ΠΟΛΥΤΕΧΝΕΙΟ ΚΡΗΤΗΣ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΝΙΚΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΕΡΓΑΣΤΗΡΙΟ ΜΙΚΡΟΕΠΕΞΕΡΓΑΣΤΩΝ ΚΑΙ ΥΛΙΚΟΥ

“Ανάπτυξη Παράλληλου Συστήματος Για Εξαγωγή Υπογράφων”

“Development of Highly Parallel System For Frequent Subgraph Mining”

Μαντάκος Γεώργιος

Επιτροπή:

Παπαευσταθίου Ιωάννης (Επιβλέπων)

Δόλλας Απόστολος

Γαροφαλάκης Μίνως

Περιεχόμενα

Περίληψη

Η εξόρυξη πληροφορίας από δεδομένα αποθηκευμένα σε μορφή γράφων (graph mining) βρίσκει πρακτική εφαρμογή σε πολλές περιοχές (molecular substructure discovery, web link analysis, fraud detection, social network analysis). Καθώς αυξάνεται ο όγκος των δεδομένων αυτών, ενώ συσσωρεύεται συνεχώς καινούργια πληροφορία, αυξάνεται μαζί του και η ανάγκη για αποδοτικότερη και γρηγορότερη εξόρυξη.

Το πρόβλημα που έχουν να λύσουν αυτοί οι αλγόριθμοι συνοψίζεται στην εξεύρεση όλων των υπογράφων που εμφανίζονται σε τουλάχιστον s γράφους σε ένα σύνολο γράφων (dataset), όπου το s καθορίζεται από το χρήστη. Ο έλεγχος ισομορφισμού των γράφων και ο τεράστιος χώρος αναζήτησης (search space) των υποφήφιων μοτίβων των γράφων (graph patterns) κάνουν τη διαδικασία χρονοβόρα ακόμη και για μικρά dataset. Για αυτό το λόγο έχουν γίνει αρκετές δουλειές πάνω σε παράλληλες υλοποιήσεις αυτών των αλγόριθμων σε διάφορες αρχιτεκτονικές.

Στην παρούσα εργασία γίνεται μια προσπάθεια παραλληλοποίησης σε GPU ενός από τους πιο αποδοτικούς αλγόριθμους στο πεδίο του frequent subgraph mining, του gSpan (Graph-Based Substructure Pattern Mining).

Κεφάλαιο 1: Εισαγωγή

1.1 Γενικά

Σύμφωνα με το wikipedia εξόρυξη δεδομένων (ή ανακάλυψη γνώσης από βάσεις δεδομένων) είναι η εξεύρεση μιας (ενδιαφέρουσας, αυτονόητης, μη προφανής και πιθανόν χρήσιμης) πληροφορίας ή προτύπων από μεγάλες βάσεις δεδομένων με χρήση αλγορίθμων ομαδοποίησης ή κατηγοριοποίησης και των αρχών της στατιστικής, της τεχνητής νοημοσύνης, της μηχανικής μάθησης και των συστημάτων βάσεων δεδομένων. Στόχος της εξόρυξης δεδομένων είναι η πληροφορία που θα εξαχθεί και τα πρότυπα που θα προκύψουν να έχουν δομή κατανοητή προς τον άνθρωπο έτσι ώστε να τον βοηθήσουν να πάρει τις κατάλληλες αποφάσεις.

Στη βιβλιογραφία, το data mining περιγράφεται με τον εξής ορισμό: «Η σύνθετη διαδικασία εξαγωγής συγκεκριμένης, προηγουμένως άγνωστης και δυνητικά ωφέλιμης, γνώσης από δεδομένα»[1]. Εναλλακτικά, συναντάται και ως «η επιστήμη της εξόρυξης χρήσιμης πληροφορίας από σύνολα ή βάσεις δεδομένων μεγάλου μεγέθους»[2]. Πιο απλά, το data mining μπορεί να οριστεί ως η διαδικασία που περιλαμβάνει την αναζήτηση, τη συλλογή, την επεξεργασία και την ανάλυση των δεδομένων. Είναι σημαντικό να ξεκαθαριστεί ότι ο ορισμός αυτός δεν είναι ευρέως αποδεκτός, ωστόσο περιγράφει εύστοχα την όλη διαδικασία.

Το graph mining είναι μια υποκατηγορία του data mining όπου τα δεδομένα προς εξόρυξη βρίσκονται αποθηκευμένα με τη μορφή γράφων. Γράφος είναι ένα σύνολο από κόμβους οι οποίοι μπορούν να συνδέονται μεταξύ τους, ακόμη και με τον εαυτό τους. Υπάρχουν διάφοροι τύποι γράφων όπως κατευθυνόμενοι, μη κατευθυνόμενοι, πεπερασμένοι, άπειροι, τυπικοί και πλήρεις. Αυτοί οι τύποι χρησιμοποιούνται σε εξειδικευμένες εφαρμογές όπου είναι απαραίτητες ειδικές σχέσεις και εξαρτήσεις μεταξύ των δεδομένων. Τις περισσότερες φορές όμως η αποθηκευμένη πληροφορία που έχει τη μορφή γράφου μπορεί να μοντελοποιηθεί με τη μορφή ενός μη κατευθυνόμενου γράφου.

Τεχνικές που έχουν αναπτυχθεί για graph mining είναι

- η εξόρυξη συχνών μοτίβων (frequent pattern mining)
- η ομαδοποίηση των δεδομένων (clustering) και
- η ταξινόμησή τους (classification) οι οποίες θα αναλυθούν παρακάτω.

Ο αλγόριθμος gspan με τον οποίο θα ασχοληθούμε ανήκει στην κατηγορία του frequent subgraph mining στην οποία θα δώσουμε και μεγαλύτερο βάρος.

1.1.1 Graph Frequent Pattern Mining

Όπως αναφέρθηκε παραπάνω οι γράφοι είναι μία ιδιαίτερα πολύπλοκη δομή δεδομένων. Η χρησιμοποίηση τους στο πρόβλημα του frequent pattern mining, αλλάζει λίγο τη διαδικασία μελέτης του ελάχιστου ορίου συχνότητας, που ονομάζεται support. Το πρόβλημα μπορεί να οριστεί με διάφορους τρόπους και ανάλογα με την εφαρμογή στην οποία απευθύνεται μπορούμε να διακρίνουμε δύο κατηγορίες. Στην πρώτη περίπτωση υπάρχει μία ομάδα από γράφους απ' όπου επιθυμείται η εξόρυξη όλων των patterns τα οποία ξεπερνούν το όριο της συχνότητας των αντίστοιχων γράφων [3,4,5]. Στη δεύτερη περίπτωση υπάρχει ένας μεγάλος γράφος και επιθυμείται η εξόρυξη εκείνων των pattern που εμφανίζονται ένα συγκεκριμένο αριθμό φορών (N) μέσα στον ίδιο γράφο [5,6,7]. Και στις δύο περιπτώσεις απαιτείται έλεγχος για τον ισομορφισμό, προκειμένου να μην εξεταστούν πολλαπλές φορές όμοια patterns. Ο συγκεκριμένος έλεγχος αποτελεί κρίσιμο σημείο του συνολικού προβλήματος αν επιτρέπονται οι επικαλύψεις μεταξύ όμοιων και διαφορετικών patterns μέσα στο σύνολο των εξεταζόμενων γράφων.

1.1.2 Clustering

Οι αλγόριθμοι που ανήκουν σε αυτή την κατηγορία, τόσο οι κλασικοί για graph clustering όσο και για clustering XML data, χρησιμοποιούνται σε πληθώρα εφαρμογών ανάμεσα στις οποίες βρίσκονται η κυκλοφοριακή συμφόρηση, η χωροθέτηση εγκαταστάσεων και η ολοκλήρωση XML δεδομένων[8]. Έχουμε δύο είδη αλγορίθμων σε αυτή την περιοχή οι οποίοι ομαδοποιούν τους κόμβους με διαφορετικά κριτήρια. Στη μία περίπτωση έχουμε ένα μεγάλο γράφο όπου η ομαδοποίηση των κόμβων του γίνεται με βάση την απόσταση ή την ομοιότητα των τιμών που έχουν οι ακμές, ενώ στην άλλη υπάρχει ένας μεγάλος αριθμός από γράφους και η ομαδοποίηση γίνεται με βάση τη συμπεριφορά της δομής τους. Η χρησιμοποίηση της ομοιότητας μεταξύ των δομών κάθε γράφου ως κριτήριο για την ομαδοποίηση όλων των γράφων αναδεικνύει από μόνη της την πολυπλοκότητα του συγκεκριμένου προβλήματος. Στην περίπτωση με τον ένα μεγάλο γράφο κάνουμε λόγο για Node clustering αλγόριθμους όπου το βάρος των ακμών συμβολίζει την αντίστοιχη απόσταση των κόμβων μέσα στο συγκεκριμένο γράφο. Έτσι με αυτό τον τρόπο, κατά την ομαδοποίηση, δημιουργούνται ομάδες κόμβων (clusters).

1.1.3 Graph Classification

Εδώ οι γράφοι χρησιμοποιούνται για να αναπαραστήσουν τις σχέσεις μεταξύ διάφορων οντοτήτων. Οι οντότητες αναπαριστάνονται από τους κόμβους ενώ οι σχέσεις μεταξύ τους από τις ακμές. Οι εφαρμογές οι οποίες χρησιμοποιούν αυτό το μοντέλο για να αναπαραστήσουν τα δεδομένα τους και κατάλληλους αλγόριθμους για την εξόρυξη πληροφορίας που εμπίπτουν στην κατηγορία του graph classification καλύπτουν ένα μεγάλο εύρος του επιστημονικού και βιομηχανικού τομέα. Για παράδειγμα στη φαρμακευτική και στο σχεδιασμό των φαρμάκων χρειάζεται να γνωρίζουμε τη σχέση μεταξύ της δραστηριότητας μιας χημικής ένωσης και τη δομή της ένωσης, η οποία αντιπροσωπεύεται από ένα γράφο. Ένα άλλο παράδειγμα που αφορά τα κοινωνικά δίκτυα είναι το εξής. Οι ακμές ενός γράφου μπορούν να σχετίζονται με το μέσο αριθμό κλήσεων ανά μήνα ανάμεσα σε δύο άτομα (κόμβους). Αυτό δίνει τη δυνατότητα σε ενδιαφερόμενους να αναζητούν ομάδες ανθρώπων που καλούν

ο ένας τον άλλο συχνά. Υπάρχουν δύο διαφορετικές προσεγγίσεις για το graph classification. Στην πρώτη ο γράφος περιλαμβάνει ένα σύνολο από κόμβους ο κάθε ένας από τους οποίους έχει τη δική του ετικέτα (label). Οι αλγόριθμοι αυτοί εξάγουν ένα μοντέλο από τους κόμβους με ετικέτες με βάση το οποίο θα ταξινομηθούν οι υπόλοιποι κόμβοι που δεν έχουν ετικέτα. Στη δεύτερη προσέγγιση οι αλγόριθμοι εργάζονται με ένα σύνολο από γράφους, κάποιοι από τους οποίους έχουν ετικέτα. Εξάγουν ένα μοντέλο από αυτούς με βάση το οποίο θα ταξινομηθούν οι υπόλοιποι γράφοι που δεν έχουν ετικέτα.

1.2 Επιστημονική Συνεισφορά

Το περιεχόμενο της παρούσας εργασίας εμπίπτει στην κατηγορία του Graph Frequent Pattern Mining καθώς ασχολείται με την παραλληλοποίηση του αλγόριθμου gspan, ενός από τους πιο αποδοτικούς και διαδεδομένους αλγόριθμους στην επιστημονική κοινότητα, για το πρόβλημα της εξόρυξης συχνών υπογράφων μέσα από ένα σύνολο πολλών γράφων ή ενός μεγάλου γράφου. Πιο συγκεκριμένα το pattern αναζήτησης είναι και αυτό ένας γράφος πράγμα που κάνει το συγκεκριμένο πρόβλημα γνωστό ως Frequent Subgraph Mining. Θα μιλήσουμε για αυτό στην επόμενη ενότητα.

Βασικός στόχος της εργασίας είναι η ανεύρεση και αξιοποίηση παραλληλοποιήσιμων κομματιών του αλγόριθμου με τον κατάλληλο σχεδιασμό τους για εκτέλεση σε GPU, επιταχύνοντας έτσι τον αλγόριθμο κατά ένα σημαντικό παράγοντα. Η αρχιτεκτονική σχεδιάστηκε και προσαρμόστηκε για μία nvidia GeForce GTX 780 κάρτα.

Κατά τη διάρκεια υλοποίησης της παρούσας εργασίας δεν υπήρχε αντίστοιχη αρχιτεκτονική για τον αλγόριθμο gspan σε GPU. Αναλύονται τα παραλληλοποιήσιμα κομμάτια και γίνεται σύγκριση μεταξύ της αρχικής και της παράλληλης αρχιτεκτονικής σε θέματα απόδοσης.

1.3 Δομή της εργασίας

- Κεφάλαιο 2

  Παρουσίαση και ανάλυση του θεωρητικού υπόβαθρου για την περιοχή του Graph Mining καθώς και παρουσίαση σχετικών εργασιών.

- Κεφάλαιο 3

  Περιγραφή του αλγόριθμου gspan καθώς και του τμήματος που σχεδιάστηκε για εκτέλεση από τη GPU.

- Κεφάλαιο 4

  Ανάλυση των αρχιτεκτονικών από τις οποίες περάσαμε και των βελτιστοποιήσεών τους μέχρι να καταλήξουμε στο τελικό σύστημα.

- Κεφάλαιο 5

  Παρουσίαση αποτελεσμάτων της υλοποίησής μας σε θέματα απόδοσης.

- Κεφάλαιο 6

  Παρουσίαση συμπερασμάτων στα οποία καταλήξαμε κατά την εκπόνηση της εργασίας

Κεφάλαιο 2: Σχετικές Εργασίες

2.1 Γενικά

Στο κεφάλαιο αυτό περιγράφεται όλο το θεωρητικό υπόβαθρο για την περιοχή του frequent subgraph mining προβλήματος. Επιπλέον, γίνεται μία αναφορά στους αλγορίθμους που έχουν αναπτυχθεί για το πρόβλημα αυτό καθώς επίσης και στις υπάρχουσες παράλληλες υλοποιήσεις και μελέτες που έχουν παρουσιασθεί μέχρι σήμερα.

2.2 Frequent Subgraph Mining

Το πρόβλημα στο frequent subgraph mining μπορεί να περιγραφεί ως η αναζήτηση όλων εκείνων των pattern (υπογράφων συγκεκριμένα) τα οποία εμφανίζονται πάνω από ένα προκαθορισμένο αριθμό φορών στο dataset. Ανάλογα με τη φύση του προβλήματος μπορεί να υπάρχουν πολλοί γράφοι και σκοπός να είναι η εξόρυξη όλων των υπογράφων που εμφανίζονται μέσα σε αυτούς τουλάχιστον m φορές, όπου m ορίζεται από το χρήστη. Εναλλακτικά μπορεί η είσοδος να αποτελείται από ένα τεράστιο γράφο και η εξόρυξη να γίνει αντίστοιχα σε αυτόν. Ο ελάχιστος αριθμός εμφανίσεων του pattern ονομάζεται support και θα μιλήσουμε γι' αυτό αναλυτικότερα κατά την ανάλυση του αλγορίθμου gspan στο κεφάλαιο 3. Οι αλγόριθμοι που ανήκουν σε αυτή την περιοχή μπορούν να κατηγοριοποιηθούν περαιτέρω, ανάλογα τον τρόπο αναπαράστασης των γράφων, την απαρίθμηση όλων των υπογράφων και το μέτρημα της συχνότητας εμφάνισής τους, στις εξής κατηγορίες

- graph theory based(apriori, pattern growth)
- inductive logic programming
- greedy search[11,12]

Θα μιλήσουμε συνοπτικά για αυτές παρακάτω.

2.2.1 Graph Theory Based Αλγόριθμοι

Οι αλγόριθμοι που ανήκουν στην κατηγορία graph theory based επεξεργάζονται ένα σύνολο γράφων με σκοπό την εξόρυξη συχνών υπογράφων με βάση το support threshold. Αυτό πρόκειται για μία παράμετρο που δίνεται από το χρήστη και αντιπροσωπεύει τον ελάχιστο αριθμό εμφανίσεων ενός υπογράφου για να είναι συχνός. Η διαδικασία της εξόρυξης σε αυτούς τους αλγόριθμους αποτελείται από δύο βήματα. Αφού υπολογιστούν οι υποψήφιοι συχνοί υπογράφοι ελέγχεται αν η συχνότητά τους είναι μεγαλύτερη από το support threshold. Τέτοιου είδους αλγόριθμοι ανάλογα με τον τρόπο που δημιουργούν τους πιθανούς συχνούς υπογράφους ανήκουν είτε στους apriori είτε στους pattern growth αλγόριθμους.

2.2.1.1 Apriori Αλγόριθμοι

Στους apriori η διαδικασία απαρίθμησης των υποψήφιων υπογράφων έχει bottom-up χαρακτήρα και ξεκινάει με υπογράφους που έχουν ένα μόνο κόμβο. Σε κάθε επανάληψη ο υποψήφιος k-υπογράφος ελέγχεται ως προς τη συχνότητά του και μόνο οι συχνοί υπογράφοι χρησιμοποιούνται για τη δημιουργία των (k+1)-υπογράφων. Για να δημιουργηθεί ένας υποψήφιος υπογράφος με μέγεθος k+1, δύο ή περισσότεροι συχνοί υπογράφοι μεγέθους k συγχωνεύονται. Η στρατηγική αναζήτησης που χρησιμοποιούν αυτοί οι αλγόριθμοι είναι η Breadth First Search. Τέτοιοι αλγόριθμοι είναι οι AGM[4], FSG[5], AcGM [13], FFSM[14], SPIN[15], και PATH[3]. Όλοι αυτοί όμως έχουν ένα βασικό μειονέκτημα. Η διαδικασία δημιουργίας ενός νέου υπογράφου συγχωνεύοντας δύο άλλους συχνούς υπογράφους είναι ακριβή διαδικασία.

2.2.1.2 Pattern Growth Αλγόριθμοι

Αντιθέτως οι αλγόριθμοι που ακολουθούν την pattern growth τακτική, κατά τη διαδικασία απαρίθμησης των υποψήφιων υπογράφων, προεκτείνουν έναν υπογράφο βάζοντάς του μία νέα ακμή σε κάθε δυνατή θέση και ελέγχουν αν ο νέος υπογράφος είναι “συχνός” πριν συνεχίσουν. Τέτοιοι αλγόριθμοι είναι ο MoFA[16], ο gSpan[9] (2002) και ο Gaston[17]. Με αυτή τη μέθοδο παρουσιάζεται το πρόβλημα της εξέτασης του ίδιου υπογράφου πολλές φορές, το οποίο λύνουν με διαφορετικό τρόπο ο καθένας από αυτούς τους αλγόριθμους.

2.2.2 Inductive Logic Programming Αλγόριθμοι

Οι προσεγγίσεις που εμπίπτουν στην κατηγορία αυτή χαρακτηρίζονται από τη χρήση του ILP να αναπαριστά τα δεδομένα των γράφων χρησιμοποιώντας horn clauses. Προέρχονται από τον τομέα εξόρυξης σχεσιακών δεδομένων, ο οποίος ουσιαστικά αφορά την εξόρυξη δεδομένων που είναι οργανωμένα σε πολλαπλούς αλληλοσυνδεόμενους πίνακες μιας σχεσιακής βάσης δεδομένων. Δεν υπάρχουν πολλοί αλγόριθμοι στον τομέα αυτό. Ο πιο γνωστός είναι ο WARMR ο οποίος στη συνέχεια εξελίχθηκε στον αλγόριθμο FARMER[19]. Ο αλγόριθμος αυτός περιλαμβάνει στοιχεία και από την κατηγορία ILP αλλά και από την apriori.

2.2.3 Greedy Search Αλγόριθμοι

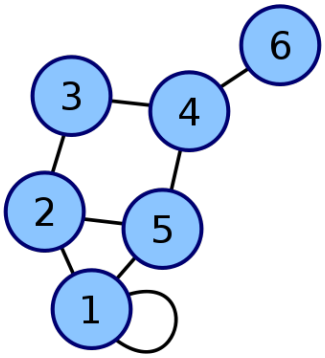
Το κύριο χαρακτηριστικό των αλγορίθμων που χρησιμοποιούν αυτή τη μέθοδο είναι ότι αποφεύγουν την υπολογιστική διαδικασία που απαιτείται για τον υπολογισμό όλων των συχνών υπογράφων και ακολουθούν μία άπλειστη προσέγγιση για να μειώσουν τον αριθμό των υπογράφων που εξετάζουν. Σε κάθε στάδιο λαμβάνεται μια απόφαση που φαίνεται να είναι η καλύτερη σε εκείνη τη χρονική στιγμή. Η απόφαση που θα ληφθεί σε ένα στάδιο δεν μπορεί να μεταβληθεί σε μεταγενέστερα στάδια οπότε κάθε απόφαση θα πρέπει να διασφαλίζεται ότι είναι η βέλτιστη δυνατή. Έτσι αποφεύγουν την πολυπλοκότητα του ισομορφισμού των γράφων, όμως χάνουν κάποιους συχνούς υπογράφους. Από αυτή την κατηγορία ξεχωρίζουν δύο αλγόριθμοι, ο SUBDUE[20] και ο GBI[12]. Και οι δύο αλγόριθμοι χρησιμοποιούν πολύ ισχυρά κριτήρια για την εξαγωγή των δομών του γράφου, όπως είναι η εντροπία της πληροφορίας και το gini-index[13].

2.3 Αναπαράσταση Γράφων

Οι γράφοι μπορούν να αναπαρασταθούν με δύο τρόπους, είτε με έναν πίνακα που ονομάζεται adjacency maxtrix, είτε με λίστες που ονομάζονται adjacency list.

2.3.1 Adjacency Matrix

Στην περίπτωση του adjacency matrix έχουμε ένα πίνακα n x n όπου n είναι ο αριθμός των κόμβων που έχει ο γράφος. Οι τιμές που μπορεί να έχει κάθε θέση του πίνακα είναι 0,1 και συμβολίζουν τη σύνδεση μεταξύ δύο κόμβων. Για τιμή 1 υπάρχει ακμή μεταξύ των δύο αντίστοιχων κόμβων.

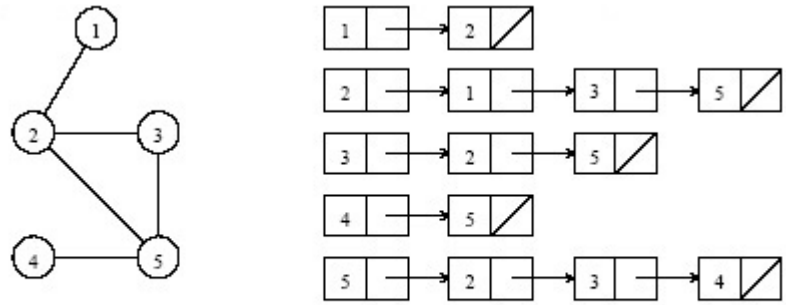


$$\begin{pmatrix} 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Εικόνα 2.3.1 Απεικόνιση ενός γράφου με adjacency matrix

2.3.2 Adjacency List

Στην άλλη μορφή αναπαράστασης (adjacency list) έχουμε μία λίστα για κάθε κόμβο η οποία κρατάει πληροφορία για τις συνδέσεις που έχει ο συγκεκριμένος κόμβος. Ουσιαστικά η λίστα αυτή περιέχει όλους τους κόμβους με τους οποίους συνδέεται ο κόμβος στον οποίο ανήκει η λίστα. Το μέγιστο μέγεθος της λίστας μπορεί να είναι όσο η τάξη του γράφου, δηλαδή όσο ο αριθμός των κόμβων που περιέχει ο γράφος.



Εικόνα 2.3.2 Απεικόνιση ενός γράφου με adjacency list

2.3.3 Απαρίθμηση Υπογράφων

Οι δύο τρόποι με τους οποίους μπορεί να πραγματοποιηθεί η απαρίθμηση των υπογράφων είναι οι εξής. Η μέθοδος της ένωσης (join operation) και η μέθοδος της επέκτασης(extension operation). Στην πρώτη περίπτωση ο κάθε υποψήφιος γράφος που προκύπτει από την ένωση μπορεί ο ίδιος να προκύπτει από πολλαπλές διαφορετικές ενώσεις. Στη δεύτερη περίπτωση ο αριθμός των κόμβων που μία νέα ακμή μπορεί να επισυνάψει περιορίζεται σε μεγάλο βαθμό.

