Ατομική Διπλωματική Εργασία

Υλοποίηση Ερωτημάτων Βάσεων Δεδομένων στην Πλατφόρμα Ροής Δεδομένων της Maxeler

Βαρνάβας Παπαϊωάννου

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Μάιος 2016

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Υλοποίηση Ερωτημάτων Βάσεων Δεδομένων
στην Πλατφόρμα Ροής Δεδομένων της Maxeler

Βαρνάβας Παπαϊωάννου

Επιβλέπων Καθηγητής
Pedro Trancoso

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

Μάιος 2016

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή μου, κ. Pedro Trancoso, για την συνεχή στήριξη και τις συμβουλές του, στοιχεία τα οποία διαδραμάτισαν καθοριστικό ρόλο στην επιτυχή ολοκλήρωση της διπλωματικής εργασίας.

Θα ήθελα επίσης να ευχαριστήσω τον διδακτορικό φοιτητή Ανδρέα Διαβαστό, ο οποίος ήταν πάντα διαθέσιμος να βοηθήσει καθ' όλη την διάρκεια της διεκπεραίωσης της εργασίας.

Περίληψη

Στην παρούσα εργασία μελετάται το κατά πόσο θα μπορούσαμε να προσφύγουμε σε εναλλακτικές μορφές λύσεων, δηλαδή λύσεις οι οποίες δεν στηρίζονται στο παραδοσιακό control flow για μια ορισμένη κατηγορία προβλημάτων.
Συγκεκριμένα, μελετάται η χρήση της πλατφόρμας της Maxeler και το κατά πόσο θα μπορούσαμε να δούμε βελτίωση έναντι των συμβατικών λύσεων πάνω σε προβλήματα του τομέα των βάσεων δεδομένων.

Περιεχόμενα

Κατάλογος Σχημάτων

Κατάλογος Πινάκων

Κεφάλαιο 1

Εισαγωγή

1.1 Κίνητρο Εκπόνησης της Διπλωματικής Εργασίας

Είναι γεγονός ότι η διαθέσιμη επεξεργαστική ισχύς των συστημάτων της σημερινής εποχής είναι ο κύριος παράγοντας που περιορίζει τα προβλήματα που μπορούν να τύχουν στόχευσης, την ακρίβεια με την οποία μπορεί κανείς να τα προσεγγίσει και συνεπώς την αξιοπιστία των αποτελεσμάτων που εξάγονται.

Με την πάροδο του χρόνου, καθώς επίσης και με την συνεχή ανάγκη για επίλυση πιο περίπλοκων προβλημάτων με μεγαλύτερη ακρίβεια θεωρητικών μοντέλων, καθίσταται αναγκαία η εύρεση νέων υπολογιστικών τεχνολογιών που θα αντικαταστήσουν τις υπάρχουσες.

Τα τελευταία χρόνια το γεγονός αυτό παραβλεπόταν λόγω της συνεχούς αύξησης των συχνοτήτων των επεξεργαστών. Το φαινόμενο αυτό πλέον έχει εκλείψει, λόγω των προβλημάτων διάχυσης της θερμότητας που προέκυψαν από την αυξανόμενη πυκνότητα τρανζίστορ ανά μονάδα χώρου.

Επομένως στις μέρες μας, η ανάγκη αυτή γίνεται όλο και πιο επιτακτική.

1.2 Σκοπός και Αναμενόμενα Αποτελέσματα

Σκοπός της παρούσας διπλωματικής εργασίας είναι να αναδείξει μια σχετικά νέα τεχνολογία όσον αφορά την χρήση της, που φαίνεται να είναι υποσχόμενη σε μια ευρεία γκάμα προβλημάτων. Ο λόγος γίνεται για τα συστήματα ροής δεδομένων.

Μέσω της παρούσας εργασίας θα γίνει προσπάθεια να μελετηθεί το κατά πόσο μπορούμε να ενσωματώσουμε αυτήν την τεχνολογία στην επίλυση προβλημάτων που σχετίζονται με τις βάσεις δεδομένων, καθώς επίσης να μελετήσουμε το κατά πόσο είναι δυνατό να επιτευχθεί αύξηση της επίδοσης σε σχέση με τους χρόνους της CPU. Τα προβλήματα που θα προσεγγιστούν στηρίζονται πάνω στην σουίτα ερωτημάτων για αξιολόγηση συστημάτων, TPC-H.

Αναμένεται ότι θα παρατηρηθεί αύξηση της επίδοσης σε κάποιο βαθμό. Αυτό γιατί ο τύπος των προβλημάτων προσφέρει μεγάλη δυνατότητα παραλληλισμού, πράγμα το οποίο μπορεί να εκμεταλλευτεί αποτελεσματικά το μοντέλο που μελετάμε. Το γεγονός, όμως, ότι η πολυπλοκότητα των επιμέρους κομματιών των προβλημάτων είναι σχετικά μικρή, αναμένεται να μετριάσει τα αποτελέσματα.

1.3 Οργάνωση και Μεθοδολογία

Για την επιτυχή επίτευξη της παρούσας διπλωματικής εργασίας έγινε εξαρχής μια ανασκόπηση της διαδικασίας που έπρεπε να ακολουθηθεί και έπειτα σπάσιμο αυτής σε κομμάτια έτσι ώστε γίνει πιο ουσιαστική προσέγγιση.
Αρχικά, λόγω του ότι η πλατφόρμα προγραμματισμού της Maxeler ήταν άγνωστη, έγινε μελέτη της βιβλιογραφίας για τον τρόπο προγραμματισμού, έτσι ώστε να τεθούν οι βάσεις μέσα από το θεωρητικό κομμάτι. Στην συνέχεια, ακολουθήθηκε πρακτική προσέγγιση, με την υλοποίηση διαφόρων μικρών προγραμμάτων για να επιτευχθεί μια πιο ολοκληρωμένη εξοικείωση με το περιβάλλον. Έπειτα, έγινε μελέτη των προβλημάτων, των οποίων θα γινόταν προσπάθεια μεταφοράς στο περιβάλλον της Maxeler, και αναγνώριση των επιμέρους κομματιών τους τα οποία θα μεταφέρονταν στην Μηχανή Ροής Δεδομένων. Ακολούθως, έγινε υλοποίηση των κομματιών που αναγνωρίστηκαν στο προηγούμενο βήμα στην μηχανή ροής δεδομένων και χρησιμοποιώντας τον προσομοιωτή επιβεβαιώθηκε η ορθότητα τους. Τέλος, έγινε υπολογισμός και καταγραφή των επιδόσεων των προγραμμάτων, σύγκριση μεταξύ τους και εξαγωγή των σχετικών συμπερασμάτων.

1.4 Δομή Εργασίας

Η παρούσα διπλωματική εργασία χωρίζεται σε έξι κεφάλαια και είναι δομημένα αρχικά με μια εισαγωγή στο θέμα, έτσι ώστε μετέπειτα να μπορεί να γίνει επιτυχώς η επεξήγηση της προσέγγισης που ακολουθήθηκε, τα αποτελέσματα, όπως και τα συμπεράσματα.

Στο πρώτο Κεφάλαιο, γίνεται μια σύντομη εισαγωγή στο θέμα της εργασίας.

Στη συνέχεια, στο Κεφάλαιο 2, γίνεται περιγραφή του τρόπου λειτουργίας της πλατφόρμας της Maxeler και έπειτα δίδονται παραδείγματα προγραμμάτων υλοποιημένα στο γραφικό περιβάλλον υλοποίησης προγραμμάτων της Maxeler, τον MaxCompiler.

Στο Κεφάλαιο 3 αναλύονται εργασίες που έγιναν στον παρελθόν που στηρίζονται στην πλατφόρμα της Maxeler, έτσι ώστε να δοθεί μια ιδέα για τις περιπτώσεις στις οποίες χρησιμοποιείται η πλατφόρμα.

Στο Κεφάλαιο 4 γίνεται ανάλυση της πλατφόρμας ερωτημάτων TPC-H και μελετώνται τόσο οι υλοποιήσεις CPU, όσο και οι αντίστοιχες υλοποιήσεις στην πλατφόρμα της Maxeler.

Έπειτα, στο Κεφάλαιο 5 εξάγονται οι χρόνοι των δύο υλοποιήσεων για το κάθε ερώτημα και γίνεται σύγκριση των αποτελεσμάτων.

Στο Κεφάλαιο 6, που είναι και το τελευταίο, γίνεται μια γενική ανασκόπηση των αποτελεσμάτων, ενώ εξάγονται τα τελικά συμπεράσματα και γίνεται αναφορά για πιθανή μελλοντική δουλειά.

Κεφάλαιο 2

Maxeler Dataflow Computing

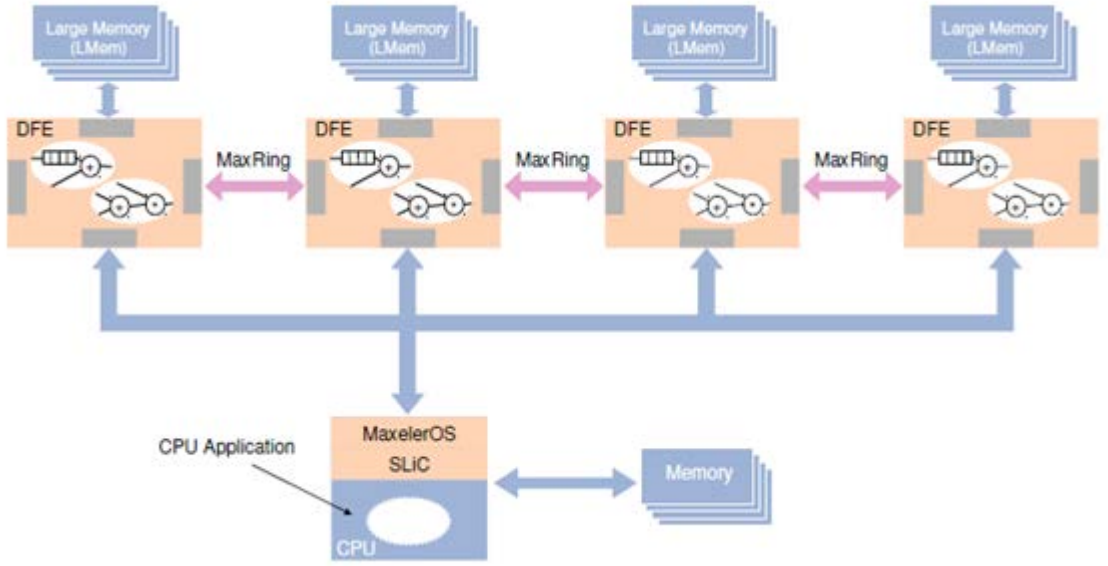
2.1 Συστήματα ροής δεδομένων

Στα συστήματα ροής δεδομένων μπορούμε να πούμε ότι ένα πρόγραμμα είναι ένας γράφος ροής δεδομένων για τις εκτελούμενες ενέργειες. Με το που γίνονται διαθέσιμα τα δεδομένα από τους κόμβους εισόδου, εκτελείται η ενέργεια και το αποτέλεσμά της προωθείται στον επόμενο κόμβο του γράφου. Η διαδικασία συνεχίζεται μέχρι να εκτελεστεί ο τελευταίος κόμβος/ενέργεια για όλα τα δεδομένα.

2.2 Εισαγωγή στην πλατφόρμα της Maxeler

Η πλατφόρμα της Maxeler καινοτομεί στο ότι συνδυάζει τις αρχιτεκτονικές ελέγχου ροής, CPUs, και ροής δεδομένων, Dataflow Engines. Σκοπός αυτού είναι με την χρήση των δυνατοτήτων της κάθε αρχιτεκτονικής να περιοριστούν στο ελάχιστο οι αδυναμίες της άλλης αρχιτεκτονικής, προσφέροντας έτσι ένα καλό πλαίσιο για παραλληλισμό.

Η γενική μορφή των προγραμμάτων έχει μια διεργασία να τρέχει στο CPU, ενώ σε διάφορα σημεία της καλεί διάφορες λειτουργίες για εκτέλεση σε μία ή και περισσότερες μηχανές ροής δεδομένων, οι οποίες είναι διασυνδεδεμένες μεταξύ τους. Ο υβριδικός της χαρακτήρας, παρότι μπορεί να προσφέρει αρκετές δυνατότητες βελτίωσης, μπορεί να καταστήσει πρόκληση τόσο την εύρεση, όσο και την υλοποίηση των κομματιών που θα εκτελεστούν στην μηχανή ροής δεδομένων.



Σχήμα 2.1: Αρχιτεκτονική του συστήματος ροής δεδομένων της Maxeler [1]

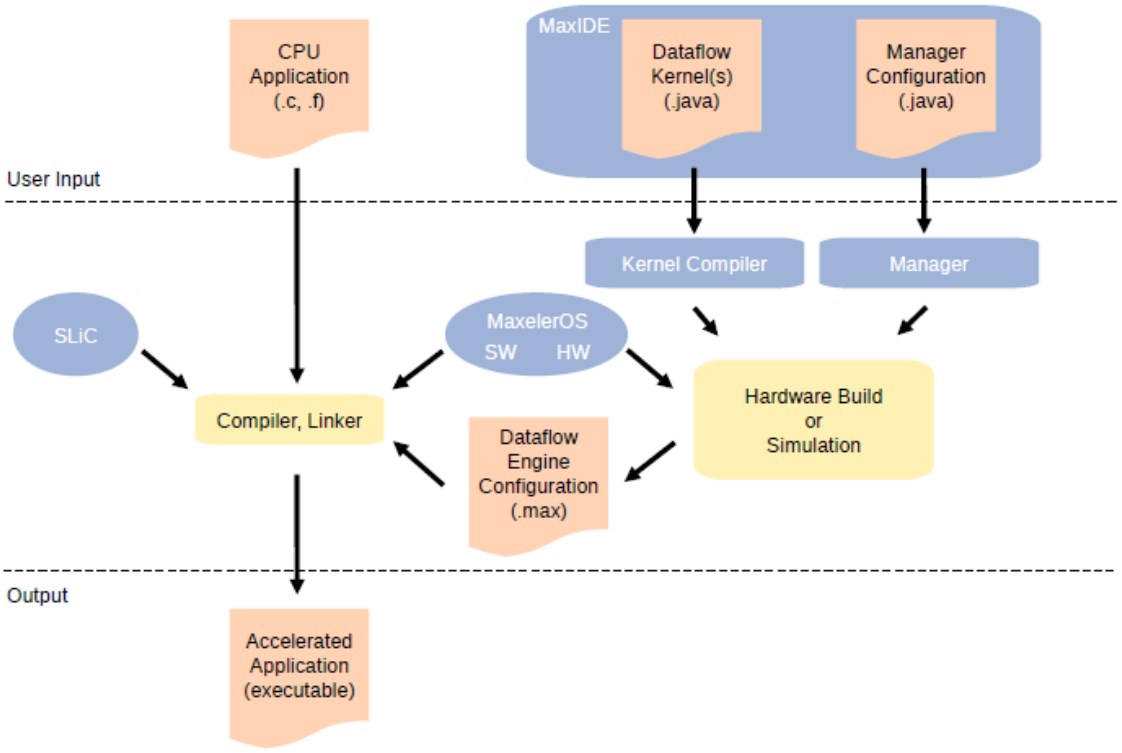
2.3 Προγραμματισμός στο Maxeler MaxCompiler

Ο προγραμματισμός στην πλατφόρμα της Maxeler αποτελείται από τρία μέρη. Το πρόγραμμα, το οποίο θα εκτελεστεί πάνω σε ένα σύνολο CPU, έπειτα από την περιγραφή των ροών από και προς τον πυρήνα και το CPU, και τρίτον, τον προγραμματισμό του πυρήνα. Η ορολογία για τα δύο τελευταία μέρη είναι Manager και Kernel αντίστοιχα. Επίσης, για τα δύο τελευταία μέρη προσφέρεται το εργαλείο MaxCompiler, το οποίο παρέχει ένα γραφικό περιβάλλον για την διευκόλυνση του προγραμματισμού των συστημάτων. Για το κομμάτι το οποίο τρέχει στην CPU μπορεί να χρησιμοποιηθούν C/C++, Python, Matlab και R. Για τα άλλα δύο μέρη χρησιμοποιείται μια επέκταση της Java, η MaxJ, και μέσω αυτής επιτυγχάνεται ολοκληρωμένα ο προγραμματισμός των μηχανών ροής.

Η διαδικασία δημιουργίας εφαρμογής για την πλατφόρμα της Maxeler ξεκινά με την ανάλυση του Manager και του Kernel. Βάσει αυτών δημιουργείται ένας γράφος ο οποίος περιγράφει την ροή των δεδομένων στην μηχανή ροής. Ο γράφος αυτός αποθηκεύεται σε ένα αρχείο με κατάληξη .max. Αφού δημιουργηθεί το αρχείο .max, το CPU είναι υπεύθυνο για την φόρτωση του/των .max στην μηχανή ροής δεδομένων. Η επικοινωνία από και προς τη μηχανή ροής γίνεται με την χρήση της διεπαφής SLiC, Simple Live CPU. Η διεπαφή αυτή αποτελείται από απλές εντολές χειρισμού της μηχανής, τις οποίες αναλαμβάνει να εκτελέσει στην μηχανή το λειτουργικό της Maxeler.

Ο ιδανικό τρόπος χρήσης είναι για το πρόγραμμα να τρέξει κανονικά στην CPU και για το σύνολο των εντολών, το οποίο πρέπει να τρέξει πάνω σε μεγάλο αριθμό δεδομένων, να χρησιμοποιηθεί η μηχανή ροής δεδομένων για την επιτάχυνσή του.

Καθοριστικό ρόλο στην επίδοση παίζει η σωστή αξιοποίηση τόσο της εσωτερικής (FMem), όσο και της εξωτερικής μνήμης (LMem) της μηχανής, οι οποίες χαρακτηρίζονται από το σχετικά μεγάλο εύρος που παρέχουν.



Σχήμα 2.2: Αλληλεπίδραση των διάφορων κομματιών [1]

2.4 Παραδείγματα Εφαρμογών

Πιο κάτω δίδονται κάποια απλά παραδείγματα εφαρμογών που επιλύουν συγκεκριμένα προβλήματα, τα οποία αντιμετωπίστηκαν για την ολοκλήρωση της παρούσας διπλωματικής εργασίας.

Μετατροπή βρόγχου:

CPU Code:

```
for (int i = 0; i < size; ++i)
        result[i] = input[i] + 1;
```

Maxeler Code:

CPU Code:

```
Increment(size, input, output);
```

Στο CPU γίνεται απλώς το κάλεσμα της συνάρτησης για την εκτέλεση της λειτουργίας στην μηχανή ροής δεδομένων.

Manager Code:

```
manager.setIO(link("input", CPU),
              link("result", CPU));
m.createSLiCinterface();
```

Ο Manager συνδέει τις ροές input και result με το CPU. Η ρύθμιση της εκτέλεσης του πυρήνα γίνεται αυτόματα από την μηχανή ροής δεδομένων λόγω της απλότητάς του.

Kernel Code:

```
DFEVar input = io.input("input", dfeUInt(32));
DFEVar result = input + 1;
io.output("result", result, dfeUInt(32));
```

Στον πυρήνα ρυθμίζεται η ροή που θα έχουν τα δεδομένα, έτσι ώστε στο τέλος να παράγεται το επιθυμητό αποτέλεσμα.

Μετατροπή βρόγχου με επεξεργασία 10 στοιχείων ανά κύκλο:

CPU Code:

```
for (int i = 0; i < size; i++)
        result[i] = input[i] + 1;
```

Maxeler Code:

CPU Code:

```
Increment(size, input, output);
```

Manager Code:

```
manager.setIO(link("input", CPU),
              link("result", CPU));
m.createSLiCinterface();
```

Kernel Code:

```
int plevel=10;
DFEVectorType<DFEVar> mtype = new DFEVectorType<DFEVar>
(dfeUInt(32), plevel);

DFEVar input = io.input("input", mtype);
DFEVector<DFEVar> result=mtype.newInstance(this);
for(int i=0;i<plevel;i++){
    result[i] <== input + 1;
}
io.output("result", result, result.getType());
```

Όπως παρατηρείται, το κύριο κομμάτι που διαφοροποιείται είναι αυτό του πυρήνα.

