρήνα 8x8x8. Δεν μπορούμε να ορίσουμε blocks μεγαλύτερου μεγέθους, λόγω του περιορισμού του υλικού για τον αριθμό των threads ανά block. Παρουσιάζονται ο πίνακας με τις μετρήσεις και το αντίστοιχο διάγραμμα. Ο κώδικας μηχανής υπερτερεί έναντι του κώδικα της βιβλιοθήκης FLCC:

Μέγεθος πίνακα / block	32	128	192	288	416	608
8x8x8	0,0005	0,0168	0,0531	0,1731	0,5052	1,5263
	0,0012	0,0396	0,1253	0,4066	1,1934	3,6390

Πίνακας 6.2.2.1 Χρόνοι συνέλιξης για μεταβλητό μέγεθος πηγαίου πίνακα και πυρήνα 8x8x8. Κώδικες μηχανής και FLCC v1.0



Εικόνα 6.2.2.1 Χρόνοι εκτέλεσης για μεταβλητά μεγέθη πινάκων και πυρήνα 8x8x8

Παρατίθενται τα cuda kernels της μηχανής και της βιβλιοθήκης FLCC v1.0 των Γ.Παπαμακάριου και Γ.Ρίζου

Kernel μηχανής

```
__global__ void optKernel(float *a, float *b, float *c, int TH, int
TW, int TD, int SW, int SD, int BW, int BD){


      int x = threadIdx.x;
      int y = threadIdx.y;
      int z = threadIdx.z;

      int u0 = __mul24(blockIdx.x, 8);
      int v0 = __mul24(blockIdx.y, 8);
      int w0 = __mul24(blockIdx.z, 8);

      int PH = 8 + TH - 1;
      int PW = 8 + TW - 1;
      int PD = 8 + TD - 1;

      int T_SIZE = TH * TW * TD;

      __shared__ float bs[3375];
      __shared__ float cs[512];
```

```
      float sum[0];

int i_idx = u0 + __mul24(v0, SD) + __mul24(w0, __mul24(SD, SW));
      int p_idx = x + __mul24(y, SD) + __mul24(z, __mul24(SD, SW));
      int t_idx = T_SIZE - 1;


      for (int k = 0; k < PH; k += 8) {
            for (int j = 0; j < PW; j += 8) {
                  for (int i = 0; i < PD; i += 8) {

                        bs[i+x+ __mul24(j+y, PD) + __mul24(k+z,
__mul24(PD,PW))] =
b[i_idx+i+x+__mul24(j+y,BD)+__mul24(k+z,__mul24(BD,BW))];

                  }
            }
      }

      for (int k = 0; k < TH; k += 8) {
            for (int j = 0; j < TW; j += 8) {
                  for (int i = 0; i < TD; i += 8) {

                        int ijk = i + x + __mul24(j + y, TD) +
__mul24(k + z, __mul24(TD, TW));

                        cs[ijk] = c[ijk];
                        __syncthreads();

                        for (int kk = k; kk < k + 8; kk++) {
                              for (int jj = j; jj < j + 8; jj++) {
                                    for (int ii = i; ii < i + 8;
ii++) {
                                          sum[0] += bs[p_idx + i +
(jj * PD) + (kk * PD * (PW - TD))] * cs[t_idx - i - (jj * TD)];

                                    }
                              }
                        }
                  }
            }
      }


      a[(u0 + x) + __mul24((v0 + y), SD) + __mul24((w0 + z),
__mul24(SD, SW))] = sum;

}
```

Kernel FLCC

```
__global__ void conv3d (int *imageDev, int *templateDev, int
*outputDev, int TH, int TW, int TD, int SW, int SD, int BW, int BD){

      int x = threadIdx.x;
      int y = threadIdx.y;
      int z = threadIdx.z;

      int u0 = __mul24(blockIdx.x, blockDim.x);
      int v0 = __mul24(blockIdx.y, blockDim.y);
      int w0 = __mul24(blockIdx.z, blockDim.z);

      const int PH = 8 + TH - 1;
      const int PW = 8 + TW - 1;
      const int PD = 8 + TD - 1;

      const int P_SIZE = PH * PW * PD;
      const int T_SIZE = TH * TW * TD;

      __shared__ float panelSh[3375];

      const int i_idx = u0 + __mul24(v0,BD) +
__mul24(w0,__mul24(BD,BW));


      for (int k = threadIdx.z; k < PH; k += 8) {
            for (int j = threadIdx.y; j < PW; j += 8) {
                  for (int i = threadIdx.x; i < PD; i += 8) {

                        panelSh[i+ __mul24(j, PD) + __mul24(k,
__mul24(PD,PW))] =
imageDev[i_idx+i+__mul24(j,BD)+__mul24(k,__mul24(BD,BW))];

                  }
            }
      }

      __shared__ float templateSh[512];

      for (int k = threadIdx.z; k < TH; k += 8) {
            for (int j = threadIdx.y; j < TW; j += 8) {
                  for (int i = threadIdx.x; i < TD; i += 8) {
                        const int ijk = i + __mul24(j, TD) +
__mul24(k, __mul24(TD, TW));

                        templateSh[ijk] = templateDev[ijk];
                  }
            }
      }
```

```
      __syncthreads();

      float conv = 0.0;
      int p_idx = x + __mul24(y,PD) + __mul24(z, __mul24(PD,PW));
      int t_idx = T_SIZE - 1;

      for (int k = 0; k < TH; k++) {
            for (int j = 0; j < TW; j++) {
                  for (int i = 0; i < TD; i++) {
                   conv += panelSh[p_idx + i] * templateSh[t_idx - i];
                  }

                   p_idx += PD;
                   t_idx -= TD;
            }
            p_idx += PD * (PW - TD);
      }

      outputDev[(u0 + x) + __mul24((v0 + y), SD) + __mul24((w0 + z),
__mul24(SD,SW))] = conv;
}
```

7 Συμπεράσματα

Η ραγδαία εξέλιξη των καρτών γραφικών τα τελευταία χρόνια, καθώς και η ανάπτυξη προγραμματιστικών μοντέλων που επιτρέπουν την αποτελεσματική χρησιμοποίηση των πόρων που αυτές προσφέρουν, παρουσιάζει νέες δυνατότητες επίλυσης προβλημάτων, που σε συμβατικές CPU η αντιμετώπισή τους θα ήταν από πολύ δύσκολη έως αδύνατη.