2.3.4 Συχνότητα Εμφάνισης

Για τον υπολογισμό της συχνότητας εμφάνισης των υπογράφων χρησιμοποιούνται δύο δομές. Τα embedding lists και τα recomputed embeddings. Η πρώτη απαιτεί πολύ μεγαλύτερο αποθηκευτικό χώρο από τη δεύτερη καθώς απεικονίζει σε ποιο σημείο μέσα στη βάση δεδομένων, βρίσκεται κάθε υπογράφος και επιτρέπει τη γρήγορη αναζήτηση των υποψήφιων παιδιών του. Όμως απαιτείται η ολοκλήρωση της επεξεργασίας των παιδιών για να ελευθερωθεί ο πατέρας. Αντιθέτως στα recomputed embeddings διατηρείται ένα σύνολο από ενεργούς υπογράφους και για κάθε έναν επαναπροσδιορίζεται επαναληπτικά η συχνότητα εμφάνισής του[10].

2.4 Σχετικές Εργασίες

2.4.1 Γενικά

Το πρόβλημα της εξόρυξης συχνών υπογράφων αποτελεί από τη φύση του μία αρκετά ακριβή υπολογιστικά και χρονοβόρα διαδικασία. Η ανάγκη για αύξηση της απόδοσης και μείωση του χρόνου εξόρυξης οδήγησε σε προσπάθειες παραλληλοποίησης αυτών των αλγόριθμων. Η κατανεμημένη ή η παράλληλη αναζήτηση είναι δύο πιθανές κατευθύνσεις που μπορούν να ακολουθήσουν οι παράλληλες αυτές υλοποιήσεις ώστε να ξεπεράσουν εμπόδια στη χρήση μνήμης και στην απόδοση[22].

Στην περίπτωση της κατανεμημένης αναζήτησης σπάμε τα δεδομένα σε πολλά κομμάτια και κάθε ένα από αυτά επεξεργάζεται σε διαφορετικό μηχάνημα. Η μέθοδος αυτή απαιτεί τη χρήση κατανεμημένων μνημών και έχει καλή απόδοση, όμως περιορίζεται από το γεγονός ότι τα δεδομένα δεν είναι πάντα εύκολο να διασπαστούν με τέτοιο τρόπο, ιδιαίτερα σε γράφους χωρίς σταθερή δομή[23].

Εναλλακτικά μπορούμε να έχουμε μηχανήματα με shared memory. Οι υλοποιήσεις των αλγόριθμων σε αυτή τη περίπτωση χρησιμοποιούν μία global μνήμη η οποία είναι προσβάσιμη από όλους τους επεξεργαστές. Αυτό συμπεριλαμβάνει τους παράλληλους υπολογιστές με cache coherent μνήμες και τα μαζικά πολυνηματικά μηχανήματα (massively multi-thread machines MMT). Στην περίπτωση των παράλληλων υπολογιστών με cache coherency έχουμε γρηγορότερη πρόσβαση στα δεδομένα στη μνήμη σε σχέση με τις κατανεμημένες μνήμες όμως πληρώνουμε το κόστος για το cache coherency. Στα MMT έχουμε το latency της μνήμης το οποίο όμως “κρύβεται” από τα πολλαπλά αιτήματα προς τη μνήμη και την υποστήριξη μεγάλου αριθμού ταυτόχρονων νημάτων[23].

2.4.2 Παράλληλες Υλοποιήσεις

Παρακάτω θα δούμε εργασίες στις οποίες υλοποιήθηκαν αλγόριθμοι της περιοχής του graph mining σε διάφορες αρχιτεκτονικές και πλατφόρμες, με έμφαση στον παραλληλισμό, καθώς και τις επιδόσεις που πέτυχαν. Χρησιμοποιήθηκαν πολυπύρηνες πλατφόρμες, clusters και frameworks όπως το openMP[28], το MPI[32] και το MapReduce[27,31,35]. Τα πρώτα βήματα στον παραλληλισμό αλγορίθμων του frequent subgraph mining έγιναν πάνω σε clusters με πολυεπεξεργαστές.

2.4.2.1 Software Based

Μία από τις πρώτες παράλληλες υλοποιήσεις έγινε από τους Di Fatta και Berthold [24] σε ένα κατανεμημένο σύστημα με 32 επεξεργαστές. Το σύστημα αυτό πέτυχε speedup 15x σε σχέση με μία υλοποίηση ενός αλγόριθμου εξόρυξης υπογράφων (single threaded) υλοποιημένο πάλι από αυτούς.

Το 2005 οι Buehrer και Parthasarathy [25] υλοποίησαν έναν αλγόριθμο graph mining σε γλώσσα C υιοθετώντας αρκετές τεχνικές από τον αλγόριθμο gspan[9]. Αξιολόγησαν κάποια σημεία περαιτέρω κατάτμησης μέσα στον αλγόριθμο καθώς και ορισμένα μοντέλα αναμονής των εργασιών. Αποτέλεσμα ήταν ακόμη και η υλοποίηση με ένα thread να είναι ανταγωνιστική σε σχέση με τον αυθεντικό gspan. Στη συνέχεια χρησιμοποίησαν ένα σύστημα με 27 έως 32 επεξεργαστές και αντιστοίχησαν ένα thread σε κάθε επεξεργαστή πετυχαίνοντας ένα speedup 25x σε σχέση με τον αυθεντικό gspan(single threaded).

Την ίδια χρονιά οι Reinhardt and Karypis [28] (2005) παραλληλοποίησαν τον αλγόριθμο VSIGRAM[47] με χρήση του openMP σε μια πλατφόρμα με 32 επεξεργαστές. Σε σύγκριση με την πρώτη υλοποίηση του αλγορίθμου το 2004 σε μηχανήματα AMD Athlon MP 1800+ (1.53 Ghz) πήραν

speedup πάνω από 26x.

Το 2006 οι Meinl, Worlein, Fischer και Philippsen [26,27] παραλληλοποίησαν τους αλγόριθμους MoFa[16] και gSpan[9] σε ένα σύστημα SMP με 12 επεξεργαστές και χρήση κοινής μνήμης πετυχαίνοντας speedup 7x και 11x αντίχτοιχα.

Το 2009 παρουσιάστηκε μία καινούργια τεχνική από τους Ranu και Singh [29] για την εξόρυξη σημαντικών υπογράφων μέσα από πολύ μεγάλους γράφους. Η τεχνική τους εμφάνισε ως αποτέλεσμα γραμμικό speed-up σε σύγκριση με τις αυθεντικές υλοποιήσεις των αλγορίθμων gSpan[9] και FSG[5].

Ο αλγόριθμος SUBDUE[9], γνωστός για τη συμπίεση υπογράφων στο πρόβλημα του frequent subgraph mining, παραλληλοποιήθηκε από τους Ray και Holder[32] με χρήση του MPI framework. Η υλοποίηση αυτή ονομάστηκε SP-SUBDUE. Μετά από βελτιώσεις στην παράλληλη υλοποίησή τους, κατέληξαν σε μία δεύτερη έκδοση την οποία ονόμασαν SP-SUBDUE-2. Ο SP-SUBDUE-2 σε σύγκριση με την προηγούμενη έκδοση παρουσιάζει super-linear speedup, επειδή ο χρόνος εκτέλεσης του αλγορίθμου SUBDUE[20] δεν είναι γραμμικός σε σύγκριση με το μέγεθος του γράφου. Όσο κάθε core τρέχει τον SUBDUE για μικρούς γράφους, τόσο ο χρόνος εκτέλεσης του SP-SUBDUE σε σύγκριση με τον SUBDUE θα μειώνεται.

Όσο αφορά το Map Reduce framework οι Bin Wu και YunLong Bai [30] παρουσίασαν μία κατανεμημένη μέθοδο για το πρόβλημα του subgraph mining πετυχαίνοντας speedup 12x για 32 reducers. Επίσης οι Hill, Srichandan και Sunderraman[31] χρησιμοποιώντας το ίδιο framework για το ίδιο πρόβλημα πάνω σε βιολογικά δεδομένα πέτυχαν σχεδόν γραμμικό speedup.

Τέλος, το 2013 στο πανεπιστήμιο Purde της Indianapolis κατάφεραν χρησιμοποιώντας την πλατφόρμα του MapReduce, οι Bhuiyan και Al Hasan[33], να υλοποιήσουν έναν αλγόριθμο που ονόμασαν MIRAGE με σχεδόν γραμμικό speedup σε σύγκριση με την υλοποίηση που παρουσίασαν οι Hill, Srichandan και Sunderraman[31] το 2012.

2.4.2.2 Hardware Based

Όσο αφορά τις GPU οι πρώτες προσπάθειες παραλληλοποίησης τέτοιων αλγόριθμων έγιναν σε μία Nvidia 8800GTX από τους Harish και Narayanan[34]. Υλοποίησαν δύο αλγόριθμους σε C++, έναν για την BFS στρατηγική προσπέλασης πάνω στους γράφους και έναν για το πρόβλημα του Single Source Shortest Path(SSSP). Ο πρώτος αλγόριθμος, για μεγάλους γράφους με εκατομμύρια κόμβους και ακμές, παρουσίασε speedup 50x συγκρινόμενος με την υλοποίηση σε CPU. Ο δεύτερος αλγόριθμος είχε speedup 70x. Από τα αποτελέσματα αυτά αναδεικνύεται η καταλληλότητα των GPU για αυτά τα προβλήματα, λόγω του μικρού κόστους τους και της καλής απόδοσής τους. Παρόλα αυτά σε γράφους με χαμηλή τάξη (π.χ. 2-3), οι οποίοι χαρακτηρίζονται ως γραμμικοί, σε κάθε επανάληψη χρειάζεται να εξεταστεί κάθε κόμβος οπότε οι παράλληλοι αλγόριθμοι δεν έχουν κάποιο όφελος.

Οι Hong, Oguntebi, Olukotun [35] υλοποίησαν μία νέα μέθοδο breadth-first αναζήτησης (BFS) πάνω σε δεδομένα από γράφους, πετυχαίνοντας περίπου 2x speedup απέναντι σε μία διαδεδομένη μέθοδο αναζήτησης.

Οι Merrill, Garland και Grimshaw[36] παρουσίασαν μία καλά προσαρμοσμένη BFS προσπέλαση για γράφους με τη χρήση των GPUs πετυχαίνοντας speedup έως 2.5x. Χρησιμοποίησαν ένα υβριδικό σύστημα το οποίο για κάθε επίπεδο του BFS επιλέγει την κατάλληλη πλατφόρμα που θα τρέξει. Έτσι το σύστημα έχει καλή απόδοση και στους μικρούς και στους μεγάλους γράφους, ανεξάρτητα από την τάξη τους.

Το 2013 έχουμε την πρώτη υλοποίηση σε GPU με πραγματικά εντυπωσιακά αποτελέσματα[37]. Για

γράφους μέχρι 30 κόμβους η υλοποίηση αυτή πέτυχε μέχρι και 80x speedup.

Από τις πρώτες υλοποιήσεις αλγορίθμων για graph mining σε fpga ήταν η υλοποίηση του αλγόριθμου του Ullmann από τους Ichikawa, Saito, Udorn και Konishi[38]. Με την υλοποίησή τους αυτή πέτυχαν speedup 20x σε σύγκριση με την πιο διαδεδομένη εκτέλεση σε έναν υπολογιστή.

Οι Bondhugula, Devulapalli, Fernando, Wyckoff, Sadayappan[39] για πρώτη φορά υλοποίησαν το all-pairs shortest-paths πρόβλημα χρησιμοποιώντας την τεχνολογία των fpga. Η δουλειά τους έγινε σε μία πλατφόρμα Cray XD1 και έδωσε speedup 22x σε σχέση με έναν επεξεργαστή.

Το2007 οι Thomas, Luk, Stumpl[40] παρουσίασαν μια υλοποίηση του canonical labeling των κόμβων ενός γράφου σε FPGA με speedup 10x. Ενώ με τη μέθοδο canonical graph labeling πετύχαινε ένα speedup κατά ένα παράγοντα γύρω στο 100.

Οι Betkaoui, Thomas, Luk, Przulj[41] ανέπτυξαν ένα framework με το οποίο θα υλοποιούνται αλγόριθμοι για graph mining σε fpgas. Για να αναλύσουν αυτό το framework και να επιβεβαιώσουν τη λειτουργικότητά του, αναπτύχθηκε ένας αλγόριθμος graphlet counting. Η υλοποίηση αυτή είχε ένα speedup της τάξεως του 10x σε σχέση με ένα quad-core CPU.

Ο BFS αλγόριθμος για αναζήτηση ενός υπογράφου υλοποιήθηκε από τους Betkaoui, Wang, Thomas, Luk[42] σε μία πλατφόρμα πολλαπλών FPGA (Convey HC-1 server), με speedup 2x σε σχέση με έναν 32 core Xeon server.

Τέλος οι Kobori και Maruyama [43] υλοποίησαν έναν αλγόριθμο graph-cut segmentation σε FPGA με speedup 3x έως 5x συγκρινόμενος με άλλες λύσεις σε software και μία υλοποίηση σε GPU.

2.4.3 Μελέτες και Συγκρίσεις

Έχουν γίνει αρκετές εργασίες με σκοπό τη μελέτη και σύγκριση αλγορίθμων της περιοχής του subgraph mining. Κάθε μία από αυτές καταλήγει σε συμπεράσματα όπως σε ποιες περιπτώσεις ο κάθε αλγόριθμος είναι πιο αποδοτικός καθώς και συμπεράσματα σχετικά με τη μέθοδο που χρησιμοποιεί ο κάθε αλγόριθμος. Ακολουθούν μερικές από αυτές τις εργασίες.

Αντικείμενο της εργασίας [11] είναι η σύγκριση των αλγορίθμων SUBDUE[20] (Greedy search), AGM[4] (apriori based) και gSpan[9] (Pattern Growth based). Αναλύονται οι λειτουργίες του κάθε αλγόριθμου, ο τρόπος με τον οποίο αναπαριστούν την πληροφορία των γράφων, η στρατηγική αναζήτησης που χρησιμοποιούν, ο τρόπος με τον οποίο εκτιμούν τη συχνότητα των υπογράφων και πώς δέχονται τα δεδομένα εισόδου. Τα συμπεράσματα στα οποία καταλήγει είναι τα εξής.

Ο αλγόριθμος SUBDUE δεν απαριθμεί όλους τους συχνούς υπογράφους, σε αντίθεση με τους άλλους δύο αλγόριθμους, καθώς ακολουθεί την greedy τεχνική αναζήτησης. Επίσης είναι προτιμότερος όταν έχουμε datasets που αποτελούνται από ένα μόνο γράφο. Οι άλλοι δύο κερδίζουν όταν έχουμε μεγάλες βάσεις δεδομένων, στις οποίες παρατηρείται υψηλή συσχέτιση μεταξύ των δεδομένων. Ο gSpan[9] ξεπερνάει σε ταχύτητα εκτέλεσης τον AGM[4] κατά μία τάξη μεγέθους και είναι πιο αξιόπιστος για την εξόρυξη πιο μεγάλων γράφων με πιο μικρή συχνότητα εμφάνισης.

Οι αλγόριθμοι που συγκρίνονται στην εργασία [22] είναι οι MoFa[16], gSpan[9], FFSM[14] και Gaston[17]. Για να έχει πιο ακριβή αποτελέσματα χρησιμοποιήθηκαν οι ίδιες δομές δεδομένων κατά την υλοποίηση αυτών των αλγόριθμων. Τα συμπεράσματα είναι τα εξής.

Η χρήση των embedding lists, αντίθετα με τη γνώμη που επικρατεί, δεν έχει πολύ καλά αποτελέσματα όσο αφορά την αναζήτηση συχνών τμημάτων. Επίσης παρουσιάζει προβλήματα όταν η μνήμη δεν είναι επαρκής και όταν το throuput είναι μικρό. Ο αλγόριθμος gSpan[9] χρησιμοποιεί ένα σύστημα κωδικής αναπαράστασης των γράφων, πράγμα που τον καθιστά ανταγωνιστικό απέναντι στους Gaston[17] και FFSM[14]. Η τεχνική αυτή, για την ανίχνευση ισομορφικών γράφων, σε σχέση με άλλες τεχνικές για τον έλεγχο του ισομορφισμού των υπογράφων, είναι πολύ αποδοτική. Η μόνη περίπτωση στην οποία μπορεί να αποτύχει ο αλγόριθμος gSpan[9] είναι όταν εξετάσει πάρα πολύ μεγάλους γράφους. Παρόλο που ο αλγόριθμος MoFa[16], σε όλες τις δοκιμές, είναι πιο αργός από τους υπόλοιπους της ίδιας κατηγορίας χρησιμοποιείται σε μεγάλο βαθμό για την εξόρυξη γράφων σε μοριακές βάσεις και βιοχημικές ερωτήσεις(biochemical questions). Τέλος όλοι οι αλγόριθμοι έχουν γραμμική απόδοση σε σχέση με το μέγεθος της βάσης αν και έχουν διαφορετικούς δείκτες γραμμικής συσχέτισης.

Στην εργασία [44] γίνεται μία γενική ανάλυση όλων των αλγόριθμων του frequent subgraph mining με βάση κριτήρια όπως η στρατηγική αναζήτησης, ο τρόπος που δέχονται την είσοδο και η ολοκλήρωση της απαρίθμησης κάθε συχνού υπογράφου. Στη συνέχεια αναλύονται οι τρεις αλγόριθμοι που ολοκληρώνουν πλήρως τη διαδικασία της απαρίθμησης των συχνών υπογράφων, του FSG[5], του gSpan[9] και του Gaston[17]. Και οι τρεις δέχονται ως είσοδο ένα σύνολο από γράφους. Ο αλγόριθμος FSG[5] είναι από τους πιο γνωστούς αλγορίθμους που λειτουργεί χρησιμοποιώντας τη BFS στρατηγική αναζήτησης. Ο αλγόριθμος gSpan[9] είναι ο πρώτος αλγόριθμος που χρησιμοποίησε τη DFS στρατηγική αναζήτησης. Ενώ ο Gaston[17] είναι από τους πιο πρόσφατους αλγορίθμους και έχει την πιο γρήγορη εκτέλεση από όλους.

Ο gSpan[9] είναι αρκετά ανταγωνιστικός ως προς το χρόνο εκτέλεσης του Gaston[17], ο οποίος είναι πιο γρήγορος και από τους τρεις, και γρηγορότερος από τον FSG[5] που χρησιμοποιεί την BFS στρατηγική αναζήτησης.

2.4.4 Συμπεράσματα

- Η διαδικασία απαρίθμησης των συχνών υπογράφων δεν ολοκληρώνεται από όλους τους αλγόριθμους.
- Οι ταχύτεροι αλγόριθμοι που ολοκληρώνουν τη διαδικασία απαρίθμησης των συχνών υπογράφων είναι ο Gaston[17] και ο gSpan[9], με το Gaston να υπερτερεί.
- Ο gSpan χρησιμοποιεί λιγότερη μνήμη από το Gaston λόγω του Canonical Labeling συστήματος. Η χρήση των embedding lists από το Gaston αυξάνει την απόδοσή του αλλά και την κατανάλωση μνήμης.

Κεφάλαιο 3: Ανάλυση Αλγόριθμου

3.1 Γενικά

Στο κεφάλαιο αυτό θα ασχοληθούμε αναλυτικότερα με τον αλγόριθμο gSpan. Εν συντομία ο αλγόριθμος αυτός είναι ο πρώτος από την κατηγορία pattern growth που χρησιμοποιεί τη depth first search (DFS) στρατηγική αναζήτησης.

Κατά την προέκταση των γράφων χρησιμοποιούνται τρία κριτήρια (backward, forward pure, forward rmpath) τα οποία θα αναλύσουμε παρακάτω.

Όλες οι προεκτάσεις γίνονται πάνω στο πιο δεξιό κομμάτι του γράφου (rightmost path).

Για κάθε πιθανή προέκταση που εξετάζεται ο αλγόριθμος μετονομάζει τους κόμβους και τις ακμές που τους συνδέουν και δημιουργεί ένα κωδικό για το συγκεκριμένο γράφο (DFS CODE).

Στη συνέχεια ελέγχεται αν αυτός ο γράφος είναι ελάχιστος λεξικογραφικά. Αν είναι τότε έχει περάσει τον έλεγχο ισομορφισμού και συνεχίζεται η προέκτασή του, διαφορετικά δεν προεκτείνεται και συνεχίζει ο αλγόριθμος με την επόμενη πιθανή προέκταση.

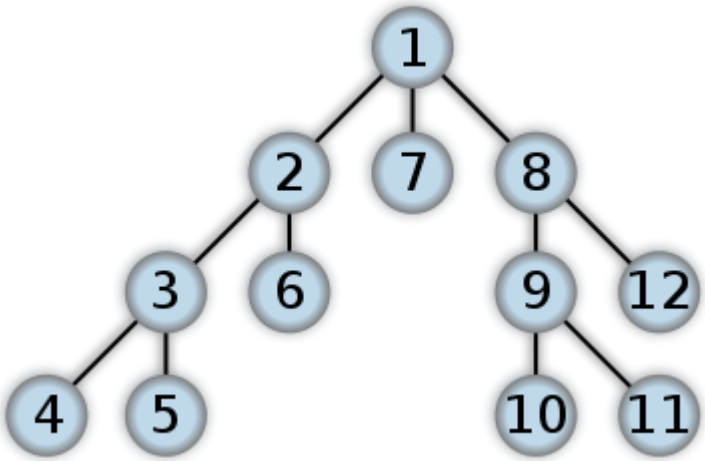
Η δομή του κεφαλαίου έχει ως εξής.

- Περιγραφή απαραίτητων ορισμών για την κατανόηση του αλγόριθμου.
- Ανάλυση του αλγόριθμου βήμα βήμα.
- Ανάλυση απόδοσης του αλγόριθμου.
- Τμήματα αλγορίθμου για υλοποίηση στη GPU.

3.2 Ορισμοί

3.2.1 DFS στρατηγική αναζήτησης

Όπως αναφέρθηκε παραπάνω ο αλγόριθμος gSpan[9] είναι ο πρώτος που χρησιμοποίησε τη μέθοδο depth first search (DFS). Η μέθοδος αυτή εξετάζει ένα μονοπάτι σε βάθος και στην συνέχεια επιστρέφει και εξετάζει το επόμενο μονοπάτι από την αμέσως προηγούμενη διακλάδωση. Η στρατηγική αναζήτησης σε ένα πρόβλημα με γράφους αποτελεί βασικό κριτήριο, που καθορίζει τη φύση του αλγόριθμου και τον διαχωρίζει ή τον κατατάσσει στην ίδια οικογένεια με άλλους αλγόριθμους. Η τεχνική που θα χρησιμοποιηθεί για τη διάσχιση ενός γράφου παίζει καθοριστικό ρόλο στη λειτουργικότητα και το συνολικό χρόνο που χρειάζεται ο αλγόριθμος για να ολοκληρώσει την εξέταση όλων των υπογράφων.



Εικόνα 3.2.1 Διάσχιση δέντρου με την DFS στρατηγική

3.2.2 Ισομορφισμός Υπογράφων

Δύο γράφοι G1(V1,L1,E1) και G2(V2,L2,E2) είναι ισομορφικοί αν υπάρχει ένα προς ένα αντιστοιχία από τους κόμβους του ενός γράφου (V1) στους κόμβους του άλλου (V2) η οποία διατηρεί τα βάρη των κόμβων, των ακμών και όλες τις συνδέσεις με τους γειτονικούς κόμβους.

$f : V1 \rightarrow V2$, τέτοια ώστε : $(v1, v2) \in E1$ αν και μόνο αν $(f(v1), f(v2)) \in E2$

Έτσι αν δύο γράφοι είναι ισομορφικοί τότε ο ένας μπορεί να θεωρηθεί ως υποσύνολο του άλλου.

Ένας γράφος GS(VS,LS,ES) είναι υπογράφος ενός γράφου G(V,L,E) αν και μόνο αν $VS \subseteq V$ και $ES \subseteq E$.