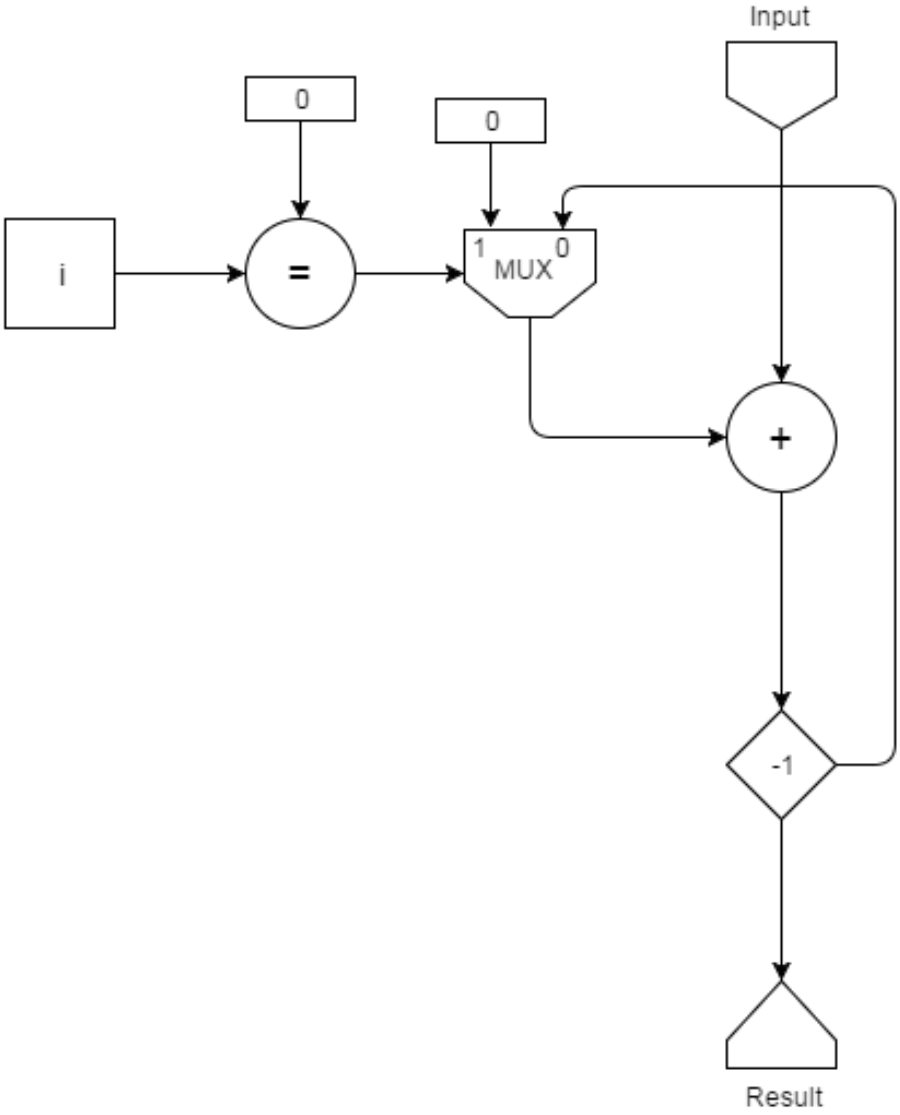
Αρχικά για να επιτευχθεί η επεξεργασία πολλαπλών υπολογισμών ανά κύκλο, γίνεται δήλωση του τύπου, στον οποίο δηλώνεται το πλήθος των αριθμών που θα τυγχάνει επεξεργασίας. Στην συνέχεια, χρησιμοποιώντας τον τύπο που δηλώθηκε πιο πριν, η συνάρτηση input υπολογίζει πόσα bytes θα ζητήσει για την είσοδο. Έπειτα, γίνεται δημιουργία ενός προσωρινού πίνακα για την αποθήκευση των αποτελεσμάτων. Τέλος, χρησιμοποιείται βρόγχος, ο οποίος κατά την μεταγλώττιση θα γίνει unroll και κάθε θέση του πίνακα result ενώνεται (<==) με το εκτιμώμενο αποτέλεσμα.

Μετατροπή βρόγχου με εξάρτηση:

CPU Code:

```
for (int i = 0; i < size; i++)
        result[i] = input[i]+i==0?0:result[i-1];
```

Το πρόβλημα φαίνεται από τον γράφο που παράγεται από την άμεση μετατροπή του πιο πάνω κώδικα που είναι:



Σχήμα 2.3: Γράφος υλοποίησης με κυκλική εξάρτηση.

Στην περίπτωση που στον πολυπλέκτη έρχεται σαν σήμα επιλογής το 0 (δηλαδή σε όλες τις περιπτώσεις εκτός της πρώτης), θα πρέπει να επιστραφεί η είσοδος που υπολογίζεται από την εκτέλεση της λειτουργίας του πολυκπλέκτη και το άθροισμα της εξόδου του με την είσοδο του προηγούμενου κύκλου. Οι δύο αυτές λειτουργίες όμως, λόγω του ότι χρειάζονται περισσότερους από 1 κύκλους για να ολοκληρωθούν, θα προκαλέσουν πρόβλημα στην

σχεδίαση του προγράμματος, αφού δεν θα προλάβουν να εκτελεστούν για να παραχθεί το αποτέλεσμα, το οποίο χρειάζεται στον αμέσως επόμενο κύκλο.
Έτσι εισάγεται μια καθυστέρηση loopLength ανά κύκλο, προκειμένου να είναι εγγυημένο ότι οι είσοδοι του πολυπλέκτη θα είναι υπολογισμένες.

Maxeler Code:

CPU Code:

```
Loop(size, input, result);
```

Manager Code:

```
Ρύθμιση ροών:
manager.setIO(link("input", CPU),
              link("result", CPU));
m.createSLiCinterface();


Ρύθμιση εκτέλεσης:
InterfaceParam  size    = engine_interface.addParam("size",
CPUTypes.INT);

InterfaceParam loopLength =
engine_interface.getAutoLoopOffset(s_kernelName,
"loopLength");
engine_interface.ignoreAutoLoopOffset(s_kernelName,
"loopLength");

engine_interface.setTicks(s_kernelName, size*loopLength);
engine_interface.setStream("input",   CPUTypes.INT32, size *
CPUTypes.INT32.sizeInBytes());
engine_interface.setStream("result", CPUTypes.INT32, size *
CPUTypes.INT32.sizeInBytes());
```

Σε αυτή την περίπτωση πρέπει να γίνει ρύθμιση και της εκτέλεσης. Η αυτόματη ρύθμιση γίνεται για τα πολύ απλά προβλήματα, στην πλειονότητα τους όμως χρειάζεται να γίνει από τον χρήστη.

Στην ρύθμιση εκτέλεσης, δηλώνεται η παράμετρος size που αναμένεται να έρθει από το CPU. Στην συνέχεια λαμβάνεται από τον πυρήνα το μέγεθος του loop και χρησιμοποιείται για να υπολογιστεί ο αριθμός των κύκλων για τον οποίο θα τρέξει η εκτέλεση. Κανονικά ο αριθμός αυτός θα ήταν μεγέθους size, αλλά επειδή τα αποτελέσματα παράγονται κάθε loop κύκλους, γίνεται size*loop. Τέλος, δηλώνεται το μέγεθος των ροών, οι οποίες θα χρησιμοποιηθούν από την πυρήνα.

Kernel Code:

```
OffsetExpr loopLength =
stream.makeOffsetAutoLoop("loopLength");
DFEVar loopLengthVal = loopLength.getDFEVar(this,
dfeUInt(8));
CounterChain chain = control.count.makeCounterChain();
DFEVar i = chain.addCounter(size, 1); //size counter max
value, 1 number of increment
DFEVar loopCounter = chain.addCounter(loopLengthVal, 1);

DFEVar input = io.input("input", dfeInt(32),
loopCounter===(loopLengthVal-1));

DFEVar carriedSum=dfeInt(32).newInstance(this);
DFEVar sum = x + (i===0?0:carriedSum);
carriedSum <== stream.offset(sum, -loopLength);

io.output("result", sum, dfeInt(32),
loopCounter===(loopLengthVal-1));
```

Στον κώδικα του πυρήνα αρχικά δηλώνεται ένας ειδικός τύπος, ο οποίος χρησιμοποιείται για την συμβολική αναφορά στο μέγεθος της κυκλικής εξάρτησης, χωρίς να είναι αναγκαία η γνώση του μεγέθους της εκ των προτέρων. Έπειτα, χρησιμοποιείται η κλάση CounterChain, η οποία επιτρέπει στον μετρητή i να μεταβάλλεται κάθε loopLengthVal κύκλους.

Στην είσοδο γίνεται χρήση της δομής αυτής έτσι ώστε να λαμβάνονται νέα δεδομένα στον τελευταίο κύκλου της περιόδου loopLengthVal κύκλων. Με αυτόν τον τρόπο, το αποτέλεσμα του αθροίσματος του προηγούμενου κύκλου θα προλάβει να υπολογιστεί πριν χρησιμοποιηθεί. Στην συνέχεια δημιουργείται μια ασύνδετη μεταβλητή, η οποία στην συνέχεια θα ενωθεί με το άθροισμα του προηγούμενου κύκλου. Υπολογίζεται το άθροισμα και στην συνέχεια γίνεται η ένωση της μεταβλητής. Τέλος, το αποτέλεσμα επιστρέφεται και αυτό στο τέλος της περιόδου των loopLengthVal κύκλων, αφού τότε παράγονται τα πραγματικά αποτελέσματα.

2.5 Ανασκόπηση

Το κεφάλαιο αυτό ολοκληρώνεται αναφέροντας ότι πιο πάνω έχει δοθεί μια γενική περιγραφή του τρόπου λειτουργίας της πλατφόρμας της Maxeler, καθώς επίσης και ορισμένα στοιχεία του προγραμματιστικού περιβάλλοντος. Στόχος ήταν να γίνει μια σύντομη επεξήγηση έτσι ώστε ο χρήστης, στην περίπτωση που δεν είχε στο παρελθόν επαφή με τη

συγκεκριμένη πλατφόρμα, να μπορεί να αντιληφθεί τις ιδέες που θα ακολουθήσουν στην συνέχεια.

Κεφάλαιο 3

Σχετική Δουλειά

3.1 Ανάλυση εργασιών πάνω στην πλατφόρμα της Maxeler

Στόχος του πρώτου εξαμήνου ήταν η κατανόηση του θέματος, έτσι ώστε να υπάρχει δυνατότητα να γίνει εκτίμηση του κατά πόσο μια εφαρμογή μπορεί να λειτουργήσει αποδοτικά σε περιβάλλον ροής δεδομένων, αλλά και της βελτίωσης που δύναται να περιμένουμε από μια τέτοια μετάβαση.

Αρχικά μελετήθηκαν τα εγχειρίδια του προγραμματιστικού περιβάλλοντος για το σύστημα της Maxeler [1],[2],[3],[4],[5] έτσι ώστε να γίνει μια εισαγωγή επί του θέματος και στην συνέχεια, αφού ετέθησαν οι βάσεις, έγινε μελέτη διάφορων εργασιών που αναπτύχθηκαν πάνω στον MaxCompiler, στους τύπους των προβλημάτων που στόχευσαν και στις επιδόσεις που πέτυχαν.

Η χρήση της πλατφόρμας της Maxeler, βρίσκει άμεσα εφαρμογή στα προγράμματα που επιλύουν προβλήματα μερικών διαφορικών εξισώσεων με χρήση πεπερασμένων διαφορών. Μέσω του MaxGenFD, εργαλείο που αποτελεί το προγραμματιστικό πακέτο της Maxeler, γίνεται διαχείριση όλων των δυσκολίων που πιθανώς να αντιμετωπίσει ο προγραμματιστής στην υλοποίηση πεπερασμένων διαφορών.

Άμεση εφαρμογή αυτού υπάρχει στον τομέα της γεωλογίας, αφού μπορούν εύκολα να δημιουργηθούν εφαρμογές που υλοποιούν αλγόριθμους Forward Modeling, Reverse Time Migration και Waveform Inversion.

Υπήρξαν μελέτες [6],[7] που έγιναν για να διαφανεί το μέτρο της επιτάχυνσης που μπορεί να επιτευχθεί με την χρήση μηχανών ροής δεδομένων σε εφαρμογές σεισμικής επεξεργασίας. Στην [6], έγινε προσπάθεια μοντελοποίησης σεισμικών κυμάτων μέχρι και 70khz. Η

δυσκολία του εγχειρήματος έγκειται στο γεγονός ότι το πλήθος των αριθμητικών πράξεων που χρειάζονται είναι της τάξεως της τέταρτης δύναμης της συχνότητας. Εντούτοις, χρησιμοποιώντας μηχανές ροής δεδομένων, επιτεύχθηκε σημαντική βελτίωση στο χρόνο, έχοντας αναλογικά περίπου 200 πυρήνες CPU να έχουν την ίδια επίδοση με μια MAX2 κάρτα.

Σε μια άλλη εργασία επί του θέματος [7], μελετήθηκε η υλοποίηση του αλγόριθμου 3D Zero-offset CRS Stacking. Ο βαρετός αλγόριθμος, τόσο σε υπολογιστικές ανάγκες, όσο και σε μνήμη, επωφελήθηκε σημαντικά από την υπολογιστική δύναμη, αλλά και το εύρος της μνήμης της μηχανής ροής δεδομένων. Στην τελική υλοποίηση αναφέρεται πως σημειώθηκε επιτάχυνση περίπου 200 φορές σε σχέση με τους χρόνους εκτέλεσης της FORTRAN εφαρμογής σε ένα πυρήνα CPU.

Στο [8], μελετήθηκε κατά πόσο μπορεί να βελτιωθεί η προσομοίωση διάδοσης του κύματος για την εξαγωγή πληροφοριών σχετικά με το έδαφος. Είναι ένα δύσκολο πρόβλημα γιατί για να δημιουργηθεί η εικόνα του εδάφους πρώτα ενεργοποιείται ένα σήμα χαμηλής συχνότητας το οποίο αντανακλάται στο έδαφος και έπειτα καταγράφεται από χιλιάδες δέκτες. Η διαδικασία αυτή δημιουργεί ένα όγκο δεδομένων εκατοντάδων τεραμπάιτ. Έτσι μέσω της υλοποίησης με σύστημα ροής δεδομένων, έγινε κατορθωτό να επιτευχθεί επιτάχυνση στο χρόνο κατά 70 φορές περισσότερο της συμβατικής υλοποίησης με CPU. Σημειώνεται ότι στην υλοποίηση έγινε πειραματισμός με αντικατάσταση των μηχανών ροής δεδομένων με GPU. Οι χρόνοι που επιτεύχθηκαν ήταν 10 φορές καλύτεροι έναντι της υλοποίησης με συμβατικά CPU, χρόνοι καλοί αλλά σαφώς χειρότεροι από την υλοποίηση με μηχανές ροής δεδομένων.

Εφαρμογές παρατηρούνται επίσης στον οικονομικό τομέα. Η ανάγκη για χρήση όλο και πιο πολύπλοκων μαθηματικών μοντέλων για τον υπολογισμό του ρίσκου των παραγώγων και των κεφαλαίων είχε ως αποτέλεσμα την ώθηση των επιστημόνων στην αναζήτηση νέων, πιο αποδοτικών λύσεων, τόσο υπολογιστικά, όσο και ενεργειακά. Στις [9],[10], μετά από ανάλυση του τρόπου λειτουργίας της εφαρμογής, έγινε εύρεση των κομματιών που έχρηζαν βελτίωσης και έπειτα υλοποίηση τους στην μηχανή ροής δεδομένων. Μέσω αυτού, έγινε επιτάχυνση του χρόνου, σε σχέση με την αντίστοιχη υλοποίηση που δεν χρησιμοποίησε τις δύο μηχανές ροής δεδομένων, κατά 31 φορές κατά τον υπολογισμό με πλήρη ακρίβεια και κατά 37 φορές με μειωμένη.

Μελέτες έγιναν και στον τομέα της βιοπληροφορικής. Στην [11] στοχεύτηκε ο αλγόριθμος Smith-Waterman με μηχανή ροής δεδομένων για την ευθυγράμμιση τοπικών ακολουθιών. Στην παρουσίαση αποτελεσμάτων παρατηρήθηκε ότι με ένα Maxeler MaxNode με 4 Max2 DFEs υπήρξε επιτάχυνση του χρόνου κατά 128 φορές έναντι της SSE επιταχυνόμενης SSEARCH υλοποίησης.

Τέλος, μελετήθηκε εργασία [12] με θέμα από τον κλάδο της ρευστοδυναμικής. Συγκεκριμένα μελετήθηκε το κατά πόσο υπάρχει δυνατότητα επιτάχυνσης της προσομοίωσης της ροής των ρευστών χρησιμοποιώντας την μέθοδο Lattice Boltzmann (LBM). Ο αλγόριθμος είναι ιδανικός για υπολογισμούς τύπου streaming λόγω του παραλληλισμού που προσφέρει. Μετά την υλοποίηση σε FPGA για την επιτάχυνσή του, πάρθηκαν διάφορες μετρήσεις καθιστώντας την εφαρμογή που έγινε με το FPGA να έχει 2.93 φορές καλύτερο χρόνο από ένα μονοπύρηνο 3.4GHz Intel Pentium 4 και 2.46 φορές καλύτερο χρόνο από ένα μονοπύρηνο 2.2GHz AMD Opteron.

Τα αποτελέσματα των επιδόσεων έχουν κάποια βαρύτητα, θα πρέπει όμως να αναφερθεί ότι ήταν περιορισμένα λόγω της χρήσης της διεπαφής PCI-Express, η οποία εικάζεται ότι ήταν το bottleneck.

Παρατηρούμε ότι η πλατφόρμα της Maxeler με σωστή και προσεγμένη χρήση μπορεί να προσδώσει βελτίωση στους χρόνους σε μια σχετικά ευρεία γκάμα προβλημάτων.

Κεφάλαιο 4

TPC-H Benchmark Platform

4.1 Εισαγωγή στην Πλατφόρμα Πειραμάτων TPC-H

Η πλατφόρμα πειραμάτων TPC-H αποσκοπεί στην αξιολόγηση διάφορων συστημάτων ως προς την υποστήριξη αποφάσεων. Συγκεκριμένα, αποτελείται από διάφορα ad-hoc ερωτήματα που απευθύνονται στις ανάγκες των επιχειρήσεων. Κάθε ερώτημα αξιολογεί διάφορα κομμάτια του συστήματος, όπως παραδείγματος χάρη την επεξεργασία μεγάλου όγκου δεδομένων και την συμπεριφορά του υπό την παράλληλη τροποποίηση δεδομένων.

Στα πλαίσια της διπλωματικής εργασίας θα προσεγγιστούν διάφορα ερωτήματα, τα οποία απαιτούν την επεξεργασία μεγάλου όγκου δεδομένων.
Για την επίδειξη των υλοποιήσεων, πρώτα γίνεται αναφορά στην υλοποίηση της CPU και μετά σε αυτή της πλατφόρμας της Maxeler. Επίσης, περιγράφεται ο κώδικας του πυρήνα αφού η υλοποίηση της CPU ήταν απλά το κάλεσμα της λειτουργίας στην μηχανή ροής δεδομένων, λόγω του ότι ήταν σχετικά μικρά τα προγράμματα και στην υλοποίηση του Manager γίνονται μηχανικά κάποιες λειτουργίες βάσει του πυρήνα.

Σημειώνεται ότι το πρώτο κομμάτι θα περιγραφεί αναλυτικά για πληροφοριακούς σκοπούς.

