ΠΟΛΥΤΕΧΝΕΙΟ ΚΡΗΤΗΣ
Τμήμα Ηλεκτρονικών Μηχανικών & Μηχανικών Υπολογιστών

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Κρυπτογραφική ανάλυση των αλγορίθμων LM hash, MD5 και SHA-1 με χρήση πινάκων ουράνιου τόξου σε αρχιτεκτονική CUDA

Μιχάλης Τρουλλινός

Επιβλέπων Καθηγητής:
Καθηγητής Διονύσιος Πνευματικάτος

Εξεταστική επιτροπή:
Καθηγητής Διονύσιος Πνευματικάτος
Καθηγητής Απόστολος Δόλλας
Αναπληρωτής Καθηγητής Ιωάννης Παπαευσταθίου

Χανιά ,Ιούνιος 2010

Ευχαριστίες

Θα ήθελα να ευχαριστήσω θερμά τον καθηγητή μου κ. Διονύση Πνευματικάτο για την ανάθεση της παρούσας διπλωματικής και για την πολύτιμη καθοδήγηση που μου παρείχε κατά την διάρκεια εκπόνησης της εργασίας. Επίσης τους καθηγητές Απόστολο Δόλλα και Ιωάννη Παπαευσταθίου για την αποδοχή τους να αξιολογήσουν αυτή την εργασία.
Επίσης θα ήθελα να ευχαριστήσω τον υπεύθυνο του εργαστηρίου Μικροεπεξεργαστών και Υλικού, τον κ. Μάρκο Κιμιωνή για την παροχή του απαιτούμενου υλικού.
Επιπλέον θα ήθελα να ευχαριστήσω τον Κώστα Θεοχαρούλη για τις χρήσιμες συμβουλές και υποδείξεις του.
Με την εργασία αυτή τελειώνω και τις σπουδές μου ως προπτυχιακού φοιτητή θα ήθελα να ευχαριστήσω όλους τους καθηγητές μου, τους συμφοιτητές μου και ιδιαίτερα την οικογένεια μου για την στήριξη και την υπομονή τους σε όλα τα χρόνια των σπουδών μου.

Χανιά, Ιούνιος 2010

Περιεχόμενα

Πρόλογος

Η κρυπτογραφία (cryptography) ασχολείται με την κωδικοποίηση της πληροφορίας σε μια ακατανόητη (κρυφή) μορφή με σκοπό την ασφαλή αποστολή και αποθήκευση ευαίσθητων πληροφοριών. Μία εφαρμογή της κρυπτογραφίας, η οποία δεν χρησιμοποιεί κλειδί για την κρυπτογράφηση, είναι η ασφαλής αποθήκευση κωδικών πρόσβασης. Αντί για την αποθήκευση των κωδικών πρόσβασης αποθηκεύουμε την κρυπτογραφημένη μορφή τους ώστε σε περίπτωση υποκλοπής της βάσης δεδομένων να μην είναι εκτεθειμένοι οι κωδικοί. Όταν κάποιος νόμιμος χρήστης θέλει να συνδεθεί με τον λογαριασμό του, το σύστημα αρχικά κρυπτογραφεί τον κωδικό που εισάγει και στην συνέχεια τον συγκρίνει με τον κρυπτογραφημένο κωδικό που υπάρχει στην βάση δεδομένων.

Αντίθετα η κρυπτογραφική ανάλυση (cryptanalysis) ασχολείται με την παραβίαση κρυπτογραφημένων πληροφοριών χωρίς την γνώση του μυστικού κλειδιού. Η πιο απλή μέθοδος παραβίασης, αλλά μη αποτελεσματική, είναι η εξαντλητική δοκιμή όλων των πιθανών κλειδιών μέχρι να βρεθεί το σωστό. Μια καλύτερη μέθοδος η οποία δεν μπορεί να χρησιμοποιηθεί στην πράξη καθώς έχει τεράστιες απαιτήσεις σε αποθηκευτικό χώρο θα ήταν να κρυπτογραφήσουμε μια φορά όλους τους κωδικούς και να αποθηκεύσουμε τα αποτελέσματα ώστε να μπορούν να χρησιμοποιηθούν επανειλημμένα για παραβίαση απλά με τον εντοπισμό του σωστού ζευγαριού. Ο M.Hellman το 1980 [1] πρότεινε μια τεχνική ανταλλαγής χώρου-χρόνου (space-time tradeoff) που βρίσκεται μεταξύ των δύο παραπάνω λύσεων με την οποία χρησιμοποιείται λιγότερο χώρος καθώς αποθηκεύετε ένα μικρό μέρος από τον συνολικό αριθμό ζευγαριών αλλά χρησιμοποιείται περισσότερος χρόνος για τον εντοπισμό των ζευγαριών. Η αρχική ιδέα του M.Hellman βελτιώθηκε από τον P.Oechslin το 2003[2] ο οποίος πρότεινε μια τεχνική που χρησιμοποιεί πίνακες ουράνιου τόξου (rainbow tables). Με τους πίνακες ουράνιου τόξου, οι οποίοι είναι πίνακες αναζήτησης (lookup tables), επιτυγχάνεται αύξηση της πιθανότητας εύρεσης ενός κωδικού για τις ίδιες απαιτήσεις χώρου με σχέση την αρχική τεχνική ανταλλαγής χώρου-χρόνου του M.Hellman.

Ακόμα όμως και με την χρήση των πινάκων ουράνιου τόξου για να πραγματοποιηθεί μια παραβίαση σε ανεκτό χρονικό διάστημα απαιτείται μεγάλη επεξεργαστική ισχύς. Για την λύση αυτού του προβλήματος έχουν προταθεί πολλές λύσεις, όπως για παράδειγμα η χρήση υπερυπολογιστών, συμπλεγμάτων υπολογιστών, και πλατφόρμες FPGA[25][26].

Το RainbowCrack[3] αποτελεί μια ολοκληρωμένη λύση για την κατασκευή και χρήση πινάκων ουράνιου τόξου με υλοποιήσεις σε C++ και σε αρχιτεκτονική CUDA. Ο κώδικας σε C++ για μία παλιότερη έκδοση της εφαρμογής είναι δημοσιοποιημένος ως ανοικτό λογισμικό όμως η υλοποίηση σε CUDA είναι εμπορικό προϊόν με κλειστό κώδικα.

Ο Κ.Θεοχαρούλης[4] προτείνει μια λύση χαμηλού κόστους με μια πλατφόρμα FPGA με την οποία δημιουργεί τους πίνακες ουράνιου τόξου για τους αλγόριθμους κρυπτογράφησης LM hash, MD5 και SHA-1.

Στην παρούσα εργασία προτείνεται μια λύση για την δημιουργία των πινάκων ουράνιου τόξου με την αξιοποίηση της τεράστιας υπολογιστικής ισχύς των σύγχρονων καρτών γραφικών που είναι ευρέος διαδεδομένες και διατίθενται με χαμηλό κόστος. Με την λύση αυτή είναι εφικτή η ταχύτερη κατασκευή των πινάκων 50 με 117 φορές σε σύγκριση με την κατασκευή τους με χρήση επεξεργαστών γενικού σκοπού.

Τα τελευταία χρόνια η βιομηχανία των υπολογιστών για να πετύχει αύξηση των επιδόσεων έχει μεταβεί σε μικροεπεξεργαστές παράλληλης επεξεργασίας που διαθέτουν πολλαπλούς πυρήνες (multi-cores)[5]. Το πιο αντιπροσωπευτικό δείγμα αυτής της τάσης είναι οι κάρτες γραφικών (Graphics Processors Units) που έχουν εξελιχτεί σε συστήματα παράλληλης επεξεργασίας με εκατοντάδες επεξεργαστές και χρησιμοποιούν μνήμες με μεγάλο εύρος ζώνης και υψηλές ταχύτητες μεταφοράς δεδομένων. Για την εξέλιξη αυτή μεγάλο ρόλο έχει παίξει η προσοδοφόρα βιομηχανία ηλεκτρονικών παιχνιδιών που απαιτεί από τις σύγχρονες κάρτες γραφικών να διαθέτουν μεγάλη επεξεργαστική ισχύ ώστε να είναι εφικτή η απεικόνιση

τρισδιάστατων γραφικών υψηλής ευκρίνειας σε πραγματικό χρόνο[6].

Πριν μερικά χρόνια δεν ήταν εύκολο να αξιοποιηθεί η υπολογιστικής ισχύς των καρτών καθώς ο προγραμματισμός γενικού σκοπού σε κάρτες γραφικών (GPGPU), δηλαδή για εργασίες που δεν σχετίζονται με επεξεργασία γραφικών, απαιτούσε προγραμματισμό στην γλώσσα της κάρτας. Επιπλέον υπήρχαν περιορισμοί στην απόδοση καθώς ο κώδικας εκτελούνταν στο API της γλώσσας γραφικών και όχι απευθείας πάνω στο υλικό[7]. Αυτό άλλαξε τον Νοέμβριο του 2006 όταν η NVIDIA™ παρουσίασε την CUDA (Compute Unified Device Architecture) [8], μια γενικού σκοπού παράλληλη αρχιτεκτονική η οποία χρησιμοποιεί ένα υπερσύνολο της υψηλού επιπέδου γλώσσας προγραμματισμού C και η οποία εκτελείται απευθείας στο υλικό της κάρτας.

Οργάνωση Εργασίας

Η εργασία είναι οργανωμένη σε έξι κεφάλαια από τα οποία στα τρία πρώτα γίνεται εισαγωγή των απαραίτητων εννοιών που χρειάζονται για την κατανόηση της υλοποίησης μου. Στο πρώτο κεφάλαιο γίνεται εισαγωγή στην κρυπτογραφία και αναλύω τις κρυπτογραφικές συναρτήσεις κατακερματισμού που θα χρησιμοποιήσω στο πλαίσιο της εργασίας. Το δεύτερο κεφάλαιο αναφέρεται στην κρυπτογραφική ανάλυση και τις διάφορες τεχνικές παραβίασης δίνοντας ιδιαίτερο βάρος στην μέθοδο με χρήση πινάκων ουράνιου τόξου. Το τρίτο κεφάλαιο αποτελεί μια εισαγωγή στην CUDA. Στο δεύτερο μέρος παρουσιάζεται η υλοποίησή μου και τα αποτελέσματα που πήρα. Αναλυτικότερα, στο τέταρτο κεφάλαιο αναφέρω τον σχεδιασμό μου και την υλοποίηση της εφαρμογής μου. Στο πέμπτο κεφάλαιο παρουσιάζονται τα αποτελέσματα και γίνεται σύγκριση με άλλες μεθόδους και στο τελευταίο κεφάλαιο αναλύονται τα συμπεράσματα και αναφέρεται οι μελλοντικές εργασίες που θα μπορούσαν να γίνουν.

Cryptanalysis of the hash algorithms LM hash, MD5 and SHA-1 using rainbow tables in the CUDA architecture.

Abstract : This thesis presents cryptanalysis of the hash algorithms LM hash, MD5 and SHA-1 using rainbow tables in the CUDA architecture. We use a parallel implementation for generating keys of variable length up to 64 bytes based on Hellman's time-memory trade off technique. GPU's parallel implementation performs 50 to 117 times faster than a general purpose microprocessor.

*1.*Κρυπτογραφία

1.1 Γενικά για την κρυπτογραφία

Με τον όρο κρυπτογραφία (cryptography) αναφερόμαστε σε ένα κλάδο της επιστήμης της κρυπτολογίας (cryptology) που έχει ως αντικείμενο την μελέτη της ασφαλούς επικοινωνίας. Ο κύριος τρόπος για την επίτευξη ασφαλούς επικοινωνίας, που σε αυτόν οφείλει η κρυπτογραφία το όνομα της, είναι η μετατροπή των πληροφοριών σε μία ακατανόητη (κρυφή) μορφή.

Η κρυπτογραφία συνήθως καλείται να ικανοποιεί τις ακόλουθες τέσσερις ιδιότητες, την *εμπιστευτικότητα (confidentially)* όπου πρόσβαση σε εμπιστευτικές πληροφορίες έχουν μόνο εξουσιοδοτημένα μέλη και αποτρέπεται η πρόσβαση σε τρίτους, την *αυθεντικότητα (authentication)* με την οποία ο αποστολέας και ο παραλήπτης μπορούν να διαβεβαιώσουν την ταυτότητα τους καθώς και την πηγή και τον προσδιορισμό της πληροφορίας έχοντας την διαβεβαίωση ότι οι ταυτότητες τους δεν είναι πλαστές. Επόμενη ιδιότητα που πρέπει να ικανοποιείται είναι η *ακεραιότητα (integrity)* με την οποία η οποιαδήποτε αλλαγή στην πληροφορία από μη εξουσιοδοτημένα μέλη μπορεί να ανιχνευτεί. Τέλος η *μη απάρνηση (Nonrepudiation)* με την οποία ο αποστολές και ο παραλήπτης δεν μπορεί να αρνηθεί την αυθεντικότητα της μετάδοσης ή της δημιουργίας της [9].
Έχει καθιερωθεί να χρησιμοποιείται η ακόλουθη ορολογία την οποία θα ακολουθήσουμε και εμείς σε όλο το πλαίσιο της εργασίας.

- *Αρχικό κείμενο (plaintext ή cleartext)* είναι το μήνυμα πριν να γίνει σε αυτό οποιαδήποτε παρέμβαση και βρίσκεται σε μια κατανοητή μορφή.
- Κρυπτοκείμενο (ciphertext) είναι το μήνυμα αφότου έχει υποστεί παρέμβαση και είναι σε μια ακατανόητη(κρυφή) μορφή.
- *Κρυπτογράφηση (encryption)* ονομάζεται η διαδικασία μετασχηματισμού του plaintext σε ciphertext.
- *Αποκρυπτογράφηση (decryption)* είναι η αντίστροφη διαδικασία από την κρυπτογράφηση με την οποία μετατρέπουμε το ciphertext σε plaintext.
- *Αλγόριθμος κρυπτογράφησης (encryption algorithm)* είναι μια μαθηματική συνάρτηση που χρησιμοποιείται για την κρυπτογράφηση και την αποκρυπτογράφηση.
- *Κλειδί (key)* είναι μια συμβολοσειρά από bits που χρησιμοποιείται ως είσοδος στον αλγόριθμο κρυπτογράφησης με σκοπό να διαφοροποιεί την έξοδό του και που χωρίς αυτό ο αλγόριθμος δεν μπορεί να λειτουργήσει.

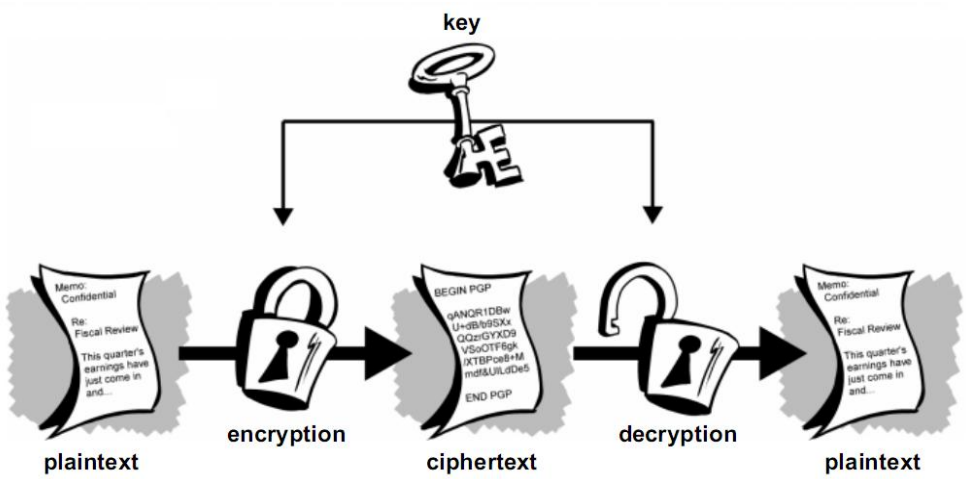
Οι απαρχές της κρυπτογραφίας ξεκινάνε από την αρχαιότητα όπου η χρήση της κρυπτογραφίας επικεντρωνόταν σε μορφές επικοινωνίας κατανοητές από ανθρώπους όπως κείμενο[10]. Αντίθετα στην σύγχρονη μορφή της η κρυπτογραφία αναφέρεται σε επικοινωνία αναφέρεται σε δεδομένα που αναπαριστούνται από δυαδική μορφή που είναι κατανοητά για επεξεργασία από ηλεκτρονικούς υπολογιστές και βασίζεται στα μαθηματικά και την θεωρία πληροφορίας[11]. Για όλο το υπόλοιπο πλαίσιο της εργασίας όταν αναφέρομαι στην κρυπτογράφηση θα αφορά την σύγχρονη μορφή της.

Ένα βασικό χαρακτηριστικό των καλών συστημάτων κρυπτογράφησης είναι ότι η ασφάλεια του συστήματος εξαρτάται εξ' ολοκλήρου από το κλειδί και όχι από την γνώση του αλγορίθμου. Αυτό έχει το πλεονέκτημα ο αλγόριθμος μπορεί να δημοσιευτεί ώστε να χρησιμοποιηθεί από οποιονδήποτε και να αναλυθεί ώστε να αξιολογηθεί το επίπεδο ασφάλειας που παρέχει. Αν η ασφάλεια ενός συστήματος εξαρτιόταν από τον αλγόριθμό, τότε θα υπήρχε ο κίνδυνος με την αποκάλυψη του αλγορίθμου όλα τα δεδομένα που έχουν κρυπτογραφηθεί από αυτόν να ήταν εκτεθειμένα σε μη εξουσιοδοτημένα μέλη[10].

Το μέγεθος του κλειδιού μετράται σε bits και γενικά όσο μεγαλύτερο είναι ένα κλειδί τόσο καλύτερη ασφάλεια παρέχεται καθώς η εξαντλητική δοκιμή όλων των κλειδιών γίνεται πιο δύσκολη. Το πλήθος των δυνατών κλειδιών που μπορούν να αναπαρασταθούν είναι 2^m όπου m είναι το μέγεθος του κλειδιού σε bits. Θα πρέπει να επισημάνουμε όμως ότι μόνο το μέγεθος του κλειδιού δεν εξασφαλίζει ασφάλεια καθώς η ασφάλεια εξαρτάται και από τον αλγόριθμο κρυπτογράφησης. Αυτό σημαίνει ότι για να εξασφαλίσουμε το ίδιο επίπεδο ασφάλειας με διαφορετικούς αλγόριθμους απαιτείται διαφορετικό μέγεθος κλειδιού.

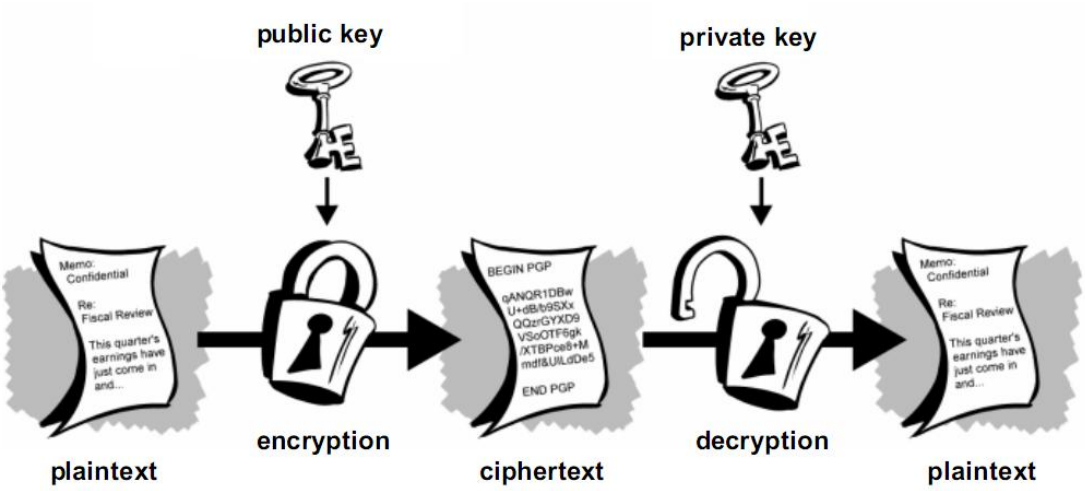
1.2 Είδη Κρυπτογραφίας

Οι δύο κύριες μορφές κρυπτογραφίας είναι η κρυπτογραφία *συμμετρικού κλειδιού (symmetric-key cryptography) και η κρυπτογραφία δημόσιου κλειδιού (public-key cryptography)* .Στην πρώτη μορφή *γίνεται* χρήση του ίδιου κλειδιού για την κρυπτογράφηση και την αποκρυπτογράφηση.



Σχήμα 1.1 *Κρυπτογράφηση συμμετρικού κλειδιού*

Στην δεύτερη μορφή την *κρυπτογραφία δημόσιου κλειδιού* έχουμε κλειδιά με την μορφή ζευγαριών όπου το ένα χρησιμοποιείται αποκλειστικά για την κρυπτογράφηση και το άλλο αποκλειστικά για την αποκρυπτογράφηση[11]. Σε αυτήν την εργασία θα ασχοληθούμε μόνο με κρυπτογραφία συμμετρικού κλειδιού.