Αυτό που έχει πλέον όμως εξέχουσα σημασία, δεν είναι μια απλή επίλυση ενός προβλήματος με τη χρήση μιας GPU. Σταδιακά το ενδιαφέρον μεταφέρεται στην όσο το δυνατόν αποδοτικότερη αντιμετώπιση τέτοιων προβλημάτων, για μεγιστοποίηση του πλεονεκτήματος που προσφέρεται από μια τέτοια αρχιτεκτονική. Όπως αναφέρθηκε πρωτύτερα, μια GPU διαθέτει κάποιους συγκεκριμένους πόρους για χρήση, όπως οι πολυεπεξεργαστές και τα threads που υποστηρίζει ο καθένας, οι μνήμες που διακρίνονται στις: καθολική, διαμοιραζόμενη και αρχείο καταχωρητών για κάθε ένα thread, όπως και η ταχύτητα διαμεταγωγής των δεδομένων που υποστηρίζει το κάθε ένα είδος μνήμης. Από το πρόβλημα που εξετάσαμε φάνηκε ότι είναι καθοριστικής σημασίας ο τρόπος που θα χρησιμοποιηθούν αυτοί οι πόροι είτε ξεχωριστά, είτε σε συνδυασμό.

Τεράστια είναι τα οφέλη – από την άποψη της ταχύτητας εκτέλεσης – από τη σωστή χρήση της διαμοιραζόμενης μνήμης. Η μνήμη αυτή είναι σχεδιασμένη έτσι ώστε να επιτρέπει ταχεία προσπέλαση και κατά συνέπεια όταν φορτώνονται δεδομένα σε αυτήν για την εκτέλεση υπολογισμών του προβλήματος, μειώνεται κατά πολύ η ‘κίνηση’ στην καθολική μνήμη. Είναι βέβαια σημαντικό όλα τα threads, να έχουν μια διατεταγμένη πρόσβαση στη μνήμη, για να αποφεύγονται οι συγκρούσεις προσπέλασης, που εισάγουν καθυστέρηση.

Πολύ σημαντικές στην απόδοση είναι και οι βελτιστοποιήσεις των βρόχων. Κατά κανόνα, ένα αλγεβρικό πρόβλημα που απαιτεί τη χρήση μιας GPU για να επιλυθεί, περιλαμβάνει αξιοσημείωτο αριθμό από βρόχους. Οι υπολογισμοί των δεικτών των βρόχων, και οι έλεγχοι τερματισμού τους μπορεί να μη φαίνονται σημαντικοί, στην πραγματικότητα όμως εισάγουν καθυστέρηση της τάξης μέχρι και μισού δευτερολέπτου. Το blocking των βρόχων σε συνδυασμό με το unrolling δίνουν θεαματικά αποτελέσματα για τους εξείς λόγους: α)κατάτμηση του προβλήματος σε

μικρότερα κομμάτια, με αποτέλεσμα να είναι δυνατή η φόρτωσή τους σε προσωρινή μνήμη και υπολογισμός του μερικού αποτελέσματος σε πολύ μικρότερο χρόνο, β)πολύ λιγότεροι υπολογισμοί των δεικτών των βρόχων καθώς αυξάνεται το βήμα προσαύξησης (blocking), γ)περισσότερα δεδομένα είναι διαθέσιμα στους registers κάθε thread, με αποτέλεσμα να μειώνεται η κίνηση ακόμα και στη διαμοιραζόμενη μνήμη, που είναι πιο αργή από τους registers (unrolling).

Σημαντικό ρόλο παίζει επίσης και η κατάτμηση του αρχικού προβλήματος σε blocks, όπου διαφορετικές διαστάσεις μπορεί να οδηγήσουν σε μη βέλτιστη ανάθεση της δουλειάς δηλαδή σε υποχρησιμοποίηση των διαθέσιμων threads. Θεωρείται δεδομένο όμως, ότι το πρόβλημα είναι αρκετά μεγάλο, ώστε να προσφέρει μεγάλα ποσά δεδομένων και πράξεων, για να είναι δικαιολογημένη η χρήση μιας GPU.

Τέλος παρουσιάζει ενδιαφέρον, η δόμηση των εντολών του πυρήνα, καθώς μια εντολή διακλάδωσης μπορεί να εισάγει σημαντική καθυστέρηση αν τα threads παρουσιάσουν απόκλιση στη ροή εκτέλεσης των εντολών. Αυτό θα έχει σαν αποτέλεσμα, ο μηχανισμός διαχείρισης νημάτων, να οδηγήσει τα νήματα σε σειριακή εκτέλεση περιμένοντας τα αποκλίνοντα να τελειώσουν την εκτέλεσή τους (π.χ όλα τα νήματα με ζυγό αριθμό ID περιμένουν τα νήματα περιττού ID να τελειώσουν την εκτέλεσή τους).

Είναι ευνόητο λοιπόν, ότι το πεδίο αυτό επιδέχεται μεγάλης έρευνας, καθώς οι παράμετροι που εμπλέκονται, - το μέγεθος των προβλημάτων, οι απαιτήσεις ταχύτητας και οι αρχιτεκτονικές GPU, - είναι άκρως μεταβαλλόμενες.

8 Παράρτημα

Εδώ παρατίθενται τα βασικότερα κομμάτια κώδικα της εφαρμογής και πιο συγκεκριμένα μέρος της τάξης CustomParser.java, που είναι ο parser της εφαρμογής, το τμήμα κώδικα της τάξης BurmComplete.java που κάνει το matching επάνω στο δέντρο των βρόχων, με τη μέθοδο getCudaCost() για υπολογισμό του κόστους του παραγόμενου κώδικα, και οι κώδικες της τάξης IRTransformer.java που κάνουν τους μετασχηματισμούς strip-mining, tiling, unrolling και interchanging. Επιπλέον υπάρχουν τμήματα των τάξεων CudaGen.java και Emitter.java, που είναι τάξεις υπεύθυνες για την παραγωγή των cuda statements:

