ΑΡΙΣΤΟΤΕΛΕΙΟ
ΠΑΝΕΠΙΣΤΗΜΙΟ
ΘΕΣΣΑΛΟΝΙΚΗΣ

Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Τομέας Ηλεκτρονικής και Υπολογιστών

Homomorphic Encryption using Microsoft SEAL and FPGA acceleration

Διπλωματική Εργασία του φοιτητή

Ηλιάδη-Αποστολίδη Δημοσθένη

Επιβλέπων

Παπαευσταθίου Ιωάννης

Αναπληρωτής Καθηγητής

Θεσσαλονίκη, Δεκέμβριος 2021

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή κ. Ιωάννη Παπαευσταθίου, καθώς ήταν ο άνθρωπος που μου έδωσε την ευκαιρία να ασχοληθώ με το συγκεκριμένο θέμα, αλλά και για την καθοδήγησή του, που συνέβαλε στην εκπλήρωση της εργασίας. Θα ήθελα να ευχαριστήσω επίσης τον κ. Σταμάτιο Ανδριανάκη για την επικοινωνία και τις υποδείξεις του, που βοήθησαν στην ολοκλήρωση της εργασίας αυτής. Τέλος, ευχαριστώ την οικογένειά μου για τη στήριξή τους όλα αυτά τα χρόνια, αλλά και τους φίλους μου και όλους με όσους συναναστράφηκα κατά τη διάρκεια των σπουδών μου.

Περίληψη

Το Homomorphic Encryption είναι μια αναδυόμενη τάση στο χώρο της Κυβερνοασφάλειας και έχει ως βασική λογική την υλοποίηση υπολογισμών πάνω σε κρυπτογραφημένα δεδομένα, χωρίς την πρόσβαση σε κάποιο κρυφό κλειδί, αλλά και χωρίς την ανάγκη να προηγηθεί αποκρυπτογράφηση των δεδομένων αυτών. Το αποτέλεσμα είναι μια κρυπτογραφημένη μορφή. Όταν αυτή η μορφή αποκρυπτογραφηθεί, το αποτέλεσμα είναι το ίδιο με το να είχαμε εκτελέσει τους υπολογισμούς αυτούς σε μη κρυπτογραφημένα δεδομένα. Το Homomorphic Encryption μπορεί να χρησιμοποιηθεί ως λύση για την διατήρηση των εννοιών της εμπιστευτικότητας και της ακεραιότητας στα δεδομένα, αλλά και στους υπολογισμούς που γίνονται σε αυτά. Η λογική του Homomorphic Encryption βασίζεται στην έννοια του ομομορφισμού, δηλαδή της ύπαρξης μιας απεικόνισης μεταξύ δυο αλγεβρικών δομών. Αντίστοιχα, στην περίπτωση του Homomorphic Encryption, υπάρχει η μεταξύ των δεδομένων, απεικόνιση, κατά την οποία διατηρούνται κάποιες ιδιότητες, το οποίο και επιτρέπει την παραπάνω λογική.

Η χρήση αλγορίθμων που βασίζονται στο Homomorphic Encryption εκτιμάται ότι θα είναι ευρεία στο κοντινό μέλλον. Ενδεικτικά, κάποιες εφαρμογές για τις οποίες θα είναι άμεσα χρήσιμο, είναι η περαιτέρω ασφάλεια σε δεδομένα που βρίσκονται σε νέφος (cloud), η ανάλυση δεδομένων σε Βιομηχανίες που έχουν κανονισμούς για τα προσωπικά δεδομένα (π.χ. Ασθενείς σε νοσοκομεία), το Machine Learning και άλλες.

Το Fully Homomorphic Encryption (FHE), βασίζεται στην παραπάνω λογική, επιτρέποντας ταυτόχρονα την υλοποίηση αυθαίρετων υπολογισμών σε κρυπτογραφημένα κείμενα (ciphertexts), δηλαδή έχοντας μία κρυπτογραφημένη είσοδο μπορούμε να επιτύχουμε την κρυπτογράφηση του αποτελέσματος. Οι γενιές της κρυπτογράφησης που αποτελούν τα Fully Homomorphic Encryption schemes είναι οι εξής :

1) Pre-FHE -> RSA, Eigenmap, Goldwasser-Micali
2) 1st gen FHE -> Gentry, (Ideal) Lattice-Based Cryptography, Isomorphic
3) 2nd gen FHE -> Brakerski-Gentry-Vaikuntanathan† (BGV), NTRU (LTV), BFV, NTRU (BLLN)
4) 3rd gen FHE -> GSW (FHEW, TFHE)
5) 4th gen FHE -> CKKS

Γίνονται επίσης αναφορές στο Partially Homomorphic Encryption (PHE), στο Somewhat Homomorphic Encryption (SHE) και στο Levelled Fully Homomorphic Encryption (LFHE).

Πλέον υπάρχουν διάφορες βιβλιοθήκες, οι περισσότερες γραμμένες σε C++, που υλοποιούν αρκετά schemes του Fully Homomorphic Encryption. Μια από τις πλέον διαδεδομένες αποτελεί η βιβλιοθήκη Microsoft SEAL και βρίσκεται ακόμα σε ερευνητικό επίπεδο. Η βιβλιοθήκη αυτή, χαρακτηρίζεται από την ευκολία χρήσης της, χωρίς να απαιτούνται ιδιαίτερες γνώσεις κρυπτογραφίας, αλλά και καλή απόδοση που παρέχει σε επίπεδο εκτέλεσης, των schemes που διαθέτει.

Το μεγάλο πρόβλημα που αφορά και το μέλλον του Homomorphic Encryption, είναι το κόστος εκτέλεσης των προαναφερθέντων υπολογισμών και κυρίως το χρονικό διάστημα που χρειάζονται για να ολοκληρωθούν. Ένας τρόπος για τη βελτίωση του χρόνου εκτέλεσης, είναι η αξιοποίηση των FPGAs, για την επιτάχυνση των κοστοβόρων αλγορίθμων-συναρτήσεων, με σκοπό την αποδοτικότερη εκτέλεσή τους.

Στα πλαίσια αυτής της διπλωματικής εργασίας, ασχολούμαστε με το παράδειγμα του CKKS scheme που προσφέρει η βιβλιοθήκη Microsoft SEAL και γίνεται προσπάθεια επιτάχυνσής του. Έπειτα από ανάλυση του παραδείγματος αυτού με το εργαλείο Intel® VTune™ Profiler, γίνεται εμφανές πως η πιο κοστοβόρα συνάρτηση στον κώδικα αυτόν, αποτελεί η συνάρτηση divide_uint128_uint64_inplace_generic();, η οποία καλείται 393485 φορές και κάνει την πράξη της διαίρεσης μεταξύ ενός αριθμητή uint128 bit και ενός παρονομαστή uint64 bit.

Χρησιμοποιώντας τη διαδικασία της επιτάχυνσης υλικού/λογισμικού, επιτυγχάνεται αισθητή βελτίωση του χρόνου εκτέλεσης της παραπάνω συνάρτησης, η οποία και μεταβάλλεται σημαντικά για να επιτευχθεί αυτό. Πιο συγκεκριμένα, δοκιμάστηκαν 5 διαφορετικές σχεδιάσεις, όπου η μία επέφερε σημαντικά αποτελέσματα. Αυτό επιτεύχθηκε, επιλέγοντας την Xilinx® Alveo™ U200 Data Center accelerator card και χρησιμοποιώντας τα εργαλεία της Xilinx®. Πιο συγκεκριμένα, με το Vivado HLS 2019.2, μπορούμε να καταλήξουμε σε χρήσιμα συμπεράσματα σχετικά με τη σημασία του acceleration και του γενικότερου speedup σε δύσκολους και περίπλοκους υπολογισμούς και αλγορίθμους, όπως και έγινε στην συγκεκριμένη διπλωματική εργασία.

Περιεχόμενα

Περίληψη 3

Περιεχόμενα 5

Κατάλογος Εικόνων 7

Κατάλογος Πινάκων 8

Κεφάλαιο 1 : Εισαγωγή 9

1.1) Περιγραφή Θεματικής Περιοχής 10

1.2) Σκοπός της Διπλωματικής Εργασίας 10

1.3) Διάρθρωση της Διπλωματικής Εργασίας 11

Κεφάλαιο 2 : Εισαγωγή στην έννοια του Homomorphic Encryption 13

2.1) Homomorphic Encryption : Ένα σύντομο ιστορικό 14

2.2) Βασικές έννοιες του Homomorphic Encryption 14

2.3) Lattice-based Cryptography 19

2.4) (Ring) Learning With Errors (LWE, RLWE) 19

Κεφάλαιο 3 : Γενιές του Fully Homomorphic Encryption 21

3.1) Εισαγωγικά – Pre-FHE 22

3.2) 1st Generation of FHE 23

3.3) 2nd Generation of FHE 24

3.4) 3rd Generation of FHE 25

3.5) 4th Generation of FHE - CKKS 26

Κεφάλαιο 4 : Microsoft SEAL 28

4.1) Εισαγωγή στο Microsoft SEAL 29

4.2) Τα στάδια χρήσης των schemes στο Microsoft SEAL 30

4.3) Δομή, κλάσεις και συναρτήσεις του Microsoft SEAL 31

Κεφάλαιο 5 : Εισαγωγή στα FPGA και υλοποίηση σχεδίασης *41*
5.1) Εισαγωγή στα FPGA και στην έννοια του HLS *42*
5.2) Xilinx® Alveo™ U200 Data Center accelerator card *44*
5.3) Περιγραφή κάποιων βασικών HLS directives *48*
5.4) Ανάλυση με το Intel® VTune™ Profiler *53*
5.5) Περιγραφή της υλοποίησης *55*
5.6) Περιγραφή των σχεδιάσεων *65*
5.7) Δομή Αποτελεσμάτων με το Vivado HLS 2019.2 ..... *67*
5.8) Αποτελέσματα και συμπεράσματα επί αυτών *74*

Κεφάλαιο 6 : Προβλήματα που αντιμετωπίστηκαν *82*
6.1) Εισαγωγικά *83*
6.2) Προσπάθειες που δεν κατέληξαν σε αποτέλεσμα *83*
6.3) Προβλήματα με το Microsoft SEAL και το Vivado HLS 2019.2 ..... *85*
6.4) Προβλήματα σχετικά με τις σχεδιάσεις *87*

Κεφάλαιο 7 : Συμπερασματικά *90*

Βιβλιογραφία *93*

Κατάλογος Εικόνων

Κεφάλαιο 2 : Εισαγωγή στην έννοια του Homomorphic Encryption 13

2.1 Βασικές υποκατηγορίες του Homomorphic Encryption 29

Κεφάλαιο 4 : Microsoft SEAL 28

4.1 Traditional Cloud 29

4.2 Cloud που χρησιμοποιεί Homomorphic Encryption 30

Κεφάλαιο 5 : Εισαγωγή στα FPGA και υλοποίηση σχεδίασης 41

5.1 Τρόπος που χτίζεται (build) ο κώδικας στο Vitis 44

5.2 Η βασική αρχιτεκτονική των FPGAs, σύνδεση μεταξύ των διαφόρων CLBs 44

5.3 Η δομή του Flip-Flop 45

5.4 Αναπαράσταση ενός LUT ως ένα σύνολο πυλών 45

5.5 Η δομή ενός DSP48 block 46

5.6 Τρόπος διασύνδεσης μεταξύ του Host και του Kernel 47

5.7 Task-Level Pipelining με το Dataflow directive 50

5.8 Παράδειγμα με το Pipeline directive 51

5.9 Διαφορές μεταξύ Loop Pipeline και Loop Unroll 52

5.10 Αποτελέσματα ανάλυσης με το Intel® VTune™ Profiler 54

5.11 Αλγόριθμος για το διάβασμα των αρχείων 57

5.12 Αλγόριθμος για το κάλεσμα της top function 58

5.13 Αλγόριθμος για την επαλήθευση των αποτελεσμάτων 59

5.14 Αλγόριθμος του top function, της συνάρτησης div 61

5.15 Αλγόριθμος για τον υπολογισμό του αριθμού bits σε pointer 62

5.16 Αλγόριθμος για τον υπολογισμό του αριθμού bits σε πίνακα 63

5.17 Αλγόριθμος για τη διόρθωση του αριθμητή 64

5.18 Τρόπος εμφάνισης των αποτελεσμάτων στην κονσόλα του εργαλείου 73

Κεφάλαιο 6 : Προβλήματα που αντιμετωπίστηκαν 82

6.1 Παράδειγμα για χρήση TLS 84

6.2 Κώδικας για μη χρήση TLS από τον compiler 84

6.3 Κώδικας για μη χρήση TLS από τον compiler του Vivado HLS 87

Κατάλογος Πινάκων

Κεφάλαιο 4 : Microsoft SEAL 28

4.1 Πίνακας αριθμών bits του coefficient modulus, για κλασσική ασφάλεια 34

4.2 Πίνακας αριθμών bits του coefficient modulus, για κβαντική ασφάλεια 34

Κεφάλαιο 5 : Εισαγωγή στα FPGA και υλοποίηση σχεδίασης 41

5.1 Resources του Xilinx® Alveo™ U200 Data Center accelerator card 47

5.2 Πίνακας περιόδου ρολογιού και εκτιμώμενου χρόνου 68

5.3 Πίνακας του εκτιμώμενου Latency 68

5.4 Πίνακας για τα Instances 68

5.5 Πίνακας εκτιμήσεων για το Latency που αφορά τη λούπα 69

5.6 Πίνακας γενικών εκτιμήσεων χρήσης των πόρων 69

5.7 Πίνακας εκτιμήσεων χρήσης πόρων για Instances 70

5.8 Πίνακας εκτιμήσεων χρήσης πόρων για τη μνήμη 70

5.9 Πίνακας εκτιμήσεων χρήσης πόρων των πολυπλεκτών 70

5.10 Πίνακας χρήσης πόρων για expressions 71

5.11 Πίνακας χρήσης πόρων των registers 72

5.12 Πίνακας χρήσης των θυρών 73

5.13 Πίνακας αποτελέσματος μετά το C/RTL Cosimulation 73

5.14 Αποτελέσματα για την πρώτη σχεδίαση 75

5.15 Αποτελέσματα για τη δεύτερη σχεδίαση 75

5.16 Αποτελέσματα για την τρίτη σχεδίαση 76

5.17 Αποτελέσματα για την τέταρτη σχεδίαση 76

5.18 Αποτελέσματα για τη βέλτιστη σχεδίαση, για II = 1 78

5.19 Αποτελέσματα για τη βέλτιστη σχεδίαση, για II = 2 79

5.20 Αποτελέσματα για τη βέλτιστη σχεδίαση, για II = 3 80

Κεφάλαιο 1 : Εισαγωγή

1.1) Περιγραφή Θεματικής Περιοχής

Στο σήμερα, που η τεχνολογία είναι πιο προσβάσιμη από ποτέ και η μάζα των δεδομένων και των πληροφοριών μεγαλώνει διαρκώς, η ανάγκη για την ασφάλεια αλλά και την γρήγορη προσπέλαση των δεδομένων αυτών γίνεται επιτακτική. Το Homomorphic Encryption αποτελεί μια καινοτομία στον χώρο της Κυβερνοασφάλειας και αποτελεί έναν πολλά υποσχόμενο τομέα στην Κρυπτογραφία. Το σημαντικό του πλεονέκτημα είναι, ότι μπορούν να γίνουν υπολογισμοί πάνω σε κρυπτογραφημένα δεδομένα, χωρίς την πρόσβαση σε κάποιο κρυφό κλειδί, αλλά και χωρίς την ανάγκη να προηγηθεί αποκρυπτογράφηση των δεδομένων αυτών. Αυτό σημαίνει ότι είναι αρκετά πιο ασφαλές από άλλες λογικές κρυπτογράφησης, έχοντας όμως και ένα πρόβλημα. Το μεγάλο πρόβλημα με το Homomorphic Encryption είναι η ταχύτητα με την οποία μπορούν να εκτελεστούν τα διάφορα schemes, η οποία ακόμα και σήμερα δεν είναι ικανοποιητική. Είναι λογικό, πως η υλοποίηση αρκετά περίπλοκων υπολογισμών, δυσχεραίνει κατά πολύ την απόδοση σε χρόνο εκτέλεσης των schemes που ανήκουν στη λογική αυτή. Υπάρχουν αρκετά schemes και αρκετές βιβλιοθήκες, με το CKKS scheme και τη βιβλιοθήκη Microsoft SEAL να ξεχωρίζουν στον τομέα αυτόν. Ένας τρόπος που εξετάζεται για να γίνει αυτό, είναι η επιτάχυνση των schemes αυτών (ή κομματιών του), με σκοπό την βελτίωση στον χρόνο που χρειάζονται για να εκτελεστούν. Υπάρχουν και δημοσιεύσεις που προτείνουν τη χρήση CUDA για την αξιοποίηση του thread-level parallelism, αλλά οι περισσότερες αφορούν την αξιοποίηση των FPGA (Field-programmable Gate Array), με συγκεκριμένη σχεδίαση σε επίπεδο κώδικα και την επιτάχυνση αυτής.

Σε θεωρητικό επίπεδο έχουν γίνει αρκετές προτάσεις σχεδιάσεων, οι οποίες έχουν ως σκοπό την επίλυση του προβλήματος της ταχύτητας εκτέλεσης. Στη δημοσίευση "HEAX: An Architecture for Computing on Encrypted Data" προτείνεται μια σχεδίαση με αρκετά θετικά αποτελέσματα, δίνοντας ιδιαίτερο βάρος στον αλγόριθμο CKKS.

Θα δούμε, λοιπόν, αναλυτικά τη βιβλιοθήκη Microsoft SEAL, το CKKS scheme, καθώς και την επιτάχυνση που επιτεύχθηκε, κομματιού του scheme αυτού, με τη χρήση των εργαλείων της Xilinx® και πιο συγκεκριμένα του Vivado HLS 2019.2.

1.2) Σκοπός της Διπλωματικής Εργασίας

Η διπλωματική αυτή έχει ως σκοπό την εμπεριστατωμένη ανάλυση της έννοιας του Homomorphic Encryption, την ανάλυση κάποιων βασικών schemes που βασίζονται στη λογική του και πιο συγκεκριμένα στην ανάλυση κάποιων Fully Homomorphic Encryption schemes, την ανάλυση της βιβλιοθήκης Microsoft SEAL και των συναρτήσεων της.

Τα schemes που βασίζονται στο Homomorphic Encryption και για την ακρίβεια αποτελούν Fully Homomorphic Encryption schemes, είναι αρκετά περίπλοκα και γι' αυτό είναι αναγκαία η ύπαρξη των βιβλιοθηκών, οι οποίες σκοπό έχουν τη χρήση πολλών μικρότερων συναρτήσεων, οι οποίες θα μπορέσουν να απλοποιήσουν το πρόβλημα της δυσκολίας κατανόησης των schemes αυτών. Το scheme το οποίο έχει επιλεγεί για να γίνει accelerate, είναι το CKKS, καθώς αποτελεί το πιο πρόσφατο scheme στον χώρο του Homomorphic Encryption και είναι πολλά υποσχόμενο.

Στα πλαίσια αυτής της διπλωματικής εργασίας, υλοποιούνται 5 σχεδιάσεις, οι οποίες έχουν ως σκοπό την επίτευξη καλύτερου χρονικού αποτελέσματος, χρησιμοποιώντας το εργαλείο Vivado HLS 2019.2 με σκοπό την επιτάχυνση κομματιού κώδικα που αφορά το CKKS scheme, την καταγραφή των αποτελεσμάτων και την άντληση χρήσιμων συμπερασμάτων επί αυτών. Για να γίνει αυτό, χρειάζονται δύο αρχεία-κώδικες, ένα testbench και ένα source αρχείο, για κάθε σχεδίαση από τις 5 που γίνονται, όπου το testbench θα είναι υπεύθυνο για την επιβεβαίωση της σχεδίασης, η οποία θα περιγράφεται στο source.

1.3) Διάρθρωση της Διπλωματικής Εργασίας

Το θέμα που πραγματεύεται η διπλωματική αυτή, είναι διαμορφωμένο και αναπτυγμένο σε 7 κεφάλαια. Στο δεύτερο και στο τρίτο κεφάλαιο γίνεται ανάλυση του Homomorphic Encryption, των γενιών του και των αλγορίθμων τους. Στο τέταρτο κεφάλαιο γίνεται ανάλυση της δομής και των συναρτήσεων της βιβλιοθήκης Microsoft SEAL. Στο πέμπτο κεφάλαιο, γίνεται λόγος για την υλοποίηση της σχεδίασης στο εργαλείο Vivado HLS 2019.2, η ανάλυση του κώδικα του testbench και του source, καθώς και η παρουσίαση των αποτελεσμάτων, της επιτάχυνσης που επετεύχθη, καθώς και σχολιασμός και συμπεράσματα επί των αποτελεσμάτων αυτών. Στο έκτο κεφάλαιο γίνεται παρουσίαση όλων των προβλημάτων και των λύσεων που επιλέχθηκαν για αυτά και τέλος, στο έβδομο κεφάλαιο γίνεται η ανακεφαλαίωση και αναλύονται κάποια συμπεράσματα που προκύπτουν από τη διπλωματική αυτή.

Στο Κεφάλαιο 2, θα δούμε κάποια βασικά στοιχεία που αποτελούν τη βάση για το Homomorphic Encryption. Πιο συγκεκριμένα, θα γίνει ανάλυση κάποιων βασικών μαθηματικών και λογικών πράξεων, πράξεων με παραμέτρους ασφαλείας, αλλά και ανάλυση κάποιων μαθηματικών προβλημάτων και εννοιών που απασχολούν το χώρο του Homomorphic Encryption. Όλα αυτά αποτελούν μια σημαντική εισαγωγή για την έννοια του Homomorphic Encryption, που συμβάλλουν στην καλύτερη κατανόησή της, αλλά και των schemes που είναι κομμάτι της.

Στο Κεφάλαιο 3, γίνεται ανάλυση των γενιών των Fully Homomorphic Encryption Schemes, της συνολικής λογικής που υπάρχει πίσω από αυτά τα διαφορετικά schemes, αναφέρονται είναι οι διαφορές που υπάρχουν μεταξύ τους, αλλά και η ίδια η εξέλιξή τους. Η σημασία του κεφαλαίου αυτού έγκειται, στην ανάδειξη της ιστορίας και της βελτίωσης που υφίσταντο τα schemes από γενιά σε γενιά και εν τέλει η κατάληξη στην πιο πρόσφατη γενιά που εμπεριέχει και το CKKS scheme, που μας απασχολεί και στο τεχνικό κομμάτι της διπλωματικής αυτής.

Στο Κεφάλαιο 4, γίνεται εκτενής ανάλυση της βιβλιοθήκης Microsoft SEAL. Η βιβλιοθήκη αυτή θεωρείται από τις πλέον υποσχόμενες στον χώρο του Homomorphic Encryption, λόγω και της ταχύτητάς της, αλλά και λόγω της ευκολίας στην χρήση της. Ιδανικά, ένας απλός προγραμματιστής θα μπορούσε να χρησιμοποιήσει κάποιο scheme που προσφέρει η βιβλιοθήκη αυτή, σε μερικές γραμμές. Αρχικά, εξηγείται η λογική πίσω από τη βιβλιοθήκη, αλλά και ο τρόπος χρήσης της. Έπειτα, αναλύονται οι σημαντικότερες κλάσεις που εμπεριέχονται σε αυτήν, καθώς και κάποιες βασικές συναρτήσεις που παίζουν βασικό ρόλο στην χρήση του CKKS scheme, όπως αυτό είναι υλοποιημένο στη συγκεκριμένη βιβλιοθήκη. Πιο συγκεκριμένα, μας ενδιαφέρει το κόστος που έχουν οι συναρτήσεις αυτές σε χρόνο εκτέλεσης, σε ένα απλό παράδειγμα

που προσφέρει η Microsoft που χρησιμοποιείται το CKKS scheme. Το κεφάλαιο αυτό, αναδεικνύει από τη μία την απλότητα χρήσης που προσφέρει η βιβλιοθήκη Microsoft SEAL, αλλά από την άλλη τη δυσκολία που υπάρχει στο να επιτευχθεί επιτάχυνση με χρήση FPGA, λόγω της μεγάλης αλληλεξάρτησης και πολυπλοκότητας που υπάρχει μεταξύ των κλάσεων αυτών. Αυτό, δε βοηθάει στην υλοποίηση σχεδιάσεων που θα μπορέσουν με ευκολία να επιφέρουν βελτιώσεις στην απόδοση του κώδικα.

Στο Κεφάλαιο 5, γίνεται ανάλυση της υλοποίησης που έγινε, των σχεδιάσεων που υλοποιήθηκαν και δοκιμάστηκαν, της δομής των αποτελεσμάτων που δίνει το Vivado HLS 2019.2, καθώς και παρουσιάζονται τα αποτελέσματα που επιτεύχθηκαν και τα συμπερασμάτα που προκύπτουν, επί αυτών. Με τον όρο σχεδιάσεις, εννοείται οι διαφορετικές περιπτώσεις κώδικα που δοκιμάστηκαν, για να επιτευχθεί επιτάχυνση, καθώς και τα αποτελέσματα που προέκυψαν με αυτές. Σημαντικό κομμάτι του κεφαλαίου αυτού, αποτελεί η ανάλυση που έγινε με το Intel® VTune™ Profiler, που αφορά το παράδειγμα υλοποίησης του CKKS scheme της βιβλιοθήκης Microsoft SEAL, το οποίο και έδειξε το κόστος σε χρόνο εκτέλεσης, των διαφόρων συναρτήσεων. Η ανάλυση αυτή, βοήθησε στην επιλογή της κατάλληλης και πιο κοστοβόρας για να επιταχυνθεί εν συνεχεία, με τη χρήση των FPGA. Γίνεται επίσης ανάλυση του High Level Synthesis και των πλεονεκτημάτων που αυτό προσφέρει, καθώς και συνολικότερα της λογικής της επιτάχυνσης υλικού/λογισμικού. Ακόμη, αναφέρονται κάποια βασικά χαρακτηριστικά για το Alveo™ U200 Data Center accelerator card, καθώς και κάποιων βασικών HLS directives που χρησιμοποιήθηκαν για την επιτάχυνση της σχεδίασης. Είναι εμφανές, πως τα αποτελέσματα που παρουσιάζονται στο κεφάλαιο αυτό, είναι η ουσία όλου του τεχνικού κομματιού αυτής της διπλωματικής εργασίας, όπου και επιτυγχάνεται σημαντική βελτίωση, από τον αρχικό χρόνο.

Στο Κεφάλαιο 6, επεξηγούνται αναλυτικά τα προβλήματα που προέκυψαν κατά τη διάρκεια εκπόνησης αυτής της διπλωματικής εργασίας, καθώς και οι τρόποι αντιμετώπισης των προβλημάτων αυτών. Κάποια προβλήματα δεν ήταν δυνατό να επιλυθούν, αλλά σε κάθε περίπτωση, γίνεται αναφορά στον τρόπο με τον οποίο ξεπεράστηκαν τα εμπόδια αυτά.

Στο Κεφάλαιο 7, καταλήγουμε σε συμπεράσματα που προκύπτουν από αυτήν τη διπλωματική εργασία, αναφορικά με το Homomorphic Encryption, αλλά και το πώς η αξιοποίηση των FPGA, μπορεί να επιφέρει σημαντικά αποτελέσματα σε σχέση με την απόδοση πολύπλοκων και κοστοβόρων κομματιών κώδικα.

Κεφάλαιο 2 : Εισαγωγή στην έννοια του Homomorphic Encryption

2.1) Homomorphic Encryption : Ένα σύντομο ιστορικό

Η πρώτη φορά που αναφέρεται η αναγκαιότητα ύπαρξης του Homomorphic Encryption, είναι το 1978, στο paper "*On Data Banks and Privacy Homomorphisms*". Σε αυτήν τη δημοσίευση, αναφέρεται η αναγκαιότητα χρήσης ενός μυστικού ομομορφισμού για την κρυπτογράφηση των δεδομένων μιας τράπεζας, ούτως ώστε ένας χρονικά κοινόχρηστος υπολογιστής να μπορεί να χειρίζεται τα δεδομένα, χωρίς να είναι απαραίτητο να προηγηθεί η αποκρυπτογράφησή τους. Για πολλά χρόνια, η παραδοχή αυτή παραμένει ένα ανοιχτό πρόβλημα, μέχρι το 2009, όπου ο Gentry στη διδακτορική του διατριβή "*A Fully Homomorphic Encryption Scheme*", ορίζει τη βασική ορολογία του Homomorphic Encryption, τις υποκατηγορίες του, αλλά και προτείνει έναν αλγόριθμο που θα αποτελέσει το πρώτο Fully Homomorphic Encryption scheme.

Σημαντική σημείωση που αφορά το Homomorphic Encryption είναι, πως δεν υπάρχει κάποιο επίσημο πρότυπο (standard), που να πιστοποιεί την λειτουργικότητα, την ταχύτητα και την ασφάλεια των διαφόρων Homomorphic Encryption schemes. Μια προσπάθεια προς αυτήν την κατεύθυνση, αποτελεί το homomorphicencryption.org, το οποίο αποτελεί μια ανοιχτή συζήτηση, με σκοπό την επισημοποίηση κάποιων standard, που θα αφορούν το Homomorphic Encryption. Αυτό είναι κάτι ιδιαίτερα σημαντικό, καθώς δείχνει ακριβώς κάποιες αδυναμίες που υπάρχουν ακόμα στο Homomorphic Encrytpion. Για παράδειγμα, δεν μπορεί να πιστοποιήσει κάποιος ότι κάποιο scheme είναι όσο ασφαλές όσο υποστηρίζεται για αυτό, πράγμα που καταδεικνύει τις δυνατότητες εξέλιξης που έχει ακόμα ο χώρος αυτός.

2.2) Βασικές έννοιες του Homomorphic Encryption

Σε ένα σύγχρονο scheme κρυπτογράφησης, υπάρχουν μόνο δύο διαδικασίες-πράξεις που μπορούν να εφαρμοστούν σε δεδομένα, το storage και το retrieval. Για να γίνουν υπολογισμοί σε αυτά τα δεδομένα, θα πρέπει πρώτα να αποκρυπτογραφηθούν. Το Homomorphic Encryption, λειτουργεί σε επίπεδο κυκλώματος, πράγμα που σημαίνει πως οι συναρτήσεις που θα χρησιμοποιηθούν, θα πρέπει να αποτελούνται μόνο από δυαδικές πράξεις, όπως AND και XOR. Οι δύο αυτές πράξεις, μπορούν να αναπαρασταθούν ως εξής :

i) Η πράξη AND μπορεί να αναπαρασταθεί ως ο πολλαπλασιασμός 2 bits :

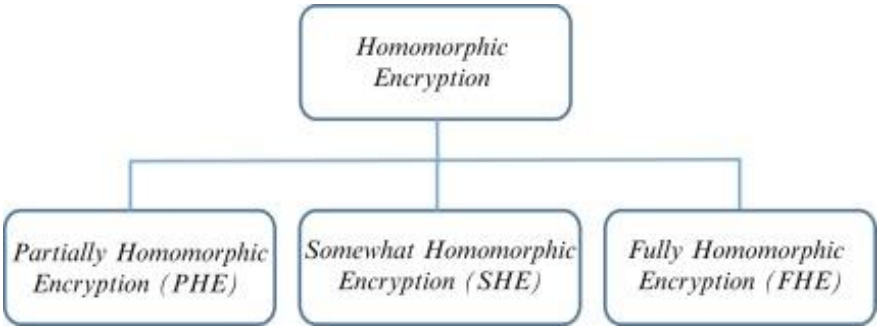
➔ ***AND (b1, b2) = b1 · b2, όπου b1, b2 : bits***

ii) Η πράξη XOR, μπορεί να αναπαρασταθεί ως το άθροισμα 2 bits modulo 2 :

➔ ***XOR (b1, b2) = (b1 + b2) mod 2 όπου b1, b2 : bits***

Το βάθος του κυκλώματος ή αλλιώς, το πόσες πράξεις εκτελούνται, μπορεί να επηρεάσει σημαντικά την αποτελεσματικότητα και την ορθότητα ενός Homomorphic Encryption scheme.

Το Homomorphic Encryption, ανάλογα με τις δυνατότητες του εκάστοτε scheme, μπορεί να κατηγοριοποιηθεί σε 3 βασικές υποκατηγορίες :



Εικόνα 2.1 Βασικές υποκατηγορίες του Homomorphic Encryption.

i) Το Partially Homomorphic Encryption (PHE), το οποίο περιλαμβάνει schemes υποστηρίζουν πράξεις σε επίπεδο κυκλώματος, μόνο μίας λογικής πύλης (***AND*** ή ***XOR***). Ενδεικτικά παραδείγματα schemes αυτής της υποκατηγορίας είναι το Unpadded RSA, το οποίο χρησιμοποιείται στο SSL/TLS, το EIGamal, το Goldwasser-Micali, το Bemaloh και το Pailler.

ii) Το Somewhat Homomorphic Encryption (SHE), το οποίο μπορεί να αξιοποιήσει δύο τύπους λογικών πυλών (***AND*** και ***XOR***), αλλά μόνο για ένα υποσύνολο κυκλωμάτων. Στην ουσία αυτό σημαίνει πως οι πράξεις μπορούν να εφαρμοστούν συγκεκριμένες φορές, οι οποίες καθορίζεται από αυτό το υποσύνολο κυκλωμάτων.

iii) Το Levelled Fully Homomorphic Encryption (LFHE), το οποίο υποστηρίζει την αξιολόγηση αυθαίρετων κυκλωμάτων που αποτελούνται από πολλαπλούς τύπους πυλών προκαθορισμένου, όμως, βάθους, που σημαίνει ότι οι πράξεις μπορούν να εφαρμοστούν για συγκεκριμένο αριθμό επαναλήψεων. Το Fully Homomorphic Encryption (FHE), επιτρέπει την αξιολόγηση αυθαίρετων κυκλωμάτων, που αποτελούνται από πολλαπλούς τύπους πυλών, απεριόριστου βάθους και είναι η ισχυρότερη έννοια του Homomorphic Encryption.