3.2.3 Rightmost Path

Κάτι που διαφοροποιεί τον αλγόριθμο gSpan από τους υπόλοιπους στην περιοχή του subgraph mining είναι η προέκταση των γράφων μόνο από το rightmost path. Το μονοπάτι αυτό (rightmost path) ξαναδημιουργείται κάθε φορά που εισάγεται μία νέα ακμή στο γράφο. Η διαδικασία δημιουργίας του έχει ως εξής. Ξεκινάει από την πιο πρόσφατη ακμή που μπήκε στο γράφο. Στη συνέχεια επιλέγει την πιο πρόσφατη ακμή που ενώνει τον κόμβο εκκίνησης της επιλεγμένης ακμής με κάποιον άλλο κόμβο. Η διαδικασία ολοκληρώνεται όταν εισαχθεί στο μονοπάτι μία ακμή που έχει ως τελευταίο κόμβο τη ρίζα του γράφου.

3.2.4 Support Threshold

Πρόκειται για μία παράμετρο με βάση την οποία ο αλγόριθμος δημιουργεί ένα κάτω όριο, το οποίο χρησιμοποιείται για να αγνοήσει εκείνους τους υπογράφους που έχουν συχνότητα εμφάνισης χαμηλότερη από αυτό. Ουσιαστικά είναι ένα ποσοστό που αντιπροσωπεύει την ποσοστιαία συχνότητα εμφάνισης ενός υπογράφου μέσα στους γράφους που δόθηκαν στο dataset εισόδου. Συγκεκριμένα ο αλγόριθμος gSpan δέχεται ως είσοδο έναν ακέραιο σ και τους γράφους που βρίσκονται στο dataset. Απαριθμεί το πλήθος των γράφων (G) και υπολογίζει το ποσοστό με βάση το οποίο ο αλγόριθμος gSpan κόβει τους υπογράφους που δεν είναι συχνοί. Το support threshold λοιπόν προκύπτει ως το πηλίκο:

$$\text{support threshold} = \frac{\sigma}{G} * 100 \quad \%$$

3.2.5 Λεξικογραφικό κριτήριο

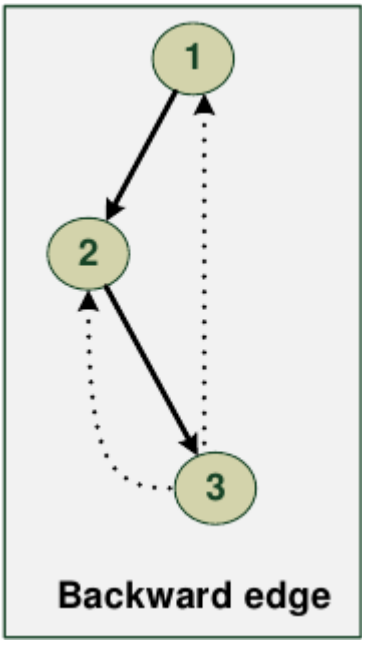
Χρησιμοποιείται από τον αλγόριθμο gSpan για να δώσει προτεραιότητα σε κάποιες από τις υποψήφιες ακμές που προεκτείνουν ένα γράφο. Έτσι ο αλγόριθμος διαχωρίζει πολύ εύκολα ποια μονοπάτια έχουν ήδη εξεταστεί με αποτέλεσμα να μειώνεται σημαντικά ο χώρος αναζήτησης (search space) των εξεταζόμενων μονοπατιών μέσα στους γράφους. Η προτεραιότητα έχει σχέση με το βάρος των κόμβων και των ακμών που τους συνδέουν. Η μικρότερη «λεξικογραφικά» ακμή, είναι εκείνη που έχει το μικρότερο βάρος κόμβου εκκίνησης, το μικρότερο βάρος ακμής και το μικρότερο βάρος κόμβου προορισμού, με αυτή τη σειρά. Το λεξικογραφικό κριτήριο χρησιμοποιείται σε τρία σημεία του αλγορίθμου. Στην αρχή, όταν ψάχνει τις μικρότερες ακμές που θα αποτελέσουν τη ρίζα κάθε εξεταζόμενου γράφου. Στον έλεγχο ισομορφισμού του γράφου. Και τέλος, όταν εξετάζει τις υποψήφιες προεκτάσεις του γράφου και υπάρχουν πολλές προεκτάσεις από τον ίδιο κόμβο.

3.2.6 Κριτήρια προέκτασης γράφων

Όπως αναφέρθηκε παραπάνω τα κριτήρια αυτά είναι τρία. Το καθένα από αυτά αναζητά ακμές που βρίσκονται στο rightmost path και έχουν συγκεκριμένα χαρακτηριστικά.

3.2.6.1 Backward

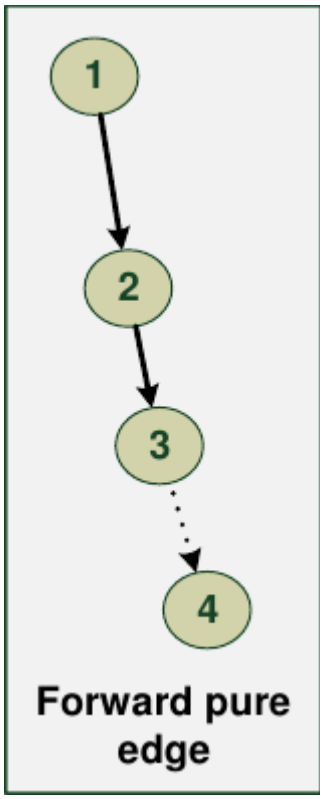
Μία ακμή για να χαρακτηριστεί ως backward πρέπει να ξεκινάει απαραίτητα από τον τελευταίο κόμβο του rightmost path καθώς και να καταλήγει σε κόμβο που βρίσκεται επίσης μέσα σε αυτό. Κατά τη διαδικασία εύρεσης αυτών των ακμών επιστρέφονται όλες οι ακμές που ικανοποιούν την παραπάνω συνθήκη και δίνεται προτεραιότητα σε αυτήν που πηγαίνει προς τον πιο παλιό κόμβο. Παλιός κόμβος θεωρείται αυτός που έχει μικρότερο id.



Εικόνα 3.2.6.1 Πιθανές προεκτάσεις τύπου backward

3.2.6.2 Forward pure

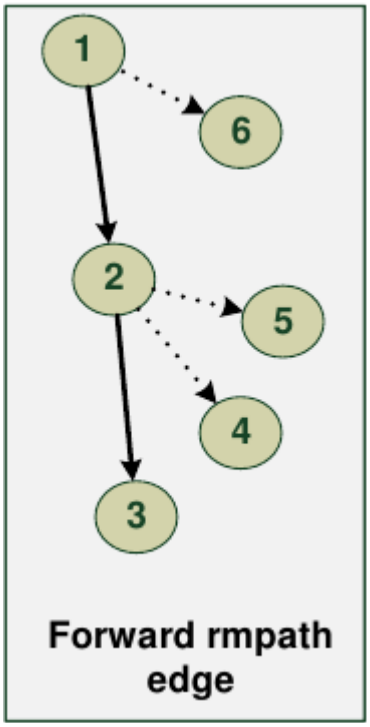
forward pure χαρακτηρίζεται μια ακμή όταν ξεκινάει από τον τελευταίο κόμβο του rightmost path και ο κόμβος προορισμού δεν βρίσκεται μέσα σε αυτό. Κατά τη διαδικασία εύρεσης αυτών των ακμών, επιστρέφονται όλες οι ακμές που ικανοποιούν την παραπάνω συνθήκη και δίνεται προτεραιότητα σε αυτήν που είναι μικρότερη λεξικογραφικά.



Εικόνα 3.2.6.2 Πιθανές προεκτάσεις τύπου forward pure

3.2.6.3 Forward rmpath

Ακμές με αυτό το χαρακτηρισμό ξεκινούν από οποιονδήποτε κόμβο του rightmost path εκτός του τελευταίου και καταλήγουν σε κόμβο που δεν βρίσκεται μέσα σε αυτό. Κατά τη διαδικασία εύρεσης αυτών των ακμών, επιστρέφονται όλες οι ακμές που ικανοποιούν την παραπάνω συνθήκη και δίνεται προτεραιότητα σε αυτήν που ξεκινάει από τον πιο πρόσφατο κόμβο μέσα στο rightmost path. Όσο πιο μεγάλο id έχει ένας κόμβος τόσο πιο πρόσφατος θεωρείται.



Εικόνα 3.2.6.3 Πιθανές προεκτάσεις τύπου forward rmpath

3.3 Ανάλυση Αλγόριθμου Gspan

Ο αλγόριθμος gSpan είναι ένας από τους πιο διαδεδομένους στο πρόβλημα της εξόρυξης συχνών υπογράφων. Είναι ανταγωνιστικός, από άποψη χρόνου εκτέλεσης, με τον Gaston ο οποίος είναι ο πιο γρήγορος και χρησιμοποιεί λιγότερη μνήμη από αυτόν.

Παρακάτω βλέπουμε τα βήματα εκτέλεσης του αλγόριθμου gSpan

Αλγόριθμος gSpan

Είσοδος : όριο support σ, γράφοι εισόδου G

Βήματα Αλγόριθμου:

Έξοδος : απαριθμημένοι συχνοί υπογράφοι

- Διαβάζει όλους τους γράφους.
- Μετονομάζει τους κόμβους και τις ακμές κάθε γράφου.
- Βρίσκει κάθε μονή ακμή από τους αρχικούς γράφους.
- Τις ταξινομεί, σύμφωνα με τα λεξικογραφικά κριτήρια.
- Εξετάζει όλες τις πιθανές επεκτάσεις, καλώντας επαναληπτικά την Subgraph_Mining Διαδικασία.

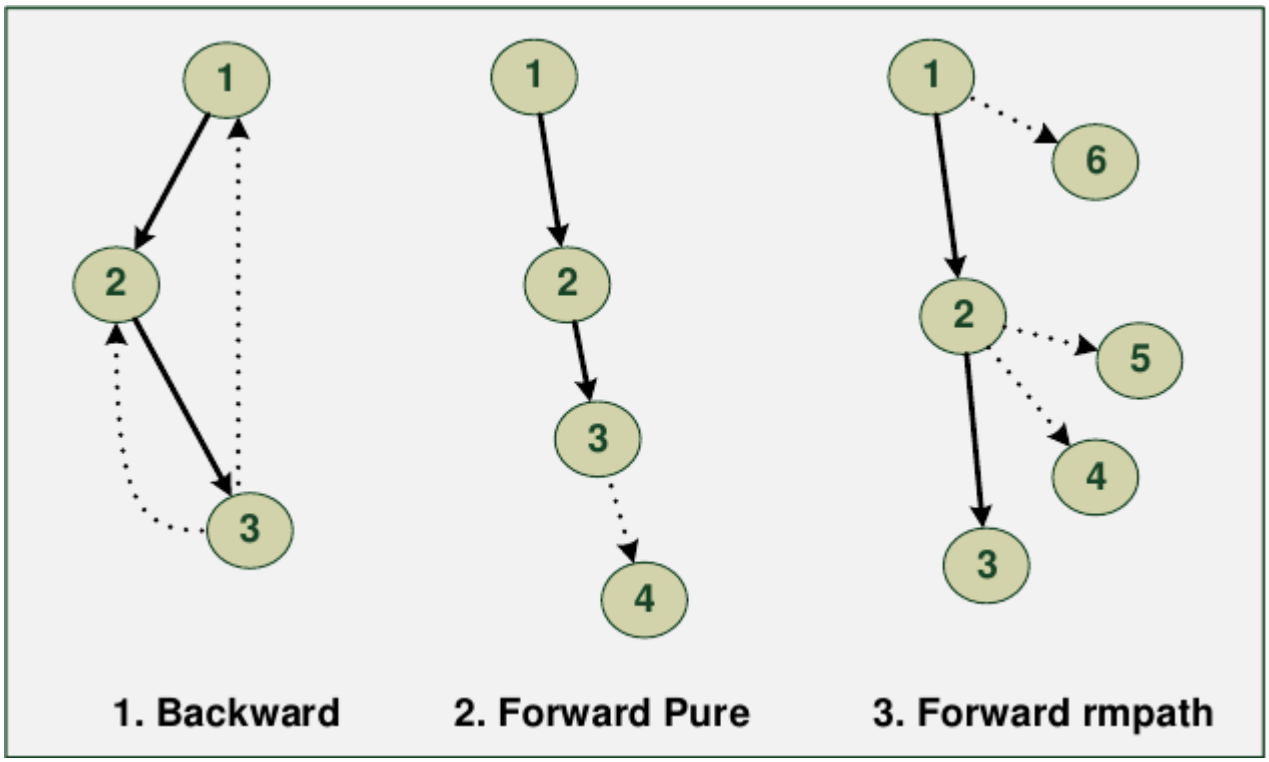
Subgraph_Mining Διαδικασία:

- Ελέγχει το support όριο του υπογράφου.
- Εκτελεί το test ισομορφισμού του υπογράφου.
- Εκτυπώνει τον συχνό υπογράφο.
- Εξετάζει κάθε πιθανή επέκταση.
- Για κάθε νέα επέκταση, καλεί αναδρομικά την Subgraph_Mining Διαδικασία.

Για να ξεκινήσει ο αλγόριθμος χρειάζεται δύο εισόδους, έναν ακέραιο αριθμό σ, που αποτελεί την ελάχιστη συχνότητα εμφάνισης κάθε υπογράφου, και τους γράφους του dataset μέσα από τους οποίους ο αλγόριθμος θα αναζητήσει κάθε μικρότερο συχνό υπογράφο. Ο αλγόριθμος αρχικά διαβάζει το dataset και κρατάει πληροφορία για τα βάρη των κόμβων και των ακμών κάθε γράφου. Η πληροφορία αυτή αποθηκεύεται σε δομές δεδομένων τύπου λίστας. Για κάθε κόμβο υπάρχει μία λίστα που περιέχει τις ακμές που ξεκινούν από αυτόν. Κάθε ακμή έχει δύο id. Το TRANS-id, το οποίο προσδιορίζει σε ποιο γράφο ανήκει και το edge-id που αντιπροσωπεύει τη σειρά της ακμής μέσα στο γράφο.

Στο επόμενο βήμα ο αλγόριθμος αναζητεί όλες τις μονές ακμές και τις ταξινομεί με βάση το λεξικογραφικό κριτήριο. Στη συνέχεια τις προεκτείνει μία μία, ξεκινώντας από τις προεκτάσεις της μικρότερης «λεξικογραφικά» ακμής. Η διαδικασία αυτή όπως είδαμε παραπάνω ονομάζεται Subgraph_Mining. Αποτελεί μία αναδρομική διαδικασία που εξετάζει σε βάθος κάθε μονοπάτι που ξεκινάει από μία μονή ακμή ή όπως την αναφέρει ο αλγόριθμος root edge, Καλείται επαναληπτικά για κάθε μονή ακμή που υπάρχει μέσα στους γράφους του dataset.

Η διαδικασία Subgraph_Mining αρχικά “κόβει” όσους υπογράφους δεν είναι συχνοί ή έχουν ήδη εξεταστεί, δηλαδή δεν πληρούν το κριτήριο της συχνότητας και του ισομορφισμού, μειώνοντας έτσι το χώρο αναζήτησης (search space). Αν κάποιο από αυτά τα δύο κριτήρια δεν πληρούνται τότε ο συγκεκριμένος υπογράφος δεν επεκτείνεται άλλο και ο αλγόριθμος συνεχίζει με τον επόμενο. Σημαντικό ρόλο από άποψη απόδοσης έχει η σειρά με την οποία ελέγχονται αυτά τα δύο κριτήρια καθώς το κριτήριο του ισομορφισμού είναι πολύ πιο απαιτητικό υπολογιστικά. Αυτός είναι και ο λόγος που πρώτα ελέγχεται ο υπογράφος ως προς τη συχνότητα. Στο κριτήριο της συχνότητας εξετάζεται αν ο συγκεκριμένος υπογράφος που έχει προεκταθεί βρίσκεται σε τουλάχιστον σ από τους γράφους εισόδου. Αν πληρεί αυτή την προϋπόθεση τότε εξετάζεται το κριτήριο του ισομορφισμού όπου ελέγχεται αν ο συγκεκριμένος υπογράφος μπορεί να προκύψει από επέκταση άλλου υπογράφου. Κατά τον έλεγχο αυτό δημιουργείται ένας νέος γράφος με τις ελάχιστες λεξικογραφικά συνδέσεις, τηρώντας το λεξικογραφικό κριτήριο καθώς και την προτεραιότητα για την εξέταση των πιθανών προεκτάσεων. Δηλαδή πρώτα εξετάζεται η ύπαρξη ακμών backward, με προτεραιότητα σε εκείνη που πηγαίνει προς τον πιο παλιό κόμβο. Μετά εξετάζεται η ύπαρξη ακμών forward_pure και τέλος forward_rmpath με προτεραιότητα σε αυτή που ξεκινάει από τον πιο πρόσφατο κόμβο. Σε περίπτωση που ο νέος γράφος που δημιουργήθηκε είναι ίδιος με αυτόν που εξετάζεται τότε ο εξεταζόμενος γράφος είναι λεξικογραφικά ελάχιστος. Αν δεν ισχύει αυτό τότε σταματάει η διαδικασία προέκτασης του συγκεκριμένου γράφου.



Εικόνα 3.3.1 Τύποι Προεκτάσεων

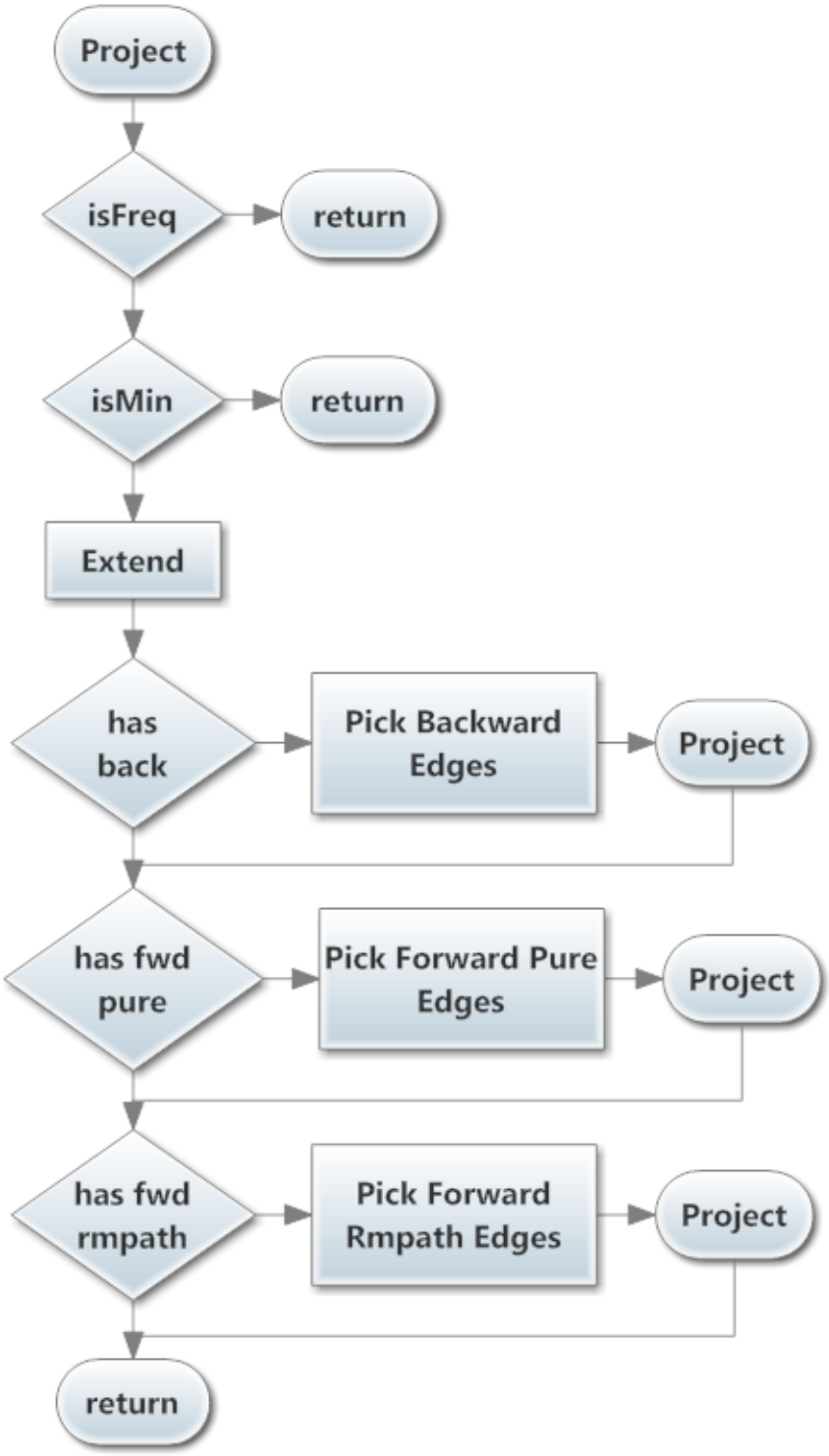
3.4 Ανάλυση αλγόριθμου για παράλληλη υλοποίηση

Ο αλγόριθμος gspan όπως θα δούμε παρακάτω, παρά την DFS στρατηγική που ακολουθεί, έχει κομμάτια που μπορούν να παραλληλοποιηθούν. Υπάρχουν πολλά που είναι κρίσιμα κατά την εκτέλεση του αλγόριθμου με τα βασικότερα να είναι είναι η διαδικασία επέκτασης του κυρίως υπογράφου και ο έλεγχος ισομορφισμού του. Αυτά είναι τα “ακριβότερα” υπολογιστικά, γι' αυτό και έχει επικεντρωθεί εκεί το ενδιαφέρον μας.

Κατά τη διαδικασία επέκτασης ενός υπογράφου εξετάζονται όλες οι εμφανίσεις του μέσα στους γράφους του αρχικού dataset εισόδου. Στη συνέχεια αναζητούνται και απαριθμούνται όλες οι πιθανές επεκτάσεις τους. Όσες δεν ξεπερνούν το όριο της συχνότητας που έχει οριστεί διαγράφονται. Στο επόμενο βήμα πραγματοποιείται ο έλεγχος ισομορφισμού με χρήση του λεξικογραφικού κριτηρίου. Αν ο εξεταζόμενος υπογράφος είναι λεξικογραφικά ελάχιστος τότε συνεχίζεται η επέκτασή του, σε διαφορετική περίπτωση σημαίνει ότι υπάρχει όμοιος υπογράφος οπότε δεν υπάρχει λόγος να συνεχιστεί η επέκτασή του.

Ακολουθεί το διάγραμμα ροής της διαδικασίας επέκτασης και στη συνέχεια μία ανάλυση των επιμέρους διαδικασιών.

Project Procedure



Εικόνα 3.4 Διάγραμμα διαδικασίας Project

3.4.1 Support

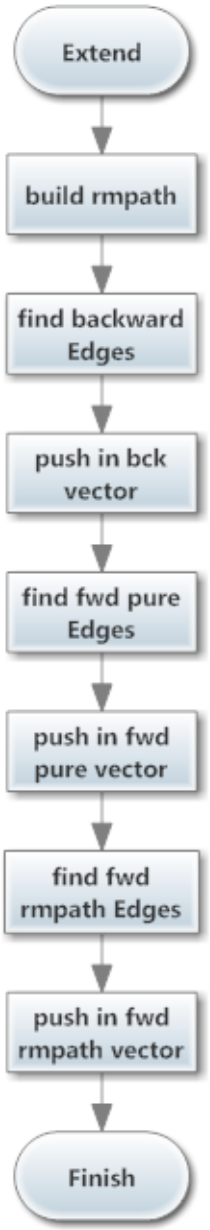
Συχνότητα ενός υπογράφου είναι ο αριθμός των υποστάσεων του συγκεκριμένου υπογράφου στο σύνολο των γράφων εισόδου. Μπορεί να υπάρχει μία υπόσταση ενός υπογράφου σε ένα γράφο εισόδου καθώς και πολλαπλές υποστάσεις μέσα στον ίδιο γράφο. Στην περίπτωση που υπάρχουν πολλαπλές υποστάσεις του ίδιου υπογράφου μέσα στον ίδιο γράφο εισόδου, τότε μετρούνται ως μία για τον έλεγχο της συχνότητας. Όταν λοιπόν το σύνολο των γράφων εισόδου, στους οποίους υπάρχει τουλάχιστον μία φορά ο εξεταζόμενος υπογράφος, είναι μεγαλύτερος ή ίσος με το όριο της συχνότητας που έχει δοθεί τότε ο συγκεκριμένος υπογράφος χαρακτηρίζεται συχνός. Σε αντίθετη περίπτωση ο υπογράφος δεν είναι συχνός και σταματάει η διαδικασία επέκτασης.