4.2 Shipping Priority Query (Ερώτημα 3)

4.2.1 Περιγραφή

Το ερώτημα επιστρέφει τις 10 παραγγελίες με τις ψηλότερες τιμές οι οποίες ακόμα να αποσταλούν.

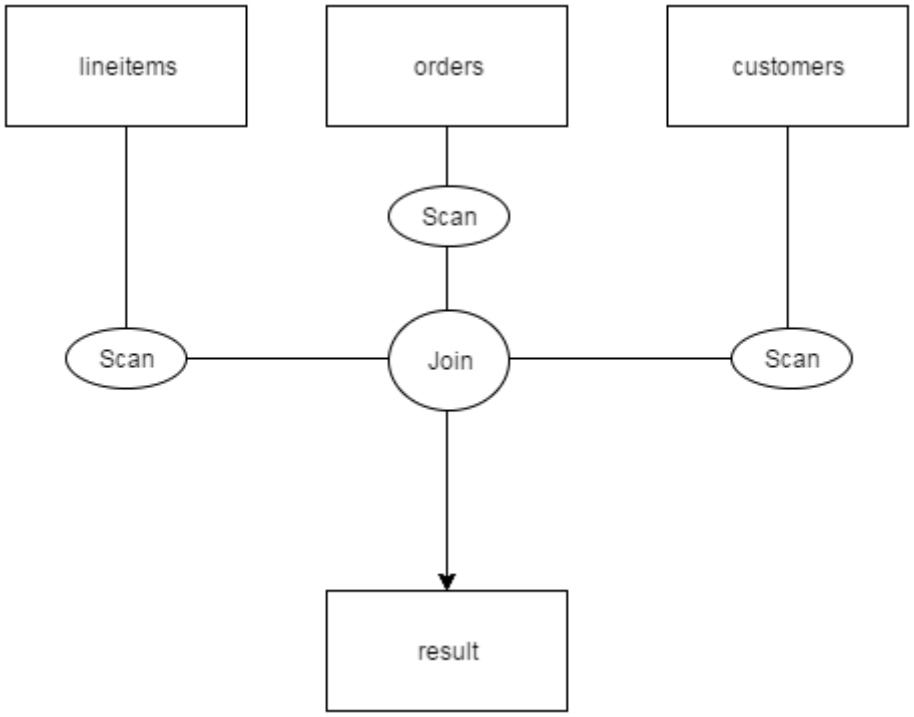
Αναλυτικά, το ερώτημα επιστρέφει την προτεραιότητα αποστολής και τα πιθανά έσοδα, που υπολογίζονται ως το άθροισμα των τιμών των προϊόντων μετά την έκπτωση για τις παραγγελίες, οι οποίες έχουν τα μεγαλύτερα έσοδα και οι οποίες δεν έχουν αποσταλεί μέχρι και μια συγκεκριμένη ημερομηνία. Επιστρέφονται οι πρώτες δέκα παραγγελίες σε φθίνουσα σειρά βάσει του εισοδήματός τους.

```
select
    l_orderkey,
    sum(l_extendedprice*(1-l_discount)) as revenue,
    o_orderdate,
    o_shippriority
from
    customer,
    orders,
    lineitem
where
    c_mktsegment = '[SEGMENT]'
    and c_custkey = o_custkey
    and l_orderkey = o_orderkey
    and o_orderdate < date '[DATE]'
    and l_shipdate > date '[DATE]'
group by
    l_orderkey,
    o_orderdate,
    o_shippriority
order by
    revenue desc,
    o_orderdate;
```

4.2.2 Φάση Α (Nested Loop Join)

Η αρχική προσέγγιση, αφού αναλύθηκε το πρόβλημα και παρατηρήθηκε ότι θα χρειαστεί η ένωση των τριών πινάκων, ήταν να υπολογιστεί αυτή με την χρήση φωλιασμένων for (Nested Loop Join). Έτσι, κάθε εγγραφή κάθε πίνακα ελέγχεται με κάθε εγγραφή των άλλων πινάκων για να βρεθούν οι ενώσεις και να υπολογιστούν οι άλλες παράμετροι του προβλήματος.
Κώδικας CPU:

```
for(i=0;i<LINEITEM;++i){
    curr_ord=lineitem[i][0];
    if(lineitem[i][3]>1980){
        for(j=0;j<ORDERS;++j){
            if((curr_ord==orders[j][0]) && (orders[j][2]<1995)){
                for(k=0;k<CUSTOMER;++k){
                    if((customer[k][0]==orders[j][1]) && (customer[k][1]==2)){
                        orderkey_temp[count_temp]=customer[k][0];
                        count_temp++;
                    }
                }
                break;
            }
        }
    }
}
```



Σχήμα 4.1: Διάγραμμα ροής Ερωτήματος 3, Φάση Α

Η μετατροπή του κώδικα από CPU στην πλατφόρμα της Maxeler είναι σχετικά απλή, τουλάχιστο για το κομμάτι του πυρήνα. Αρχικά σημειώνεται ότι λόγω του ότι οι εντολές ελέγχου ροής δεν μπορούν να τύχουν εξομοίωσης από την μηχανή ροής δεδομένων, το πρόγραμμα της CPU μετατράπηκε σε:

```
for(i=0;i<LINEITEM;++i){
    for(j=0;j<ORDERS;++j){
        for(k=0;k<CUSTOMER;++k){
            if((lineitem[i][3]>1980) && (lineitem[i][0]==orders[j][0]) &&
(orders[j][2]<1995) && (customer[k][0]==orders[j][1]) &&
(customer[k][1]==2)){
                orderkey_temp[count_temp]=customer[k][0];
                count_temp++;
            }
        }
    }
}
```

Στόχος είναι η μετατροπή του εσωτερικού κομματιού του βρόγχου με τον μετρητή k στην πλατφόρμα της Maxeler, έτσι ώστε η λειτουργία αυτή να γίνεται από υλικό με πολλαπλά στάδια διασωλήνωσης. Έτσι στο τέλος, ουσιαστικά θα γίνεται υπολογισμός του κομματιού αυτού σε ένα κύκλο της μηχανής ροής.

Κατά την μετατροπή, η μόνη ιδιαιτερότητα πλέον είναι στην προσαρμογή των ροών, έτσι ώστε τα δεδομένα που έρχονται στον πυρήνα ανά κύκλο να είναι ίδια με τα δεδομένα που τυγχάνουν επεξεργασίας από το CPU ανά επανάληψη. Αφού γίνει επιτευκτή η αντιστοίχιση των ροών, οι λειτουργίες στο εσωτερικό του βρόγχου μπορούν να τύχουν άμεσης μετάφρασης από την μηχανή ροής δεδομένων.

Κατά την υλοποίηση, αρχικά γίνεται δημιουργία τριών μετρητών που αντιστοιχούν στις εγγραφές που έτυχαν επεξεργασίας σε κάθε πίνακα. Οι μετρητές ρυθμίζονται με τέτοιο τρόπο ώστε μόλις ο αριθμός εγγραφών του εκάστοτε πίνακα ξεπεραστεί, τότε μηδενίζεται και αρχίζει ξανά από την αρχή. Επιπλέον, δημιουργείται και μια γραμμική ιεραρχία στον τρόπο μετρήματος έτσι ώστε κάθε μετρητής να βασίζεται στο αν ο επόμενος μετρητής ολοκλήρωσε όλο το φάσμα αναζήτησής του (δηλαδή μηδενίστηκε).

Για παράδειγμα, έστω τρεις πίνακες A,B,C έχουν μέγεθος τριών εγγραφών:

Κύκλος	1	2	3	4	5	6	7	8	9	10
counterA	0	1	2	0	1	2	0	1	2	0
counterB	0	0	0	1	1	1	2	2	2	0
counterC	0	0	0	0	0	0	0	0	0	1

Πίνακας 4.1: Παράδειγμα εκτέλεσης μετρητών για 10 κύκλους

Παρακάτω είναι ο κώδικας που περιγράφει την πιο πάνω διαδικασία:

```
1 Count.Params paramsCu = control.count.makeParams(cWidth)
2   .withMax(cuSize); //customer counter parameters
3
4 Count.Params paramsOr = control.count.makeParams(cWidth)
5   .withEnable(counterCu.getWrap())
6   .withMax(orSize);//orders counter parameters
7
8 Count.Params paramsLi = control.count.makeParams(cWidth)
9   .withEnable(counterOr.getWrap())
10  .withMax(liSize);//lineitems counter parameters
```

Η συνάρτηση `getWrap()` επιστρέφει `true` αν ο μετρητής στον οποίο εκτελείται έχει μηδενιστεί λόγω του max. Βλέπουμε δηλαδή ότι ο μετρητής του πίνακα customers θα μετράει συνεχώς και όταν φτάσει στο `cuSize` θα μηδενίζεται και θα ξεκινά από την αρχή. Ο μετρητής του πίνακα orders θα αλλάζει τιμή μόνο όταν μηδενιστεί λόγω του max ο μετρητής του customers και παρόμοια και ο μετρητής του lineitems.

Πρακτικά, γίνεται εξομοίωση των μετρητών `i,j,k` του προγράμματος της CPU.

Στην συνέχεια, οι μετρητές χρησιμοποιούνται για να ενεργοποιήσουν τις ροές για τον κάθε πίνακα. Η ροή του πίνακα customer είναι πάντα ενεργοποιημένη, του πίνακα order είναι ενεργοποιημένη μόνο όταν ο μετρητής του customer είναι ίσος με 0 και η ροή του lineitems είναι ενεργοποιημένη όταν ο μετρητής του customers και του orders είναι 0.
Παρακάτω είναι ο κώδικας που περιγράφει την πιο πάνω διαδικασία:

```
1 io.input("customers", dfeFloat(8, 24));
2 io.input("orders", dfeFloat(8, 24), icu.eq(0));
3 io.input("lineitems", dfeFloat(8, 24), ior.eq(0)&icu.eq(0));
```

Η δεύτερη παράμετρος της εντολής input είναι ο τύπος των δεδομένων. Η τρίτη παράμετρος, αν δίδεται, ρυθμίζει το αν θα είναι ενεργοποιημένη η ροή. Αν όχι, επιστρέφει την τιμή που επέστρεψε την τελευταία φορά. Επιπλέον, λόγω του ότι οι πίνακες orders και customers

διατρέχονται περισσότερο από μία φορές, πράγμα το οποίο δεν μπορεί να επιτευχθεί από την απευθείας μεταφορά από το CPU, αποθηκεύονται στην εξωτερική μνήμη της μηχανής ροής. Με αυτό τον τρόπο έχουμε επίτευξη της αντιστοίχισης των ροών ανά κύκλο.

Έπειτα, γίνονται οι έλεγχοι πάνω στις εγγραφές και η αληθοτιμή αυτών μπαίνει σαν σήμα ενεργοποίησης σε μετρητή παρόμοιο με τον `count_temp` του προγράμματος της CPU. Στον τέλος του προγράμματος επιστρέφεται ο μετρητής αυτός.

Στην υλοποίηση του Manager, αρχικά γίνεται δήλωση και ρύθμιση των ροών οι οποίες θα χρησιμοποιηθούν τον/τους πυρήνες.

```
1   KernelBlock block = addKernel(new Q3Kernel(makeKernelParameters(s_kernelName)));
2   LMemInterface iface = addLMemInterface();
3
4   DFELink cpu2lmemCust = iface.addStreamToLMem("cpu2lmemCust", LMemCommand-
Group.MemoryAccessPattern.LINEAR_1D);
5   DFELink cpu2lmemOrd = iface.addStreamToLMem("cpu2lmemOrd", LMemCommand-
Group.MemoryAccessPattern.LINEAR_1D);
6
7   cpu2lmemCust <== addStreamFromCPU("customers");
8   cpu2lmemOrd <== addStreamFromCPU("orders");
9
10  DFELink LMemCustomers = iface.addStreamFromLMem("LMemCustomers", LMemCommand-
Group.MemoryAccessPattern.LINEAR_1D);
11  DFELink LMemOrders = iface.addStreamFromLMem("LMemOrders", LMemCommand-
Group.MemoryAccessPattern.LINEAR_1D);
12
13  block.getInput("lineitems") <== addStreamFromCPU("lineitems");
14  block.getInput("customers") <== LMemCustomers;
15  block.getInput("orders") <== LMemOrders;
16  addStreamToCPU("matched") <== block.getOutput("matched");
17
18  createSLiCinterface(modeDefault());
19  createSLiCinterface(modeWrite("writeLMem"));
```

Αρχικά γίνεται δήλωση του block του πυρήνα, το οποίο θα χρησιμοποιηθεί στην συνέχεια για να ρυθμιστούν οι ροές του. Στην συνέχεια δημιουργούνται ροές από και προς της εξωτερική μνήμη (LMem) με την χρήση της συνάρτησης addStreamToLMem, addStreamFromLMem και για το CPU, addStreamToCPU και addStreamFromCPU. Έπειτα, με την χρήση του τελεστή "<==", γίνεται η ένωση των διαφόρων ροών. Στην γραμμή 7, παραδείγματος χάρη, ρυθμίζουμε την ροή που πάει προς την εξωτερική μνήμη με το όνομα cpu2lmemCust να ενωθεί με την ροή που έρχεται από το CPU με το όνομα customers. Τέλος, δημιουργούνται οι διεπαφές για την εκτέλεση των λειτουργιών της προκαθορισμένης επιλογής (κάλεσμα Q3(…) από το CPU) και της επιλογής writeLMem (κάλεσμα Q3_writeLMem(…) απο το CPU).

Με το κάλεσμα του modeWrite πρέπει να γίνεται εγγραφή των πινάκων customers και orders στην εξωτερική μνήμη της μηχανής ροής δεδομένων.

Η λειτουργία για την εκτέλεση του modeWrite:

```
1 engine_interface.setStream("customers", CPUTypes.FLOAT, cuSize*fsize*customerDim);
2 engine_interface.setStream("orders", CPUTypes.FLOAT, orSize*fsize*orderDim);
3 engine_interface.setLMemLinear("cpu2lmemCust", zero, cuSize*customerDim*fsize);
4 engine_interface.setLMemLinear("cpu2lmemOrd", cuSize*customerDim*fsize,
orSize*orderDim*fsize);
```

Στις πρώτες δύο γραμμές του κώδικα γίνεται δήλωση του μεγέθους σε bytes των ροών “customers” και “orders” οι οποίες δημιουργήθηκαν πιο πριν. Τέλος, γίνεται ρύθμιση των ροών που κατευθύνονται προς την εξωτερική μνήμη. Η πρώτη παράμετρος της συνάρτησης setLMemLinear είναι το όνομα της ροής, η δεύτερη είναι η θέση σε bytes στην οποία θα γίνει η εγγραφή και η τρίτη είναι το μέγεθος σε bytes της ροής.

Με το κάλεσμα της προκαθορισμένης επιλογής γίνεται η εκτέλεση του κυρίως κομματιού.

```
1   int fsize = CPUTypes.FLOAT.sizeInBytes();
2   engine_interface.setTicks(s_kernelName, liSize*cuSize*orSize);
3
4   engine_interface.setStream("lineitems", CPUTypes.FLOAT, liSize*lineitemDim*fsize);
5   engine_interface.setLMemLinearWrapped("LMemCustomers", zero, cuSize*cus-
tomerDim*fsize, liSize*orSize*cuSize*customerDim*fsize, zero);
6   engine_interface.setLMemLinearWrapped("LMemOrders", cuSize*customerDim*fsize, or-
Size*orderDim*fsize, liSize*orSize*orderDim*fsize, zero);
7   engine_interface.setStream("matched", CPUTypes.FLOAT, liSize*cuSize*orSize*fsize);
```

Αρχικά γίνεται καθορισμός του αριθμού των κύκλων που θα τρέξει ο πυρήνας. Μετά γίνεται η ρύθμιση των ροών από και προς τον πυρήνα από την CPU με την χρήση της εντολής setStream. Για τις ροές που έρχονται από την μνήμη, παρατηρούμε ότι χρησιμοποιείται η εντολή setLMemLinearWrapped, πιο εξειδικευμένη από την setLMemLinear, η οποία επιτρέπει την προσπέλαση ενός συγκεκριμένου κομματιού της μνήμης πολλαπλές φορές. Η πρώτη παράμετρός της είναι η θέση στην μνήμη από την οποία ξεκινά, η δεύτερη είναι το μέγεθος του χώρου που καταλαμβάνει, η τρίτη είναι το μέγεθος το οποίο θα διαβαστεί/γραφτεί και η τελευταία ρυθμίζει το offset από το οποίο θα ξεκινά την ανάγνωση/εγγραφή. Όλες οι παράμετροι είναι σε bytes.

Στο συγκεκριμένο παράδειγμα, η ροή customers έχει ρυθμιστεί ότι ξεκινά από την θέση 0, έχει μέγεθος το μέγεθος του πίνακα customers επί τις στήλες του πίνακα επί το μέγεθος της

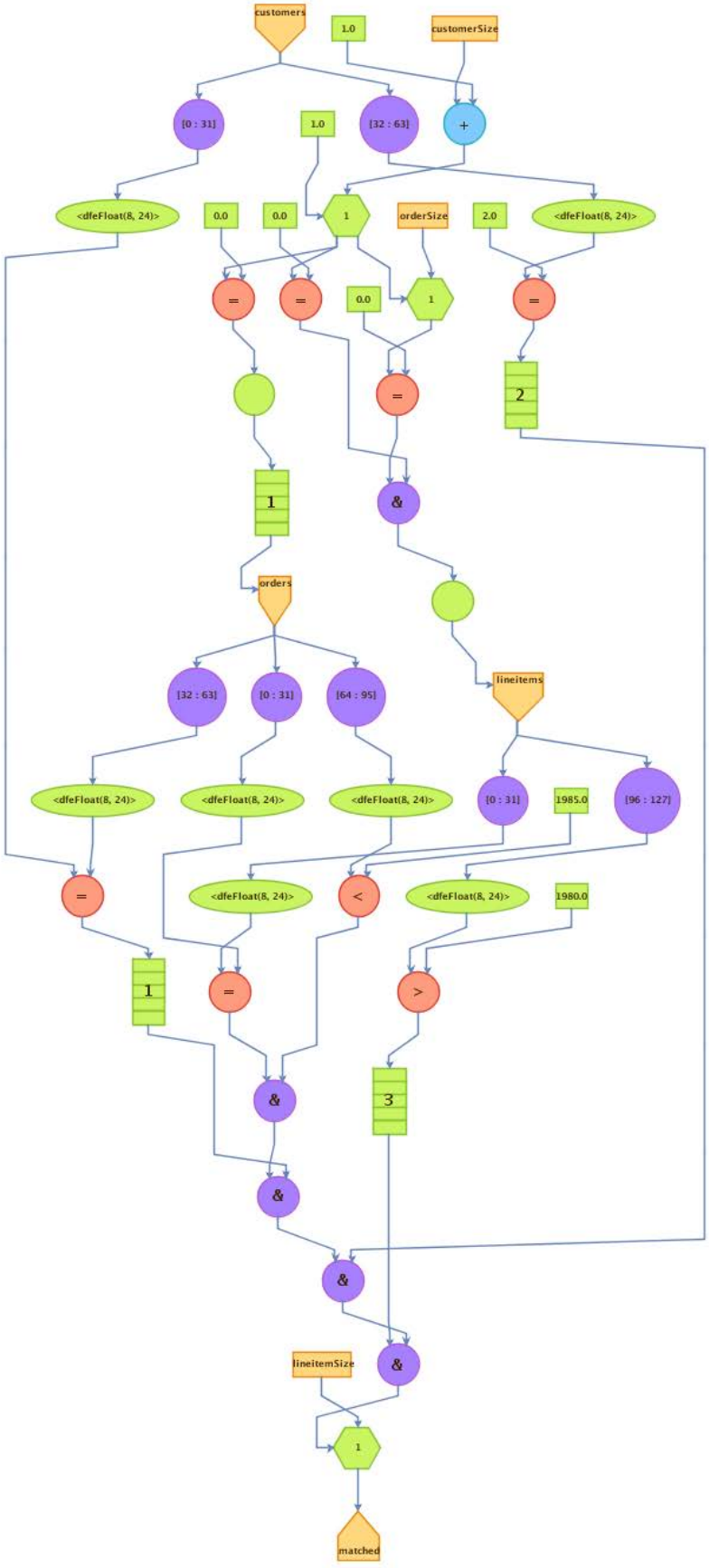
κάθε μιας σε bytes και διαβάζει από αυτή την ορισμένη μνήμη το συνολικό μέγεθός της επί τα μεγέθη των πινάκων lineitems και orders, λόγω του ότι είναι ο εσωτερικός βρόγχος.