Γενικά, ένα Homomorphic Encryption scheme, αποτελείται από μια τετράδα (υπο)αλγορίθμων :

i) ***KeyGenenerator*** : Παίρνει σαν είσοδο μια παράμετρο ασφαλείας και βγάζει στην έξοδο ένα ζεύγος κλειδιών ***keys(pk, sk)*** .

ii) ***Encoder*** : Παίρνει σαν είσοδο το δημόσιο κλειδί ***pk*** και ένα μήνυμα $\boldsymbol{m} \in \boldsymbol{\{0,1\}}$ και βγάζει στην έξοδο ένα κρυπτογραφημένο κείμενο (***Ciphertext***) $\boldsymbol{c} \in \boldsymbol{C}$, όπου ***C*** ο χώρος των κρυπτογραφημένων κειμένων. Η διαδικασία αυτή μπορεί να συμβολιστεί και ως :

$$\boldsymbol{c \leftarrow Enc_{pk}(m)}$$

iii) ***Decoder*** : Παίρνει σαν είσοδο το κρυφό κλειδί ***sk*** και ένα κρυπτογραφημένο κείμενο (***Ciphertext***) $\boldsymbol{c} \in \boldsymbol{C}$ **.** Σαν έξοδος, αν έχει χρησιμοποιηθεί ο σωστός συνδυασμός κλειδιών, βγαίνει ένα μήνυμα $\boldsymbol{m} \in \boldsymbol{P}$,

όπου ***P*** ο χώρος των απλών κειμένων (***Plaintext***). Αυτή η διαδικασία μπορεί να συμβολιστεί και ως :

$$m := Dec_{sk}(c)$$

iv) ***Evaluator*** : Παίρνει σαν είσοδο το δημόσιο κλειδί ***pk, n*** κρυπτογραφημένα κείμενα (***Ciphertext***) $c_1, c_2, \dots, c_n \in C$, αλλά και μια συγκεκριμένη και επιτρεπόμενη συνάρτηση ***F***. Η έξοδος που προκύπτει με αυτόν τον (υπο)αλγόριθμο είναι η $F(c_1, c_2, \dots, c_n) \in C$. Αν $c_i = \{c_1, c_2, \dots, c_n\} \in C$ είναι το σύνολο των κρυπτογραφημένων κειμένων και $m_i = \{m_1, m_2, \dots, m_n) \in P$ είναι το αντίστοιχο σύνολο των αποκρυπτογραφημένων μηνυμάτων τους με ***P : Plaintext*** , τότε η αξιολόγηση που κάνει ο αλγόριθμος ***Evaluator*** θα είναι σωστή αν ισχύει :

$$Dec(Eval(F, c_i, pk), sk) = F(m_i) .$$

Ένα Homomorphic Encryption scheme, μπορεί να χαρακτηριστεί ως Fully Homomorphic Encryption scheme, αν εμπεριέχει αυτούς τους τέσσερις αλγορίθμους, αλλά και ικανοποιεί την ιδιότητα ότι η πράξη που εκτελείται, μπορεί να είναι μια αυθαίρετη συνάρτηση ***f***. Αν ένα κρυπτοσύστημα μπορεί να κρυπτογραφήσει μηνύματα $m \in \{0, 1\}$ και μπορεί να εκτελέσει την πράξη της πρόσθεσης και του πολλαπλασιασμού αποτελεσματικά, τότε ένα κρυπτοσύστημα μπορεί να χαρακτηριστεί ως Fully Homomorphic Encryption scheme, εάν ικανοποιεί τα παρακάτω :

i) ***Λειτουργικότητα (Functionality)*** : Αν ***S*** είναι ένα σύνολο έγκυρων κρυπτογραφημένων μηνυμάτων (***Ciphertexts***) και ***sk*** είναι ένα κρυφό κλειδί, τότε για $c_i \in S$, με $c_{add} = c_1 + c_2$, $c_{add} \in S$ και $c_{mult} = c_1 \cdot c_2$, $c_{mult} \in S$, τότε θα ισχύει και τα εξής :

a. $Dec(sk, c_{add}) = Dec(sk, c_1) + Dec(sk, c_2)$

b. $Dec(sk, c_{mult}) = Dec(sk, c_1) \cdot Dec(sk, c_2)$

ii) ***Αποτελεσματικότητα (Efficiency)*** : Για κάποια παράμετρο ασφαλείας λ, θα πρέπει να ισχύουν τα εξής :

a. Όλες οι πράξεις και οι συναρτήσεις (***KeyGenerator, Encoder, Decoder, Evaluation, Addition, Multiplication***) θα πρέπει να έχουν πολυωνυμικό χρόνο εκτέλεσης σε σχέση με την παράμετρο λ. Με άλλα λόγια, όλες οι συναρτήσεις θα πρέπει να μπορούν να εκτελούνται συμπαγώς.

b. Όλα τα έγκυρα κρυπτογραφημένα κείμενα (***Ciphertexts***) του scheme, θα πρέπει να έχουν πολυωνυμικό μέγεθος σε σχέση πάλι με την παράμετρο λ. Αυτό σημαίνει, ότι το μέγεθος του κρυπτογραφημένου

κειμένου, θα πρέπει να είναι ανεξάρτητο από την συνάρτηση/πράξη που αξιολογείται ομομορφικά.

Θα αναφερθούν επίσης κάποιες συναρτήσεις που χρησιμοποιούνται σε κάποιες γενιές του Fully Homomorphic Encryption και θα μας απασχολήσουν στη συνέχεια :

i) ***Bootstrapping (Recrypt)*** : Ανανεώνει το περιεχόμενο ενός κρυπτογραφημένου κειμένου (***Ciphertext***), τρέχοντας τον αλγόριθμο αποκρυπτογράφησης (***Decryptor***) ομομορφικά. Το ***Bootstrapping*** προτάθηκε πρώτα από τον Gentry και ως ***Recrypt***. Παίρνει ως είσοδο ένα δημόσιο κλειδί pk_2 και το κύκλωμα αποκρυπτογράφησης $D = \{Dec, \overrightarrow{sk1}, c_1\}$, όπου $\overrightarrow{sk1}$ ένα κρυφό κλειδί που κρυπτογραφήθηκε με το δημόσιο κλειδί pk_2. Αρχικά παράγεται το διάνυσμα $\overrightarrow{c1}$ χρησιμοποιώντας τον αλγόριθμο κρυπτογράφησης (***Encryptor***) $Enc(pk_2, c_{1j})$, στα ***j*** bits του κρυπτογραφημένου κειμένου c_1. Ως έξοδος, δίνεται το αποτέλεσμα του αλγορίθμου αξιολόγησης (***Evaluator***) $Eval(pk2, \overrightarrow{sk1}, \overrightarrow{c1})$, με την οποία εξαλείφεται αποτελεσματικά η επίδραση του pk_1 στο $\overrightarrow{c1}$ αφήνοντας το κρυπτογραφημένο μόνο με το pk_2 . Το ***Bootstrapping*** είναι μια μέθοδος που αξιοποιείται για τη μείωση θορύβου (noise reduction method) στα κρυπτογραφημένα κείμενα και είναι μια πολύ κοστοβόρα διαδικασία.

ii) ***Modulus Switching*** : Είναι μια εναλλακτική μέθοδος για τη μείωση θορύβου και χρησιμοποιείται στο BGV scheme, στο οποίο θα αναφερθούμε και στη συνέχεια. Χρησιμοποιεί τον αλγόριθμο αξιολόγησης (***Evaluator***), ο οποίος γνωρίζοντας κάποιο όριο του μήκους του κρυφού κλειδιού ***sk*** και όχι ολόκληρο το κλειδί, μπορεί να μετασχηματίσει ένα κρυπτογραφημένο κείμενο (***Ciphertext***), το $c \bmod q$ σε ένα διαφορετικό $c' \bmod p$ χωρίς να επηρεάζεται η ορθότητα του scheme, όπου p, q είναι περιττά moduli. Το ***Modulus Switching*** χρησιμοποιείται στο BGV.

iii) ***Relinearization*** : Εμφανίζεται στην δεύτερη γενιά του Fully Homomorphic Encryption και χρησιμοποιήθηκε για την υλοποίηση της πράξης του πολλαπλασιασμού που αναφέρθηκε παραπάνω. Παίρνει το μέγεθος του κρυπτογραφημένου κειμένου (***Ciphertext***) και ως έξοδο κατεβάζει το βαθμό του κρυπτογραφημένου κειμένου τουλάχιστον σε βαθμό = 2. Δηλαδή αν είχαμε για παράδειγμα, μετά την πράξη του πολλαπλασιασμού, ένα κρυπτογραφημένο κείμενο 3ου βαθμού, (c_0, c_1, c_2), με τη μέθοδο αυτή η έξοδος θα είναι ένα νέο κρυπτογραφημένο κείμενο 2ου βαθμού, το οποίο θα είναι το (c'_0, c'_1), το οποίο αν αποκρυπτογραφηθεί, θα δίνει το ίδιο αποτέλεσμα με το αρχικό. Για αυτήν τη διαδικασία, χρησιμοποιούνται κάποια δημόσια κλειδιά, τα ***relinearization keys***, των οποίων η παραγωγή εμπεριέχεται και γίνεται, αν χρειαστεί, στον αλγόριθμο ***Evaluator***. Το ***Relinearization***, αποτελεί μια αρκετά κοστοβόρα διαδικασία.

iv) ***Rotation*** : Κάνουμε μια σύντομη αναφορά και στην πράξη/συνάρτηση αυτή, η οποία μπορεί να χρησιμοποιηθεί προαιρετικά για την αύξηση της

ποιότητας της διαδικασίας, κατά την εκτέλεση κάποιου Homomorphic Encryption scheme. Η πράξη αυτή χρειάζεται κάποια δημόσια κλειδιά, τα οποία ονομάζονται ***galois keys***. Η παραγωγή τους αποτελεί μια ιδιαίτερα κοστοβόρα διαδικασία. Με την πράξη αυτή, γίνεται περιστροφή των θέσεων των κρυφών κλειδιών κατά έναν προκαθορισμένο αριθμό. Με τη διαδικασία αυτή, προστίθεται θόρυβος, ο οποίος όμως είναι αμελητέος στη βιβλιοθήκη που θα χρησιμοποιήσουμε, τη Microsoft SEAL.

υ) ***NTT*** : Αναφέρεται εδώ συνοπτικά, η μορφή ***NTT (Number Theoretic Transform)***. Η μορφή NTT είναι ένας διακριτός μετασχηματισμός Fourier (DFT), που γίνεται σε πεπερασμένα πεδία της μορφής Z/p, όπου p : prime (πρώτος αριθμός). Αν υπάρχει υπάρχει n-οστή ρίζα, τότε όταν το n διαιρεί το p − 1, έχουμε p = ξn +1¸ για κάποιον θετικό ακέραιο ξ. Αν ω είναι η (p-1)-οστή ρίζα, τότε μια n-οστή ρίζα a μπορεί να βρεθεί με την a = ωξ. Με αυτόν τον τρόπο, υπολογίζεται η μορφή NTT.

Αναφερόμαστε επίσης, στους αλγορίθμους κρυπτογράφησης συμμετρικού κλειδιού (symmetric-key encryption algorithms) και ασύμμετρου κλειδιού (asymmetric-key encryption algorithms).

Οι αλγόριθμοι κρυπτογράφησης συμμετρικού κλειδιού χρησιμοποιούν τα ίδια κρυπτογραφικά κλειδιά, για την κρυπτογράφηση και του απλού κειμένου (***Plaintext***), αλλά και για την αποκρυπτογράφηση του κρυπτογραφημένου κειμένου (***Ciphertext***). Τα κλειδιά αυτά μπορεί να ίδια ή να υπάρχει ένας πολύ απλός μετασχηματισμός με τα αρχικά, για την αξιοποίησή τους στις διαδικασίες κρυπτογράφησης και αποκρυπτογράφησης.

Οι αλγόριθμοι κρυπτογράφησης ασύμμετρου κλειδιού ή αλλιώς, οι αλγόριθμοι κρυπτογράφησης δημοσίου κλειδιού (public key encryption algorithms), χρησιμοποιεί ένα ζεύγος κλειδιών, που αποτελείται από το δημόσιο κλειδί και το κρυφό κλειδί (***public key, secret key***). Το δημόσιο κλειδί είναι γνωστό σε όλους, ενώ το κρυφό κλειδί το γνωρίζει μόνο ο ιδιοκτήτης/παραλήπτης των δεδομένων-μηνυμάτων που υπόκεινται σε κρυπτογράφηση. Σε αυτούς τους αλγορίθμους, οποιοσδήποτε αποστολέας μπορεί να κρυπτογραφήσει κάποιο μήνυμα χρησιμοποιώντας το δημόσιο κλειδί, αλλά μόνο ο παραλήπτης μπορεί να αποκρυπτογραφήσει το κρυπτογραφημένο μήνυμα, αξιοποιώντας το δικό του κρυφό κλειδί. Η παραγωγή αυτών των κλειδιών γίνεται με την αξιοποίηση κρυπτογραφικών αλγορίθμων, που βασίζονται σε μαθηματικά προβλήματα τα οποία χαρακτηρίζονται και ως one-way functions, λόγω της ευκολίας τους να υπολογιστούν με κάποια συγκεκριμένη είσοδο, αλλά στη δυσκολία τους να αναστραφεί ο υπολογισμός αυτός, δηλαδή έχοντας μόνο την έξοδό τους. Παρακάτω περιγράφονται δύο βασικά προβλήματα που αφορούν σημαντικά το Homomorphic Encryption.

Ένα Homomorphic Encryption scheme, δεν είναι απαραίτητο να βασίζεται σε ασύμμετρη κρυπτογράφηση κλειδιού. Στην ουσία, στα schemes αυτά δεν υπάρχει κάποια ουσιαστική διαφορά μεταξύ των συμμετρικών και ασύμμετρων από άποψη ασφάλειας, λόγω της αξιοποίησης των ομομορφικών ιδιοτήτων. Η μετατροπή από ένα ασύμμετρο σε ένα συμμετρικό Homomorphic Encryption scheme είναι ιδιαίτερα απλή. Αυτό γίνεται προσθέτοντας το κλειδί κρυπτογράφησης στο τέλος κάθε

κρυπτογραφημένου κειμένου. Υπάρχει όμως τρόπος μετατροπής ενός συμμετρικού Homomorphic Encryption scheme σε ασύμμετρο. Το δημόσιο κλειδί $\boldsymbol{pk}$ είναι μια σειρά από bits $\boldsymbol{\{b_1, \ldots, b_l\}}$ και $\boldsymbol{l}$ είναι η κρυπτογράφηση αυτών των bits $\boldsymbol{\{c_1, \ldots, c_l\}}$, ή αλλιώς $\boldsymbol{pk = \{(b_1, c_1), \ldots, (b_l, c_l)\}}$, με $\boldsymbol{c_i = Enc_{sk}(b_i)}$. Επιλέγεται ο αριθμός $\boldsymbol{l}$ πολύ μεγαλύτερος από το μέγεθος του κρυπτογραφημένου κειμένου (***ciphertext***), δηλαδή $\boldsymbol{l >> |c_i|}$. Για την κρυπτογράφηση κάποιου bit $\boldsymbol{\sigma}$, πρώτα επιλέγεται ένα κατάλληλο και τυχαίο string από bits $\vec{\boldsymbol{r}}$ τέτοιο ώστε το εσωτερικό γινόμενο μεταξύ του $\boldsymbol{r_i}$ και του $\boldsymbol{b_i}$, να είναι ίσο με $\boldsymbol{\sigma}$ ή αλλιώς $\sum r_i b_i = \sigma$. Το ομομορφικό εσωτερικό γινόμενο, δηλαδή το εσωτερικό γινόμενο modulo 2 θα είναι το $c = \sum_{r_i=1} c_i = \boldsymbol{Enc(}\sum r_i b_i\boldsymbol{)}$. Έτσι, λοιπόν, μπορεί να κρυπτογραφηθεί ένα μήνυμα μετατρέποντας το ένα συμμετρικό scheme σε ασύμμετρο.

2.3) Lattice-based Cryptography

Η Κρυπτογραφία βασισμένη σε πλέγμα (Lattice-based Cryptography), αποτελεί ένα σημαντικό κομμάτι της ιστορίας του Homomorphic Encryption, το οποίο αξιοποιήθηκε από τον Gentry το 2009 για μπορέσει να περιγράψει με έναν πρωτόλειο τρόπο την πιθανή δημιουργία ενός Fully Homomorphic Encryption scheme. Όπως και το παρακάτω πρόβλημα του (Ring) Learning With Errors, η κρυπτογραφία αυτή εκτιμάται ότι θα αποτελέσει σημαντικό κομμάτι για την ασφάλεια απέναντι σε κυβερνοεπιθέσεις στο μέλλον, που θα προκύπτουν από κρυπτοαναλύσεις κβαντικών υπολογιστών. Αυτό προκύπτει από την παραδοχή, ότι κάποια συγκεκριμένα υπολογιστικά προβλήματα πλέγματος δεν μπορούν να λυθούν με αποτελεσματικό τρόπο. Ακολουθεί μια συνοπτική ανάλυση του ζητήματος :

Ένα πλέγμα $\boldsymbol{L \subset R}$ είναι το σύνολο όλων των ακεραίων γραμμικών συνδυασμών διανυσμάτων $\boldsymbol{b}$ που αποτελούν βάσεις στον $\boldsymbol{R^n}$ διανυσματικό χώρο, με $\boldsymbol{b_1, \ldots, b_n \in R^n}$, δηλαδή για το σύνολο αυτό ισχύει ότι $\boldsymbol{L = \{\sum a_i b_i, a_i \in Z\}}$. Για παράδειγμα, το $\boldsymbol{Z^n}$ είναι ένα πλέγμα το οποίο παράγεται από ορθοκανονική βάση για το $\boldsymbol{R^n}$. Η βάση για το πλέγμα δεν είναι μοναδική. Μπορούν για παράδειγμα να υπάρχουν διάφορες βάσεις για ένα πλέγμα $\boldsymbol{Z^4}$.

2.4) (Ring) Learning With Errors (LWE, RLWE)

Εδώ θα αναφερθεί συνοπτικά το πρόβλημα του Learning With Errors (LWE) που έπαιξε καθοριστικό ρόλο στην σχεδίαση και ανάπτυξη των schemes, αλλά και το Ring Learning With Errors (RLWE), το οποίο αποτελεί μια παραλλαγή του LWE βασισμένο σε δακτύλιο.

Η βασική ιδέα γύρω από το LWE, είναι το πρόβλημα του να βρεθεί ένα κρυφό κλειδί $\boldsymbol{sk}$, δεδομένου μιας γραμμικής εξίσωσης με θόρυβο, της μορφής $\boldsymbol{b = a\, sk + e}$, όπου $\boldsymbol{e}$ ***: error***, με δεδομένη την πρόσβαση σε πολυωνιμικά πολλά δείγματα επιλογής ζευγαριών των υπόλοιπων όρων. Εάν δεν υπήρχε ο όρος $\boldsymbol{e}$, το κρυφό κλειδί $\boldsymbol{sk}$, θα μπορούσε να βρεθεί με τη μέθοδο της Γκαουσσιανής Εξάλειψης (Gaussian Elimination), που αποτελεί έναν αλγόριθμο επίλυσης συστημάτων γραμμικών εξισώσεων. Η εισαγωγή του όρου του σφάλματος, δυσκολεύει αρκετά αυτήν την επίλυση του προβλήματος αυτού. Αναπτύχθηκε το 2005 από τον Regev και λόγω της δυσκολίας των υπολογισμών για την επίλυσή του, αποτελεί χρήσιμο κομμάτι της

Κρυπτογραφίας, αλλά και βοήθησε καταλυτικά στην ανάπτυξη Fully Homomorphic Encryption schemes.

Learning With Errors : Για θετικούς ακεραίους n και q, μια διακριτή Γκαουσσιανή κατανομή χ και ένα διάνυσμα s, που επιλέγεται σαν ένα n – διαστάσεων ακέραιο διάνυσμα, στο οποίο εφαρμόζεται η πράξη modulo **q**, δηλαδή $s \in Z_q^n$ τότε η κατανομή LWE, που συμβολίζεται ως $A_{s,\chi}$, θα παράγει ένα δείγμα (a,b), επιλέγοντας ένα ομοιόμορφα τυχαίο $a \in Z_q^n$, επιλέγοντας ένα $e \leftarrow \chi$ και υπολογίζοντας την πράξη $b = \langle a\ s \rangle + e$.

Το πρόβλημα Ring Learning With Errors (RLWE), παρουσιάστηκε το 2013 από τον Lyubaskevsky, Peikert και Regev. Κομμάτι αυτής διαδικασίας και πιο συγκεκριμένα στην διαδικασία παραγωγής δημοσίων κλειδιών του Microsoft SEAL και την επιβεβαίωση της ορθότητας τους, κάνουμε accelerate σε αυτήν τη διπλωματική εργασία.

Ring Learning With Errors : Για κάποια παράμετρο ασφαλείας λ , με την $f(x)$ να είναι ένα κυκλοτομικό πολυώνυμο $\Phi_m(x)$, με βαθμό $deg(f) = \rho(m)$, το οποίο θα εξαρτάται από την παράμετρο λ, ένα σύνολο $R = Z[x] / f(x)$ και για έναν ακέραιο $q = q(\lambda) \geq 2$. Για κάποιο τυχαίο στοιχείο $s \in R_q$ και μια κατανομή $\chi = \chi(\lambda)$ στο R, η κατανομή που προκύπτει από την επιλογή ενός ομοιόμορφα τυχαίου $a \leftarrow R_q$ και ενός όρου θορύβου $e \leftarrow \chi$, θα συμβολίζεται ως $A_{s,x^{(q)}}$ και θα έχει ως έξοδο το ζεύγος $(a, [a \cdot s + e]_q)$.

Το πρόβλημα Ring Learning With Errors, εκτιμάται, μεταξύ άλλων, ότι θα χρησιμοποιηθεί για την προστασία από την κρυπτοανάλυση από κβαντικούς υπολογιστές.

Κεφάλαιο 3 : Γενιές του Fully Homomorphic Encryption

3.1) Εισαγωγικά – Pre-FHE

Ανάλογα με κάποιους παράγοντες, που επηρεάζουν σημαντικά την ταχύτητα και την αποτελεσματικότητα της διαδικασίας του Fully Homomorphic Encryption, τα schemes τα οποία έχουν δημιουργηθεί, μπορούν να διαχωριστούν σε κάποιες γενιές. Στη συνέχεια θα γίνει ανάλυση κάποιων σημαντικών schemes αυτών των γενιών. Ξεκινώντας, θα γίνει μια σύντομη ανάλυση κάποιων σημαντικών schemes πριν το 2009. Όπως έχει ήδη αναφερθεί, για περίπου 30 χρόνια από το 1978, το πρόβλημα της δημιουργίας ενός Fully Homomorphic Encryption scheme παρέμενε ανοιχτό. Σε αυτό το διάστημα αναφέρονται ενδεικτικά κάποια schemes, τα οποία αποτελούν κομμάτι του Partially Homomorphic Encryption (PHE) και τοποθετούνται χρονικά στο διάστημα πριν της δημιουργίας του πρώτου Fully Homomorphic Encryption Scheme. Πιο συγκεκριμένα, θα αναφερθούν οι ομομορφικές ιδιότητες (***Homomorphic Properties***) αυτών των schemes. Ομομορφισμός είναι μια απεικόνιση μεταξύ δυο αλγεβρικών δομών. Με τον όρο ομομορφικές ιδιότητες, αναφερόμαστε στη σχέση που θα υπάρξει μεταξύ αυτών των αλγεβρικών δομών, μεταφέρνοντάς την στον χώρο της Κρυπτογραφίας. Οι αποδείξεις των ιδιοτήτων αυτών, βρίσκονται στο Appendix.

i) ***Unpadded RSA*** : Αν το ***e*** αποτελεί στοιχείο της κρυπτογράφησης (***encryption component***)***, n : modulus (public key), m :*** μήνυμα (***message***)***,*** τότε η κρυπτογράφηση του μηνύματος προκύπτει από την $E(m) = m^e \bmod n$. Σε αυτήν την περίπτωση, η ομομορφική ιδιότητα είναι :

$$E(m_1) \cdot E(m_2) = E(m_1 \cdot m_2)$$

Δηλαδή το γινόμενο της κρυπτογράφησης δύο μηνυμάτων ισούται με την κρυπτογράφηση του γινομένου των μηνυμάτων αυτών.

ii) ***EIGamal*** : Για μια κυκλική ομαδοποίηση ***G*** σε σειρά ***q*** γεννήτρια (κλειδιών) ***g***, αν το δημόσιο κλειδί (***public key***) είναι το ***(G, q, g, h)***, όπου $h = g^x$ και ***x*** είναι το κρυφό κλειδί, τότε η κρυπτογράφηση του μηνύματος είναι η $E(m) = (g^r, m \cdot h^r)$, όπου το ***r*** αποτελεί κάποια τυχαία τιμή και ανήκει στο σύνολο $r \in \{0, \ldots, q-1\}$. Η ομομορφική ιδιότητα σε αυτήν την περίπτωση είναι όμοια με την παραπάνω, δηλαδή :

$$E(m_1) \cdot E(m_2) = E(m_1 \cdot m_2)$$

iii) ***Goldwasser – Micali*** : Αν το δημόσιο κλειδί (***public key***) είναι το ***n modulus*** και το ***x*** είναι το τετραγωνικό μη-υπόλοιπο (quadratic non-residue), δηλαδή δεν υπάρχει ακέραιος αριθμός ($\nexists y$) τέτοιος ώστε $y^2 \equiv x \bmod n$, τότε, η κρυπτογράφηση ενός bit ***b*** θα είναι $E(b) = x^b r^2 \bmod n$, για κάποια τυχαία τιμή του $r \in \{0, \ldots, n-1\}$. Η ομομορφική ιδιότητα τότε είναι :

$$E(b_1) \cdot E(b_2) = E(b_1 \oplus b_2)$$

Δηλαδή το γινόμενο της κρυπτογράφησης δύο bit του μηνύματος ισούται με την κρυπτογράφηση του αμοιβαίου αποκλεισμού των 2 αυτών bits.

iv) ***Benaloh*** : Αν το δημόσιο κλειδί είναι το ***n modulus*** και το ***g*** αποτελεί μια βάση με block size ***c***, δηλαδή ένα συμμετρικό κλειδί, που παράγεται με τη γεννήτρια ***g***, έχει μήκος bit string ίσο με ***c***, τότε η κρυπτογράφηση του μηνύματος ***m*** θα είναι στο συγκεκριμένο scheme $E(m) = g^m r^c$, για κάποια τυχαία τιμή του $r \in \{0, \ldots, n-1\}$. Η ομομορφική ιδιότητα σε αυτήν την περίπτωση είναι :

$$E(m_1) \cdot E(m_2) = E(m_1 + m_2 \bmod c)$$

Ή αλλιώς, το γινόμενο της κρυπτογράφησης δύο μηνυμάτων, είναι ίσο με την κρυπτογράφηση του αθροίσματος του πρώτου μηνύματος με το υπόλοιπο της διαίρεσης του περιεχομένου (σε δεκαδικό σύστημα) του δεύτερου μηνύματος με το μήκος του bit string του συμμετρικού κλειδιού του.

v) ***Paillier*** : Αν το δημόσιο κλειδί είναι το ***modulus n*** και η βάση (γεννήτρια) είναι η ***g*** τότε η κρυπτογράφηση του μηνύματος ***m*** είναι η $E(m) = g^m r^n \bmod n^2$, για κάποια τυχαία τιμή $r \in \{0, \ldots, n-1\}$. Η ομομορφική ιδιότητα σε αυτήν την περίπτωση είναι :

$$E(m_1) \cdot E(m_2) = E(m_1 + m_2)$$

Δηλαδή το γινόμενο της κρυπτογράφησης δύο μηνυμάτων, είναι ίσο με την κρυπτογράφηση του αθροίσματος του περιεχομένου των μηνυμάτων αυτών.

Τα παραπάνω schemes και οι ομομορφικές τους ιδιότητες αναφέρονται, για την καλύτερη κατανόηση του Homomorphic Encryption σαν έννοια, αλλά και για το πώς μπορεί να εκφραστεί η έννοια του ομομορφισμού στον τομέα της κρυπτογραφίας και του Homomorphic Encryption.

Παρακάτω, ακολουθεί η κατηγοριοποίηση των διαφόρων Fully Homomorphic Encryption schemes, ανάλογα με τα διάφορα χαρακτηριστικά τους, τον σκοπό και τις καινοτομίες που κάθε γενιά εισήγαγε, αλλά και το τι πράξεις υποστήριζε το κάθε scheme. Όλες οι γενιές έπαιξαν σημαντικό ρόλο στην ανάπτυξη του Fully Homomorphic Encryption, στα πλαίσια όμως αυτής της διπλωματικής, θεωρούμε ιδιαίτερα σημαντική την 4η γενιά και το CKKS scheme.

3.2) 1st Generation of FHE

Στην πρώτη γενιά του Fully Homomorphic Encryption, συναντάμε αρχικά το scheme του Gentry, αξιοποιώντας την κρυπτογραφία (ιδανικού) πλέγματος (Ideal Lattice-based Cryptography). Αυτό, υποστηρίζει τις πράξεις της πρόσθεσης και του πολλαπλασιασμού που αναπτύχθηκαν και παραπάνω, σε κρυπτογραφημένα κείμενα

(***Ciphertexts***), με τις οποίες καθίσταται δυνατή η κατασκευή κυκλωμάτων για την υλοποίηση αυθαίρετων υπολογισμών, που όπως περιεγράφηκε αποτελεί σημαντικό διαφοροποιητικό στοιχείο, μεταξύ των διαφόρων ειδών των Homomorphic Encryption schemes και αποτελεί βασικό χαρακτηριστικό των Fully Homomorphic Encryption schemes.

Η κατασκευή αυτή, ξεκινάει με την παραδοχή ότι οι πράξεις αυτές αποτελούν κομμάτι ενός Somewhat Homomorphic Encryption scheme, από την άποψη ότι μπορούν, στο συγκεκριμένο scheme, να γίνουν υπολογισμοί σε πολυώνυμα χαμηλού βαθμού σε κρυπτογραφημένα δεδομένα, με την κάθε πράξη όμως, (είτε πρόσθεσης είτε πολλαπλασιασμού) να προσθέτει έναν όρο θορύβου και ως αποτέλεσμα, μετά από έναν αριθμό πράξεων, να μην είναι δυνατή η αποκρυπτογράφηση του κρυπτογραφημένου κειμένου. Για να λύσει το πρόβλημα αυτό, ο Gentry τροποποιεί κατάλληλα αυτό το scheme, για να εισαγάγει τη δυνατότητα αξιοποίησης της συνάρτησης ***Bootstrapping***, στην οποία έχει ήδη γίνει αναφορά. Αυτό δίνει τη δυνατότητα στο scheme αυτό, να μπορεί να κάνει άλλη μια πράξη, πέρα από την αξιολόγηση με τον ***Evaluator*** του κυκλώματος, μειώνοντας έτσι και τον θόρυβο που προστίθεται στο πέρας κάθε πράξης, λόγω και της ανανέωσης του κρυπτογραφημένου κειμένου, όποτε ο θόρυβος μεγαλώνει πολύ. Στην ουσία, σε αυτό το scheme, η συνάρτηση αυτή, ανανεώνει κατά τη διαδικασία αποκρυπτογράφησης το κρυπτογραφημένο κείμενο ομομορφικά και έτσι προκύπτει ένα νέο κρυπτογραφημένο κείμενο, το οποίο είναι κρυπτογραφημένο για το ίδιο περιεχόμενο, όπως το αρχικό, αλλά έχει μικρότερο θόρυβο. Αποδεικνύει, έτσι, ότι κάθε Somewhat Homomorphic Encryption Scheme, στο οποίο υπάρχει η δυνατότητα να εφαρμοστεί η συνάρτηση ***Bootstrapping***, μπορεί να μετασχηματιστεί σε Fully Homomorphic Encryption scheme. Σε αυτήν την υλοποίηση από τον Gentry και τον Halevi, χρειάζονται περίπου 30 λεπτά για να μπορέσει να γίνει μια πράξη σε ένα bit μηνύματος. Σε αυτήν τη γενιά, αναφέρουμε και ένα άλλο scheme, από τους Van Dijk, Gentry, Halevi και Vaikuntanathan, το οποίο βασίζεται στο scheme του Gentry, χωρίς όμως να απαιτεί την ύπαρξη ιδανικών πλεγμάτων. Τα αποτελέσματα σε σχέση με την ταχύτητα αυτού του scheme, είναι παρόμοια με αυτά του πρώτου scheme, του Gentry.

3.3) 2nd Generation of FHE

Στη δεύτερη γενιά των Fully Homomorphic Encryption schemes, έχουμε πληθώρα από αυτά, με αρκετά διαφορετικά χαρακτηριστικά μεταξύ τους. Το βασικό στοιχείο, που αποτελεί ειδοποιό διαφορά σε σχέση με την πρώτη γενιά, είναι ο πολύ μικρότερος ρυθμός αύξησης του θορύβου που παράγεται, κατά τη διάρκεια εκτέλεσης των υπολογισμών, που έκανε την πρώτη γενιά αρκετά αργή και περίπλοκη. Τα περισσότερα schemes αυτής της γενιάς χρησιμοποιούν την δυσκολία του προβλήματος (Ring) Learning With Errors, για μεγαλύτερη ασφάλεια, με εξαίρεση το ***LTV*** (Lopez-Alt, Tromer, Vaikuntanathan, 2012), το οποίο έχει γίνει accelerate με FPGA και το ***BLLN*** (Bos, Lauter, Loftus, Naehrig, 2013), τα οποία βασίζονται στο NTRU. Χωρίς να αναλωθούμε ιδιαίτερα στην ανάλυση του NTRU, αποτελεί ένα κρυπτοσύστημα ανοιχτού πηγαίου κώδικα, το οποίο είναι βασισμένο στην κρυπτογραφία βασισμένη σε πλέγμα (Lattice-Based Cryptography). Τα schemes που βασίζονται στο NTRU, δε χρησιμοποιούνται πλέον, λόγω του ότι αποτέλεσαν στόχο σε επιθέσεις σχετικές με το υπό-πεδίο του πλέγματος, δηλαδή κανονικοποιώντας το

δημόσιο κλειδί σε ένα υπό-πεδίο, το οποίο κάνει το πρόβλημα της κρυπτογράφησης βασισμένη σε πλέγμα πιο εύκολο στην επίλυση.

Το ***BLLN***, είχε μια παρόμοια λογική με αυτήν του Gentry στην προηγούμενη γενιά, δηλαδή ότι ένα Somewhat Homomorphic Encryption scheme, μπορεί να αποτελέσει ένα Fully Homomorphic Encryption scheme, εάν σε αυτό εφαρμοστεί η συνάρτηση ***Bootstrapping***. Πέρα από αυτήν όμως τη συνάρτηση, το ***BLLN*** μπορεί να αξιοποιήσει και τη συνάρτηση ***Modulus Switching***, για να μειωθεί ο ρυθμός αύξησης του θορύβου και να μπορέσει να αποτελέσει και αυτό ένα Fully Homomorphic Encryption scheme. Άλλα schemes αυτής της γενιάς που ξεχωρίζουν, είναι το ***BGV*** (Brakerski-Gentry-Vaikuntanathan, 2011) και το ***BFV*** (Brakerski/Fan-Vercauteren, 2012).

Το ***BGV,*** αξιοποιεί την συνάρτηση ***Relinearization***. Μάλιστα, αποδεικνύεται ότι μπορεί να προκύψει ένα Somewhat Homomorphic Encryption scheme, εάν η παραγωγή του βασιστεί στο πρόβλημα του Learning With Errors (LWE). Χρησιμοποιώντας αυτήν την τεχνική, δεν καθίσταται αναγκαία η επιλογή ενός ιδανικού δακτυλίου για την επίλυση του προβλήματος LWE. Παρουσιάστηκε επίσης μια τεχνική που ονομάστηκε ***dimension-modulus reduction*** (***Modulus Switching***), η οποία καταφέρνει τον μετασχηματισμό του παραγόμενου Somewhat Homomorphic Encryption scheme σε Fully Homomorphic Encryption scheme, επιτρέποντας το ***Bootstrapping***.

Το ***BFV***, καταδεικνύει κυρίως ένα συστατικό που χαρακτηρίζει αυτήν τη γενιά, σχετικά με την πολυπλοκότητα των schemes. Βασική παρατήρηση σε αυτό το scheme, αποτελεί ότι ο θόρυβος ενός κρυπτογραφημένου κειμένου (***Ciphertext***), ενώ στα περισσότερα schemes προηγούμενων γενιών μεγάλωνε τετραγωνικά με κάθε πολλαπλασιασμό (***B-> B² · poly(n)***), σε αυτό το scheme μεγαλώνει μόνο γραμμικά (***B-> B² · poly(n)***). Ακόμη, σε αυτό το scheme δεν είναι αναγκαία η χρήση του ***modulus switching*** σε σχέση με άλλα schemes αυτής της γενιάς.

Μια σημαντική παρατήρηση για αυτήν τη γενιά του Fully Homomorphic Encryption, αποτελεί, ότι όλα τα schemes βασίζονται στη λογική την οποία πρότεινε ο Gentry στο 1st gen FHE. Σημαντική, ακόμα, αποτελεί η παρατήρηση ότι τα schemes αυτής της γενιάς είναι αρκετά αποτελεσματικά, ακόμα και χωρίς το ***Bootstrapping***, επιτρέποντας τη λειτουργία ως Levelled Fully Homomorphic schemes, στο οποίο έχουμε ήδη αναφερθεί. Γενικά, η πολυπλοκότητα των schemes σε αυτήν τη γενιά, για ***T*** υπολογισμούς σε κρυπτογραφημένα δεδομένα και με ***k*** μια παράμετρος ασφαλείας (***security parameter***), είναι ***T · polylog(k)***. Τέλος, τα schemes σε αυτήν τη γενιά λειτουργούν με τη λογική ***SIMD (Single Instruction Multiple Data)***, όπως καταδεικνύεται αναλυτικά στη δημοσίευση “Fully Homomorphic SIMD Operations”.

3.4) 3rd Generation of FHE

Στην τρίτη γενιά των Fully Homomorphic Encryption schemes, έχουμε κι άλλες βελτιστοποιήσεις, που αποτελούν σημαντικά βήματα μείωσης του ρυθμού αύξησης του θορύβου, πράγμα που οδηγεί σε μεγαλύτερη αποτελεσματικότητα και ισχυρότερη ασφάλεια. Σε αυτήν τη γενιά, έχουμε ένα βασικό scheme, το ***GSW*** (Gentry, Sahai, Waters, 2013) και δύο παραλλαγές του το ***FHEW*** (2014) και το ***TFHE*** (2016).