3.4.2 Έλεγχος Ισομορφισμού

Ο έλεγχος ισομορφισμού γίνεται με χρήση του λεξικογραφικού κριτηρίου. Αν ο εξεταζόμενος γράφος δεν είναι ελάχιστος λεξικογραφικά τότε σταματάει η διαδικασία επέκτασής του. Για την πραγματοποίηση αυτού του έλεγχου δημιουργείται από την αρχή ένας καινούργιος γράφος με δεδομένες τις ακμές, τους κόμβους και τα βάρη τους. Κατά τη δημιουργία του ελάχιστου γράφου χρησιμοποιούνται τα εξής κριτήρια. Αρχικά εντοπίζεται η μικρότερη ακμή ως προς τα βάρη με βάση το λεξικογραφικό κριτήριο. Στη συνέχεια αναζητούνται όλες οι επεκτάσεις και επιλέγονται με σειρά προτεραιότητας πρώτα η backward επέκταση, μετά η forward_pure και τέλος η forward_rmpath. Στην περίπτωση που υπάρχουν πολλές backward τότε επιλέγεται πρώτη αυτή που ο κόμβος προορισμού της έχει το μικρότερο id, δηλαδή είναι ο παλαιότερος μέσα στο rmpath. Αντίστοιχα αν υπάρχουν πολλές forward_pure τότε επιλέγεται πρώτα αυτή που είναι μικρότερη λεξικογραφικά. Τέλος αν υπάρχουν πολλές forward_rmpath τότε επιλέγεται πρώτα αυτή που έχει κόμβο εκκίνησης με το μεγαλύτερο id, δηλαδή τον πιο πρόσφατο μέσα στο rmpath.

3.4.3 Διαδικασία Επέκτασης Υπογράφων

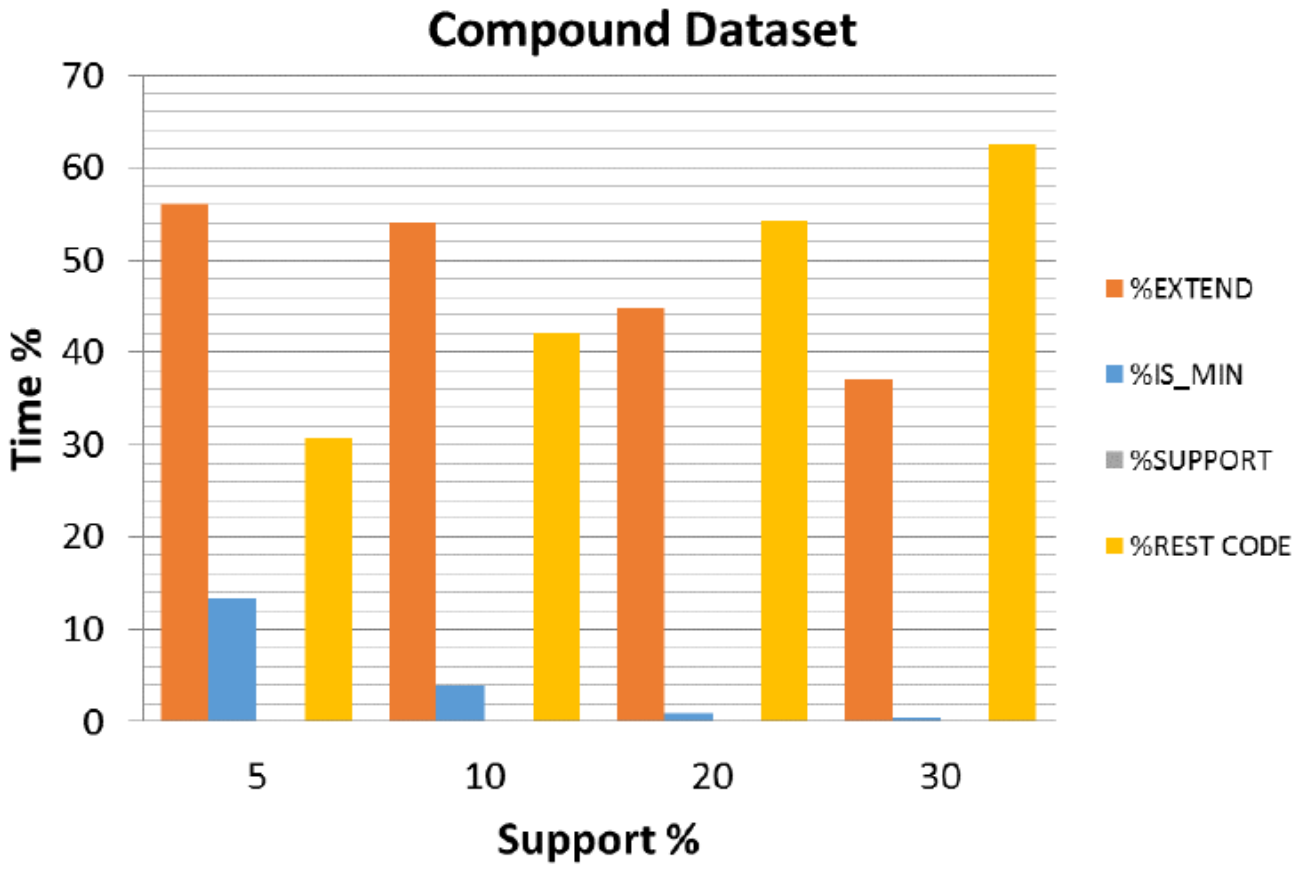
Το πρώτο βήμα στη διαδικασία επέκτασης των υπογράφων είναι η δημιουργία του rightmost path του εξεταζόμενου υπογράφου. Στη συνέχεια αναζητούνται όλες οι πιθανές επεκτάσεις backward, forward_pure και forward_rmpath. Κάθε νέα ακμή που προκύπτει ανάλογα με το είδος της τοποθετείται σε ένα vector. Στο τέλος επιστρέφει κάθε ακμή από κάθε είδος. Μετά την ολοκλήρωση της διαδικασίας αναζήτησης των επεκτάσεων το κομμάτι του project είναι υπεύθυνο να επιλέξει μία μία όλες τις ακμές, να τις μετονομάσει και να τις προεκτείνει περεταίρω.



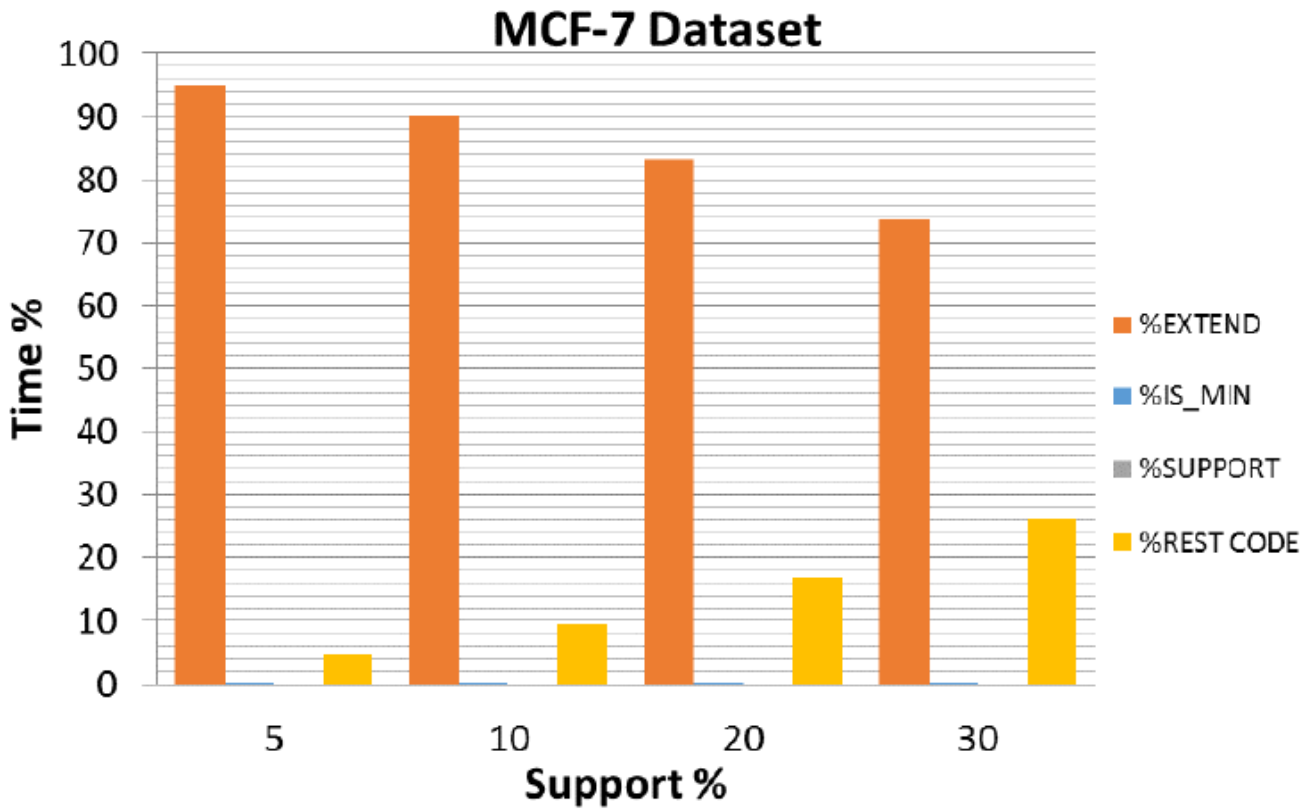
Εικόνα 3.4.3 Διαδικασία επέκτασης υπογράφων

3.4.4 Profiling

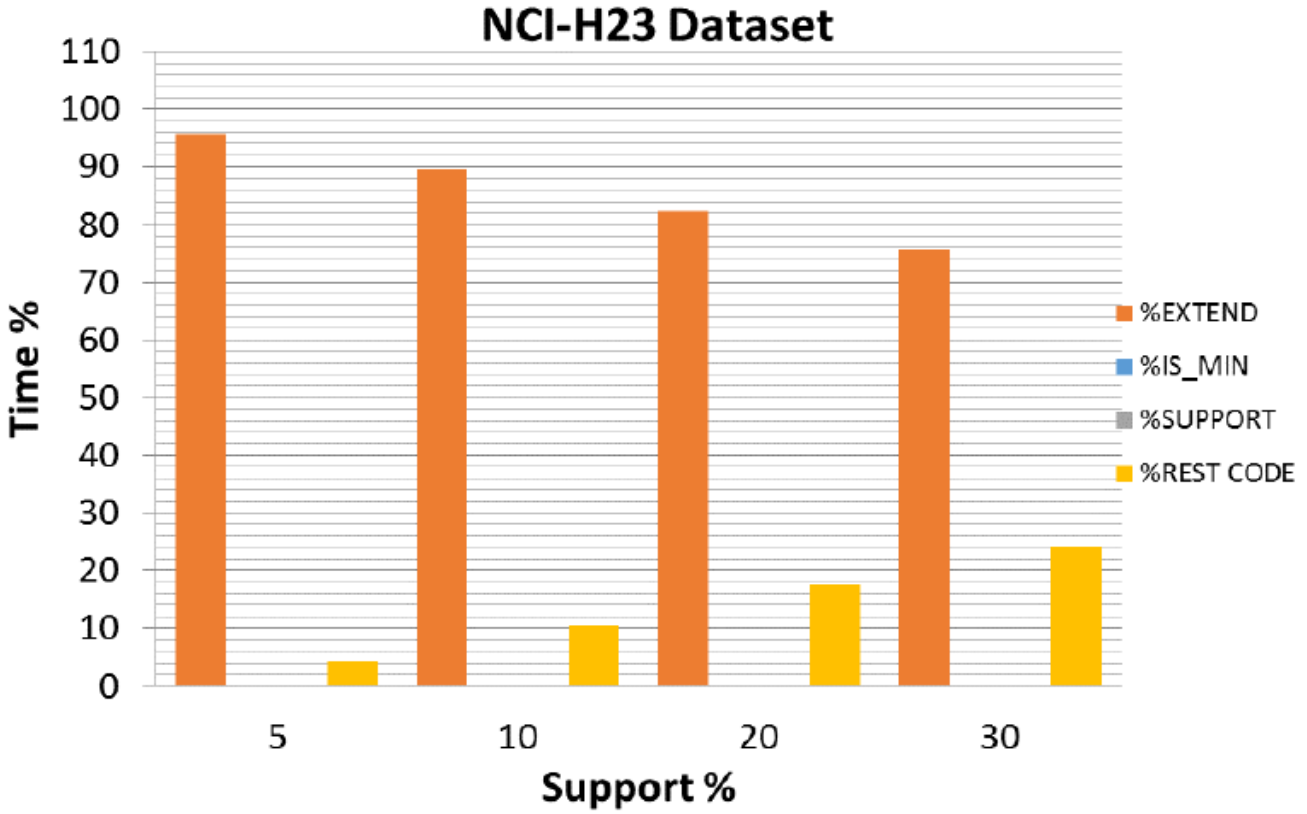
Για τον εντοπισμό των σημείων του κώδικα στα οποία ο επεξεργαστής σπαταλάει τον περισσότερο χρόνο (hotspots) χρησιμοποιήθηκαν οι profilers vtune, gperf και gprof. Τα αποτελέσματα που έδωσαν ήταν τα ίδια με πολύ μικρές αποκλίσεις. Όπως παρουσιάζονται παρακάτω, τα αποτελέσματα έδειξαν ότι τον περισσότερο χρόνο τον σπαταλάει ο επεξεργαστής στη διαδικασία επέκτασης των υπογράφων ενώ λιγότερο χρόνο περνάει στη διαδικασία ελέγχου ισομορφισμού και στον υπόλοιπο κώδικα. Όσο μικρότερο το support τόσο περισσότερος ο χρόνος στη διαδικασία επέκτασης στα dataset που εξετάστηκαν. Ακολουθούν τα αποτελέσματα του profiling.



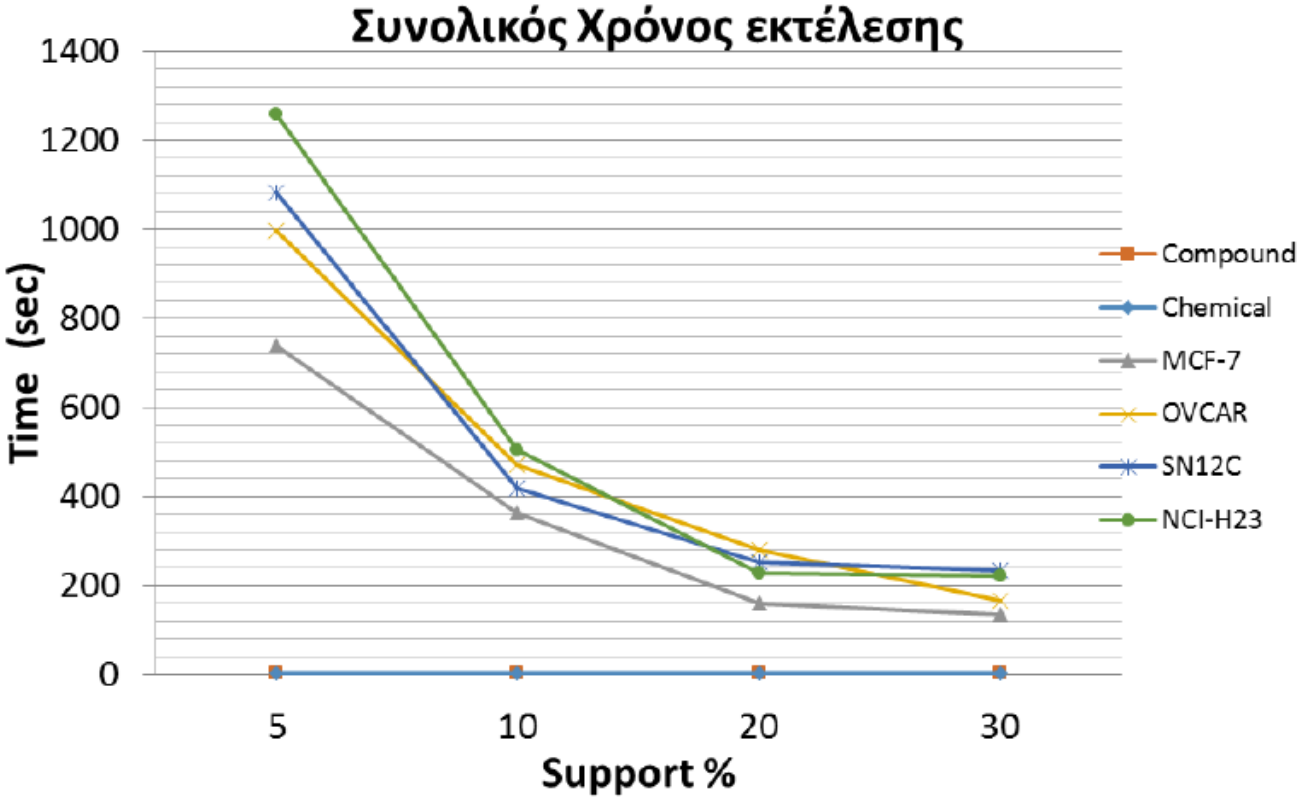
Εικόνα 3.4.4.1 : Καταμερισμός του χρόνου εκτέλεσης ανά συνάρτηση, με βάση τη παράμετρο support για το Compound Dataset (422 graphs).



Εικόνα 3.4.4.2 : Καταμερισμός του χρόνου εκτέλεσης ανά συνάρτηση, με βάση τη παράμετρο support για το MCF-7 Dataset (25.475 graphs).



Εικόνα 3.4.4.3 : Καταμερισμός του χρόνου εκτέλεσης ανά συνάρτηση, με βάση τη παράμετρο support για το NCI-H23 Dataset (38.295 graphs).



Εικόνα 3.4.4.4 : Συγκεντρωτικά αποτελέσματα του συνολικού χρόνου εκτέλεσης του αλγορίθμου gSpan για έξι Datasets.

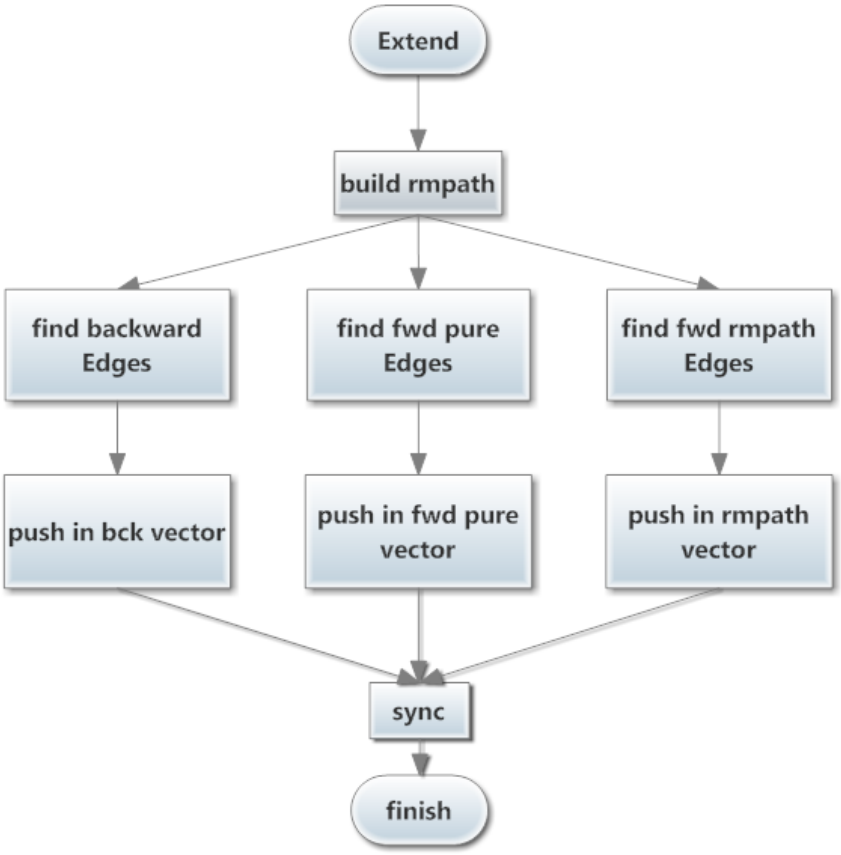
Ο συνολικός χρόνος εκτέλεσης στο παραπάνω διάγραμμα αναφέρεται στο χρόνο που απαιτείται από το gspan για να ολοκληρώσει τη διαδικασία εξόρυξης από την ανάγνωση του dataset εισόδου μέχρι και την εμφάνιση των αποτελεσμάτων. Οι μετρήσεις πραγματοποιήθηκαν σε έναν intel core i7 2.7GHz

Κεφάλαιο 4: Αρχιτεκτονικές

4.1 Γενικά

Η GPU που χρησιμοποιήθηκε για την υλοποίηση είναι η nvidia GeForce 780 GTX, η οποία όπως θα αναλύσουμε παρακάτω αναλαμβάνει το κομμάτι επέκτασης των υπογράφων. Σε αυτό το κεφάλαιο θα παρουσιαστούν οι διάφορες αρχιτεκτονικές και βελτιστοποιήσεις που ακολούθησαν μέχρι να καταλήξουμε στην πιο αποδοτική από πλευράς χρόνου εκτέλεσης. Το συγκεκριμένο κομμάτι είναι το ακριβότερο υπολογιστικά ολόκληρου του αλγόριθμου, όπως αναφέραμε και επιβεβαιώσαμε παραπάνω με τα αποτελέσματα των profiler. Έτσι επιταχύνοντάς το θεωρητικά θα έχουμε καλύτερη συνολική επιτάχυνση όλου του αλγόριθμου απ'ότι αν επιλέγαμε κάποιο άλλο κομμάτι κώδικα στο οποίο ο επεξεργαστής αφιερώνει λιγότερο χρόνο. Όλες οι υπόλοιπες λειτουργίες του αλγόριθμου, πέρα από την επέκταση των υπογράφων, καθώς και η τροποποίηση και μεταφορά των απαραίτητων δεδομένων από και προς τη GPU εκτελούνται από τον επεξεργαστή.

Η διαδικασία επέκτασης υπογράφων από τη φύση της είναι ιδιαίτερα παραλληλοποιήσιμη, καθώς η ανακάλυψη κάθε νέας ακμής δεν εξαρτάται από την ανακάλυψη κάποιας άλλης σε πρώτο επίπεδο. Οπότε στην ιδανική περίπτωση θα θέλαμε να μεταφέρουμε τα απαραίτητα δεδομένα στη GPU και εκείνη να πραγματοποιήσει τον υπολογισμό όλων των νέων ακμών – επεκτάσεων με παράλληλο τρόπο. Θα αναλύσουμε παρακάτω αυτή τη διαδικασία και θα δούμε πως αυτή η “μέγιστη” παραλληλοποίηση απαιτεί πολύ μεγάλη μνήμη. Τα κομμάτια που μπορούν να εκτελεστούν παράλληλα φαίνονται στο παρακάτω διάγραμμα.



Εικόνα 4.1 Διάγραμμα παράλληλης διαδικασίας επέκτασης υπογράφων

4.2 Δομές Δεδομένων

Πριν προχωρήσουμε σε βάθος στην ανάλυση αυτών των παραλληλοποιήσιμων κομματιών είναι απαραίτητο να αναφέρουμε τις δομές δεδομένων που χρησιμοποιούνται, ώστε να μελετήσουμε τις όποιες εξαρτήσεις μεταξύ τους. Τα δεδομένα αυτά είναι απαραίτητο να υποστούν κάποια επεξεργασία πριν μεταφερθούν στη μνήμη της GPU ώστε να είναι αξιοποιήσιμα από αυτή.