Σχήμα 1.2 *Κρυπτογράφηση δημόσιου κλειδιού*

1.3 Αλγόριθμοι Συμμετρικής Κρυπτογράφησης (Symmetric-Key Encryption Algorithms)

DES

Ο αλγόριθμος *DES (Data Encryption Standard)* αναπτύχθηκε από την IBM σε συνεργασία με την Υπηρεσία Εθνικής Ασφάλειας (National Security Agency) των ΗΠΑ. Επενεργεί πάνω σε αρχικό κείμενο μήκους 64 bit και χρησιμοποιεί κλειδιά 56 bits παράγοντας ένα κρυπτοκείμενο μήκους 64 bits. Ο DES αποτελούσε το επίσημο κρυπτογραφικό σύστημα της κυβέρνησης των ΗΠΑ από το 1976 μέχρι το 2002. Με την δημοσίευση του το 1976 αποτέλεσε πεδίο ενδιαφέροντος για την ακαδημαϊκή μελέτη και μελετήθηκε εντατικά για πολλά χρόνια. Κύριο μειονέκτημα του είναι το μικρό μήκος κλειδιού που χρησιμοποιεί και που τον καθιστά ανασφαλή. Το 2007 ένα ακαδημαϊκό project κατάφερε να σπάσει τον DES σε 6,4 μέρες με κόστος μικρότερο από 10.000 δολάρια[12].

Triple DES

Ο αλγόριθμος *Triple DES* δημοσιεύτηκε το 1998, αποτελεί μια παραλλαγή του DES και οφείλει το όνομα του στις τρεις κρυπτογραφήσεις που εφαρμόζει με τον DES σε κάθε μπλοκ δεδομένων. Επενεργεί σε αρχικό κείμενο μήκους 64 bits, χρησιμοποιεί κλειδιά 112 bits και 168 bits παράγοντας κρυπτοκείμενο μήκους 64 bits. Η βασική ιδέα είναι ότι το αρχικό μήνυμα κωδικοποιείται με ένα κλειδί, στην συνέχεια αποκωδικοποιείται με ένα δεύτερο κλειδί και τέλος κωδικοποιείται με ένα τρίτο κλειδί. Από την επιλογή των τριών αυτών κλειδιών προκύπτει το μήκος του κλειδιού. Οι επιλογές που υπάρχουν είναι δύο, αν τα τρία κλειδιά είναι διαφορετικά τότε προκύπτει ένα κλειδί 3X56 bits =168 bits. Η δεύτερη επιλογή είναι η χρήση ίδιου κλειδιού για τα την πρώτη και την τρίτη κωδικοποίηση και διαφορετικό για την δεύτερη που απαιτεί κλειδί 2X56 bits=112 bits. Όμως η πραγματική ασφάλεια που παρέχεται με την πρώτη επιλογή κλειδιού δεν είναι 168 bits καθώς με μια τεχνική γνωστή ως **meet-in-the-middle attack** περιορίζεται η ασφάλεια στα 128 bits. Ακόμα και έτσι όμως με έναν απλό τρόπο μπορεί να ξεπεραστεί το βασικότερο μειονέκτημα του DES που είναι το μικρό μέγεθος κλειδιού [13].

AES

Ο αλγόριθμος *AES (Advanced Encryption Standard)* σχεδιάστηκε από τους Vincent Rijmen και Joan Daemen και δημοσιεύτηκε για πρώτη φορά το 1998 ενώ αποτελεί το επίσημο κρυπτογραφικό σύστημα των ΗΠΑ από το 2002 οπότε αντικατάστησε τον DES. Επενεργεί πάνω σε αρχικό κείμενο μήκους 128 bits και χρησιμοποιώντας κλειδιά μήκους 128,192 και 256 bits παράγοντας κρυπτοκείμενο μήκους 128 bits [14].

1.3.1 Ο Κρυπτογραφικός αλγόριθμος DES

Η κρυπτογράφηση στον αλγόριθμο DES πραγματοποιείται σε δύο φάσεις. Στην πρώτη φάση παράγουμε 16 συμβολοσειρές των 48 bits που ονομάζονται subkeys που οι τιμές τους εξαρτώνται αποκλειστικά από το κλειδί. Στο δεύτερο μέρος χρησιμοποιούμε τα subkeys για να κρυπτογραφήσουμε το αρχικό κείμενο.
Στην πρώτη φάση ξεκινάει με την μετάθεση των bits του κλειδιού σύμφωνα με τον πίνακα PC-1

57	49	41	33	25	17	9
1	58	50	42	34	26	18
10	2	59	51	43	35	27
19	11	3	60	52	44	36
63	55	47	39	31	23	15
7	62	54	46	38	30	22
14	6	61	53	45	37	29
21	13	5	28	20	12	4

Πίνακας 1.1 Ο Πίνακας PC-1

σύμφωνα με την λογική ότι η τιμή που έχει κάθε θέση δείχνει το bit του αρχικού κλειδιού και η θέση του πίνακα δείχνει το bit του νέου κλειδιού. Συγκεκριμένα το πρώτο bit του νέου κλειδιού θα είναι το 57ο του παλιού. Αντίστοιχα το δεύτερο bit του νέου κλειδιού θα είναι το 49ο του παλιού. Παρατηρούμε ότι ο πίνακας δεν έχει τις τιμές που αντιστοιχούν στα parity bits του αρχικού κλειδιού όπως το 8ο,16ο,… και έτσι το νέο κλειδί έχει 56 bits. Επόμενο βήμα είναι η δημιουργία 17 ζευγαριών από συμβολοσειρές των 28bit, C_n και D_n με $0 \le n \le 16$. Τα πρώτα 28 bits του νέου κλειδιού είναι το C_0 και τα επόμενα 28 το D_0. Για όλα τα υπόλοιπα θα ακολουθήσουμε την επόμενη διαδικασία. Θα πραγματοποιήσουμε αριστερή ολίσθηση στα C_0 και D_0 κατά μια θέση για να δημιουργήσουμε τα C_1 και D_1 αντίστοιχα. Τα πιο σημαντικά bits των C_0 και D_0 πριν από την ολίσθηση τοποθετούνται στις λιγότερο σημαντικές θέσεις των C1 και D1 αντίστοιχα. Για παράδειγμα αν C_0= 1111000 0110011 0010101 0101111 και D_0= 0101010 1011001 1001111 0001111 θα πάρουμε C_1 =1110000 1100110 0101010 1011111 και D_1 =1010101 0110011 0011110 0011110. Επαναλαμβάνουμε την ίδια διαδικασία ώστε από τα C_1 και D_1 να παίρνουμε τα C_2 και τα D_2 αντίστοιχα. Αυτό θα επαναληφτεί μέχρι να πάρουμε τα C_{16} και D_{16} απλά το μόνο που αλλάζει είναι ο αριθμός των bits που ολισθαίνουν προς τα αριστερά. Κατά μία θέση ολισθαίνουν τα bits στην 1η, 2η, 9η και 16η περιστροφή, ενώ σε όλες τις υπόλοιπες η ολίσθηση είναι κατά δύο θέσεις. Τα subkeys K_n με $1 \le n \le 16$ τα παίρνουμε αν μεταθέσουμε τα συνενωμένα bits των C_nD_n για $1 \le n \le 16$ σύμφωνα με τον πίνακα PC-2.

14	17	11	24	1	5
3	28	15	6	21	10
23	19	12	4	26	8
16	7	27	20	13	2
41	52	31	37	47	55
30	40	51	45	33	48
44	49	39	56	34	53
46	42	50	36	29	32

Πίνακας 1.2 *Ο πίνακας PC-2*

Παρατηρούμε ότι χρησιμοποιούμε μόνο τα 48 από τα 56 bits του συνενωμένου ζευγαριού για την κατασκευή των subkeys. Για παράδειγμα το συνενωμένο ζευγάρι
C_1D_1=1110000 1100110 0101010 1011111 1010101 0110011 0011110 0011110 με την μετάθεση ακολουθώντας την ίδια λογική όπως του πίνακα PC1 θα πάρουμε το

K_1=000110 110000 001011 101111 111111 000111 000001 110010. Συνεχίζουμε αυτή την διαδικασία μέχρι να πάρουμε και τα 16 subkeys. Σε αυτό το σημείο έχουμε ολοκληρώσει την πρώτη φάση καθώς έχουμε κατασκευάσει και τα 16 subkeys.

Στη δεύτερη φάση όπου θα χρησιμοποιήσουμε τα 16 subkeys που κατασκευάσαμε αρχίζουμε με την μετάθεση του plaintext $ μήκους 64 bits σύμφωνα με τον πίνακα IP(initial permutation).

58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7

Πίνακας 1.3 *Ο πίνακας IP.*

Πλέον έχουμε το νέο plaintext $ που έχει 64 bits, όπως και το αρχικό, και το οποίο χωρίζουμε σε δύο ίσα μέρη, τα L_0 και το R_0.Τα πρώτα 32 bits του νέου αρχικού μηνύματος είναι το L_0 και τα υπόλοιπα 32 το R_0. Σε αυτό το στάδιο θα πραγματοποιήσουμε 16 επαναλήψεις χρησιμοποιώντας την συνάρτηση γύρου **f (**Feistel function). Η συνάρτηση αυτή επενεργεί σε δύο συμβολοσειρές από bits, την πρώτη με μήκος 32bits και η δεύτερη με μήκος 48 bits και επιστρέφει ένα μπλοκ με μήκος 32 bits. Η συμβολοσειρά των 32 bits είναι το αποτέλεσμα της συνάρτησης γύρου από προηγούμενες επαναλήψεις ενώ η συμβολοσειρά των 48 bits είναι ένα subkey το οποίο είναι διαφορετικό σε κάθε γύρο. Ο γενικός τύπος για την επανάληψη n είναι

$$L_n=R_{n-1}$$
$$R_n=(L_{n-1}) \text{ XOR } f(R_{n-1},K_n)$$

Παράδειγμα για την πρώτη επανάληψη με n=1 ισχύει

$$L_1=R_0$$
$$R_1=L_0 \text{ XOR } f(R_0,K_1)$$

Αυτή η διαδικασία θα επαναληφτεί 16 φορές ώστε να πάρουμε τα L_{16},R_{16}. Σε αυτό το σημείο θα πρέπει να εξηγήσουμε πώς δουλεύει η συνάρτηση γύρου **f (**Feistel function). Ξεκινάμε χρησιμοποιώντας τον πίνακα E-BIT και επεκτείνουμε το μπλοκ R_{n-1} από 32 σε 48 bits επαναχρησιμοποιώντας κάποια bits.

32	1	2	3	4	5
4	5	6	7	8	9
8	9	10	11	12	13
12	13	14	15	16	17
16	17	18	19	20	21
20	21	22	23	24	25
24	25	26	27	28	29
28	29	30	31	32	1

Πίνακας 1.4 *Ο πίνακας E-BIT*

Μετά την επέκταση του R_{n-1} εφαρμόζουμε την λογική πράξη xor με το subkey K_n σε κάθε bit. Το αποτέλεσμα μήκους 48 bits το θεωρούμε οκτώ ομάδες τον έξι bits τις B1,B2,B3,B4,B5,B6,B7,B8. Με κάθε μία από αυτές τις ομάδες θα παίρνουμε την τιμή που

περιέχουν οκτώ πίνακες με προκαθορισμένες τιμές από κάποια θέση που προσδιορίζεται από την τιμή των B_i. Οι οκτώ αυτοί πίνακες έχουν μια ειδική ονομασία που ονομάζονται S-Boxes. Τα S-Boxes αποτελούνται από 8 πίνακες τους S1,S2,S3,S4,S5,S6,S7,S8 διαστάσεων 16x4 που περιέχουν προκαθορισμένες τιμές των 4 bits. Για λόγους οικονομίας χώρου τα S-Boxes παρουσιάζονται στο παράρτημα. Κάθε μια από τις οκτώ ομάδες Bi αντιστοιχεί στον πίνακα Si ενώ ο προσδιορισμός της θέσης του πίνακα όπου θα πάρουμε την νέα τιμή γίνεται για την γραμμή του πίνακα από την δεκαδική τιμή των δύο πρώτων bits και για την στήλη από την δεκαδική τιμή που έχουν τα υπόλοιπα 4 bits. Η τιμή που επιστρέφει η f είναι η S1(B1)S2(B2)S3(B3)S4(B4)S5(B5)S6(B6)S7(B7)S8(B8). Το παραπάνω επαναλαμβάνεται 16 φορές μέχρι να έχουμε τις τιμές των L16 και R16. Στην συνένωση R16L16 πραγματοποιούμε την τελική μετάθεση σύμφωνα με τον πίνακα IP^{-1}.

40	8	48	16	56	24	64	32
39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26
33	1	41	9	49	17	57	25

Πίνακας 1.5 Ο πίνακας IP^{-1}.

Στο σημείο αυτό έχει ολοκληρωθεί η κρυπτογράφηση και το αποτέλεσμα της μετάθεσης είναι το κρυπτοκείμενο. Για την αποκρυπτογράφηση ακολουθούμε ακριβώς τα ίδια βήματα με μόνη διαφορά ότι εφαρμόζουμε τα subkeys στην συνάρτηση γύρου με αντίστροφη φορά.

1.4 Κρυπτογραφικές συναρτήσεις κατακερματισμού

Με τις κρυπτογραφικές συναρτήσεις κατακερματισμού*(cryptographic hash functions)* μια μεταβλητού μήκους συμβολοσειρά από bit παράγει μια σταθερού μήκους συμβολοσειρά από bit. Η συμβολοσειρά από bits που αποτελεί την είσοδο ονομάζεται *μήνυμα (message)* και η συμβολοσειρά από bits που αποτελεί την έξοδος ονομάζεται *hash value* ή *σύνοψη (digest).*

Οι ιδανικές συναρτήσεις κατακερματισμού εξασφαλίζουν ότι με την παραμικρή μεταβολή της εισόδου έστω και για ένα bit θα αλλάξει η τιμή της σύνοψης. Μια ακόμα ιδιότητα που έχουν οι ιδανικές συναρτήσεις κατακερματισμού είναι ότι είναι μη ανηστρέψιμες(one-way) δηλαδή ο υπολογισμός της σύνοψης για οπουδήποτε μήκος μηνύματος είναι απλή διαδικασία ενώ η εύρεση του μηνύματος από την σύνοψη πολύπλοκη διαδικασία έως αδύνατη. Τέλος οι ιδανικές συναρτήσεις κατακερματισμού εξασφαλίζουν ότι δύο διαφορετικά μηνύματα έχουν διαφορετικές συνόψεις [11].

Οι συναρτήσεις κατακερματισμού βρίσκουν πολλές εφαρμογές στην κρυπτογραφία και ειδικότερα σε εφαρμογές του διαδικτύου όπου απαιτείται εξασφάλιση της ιδιότητας της αυθεντικότητας (*authentication)* που περιγράψαμε παραπάνω [11].

Κάποιοι διαδεδομένοι αλγόριθμοι κρυπτογραφικών συναρτήσεων κατακερματισμού είναι ο LM hash, ο MD5 και ο SHA-1. Καθώς θα χρησιμοποιήσω αυτούς τους αλγορίθμους στην εργασία μου ακολουθεί αναλυτική περιγραφή για κάθε έναν.

1.4.1 Ο αλγόριθμος LM hash

Ο LM hash χρησιμοποιήθηκε από κάποιες παλιότερες εκδόσεις των Microsoft Windows™ για να προφυλάξει τους κωδικούς των χρηστών μέσω της αποθήκευσης των συνόψεων που παράγουν οι κωδικοί αντί να αποθηκεύει τους κωδικούς. Το μέγιστο μήκος του μηνύματος που μπορεί να χρησιμοποιηθεί ως είσοδος είναι τα 112 bits και παράγει σύνοψη σταθερού μήκους

128 bits. Σαν είσοδος όμως μπορούν να χρησιμοποιηθούν μόνο οι χαρακτήρες ASCII που αντιστοιχούν σε εκτυπώσιμα γράμματα, ψηφία και σύμβολα(printable) [15].

Περιγραφή Αλγορίθμου

Τα βήματα που ακολουθούνται για την δημιουργία των συνόψεων είναι τα ακόλουθα: Αρχικά οι πεζοί χαρακτήρες του μηνύματος μετατρέπονται στους αντίστοιχους κεφαλαίους . Στην συνέχεια αν το μήνυμα είναι μικρότερο από 14 χαρακτήρες συμπληρώνεται με μηδενικά ώστε να απόκτηση μήκος 14 χαρακτήρων. Επόμενο βήμα είναι η διάσπαση της συμβολοσειράς σε δύο συμβολοσειρές από 7 χαρακτήρες και η εισαγωγή 7 bits ισότητας ώστε να αποκτήσουν μήκος 64 bits. Πλέον οι δύο συμβολοσειρές να μπορούν να χρησιμοποιηθούν σαν κλειδιά στον αλγόριθμο κρυπτογράφησης DES ενώ σαν αρχικό κείμενο χρησιμοποιείται πάντα η σταθερή συμβολοσειρά "KGS!@#$%". Τα δύο κρυπτομηνύματα που παράγονται από τις δύο κρυπτογραφήσεις με τον DES συνενώνονται και αποτελούν την σύνοψη του LM hash.

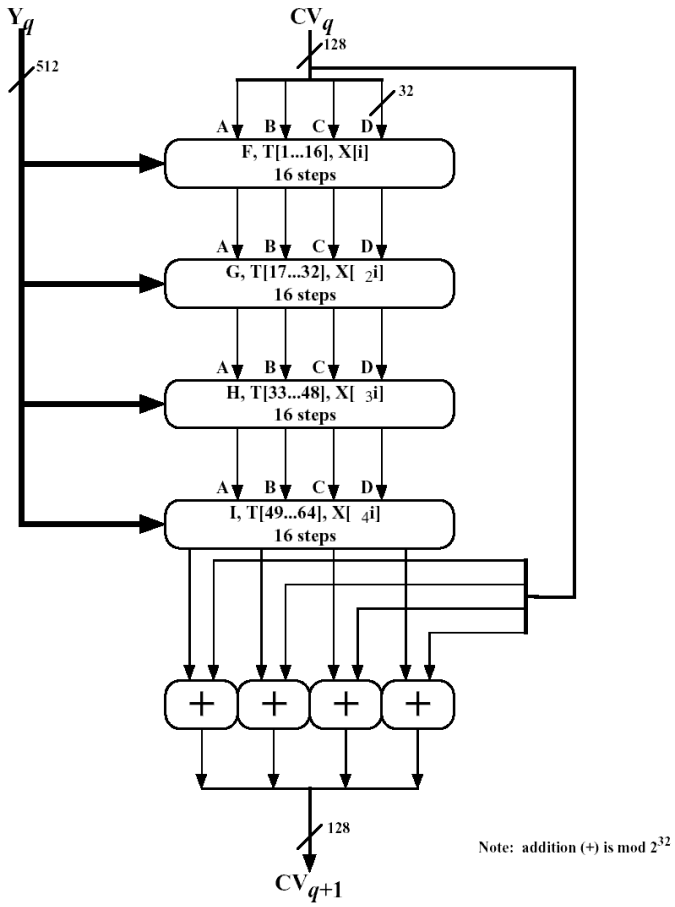
Ο LM hash παρουσιάζει αδυναμίες στην υλοποίηση του. Η βασικότερη αδυναμία είναι ότι χωρίζει σε δύο μέρη τους κωδικούς που έχουν μήκος μεγαλύτερο από επτά χαρακτήρες και παράγει την κάθε σύνοψη ξεχωριστά. Η δεύτερη αδυναμία είναι η μετατροπή των πεζών χαρακτήρων σε κεφαλαίους. Αυτές τις αδυναμίες θα τις εκμεταλλευτούμε αργότερα κατά την παραβίαση των συνόψεων που έχουν δημιουργηθεί από τον αλγόριθμο LM hash. Οι χαρακτήρες ASCII που μπορούν να εκτυπωθούν και χρησιμοποιούνται για την δημιουργία του κλειδιού είναι 95 και επομένως τα κλειδιά που μπορούν να αναπαρασταθούν για κωδικούς μήκους 14 χαρακτήρων είναι 95^{14}. Λόγω της πρώτης αδυναμίας ο αριθμός γίνεται 95^{7} ενώ λόγω της δεύτερης μειώνεται ακόμα περισσότερο καθώς από τους 95 χαρακτήρες αφαιρούμε τα πεζά γράμματα που είναι 26 και επομένως το πλήθος των κωδικών που μπορούν να αναπαρασταθούν γίνεται 69^{7} που είναι περίπου 2^{43}. Άρα η πραγματική ασφάλεια της κρυπτογράφησης είναι 43 bits.

1.4.2 Ο αλγόριθμος MD5

Ο MD5 αναπτύχτηκε από τον Ronald Rivest και δημοσιεύτηκε το 1992. Αποτελεί μια αναβαθμισμένη έκδοση του MD4, έναν παλιότερο αλγόριθμο κρυπτογράφησης. Ο MD5 δέχεται ως είσοδο ένα μήνυμα μήκους από 0 μέχρι 2^{64}-1 bits και παράγει σύνοψη μήκους 128 bits**.** Ο MD5 παρουσιάζει κάποιες αδυναμίες που έχουν ως αποτέλεσμα με κατάλληλες τεχνικές να κάνουμε δύο διαφορετικά μηνύματα να έχουν την ίδια σύνοψη. Για τον λόγω αυτό συνιστάται η χρήση της οικογένειας των αλγορίθμων κρυπτογράφησης SHA [16].