Η υλοποίηση του Manager απαιτεί ακρίβεια ως προς την ρύθμιση των ροών και στην ρύθμιση τις διεπαφής. Λανθασμένοι υπολογισμοί θα προκαλέσουν είτε απρόσμενο τερματισμό της εκτέλεσης είτε μη τερματισμό της μηχανής ροής. Και στις δύο περιπτώσεις η αποσφαλμάτωση είναι μια χρονοβόρα διαδικασία.

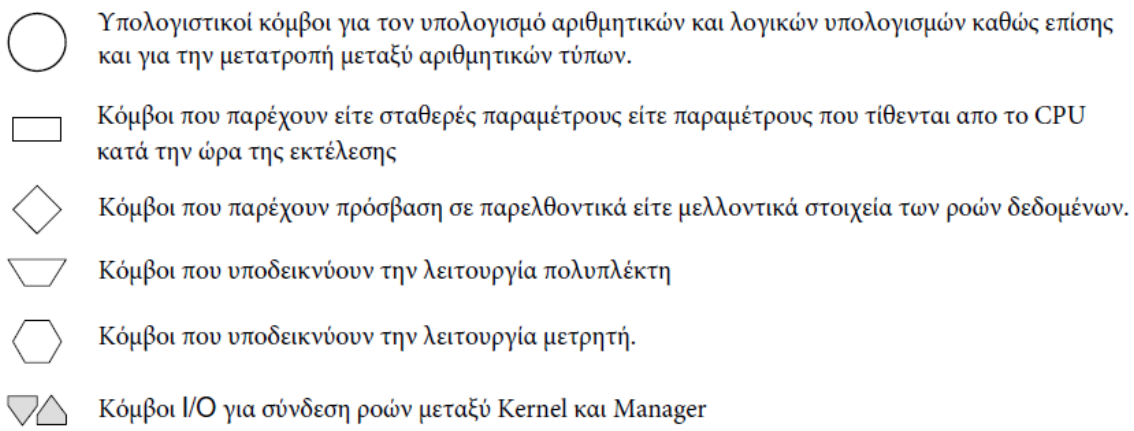
Τέλος, γίνεται η αξιοποίηση των πιο πάνω λειτουργιών από το CPU:

```
Q3_writeLMem(cusize,orsize,customers,orders);
Q3(cusize, lisize, orsize, lineitems, matched);
```

Παρακάτω είναι ο γράφος που δημιουργείται από τον πυρήνα:



Σχήμα 4.2: Γράφος ροής δεδομένων του Ερωτήματος 3 Φάση Α



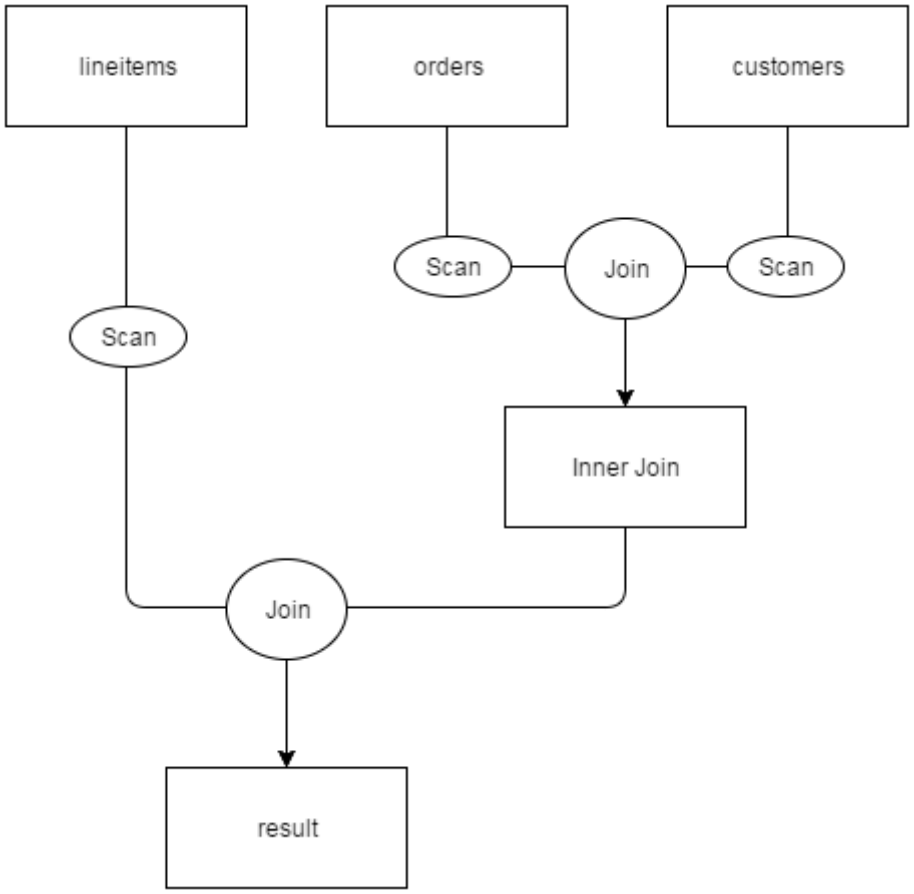
Σχήμα 4.3: Επεξήγηση κόμβων γράφου

4.2.3 Φάση B (Separate Join)

Στην συνέχεια, παρατηρήθηκε ότι η πρώτη προσέγγιση θα μπορούσε να βελτιωθεί αν γινόταν πρώτα η ένωση των εσωτερικών πινάκων. Έτσι, πλέον για κάθε εγγραφή του lineitems θα γίνεται αναζήτηση για το μέγιστο αριθμό εγγραφών $|orders|$ αφού θα εφαρμοστούν και τα ξεχωριστά φίλτρα των πινάκων, αντί σε $|orders| * |customers|$.

Κώδικας CPU:

```
for(j=0;j<ORDERS;++j){
    if(orders[j][2]<1995){
        for(k=0;k<CUSTOMER;++k){
            if((customer[k][0]==orders[j][1]) && (customer[k][1]==2)){
                ij[ijc*2+0]=orders[j][0];
                ij[ijc*2+1]=orders[j][1];
                ijc++;
                break;
            }
        }
    }
}
for(i=0;i<LINEITEM;++i){
    curr_ord=lineitem[i][0];
    if(lineitem[i][3]>1980){
        for(j=0;j<ijc;++j){
            if(curr_ord==ij[j*2+0]){
                orderkey_temp[count_temp]=ij[j*2+1];
                count_temp++;
                break;
            }
        }
    }
}
```



Σχήμα 4.4: Διάγραμμα ροής Ερωτήματος 3, Φάση Β

Για την υλοποίηση του παραπάνω κώδικα χρησιμοποιούνται δύο πυρήνες. Ένας υπεύθυνος για τον υπολογισμό της πρώτης ένωσης και ο άλλος για τον υπολογισμό του αποτελέσματος.

Ο πρώτος πυρήνας αρχικά, με την χρήση της τεχνικής με τους μετρητές κάνει την ένωση των δύο πινάκων αποθηκεύοντας το αποτέλεσμα στην εξωτερική μνήμη.

```
1  DFEVar ctl=(order[2]<1995)&(customer[0]===order[1])&(customer[1]===2);
2  Count.Params paramsWritten = control.count.makeParams(32)
3    .withEnable(ctl | orderCounter>=orSize)
4    .withWrapMode(WrapMode.STOP_AT_MAX)
5    .withMax(orSize);
6  Counter counterWritten = control.count.makeCounter(paramsWritten);
7  DFEVar written = counterWritten.getCount();
8
9  DFEVar extractl=(orderCounter>=orSize) & (written<orSize);
10 DFEVector<DFEVar> sv=ctype.newInstance(this);
11 sv[0]<==extractl?constant.var(dfeFloat(8, 24), -1):order[0];
12 sv[1]<==extractl?constant.var(dfeFloat(8, 24), -1):order[1];
13
14 io.output("join", sv, sv.getType(), (ctl & orderCounter<orSize) | extractl);
15 io.scalarOutput("written", written, written.getType(),
(customerCounter===cuSize-1) & (orderCounter===orSize-1));
```

Συγκεκριμένα, εφαρμόζει τους διάφορους ελέγχους που πρέπει να γίνουν (γραμμή 1, ctl) και αν το αποτέλεσμα αυτών είναι θετικό και ο μετρητής των παραγγελιών δεν ξεπέρασε το

μέγεθος του πίνακα, τότε η ένωση γράφεται στην μνήμη (γραμμή 14, 1η συνθήκη). Λόγω όμως του ότι όλες οι ροές από και προς την εξωτερική μνήμη πρέπει να έχουν μέγεθος σε bytes πολλαπλάσιο του 384 έγιναν κάποιες τροποποιήσεις.

Αρχικά εισήχθη ο μετρητής `written` για να υπάρχει γνώση του πόσες εγγραφές γράφτηκαν στην μνήμη μέχρι στιγμής. Ο μετρητής αυτός είναι σε λειτουργία όταν οι εγγραφές έχουν περάσει τους ελέγχους για εγγραφή στην μνήμη (γραμμή 3, συνθήκη 1) ή όταν το στάδιο της ένωσης των πινάκων τέλειωσε και ο αριθμός των εγγραφών στην μνήμη είναι μικρότερος του πίνακα orders (γραμμή 3, συνθήκη 2).

Βάσει αυτού, εξάγονται πληροφορίες του κατά πόσο πρέπει να γραφτούν κενές εγγραφές στην μνήμη η όχι (`extractl`). Έτσι, συνεχίζεται η εγγραφή στην μνήμη μέχρι να συμπληρωθούν orders εγγραφές, αφού το orders είναι ο μέγιστος αριθμός εγγραφών που μπορεί να έχει η ένωση. Για τον πίνακα orders πρέπει να βεβαιωθεί η CPU ότι το μέγεθός του είναι πολλαπλάσιο των 384 bytes. Τέλος, γίνεται επιστροφή και του πραγματικού αριθμού των εγγραφών (written) με το που τελειώνει η αναζήτηση στους πίνακες (γραμμή 15)

Έπειτα στον δεύτερο πυρήνα, χρησιμοποιώντας τον πίνακα lineitems και τον πίνακα που υπολογίστηκε πιο πριν, γίνεται η ένωση των δύο παρόμοια με την φάση Α. Για το μέγεθος της ένωσης χρησιμοποιείται η τιμή written που επέστρεψε ο πρώτος πυρήνας και χρησιμοποιείται για να υπολογιστεί ο πλησιέστερος αριθμός από αυτήν που είναι πολλαπλάσιο του 384.

4.2.4 Φάση Γ (Hash Join)

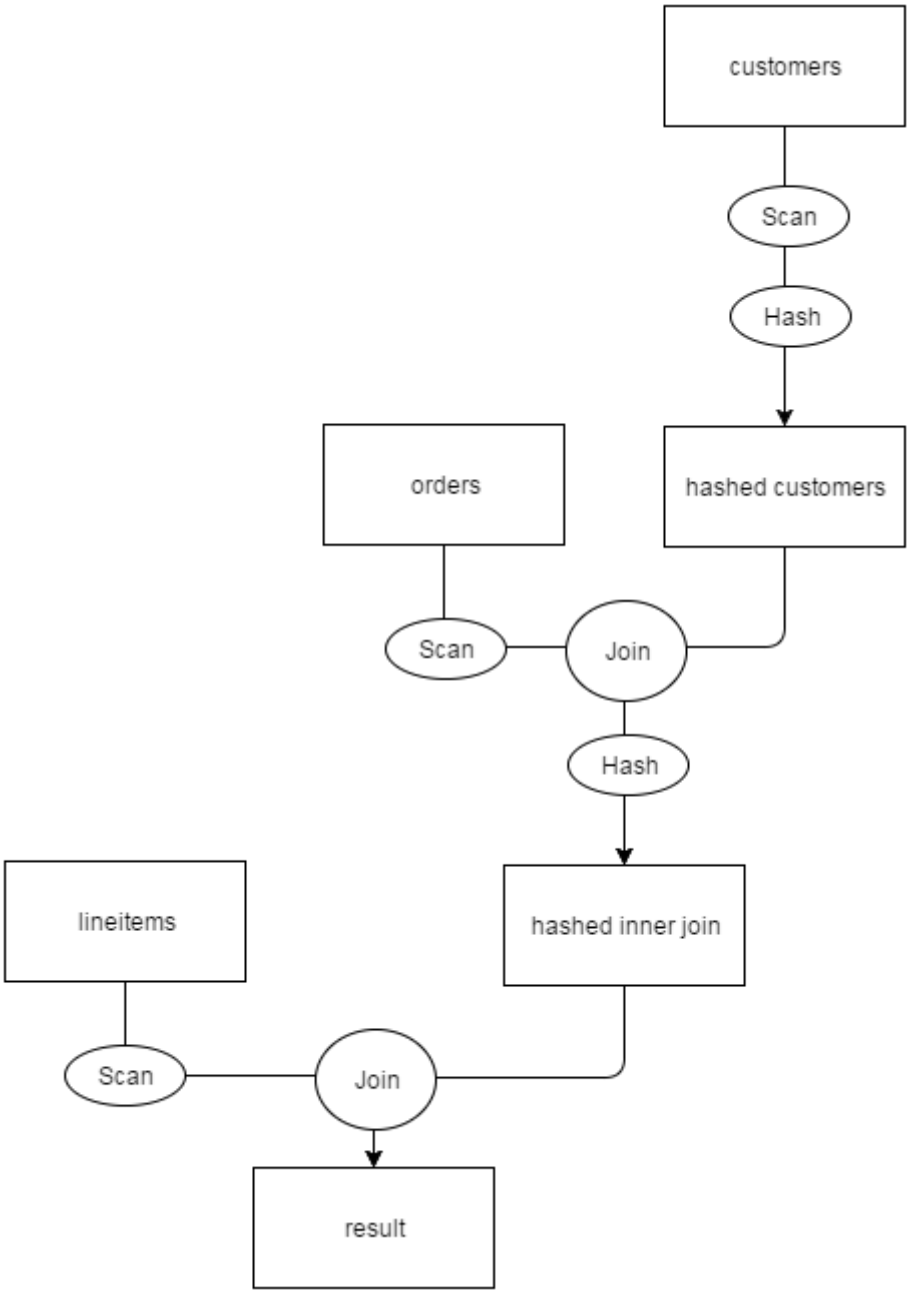
Ακολούθως, έγινε προσπάθεια δημιουργίας μιας υλοποίησης η οποία θα ήταν βασισμένη πάνω στην χρήση κατακερματισμού, hash join. Κατά την συγκεκριμένη υλοποίηση προέκυψαν διάφορα προβλήματα τα οποία από ότι φάνηκε σχετίζονταν με την χρήση της εξωτερικής μνήμης με μη γραμμικό τρόπο. Για τυχόν επίλυση αυτών έγινε προσπάθεια επικοινωνίας με την κλειστή κοινότητα της πλατφόρμας χωρίς αποτέλεσμα. Το πρόβλημα πιθανώς να σχετιζόταν με το MaxelerOs 2015.2 αφού παρατηρήθηκε ότι και άλλοι χρήστες είχαν προβλήματα της ίδιας τάξεως. Έτσι έγιναν κάποιες προσπάθειες για αποφυγή του προβλήματος χωρίς επιτυχία. Θα γίνει αναφορά στην γενική δομή που ακολουθήθηκε.

Κώδικας CPU:

```
//Δημιουργεία hash table το πίνακα customers
for(k=0;k<CUSTOMER;++k){
    if(customer[k][1]==2){
        hi=((int)customer[k][0])%512;
        cuHash[hi*384+cuHashIndex[hi]]=customer[k][0];
        cuHashIndex[hi]++;
    }
}

//Δημιουργεία hash table της ένωσης του πίνακα orders με customers
for(j=0;j<ORDERS;++j){
    if(orders[j][2]<1995){
        hi=((int)orders[j][1])%512;
        for(k=0;k<cuHashIndex[hi];++k){
            if(cuHash[hi*384+k]==orders[j][1]){
                hi=((int)orders[j][0])%512;
                ijHash[hi*3072+ijHashIndex[hi]*2+0]=orders[j][0];
                ijHash[hi*3072+ijHashIndex[hi]*2+1]=orders[j][1];
                ijHashIndex[hi]++;
                break;
            }
        }
    }
}

//Ένωση πίνακα lineitems με την ενδιάμεση ένωση
for(i=0;i<LINEITEM;++i){
    curr_ord=lineitem[i][0];
    if(lineitem[i][3]>1980){
        hi=((int)curr_ord)%512;
        for(j=0;j<ijHashIndex[hi];++j){
            if(curr_ord==ijHash[hi*3072+j*2+0]){
                orderkey_temp[count_temp]=ijHash[hi*3072+j*2+1];
                count_temp++;
                break;
            }
        }
    }
}
```



Σχήμα 4.5: Διάγραμμα ροής Ερωτήματος 3, Φάση Γ

Σε αυτή την υλοποίηση, ο τρόπος που προσεγγίστηκε το πρόβλημα ήταν ο υπολογισμός των έγκυρων εγγραφών και χρησιμοποιώντας τα τελευταία 10 bit του πεδίου κλειδιού να γίνει ο υπολογισμός του partition που πρέπει να γίνει είτε η εγγραφή είτε η εύρεση ισότιμης εγγραφής για την ένωση.

Αρχικά χρησιμοποιήθηκαν προσαρμοσμένοι παραγωγοί διευθύνσεων για την πρόσβαση στην μνήμη. Εξαιτίας του ότι η εξωτερική μνήμη δουλεύει σε bursts των 384 bytes, υπάρχει μια καθυστέρηση περίπου 384/(4*2)=48 κύκλων για την κάθε εγγραφή. Επιπλέον, χρειάζεται και

ένα ευρετήριο για να υπάρχει η γνώση μέχρι σε ποιο σημείο κάποιο partition έχει έγκυρες εγγραφές.

Η άλλη ιδέα ήταν η αποφυγή της πρόσβασης στην μνήμη σε μη γραμμικό τρόπο με το να δίδεται στην μηχανή ροής δεδομένων σαν παράμετρος ο αριθμός του partition ο οποίος θα υπολογίσει (hash). Έτσι σε κάθε λειτουργία, αν η εγγραφή ανήκει στο partition το οποίο έχει δοθεί ως παράμετρος υπολογίζεται, ειδάλλως η εγγραφή ξαναγράφεται σε μια άλλη θέση μνήμης γραμμικά, στην θέση μνήμης για τις εγγραφές που δεν έτυχαν επεξεργασίας ακόμη. Μετά το πέρας της λειτουργίας για το k partition, στην μνήμη με τις ανεπεξέργαστες εγγραφές θα βρίσκονται μόνο οι εγγραφές που ανήκουν στα partition [k+1, 512).
Παρόλα αυτά, θα χρειαστούν $\frac{\kappa*\frac{512*513}{2}}{512*384}$ φορές περισσότεροι κύκλοι, το οποίο για κ=100 ισοδυναμεί με 67 φορές περισσότερους κύκλους (κ είναι ο μέσος όρος εγγραφών ανά partition).