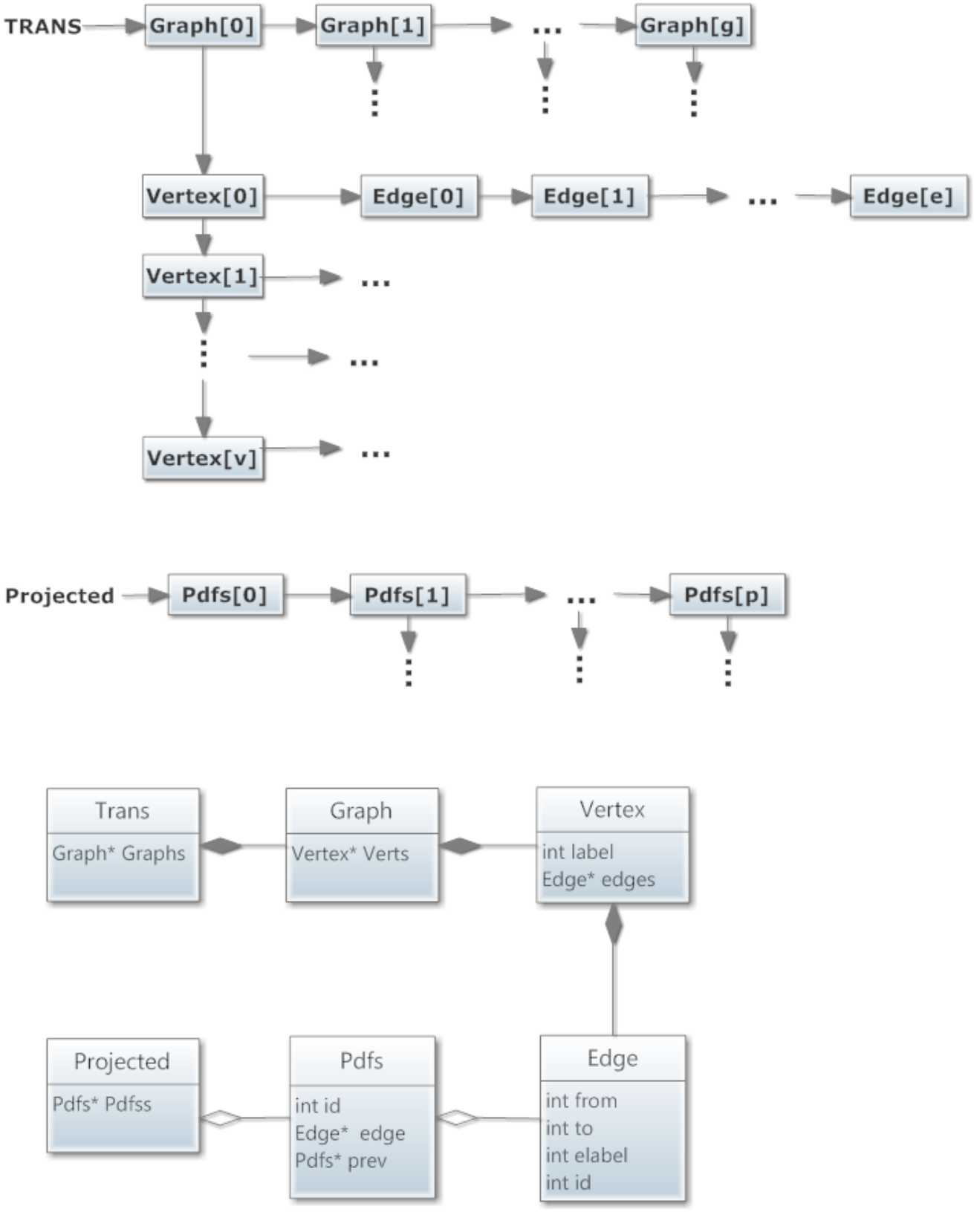
Το κυριότερο πρόβλημα που έπρεπε να αντιμετωπισθεί είναι η πολύπλοκη χρήση διάφορων pointer από συγκεκριμένες δομές. Αυτό διότι αν μεταφερθεί η τιμή του pointer ως έχει στη GPU τότε θα είναι άχρηστη αφού θα “δείχνει” σε κομμάτι μνήμης που είναι προσβάσιμο μόνο από τη CPU. Επίσης τις περισσότερες φορές η χρήση pointer στον kernel που εκτελεί η GPU δεν είναι η πιο αποδοτική λύση. Όλες οι δομές δεδομένων λοιπόν τροποποιήθηκαν σε τέτοιο βαθμό που να εξυπηρετούν την αρχιτεκτονική της GPU.

4.2.1 Οντότητες

Οντότητες που χρησιμοποιούνται στη διαδικασία επέκτασης, και άρα είναι άμεσου ενδιαφέροντος, είναι οι ακόλουθες:

- TRANS συνδεδεμένη λίστα με γράφους
- graph γράφος, συνδεδεμένη λίστα με κόμβους(vertex)
- vertex κόμβος, έχει μία ετικέτα (label) και μία λίστα με ακμές(edge)
- edge ακμή, έχει τα παρακάτω στοιχεία
  - from id κόμβου εκκίνησης
  - to id κόμβου προορισμού
  - elabel ετικέτα ακμής
  - id id ακμής
- pdfs κόμβος υπογράφου
  - id id αρχικού κόμβου εισόδου
  - edge ακμή
  - prev προηγούμενος κόμβος υπογράφου (pdfs)
- projected υπογράφος (λίστα με pdfs)

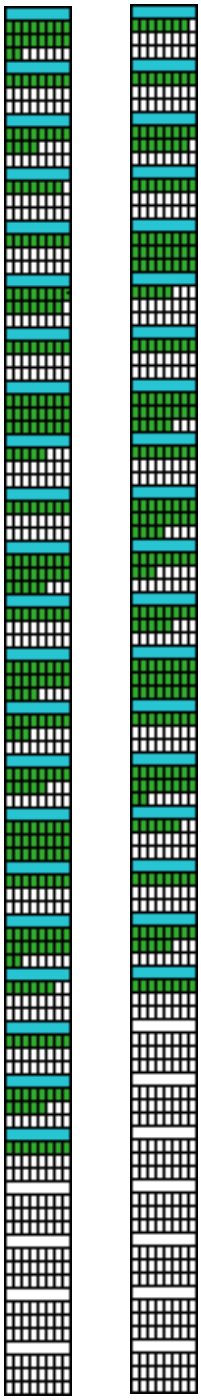
Οι οντότητες αυτές παρουσιάζονται και στο ακόλουθο διάγραμμα.



Εικόνα 4.2.1 Αναπαράσταση οντοτήτων που χρησιμοποιούνται από τον αλγόριθμο gpan

4.2.2 Χρήση Στατικών Πινάκων

Όσο αφορά τη μετατροπή που πρέπει να υποστούν ώστε να είναι αξιοποιήσιμες από τη GPU, το βέλτιστο θα ήταν να μετατραπούν όλες οι συνδεδεμένες λίστες σε στατικούς πίνακες. Αυτό όμως δεν είναι δυνατό στις περισσότερες περιπτώσεις καθώς υπάρχουν πολλά δυναμικά στοιχεία που μεταβάλλονται σε μεγάλο βαθμό κατά τη διάρκεια εκτέλεσης του αλγόριθμου. Αυτό απαιτεί αυξομείωση του αριθμού των εκάστοτε στοιχείων μέσα στη λίστα, το οποίο καθιστά αναποτελεσματική τη χρήση ενός στατικού πίνακα. Μία τακτική που μπορεί να εφαρμοστεί είναι να χρησιμοποιηθεί ένας αυθαίρετος σταθερός μέγιστος αριθμός για κάθε στοιχείο και να χρησιμοποιηθούν στατικοί πίνακες.



Αριστερά βλέπουμε μία αναπαράσταση δύο γράφων στη μνήμη υποθέτοντας τα παρακάτω

- Μέγιστος αριθμός κόμβων ενός γράφου: 26
- Μέγιστος αριθμός ακμών ενός κόμβου: 24

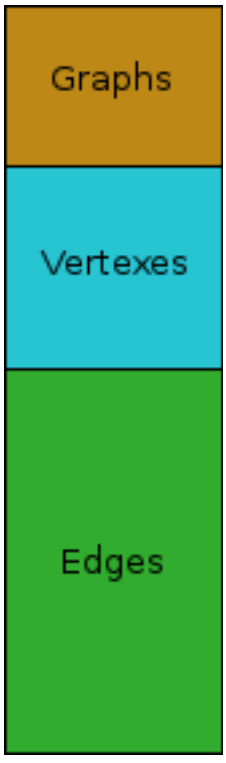
Ανεξάρτητα από τους κόμβους και τις ακμές που έχει συνολικά ο κάθε γράφος, όλοι καταλαμβάνουν τον ίδιο χώρο.

Εικόνα 4.2.2 Αναπαράσταση της μνήμης δύο γράφων με χρήση στατικών μέγιστων παραμέτρων για τους κόμβους και τις ακμές

4.2.3 Χρήση Pointer μαζί με στατικούς πίνακες

Όπως παρατηρούμε αρκετά μεγάλο κομμάτι μνήμης μένει αχρησιμοποίητο με την προηγούμενη μέθοδο γι'αυτό και δεν προτιμάται αυτός ο τρόπος αποθήκευσης των γράφων συγκεκριμένα. Ακόμη και αν δε μας ενδιέφερε αυτή η σπατάλη μνήμης, είναι αρκετά δύσκολο έως αδύνατο να προβλέψουμε το μέγιστο αριθμό στοιχείων σε κάποιες περιπτώσεις. Αντί γι'αυτό χρησιμοποιήθηκαν η δομή πίνακα αλλά και pointers μαζί. Συγκεκριμένα η κάθε οντότητα αναπαριστάται ως εξής:

- graph
  - vertex_size αριθμός κόμβων που έχει ο γράφος
  - *vertex δείκτης προς τους κόμβους
- vertex
  - label ετικέτα κόμβου
  - edge_size αριθμός ακμών κόμβου
  - *edge δείκτης προς τις ακμές
- edge
  - from id κόμβου εκκίνησης
  - to id κόμβου προορισμού
  - elabel ετικέτα ακμής
  - id id ακμής
  - *cpu_pointer δείκτης προς τη συγκεκριμένη ακμή (στη μνήμη της CPU)



Οι αριθμοί κόμβων και ακμών είναι απαραίτητοι για χρήση της δομής στη GPU καθώς δεν είναι μία κλασική συνδεδεμένη λίστα και είναι απαραίτητο να γνωρίζουμε το πλήθος των στοιχείων με αυτό τον τρόπο. Επίσης ο pointer της ακμής προς τον εαυτό της με μια πρώτη ματιά φαίνεται περιττός, όμως χρησιμοποιείται στα δεδομένα επιστροφής από τη GPU ώστε να είναι εύκολος ο εντοπισμός της μνήμης της συγκεκριμένης ακμής (που είναι προσβάσιμη από τη CPU).

Με αυτό τον τρόπο χρησιμοποιείται ακριβώς η μνήμη που χρειάζεται χωρίς να έχουμε όλο το επιπλέον κόστος των αναζητήσεων στη μνήμη (λόγω pointer) που θα είχαμε με μια κανονική συνδεδεμένη λίστα.

Εικόνα 4.2.3.1 Αναπαράσταση της μνήμης όλων των γράφων αποθηκευμένων με τον τρόπο που περιγράφηκε παραπάνω

4.2.4 Τροποποιημένη Δομή Pdfs

Όσο αφορά τα pdfs η δομή τους έχει τροποποιηθεί αρκετά για να χρησιμοποιηθεί από τη GPU. Το νέο pdfs περιλαμβάνει επιπλέον πληροφορία.

- Pdfs
  - id id αρχικού κόμβου εισόδου
  - has_edge lookup table με ακμές
  - has_vertex lookup table με κόμβους
  - cpu_pointer pointer στο ίδιο το pdfs
  - history_size μέγεθος του edge_array
  - edge_array πίνακας με ακμές

Το id είναι το ίδιο με πριν και το cpu_pointer είναι απλά ένας δείκτης προς το ίδιο το pdfs. Χρησιμοποιείται αντίστοιχα, όπως είδαμε παραπάνω και στα edge, στα δεδομένα επιστροφής από τη GPU ώστε να είναι εύκολος ο εντοπισμός της μνήμης του συγκεκριμένου pdfs (που είναι προσβάσιμη από τη CPU).

Τα has_edge και has_vertex είναι πίνακες με μέγεθος ίσο με το μέγιστο αριθμό ακμών και κόμβων αντίστοιχα που έχει το dataset εισόδου. Χρησιμοποιούνται ως lookup tables για να αποφύγουμε την αναζήτηση ακμών και κόμβων κατά τη διαδικασία επιλογής ή απόρριψης μιας νέας ακμής ως προέκταση. Για να αποφύγουμε τη δημιουργία αυτών των πινάκων στη GPU τους δημιουργούμε από πριν και τους μεταφέρουμε στη δομή του pdfs. Έτσι πραγματοποιείται γρηγορότερα η διαδικασία καθώς η CPU είναι αποδοτικότερη για αυτή την εργασία.

Επίσης το edge_array, στην αρχική υλοποίηση δημιουργείται διατρέχοντας τη λίστα με τα pdfs με χρήση του pointer prev. Αυτό δε θα ήταν καθόλου αποδοτικό στη GPU οπότε και αυτή η διεργασία εκτελείται από τον επεξεργαστή, δημιουργείται η λίστα με τις ακμές και μεταφέρεται στη GPU μέσω της δομής του pdfs

Αρχικά χρησιμοποιήσαμε στατικούς πίνακες για την αναπαράσταση της δομής αυτής με σταθερό μέγιστο αριθμό στοιχείων στο edge_array. Χρησιμοποιήθηκε αυτή η μορφή σε πρώτο στάδιο καθώς η υλοποίησή της είναι αρκετά πιο εύκολη.

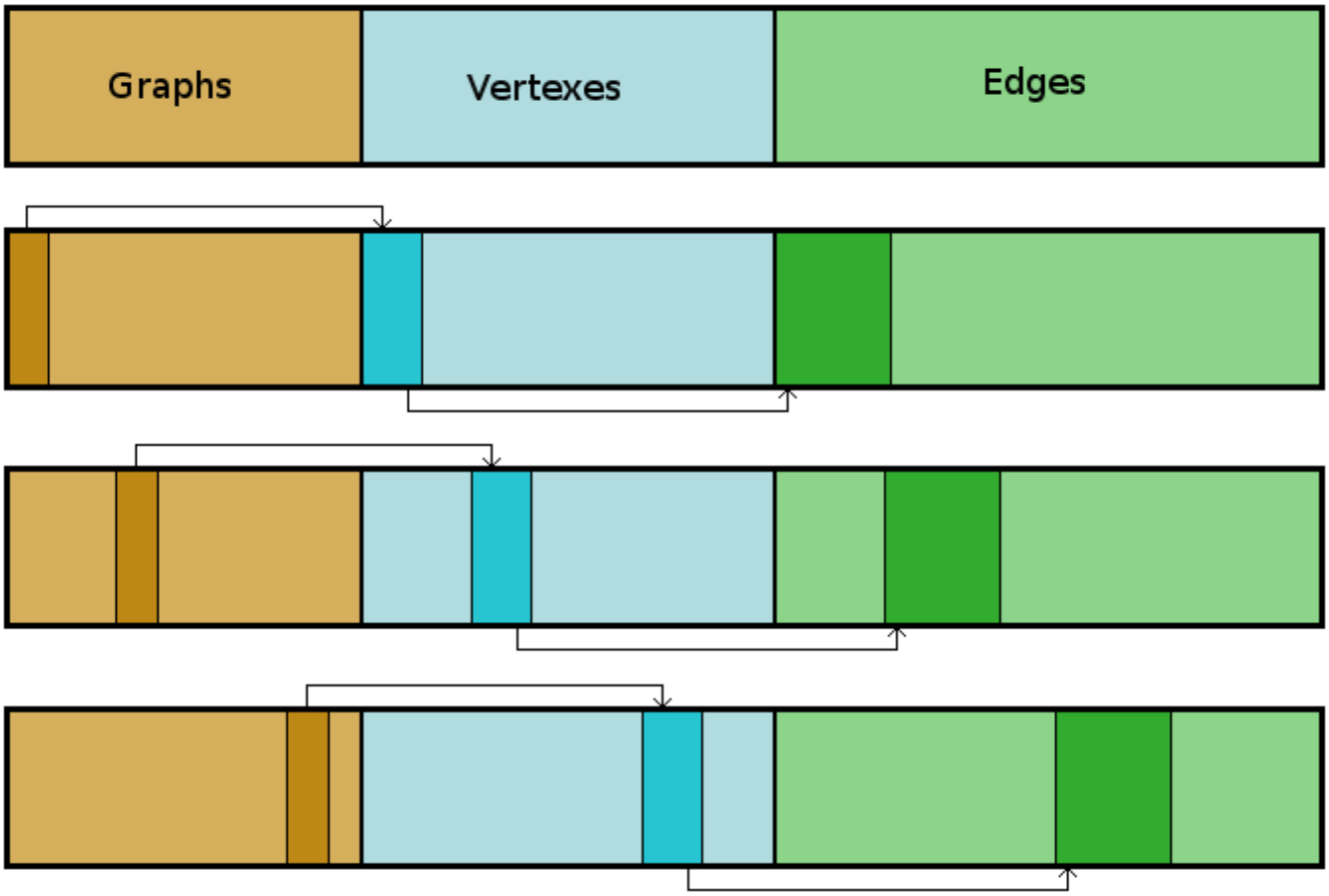
Σε επόμενο στάδιο υπολογίζεται δυναμικά κατά την εκτέλεση του αλγόριθμου ο ακριβής αριθμός ακμών και κόμβων που θα χρειαστεί να μεταφερθούν και χρησιμοποιείται η απαραίτητη μνήμη. Έτσι μειώνεται ο όγκος των δεδομένων που πρέπει να μεταφερθούν προς τη GPU. Θα αναλύσουμε περεταίρω αυτή τη βελτιστοποίηση στο κεφάλαιο μεταφοράς δεδομένων προς τη GPU.

4.3 Μεταφορά δεδομένων προς GPU

4.3.1 Αντιγραφή dataset

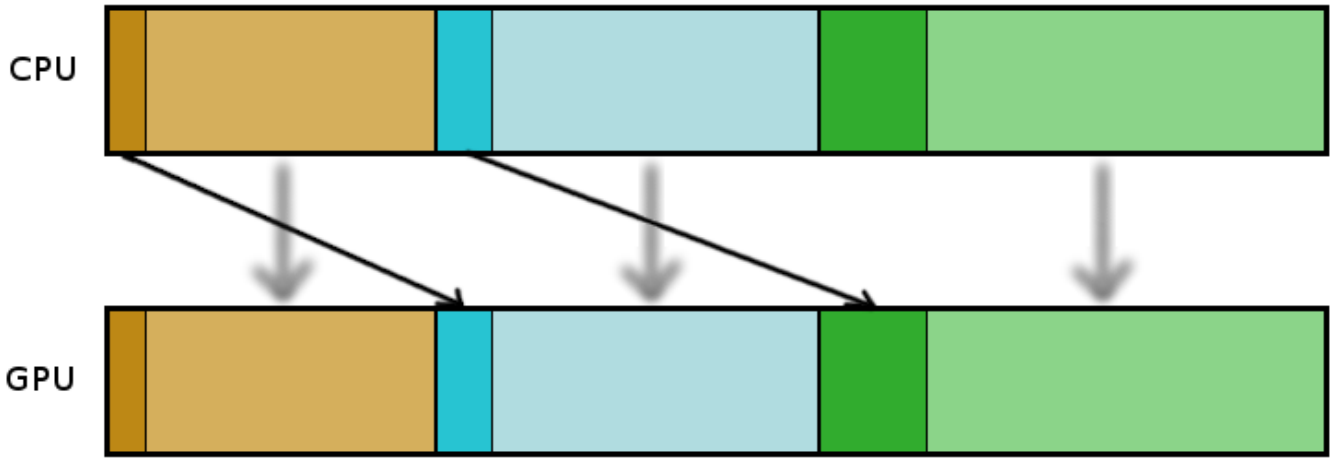
Ο αλγόριθμος gspan απαιτεί ολόκληρο το dataset των αρχικών γράφων να βρίσκεται στη μνήμη πριν ξεκινήσει η εκτέλεση. Η πληροφορία αυτή είναι ιδιαίτερα χρήσιμη κατά τη διαδικασία επέκτασης των γράφων, ουσιαστικά δεν είναι δυνατή η επέκταση χωρίς αυτή. Επομένως το dataset πρέπει να μεταφερθεί και στη μνήμη της GPU ώστε να είναι δυνατή η διαδικασία. Η συνάρτηση αντιγραφής cudamemcpy από τη μνήμη της CPU στη GPU έχει αρκετό overhead οπότε όσο λιγότερες κλήσεις σε αυτή τη συνάρτηση τόσο καλύτερα. Αρχικά, καθώς ήταν ευκολότερη η υλοποίηση, χρησιμοποιήθηκαν πολλές κλήσεις αυτής της συνάρτησης. Κάθε μια μετέφερε ένα μικρό κομμάτι δεδομένων μέχρι να ολοκληρωθεί η αντιγραφή όλων των γράφων του dataset εισόδου. Ο συνολικός χρόνος που ήταν απαραίτητος για να επιτευχθεί η μεταφορά δεν ήταν ικανοποιητικός γι'αυτό η προσοχή εστιάστηκε στη μείωση των κλήσεων της cudamemcpy. Μέσα από διάφορες υλοποιήσεις παρατηρήθηκε ότι όσο μειωνόταν ο αριθμός κλήσεων της τόσο αυξανόταν η απόδοση. Στην τελευταία βέλτιστη υλοποίηση πραγματοποιείται ολόκληρη η μεταφορά του dataset εισόδου μόνο με μία κλήση. Η διαφορά χρόνου σε σχέση με την πρώτη είναι εντυπωσιακή καθώς ξεπερνάει τις δύο τάξεις μεγέθους, μόνο μειώνοντας τις κλήσεις τις συνάρτησης cudamemcpy.

Για να επιτευχθεί το παραπάνω αποτέλεσμα ήταν απαραίτητο να δημιουργηθεί μία δομή σε συνεχόμενη μνήμη στη CPU έτοιμη προς αξιοποίηση από τη GPU. Έτσι με μία μόνο αντιγραφή και χωρίς καμία περεταίρω τροποποίηση θα βρισκόταν στην επιθυμητή κατάσταση στη μνήμη της GPU. Η δομή αυτή έχει την παρακάτω μορφή.



Εικόνα 4.3.1.1 Αναπαράσταση της μνήμης των γράφων

Έχουμε αναλύσει παραπάνω τις επιμέρους οντότητες και τα στοιχεία τους. Κάθε γράφος έχει ένα δείκτη που δείχνει στην αρχή της μνήμης των κόμβων του και κάθε κόμβος έχει έναν που δείχνει στην αρχή της μνήμης των ακμών του. Αν έδειχναν αυτοί οι δείκτες σε μνήμη της CPU, μετά τη μεταφορά, θα ήταν άχρηστοι στη GPU, οπότε έχει γίνει το εξής. Καθώς δημιουργείται στη μνήμη του επεξεργαστή η παραπάνω δομή αντί οι δείκτες να δείχνουν σε αυτή τη μνήμη, δείχνουν στα κατάλληλα κομμάτια μνήμης GPU. Έτσι μετά τη μεταφορά της δομής όλοι οι δείκτες θα είναι έγκυροι και θα δείχνουν στη σωστή μνήμη.

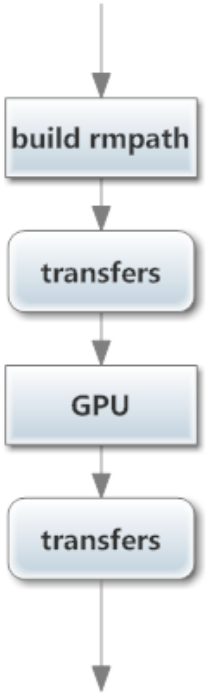


Εικόνα 4.3.1.2 Pointer πριν την αντιγραφή στη GPU

Αυτό είναι δυνατό καθώς με μία απλή προσπέλαση του dataset μπορούμε να γνωρίζουμε το ακριβές μέγεθος των γράφων, κόμβων και ακμών οπότε αρχικοποιούμε την απαραίτητη μνήμη στη GPU και έχουμε γνώση των θέσεων που θα καταλάβει μετά την αντιγραφή ο κάθε γράφος, κόμβος και η κάθε ακμή. Οπότε πριν καν πραγματοποιηθεί η αντιγραφή έχουμε την απαραίτητη πληροφορία για να αρχικοποιήσουμε όλους τους δείκτες σωστά. Θα δείχνουν στη μνήμη όπου μετά την αντιγραφή θα έχει την αντίστοιχη πληροφορία.