Περιγραφή Αλγορίθμου

Ο αλγόριθμος παίρνει ως είσοδο ένα μήνυμα με μήκος 0 έως 2^{64}-1 bits, έστω L1, στο οποίο προσθέτουμε τον ελάχιστο δυνατό αριθμό bits, έστω L2, ώστε το (L1+L2) mod 512 = 448. Η συμβολοσειρά από bits που προσθέτουμε αποτελείται από ένα bit με τιμή 1 και τα υπόλοιπα bits με τιμή 0. Μετά από αυτή την διαδικασία το τελευταίο τμήμα του μηνύματος έχει μήκος 448 bits και προσθέτουμε ακόμα 64 bits που αναπαριστούν το μήκος σε bits του πραγματικού μηνύματος. Για παράδειγμα αν το αυθεντικό μήνυμα έχει μήκος 552 bits τότε θα προσθέσουμε ακόμα 408 bits καθώς (552+408) mod 512 =448. Επιπλέον θα προσθέσουμε τα 64 bits που αναπαριστούν την τιμή 552 του μήκους του αυθεντικού μηνύματος. Καταλήγουμε λοιπόν να έχουμε 552 + 408 +64 =1024 bits δηλαδή δύο ομάδες των 512 bits.

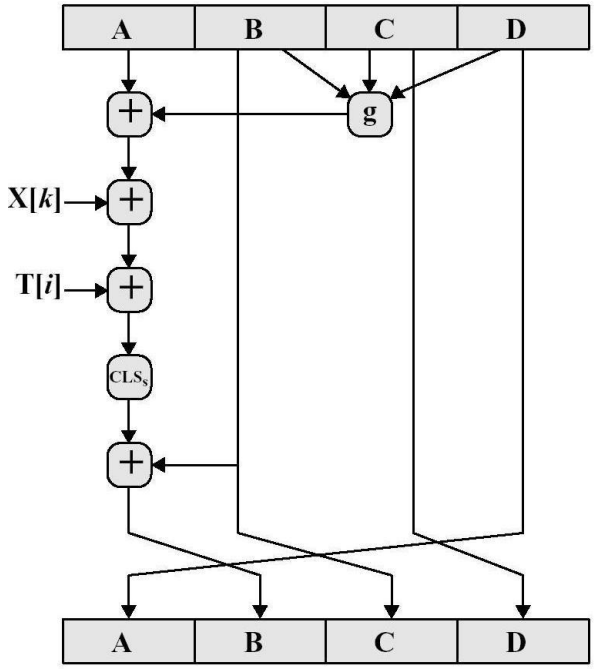


Σχήμα 1.3 *Επεξεργασία ενός μπλοκ 512-bit για τον MD5*

Στην συνέχεια θα επαναλάβουμε την παρακάτω διαδικασία για κάθε μία από τις ομάδες των 512 bits. Οι ενδιάμεσες τιμές καθώς και το αποτέλεσμα της σύνοψης αποθηκεύονται σε 4 καταχωρητές, τους A, B, C και D των 32 bits που ονομάζονται αλυσιδωτές μεταβλητές (chaining variables). Οι τιμές αρχικοποίησης είναι:

A=0x67452301
B= 0xEFCDAB89
C= 0x98BADCFE
D= 0x19325476B

Κάθε τμήμα των 512 bits του μηνύματος εισάγεται στην συνάρτηση συμπίεσης η οποία αποτελείται από τέσσερις γύρους και κάθε γύρος εκτελεί 16 πράξεις. Κάθε πράξη εκτελεί μία μη γραμμική συνάρτηση μεταξύ των A, B, C, D και προσθέτει το αποτέλεσμα στην αλυσιδωτή μεταβλητή B καθώς και τρεις αναθέσεις τιμών στις αλυσιδωτές μεταβλητές όπως φαίνεται και στο σχήμα:



Σχήμα 1.4 *Βασική πράξη γύρου MD5*

Για τις απλές αναθέσεις τιμών είναι

A=D
C=B
D=C

Για την ανάθεση τιμής της αλυσιδωτής μεταβλητής μέσω της μη γραμμικής συνάρτησης ισχύει:

$B= ((A + g_j(B, C, D) + X[k] + T[i]) << s_i) + B) \bmod 2^{32}$, για $1 \leq j \leq 4$

όπου:
Το $g_j( )$ είναι τέσσερις μη γραμμικές συναρτήσεις, όπου σε κάθε γύρο χρησιμοποιούμε μια από αυτές.
$g_1(B,C,D) = (B \cdot C) + ((\neg B) \cdot D)$
$g_2(B,C,D) = (B \cdot D) + (C \cdot (\neg D))$
$g_3(B,C,D) = (B \text{ xor } C \text{ xor } D)$
$g_4(B,C,D) = (C \text{ xor } (B \cdot (\neg D)))$

Το X[j] είναι ένα από τα 16 τμήματα των 32 bit του μπλοκ εισόδου των 512 bit.
Το T[i] κατά το i-στό βήμα, η τιμή που προκύπτει από το ακέραιο τμήμα της απόλυτης τιμής της ποσότητας $\sin(i+1) \times 2^{32}$ με το i να αντιστοιχεί σε ακτίνια.
Το s_i είναι η ποσότητα κυκλικής ολίσθησης του αποτελέσματος.

Στο τέλος των τεσσάρων γύρων στις τελικές τιμές προστίθενται οι αρχικές τιμές των μεταβλητών A, B,C και D.

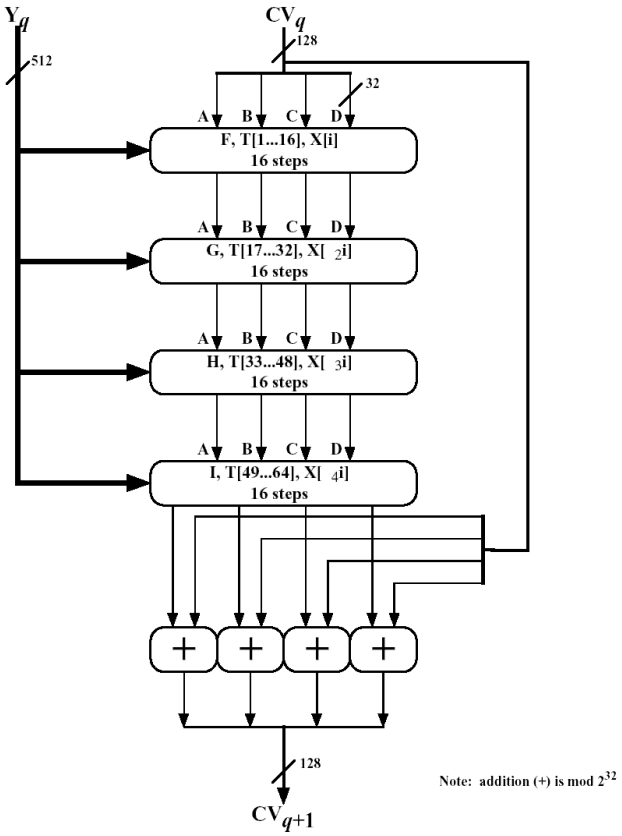
1.4.3 Ο αλγόριθμος SHA-1

Ο SHA-1(Secure Hash Algorithm) υλοποιήθηκε από το National Security Agency των ΗΠΑ και δημοσιεύτηκε το 1995. Ο SHA-1 βασίζονται στον MD4, ένα παλιότερο αλγόριθμο κρυπτογράφησης όπως και ο MD5, όμως οι διαφορές που έγιναν στον SHA-1 είναι πολύ

διαφορετικές από αυτές του MD5. Ο SHA-1 παίρνει ως είσοδο ένα μήνυμα μήκους 0 μέχρι (2^{64}-1) bits και παράγει μια σύνοψη μήκους 160 bits. Αποτελεί το πιο ευρέως χρησιμοποιούμενο μέλος της οικογένειας των κρυπτογραφικών συναρτήσεων SHA [16].

Περιγραφή Αλγορίθμου

Η αρχική διαδικασία είναι ίδια με αυτήν του MD5 δηλαδή έχουμε ως είσοδο ένα μήνυμα με αυθαίρετο μήκος L1, στο οποίο προσθέτουμε τον ελάχιστο δυνατό αριθμό bits, έστω L2, ώστε το (L1+L2) mod 512 = 448. Η συμβολοσειρά από bits αποτελείται από ένα bit με τιμή 1 και τα υπόλοιπα bits με τιμή 0. Μετά από αυτή την διαδικασία το τελευταίο τμήμα του μηνύματος έχει μήκος 448 bits και προσθέτουμε ακόμα 64 bits που αναπαριστούν το μήκος σε bits του πραγματικού μηνύματος. Στην συνέχεια θα επαναλάβουμε την παρακάτω διαδικασία για κάθε μία από τις ομάδες των 512 bits. Λόγω του μεγαλύτερου μήκους της σύνοψης χρησιμοποιούνται πέντε αλυσιδωτές μεταβλητές, έναντι των τεσσάρων που χρησιμοποιούνται στους MD4 και MD5.

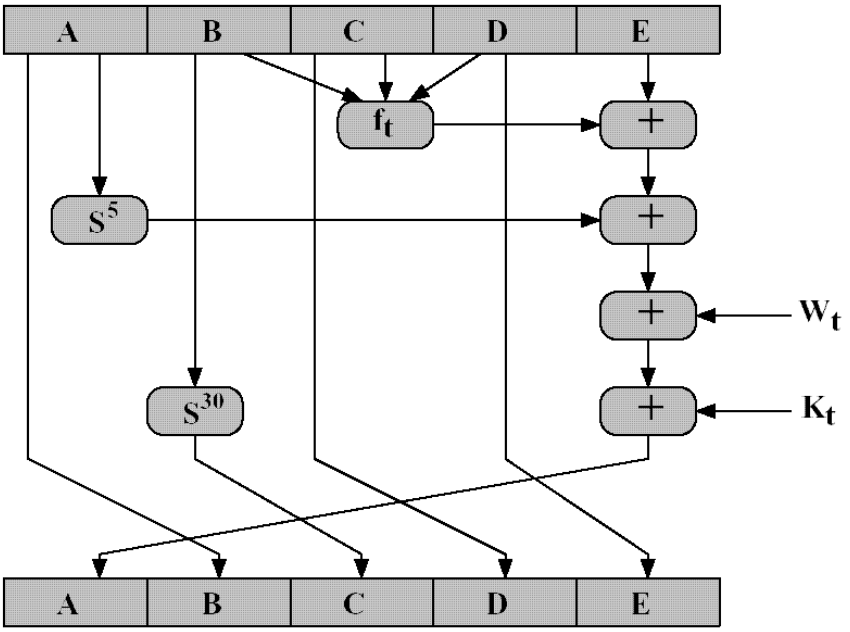


Σχήμα 1.5 *Επεξεργασία ενός μπλοκ 512-bit για τον SHA-1*

Οι μεταβλητές αυτές είναι οι A, B, C, D και E και οι αρχικές τιμές των τεσσάρων πρώτων μεταβλητών είναι όμοιες με αυτές της MD5, ενώ η πέμπτη μεταβλητή έχει την τιμή:
E = 0xC3D2E1F0

Κάθε τμήμα των 512 bits του μηνύματος εισάγεται στην συνάρτηση συμπίεσης η οποία αποτελείται από τέσσερις γύρους και κάθε γύρος εκτελεί 16 πράξεις. Κάθε πράξη εκτελεί μία μη γραμμική συνάρτηση μεταξύ των A,B,C,D και προσθέτει το αποτέλεσμα στην αλυσιδωτή μεταβλητή B καθώς και τρείς αναθέσεις τιμών στις αλυσιδωτές μεταβλητές όπως φαίνεται και στο σχήμα

Παρόμοια με την MD5, το κάθε τμήμα του μηνύματος εισάγεται στην συνάρτηση συμπίεσης η οποία αποτελείται από τέσσερις γύρους και κάθε γύρος εκτελεί 20 πράξεις. Κάθε πράξη εκτελεί μια μη γραμμική συνάρτηση μεταξύ των A, C, D, E και προσθέτει το αποτέλεσμα στην αλυσιδωτή μεταβλητή A καθώς και τέσσερις αναθέσεις τιμών στις αλυσιδωτές μεταβλητές όπως φαίνεται και στο σχήμα:



Σχήμα 1.6 *Βασική πράξη γύρου SHA-1*

Για την ανάθεση τιμής της αλυσιδωτής μεταβλητής μέσω της μη γραμμικής συνάρτησης ισχύει:

$A=(A<<5) + f_j + E + K_j + W[t]$, για $1 \le j \le 4$
Όπου
$f_1=(B \cdot C + B \cdot D)$
$f_2=(B + C + D)$
$f_3=(B \cdot C + B \cdot D + C \cdot D)$
$f_4=(B + C + D)$

Το W[t] είναι η επέκταση των 512 bits εισόδου w[] σε 80 δυαδικές λέξεις των 32 bit που έχουν επεκταθεί από την συνάρτηση
W[t] = (w[t-3] xor w[t-8] xor w[t-14] xor w[t-16]) << 1.
Το K_j είναι σταθερές τιμές

K_1 = 0x5A827999
K_2 = 0x6ED9EBA1
K_3 = 0x8F1BBCDC
K_4 = 0xCA62ClD6

2. Κρυπτογραφική Ανάλυση

2.1 Γενικά για την κρυπτογραφική ανάλυση

Από το προηγούμενο κεφάλαιο είδαμε ότι η κρυπτογραφία κωδικοποιεί δεδομένα ενώ αντίθετα η *κρυπτογραφική ανάλυση (cryptanalysis)* προσπαθεί να παραβιάσει κρυπτογραφημένα δεδομένα χωρίς την γνώση των κλειδιών που απαιτούνται για την αποκρυπτογράφηση. Στο κεφάλαιο αυτό θα περιγράψουμε τα πλεονέκτημα και τα μειονεκτήματα της επίθεσης με χρήση λεξικού που προσπαθεί να παραβιάσει δεδομένα χρησιμοποιώντας κωδικούς που έχουν μεγάλη πιθανότητα επιτυχίας, την επίθεση με χρήση ωμής βίας που προσπαθεί να δοκιμάσει όλα τα δυνατά κλειδιά και τέλος την μέθοδο με χρήση πινάκων ουράνιου τόξου, την οποία και επέλεξα. Να σημειώσω ότι δύο πρώτες τεχνικές χρησιμοποιούνται γενικά για κρυπτογραφικά δεδομένα, όμως η χρήση των πινάκων ουράνιου τόξου (rainbow tables) περιορίζεται για κρυπτογραφικές συναρτήσεις κατακερματισμού.

2.2 Μέθοδοι κρυπτογραφικής ανάλυσης

2.2.1 Επίθεση με χρήση λεξικού (dictionary attack)

Είναι μια τεχνική κρυπτογραφικής ανάλυσης η οποία δοκιμάζει κωδικούς που είναι πολύ πιθανό να έχουν επιλεγεί από τους χρήστες. Για τους περισσότερους χρήστες ένας κωδικός σαν τον “h2dwp#wxkm7dms” δεν θα αποτελούσε την πρώτη τους επιλογή, καθώς οι χρήστες έχουν την τάση να επιλέγουν κωδικούς που είναι εύκολοι στην απομνημόνευση. Συνήθως οι κωδικοί που επιλέγονται από τους χρήστες έχουν μικρό μήκος, μέχρι επτά χαρακτήρες, είναι λέξεις που μπορούν να βρεθούν σε λεξικά (εδώ οφείλεται η ονομασία της συγκεκριμένης μεθόδου) και ονόματα ανθρώπων. Επιπλέον χρησιμοποιούν και κάποιες μικρές παραλλαγές από τα παραπάνω όπως η προσάρτηση ενός αριθμού στο τέλος μιας λέξης, η χρησιμοποίηση λέξεων γραμμένων ανάποδα, η αντικατάσταση κάποιων πεζών γραμμάτων με τα αντίστοιχα κεφαλαία γράμματα και τέλος η αντικατάσταση συγκεκριμένων γραμμάτων με αριθμούς ή σύμβολα όπως το μηδέν ‘0’ αντί για το ‘o’ και το ‘$’ αντί για το ‘s’[17][18]. Χρησιμοποιώντας την τελευταία παραλλαγή ο κωδικός “password” θα μπορούσε να είχε μετατραπεί στον “pa$$w0rd”. Επομένως η κατασκευή ενός λεξικού (λίστας λέξεων) που περιέχει όλους τους συνδυασμούς των παραπάνω περιπτώσεων και η χρήση του θα ήταν ιδιαίτερα αποδοτική στην παραβίαση κωδικών ενώ ο μικρός αριθμός κωδικών κάνει αυτή την μέθοδο ιδιαίτερα γρήγορη. Στα μειονεκτήματα αυτής της μεθόδου είναι η αναποτελεσματικότητα εναντίων συστημάτων που η επιλογή του κωδικού έχει γίνει έχοντας λάβει υπόψη τα παραπάνω.

2.2.2 Επίθεση με χρήση ωμής βίας (brutal force attack)

Είναι μια τεχνική κρυπτογραφικής ανάλυσης η οποία αναφέρατε στην εξαντλητική δοκιμή όλως των πιθανών κωδικών πρόσβασης. Αυτή η τεχνική είναι πολύ απλή στην υλοποίηση της και θεωρητικά δίνει πάντα λύση για οποιοδήποτε μήκος κωδικού, πρακτικά όμως μπορεί να χρησιμοποιηθεί μόνο σε συστήματα που χρησιμοποιούν κωδικούς με μικρό μήκος ώστε το πλήθος των πιθανών κωδικών να είναι μικρό και ο χρόνος που απαιτείται για την εξαντλητική

δοκιμή να είναι ανεκτός. Συνήθως η τεχνική αυτή χρησιμοποιείται αφού έχει αποτύχει η επίθεση με χρήση λεξικού[19][20].

2.2.3 Επίθεση με πίνακες ουράνιου τόξου

Οι δύο προηγούμενες μέθοδοι αν και έχουν κάποια πλεονεκτήματα δεν είναι επαρκείς καθώς η επίθεση με χρήση λεξικού αποτυγχάνει για κωδικούς πρόσβασης που έχουν επιλεγεί προσεκτικά ενώ η επίθεση με χρήση ωμής βίας δεν μπορεί να χρησιμοποιηθεί για κωδικούς με μεγάλο μήκος. Γι αυτό θα πρέπει να αναζητήσουμε κάποια καλύτερη μέθοδο επίθεσης. Μια διαφορετική προσέγγιση που θα μπορούσαμε να χρησιμοποιήσουμε για να βελτιώσουμε την ταχύτητα της επίθεσης με χρήση βίας είναι να έχουμε προϋπολογίσει όλες τις συνόψεις για όλους τους δυνατούς συνδυασμούς κλειδιών και να έχουμε αποθηκεύσει όλα τα ζευγάρια κλειδιού-σύνοψης. Κατά την στιγμή που θα θέλαμε να παραβιάσουμε κάποιο κλειδί θα αναζητούσαμε το ζευγάρι με τον κωδικό και την σύνοψη και δεν θα χρειαζόμασταν να προχωρήσουμε σε καμία δοκιμή κωδικού. Με αυτόν τον τρόπο η παραγωγή των συνόψεων απαιτεί τον ίδιο χρόνο όπως και με την επίθεση με χρήση ωμής βίας αλλά έχουμε το πλεονέκτημα ότι τα αποθηκευμένα ζευγάρια θα μπορούσαν να χρησιμοποιηθούν για να παραβιαστούν περισσότεροι από ένας κωδικός χωρίς να χρειάζεται να τα ξαναδημιουργήσουμε. Εκ πρώτης άποψης η λύση αυτή φαίνεται ιδιαίτερα αποδοτική, όμως έχει ένα μεγάλο μειονέκτημα που την κάνει να μην μπορεί να χρησιμοποιηθεί στην πράξη και αυτό δεν είναι άλλο από τις τεράστιες απαιτήσεις σε χώρο που απαιτούνται για να αποθηκευτούν όλα τα ζευγάρια από τα κλειδιά και τις συνόψεις. Ακόμα και για κωδικούς που χρησιμοποιούν μόνο γράμματα του λατινικού αλφαβήτου και αριθμούς με μήκος έξι χαρακτήρων, οι απαιτήσεις για να αποθηκεύσουμε όλα τα ζευγάρια κωδικού-σύνοψης για τον αλγόριθμο MD5 είναι περίπου 74 TBytes*. Για να μπορεί να χρησιμοποιηθεί στην πράξη μια μέθοδος που χρησιμοποιεί προϋπολογισμένα δεδομένα θα πρέπει να απαιτεί σχετικά μικρό χώρο αποθήκευσης. Αυτό είναι εφικτό με μια μέθοδο που χρησιμοποιεί πίνακες ουράνιου τόξου.

Τι είναι οι πίνακες ουράνιου τόξου?

Οι *πίνακες ουράνιου τόξου (rainbow tables)* είναι πίνακες αναζήτησης (lookup tables) που επιτρέπουν την χρήση μιας τεχνικής που ονομάζεται space-time tradeoff (ανταλλαγή χώρου-χρόνου). Η ιδέα για χρήση της ανταλλαγής χώρου-χρόνου ανήκει στον Martin Hellman που το 1980 χρησιμοποίησε προϋπολογισμένες συνόψεις για να παραβιάζει κωδικούς πρόσβασης [1]. Οι πίνακες ουράνιου τόξου ονομάστηκαν το 2003 από τον Philippe Oeshlin που βελτίωσε την αρχική δουλειά του Hellman [2] αυξάνοντας την πιθανότητα παραβίασης ενός κωδικού για δεδομένο αποθηκευτικό χώρο. Με τους πίνακες ουράνιου τόξου είναι εφικτή η παραβίαση κωδικών πιο αποτελεσματικά από την επίθεση με χρήση ωμής βίας, γνωρίζοντας μόνο την σύνοψη που αυτός δημιουργεί. Επιπλέον καθώς η τεχνική αυτή είναι γενική μπορεί να χρησιμοποιηθεί από οποιονδήποτε κρυπτογραφικό αλγόριθμο κατακερματισμού.