Το ***GSW***, βασίζεται και αυτό στο πρόβλημα Learning With Errors (LWE). Έχει ως βασικό χαρακτηριστικό την αποφυγή του ακριβού βήματος ***Relinearization*** στην πράξη του πολλαπλασιασμού. Αυτό επιτυγχάνεται με την προτεινόμενη μέθοδο των κατά προσέγγιση ιδιοδιανυσμάτων (***approximate eigenvector method***). Με αυτόν τον τρόπο, οι περίπλοκες πράξεις της ομομορφικής πρόσθεσης και ομομορφικού πολλαπλασιασμού, μετασχηματίζονται σε απλές πράξεις πινάκων (πρόσθεσης και πολλαπλασιασμού), το οποίο όχι μόνο κάνει τους υπολογισμούς γρηγορότερους, αλλά και εύκολους στην κατανόηση. Ακόμη, σε προηγούμενα schemes, ο ***Evaluator*** για να προβεί σε πράξεις, χρειαζόταν να γνωρίζει το δημόσιο κλειδί του χρήστη. Αυτό ονομάζεται και ως κλειδί αξιολόγησης (***Evaluation key***) και αποτελείται από μια αλυσίδα κρυπτογραφημένων κρυφών κλειδιών. Σε αυτό το scheme, δεν υπάρχει καθόλου κλειδί αξιολόγησης και οι ομομορφικές πράξεις πραγματοποιούνται γνωρίζοντας μερικές απλές παραμέτρους ασφαλείας.

Όσον αφορά το ***FHEW*** και το ***TFHE***, αποτελούν παραλλαγές του ***GSW*** που χρησιμοποιούν το πρόβλημα Ring Learning With Errors, δηλαδή αποτελούν παραλλαγές που αξιοποιούν το δακτύλιο, για την περαιτέρω βελτίωση της ταχύτητας των schemes. Πιο συγκεκριμένα, το ***FHEW*** ήταν το πρώτο scheme στο οποίο καταδεικνύεται πως η ανανέωση των κρυπτογραφημένων κειμένων (***Ciphertexts***), μετά το πέρας κάθε πράξης, συμβάλλει στην αισθητή μείωση του χρόνου που χρειάζεται για να γίνει το ***Bootstrapping***, σε δέκατα του δευτερολέπτου. Να σημειώσουμε, ότι η ίδια η ανανέωση του κρυπτογραφημένου κειμένου, όπως έχει προαναφερθεί, γίνεται με το ***Bootstrapping***. Αυτό επιτυγχάνεται, λόγω του ότι χρησιμοποιείται μια παραλλαγή του ***Bootstrapping***, η οποία ακολουθεί μια αριθμητική διαδικασία, αντί της διαδικασίας λογικών πράξεων ως υλοποίηση του αλγορίθμου αυτού. Ως συνέχεια, η αποτελεσματικότητα του ***FHEW*** βελτιώθηκε με το ***TFHE*** scheme, το οποίο πάλι έχει μια παραλλαγή του ***Bootstrapping*** με δακτύλιο, χρησιμοποιώντας μια μέθοδο παρόμοια με το ***FHEW***.

Παρακάτω ακολουθεί η 4η γενιά των Fully Homomorphic Encryption schemes, που θα δώσουμε ιδιαίτερη βάση, μιας και σκοπός της εργασίας είναι η επιτάχυνση με FPGA, κομματιού του αλγορίθμου αυτού, όπως αυτός είναι υλοποιημένος στη βιβλιοθήκη Microsoft SEAL. Συνεπώς, η περαιτέρω ανάλυση του παρακάτω αλγορίθμου στο επόμενο κεφάλαιο είναι ιδιαίτερα σημαντική, για την καλύτερη κατανόηση του τρόπου λειτουργίας του scheme αυτού.

3.5) 4th Generation of FHE - CKKS

Ένα αρνητικό και του ***BGV*** και του ***BFV*** της δεύτερης γενιάς του Fully Homomorphic Encryption είναι πως έχουν τη δυνατότητα να υλοποιούν πράξεις μόνο σε ακεραίους, δηλαδή υποστηρίζουν υπολογισμούς λογικούς, με ακεραίους ή με modulo. Αυτό τα κάνουν μη πρακτικά, καθώς τα δεδομένα του πραγματικού κόσμου ανήκουν σε συνεχή σύνολα, όπως στο σύνολο των πραγματικών αριθμών ή στην επέκτασή του, στο σύνολο των μιγαδικών αριθμών. Η 4η γενιά του Fully Homomorphic Encryption και το ***CKKS*** (Jung Hee Cheon, Andrey Kim, Miran Kim, Yongsoo Song, 2017), υποστηρίζει υπολογισμούς σε μιγαδικούς αριθμούς με περιορισμένη ακρίβεια, αντιμετωπίζοντας το θόρυβο κρυπτογράφησης σαν κομμάτι του σφάλματος, το οποίο προκύπτει έτσι κι αλλιώς, στους υπολογισμούς στρογγυλοποίησης. Για αυτόν το λόγο,

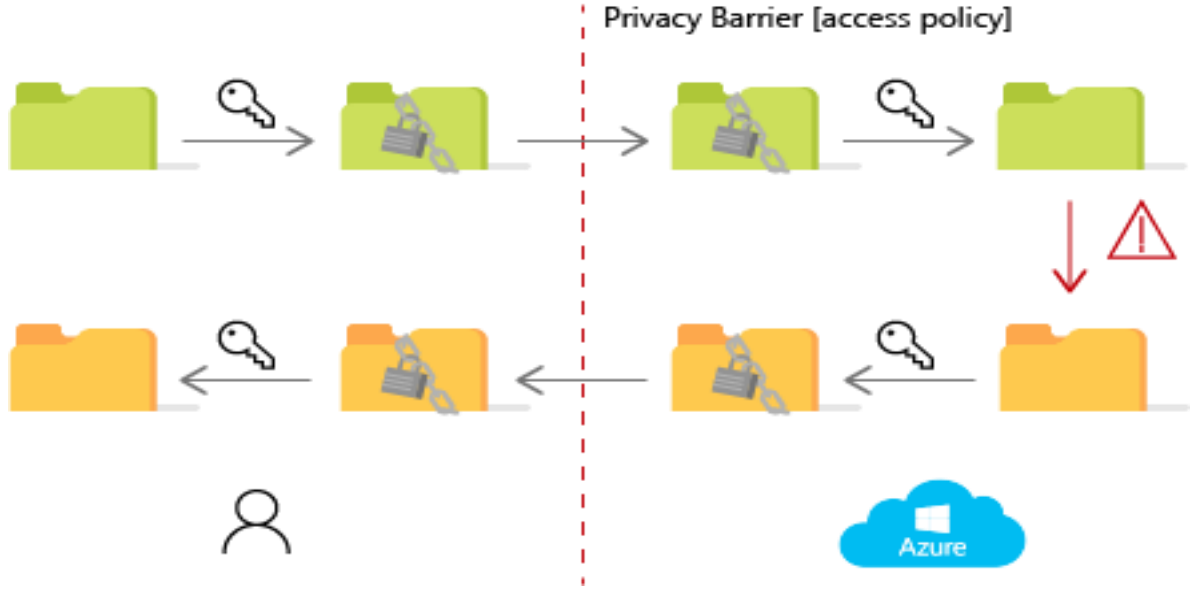
το ***CKKS***, που αποτελεί ένα Fully Homomorphic Encryption scheme, θα μπορούσε να κατηγοριοποιηθεί και ως ένα Approximate Fully Homomorphic Encryption scheme. Το αρχικό scheme, ονομάστηκε HEAAN και εν συνεχεία δημιουργήθηκε και βιβλιοθήκη με το ίδιο όνομα, σαν τη Microsoft SEAL, υλοποιώντας μόνο τον αλγόριθμο του ***CKKS***. Για τις βιβλιοθήκες, όμως θα γίνει λόγος στο επόμενο κεφάλαιο.

Το ***CKKS*** υποστηρίζει τις πράξεις προσέγγισης (approximation) της πρόσθεσης και του πολλαπλασιασμού κρυπτογραφημένων μηνυμάτων, αλλά και εισάγει μια νέα διαδικασία/συνάρτηση, τη συνάρτηση ***Rescaling***, για την καλύτερη διαχείριση του μεγέθους των απλών κειμένων (***Plaintexts***). Στην ουσία, η διαδικασία αυτή «ακρωτηριάζει» το κρυπτογραφημένο μήνυμα (***Ciphertext***) σε μέγεθος με μικρότερο modulus, το οποίο οδηγεί και στην στρογγυλοποίηση του μεγέθους του απλού κειμένου προς τα κάτω. Σε αυτό το απλό κείμενο, προστίθεται θόρυβος για ασφάλεια, αλλά, όπως έχει ήδη αναφερθεί, είναι κομμάτι σφάλματος που προέκυπτε κατά την εκτέλεση (προσεγγιστικών) υπολογισμών. Αυτό το σφάλμα μειώνεται με το ***Rescaling***. Σαν αποτέλεσμα, η διαδικασία αποκρυπτογράφησης του συγκεκριμένου scheme, δίνει μια προσέγγιση του απλού κειμένου με προκαθορισμένη ακρίβεια. Ακόμη είναι σημαντικό να σημειωθεί, ότι στο συγκεκριμένο scheme το μέγεθος σε bit του ***modulus*** του κρυπτογραφημένου κειμένου (***Ciphertext***) αυξάνεται γραμμικά σε σχέση με το βάθος του κυκλώματος που αξιολογείται με την διαδικασία του ***Rescaling***, ενώ τα προηγούμενα schemes απαιτούσαν είτε ένα ***modulus*** εκθετικά μεγάλου μεγέθους ή κάποιον περίπλοκο και κοστοβόρο υπολογισμό, όπως είναι το ***Bootstrapping***. Η αρχική δημοσίευση και το scheme, αποτελούσε ένα Levelled Fully Homomorphic Encryption scheme. Από τότε όμως, έχει δημοσιευτεί από τους δημιουργούς μια παραλλαγή του scheme, για να υποστηρίζει (full) ***RNS (Residue Number System)***, αλλά και έχει προστεθεί η συνάρτηση ***Bootstrapping***, πράγμα που καθιστά το ***CKKS***, ένα Fully Homomorphic Encryption scheme. Λόγω της προσεγγιστικότητας του ***CKKS***, το scheme αυτό έχει χαρακτηριστεί και ως μια λύση για την κρυπτογραφημένη μηχανική μάθηση (encrypted machine learning).

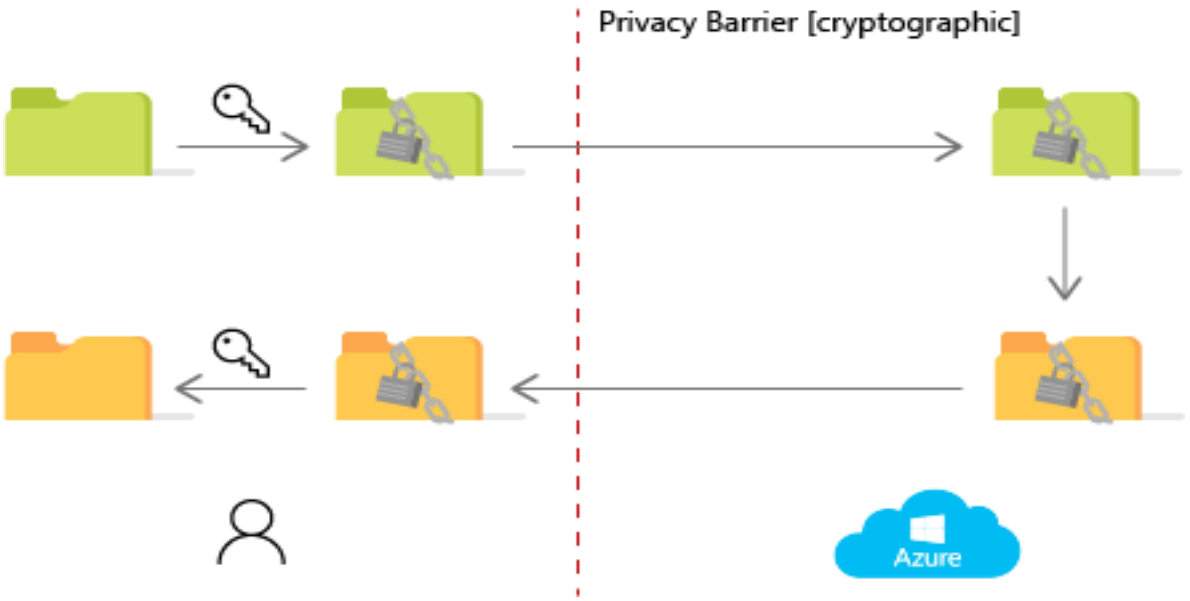
Κεφάλαιο 4 : Microsoft SEAL

4.1) Εισαγωγή στο Microsoft SEAL

Το Microsoft SEAL, πρωτοεμφανίστηκε το Δεκέμβριο του 2018. Από τότε έχουν προκύψει αρκετές παραλλαγές του, γραμμένες σε άλλες γλώσσες, όπως π.χ. python, αλλά η βιβλιοθήκη αυτή, είναι γραμμένη σε C++. Το Microsoft SEAL, έχει ως σκοπό, μεταξύ άλλων, να μπορέσει να εμπλέξει και developers, κυρίως για εφαρμογές cloud, κάνοντας αρκετά απλή την διαδικασία κρυπτογράφησης και αποκρυπτογράφησης των δεδομένων. Αυτό επιτυγχάνεται με τις κλάσεις, τα αντικείμενα και τις συναρτήσεις που προσφέρει, με τις οποίες μπορεί κάποιος μέσα σε λίγες γραμμές, να έχει ήδη έτοιμη την κρυπτογράφηση στα δεδομένα του. Αυτή η ευκολία όμως, σε συνδυασμό με την απόδοση των schemes και των συναρτήσεων που είναι σχεδιασμένες στο SEAL, είναι που την καθιστά ίσως την πιο διαδεδομένη βιβλιοθήκη που υλοποιεί Fully Homomorphic Encryption schemes. Αποτελεί μια βιβλιοθήκη ανοιχτού πηγαίου κώδικα. Παρακάτω επεξηγείται ένα παράδειγμα για cloud του ποια θα ήταν η αξία του Homomorphic Encryption.



*Εικόνα 4.1 Traditional Cloud. Εδώ φαίνεται το πώς το «παραδοσιακό» cloud θα είναι υπεύθυνο για την ασφάλεια των αρχείων ενός χρήστη. Φαίνεται επίσης πως η πρόσβαση στα αρχεία-δεδομένα κάποιου χρήστη γίνεται με κάποια συγκεκριμένη πολιτική πρόσβασης (**access policy**).*



Εικόνα 4.2 Cloud που χρησιμοποιεί Homomorphic Encryption. Εδώ μπορούμε να δούμε το πώς με το Microsoft SEAL και με την αξιοποίηση των Homomorphic Encryption schemes, το cloud δεν θα έχει ποτέ αποκρυπτογραφημένη πρόσβαση σε δεδομένα κάποιου χρήστη. Η πρόσβαση στα αρχεία/δεδομένα σε αυτήν την περίπτωση βασίζεται στην κρυπτογραφία και στα μαθηματικά προβλήματα τα οποία αναλύθηκαν παραπάνω.

Η βιβλιοθήκη Microsoft SEAL, παρέχει αρκετά header και .cpp files (πάνω από 100), τα οποία μπορούν να επιτελούν κάποιες βασικές διεργασίες, για τη χρήση των Homomorphic Encryption schemes. Για αυτά τα αρχεία, θα γίνει ανάλυση παρακάτω.

4.2) Τα στάδια χρήσης των schemes στο Microsoft SEAL

Τα Homomorphic Encryption schemes θα μπορούσαν να χωριστούν σε πέντε γενικά στάδια διαδικασιών, για να μπορούν να είναι επιτυχή και αποτελεσματικά. Κάθε scheme, φυσικά, διατηρεί τις ιδιαιτερότητας για τις οποίες έχει ήδη γίνει λόγος. Αυτά τα στάδια είναι τα εξής :

Setup -> KeyGeneration -> Encryption -> Evaluation -> Decryption

Γίνεται εμφανές, πως έχουμε αναφερθεί ήδη στα περισσότερα από αυτά, με εξαίρεση το ***Setup***. Στην ουσία, καταδεικνύεται η σειρά με την οποία θα πρέπει να εφαρμοστούν οι αλγόριθμοι αυτοί, για να μπορέσουμε να χρησιμοποιήσουμε αποτελεσματικά ένα Homomorphic Encryption scheme.

i) ***Setup*** : Σε αυτό το στάδιο, καθορίζονται κάποιες βασικές παράμετροι, ανάλογα με το scheme που θέλουμε να χρησιμοποιήσουμε (π.χ. ***BGV, CKKS***), ή με την αποτελεσματικότητα της ασφάλειας που θέλουμε να προσδώσουμε στο πρόγραμμά μας. Συνολικά δηλαδή αρχικοποιείται το ***context*** για συγκεκριμένες παραμέτρους.

ii) ***KeyGeneration*** : Εδώ αξιοποιούνται οι παράμετροι του προηγούμενου σταδίου, ούτως ώστε να παραχθούν και τα απαραίτητα κλειδιά για το επιλεγμένο scheme. Η βιβλιοθήκη Microsoft SEAL, δίνει τη δυνατότητα σε

αυτό το στάδιο, να παράξουμε διάφορα κλειδιά, πέρα από τα αναγκαία ***public keys*** και ***secret keys***, όπως είναι τα ***relin keys (Relinearization)***, τα ***galois keys (Rotation)*** και τα ***kswitch keys***. Δε θα επεκταθούμε ιδιαίτερα στα τελευταία, σημειώνουμε όμως ότι όσον αφορά την παραγωγή κλειδιών, η πιο κοστοβόρα και χρονικά και χωρικά είναι η παραγωγή των ***galois keys***.

iii) ***Encryption*** : Είναι υπεύθυνο για την κρυπτογράφηση των απλών κειμένων/μηνυμάτων (***Plaintext***). Επίσης σε αυτό το στάδιο αρχικοποιούνται τα αντικείμενα των απλών κειμένων και των κρυπτογραφημένων κειμένων (***Ciphertext***). Η βιβλιοθήκη αυτή, έχει διαφορετικές κλάσεις για διαφορετικά schemes, για παράδειγμα την ***CKKSEncoder***, η οποία χρησιμοποιείται για να κωδικοποιήσει κάποιο ***Plaintext*** κατάλληλα, για να κρυπτογραφηθεί στη συνέχεια από τον ***Encryptor***. Στις κλάσεις και σε κάποιες βασικές συναρτήσεις τους, καθώς και στην ίδια την δομή του SEAL θα αναφερθούμε αναλυτικά σε επόμενο υποκεφάλαιο.

iv) ***Evaluation*** : Σε αυτό το στάδιο, γίνονται κάποιοι αναγκαίοι υπολογισμοί πάνω στα κρυπτογραφημένα κείμενα/μηνύματα (***Ciphertext***). Για παράδειγμα, μια πράξη της πρόσθεσης ή του τετραγωνισμού ενός κρυπτογραφημένου κειμένου γίνεται σε αυτό το στάδιο.

v) ***Decryption*** : Είναι το στάδιο της αποκρυπτογράφησης. Μπορεί να χρησιμοποιηθεί σε κάποιο πρόγραμμα, είτε σε κάποιο request από τον χρήστη ή στη δικιά μας περίπτωση, για την ανάκτηση του αρχικού απλού κειμένου (***Plaintext***) αποκρυπτογραφώντας το κρυπτογραφημένο κείμενο (***Ciphertext***) και ελέγχοντας την ορθότητά του.

4.3) Δομή, κλάσεις και συναρτήσεις του Microsoft SEAL

Στη βιβλιοθήκη Microsoft SEAL, υπάρχουν πάνω από 100 .cpp και header files. Σαν κάποιες αρχικές παρατηρήσεις, να αναφέρεται, ότι υπάρχει μεγάλη αλληλεξάρτηση μεταξύ αυτών των αρχείων, για αυτό και στα περισσότερα header files βλέπουμε το preprocessor directive ***#pragma once*** στην αρχή του αρχείου, δηλαδή δίνεται στον compiler η πληροφορία του να μην κάνει include τα αρχεία αυτά πάνω από μία φορά, αλλά να γίνεται μόνο μία φορά σε κάθε compilation. Αυτή η αλληλεξάρτηση πάντως, όπως θα δούμε και στη συνέχεια, είναι και η μεγαλύτερη δυσκολία που έχει ο πηγαίος κώδικας της βιβλιοθήκης, όσον αφορά την προσπάθεια acceleration κομματιού του κώδικα με FPGA. Με αυτό εννοείται, πως ο κώδικας είναι γραμμένος με τέτοιον τρόπο, ούτως ώστε να δίνει τη δυνατότητα σε developers που δεν έχουν κάποια άμεση σχέση με την κρυπτογραφία, να μπορούν να υλοποιήσουν προγράμματα και εφαρμογές στα οποία να χρησιμοποιείται κάποιο Homomorphic Encryption scheme. Αυτό σημαίνει ότι η διαδικασία των πέντε σταδίων που περιγράφεται παραπάνω, μπορεί να γίνει σε λίγες σχετικά γραμμές. Αυτό όμως έρχεται σε αντιδιαστολή με την επιτάχυνση κάποιας σχεδίασης (source κώδικας στο Vivado HLS), καθώς αυτή απαιτεί την κατά το δυνατόν λιγότερη συσχέτιση με εξωτερικές βιβλιοθήκες, κομμάτια κώδικα και γενικά οτιδήποτε πέρα από τον ίδιον τον source κώδικα. Θα ήταν δηλαδή πολύ πιο

εύκολο, αν ένα ολόκληρο scheme και κάθε του στάδιο υπήρχε αυτούσιο σε ένα μόνο .cpp αρχείο. Αυτό όμως, δυστυχώς δεν υπάρχει. Αυτή είναι μια από τις βασικές δυσκολίες που αντιμετωπίστηκαν και η περιγραφή τους θα γίνει σε επόμενο κεφάλαιο.

Θα αναφερθούμε τώρα σε κάποια βασικά αρχεία, κλάσεις και συναρτήσεις οι οποίες θεωρούμε πως είναι σημαντικές και παίζουν ρόλο στην υλοποίηση ενός Homomorphic Encryption scheme και πιο συγκεκριμένα του CKKS. Μετά την εγκατάσταση του SEAL (επόμενο κεφάλαιο), όλα τα παρακάτω αρχεία βρίσκονται στον φάκελο ../SEAL/native/src/seal. Η σειρά των αρχείων/κλάσεων/συναρτήσεων θα γίνει βάσει των προαναφερθέντων σταδίων.

1) ***EncryptionParameters / encryptionparams.h*** : Αποτελεί μια σημαντική κλάση, η οποία θέτει τις αναγκαίες παραμέτρους στο στάδιο του ***Setup***, ανάλογα με το scheme που θέλουμε να χρησιμοποιήσουμε. Η δήλωση για τη χρήση του ***CKKS*** scheme γίνεται ως εξής :

```
EncryptionParameters parms(scheme_type::ckks);
```

Είναι προφανές, ότι το scheme επιλέγεται μέσω του scheme_type. Κάποιες συναρτήσεις που ανήκουν σε αυτήν τη κλάση είναι οι :

i) `inline void set_poly_modulus_degree(std::size_t poly_modulus_degree)`

Θέτει το βαθμό του πολυωνιμικού modulus. Ο αριθμός αυτός επηρεάζει τον αριθμό των συντελεστών (***coefficients***) που θα εμπεριέχονται σε πολυώνυμα τύπου απλών κειμένων (***plaintext***), το μέγεθος των στοιχείων των κρυπτογραφημένων κειμένων (***ciphertexts***). Ακόμη, όσο μεγαλύτερο είναι το poly_modulus_degree τόσο χειροτερεύει και η υπολογιστική επίδοση, αλλά αυξάνεται το επίπεδο ασφαλείας του scheme. Στο Microsoft SEAL, ο βαθμός αυτός πρέπει να είναι δύναμη του 2.

ii) `inline void set_coeff_modulus(const std::vector<Modulus> &coeff_modulus)`

Θέτει την παράμετρο coefficient modulus. Αποτελείται από μια λίστα διακριτών πρώτων αριθμών και στην βιβλιοθήκη αναπαρίσταται ως ένα διάνυσμα των αντικειμένων της κλάσης ***Modulus***, που θα δούμε παρακάτω. Ομοίως, αυτή η παράμετρος επηρεάζει το μέγεθος των στοιχείων των ***Ciphertexts***. Όσο μεγαλύτερη είναι αυτή η παράμετρος, τόσο περισσότερους υπολογισμούς μπορεί να εκτελέσει το επιλεγμένο scheme, αλλά τόσο μειωμένο θα είναι το επίπεδο ασφαλείας. Στη βιβλιοθήκη, κάθε ένας πρώτος αριθμός που αποτελεί το διάνυσμα αυτό, θα πρέπει να είναι τουλάχιστον 60 bit και θα πρέπει να συγκλίνει στην τιμή που θα είναι το αποτέλεσμα της πράξης : $1 \bmod 2 \cdot poly\_modulus\_degree$.

iii) `inline void set_plain_modulus(const Modulus &plain_modulus)`

Χρησιμοποιείται στο BFV scheme, οπότε δε θα μας απασχολήσει ιδιαίτερα. Θέτει την παράμετρο plaintext modulus. Η παράμετρος αυτή είναι ένας ακέραιος αριθμός και αναπαρίσταται από την κλάση ***Modulus***. Καθορίζει το μεγαλύτερο συντελεστή, που θα μπορεί να συμβολίζει κάποιο πολυώνυμο τύπου ***Plaintext***. Ισχύουν τα ίδια που ισχύουν και για το coefficient modulus,

με εξαίρεση την τιμή του plaintext modulus, που δεν υπάρχει αναγκαιότητα σύγκλισης σε κάποια συγκεκριμένη τιμή.

iv) `inline void set_random_generator(std::shared_ptr<UniformRandomGeneratorFactory> random_generator) noexcept`

Προαιρετικά, ο χρήστης μπορεί να θέσει εδώ κάποιον pointer, για να επιλέξει κάποια άλλη πηγή παραγωγής τυχαίων αριθμών, πέρα από την default_factory(), που χρησιμοποιεί η βιβλιοθήκη.

v) `void compute_parms_id();`

Η τελευταία συνάρτηση στην οποία θα γίνει αναφορά και ίσως η πιο σημαντική της κλάσης. Ορίζεται μέσα στο encryptionparameters.cpp, καλείται από όλες τις παραπάνω συναρτήσεις και υπολογίζει τις περισσότερες παραμέτρους του σταδίου ***Setup***, βάζοντας τες στη μνήμη, με κάποιον pointer που δείχνει σε ένα μοναδικό "id".

Υπάρχουν κι άλλες χρήσιμες συναρτήσεις σε αυτήν την κλάση, που μπορούν να επιστρέψουν κάποια συγκεκριμένη παράμετρο ή να αποθηκέψουν/φορτώσουν κάποια από αυτές. Σημειώνεται ξανά, ότι διάφορες συναρτήσεις στη βιβλιοθήκη αλληλεξαρτώνται και καλούν η μία την άλλη. Για παράδειγμα η compute_parms_id() καλεί, μεταξύ άλλων, τη συνάρτηση allocate_uint(), η οποία βρίσκεται σε άλλο header file, που με τη σειρά της καλεί άλλες συναρτήσεις που βρίσκονται και αυτές σε άλλα ή στο ίδιο header file, για να υλοποιήσουν αυτό που λέει και το όνομά τους. Δε θα επεκταθούμε τόσο στην αλληλεξάρτηση αυτή, αλλά την αναφέρουμε, γιατί αποτελεί σημαντικό πρόβλημα, όπως θα δούμε και στο 5ο και 6ο κεφάλαιο, της ίδιας της υλοποίησης της σχεδίασης.

2) Modulus / modulus.h : Στην κλάση αυτή θα αναφερθούμε μόνο σε μια σημαντική συνάρτηση, καθώς η κλήση της δεν είναι απαραίτητη για την υλοποίηση κάποιου scheme, αλλά γίνεται αυτόματα με τη δημιουργία αντικειμένων άλλων κλάσεων. Η συνάρτηση αυτή είναι η :

```
SEAL_NODISCARD static constexpr int MaxBitCount(
    std::size_t poly_modulus_degree,
    sec_level_type sec_level = sec_level_type::tc128) noexcept
```

Η MaxBitCount(), επιστρέφει έναν συγκεκριμένο αριθμό που αφορά το μεγαλύτερο μήκος σε bit, που μπορεί να έχει το coefficient modulus. Η κλήση της, ανάλογα με τα ορίσματα επιστρέφει τον κατάλληλο αριθμό, ο οποίος εξασφαλίζει ένα συγκεκριμένο επίπεδο ασφαλείας, για κάποιο δεδομένο βαθμό poly_modulus_degree. Όπως έχει ήδη προαναφερθεί, δεν υπάρχει κάποιο συγκεκριμένο security standard όσον αφορά το Homomorphic Encryption. Η βιβλιοθήκη βασίζεται στο homomorphicencryption.org για να επιλέξει το κατάλληλο επίπεδο ασφαλείας. Σημειώνεται, ότι η βιβλιοθήκη υποστηρίζει και το standard για Quantum Security. Ακολουθεί ο πίνακας με τους αριθμούς για συγκεκριμένα poly_modulus_degree :

Classical Security			
poly_mod_deg	128-bit	192-bit	256-bit
1024	27	19	41
2048	54	37	29
4096	109	75	58
8192	218	152	118
16384	438	305	237
32768	881	611	476

Πίνακας 4.1 Πίνακας αριθμών bits που καθορίζουν το coefficient modulus, για συγκεκριμένα poly_modulus_degree, για κλασσική ασφάλεια.

Quantum Security			
poly_mod_deg	128-bit	192-bit	256-bit
1024	25	17	13
2048	51	35	27
4096	101	70	54
8192	202	141	109
16384	411	284	220
32768	827	571	443

Πίνακας 4.2 Πίνακας αριθμών bits που καθορίζουν το coefficient modulus, για συγκεκριμένα poly_modulus_degree, για κβαντική ασφάλεια.

Σημαντικό είναι να παρατηρηθεί ότι οι αριθμοί στο Quantum Security είναι μικρότεροι από αυτούς για το Classical Security. Σημειώνεται επίσης, ότι οι αριθμοί αυτοί αποτελούν αριθμούς κατανομής που προκύπτει από το πρόβλημα Learning With Errors (LWE) και μάλιστα για τριαδικό (ternary) LWE πρόβλημα.

Δε θα επεκταθούμε περισσότερο στα πλαίσια της διπλωματικής εργασίας στο συγκεκριμένο ζήτημα. Αναφέρουμε όμως τη δημοσίευση “Quantum Key Search for Ternary LWE” και “Revisiting the Hybrid attack on sparse and ternary secret LWE” στις οποίες γίνεται αναφορά για το παραπάνω ζήτημα.

3) ***SEALContext / context.h*** : Η κλάση αυτή είναι σημαντική, καθώς μόνο και μόνο η δημιουργία ενός αντικειμένου αυτής της κλάσης καλεί τη συνάρτηση αποτελεί τη σχεδίασή μας. Μια τυπική κλήση της γίνεται ως εξής :

```
SEALContext context(parms);
```

Σημαντική παρατήρηση είναι, ότι η κατασκευή αντικειμένου γίνεται και με το πέρασμα του ορίσματος parms, δηλαδή των παραμέτρων που αρχικοποιήθηκαν προηγουμένως με την κλάση EncryptionParameters, για το κατάλληλο scheme που δύναται να υλοποιηθεί. Στο header file έχουμε πολλές συναρτήσεις που επιστρέφουν διάφορες παραμέτρους ασφαλείας. Στην ουσία εδώ ολοκληρώνεται το στάδιο αρχικοποίησης – ***Setup***, καθώς η κλάση αυτή δημιουργεί το context για τις συγκεκριμένες παραμέτρους. Η πιο σημαντική συνάρτηση ορίζεται στο ομότιτλο αρχείο .cpp και είναι η :

```
SEALContext::ContextData SEALContext::validate(EncryptionParameters parms)
```

Η validate(); επιβεβαιώνει τις ήδη δημιουργημένες παραμέτρους και το context. Μέσω της validate, όπως θα δούμε και στη συνέχεια, καλείται μεταξύ άλλων και η συνάρτηση div_uint128_uint64_inplace_generic() που κάναμε accelerate. Σημειώνεται επίσης, ότι η κλάση αυτή περιέχει και συνάρτηση για να επιστρέψει σφάλμα στην περίπτωση που κάποια παράμετρος είναι λάθος.

Για να κλείσει το στάδιο του ***Setup***, θα αναφερθούμε στις κλάσεις ***Plaintext*** και ***Ciphertext*** και ***CKKSEncoder***.

4) ***Plaintext / plaintext.h*** : Όπως λέει και το όνομα, σκοπός αυτής της κλάσης είναι η δημιουργία, αλλά και ο συνολικότερος χειρισμός, αντικειμένων απλών κειμένων. Όταν χρησιμοποιείται το ***CKKS*** scheme, το οποίο είναι το scheme που χρησιμοποιείται και για την υλοποίηση που έχουμε κάνει, το απλό κείμενο αποθηκεύεται από προεπιλογή, σε μορφή μετασχηματισμού NTT (Number-Theoretic Transform), σε σχέση με τους πρώτους αριθμούς που εμπεριέχει το coefficient modulus. Με αυτόν τον τρόπο, η μνήμη που χρειάζεται να κατανεμηθεί είναι ακριβώς όσο είναι και το μέγεθος του coefficient modulus πολλαπλασιασμένο με τον βαθμό του polynomial modulus. Ένα έγκυρο απλό κείμενο για το scheme του ***CKKS***, αποθηκεύει επίσης και το parms_id που είδαμε προηγουμένως. Στη μορφή NTT έγινε αναφορά και στο 2ο Κεφάλαιο. Αποτελεί πράξη που μπορεί να γίνει με την κλάση ***Evaluator***, αλλά και είναι ένας χρήσιμος μετασχηματισμός που τον βλέπουμε συχνά και στη βιβλιοθήκη Microsoft SEAL αλλά και συνολικότερα στην κρυπτογραφία. Ο τρόπος χρήσης της κλάσης ***Plaintext*** στην πράξη, είναι ιδιαίτερα απλός και φαίνεται παρακάτω.

```
Plaintext x_plain;
```

Με αυτόν τον τρόπο, το x_plain αποτελεί πλέον ένα απλό κείμενο και μπορεί να αξιοποιηθεί, όπως θα δούμε στη συνέχεια και από την κλάση ***Evaluator*** για την υλοποίηση διαφόρων πράξεων με αυτό. Η κλάση ***Plaintext***, περιέχει περίπου 20 συναρτήσεις, που αφορούν την κράτηση του απαραίτητου χώρου

στη μνήμη, μέχρι και μικρο-διαχείριση του απλού κειμένου, π.χ. τη μείωση του μεγέθους για να μπορεί να χωράει στη μνήμη.