4.3.2 Δεδομένα για τη συνάρτηση επέκτασης

Παραπάνω μιλήσαμε για τη μεταφορά των γράφων του αρχικού dataset στη GPU η οποία πραγματοποιείται μία φορά στην αρχή. Τώρα θα αναλύσουμε τα δεδομένα που μεταφέρονται από και προς τη GPU για την εκτέλεση της διαδικασίας επέκτασης γράφων.



Εικόνα 4.3.2 Μεταφορές δεδομένων πριν και μετά τη GPU

Όλη η απαραίτητη πληροφορία που χρειάζεται η GPU για να ξεκινήσει τη διαδικασία βρίσκεται στη δομή pdfs με τις επιπλέον προσθήκες, όπως αυτή περιγράφηκε παραπάνω, και βρίσκεται σε μορφή έτοιμη για μεταφορά και αξιοποίηση. Τα δεδομένα που επιστρέφονται έχουν τη μορφή ενός δισδιάστατου ή τρισδιάστατου πίνακα του οποίου τα κελιά περιέχουν μία συνδεδεμένη λίστα με pdfs. Αυτοί οι πίνακες είναι πολύ αραιοί οπότε για να αποφύγουμε την περιττή κατανάλωση μνήμης χρησιμοποιήσαμε τις παρακάτω δομές για το δισδιάστατο και τρισδιάστατο πίνακα αντίστοιχα.

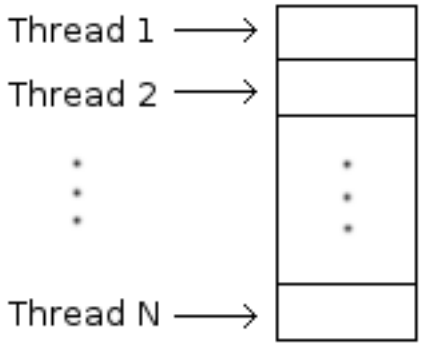
- bck_root
  - a index πίνακα
  - b index πίνακα
  - id δεδομένα pdfs (id)
  - edge δεδομένα pdfs (edge)
  - pdfs δεδομένα pdfs (pdfs)
- fwd_root
  - a index πίνακα
  - b index πίνακα
  - c index πίνακα
  - id δεδομένα pdfs (id)
  - edge δεδομένα pdfs (edge)
  - pdfs δεδομένα pdfs (pdfs)

Όπως βλέπουμε κάθε στοιχείο μιας συνδεδεμένης λίστας με pdfs μέσα στο δισδιάστατο πίνακα αναπαριστάται από τη δομή bck_root ενώ αντίστοιχα στον τρισδιάστατο από την fwd_root. Αυτές οι δομές κρατάνε τα δεδομένα που έχει το συγκεκριμένο pdfs στη συνδεδεμένη λίστα αλλά και τις συντεταγμένες του πίνακα όπου ανήκει αυτή συνδεδεμένη λίστα. Χρησιμοποιώντας αυτή την πληροφορία που επιστρέφεται από τη GPU ο επεξεργαστής μπορεί να ανακατασκευάσει τους πίνακες στην αρχική μορφή τους ώστε να συνεχίσει ο αλγόριθμος την εκτέλεσή του. Με αυτό τον τρόπο, κρατώντας μόνο τα indexes και την πληροφορία του pdfs (id,edge,pdfs) χρησιμοποιούμε πολύ λιγότερο χώρο στη μνήμη καθώς όπως είπαμε οι πίνακες αυτοί είναι πολύ αραιοί καθώς επίσης μειώνουμε και τον όγκο των δεδομένων που μεταφέρονται.

Μία βελτιστοποίηση που έγινε σε αυτό το κομμάτι είναι η χρησιμοποίηση μιας δομής pdfs με επιπρόσθετα στοιχεία σε σχέση με την αρχική, η οποία όμως δεν χρησιμοποιεί αυθαίρετους μέγιστους αριθμούς για τα στοιχεία της όπως αναλύσαμε προηγουμένως, αλλά κάθε φορά χρησιμοποιεί ακριβώς όσο χώρο χρειάζεται. Με αυτό τον τρόπο μειώνεται σημαντικά η πληροφορία που μεταφέρεται προς τη GPU και αυξάνεται η απόδοση του συστήματος. Η δυναμική εκχώρηση μνήμης δεν αποτελεί αποτελεσματική μέθοδο όταν ο χρόνος εκτέλεσης είναι σημαντικός. Για να δημιουργούμε μία λίστα με pdfs προς μεταφορά στη GPU κάθε φορά, πριν την εκτέλεση της παράλληλης επέκτασης υπογράφων, θα έπρεπε να κάνουμε δυναμική εκχώρηση μνήμης ώστε να έχουμε ακριβώς το χώρο που απαιτείται για τα pdfs τόσο στη μνήμη της CPU όσο και στης GPU. Αυτό δε θα ήταν αποδοτικό αν σκεφτούμε το πλήθος των εκχωρήσεων μνήμης που, ανάλογα το dataset, έφτανε τεράστια νούμερα. Αντί γι' αυτό λοιπόν εκχωρούμε ένα μεγάλο κομμάτι μνήμης στην αρχή και το χρησιμοποιούμε σε όλη τη διάρκεια εκτέλεσης του αλγόριθμου. Τις περισσότερες φορές δεν χρησιμοποιείται ολόκληρο παρά μόνο ένα μέρος αυτού. Μεταφέρεται στη GPU μόνο η χρήσιμη πληροφορία, οπότε κερδίζουμε σε χρόνο. Σε περίπτωση που ο χώρος αυτός δεν επαρκεί σε κάποια επανάληψη της εκτέλεσης τότε εκχωρούμε ένα ακόμη μεγαλύτερο κομμάτι. Με αυτό τον τρόπο χρησιμοποιούμε περισσότερη μνήμη η οποία δεν χρησιμοποιείται ολόκληρη στο μεγαλύτερο ποσοστό του χρόνου, όμως μειώνουμε τις κλήσεις στη συνάρτηση εκχώρησης μνήμης και τον όγκο μεταφοράς δεδομένων προς τη GPU. Ως αποτέλεσμα έχουμε μείωση του συνολικού χρόνου εκτέλεσης του προγράμματος.

4.4 Ανάλυση παράλληλης συνάρτησης επέκτασης

Χωρίσαμε τη λειτουργία της συνάρτησης extend σε τρία παραλληλοποιήσιμα κομμάτια. Την αναζήτηση των backward ακμών, των forward pure και των forward rmpath. Αναλύοντας περαιτέρω τη λειτουργία αυτών βλέπουμε ότι ουσιαστικά αποτελούνται από έναν έλεγχο που θα κρίνει αν η εξεταζόμενη ακμή είναι έγκυρη backward,forward pure, ή forward rmpath ακμή και από μία καταχώρηση στη μνήμη της ακμής αυτής στην περίπτωση που είναι έγκυρη. Ο κάθε έλεγχος με την αντίστοιχη εγγραφή στη μνήμη είναι ανεξάρτητος από όλες τις υπόλοιπες, πράγμα το οποίο σημαίνει ότι μπορούν να παραλληλοποιηθούν. Στην ιδανική περίπτωση όλοι οι έλεγχοι με τις αντίστοιχες εγγραφές θα θέλαμε να εκτελεστούν παράλληλα.



Εικόνα 4.4 Κάθε thread γράφει στο “δικό του” χώρο στη μνήμη

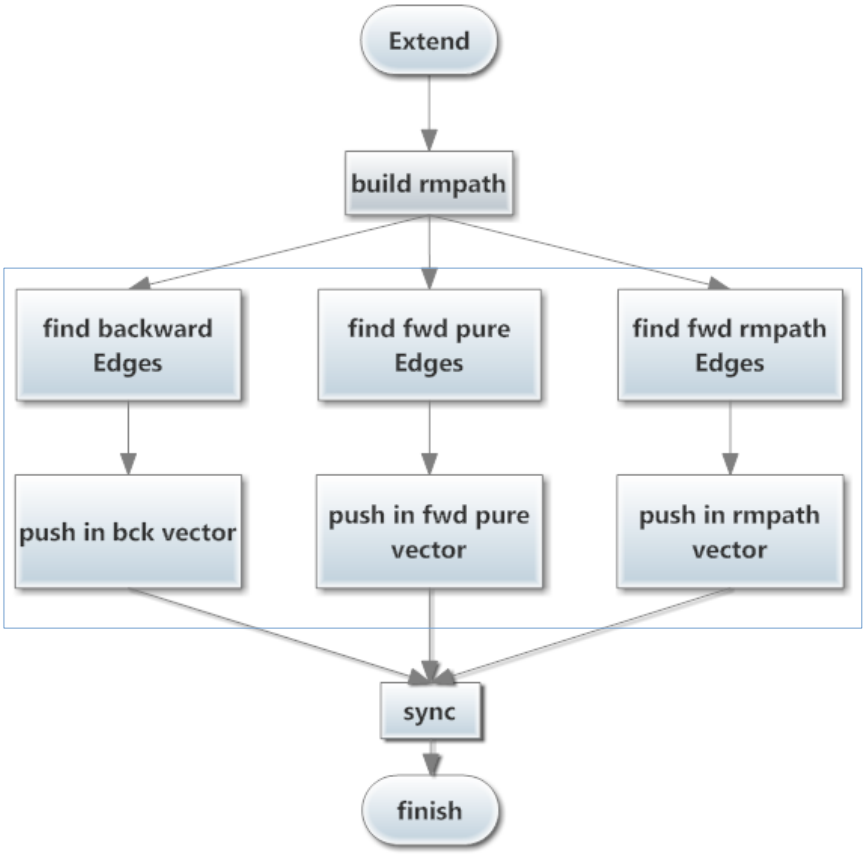
Αυτή η ιδέα για να υλοποιηθεί απαιτεί κάθε thread να αναλάβει έναν έλεγχο με την αντίστοιχη εγγραφή, το οποίο με τη σειρά του σημαίνει πως κάθε thread χρειάζεται ένα κομμάτι μνήμης που θα κρατήσει την ενδεχόμενη νέα ακμή, ανεξάρτητα από το αν θα πραγματοποιηθεί ή όχι η εγγραφή. Αυτό είναι απαραίτητο καθώς δεν γνωρίζουμε τον ακριβή αριθμό εγγραφών παρά μόνο μετά την ολοκλήρωση του κάθε ελέγχου εγκυρότητας της νέας ακμής.

Εικόνα 4.4 Το κάθε είδος ακμής έχει το δικό του χώρο στη μνήμη

Ο χώρος μνήμης που χρησιμοποιείται έχει χωριστεί σε τρία κομμάτια. Το χώρο αποθήκευσης των backward ακμών, των forward pure και των forward rmpath. Και στους τρεις πρέπει να εκχωρηθεί το θεωρητικό μέγιστο μέγεθος που θα καταλάμβαναν οι νέες ακμές αν όλοι οι έλεγχοι ήταν θετικοί για να λειτουργήσει η συγκεκριμένη υλοποίηση.

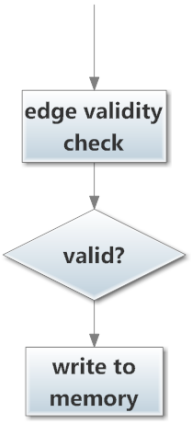
4.5 Υλοποιήσεις Kernel

Σε αυτό το κεφάλαιο θα παρουσιαστούν όλες οι υλοποιήσεις και βελτιστοποιήσεις από τις οποίες περάσαμε για να καταλήξουμε στην τελική. Παρακάτω βλέπουμε τον κώδικα προς εκτέλεση στη GPU και τα παραλληλοποιήσιμά του κομμάτια. Είναι οι τρεις αναζητήσεις επεκτάσεων χωρισμένες ανά κατηγορία. Όμως όπως αναλύσαμε στην προηγούμενη ενότητα παραλληλοποίηση δεν περιορίζεται μόνο εδώ.



Εικόνα 4.5.1 Κομμάτια προς εκτέλεση στη GPU (στο μπλε περίγραμμα)

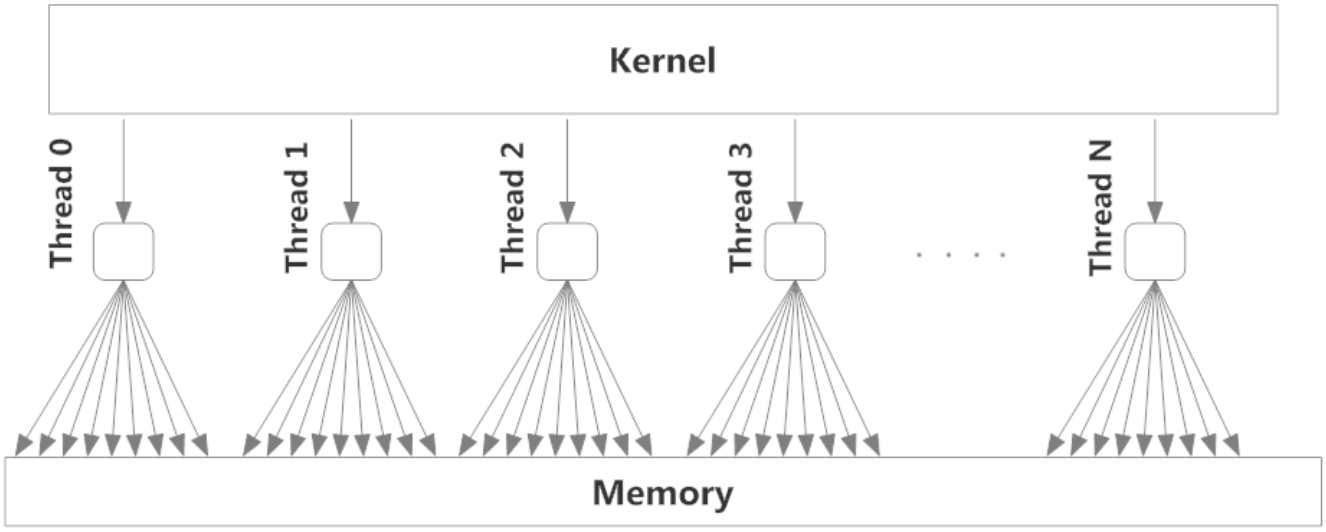
Όπως αναφέραμε στο κεφάλαιο 4.4 η καλύτερη παραλληλοποίηση που μπορούμε να πετύχουμε είναι στο επίπεδο του ελέγχου εγκυρότητας μιας νέας ακμής ως επέκταση και η εγγραφή αυτής στη μνήμη. Οπότε σκοπός μας είναι το κάθε thread να αναλάβει μία τέτοια εργασία και να εκτελεστούν όλες παράλληλα.



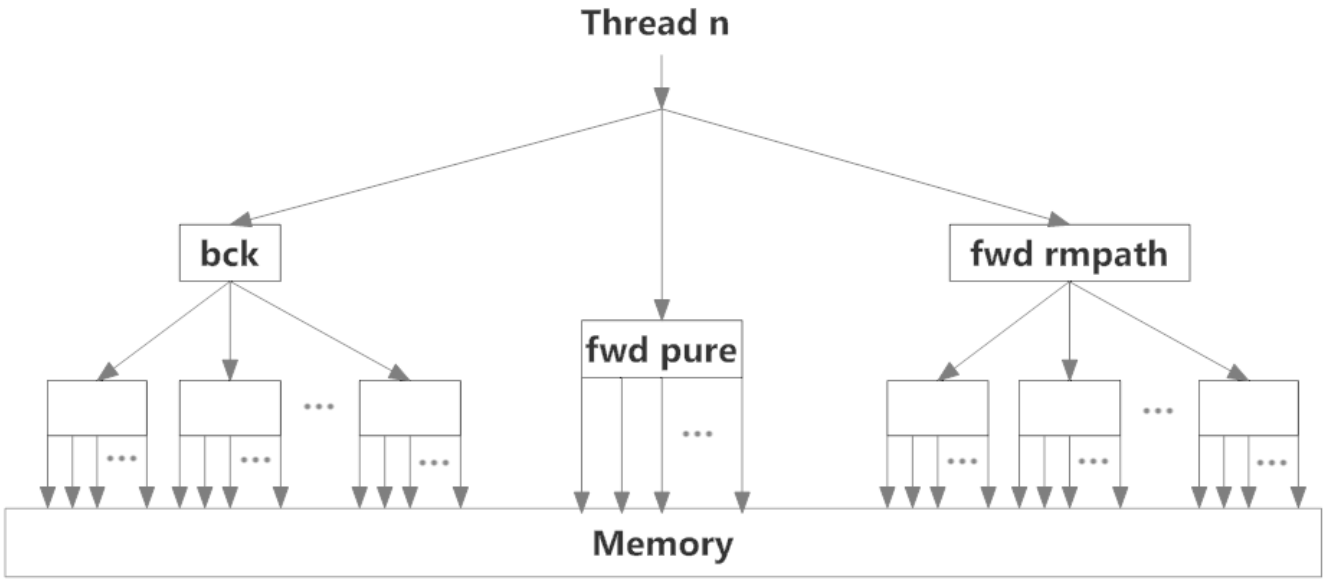
Εικόνα 4.5.2 Εργασία προς εκτέλεση από ένα thread

4.5.1 Kernel με Dynamic Parallelism

Ο ακριβής αριθμός των ελέγχων που θα γίνουν κατά την εκτέλεση δεν είναι γνωστός εκ των προτέρων παρά μόνο μετά την ολοκλήρωση της. Είναι αδύνατο λοιπόν να γνωρίζουμε από την αρχή πόσα threads θα δημιουργήσουμε για να εκτελέσουμε τη διαδικασία. Γι' αυτό χρησιμοποιούμε το dynamic parallelism της cuda το οποίο μας επιτρέπει να καλέσουμε έναν kernel μέσα από έναν άλλο. Αυτό σημαίνει ότι καθώς εκτελείται η συνάρτηση extend από κάποια threads αυτά με τη σειρά τους θα μπορούν να δημιουργήσουν και άλλα χωρίς να παρέμβει η cpu. Έτσι μπορούμε κατά την εκτέλεση της συνάρτησης επέκτασης να έχουμε ακριβώς τον αριθμό των thread που χρειάζονται ώστε κάθε thread να πραγματοποιήσει έναν έλεγχο και μία εγγραφή αν ο έλεγχος ήταν επιτυχής.



Εικόνα 4.5.1.1 Το κάθε thread δημιουργεί δυναμικά άλλα threads



Εικόνα 4.5.1.2 Kernels που καλεί ένα thread για να δημιουργήσει άλλα χωρίς την παρέμβαση της CPU

Ο επεξεργαστής αρχικά καλεί τον βασικό kernel ο οποίος δημιουργεί threads τόσα όσα είναι και τα pdfs στη δομή projected που έχουμε μεταφέρει στη GPU. Στη συνέχεια ο kernel αυτός καλεί άλλους τρεις καθώς κατά την εκτέλεση γίνεται γνωστό πόσα ακόμη threads είναι απαραίτητα. Έναν που θα αναλάβει τη δημιουργία thread για τις backward ακμές, έναν για τις forward pure και έναν για τις forward rmpath. Τέλος ο backward και ο forward rmpath θα δημιουργήσουν από άλλον ένα ο καθένας. Σε αυτό το στάδιο έχουμε όσα threads χρειάζονται για να εκτελεσθούν παράλληλα όλοι οι έλεγχοι και οι ενδεχόμενες εγγραφές. Μετά την ολοκλήρωση αυτής της διαδικασίας μεταφέρεται ολόκληρη αυτή η μνήμη και η cpu αναλαμβάνει την επεξεργασία αυτής της πληροφορίας και τη συνέχιση του αλγόριθμου.

Με αυτό τον τρόπο ναι μεν έχουμε ακριβώς τον αριθμό των thread που είναι απαραίτητα για την πλήρη παραλληλοποίηση της διαδικασίας extend, όμως έχουμε κάποια μειονεκτήματα που καθιστούν αυτή τη μέθοδο καθόλου αποτελεσματική.

- Overhead δημιουργίας kernel - thread

  Ο χρόνος δημιουργίας ενός καινούργιου kernel αλλά και των αντίστοιχων thread είναι μη αμελητέος με αποτέλεσμα να δημιουργείται τεράστιο overhead στη διαδικασία που περιγράψαμε παραπάνω. Ως αποτέλεσμα είχαμε το συνολικό χρόνο εκτέλεσης όταν χρησιμοποιούμε την κάρτα γραφικών να είναι συγκρίσιμος ή και χειρότερος από το χρόνο εκτέλεσης του προγράμματος εξολοκλήρου στον επεξεργαστή.

- Σπατάλη μνήμης

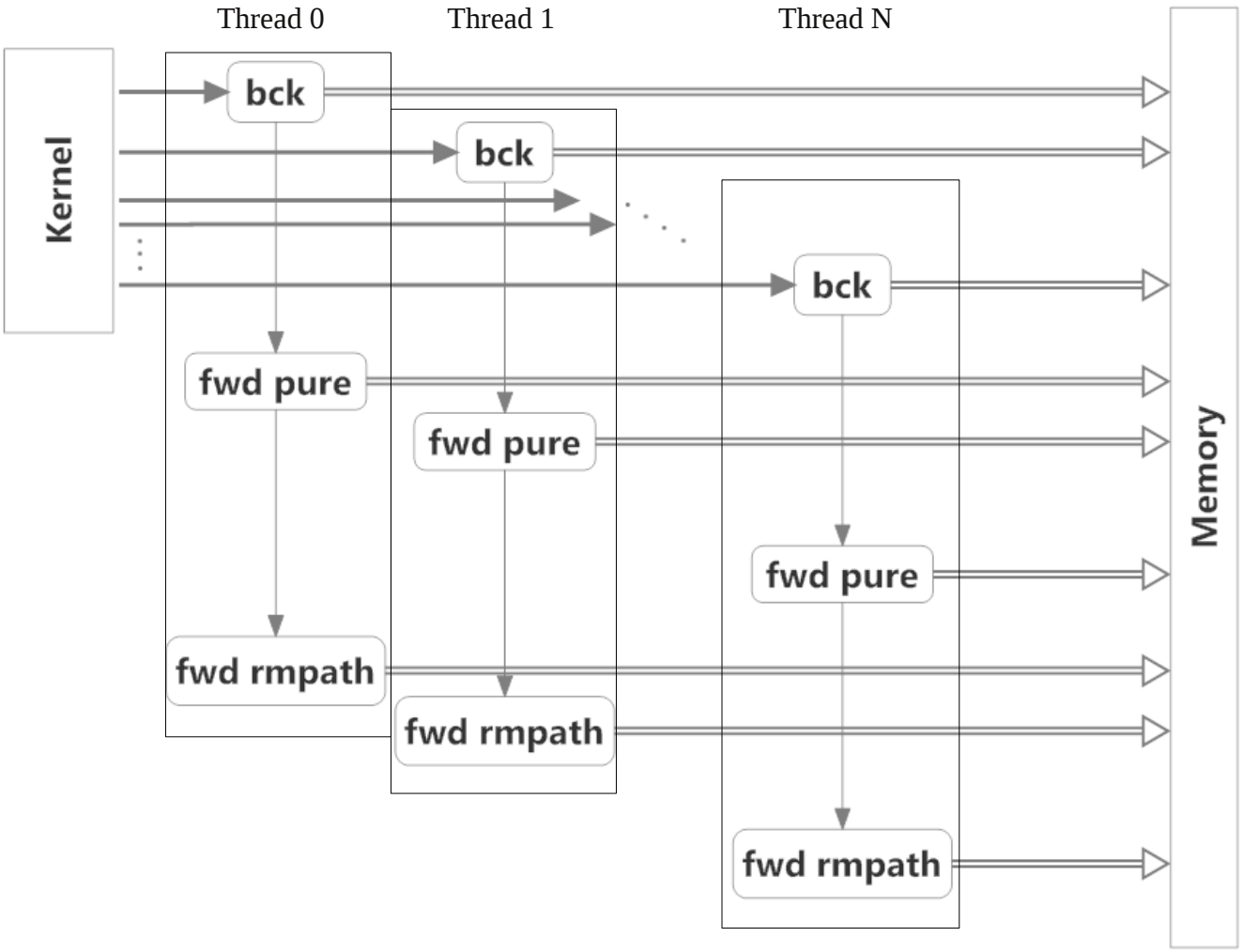
  Επίσης η μνήμη που είναι απαραίτητη για τη διαδικασία που περιγράφηκε παραπάνω είναι πολύ μεγάλη, αφού θεωρούμε ότι κάθε έλεγχος θα είναι θετικός για να λειτουργήσει η παραλληλία. Στην πραγματικότητα μόνο ένα μικρό ποσοστό της μνήμης αυτής γράφεται με νέες ακμές ενώ το υπόλοιπο παραμένει μηδενικό.