Γενικά με τον όρο ανταλλαγή χώρου-χρόνου αναφερόμαστε σε είναι μια κατάσταση όπου η χρήση μνήμης μπορεί να μειωθεί με κόστος την αύξηση του χρόνου εκτέλεσης ή αντίστροφα να αυξηθεί η χρήση μνήμης με μείωση του χρόνου εκτέλεσης (υπολογισμού). Αυτό είναι εφικτό αν με κατάλληλες επιλογές μπορεί να αλλάξει η πολυπλοκότητα του προβλήματος [21]. Στην περίπτωση των πινάκων ουράνιου τόξου μας χρησιμοποιούμε την τεχνική αυτή για να μειώσουμε την χρήση μνήμης με αντάλλαγμα την αύξηση του χρόνου υπολογισμού.

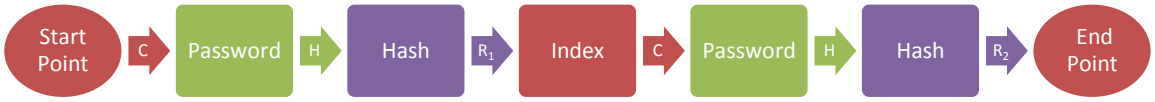
*Τα γράμματα είναι 52 και οι αριθμοί 10 επομένως έχουμε 62 χαρακτήρες που απαιτούν 6 bits για την αναπαράσταση τους αφού 2^6=64 .Για κωδικούς με μήκος 7 χαρακτήρων οι δυνατοί συνδυασμοί είναι 62^7. Κάθε κωδικός αφού έχει μήκος 7 χαρακτήρων χρειάζεται 7*6=42 bits για την αποθήκευση του. Επιπλέον κάθε σύνοψη του MD5 έχει μήκος 128 bits. Επομένως ο συνολικός χώρος που απαιτείται είναι 62^7 * (42+128) bits = 3.521.614.606.208 * 170 bits =74.834.310.381.920 bytes που είναι περίπου 74.000 GB ή 74 TB.

Πώς δουλεύουν οι πινάκες ουράνιου τόξου;

Αρχικά θα προσπαθήσω να εξηγήσω την βασική δομή της κατασκευής και της χρήσης των πινάκων ουράνιου τόξου. Στη συνέχεια ακολουθεί ένα παράδειγμα που θα βοηθήσει στην πλήρη κατανόηση τού πώς δουλεύουν οι πίνακες. Το βασικότερο θέμα για την κατανόηση των πινάκων είναι η διαφορά μεταξύ κατασκευής και χρήσης τους. Η κατασκευή γίνεται μονάχα μια φορά και αφού ολοκληρωθεί, οι πίνακες μπορούν να χρησιμοποιηθούν για να παραβιάσουμε όσους κωδικούς πρόσβασης επιθυμούμε.

Κατασκευή πινάκων

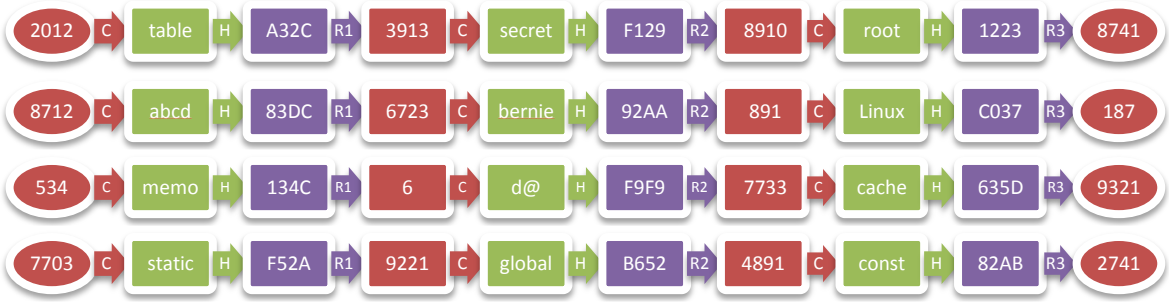
Ένας πίνακας ουράνιου τόξου κατασκευάζεται από αλυσίδες (ακολουθίες) στις οποίες κρυπτογραφούνται κωδικοί πρόσβασης.



Σχήμα 2.1 *Σχηματική αναπαράσταση αλυσίδας με δύο κρίκους*

Κάθε αλυσίδα αρχίζει με ένα αριθμό που ονομάζεται αρχικό σημείο. Το αρχικό σημείο μετατρέπεται σε έναν επιτρεπτό κωδικό με μια συνάρτηση που ονομάζεται συνάρτηση μετατροπής. Στην συνέχεια από τον κωδικός παράγεται η σύνοψη με μία κρυπτογραφική συνάρτηση κατακερματισμού. Το τρίτο βήμα είναι η μετατροπή της σύνοψης σε έναν νέο αριθμό με μια συνάρτηση ανακατασκευής (reduction function). Η διαδικασία αυτή επαναλαμβάνεται για σταθερό αριθμό βημάτων για κάθε μία από τις αλυσίδες. Να σημειώσω ότι σε αυτή την εργασία χρησιμοποιείται η βελτιωμένη υλοποίηση του Philippe Oechslin στην οποία γίνεται χρήση διαφορετικών συναρτήσεων ανακατασκευής σε κάθε κρίκο.

Η κατασκευή του πίνακα ουράνιου τόξου γίνεται απορρίπτοντας όλες τις συνόψεις και τους ενδιάμεσους κωδικούς πρόσβασης και αποθηκεύουμε μόνο τα αρχικά και τα τελικά σημεία κάθε αλυσίδας. Αυτά τα ζευγάρια αρχικών και τελικών σημείων αποτελούν τον πίνακα ουράνιου τόξου.



ΠΙΝΑΚΑΣ ΟΥΡΑΝΙΟΥ ΤΟΞΟΥ	
Αρχικά Σημεία	**Τελικά Σημεία**
2012	**8741**
8712	**187**
534	**9321**
7703	**2741**

Σχήμα 2.2 *Δημιουργία πίνακα ουράνιου τόξου με τα αρχικά και τελικά σημεία όλων των αλυσίδων.*

Χρήση πινάκων

Έχοντας κατασκευάσει τους πίνακες μπορούμε να τους χρησιμοποιήσουμε επανειλημμένα για την παραβίαση οποιουδήποτε αριθμού συνόψεων. Η παραβίαση για μία συγκεκριμένη σύνοψη γίνεται με τον εντοπισμό της στον κρίκο οποιασδήποτε αλυσίδας από αυτές που χρησιμοποιήσαμε για την κατασκευή των πινάκων. Καθώς ο πίνακας δεν περιέχει τις ενδιάμεσες τιμές παρά μόνο τις αρχικές και τις τελικές τιμές κάθε αλυσίδας, θα πρέπει να χρησιμοποιήσουμε μια ειδική τεχνική για τον έλεγχο των αλυσίδων. Η τεχνική βασίζεται στην κατασκευή μιας αλυσίδα της οποίας το μήκος αυξάνεται συνεχώς μέχρι την εύρεση της σύνοψης. Σαν είσοδο στην αλυσίδα αυτή θα χρησιμοποιηθεί η έξοδος της συνάρτησης ανακατασκευής που επενεργεί στην σύνοψη ενώ το μέγιστο μήκος που μπορεί να επεκταθεί είναι το μήκος των αλυσίδων που χρησιμοποιήθηκαν για την κατασκευή του πίνακα. Μετά από κάθε επέκταση της αλυσίδας ελέγχουμε αν η τιμή του τελευταίου κρίκου υπάρχει και στον πίνακα ουράνιου τόξου. Αν η τιμή βρεθεί στον πίνακα τότε μπορούμε να παραβιάσουμε την σύνοψη και σταματάμε την διαδικασία επέκτασης της αλυσίδας. Αντίθετα αν φτάσουμε στο μέγιστο μήκος χωρίς να έχουμε εντοπίσει την τιμή σημαίνει ότι η συγκεκριμένη σύνοψη δεν είχε δημιουργηθεί κατά την κατασκευή του πίνακα και επομένως δεν μπορεί να παραβιαστεί. Για την παραβίαση της σύνοψης απαιτείται η χρήση κάποιου άλλου πίνακα του οποίου οι αλυσίδες περιέχουν την συγκεκριμένη σύνοψη.

2.3 Παράδειγμα κατασκευής και χρήσης πινάκων ουράνιου τόξου

Για την πλήρη κατανόηση των πινάκων θα χρησιμοποιήσω ένα απλουστευμένο παράδειγμα κατασκευής και χρήσης πινάκων ουράνιου τόξου. Σε αυτό το παράδειγμα ο κωδικός αποτελείται μόνο από πεζά γράμματα και αριθμούς με επιτρεπτό μήκος ενός χαρακτήρα. Ο κρυπτογραφικός αλγόριθμος κατακερματισμού H() παίρνει ως είσοδο κωδικούς μήκους ενός χαρακτήρα και παράγει σύνοψη μήκους δύο χαρακτήρων. Ο τρόπος που την παράγει είναι ο εξής: Αρχικά δημιουργεί μια συμβολοσειρά με μήκος δύο χαρακτήρες αποτελούμενη από τους χαρακτήρες εισόδου. Στην συνέχεια ο δεύτερος χαρακτήρας αντικαθίσταται από τον χαρακτήρα που βρίσκεται τρεις θέσεις μετά σύμφωνα με την αλφαβητική σειρά. Για παράδειγμα ο χαρακτήρας ‘b’ αρχικά επεκτείνεται στην συμβολοσειρά ‘bb’ και στην συνέχεια το δεύτερο ‘b’ αντικαθίσταται από το ‘e’ άρα η τελική συμβολοσειρά γίνεται ‘be’. Αν ο χαρακτήρας που πρέπει να αντικατασταθεί είναι εκτός ορίων του αλφάβητου τότε χρησιμοποιούμε νούμερα, ενώ αν είναι και εκτός ορίων των αριθμών ξεκινάμε από την αρχή του αλφάβητου. Για παράδειγμα το ‘z’ θα γίνει ‘z2’ ενώ το ‘9’ θα γίνει ‘9c’. Ακολουθεί ο πίνακας με όλους τους πιθανούς κωδικούς και τις αντίστοιχες συνόψεις που παράγονται με τον H().

Plaintext	Ciphertext	Plaintext	Ciphertext	Plaintext	Ciphertext	Plaintext	Ciphertext
a	ad	J	jm	s	sv	1	14
b	be	K	kn	t	tw	2	25
c	cf	L	lo	u	ux	3	36
d	dg	M	mp	v	vy	4	47
e	eh	N	nq	w	wz	5	58
f	fi	O	or	x	x0	6	69
g	gj	P	ps	y	y1	7	7a
h	hk	Q	qt	z	z2	8	8b
i	il	R	ru	0	03	9	9c

Πίνακας 3.1

Η συνάρτηση ανακατασκευής $R_i()$, όπως έχουμε αναφέρει και νωρίτερα, αντιστοιχεί συνόψεις σε κωδικούς. Ο απλούστερος τρόπος για να γίνει αυτό είναι η μετατροπή των συνόψεων σε μια αριθμητική τιμή και μετά να βρούμε το υπόλοιπο της αριθμητικής τιμής με το

πλήθος όλων των δυνατών κωδικών. Στο παράδειγμά μας το πλήθος των δυνατών κωδικών είναι 36 καθώς έχουμε 26 γράμματα και 10 αριθμούς και επιτρέπεται το μήκος των κωδικών να είναι ένα. Επιπλέον καθώς χρησιμοποιούμε την υλοποίηση του Philippe Oechslin κάθε κρίκος(θέση) τις αλυσίδας θα πρέπει να χρησιμοποιεί μια διαφορετική συνάρτηση ανακατασκευής. Αυτό θα γίνει προσθέτοντας την θέση του κρίκου στην αριθμητική τιμή της σύνοψης. Κατά σύμβαση για την πρώτο κρίκο θα προσθέσουμε τον αριθμό 1, για τον δεύτερο κρίκο τον αριθμό 2 κτλ. Η μετατροπή των συνόψεων σε αριθμητική τιμή γίνεται θεωρώντας ότι οι χαρακτήρες παίρνουν τιμές με βάση την θέση τους όπως στο δεκαδικό σύστημα, μόνο που σε αυτό το παράδειγμα η βάση δεν είναι το 10, αλλά ο αριθμός 36. Ο αριθμός 36 προκύπτει από το πλήθος όλων των δυνατών κωδικών πρόσβασης που μπορούν να παραχθούν. Σε κάθε χαρακτήρα αντιστοιχούμε μια τιμή ξεκινώντας από το 'a' που παίρνει την τιμή 0, το 'b' την τιμή 1 και με την ίδια λογική ο χαρακτήρας '9' έχει την τιμή 35. Στον παρακάτω πίνακα φαίνονται οι αριθμητικές τιμές όλων των επιτρεπτών χαρακτήρων.

Χαρακτήρας	Τιμή	Χαρακτήρας	Τιμή	Χαρακτήρας	Τιμή	Χαρακτήρας	Τιμή
a	0	J	9	s	18	1	27
b	1	K	10	t	19	2	28
c	2	L	11	u	20	3	29
d	3	M	12	v	21	4	30
e	4	N	13	w	22	5	31
f	5	O	14	x	23	6	32
g	6	P	15	y	24	7	33
h	7	Q	16	z	25	8	34
i	8	R	17	0	26	9	35

Πίνακας 3.2

Κατασκευή πινάκων.

Όπως ήδη έχουμε αναφέρει για να χρησιμοποιήσουμε τους πίνακες για να παραβιάσουμε κωδικούς πρέπει πρώτα να τους κατασκευάσουμε. Στο παράδειγμα αυτό θα κατασκευάσουμε έναν πίνακα από πέντε αλυσίδες των έντεκα κρίκων και επομένως χρειαζόμαστε τέσσερις τυχαίους κωδικούς που κάθε ένας θα χρησιμοποιηθεί σαν είσοδος σε μία αλυσίδα. Την είσοδο σε κάθε αλυσίδα θα την ονομάσουμε αρχικό σημείο. Σε κάθε αλυσίδα ο κωδικός θα κρυπτογραφηθεί και στην συνέχεια η σύνοψη θα χρησιμοποιηθεί σαν είσοδος στην συνάρτηση ανακατασκευής. Κάθε κωδικό που προκύπτει από τις συναρτήσεις ανακατασκευής θα τον ονομάσουμε ενδιάμεσο σημείο. Το τελευταίο ενδιάμεσο σημείο θα το ονομάσουμε τελικό σημείο. Υπάρχουν τόσα ενδιάμεσα σημεία όσο και ο αριθμός των κρίκων. Τα δεδομένα στον παρακάτω πινάκα είναι τα αρχικά σημεία, τα ενδιάμεσα σημεία, και τα τελικά σημεία για όλες τις αλυσίδες.

ΑΣ	X1	X2	X3	X4	X5	X6	X7	X8	X9	X10	ΤΣ=X11
a	e	j	p	W	4	d	N	y	a	n	1
d	h	m	s	Z	7	g	Q	1	d	q	4
l	p	u	0	7	f	o	Y	9	1	y	C
4	8	d	j	Q	y	7	H	s	4	h	V
5	9	e	k	R	z	8	I	t	5	i	W

Πίνακας 3.2

Υπενθυμίζουμε ότι ο κωδικός του αρχικού σημείου έχει επιλεγεί τυχαία αλλά με κατάλληλο τρόπο ώστε να είναι έγκυρος. Για να δημιουργηθεί η τιμή του ενδιάμεσου σημείου X1 το αρχικό σημείο πρέπει να κρυπτογραφηθεί και να εφαρμοστεί η συνάρτηση

ανακατασκευής. Συγκεκριμένα το αρχικό σημείο ‘a’ με την κρυπτογράφηση θα γίνει ‘ad’. Η σύνοψη θα μεταφραστεί στην αριθμητική της τιμή με σύστημα αρίθμησης που χρησιμοποιεί σαν βάση το 36 άρα το ‘ad’ αντιστοιχεί στην τιμή 36*’a’+’d’=36*0+3=3. Στην συνέχεια θα εφαρμόσουμε την συνάρτηση ανακατασκευής στην αριθμητική τιμή της σύνοψης, αφού προσθέσουμε τον αριθμό της στήλης. Ακολούθως για να εξασφαλίσουμε ότι το αποτέλεσμα είναι στο εύρος 0-35 θα βρούμε το υπόλοιπο και μετά γίνεται άμεσα η αντιστοίχιση με έναν κωδικό. Οι επόμενοι πίνακες δείχνουν τα αρχικά σημεία, τις συνόψεις, τις αριθμητικές τιμές των συνόψεων, το αποτέλεσμα των συναρτήσεων ανακατασκευής και την θέση μέσα στην αλυσίδα μέχρι το δεύτερο ενδιάμεσο σημείο.

ΑΣ	**Cipher**	**Value**	**R_0**	**X1**	**Cipher**	**Value**	**R_1**	**X2**
a	ad	3	4	e	eh	151	9	J
d	dg	114	7	h	hk	262	12	M
l	lo	410	15	p	ps	558	20	U
4	47	1113	34	8	8b	1125	3	D
5	58	1150	35	9	9c	1262	4	E

Πίνακας 3.3

Από τον πίνακα 3.3 βλέπουμε ότι για το αρχικό σημείο ‘a’ η σύνοψη του είναι ‘ad’ που αντιστοιχεί στην αριθμητική τιμή 3. Καθώς σε αυτήν την τιμή προσθέτουμε τον αριθμό της στήλης παίρνουμε την τιμή 4 και καθώς (4 mod 36)=4 προκύπτει το αποτέλεσμα της συνάρτησης ανακατασκευής. Ο αριθμός 4 από τον πίνακα 3.2 αντιστοιχεί στον χαρακτήρα ‘e’. Αυτό είναι το πρώτο ενδιάμεσο σημείο που το έχουμε ονομάσει X1. Ο κωδικός ‘e’ στο X1 κρυπτογραφείται και δίνει την σύνοψη ‘eh’. Αυτή αντιστοιχεί στην αριθμητική τιμή 36*’e’+’h’=36*4+7=151. Προσθέτουμε τον αριθμό της στήλης που είναι 2 και παίρνουμε την τιμή 153 και καθώς (153 mod 36)=9 προκύπτει το αποτέλεσμα της συνάρτησης ανακατασκευής. Ο αριθμός 9 από τον πίνακα 3.2 αντιστοιχεί στον χαρακτήρα ‘j’. Αυτό είναι το δεύτερο ενδιάμεσο σημείο που έχουμε ονομάσει X2.Η παραπάνω διαδικασία θα επαναληφθεί μέχρι να ολοκληρώσουμε όλους του κρίκους της αλυσίδας και να φτάσουμε στο τελικό σημείο. Από κάθε αλυσίδα θα αποθηκεύσουμε μόνο το αρχικό και το τελικό σημείο και τελικά θα πάρουμε τον ακόλουθο πίνακα που είναι ο πίνακας ουράνιου τόξου.

ΑΣ	**ΤΣ**
a	1
d	4
l	c
4	v
5	w

Πίνακας 3.4

Χρήση πινάκων για Κρυπτογραφική Ανάλυση

Αφού έχουμε ολοκληρώσει την κατασκευή των πινάκων μπορούμε να τους χρησιμοποιήσουμε για να παραβιάσουμε έναν αυθαίρετο αριθμό από συνόψεις. Η αποθήκευση του παραπάνω πίνακα απαιτεί μόνο 2 bytes για κάθε αλυσίδα, άρα συνολικά 10 bytes, ενώ αν είχαμε αποθηκεύσει ολόκληρο τον πίνακα αναζήτησης θα απαιτούσε κάθε αλυσίδα 33 bytes καθώς έχουμε 11 ζευγάρια κωδικού-σύνοψης άρα συνολικά 33*5= 165 bytes. Παρατηρούμε ότι η διαφορά σε απαιτήσεις χώρου στις δύο περιπτώσεις είναι μεγάλη και η διαφορά αυξάνεται όσο περισσότερους κρίκους έχουν οι αλυσίδες.

Τώρα θα χρησιμοποιήσουμε τον πίνακα ουράνιου τόξου για να παραβιάσουμε την σύνοψη ‘hk’. Το πρώτο βήμα είναι να δούμε αν το αποτέλεσμα τις συνάρτησης ανακατασκευής ταιριάζει με κάποια τιμή από την δεύτερη στήλη του πίνακα του ουράνιου τόξου, δηλαδή το

τελικό σημείο κάποιας αλυσίδας κατά την κατασκευή του πίνακα. Η αριθμητική τιμή της σύνοψης είναι 36*'h'+'k'=36*7+10=262 και με την εφαρμογή της συνάρτησης ανακατασκευής R_{10} καθώς αυτήν την συνάρτηση χρησιμοποιήσαμε για να πάρουμε το τελικό σημείο κάθε αλυσίδας. Άρα έχουμε (262+11)mod 36=21 . Η τιμή 21 αντιστοιχεί στον χαρακτήρα 'v'. Ο χαρακτήρας αυτός υπάρχει στην δεύτερη στήλη του πίνακα, άρα ο 'v' δημιούργησε την σύνοψη 'hk' και επομένως η παραβίαση ήταν επιτυχημένη.