Τέλος η καλύτερη λύση από πλευράς αποφυγής καθυστερήσεων, αλλά και ταυτόχρονα η πιο δαπανηρή από άποψης πόρων, ήταν η χρήση της εσωτερικής μνήμης για την αποθήκευση των κατακερματισμένων πινάκων. Πλέον οι περιορισμοί τις εξωτερικής μνήμης για πιο διακεκριμένη χρήση δεν υφίστανται, δίνοντας την δυνατότητα αξιοποίησης του μεγάλου εύρους της εσωτερικής μνήμης.

4.3 Forecasting Revenue Change Query (Ερώτημα 6)

4.3.1 Περιγραφή

Το ερώτημα 6 ποσοτικοποιεί την αύξηση των εσόδων με την αφαίρεση διάφορων επιχειρησιακών εκπτώσεων σε ένα καθορισμένο εύρος ποσοστού ενός έτους.

Αναλυτικά, γίνεται ανάλυση των στοιχείων του πίνακα lineitems και λαμβάνονται υπόψη όλα τα αντικείμενα τα οποία έχουν παραδοθεί σε ένα συγκεκριμένο έτος και τα οποία είχαν έκπτωση μεταξύ του ‘DISCOUNT-0.01’ και ‘DISCOUNT+0.01’. Το ερώτημα επιστρέφει το επιπλέον σύνολο εσόδων που θα υπήρχε αν γινόταν απαλοιφή των εκπτώσεων για αντικείμενα για τα οποία έγινε παραγγελία σε ποσότητα μικρότερη ενός αριθμού.

```
select
    sum(l_extendedprice*l_discount) as revenue
from
    lineitem
where
    l_shipdate >= date '[DATE]'
    and l_shipdate < date '[DATE]' + interval '1' year
    and l_discount between [DISCOUNT] - 0.01 and [DISCOUNT] + 0.01
    and l_quantity < [QUANTITY];
```

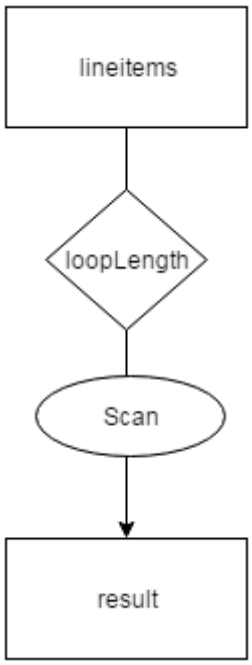
Σχήμα 4.2: Ερώτημα 6 σε SQL

Κώδικας CPU:

```
for(i=0;i<LINEITEM;++i){
    if((line_item[i][2]>=DATE1) && (line_item[i][2]<DATE2) &&
((line_item[i][1]>(DISCOUNT-0.01)) && (line_item[i][1]<(DISCOUNT+0.01))) &&
(line_item[i][3]<QUANTITY)){
        sum+=line_item[i][0]*line_item[i][2];
        result_count++;
    }
}
```

4.3.2 Φάση A (Loop)

Παρότι είναι ένα σχετικά απλό ερώτημα, το γεγονός ότι μεταξύ των κύκλων υπάρχει εξάρτηση απαιτεί ειδικό χειρισμό. Η πρώτη προσέγγιση ήταν για κάθε εγγραφή να γίνεται καθυστέρηση k κύκλων, μέχρι το αποτέλεσμα το οποίο χρειάζεται ο εκάστοτε κύκλος να προλάβει να υπολογιστεί.



Σχήμα 4.6: Διάγραμμα ροής Ερωτήματος 6, Φάση A

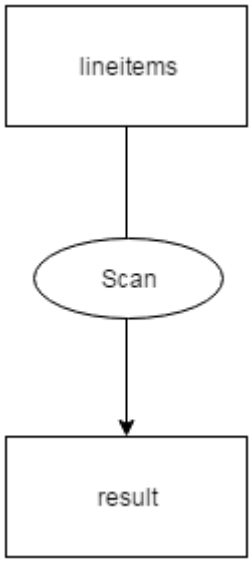
Αρχικά γίνεται δημιουργία ενός μετρητή ο οποίος θα έχει ως μέγιστο τον αριθμό κύκλων που χρειάζεται για να υπολογιστεί το αποτέλεσμα του προηγούμενου κύκλου, όπου και θα μηδενίζεται και θα αρχίζει από την αρχή.

```
Count.Params lparams = control.count.makeParams(8)
    .withMax(loopLengthVal);
```

Έτσι, χρησιμοποιείται αυτός ο μετρητής για τον έλεγχο της ροής και η επίλυση της εξάρτησης γίνεται με την τεχνική που αναλύθηκε στο παράδειγμα στην σελίδα 9.

4.3.3 Φάση B (PushPipeline)

Η βάση αυτής της βελτιστοποίησης ήταν η αποφυγή της καθυστέρησης που συνεπάγεται από την εξάρτηση μεταξύ των κύκλων. Η πλατφόρμα της Maxeler έχει ως αρχή την δημιουργία όσο περισσοτέρων σταδίων διασωλήνωσης για την κάθε λειτουργία που εκτελεί, έτσι ώστε να επιτυγχάνει όσο το δυνατό μικρότερη καθυστέρηση στην συχνότητα παραγωγής αποτελεσμάτων. Έτσι, έγινε προσπάθεια αλλαγής αυτής της συμπεριφοράς με την χρήση της βιβλιοθήκης βελτιστοποιήσεων που προσφέρει η πλατφόρμα.



Σχήμα 4.7: Διάγραμμα ροής Ερωτήματος 6, Φάση B

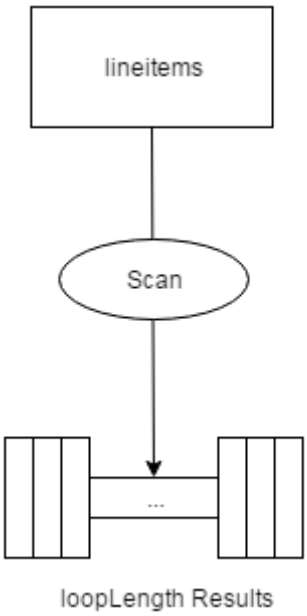
Συγκεκριμένα, αφού το πρόβλημα ήταν με την καθυστέρηση που χρειαζόταν για να γίνει το άθροισμα των πολλαπλασιασμών του κάθε κύκλου και του πολυπλέκτη, έγινε προσπάθεια μείωσης των σταδίων διασωλήνωσής τους.

```
1 optimization.pushPipeliningFactor(0);
2 DFEVar csum=sum+(control===0?0:lineitem[0]*lineitem[2]);
3 optimization.popPipeliningFactor();
```

Χρησιμοποιήθηκε η εντολή pushPipeliningFactor, η οποία παίρνει ως παράμετρο από 0-1 το βαθμό που θα γίνει επιδίωξη διασωλήνωσης. Δηλαδή στην περίπτωσή μας, θέλουμε να μειωθεί όσο το δυνατό περισσότερο. Έτσι, στην συνέχεια το αποτέλεσμα μπορεί να χρησιμοποιηθεί χωρίς κάποια δεδομένη καθυστέρηση, αλλά με μια πιθανή μείωση της συχνότητας, λόγω της μείωσης του βαθμού διασωλήνωσης.

4.3.4 Φάση Γ (Cumulative)

Λαμβάνοντας υπόψη την πιθανή μείωση που μπορεί να προκύψει με την μείωση των σταδίων διασωλήνωσης, έγινε η τελευταία εκδοχή επίλυσης του ερωτήματος με σκοπό την αποφυγή της τροποποίησης της διασωλήνωσης.



Σχήμα 4.8: Διάγραμμα ροής Ερωτήματος 6, Φάση Γ

Σε αυτή την προσέγγιση, αρχικά βρέθηκε ο αριθμός κύκλων όπου υπήρχε καθυστέρηση λόγω της εξάρτησης. Έπειτα, αντί ο κάθε κύκλος να περιμένει να ολοκληρωθεί το προηγούμενο αποτέλεσμα, συνεχίζει και παράγει το δικό του. Με το που φτάνει ο κύκλος k, για το οποίο ισχύει $\boldsymbol{k \bmod looplength = 0}$ τότε χρησιμοποιείται το αποτέλεσμα που παρήγαγε ο κύκλος k-looplength, το οποίο αφού πέρασαν οι looplength κύκλοι θα είναι έτοιμο.

```
carriedSum<==stream.offset(csum, -13);
```

Στην προκειμένη περίπτωση, ο αριθμός του looplength βρέθηκε να είναι ίσος με 13. Άρα στο CPU επιστρέφονται 13 συσσωρευτικά αθροίσματα τα οποία αθροίζονται για να παραχθεί το τελικό αποτέλεσμα.

4.4 Shipping Modes and Order Priority Query (Ερώτημα 12)

4.4.1 Περιγραφή

Σε αυτό το ερώτημα γίνεται προσπάθεια να απαντηθεί η ερώτηση του κατά πόσον η επιλογή φθηνότερου τρόπου αποστολής παραγγελίας επηρεάζει αρνητικά τις παραγγελίες ψηλής προτεραιότητας, προκαλώντας έτσι εκπρόθεσμες παραδόσεις.

Αναλυτικά, το ερώτημα μετρά βάσει του τρόπου αποστολής παραγγελίας, για lineitems τα οποία έχουν παραδοθεί σε συγκεκριμένο έτος, τον αριθμό των lineitems που ανήκουν σε παραγγελία η οποία παραδόθηκε εκπρόθεσμα για δύο καθορισμένα ship modes. Για το ερώτημα, μόνο τα lineitems τα οποία είχαν αποσταλεί πριν το commit_date λαμβάνονται υπόψη.

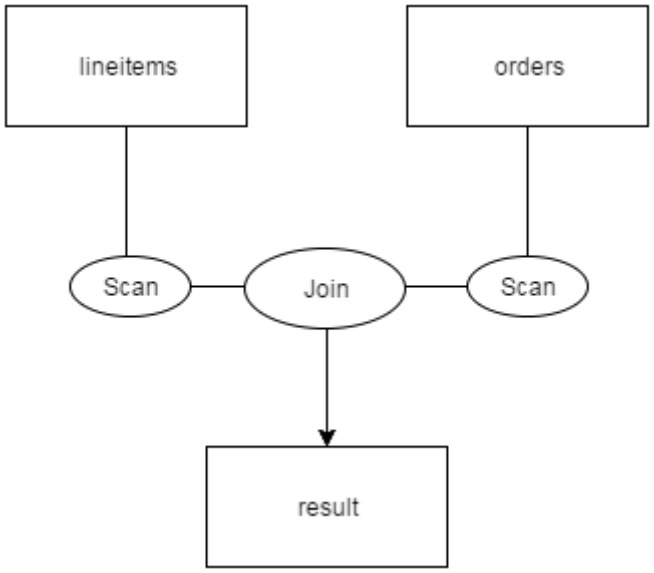
```
select
    l_shipmode,
    sum(case
            when o_orderpriority ='1-URGENT'
                or o_orderpriority ='2-HIGH'
            then 1
            else 0
    end) as high_line_count,
    sum(case
            when o_orderpriority <> '1-URGENT'
                and o_orderpriority <> '2-HIGH'
            then 1
            else 0
    end) as low_line_count
from
    orders,
    lineitem
where
    o_orderkey = l_orderkey
    and l_shipmode in ('[SHIPMODE1]', '[SHIPMODE2]')
    and l_commitdate < l_receiptdate
    and l_shipdate < l_commitdate
    and l_receiptdate >= date '[DATE]'
    and l_receiptdate < date '[DATE]' + interval '1' year
group by
    l_shipmode
order by
    l_shipmode;
```

4.4.2 Φάση Α (Nested Loop Join)

Η πρώτη προσέγγιση για το πρόβλημα ήταν η χρήση φωλιασμένων for για τον υπολογισμό των ζητουμένων του προβλήματος. Οι εγγραφές του πίνακα orders και lineitems ελέγχονται όλες μεταξύ τους για να βρεθούν αυτές που ικανοποιούν τα κριτήρια.

Κώδικας CPU:

```
for(i=0;i<ORDERS;++i){
    for(j=0;j<LINEITEM;++j){
        if((lineitem[j][0]==orders[i][0]) // order key evaluation
            &&((lineitem[j][1]==1)||(lineitem[j][1]==2)) // shipmode evaluation
            &&(lineitem[j][2]<lineitem[j][4]) // commit date evaluation
            &&(lineitem[j][3]<lineitem[j][2]) // ship date evaluation
            &&(lineitem[j][4]>=1994)  // receipt date evaluation 1
            &&(lineitem[j][4]<1995)){ // receipt date evaluation 2
                sum=sum+1;
        }
    }
}
```



Σχήμα 4.9: Διάγραμμα ροής Ερωτήματος 12, Φάση Α

Για την μετατροπή του πιο πάνω κώδικα στην πλατφόρμα της Maxeler χρησιμοποιήθηκε η τεχνική με τους μετρητές που επεξηγήθηκε στην υλοποίηση της Φάση Α του Ερωτήματος 3 στο υποκεφάλαιο 4.2.2.

4.4.3 Φάση B (Hash Join)

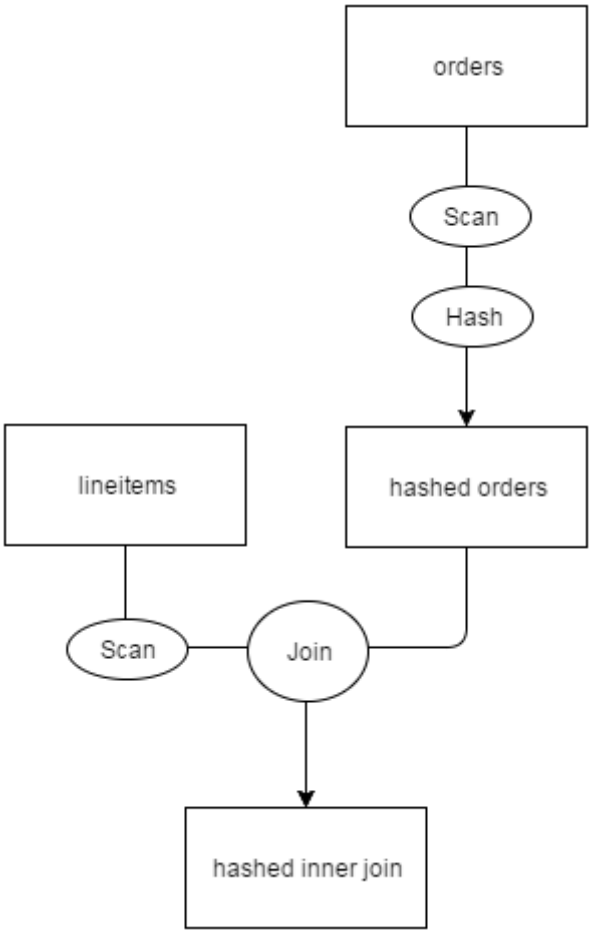
Τέλος, έγινε η υλοποίηση και με την χρήση πίνακα κατακερματισμού. Για τον πίνακα κατακερματισμού χρησιμοποιήθηκε ο πίνακας orders, ο οποίος έσπασε σε 512 partitions το κάθε ένα εκ των οποίων αποτελείτο από 3072 εγγραφές. Ως συνάρτηση κατακερματισμού επιλέχθηκε η επιστροφή των τελευταίων 10 bit του κωδικού παραγγελίας. Έτσι χρησιμοποιείται ο πίνακας αυτός για να γίνει η ένωση στην συνέχεια με τον πίνακα lineitems.

Για την υλοποίηση χρησιμοποιήθηκε η τρίτη τεχνική που αναφέρθηκε στο κεφάλαιο 4.2.4.

Κώδικας CPU:

```
for(i=0;i<ORDERS;++i){
    hi=((int)orders[i][0])%512;
    orHash[hi*3072+orHashIndex[hi]]=orders[i][0];
    orHashIndex[hi]++;
}

for(i=0;i<LINEITEM;++i){
    curr_ord=lineitem[i][0];
    if((lineitem[i][4]>=1994)&&(lineitem[i][4]<1995)){
        hi=((int)curr_ord)%512;
        for(j=0;j<orHashIndex[hi];++j){
            if(curr_ord==orHash[hi*3072+j]){
                sum=sum+1;
            }
        }
    }
}
```



Σχήμα 4.10: Διάγραμμα ροής Ερωτήματος 12, Φάση Β

Αρχικά ο πυρήνας χωρίζεται σε δύο νοητά στάδια. Το πρώτο στάδιο, στο οποίο θα γίνει ο κατακερματισμός του πίνακα orders και το δεύτερο στάδιο, στο οποίο θα γίνει η ένωση με τον πίνακα lineitems. Ο λόγος που γίνεται αυτό είναι επειδή η κατάσταση της εσωτερικής μνήμης δεν αποθηκεύεται από εκτέλεση σε εκτέλεση. Αυτό ο περιορισμός θα μπορούσε να παρακαμφθεί με την χρήση της εξωτερικής μνήμης, της οποίας η κατάσταση αποθηκεύεται, αλλά πάλι θα υπήρχε η φάση αποθήκευσης/φόρτωσης από αυτή.

Έτσι λοιπόν, οι λειτουργίες που αφορούν την δεύτερη φάση είναι απενεργοποιημένες όταν υπάρχει η ένδειξη ότι ο πυρήνας είναι ακόμη στην πρώτη και μετά το αντίστροφο.

Το στάδιο στο οποίο βρίσκεται ο πυρήνας υπολογίζεται βάσει της κατάστασης που βρίσκεται ο πίνακας orders. Γίνεται χρήση ενός μετρητή, η τιμή του οποίου όταν ξεπεράσει το μέγεθος του πίνακα orders δεικνύει ότι ο πυρήνας έχει φθάσει στην δεύτερη φάση.

```
DFEVar firstStage = globalCounter<orderSize;
```

Στην πρώτη φάση βασικά γίνεται ο υπολογισμός της θέσης της εγγραφής στον πίνακα κατακερματισμού και μετά εισαγωγή της.

```
1   DFEVar orderPartitionIndex = order.cast(dfeUInt(32)).cast(dfeUInt(9));
2   DFEVar orderCurrentPartitionSize=hashtable.read(orderPartitionIndex);
3   DFEVar orderRamFirstPos = orderPartitionIndex.cast(dfeUInt(32)) *
partitionSize;
4   DFEVar orderRamLatestPos = orderRamFirstPos +
orderCurrentPartitionSize.cast(utype);
5   ram.write(orderRamLatestPos.cast(dfeUInt(21)), order, firstStage &
orLoopCounter===0);
6   hashtable.write(orderPartitionIndex,
stream.offset(orderCurrentPartitionSize, -(orLoopLength-2))+1, firstStage &
(orLoopCounter===orLoopLengthVal-2));
```

Αρχικά γίνεται κατακερματισμός του κλειδιού, μετατρέποντάς τον πρώτα σε ακέραιο και μετά κάνοντάς τον cast σε αριθμό 9 bit έτσι ώστε να γίνει απομόνωση των τελευταίων 9 bit σχετικά ανέξοδα (μετά την μετατροπή). Στην συνέχεια, στην δεύτερη γραμμή γίνεται διάβασμα του αριθμού εγγραφών οι οποίοι έχουν ήδη εγγραφεί στο partition και στις γραμμές 3 και 4 γίνεται υπολογισμός της θέσης στην οποία θα εγγραφούν τα δεδομένα, όπου και εγγράφονται στην γραμμή 5.