Τάξη ***CustomParser.java***: Αυτή διαβάζει ένα πηγαίο αρχείο κώδικα, και δημιουργεί το δέντρο της ενδιάμεσης αναπαράστασης:

```
public TL1INode Parse() {
    TL1INode tree = null;

     try {
         tree = Compile();
         tree = FormatTree(tree);
     } catch (ParseException e) {

     } catch (Exception ee) {
      System.err.println("CustomParser.java: " + ee.getMessage());
    }
            return tree;
    }

  private TL1INode Compile() throws Exception, ParseException {

        String code = "";
        TL1INode node = null;
        boolean endFlag = false;

        while (!endFlag) {
             token = consumeToken();

             switch (token.kind) {
                 case FOR:
                 case STMT:
                     node = Statement(token);
                     break;
                 case EOF:
                     endFlag = true;
                     break;
                }
            }
            return node;
   }
```

```
final public TL1INode Statement(Token token) throws ParseException,
Exception {

        TL1INode result = null;

  //this parser only works for code containing for statements
        switch (token.kind) {
            case FOR:
                result = ForStmt();
                forNodes.add(result);
                break;
            case STMT:
                result = Stmt(token);
                stmts.add(result);
                break;
            default:
                throw new ParseException();
        }

        return result;
    }

final public TL1INode ForStmt() throws ParseException, Exception {

        For forr;
        TL1INode body, result;

        Token tokken = null;

 //omit the left parenthesis and the index integer declaration
        consumeToken();
        consumeToken();
        forr = Expression();

        if (forr.getIndex().equalsIgnoreCase("z")) {
            tokken = new Token(FOR, "forZ");
            ForFound++;
        } else {
            tokken = new Token(FOR, "for" + ForFound++);
        }

        body = Statement(consumeToken());

        result = new TL1INode(tokken, body, null);
        result.setObjectContent(forr);

        if (forr.getForm().equals("N1")) {
            result.setRoot(true);
        }

        return result;
}

final public TL1INode Stmt(Token tok) throws Exception {

     StatementList list = new StatementList("statementList");

        Token tokken = new Token(STMT, "stmt");
        String statement = "";
        TL1INode node = new TL1INode(tokken);
        TL1INode body = null;
```

```
        //finish the rest of the tokens
        while (true) {
            if (tok == null || tok.toString().equals("")) {
                break;
            } else if (tok.kind == FOR) {
                body = Statement(tok);
                break;
            } else if (tok.toString().equals(";") ||
tok.toString().equals("\\n")) {
                statement += tok + "\n";
                Assign assign = new Assign(statement);
                list.addElement(assign);

                //initialize the buffer
                statement = "";

                tok = consumeToken();
                continue;
            } else if (tok.toString().equals("}") ||
tok.toString().equals("{")) {
                statement += "\n";
                tok = consumeToken();
                continue;
            }
            statement += tok.toString() + " ";
            tok = consumeToken();
        }

        node.setObjectContent(list);
        node.setLeftChild(body);
        return node;
}

/**
 * This method reads tokens from the input source file and
 *  creates a For object properly initialized
 * @return - returns the constructed for loop
 */
final public For Expression() throws Exception {

        String step = "";

        String loopindex = consumeToken().toString();
        consumeToken();
        String lowerBound = consumeToken().toString();
        consumeToken(); //this is the semicolon
        consumeToken(); //this is the index again
        String test = consumeToken().toString();
        String upperBound = consumeToken().toString();
        consumeToken(); //omit semicolon
        consumeToken(); //omit the index
        String operatorOnStep = consumeToken().toString();


        String next = consumeToken().toString();
        if (next.equals("=")) {
            step = consumeToken().toString();
        } else if (operators.contains(next)) {
            step = "1";
        }
```

```
        //omit the right parenthesis
        consumeToken();
        //get the identifier from the source file to decide if this
is a simd loop
        String simdDecl = consumeToken().toString();
        if (simdDecl.equalsIgnoreCase("simd")) {
            consumeToken();
        }

        For forr = new For(loopindex, lowerBound, test, upperBound,
operatorOnStep, step, ForFound);
        forr.setSimd((simdDecl.equalsIgnoreCase("simd")) ? true :
false);

        if (loopindex.equalsIgnoreCase("z")) {
            forr.setForm("NZ");
        } else {
            forr.setForm("N" + ForFound);
        }
        return forr;
    }

    private Token consumeToken() {

        Token tokken = null;

        try {
            tokken = source.getNextToken();
            String serialized = tokken.toString();

            if (serialized.equals("for")) {
                tokken.kind = FOR;
            } else if (serialized.equals("")) {
                //end the parsing process
                tokken.kind = EOF;
            } else {
                tokken.kind = STMT;
            }
        } catch (TokenMgrError e) {
            try {

//System.err.println("CustomParser.java:Parsing ended. Terminating
process");
                System.out.println("CustomParser.java : " +
e.getMessage());
                //System.out.println("...continuing manual parsing");
                String value = new String(new
char[]{source.getStream().readChar()});
                //System.out.println(value);

                if (value.equals("[")) {
                    tokken = Token.newToken(LBRACKET, value);
                } else if (value.equals("]")) {
                    tokken = Token.newToken(RBRACKET, value);
                } else if (value.equals("%")) {
                    tokken = Token.newToken(MODULO, value);
                }
            } catch (IOException ee) {
                System.out.println(ee.getMessage());
            }
```

```
            }
            return tokken;
        }
```

Τάξη ***BurmComplete.java***: Σε αυτό το σημείο, καθώς σαρώνεται το αρχικό δέντρο του κώδικα, γίνεται το ταίριασμα των δενδρικών προτύπων επάνω στους κόμβους, και ξεκινάει η διαδικασία μετασχηματισμού του δέντρου. Η μέθοδος *getCudaCost()* είναι αυτή που ξεκινάει τη διαδικασία δημιουργίας cuda κώδικα και εκτέλεσής του και ανακτά του κόστους του.

```
   case FOR: {
        // FOR(FOR(stmt, stmt), stmt)
        if (node.getArity() == 1 && node.getOperator() == FOR
                        && node.getNthChild(0).getOperator() == FOR){


        iCost = getCudaCost(node.m_node);


        TL1INode root = node.getNode();
        root.setLeftChild(node.getNthChild(0).getNode());

        if (recursiveCalls) {
          JBurgAnnotation newAnnotationTree = null;
          TL1INode newTree = transformGeneric.copy(root, root.getParent());
          newAnnotationTree = transformGeneric.TransformLoops(newTree,
node);

          if (newAnnotationTree.getCost(statement_NT) <
node.getCost(statement_NT)) {
      //overwrite the old tree
      node.setNode(newAnnotationTree.getNode());

      node.setNthChild(newAnnotationTree.getNthChild(0), 0);

      //node.setNthChild(newAnnotationTree.getNthChild(1), 1);
                                  node.reset(statement_NT,
      newAnnotationTree.getCost(statement_NT), 29);
   }
  }
    return;
}

       // FOR(stmt, stmt)
       if (node.getArity() == 1 && node.getOperator() == FOR) {

           iCost = getCudaCost(node.m_node);

         if (recursiveCalls) {
```

```
      TL1INode root = node.getNode();
      JBurgAnnotation newAnnotationTree = null;
      TL1INode newTree = transformGeneric.copy(root,
 root.getParent());
      newAnnotationTree = transformGeneric.TransformLoops(newTree,
 node);

      if (newAnnotationTree.getCost(statement_NT) <
 node.getCost(statement_NT)) {
        //overwrite the old tree
        node.setNode(newAnnotationTree.getNode());
        node.reset(statement_NT,
 newAnnotationTree.getCost(statement_NT), 31);
    }
   }
  }
  break;
}
```