5) ***Ciphertext / ciphertext.h*** : Η κλάση του κρυπτογραφημένου κειμένου, έχει σχεδόν την ίδια λειτουργικότητα με αυτήν του απλού κειμένου. Και εδώ έχουμε σχεδόν 20 συναρτήσεις που είναι σχεδόν ταυτόσημες με την προηγούμενη κλάση. Ο τρόπος χρήσης της κλάσης ***Ciphertext*** είναι ο εξής :

```
Ciphertext x_encrypted;
```

6) ***KeyGenerator / keygenerator.h*** : Με αυτήν την κλάση περνάμε στο δεύτερο στάδιο για τη χρήση ενός Homomorphic Encryption scheme, αυτό της παραγωγής κλειδιών. Αποτελεί μια πολύ σημαντική κλάση, καθώς με αυτήν δημιουργούνται όλα τα αναγκαία κλειδιά είτε δημόσια είτε κρυφά. Σημειώνουμε εδώ, ότι υπάρχουν και κλάσεις και header files, τα οποία χρησιμοποιούνται και για κάθε κλειδί ξεχωριστά. Αυτό δεν είναι όμως αναγκαίο για όλα τα κλειδιά. Ως εξαίρεση, για τη δημιουργία ενός κρυφού κλειδιού (***secret key***), δεν είναι απαραίτητη η δημιουργία και ενός αντικειμένου κρυφού κλειδιού. Η δημιουργία ενός αντικειμένου αυτής της κλάσης, αλλά και των κρυφών κλειδιών γίνεται με :

```
KeyGenerator keygen(context);
auto secret_key = keygen.secret_key();
```

Η συνάρτηση παραγωγής κρυφού κλειδιού, καλεί με τη σειρά της άλλες συναρτήσεις της κλάσης, οι οποίες όμως οδηγούν και στο header file secretkey.h και σε άλλα header files. Η λογική που ακολουθείται και για την παραγωγή των υπόλοιπων κλειδιών είναι παρόμοια, με τη διαφορά ότι δημιουργείται και ένα αντικείμενο της αντίστοιχης κλάσης για το εκάστοτε κλειδί, δηλαδή έχουμε :

```
PublicKey public_key;
keygen.create_public_key(public_key);
```

```
RelinKeys relin_keys;
keygen.create_relin_keys(relin_keys);
```

```
GaloisKeys gal_keys;
keygen.create_galois_keys(gal_keys);
```

Η κλήση των παραπάνω συναρτήσεων είναι αρκετή για την ολοκληρωμένη παραγωγή κλειδιών στο δεύτερο στάδιο υλοποίησης ενός Homomorphic Encryption scheme. Φυσικά, η κάθε συνάρτηση μπορεί να έχει πολλές κλήσεις σε άλλες συναρτήσεις άλλων ή του ίδιου header file. Για παράδειγμα, η κλάση ***KeyGenerator***, εμπεριέχει και τη συνάρτηση create_public_key(), αλλά να καλεί τη συνάρτηση generate_pk(), πάλι της ίδιας κλάσης, η οποία με τη σειρά της καλεί άλλες αναγκαίες συναρτήσεις για να εκτελεστεί η παραγωγή. Σημειώνουμε, ότι το αντικείμενο keygen παίρνει ως όρισμα το context των παραμέτρων ασφαλείας, στο οποίο αναφερθήκαμε προηγουμένως. Ακόμη, η παραγωγή κρυφού κλειδιού δεν παίρνει κάποιο όρισμα, ενώ η παραγωγή των

υπολοίπων κλειδιών παίρνουν τα αντικείμενα των αντίστοιχων κλάσεων. Παρατηρούμε επίσης ότι τα αντικείμενα των κλειδιών, δεν έχουν κάποιο όρισμα. Αυτό δε σημαίνει ότι η παραγωγή κλειδιών σε αυτήν την περίπτωση είναι τελείως αυθαίρετη. Πρέπει να γνωρίζουμε για ποιο scheme θα παραχθεί το κλειδί, ακόμα και δημόσιο να είναι, όπως και τι παράμετροι έχουν γίνει ***Setup***. Όπως προαναφέρθηκε, η βιβλιοθήκη Microsoft SEAL έχει μεγάλη αλληλεξάρτηση όσον αφορά τα header και τα .cpp files. Εδώ παρατηρούμε, για παράδειγμα, πως το αντικείμενο ***KeyGenerator*** έχει πάρει τους αναγκαίους pointers που δείχνουν στα αντίστοιχα σημεία στη μνήμη το περιεχόμενο του context και κατ' επέκτασιν των parms. Πέρα από αυτό όμως, υπάρχει άλλη μια κλάση, η ***MemoryPoolHandle***, για την οποία δε θα επεκταθούμε, πέρα από το ότι η χρήση της μπορεί να υποδείξει το που βρίσκεται τι στη μνήμη. Είναι σημαντικό να σημειώσουμε όμως, πως σχεδόν κάθε συνάρτηση κάποιας κλάσης, που πρέπει να πάρει δεδομένα από την μνήμη, έχει αναφορά στην κλάση αυτή. Με τις κλάσεις των κλειδιών και τις συναρτήσεις τους, ολοκληρώνεται και το δεύτερο στάδιο ενός Homomorphic Encryption scheme και έτσι περνάμε και στο τρίτο στάδιο, αυτό του ***Encryption***.

7) ***Encryptor / encryptor.h*** : Η κλάση αυτή έχει πολύ σημαντικές συναρτήσεις, που αφορούν στην κρυπτογράφηση όλων των μεταβλητών-κειμένων των προηγούμενων σταδίων. Ο τρόπος δημιουργίας ενός αντικειμένου αυτής της κλάσης, είναι ο εξής:

```
Encryptor encryptor(context, public_key);
```

Η συνάρτηση που ξεχωρίζει είναι η encrypt(), η οποία καταλήγει και σε διάφορες παραλλαγές συναρτήσεων, ανάλογα με τη συμμετρία της κρυπτογράφησης, που θέλουμε να επιτευχθεί. Σημαντική συνάρτηση και η encrypt_zero_(a)symmetric(), στην οποία φτάνει η συνάρτηση αυτή, εκεί λαμβάνει χώρα η κρυπτογράφηση και βρίσκεται στο αρχείο rlwe.cpp. Σε επίπεδο ενός απλού προγράμματος, ο τρόπος κρυπτογράφησης ενός απλού κειμένου σε κρυπτογραφημένο, γίνεται ως εξής :

```
encryptor.encrypt(x_plain, x_encrypted);
```

Προφανώς και τα x_plain και x_encrypted, αποτελούν Plaintext και Ciphertext αντίστοιχα.

8) ***Evaluator / evaluator.h*** : Εδώ εμπεριέχονται όλες η συναρτήσεις που είναι υπεύθυνες για τις πράξεις και τον χειρισμό των κρυπτογραφημένων κειμένων. Αυτές οι συναρτήσεις, καλούν άλλες, που βρίσκονται πιο βαθιά μέσα στη βιβλιοθήκη, για να εκτελεστούν οι πράξεις αυτές. Οι πράξεις που μπορούν να γίνουν με την κλάση ***Evaluator***, εμπεριέχουν αριθμητικές πράξεις, relinearization, rotation, αλλά και άλλες πράξεις όπως είναι η μετατροπή των ***Ciphertexts*** σε μορφή NTT και πίσω. Έχει και αυτή περίπου 20 συναρτήσεις. Η κατασκευή ενός αντικειμένου της κλάσης αυτής μπορεί να γίνει ως εξής :

```
Evaluator evaluator(context);
```

Το αντικείμενο παίρνει ως όρισμα το context των παραμέτρων, για να μπορέσει να έχει άμεση πρόσβαση στο context και στις παραμέτρους, του scheme που θέλουμε να χρησιμοποιήσουμε. Οι συναρτήσεις είναι πάλι της μορφής ***add(c₁, c₂), sub(), multiply(), square, relinearize()*** και ούτω καθεξής. Υπάρχουν και παραλλαγές για τη χρήση σε απλό κείμενο (***plaintext***), όπως για παράδειγμα η ***add_plain()***¸αλλά και για να γίνει κάποια πράξη μεταξύ αρκετών κειμένων, μπορεί να χρησιμοποιηθεί η παραλλαγή ***add_many()***. Ενδεικτικά αναφέρουμε, ότι για να γίνει για παράδειγμα η πράξη του πολλαπλασιασμού, καλούνται στη σειρά τουλάχιστον άλλες δέκα συναρτήσεις που βρίσκονται σε άλλα αρχεία. Το όρισμα context, περνάει στην κλάση αυτή το scheme που θέλουμε να χρησιμοποιήσουμε και έτσι μπορεί να επιλεχθεί η κατάλληλη συνάρτηση-παραλλαγή της βιβλιοθήκης, για το κάθε scheme. Εδώ θα σταθούμε σε δύο συναρτήσεις που είναι ιδιαίτερες και θα δείξουμε τον τρόπο χρήσης τους.

Η πρώτη συνάρτηση, είναι αυτή που υλοποιεί το ***Modulus Switching***.

```
evaluator.mod_switch_to_inplace(x_encrypted, last_parms_id);
```

Όπως βλέπουμε, η χρήση της είναι αρκετά απλή. Παίρνει ως ορίσματα κάποιο κείμενο, είτε κρυπτογραφημένο ή απλό, αλλά και το "id" των παραμέτρων στη μνήμη του τελευταίου κειμένου που πειράξαμε, συνήθως μετά το scaling που μπορεί να υποστεί το κείμενο αυτό. Αυτό συμβαίνει, καθώς κατά το scaling κάποιου κειμένου μπορεί να ήταν μεγαλύτερου βαθμού, λόγω κάποιας άλλης πράξης και πρέπει να εκτελέσουμε κάποια άλλη πράξη μεταξύ δύο κειμένων, όπως για παράδειγμα την πράξη της πρόσθεσης. Αφού λοιπόν αντιληφθούμε ότι έχουμε διαφορετικό scale, θέτουμε το ίδιο και στα δύο κείμενα και μετά χρησιμοποιούμε την πράξη του ***Modulus Switching***, ούτως ώστε να μπορέσουμε να έχουμε σωστά σε κάθε κείμενο τις παραμέτρους κρυπτογράφησης, αλλά και να «πετάξουμε» κομμάτια του coefficient modulus, που δε θα χρειαστούμε. Σε πρακτικό επίπεδο, αυτό επιτυγχάνεται χρησιμοποιώντας τη συνάρτηση αυτή, σε αυτήν την περίπτωση. Το ***CKKS***, υποστηρίζει κανονικά την πράξη αυτή, όπως και το ***BFV***.

Η δεύτερη συνάρτηση στην οποία θα αναφερθούμε είναι αυτή που υλοποιεί το rescale. Εδώ θα γίνει και πιο κατανοητή η χρήση της παραπάνω συνάρτησης. Όταν εκτελούμε κάποια πράξη σε κάποιο ***ciphertext***, για παράδειγμα όταν θέλουμε να αξιολογήσουμε ένα πολυώνυμο 3ου βαθμού. Αυτό σημαίνει ότι για να κατασκευάσουμε το ***ciphertext*** x^3, θα πρέπει πρώτα να το τετραγωνίσουμε, με τη συνάρτηση square(). Αν το ***x*** είχε scale 2^{40}, πλέον θα έχει 2^{80} και οπότε δε θα μπορούμε να εκτελέσουμε κάποια άλλη πράξη με του άλλους όρους του πολυωνύμου. Για τον λόγο αυτόν, πρέπει να κατεβάσουμε το scale ξανά σε 2^{40} (1ου βαθμού όρος ή απλή παράμετρος στο συγκεκριμένο παράδειγμα). Αυτό θα γίνει με συνάρτηση rescale_to_next_inplace() και έτσι μπορεί να συνεχιστεί η κατασκευή του πολυωνύμου που αξιολογείται. Αυτό γίνεται με την κλήση :

```
evaluator.rescale_to_next_inplace(x_encrypted);
```

9) ***Decryptor / decryptor.h*** : Σκοπός αυτής της κλάσης είναι η αποκρυπτογράφηση του κρυπτογραφημένου κειμένου (***ciphertext***) και η αποθήκευση του αποτελέσματος σε κάποιον προορισμό. Η δήλωση του αντικειμένου γίνεται με την :

```
Decryptor decryptor(context, secret_key);
```

Παρατηρούμε ότι το αντικείμενο παίρνει ως ορίσματα το context, αλλά και το κρυφό κλειδί, το οποίο είναι απαραίτητο για την αποκρυπτογράφηση των δεδομένων. Το στάδιο αυτό, χρησιμοποιείται και στην υλοποίησή μας, για να μπορέσει να επαληθευτεί η σωστή λειτουργία του προγράμματος. Βασική και πιο σημαντική συνάρτηση της κλάσης αυτής αποτελεί η ***decrypt()***, η οποία με τη σειρά της καλεί την κατάλληλη παραλλαγή, ανάλογα με το ποιο scheme θέλουμε να χρησιμοποιήσουμε (εδώ : ***ckks_decrypt();***), που με τη σειρά της ξεκινάει μια αλυσίδα κλήσεων συναρτήσεων, για να μπορέσει να γίνει η αποκρυπτογράφηση των δεδομένων. Η κλήση της συνάρτησης αυτής γίνεται με την :

```
decryptor.decrypt(encrypted_result, plain_result);
```

Η συνάρτηση αυτή, παίρνει ως όρισμα ένα κρυπτογραφημένο κείμενο ***encrypted_result*** (***Ciphertext***), το οποίο αποτελεί το αποτέλεσμα που προκύπτει, από την κατασκευή του πολυωνύμου και την εφαρμογή των πράξεων που υπάρχουν μεταξύ των όρων του, που συμβαίνει στο στάδιο της ***Evaluation*** και μετά την κρυπτογράφηση που υπέστη το αρχικό απλό κείμενο, που συμβαίνει στο στάδιο ***Encryption***. Το ***plain_result***, αποτελεί ένα άδειο αντικείμενο απλού κειμένου (***Plaintext***), στο οποίο θα γραφτεί η αποκρυπτογράφηση του πολυωνύμου που αξιολογήθηκε. Σε αυτό το στάδιο, το αποκρυπτογραφημένο κείμενο δεν είναι αναγνώσιμο από τον χρήστη. Αυτό συμβαίνει, διότι έχει προηγηθεί κωδικοποίηση των συντελεστών και του περιεχομένου του απλού κειμένου, ανάλογα και με το scale που έχει επιλεγεί από τον χρήστη. Παρακάτω βλέπουμε το πώς γίνεται αυτό, στην τελευταία κλάση για την οποία θα γίνει λόγος στα πλαίσια αυτής της διπλωματικής εργασίας.

10) ***CKKSEncoder / ckks.h*** : Γίνεται αναφορά σε αυτήν την κλάση, διότι αποτελεί σημαντικό κομμάτι του ***Pre-Encryption*** και του ***Post-Decryption*** για τη χρήση του ***CKKS*** scheme. Με αυτήν την κλάση, μπορούμε να δημιουργήσουμε ένα αντικείμενο ***encode***, το οποίο μπορεί να χρησιμοποιηθεί κυρίαρχα για την κωδικοποίηση, ώστε να επέλθει η κρυπτογράφηση και την αποκωδικοποίηση, αφού έχει προηγηθεί η αποκρυπτογράφηση του περιεχομένου. Θα λέγαμε λοιπόν, ότι μπορεί να θεωρηθεί σαν μια διαδικασία, η οποία τοποθετείται ανάμεσα στο ***Setup*** και στο ***Encryption***, αλλά και μετά το ***Decryption***. Η αρχική δήλωση ενός αντικειμένου, μπορεί να γίνει ως εξής :

```
CKKSEncoder encoder(context);
```

Οι βασικές συναρτήσεις που την αποτελούν είναι προφανώς η ***encode();*** και η ***decode();***. Η ***encode()***, χρησιμοποιείται για την κωδικοποίηση των συντελεστών (***coefficients***) του πολυωνύμου, σε ένα απλό κείμενο. Για παράδειγμα, η κωδικοποίηση ενός συντελεστή μπορεί να γίνει ως εξής :

```
encoder.encode(3.14159265, scale, plain_coeff3);
```

Εδώ βλέπουμε πώς το π, μπορεί να κωδικοποιηθεί σε ένα άδειο ***Plaintext plain_coeff3***, δίνοντας στη συνάρτηση και το προκαθορισμένο scale, που εδώ είναι 2^{40}. Ένας όρος του πολυωνύμου με βαθμό ≥ 1, κωδικοποιείται ως εξής :

```
encoder.encode(input, scale, x_plain);
```

Με τον τρόπο αυτό, κωδικοποιείται ένα διάνυσμα input, με δεκαδικές τιμές (***CKKS***), δίνοντας το scale και το άδειο ***Plaintext x_plain***. Ακριβώς μετά το βήμα αυτό της κωδικοποίησης, το ***x_plain*** είναι έτοιμο να κρυπτογραφηθεί.

Αντίστοιχα η συνάρτηση ***decode();,*** καλείται ως εξής :

```
encoder.decode(plain_result, result);
```

Εδώ, το ***plain_result*** είναι το αποκρυπτογραφημένο κείμενο-διάνυσμα του ***Ciphertext encrypted_result***, ενώ το ***result***, είναι ένα άδειο διάνυσμα στο οποίο και περνάμε το αποτέλεσμα σε αναγνώσιμη μορφή.

Με τις παραπάνω αναφορές, εξηγήθηκε και ένα μεγάλο κομμάτι του ***testbench***, το οποίο είναι βασισμένο στο παράδειγμα 4_ckks_basics.cpp, που εμπεριέχεται στη βιβλιοθήκη Microsoft SEAL. Στο επόμενο κεφάλαιο, θα αναλυθεί συνολικά η υλοποίηση και η επίδοσή της, αλλά θα γίνει και μια εισαγωγή σε βασικούς όρους των FPGA και της χρήσης της κάρτας Alveo™ U200 Data Center accelerator card και του περιβάλλοντος της Xilinx®, Vivado HLS 2019.2 .

Κεφάλαιο 5 : Εισαγωγή στα FPGA και υλοποίηση σχεδίασης

5.1) Εισαγωγή στα FPGA και στην έννοια του HLS

Τα FPGA (Field-Programmable Gate Array), είναι ολοκληρωμένα κυκλώματα (Integrated Circuit, τα οποία εμπεριέχουν blocks λογικής (Configurable Logic Blocks – CLBs) αλλά και διασυνδέσεις μεταξύ αυτών των blocks. Ένα λογικό block, μπορεί να είναι υλοποιημένο με την αρχιτεκτονική πολυπλέκτη (Multiplexor - MUX) ή με την αρχιτεκτονική του πίνακα αναζήτησης (LookUp Table - LUT). Στην αρχιτεκτονική πολυπλέκτη, η είσοδος στο λογικό block είναι 0 ή 1 και ανάλογα με τον προγραμματισμό του κυκλώματος, μπορεί να υλοποιηθεί κάποια πράξη με την χρήση των πολυπλεκτών που βρίσκονται μέσα στα λογικά blocks. Στην αρχιτεκτονική του πίνακα αναζήτησης, η είσοδος και το σύνολο των στοιχείων της, καθορίζουν την πράξη που θα πρέπει να επιτελεστεί, αξιοποιώντας τους πίνακες αληθείας που εμπεριέχονται μέσα στα λογικά block και παράγοντας την επιθυμητή έξοδο. Πέρα από τα παραπάνω, για την καλύτερη απόδοση και συνολική λειτουργία των κυκλωμάτων αυτών, τα FPGA μπορεί να περιέχουν και κάποια ενσωματωμένα στοιχεία. Τέτοια είναι οι Block RAMs (BRAM_18K), διάφορα block για την επιτέλεση κάποιων πράξεων, όπως για παράδειγμα αθροιστές (DSP48E), flip flops (FFs) για καταχωρητές, αλλά και άλλους ενσωματωμένους (υπο-) πυρήνες.

Κάποια FPGA, μπορούν να προγραμματιστούν μόνο μια φορά, ενώ άλλα μπορούν να προγραμματιστούν πολλές φορές και μετά την κατασκευή τους, σε αντίθεση με τα ASIC που είναι One Time Programmable (OTP). Τα FPGA προγραμματίζονται με γλώσσες περιγραφής υλικού (Hardware Description Language-HDL), όπως είναι η Verilog και η VHDL. Στα πλαίσια αυτής της διπλωματικής, ασχολούμαστε με τη διαδικασία του High Level Synthesis (HLS) ή αλλιώς C synthesis και έπειτα, της διαδικασίας του C/RTL Cosimulation.

Το High Level Synthesis, αποτελεί μια αυτοματοποιημένη διαδικασία σχεδιασμού, με την οποία προκύπτει μία δομή Register-Transfer Level (RTL) σε γλώσσα HDL, αξιοποιώντας μια αφηρημένη προδιαγραφή συμπεριφοράς ενός ψηφιακού κυκλώματος. Οι δοκιμές και τα αποτελέσματα που προέκυψαν με High Level Synthesis, έγιναν με τη χρήση του εργαλείου Vivado HLS 2019.2 και Vitis HLS 2021.1(&2021.2&2020.2), της Xilinx®. Με το εργαλείο αυτό, δίνεται η δυνατότητα να καθοριστεί η προαναφερόμενη προδιαγραφή με τη χρήση κάποιας πιο διαδεδομένης γλώσσας προγραμματισμού απ' ότι οι γλώσσες HDL και εν προκειμένω, με C/C++. Η δομή RTL που προκύπτει, είναι σε κάποια γλώσσα HDL της επιλογής μας, σε Verilog ή VHDL. Από τη στιγμή που η βιβλιοθήκη Microsoft SEAL είναι γραμμένη σε C++, η υλοποίηση της σχεδίασης, όπως και το testbench, για τα οποία θα γίνει λόγος στη συνέχεια, είναι επίσης γραμμένα με αυτήν τη γλώσσα. Με το HLS, μειώνεται ο χρόνος ανάπτυξης του κώδικα, βελτιώνεται η ποιότητα της σχεδίασης, αλλά και η συνολική διαχείριση με αλλαγές/διορθώσεις της αρχικής σχεδίασης γίνεται με μεγαλύτερη ευκολία.

Για την επιβεβαίωση της ορθότητας της προδιαγραφής, αλλά και της αυτοματοποιημένης αυτής διαδικασίας του HLS, το εργαλείο δίνει τη δυνατότητα του C/RTL Cosimulation, δηλαδή της προσομοίωσης της σχεδίασης (της προδιαγραφής και της δομής RTL που προκύπτει), ούτως ώστε να επαληθευτεί η λειτουργία του κυκλώματος και με τους δύο τρόπους. Η διαδικασία του C/RTL Cosimulation, δίνει τη

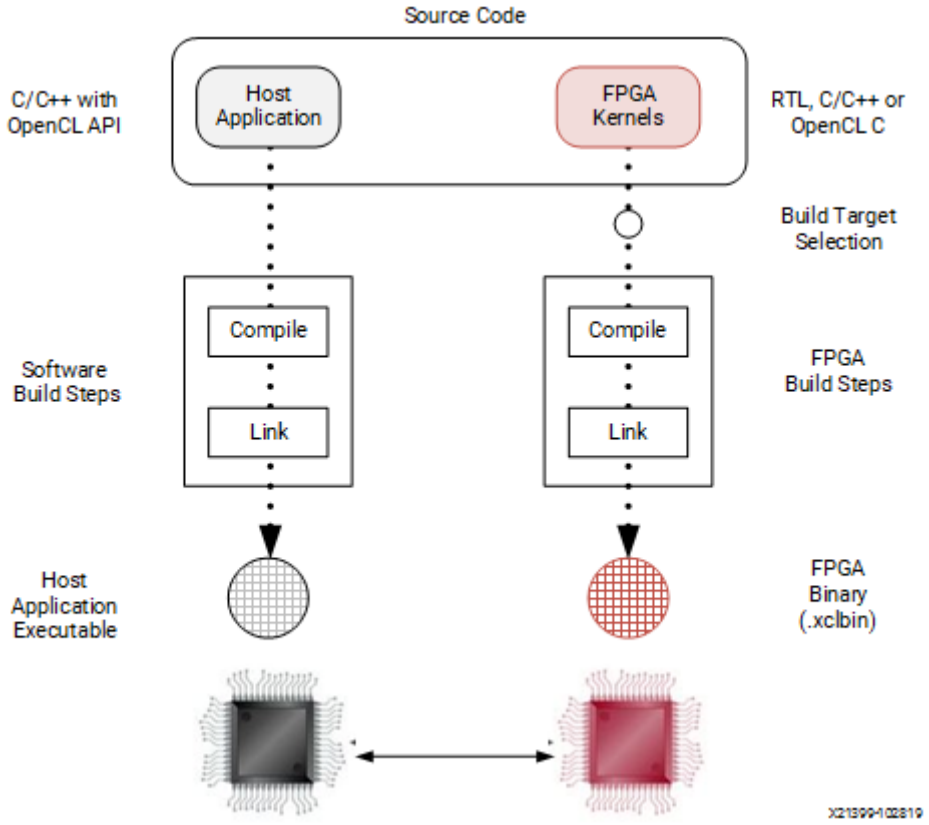
δυνατότητα επίσης του Wave Viewer Debug, δηλαδή το πρόγραμμα, μπορεί να δώσει τα διαγράμματα των σημάτων του κυκλώματος και τις τιμές τους, ανά κύκλο εκτέλεσης. Το Wave Viewer, είναι σημαντικό για την επαλήθευση της σχεδίασης, σε σχέση με την εκτέλεσή της και τη λειτουργικότητά της. Φυσικά, όταν έχουμε πολλά στοιχεία να επαληθεύσουμε ότι έχουν σωστές τιμές, δε γίνεται να ελέγξουμε ένα ένα τα στοιχεία μέσω του Wave Viewer, αλλά με κάποιον έλεγχο στο testbench, όπως και γίνεται. Η ουσία του Wave Viewer, είναι η αναπαράσταση που δίνει στο που περνάνε στο FPGA, τα ορίσματα που του δίνουμε στην top function, δηλαδή στη συνάρτηση που περιγράφει τη σχεδίασή μας, αλλά και οι πράξεις που γίνονται μέσα σε αυτήν.

Πέρα από τα παραπάνω, το εργαλείο δίνει τη δυνατότητα απλού C Simulation, δηλαδή compile & run, για την επαλήθευση του κώδικα, με την επιλογή συγκεκριμένων compiler, που διαθέτει το εργαλείο και δίνοντας τα απαραίτητα compiler flags. Σε όλα αυτά τα στάδια, αντιμετωπίστηκαν διάφορα προβλήματα, η αντιμετώπιση των οποίων, αναφέρεται στο επόμενο κεφάλαιο.

Για να γίνουν όλα τα παραπάνω, χρειάζονται τουλάχιστον δύο αρχεία κώδικα C++, το testbench και το source. Το testbench, είναι ένα αρχείο το οποίο τρέχει στον Host Processor (υπολογιστή) και είναι υπεύθυνο για την λειτουργικότητα της σχεδίασης του source κώδικα-επιταχυντή (accelerator). Η περιγραφή της λειτουργικότητας της προδιαγραφής του επιταχυντή γίνεται στο source αρχείο.

Με την διαδικασία High Level Synthesis, μπορούμε να αντλήσουμε συμπεράσματα για τη σχεδίασή μας, παίρνοντας τις εκτιμήσεις που μας βγάζει το εργαλείο, οι οποίες έχουν σχέση με τους χρόνους του ρολογιού, τον αριθμό των κύκλων που απαιτούνται για την εκτέλεση της σχεδίασης, το latency, καθώς και το ποσοστό ή τον απόλυτο αριθμό χρησιμοποίησης (utilization), των διάφορων στοιχείων του κυκλώματος, όπως είναι τα BRAM_18K, DSP48E, Flip-Flops, LUTs και URAM. Η διαθεσιμότητα αυτών των στοιχείων, καθορίζεται από το FPGA που επιλέγουμε για την προσομοίωση στο εργαλείο, το οποίο στα πλαίσια της παρούσας διπλωματικής είναι το Xilinx® Alveo™ U200 Data Center accelerator card.

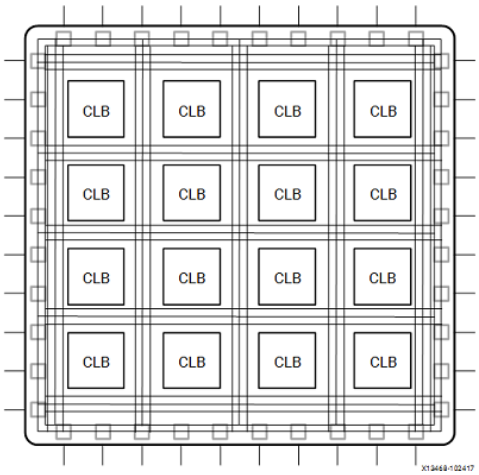
Στο πέρας της παραπάνω διαδικασίας, μπορεί να γίνει ένα βήμα παρακάτω, υλοποιώντας τη σχεδίαση μαζί με το πραγματικό FPGA και όχι σε simulation, προσθέτοντας στον κώδικα OpenCL. Στα πλαίσια αυτής της διπλωματικής, δε θα ασχοληθούμε με αυτό, σημειώνεται όμως, πως αυτό μπορεί να γίνει με το εργαλείο Vitis, και η διαδικασία που ακολουθείται για να γίνει build το project, φαίνεται παρακάτω :



Εικόνα 5.1 Τρόπος που χτίζεται (build) ο κώδικας στο Vitis.

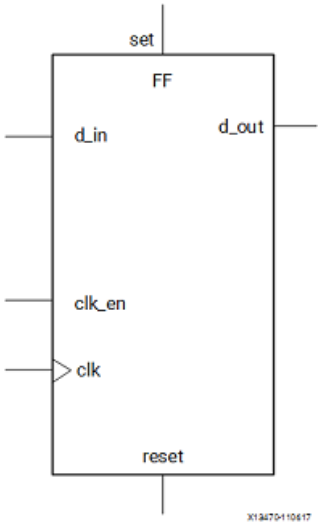
5.2) Xilinx® Alveo™ U200 Data Center accelerator card

Όπως προαναφέρθηκε, τα CLBs (Configurable Logic Blocks) είναι βασικά στοιχεία λογικής που εμπεριέχονται σε ένα FPGA. Με την εσωτερική σύνδεση που υπάρχει μεταξύ των διαφόρων CLBs σε ένα FPGA, μπορούν να εκτελεστούν διάφορες περίπλοκες συναρτήσεις, συναρτήσεις μνήμης αλλά και να συγχρονιστεί ο κώδικας στο FPGA. Τα CLBs αποτελούνται από μικρότερα στοιχεία, τα οποία μπορεί να είναι Flip-Flops (FFs), Look-up Tables (LUTs) ή DSP48 Block.



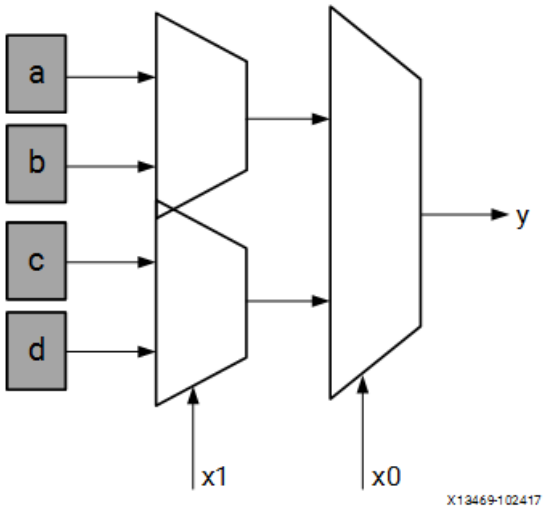
Εικόνα 5.2 Η βασική αρχιτεκτονική των FPGAs, σύνδεση μεταξύ των διαφόρων CLBs.

Το Flip-Flop είναι το μικρότερο στοιχείο αποθήκευσης σε ένα FPGA. Αποτελεί ένα κύκλωμα που είναι ικανό να αποθηκεύσει ένα bit δεδομένων και έχει δύο σταθερές καταστάσεις. Κάθε FF σε ένα CLB είναι ένας δυαδικός καταχωρητής που χρησιμοποιείται για να αποθηκεύσει λογικές καταστάσεις μεταξύ των διαφορετικών κύκλων ρολογιού.



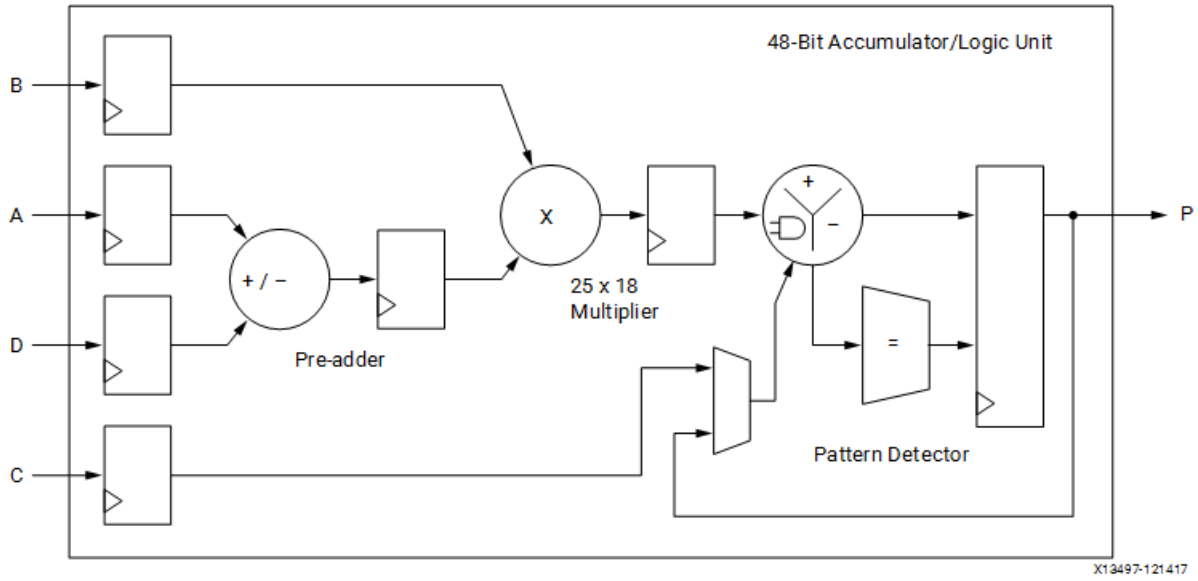
Εικόνα 5.3 Η δομή του Flip-Flop.

Τα LUTs είναι ένα σύνολο από πύλες που είναι καλωδιωμένες στο FPGA. Για κάθε συνδυασμό εισόδων, ένα LUT μπορεί να αποθηκεύσει μια προκαθορισμένη λίστα εξόδων. Τα LUTs αποτελούν έναν γρήγορο τρόπο για να μπορέσουμε να πάρουμε την έξοδο κάποιας λογικής πράξης, καθώς τα αποτελέσματα-έξοδοι είναι ήδη αποθηκευμένα στη λίστα, αντί να υπολογίζονται τη στιγμή που τα χρειαζόμαστε και άρα απλά παίρνουμε την επιθυμητή έξοδο αντί να την επαναϋπολογίσουμε. Στην ουσία τα LUTs, αποτελούνται από τα στοιχεία ενός πίνακα αληθείας, τα οποία εγγράφονται κατά την διαμόρφωση της συσκευής. Μπορούν να χρησιμοποιηθούν ως μνήμες 64-bit, πράγμα ιδιαίτερα σημαντικό στα πλαίσια αυτής της διπλωματικής αφού τέτοια στοιχεία έχουμε κυρίαρχα και αποτελούν την πιο γρήγορη μνήμη που διαθέτει το FPGA.



Εικόνα 5.4 Αναπαράσταση ενός LUT ως ένα σύνολο πυλών.

Το DSP48 block είναι το πιο περίπλοκο υπολογιστικό block στα FPGA της Xilinx. Αποτελεί μια αριθμητική μονάδα λογικής (ALU), ενσωματωμένη στο FPGA και αποτελείται από τρία διαφορετικά blocks. Στο κύκλωμα αυτό, δημιουργείται μια υπολογιστική αλυσίδα, που εμπεριέχει το κομμάτι που εκτελεί την πρόσθεση/αφαίρεση, το οποίο συνδέεται με το κομμάτι που εκτελεί τον πολλαπλασιασμό και τέλος συνδέεται με το κομμάτι της πρόσθεσης/αφαίρεσης/συσσώρευσης της πράξης.



Εικόνα 5.5 Η δομή ενός DSP48 block.

Σε ένα FPGA υπάρχουν ενσωματωμένα στοιχεία μνήμης τα οποία μπορούν να χρησιμοποιηθούν ως μνήμη τυχαίας προσπέλασης (RAM), ως μνήμη μόνο για ανάγνωση (ROM) ή ως καταχωρητές ολίσθησης (shift registers). Η κάρτα διαθέτει άλλα δύο στοιχεία τα οποία θα αναλύσουμε. Την BRAM_18K και τη URAM. Η Block RAM (BRAM_18K), είναι ένας τύπος μνήμης τυχαίας προσπέλασης, είναι ενσωματωμένη στο FPGA και μπορεί να χρησιμοποιηθεί για την αποθήκευση δεδομένων. Είναι δίθυρη, πράγμα που σημαίνει ότι μπορεί να χρησιμοποιηθεί για παράλληλη προσπέλαση στοιχείων και ανάλογα με την κάρτα που χρησιμοποιούμε, μπορεί να αποθηκεύσει 18k ή 36k bits. Σημειώνεται επίσης, πως σε κώδικα OpenCL, που χρησιμοποιείται στο στάδιο του Vitis, όπως προαναφέρθηκε και παραπάνω, οι BRAMs μπορεί να αποτελούν είτε RAM ή ROM. Η UltraRAM (URAM), αποτελεί μια δίθυρη, σύγχρονη μνήμη, που είναι διαθέσιμη στην σειρά UltraScale+™. Δε θα μας απασχολήσει στα πλαίσια αυτής της διπλωματικής.