- Μεγαλύτερος όγκος δεδομένων προς μεταφορά

  Κάτι ακόμη που συμβάλει στην κακή απόδοση αυτής της αρχιτεκτονικής είναι το γεγονός ότι ολόκληρη η μνήμη μεταφέρεται πίσω στη cpu μαζί με τα μηδενικά, δηλαδή εκείνες τις θέσεις μνήμης στις οποίες δεν πραγματοποιήθηκε εγγραφή νέας ακμής. Δεν υπάρχει τρόπος διαχωρισμού της χρήσιμης από την άχρηστη πληροφορία με αυτή τη μέθοδο.

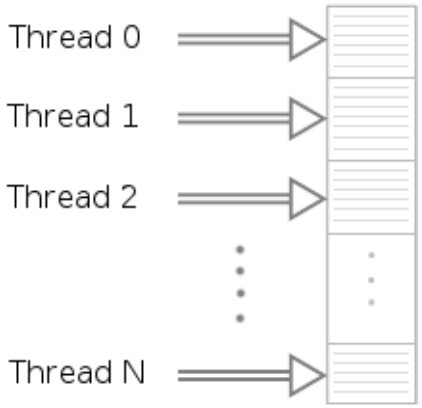
4.5.2 Kernel που αναλαμβάνει και τα τρία είδη ακμών

Όπως αναλύσαμε προηγουμένως το dynamic parallelism στην περίπτωσή μας δεν είναι αποδοτική προσέγγιση παρόλο που έχουμε ακριβώς όσα threads είναι απαραίτητα. Σε αυτή την αρχιτεκτονική χρησιμοποιούμε έναν kernel ο οποίος αναλαμβάνει και τις backward και τις forward pure και τις forward rmpath ακμές. Ο αριθμός των thread εδώ είναι μικρότερος καθώς ένα thread αναλαμβάνει πολλαπλούς ελέγχους και εγγραφές στη μνήμη πρώτα για τις backward μετά για τις forward pure και τέλος για τις forward rmpath ακμές.



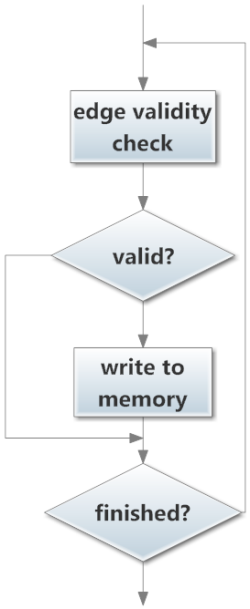
Εικόνα 4.5.2.1 Kernel όπου το κάθε thread αναλαμβάνει και από τα τρία είδη ακμών

Το κάθε thread εδώ δεν πραγματοποιεί ένα μόνο έλεγχο όπως στην προηγούμενη περίπτωση, εκτελεί σειριακά μία μικρή ομάδα ελέγχων και εγγραφών.



Εικόνα 4.5.2.2 Threads που εκτελούν πολλαπλές εγγραφές το καθένα

Η μνήμη που απαιτείται σε αυτή την αρχιτεκτονική είναι αρκετά μικρότερη καθώς δεν θεωρούμε ότι ο κάθε έλεγχος θα είναι θετικός και ότι υπάρχει αντίστοιχη μνήμη για την εγγραφή. Στην πραγματικότητα μόνο ένα πολύ μικρό ποσοστό των ελέγχων είναι θετικό οπότε στο κάθε thread αντιστοιχεί χώρος πολύ μικρότερος από τον αριθμό των ελέγχων που πραγματοποιεί. Για παράδειγμα μπορεί να εκτελέσει 100 ελέγχους όμως η μνήμη που του αντιστοιχεί να έχει μόνο 5 θέσεις για νέες ακμές. Αυτό μετρήθηκε πειραματικά με τα dataset που εξετάστηκαν.



Εικόνα 4.5.2.3 Εργασία που αναλαμβάνει ένα thread

Χρησιμοποιώντας λιγότερη μνήμη μειώνεται και ο όγκος των δεδομένων προς μεταφορά πίσω στη cpu. Έχουμε μειώσει την παραλληλία, όμως η απόδοση είναι καλύτερη διότι έχουμε αποφύγει το τεράστιο overhead που είχαμε προηγουμένως με το dynamic parallelism, καθώς επίσης μειώσαμε την απαιτούμενη μνήμη αλλά και τον όγκο δεδομένων προς μεταφορά.

4.5.3 Kernel με λιγότερη απαιτούμενη μνήμη

Βλέποντας βελτίωση με τις παραπάνω αλλαγές αναζητήσαμε κι άλλους τρόπους για να μειωθεί περαιτέρω η απαιτούμενη μνήμη. Η μέθοδος που ακολουθεί χρησιμοποιεί πολύ λιγότερα threads τα οποία εκτελούν μέρος του συνόλου των επαναλήψεων παράλληλα. Το κάθε ένα από αυτά αναλαμβάνει έναν έλεγχο και μία ενδεχόμενη εγγραφή. Άρα όπως και στην πρώτη υλοποίηση έχουμε ένα χώρο μνήμης όπου κάθε thread αντιστοιχεί σε μία θέση για μία νέα ακμή ανεξάρτητα από το αν θα γίνει η εγγραφή. Η διαφορά είναι ότι αυτός ο χώρος είναι πάρα πολύ μικρότερος και εξυπηρετεί μόνο ένα μικρό κομμάτι από το σύνολο των επαναλήψεων. Στο τέλος κάθε επανάληψης ένα thread αναλαμβάνει την αντιγραφή μόνο της χρήσιμης πληροφορίας σε ένα συμπαγή χώρο μνήμης χωρίς περιττά κενά. Στη συνέχεια διαγράφεται η μνήμη όπου πραγματοποιούν εγγραφές τα thread και συνεχίζουμε με την επόμενη επανάληψη. Η μνήμη αυτή λοιπόν επαναχρησιμοποιείται.

Για να πετύχουμε το παραπάνω είναι απαραίτητος ο συγχρονισμός μεταξύ των thread, ώστε όταν τελειώσουν με τις εγγραφές τα πολλά thread να αναλάβει ένα την αντιγραφή τους στη συμπαγή μνήμη. Η cuda υποστηρίζει συγχρονισμό μεταξύ των thread μόνο του ίδιου block οπότε με την συγκεκριμένη κάρτα είχαμε περιοριστεί στα 1024 threads και 1 block.

Με αυτό τον τρόπο έχουμε μειώσει σημαντικά την απαιτούμενη μνήμη αλλά δυστυχώς και την παραλληλία. Το αποτέλεσμα δεν ήταν το επιθυμητό καθώς η απόδοση ήταν περίπου 20 φορές χειρότερη.

Σε μία προσπάθεια αύξησης της παραλληλίας ενώ θα διατηρούσαμε σε χαμηλό επίπεδο την κατανάλωση μνήμης έπρεπε να υλοποιηθεί συγχρονισμός μεταξύ των block, αφού είχαμε εξαντλήσει το όριο του ενός το οποίο ήταν 1024 threads. Έγιναν δοκιμές με 8 block των 128 thread και 8 των 1024. Τα αποτελέσματα έδειξαν ότι όσο αυξάνεται η πολυπλοκότητα του μηχανισμού συγχρονισμού τόσο χειροτερεύει η απόδοση. Η υλοποίηση μηχανισμών συγχρονισμού μεταξύ των thread ή και η χρήση αυτών που ήδη υπάρχουν είναι απαγορευτική για την εφαρμογή μας στη συγκεκριμένη περίπτωση.

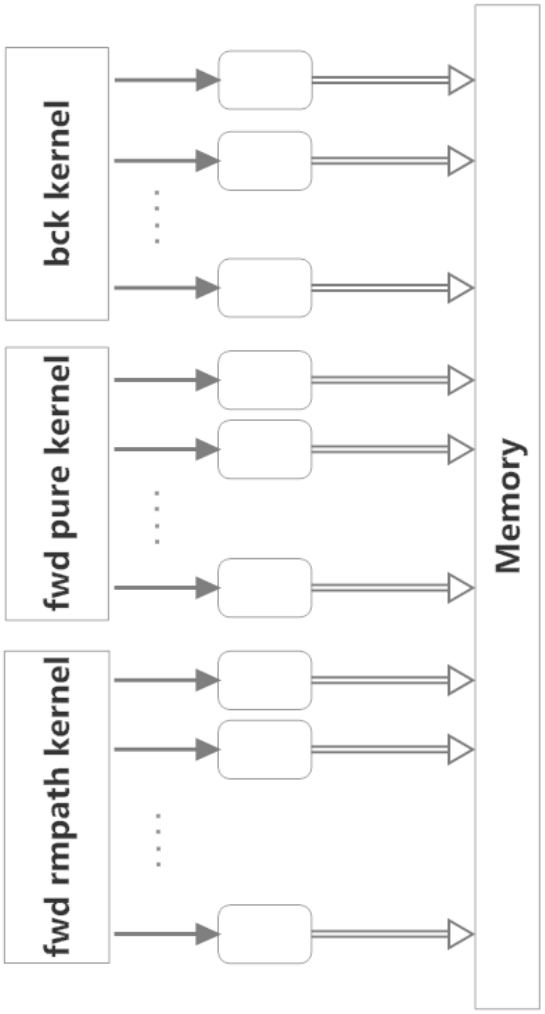
Dataset OVCAR size: 29MB graphs: 38437 support:2000	Χρόνος CPU	Χρόνος GPU	
		Grid,Block	Time (seconds)
	83s	1,1024	1245
		8,128	1343
		8,1024	1655

Σε μία προσπάθεια αποφυγής του συγχρονισμού των thread αλλά διατήρηση του χαμηλού όγκου δεδομένων προς μεταφορά έγινε το εξής. Χρησιμοποιήθηκε ένας kernel που αναλαμβάνει και τις backward και τις forward pure και τις forward rmpath ακμές και το κάθε thread του εκτελεί έναν έλεγχο και μία ενδεχομένως εγγραφή. Δεν κερδίζουμε κάτι σε μνήμη με αυτό τον τρόπο καθώς χρησιμοποιούμε το μέγιστο θεωρητικά χώρο που θα μπορούσε να χρησιμοποιηθεί. Η διαφορά σε αυτή την περίπτωση είναι ότι όταν το πρώτο thread τελειώσει με τον έλεγχο και την αντίστοιχη εγγραφή ξεκινάει να σαρώνει τη μνήμη όπου γράφουν όλα τα thread και αντιγράφει τη χρήσιμη πληροφορία σε ένα συμπαγή χώρο μνήμης. Αυτό γίνεται χωρίς τη χρήση κάποιου μηχανισμού συγχρονισμού και ενώ δεν είναι εγγυημένο ότι θα δουλέψει, πειραματικά βρέθηκε ότι το thread 0 από τη στιγμή που ξεκινάει την αντιγραφή καθυστερεί τόσο που όλα τα υπόλοιπα προλαβαίνουν τα τελειώσουν με πολύ μεγάλη πιθανότητα, καθώς δεν υπήρξε πρόβλημα σε καμία από τις δοκιμές.

Το αποτέλεσμα δεν ήταν διαφορετικό και σε αυτή την περίπτωση καθώς είχαμε μείωση της απόδοσης πάλι κατά 20 φορές περίπου. Το κύριο πρόβλημα λοιπόν με αυτές τις αρχιτεκτονικές, πέρα από το συγχρονισμό, ήταν ότι ένα thread είχε αναλάβει πολλή περισσότερη σειριακή “δουλειά”, πράγμα καθόλου αποδοτικό.

4.5.4 Τελική Αρχιτεκτονική με τρεις Kernel

Η τελική αρχιτεκτονική που καταλήξαμε παρουσιάζεται παρακάτω. Αποτελείται από τρεις kernel κάθε ένας από τους οποίους αναλαμβάνει ένα είδος ακμών από τις backward, forward pure και forward rmpath. Και οι τρεις έχουν παρόμοια λειτουργία. Χρησιμοποιούν τόσα threads όσο και ο αριθμός των pdfs που έχουμε μεταφέρει στη GPU. Έτσι το καθένα αναλαμβάνει την επεξεργασία ενός pdfs. Αυτό συνεπάγεται ότι θα αναλάβει πολλαπλούς ελέγχους και εγγραφές στη μνήμη. Με αυτό τον τρόπο κερδίζουμε σε μνήμη από το να είχαμε ένα thread ανά έλεγχο και δε χάνουμε σε απόδοση καθώς οι έλεγχοι αυτοί δεν είναι πολύπλοκοι υπολογιστικά ώστε να έχουμε σημαντική βελτίωση αν πραγματοποιηθούν όλοι παράλληλα.



Εικόνα 4.5.4 Αρχιτεκτονική με τρεις kernel

Έχοντας τον επεξεργαστή να ξεκινήσει τρεις kernel αυξάνουμε την παραλληλία σε σχέση με τις προηγούμενες υλοποιήσεις που αναλύσαμε και πετυχαίνουμε την καλύτερη απόδοση. Σε αυτό συμβάλει και η καλή αξιοποίηση της μνήμης αφού έχουμε μειώσει αρκετά τις μη χρησιμοποιούμενες θέσεις αναθέτοντας σε ένα thread πολλαπλούς ελέγχους.

Κεφάλαιο 5: Αποτελέσματα

Για όλες τις μετρήσεις χρησιμοποιήθηκε ένας επεξεργαστής intel core i7 2.7GHz και μία κάρτα γραφικών Nvidia GeForce 780 GTX. Όλοι οι χρόνοι που μετρήθηκαν αφορούν τη συνολική εκτέλεση του αλγόριθμου gspan, δηλαδή συμπεριλαμβάνουν και το χρόνο αντιγραφής του dataset στη μνήμη αλλά και την εμφάνιση των αποτελεσμάτων.

Dataset	**# Graphs**
Chemical	340
Compound	422
MCF-7	25.475
OVCAR	38436
SN12C	38048
NCI-H23	38295

Πίνακας 5.1 Datasets που χρησιμοποιήθηκαν

Τα dataset που χρησιμοποιήθηκαν για τις δοκιμές αναπαριστούν χημικές ενώσεις και εμφανίστηκαν και σε διάφορες άλλες εργασίες όπως αναφέραμε στο κεφάλαιο 2.

Μετρήθηκαν οι χρόνοι που χρειάζεται ο gspan για να ολοκληρώσει την εξόρυξη χρησιμοποιώντας μόνο τον επεξεργαστή (αρχική υλοποίηση) για διάφορα dataset και support και οι αντίστοιχοι χρόνοι χρησιμοποιώντας την υλοποίηση που περιγράφηκε παραπάνω αξιοποιώντας τη GPU. Τα αποτελέσματα ελέγχθηκαν συγκρίνοντας την έξοδο της αρχικής υλοποίησης του gspan με τη δική μας ώστε να βεβαιώσουμε τη σωστή λειτουργία του συστήματος. Ακολουθεί ένας πίνακας με τα αποτελέσματα.

Dataset	No Graphs	Support %	Χρόνος Εκτέλεσης Original Gspan (sec)	Χρόνος Εκτέλεσης Υλοποίησης Cuda (sec)	Speedup
Compound	422	30	0.465	1.520	0.31
		20	0.515	1.520	0.34
		10	0.615	1.672	0.37
		5	0.816	2.022	0.40
		4	1.066	2.372	0.45
		3	2.018	2.974	0.68
		2	5.325	6.584	0.81
		1	470.374	334.265	1.41
Chemical	340	30	0.415	1.369	0.30
		20	0.365	1.419	0.26
		10	1.067	1.470	0.73
		5	2.471	1.724	1.43
		4	3.624	2.023	1.79
		3	4.877	2.323	2.10
		2	17.814	5.482	3.25
		1	163.701	45.666	3.58
MCF-7	25475	30	134.075	37.100	3.61
		20	186.700	43.235	4.32
		10	355.130	67.452	5.26
		5	655.344	118.996	5.51
		4	818.902	146.630	5.58
		3	1082.621	197.057	5.49
		2	1701.807	313.134	5.43
		1	4617.237	1003.967	4.60

OVCAR	38436	30	211.225	54.784	3.86
		20	294.067	65.447	4.49
		10	539.836	94.903	5.69
		5	976.149	163.323	5.98
		4	1210.745	203.340	5.95
		3	1581.793	270.707	5.84
		2	2407.022	419.140	5.74
		1	7083.757	1502.971	4.71
SN12C	38048	30	197.049	54.552	3.61
		20	271.954	61.622	4.41
		10	503.920	92.227	5.46
		5	884.597	155.241	5.70
		4	1095.441	189.685	5.78
		3	1441.102	255.064	5.65
		2	2234.346	405.219	5.51
		1	7570.218	1643.087	4.61
NCI-H23	38295	30	209.992	54.545	3.85
		20	293.576	63.867	4.60
		10	537.155	93.944	5.72
		5	970.265	163.420	5.94
		4	1208.177	200.683	6.02
		3	1579.387	267.858	5.90
		2	2395.761	413.508	5.79
		1	6921.807	1454.064	4.76

Πίνακας 5.2 Αποτελέσματα original gspan – cuda gspan

Όπως παρατηρούμε στον παραπάνω πίνακα μεγαλύτερο speedup έχουμε για μικρότερες τιμές του support με μέγιστο να είναι 6x για το dataset NCI-H23 με support 4. Αυτό συμβαίνει διότι όσο μικραίνει το support τόσο περισσότερο αυξάνεται η παραλληλία. Μαζί με την παραλληλία όμως αυξάνεται και ο όγκος δεδομένων προς μεταφορά από και προς τη GPU για να εκτελεστεί η όλη διαδικασία. Γι' αυτό και για πάρα πολύ μικρές τιμές του support παρατηρούμε ότι το overhead για τις μεταφορές δεδομένων είναι τόσο που δεν ευνοεί τη συνολική απόδοση.

Κεφάλαιο 6: Συμπεράσματα

6.1 Γενικά

Σε αυτό το κεφάλαιο θα αναφερθούμε στη γνώση που προσκομίστηκε από αυτή την εργασία, τις τακτικές που ακολουθήθηκαν, προβλήματα που παρουσιάστηκαν και πιθανές λύσεις που προσπαθήσαμε να δώσουμε. Γενικά η δουλειά που αναθέσαμε στην κάρτα γραφικών ήταν κάποιοι έλεγχοι και εγγραφές στη μνήμη χωρίς καθόλου αριθμητικές πράξεις. Η εργασία αυτή περιλαμβάνει αρκετά branches πριν από μία ενδεχόμενη εγγραφή πράγμα το οποίο σημαίνει ότι μειώνουμε την παραλληλία διότι δεν μπορούν να εκτελούνται παράλληλα και τα 2 μέρη ενός branch μέσα σε ένα warp. Παρόλο που η GPU δεν είναι κατάλληλη για αυτή την εργασία όμως, όπως είδαμε στα αποτελέσματα, είχαμε αρκετό speedup σε κάποιες περιπτώσεις.

Θέματα με τα οποία ασχοληθήκαμε στην παρούσα εργασία, και καταλήξαμε σε κάποια συμπεράσματα για αυτά, είναι τα παρακάτω:

- Zero copy
- Overhead της cudamemcopy
- Όγκος δεδομένων που μεταφέρονται προς τη GPU
- Εκχώρηση μνήμης μία φορά και επαναχρησιμοποίησή της
- Χρήση page locked (pinned) memory
- CudaHostAllocWriteCombined
- Συγχρονισμός μεταξύ των thread
- Περισσότερη σειριακή εργασία από ένα thread

6.2 Zero Copy

Η Cuda παρέχει μία δυνατότητα που ονομάζεται zero copy. Όταν χρησιμοποιείται αυτή η τεχνική εκχωρούμε κάποιο χώρο μνήμης ως mapped memory. Αυτή η μνήμη τότε θα είναι διαθέσιμη και στον kernel που θα τρέξει στη GPU χωρίς περαιτέρω κλήσεις σε συναρτήσεις μεταφοράς δεδομένων. Στην πραγματικότητα αυτό που γίνεται είναι η cuda να μεταφέρει τα δεδομένα από αυτή τη μνήμη, που δηλώσαμε ως mapped, στη GPU όταν αυτά είναι απαραίτητα μόνο. Αυτό το κάνει με τέτοιο τρόπο ώστε να είναι δυνατό το overlapping μεταφοράς δεδομένων και εκτέλεσης εντολών. Δηλαδή προσπαθεί να “κρύψει” το χρόνο που απαιτείται για τη μεταφορά εκτελώντας παράλληλα εντολές. Μπορούμε να πούμε πως είναι μια μορφή ασύγχρονης μεταφοράς δεδομένων που όμως γίνεται αυτόματα από την cuda σε χρόνο που θεωρεί κατάλληλο.

Στην περίπτωση μας, ίσως επειδή τα thread δεν είχαν να εκτελέσουν πράξεις παρά μόνο συγκρίσεις, δεν υπήρχε καμία βελτίωση στην απόδοση. Μάλιστα όταν η αντιγραφή δεδομένων έγινε χειροκίνητα με κλήσεις στις συναρτήσεις μεταφοράς η απόδοση ήταν πολύ καλύτερη.

6.3 Overhead της cudamemcopy

Πολύ σημαντικό ρόλο από άποψη απόδοσης παίζει ο όγκος μεταφοράς δεδομένων από και προς τη GPU. Ο χρόνος που σπαταλά η συνάρτηση μεταφοράς δεδομένων cudamemcopy πέρα από αυτόν που απαιτείται για την πραγματική μεταφορά (overhead) είναι μη αμελητέος. Ιδιαίτερα όταν καλείται επαναληπτικά αυτή η συνάρτηση και μεταφέρει κομμάτι κομμάτι ένα μεγάλο όγκο δεδομένων τότε τα αποτελέσματα είναι απογοητευτικά. Στην αρχική υλοποίηση μας, καθώς ήταν πιο εύκολο, χρησιμοποιήθηκε αυτή η συνάρτηση πολλές φορές για να μεταφέρει λίγο λίγο τα δεδομένα. Αυτό είχε ως αποτέλεσμα ο χρόνος που απαιτείται για τη μεταφορά να είναι δύο τάξεις μεγαλύτερος απ'ότι θα μπορούσε να είναι αν χρησιμοποιούσαμε τον ελάχιστο αριθμό κλήσεων.

Το συμπέρασμα είναι ότι είναι πολύ προτιμότερο να “φορτώσουμε” τον επεξεργαστή περισσότερο και να χρησιμοποιήσουμε παραπάνω μνήμη ώστε να φέρουμε τα δεδομένα προς μεταφορά σε μία μορφή τέτοια που να μπορεί να πραγματοποιηθεί η μεταφορά τους μόνο με μία κλήση της συνάρτησης cudamemcopy. Αυτό όπως και στην περίπτωσή μας μπορεί να είναι αρκετά περίπλοκο αν εμπλέκονται δομές που χρησιμοποιούν pointers. Όμως αν τα φέρουμε σε συνεχόμενες θέσεις μνήμης με τους pointer κατάλληλα αρχικοποιημένους και μεταφέρουμε όλη αυτή τη μνήμη μόνο με μία κλήση τότε τα αποτελέσματα είναι βέλτιστα παρόλο που φαίνεται ότι έχουμε κάνει περισσότερη δουλειά για να δημιουργήσουμε αυτή την επιπλέον μνήμη στην κατάλληλη μορφή.

Στην εργασία μας αυτό έγινε κατά τη μεταφορά όλων των γράφων του αρχικού dataset εισόδου στη GPU. Φέραμε τα δεδομένα σε κατάλληλη μορφή και μετά πραγματοποιήθηκε η μεταφορά μόνο με μία κλήση της cudamemcopy. Έτσι ήταν κατά δύο τάξεις μεγέθους γρηγορότερη η μεταφορά στο σύνολό της.