Μέθοδος getCudaCost:

```
  private double getCudaCost(jburg.compiler.tl1.ir.TL1INode p) {
      double cost = Double.MAX_VALUE;

      if(transformGeneric.treeDepth(p)>=2&&transformGeneric.treeInNor
      malForm(p)) {
          cost = 0.0;

          try {
              this.cGen.dumpCode(p);

              NumberFormat formatter = new DecimalFormat("##");

              double response = this.cGen.Execute();
              System.out.print(formatter.format(response) + " ");
              cost = response;
          } catch (NumberFormatException e) {
              return Double.MAX_VALUE;
          } catch (ShellException ee) {
            //returning  max  value,  means  that  the  current
            //experiment is being ignored, due to
            //a very bad processing time
              return Double.MAX_VALUE;
          }
```

```
            }
            return cost;
        }
```

Τάξη ***IRTransformer.java***: Παρατίθεναι οι μέθοδοι *StripMineAllCombinations* και *Stripmine, UnrollAllCombinations* και *Unroll, Tile* και *InterchangeInward*. Η πρώτη σαρώνει με αναδρομικό τρόπο τους κόμβους του δέντρου διερχόμενη έτσι από όλους τους πιθανούς συνδυασμούς μεγεθών block, και η δεύτερη πραγματοποιεί ουσιαστικά τη διαδικασία strip-mining στους βρόχους. Παρόμοια διαδικασία ακολουθείται και από τις μεθόδους που κάνουν tiling και unrolling.

```
    /**
     * Performs all possible combinations of strip mining over a perfect
     *loop nest, using recursion to move to child nodes.
     *
     * @param parent – The root of the tree
     * @param child is the node currently under transformation
     * @return – The transformed tree
     */
    private JBurgAnnotation StripmineAllcombinations(TL1INode parent,
TL1INode child) {

        TL1INode root = parent;
        TL1INode temp = null;
        TL1INode cheapestTree = copy(parent, parent.getParent());

        JBurgAnnotation stripMined = null;

        //avoid the statement nodes in the tree
        if (child == null || !child.toString().contains("for")) {
            return burm.label(root);
        }

        For forr = (For) parent.getObjectContent();

        if (contains(forr.getForm(), normalForm3)) {
            for (int i = STRIP_MINE_LOWER_BOUND; i <=
STRIP_MINE_UPPER_BOUND; i *= 2) {

                //check if the template allows strip mining on this node
                if (contains(((For) child.getObjectContent()).getForm(),
SMINE_LOOPS)) {
                    //restore the temp before each iteration starts
                    temp = copy(parent, parent.getParent());
                    stripMined = Stripmine(temp, child, "" + i);
                } else {
                    return StripmineAllcombinations(parent,
child.getNthChild(0));
                }
```

```
                if (stripMined == null) {
                    continue;
                }

                if (_UNROLLING == ON) {
                    JBurgAnnotation unrolled =
UnrollAllCombinations(stripMined.getNode(), stripMined.getNode(), 1
/*child.getNthChild(0)*/);
                    //decide which tree is the cheapest one
                    stripMined = cheapestTree(unrolled, stripMined);
                }
                JBurgAnnotation smined =
StripmineAllcombinations(stripMined.getNode(), child.getNthChild(0));

                if (stripMined.getCost(BurmComplete.statement_NT) <
smined.getCost(BurmComplete.statement_NT)) {

                    System.out.println("\n\n " +
stripMined.getCost(BurmComplete.statement_NT) + " is better cost.
Overwrite(in StripmineAllcombinations)\n\n");
                    cheapestTree = copy(stripMined.getNode(),
stripMined.getNode().getParent());
                    //minCost =
stripMined.getCost(BurmComplete.statement_NT);
                    cheapestTrees.add(stripMined);
                    //traverseTree(stripMined);
                }
            }
        }
        return burm.label(cheapestTree);
    }
```

Μέθοδος Stripmine:

```
    /**
     * Performs strip mining. It increases the step of the loop at *depth
     *'depth', and interpolates a new loop between
     * 'node' and 'node + 1'. The loop is strip mined to the width of
     *'step'.
     *
     * @param node. The outer for loop
     * @param depth. The nest level of the for loop we are going to *strip
     *mine
     * @param step. The block width that is going to be applied by strip
     *mining
     * @return The transformed tree
     */
    public JBurgAnnotation Stripmine(TL1INode parent, TL1INode currentNode,
String step) {

        JBurgAnnotation created = null;
```

```
        int counter = 0;
        int simdWidth = 0;
        //TL1INode parent = copy(initial, initial.getParent());

        //keep a pointer to the initial node
        TL1INode root = parent;
        TL1INode temp = copy(root, root.getParent());

        //find the correct loop to apply strip mining
        while (parent.getNthChild(0) != null) {

            try {
                For forr = (For) parent.getObjectContent();
                For currentFor = (For) currentNode.getObjectContent();
                //System.out.println("Checking loop " + currentFor + "
counter = " + counter + " form = " + forr.getForm() + "-");

                if (forr.getForm().equals(currentFor.getForm())) {
                    break;
                }

            } catch (ClassCastException e) {
            }
            parent = parent.getNthChild(0);
        }

        System.out.println("STRIP MINING " + parent + " step " + step);

        //save the child node
        TL1INode leftChild = parent.getNthChild(0);
        TL1INode rightChild = parent.getNthChild(1);

        //modify the step of the current loop. Use the copy constructor
(For) to properly copy an object's properties
        Object obj = parent.getObjectContent();

        //check if strip mining is plausible
        if (!ValidBlock(obj, temp, step, "stripmining")) {
            //System.out.println("INVALID STRIP MINING");
            return null;
        }

        if (obj instanceof For) {
            For forr = new For((For) obj);
            forr.setStep(step);
            /**
             * we change the form of this for in order to know later that
it is a N(ormal) for loop, which has been strip mined.
             * So the final form will be N1|2|3-SM. If a skewing has been
preceded, we must take some care of the upper bound
             * of the newly generated for.
             */
            forr.setForm(forr.getForm() + "-SM");
```

```
            String lowerBound = forr.getIndex();
            String upperBound = forr.getIndex() + " + " + forr.getStep();

            if (forr.getForm().contains("SK")) {
                String bound = forr.getUpperBound();
                upperBound = "(" + upperBound + " > " + bound + ") ? " +
upperBound + " : " + bound;
            }

            parent.setObjectContent(forr);
            //discard any statement nodes the current loop carries
            parent.setRightChild(null);

            //create the new node. This is the new loop added to the tree.
We don't need a right child attached to the new node
            TL1INode newNode = new TL1INode(new Token(BurmComplete.FOR,
"for"), leftChild, null /*rightChild*/);

            //create the new loop
            For newFor = new For(forr.getIndex() + forr.getIndex(),
lowerBound, upperBound, "1", forr.getOrder());
            newFor.setSimd(forr.isSimd());
            if (forr.getIndex().equalsIgnoreCase("z")) {
                newFor.setForm("SMZ");
            } else {
                newFor.setForm("SM" + forr.getOrder());
            }
            newNode.setObjectContent(newFor);

            //attach the new node (for) to the current one
            parent.setLeftChild(newNode);
            created = InterchangeInward(root, parent);
            emitter.EmitCudaStatements(STRIP_MINING, created.getNode(),
parent);
        }

        //System.out.println("INTERCHANGED ");
        created = burm.label(root);
        //traverseTree(created);

        //we just need to match a rule and calculate the cost for this
strip mining. We don't want any other recursive calls
        if (treeInNormalForm(root) && templateApplied(SMINE_TEMPLATE,
created.getNode())) {

            if (!validBlocksSpecified(root)) {
                return created;
            }

            created = burm.label(root);
            //traverseTree(created);
            //created = normalize(created.getNode());
            burm.computeTreeCost(created, false);
```

```
        }

        return created;
    }
```