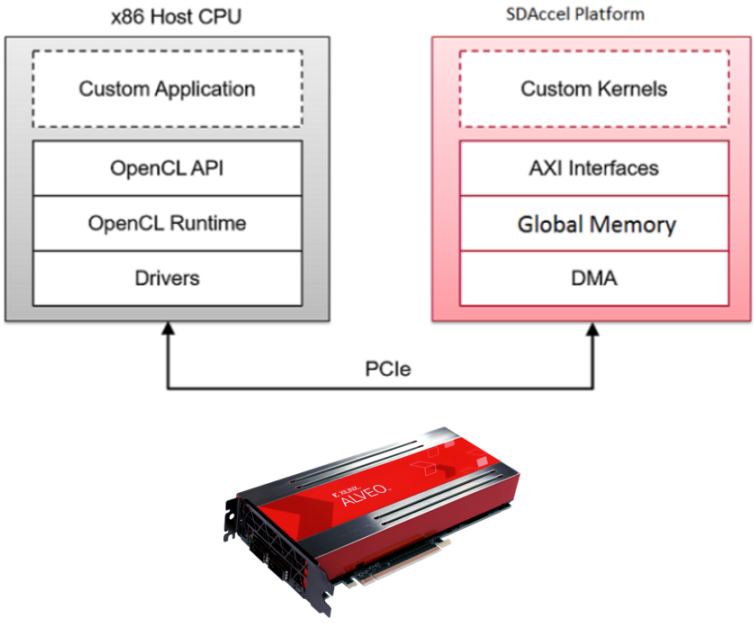
Αναφερόμαστε επίσης, στην έννοια του Super Logic Region (SLR). Το SLR είναι στην ουσία κομμάτι του FPGA, που εμπεριέχεται σε μία Synchronous Serial Interface (SSI) συσκευή. Είναι μια έννοια που θα μας απασχολήσει και στη συνέχεια, καθώς σε κάποια αποτελέσματα που πήραμε, περνούσαμε το όριο των διαθέσιμων πόρων που διαθέτει η κάρτα, αλλά όχι τα συνολικά. Η Alveo™ U200 Data Center accelerator card, έχει 3 τέτοιες περιοχές, πράγμα που φαίνεται και στον παρακάτω πίνακα, με τα συνολικά στοιχεία-πόρους που διαθέτει.

Name	BRAM_18K	DSP48E	FF	LUT	URAM
Available	4320	6840	2364480	1182240	960
Available SLR	1440	2280	788160	394080	320

Πίνακας 5.1 Resources του Xilinx® Alveo™ U200 Data Center accelerator card.

Συνεχίζοντας, αναφερόμαστε στο Advanced eXtensible Interface (AXI). Τα AXI4 και AXI4-Lite, αποτελούν πρωτόκολλα επικοινωνίας της ARM, τα οποία χρησιμοποιούνται στη μεταφορά δεδομένων από τον host-CPU στον kernel-FPGA. Το AXI4-Lite αποτελεί υποσύνολο των δυνατοτήτων του AXI4. Στα πλαίσια της διπλωματικής αυτής, το σημαντικό στοιχείο αυτών των πρωτοκόλλων είναι η δυνατότητα που δίνει στο FPGA να κάνει Burst R/W. Η εγγραφή και η ανάγνωση που γίνεται κατά τη μεταφορά δεδομένων από τον host στον kernel, εισάγει μια καθυστέρηση η οποία αρκετές φορές είναι δαπανηρή.

Σε μια τυπική εφαρμογή που εκτελείται στον host-CPU, τα δεδομένα μεταφέρονται από τη μνήμη RAM του PC, στη global μνήμη του FPGA. Η διασύνδεση μεταξύ τους μπορεί να γίνεται με PCI-express. Ο kernel-FPGA επεξεργάζεται τα δεδομένα και αποθηκεύει τα αποτελέσματα πίσω στην global μνήμη. Μετά την ολοκλήρωση της διαδικασίας αυτής, μεταφέροντα τα αποτελέσματα πίσω στη μνήμη RAM του host-PC. Οι μεταφορές δεδομένων μεταξύ της RAM του PC και της global μνήμης του FPGA εισάγουν καθυστέρηση, η οποία μπορεί να είναι δαπανηρή για τη συνολική εφαρμογή. Για να επιτευχθεί επιτάχυνση σε ένα πραγματικό σύστημα, τα οφέλη που επιτυγχάνονται από τους πυρήνες επιτάχυνσης υλικού πρέπει να υπερτερούν του πρόσθετου χρόνου της μεταφοράς δεδομένων. Για το λόγο αυτόν, τα παραπάνω πρωτόκολλα επικοινωνίας χρησιμοποιούνται κυρίως, λόγω της δυνατότητας του Burst R/W που δίνουν στο να μειωθεί αρκετά καθυστέρηση αυτή. Σχηματικά, ο τρόπος διασύνδεσης φαίνεται παρακάτω.



Εικόνα 5.6 Τρόπος διασύνδεσης μεταξύ του Host και του Kernel.

5.3) Περιγραφή κάποιων βασικών HLS directives

Τα High Level Synthesis directives, είναι κάποιες βασικές εντολές-γραμμές κώδικα, με τη χρήση των οποίων επιτυγχάνεται η καλύτερη αξιοποίηση του FPGA για την ταχύτερη εκτέλεση του προγράμματος/σχεδίασης, σε σχέση με την απλή εκτέλεσή του σε κάποιον υπολογιστή. Υπάρχουν διάφοροι περιορισμοί σε σχέση με το ποια directives μπορούν να χρησιμοποιηθούν, πράγμα που καθορίζεται κυρίως από τη δομή και τον τρόπο λειτουργίας του κώδικα, αλλά και από τους περιορισμούς των πόρων που διαθέτει το FPGA. Τα directives αυτά, γράφονται στον source κώδικα, σε συγκεκριμένα σημεία, ανάλογα με το περιεχόμενο του κώδικα αυτού. Στη διπλωματική αυτή, χρησιμοποιήθηκαν συγκεκριμένα directives, για τα οποία και θα γίνει ανάλυση στη συνέχεια, λόγω της περίπλοκης φύσης του κώδικα. Όπως θα δούμε στη συνέχεια, υπήρξαν διάφορες σχεδιάσεις, όσον αφορά τον κώδικα, για αυτό και ανάλογα με την περίπτωση χρησιμοποιούνται κάποια από τα παρακάτω directives, καθώς σε κάποιες περιπτώσεις, δεν ήταν εφικτή η χρήση τους. Τα directives αυτά είναι τα εξής :

i) ***#pragma HLS TOP name=<string>*** : Καθορίζει το όνομα της top function της σχεδίασής μας. Δεν χρειάστηκε η χρήση του, καθώς αρκεί ο απλός καθορισμός της συνάρτησης που είναι top, μέσω του gui του Vivado HLS. Η top συνάρτηση, είναι η συνάρτηση η οποία θα τρέξει στο FPGA ή αλλιώς, είναι η βασική συνάρτηση που υλοποιεί τη σχεδίασή μας.

ii) ***#pragma HLS LOOP_TRIPCOUNT min=<int> max=<int> avg=<int>*** : Με αυτό το directive, μπορεί να καθοριστεί ο αριθμός των επαναλήψεων κάποιας loop μέσα στον κώδικα. Γενικά δεν έχει κάποια άμεση επίπτωση στην απόδοση του κώδικα, βοηθάει όμως στο να μπορέσει το εργαλείο Vivado HLS, να βγάλει μια καλύτερη εκτίμηση για το συνολικό ελάχιστο, μέγιστο και το μέσο όρο του latency που θα προκύψει μεταξύ διαδοχικών κύκλων ρολογιού στο FPGA. Τα min, max, avg, στο directive δίνουν αυτόν ακριβώς τον αριθμό των επαναλήψεων της λούπας, με τον αναγκαίο προσδιορισμό τουλάχιστον του ενός από τα τρία variables.

iii) ***#pragma HLS ARRAY_PARTITION variable=<name> <type> factor=<int> dim=<int>*** : Ο κώδικας αυτός, χωρίζει τον πίνακα που δίνουμε ως ***variable***, σε μικρότερους υποπίνακες. Με αυτόν τον τρόπο, το εργαλείο μπορεί να αξιοποιήσει καλύτερα τους διαθέσιμους πόρους του FPGA, χρησιμοποιώντας θέσεις στη μνήμη ή πολλαπλούς registers, αντί να χρησιμοποιήσει ένα μεγάλο κομμάτι της μνήμης για έναν πίνακα, κατά τη δημιουργία του RTL. Αυξάνονται έτσι οι διαθέσιμες θύρες (ports) που μπορούν να αξιοποιηθούν μεταξύ του Host (PC) και του Kernel (FPGA) ως είσοδοι και έξοδοι (I/O), αυξάνοντας όμως τον αριθμό των κομματιών της μνήμης που χρησιμοποιούνται. Ένας τύπος (***type***) χωρισμού αυτού, μπορεί να είναι ο ***cyclic***, δηλαδή βάζοντας ένα στοιχείο τη φορά για κάθε πίνακα, ο αριθμός των οποίων καθορίζεται από το ***factor=N***, κυκλικά. Ένας άλλος είναι το ***block***, που χωρίζει σε διαδοχικά ***N*** block τον πίνακα παίρνοντας διαδοχικά τα στοιχεία. Ο τελευταίος τρόπος είναι το ***complete***, που χωρίζει

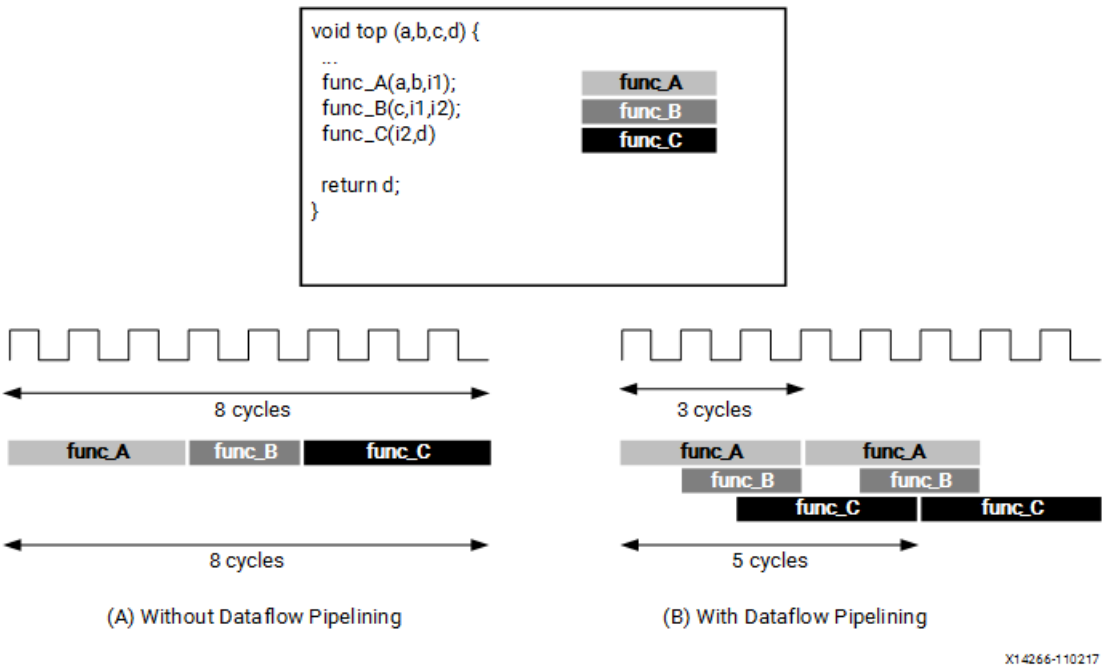
κάθε στοιχείο του πίνακα σε νέους πίνακες με μόνο ένα στοιχείο. Στην ουσία, έτσι έχουμε στη μνήμη ξεχωριστούς registers, για κάθε στοιχείο. Αν έχουμε πολλές διαστάσεις στον πίνακα που θέλουμε να χωρίσουμε, το ποια διάσταση θέλουμε να γίνει partition, καθορίζεται από το ***dim***. Το directive αυτό από μόνο του, δεν έχει κάποια επίδραση στην απόδοση της σχεδίασης.

Όπως θα δούμε και παρακάτω και ιδιαίτερα στο επόμενο κεφάλαιο που εξηγούνται τα προβλήματα που αντιμετωπίστηκαν, ο κώδικας αυτός, μπορεί να χρησιμοποιηθεί για στατικούς πίνακες και όχι για scalar type μεταβλητές, όπως είναι οι pointers. Επειδή είχαμε μεγάλους πίνακες (393485 στοιχείων uint64_t), η χρήση στατικών πινάκων ήταν απαγορευτική. Χρησιμοποιήθηκε όμως το παραπάνω pragma, σε μια από τις υλοποιήσεις που έγιναν, όπου χωρίζονται οι πίνακες αυτοί σε πίνακες 1024 στοιχείων, για αυτό και αναφερόμαστε σε αυτό.

iv) ***#pragma HLS INTERFACE <mode> port=<name> bundle=<string> register_mode=<mode> depth=<int> offset=<string> clock=<string> name=<string> num_read_outstanding=<int> num_write_outstanding=<int> max_read_burst_length=<int> max_write_burst_length=<int>*** : Σε μια σχεδίαση RTL, οι είσοδοι και οι έξοδοι, πρέπει να καθοριστούν και να υλοποιηθούν μέσω κάποιας θύρας (port), η οποία θα ακολουθεί και κάποιο συγκεκριμένο πρωτόκολλο επικοινωνίας. Στα FPGA, τα πιο γνωστά πρωτόκολλα επικοινωνίας είναι το PCI-e, το AXI4 και το AXI4-lite, στα οποία έγινε αναφορά και παραπάνω. Το directive αυτό, στην ουσία, είναι ιδιαίτερα χρήσιμο στον καθορισμό του πώς θα περαστούν ορίσματα της top function. Αυτό το directive, αν δεν χρησιμοποιηθεί, δεν αξιοποιούνται οι BRAM και οι URAM που είναι διαθέσιμες στο FPGA, αλλά το εργαλείο Vivado HLS κρίνει το που θα βρίσκονται τα διάφορα στοιχεία του κώδικα όπως είναι οι διάφορες μεταβλητές, οι πίνακες και οι pointers. Τα βασικά ***mode*** που μπορεί να πάρει αυτό το directive είναι το ***axis***, όπου χρησιμοποιείται το πρωτόκολλο AXI4 για κάποιον στατικό πίνακα, το ***m_axi***, όπου χρησιμοποιείται το ίδιο πρωτόκολλο, αλλά για κάποια scalar μεταβλητή όπως είναι οι pointers, που για να μπει στην global memory χρησιμοποιείται το ***bundle=gmemN***, όπου ***N*** είναι 0,1,2,… οι θέσεις στη μνήμη που θα πάρουν οι μεταβλητές αυτές. Τέλος, το ***s_axilite***, που χρησιμοποιείται το πρωτόκολλο AXI4-lite, χρησιμοποιείται συνήθως μετά τη χρήση και του ***m_axi***, για να δημιουργείται άλλη μια θύρα (port) για τις μεταβλητές αυτές για τον έλεγχό τους (***bundle=control***). Χρησιμοποιείται ακόμη και για τη δημιουργία ενός port, για την επιστροφή από την εκτέλεση στο FPGA, στο αρχικό πρόγραμμα testbench που τρέχει στον Host (CPU), με ***bundle=return***.

Η ακόμα μεγαλύτερη χρησιμότητα του παραπάνω directive, είναι για την χρήση και του DATAFLOW (ή/και του STREAM) directive, το οποίο όμως δεν χρησιμοποιήθηκε σε αυτήν την διπλωματική, καθώς δεν μπόρεσαν να ικανοποιηθούν κάποιοι περιορισμοί, όπως το να μην υπάρχουν operations

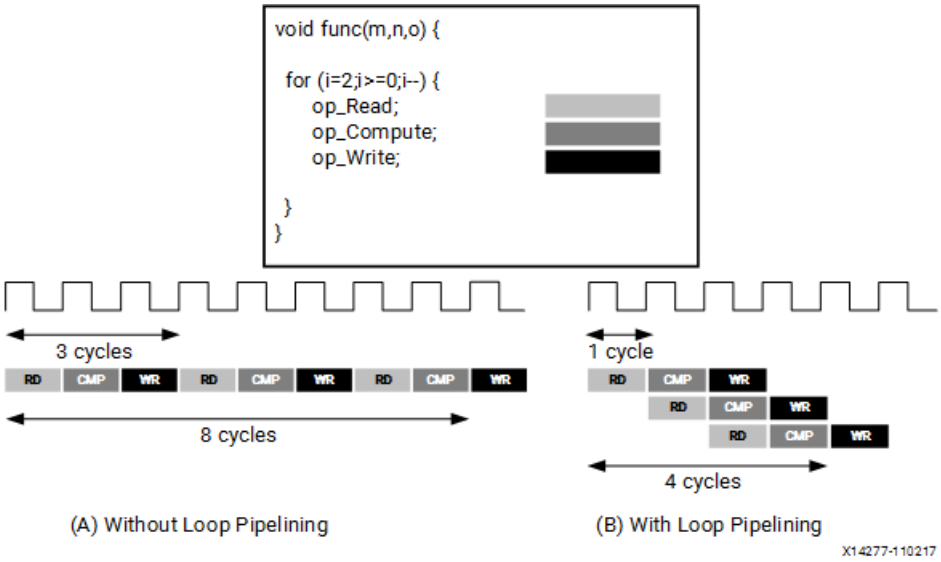
λογικής, που είναι απαραίτητο να ικανοποιείται για τη χρήση του. Στην ουσία, το DATAFLOW directive, υλοποιεί task-level pipelining, επιτρέποντας συναρτήσεις και λούπες να επικαλύπτονται (overlap), αυξάνοντας έτσι την χρονική επίδοση και το πόσο ταυτόχρονα γίνεται η εκτέλεση (concurrency). Φαίνεται παρακάτω, η λογική του DATAFLOW :



Εικόνα 5.7 Task-Level Pipelining με το Dataflow directive.

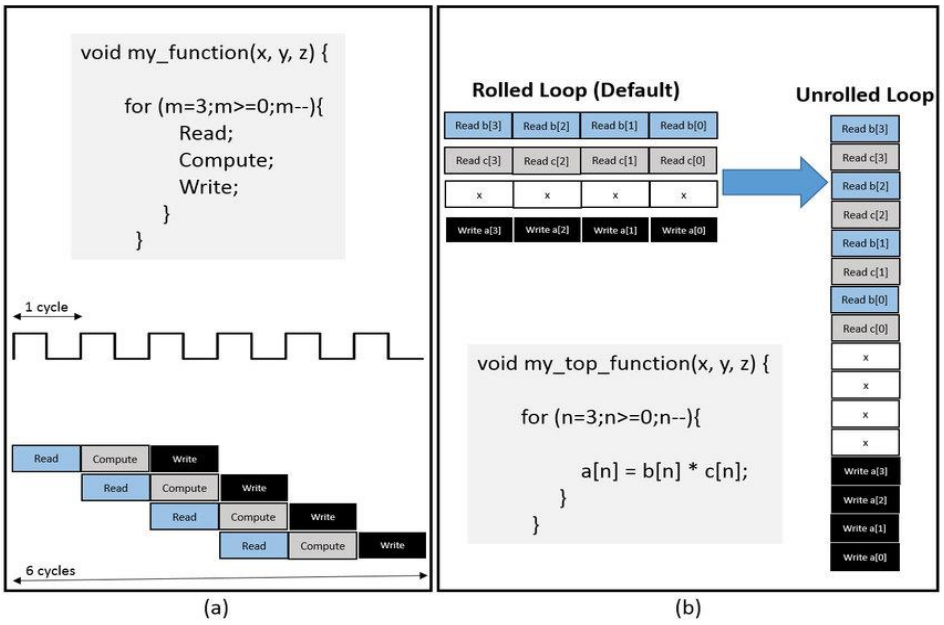
v) ***#pragma HLS DEPENDENCE variable=<variable> <class> <type> <direction> distance=<int> <dependent>*** : Με το directive αυτό, καθορίζονται πληροφορίες για διάφορες εξαρτήσεις που μπορεί να υπάρχουν μεταξύ μεταβλητών μέσα σε κάποια επανάληψη (loop-independent dependence) ή μεταξύ διαφορετικών επαναλήψεων κάποιας λούπας (loop-carry dependence). Με αυτόν τον τρόπο, μπορεί να επιτραπεί ή όχι το να γίνει pipelined κάποια λούπα που είχε κάποια ψευδή εξάρτηση. Το ***type*** μπορεί να είναι ***intra***, αν η εξάρτηση είναι μέσα στην ίδια την επανάληψη της λούπας ή ***inter***, αν η εξάρτηση βρίσκεται σε ξεχωριστές επαναλήψεις της λούπας. Το ***direction***, μπορεί να είναι ***Read-After-Write (RAW)***, ***Write-After-Read (WAR)*** ή τέλος ***Write-After-Write (WAW)***, ανάλογα με τη δομή του κώδικα και των operations που γίνονται με τις μεταβλητές-εξαρτήσεις. Το ***distance***, προσδιορίζει την απόσταση μεταξύ των ***inter*** επαναλήψεων που βρίσκονται οι μεταβλητές-εξαρτήσεις και χρησιμοποιείται μόνο αν η εξάρτηση είναι αληθής, όπου η αλήθεια καθορίζεται από το ***dependent***. Σημαντική σημείωση είναι, ότι μπορούμε να γλυτώσουμε την χρήση του directive αυτού, αυξάνοντας το Initiation Interval στην παρακάτω εντολή. Παρακάτω, θα συζητηθεί αναλυτικότερα το συγκεκριμένο ζήτημα, μιας και μας απασχόλησε και στον κώδικά μας.

vi) ***#pragma HLS PIPELINE II=<int> enable_flush rewind*** : Με αυτήν τη γραμμή κώδικα, μειώνεται το διάστημα έναρξης εκτέλεσης των operations (Initiation Interval), που βρίσκονται μέσα σε κάποια επανάληψη ή συνάρτηση, επιτρέποντας έτσι την ταυτόχρονη εκτέλεση operations (concurrency). Το Initiation Interval καθορίζεται σε έναν ακέραιο αριθμό κύκλων ρολογιού, με το προκαθορισμένο να είναι το 1, που είναι και το βέλτιστο. Αν το εργαλείο δεν καταφέρνει λόγω εξαρτήσεων (dependencies) να πετύχει το βέλτιστο αυτό διάστημα, για μια συγκεκριμένη περίοδο ρολογιού, τότε προτείνει τον αμέσως επόμενο μικρότερο δυνατό αριθμό, για τον οποίο ικανοποιούνται οι συνθήκες για να εφαρμοστεί PIPELINE. Το ***enable_flush*** αδειάζει τα δεδομένα της εισόδου του PIPELINE για συναρτήσεις όταν αυτό γίνεται ανενεργό και το ***rewind*** μπορεί να εκτελεί διαδοχικά επαναλήψεις κάποιας τέλειας λούπας. Τα δύο αυτά flags είναι προαιρετικά. Το directive αυτό, επηρεάζει άμεσα και καταλυτικά την απόδοση της εκτέλεσης στο FPGA, καθώς έχουμε concurrency. Στα αποτελέσματα, φαίνεται και η διαφορά που μπορεί να προκύψει για διαφορετικές τιμές στο Initiation Interval, για διάφορες περιόδους ρολογιού. Είναι ιδιαίτερα σημαντικό, το που θα τοποθετηθεί μέσα στον κώδικα, αυτό το directive, καθώς θα παίρνουμε διαφορετικά αποτελέσματα αν κάνουμε διαφορετικά κομμάτια του κώδικα PIPELINE. Επίσης σημαντικό είναι να σημειωθεί, ότι το κομμάτι του κώδικα που βρίσκεται μετά το directive, θα πρέπει να μπορεί να γίνεται unroll, το οποίο για να γίνει, χρειάζεται να ικανοποιούνται διάφοροι περιορισμοί που συζητιούνται παρακάτω. Σε περίπτωση που δεν μπορεί να γίνει unroll, η λούπα ή η συνάρτηση στην οποία εφαρμόζεται το directive, δε θα γίνει PIPELINE. Παρ' όλα αυτά, αποτελεί ίσως το πιο σημαντικό directive για την βελτιστοποίηση της απόδοσης, καθώς ακόμα και με τους περιορισμούς, μπορεί να εφαρμοστεί ευκολότερα, απ' ότι το directive που ακολουθεί.



Εικόνα 5.8 Παράδειγμα με το Pipeline directive.

vii) ***#pragma HLS UNROLL factor=<N> region skip_exit_check*** : Αυτό το directive, μπορεί να χρησιμοποιηθεί μέσα σε κάποια λούπα και την μεταμορφώνει δημιουργώντας πολλά αντίγραφα με το περιεχόμενο της λούπας στο RTL. Με αυτόν τον τρόπο, κάποιες ή όλες οι επαναλήψεις της λούπας, γίνονται παράλληλα. Αν δεν προσδιοριστεί το ***factor***, η λούπα ξεδιπλώνεται ολόκληρη, αλλιώς ξεδιπλώνονται οι ***factor*** επαναλήψεις της. Το ***region*** είναι προαιρετικό και αυτό, και ξεδιπλώνει όλες τις λούπες που βρίσκονται κάτω από τη θέση που τοποθετείται το directive αφήνοντας την εξωτερική λούπα διπλωμένη (που βρίσκεται πάνω από τη θέση του directive. Το ***skip_exit_check***, χρησιμοποιείται όταν έχουμε μερικό unrolling, δηλαδή έχει προσδιοριστεί και το ***factor*** και είναι μικρότερο από το σύνολο των επαναλήψεων της λούπας και αφαιρεί τον έλεγχο του εργαλείου σχετικά με την έξοδο από τη λούπα, που μπορεί να είναι για παράδειγμα κάποιο break. Το keyword αυτό, δεν μπορεί στην ουσία να χρησιμοποιηθεί για μεταβλητό αριθμό επαναλήψεων, που δεν είναι και πολλαπλάσιο του ***factor***, αλλά και όταν έχουμε πραγματικά να κάνουμε κάποιον έλεγχο εξόδου μέσα στη λούπα, που δεν μπορούμε να αποφύγουμε. Με το unroll, δημιουργούνται κι άλλα operations μέσα στη λούπα, αλλά έχει κρίσιμους περιορισμούς για να μπορέσει να εφαρμοστεί. Όπως θα δούμε και στα προβλήματα που αντιμετωπίστηκαν, δεν μπορεί να γίνει unroll μια λούπα η οποία έχει μεγάλο αριθμό επαναλήψεων, αλλά και δεν μπορεί να γίνει unroll μια λούπα που έχει μεταβλητό αριθμό επαναλήψεων και δεν είναι πολλαπλάσιο του ***factor*** για να γίνει μερικώς unroll. Σε δημοσιεύσεις που βρέθηκαν, αλλά και στο forum της Xilinx, προτείνεται η μετατροπή στις λούπες που είναι υλοποιημένες με while, σε for για το μέγιστο πιθανό αριθμό επαναλήψεων και if για τον έλεγχο της λογικής ή αν αυτό δεν είναι εφικτό, να χρησιμοποιηθεί το PIPELINE directive, το οποίο δεν έχει αυτούς τους περιορισμούς.



Εικόνα 5.9 Διαφορές μεταξύ Loop Pipeline και Loop Unroll.

viii) ***#pragma HLS loop_flatten off*** : Αναφερόμαστε, τέλος, στο directive αυτό, το οποίο δίνει τη δυνατότητα να γίνει flatten κάποια συγκεκριμένη λούπα, δηλαδή αυτή και όλες οι υπόλοιπες λούπες που μπορεί να υπάρχουν εξωτερικά της, να μετασχηματιστούν σε μία λούπα, αρκεί να είναι τέλειες (perfect loops), δηλαδή να μην υπάρχει κώδικας λογικής μεταξύ τους και να μην είναι μεταβλητού αριθμού επαναλήψεων. Με το προαιρετικό keyword ***off***, το εργαλείο καταλαβαίνει πως δεν πρέπει να κάνει flatten αυτήν τη λούπα, καθώς μπορεί να εμπεριέχει κώδικα λογικής ή κάποιο μεγάλο. Η αναφορά αυτή γίνεται σε αυτό το pragma, καθώς έγιναν και κάποιες δοκιμές με πιο πρόσφατες εκδόσεις του εργαλείου Vivado HLS, στις οποίες το εργαλείο προσπαθούσε αυτόματα κατά τη σύνθεση να κάνει optimize κάποιο κομμάτι του κώδικα για να μπορέσει να βελτιώσει την απόδοσή του στο FPGA. Σε κάποιες τέτοιες περιπτώσεις, λοιπόν, έγινε χρήση του directive αυτού, ώστε να μη προσπαθεί το εργαλείο να κάνει flatten κάποια λούπα που δεν μπορεί ή δεν πρέπει να βελτιστοποιηθεί με αυτό τον τρόπο.

5.4) Ανάλυση με το Intel® VTune™ Profiler

Η χρήση του εργαλείου αυτού ήταν καθοριστική, για να μπορέσει να βρεθεί το κομμάτι του κώδικα μέσα στη βιβλιοθήκη Microsoft SEAL, το οποίο είναι αρκετά χρονοβόρο, το οποίο και θα έπρεπε να προσπαθήσουμε για να γίνει accelerate. Για να γίνει η ανάλυση αυτή, φτιάχτηκε ένα νέο .cpp αρχείο το οποίο περιείχε ολόκληρο το παράδειγμα-testbench με τη σχεδίαση (source) να είναι ενσωματωμένη σε αυτό. Σε αυτό το σημείο, πρέπει να κάνουμε μια πολύ σημαντική παρατήρηση. Οι προκαθορισμένες (default) επιλογές με τις οποίες γίνεται build ένα παράδειγμα το οποίο βασίζεται στην βιβλιοθήκη Microsoft SEAL, είναι αρκετές και επηρεάζουν αρκετά την απόδοση του προγράμματος που προκύπτει. Πήραμε, όμως, αρκετά διαφορετικά αποτελέσματα, χρησιμοποιώντας το -O3 compiler optimization flag, για να κάνουμε build το παράδειγμα και χωρίς αυτό. Και στις δύο περιπτώσεις όμως, οι συναρτήσεις που επιλέχθηκαν ήταν αρκετά χρονοβόρες και είχαν νόημα να γίνουν accelerate. Υπήρξε το πρόβλημα, ότι το flag αυτό δεν γίνεται να περαστεί στο C/RTL Cosimulation, πρόβλημα που θα συζητηθεί στο 6ο Κεφάλαιο. Σε κάθε περίπτωση, κάνοντας την ανάλυση μέσα από το Intel® VTune™ Profiler, μπορούμε να παρατηρήσουμε ότι η πιο κοστοβόρα συνάρτηση είναι η divide_uint128_uint64_inplace_generic();. Ο αρχικός χρόνος εκτέλεσης της συνάρτησης αυτή είναι περίπου ***560ms***, χωρίς το optimization flag -O3, ενώ με τη χρήση του flag, το αποτέλεσμα που παίρνουμε είναι ***164ms***, για απλή εκτέλεση στον Host-PC. Παρ' όλο που αυτό το flag, δεν περάστηκε και ίσως δε γίνεται να περαστεί στο C/RTL Cosimulation, η σύγκριση των αποτελεσμάτων που θα γίνει, θα αφορά την τιμή 164ms, καθώς πρέπει να υπερβούμε τον χρόνο εκτέλεσης στον Host, που υποθετικά θα χρησιμοποιεί αυτό το flag. Όσον αφορά τη δεύτερη συνάρτηση, την encrypt_zero_symmetric();, η σχεδίαση δεν προχώρησε πλήρως, καθώς δεν υπήρχε ευκολία στο να σπάσει από τη βιβλιοθήκη Microsoft SEAL, λόγω μεγάλης αλληλεξάρτησης με διάφορα κομμάτια κώδικα. Μέχρι εκεί που έφτασε, δεν μας έδωσε κάποια σημαντική βελτίωση στον χρόνο, για αυτό και δε θα τη λάβουμε υπόψιν. Την αναφέρουμε όμως, καθώς ασχοληθήκαμε και με αυτήν και θα αναφερθεί στο 6ο κεφάλαιο. Ακολουθούν τα αποτελέσματα που πήραμε, από το εργαλείο Intel® VTune™ Profiler.

Function	CPU Time: Total
seal::util::divide_uint128_uint64_inplace_generic	43.2%
seal::util::ntt_negacyclic_harvey_lazy	16.9%
blake2b_compress	10.5%
seal::util::multiply_poly_scalar_coeffmod	5.3%
seal::util::encrypt_zero_symmetric	37.9%
seal::util::dyadic_product_coeffmod	3.2%
__memset_avx2_erms	3.2%
blake2b_init_param	3.2%
seal::util::try_minimal_primitive_root	2.1%
seal::util::NTTTables::initialize	47.4%
seal::util::GaloisTool::generate_table_ntt	2.1%
seal::util::sample_poly_cbd	2.1%
seal::util::Arithmetic<std::complex<double>, std::complex<double>, double>::guard	2.1%
seal::Encryptor::encrypt_zero_internal	2.1%
seal::KeyGenerator::generate_kswitch_keys	3.2%
seal::KeyGenerator::create_relin_keys	3.2%
seal::KeyGenerator::create_relin_keys	3.2%
seal::SEALContext::validate	47.4%
seal::Ciphertext::resize_internal	3.2%
seal::Ciphertext::resize	3.2%
seal::SEALContext::SEALContext	47.4%
seal::Encryptor::encrypt	2.1%
seal::Encryptor::encrypt_internal	2.1%
seal::Evaluator::rescale_to_next_inplace	3.2%
seal::util::encrypt_zero_asymmetric	2.1%
seal::SEALContext::SEALContext	47.4%
_start	100.0%
seal::util::GaloisTool::apply_galois_ntt	2.1%
__libc_start_main	100.0%
seal::CKKSEncoder::decode<double, void>	2.1%
seal::CKKSEncoder::decode_internal<double, void>	2.1%
seal::util::DWTHandler<std::complex<double>, std::complex<double>, double>::transform_to_rev	2.1%
main	100.0%
seal::UniformRandomGenerator::generate	13.7%
seal::KeyGenerator::create_galois_keys	42.1%
seal::SEALContext::create_next_context_data	31.6%
seal::util::RNSTool::initialize	30.5%
seal::util::RNSTool::RNSTool	30.5%
seal::util::allocate<seal::util::RNSTool, unsigned long&, seal::util::RNSBase&, seal::Modulus const&, seal::Mem	30.5%
seal::KeyGenerator::create_galois_keys	42.1%
blake2xb_final	13.7%
seal::Blake2xbPRNG::refill_buffer	13.7%
blake2xb	13.7%
seal::KeyGenerator::create_galois_keys	42.1%
seal::util::sample_poly_uniform	13.7%
blake2b_final	10.5%
seal::KeyGenerator::generate_one_kswitch_key	43.2%
seal::util::NTTTables::NTTTables	47.4%
seal::util::CreateNTTTables	47.4%
seal::util::RNSTool::divide_and_round_q_last_ntt_inplace	3.2%
seal::Evaluator::mod_switch_scale_to_next	3.2%
seal::Evaluator::rescale_to_next	3.2%

Εικόνα 5.10 Αποτελέσματα ανάλυσης με το Intel® VTune™ Profiler, με το optimization flag -O3 ενεργοποιημένο.

5.5) Περιγραφή της υλοποίησης

Θα μπορούσαμε να κάνουμε ξεχωριστά υποκεφάλαια για την υλοποίηση του testbench και για την υλοποίηση του source. Επειδή όμως φτιάξαμε διάφορες σχεδιάσεις, για να μπορέσουμε να πάρουμε διάφορα αποτελέσματα και να τα ερμηνεύσουμε, έχει περισσότερο νόημα να αναλύσουμε την κάθε σχεδίαση ξεχωριστά. Όπως θα έχει γίνει ήδη αντιληπτό, το CKKS scheme και γενικότερα τα Fully Homomorphic Encryption schemes, είναι ιδιαίτερα περίπλοκα. Εδώ έρχεται και το Microsoft SEAL, το οποίο συζητήθηκε στο προηγούμενο κεφάλαιο, καθώς έχει ως στόχο την διευκόλυνση χρήσης κάποιου scheme, σε λίγες μόλις γραμμές. Ιδανικά, για να γίνει πιο εύκολα κάποια υλοποίηση για να αξιοποιηθούν και τα εργαλεία της Xilinx, θα θέλαμε ένα μόνο αρχείο .cpp, το οποίο θα εμπεριείχε μερικές χιλιάδες γραμμές κώδικα, από τις οποίες θα διαλέγαμε κάποιο κομμάτι και θα το κάναμε accelerate, δηλαδή θα το τρέχαμε ως source στο kernel/FPGA. Στο 4ο κεφάλαιο, περιγράψαμε, στην ουσία, ολόκληρο το testbench-παράδειγμα με τη σειρά. Εδώ θα εξηγήσουμε λίγο πιο αναλυτικά κάποια πράγματα που θεωρούμε ιδιαίτερα σημαντικά.