Στην γραμμή 6 ανανεώνεται ο πίνακας ο οποίος κρατά τις θέσεις των partition. Στο κομμάτι αυτό σαν δεύτερη παράμετρος πρέπει να μπει η τιμή του orderCurrentPartitionSize+1. Στην περίπτωση όμως που στον επόμενο κύκλο γίνει προσπέλαση της μνήμης στην θέση την οποία έγραφε ο προηγούμενος κύκλος, θα επιστραφεί η παλιά τιμή. Για αυτό το λόγο γίνεται αναγκαστικά η δημιουργία ενός loop (Το μέγεθος του loop μπορεί να το δει κανείς μετά την εκτέλεση του προγράμματος, πριν αυτή πρέπει να μπει μια μεταβλητή η οποία θα αντικατασταθεί από τον μεταγλωττιστή αυτόματα για να μην υπάρξει loop). Η εγγραφή γίνεται αν ο πυρήνας βρίσκεται στο πρώτο στάδιο και ο μετρητής του loop 2 θέσεις πριν το τέλος (για να προλάβει να εγγραφεί και να είναι έτοιμο για ανάγνωση από τον επόμενο κύκλο).

Στο δεύτερο στάδιο, υπολογίζεται σε ποιο partition πιθανώς να βρίσκεται η εγγραφή του lineitems και διατρέχεται ολόκληρο. Οι εγγραφές οι οποίες ταυτίστηκαν αποθηκεύονται με την χρήση ενός μετρητή που ενεργοποιείται μόνο σε αυτή την περίπτωση.

```
1   Count.Params paramsliiter = control.count.makeParams(32)
2       .withEnable(~firstStage)
3       .withMax(partitionSize);
4   Counter counterliPartitionIteration =
control.count.makeCounter(paramsliPartitionIteration);
5   DFEVar liPartitionIteration = counterliPartitionIteration.getCount();
6
7   DFEVar validlineitem = lineitem[1]>=1994 & lineitem[1]<1995;
8
9   DFEVar liPartitionIndex = lineitem[0].cast(dfeUInt(32)).cast(dfeUInt(9));
10  DFEVar liCurrentPartitionSize=hashtable.read(liPartitionIndex);
11  DFEVar liRamFirstPos = liPartitionIndex.cast(dfeUInt(32)) *
partitionSize;
12  DFEVar currentRamPos = liRamFirstPos + liPartitionIteration;
13  DFEVar v=ram.read(currentRamPos.cast(dfeUInt(21)));
14
15  DFEVar equal=v===lineitem[0]?constant.var(true):0;
16  Count.Params paramstotal = control.count.makeParams(32)
17      .withEnable(~firstStage & equal & validlineitem);
18  Counter countertotal = control.count.makeCounter(paramstotal);
19  DFEVar total = countertotal.getCount();
```

Στις γραμμές 1-5 γίνεται αρχικοποίηση του μετρητή βάσει του οποίου θα διατρεχτεί το partition. Ακολούθως υπολογίζεται αν η εγγραφή τηρεί κάποιους περιορισμούς (γραμμή 7) και στην συνέχεια υπολογίζεται το partition και η θέση σε αυτό και την διαβάζει (γραμμή 9-13). Έπειτα γίνεται έλεγχος αν τα πεδία ταυτίζονται (γραμμή 15) και βάσει αυτού και του αν ο πυρήνας δεν βρίσκεται στο πρώτο στάδιο, τότε ενεργοποιείται ο μετρητής που κρατά το σύνολο των ταυτίσεων. Στο τέλος επιστρέφεται η τιμή του total.

4.5 Ανασκόπηση

Σε αυτό το κεφάλαιο μελετήθηκε η μετατροπή διάφορων υλοποιήσεων από CPU στην πλατφόρμα της Maxeler.

Ο γενικός τρόπος που ακολουθήθηκε για την μεταφορά των προγραμμάτων στην πλατφόρμα της Maxeler ήταν η μετατροπή των προγραμμάτων της CPU σε εκδόσεις οι οποίες επιτελούν την ίδια λειτουργία με τις αρχικές τους εκδόσεις, αλλά οι οποίες ταυτόχρονα προσφέρουν άμεση μεταφερσιμότητα στην πλατφόρμα της Maxeler.

Κύριο χαρακτηριστικό της διαδικασίας αυτής ήταν η απαλοιφή των εντολών ελέγχου ροής. Έπειτα, γινόταν σχεδιασμός του Manager και του Kernel για την συγκεκριμένη λειτουργία κάνοντας ταυτόχρονα επίλυση διάφορων δυσκολιών που προκύπτουν από την χρήση των συστημάτων ροής δεδομένων της Maxeler.

Κατά τον σχεδιασμό του Manager, η μεγαλύτερη πρόκληση ήταν η ρύθμιση των ροών των διαφόρων προγραμμάτων. Απαιτείται μεγάλη ακρίβεια και η αποσφαλμάτωση σε αυτό το κομμάτι είναι μια πολύ χρονοβόρα διαδικασία. Σημειώνεται ότι μέχρι και τα εργαλεία αποσφαλμάτωσης σε διάφορες περιπτώσεις, κυρίως σε περιπτώσεις χρήσης της εξωτερικής

μνήμης, αντιμετώπιζαν προβλήματα και έκλειναν χωρίς προειδοποίηση κατά την ώρα της αποσφαλμάτωσης.

Κατά τον σχεδιασμό του Kernel, μια από τις πρώτες δυσκολίες που παρουσιάστηκαν ήταν η επίλυση των κυκλικών εξαρτήσεων όπως φαίνεται στο Ερώτημα 6.

Η επόμενη πρόκληση ήταν να γίνεται ανάγνωση πολλαπλών στοιχείων ανά κύκλο. Ο αρχικός τρόπος λειτουργίας, ήταν για ένα πίνακα με k στήλες, να χρησιμοποιούνται k κύκλοι για να διαβαστούν η κάθε μια από αυτές, χρησιμοποιώντας ένα buffer για την αποθήκευσή τους. Με τον τρόπο αυτό, εκτός του ότι γινόταν περαιτέρω χρήση πόρων, εισήχθηκε και μια καθυστέρηση k κύκλων ανά επανάληψη. Η λύση του προβλήματος αυτού περιγράφεται στο δεύτερο παράδειγμα στο Κεφάλαιο 2, κάτι το οποίο δεν περιγραφόταν στα εγχειρίδια της πλατφόρμας. Με την λύση αυτού έγινε επιτευκτή και η εισαγωγή επεξεργασίας πολλαπλών εγγραφών ανά κύκλο.

Προκλήσεις αντιμετωπίστηκαν και λόγω του σχεδιασμού των διάφορων διεπαφών της μηχανής ροής.

Μια από τις μεγαλύτερες προκλήσεις ήταν το γεγονός ότι για την χρήση της εξωτερικής μνήμης όλες οι παράμετροι πρέπει να είναι πολλαπλάσια κάποιου `burst size`, για την αποδοτικότητα της μνήμης, το οποίο στις υλοποιήσεις που μελετήθηκαν είναι 384.

Το γεγονός αυτό επηρεάζει και τον Manager και τον Kernel αφού στον Manager τα μεγέθη που θα εγγραφούν στην μνήμη πρέπει να οριστούν εκ των προτέρων και στον Kernel πρέπει να γίνει σίγουρο ότι στις ροές από και προς την εξωτερική μνήμη, η ανάγνωση και η εγγραφή είναι σε πολλαπλάσια των `burst size` bytes.

Ιδιαίτερη πρόκληση κατέληξε να είναι και η χρήση της εσωτερικής μνήμης της μηχανής ροής όπως φάνηκε και από τις υλοποιήσεις του Hash Join. Το γεγονός ότι η εσωτερική μνήμη δεν μπορεί να χρησιμοποιηθεί από εκτελέσεις σε διαφορετικούς χρόνους, κατέστησε αναγκαίο τον νοητό διαχωρισμό των υλοποιήσεων σε φάσεις με περιπλοκές σε Manager και Kernel.

Επιπλέον παρατηρήθηκε το ότι σε ορισμένες περιπτώσεις, η ανανέωση των τιμών στην εσωτερική μνήμη χρειαζόταν δύο κύκλους για να ολοκληρωθεί, παρότι ήταν αναμενόμενο να ολοκληρωθεί σε ένα. Αναγκαστικά έγινε εισαγωγή καθυστέρησης όπως φάνηκε και στην υλοποίηση για την αποφυγή προβλημάτων που πιθανώς να προέκυπταν.

Τέλος, δεν θα μπορούσε να παραληφθεί το γεγονός ότι ο χρόνος μεταγλώττισης (στην προσομοίωση) ήταν της τάξεως των λεπτών, κυρίως για τις σύνθετες υλοποιήσεις, γεγονός που κυρίως στα στάδια της αποσφαλμάτωσης ήταν μια μεγάλη επιβάρυνση, αφού συχνά έπρεπε να γίνουν αρκετές αλλαγές μέχρι να επιλυθεί το πρόβλημα.

Κεφάλαιο 5

Πειραματικές Μετρήσεις και Αποτελέσματα

5.1 Μεθοδολογία

Σε αυτό το κεφάλαιο θα γίνει παρουσίαση των αποτελεσμάτων, χρησιμοποιώντας τις επιδόσεις των υλοποιήσεων, για την CPU και την πλατφόρμα της Maxeler.

Για την λήψη των μετρήσεων στην CPU, μετρήθηκαν οι χρόνοι ολοκλήρωσης των προγραμμάτων πολλαπλές φορές, παίρνοντας σαν μέτρηση τον μέσο όρο αυτών. Για τις μετρήσεις χρησιμοποιήθηκε μηχανή με επεξεργαστή Intel(R) Core(TM) i5-4590 CPU @ 3.30GHz και με 4GB Ram.

Οι μετρήσεις για την πλατφόρμα της Maxeler, λόγω του τρόπου λειτουργίας της, έγιναν βάσει υπολογισμού. Κλειδί σε αυτό, είναι το γεγονός ότι τα προγράμματα για την μηχανή

ροής δεδομένων τρέχουν για προκαθορισμένο αριθμό κύκλων. Αυτό, σε συνδυασμό με το γεγονός ότι η συχνότητα επεξεργασίας της διασωλήνωσης όπως και της μνήμης, ως επί το πλείστον, είναι γνωστή εκ των προτέρων μας δίδει την δυνατότητα να υπολογίσουμε την επίδοση.

Για τον υπολογισμό της επίδοσης του προγράμματος, λαμβάνονται υπόψη δύο παράγοντες:

- Η ταχύτητα διεκπεραίωσης της διασωλήνωσης (Frequency)
- Το εύρος ζώνης από και προς τον γράφο δεδομένων (Bandwidth)

Έχοντας υπόψη ότι φάσεις της μνήμης μπορούν να συμπεριληφθούν ως στάδια της διασωλήνωσης, ο χρόνος ολοκλήρωσης ενός προγράμματος θα είναι:

$$T = Max(\frac{Cycles}{Frequency}, \frac{Data}{Bandwidth})$$

Όπου Cycles ο αριθμός των κύκλων που θα τρέξει η μηχανή ροής και Data το μέγεθος των δεδομένων που πρέπει να τύχουν επεξεργασίας.

Πληροφορίες οι οποίες χρησιμοποιήθηκαν για την πλατφόρμα της Maxeler:

	PCI-E	LMem	FMem	Kernel Frequency
Speed	2 GB/s	38 GB/s	12.6TB/s	250 MHz

Πίνακας 5.1: Τεχνικά χαρακτηριστικά της μηχανής ροής δεδομένων

Πληροφορίες σχετικά με τους πίνακες της βάσης:

	Small	Medium	Large
Lineitems	60175	600572	6001215
Orders	15000	150000	1500000
Customers	1500	15000	150000

Πίνακας 5.2: Μεγέθη των διάφορων πινάκων.

Επιπλέον, επισημαίνεται ότι με την χρήση των όρων `li`, `or`, `cu` γίνεται αναφορά στο πλήθος των εγγραφών των αντίστοιχων πινάκων, ενώ με την χρήση των όρων `liSize`, `orSize`, `cuSize` γίνεται αναφορά στο μέγεθος του πίνακα σε bytes. Το μέγεθος αυτό για τον πίνακα lineitems, παραδείγματος χάρη, υπολογίζεται ως εξής:

$$liSize = li * usedColumns * columnsSize$$

όπου το usedColumns είναι ο αριθμός των στηλών του πίνακα που χρησιμοποιούνται και το columnSize είναι το μέγεθος της κάθε στήλης σε bytes, όπου στην προκειμένη περίπτωση είναι όλα μεγέθους 4.

Ο αριθμός παραλληλισμού που χρησιμοποιείται σε κάποια ερωτήματα, είναι τέτοιος έτσι ώστε οι υλοποιήσεις να είναι συμβατές με την μηχανή ροής δεδομένων GALAVA.

Τέλος επισημαίνεται ότι για τις υλοποιήσεις στο CPU δεν γίνεται οποιαδήποτε χρήση παραλληλισμού αφού οποιοδήποτε κέρδος μπορεί να έχει το CPU από τον παραλληλισμό μπορεί να το επιτύχει με προσαρμογή και η μηχανή ροής δεδομένων, αφού ο Host της τρέχει στο CPU.
Παραδείγματος χάρη, έστω ότι γίνεται χρήση k threads από το CPU, ο Host μπορεί να χρησιμοποιήσει και αυτός k threads ορίζοντας σε κάποιο thread μεγαλύτερο φόρτο εργασίας, έτσι ώστε να είναι υπεύθυνο να τρέξει την μηχανή ροής δεδομένων για τον συγκεκριμένο φόρτο. Σημειώνεται επίσης ότι το κάλεσμα των λειτουργιών στην μηχανή ροής μπορεί να είναι και non-blocking. Έτσι επιτυγχάνεται ανάλογη βελτίωση και στην μηχανή ροής δεδομένων.

5.2 Shipping Priority Query (Ερώτημα 3)

5.2.1 Αποτελέσματα Φάσης Α (Nested Loop Join)

Για την μέτρηση των χρόνων της μηχανής ροής, το ερώτημα χωρίστηκε σε δύο φάσεις.
Κατά την πρώτη έγινε η μεταφορά των πινάκων customers και orders στην εξωτερική μνήμη. Στην δεύτερη, έγινε μεταφορά των δύο πινάκων από την εξωτερική μνήμη στον πυρήνα, η μεταφορά του πίνακα lineitems από την CPU στον πυρήνα και η εκτέλεση του πυρήνα.

	Φάση I	Φάση II
CPU	$\frac{orSize + cuSize}{PCIE}$	$\frac{liSize}{PCIE}$
Memory	$\frac{orSize + cuSize}{LMem}$	$\frac{orSize * li + cuSize * or * li}{LMem}$
Ticks	0	$\frac{li * or * cu}{Frequency}$

Πίνακας 5.3: Φάσεις εκτέλεσης ερωτήματος 3 Α

Ο συνολικός χρόνος εκτέλεσης Τ είναι:
$T = Max(Φαση\ I) + Max(Φαση\ II)$

Μετά τον υπολογισμό των χρόνων της κάθε φάσης, παρατηρήθηκε ότι το εύρος του δικτύου και της μνήμης επέτρεπε την μεταφορά περισσότερων από μιας εγγραφής ανά κύκλο χωρίς περαιτέρω επιβράδυνση. Συγκεκριμένα, παρατηρήθηκε ότι μπορεί να γίνεται αποστολή μέχρι και 300000 εγγραφών του lineitems μειώνοντας την διαφορά των χρόνων της κάθε λειτουργίας στο ελάχιστο.

	Φάση I	Φάση II
CPU	$\frac{orSize + cuSize}{PCIE}$	$\frac{300000 * liSize}{PCIE}$
Memory	$\frac{orSize + cuSize}{LMem}$	$\frac{orSize * \frac{li}{300000} + cuSize * or * \frac{li}{300000}}{LMem}$
Ticks	0	$\frac{\frac{li}{300000} * or * cu}{Frequency}$

Πίνακας 5.4: Φάσεις εκτέλεσης ερωτήματος 3 Α με παραλληλισμό

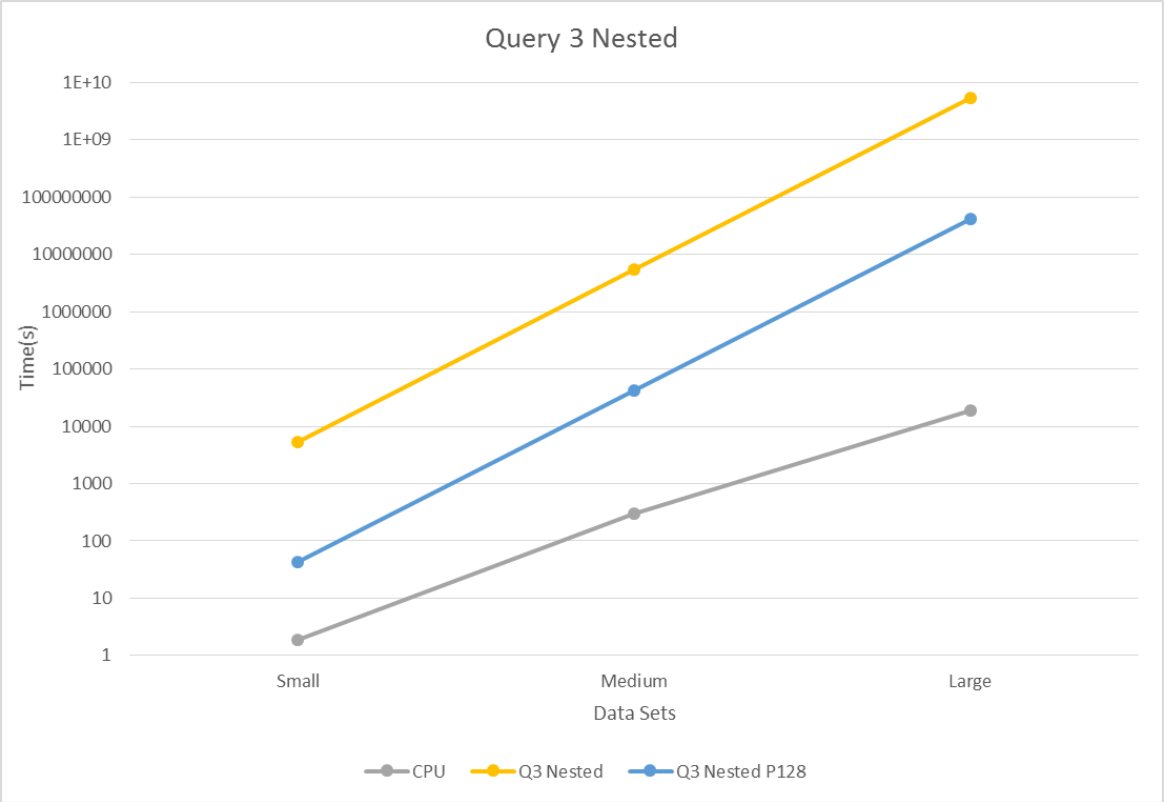
Για τον πίνακα πιο πάνω ισχύει ότι:

$T_{CPU} \approx T_{Ticks}$

Εντούτοις, υπάρχει περιορισμός στο υλικό το οποίο μπορεί να χρησιμοποιηθεί και έτσι για τις μετρήσεις χρησιμοποιείται ο αριθμός επεξεργασίας 128 εγγραφών ταυτόχρονα, αριθμός ο οποίος επιβεβαιώθηκε ότι υποστηρίζεται από τουλάχιστον μια από τις μηχανές ροής δεδομένων.

	Small	Medium	Large
CPU	1.9	300	18700
Q3 Nested	5415.75	5405148	5401093500
Q3 Nested P128	42.31	42227.72	42196042.97

Πίνακας 5.5: Χρόνοι εκτέλεσης ερωτήματος 3 Α



Σχήμα 5.1: Γράφημα Ερωτήματος 3 Φάση Α

Παρατηρείται ότι ο χρόνος του CPU είναι πολύ καλύτερος από αυτό του Maxeler. Αυτό οφείλεται στο ότι η CPU τρέχει κατά πολύ λιγότερους κύκλους από την μηχανή ροής δεδομένων, αφού στο CPU με την χρήση των εντολών ελέγχου ροής υπάρχει η δυνατότητα να διακοπεί η αναζήτηση σε περίπτωση που βρεθεί μία ταύτιση, σε αντίθεση με την μηχανή ροής που τρέχει για το σύνολο όλων των εγγραφών των πινάκων.