Μέθοδος Tile. Σαρώνει αναδρομικά το δέντρο, περνώντας από όλα τα δυνατά μεγέθη block.

```
    private JBurgAnnotation Tile(TL1INode parent) {

        double minCost = Double.MAX_VALUE;
        TL1INode cheapestTree = null;
        TL1INode root = parent;

      for (int i = TILE_LOWER_BOUND; i <= TILE_UPPER_BOUND; i *= 2) {

            JBurgAnnotation tiled = Tile(parent, i);

            if (tiled == null) {
                continue;
            }

            if (_UNROLLING == ON) {
                JBurgAnnotation unrolled =
UnrollAllCombinations(tiled.getNode(), tiled.getNode(), 2);
                tiled = cheapestTree(tiled, unrolled);
            }

            if (tiled.getCost(BurmComplete.statement_NT) < minCost) {
                System.out.println("\n\n " +
tiled.getNode().getObjectContent() + " has better cost. Overwrite(in
Tile)\n\n");

                cheapestTree = copy(tiled.getNode(),
tiled.getNode().getParent());
                cheapestTrees.add(tiled);
                minCost = tiled.getCost(BurmComplete.statement_NT);
            }
            //traverseTree(tiled);
        }

        return burm.label(cheapestTree);
    }
```

Μέθοδος Tile. Αυτή πραγματοποιεί την διαδικασία tiling επάνω στους κόμβους του δέντρου. Καλείται από την παραπάνω αναδρομική Tile.

```
    /**
     * Transforms a loop nest of depth n into one of depth 2n
     *
     * @param node - The node to be tiled
     * @param tileSize
     * @return - The transformed tree
     */
    private JBurgAnnotation Tile(TL1INode node, int tileSize) {

        System.out.println("TILING " + tileSize);
        JBurgAnnotation created = null;

        TL1INode temp = copy(node, node.getParent());
        TL1INode copy = copy(node, node.getParent());
        TL1INode root = copy;

        while (copy != null) {
            Object obj = copy.getObjectContent();

            if (!ValidBlock(obj, temp, "" + tileSize, "tiling")) {
                //System.out.println("INVALID TILE " + obj);
                copy = copy.getNthChild(0);
                continue;
            }

            //Tiling is applied only on the initial loops
            if (obj instanceof For && ((For) obj).getForm().contains("N"))
{

                //get the child of the current node
                TL1INode inner = copy.getNthChild(0);

                //modify the step of the current loop
                For forr = new For((For) obj);
                forr.setStep("" + tileSize);
                forr.setForm(forr.getForm() + "-TL");
                copy.setObjectContent(forr);
                //discard any statement nodes the current loop carries
                copy.setRightChild(null);

                //--create the new tiled ii loop. THIS CHECK ON THE LOOP
BOUNDS CONSUMES A NOTICEABLE AMOUNT OF TIME.
//                String upperBound = "(( " + forr.getIndex() + " + " +
forr.getStep() + " ) < " + forr.getUpperBound()
//                        + ") ? " + forr.getIndex() + " + " +
forr.getStep() + " : " + forr.getUpperBound() + ")";

                String upperBound = forr.getIndex() + " + " +
forr.getStep();
                String index = forr.getIndex() + forr.getIndex();

                //this is the new tiled loop
```

```
                For tiledFor = new For(index, forr.getIndex(), upperBound,
"1", forr.getOrder());
                tiledFor.setSimd(forr.isSimd());
                if (forr.getIndex().equalsIgnoreCase("z")) {
                    tiledFor.setForm("TLZ");
                } else {
                    tiledFor.setForm("TL" + forr.getOrder());
                }

                //--create the new tree nodes that will be holding the ii
and jj loops
                TL1INode tiledNode = new TL1INode(new
Token(BurmComplete.FOR, "for"));
                tiledNode.setObjectContent(tiledFor);

                //------------- RESTORATION PROCESS ---------------
                //index to the last loop of 'node'
                tiledNode.setLeftChild(inner);
                copy.setLeftChild(tiledNode);
                created = InterchangeInward(root, copy);
                if (treeInNormalForm(created.getNode())) {
                    emitter.EmitCudaStatements(TILING, root,
created.getNode());
                }
            }
            copy = copy.getNthChild(0);
        }

        created = burm.label(root);
        //traverseTree(created);
        if (/*(treeInNormalForm(root) && templateApplied(SMINE_TEMPLATE,
root))
                ||*/(treeInNormalForm(root) &&
templateApplied(TILE_TEMPLATE, root))) {

            if (!validBlocksSpecified(root)) {
                return created;
            }

            burm.computeTreeCost(created, false);
            traverseTree(created);
        }

        return created;
    }
```