Τώρα θα χρησιμοποιήσουμε τον πίνακα ουράνιου τόξου για να παραβιάσουμε την σύνοψη '47'. Η αριθμητική τιμή της σύνοψης είναι '4'*36+'7'=30*36+33=1113 και με την εφαρμογή της συνάρτησης ανακατασκευής R_{10} παίρνουμε (1113+11) mod 36=8. Η τιμή 8 αντιστοιχεί στον χαρακτήρα 'i'. Ο χαρακτήρας αυτός δεν υπάρχει στην τελευταία στήλη άρα πρέπει να συνεχίσουμε την διαδικασία. Αυτή την φορά θα εφαρμόσουμε την σύνοψη χρησιμοποιώντας αρχικά την συνάρτηση ανακατασκευής R_9 όπου θα πάρουμε (1113+10) mod 36=7. Η τιμή αυτή αντιστοιχεί στον 'h' και θα την κρυπτογραφήσουμε οπότε θα δώσει αποτέλεσμα 'hk' . Η αριθμητική τιμή της σύνοψης είναι 'h'*36+'k'=36*7+10=262. Σε αυτή την σύνοψη θα εφαρμόσουμε την R_{10} όπου θα πάρουμε (262 +11) mod 36=21. Η τιμή 21 αντιστοιχεί στον χαρακτήρα 'v' που υπάρχει στην δεύτερη στήλη του πίνακα και επομένως γνωρίζουμε ότι υπάρχει ο κωδικός που δημιούργησε την σύνοψη. Για να βρούμε όμως σε πιο χαρακτήρα αντιστοιχεί το '47' θα πρέπει να δημιουργήσουμε την συγκεκριμένης αλυσίδας χρησιμοποιώντας σαν αρχικό σημείο την πρώτη στήλη του πίνακα 3.4 και θα δημιουργήσουμε την αλυσίδα όπως κάναμε κατά την κατασκευή του πίνακα μέχρι το σημείο που από την κρυπτογράφηση θα προκύψει η σύνοψη '47', δηλαδή μέχρι και το ενδιάμεσο σημείο X8.

Αν κατά την παραβίαση μιας σύνοψης δεν βρίσκαμε αποτέλεσμα ενώ είχαμε εξαντλήσει την παραπάνω διαδικασία για όλες τις συναρτήσεις ανακατασκευής, η σύνοψη που ψάχνουμε δεν είχε κρυπτογραφηθεί κατά την κατασκευή του πίνακα και επομένως είναι αδύνατη η παραβίαση από τον συγκεκριμένο πίνακα.

Παρατηρήσεις

Τώρα που έχει περιγραφτεί πώς δουλεύουν οι πίνακες μπορούμε να κάνουμε κάποιες παρατηρήσεις. Παρόλο που χρειάζεται πολύς χρόνος για να κατασκευαστούν οι πίνακες η παραβίαση γίνεται πολύ πιο αποτελεσματικά. Θα πρέπει να σημειώσουμε ότι κάθε πίνακας είναι φτιαγμένος για συγκεκριμένη κρυπτογραφική συνάρτηση κατακερματισμού για συγκεκριμένο μήκος κωδικού και συγκεκριμένους χαρακτήρες. Κωδικοί που δεν ταιριάζουν σε αυτά τα χαρακτηριστικά δεν μπορούν να παραβιαστούν από τον συγκεκριμένο πίνακα.

Όταν κατά τον υπολογισμό των πινάκων δύο αλυσίδες περιέχουν τμήματα με ίδιες τιμές και έχει ως αποτέλεσμα να δώσουν ίδια τελικά σημεία, τότε ονομάζουμε αυτή την κατάσταση collosion και πρέπει να αποφεύγεται καθώς ο πίνακας περιέχει επαναλαμβανόμενες τιμές. Χωρίς την χρήση της μεθόδου του Philippe Oeshlin αυτό μπορεί να συμβεί όταν οι συναρτήσεις ανακατασκευής δύο αλυσίδων επιστρέψουν την ίδια τιμή. Αντίθετα με την χρήση της μεθόδου του Philippe Oeshlin οι πιθανότητες να συμβεί είναι πολύ μικρότερες καθώς χρησιμοποιεί διαφορετικές συναρτήσεις ανακατασκευής σε κάθε στήλη και επομένως για να συμβεί θα πρέπει οι συναρτήσεις ανακατασκευής δύο αλυσίδων να επιστρέψουν στην ίδια τιμή στην ίδια στήλη. Αυτό είναι το βασικό πλεονέκτημα της μεθόδου του Philippe Oeshlin.

Όταν κατά την χρήση των πινάκων για την παραβίαση μιας σύνοψης το τελικό σημείο ταιριάζει με την έξοδο μιας συνάρτησης ανακατασκευής, αλλά το κλειδί που βρέθηκε στην προηγούμενη στήλη δεν μπορεί να αποκρυπτογραφήσει την σύνοψη, τότε ονομάζουμε αυτή την κατάσταση false alarm. Αυτό μπορεί να συμβεί καθώς η συνάρτηση ανακατασκευής αντιστοιχεί ένα μεγαλύτερο σύνολο από συνόψεις σε ένα μικρότερο από κωδικούς με αποτέλεσμα πολλαπλές συνόψεις να αντιστοιχιστούν στον ίδιο κωδικό. Γι αυτό θα πρέπει να ελέγχουμε αν ο κωδικός μπορεί να αποκρυπτογραφήσει την σύνοψη ώστε, αν χρειαστεί, να συνεχιστεί κρυπτογραφική ανάλυση.

Ο κυριότερος τρόπος προστασίας απέναντι στους πίνακες ουράνιου τόξου είναι η χρήση salt στους κωδικούς πρόσβασης. Το salt περιλαμβάνει κάποια τυχαία bits τα οποία δεν είναι απαραίτητο να είναι κρυφά και τα οποία χρησιμοποιούνται για την επέκταση του κωδικού πρόσβασης. Για παράδειγμα αν ο κωδικός μας είναι 'password' θα κρυπτογραφήσουμε το 'this_is_my_salt'+ 'password'. Με αυτόν τον τρόπο μεγαλώνουμε το μήκος του κωδικού χωρίς να είμαστε αναγκασμένοι να απομνημονεύσουμε το salt. Ακόμα και αν κάποιος γνωρίζει το salt θα πρέπει να κατασκευάσει καινούργιους πίνακες ουράνιου τόξου ειδικά για το συγκεκριμένο salt για να μπορέσουν να τον βοηθήσουν να παραβιάσει τον κωδικό μας. Για την αύξηση της ασφάλειας τα πρώτα συστήματα UNIX χρησιμοποιούσαν 12-bit salt. Στις νεώτερες γενιές χρησιμοποιείται salt μεγαλύτερου μήκους. [27]

2.4 Πιθανότητες επιτυχίας των πινάκων ουράνιου τόξου

Ο αλγόριθμος για την κατασκευή και κατ' επέκταση για την χρήση των πινάκων ουράνιου τόξου βασίζονται στην πιθανότητα, αφού όπως έχουμε αναφέρει σαν αρχικά σημεία χρησιμοποιούνται τυχαίες είσοδοι με αποτέλεσμα να μην γνωρίζουμε για ποιους κωδικούς θα δημιουργηθούν οι αντίστοιχες συνόψεις. Έτσι υπάρχει πάντα η πιθανότητα επιτρεπτοί κωδικοί να μην έχουν κρυπτογραφηθεί και να μην μπορούν να παραβιαστούν οι συνόψεις τους. Οι πιθανότητες επιτυχίας των πινάκων που έχουν κατασκευαστεί με την υλοποίηση του Philippe Oechslin [2] ,σύμφωνα με την εργασία του, δίνονται από τους παρακάτω τύπους :

- Η πιθανότητα επιτυχίας για έναν πίνακα δίνεται από το τύπο

$$P_{table}=1-\prod_{i=1}^{t}\left(1-\frac{m_i}{N}\right)$$

όπου

m_1 =είναι ο αριθμός των αλυσίδων και $m_{i+1}=N(1-e^{-\frac{m_i}{N}})$
t είναι το μήκος της αλυσίδας
N είναι το πλήθος των δυνατών κωδικών που μπορούν να δημιουργηθούν

- Η πιθανότητα επιτυχίας για περισσότερους πίνακες δίνεται από το τύπο

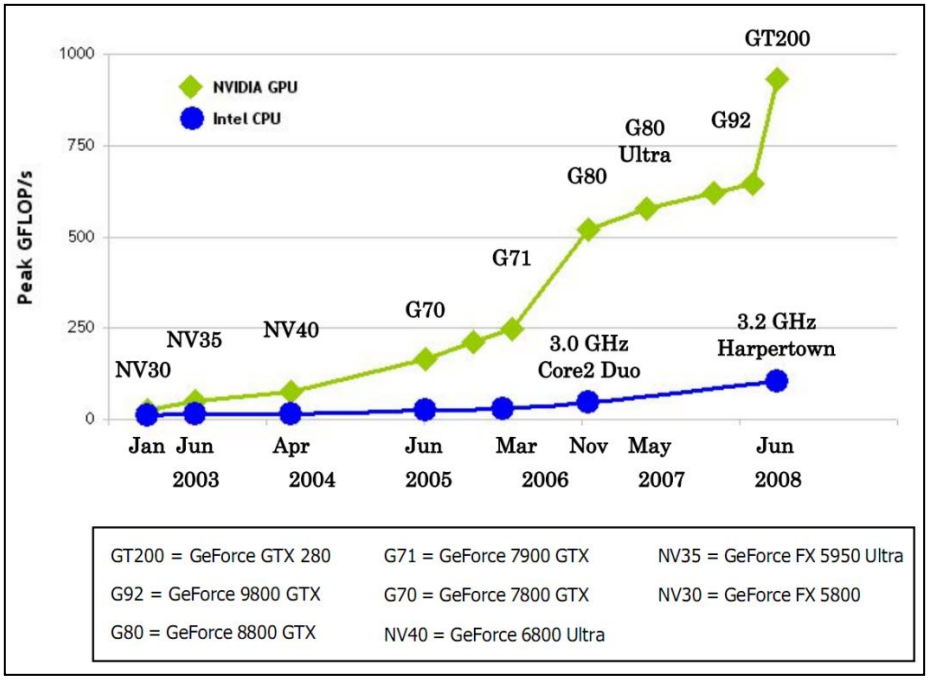
$$P_{total}\geq 1-(1-P_{table})^{r}$$

όπου
P_{table} είναι η πιθανότητα ενός πίνακα
r είναι το πλήθος των αλυσίδων που έχουν πιθανότητα επιτυχίας P_{table}

3. Η Αρχιτεκτονική CUDA

3.1 Τι είναι η CUDA;

Τα τελευταία χρόνια η βιομηχανία των υπολογιστών για να πετύχει αύξηση των επιδόσεων έχει μεταβεί σε μικροεπεξεργαστές παράλληλης επεξεργασίας που διαθέτουν πολλαπλούς πυρήνες[5]. Το πιο αντιπροσωπευτικό δείγμα αυτής της τάσης είναι οι κάρτες γραφικών (GPUs ή Graphics Processors Units) που έχουν εξελιχτεί σε συστήματα παράλληλης επεξεργασίας που διαθέτουν εκατοντάδες πυρήνες και μνήμες με μεγάλο εύρος ζώνης και υψηλές ταχύτητες μεταφοράς δεδομένων. Για την εξέλιξη αυτή μεγάλο ρόλο έχει παίξει η προσοδοφόρα βιομηχανία ηλεκτρονικών παιχνιδιών που απαιτεί από τις σύγχρονες κάρτες γραφικών να διαθέτουν μεγάλη επεξεργαστική ισχύ ώστε να μπορούν να απεικονίσουν τρισδιάστατα γραφικά υψηλής ευκρίνειας σε πραγματικό χρόνο.[6] Αποτέλεσμα, οι κάρτες γραφικών έχουν καταφέρει τα τελευταία χρόνια να παρέχουν μεγαλύτερη συνολική επεξεργαστική ισχύ από τους επεξεργαστές γενικού σκοπού[8]. Όμως για να μπορεί να γίνει αξιοποίηση της μεγάλης επεξεργαστικής ισχύος των καρτών γραφικών πρέπει να χρησιμοποιηθούν προγράμματα που να μπορούν να εκμεταλλευτούν τον παραλληλισμό. Η παραλληλοποίηση των εφαρμογών είναι μια απαιτητική διαδικασία που πρέπει να γίνει από προγραμματιστές καθώς οι αυτοματοποιημένες διαδικασίες δεν παρέχουν ικανοποιητικά αποτελέσματα. Τέλος να σημειώσω ότι δεν μπορούν όλες οι εφαρμογές να παραλληλοποιηθούν[28].



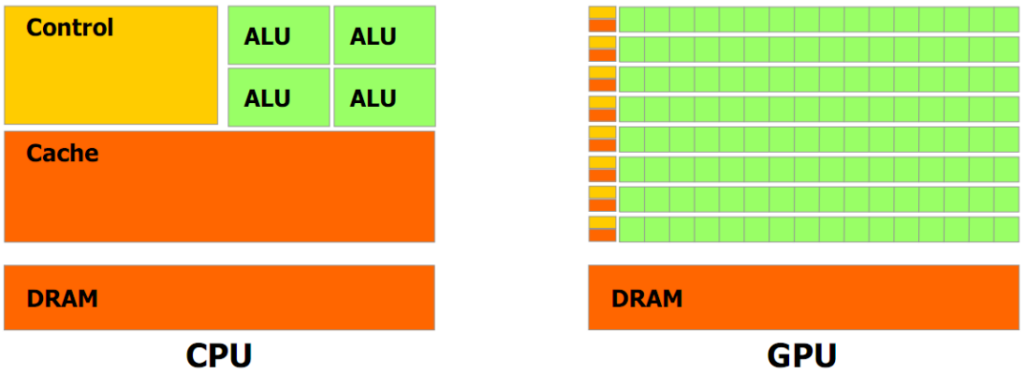
Σχήμα 3.1 *Σύγκριση επεξεργαστικής ισχύος GPU-CPU*

Η CUDA (Compute Unified Device Architecture) είναι μια παράλληλη αρχιτεκτονική που επιτρέπει την αξιοποίηση των καρτών γραφικών για εφαρμογές γενικού σκοπού. Παρουσιάστηκε

το Νοέμβριο του 2006[8] από την NVIDIA™ και αναφέρεται στην αρχιτεκτονική που διαθέτουν οι κάρτες γραφικών και σε μια επέκταση της υψηλού επιπέδου γλώσσας προγραμματισμού C. Η αρχιτεκτονική CUDA κατάφερε να κάνει ευκολότερη στους προγραμματιστές την πρόσβαση στην τεράστια υπολογιστική ισχύ που διαθέτουν οι κάρτες γραφικών για εφαρμογές γενικού σκοπού, δηλαδή εφαρμογές που δεν έχουν σχέση με επεξεργασία γραφικών. Αποτέλεσμα της παραπάνω εξέλιξης είναι τα τελευταία χρόνια οι εφαρμογές γενικού σκοπού που εκτελούνται σε κάρτες γραφικών να έχουν γίνει ιδιαίτερα δημοφιλείς.

Πριν από την δημιουργία της CUDA ο προγραμματισμός γενικού σκοπού σε κάρτες γραφικών (GPGPU) γινόταν στην γλώσσα της GPU που ήταν μια πολύ απαιτητική, με μικρή παραγωγικότητα εργασία και επιπλέον απαιτούσε την γνώση της συγκεκριμένης γλώσσας από τον προγραμματιστή. Ένα ακόμα μειονέκτημα ήταν ότι δεν αξιοποιούνται αποτελεσματικά όλοι των πόρων της κάρτας, με αποτέλεσμα να υπάρχει μειωμένη απόδοση καθώς η εκτέλεση του προγράμματος γινόταν με την βοήθεια του API των γραφικών[7]. Αντίθετα στην CUDA παρέχεται αυξημένη ευελιξία και μεγάλη παραγωγικότητα καθώς ο προγραμματισμός γίνεται στην δημοφιλή γλώσσα C ενώ διαθέτει ένα σύνολο εντολών γενικού σκοπού (GPU assembly) ώστε το πρόγραμμα να εκτελείται πάνω στο υλικό που παρέχει την πλήρη εκμετάλλευση του υλικού, επιτυγχάνοντας την μέγιστη απόδοση.

Οι κάρτες γραφικών που υποστηρίζουν CUDA (CUDA-enabled GPUs) έχουν διαφορετική αρχιτεκτονική από τις CPU, καθώς προορίζονται στην εκτέλεση της ίδιας εντολής για διαφορετικά δεδομένα. Αυτό έχει ως αποτέλεσμα τον απλούστερο έλεγχο ροής του προγράμματος καθώς οι καθυστερήσεις που δημιουργούνται από την μνήμη μπορούν να κρύβονται με την εναλλαγή των δεδομένων που επεξεργάζονται και όχι με την χρήση κρυφών μνημών που χρησιμοποιούν οι CPUs. Έτσι ένας μεγάλος αριθμός από τρανζίστορ μπορεί να ανατεθεί για την επεξεργασία δεδομένων, αντίθετα από τις CPUs που ανατίθεται για την υλοποίηση κρυφών μνημών και στον έλεγχο ροής.



Σχήμα 3.2 *Διαφορές αρχιτεκτονικής CPU-GPU.*

Η NVIDIA χρησιμοποιεί την παρακάτω ορολογία την οποία θα χρησιμοποιήσω και εγώ σε όλο το πλαίσιο της εργασίας*.

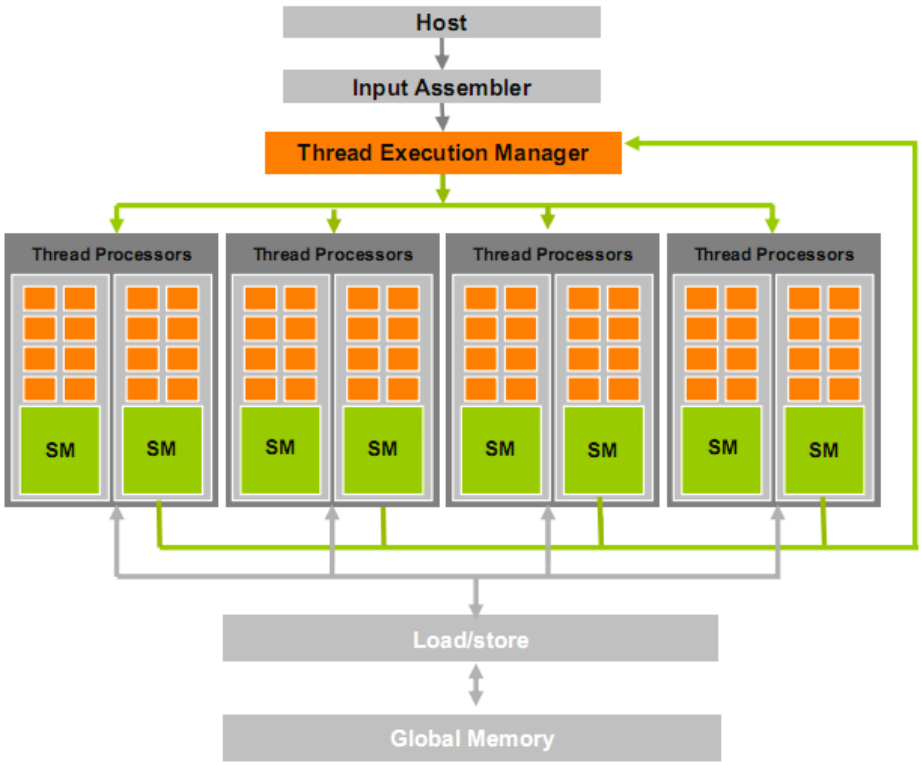
- *Συσκευή (device)* είναι η κάρτα γραφικών.
- *Σύστημα (host)* είναι η CPU και η κύρια μνήμη του υπολογιστή στον οποίο είναι εγκαταστημένη η κάρτα γραφικών.
- *Μνήμη συσκευής (device memory)* είναι η μνήμη που υπάρχει στην συσκευή.

*Σε όλο το πλαίσιο της εργασίας η αρχιτεκτονική υλικού αναφέρεται σε κάρτες γραφικών με compute capability 1.2 [8].

- *Μνήμη συστήματος (host memory)* είναι η κύρια μνήμη του υπολογιστή.
- *Kernel* είναι μια συνάρτηση που εκτελείται στην συσκευή.
- *Thread* είναι η μικρότερη μονάδα παραλληλίας στην CUDA.
- *Warp* είναι μια ομάδα από 32 threads που εκτελούνται παράλληλα σε φυσικό επίπεδο. Κάθε warp αποτελείται από δύο half-warps.
- *Block* είναι μια ομάδα από 1 μέχρι 512 threads που εκτελούνται παράλληλα σε έναν πολυεπεξεργαστή ενώ συνεργάζονται μεταξύ τους, με τον συγχρονισμό της εκτέλεσής τους και την χρήση της κοινόχρηστης μνήμης. Κάθε thread που ανήκει στο block μπορεί να ταυτοποιηθεί από ένα μοναδικό αριθμό αναγνώρισης.
- *Grid* είναι το σύνολο των blocks από τα οποία αποτελείται η εφαρμογή. Ο μέγιστος αριθμός των blocks μπορεί να είναι 65.535 και κάθε ένα μπορεί να ταυτοποιηθεί από ένα μοναδικό αριθμό αναγνώρισης.