Γενικεύοντας τα παραπάνω, η λογική για να επιτευχθεί acceleration και να μπορέσουμε να αξιοποιήσουμε τα εργαλεία της Xilinx, είναι να μπορέσουμε να έχουμε ως testbench τον κώδικα-παράδειγμα που υλοποιεί ολόκληρο το scheme, χρησιμοποιώντας της συναρτήσεις της βιβλιοθήκης Microsoft SEAL και ως source, να έχουμε κάποιο κομμάτι αυτού του κώδικα, τροποποιημένο, ο οποίος θα τρέχει στο FPGA. Το να μπορέσει να γίνει το παραπάνω, ήταν ιδιαίτερα περίπλοκο, λόγω της φύσης και των πολλών υποσυναρτήσεων και κλάσεων που εμπεριέχει η βιβλιοθήκη SEAL και θα αναλυθεί και στο επόμενο κεφάλαιο. Ένα άλλο σημαντικό στοιχείο που πρέπει να σημειωθεί, είναι ότι λόγω διάφορων προβλημάτων που εμφάνιζε το εργαλείο, δεν καταφέραμε να έχουμε στο testbench χρήση καμιάς συνάρτησης του SEAL, για να γίνει το C/RTL Cosimulation. Αυτό συνέβαινε, καθώς δεν υπήρχε τρόπος για να μπορέσει το εργαλείο να κάνει την παραπάνω διαδικασία, έχοντας κάποια συγκεκριμένα include και linker flags, τα οποία είναι απαραίτητα για να γίνει compile οποιοδήποτε αρχείο .cpp, που χρησιμοποιεί τη βιβλιοθήκη SEAL. Θα συζητηθεί και αυτό στο επόμενο κεφάλαιο. Συνεπώς, το testbench μας δεν χρησιμοποιεί στην ουσία κάτι από τη βιβλιοθήκη Microsoft SEAL, αλλά προσομοιώνουμε τη λογική που περιεγράφηκε παραπάνω με διαφορετικό τρόπο.

Πιο συγκεκριμένα, δώσαμε βάση στο να μπορέσουμε να κάνουμε accelerate τη συνάρτηση divide στην οποία αναφερθήκαμε παραπάνω. Αναλύοντας τον κώδικα του παραδείγματος και μεταβάλλοντάς τον, καταλήξαμε πως η συνάρτηση αυτή, που θα αποτελέσει την top function του source αρχείου, καλείται ακριβώς 393485 φορές. Τα ορίσματά της είναι ο αριθμητής (***numerator***), παρονομαστής (***denominator***) και (***quotient***) και είναι μεταβλητές τύπου unsigned int 64-bit (uint64_t). Η συνάρτηση παίρνει αυτά ως ορίσματα και χωρίς να εμβαθύνουμε περισσότερο, χρησιμοποιείται μια μαθηματική διαδικασία, που ονομάζεται ακολουθία ***de Bruijn*** και έτσι προκύπτει ο 128-bit αριθμός, που είναι ο ***numerator***. Αναλύοντας περαιτέρω τον κώδικα, προκύπτει ότι για το συγκεκριμένο παράδειγμα, τα στοιχεία-ορίσματα που χρησιμοποιούνται και παράγονται αυτές τις 393485 φορές, είναι πάντα τα ίδια, για αυτό το παράδειγμα. Συνεπώς, πήραμε σε αρχεία τα στοιχεία αυτά, τα περνάμε στον host με τον κώδικα του testbench και καλούμε τη συνάρτηση-σχεδίαση που τρέχει στο

FPGA. Ο κώδικας, λοιπόν, του testbench κάνει μόνο αυτήν τη δουλειά, δηλαδή προσομοιώνει την εκτέλεση για 393485 φορές της συνάρτησης divide_uint128_uint64_inplace_generic();, πράγμα το οποίο γίνεται, κατά την απλή εκτέλεση του παραδείγματος. Στην ουσία δηλαδή, είτε μέσα στο source κώδικα, είτε μέσα στο testbench, η συνάρτηση για τον υπολογισμό αυτόν, θα τρέξει 393485 φορές. Αυτό που ίσως δυσκόλεψε τη σχεδίαση, είναι μια while που υπάρχει μέσα σε κάθε επανάληψη, πράγμα που δυσκολεύει το scheduling και γενικότερα την επιτάχυνση και θα συζητηθεί και παρακάτω.

Σε γενικές γραμμές, η λογική πίσω από τον κώδικα είναι παρόμοια. Θα αναλύσουμε το βασικό σκεπτικό και εν συνεχεία θα αναλυθούν οι παραλλαγές-σχεδιάσεις που έγιναν. Αλγοριθμικά, λοιπόν, η βασική λογική πίσω από τον κώδικα σχετικά με το testbench είναι η παρακάτω (το times που θα δούμε, είναι οι 393485 φορές, μία φορά για κάθε εκτέλεση που γίνεται στο παράδειγμα της βιβλιοθήκης Microsoft SEAL, για το CKKS scheme) :

i) Διαβάζονται τα αρχεία που εμπεριέχουν τους μεγάλους πίνακες, με τον αριθμητή και τον παρονομαστή και αρχικοποιείται ο pointer στον οποίο θα εγγραφούν τα αποτελέσματα της διαίρεσης.

ii) Καλείται και χρονομετρείται η εκτέλεση της top συνάρτησης div που δημιουργήσαμε και είναι υλοποιημένη στο source αρχείο. Η χρονομέτρηση αυτή, έχει σχέση με απλή εκτέλεση συνολικά του προγράμματος στον host (υπολογιστή).

iii) Επαληθεύονται τα αποτελέσματα της πράξης της διαίρεσης, διαβάζοντας από το αρχείο των αποτελεσμάτων και συγκρίνοντάς τα με τα αποτελέσματα που πήραμε από την υλοποίησή μας.

```
int main(int argc, char** argv){

  //Read numerator
  uint64_t *num = (uint64_t *) malloc(sizeof(uint64_t) * times);
  // Read from the text file
  ifstream f1;
  f1.open("numerator.txt");
  int i = 0;
  if(f1.is_open()){
    while (!f1.eof() )
    {
      f1>>num[i];
      i++;
    }
  }
  f1.close();

  //Read denominator
  uint64_t *den = (uint64_t *) malloc(sizeof(uint64_t) * times);
  // Read from the text file
  ifstream f2;
  f2.open("denominator.txt");
  i = 0;
  if(f2.is_open()){
    while (!f2.eof() )
    {
      f2>>den[i];
      i++;
    }
  }
  f2.close();

  //Initialize quotient
  uint64_t *quo = (uint64_t *) malloc(sizeof(uint64_t) * times);
```

Εικόνα 5.11 Αλγόριθμος για το διάβασμα των αρχείων και την αρχικοποίηση των μεταβλητών που θα χρησιμοποιηθούν από τη σχεδίαση.

```
//Initialize quotient
uint64_t *quo = (uint64_t *) malloc(sizeof(uint64_t) * times);

//Start measuring time for simple execution @pc/host
auto start = high_resolution_clock::now();

//Call the design's/source top function
div(num, den, quo);

//Stop measuring time for simple execution @pc/host
auto stop = high_resolution_clock::now();
auto duration = duration_cast<microseconds>(stop - start);
cout << endl;
cout << duration.count() << endl;
cout << endl;

free(num);
free(den);
```

Εικόνα 5.12 Αλγόριθμος για το κάλεσμα της top function και χρονομέτρηση στον host.

```
//Read quotient to validate the results
uint64_t *quo_check = (uint64_t *) malloc(sizeof(uint64_t) * times);
//Read from the text file
ifstream f3;
f3.open("quotient.txt");
i = 0;
if(f3.is_open()){
  while (!f3.eof() )
  {
    f3>>quo_check[i];
    i++;
  }
}
f3.close();

int wrong = 0;
for (int i = 0; i < times; i++){
  if (quo_check[i] != quo[i]){
    std::cout<<"Wrong Quotient!"<<std::endl;
    break;
  }
}
if (!wrong)
  std::cout<<"Correct Quotient!"<<std::endl;

free(quo);
free(quo_check);

return 0;
}
```

Εικόνα 5.13 Αλγόριθμος για την επαλήθευση των αποτελεσμάτων της πράξης της διαίρεσης.

Εδώ θα αναλύσουμε τον αλγόριθμο της σχεδίασης-source. Αποτελείται από μια top function και άλλες τέσσερις συναρτήσεις που καλούνται από αυτήν. Η λογική πίσω από τη σχεδίαση είναι η εξής :

i) Έχουμε μια μεγάλη λούπα για τις 393485 φορές που πρέπει να τρέξει το πρόγραμμα.

ii) Αρχικοποιούμε τα απαραίτητα στοιχεία για να μπορέσουμε να εκτελέσουμε την πράξη και καλούμε τις συναρτήσεις ptr_bits και arr_bits, οι οποίες επιστρέφουν τον αριθμό των bits που έχουμε σε κάθε στοιχείο, με τη διαδικασία της ακολουθίας de Bruijn, ανάλογα με το αν δίνουμε pointer ή πίνακα.

iii) Κάνουμε κάποιες βασικές πράξεις για να μπορέσουμε να εφαρμόσουμε τον αλγόριθμο bit-wise division.

iv) Εφαρμόζουμε τον αλγόριθμο bit-wise division.

v) Διορθώνουμε τον αριθμητή, ο οποίος είναι και το υπόλοιπο της διαίρεσης και γράφουμε το αποτέλεσμα του πηλίκου στον αρχικοποιημένο pointer quo που κάναμε στο testbench.

```
void div(uint64_t num[times], uint64_t den[times], uint64_t quo[times])
{
    for (int i = 0; i < 393485; i++) {
        //Initialize the variables.
        //uint64_t *numerator = (uint64_t *) malloc(sizeof(uint64_t) * 2);
        uint64_t _numerator[2];
        uint64_t *numerator = &_numerator[0];
        numerator[0] = 0;
        numerator[1] = num[i];
        uint64_t denominator = den[i];
        // Clear quotient. Set it to zero.
        //uint64_t *quotient = (uint64_t *) malloc(sizeof(uint64_t) * 2);
        uint64_t _quotient[2];
        uint64_t *quotient = &_quotient[0];
        quotient[0] = 0;
        quotient[1] = 0;
        int numerator_bits = ptr_bits(numerator);
        int denominator_bits = arr_bits(denominator);
        // Create temporary space to store mutable copy of denominator.
        uint64_t shifted_denominator[uint64_count]{ denominator, 0 };
        // Create temporary space to store difference calculation.
        uint64_t difference[uint64_count]{ 0, 0 };
        // Shift denominator to bring MSB in alignment with MSB of numerator.
        int denominator_shift = numerator_bits - denominator_bits;
        //left_shift_uint128(shifted_denominator, denominator_shift, shifted_denominator);
        const std::size_t bits_per_uint64_sz = static_cast<std::size_t>(bits_per_uint64);
        const std::size_t shift_amount_sz = static_cast<std::size_t>(denominator_shift);
        // Early return
        if (shift_amount_sz & bits_per_uint64_sz)
        {
            shifted_denominator[1] = shifted_denominator[0];
            shifted_denominator[0] = 0;
        }
        // How many bits to shift in addition to word shift
        std::size_t bit_shift_amount = shift_amount_sz & (bits_per_uint64_sz - 1);
        // Do we have a word shift
        if (bit_shift_amount)
        {
            std::size_t neg_bit_shift_amount = bits_per_uint64_sz - bit_shift_amount;

            // Warning: if bit_shift_amount == 0 this is incorrect
            shifted_denominator[1] = (shifted_denominator[1] << bit_shift_amount) | (shifted_denominator[0] >> neg_bit_shift_amount);
            shifted_denominator[0] = shifted_denominator[0] << bit_shift_amount;
        }
        denominator_bits += denominator_shift;
        // Perform bit-wise division algorithm.
        int remaining_shifts = denominator_shift;
        while_loop(numerator, shifted_denominator, difference, quotient, numerator_bits, denominator_bits, remaining_shifts);
        quo[i] = cor_ret(numerator, quotient, numerator_bits, denominator_shift);
    }
}
```

Εικόνα 5.14 Αλγόριθμος του top function, της συνάρτησης div.

```
int ptr_bits(uint64_t *ptr){
    size_t c = uint64_count;
    ptr += c - 1;

    if (*ptr == 0 && c > 1){
        c--;
        ptr--;
    }

    int bit_count = 0;
    uint64_t V = *ptr;
    if (V == 0)
    {
        bit_count = static_cast<int>(c - 1) * bits_per_uint64;
    }
    else
    {
        unsigned long r = 0;
        unsigned long *result = &r;
        unsigned long *result_bits = result;
        V |= V >> 1;
        V |= V >> 2;
        V |= V >> 4;
        V |= V >> 8;
        V |= V >> 16;
        V |= V >> 32;

        *result_bits = deBruijnTable64[((V - (V >> 1)) * uint64_t(0x07EDD5E59A4E28C2)) >> 58];
        bit_count = static_cast<int>(c - 1) * bits_per_uint64 + static_cast<int>(r + 1);
    }
    return bit_count;
}
```

Εικόνα 5.15 Αλγόριθμος για τον υπολογισμό του αριθμού bits του κάθε στοιχείου του pointer. Χρησιμοποιείται για τον αριθμητή numerator.

```
int arr_bits(uint64_t arr){
    int bit_count = 0;
    if (arr != 0)
    {
        size_t c = uint64_count;

        unsigned long r = 0;
        unsigned long *result = &r;
        arr |= arr >> 1;
        arr |= arr >> 2;
        arr |= arr >> 4;
        arr |= arr >> 8;
        arr |= arr >> 16;
        arr |= arr >> 32;

        *result = deBruijnTable64[((arr - (arr >> 1)) * uint64_t(0x07EDD5E59A4E28C2)) >> 58];
        bit_count = static_cast<int>(r + 1);
    }
    return bit_count;
}
```

Εικόνα 5.16 Αλγόριθμος για τον υπολογισμό του αριθμού bits του κάθε στοιχείου του πίνακα. Χρησιμοποιείται για τον παρονομαστή denominator.

```
uint64_t cor_ret(uint64_t *numerator, uint64_t *quotient, int numerator_bits, int denominator_shift){
    // Correct numerator (which is also the remainder) for shifting of denominator, unless it is just zero.
    if (numerator_bits > 0)
    {
        //right_shift_uint128(numerator, denominator_shift, numerator);
        const std::size_t shift_amount_sz3 = static_cast<std::size_t>(denominator_shift);

        if (shift_amount_sz3 & bits_per_uint64_sz)
        {
            numerator[0] = numerator[1];
            numerator[1] = 0;
        }
        // How many bits to shift in addition to word shift
        std::size_t bit_shift_amount = shift_amount_sz3 & (bits_per_uint64_sz - 1);

        if (bit_shift_amount)
        {
            std::size_t neg_bit_shift_amount = bits_per_uint64_sz - bit_shift_amount;

            // Warning: if bit_shift_amount == 0 this is incorrect
            numerator[0] = (numerator[0] >> bit_shift_amount) | (numerator[1] << neg_bit_shift_amount);
            numerator[1] = numerator[1] >> bit_shift_amount;
        }

    }
    return quotient[0];
}
```

Εικόνα 5.17 Αλγόριθμος για τη διόρθωση του αριθμητή numerator μετά την εφαρμογή του bit-wise division algorithm.

Σημειώνεται, ότι τα παραπάνω κομμάτια κώδικα που αφορούν το source, είναι μετασχηματισμένα κομμάτια κώδικα που υπάρχουν στη βιβλιοθήκη Microsoft SEAL.

5.6) Περιγραφή των σχεδιάσεων

Οι σχεδιάσεις βασίζονται στην λογική που περιεγράφηκε παραπάνω, εκτός από την τέταρτη περίπτωση, στην οποία ακολουθείται μια διαφορετική προσέγγιση. Καθεμία περίπτωση, είναι πλήρως λειτουργική. Ο λόγος που έγιναν αυτές οι διαφορετικές προσεγγίσεις είναι για να εξεταστεί η απόδοση των διαφορετικών αυτών περιπτώσεων, για δεδομένη εκτέλεση στο FPGA. Ακολουθούν οι σχεδιάσεις και η ανάλυσή τους, καθώς και ποια είναι η λογική πίσω από κάθε περίπτωση.

i) Στην πρώτη περίπτωση, διαβάζονται τα ***numerator***, ***denominator***, ***quotient***, περνιούνται τα δύο πρώτα στο source και εκεί υπάρχει η for που τρέχει για 393485 φορές και η while μέσα στη for. Εδώ, τα directives που μπορούν να χρησιμοποιηθούν είναι μόνο ένα PIPELINE μέσα στη while, καθώς έχει μεταβλητό loop bound και δεν μπορεί να γίνει unroll. Έγιναν δοκιμές και χωρίς τη χρήση του break, που εμπεριέχει ο κώδικας, αλλάζοντάς το με ένα if, που αν η συνθήκη είναι αναληθής, δεν εκτελείται το κομμάτι του κώδικα που ακολουθεί το break. Αυτό γίνεται, καθώς το εργαλείο δεν μπορεί να κάνει schedule το break, για τις πολλές επαναλήψεις που εκτελείται ο υπολογισμός, μέσω της εξωτερικής for, η οποία διατηρείται. Χρησιμοποιείται και το directive DEPENDENCE για να καταλάβει το εργαλείο να κάνει RAW στον έλεγχο της while, που το ελεγχόμενο στοιχείο αλλάζει μέσα στη while. Παρ'όλα αυτά, ανάλογα με το Iteration Interval που χρησιμοποιείται στο PIPELINE, μπορεί το DEPENDENCE να μην είναι αναγκαίο. Αυτό μπορεί να συμβεί, καθώς αυξάνοντας το II, λέμε στο εργαλείο να ξεκινήσει την εκτέλεση της επόμενης επανάληψης, αφού έχει πάρει τον χρόνο να κάνει ένα κομμάτι του κρίσιμου μονοπατιού (critical path) και άρα δεν υπάρχει hazard για ικανό αριθμό II. Ακόμη, χρησιμοποιείται και το TRIPCOUNT, αλλά όπως έχει προαναφερθεί, δεν έχει κάποιο ουσιαστικό αντίκτυπο στην ταχύτητα, αλλά του δίνει την δυνατότητα να παρέχει μια καλύτερη εκτίμηση για το latency που θα υπάρξει. Τέλος, δοκιμάστηκε να χρησιμοποιηθούν και τα AXI4 και AXI4-lite directives. Αντιμετωπίστηκαν κάποια προβλήματα και σε αυτήν την περίπτωση, τα οποία θα αναλυθούν και στο επόμενο κεφάλαιο. Είναι σημαντικό να κρατήσουμε όμως, πως σε αυτήν την περίπτωση, όπως και στην επόμενη, δεν έγινε δυνατό να περάσουμε 3 στατικούς πίνακες με 393485 στοιχεία ο καθένας, πράγμα λογικό, λόγω και των περιορισμό σε μνήμη. Οπότε αυτό που έγινε είναι ότι στις θύρες που πρέπει να προσδιοριστούν οι μεταβλητές, όπως έχει ήδη συζητηθεί, μπήκαν οι scalar μεταβλητές (pointers) των πινάκων αυτών

ii) Στη δεύτερη περίπτωση, ακολουθείται παρόμοια λογική με την παραπάνω, αλλά χρησιμοποιείται for και if αντί για while και παίρνοντας χρήσιμες υποδείξεις από τη δημοσίευση "*HLS-Based Optimization and Design Space Exploration for Applications with Variable Loop Bounds*". Η while, σε ένα worst-case scenario, τρέχει ~50 φορές σε κάθε επανάληψη της εξωτερικής for, πράγμα που και μετρήθηκε. Τυπικά πάντως, η while θα τρέξει το πολύ 64 φορές, καθώς ο έλεγχος που γίνεται έχει σχέση με τα bits των

numerator και ***denominator***. Πέρα από την αλλαγή της while σε for και if, ισχύουν τα ίδια που έγιναν και παραπάνω.

iii) ***Buffer Chunk***. Μετασχηματίζουμε τον κώδικα, ούτως ώστε να χωριστούν τα 393485 στοιχεία σε 1024 τη φορά για 384 επαναλήψεις, πράγμα το οποίο ουσιαστικά εκτελεί την πράξη της διαίρεσης για τα πρώτα 393216 στοιχεία και η 385η επανάληψη τρέχει για τα τελευταία 269 στοιχεία που περισσεύουν. Αυτός ο μετασχηματισμός δοκιμάστηκε να γίνει και μέσα στο testbench, δηλαδή ο host να περνάει στον kernel (FPGA) τα 1024 στοιχεία τη φορά, αλλά και στο source, δηλαδή το FPGA να είναι υπεύθυνο για το πώς θα χωρίσει τα στοιχεία αυτά και τις πράξεις. Προφανώς και σε αυτήν την περίπτωση, όπως θα δούμε παρακάτω, είχαμε καλύτερα αποτελέσματα για το πρώτο σενάριο, αλλάζοντας τον κώδικα του testbench. Και στα δύο σενάρια, χρειάστηκαν αλλαγές και στους δύο κώδικες, απλά αναφερόμαστε με αυτόν τον τρόπο στις αλλαγές, για να δώσουμε ιδιαίτερο βάρος στον διαχωρισμό του πίνακα-pointer, που διαχωρίζει αυτήν την σχεδίαση με τις προηγούμενες. Χρησιμοποιήθηκαν και εδώ AXI4 και AXI4-lite directives. Το ίδιο το εργαλείο, πρότεινε την χρήση των 1024 στοιχείων τη φορά, για το Alveo™ U200 Data Center accelerator card, για να περνιούνται στις μέσω των θυρών (ports) μέσα στα resources του FPGA. Με όλη την παραπάνω λογική, μπορεί να αποφευχθεί ο χρόνος που χρειάζεται για να διαβαστούν και να γραφτούν τα στοιχεία πίσω στον host, για τη συνέχιση της εκτέλεσης του προγράμματος. Αυτό μπορεί να γίνει με το λεγόμενο burst Read-Write των μεταβλητών που μας ενδιαφέρουν και αξιοποιώντας το directive DATAFLOW, στο οποίο έχει γίνει αναφορά. Όπως θα δούμε όμως και παρακάτω, όλη αυτή η διαδικασία, δεν είναι το πιο κοστοβόρο κομμάτι του κώδικα και για αυτό δεν είχαμε κάποια μεγάλη βελτίωση στους χρόνους που πήραμε. Αυτό που θέλαμε παραπάνω να δοκιμάσουμε είναι να κάνουμε unroll τη μικρότερη λούπα των 1024 επαναλήψεων, καθώς όπως θα δούμε και παρακάτω στα προβλήματα που αντιμετωπίστηκαν, το εργαλείο δεν άφηνε να γίνει unroll η λούπα για την αρχική σχεδίαση των περιπτώσεων i) και ii), διότι οι 393485 φορές είναι πολλές για να μπορούν να γίνουν schedule και unroll. Δοκιμάστηκε επίσης και η χρήση του PIPELINE directive στην περίπτωση αυτή και τα αποτελέσματα σε κάθε περίπτωση θα παρουσιαστούν παρακάτω.

iv) Μεταβάλλουμε αρκετά τον κώδικά μας και ουσιαστικά αλλάζει ολόκληρη η σχεδίαση-υλοποίηση που είναι στο source αρχείο. Χωρίζουμε στην ουσία την υλοποίηση, με τη λογική, του να εκτελείται ολόκληρος ο κώδικας στο testbench, δηλαδή στον host και το FPGA εκτελεί μόνο την while, καθώς αποτελεί ένα αρκετά χρονοβόρο κομμάτι του κώδικα. Με αυτόν τον τρόπο, περνιούνται 393485 φορές, στοιχείο-στοιχείο οι πίνακες στο FPGA και εκτελείται η while στο kernel-FPGA. Δοκιμάστηκε και η παραλλαγή με την for και if, χωρίς όμως να υπάρχει μεγάλη διαφορά στα αποτελέσματα. Το πρόβλημα και σε αυτήν την περίπτωση είναι το μεταβλητό όριο επαναλήψεων της λούπας, πράγμα που δυσκολεύει αρκετά την επιτάχυνση της υλοποίησής μας.

v) **Η βέλτιστη.** Αυτή που έδωσε τα καλύτερα αποτελέσματα. Σε όλες τις παραπάνω υλοποιήσεις, είχαμε ένα πολύ σημαντικό πρόβλημα. Αυτό ήταν ότι είχαμε από τη μία ένα loop exit μέσα στη while που δεν μπορούσε να γίνει schedule, αλλά και ένα return statement, πάλι μέσα στη while, το οποίο σε συνδυασμό με το μεταβλητό όριο της while, δεν επέτρεπαν τη χρήση του PIPELINE directive, έξω από την λούπα αυτή. Πέρα από αυτά, η while_loop() χρειάστηκε να γίνει inline στην top function div(), δηλαδή το περιεχόμενο της συνάρτησης-λούπας να υπάρχει αυτούσιο μέσα στην συνάρτηση div() και να μετασχηματιστεί σε for με if, καθώς με άλλον τρόπο, το εργαλείο έφτανε σε ένα τέλμα κατά τη διάρκεια της διαδικασία C Synthesis, το οποίο θα συζητηθεί στο 6ο κεφάλαιο. Στην ουσία, λοιπόν, η υλοποίηση αυτή θα μπορούσαμε να πούμε πως αποτελεί μια παραλλαγή της δεύτερης περίπτωσης, βάζοντας τη μετασχηματισμένη while (for & if) μέσα στην div και βάζοντας if statements για κάθε loop ή function exit που υπάρχει. Αυτή η σχεδίαση μας έδωσε εντυπωσιακά καλύτερους αριθμούς από τις παραπάνω, της τάξης μονοψήφιων ms και τα αποτελέσματα θα αναλυθούν ενδελεχώς παρακάτω.

5.7) Δομή Αποτελεσμάτων με το Vivado HLS 2019.2

Σε αυτό το κεφάλαιο, εξηγείται η δομή των αποτελεσμάτων που παίρνουμε από το εργαλείο Vivado HLS. Με το εργαλείο, παίρνουμε δύο html αρχεία, μετά από την επιτυχή ολοκλήρωση της διαδικασίας του C Synthesis και του C/RTL Cosimulation. Στο πρώτο αρχείο, που προκύπτει από τη σύνθεση, αναγράφονται αναλυτικά σε πίνακες οι εκτιμήσεις της απόδοσης (Performance Estimates), οι εκτιμήσεις της αξιοποίησης των πόρων του FPGA (Utilization Estimates), καθώς και το Interface, δηλαδή το ποιες θύρες χρησιμοποιούνται από το εργαλείο, είτε αυτόματα, είτε από δικό μας προσδιορισμό αυτών. Στο δεύτερο αρχείο, που προκύπτει από το C/RTL Cosimulation, το εργαλείο μας δίνει έναν πίνακα, που εμπεριέχει το αποτέλεσμα του Latency σε κύκλους για τη σχεδίασή μας, καθώς και το αν αυτή είναι σωστή (PASS). Ο συνολικός χρόνος εκτέλεσης στο FPGA της σχεδίασης, εμφανίζεται στην κονσόλα του εργαλείου σε nanoseconds και έχει καταγραφεί χειρόγραφα. Ακολουθούν ενδεικτικά οι πίνακες στους οποίους αναφερόμαστε, χρησιμοποιώντας ως παράδειγμα την απλή αρχική σχεδίαση, χωρίς κανένα directive. Όπως έχει αναφερθεί, λόγω ύπαρξης μεταβλητών ορίων στη λούπα while, χωρίς τη χρήση του TRIPCOUNT, το εργαλείο δε θα μπορέσει να κάνει εκτίμηση για το Latency και θα δώσει ερωτηματικά στη θέση του. Δείχνουμε και έναν πίνακα, όπου έχουμε ερωτηματικά, με αποτελέσματα από άλλο project της λογικής της πρώτης σχεδίασης, για να φανεί η διαφορά με τη χρήση του TRIPCOUNT directive και των ερωτηματικών. Παρ' όλα αυτά, στα projects της λογικής της δεύτερης σχεδίασης, δεν έχουμε το ίδιο πρόβλημα, καθώς η while μετασχηματίζεται σε μια for με if λούπα, η οποία έχει σταθερό αριθμό επανάληψης, 50 φορές. Σε όλες τις υπόλοιπες προσομοιώσεις που έγιναν, χρησιμοποιήθηκε το directive αυτό και έχουμε κανονικά τα αποτελέσματα, αυτή όμως ήταν η πρώτη εκτέλεση και αυτό αποτελεί και ένα από τα προβλήματα που αντιμετωπίστηκαν.

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	7.154 ns	1.25 ns

Πίνακας 5.2 Πίνακας της επιλεγμένης περιόδου ρολογιού και της εκτιμώμενης περιόδου ρολογιού που μπορεί να εκτελέσει το FPGA τη σχεδίασή μας, για δεδομένη αβεβαιότητα.

Latency (cycles)		Latency (absolute)		Interval (cycles)		Type
min	max	min	max	min	max	
?	?	?	?	?	?	none

Latency (cycles)		Latency (absolute)		Interval (cycles)		Type
min	max	min	max	min	max	
6295761	44857297	78.697 ms	0.561 sec	6295761	44857297	none

Πίνακας 5.3 Πίνακας του εκτιμώμενου Latency, σε κύκλους και σε χρόνο και παράδειγμα χρήσης του TRIPCOUNT.

Instance	Module	Latency (cycles)		Latency (absolute)		Interval (cycles)		Type
		min	max	min	max	min	max	
grp_while_loop_fu_270	while_loop	?	?	?	?	?	?	none

Instance	Module	Latency (cycles)		Latency (absolute)		Interval (cycles)		Type
		min	max	min	max	min	max	
grp_while_loop_fu_270	while_loop	6	104	75.000 ns	1.300 us	6	104	none

Πίνακας 5.4 Πίνακας για τα Instances, δηλαδή πληροφορίες για το Latency, διάφορων υπο-modules, που το εργαλείο δημιουργεί αυτόματα για διάφορες υποσυναρτήσεις από την top function, που θεωρεί ότι είναι κοστοβόρες και παράδειγμα με TRIPCOUNT.

Loop Name	Latency (cycles)		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
- Loop 1	?	?	?	-	-	393485	no
+ memset_difference	1	1	1	-	-	2	no

Loop Name	Latency (cycles)		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
- Loop 1	6295760	44857295	16 ~ 114	-	-	393485	no
+ memset_difference	1	1	1	-	-	2	no

Πίνακας 5.5 Πίνακας εκτιμήσεων για το Latency της εκτέλεσης της λούπας και παράδειγμα με TRIPCOUNT.

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	18	0	1953	-
FIFO	-	-	-	-	-
Instance	0	9	1021	2791	-
Memory	0	-	12	6	-
Multiplexer	-	-	-	101	-
Register	-	-	1000	-	-
Total	0	27	2033	4851	0
Available	4320	6840	2364480	1182240	960
Available SLR	1440	2280	788160	394080	320
Utilization (%)	0	~0	~0	~0	0
Utilization SLR (%)	0	1	~0	1	0

Πίνακας 5.6 Πίνακας γενικών εκτιμήσεων χρήσης των πόρων του FPGA.

Instance	Module	BRAM_18K	DSP48E	FF	LUT	URAM
grp_while_loop_fu_270	while_loop	0	9	1021	2791	0
Total	1	0	9	1021	2791	0

Πίνακας 5.7 Πίνακας εκτιμήσεων χρήσης των πόρων για τα Instances.

Memory	Module	BRAM_18K	FF	LUT	URAM	Words	Bits	Banks	W*Bits*Banks
deBruijnTable64_U	div_deBruijnTable64	0	12	6	0	64	6	1	384
Total	1	0	12	6	0	64	6	1	384

Πίνακας 5.8 Πίνακας εκτιμήσεων χρήσης των πόρων για τη μνήμη. Εδώ μπορούμε να δούμε, πως το εργαλείο χωρίς κανένα directive, επιλέγει να βάλει σε look-up table τον global πίνακα που χρησιμοποιείται για την ακολουθία de Bruijn.

Name	LUT	Input Size	Bits	Total Bits
ap_NS_fsm	50	11	1	11
deBruijnTable64_address0	15	3	6	18
difference_0_1_reg_249	9	2	64	128
difference_1_1_reg_239	9	2	64	128
i_0_reg_228	9	2	19	38
phi_ln331_reg_259	9	2	1	2
Total	101	22	155	325

Πίνακας 5.9 Πίνακας εκτιμήσεων χρήσης των πολυπλεκτών του FPGA.

Variable Name	Operation	DSP48E	FF	LUT	Bitwidth P0	Bitwidth P1
mul_ln66_fu_780_p2	*	9	0	45	59	64
mul_ln94_fu_860_p2	*	9	0	45	59	64
bit_count_3_fu_906_p2	+	0	0	7	1	7
bit_count_fu_885_p2	+	0	0	8	7	8
i_fu_289_p2	+	0	0	19	19	1
denominator_shift_fu_957_p2	-	0	0	9	9	9
neg_bit_shift_amount_fu_1002_p2	-	0	0	8	8	7
sub_ln66_fu_774_p2	-	0	0	64	64	64
sub_ln94_fu_854_p2	-	0	0	64	64	64
icmp_ln304_fu_283_p2	icmp	0	0	20	19	18
icmp_ln324_fu_927_p2	icmp	0	0	11	8	8
icmp_ln346_fu_996_p2	icmp	0	0	11	6	1
icmp_ln42_fu_301_p2	icmp	0	0	29	64	1
icmp_ln80_fu_511_p2	icmp	0	0	29	64	1
lshr_ln351_fu_1018_p2	lshr	0	0	182	64	64
or_ln59_fu_321_p2	or	0	0	63	63	63
or_ln60_fu_357_p2	or	0	0	62	62	62
or_ln61_fu_395_p2	or	0	0	60	60	60
or_ln62_fu_437_p2	or	0	0	56	56	56
or_ln63_fu_481_p2	or	0	0	48	48	48
or_ln64_fu_731_p2	or	0	0	32	32	32
or_ln87_fu_531_p2	or	0	0	63	63	63
or_ln88_fu_567_p2	or	0	0	62	62	62
or_ln89_fu_605_p2	or	0	0	60	60	60
or_ln90_fu_647_p2	or	0	0	56	56	56
or_ln91_fu_691_p2	or	0	0	48	48	48
or_ln92_fu_811_p2	or	0	0	32	32	32
shifted_denominator_5_fu_1024_p2	or	0	0	64	64	64
bit_count_4_fu_891_p3	select	0	0	8	1	1
denominator_bits_fu_912_p3	select	0	0	7	1	1
select_ln331_1_fu_948_p3	select	0	0	63	1	64
select_ln331_fu_939_p3	select	0	0	63	1	1
select_ln338_1_fu_981_p3	select	0	0	63	1	1
select_ln338_fu_974_p3	select	0	0	63	1	64
select_ln346_fu_1037_p3	select	0	0	63	1	64
shifted_denominator_6_fu_1030_p2	shl	0	0	182	64	64
shl_ln351_fu_1012_p2	shl	0	0	182	64	64
xor_ln331_fu_933_p2	xor	0	0	2	1	2
Total	38	18	0	1953	1357	1413

Πίνακας 5.10 Πίνακας χρήσης πόρων για διάφορα expressions (αριθμητικών και λογικών πράξεων), που ταξινομεί το εργαλείο και εμπεριέχονται στη σχεδίασή μας.

Name	FF	LUT	Bits	Const Bits
ap_CS_fsm	10	0	10	0
bit_count_4_reg_1208	8	0	8	0
denominator_shift_reg_1243	9	0	9	0
difference_0_0_reg_216	64	0	64	0
difference_0_1_reg_249	64	0	64	0
difference_0_reg_1263	64	0	64	0
difference_1_0_reg_204	64	0	64	0
difference_1_1_reg_239	64	0	64	0
difference_1_reg_1268	64	0	64	0
grp_while_loop_fu_270_ap_start_reg	1	0	1	0
i_0_reg_228	19	0	19	0
i_reg_1063	19	0	19	0
icmp_ln304_reg_1059	1	0	1	0
icmp_ln42_reg_1094	1	0	1	0
icmp_ln80_reg_1146	1	0	1	0
p_numerator_1_reg_1083	64	0	64	0
phi_ln331_reg_259	1	0	1	0
select_ln331_1_reg_1237	64	0	64	0
select_ln331_reg_1231	64	0	64	0
select_ln346_reg_1253	64	0	64	0
shifted_denominator_6_reg_1248	64	0	64	0
shifted_denominator_s_reg_1088	64	0	64	0
tmp_11_reg_1113	2	0	2	0
tmp_12_reg_1099	1	0	1	0
tmp_13_reg_1106	1	0	1	0
tmp_15_reg_1120	4	0	4	0
tmp_17_reg_1127	8	0	8	0
tmp_18_reg_1139	16	0	16	0
tmp_20_reg_1151	1	0	1	0
tmp_22_reg_1158	1	0	1	0
tmp_24_reg_1165	2	0	2	0
tmp_26_reg_1172	4	0	4	0
tmp_28_reg_1179	8	0	8	0
tmp_29_reg_1191	16	0	16	0
trunc_ln64_reg_1134	32	0	32	0
trunc_ln92_reg_1186	32	0	32	0
zext_ln313_reg_1068	19	0	64	45
zext_ln42_reg_1213	8	0	9	1
zext_ln80_reg_1218	7	0	9	2
Total	1000	0	1048	48

Πίνακας 5.11 Πίνακας χρήσης των πόρων των registers του FPGA.