Μέθοδος UnrollAllCombinations. Όπως και η StripMineAllCombinations, σαρώνει αναδρομικά το δέντρο, για να πετύχει όλους τους δυνατούς συνδυασμούς unrolling.

```
     /**
      * Performs unrolling. For each loop unrolling calculates recursively
      * all the other possible unrollings of the
      * other loops in the nest. The successive unrolling factors are
      *multiples of two.
      *
      * @param parent - The outer for loop
      * @param child - The loop right below parent
      * @param startingFactor - The unrolling factor from which the
unrolling of the current node begins. This saves us time,
     * instead of each level of the recursion to begin from 1, only the
first begins from 1 and the others from 2.
     * @return - The transformed tree
     */
    private JBurgAnnotation UnrollAllCombinations(TL1INode parent, TL1INode
child, int Transformation) {

        //take a copy of parent so as to leave the initial parent intact
        TL1INode copy = copy(parent, parent.getParent());
        TL1INode root = copy;
        TL1INode temp = null;
        TL1INode cheapestTree = copy(copy, copy.getParent());

        JBurgAnnotation unrolled = null;

        //avoid the statement nodes in the tree
        if (child == null || !child.toString().contains("for")) {
            return burm.label(root);
        }

        //perform some necessary checks in order to apply unrolling to the
proper loop
        For forr = (For) child.getObjectContent();
        String form = forr.getForm();

        if ((form.contains("N") && form.contains("SM")) ||
(form.contains("N") && form.contains("TL"))) {
            //System.out.println("found forbidden form "+form + " "+forr);
            return UnrollAllCombinations(copy, child.getNthChild(0), 1);
        }

        for (int i = UNROLL_LOWER_BOUND; i <= UNROLL_UPPER_BOUND; i *= 2) {

            //innerMost unrollings must be aware of the inner unrollings
            temp = copy(copy, copy.getParent());
            unrolled = Unroll(temp, child, "" + i);

            //if unrolling fails do nothing
            if (unrolled == null) {
                continue;
            }

            //go on with the rest of the unrollings
```

```
            JBurgAnnotation unRolled =
UnrollAllCombinations(unrolled.getNode(), child.getNthChild(0), 2);

            if (unrolled.getCost(BurmComplete.statement_NT) <
unRolled.getCost(BurmComplete.statement_NT)) {
                System.out.println("\n\n " +
unrolled.getNode().getObjectContent() + " has better cost. Overwrite(in
UnrollAllCombinations)\n\n");

                cheapestTree = copy(unrolled.getNode(),
unrolled.getNode().getParent());
                cheapestTrees.add(unrolled);
            }
        }

        return burm.label(cheapestTree);
    }
```

Μέθοδος Unroll. Πραγματοποιεί τη διαδικασία unrolling επάνω στους βρόχους-κόμβους του δέντρου καθώς καλείται από την UnrollAllCombinations.

```
    /**
     * Unrolls a given loop.
     *
     * @param node
     * @return the unrolled tree
     */
    public JBurgAnnotation Unroll(TL1INode parent, TL1INode currentNode,
String unrollFactor) {

        //keep a reference to the root of the tree
        //TL1INode temp = copy(parent, parent.getParent());
        TL1INode root = parent;

        //find the loop to unroll
        while (parent.getNthChild(0) != null) {
            try {
                For forr = new For((For) parent.getObjectContent());
                For currentFor = (For) currentNode.getObjectContent();

                //System.out.println("checking " + forr.getForm() + " and "
+ currentFor.getForm());
                if (forr.getForm().equals(currentFor.getForm())) {
                    break;
                }
            } catch (ClassCastException e) {
                //System.err.println(e.getMessage());
            }
            parent = parent.getNthChild(0);
        }
```

```
        //System.out.println("FOR FOUND " + parent);
        Object object = parent.getObjectContent();

        if (object instanceof For) {
            System.out.println("UNROLLING " + ((For) object).getForm() + "
factor " + unrollFactor);

            For forr = new For((For) object);

            if (!validUnrollFactor(forr, unrollFactor)) {
                //an attempt to unroll a loop by a factor greater than its
iteration space results in null
                return null;
            }

            forr.setStep(unrollFactor);
            //System.out.println("setting step "+unrollFactor);
            forr.setForm(forr.getForm() + "-U");
            parent.setObjectContent(forr);
            parent.setRightChild(null);

            //every loop unrolling must be followed by a fixing of the
inner loops' bounds. FixBounds is applied to
            //N3 and N4 loops. Parent is the current node and temp holds
the entire tree
            TL1INode temp = copy(root, root.getParent());
            fixBounds(parent, temp, "N3", "SM3", "TL3", "SM1", "TL1",
"N1");
            fixBounds(parent, temp, "N4", "SM4", "TL4", "SM2", "TL2",
"N2");

            if (treeInNormalForm(temp)) {
                emitter.EmitCudaStatements(UNROLLING, root, parent);
            }
        }

        JBurgAnnotation created = burm.label(root);
        //traverseTree(created);

        //we just need to match a rule and calculate the cost for this
strip mining. We don't want any other recursive calls
        if ((treeInNormalForm(root) && templateApplied(SMINE_TEMPLATE,
root))
                || (treeInNormalForm(root) &&
templateApplied(TILE_TEMPLATE, root))) {

            if (!validBlocksSpecified(root)) {
                return created;
            }

            created = burm.label(root);
            traverseTree(created);
```

```
        burm.computeTreeCost(created, false);
    }

    return created;
}
```

Μέθοδος InterchangeInward.

```
    private JBurgAnnotation InterchangeInward(TL1INode parent,
TL1INode currentNode) {

        TL1INode root = parent;
        int counter = 0;

        //find the node we must work on
        while (parent != null) {

            try {
                For forr = (For) parent.getObjectContent();
                For currentFor = (For)
currentNode.getObjectContent();
                //System.out.println("Checking for2 " + forr + "
counter = " + counter + " form = " + forr.getForm() + "-");

                if (forr.getForm().equals(currentFor.getForm())) {
                    break;
                }
            } catch (ClassCastException e) {
            }
            parent = parent.getNthChild(0);
        }

        Object obj = parent.getObjectContent();
        TL1INode parentPointer = parent; //keep track of the parent
node

        /**
         * This code is looking only for N(ormal) loops to
interchange their strip mined or tiled loops.
         * It groups the loops by SIMD and nonSIMD, by moving down a
loop until it finds nodes with different
         * simd attribute
         */
        if (obj instanceof For && (((For)
obj).getForm().contains("N"))) {
            //keep the current group
            boolean simd = ((For) obj).isSimd();
            TL1INode tobeMoved = parent.getNthChild(0); //the strip
mined loop to be interchanged

            while (parent != null) {
                TL1INode nn = parent.getNthChild(0);
```

```
                //we are on the non simd group and reached the bottom
    of the tree
                if (nn == null || nn.getObjectContent() instanceof
    StatementList) {
                    if (parentPointer.getNthChild(0).getNthChild(0)
    != null) {


    parentPointer.setLeftChild(parentPointer.getNthChild(0).getNthChild(0
    ));
                        tobeMoved.setLeftChild(nn);
                        parent.setLeftChild(tobeMoved);
                    }

                    break;
                }

                For forrr = (For) nn.getObjectContent();

                //we have reached the loop group with different simd
    attribute (the non simd group)
                if (forrr.isSimd() != simd) {

    parentPointer.setLeftChild(parentPointer.getNthChild(0).getNthChild(0
    ));
                    tobeMoved.setLeftChild(nn);
                    parent.setLeftChild(tobeMoved);
                    break;
                }

                parent = parent.getNthChild(0);
            }
        }

        JBurgAnnotation created = burm.label(root);
        return created;
    }
```