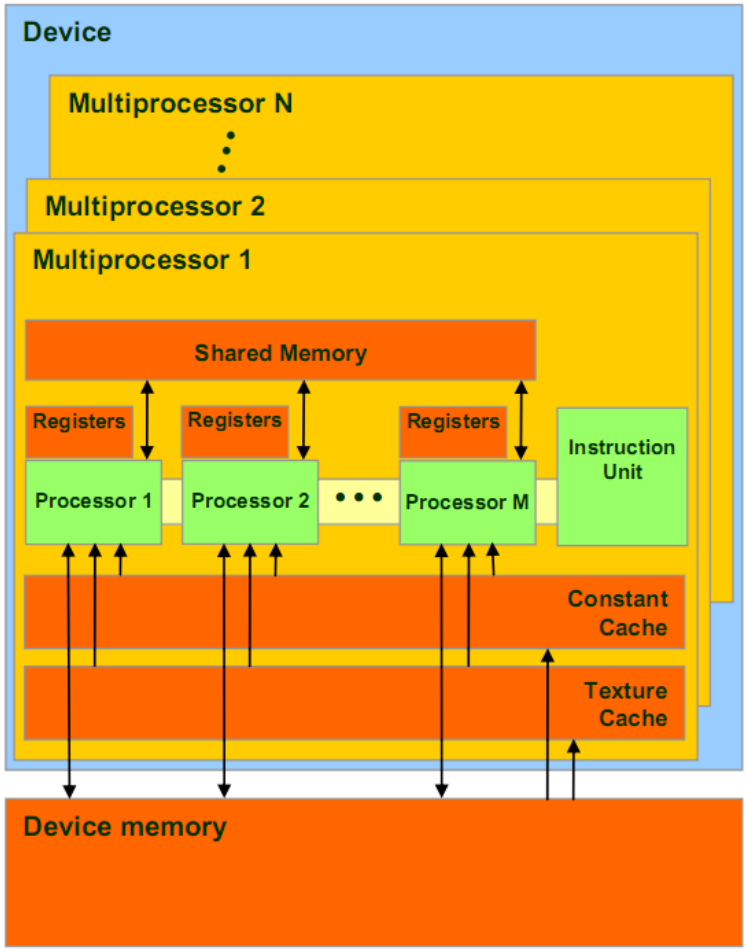
3.2 Αρχιτεκτονική Υλικού (Hardware Implementation)

Η αρχιτεκτονική της CUDA βασίζεται σε πολυεπεξεργαστές ροής (streaming multiprocessors) οι οποίοι επικοινωνούν με την μνήμη συσκευής. Για τις πιο διαδεδομένες κάρτες γραφικών ο αριθμός των πολυεπεξεργαστών κυμαίνεται από 8 μέχρι 24 ενώ το μέγεθος της μνήμης συσκευής κυμαίνεται από 512MB μέχρι 1GB. Στο παρακάτω σχήμα φαίνεται η αρχιτεκτονική μιας κάρτας γραφικών με 8 πολυεπεξεργαστές οργανωμένους σε ομάδες των δύο.



Σχήμα 3.3 *Γενική άποψη αρχιτεκτονικής CUDA.*

Κάθε πολυεπεξεργαστής διαθέτει 8 ανεξάρτητους επεξεργαστές ροής (streaming processors) που επικοινωνούν με την κοινόχρηστη μνήμη (shared memory) το αρχείο καταχωρητών και με δύο κρυφές μνήμες, την constant και texture cache. Όλες οι παραπάνω μονάδες είναι προσβάσιμες μόνο από τα threads που εκτελούνται στον συγκεκριμένο πολυεπεξεργαστή.



Σχήμα 3.4 *Αρχιτεκτονική Πολυεπεξεργαστή*

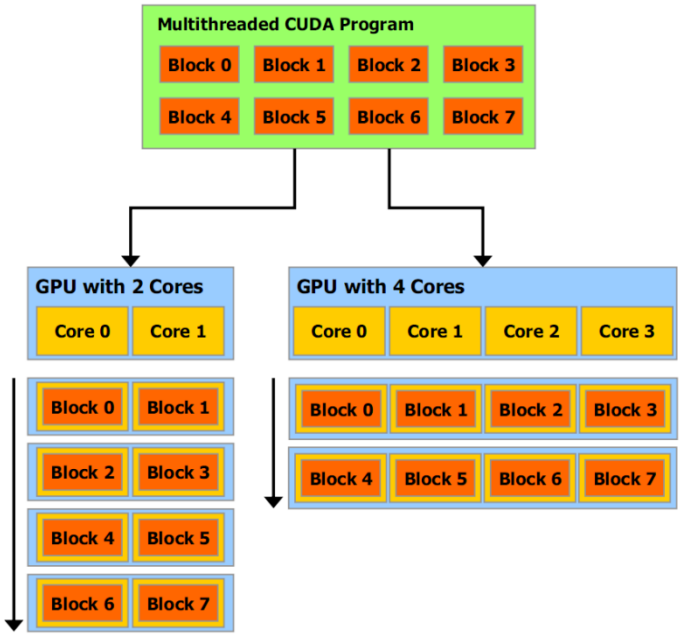
3.3 Η αρχιτεκτονική SIMT

Ο πολυεπεξεργαστής είναι σχεδιασμένος ώστε να μπορεί να εκτελεί εκατοντάδες threads συγχρόνως μέσω μιας μοναδικής αρχιτεκτονικής που ονομάζεται SIMT (Single Instruction Multiple-Thread) η οποία του επιτρέπει να διαχειρίζεται αποτελεσματικά τον μεγάλο αριθμό από threads.

Ο πολυεπεξεργαστής δημιουργεί, διαχειρίζεται και εκτελεί threads που είναι οργανωμένα σε ομάδες των 32 που ονομάζονται **warps**. Τα μεμονομένα threads που συγκροτούν ένα warp ξεκινάνε εκτελώντας την ίδια εντολή αλλά έχουν τον δικό τους μετρητή προγράμματος (instruction counter) και χρησιμοποιούν διαφορετικούς καταχωρητές με αποτέλεσμα να είναι ελεύθερα να παρεκλίνουν από το αρχικό μονοπάτι και να εκτελούνται ανεξάρτητα. Καθώς όμως οι επεξεργαστές ροής ενός μεμονομένου πολυεπεξεργαστή μπορούν είτε να εκτελούν όλοι την ίδια εντολή ή να παραμένουν ανενεργοί, για την επίτευξη της μέγιστης απόδοσης όλα τα threads ενός warp πρέπει να ακολουθούν το ίδιο μονοπατι εκτέλεσης. Σε

περίπτωση που έχουμε διαφορετικά μονοπάτια η εκτέλεσή τους θα γίνει σειριακά χωρίς να αξιοποιούνται οι οκτώ επεξεργαστές ροής, αλλά ένας μικρότερος αριθμός, μέχρι η εκτέλεση να επιστρέψει σε ένα κοινό μονοπάτι. Καθώς ο πολυεπεξεργαστής διαχειρίζεται warps ο χωρισμός μονοπατιού έχει νόημα μόνο για warps.

Όταν σε έναν πολυεπεξεργαστή ανατεθεί να εκτελέσει ένα ή περισσότερα blocks, αυτός χωρίζει τα threads σε warps και στη συνέχεια αναλαμβάνει ο warp scheduler τον προγραμματισμό και την εκτέλεσή τους, ενώ μόλις ολοκληρωθεί η εκτέλεση κάποιου block ο πολυεπεξεργαστής είναι έτοιμος να αναλάβει να εκτελέσει κάποιο διαθέσιμο block. Για να μπορούν να προσδιοριστούν με μοναδικό τρόπο τα thread ενός block, τους ανατίθεται ένας μοναδικός αριθμός αναγνώρισης ξεκινώντας από το μηδέν. Υπάρχουν όμως κάποιοι περιορισμοί που πρέπει να τηρηθούν όσο αφορά τον μέγιστο αριθμό από blocks και warps που μπορούν να ανατεθούν σε ένα πολυεπεξεργαστή και αυτοί είναι 8 και 24 αντίστοιχα. Επιπλέον υπάρχουν περιορισμοί που επιβάλλονται από τους διαθέσιμους πόρους του πολυεπεξεργαστή ενώ, αν δεν υπάρχουν διαθέσιμοι για να ανατεθεί ένα τουλάχιστον block, τότε ο kernel θα αποτύχει να ξεκινήσει. Τέλος όλα τα warps αποτελούνται από threads με διαδοχικούςαριθμούς αναγνώρισης δηλαδή το πρώτο warp θα περιέχει threads με αριθμούς αναγνώρισης 0-31, το δεύτερο 32-63 και αυτό συνεχίζεται μέχρι να αντιστοιχιστούν όλα τα threads σε κάποιο warp, με την προϋπόθεση ότι οι παραπάνω περιορισμοί εξακολουθούν να τηρούνται.



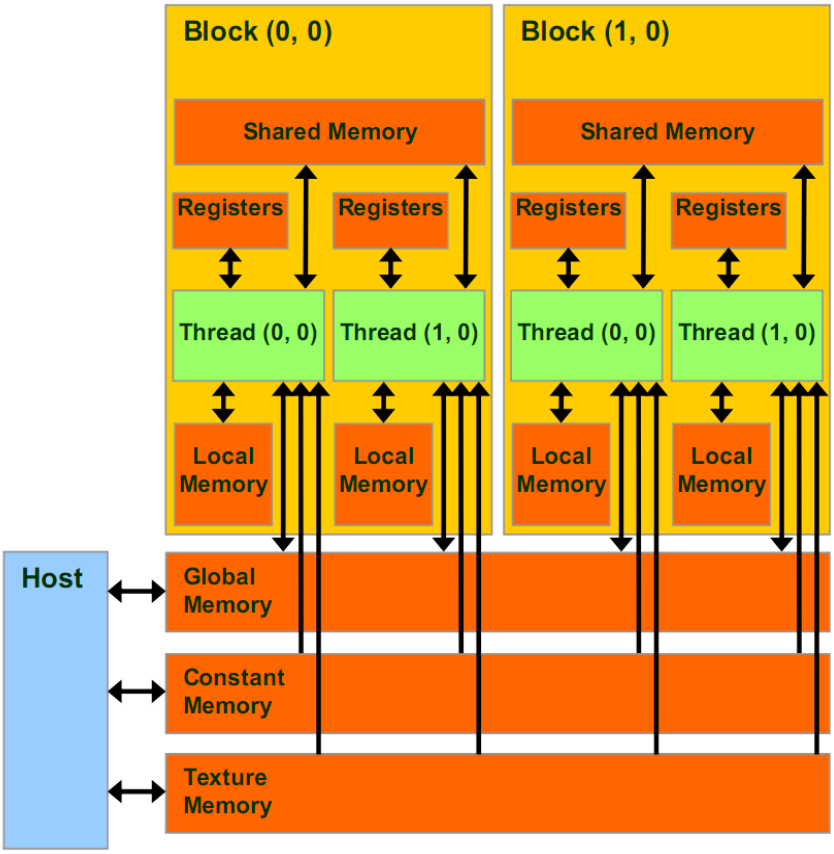
Σχήμα 3.5 *Ανάθεση των blocks στους πολυεπεξεργαστές*

Ο warp scheduler αναλαμβάνει να εναλλάσει τα warps με στόχο να παρέχει στον πολυεπεξεργαστή κάθε στιγμή κάποιο warp που όλα του τα threads είναι έτοιμα να εκτελεστούν. Το κόστος της εναλλαγής είναι αμελητέο καθώς πραγματοποιείται σε μόλις ένα κύκλο ρολογιού. Μόλις γίνει εναλλαγή warp αρχίζουν να εκδίδονται οι εντολές που πρέπει να εκτελέσει μέχρις ότου κάποιο από τα thread του να μπλοκαριστεί, οπότε ο warp scheduler θα αναλάβει την εναλλαγή του.

3.4 Μοντέλο Μνήμης

Εκτός από την φυσικές (πραγματικές) μνήμες που είδαμε στην προηγούμενη παράγραφο και θα παρουσιάσουμε αναλυτικότερα παρακάτω, υπάρχουν και κάποιες μνήμες που φιλοξενούνται στην μνήμη της συσκευής, οι οποίες χρησιμοποιούνται από την αρχιτεκτονική SIMT. Οι εικονικές μνήμες είναι η global memory, η local memory, η constant memory και η

texture memory. Στο επόμενο σχήμα φαίνεται το μοντέλο μνήμης με το σύνολο των μνημών.



Σχήμα 3.6 *Μοντέλο μνήμης*

Ακολουθεί αναλυτική παρουσίαση όλων των ειδών μνήμης για την χρήση τους και τον τρόπο με τον οποίο επιτυγχάνονται οι μέγιστες αποδόσεις τους.

Μνήμη συσκευής (device memory)

Η μνήμη αυτή έχει πολύ μεγάλη χωρητικότητα και μεγάλο εύρος ζώνης, όμως η πρόσβαση σε αυτήν εισάγει καθυστέρηση (latency) 400-600 κύκλους ρολογιού. Για να αξιοποιηθεί πλήρως το μεγάλο εύρος ζώνης πρέπει οι προσβάσεις σε αυτήν να είναι coalesced δηλαδή οι προσβάσεις πρέπει να προσπελαύνουν δεδομένα των 32, 64 και 128 bytes που βρίσκονται σε συνεχόμενες και ευθυγραμμισμένες θέσεις στην μνήμη. Με τον όρο ευθυγραμμισμένες εννοούμε ότι η πρώτη διεύθυνση των δεδομένων είναι πολλαπλάσια του μεγέθους τους. Με αυτό τον τρόπο ένα half-warp μπορεί να εξυπηρετηθεί σε μια μόνο συναλλαγή (transaction), δηλ. τον μικρότερο δυνατό αριθμό, με αποτέλεσμα να επιτυγχάνεται το μέγιστο εύρος ζώνης. Η ποινή για μη coalesced προσβάσεις στην μνήμη διαφέρει ανάλογα με τον τύπο και το μέγεθος των δεδομένων.

Global memory

Η μνήμη αυτή έχει τις ίδιες αποδόσεις με την μνήμη συσκευής καθώς φιλοξενείται σε αυτήν, όπου καταλαμβάνει και το μεγαλύτερο μέρος της. Χρησιμοποιείται για την αποθήκευση μεγάλων δεδομένων που θα χρησιμοποιηθούν από όλα τα blocks. Είναι προσβάσιμη για ανάγνωση και εγγραφή τόσο από το σύστημα, όσο και από την συσκευή, ενώ τα δεδομένα που αποθηκεύονται σε αυτήν έχουν την διάρκεια ζωής της εφαρμογής.

Local memory

Η μνήμη αυτή φιλοξενείται στην μνήμη της συσκευής και επομένως έχει τις ίδιες αποδόσεις με την μνήμη συσκευής. Η μνήμη αυτή περιορίζεται στο μέγεθος των 16KB για κάθε thread. Είναι προσβάσιμη για ανάγνωση και εγγραφή μόνο από την συσκευή και χρησιμοποιείται για την διατήρηση πινάκων και δομών δεδομένων που θα απαιτούσαν πολύ χώρο για την αποθήκευση τους στους καταχωρητές, ενώ μπορεί να χρησιμοποιηθεί για οποιαδήποτε μεταβλητή αν οι καταχωρητές δεν επαρκούν. Τα δεδομένα που αποθηκεύονται είναι προσπελάσιμα μόνο από το thread που τα δημιούργησε και έχουν την διάρκεια της ζωής του.

Constant memory

Η μνήμη αυτή φιλοξενείται στην μνήμη της συσκευής, καταλαμβάνοντας 64KB, όμως διαθέτει κρυφή μνήμη μεγέθους 8KB σε κάθε πολυεπεξεργαστή που της επιτρέπει να επιτύχει πολύ καλές επιδόσεις. Η μνήμη είναι προσπελάσιμη από οποιοδήποτε block μόνο για ανάγνωση σε δεδομένα που έχουν αποθηκευτεί από το σύστημα, πριν την κλήση του kernel.

Texture memory

Η μνήμη αυτή φιλοξενείται στην μνήμη της συσκευής, όμως διαθέτει μια ειδική κρυφή μνήμη μεγέθους 6-8KB σε κάθε πολυεπεξεργαστή που της επιτρέπει να εκμεταλλευτεί σε πολύ μεγάλο βαθμό την τοπική χωρητικότητα με αποτέλεσμα να επιτυγχάνει πολύ καλές επιδόσεις. Η μνήμη είναι προσπελάσιμη από οποιοδήποτε block μόνο για ανάγνωση σε δεδομένα που έχουν αποθηκευτεί από το σύστημα, πριν την κλήση του kernel.

Αρχείο Καταχωρητών (Registers file)

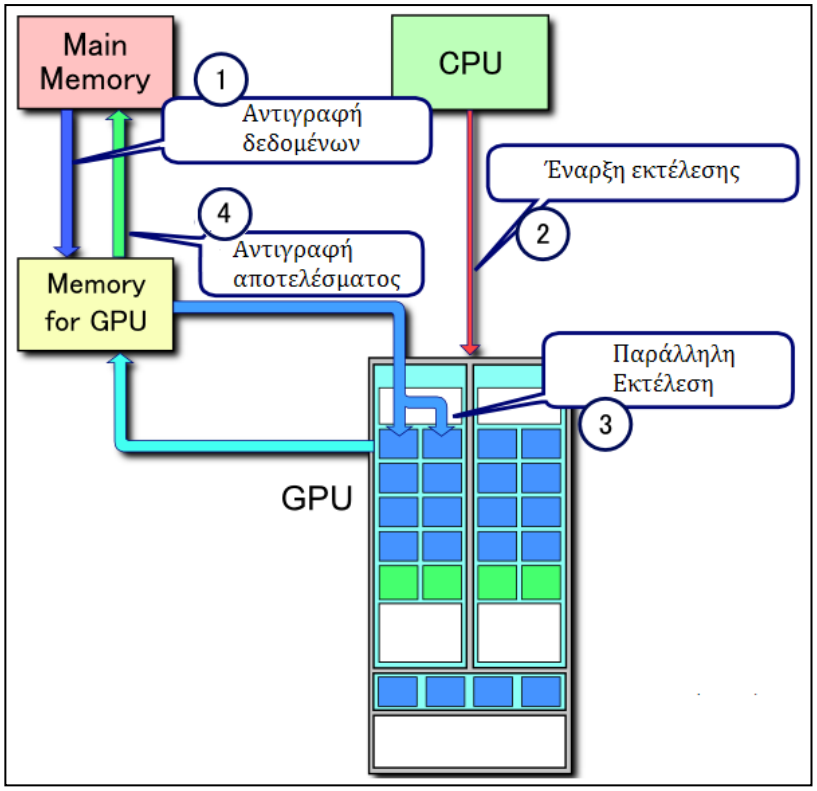
Είναι η γρηγορότερη μορφή μνήμης στον πολυεπεξεργαστή και διαθέτει 16K καταχωρητές των 32-bits. Τα δεδομένα που αποθηκεύονται είναι προσπελάσιμα μόνο από το thread που τα δημιούργησε και έχουν την διάρκεια της ζωής του.

Κοινόχρηστη μνήμη (Shared memory)

Η μνήμη αυτή βρίσκεται στον πολυεπεξεργαστή και έχει χωρητικότητα 16KB. Για να μπορεί να πετύχει ισάξιες επιδόσεις με τους καταχωρητές είναι χωρισμένη σε 16 τμήματα ίσου μεγέθους που ονομάζονται banks και τα οποία μπορούν να προσπελάζονται ταυτόχρονα. Οι διαδοχικές 32-bit λέξεις ανατίθενται σε διαδοχικά banks ώστε να μπορούν να εξυπηρετηθούν ταυτόχρονα 16 προσβάσεις, όταν χρησιμοποιούν διευθύνσεις που αντιστοιχούν σε διαφορετικές banks, επιτυγχάνοντας εύρος ζώνης 16 φορές υψηλότερο από ότι ένα bank. Στην περίπτωση όμως που δύο ή περισσότερες προσβάσεις αντιστοιχιστούν σε μια bank έχουμε bank conflict και τότε θα πρέπει να εξυπηρετηθούν σειριακά. Η κοινόχρηστη μνήμη είναι προσβάσιμη από οποιοδήποτε thread του block που την δημιούργησε και έχει τον χρόνο ζωής του block.

3.5 Ένα τυπικό πρόγραμμα

Στο σημείο αυτό μπορούμε να περιγράψουμε πώς δουλεύει ένα τυπικό πρόγραμμα υλοποιημένο σε CUDA. Όπως φαίνεται και στο σχήμα 3.6 το πρώτο βήμα είναι η μεταφορά των δεδομένων προς επεξεργασία από την κύρια μνήμη του συστήματος στην μνήμη της συσκευής. Επόμενο βήμα είναι η έναρξης της εκτέλεσης στην συσκευή ενώ στο επόμενο βήμα γίνεται η επεξεργασία των δεδομένων παράλληλα, ενώ ταυτόχρονα ο επεξεργαστής του συστήματος(CPU) μπορεί να είναι απασχολημένος με άλλη εργασία . Στο τέταρτο βήμα μεταφέρονται τα επεξεργασμένα δεδομένα από την μνήμη της συσκευής στην κύρια μνήμη του συστήματος.