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	div	return value
ap_rst	in	1	ap_ctrl_hs	div	return value
ap_start	in	1	ap_ctrl_hs	div	return value
ap_done	out	1	ap_ctrl_hs	div	return value
ap_idle	out	1	ap_ctrl_hs	div	return value
ap_ready	out	1	ap_ctrl_hs	div	return value
num_address0	out	19	ap_memory	num	array
num_ce0	out	1	ap_memory	num	array
num_q0	in	64	ap_memory	num	array
den_address0	out	19	ap_memory	den	array
den_ce0	out	1	ap_memory	den	array
den_q0	in	64	ap_memory	den	array
quo_address0	out	19	ap_memory	quo	array
quo_ce0	out	1	ap_memory	quo	array
quo_we0	out	1	ap_memory	quo	array
quo_d0	out	64	ap_memory	quo	array

Πίνακας 5.12 Πίνακας χρήσης των θυρών του FPGA.

		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	42487025	42487025	42487025	NA	NA	NA

Πίνακας 5.13 Πίνακας αποτελέσματος μετά το C/RTL Cosimulation. Εδώ μπορούμε να δούμε ότι έγινε η επιλογή της Verilog για HDL γλώσσα, καθώς και ότι η σχεδίασή μας είναι σωστή (PASS). Παρατηρούμε επίσης, ότι η μη χρήση του TRIPCOUNT επηρεάζει τη σύνθεση και όχι το C/RTL Cosimulation, στην περίπτωση που έχουμε μεταβλητή λούπα.

```
$finish called at time : 2755994500 ps
```

*Εικόνα 5.18 Ένα από τα καλύτερα αποτελέσματα που πήραμε, **2.756ms**. Αυτός είναι ο τρόπος που εμφανίζονται τα αποτελέσματα στην κονσόλα του εργαλείου, κατά το C/RTL Cosimulation.*

Λόγω της μεγάλης διαθεσιμότητας πόρων που προσφέρουν τα FPGA, το πιο ενδιαφέρον στοιχείο με το οποίο θα ασχοληθούμε είναι ο χρόνος εκτέλεσης στο FPGA και το speedup που είχαμε σε σχέση με την απλή εκτέλεση στον υπολογιστή. Παρ' όλα αυτά θα σχολιάσουμε και τις διαφορές στη χρήση των πόρων σε κάθε περίπτωση.

5.8) Αποτελέσματα και συμπεράσματα επί αυτών

Όπως προαναφέρθηκε, το πιο κοστοβόρο κομμάτι των σχεδιάσεων, αλλά και που στεκόταν εμπόδιο στην επιτάχυνση της σχεδίασης, ήταν το κομμάτι του κώδικα που εμπεριέχει τη λούπα while. Όταν γίνεται build η βιβλιοθήκη Microsoft SEAL, χρησιμοποιείται το -O3 flag και δίνει τη δυνατότητα να γίνει η καλύτερη δυνατή βελτιστοποίηση που μπορεί να λάβει χώρα για τον δεδομένο κώδικα. Το -O3 flag, αφορά τον κώδικα που είναι γραμμένος σε C++ και όχι αυτόν που προκύπτει από τη διαδικασία της C Synthesis και που χρησιμοποιείται για να γίνει το C/RTL Cosimulation. Μετά από μεγάλη αναζήτηση στη βιβλιογραφία και στις οδηγίες χρήσης της Xilinx, δεν βρέθηκε κάποιος τρόπος που να περνιέται αυτούσιο το flag αυτό, για τις παραπάνω διαδικασίες. Το optimization που μπορεί να κάνει ο compiler στις διαδικασίες αυτές, είναι το vivado_optimization_level, το οποίο καθορίζεται και μέσα στο περιβάλλον του εργαλείου Vivado HLS 2019.2, αλλά δεν καταλήγει από μόνο του σε καλύτερα αποτελέσματα από το -O3 flag. Παρ' όλα αυτά, για την εξαγωγή ορθών συμπερασμάτων σε σχέση με τα αποτελέσματα, απαιτείται η σύγκριση με το χρόνο της απλής εκτέλεσης που προκύπτει με τη χρήση του flag αυτού, της συνάρτησης divide, που εκτελείται για 393485 φορές, κατά την χρήση του παραδείγματος του CKKS scheme. Ο χρόνος αυτός είναι τα ***164ms***. Παρ'όλα αυτά, θεωρούμε σημαντικό να αποτελέσει κομμάτι της συζήτησης και ο χρόνος εκτέλεσης της συνάρτησης χωρίς το flag, που είναι τα ***560ms***. Συνεπώς, ο χρόνος που πρέπει να μειώσουμε, είναι ο χρόνος ***164ms***, αλλά θα γίνει και σύγκριση με την τιμή ***560ms***, με τη λογική ότι δεν έχουμε compiler optimization στις διαδικασίες του High Level Synthesis. Επίσης, αναφέρουμε και τον χρόνο που χρειάστηκε και το εργαλείο για να κάνει τη διαδικασία C Synthesis και C/RTL Cosimulation, καθώς σε κάποιες περιπτώσεις αυτός ήταν απαγορευτικά μεγάλος και τον συμβολίζουμε με $\Delta t_{synth+cosim}$, ενώ ο χρόνος εκτέλεσης συμβολίζεται με Δt_{FPGA}. Τέλος, σημειώνουμε πως θα αναλύσουμε τα αποτελέσματα που είναι σωστά και έχουν νόημα. Δε θα παρουσιαστούν αποτελέσματα που είτε ήταν λάθος, είτε δε συμβάλλουν στην άντληση ορθών συμπερασμάτων.

Συνεχίζουμε με κάποια συνοπτικά σχόλια για κάθε σχεδίαση και εν συνεχεία παραθέτουμε τα αποτελέσματα.

i) Στην πρώτη περίπτωση, έχουμε καλύτερα αποτελέσματα από την αρχική σχεδίαση των 560 ms. Μεταβάλλοντας το clock και χρησιμοποιώντας τα κατάλληλα directives (PIPELINE) για να ξεκινάνε πιο γρήγορα τα operations, πήραμε καλό αποτέλεσμα. Στην περίπτωση αυτή, δεν μπορούμε να χρησιμοποιήσουμε κάποιο άλλο directive που να βελτιώσει την απόδοση, καθώς έχουμε μεταβλητό αριθμό επαναλήψεων στη λούπα while και ούτε μπορούμε να τοποθετήσουμε το PIPELINE directive εκτός της while, επειδή τότε θα έπρεπε αυτή να γίνει unroll, πράγμα που δε γίνεται λόγω της μεταβλητότητας αυτής. Σε διάφορες δοκιμές, είχαμε και warning για ένα hazard που συνέβαινε μέσα σε κάθε επανάληψη, το οποίο για να αντιμετωπιστεί, επιλέχθηκε η αύξηση της περιόδου ή/και η αύξηση του Initiation Interval. Σημειώνουμε, πως η απλή βελτιστοποίηση της while_loop, δεν σημειώνει τα επιθυμητά αποτελέσματα κάτω από τα 164ms, καθώς ο υπόλοιπος κώδικας δε βελτιστοποιείται με κάποιον τρόπο,

ούτε με compiler optimization flag, ούτε με κάποιο διαφορετικό directive. Τα αποτελέσματα σε αυτήν την περίπτωση, είναι τα εξής :

Clock Period (ns)	Optimizations	$\Delta t_{synth+cosim}$	Δt_{FPGA} (ms)	Speedup_O0 (560ms)	Speedup_O3 (164ms)
25	PIPELINE II=1	~1 ώρα	**442**	**1.267**	**0.371**
19	PIPELINE II=1	~1 ώρα	**336**	**1.667**	**0.488**

Πίνακας 5.14 Αποτελέσματα για την πρώτη σχεδίαση.

Παρατηρούμε, ότι τα αποτελέσματα δεν είναι καλύτερα από τον χρόνο του O3 flag, αλλά υπάρχει βελτίωση στη σύγκριση με την απλή εκτέλεση της συνάρτησης χωρίς flag. Πάρθηκαν κι άλλα αποτελέσματα, με χειρότερους χρόνους από τους παραπάνω και για αυτό δεν παρουσιάζονται.

ii) Εδώ παίρνουμε καλύτερα αποτελέσματα από τα παραπάνω, καθώς η while μετασχηματίζεται σε for και if. Ακόμα όμως δεν μπορούμε να εφαρμόσουμε το PIPELINE directive έξω από αυτήν τη λούπα, καθώς λόγω των loop exits προκύπτουν προβλήματα με το scheduling που πρέπει να κάνει το εργαλείο. Τα αποτελέσματα είναι τα εξής :

Clock Period (ns)	Optimizations	$\Delta t_{synth+cosim}$	Δt_{FPGA} (ms)	Speedup_O0 (560ms)	Speedup_O3 (164ms)
20	PIPELINE II=1	~1/2 ώρα	**354**	**1.582**	**0.463**
14	PIPELINE II=2	~1/2 ώρα	**418**	**1.334**	**0.392**
12.5	PIPELINE II=2	~1/2 ώρα	**383**	**1.462**	**0.428**
7	PIPELINE II=3	~1/2 ώρα	**305**	**1.836**	**0.538**
6	PIPELINE II=3	~1/2 ώρα	**262**	**2.137**	**0.626**
6	PIPELINE II=4	~1/2 ώρα	**339**	**1.652**	**0.484**

Πίνακας 5.15 Αποτελέσματα για τη δεύτερη σχεδίαση.

Πρέπει να σημειώσουμε δύο πράγματα. Πρώτον, ότι η αύξηση του Initiation Interval, βοηθάει στη μείωση της περιόδου του ρολογιού, που σημαίνει αύξηση της συχνότητας λειτουργίας του FPGA. Η κάρτα που έχει επιλεγεί, επιτρέπει την εκτέλεση μέχρι και 300MHz, δηλαδή μέχρι και 3.33ns περίοδο ρολογιού. Δεύτερον, πως και σε αυτήν τη περίπτωση δεν μπορεί να εφαρμοστεί κάποιο άλλο directive, πέρα από το PIPELINE στην while_loop, που είναι υλοποιημένη με for και if. Τα αποτελέσματα και εδώ, δεν καταφέρνουν να μειώσουν τον χρόνο των 164 ms και το 262ms, αποτελεί την καλύτερη εκτέλεση μέχρι να φτάσουμε στην 5η περίπτωση, όπου και η διαφορά είναι μεγάλη.

iii) Εδώ γίνεται χρήση των ports της κάρτας, χωρίς να αλλάζει κάτι σε σχέση με το PIPELINE, το οποίο και συνεχίζουμε να χρησιμοποιούμε. Η σχεδίαση παίρνει πάρα πολλές ώρες για κάποιες δοκιμές για να γίνει το C Synthesis και το Cosimulation. Όπως όμως θα δούμε, τα αποτελέσματα είναι χειρότερα από την προηγούμενη περίπτωση. Αυτό συμβαίνει, καθώς δεν τα

directives αυτά, που έχουν ήδη συζητηθεί, δεν μπορούν να επιφέρουν καλύτερα αποτελέσματα από μόνα τους, χωρίς τη χρήση DATAFLOW και χωρίς να έχουμε κάποιον μεγάλο στατικό πίνακα, ο οποίος να μας δημιουργεί καθυστέρηση (Latency), κατά τη μεταφορά του από τον Host-CPU στον Kernel-FPGA. Επειδή έχουμε, λοιπόν pointers και από τη στιγμή που δεν μπορεί να χρησιμοποιηθεί το DATAFLOW, επειδή έχουμε πολλές λογικές συγκρίσεις, όπως για παράδειγμα if, ο προσδιορισμός των ports, χειροτερεύει τα αποτελέσματα.

Clock Period (ns)	Optimizations	$\Delta t_{synth+cosim}$	Δt_{FPGA} (ms)	Speedup_O0 (560ms)	Speedup_O3 (164ms)
20	PIPELINE II=1	~4 ώρες	**346**	**1.619**	**0.474**
14	PIPELINE II=2	~22 ώρες	**529**	**1.059**	**0.31**

Πίνακας 5.16 Αποτελέσματα για την τρίτη σχεδίαση.

Παρατηρούμε εδώ, πως η επικοινωνία μεταξύ του FPGA και του Host-CPU που γίνεται για να παίρνει κάθε φορά τα 1024 στοιχεία, δηλαδή για 384+1 φορές, εισάγει καθυστέρηση στην εκτέλεση, που δεν φέρνει καλά αποτελέσματα. Αυτό θα γίνει ακόμα πιο εμφανές στην επόμενη (τέταρτη περίπτωση), που πήραμε χειρότερα αποτελέσματα ακόμα και από το αρχικό 560 ms.

iv) Στην περίπτωση αυτή, ο host πρέπει να επικοινωνήσει με το kernel-FPGA, 393485 φορές, για τη μεταφορά ενός στοιχείου αριθμητή, παρονομαστή και πηλίκου όπου θα γραφτεί το αποτέλεσμα, κάθε φορά. Όπως θα δούμε, αυτό εισάγει απαγορευτική καθυστέρηση στην εκτέλεση στο FPGA.

Clock Period (ns)	Optimizations	$\Delta t_{synth+cosim}$	Δt_{FPGA} (ms)	Speedup_O0 (560ms)	Speedup_O3 (164ms)
11	PIPELINE II=8	~2 μέρες	**1146**	**0.489**	**0.143**
6	PIPELINE II=9	~1 μέρα	**703**	**0.797**	**0.233**
5.8	PIPELINE II=10	~1 μέρα	**739**	**0.758**	**0.222**
5	PIPELINE II=10	~1 μέρα	**653**	**0.858**	**0.251**
6	PIPELINE II=13	~2 μέρες	**846**	**0.662**	**0.194**

Πίνακας 5.17 Αποτελέσματα για τη τέταρτη σχεδίαση.

Βλέπουμε πως στην καλύτερη περίπτωση, η εκτέλεση είναι 4 φορές χειρότερη από τον επιθυμητό χρόνο (0.251). Καταλαβαίνουμε, λοιπόν, πως αυτή η σχεδίαση δεν είχε κανένα όφελος, από άποψη επιτάχυνσης της σχεδίασης.

v) Εδώ θα δούμε το πιο σημαντικό κομμάτι που αφορά τη διπλωματική, σε σχέση με το τεχνικό της κομμάτι. Με τις τροποποιήσεις που γίνονται σε αυτήν την περίπτωση, μετασχηματίζοντας κάθε loop ή function exit σε if,

μεταβάλλοντας την while_loop σε if και for, και περνώντας τον pointer με τα στοιχεία στο FPGA, τρέχοντας εκεί τις 393485 επαναλήψεις που απαιτούνται, καθίσταται δυνατό, να περάσουμε το PIPELINE directive εξωτερικά της while_loop, ακριβώς στην αρχή του loop body της εξωτερικής λούπας. Με αυτόν τον τρόπο, επιτυγχάνεται κάτι που δεν μπορούσε να γίνει με τις προηγούμενες σχεδιάσεις, που είναι ιδιαίτερα σημαντικό για τη βελτιστοποίηση της επιτάχυνσης. Αυτό είναι το γεγονός ότι γίνεται unroll κάθε μια επανάληψη από τις 393485 και έτσι τρέχουν σε έναν κύκλο ρολογιού, διαδοχικά, για Initiation Interval = 1. Αντίστοιχα, μεταβάλλοντας την τιμή αυτή, μεταβάλλεται και η έναρξη της εκτέλεσης κάθε επανάληψης, δηλαδή για παράδειγμα, για II=2, θα έχουμε εκτέλεση μιας επανάληψης ανά 2 κύκλους ρολογιού και ούτω καθεξής.

Σε αυτήν λοιπόν την περίπτωση, πάρθηκαν αρκετά αποτελέσματα, για την αποτελεσματικότερη μελέτη και άντληση συμπερασμάτων της βέλτιστης επιτάχυνσης που μπορεί να επιτευχθεί. Επιλέχθηκε να χρησιμοποιηθούν οι τιμές για το Initiation Interval μέχρι τον αριθμό 3 και μεταβάλλοντας το ρολόι να ερμηνευτούν τα διάφορα αποτελέσματα που προκύπτουν. Θα χωρίσουμε τα αποτελέσματα σε 3 πίνακες, έναν για κάθε Initiation Interval, από 1 μέχρι 3 και εδώ έχει νόημα να σχολιαστούν και οι πίνακες σύνοψης που προκύπτουν για τα διάφορα κυκλωματικά στοιχεία κατά τη διαδικασία C Synthesis. Πιο συγκεκριμένα, τα Flip-Flops που χρησιμοποιούνται σε κάθε περίπτωση αλλάζουν και φτάνουν στα όρια των διαθέσιμων πόρων του συγκεκριμένου FPGA. Είναι λογικό, πως όσο πιο γρήγορα απαιτούμε να ξεκινάει η επόμενη εκτέλεση επανάληψης και όσο χαμηλότερα είναι η περίοδος ρολογιού, άρα και τόσο μεγαλύτερη είναι η συχνότητα λειτουργίας, τόσο μεγαλύτερες απαιτήσεις θα έχουμε σε πόρους.

Είχαμε δύο περιπτώσεις που έχει σημασία να παρουσιαστούν, καθώς μπορεί για τη συγκεκριμένη κάρτα να μην επαρκούν οι διαθέσιμοι πόροι για κάποιες μετρήσεις, αλλά θα επαρκούν για κάποια άλλη κάρτα με περισσότερους πόρους. Η πρώτη περίπτωση, αφορούσε την αξιοποίηση περισσότερου του 100% του Super Logic Region (SLR) σε Flip-Flops, πράγμα το οποίο ***δεν*** κάνει απαγορευτικές τις μετρήσεις μας για τη συγκεκριμένη κάρτα, από τη στιγμή που η αξιοποίηση του συνολικού αριθμού διαθέσιμων Flip-Flop είναι μικρότερη του 100%. Όπως έχει ήδη προαναφερθεί, η Alveo™ U200 Data Center accelerator card έχει 3 SLR. Όταν το εργαλείο καταλήγει στο συμπέρασμα ότι ξεπερνιέται το 100% των FFs ενός διαθέσιμου SLR, αλλά όχι το συνολικό, σημαίνει ότι γίνεται προσπάθεια να χρησιμοποιηθούν όλα τα στοιχεία από ένα μόνο SLR, πράγμα που σε κάποιες περιπτώσεις δεν είναι εφικτό. Αυτό γίνεται προφανές, με το γεγονός ότι το συνολικό ποσοστό σε αυτήν την περίπτωση είναι 3 φορές μικρότερο, από το διαφαινόμενο ποσοστό του SLR, δηλαδή διαμοιράζεται στα 3 διαθέσιμα SLR της κάρτας. Τα αποτελέσματα αυτής της περίπτωσης, θα έχουν από δίπλα μονό αστερίσκο. Η δεύτερη περίπτωση, αφορά την υπέρβαση του 100% των πόρων και των συνολικών διαθέσιμων πόρων και του SLR, όπου και σε αυτήν την περίπτωση γίνεται

ο διαχωρισμός σε 3, απλά δεν είναι αρκετός για τη συγκεκριμένη κάρτα. Παρουσιάζονται όμως και αυτά τα αποτελέσματα, με τη λογική που περιγράφεται παραπάνω, ότι δηλαδή μπορεί να χρησιμοποιηθεί μια κάρτα με περισσότερους διαθέσιμους πόρους σε FFs, για την επίτευξη αυτών ή και καλύτερων αποτελεσμάτων. Στα αποτελέσματα αυτής της περίπτωσης θα αναγράφεται διπλός αστερίσκος. Μόνο ένα αποτέλεσμα ανήκει σε αυτήν την περίπτωση, για Initiation Interval = 1 και περίοδο ρολογιού 3.4 ns, που πλησιάζει τα όρια λειτουργίας της κάρτας, των 300MHz. Τα αποτελέσματα που δεν έχουν κανένα αστερίσκο, δεν παρουσίασαν καμία υπέρβαση στη χρήση των διαθέσιμων πόρων. Σημειώνεται επίσης, πως έγινε στρογγυλοποίηση στα 2 δεκαδικά στοιχεία. Κατηγοριοποιούμε, λοιπόν τα στοιχεία των πινάκων που ακολουθούν σε δύο ακόμα κατηγορίες :

** Flip-Flop Super Logic Region (SLR) > 100% (ή αλλιώς FF SLR >100%)*
*** FF SLR > 100% && FF Total >100%.*

Συνεχίζουμε, με την παρουσίαση των αποτελεσμάτων :

Clock Period (ns)	Optimizations	$\Delta t_{synth+cosim}$	Δt_{FPGA} (ms)	Speedup_O0 (560ms)	Speedup_O3 (164ms)
3.4	PIPELINE II=1	~1 ώρα	**1.34****	417.91	122.39
4.5	PIPELINE II=1	~1 ώρα	**1.77***	316.38	92.66
5	PIPELINE II=1	~1 ώρα	**1.97***	284.26	83.25
5.5	PIPELINE II=1	~1 ώρα	**2.17***	258.06	75.58
7	PIPELINE II=1	~1 ώρα	**2.76***	202.9	59.42
10	PIPELINE II=1	~1 ώρα	**3.94***	142.13	41.62
12	PIPELINE II=1	~1 ώρα	**4.72**	118.64	34.75
15	PIPELINE II=1	~1 ώρα	**5.9**	94.92	27.8
20	PIPELINE II=1	~1 ώρα	**7.87**	71.16	20.84

Πίνακας 5.18 Αποτελέσματα για την πέμπτη και βέλτιστη σχεδίαση, για Initiation Interval = 1.

Για την καλύτερη κατανόηση των αποτελεσμάτων, θα εξηγήσουμε το πώς προκύπτει το αποτέλεσμα 2.76ms, για περίοδο ρολογιού 7ns. Ο χρόνος εκτέλεσης στο FPGA, θα προκύψει από την εκτέλεση των 393485 επαναλήψεων, όπου η κάθε επανάληψη γίνεται σε έναν κύκλο ρολογιού, δηλαδή σε 7ns. Αυτό σημαίνει πως μπορούμε να υπολογίσουμε 393485επαναλήψεις * 7ns/επανάληψη = 2754395ns = 2.75ms, χρόνος εκτέλεσης. Το εργαλείο μας δίνει 2.76ms, καθώς γίνονται κάποια operations από το Vivado, για να εκτελεστεί η σχεδίασή μας. Σαν δεύτερο σχόλιο, βλέπουμε ότι σε κάθε δοκιμή, έχουμε αποδοτικότερη εκτέλεσης από τις προηγούμενες σχεδιάσεις, αλλά και πως μειώνουμε αρκετά τον χρόνο εκτέλεσης, ακόμα και από τον χρόνο εκτέλεσης της συνάρτησης divide της Microsoft SEAL, με το optimization flag -O3. Αυτό διατηρείται σε όλα τα αποτελέσματα αυτής της σχεδίασης. Ακόμη, σημειώνεται πως οι περίοδοι που επιλέχθηκαν δεν είναι τυχαίοι, αλλά είναι αυτοί που βρίσκονται σε

οριακές καταστάσεις σχετικά με το θέμα αξιοποίησης των πόρων της κάρτας. Για τα παραπάνω αποτελέσματα, αυτό εκφράζεται στο γεγονός ότι στη μέτρηση 1.77ms, με περίοδο 4.5ns, έχουμε αλλαγή από το προηγούμενο αποτέλεσμα και αξιοποιείται πάνω από το 90% των συνολικών διαθέσιμων FFs και πιο συγκεκριμένα το 99%, σε αντίθεση με το μοναδικό αποτέλεσμα με διπλό αστερίσκο, που έχουμε αξιοποίηση 169%. Αντίστοιχα, το αποτέλεσμα των 4.72ms, που συμβαίνει για περίοδο ρολογιού 12ns, έχουμε αξιοποίηση 91% των διαθέσιμων FFs σε ένα SLR, σε αντίθεση με το προηγούμενο, που για περίοδο 10ns, έχουμε 132% χρήση σε ένα SLR. Τέλος, σημειώνεται πως οι χρόνοι εκτέλεσης που παρουσιάστηκαν πιο πάνω, θα είναι καλύτεροι από τα υπόλοιπα αποτελέσματα, σε κάθε περίπτωση που αφορά το SLR και την αξιοποίηση πόρων, πράγμα λογικό αφού έχει επιλεγεί II Ακολουθούν τα αποτελέσματα για II=2 :

Clock Period (ns)	Optimizations	$\Delta t_{synth+cosim}$	Δt_{FPGA} (ms)	Speedup_O0 (560ms)	Speedup_O3 (164ms)
3.4	PIPELINE II=2	~1 ώρα	**2.68***	**208.96**	**61.19**
4.5	PIPELINE II=2	~1 ώρα	**3.54***	**158.19**	**46.33**
5	PIPELINE II=2	~1 ώρα	**3.94***	**142.13**	**41.62**
5.5	PIPELINE II=2	~1 ώρα	**4.33***	**129.33**	**37.88**
7	PIPELINE II=2	~1 ώρα	**5.51**	**101.63**	**29.76**
10	PIPELINE II=2	~1 ώρα	**7.87**	**71.16**	**20.84**
12	PIPELINE II=2	~1 ώρα	**9.44**	**59.32**	**17.37**
15	PIPELINE II=2	~1 ώρα	**11.8**	**47.46**	**13.9**
20	PIPELINE II=2	~1 ώρα	**15.74**	**35.58**	**10.42**

Πίνακας 5.19 Αποτελέσματα για την πέμπτη και βέλτιστη σχεδίαση, για Initiation Interval = 2.

Εδώ, ισχύουν τα ίδια με τα παραπάνω, για τον υπολογισμό των χρόνων εκτέλεσης στο FPGA. Έχει αξία να παρατηρήσουμε, πως αυξάνοντας το Initiation Interval, αυξάνεται ο χρόνος εκτέλεσης αντίστοιχα με τις προηγούμενες τιμές. Δηλαδή για περίοδο ρολογιού 3.4ns, έχουμε 2.68ms, ενώ παραπάνω έχουμε το μισό χρόνο εκτέλεσης, τα 1.34ms, ενώ όπως προαναφέρθηκε, μειώνεται η αξιοποίηση των διαθέσιμων πόρων της κάρτας. Ακολουθούν τα αποτελέσματα για II=3 :

Clock Period (ns)	Optimizations	$\Delta t_{synth+cosim}$	Δt_{FPGA} (ms)	Speedup_O0 (560ms)	Speedup_O3 (164ms)
3.4	PIPELINE II=3	~1 ώρα	**4.02***	**139.3**	**40.8**
4.5	PIPELINE II=3	~1 ώρα	**5.31***	**105.46**	**30.9**
5	PIPELINE II=3	~1 ώρα	**5.9**	**94.92**	**27.8**
5.5	PIPELINE II=3	~1 ώρα	**6.49**	**86.3**	**25.27**
7	PIPELINE II=3	~1 ώρα	**8.26**	**67.8**	**19.85**
10	PIPELINE II=3	~1 ώρα	**11.8**	**47.46**	**13.9**
12	PIPELINE II=3	~1 ώρα	**14.17**	**39.52**	**11.57**
15	PIPELINE II=3	~1 ώρα	**17.71**	**31.62**	**9.26**
20	PIPELINE II=3	~1 ώρα	**23.61**	**23.72**	**6.95**

Πίνακας 5.20 Αποτελέσματα για την πέμπτη και βέλτιστη σχεδίαση, για Initiation Interval = 3.

Εξακολουθούν, να ισχύουν τα ίδια με τα παραπάνω, για τον υπολογισμό των χρόνων εκτέλεσης στο FPGA. Επίσης παρατηρούμε τον τριπλασιασμό τον αρχικών χρόνων για II=1, αφού εδώ έχουμε II=3.

Σε αυτό το σημείο είναι σημαντικό να επισημάνουμε μια σημαντική παρατήρηση. Ο προσεγγιστικός υπολογισμός χρόνων που εξηγήθηκε παραπάνω, που αφορά την πέμπτη και βέλτιστη περίπτωση σχεδιάσεων, δεν μπορεί να εφαρμοστεί στις προηγούμενες σχεδιάσεις. Ενώ ο χρόνος που έδινε το εργαλείο είναι συγκεκριμένος και για αυτές, για όσες προσομοιώσεις και να δοκιμάσαμε με τα ίδια χαρακτηριστικά, προσεγγιστικά δεν μπορούμε να καταλήξουμε κάπου, καθώς έχουμε κομμάτια κώδικα, που δεν μπορούν υπολογιστούν με αυτόν τον τρόπο. Αναφερόμαστε στα loop exits αλλά και στο γενικότερο latency που υπάρχει με τις προηγούμενες σχεδιάσεις.

Από τα παραπάνω αποτελέσματα, σημειώνεται πως τα σημαντικότερα, είναι αυτά της σύγκρισης με την απλή εκτέλεση της συνάρτησης χρησιμοποιώντας το flag -O3. Είναι γνωστό, πως τα περισσότερα προγράμματα και κώδικες σήμερα χρησιμοποιούν αυτό το compiler optimization flag και η διαδικασία επιτάχυνσης με FPGA, θα πρέπει να μπορεί να υπερνικά και αυτούς τους χρόνους, καθώς έχουν αυτήν τη δυνατότητα. Συμπερασματικά, λοιπόν, επιτεύχθη βελτίωση του χρόνου εκτέλεσης, από ***6.95***, έως και ***92.66*** φορές πιο γρήγορα, σε σχέση με την απλή εκτέλεση της συνάρτησης αυτής, 393485 φορές, με το optimization flag -O3, σε έναν υπολογιστή. Ενώ πάρθηκε και το αποτέλεσμα του ***122.39*** speedup, δεν το θεωρούμε αξιοποιήσιμο, στα πλαίσια αυτής της διπλωματικής, αλλά και για τους λόγους που συζητήθηκαν παραπάνω.

Μια ακόμη σημείωση, αποτελεί το ότι οι υπόλοιπες σχεδιάσεις πλην της τελευταίας-βέλτιστης, είχαν νόημα να παρατεθούν, καθώς συμβάλλουν στην καλύτερη άντληση συμπερασμάτων. Σκιαγραφούν επίσης καλύτερα, τη συνολική πορεία που ακολουθήθηκε για τη διεκπεραίωση αυτής της διπλωματικής εργασίας και πιο συγκεκριμένα των σταδίων από τα οποία μπορεί να περάσει η προσπάθεια επιτάχυνσης υλικού/λογισμικού. Πέρα από αυτό, δεν προσφέρουν την επιθυμητή βελτίωση της απόδοσης, πράγμα που αποτελεί σκοπό της εργασίας αυτής.

Κεφάλαιο 6 : Προβλήματα που αντιμετωπίστηκαν

6.1) Εισαγωγικά

Στα πλαίσια αυτής της διπλωματικής, προέκυψαν πάρα πολλά προβλήματα και δοκιμάστηκαν αρκετά πράγματα για να αντιμετωπιστούν. Πέρα από τη φύση του ίδιου του θέματος, για το οποίο υπάρχουν ακόμα αρκετά ερευνητικά βήματα που μπορούν να γίνουν, υπήρξε δυσκολία στην ανάπτυξη και υλοποίηση της σχεδίασης, η οποία θα έπρεπε να έχει τα αναγκαία χαρακτηριστικά για να αποτελεί σχεδίαση και όχι απλό κώδικα, αλλά και συνολικότερα στην κατανόηση του Microsoft SEAL σαν κώδικα και η χαρτογράφησή του. Τα πρώτα βήματα που ακολουθήθηκαν για την εκπόνηση της διπλωματικής αυτής, ήταν η μελέτη βιβλιογραφίας σχετική με Homomorphic Encryption έπειτα σε θέματα που αφορούσαν την επιτάχυνσή του με FPGA. Εν συνεχεία, άρχισε η ενασχόληση με το Microsoft SEAL, της κατανόησης της δομής του, προγραμματιστικά, αλλά και η μελέτη του CKKS scheme. Τέλος, έγινε η προσπάθεια να περαστεί αλλαγμένο κομμάτι του κώδικα της βιβλιοθήκης στο Vivado HLS 2019.2, με σκοπό να γίνει η επιτάχυνση.

Όπως έχει προαναφερθεί, σκοπός της διπλωματικής αυτής δεν ήταν να δημιουργήσουμε από το μηδέν δικό μας κώδικα, αλλά να στηριχτούμε σε αυτόν της βιβλιοθήκης και με αλλαγές να καταφέρουμε να τον επιταχύνουμε με FPGAs. Η υλοποίηση από το μηδέν, ενός Fully Homomorphic Encryption scheme, δεν είναι κάτι απλό και απαιτεί αρκετό χρόνο. Για την εργασία, λοιπόν, δοκιμάστηκαν αρκετά πράγματα, τα οποία και θα αναλυθούν σε αυτό το κεφάλαιο.

6.2) Προσπάθειες που δεν κατέληξαν σε αποτέλεσμα

Θα αναλύσουμε εδώ κάποια πράγματα με τα οποία υπήρξε ενασχόληση, αλλά δεν προχώρησαν για διάφορους λόγους. Η πρώτη προσπάθεια που έγινε, αφορούσε τη συνάρτηση encrypt_zero_symmetric();. Όπως έχει ειπωθεί, η συνάρτηση αυτή είναι υπεύθυνη για την κρυπτογράφηση του δημοσίου κλειδιού, αλλά γίνεται κλήση της και σε άλλα σημεία κατά τη διάρκεια εκτέλεσης του παραδείγματος που αφορά το CKKS scheme. Το αρχικό σκεπτικό ήταν να γραφτεί κάποιος κώδικας ο οποίος να μπορεί να πάρει χρήσιμα στοιχεία τα οποία χρησιμοποιεί η συνάρτηση αυτή, αλλά και το scheme γενικά, ούτως ώστε να αξιοποιηθούν για την υλοποίηση του CKKS scheme. Τέτοια στοιχεία αποτέλεσαν τα encrypted_result, input_vector και διάφορα άλλα στοιχεία ασφαλείας που αφορούσαν τα μηνύματα, τα κρυπτογραφημένα κείμενα όπως και έχουν αναλυθεί σε προηγούμενα κεφάλαια. Για το σκοπό αυτό, γράφτηκε κώδικας σε C και μεταβλήθηκαν διάφορα σημεία του κώδικα του Microsoft SEAL, σε σημεία που παράγονται ή τυπώνονται οι παράμετροι αυτοί, για να παίρνουμε όλα αυτά σε αρχεία. Τελικά, χρησιμοποιήθηκε μερικώς, στο κομμάτι της δημιουργίας των αρχείων με τα 393485 στοιχεία του αριθμητή (numerator), παρονομαστή (denominator) και του αποτελέσματος-πηλίκου (quotient), που αφορούν τη συνάρτηση divide_uint128_uint64_inplace_generic();.

Σε αυτό το σημείο, δεν υπήρχε εύκολος τρόπος να χρησιμοποιηθεί η συνάρτηση encrypt_zero_symmetric();, ακόμα και αρκετά αλλαγμένη. Λόγω της μεγάλης αλληλεξάρτησης που είχε με άλλες συναρτήσεις που βρίσκονταν σε διαφορετικά σημεία της βιβλιοθήκης ή/και σε διαφορετικά αρχεία, δεν προχώρησε η συγκεκριμένη συνάρτηση. Αυτό που έγινε είναι ένα πολύ μικρό κομμάτι μια εσωτερικής της συνάρτησης, να περαστεί στο Vivado HLS, το οποίο όμως στην ουσία επειδή ήταν δύο

μικρές for, ούτε ήταν κοστοβόρες για το CKKS scheme, αλλά και ούτε μας έδωσαν κάποια βελτίωση στον χρόνο. Παρ'όλα αυτά, η ενασχόληση με τη συνάρτηση encrypt_zero_symmetric();, έδειξαν πάρα πολλά προβλήματα που προέκυψαν, τα οποία αφορούσαν καθολικά την αξιοποίηση της βιβλιοθήκης Microsoft SEAL σε συνδυασμό με το Vivado HLS.