Τάξεις ***Emitter.java*** και ***CudaGen.java***: Είναι οι τάξεις που παράγουν cuda statements. Παρατίθενται οι μέθοδοι *EmitCudaStatements()* της τάξης *Emitter,* και *GenerateKernel(), Kerne()* και *UnpackCode()* της τάξης *CudaGen.*

Τάξη *Emitter.java* μέθοδος *EmitCudaStatements*:

```
    public void EmitCudaStatements(int transformation, TL1INode root,
TL1INode current) {

      TL1INode rootPointer = root;
      TL1INode cleanTree = root;
      PickoverLoops(rootPointer);
```

```
        //remove any needless statement nodes left in the tree from any
previous statement generation processes
        CleanTree(cleanTree);

        StatementList smtlistcached = null;
        StatementList smtlistcachedmsums = null;
        //TL1INode buffer = current;

        String loopFormToBeUsed = "", transf = "";

        switch (transformation) {
            case UNROLLING:
                For forr = (For) current.getObjectContent();
                //unroll is taken care of by strip-mining or tiling below.
It creates the unrollFactor
                transf = forr.getForm();

                if (transf.contains("SM")) {
                    EmitCudaStatements(STRIP_MINING, root, current);
                } else if (transf.contains("TL")) {
                    EmitCudaStatements(TILING, root, current);
                } //the nodes are not strip mined
                else {
                    EmitCudaStatements(STRIP_MINING, root, current);
                }
                return;
            case STRIP_MINING:
                //create the statements
                smtlistcached =
GenCachedArrayAccesses_Smined(SIMDNormalLoops, SIMDLoops);
                smtlistcachedmsums =
GenCachedMultiplySums_Smined(nonSIMDNormalLoops, nonSIMDsminedLoops);
                transf = "SM";
                break;
            case TILING:
                //create the statements
                smtlistcached = GenCachedArrayAccesses_Tiled(SIMDLoops);
                smtlistcachedmsums =
GenCachedMultiplySums_Tiled(nonSIMDsminedLoops);
                transf = "TL";
                break;
        }

        current = MoveTo(root, "N" + normalInitialLoops.size());
        loopFormToBeUsed = transf + normalInitialLoops.size();

        System.out.println("Moving to N" + normalInitialLoops.size() + " " +
loopFormToBeUsed);

        //set up the nodes
        TL1INode cached = new TL1INode(new Token(BurmComplete.STMT,
"stmt"));
        cached.setObjectContent(smtlistcached);
```

```
        TL1INode cachedmsums = new TL1INode(new Token(BurmComplete.STMT,
"stmt"));
        cachedmsums.setObjectContent(smtlistcachedmsums);

        current.setRightChild(cached);

        if (smtlistcachedmsums != null) {
            current = MoveTo(root, loopFormToBeUsed);
            //System.out.println("MOVED TO " +
current.getObjectContent().toString());
            current.setLeftChild(cachedmsums);
        }
    }
```

Τάξη ***CudaGen.java***, μέθοδος *GenerateKernel*:

```
           //scan the tree to generate cuda kernel code
           public void GenerateKernel(TL1INode node) {

               For iLoop = null, jLoop = null;
               For iiLoop = null, jjLoop = null;

               TL1INode N1Child = null, N2Loop = null, N2Child = null,
       N3Loop = null, N3Child = null, N4Loop = null, N4Child = null;

               //kernel loop is the first non simd loop in the tree. It is
       N3Loop
               boolean kernelLoopFound = false;

               //instantiate some necessary references
               while (node.getNthChild(0) != null) {
                   Object obj = node.getObjectContent();

                   if (obj instanceof For) {
                       For forr = (For) obj;
                       String form = forr.getForm();

                     if (form.contains("N1")) {
                           iLoop = new For(forr);
                       }
                     else if (form.contains("SM1") || form.contains("TL1"))
                   {
                           iiLoop = new For(forr);
                           N1Child = node;
                   }
                     else if (form.contains("N2")) {
                           jLoop = new For(forr);
                           N2Loop = node;
                   }else if (form.contains("SM2")||form.contains("TL2"))   {
                           jjLoop = new For(forr);
                           N2Child = node;
                    }
```

```
                if(!kernelLoopFound && !forr.isSimd()){
                    N3Loop = node;
                    kernelLoopFound = true;
                    break;
                }
            }
            node = node.getNthChild(0);
        }

        String kern = GenKernelHeader(INITIAL_TREE_DEPTH) + " {";

        String arrayName = "cb";
        String localMemory = "rb";

        //declare the thread and block id variables
      kern += UnpackCode("\t", emitter.GenThreadBlockDeclarations());

     kern += GenSumDeclarations((iiLoop == null) ? iLoop : iiLoop,
 (jjLoop == null) ? jLoop : jjLoop);

        kern += UnpackCode("\t", emitter.GenVariableDefinitions());

        //create the shared memory array declarations
       kern += UnpackCode("\t", emitter.GenCacheArrayDeclarations());

        //main kernel generation
        kern += "\n" + Kernel("\t", N3Loop);

        kern += UnpackCode("\t", emitter.GenAssignSums());

        kern += "\n}\n";

        kernel = kern;
        System.out.println("KERNEL generated CudaGen.java");
    }
```

Μέθοδος *Kernel*:

```
private String Kernel(String indentation, TL1INode n3node) {
        String kern = "";

  if (n3node != null && n3node.getObjectContent() instanceof For) {
     //print itself
     For forr = (For) n3node.getObjectContent();
     kern += "\n" + indentation + forr.toString() + "{";

        //print its children
     kern += Kernel(indentation + "\t", n3node.getNthChild(1));
     kern += "\n" + Kernel(indentation + "\t", n3node.getNthChild(0));

     kern += "\n" + indentation + "}";
```

```
        //kern += "\n"+indentation + "__syncthreads();";

    }
  else if (n3node != null && n3node.getObjectContent() instanceof
  StatementList) {

       StatementList stmts = (StatementList) n3node.getObjectContent();
       kern += UnpackCode(indentation, stmts);
   }

    return kern;
}
```