Σχήμα 3.7 *Ροή εκτέλεσης ενός προγράμματος σε CUDA*

3.6 Τεχνικές Βελτιστοποίησης

Η βελτιστοποίηση της απόδοσης των προγραμμάτων επιτυγχάνεται μέσω των ακόλουθων στρατηγικών:

- *Παράλληλη επεξεργασία δεδομένων* για να μπορούν να αξιοποιηθούν πλήρως όλες οι λειτουργικές μονάδες της συσκευής.
- Αύξηση του *εύρους ζώνης της μνήμης* για αποδοτικότερη αξιοποίηση των λειτουργικών μονάδων της συσκευής.
- Αύξηση του *ρυθμού έκδοσης εντολών* για βέλτιστη χρήση των λειτουργικών μονάδων της συσκευής.

3.6.1 Παράλληλη Επεξεργασία

Για να επιτευχθεί η πλήρης αξιοποίηση όλων των λειτουργικών μονάδων και να παραμένουν απασχολημένες όσο περισσότερο γίνεται θα πρέπει η εφαρμογή να έχει υλοποιηθεί με τέτοιο τρόπο, ώστε να μπορεί να εκτελεστεί παράλληλα.

Πρώτη ενέργεια κατά την σχεδιασμό μιας εφαρμογής είναι ο εντοπισμός των τμημάτων που μπορούν να εκτελεστούν παράλληλα. Στη συνέχεια η εφαρμογή θα υλοποιηθεί με τέτοιο τρόπο ώστε να αναθέτει τα τμήματα που μπορούν να εκτελεστούν παράλληλα, στον επεξεργαστή της συσκευής, ενώ τα τμήματα που εκτελούνται σειριακά, στην CPU. Με αυτό τον τρόπο επιταχύνεται η μέγιστη συνολικά απόδοση της εφαρμογής αφού αναθέτουμε σε κάθε επεξεργαστή να εκτελέσει τις εργασίες για τις οποίες έχει κατασκευαστεί και είναι πιο αποδοτικός.

Για την αξιοποιηθούν όλες οι λειτουργικές μονάδες της συσκευής, τα παράλληλα τμήματα θα πρέπει να χωριστούν κατάλληλα ώστε να ανατεθεί μέρος της δουλειά σε κάθε

επεξεργαστή ροής όλων των πολυεπεξεργαστών. Στον kernel θα πρέπει να εκτελούνται blocks ίσα με τον αριθμό των πολυεπεξεργαστών ώστε σε κάθε πολυεπεξεργαστή να ανατίθεται η εκτέλεση ενός block. Επιπλέον κάθε block θα πρέπει να εκτελεί οκτώ threads ώστε να ανατίθεται δουλειά σε κάθε έναν από τους 8 επεξεργαστές ροής κάθε πολυεπεξεργαστή.

Όμως για την πλήρη αξιοποίηση των λειτουργικών μονάδων χρειάζεται τα παράλληλα τμήματα να χωριστούν σε μεγαλύτερο βαθμό. Το μέτρο της αξιοποίησης είναι η απασχόληση (occupancy) η οποία αναφέρεται στον λόγο των ενεργών warps προς τον μέγιστο αριθμό από warps που μπορούν να ανατεθούν σε έναν πολυεπεξεργαστή. Γενικά όσο μεγαλύτερος είναι ο λόγος, τόσο καλύτερη απόδοση μπορούμε να πετύχουμε, καθώς στην CUDA οι εντολές μέσα σε ένα thread εκτελούνται σειριακά και επιτρέπεται η εναλλαγή των ενεργών warps. Με την εναλλαγή είναι δυνατόν να αποφύγουμε την καθυστέρηση μεμονωμένων warps, αφού όση ώρα παραμένουν κολλημένα μπορούμε να εκτελέσουμε κάποια άλλα και με αυτόν τον τρόπο να κρατάμε τον πολυεπεξεργαστή απασχολημένο περισσότερη ώρα. Όπως έχουμε αναφέρει νωρίτερα ο μέγιστος αριθμός των warps που μπορούν να ανατεθούν σε έναν πολυεπεξεργαστή περιορίζονται από τα κατασκευαστικά όρια. Όμως τα ενεργά warps που μπορούν να ανατεθούν σε έναν πολυεπεξεργαστή εξαρτούνται από τους διαθέσιμους πόρους και συγκεκριμένα από τους διαθέσιμους καταχωρητές και την διαθέσιμη τοπική και κοινόχρηστη μνήμη. Επομένως είναι σκόπιμο να καταναλώνουμε λίγους πόρους σε κάθε warp για να αυξάνουμε την απασχόληση. Να επισημάνω ότι όσο αυξάνεται η απασχόληση δεν σημαίνει πάντα καλύτερες επιδόσεις καθώς υπάρχει ένα ανώτερο όριο που περιορίζει την αύξηση των επιδόσεων. Από την άλλη όμως χαμηλή απασχόληση σημαίνει χαμηλή επίδοση καθώς δεν υπάρχει τρόπος να αποφευχθούν οι καθυστερήσεις.

3.6.2 Αύξηση του εύρους ζώνης της μνήμης

Ο στόχος από την αύξηση του εύρους ζώνης της μνήμης είναι η έγκαιρη μεταφορά δεδομένων για επεξεργασία στις λειτουργικές μονάδες για να παραμένουν απασχολημένες όσο το δυνατόν περισσότερο χρόνο. Αυτό επιτυγχάνεται με τα παρακάτω:

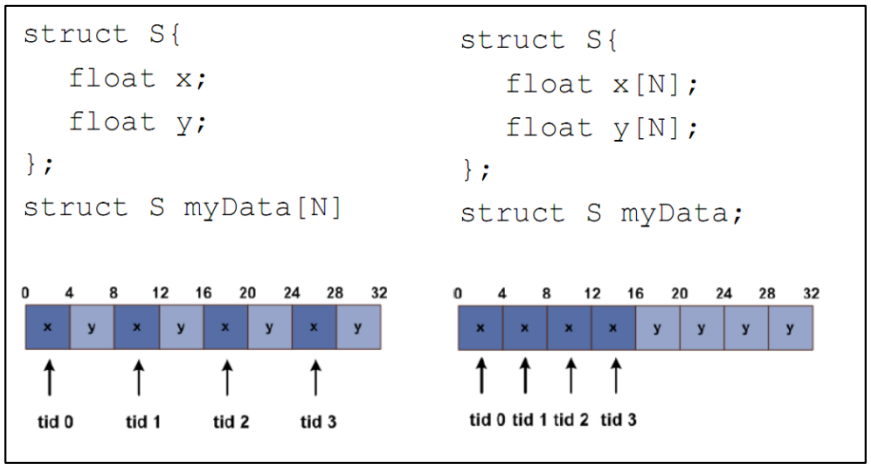
- Αποδοτική μεταφορά δεδομένων από το σύστημα στην συσκευή και αντίστροφα.
- Αξιοποίηση της κοινόχρηστης μνήμης.
- Αξιοποίηση της constant και texture memory.
- Αποδοτική προσπέλαση σε όλες τις μνήμες μέσω του μικρότερο δυνατού αριθμού συναλλαγών.

Για να πετύχουμε αποδοτική μεταφορά δεδομένων μεταξύ συστήματος και συσκευής θα πρέπει να γίνεται ομαδοποίηση στις μεταφορές ώστε να μειώνεται ο αριθμός τους με αντίστοιχη αύξηση του όγκου δεδομένων.

Η χρήση της κοινόχρηστης μνήμης είναι η σημαντικότερη τεχνική για την βελτίωση της απόδοσης και γι’ αυτό θα πρέπει να χρησιμοποιείται κατά την μεταφορά των δεδομένων προς επεξεργασία . Τα δεδομένα πριν από την επεξεργασία τους, πρέπει να μεταφέρονται στην κοινόχρηστη μνήμη και να επιστρέφουν στην global memory τα αποτελέσματα.

Η constant memory μπορεί να χρησιμοποιείται για εξοικονόμηση χώρου στην κοινόχρηστη μνήμη από δεδομένα που χρησιμοποιούνται μόνο για ανάγνωση από πολλαπλά warps και επίσης για την αποθήκευση πινάκων εντοπισμού(lookup tables) που επιτρέπουν την αποφυγή διακλάδωσης της εκτέλεσης. Από την άλλη η texture memory μπορεί να χρησιμοποιηθεί για την αποθήκευση μεγάλων δομών δεδομένων που προορίζονται μόνο για ανάγνωση για να υπάρχει εκμετάλλευση της χωρικής και χρονικής τοπικότητας.
Η αποδοτική προσπέλαση στις μνήμες επηρεάζει σε μεγάλο βαθμό την συνολική απόδοση του προγράμματος και πρέπει να δίνεται ιδιαίτερη προσοχή. Για να επιτευχθούν αποδοτικές

προσπελάσεις μπορούν να χρησιμοποιούνται οι τύποι μεταβλητών που παρέχει η CUDA. Αν απαιτούνται καινούργιες δομές δεδομένων το μέγεθός τους πρέπει να είναι μικρότερο από 128 bytes ώστε να μπορούν να συμπληρωθούν με αχρησιμοποίητα bytes για να αποκτήσουν μέγεθος 32, 64 ή 128 bytes ενώ γενικότερα θα πρέπει να χρησιμοποιούνται δομές διατάξεων και όχι διατάξεις δομών ώστε να πραγματοποιούνται προσβάσεις που να είναι σε συνεχόμενες θέσεις μνήμης.



Σχήμα 3.8 *Διαφορές στην πρόσβαση με δομή διατάξεων και διάταξη δομών*

3.6.3 Χρήση αποδοτικών εντολών

Επόμενη στρατηγική για την αύξηση των επιδόσεων είναι η βέλτιστη χρήση των λειτουργικών μονάδων με την αύξηση του ρυθμού έκδοσης εντολών. Η σημαντικότερη τεχνική που μπορεί να χρησιμοποιηθεί είναι η αποφυγή χρήσης κώδικα που διακλαδώνεται μέσα σε ένα warp. Άλλες τεχνικές είναι η χρήση bit masking στις πράξεις της διαίρεσης και του υπολοίπου με αριθμούς που είναι δυνάμεις του δύο, ενώ αν είναι δυνατό, να αποφεύγεται η χρήση τους.

Επισήμανση

Η εφαρμογή όλων των παραπάνω τεχνικών είναι προαιρετική, ενώ μπορούν να πραγματοποιηθούν με οποιαδήποτε σειρά μετά από αλλεπάλληλες δοκιμές με τα πραγματικά δεδομένα του προβλήματος.

4. Σχεδιασμός και υλοποίηση της εφαρμογής σε CUDA

Στο κεφάλαιο αυτό γίνεται μια σύντομη παρουσίαση παλιότερων εργασιών κρυπτογραφικής ανάλυση με χρήση πινάκων ουράνιου τόξου. Στην συνέχεια ακολουθεί μια αναλυτική παρουσίαση της δικής μου σχεδίασης και τους λόγους που με οδήγησαν στην χρήση της. Ακολουθεί μία αναλυτική παρουσίαση της υλοποίησης και το κεφάλαιο αυτό ολοκληρώνεται με την βελτιστοποίηση της αρχικής υλοποίησης. Ιδιαίτερο βάρος δίνεται στην ορθή σχεδίαση και στην ορθή λειτουργία της εφαρμογής.

4.1 Σχετικές εργασίες κατασκευής πινάκων ουράνιου τόξου

Μια ολοκληρωμένη εργασία για την κατασκευή και χρήση πινάκων ουράνιου τόξου αποτελεί το project RainbowCrack [3] το οποίο παρέχει υλοποιήσεις τόσο σε C++ όσο και σε αρχιτεκτονική CUDA. Το RainbowCrack παρέχει ελεύθερα τον κώδικα μιας παλιότερης έκδοσης σε C++, όμως η υλοποίηση σε CUDA είναι εμπορικό προϊόν με κλειστό κώδικα και κατά συνέπεια δεν μπορώ να τον χρησιμοποιήσω για αξιολόγηση. Επιπλέον καθώς στην εμπορική εφαρμογή δίνεται ιδιαίτερο βάρος στις επιδόσεις παραβίασης κωδικών και όχι στην κατασκευή των πινάκων δεν υπάρχουν διαθέσιμα στοιχεία για τον χρόνο κατασκευής πινάκων παρά μόνο για την παραβίαση Η υλοποίηση σε C++ παρέχει την δυνατότητα να επιλέγει ο χρήστης τον αλγόριθμο κρυπτογράφησης, το πλήθος και το μήκος των αλυσίδων και τα σύμβολα και το μήκος των κωδικών. Οι κρυπτογραφικοί αλγόριθμοι κατακερματισμού που μπορούν να χρησιμοποιηθούν είναι μεταξύ άλλων οι LM hash, MD5 και SHA-1. Οι αλγόριθμοί MD5 και SHA-1 παρέχονται από την βιβλιοθήκη του OpenSSL[22] ενώ αντίθετα ο LM hash δεν παρέχεται από την βιβλιοθήκη και πρέπει να διαμορφωθεί ο κωδικός σε τέτοια μορφή ώστε να είναι κατάλληλη για κρυπτογράφηση από τον DES που παρέχεται από το OpenSSL. Για τα αρχικά σημεία των αλυσίδων χρησιμοποιούνται τυχαίες τιμές που παράγονται από μια γεννήτρια τυχαίων αριθμών. Για κάθε κρίκο χρησιμοποιείται διαφορετική συνάρτηση ανακατασκευής με την πρόσθεση στην αριθμητική τιμή της σύνοψης του αριθμού της στήλης. Τέλος η συνάρτηση μετατροπής υλοποιείται με διαδοχικές διαιρέσεις στην έξοδο της συνάρτησης ανακατασκευής και στην συνέχεια επιλέγονται τα επιτρεπτά σύμβολα του κωδικού από ένα πίνακα. Καθώς η εκτέλεση γίνεται σε στον κεντρικό επεξεργαστή η επίδοση που επιτυγχάνεται είναι σχετικά φτωχή.
Μια ακόμα εργασία για την κατασκευή πινάκων ουράνιου τόξου είναι η διπλωματική εργασία του Κ.Θεοχαρούλη που χρησιμοποιεί μια πλατφόρμα FPGA αποκλειστικά για την κατασκευή πινάκων ουράνιου τόξου για τους κρυπτογραφικούς αλγόριθμους LM hash, MD5, SHA-1. Οι κωδικοί που μπορούν να χρησιμοποιηθούν έχουν μήκος επτά χαρακτήρων και αποτελούνται από 64 σύμβολα. Η υλοποίηση της συνάρτησης ανακατασκευής γίνεται με την βοήθεια πυλών xor. Στην εργασία αυτή κατάφερε να πετύχει βελτίωση περίπου 400 φορές με σχέση με την υλοποίηση σε software.

4.2 Σχεδιασμός Εφαρμογής

4.2.1 Γενικός Σχεδιασμός

Η εργασία αυτή αφορά μόνο την κατασκευή των πινάκων, ενώ η χρήση τους για παραβίαση κωδικών αφήνεται για μελλοντική εργασία. Οι τρείς απαιτήσεις που έθεσα για την εφαρμογή είναι οι ακόλουθες: Πρώτον να παρέχεται η δυνατότητα στον χρήστη να επιλέγει όλες τις παραμέτρους που αφορούν τις αλυσίδες και τους κωδικούς, δεύτερον όλοι οι κωδικοί να έχουν την ίδια πιθανότητα κρυπτογράφησης από τις αλυσίδες και τρίτον να πετύχω μεγάλη βελτίωση στον χρόνο κατασκευή τους με σχέση την υλοποίηση για CPU. Κατά τον σχεδιασμό έπρεπε να λάβω υπόψη τους διάφορους περιορισμούς που επιβάλει η αρχιτεκτονική CUDA και ιδιαίτερα τον περιορισμό που δεν επιτρέπει την χρήση συναρτήσεων από βιβλιοθήκες στον κώδικα που εκτελείται από την κάρτα γραφικών.

Η υλοποίηση του RainbowCrack δεν ικανοποιεί τους περιορισμούς της αρχιτεκτονικής CUDA καθώς κάνει χρήση συναρτήσεων από βιβλιοθήκες. Επίσης καλύπτει μόνο εν μέρει τις απαιτήσεις που έθεσα καθώς δεν επιτρέπει την βελτίωση των αποδόσεων λόγω της εκτεταμένης χρήση της πράξης της διαίρεσης που είναι υπερβολικά αργή στην CUDA. Για τους παραπάνω λόγους έκρινα ότι ήταν απαραίτητη μια νέα σχεδίαση η οποία θα χρησιμοποιούσε κάποιες τεχνικές του RainbowCrack αλλά θα βασίζεται σε διαφορετική φιλοσοφία και θα χρησιμοποιούσε κάποιες νέες τεχνικές για να ξεπεράσει τους περιορισμούς που επιβάλλει η αρχιτεκτονική CUDA και να ικανοποιούνται όλες οι απαιτήσεις που έθεσα. Όμως η σχεδίασή μου εισάγει μερικούς περιορισμούς στο μέγεθος των μέγιστων κωδικών που μπορούν να χρησιμοποιηθούν. Συγκεκριμένα για τον LM hash το μέγιστο μήκος είναι 14 χαρακτήρες ενώ για τους MD5 και SHA-1 είναι 8 χαρακτήρες. Το μέγιστο μήκος των 14 χαρακτήρων στον αλγόριθμο LM hash επιβάλλεται από τον περιορισμό που επιβάλει ο ίδιος ο αλγόριθμος ότι δέχεται κωδικούς με μέγιστο μήκος 14 χαρακτήρων. Για τους δύο άλλους αλγορίθμους το μέγιστο μήκος των 8 χαρακτήρων χρησιμοποιήθηκε για να είναι εύκολα διαχειρίσιμοι από τον compiler οι 8-byte αριθμοί που δημιουργούν τους κωδικούς αφού κάθε ένα byte δημιουργεί ένα χαρακτήρα.

Σε μια εφαρμογή σε CUDA κυρίαρχο ρόλο παίζουν τα τμήματα που μπορούν να εκτελεστούν παράλληλα τα οποία μπορούν να ανατεθούν για εκτέλεση στην κάρτα γραφικών που παρέχει μεγαλύτερη επεξεργαστική ισχύ από τον κεντρικό επεξεργαστή. Τα τμήματα όμως που εκτελούνται σειριακά δεν μπορούν να αξιοποιήσουν τις δυνατότητες της κάρτας γραφικών και γι’ αυτό πρέπει να ανατεθούν στον κεντρικό επεξεργαστή ο οποίος είναι γρηγορότερος για σειριακό κώδικα.. Με αυτήν την υβριδική μέθοδο πετυχαίνουμε την μέγιστη συνολική απόδοση της εφαρμογής καθώς σε κάθε συσκευή αναθέτουμε τις εργασίες που μπορεί να εκτελέσει καλύτερα.

Στην παρούσα εφαρμογή έχουμε μονάχα ένα τμήμα που μπορεί να εκτελεστεί παράλληλα, την κατασκευή των αλυσίδων, καθώς επαναλαμβάνουμε την ίδια διαδικασία πολλές φορές πάνω σε διαφορετικά δεδομένα. Επομένως όλος ο σχεδιασμός πρέπει να βασίζετε στην επίτευξη του στόχου της παράλληλης κατασκευής των αλυσίδων στην κάρτα γραφικών. Απώτερος σκοπός είναι να κρατιούνται απασχολημένες όλες οι λειτουργικές της μονάδες όση περισσότερη ώρα γίνεται.

Η επόμενη απόφαση για τον σχεδιασμό αφορά τον τρόπο που θα γίνει η παράλληλη επεξεργασία. Καθώς η τιμή εισόδου κάθε κρίκου είναι η έξοδος του προηγούμενου κρίκου, υπάρχει εξάρτηση μέσα στις αλυσίδες και δεν είναι εφικτή η τμηματοποίηση της εργασίας με αυτόν τον τρόπο. Ο μόνος τρόπος είναι η τμηματοποίηση της δουλειάς σε επίπεδο αλυσίδων καθώς τα αρχικά σημεία που τροφοδοτούν κάθε αλυσίδα είναι ανεξάρτητα μεταξύ τους. Με αυτόν τον τρόπο μπορούμε να πετύχουμε πλήρης παραλληλοποίηση καθώς σε μια πραγματική εφαρμογή ο αριθμός των αλυσίδων που θα πρέπει να δημιουργηθούν είναι της τάξης των χιλιάδων.

Η εφαρμογή θα χωριστεί σε τρία τμήματα. Στο πρώτο μέρος θα γίνεται η δημιουργία όλων των δεδομένων που χρειάζονται για την κατασκευή τους οι αλυσίδες. Στο δεύτερο μέρος

θα γίνεται η κατασκευή των αλυσίδων και στο τρίτο τμήμα η κατασκευή των πινάκων ουράνιου τόξου. Το πρώτο και το τρίτο τμήμα θα ανατεθεί για εκτέλεση στον κεντρικό επεξεργαστή ενώ το δεύτερο τμήμα, που εφαρμόζεται παράλληλη επεξεργασία, θα ανατεθεί στην κάρτα γραφικών. Στην συνέχεια θα εξηγήσω αναλυτικά τον σχεδιασμό κάθε τμήματος.

4.2.2 Σχεδιασμός τμήματος κατασκευής παραμέτρων αλυσίδων

Στο πρώτο τμήμα, που εκτελείται στον κεντρικό επεξεργαστή διαμορφώνονται όλες οι παράμετροι οι οποίες ορίζονται άμεσα ή έμμεσα από τον χρήστη. Ο χρήστης εισάγει παραμέτρους που αφορούν τον κρυπτογραφικό αλγόριθμο που θα χρησιμοποιηθεί, το πλήθος και το μήκος των αλυσίδων και τέλος τα σύμβολα και το εύρος μήκους των κωδικών.