6.4 Όγκος δεδομένων που μεταφέρονται προς τη GPU

Πέρα από το overhead της cudamemcopy, που όπως αναφέραμε παραπάνω δεν είναι αμελητέο, κυρίαρχη παράμετρος που επηρεάζει το χρόνο μεταφοράς είναι ο όγκος των δεδομένων προς αντιγραφή. Στην εργασία μας οι kernel καλούνται επαναληπτικά μέχρι να τελειώσει ο αλγόριθμος gspan την εξόρυξη. Σε κάθε επανάληψη πρέπει να μεταφερθούν δεδομένα προς τη GPU, να γίνει η επεξεργασία και να επιστραφούν τα αποτελέσματα πίσω στη συνέχεια. Ο όγκος των δεδομένων αυτών παίζει πολύ σημαντικό ρόλο καθώς μία πολύ μικρή αλλαγή στο μέγεθος μπορεί να έχει μεγάλη διαφορά στα αποτελέσματα, ειδικά αν οι επαναλήψεις αυτές είναι πολλές, όπως και στην περίπτωσή μας.

Τα δεδομένα “εισόδου” προς των kernel είναι τα pdfs. Στην αρχή είχαμε χρησιμοποιήσει στατικές δομές για αυτά. Πράγμα που σημαίνει ότι στα περισσότερα υπήρχε πολύς κενός αχρησιμοποίητος χώρος ο οποίος όμως λόγω της υλοποίησης μεταφερόταν άσκοπα προς τη GPU. Ως βελτίωση δημιουργούσαμε δυναμικά πριν από κάθε μεταφορά, με παραπάνω εργασία του επεξεργαστή, μία δομή με ακριβώς το μέγεθος που απαιτείται σε συνεχόμενες θέσεις μνήμης και με κατάλληλη αρχικοποίηση των pointer, όπως και με τη μεταφορά του dataset εισόδου. Σε μία μόνο μεταφορά το μέγεθος αυτό ήταν σχεδόν αμελητέο. Όμως στο σύνολο των μεταφορών είχε αρκετά μεγάλη βελτίωση στο χρόνο.

Συμπεραίνουμε λοιπόν πως αξίζει να αναθέσουμε στον επεξεργαστή πολύ περισσότερη δουλειά για να μειώσουμε τον όγκο μεταφοράς δεδομένων, όπως και των κλήσεων της cudamemcopy που αναφέραμε προηγουμένως, διότι το τελικό αποτέλεσμα είναι πολύ καλύτερο με αυτό τον τρόπο.

6.5 Εκχώρηση μνήμης μία φορά και επαναχρησιμοποίησή της

Αναφερθήκαμε σε δυναμική εκχώρηση μνήμης πριν από κάθε επανάληψη για την κλήση του kernel ώστε να δημιουργήσουμε ακριβώς το χώρο που απαιτείται για τα δεδομένα προς μεταφορά και να μειώσουμε έτσι τον όγκο μεταφοράς. Η εκχώρηση μνήμης και αυτή έχει overhead που δεν είναι αμελητέο όταν οι επαναλήψεις είναι πάρα πολλές. Και σε αυτή την περίπτωση λοιπόν προσπαθήσαμε να ελαχιστοποιήσουμε τον αριθμό κλήσεων προς τη συνάρτηση εκχώρησης.

Αυτό που κάναμε είναι να εκχωρούμε ένα αρκετά μεγάλο κομμάτι μνήμης στην αρχή το οποίο δεν ελευθερώναμε στη συνέχεια, αλλά το επαναχρησιμοποιούσαμε σε κάθε επανάληψη. Λογικό είναι αυτό το κομμάτι μνήμης να μην χρειάζεται ολόκληρο κάθε φορά και να χρησιμοποιείται μόνο ένα κομμάτι αυτού. Έχουμε δηλαδή περιττή δέσμευση μνήμης. Μεταφέρουμε όμως προς τη GPU μόνο το χρήσιμο κομμάτι, και η άχρηστη μνήμη δε μεταφέρεται απλά περιμένει να χρησιμοποιηθεί σε κάποια από τις επόμενες επαναλήψεις. Αποφεύγουμε με αυτό τον τρόπο να εκχωρούμε σε κάθε επανάληψη το χώρο που μας χρειάζεται, απλά χρησιμοποιούμε ένα κομμάτι από τον ήδη υπάρχων.

Αν η μνήμη μας επιτρέπει την παραπάνω τακτική και δεν έχουμε αυστηρούς περιορισμούς τότε είναι καλύτερο να έχουμε συνεχώς δεσμευμένο το μέγιστο κομμάτι μνήμης που θα χρειαστούμε ώστε να αποφύγουμε τις πολλές εκχωρήσεις μνήμης.

6.6 Χρήση page locked (pinned) memory

Μία ακόμη βελτίωση που εφαρμόσαμε όσο αφορά τις μεταφορές δεδομένων προς τη GPU είναι η χρήση page locked (pinned) μνήμης. Η non-locked μνήμη δεν βρίσκεται μόνο στη ram πχ μπορεί να είναι στο swap area στο δίσκο οπότε σε αυτή την περίπτωση ο driver πρέπει να διαβάσει όλες τις σελίδες της non-locked μνήμης, να τις αντιγράψει σε ένα pinned buffer και στη συνέχεια να τις περάσει στον DMA όπου γίνεται σύγχρονα page-by-page αντιγραφή. Έτσι οι αντιγραφές είναι πιο αργές και δεν χρησιμοποιούν το πλήρες bandwidth του PCI-e δίαυλου. Ακόμη και στη ram να βρίσκεται εξολοκλήρου η μνήμη, πάλι πρέπει να εκχωρήσει ένα χώρο με page-locked μνήμη, να αντιγράψει να δεδομένα προς μεταφορά εκεί, να πραγματοποιηθεί η μεταφορά και να ελευθερωθεί αυτή η page-locked μνήμη που χρησιμοποιήθηκε.

Όλο αυτό είναι overhead που μπορούμε να αποφύγουμε αν δηλώσουμε τη μνήμη μας ως page-locked. Το μειονέκτημα είναι ότι αν χρησιμοποιήσουμε πολλή τέτοια μνήμη θα υπάρχει πρόβλημα με το σύστημα γενικότερα καθώς αφού δεν θα μπορεί να την κάνει swap στο δίσκο.

6.7 CudaHostAllocWriteCombined

Υπάρχει περίπτωση να θέλουμε ένα κομμάτι μνήμης στο οποίο θα γράφει ο επεξεργαστής θα το μεταφέρει στη GPU όμως ποτέ ή πολύ σπάνια θα το διαβάσει. Στην περίπτωση αυτή μπορούμε να χρησιμοποιήσουμε το flag WriteCombined κατά την εκχώρηση μνήμης το οποίο σε κάποια συστήματα, αναφέρεται στο manual, ότι επιτρέπει γρηγορότερη μεταφορά προς τη GPU της συγκεκριμένης μνήμης, όμως διαβάζεται αργά από τον επεξεργαστή.

Ένα τέτοιο παράδειγμα στη δική μας εργασία είναι η περίπτωση του αρχικού dataset. Δημιουργούμε ένα χώρο μνήμης όπως περιγράφηκε παραπάνω, αντιγράφουμε εκεί το dataset σε κατάλληλη δομή ώστε να μπορεί να μεταφερθεί με ένα memcopy στη GPU και πραγματοποιούμε την αντιγραφή. Στη συνέχεια δε χρειάζεται ποτέ ο επεξεργαστής να διαβάσει αυτή τη μνήμη καθώς χρησιμοποιήθηκε μόνο για αυτή τη μεταφορά. Έτσι το μειονέκτημα της αργής ανάγνωσης δε μας απασχολεί.

Ένα ακόμη σημείο όπου χρησιμοποιείται η write combined μνήμη είναι στο χώρο που δημιουργούμε για να φτιάξουμε την κατάλληλη δομή με τα pdfs πριν τη μεταφορά τους στη GPU. Και εδώ μπορεί αυτός ο χώρος να γράφεται σε κάθε επανάληψη με νέα δεδομένα και να επαναχρησιμοποιείται, όμως ποτέ δε διαβάζεται από τον επεξεργαστή.

Παρόλο που ενδείκνυται η χρήση τέτοιου είδους μνήμης στις περιπτώσεις που αναφέραμε παραπάνω δεν παρατηρήθηκε βελτίωση στην απόδοση του συστήματος συνολικά.

6.8 Συγχρονισμός μεταξύ των thread

Η Cuda παρέχει τη δυνατότητα συγχρονισμού μεταξύ των thread που βρίσκονται στο ίδιο block, όχι όμως μεταξύ των blocks (μέχρι την έκδοση 5.5 η οποία χρησιμοποιήθηκε σε αυτή την εργασία). Υπάρχουν περιπτώσεις που αυτό είναι αρκετά χρήσιμο και θέλουμε να το χρησιμοποιήσουμε, όμως όταν η απόδοση και ο συνολικός χρόνος εκτέλεσης είναι κυρίαρχης σημασίας τότε δεν είναι καλή ιδέα.

Χρησιμοποιήσαμε αυτή τη δυνατότητα στον kernel ο οποίος χρησιμοποιεί ένα thread για να αντιγράψει εκείνες τις θέσεις στη μνήμη που περιέχουν ακμές σε ένα άλλο συμπαγή χώρο μνήμης χωρίς κενά. Αναφερθήκαμε σε αυτό στο κεφάλαιο 4.5.3. Για να υλοποιηθεί αυτό είναι απαραίτητο το thread αυτό να ξεκινήσει όταν όλα τα υπόλοιπα έχουν τελειώσει με τις εγγραφές τους. Εδώ είναι που χρησιμοποιούμε το μηχανισμό συγχρονισμού για να βεβαιωθούμε ότι τελείωσαν τα thread με τη δουλειά τους και τότε ξεκινάμε την αντιγραφή της χρήσιμης μνήμης.

Επίσης, σε μία προσπάθεια αύξησης της παραλληλίας υλοποιήσαμε ένα μηχανισμό συγχρονισμού μεταξύ των block ώστε να μπορούμε να χρησιμοποιήσουμε threads από πολλά blocks για τις εγγραφές να τα συγχρονίσουμε και μετά να αντιγράψουμε τη μνήμη όπως αναφέραμε.

Τα αποτελέσματα έδειξαν ότι το overhead του συγχρονισμού είναι αρκετά μεγάλο και δεν ενδείκνυται για εφαρμογές όπου ο χρόνος εκτέλεσης μας ενδιαφέρει. Πρέπει να χρησιμοποιείται μόνο αν δεν υπάρχει εναλλακτική λύση. Όσο περισσότερη δουλειά έπρεπε να γίνει για το συγχρονισμό των thread στην εργασία μας τόσο χειροτέρευε η απόδοση του συστήματος.

6.9 Περισσότερη σειριακή εργασία από ένα thread

Υπάρχουν περιπτώσεις όπου ένα μόνο thread από το σύνολο των ενεργών νημάτων έχει περισσότερη εργασία να διεκπεραιώσει σε σχέση με τα υπόλοιπα. Αν αυτή η εργασία είναι κατά πολύ μεγαλύτερη από των άλλων thread και ο χρόνος που απαιτείται για την ολοκλήρωσή της δεν είναι συγκρίσιμος με το χρόνο των άλλων εργασιών, αλλά αρκετά μεγαλύτερος τότε τα αποτελέσματα μπορεί να είναι καταστροφικά για την απόδοση. Αυτό συμβαίνει διότι θα δουλεύει μόνο ένα thread ενώ τα υπόλοιπα θα περιμένουν να ολοκληρώσει αυτό το ένα την εργασία του. Κατά συνέπεια θα έχουμε πολύ κακό utilization της GPU στο σύνολό της.

Όπως αναφέρθηκε παραπάνω σε μία από τις υλοποιήσεις του τελικού συστήματος χρησιμοποιήσαμε ένα thread το όποιο θα αντιγράψει τη χρήσιμη μνήμη με τις ακμές σε ένα συμπαγή χώρο. Όταν αυτό το thread πραγματοποιεί την αντιγραφή όλα τα υπόλοιπα περιμένουν να ολοκληρώσει, ώστε στη συνέχεια να συνεχίσουν με την επόμενη επανάληψη εγγραφών την οποία θα ακολουθήσει πάλι αντιγραφεί από ένα thread. Βλέπουμε λοιπόν ότι ένα υπολογίσιμο κομμάτι του χρόνου εργάζεται μόνο ένα thread που πραγματοποιεί μεταφορές δεδομένων στη μνήμη.

Η χρήση αυτής της τεχνικής είχε ως αποτέλεσμα τη χειροτέρευση του συστήματός μας όσο αφορά το χρόνο εκτέλεσης κατά 20 φορές περίπου.

6.10 Τελικά Συμπεράσματα

Συνοψίζοντας, θα αναφέρουμε όλα εκείνα τα στοιχεία που συνέβαλαν στην βελτίωση της απόδοσης της εργασίας μας.

- Το Zero Copy δεν πρέπει να χρησιμοποιείται “τυφλά” αλλά να συγκριθεί με την απόδοση της “χειροκίνητης” αντιγραφής δεδομένων
- Το overhead της cudamemcopy δεν είναι αμελητέο και μπορεί, συσσωρευτικά, να δημιουργήσει μεγάλη διαφορά στο χρόνο εκτέλεσης
- Μία μικρή αλλαγή στον όγκο των δεδομένων που μεταφέρονται (CPU - GPU) μπορεί να έχει αρκετά μεγάλο αντίκτυπο στην απόδοση, όταν οι μεταφορές εκτελούνται επαναληπτικά
- Η εκχώρηση μνήμης και αυτή δεν έχει αμελητέο overhead και όπου είναι δυνατό πρέπει να χρησιμοποιείται ο ελάχιστος αριθμός κλήσεων στη συνάρτηση εκχώρησης μνήμης
- Η Χρήση page-locked μνήμης μπορεί να αυξήσει την απόδοση στις μεταφορές δεδομένων προς τη GPU.
- Ανάλογα το σύστημα η χρήση writecombined μνήμης μπορεί επίσης να αυξήσει την αποδοτικότητα της συνάρτησης μεταφοράς μνήμης. Στο δικό μας δεν υπήρχε κάποια διαφορά στην απόδοση.
- Ο συγχρονισμός μεταξύ των thread δεν αποτελεί καλή επιλογή όταν το σύστημά μας έχει ως στόχο τη μείωση του χρόνου εκτέλεσης του.
- Όλα τα thread πρέπει να έχουν τον ίδιο ή συγκρίσιμο φόρτο εργασίας διαφορετικά μπορεί να επηρεαστεί αρνητικά η απόδοση του συστήματος.

ΒΙΒΛΙΟΓΡΑΦΙΑ

[1] Thusoo, A., Shao, Z., Anthony, S., Borthakur, D., Jain, N., Sarma, J.S., Murthy, R., and Liu, H. 2010. “Data warehousing and analytics infrastructure at facebook”. In Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. Pages 1013-1020.

[2] Aggarwal, C. and Wang, H. 2010. “Managing and Mining Graph Data”. Springer, Advances in Database Systems, Volume 40.

[3] Vanetik, N., Gudes, E. and Shimony, S.E. 2002. “Computing frequent graph patterns from semistructured data”. In Proceedings of the IEEE International Conference on Data Mining (ICDM 2002). Pages 458–465.

[4] Inokuchi, A., Washio, T. and Motoda, H. 2000. “An Apriori-Based Algorithm for Mining Frequent Substructures from Graph Data”. Pages 13-23.

[5] Kuramoshi, M. and Karypis, G. 2001. “Frequent Subgraph Discovery”. In Proceedings of the IEEE International Conference on Data Mining (ICDM 2001). Pages 313-320.

[6] Bringmann, B. and Nijssen, S. 2008. “What is frequent in a single graph?”. In Proceedings of the 12th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD 2008). Volume 5012. Pages 858-863.

[7] Fiedler, M. and Borgelt, C. 2007. “Support Computation for Mining Frequent Subgraphs in a Single Graph”. Workshop on Mining and Learning with Graphs (MLG’07).

[8] Lee, M., Yang, L. H., Hsu, W. and Yang, X. 2002. “XClust: Clustering XML Schemas for Effective Integration”. In Proceedings of the 11th International Conference on Information and Knowledge Management (CIKM 2002). Pages 292-299.

[9] Yan, X. and Han, J. 2002. “gSpan: Graph-Based Substructure Pattern Mining”. In Proceedings of the IEEE International Conference on Data Mining (ICDM 2002). Pages 721-724.

[10] Lakshmi, K. and Dr. Meyyappan, T. 2012. “FREQUENT SUBGRAPH MINING ALGORITHMS – A SURVEY AND FRAMEWORK FOR CLASSIFICATION”. Advances in the International Journal of Information Technology Convergence and Services. Volume 2, No.2. Pages 23-39.

[11] Gholami, M. and Salajegheh, A. 2012. “A Survey on Algorithms of Mining Frequent Subgraphs”. In International Journal of Engineering Inventions. Volume 1, Issue 5. Pages 60-63.

[12] Matsuda, T., Horiuchi, T., Motoda, H. and Washio, T. 2000. “Extension of graph-based induction for general graph structured data”. In Proceedings of the 4th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD 2000). Volume 1805. Pages 420–431.

[13] Inokuchi, A., Washio, T., and Motoda, H. 2003. “Complete mining of frequent patterns from graphs: Mining graph data”. In Machine Learning, Volume 50, Issue 3. Pages 321–354.

[14] Huan, J., Wang, W., and Prins, J. 2003. “Efficient mining of frequent subgraphs in the presence of isomorphism”. In Proceedings of the 3rd IEEE International Conference on Data Mining (ICDM 2003). Pages 549-552.

[15] Huan, J., Wang W., Prins, J. and Yang, J. 2004. “SPIN: Mining Maximal Frequent Subgraphs from Graph Databases”. In Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining (KDD’04). Pages 581-586.

[16] Borgelt, C. and Berthold, M. 2002. “Mining Molecular Fragments: Finding Relevant Substructures of Molecules”. In Proceedings of the 3rd IEEE International Conference on Data Mining (ICDM 2003).

Pages 51-58.

[17] Nijssen, S. and Kok, J. 2004. “A quickstart in frequent structure mining can make a difference”. In Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining (KDD’04). Pages 647-652.

[18] Muggleton, S. 1996. “Stochastic logic programs”. Advances in Inductive Logic Programming. Pages 254-264.

[19] Nijssen, S. and Kok, J. 2001. “Faster association rules for multiple relations”. In Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI’01). Volume 2. Pages 891–896.

[20] Kethar, N., Holder, L. and Cook, D. 2005. “Subdue: Compression-Based Frequent Pattern Discovery in Graph Data”. In Proceedings of the 1st international workshop on open source data mining: frequent pattern mining implementations (OSDM’05). Pages 71-76.

[21] Rissanen, J.1999. “Hypothesis selection and testing by the MDL principle”. Advances in the Computer Journal. Volume 42, Issue 4. Pages 260–269.

[22] Wörlein, M., Meinl, T., Fischer, I. and Philippsen, M. 2005. “A Quantitative Comparison of the Subgraph Miners MoFa, gSpan, FFSM, and Gaston”. In Proceedings of the 9th European Conference on Principles and Practice of Knowledge Discovery in Databases. Volume 3721. Pages 392-403.

[23] Chakrabarti, D. and Faloutsos C. 2006. “Graph Mining: Laws, Generators and Algorithms”. In Journal ACM Computing Surveys (CSUR).Volume 38, Issue 1. Article No. 2.

[24] Di Fatta, G. and Berthold, M. 2005. “High Performance Subgraph Mining in Molecular Compounds”. In Proceedings of the 1st International Conference on High Performance Computing and Communications (HPCC 2005). Volume 3726. Pages 866-877.

[25] Buehrer, G. and Parthasarathy, S. 2005. “Parallel Graph Mining on Shared Memory Architectures”. Technical report, Ohio State University, Columbus, OH.

[26] Meinl, T., Wörlein, M., Fischer, I. and Philippsen, M. 2006. “Mining Molecular Datasets on Symetric Multiprocessor Systems”. In Proceedings of the IEEE International Conference on Systems, Man and Cybernetics (SMC 2006). Pages 1269-1274.

[27] Meinl, T., Fischer, I. and Philippsen, M. 2005. “Parallel Mining for Frequent Fragments on a Shared-Memory Multiprocessor – Results and Java-Obstacles-”. In Workshopwoche Lernen, Wissensentdeckung, AdaptivitÄat (LWA'05). Pages 196-201.

[28] Reinhardt, S. and Karypis, G. 2007. “A Multi-Level Parallel Implementation of a Program for Finding Frequent Patterns in a Large Sparse Graph”. In Proceedings of the IEEE International Conference on Parallel and Distributed Processing Symbosium (IPDPS 2007). Pages 1-8.

[29] Ranu, S. and Singh, A. 2009. “GraphSig: A Scalable Approach to Mining Significant Subgraphs in Large Graph Databases”. In Proceedings of the 25th IEEE International Conference on Data Engineering (ICDE’09). Pages 844-855.

[30] Wu, B. and Bai, Y. 2010. “An Efficient Distributed Subgraph Mining Algorithm in Extreme Large Graphs”. In Proceedings of the International Conference on Artificial Intelligence and Computational Intelligence (AICI 2010).Volume 6319. Pages 107-115.

[31] Hill, S., Srichandan, B. and Sunderraman, R. 2012. “An iterative mapreduce approach to frequent subgraph mining in biological datasets”. In Proceeding of the ACM Conference on Bioinformatics, Computational Biology and Biomedicine (BCB’12). Pages 661-666.

[32] Ray, A. and Holder, L.B.. 2012. “Efficiency Improvements for Parallel Subgraph Miners”. In the 25th Florida Artificial Intelligence Research Society Conference (FLAIRS 2012). Pages 74-79.

[33] Bhuiyan, M. and Al Hasan, M. 2013. “MIRAGE: An Iterative MapReduce based Frequent Subgraph Mining Algorithm”.

[34] Harish, P. and Narayanan, P.J. 2007. “Accelerating Large Graph Algorithms on the GPU using CUDA”. In Proceedings of the 14th International Conference on High Performance Computing (HiPC 2007). Volume 4873. Pages 197-208.

[35] Hong, S., Oguntebi T. and Olukotun K. 2011. “Efficient Parallel Graph Exploration on Multi-Core CPU and GPU”. In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT 2011). Pages 78-88.

[36] Merrill, D., Garland, M. and Grimshaw, A. 2012. “Scalable GPU Graph Traversal”. In Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practise of Parallel Programming (PPOPP’12).Volume 47, Issue 8. Pages 117-128.

[37] Wang, W., Dong, J. and Yuan B. 2013. “Graph-Based Substructure Pattern Mining Using CUDA Dynamic Parallelism”.

[38] Ichikawa, S., Saito, H., Udorn, L. and Konishi, K. 2000. “Evaluation of Accelerator Designs for Subgraph Isomorphism Problem”. In Proceedings of 10th International Conference on Field-Programmable Logic and Applications: The Roadmap to Reconfigurable Computing (FPL 2000). Volume 1896. Pages 729-738.

[39] Bondhugula, U., Devulapalli, A., Fernando, J., Wyckoff, P. and Sadayappan, P. 2006. “Parallel FPGA-based All-Pairs Shortest-Paths in a Directed Graph”. In Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS 2006).

[40] Thomas, D., Luk, W. and Stumpl, M. 2007. “Reconfigurable Hardware Acceleration of Canonical Graph Labelling”. In Proceedings of the 3rd International Workshop in Reconfigurable Computing: Architectures, Tools and Applications (ARC 2007).

[41] Betkaoui, B., Thomas, D., Luk, W. and Przulj, N. 2011. “A Framework for FPGA Acceleration of Large Graph Problems: Graphlet Counting Case Study”. In Proceedings of the International Conference on Field-Programmable Technology (FPT 2011).

[42] Betkaoui, B., Wang,Y., Thomas, D. and Luk, W. 2012. “A Reconfigurable Computing Approach for Efficient and Scalable Parallel Graph Exploration”. In Proceedings of the IEEE 23rd International Conference on Application-Specific Systems, Architectures and Processors (ASAP 2012).

[43] Kobori, D. and Maruyama, T. 2012. “An Acceleration of a Graph Cut Segmentation With FPGA”. In Proceedings of the 22nd International Conference on Field-Programmable Logic and Applications (FPL 2012). Pages 407-413.