Μέθοδος *UnpackCode*:

```
  private String UnpackCode(String indentation, StatementList list) {
         String result = "";
         ArrayList arrlist = list.getList();

         for (int i = 0; i < arrlist.size(); i++) {
             Assign assign = (Assign) arrlist.get(i);
             result += "\n" + indentation +
             assign.toString().replaceAll("#", "");
         }

         return result;
  }
```

9 Βιβλιογραφία

[1]. R. Allen, K. Kennedy, *Optimizing Compilers for Modern Architectures, San Francisco:Morgan Kaufmann, 2001*

[2]. U. Banerjee, *Loop Parallelization: Loop Transformations for Restructuring Compilers, Massachusetts:Kluwer Academic, 2010*

[3]. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms (Second Edition), McGraw Hill, 2003*

[4]. V. Volkov, *Better Performance at Lower Occupancy, CU Berkeley, 22 September 2010.* *http://www.eecs.berkeley.edu/~volkov/volkov10-GTC.pdf*. (Ιανουάριος 2011)

[5]. V. Volkov, *Optimizing GPU Codes, CU Berkeley, 11 August 2009.* *http://www.eecs.berkeley.edu/~volkov/volkov09-optimizing.pdf*. (Ιανουάριος 2011)

[6]. V. Volkov, J. W. Demmel, *Benchmarking GPUs to tune Dense Linear Algebra, 2008 ACM/IEEE Conference on Supercomputing.* *http://mc.stanford.edu/cgi-bin/images/6/65/SC08_Volkov_GPU.pdf*. (Ιανουάριος 2011)

[7]. J. Kepler, *Automatic Code Generation using Dynamic Programming Techniques, U Linz, October 2007.* *http://www.bytelabs.org/pub/papers/hburg07.pdf*. (Απρίλιος 2010)

[8]. C. W. Fraser, D. R. Hanson, T. A. Proebsting, *Engineering a simple efficient Code Generator Generator, September 1992.* *http://drhanson.s3.amazonaws.com/storage/documents/iburg.pdf*. (Μάιος 2010)

[9]. M. Harris, Optimizing CUDA, S05: High Performance Computing with CUDA, 2007. *http://gpgpu.org/static/sc2007/SC07_CUDA_5_Optimization_Harris.pdf*. (Ιανουάριος 2011)

[10]. J. Owens, *Data-parallel Algorithms & Data Structures, S05: High Performance Computing with CUDA, 2007.* http://gpgpu.org/static/sc2007/SC07_CUDA_4_DataParallel_Owens.pdf. (Ιανουάριος 2011)

[11]. *JBurg: a Bottom-Up Rewrite Machine Generator for Java.* *http://jburg.sourceforge.net/* (Απρίλιος 2010)

[12]. M. Wolfe, *'GPU Architecture and Applications'. Compilers and More, 10 September 2008.* http://www.hpcwire.com/features/Compilers_and_More_GPU_Architecture_and_Applications.html. (Απρίλιος 2010)

[13]. M. Wolfe, *'Programming GPUs today'. Compilers and More, 9 October 2008*. http://www.hpcwire.com/features/Compilers_and_More_Programming_GPUs_Today.html. (Απρίλιος 2010)

[14]. M. Wolfe, *'Optimizing GPU kernels'. Compilers and More, 30 October 2008*. http://www.hpcwire.com/features/Compilers_and_More_Optimizing_GPU_Kernels.html. (Απρίλιος 2010)

[15]. M. Wolfe, '*A GPU and Accelerator Programming Model'. Compilers and More, 20 November 2008.* http://www.hpcwire.com/specialfeatures/sc08/features/Compilers_and_More_A_GPU_and_Accelerator_Programming_Model.html. (Απρίλιος 2010)

[16]. C. Vasseur, *Code Generator Generators-Generating the Instruction Selector, Technical Report,* June 2004. http://www.lrde.epita.fr/dload/20040602-Seminar/vasseur0604_code-gen-gen_report.pdf. (Μάιος 2010)

[17]. NVIDIA, *NVIDIA CUDA C Programming Guide*, version 3.2, 11 September 2010

[18]. NVIDIA, *NVIDIA CUDA C Programming Best Practises Guide*, version 2.3, July 2009

[19]. E.LaForest, *Survey of Loop Transformation Techniques, ECE 1754, 19 March 2010. http://www.eecg.toronto.edu/~laforest/SurveyLoopTransformations.pdf.* (Δεκέμβριος 2010)

[20]. V. Volkov, *Use registers and multiple outputs per thread on GPU, 30 June 2010. http://www.eecs.berkeley.edu/~volkov/volkov10-PMAA.pdf.* (Ιανουάριος 2010)

[21]. D. Walker, *Intermediate Representation, CS 320 U. Princeton, 3 April 2003. http://www.cs.princeton.edu/courses/archive/spr03/cs320/notes/IR-trans1.pdf.* (Μάιος 2010)

[22]. D. Walker, *Optimizing Compilers, CS 320 U. Princeton, 10 April 2003. http://www.cs.princeton.edu/courses/archive/spr03/cs320/notes/loops.pdf.* (Μάιος 2010)

[23]. D. Walker, *Optimizing Compilers, CS 320 U. Princeton, 15 April 2003. http://www.cs.princeton.edu/courses/archive/spr03/cs320/notes/loops2.pdf.* (Μάιος 2010)

[24]. M. Johnson, J. Zelenski, *Intermediate Representation, CS 143 Handout, 28 July 2008 http://dragonbook.stanford.edu/lecture-notes/Stanford-CS143/16-Intermediate-Rep.pdf.* (Μάιος 2010)

[25]. Γ. Ναλμπάντης, Δ. Κουφός, *Παραγοντοποίηση Cholesky αλληλουχίας πινάκων σε μονάδες επεξεργασίας γραφικών, Α.Π.Θ ΤΗΜΜΥ, Φεβρουάριος 2010.*

http://vivliothmmy.ee.auth.gr/713/1/Diplomati_Nalpmantis_Georgios_Koufos_Dimitrios.pdf. (Ιανουάριος 2011)

[26]. D.Göddeke, R.Strzodka, *ARCS 2008 GPGPU and CUDA Tutorials, TU Dresden, 25 February 2008.* *http://www.mathematik.tudortmund.de/~goeddeke/arcs2008/index.html* (Απρίλιος 2011)