Εδώ θα αναφερθούμε σύντομα στη βιβλιοθήκη HEAAN. Η βιβλιοθήκη HEAAN έχει υλοποιηθεί από τους δημιουργούς του CKKS scheme, αν και δεν ανανεώνεται τόσο συχνά όσο η Microsoft SEAL. Υλοποιεί και αυτή το CKKS scheme και για αυτό και δοκιμάστηκε, αφού είχαμε βρεθεί σε ένα αδιέξοδο σε σχέση με τη Microsoft SEAL. Η βιβλιοθήκη αυτή είναι πιο απλή προγραμματιστικά και για αυτό και δοκιμάστηκε. Το σημείο στο οποίο φάνηκε το αδιέξοδο, ήταν η προσπάθεια του να περαστεί ο κώδικας στο Vivado HLS 2019.2 και να γίνει σύνθεση. Η βιβλιοθήκη HEAAN, χρησιμοποιεί κάποιες άλλες μικρότερες βιβλιοθήκες για να καταφέρει να υλοποιεί τα schemes που προσφέρει. Μία από αυτές τις βιβλιοθήκες ήταν η NTL (Number Theory Library), η οποία χρησιμοποιούσε threading, πράγμα που οδήγησε το Vivado να μας δίνει το εξής:

error : thread-local storage is not supported for the current target

Αναζητώντας το σφάλμα, φάνηκε ότι η NTL χρησιμοποιεί την μέθοδο Thread Local Storage (TLS), κατά την οποία χρησιμοποιείται static ή global μνήμη σε ένα thread. Δεν μπορεί να χρησιμοποιηθεί η μέθοδος αυτή στα FPGA για προφανείς λόγους, καθώς δε συνάδει με τη λογική που αναπτύχθηκε στο προηγούμενο κεφάλαιο. Για παράδειγμα, ένας κώδικας που μπορεί να υλοποιήσει TLS σε C/C++ είναι ο εξής:

```
#include <threads.h>
thread_local int foo = 0;
```

Εικόνα 6.1 Παράδειγμα για χρήση TLS.

Για να λυθεί αυτό το πρόβλημα, δοκιμάστηκε να χρησιμοποιηθεί σε κάποια κομμάτια της βιβλιοθήκης NTL μια γραμμή κώδικα η οποία αποτελεί ένα preprocessor macro, το οποίο δίνει εντολή στον compiler να μη χρησιμοποιήσει TLS. Το ίδιο ακριβώς error, το πήραμε και με το Microsoft SEAL και δοκιμάστηκε να αντιμετωπιστεί με τη συμπλήρωση της γραμμής αυτής στο defines.h, χωρίς απόλυτη επιτυχία. Παρ' όλα αυτά δεν αποτελεί κάτι το αναγκαίο στο SEAL, όπως θα δούμε και στη συνέχεια. Ο κώδικας είναι αυτός :

```
#define _M_CEE 001
```

Εικόνα 6.2 Κώδικας για μη χρήση TLS από τον compiler.

Τέλος δοκιμάστηκε η χρήση του SEALEmbedded, μιας παραλλαγής της βιβλιοθήκης Microsoft SEAL, η οποία είναι πιο περιεκτική και πιο απλή, αλλά δεν έχει την ίδια απόδοση. Σκοπός είναι η χρήση σε ενσωματωμένα συστήματα, όπως λέει και το όνομα. Δεν καταλήξαμε ούτε με αυτό κάπου, καθώς δεν είχε μεγάλες διαφορές σε σχέση με τα προβλήματα που προέκυψαν με την Microsoft SEAL. Παρακάτω αναλύονται τα προβλήματα που προέκυψαν στην προσπάθεια να περαστεί το Microsoft SEAL στο Vivado HLS 2019.2.

6.3) Προβλήματα με το Microsoft SEAL και το Vivado HLS 2019.2

Κατά την προσπάθεια χρήσης του Microsoft SEAL στο Vivado HLS 2019.2, προέκυψαν πολλά προβλήματα, ακόμα και στην απλή χρήση του παραδείγματος του CKKS scheme ως testbench και της υλοποίησης κάποιας συνάρτησης του Microsoft SEAL ως σχεδίαση-source. Φυσικά, βρέθηκε τρόπος να αντιμετωπιστούν όλα τα προβλήματα αυτά ή να χρησιμοποιηθεί μια εναλλακτική ως επίλυσή τους. Η Microsoft, προτείνει τη χρήση του CMAKE, για το compile της βιβλιοθήκης, καθώς υπάρχουν διάφορα flags και επιλογές, που μπορούν με εύκολο τρόπο να γίνουν από εκεί. Αυτό όμως δεν είναι εφικτό όσον αφορά το Vivado HLS. Μια αρκετά πρωτόλεια προσπάθεια που έγινε, αφορούσε τη χρήση της βιβλιοθήκης ως testbench και τη χρήση κομματιού της και ως source. Αυτό φυσικά είναι κάτι που δεν προχώρησε, καθώς η σχεδίαση, που θα πρέπει να συντεθεί σε HDL γλώσσα, θα πρέπει να είναι όσο το δυνατόν απομονωμένη από λοιπές βιβλιοθήκες και εξωτερικούς κώδικες. Για διάφορους λόγους, δεν μπορεί να συντεθεί η βιβλιοθήκη αυτή καθ' αυτή σε HDL γλώσσα, κυρίως λόγω της μεγάλης της πολυπλοκότητας, εκτός αν κάποιος τη γράψει από την αρχή. Συνεπώς δημιουργήθηκαν πολλά προβλήματα, τα οποία και παρουσιάζουμε παρακάτω. Η τελική υλοποίηση, όπως έχει ήδη συζητηθεί, δε χρησιμοποιεί καθόλου τη βιβλιοθήκη, καθώς αυτό δεν ήταν εφικτό. Επειδή όμως το κομμάτι που επιταχύνεται, αλλά και η συνολική ενασχόληση με αυτήν ήταν μεγάλη, αναφερόμαστε σε αυτήν και στα προβλήματα που προέκυψαν κατά την προσπάθεια χρήσης της μαζί με το Vivado HLS 2019.2, που όπως θα δούμε, το καλύτερο που έγινε, είναι η χρήση της μέχρι και τη διαδικασία του C Synthesis.

Το πρώτο και μεγαλύτερο πρόβλημα που προέκυψε, αφορά το πώς θα ξαναγίνει compile η βιβλιοθήκη ή κομμάτι της, μέσα από το Vivado HLS, για να μπορέσει να χρησιμοποιηθεί. Όπως προαναφέρθηκε, η χρήση του CMAKE δεν είναι εφικτή, για αυτό και ένας άλλος τρόπος που προτείνει η Microsoft για να χρησιμοποιηθεί ο g++ ως compiler, είναι η χρήση του linker flag ***-lseal***, δηλαδή να γίνει link η ήδη χτισμένη βιβλιοθήκη μαζί με τον κώδικα του testbench και του source. Ήδη γίνεται αντιληπτό, πώς αυτό δεν μπορεί να γίνει στο source, καθώς άλλο η HDL γλώσσα που προκύπτει και άλλο η βιβλιοθήκη SEAL, που πρέπει να χτιστεί. Συνεπώς και η συνέχεια των προβλημάτων αφορούν το testbench και ένα source κώδικα που δε χρησιμοποιεί καθόλου τη βιβλιοθήκη, αλλά είναι κομμάτι της ή κάποια συνάρτησή της. Παρ' όλα αυτά τα προβλήματα αφορούν τη χρήση της βιβλιοθήκης και στα δύο αρχεία. Η χρήση του flag αυτού, δεν είναι αρκετή για το Vivado HLS, καθώς χρειάζεται παρ' όλα αυτά να γνωρίζει που βρίσκονται τα διάφορα .h αρχεία, που γίνονται include στην αρχή του testbench. Αυτό λύνεται, με τη χρήση του flag ***-I/PATH/TO/SEAL/native/src***, στο πεδίο των CFLAGS του testbench, καθώς και όποιων third-party βιβλιοθηκών που χρησιμοποιεί η βιβλιοθήκη και δεν έχουμε απενεργοποιήσει κατά το compile της, όπως είναι το gsl, δηλαδή ***-I/PATH/TO/SEAL/thirdparty/msgsl-src/include***. Προφανώς, μπορούμε να μη χρησιμοποιήσουμε όποια third party βιβλιοθήκη δε χρειαζόμαστε και να μη μπει το σχετικό include flag.

Το παραπάνω, λύνει το πρόβλημα του που βρίσκονται τα header files, αλλά δημιουργεί άλλα προβλήματα που συζητιούνται παρακάτω. Σημειώνεται επίσης, πως για την επίλυση του προαναφερθέντος, δοκιμάστηκε και η αλλαγή όλων των αρχείων της

βιβλιοθήκης, ούτως ώστε να υπάρχει αλληλοσυσχέτιση στα paths που αναφέρονται σε κάθε αρχείο ως relative paths και τη χρήση του #pragma once, που έχει ήδη αναφερθεί. Αυτό όμως δεν είχε ιδιαίτερη επιτυχία, καθώς οδηγεί το εργαλείο σε ατέρμονο compile.

Με τα παραπάνω, παρατηρήθηκε αστάθεια στο Vivado HLS. Το εργαλείο αυτό, βασίζει το IDE του στο Eclipse. Λόγω της πολυπλοκότητας των αρχείων του include flag, προέκυψε αρκετές φορές το JAVA HEAP SPACE ERROR, το οποίο δημιουργούσε αρχεία 5.7 Gb, όσο δηλαδή και άντεχε το HEAP, το οποίο εν τέλει έκλεινε το Vivado HLS. Δοκιμάστηκε να γίνει edit ένα configuration αρχείο που χρησιμοποιεί το εργαλείο, το eclipse.ini, το οποίο μεγάλωσε μεν το HEAP, αλλά απλά μεγάλωσαν και αρχεία που έγιναν dump. Τελικά, το πρόβλημα φαίνεται πως προέκυπτε, επειδή το εργαλείο μετασχημάτιζε τα paths που δινόντουσαν σε relative paths και άρα γινόταν αναζήτηση των header files χωρίς επιτυχία από το eclipse. Οπότε και η λύση του προβλήματος αυτού, ήταν η διαγραφή του relative path που έμπαινε αυτόματα στα properties του κάθε αρχείου, μέσα στο εργαλείο ως include path.

Ένα άλλο πρόβλημα που προέκυψε αφορά την χρήση C++17. Το Microsoft SEAL, χρησιμοποιεί ως προεπιλογή διάφορες συναρτήσεις αυτής της γενιάς του C++, πράγμα που δε συμβαίνει στο Vivado HLS 2019.2. Ο πρώτος τρόπος που δοκιμάστηκε ήταν η χρήση νεότερων γενιών του εργαλείου, όπως είναι το Vitis HLS 2020.2, 2021.1, 2021.2, τα οποία ναι μεν δε δημιουργούν το πρόβλημα αυτό, αλλά δε χρησιμοποιήθηκαν για άλλους λόγους. Οι λόγοι αφορούν τη διαφορετική δομή των αποτελεσμάτων που δίνουν τα εργαλεία αυτά, η οποία δε θα έδιναν τα ίδια περιθώρια ερμηνείας, αλλά και η αυτοματοποιημένη προσπάθεια από τις νεότερες εκδόσεις, να χρησιμοποιήσουν directives για την επιτάχυνση της σχεδίασης, πράγμα που δεν έδινε τον απόλυτο έλεγχο σε μας σχετικά με αυτήν και τέλος, η υποστήριξη που υπάρχει σε σχέση με την επιλεγμένη κάρτα είναι πιο περιορισμένη. Θα αναφερθούν κάποια προβλήματα που αντιμετωπίστηκαν και σε αυτά τα εργαλεία, καθώς έγινε προσπάθεια χρήσης τους. Όσον αφορά τη χρήση με το Vivado HLS 2019.2, αυτό που εν τέλει αξιοποιήθηκε ως λύση στο πρόβλημα, είναι το compile της βιβλιοθήκης Microsoft SEAL χωρίς της χρήση του C++17, αλλά με τη χρήση του C++14. Αυτό έγινε χρησιμοποιώντας το flag ***-DSEAL_USE_CXX17=OFF***, στο CMAKE με το οποίο γίνεται build η βιβλιοθήκη.

Ακόμα και έτσι, το Vivado HLS 2019.2, υποστηρίζει πειραματικά τη C++14, οπότε προκύπτει ένα δεύτερο πρόβλημα, στο εργαλείο αυτήν τη φορά, που σταματάει τη διαδικασία της Synthesis και πως ακόμα και το C++14 είναι σε πειραματικό στάδιο για χρήση στο εργαλείο. Για τη λύση αυτού του προβλήματος, η Xilinx προτείνει τη χρήση του flag ***-std=c++ox***, κατά τη διάρκεια του C Synthesis στο source αρχείο ως CFLAG και την αφαίρεσή του για την εξαγωγή αποτελεσμάτων πριν την εκτέλεση του C/RTL Cosimulation.

Το επόμενο πρόβλημα που προκύπτει, έχει σχέση με αυτό που αναφέρθηκε και προηγουμένως με το Thread Local Storage, για χρήση της βιβλιοθήκης στο source κώδικα. Το Microsoft SEAL, χρησιμοποιεί την τεχνική αυτή για τις διάφορες παραμέτρους ασφαλείας που υπάρχουν στο MemoryPoolHandle, χωρίς όμως να χρησιμοποιεί παραλληλία κατά την εκτέλεση κάποιου προγράμματος που βασίζεται σε αυτήν. Για να λυθεί το πρόβλημα αυτό, αρχικά δοκιμάστηκε να γίνει μεταβολή του

αρχείου defines.h, με το define που περιγράφεται παραπάνω. Εν συνεχεία όμως, προέκυψε νέο πρόβλημα σχετικά με τον compiler που χρησιμοποιεί το Vivado HLS. Για το λόγο αυτό, δοκιμάστηκε η μεταβολή του αρχείου c++ config που υπάρχει στο path /PATH/TO/Xilinx/Vitis_HLS/2021.1/tps/lnx64/gcc-6.2.0/include/c++/6.2.0/x86_64_pc_linux_gnu/bits, που αντίστοιχο path υπάρχει και για την έκδοση που τελικά χρησιμοποιήσαμε, αυτήν του Vivado HLS 2019.2. Η μεταβολή ήταν η εξής :

```
#undef _GLIBCXX_HAVE_TLS
```

Εικόνα 6.3 Κώδικας για μη χρήση TLS από τον compiler του Vivado HLS.

Η αλλαγή αυτή, έγινε εν γνώσει, ότι μπορεί να μην είναι σωστό το αποτέλεσμα, καθώς αλλάζουμε προεπιλογές του εργαλείου, οι οποίες αφορούν το πώς ο compiler θα χτίσει τον κώδικα για να τρέξει αυτός σε ένα FPGA. Ούτε αυτός ο τρόπος επίλυσε το πρόβλημα. Εν τέλει, η λύση ήταν αρκετά πιο απλή και αφορούσε το compile της βιβλιοθήκης με τη χρήση του flag ***-DSEAL_USE_INTRIN=OFF***, το οποίο και χρησιμοποιούσε τη μέθοδο TLS.

Όλα τα παραπάνω, δημιουργούσαν μεγάλα αδιέξοδα στην προσπάθεια χρήσης του SEAL με το Vivado HLS. Αυτός ήταν ο λόγος, που αρχικά δοκιμάστηκαν και άλλες βιβλιοθήκες και που μετασχηματίστηκε ο τελικός κώδικας στο να προσομοιώνει όσο το δυνατόν γίνεται την εκτέλεση του παραδείγματος του CKKS scheme της Microsoft SEAL, χωρίς ωστόσο την ουσιαστική, άμεση χρήση της. Αναζητώντας στη βιβλιογραφία, βρέθηκε επίσης το RIFFA, ένα project που έχει εκλείψει τα τελευταία χρόνια, που είναι στην ουσία drivers για FPGA, το οποίο για κάποια άλλη κάρτα ίσως υπάρχει η προοπτική να δώσει αποτελέσματα, όχι όμως σε επίπεδο simulation και για αυτό δε χρησιμοποιήθηκε περαιτέρω.

6.4) Προβλήματα σχετικά με τις σχεδιάσεις

Πιάνοντας το νήμα από τα παραπάνω, η λογική που ακολουθήθηκε είναι να χρησιμοποιείται αρχικά η βιβλιοθήκη στο testbench, αλλά όχι στο source. Αυτό έγινε για την προσπάθεια υλοποίησης κομματιού της συνάρτησης encrypt_zero_symmetric, πράγμα που δεν απέφερε σημαντικά αποτελέσματα. Η λογική ήταν, να «ανοιχτεί» το παράδειγμα του CKKS scheme στο testbench και να καλείται η συνάρτηση encrypt_zero_symmetric, που θα βρισκόταν στο source. Αυτό είχε πολύ μεγάλη δυσκολία, λόγω της συνολικότερης φύσης του Microsoft SEAL, όπως αυτή έχει προαναφερθεί. Λόγω της μεγάλης αλληλοσυσχέτισης των header files, αλλά και των διαφόρων συναρτήσεων, δεν είναι εφικτή η υλοποίηση της παραπάνω λογικής. Απλό παράδειγμα αποτελεί η προσπάθεια για αυτήν τη συνάρτηση, που απαιτούσε το άνοιγμα τουλάχιστον 20 συναρτήσεων, αλλά και κλάσεων (!), πράγμα που αν γινόταν, δε θα επέφερε τη σωστή λειτουργία του κώδικα. Με αυτό, εννοούμε πως η μη δημιουργία κάποιου object που είναι αναγκαίο για την υλοποίηση κάποιου scheme, αλλά απλά το άνοιγμά του, θα οδηγούσε σε error τη λογική συνέχεια του προγράμματος. Αυτό που καταφέραμε σχετικά με τα παραπάνω, λοιπόν, είναι μόνο 30 γραμμές κώδικα να υπάρχουν στο source και το testbench να είναι ιδιαίτερα ανοιγμένο μέχρι και στα σημεία που θα γινόταν η κλήση της συνάρτησης. Σημαντική σημείωση αποτελεί επίσης, πως όλο το παραπάνω, δεν μπορούσε να φτάσει στο

στάδιο του C/RTL Cosimulation, καθώς το εργαλείο δεν μπορούσε να κάνει καθόλου link, ούτε μόνο για τον κώδικα του testbench, τη βιβλιοθήκη. Είναι δηλαδή σαν το linker flag που αναφέρθηκε να μη γινόταν ποτέ στη διαδικασία αυτή και άρα παίρναμε error σχετικά με την απουσία των includes των header files στην αρχή του κώδικα.

Προχωρώντας στη λογική που τελικά απέφερε σημαντικά αποτελέσματα, δηλαδή τη σχεδίαση της divide_uint128_uint64_inplace_generic, υπήρξαν προβλήματα που είχαν να κάνουν με τη σχεδίαση αυτή. Σημειώνουμε ότι δεν θα ήταν εφικτό να βρεθεί το που καλείται 393485 φορές η συνάρτηση αυτή, ούτως ώστε να έχουμε ένα testbench που να χρησιμοποιεί τη βιβλιοθήκη, καθώς αυτό δε γίνεται σε ένα μόνο σημείο αλλά σε διάφορα σημεία, ακόμα και κατά την κατασκευή κάποιου object. Μην ξεχνάμε επίσης, ότι και να καταφέρναμε κάτι τέτοιο, δε θα μπορούσε να γίνει link έστω στο testbench η βιβλιοθήκη. Έχοντας, λοιπόν, την εμπειρία των προβλημάτων αυτών και του αποτελέσματος της ανάλυσης του Vtune, προχώρησε η λογική της μη χρήσης του SEAL στο Vivado HLS 2019.2 και γενικότερα σε νεότερες εκδόσεις που δοκιμάστηκαν, που είχαμε σχεδόν τα ίδια προβλήματα με τα παραπάνω. Ακόμα και με αυτόν τον τρόπο, προέκυψαν κάποια προβλήματα που αφορούν περισσότερο τη σχεδίαση και θα αναφερθούν.

Στις αρχικές προσπάθειες που έγιναν για την επιτάχυνση της συνάρτησης που υλοποιεί τη διαίρεση, στην προσπάθεια της πρώτης περίπτωσης της σχεδίασης, προέκυψαν διάφορα προβλήματα. Ένα από αυτά, αφορούν τη χρήση στατικών πινάκων, πράγμα το οποίο οδηγούσε σε segmentation dump. Ήταν κάτι το αναμενόμενο, καθώς η χρήση τουλάχιστον 4 πινάκων από 393485 στοιχεία uint64_t, έχει μεγάλες απαιτήσεις σε μνήμη που δεν ικανοποιήθηκαν. Η λύση σε αυτό ήταν η χρήση pointers και δυναμικής κατανομής της μνήμης, που σε αυτήν την περίπτωση, κάνει και την υλοποίηση ορθότερη.

Ένα άλλο πρόβλημα που αντιμετωπίστηκε, αφορά τη διαδικασία της σύνθεσης, όπου και δημιουργούταν ένα αρχείο 318 Gb, το οποίο εν τέλει δημιουργούσε segmentation dump. Αυτό οφείλεται σε τρία πράγματα. Πρώτον, ότι το εργαλείο προσπαθούσε να κάνει schedule κάποια loop exits, δεύτερον ότι η while_loop έχει μεταβλητό εύρος αριθμού επαναλήψεων και τρίτον ότι χρησιμοποιούνταν το PIPELINE directive εξωτερικά της while, και πιο συγκεκριμένα μέσα στην μεγάλη εξωτερική for loop, που σημαίνει ότι προσπαθούσε να κάνει όλο το loop body, συμπεριλαμβανομένου της while_loop, unroll. Αυτό δημιουργούσε μια ατέρμονη προσπάθεια του εργαλείου, να δημιουργήσει κώδικα για κάθε επανάληψη από τις 393485, πράγμα που οδηγούσε σε αδιέξοδο.

Πέρα από τα παραπάνω, ήδη συζητήθηκαν στο προηγούμενο κεφάλαιο διάφορα προβλήματα, όπως για παράδειγμα την αναγκαιότητα χρήσης του DEPENDENCE directive, λόγω ύπαρξης hazard. Στην ουσία, όμως, αυτό λύθηκε και χωρίς τη χρήση του, χρησιμοποιώντας temporary μεταβλητές και δοκιμάζοντας την αύξηση της περιόδου του ρολογιού ή/και την αύξηση του Initiation Interval στο PIPELINE directive.

Κλείνοντας, ένα τελευταίο πρόβλημα που αντιμετωπίστηκε, ήταν η κακή χρήση του PIPELINE ή του UNROLL directive, που όπως είδαμε και στα αποτελέσματα, η διαδικασία της σύνθεσης ή και του C/RTL Cosimulation διήρκησε ακόμα και μέρες,

πράγμα που συνέβαινε πάλι λόγω της ύπαρξης των loop exit, που έπρεπε το εργαλείο να ασχοληθεί με την κάθε επανάληψη ξεχωριστά. Σε κάθε περίπτωση, η διπλωματική αυτή οδήγησε στην άντληση κάποιων χρήσιμων συμπερασμάτων και σκέψεων, τα οποία παρουσιάζονται στο επόμενο κεφάλαιο.

Κεφάλαιο 7 : Συμπερασματικά

7) Συμπερασματικά

Το Homomorphic Encryption αποτελεί ένα σημαντικό κομμάτι της Κυβερνοασφάλειας και είναι έχει ένα αρκετά υποσχόμενο μέλλον. Η χρήση που μπορεί να έχει στο Cloud, αλλά και σε άλλες εφαρμογές, όπως στην ανάλυση δεδομένων ασθενών με σεβασμό στα προσωπικά δεδομένα και γενικότερα σε συνδυασμό με Machine Learning, μπορούν να δημιουργήσουν σημαντικές αλλαγές στους τομείς αυτούς και σε διάφορους άλλους, μεταβάλλοντας τελείως την έννοια της ασφάλειας, όπως αυτή υπάρχει σήμερα. Βρισκόμαστε ακόμα στην αρχή της ανάπτυξης αυτής της τεχνολογίας, με την ανάπτυξη πολλών schemes, όπως και βελτιώσεις σε υπάρχοντα να είναι πράγματα που διαφαίνονται για το μέλλον που έρχεται.

Το CKKS scheme, αποτελεί κομμάτι της 4ης γενιάς του Fully Homomorphic Encryption (FHE). Αποτελεί ένα αρκετά περίπλοκο scheme στην κατανόηση, που όμως λόγω της ασφάλειας που μπορεί να προσφέρει, αλλά και των δυνατοτήτων που διαθέτει σε σχέση με προηγούμενες γενιές όπως αυτό της προσέγγισης δεκαδικών αριθμών, ξεχωρίζει, από τα διάφορα schemes που υπάρχουν μέχρι σήμερα. Φυσικά, πάντα θα υπάρχει περιθώριο βελτίωσης στον χώρο της Κυβερνοασφάλειας και στην αντιμετώπιση των νέων προκλήσεων που θα δημιουργούνται διαρκώς, στην εποχή της διαρκούς ανάπτυξης και διάδοσης της πληροφορίας.

Η βιβλιοθήκη Microsoft SEAL, αποτελεί την πλέον δημοφιλή στον χώρο του Homomorphic Encryption, με έμφαση στην ευκολία χρήσης των διαφόρων schemes, αλλά και στην ταχύτητα εκτέλεσης τους, που προσφέρει. Η βιβλιοθήκη αυτή, δίνει τη δυνατότητα ακόμα και σε κάποιον που δεν έχει ιδιαίτερες γνώσεις κρυπτογραφίας, να εφαρμόσει κάποιες από τις τεχνικές που διαθέτει, με σχετικά εύκολο τρόπο. Παρ' όλα αυτά, όπως είδαμε και στα πλαίσια αυτής της διπλωματικής εργασίας, λόγω της μεγάλης αλληλεξάρτησης των συναρτήσεων που χρησιμοποιούνται, αλλά και συνολικότερα της δομής της, δε διευκολύνει την προσπάθεια επιτάχυνσής της, με χρήση των FPGAs.

Η επιτάχυνση υλικού/λογισμικού αποτελεί έναν πολλά υποσχόμενο και διαδεδομένο τρόπο, με τον οποίο πολλές εφαρμογές μπορούν να βελτιωθούν από την άποψη της ταχύτητας εκτέλεσης, μεταξύ άλλων. Η χρήση της είναι ήδη αρκετά δημοφιλής και σίγουρα υπάρχουν πολλές εφαρμογές που μπορούν να επιταχυνθούν με αυτήν τη διαδικασία. Τα FPGAs, θα απασχολήσουν ιδιαίτερα το επόμενο διάστημα, καθώς η επιτάχυνση με αυτά αποτελούν μια λύση, στους διάφορους περιορισμούς που διαφαίνονται σε σχέση με το νόμο του Moore, που θα εντείνουν περισσότερο την κατεύθυνση και αναζήτηση εναλλακτικών τρόπων για την αύξηση της απόδοσης των υπολογιστικών συστημάτων.

Ένα πολύ σημαντικό εργαλείο για την παραπάνω διαδικασία, αποτελεί το Vivado HLS 2019.2, της Xilinx. Όπως φάνηκε, το εργαλείο αυτό δίνει τη δυνατότητα για την αποτελεσματική υλοποίηση της σχεδίασης, την επιβεβαίωση της ορθότητας της, αλλά και την άντληση καταλυτικών συμπερασμάτων για αυτήν, σχετικά με τους πόρους που θα χρησιμοποιηθούν, αλλά και με την ταχύτητα εκτέλεσής της, σε περιβάλλον προσομοίωσης. Είναι εμφανές, πως αυτό είναι ένα πολύ μεγάλο λειτουργικό, χρονικό

και οικονομικό πλεονέκτημα των FPGA και των εργαλείων που μπορούν να χρησιμοποιηθούν με αυτά, έναντι των One Time Programmable (OTP) συσκευών.

Σημειώνεται επίσης, πως ήταν καταλυτική η ανάλυση του Intel® VTune™ Profiler, η οποία και κατέδειξε την πιο κοστοβόρα συνάρτηση, κατά την εκτέλεση του παραδείγματος του CKKS scheme, της βιβλιοθήκης Microsoft SEAL, με το optimization flag -O3 ενεργοποιημένο. Στην πολυπλοκότητα που διακατέχει η βιβλιοθήκη αυτή, σε σχέση με την αλληλεξάρτηση των συναρτήσεων και των header files που εμπεριέχει, η χρήση του εργαλείου αυτού ήταν κομβική, για την καλύτερη κατανόηση της απόδοσής τους.

Το μεγάλο πρόβλημα με το Homomorphic Encryption είναι η ταχύτητα με την οποία μπορούν να γίνουν οι απαραίτητοι υπολογισμοί, για την επίτευξη της υλοποίησής του. Ένας τρόπος με τον οποίο μπορεί να βελτιωθεί αισθητά και ίσως να εξαλειφθεί το πρόβλημα αυτό, αποτελεί η κατεύθυνση της επιτάχυνσης υλικού/λογισμικού, με χρήση των FPGAs. Στα πλαίσια αυτής της διπλωματικής, δοκιμάστηκε αυτή η προσέγγιση, επιταχύνοντας κομμάτι του CKKS scheme στο παράδειγμα της βιβλιοθήκης Microsoft SEAL, παρατηρώντας αισθητή βελτίωση στο χρόνο εκτέλεσης, για την πράξη της διαίρεσης ενός uint128 bit, με έναν uint64-bit αριθμό. Στην προσπάθεια αυτή, επιτεύχθηκαν αρκετά αισθητά αποτελέσματα, από ***6.95***, έως και ***92.66*** φορές, πιο γρήγορη εκτέλεση, σε σχέση με την απλή εκτέλεση της συνάρτησης αυτής, 393485 φορές, με το optimization flag -O3, σε έναν υπολογιστή.

Η αξιοποίηση, λοιπόν, των FPGAs και της τεχνικής της επιτάχυνσης υλικού/λογισμικού, μπορεί να επιφέρει πολύ μεγάλες βελτιώσεις, σε πολύ κοστοβόρες εφαρμογές, όπως και αποτελούν τα διάφορα schemes του Homomorphic Encryption.

Βιβλιογραφία

[1] *Kim Laine Microsoft Research "Simple Encrypted Arithmetic Library 2.3.1". https://www.microsoft.com/en-us/research/uploads/prod/2017/11/sealmanual-2-3-1.pdf*

[2] *Microsoft SEAL on GitHub.*

[3] *Carlos Aguilar Melchor, Marc-Olivier Kilijian, C´edric Lefebvre and Thomas Ricosse. "A Comparison of the Homomorphic Encryption Libraries HElib, SEAL and FV-NFLlib."*

[4] *Sai Sri Sathya, Praneeth Vepakomma, Ramesh Raskar, Ranjan Ramachandra, and Santanu Bhattacharya "A Review of Homomorphic Encryption Libraries for Secure Computation."*

[5] *Alycia Carey "On the Explanation and Implementation of Three Open-Source Fully Homomorphic Encryption Libraries"*

[6] *David Archer, Lily Chen, Jung Hee Cheon, Ran Gilad-Bachrach, Roger A. Hallman, Zhicong Huang, Xiaoqian Jiang, Ranjit Kumaresan, Bradley A. Malin, Heidi Sofia, Yongsoo Song, Shuang Wang "APPLICATIONS OF HOMOMORPHIC ENCRYPTION."*

[7] *BFV. Zvika Brakerski "Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP"*

[8] *BGV. Zvika Brakerski, Vinod Vaikuntanathany "Efficient Fully Homomorphic Encryption from (Standard) LWE"*

[9] *Jung Hee Cheon Andrey Kim Miran Kim and Yongsoo Song "Homomorphic Encryption for Arithmetic of Approximate Numbers."*

[10] *Kristin Lauter, Michael Naehrig and Vinod Vaikuntanathan "Can Homomorphic Encryption be Practical?"*

[11] *Craig Gentry "Computing Arbitrary Functions of Encrypted Data"*

[12] *Craig Gentry September 2009 "A FULLY HOMOMORPHIC ENCRYPTION SCHEME"*

[13] *Ahmet Can Mert, Erdinç Öztürk, and Erkay Sava¸s "Design and Implementation of Encryption/Decryption Architectures for BFV Homomorphic Encryption Scheme."*

[14] *FHEW. Jacob Alperin-Sheriff, Chris Peikerty "Faster Bootstrapping with Polynomial Error."*

[15] *GSW. Craig Gentry Amit Sahai, Brent Waters "Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based."*

[16] *Craig Gentry, Shai Halevi, Nigel P. Smart "Homomorphic Evaluation of the AES Circuit."*

[17] *Ronald L. Rivest, Len Adleman, Michael L. Dertouzos "ON DATA BANKS AND PRIVACY HOMOMORPHISMS."*

[18] *Dustin Richmond, Matt Jacobsen August, 2016 "RIFFA 2.2.2 Documentation."*

[19] *David Bruce Cousins, Kurt Rohloff, Daniel Sumorok "Designing an FPGA-Accelerated Homomorphic Encryption Co-Processor."*

[20] *N.P. Smart and F. Vercauteren "Fully Homomorphic SIMD Operations."*

[21] *Martin Albrecht, Shi Bai and Leo Ducas "A subfield lattice attack on overstretched NTRU assumptions. Cryptanalysis of some FHE and Graded Encoding Schemes."*

[22] *Iggy van Hoof, Elena Kirshanova and Alexander May "Quantum Key Search for Ternary LWE."*

[23] *Yongha Son and Jung Hee Cheon "Revisiting the Hybrid attack on sparse and ternary secret LWE."*

[24] *TFHE. Nicolas Gama, Malika Izabachene, Phong Q. Nguyen, Xiang Xie "Structural Lattice Reduction: Generalized Worst-Case to Average-Case Reductions and Homomorphic Cryptosystems."*

[25] *M. Sadegh Riazi, Kim Laine, Blake Pelton, Wei Dai "HEAX: An Architecture for Computing on Encrypted Data."*

[26] *"Vivado Design Suite Tutorial High-Level Synthesis", UG871 (v2019.2) January 27, 2020*

[27] *Young-kyu Choi and Jason Cong "HLS-Based Optimization and Design Space Exploration for Applications with Variable Loop Bounds."*

[28] *Ahmad Al Badawi, Luong Hoang, Chan Fook Mun, Kim Laine, Khin Mi Mi Aung "PrivFT: Private and Fast Text Classification with Homomorphic Encryption."*

[29] *Hao Chen, Kim Laine, Rachel Player, and Yuhou Xia "High-Precision Arithmetic in Homomorphic Encryption."*

[30] *Erdinç Öztürk, Yarkın Doroz, and Erkay Savas "A Custom Accelerator for Homomorphic Encryption Applications."*

[31] *David Bruce Cousins, John Golusky, Kurt Rohloff, Daniel Sumorok "An FPGA Co-Processor Implementation of Homomorphic Encryption."*

[32] *Sujoy Sinha Roy, Furkan Turan, Kimmo Jarvinen, Frederik Vercauteren and Ingrid Verbauwhede "FPGA-based High-Performance Parallel Architecture for Homomorphic Computing on Encrypted Data."*

[33] *Sunwoong Kim, Keewoo Lee, Wonhee Cho, Yujin Nam, Jung Hee Cheon, and Rob A. Rutenbar "Hardware Architecture of a Number Theoretic Transform for a Bootstrappable RNS-based Homomorphic Encryption Scheme."*

[34] *Wonkyung Jung, Eojin Lee, Sangpyo Kim, Keewoo Lee, Namhoon Kim, Chohong Miny, Jung Hee Cheon, and Jung Ho Ahn "HEAAN Demystified: Accelerating Fully Homomorphic Encryption Through Architecture-centric Analysis and Optimization."*

[35] *Deepika Natarajan and Wei Dai "SEAL-Embedded: A Homomorphic Encryption Library for the Internet of Things."*

[36] *Awesome HE on GitHub*

[37] *Critical Path Method on Wikipedia*

[38] *Moore's Law on Wikipedia*

[39] *Thread-Local Storage (TLS) on Wikipedia*

[40] *Advanced eXtensible Interface on Wikipedia*

[41] *Homomorphic Encryption on Wikipedia*

[42] *Learning With Errors (LWE) on Wikipedia*

[43] *Ring Learning With Errors (RLWE) on Wikipedia*

[44] *Cryptanalysis on Wikipedia*

[45] *Lattice-based Cryptography on Wikipedia*

[46] *Block Size (Cryptography) on Wikipedia*

[47] *NTRU on Wikipedia*

[48] *Symmetric-key Algorithm on Wikipedia*

[49] *Public-key Cryptography on Wikipedia*

[50] *Introduction to FPGAs' Architecture, by Xilinx*

[51] *HLS Pragmas, by Xilinx*

[52] *Alveo U200 Data Center Accelerator Card*

[53] *Xilinx Board Store on GitHub*