Ο λόγο για τον οποίο δεν υπάρχει υποστήριξη των εντολών ελέγχου ροής βασίζεται στο γεγονός ότι τα προγράμματα στην μηχανή ροής δεδομένων είναι ουσιαστικά γράφοι στους οποίους η ροή των δεδομένων πρέπει να είναι προκαθορισμένη, πράγμα το οποίο έρχεται σε αντίθεση με την ιδέα πίσω από την χρήση των εντολών αυτών. Κατά κύριο λόγο, η μηχανή ροής δεδομένων τρέχει για προκαθορισμένο αριθμό κύκλων και ο αριθμός αυτών υπολογίζεται βάσει της χειρότερης περίπτωσης.

Ακόμη και η πρόοδος με την επεξεργασία 128 εγγραφών του lineitems ανά κύκλο δεν ήταν αρκετή για την επικάλυψη του μειονεκτήματος που υπήρχε από τις εντολές ελέγχου ροής και την πολυπλοκότητα που δημιουργείται με τα φωλιασμένα for.

5.2.2 Αποτελέσματα Φάσης Β (Separate Join)

Για τον υπολογισμό του χρόνου εκτέλεσης της υλοποίησης με το Separate Join το πρόβλημα χωρίστηκε σε τρεις φάσεις.

Στην πρώτη έγινε η μεταφορά του πίνακα customers στην εξωτερική μνήμη. Μετά ακολούθησε μεταφορά του πίνακα customers από την μνήμη στον πυρήνα και του πίνακα orders από το CPU στον πυρήνα. Έπειτα έγινε ο υπολογισμός της ένωσης και αποθήκευση του αποτελέσματος στην μνήμη. Στην τρίτη φάση έγινε μεταφορά της ένωσης από την μνήμη, του πίνακα lineitems από το CPU και ο υπολογισμός του αποτελέσματος.

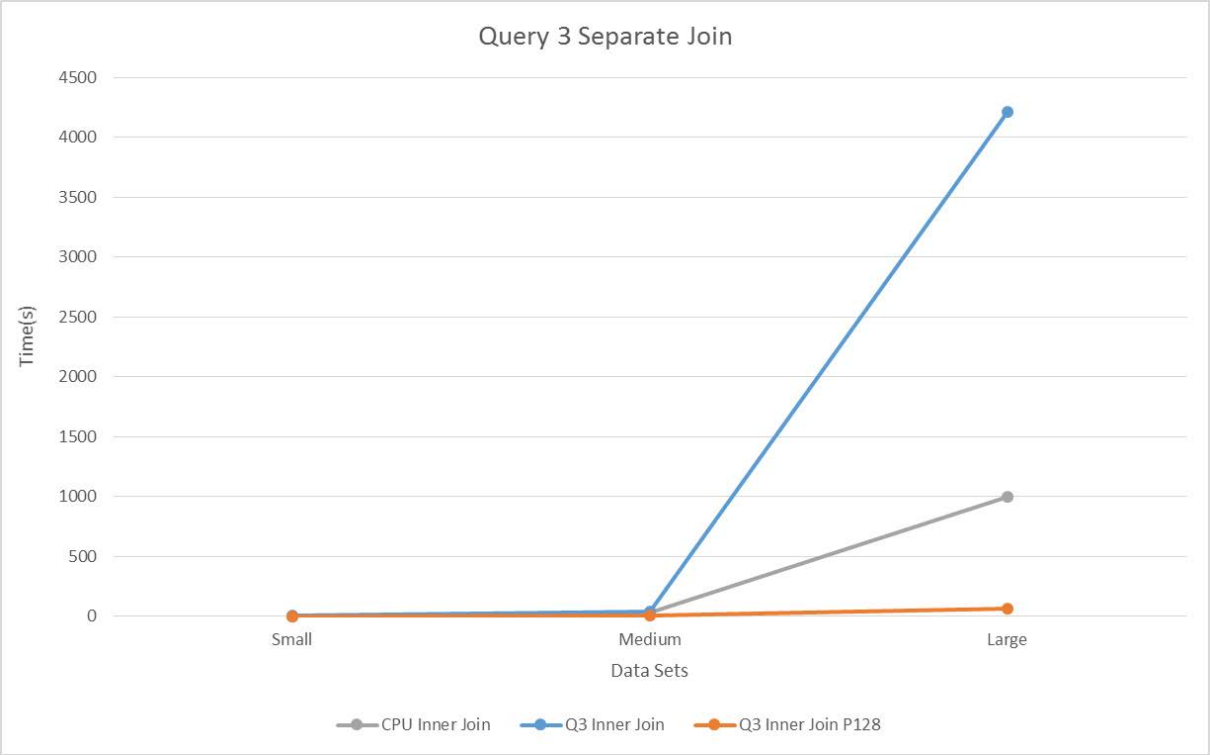
	Φάση I	Φάση II	Φάση III
CPU	$\frac{cuSize}{PCIE}$	$\frac{orSize}{PCIE}$	$\frac{liSize}{PCIE}$
Memory	$\frac{cuSize}{LMem}$	$\frac{cuSize * or + orSize}{LMem}$	$\frac{ijSize * li}{LMem}$
Ticks	0	$\frac{or * cu + or}{Frequency}$	$\frac{li * ij}{Frequency}$

Πίνακας 5.6: Φάσεις εκτέλεσης ερωτήματος 3 Β

Παρόμοια με την πρώτη υλοποίηση, υπήρχε αρκετός χώρος για βελτίωση με τον περιορισμό του υλικού. Σε αυτή την περίπτωση επιλέγηκε η χρήση παραλληλισμού στις δύο τελευταίες φάσεις. Συγκεκριμένα, χρησιμοποιήθηκε επεξεργασία 47 εγγραφών του πίνακα orders στην Φάση ΙΙ και 81 εγγραφές του πίνακα lineitems στην Φάση ΙΙΙ. Ο χρόνος της φάσης ΙΙΙ ήταν ο τριπλάσιος της φάσης ΙΙ, έτσι για τον καταμερισμό των αριθμών υπολογίστηκε το θετικό ελάχιστο της εξίσωσης: $T = \frac{1}{k} + \frac{3}{128-k}$

Time(s)	Small	Medium	Large
CPU Inner Join	0.058	30	1000
Q3 Inner Join	0.4	40.23	4207.98
Q3 Inner Join P128	0.003	0.55	59.21

Πίνακας 5.7: Χρόνοι εκτέλεσης ερωτήματος 3 Β



Σχήμα 5.2: Γράφημα Ερωτήματος 3 Φάση Β

Πλέον, αφού το πρόβλημα μειώθηκε σε πολυπλοκότητα χρόνου, η διαφορά δυναμικού που παρατηρήθηκε στην προηγούμενη υλοποίηση έχει μειωθεί.

Παρότι η υλοποίηση χωρίς παραλληλισμό φαίνεται να επηρεάζεται ακόμη από την δυσκολία μετάφρασης των εντολών ελέγχου ροής, παρατηρείται ότι στην αναβαθμισμένη έκδοση με παραλληλισμό, η μηχανή της Maxeler ξεπέρασε σε σημαντικό βαθμό την υλοποίηση του CPU.

5.2.3 Αποτελέσματα Φάσης Γ (Hash Join)

Για τον υπολογισμό του χρόνου εκτέλεσης του Hash Join, έγινε διαμοιρασμός επίσης σε τρία στάδια. Αρχικά γίνεται υπολογισμός του πίνακα κατακερματισμού του customers και αποθήκευση στην εσωτερική μνήμη. Έπειτα γίνεται ένωση του πίνακα orders με τον πίνακα customers που υπολογίστηκε και αποθήκευση στην εσωτερική μνήμη. Τέλος, γίνεται ένωση του πίνακα που υπολογίστηκε στην προηγούμενη φάση με τον πίνακα lineitems.

	Φάση I	Φάση II	Φάση III
CPU	$\frac{cuSize}{PCIE}$	$\frac{orSize}{PCIE}$	$\frac{liSize}{PCIE}$
Memory	$\frac{cuSize}{FMem}$	$\frac{orSize * partitionASize + ijSize}{FMem}$	$\frac{liSize * partitionBSize}{FMem}$
Ticks	$\frac{cu * loopA}{Frequency}$	$\frac{or * (partitionSize + loopB)}{Frequency}$	$\frac{li * ij + li}{Frequency}$

Πίνακας 5.8: Φάσεις εκτέλεσης ερωτήματος 3 Γ

Τα περιθώρια χρήσης παραλληλισμού σε αυτό το ερώτημα ήταν περιορισμένα λόγω του ότι η μεγάλη χρήση της εσωτερικής μνήμης από την αρχική υλοποίηση δεν άφησε περιθώρια για περεταίρω επέκταση. Σε μια εκ των δοκιμών που έγιναν παρατηρήθηκε ότι στην αρχική υλοποίηση έγινε χρήση του 43% των block memory.

Για τις λήψεις των μετρήσεων χρησιμοποιήθηκε παραλληλισμός 2 εγγραφών ανά κύκλο.

	Small	Medium	Large
CPU HJ	0.00058	4.78	20.47
Q3 HJ	0.76	7.61	76.04
Q3 HJ P2	0.39	3.92	39.17

Πίνακας 5.9: Χρόνοι εκτέλεσης ερωτήματος 3 Γ

Σχήμα 5.3: Γράφημα Ερωτήματος 3 Φάση Γ

Παρατηρείται πλέον ότι ο περιορισμός στον βαθμό του παραλληλισμού καθιστά την υλοποίηση της CPU σαφώς καλύτερη. Πλέον τα μειονεκτήματα που προκύπτουν από την δυσκολία χρήσης των εντολών ροής και το ότι η CPU έχει μεγαλύτερη συχνότητα δεν μπορούν να επικαλυφθούν από τον βαθμό παραλληλισμού που προσφέρεται.
Σημειώνεται βέβαια ότι η εικόνα θα ήταν διαφορετική αν γινόταν επιτευκτή η χρήση της εξωτερικής μνήμης, αφού ο περιορισμός για το υλικό δεν θα υπήρχε.

5.3 Forecasting Revenue Change Query (Ερώτημα 6)

5.3.1 Αποτελέσματα Φάσης Α (Loop)

Για τον υπολογισμό του χρόνου του προβλήματος, η διαδικασία χωρίζεται σε δύο φάσεις. Αρχικά γίνεται μεταφορά του πίνακα lineitems στην εξωτερική μνήμη του προγράμματος από το CPU. Τέλος, γίνεται η χρήση του αποθηκευμένου πίνακα για να υπολογιστούν οι παράμετροι του προβλήματος.

	Φάση I	Φάση II
CPU	$\frac{liSize}{PCIE}$	0
Memory	$\frac{liSize}{LMem}$	$\frac{liSize}{LMem}$
Ticks	0	$\frac{li * loop}{LMem}$

Πίνακας 5.10: Φάσεις εκτέλεσης ερωτήματος 6 Α

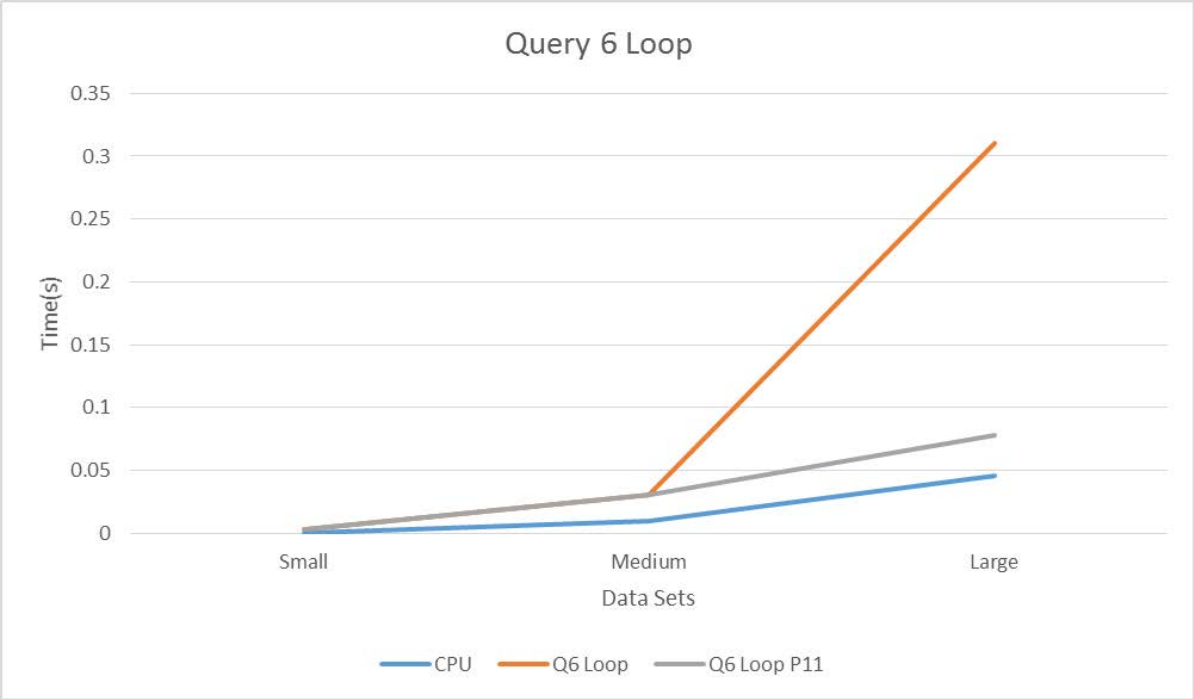
Βάσει των υπολογισμών, παρατηρήθηκε ότι παρότι ο χρόνος του προγράμματος είναι ήδη αρκετά χαμηλός, θα μπορούσε να μειωθεί περαιτέρω με την χρήση του εύρους της εξωτερικής μνήμης.
Για τον υπολογισμό του μεγέθους του παραλληλισμού χρησιμοποιήθηκε το:

$$\frac{liSize * k}{LMem} = \frac{\frac{li}{k} * loop}{LMem}$$

Από το οποίο βρέθηκε ότι το $k = 11$

	Small	Medium	Large
CPU	0.0004	0.01	0.046
Q6 Loop	0.003	0.03	0.31
Q6 Loop P11	0.003	0.03	0.078

Πίνακας 5.11 Χρόνοι εκτέλεσης ερωτήματος 6 Α



Σχήμα 5.4: Γράφημα Ερωτήματος 6 Φάση Α

Στην υλοποίηση αυτή παρατηρείται ότι παρόλο που στην υλοποίηση της CPU δεν υπάρχουν εντολές αλλαγής της ροής, η απλότητα του ερωτήματος από πλευράς κύκλων ανά επανάληψη από της πλευρά της CPU βάζει πάλι την μηχανή ροής δεδομένων σε μειονεκτική θέση αφού η CPU τρέχει σε πιο ψηλή συχνότητα.

Επιπλέον, το γεγονός ότι η εξάλειψη των εξαρτήσεων έγινε με την εισαγωγή καθυστέρησης επιβάρυνε ακόμη περισσότερο τους χρόνους της μηχανής ροής, αφού πλέον η διεκπεραιωτικότητα έπεσε στο 1/loopLength. Δεν έγινε κατορθωτή η πρόοδος ούτε με την χρήση παραλληλισμού και η αιτία αυτού πέρα από την απλότητα του ερωτήματος εξηγείται από το νόμο του Amdahl, αφού δεν ήταν δυνατός ο παραλληλισμός της πρώτης φάσης η οποία καταλάμβανε το 15% του χρόνου εκτέλεσης του προγράμματος.

5.3.2 Αποτελέσματα Φάσης Β και Γ

Για τον υπολογισμό του χρόνου του προβλήματος λαμβάνεται υπόψη ο χρόνος που χρειάζεται για την μεταφορά του πίνακα lineitems από το CPU στον πυρήνα και ο αριθμός κύκλων που χρειάζονται για να υπολογιστεί το αποτέλεσμα.

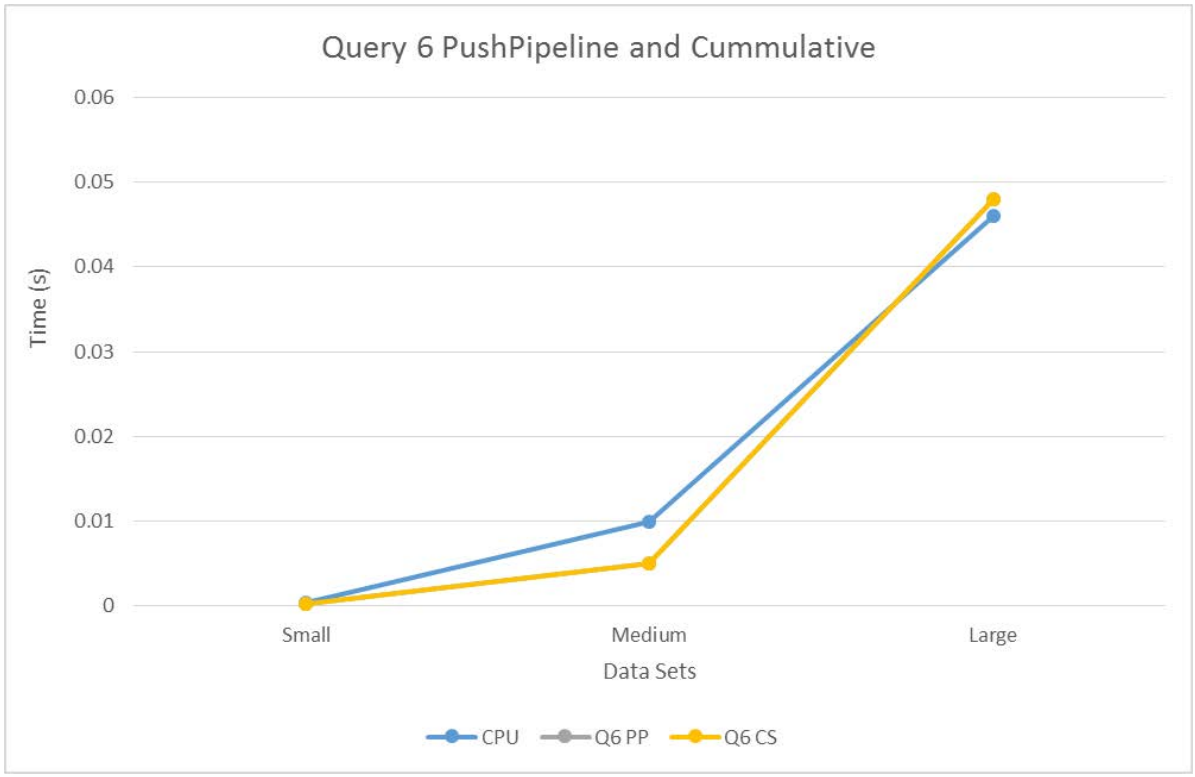
	Φάση I
CPU	$\frac{liSize}{PCIE}$
Memory	0
Ticks	$\frac{li}{Frequency}$

Πίνακας 5.12: Φάσεις εκτέλεσης ερωτήματος 6 Β,Γ

Στην υλοποίηση αυτή παρατηρήθηκε ότι δεν ήταν δυνατή η χρήση παραλληλισμού, αφού το σημείο συμφόρησης εξαρχής ήταν η μεταφορά των δεδομένων από το CPU στην μηχανή ροής.

	Small	Medium	Large
CPU	0.0004	0.01	0.046
Q6 PP	0.0003	0.005	0.048
Q6 CS	0.0002	0.005	0.048

Πίνακας 5.13: Χρόνοι εκτέλεσης ερωτήματος 6 Β,Γ



Σχήμα 5.5: Γράφημα Ερωτήματος 6 Φάση Β,Γ

Στην υλοποίηση με το PushPipeline, με την μείωση των σταδίων διασωλήνωσης έγινε προσπάθεια επιστροφής της διεκπεραιωτικότητας στο 1. Έτσι, έγινε κατορθωτό να μειωθεί σε μεγάλο βαθμό ο χρόνος T_{Ticks}.

Στην υλοποίηση με το Cumulative, έγινε επίσης μείωση του T_{Ticks} σε ακόμη μεγαλύτερο βαθμό από την υλοποίηση με το PushPipeline. Το γεγονός όμως ότι και στα δύο ο χρόνος $T_{Ticks} < T_{CPU}$, καθιστά τον χρόνο εκτέλεσης και των δύο ίσο με τον χρόνο μεταφοράς του πίνακα από την CPU στον πυρήνα.

Λαμβάνοντας υπόψη ότι το εύρος της cache του CPU μπορεί να φτάσει σε πιο ψηλές ταχύτητες από το εύρος του PCI-E, σε συνδυασμό με τις πιο ψηλές ταχύτητες της CPU, είναι λογικό που προηγείται στις πλείστες δοκιμές, παρόλο που η φύση του προβλήματος δεν αφήνει να εξαχθούν ξεκάθαρα συμπεράσματα λόγω της απλότητάς του.

5.4 Shipping Modes and Order Priority Query (Ερώτημα 12)

5.4.1 Αποτελέσματα Φάσης Α (Nested Loop Join)

Για τον υπολογισμό του χρόνου εκτέλεσης το πρόγραμμα χωρίζεται σε δύο φάσεις.
Αρχικά γίνεται η μεταφορά του πίνακα orders από το CPU στην εξωτερική μνήμη της μηχανής ροής. Στην συνέχεια γίνεται μεταφορά του πίνακα orders από την εξωτερική μνήμη στον πυρήνα, μεταφορά του πίνακα lineitems από την CPU στον πυρήνα και έπειτα εκτέλεση των κύκλων για τον υπολογισμό του προβλήματος.

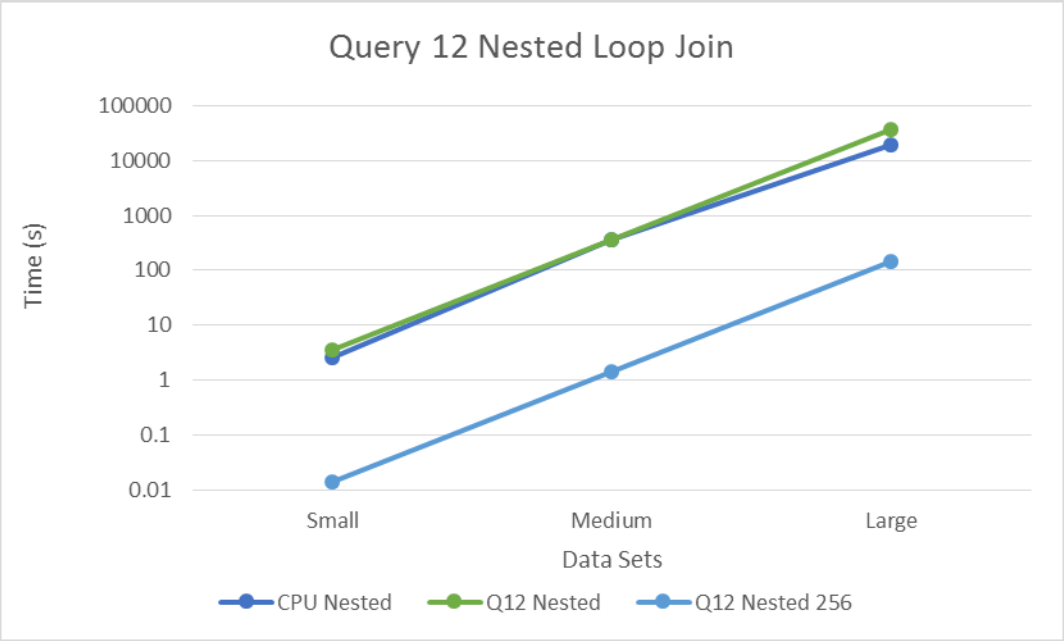
	Φάση I	Φάση II
CPU	$\frac{orSize}{PCIE}$	$\frac{liSize}{PCIE}$
Memory	$\frac{orSize}{LMem}$	$\frac{orSize * li}{LMem}$
Ticks	0	$\frac{li * or}{Frequency}$

Πίνακας 5.14: Φάσεις εκτέλεσης ερωτήματος 12 Α

Η χρήση του εύρους της εξωτερικής μνήμης ήταν και σε αυτή την περίπτωση σε πολύ χαμηλά επίπεδα. Ήταν δυνατή η αύξηση του αριθμού επεξεργασίας εγγραφών σε αρκετές χιλιάδες ανά κύκλο αλλά πάλι λόγω περιορισμού του υλικού χρησιμοποιήθηκαν 256.

	Small	Medium	Large
CPU Nested	2.5	359.31	19110.23
Q12 Nested	3.611	360.343	36007.3
Q12 Nested 256	0.014105	1.40759	140.6635

Πίνακας 5.15: Χρόνοι εκτέλεσης ερωτήματος 12 Α



Σχήμα 5.6: Γράφημα Ερωτήματος 12 Φάση Α

Σε αυτή την υλοποίηση παρατηρείται ότι η εκτέλεση στην CPU έχει μια σαφώς καλύτερη επίδοση όταν δεν γίνεται χρήση του παραλληλισμού από την πλατφόρμα της Maxeler. Η διαφορά δεν είναι στα επίπεδα που παρατηρήθηκαν στο Ερώτημα 3 της αντίστοιχης εκτέλεσης. λόγω του ότι εδώ υπάρχει μόνο ένα φωλιασμένο for, έτσι ο μέγιστος αριθμός κύκλων που μπορεί να χαθεί είναι liSize, σε αντίθεση με την υλοποίηση στο Ερώτημα 3 όπου ο αριθμός αυτός ήταν orSize*liSize. Έτσι, με την διαφορά να είναι σε μικρά επίπεδα, η εισαγωγή επεξεργασίας πολλαπλών εγγραφών ανά κύκλο στην πλατφόρμα της Maxeler βγήκε μπροστά με μια σοβαρή διαφορά.

5.4.2 Αποτελέσματα Φάσης Β (Hash Join)

Για τον υπολογισμό του χρόνου εκτέλεσης της τελευταίας υλοποίησης για το Ερώτημα 12, αρχικά έγινε υπολογισμός του χρόνου μεταφοράς του πίνακα orders και lineitems στον πυρήνα της μηχανής ροής. Στην συνέχεια έγινε εγγραφή του κατακερματισμένου πίνακα orders στην εσωτερική μνήμη από όπου και ακολούθως χρησιμοποιήθηκε για να γίνει η ένωση με τον πίνακα lineitems.

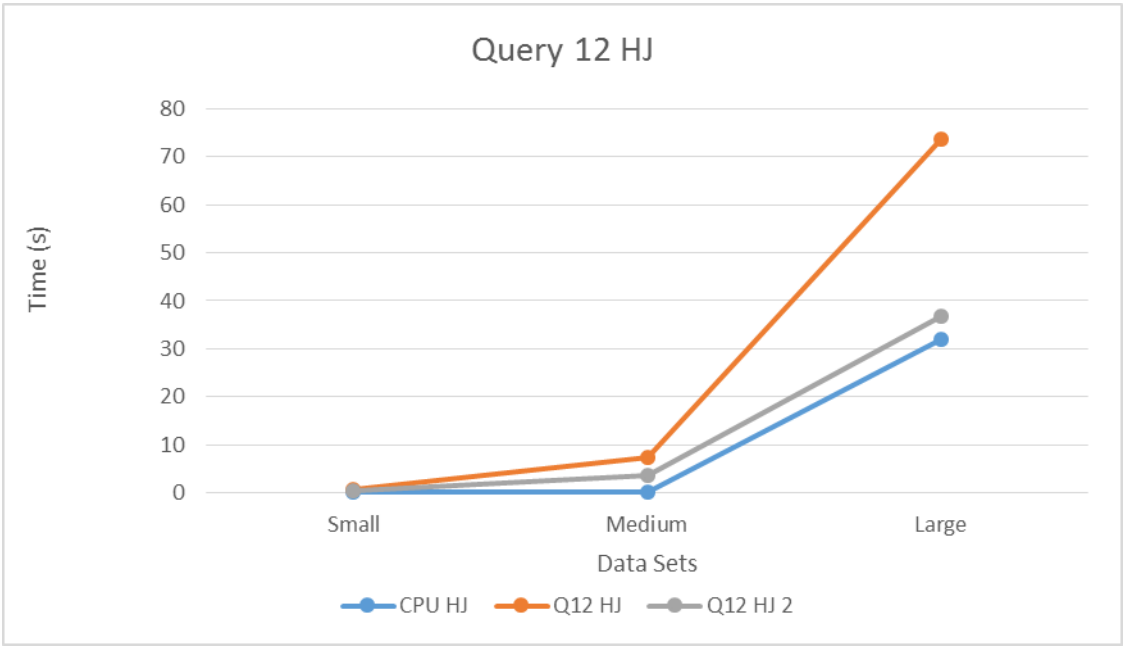
	Φάση I
CPU	$\frac{orSize + liSize}{PCIE}$
Memory	$\frac{orSize + liSize * partitionSize}{FMem}$
Ticks	$\frac{or + li * partitionSize}{Frequency}$

Πίνακας 5.16 Φάσεις εκτέλεσης ερωτήματος 12 B

Μετά τον υπολογισμό τον χρόνων, παρότι παρατηρήθηκε μειωμένη αξιοποίηση του εύρους της εσωτερικής μνήμης, δεν κατέστη δυνατό να χρησιμοποιηθεί λόγω της μεγάλης χρήσης υλικού που υπήρχε από την αρχική υλοποίηση. Για να είναι συμβατή με την πλατφόρμα στην οποία δοκιμάστηκε, χρησιμοποιήθηκε ο αριθμός 2 εγγραφών ανά κύκλο.

	Small	Medium	Large
CPU HJ	0.1	0.32	32
Q12 HJ	0.739	7.38	73.749
Q12 HJ 2	0.37	3.691	36.877

Πίνακας 5.17: Χρόνοι εκτέλεσης ερωτήματος 12 B



Σχήμα 5.7: Γράφημα Ερωτήματος 12 Φάση B

Παρόμοια με την αντίστοιχη υλοποίηση στο Ερώτημα 3, η εκτέλεση στην CPU είναι με μεγάλη διαφορά καλύτερη από την υλοποίηση στην πλατφόρμα της Maxeler στην υλοποίηση χωρίς παραλληλισμό. Ακόμη και στην περίπτωση που γίνεται επεξεργασία περισσοτέρων από μιας εγγραφής ανά κύκλο, λόγω του ότι ο αριθμός αυτός είναι πολύ μικρός, η εκτέλεση στο CPU πάλι προηγείται έναντι αυτής στην Maxeler.

5.5 Επιβεβαίωση Αποτελεσμάτων

Για την επιβεβαίωση των αποτελεσμάτων χρησιμοποιήθηκε η μηχανή ροής δεδομένων Galava πρόσβαση σε αυτή δόθηκε από το «Instituto de Engenharia de Sistemas e Computadores, Investigação e Desenvolvimento em Lisboa»
Σημειώνεται επιπλέον ότι η μηχανή αυτή δημιουργήθηκε κυρίως για την προώθηση της τεχνολογίας στα πανεπιστήμια. Είναι μια οικονομική μηχανή που ναι μεν επιτρέπει την χρήση της τεχνολογίας ροής δεδομένων αλλά με περιορισμένες δυνατότητες.

Χαρακτηριστηκά μηχανής:

Programmable Logic Fabric	490000 elements
Multipliers	512
FMem	5.6 MB
LMem	12 GB DDR3 DRAM Large Memory

Πίνακας 5.18: Προδιαγραφές μηχανής GALAVA

Πιο κάτω δίδονται οι χρόνοι εκτέλεσης του Ερωτήματος 12 για την υλοποίηση του Nested Loop Join και Hash Join για την κατηγορία δεδομένων “medium”.

Q12	Αρχικός Υπολογισμός	Χρόνος Εκτέλεσης GALAVA
Χρόνος A(s)	360	823
Χρόνος B(s)	7.38	17

Πίνακας 5.19: Σύγκριση με τους χρόνους εκτέλεσης στην μηχανή GALAVA

Στους χρόνους παρατηρείται μια διαφορά. Λόγος αυτού είναι το γεγονός ότι η μηχανή έτρεχε σε πιο χαμηλή συχνότητα από αυτή που ήταν υπολογισμένη.
Συγκεκριμένα, η μηχανή έτρεχε στα 110MHz ενώ για τον υπολογισμό ήταν 250MHz.
Επομένως, αν γίνει εφαρμογή των τύπων για την νέα συχνότητα, ο χρόνος εκτέλεσης για την υλοποίηση A θα υπολογιζόταν ως: $T = \frac{li*or}{Frequency} = \frac{600576*150144}{110*10^6} = 819.75$, και για την B:

$$T = \frac{150144 + 600576 * 3072}{110 * 10^6} = 16.78$$

Και οι δύο χρόνοι παρατηρείται ότι είναι αρκετά κοντά στους πραγματικούς χρόνους εκτέλεσης της μηχανής GALAVA.

Σημειώνεται επιπλέον ότι για το κτίσιμο των εφαρμογών, χρησιμοποιήθηκαν οι ρυθμίσεις για το κτίσιμο των εφαρμογών στον βέλτιστο χρόνο, με τον μικρότερο βαθμό βελτιστοποιήσεων. Με την διαμόρφωση των ρυθμίσεων αυτών, υπολογίζεται ότι η συχνότητα θα μπορούσε να φτάσει μέχρι τα 150-200MHz, να μειωθεί η χρήση λογικής (επεξεργασία περισσότερων εγγραφών ανά κύκλο) ή και συνδυασμός των δύο. Όλα αυτά έχοντας την επιβάρυνση του χρόνου, έχοντας τις αρχικές ρυθμίσεις να χρειάζονται μερικές ώρες για το κτίσιμο και με την χρήση των βελτιστοποιήσεων ο αριθμός αυτός να γίνεται της τάξεως των ημερών.

Πάνω στην μηχανή GALAVA δοκιμάστηκαν επίσης υλοποιήσεις από το Ερώτημα 6, το γεγονός όμως ότι οι χρόνοι εκτέλεσης των υλοποιήσεων ήταν πολύ μικροί δεν άφησαν περιθώρια εξαγωγής συμπερασμάτων.
Οι χρόνοι υλοποιήσεων του Ερωτήματος 3 δεν συμπεριελήφθησαν λόγω του ότι οι υλοποιήσεις βασίζονται πάνω στο Nested Loop Join και Hash Join, υλοποιήσεις οι οποίες επιβεβαιώθηκαν πιο πάνω. Επιπλέον, στην μηχανή GALAVA, υπήρχε μια αρχαιότερη έκδοση λειτουργικού της Maxeler και για να τρέξουν οι υλοποιήσεις στην μηχανή έπρεπε να ακολουθηθεί μια διαδικασία μετατροπής του αρχικού κώδικα για να επιλυθούν κάποια θέματα συμβατότητας.

Κεφάλαιο 6

Συμπεράσματα

6.1 Συμπεράσματα

Η πλατφόρμα της Maxeler έρχεται με μια πληθώρα δυνατοτήτων. Μετά από μελέτη και ανάλυση σχετικών εργασιών, μπορεί να λεχθεί ότι αυτή η αρχιτεκτονική ροής δεδομένων σήμερα βρίσκει πολλές εφαρμογές στον τομέα της επεξεργασίας σήματος, στην γονιδιωματική και στα οικονομικά.

Είδαμε στην παρούσα εργασία ότι ακόμα και σε μια κατηγορία προβλημάτων η οποία φαινομενικά δεν εντάσσεται πλήρως στο βελησνεκές των δυνατοτήτων της, εντούτοις έγινε κατορθωτό να επιτευχθεί μια σημαντική πρόοδος.

Κλειδί φυσικά σε αυτό ήταν η σωστή αξιοποίηση αυτών των δυνατοτήτων. Όπως παρατηρήθηκε, η αξιοποίηση του παραλληλισμού που μπορεί να προσφέρει η πλατφόρμα, της δίνει ένα πολύ μεγάλο προβάδισμα έναντι της κλασσικής υλοποίησης σε CPU. Το γεγονός αυτό φαίνεται από την επιτάχυνση που επιτεύχθηκε στις υλοποιήσεις στις οποίες έγινε κατορθωτό να γίνει αξιοποίηση του παραλληλισμού σε μεγάλο βαθμό.
Το γεγονός αυτό επιτρέπει να ειπωθεί ότι η πλατφόρμα της Maxeler, με την κατάλληλη υλοποίηση, μπορεί να δώσει μεγάλη ώθηση ακόμη και σε ερωτήματα βάσεων δεδομένων.

Μετά την αναγνώριση της θετικής της πλευράς, πρέπει να αναφερθεί ότι παρόλο που έρχεται με μια μεγάλη γκάμα δυνατοτήτων, έρχεται ταυτόχρονα και με μια μεγάλη καμπύλη μάθησης. Το γεγονός αυτό δεν οφείλεται τόσο στην πολυπλοκότητα χρήσης της, αλλά στο ότι είναι μια σχετικά νέα τεχνολογία, την οποία υποστηρίζει μια σχετικά μικρή, κλειστή κοινότητα. Το γεγονός αυτό δυσκολεύει την εκμάθηση της πλατφόρμας παρότι πίσω της έχει μια εξαιρετική τεκμηρίωση.

6.2 Μελλοντική Εργασία

Στην παρούσα διπλωματική εργασία έχουν αναλυθεί οι χρόνοι τριών ερωτημάτων της σουίτας ερωτημάτων TPC-H. Στον μέλλον, θα μπορούσε να γίνει επέκταση της εργασίας αυτής με την εισαγωγή περισσοτέρων ερωτημάτων, καθώς και με την λήψη μετρήσεων σχετικά με την κατανάλωση ενέργειας, αφού στις τελευταίες τις εκδόσεις η Maxeler, πλέον υποστηρίζεται η εύρεση τέτοιου είδους πληροφοριών.

Βιβλιογραφία

[1] Maxeler Technologies, Multiscale Dataflow Programming. Maxeler Technologies Ltd., February, 2014

[2] Maxeler Technologies, MaxCompiler Manager Compiler Tutorial. Maxeler Technologies Ltd., February, 2014

[3]Maxeler Technologies, Acceleration Tutorial: Loops and Pipelining. Maxeler Technologies Ltd., February, 2014

[4] Maxeler Technologies, Kernel Numerics. Maxeler Technologies Ltd., February, 2014

[5] Maxeler Technologies, State Machine. Maxeler Technologies Ltd., February, 2014

[6] D. Oriato, O. Pell. 'FD modeling beyond 70Hz'. HPC Workshop, Denver, USA, October 2010.

[7] P. Marchetti, D. Oriato et al. 'Fast 3D ZO CRS Stack'. CRS4 72nd European Association of Geoscientists and Engineers (EAGE) Conference, Barcelona, June 2010.

[8] Pell, Oliver et al. 'Finite-Difference Wave Propagation Modeling On Special-Purpose Dataflow Machines'. IEEE Transactions on Parallel and Distributed Systems, 2013, pp. 906-915

[9] S. Weston, J-T. Marin et al. 'Accelerating the Computation of Portfolios of Tranched Credit Derivatives'. IEEE Workshop on High Performance Computational Finance, New Orleans, USA, November 2010.

[10] Weston, Stephen et al. 'Rapid Computation Of Value And Risk For Derivatives Portfolios'. Concurrency and Computation: Practice and Experience, Special Issue Paper, July 2011, pp. 880-894

[11] A. Fukui. 'High Performance Smith-Waterman Local Sequence Alignment'. University of Tokyo. Symposium on Verification, Tokyo, Japan, Febuary 2011.

[12] O. Pell, Satoru Yamamoto et al. 'Streaming Computation for Lattice Boltzmann Method'. IEEE International Conference, Kitakyushu, Japan, December 2007.