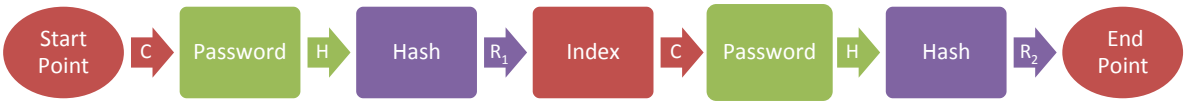
Ένας ιδανικός πίνακας πρέπει να δημιουργεί κατάλληλο αριθμό από κωδικούς με συγκεκριμένο μήκος ώστε η αναλογία που αντιστοιχεί στο συγκεκριμένο μήκος ως προς το συνολικό αριθμό των δυνατών κωδικών να παραμένει ίδια. Με άλλα λόγια ο λόγος του αριθμού αλυσίδων με συγκεκριμένο μήκος ως προς τον συνολικό αριθμό αλυσίδων να είναι ίδιος με τον λόγο των κωδικών που υπάρχουν με συγκεκριμένο μήκος ως προς τον συνολικό αριθμό κωδικών για όλα τα επιτρεπτά μήκη. Για παράδειγμα αν θέλουμε να δημιουργήσουμε ένα πίνακα με κωδικούς που χρησιμοποιούν 32 διαφορετικά σύμβολα μήκους τεσσάρων και πέντε χαρακτήρων οι κωδικοί με μήκος πέντε χαρακτήρων πρέπει να είναι 32 φορές περισσότεροι από τους κωδικούς με μήκος τεσσάρων χαρακτήρων. Αυτό ισχύει γιατί οι δυνατοί κωδικοί με μήκος τεσσάρων χαρακτήρων είναι 32^4 ενώ των πέντε χαρακτήρων είναι 32^5 δηλαδή η αναλογία τους είναι $32^5/32^4= 32$. Ο τρόπος που επέλεξα να λύσω αυτό το πρόβλημα ήταν με χρήση της σωστής αναλογίας των αλυσίδων που παράγουν ένα μόνο μήκος κωδικών και επιπλέον με την διαδοχική χρήση κάθε μήκους. Σε συνέχεια του προηγούμενου παραδείγματος αν έπρεπε να κατασκευάσω 64 αλυσίδες τότε οι πρώτες 2 από αυτές θα κατασκευάζονταν ώστε να παράγουν κωδικούς με μήκος 4 χαρακτήρων και οι επόμενες 60 με μήκος 5 χαρακτήρων. Η λύση αυτή επιλέχθηκε για να ελαττώσουμε όσο το δυνατόν τα διαφορικά μονοπάτια της εκτέλεσης μέσα σε ένα warp. Είναι πολύ αποτελεσματική καθώς αξιοποιεί την διαδοχική χρήση κάθε μήκους με αποτέλεσμα διαφορετικά μονοπάτια να μπορούν να προκύψουν μόνο όταν αλλάζει το μήκος των κωδικών. Αν το πλήθος των αλυσίδων που παράγουν ένα συγκεκριμένο μήκος κωδικού δεν αντιστοιχεί σε ακέραιο τότε το πλήθος των αλυσίδων που χρησιμοποιούνται για να παράγουν το συγκεκριμένο μήκος είναι ο αμέσως επόμενος ακέραιος. Με αυτόν τον τρόπο κατασκευάζονται κωδικοί για κάθε μήκος από τουλάχιστον μία αλυσίδα. Για την δημιουργία των αρχικών σημείων χρησιμοποιούνται τυχαίοι αριθμοί που παράγονται από συνάρτηση παραγωγής τυχαίων αριθμών. Με την χρήση τυχαίων αριθμών είναι εφικτή η κατασκευή πολλών πινάκων με ίδιες παραμέτρους αλλά κρυπτογραφώντας διαφορετικούς κωδικούς.

4.2.3 Σχεδιασμός τμήματος Κατασκευής Αλυσίδων

Η κατασκευή των αλυσίδων, που πραγματοποιείται στην κάρτα γραφικών, αποτελεί το τμήμα της εφαρμογής στο οποίο αφιερώνεται ο περισσότερος χρόνος εκτέλεσης και επομένως η βελτίωση στην απόδοση του συμβάλει σε πολύ μεγάλο βαθμό στην συνολική βελτίωση της απόδοσης της εφαρμογής. Για την κατασκευή των αλυσίδων χρησιμοποιούνται οι παράμετροι που δημιουργούνται στο προηγούμενο μέρος. Η διαφοροποίηση των αλυσίδων συντελείται με την χρήση διαφορετικών αρχικών σημείων ενώ οι υπόλοιπες παράμετροι , ο αλγόριθμος κρυπτογράφησης, το μήκος των αλυσίδων και τα σύμβολα που θα περιέχουν οι κωδικοί είναι κοινοί για όλες τις αλυσίδες.

Σε κάθε αλυσίδα υπάρχει ένας αριθμός από κρίκους οι οποίοι αποτελούν το μήκος της αλυσίδας. Σε κάθε κρίκο εκτελούνται τρεις εργασίες με την ακόλουθη σειρά: Αρχικά γίνεται η μετατροπή της αριθμητικής τιμής σε κωδικό με την συνάρτηση μετατροπής. Στη συνέχεια γίνεται η κρυπτογράφηση του κωδικού και τέλος η μετατροπή της σύνοψης σε αριθμητική τιμή με την συνάρτηση ανακατασκευής. Κάθε αλυσίδα ξεκινάει με έναν κρίκο στον οποίο σαν είσοδος χρησιμοποιείται το αρχικό σημείο. Όλοι οι ενδιάμεσοι κρίκοι παίρνουν σαν είσοδο την έξοδο

του προηγούμενου κρίκου ενώ η έξοδος του τελευταίο κρίκου είναι το τελικό σημείο. Το αρχικό και το τελικό σημείο είναι τα μόνα δεδομένα που αποθηκεύονται στον πίνακα ουράνιου τόξου από κάθε αλυσίδα.



Σχήμα 4.1 *Παράδειγμα αλυσίδας που περιέχει δύο κρίκους*

Για την κρυπτογράφηση χρησιμοποιείται ένας από τους ακόλουθους κρυπτογραφικούς αλγόριθμους κατακερματισμού, ο LM hash, ο MD5 ή ο SHA-1. Για την ανακατασκευή των κωδικών χρησιμοποιώ την ίδια τεχνική που χρησιμοποιεί και το RainbowCrack δηλαδή χρήση διαφορετικής συνάρτησης ανακατασκευής για κάθε αλυσίδα με την πρόσθεση στην αριθμητική τιμή που αντιστοιχεί η σύνοψη του αριθμού της στήλης.

Καθώς στόχος της εφαρμογής είναι η αποδοτική κατασκευή των πινάκων ουράνιου τόξου πρέπει να χρησιμοποιηθεί μια διαφορετική συνάρτηση μετατροπής από αυτήν που χρησιμοποιεί η υλοποίηση του RainbowCrack. Η υλοποίηση του RainbowCrack χρησιμοποιεί διαδοχικές διαιρέσεις του 64-bit αριθμού που παράγει η συνάρτηση ανακατασκευής με το πλήθος των συμβόλων μέχρι να δημιουργηθούν όλοι οι χαρακτήρες του κωδικού. Καθώς οι διαιρέσεις πραγματοποιούνται με πολύ μεγάλα νούμερα εισάγεται μη ανεκτή καθυστέρηση στην CUDA που διαθέτει αργές λειτουργικές μονάδες υλοποίησης της διαίρεσης. Για την αντιμετώπιση αυτού του προβλήματος χρησιμοποιώ μια διαφορετική συνάρτηση μετατροπής στην οποία κάθε ένα από τα 8 bytes της σύνοψης αντιστοιχεί σε έναν χαρακτήρα του κωδικού. Καθώς όμως η τιμή κάθε byte έχει εύρος από 0 μέχρι 255 δεν μπορεί να χρησιμοποιηθεί άμεσα ως ο χαρακτήρας του κωδικού καθώς για ορισμένες τιμές θα αντιστοιχεί σε μη επιτρεπτά σύμβολα. Έτσι χρησιμοποιείται ένας πίνακας που περιέχει όλα τα επιτρεπτά σύμβολα του κωδικού και από κάθε byte παίρνουμε το σύμβολο που υπάρχει στην θέση που προκύπτει από το υπόλοιπο της τιμής του byte με το μέγεθος του πίνακα. Για να έχουν όλα τα σύμβολα την ίδια πιθανότητα επιλογής απαιτείται η χρήση πινάκων που το μέγεθός τους είναι πολλαπλάσιο του 256. Για τον αλγόριθμο LM hash που όπως έχουμε αναφέρει στην ενότητα 1.4.1 δημιουργείται με δύο ανεξάρτητες κρυπτογραφήσεις και επομένως κάθε παραβίαση μπορεί να παραβιαστεί ανεξάρτητα. Καθώς το μέγιστο μήκος κωδικού που χρησιμοποιεί κάθε κρυπτογράφηση είναι 7 χαρακτήρες τα 8 bytes της σύνοψης επαρκούν για την παραβίαση για κάθε μήκος κωδικού και αφού οι κρυπτογραφήσεις είναι ανεξάρτητες μπορούν να παραβιαστούν κωδικοί με όλα τα επιτρεπτά μήκη που επιτρέπει ο συγκεκριμένος αλγόριθμος. Αντίθετα στους αλγορίθμους MD5 και SHA-1 το μήκος των κωδικών περιορίζεται στους 8 χαρακτήρες.

4.2.4 Σχεδιασμός τμήματος κατασκευής πίνακα ουράνιου τόξου

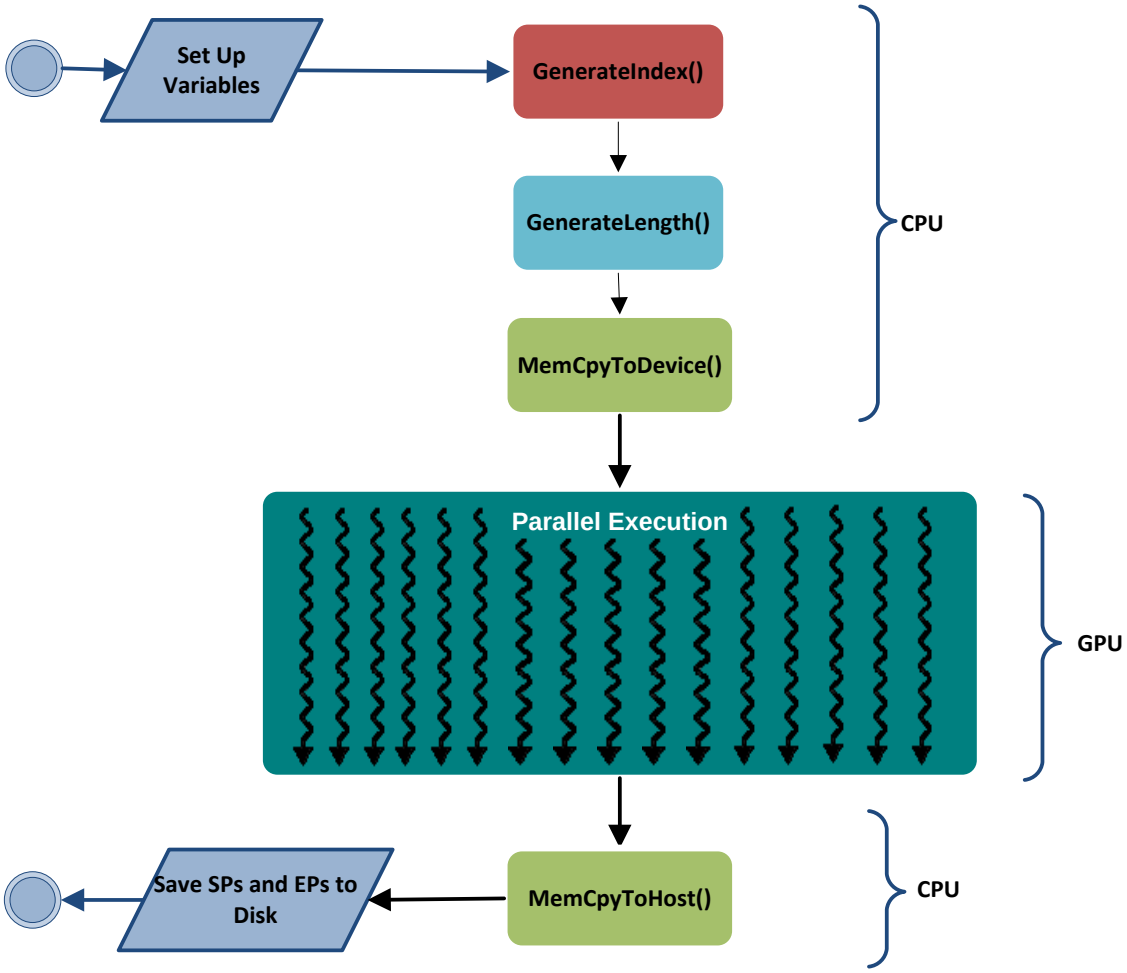
Στο τελευταίο τμήμα της εφαρμογής γίνεται η αποθήκευση του πίνακα ουράνιου τόξου σε ένα αρχείο στον δίσκο. Ο πίνακας ουράνιου τόξου αποτελείται από τα ζευγάρια των αρχικών και τελικών σημείων για όλες τις αλυσίδες. Εκτός όμως από τα αρχικά και τελικά σημεία στην αρχή του αρχείου αποθηκεύονται όλες οι παράμετροι κατασκευής του πίνακα. Οι παράμετροι αυτοί είναι απαραίτητοι από το πρόγραμμα παραβίασης που χρησιμοποιεί του πίνακες για την παραβίαση κωδικών.

4.3 Υλοποίηση σε CUDA

Στόχος της πρώτης έκδοσης σε CUDA είναι η υλοποίηση μιας πλήρους παράλληλης εφαρμογής με αξιοποίηση όλων των επεξεργαστών ροής για κάθε πολυεπεξεργαστή που διαθέτει η κάρτα γραφικών. Στις επόμενες εκδόσεις θα γίνουν βελτιστοποιήσεις τις παρούσας έκδοσης με χρήση γρηγορότερων μορφών μνήμης και χρήση μεγαλύτερου βαθμού παραλληλισμού από το φυσικό επίπεδο που διαθέτει η παρούσα έκδοση.

4.3.1 Διάγραμμα ροής της υλοποίησης σε CUDA

Αρχικά παρουσιάζω το διάγραμμα ροής της εφαρμογής στο οποίο παρουσιάζονται τα τρία τμήματα της εφαρμογής. Το πρώτο στάδιο που εκτελείται στον κεντρικό επεξεργαστή στο οποίο κατασκευάζονται όλες οι παράμετροι που χρειάζονται για την κατασκευή των αλυσίδων. Το δεύτερο στάδιο αφορά την παράλληλη κατασκευή των αλυσίδων και τέλος το τρίτο στάδιο την αποθήκευση του πίνακα ουράνιου τόξου σε αρχείο δίσκου.



Σχήμα 4.2 *Διάγραμμα ροής της εφαρμογής CUDA.*

4.3.2 Υλοποίηση τμήματος κατασκευής παραμέτρων αλυσίδων

Το πρώτο μέρος της εφαρμογής, η δημιουργία των παραμέτρων για την κατασκευή των αλυσίδων, όπως έχουμε αναφέρει θα εκτελεστεί από τον κεντρικό επεξεργαστή και έτσι μπορούμε να χρησιμοποιήσουμε βιβλιοθήκες της C. Όλες οι παράμετροι για την κατασκευή των

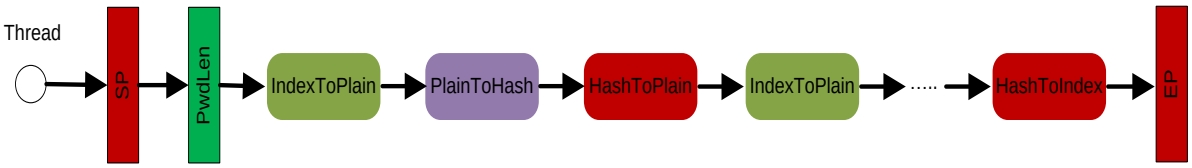
αλυσίδων περνάνε ως ορίσματα της συνάρτησης main και μετά τον έλεγχο της ορθότητα τους αποθηκεύονται σε μια δομή δεδομένων για την καλύτερη διαχείριση τους. Η δομή δεδομένων περιέχει επίσης μεταβλητές που χρησιμοποιούνται από τους κρίκους της αλυσίδας, τον κωδικό, την σύνοψη και την αριθμητική τιμή που παράγεται από την συνάρτηση ανακατασκευής. Για κάθε αλυσίδα δημιουργούμε ένα αντίγραφο της δομής.

Επίσης παράγονται τα αρχικά σημεία με μια γεννήτρια παραγωγής τυχαίων αριθμών και το μήκος κωδικού που θα δημιουργήσει κάθε αλυσίδα. Τα αποτελέσματα αποθηκεύονται σε διατάξεις στις οποίες η θέση αποθήκευσης προσδιορίζει την αλυσίδα από την οποία θα χρησιμοποιηθούν οι τιμές που περιέχουν.

4.3.3 Υλοποίηση τμήματος κατασκευής αλυσίδων

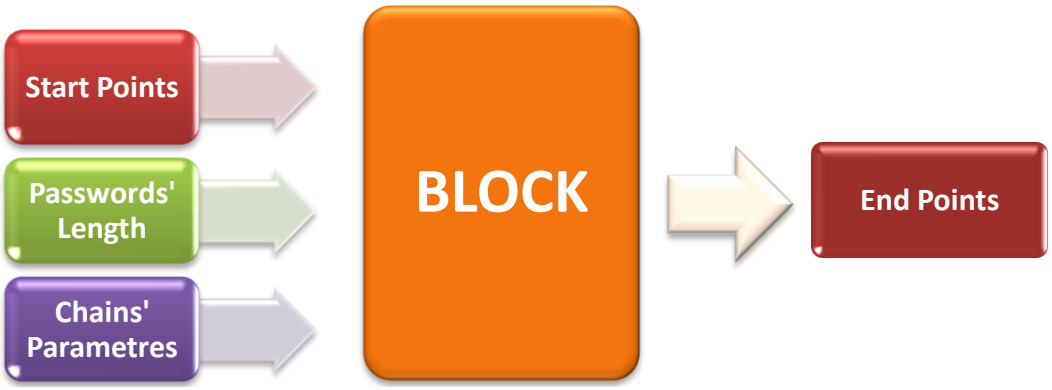
Το δεύτερο μέρος ξεκινάει με την μεταφορά των δομών και των δύο διατάξεων από την μνήμη του συστήματος στην global memory της συσκευής.

Η κατασκευή μιας μεμονωμένης αλυσίδας γίνεται εξ' ολοκλήρου από ένα thread, όμως κάθε thread ενδέχεται να αναλάβει την κατασκευή περισσότερων από μίας αλυσίδας. Σε κάθε περίπτωση όμως κάθε thread αναλαμβάνει την κατασκευή μόνο μίας αλυσίδα κάθε στιγμή και συνεχίζει στην επόμενη αφού έχει ολοκληρώσει την κατασκευή της προηγούμενης. Σαν είσοδο κάθε thread χρησιμοποιεί ένα στιγμιότυπο της δομής, ένα αρχικό σημείο και το μήκος της αλυσίδας που θα κατασκευάσει. Ο προσδιορισμός των αρχικών σημείων και του μήκους της αλυσίδας γίνεται από την θέση τους μέσα στις διατάξεις. Η έξοδος για κάθε thread είναι τα τελικά σημεία για κάθε αλυσίδα που κατασκευάζει.



Σχήμα 4.3 *Κατασκευή αλυσίδας από thread.*

Τα threads οργανώνονται σε blocks. Η είσοδος σε κάθε block είναι τα αρχικά σημεία, το μήκος του κωδικού και οι παράμετροι για όλες τις αλυσίδες που θα κατασκευαστούν από το block. Η έξοδος είναι τα τελικά σημεία για όλες τις αλυσίδες που θα κατασκευαστούν από το block.



Σχήμα 4.4 *Είσοδος και έξοδος ενός block.*

Στην πρώτη αυτή έκδοση μας ενδιαφέρει μόνο αξιοποίηση όλων των πολυεπεξεργαστών για τον λόγο αυτό χρησιμοποιούνται τόσα blocks όσο το πλήθος των πολυεπεξεργαστών που

διαθέτει η κάρτα γραφικών. Με τον τρόπο αυτόν σε κάθε ένα πολυεπεξεργαστή ανατίθεται ένα και μόνο block. Στις επόμενες εκδόσεις που θα περιέχουν βελτιστοποιήσεις ενδεχομένως να χρησιμοποιηθούν περισσότερα blocks σε κάθε πολυεπεξεργαστή για να βελτιωθεί η απόδοση.
Κάθε block αναλαμβάνει να επεξεργαστεί διαφορετικές αλυσίδες, ώστε να εκτελέσει ένα κομμάτι από την συνολική δουλειά. Για την πλήρη αξιοποίηση όλων των επεξεργαστών ροής κάθε block εκτελεί 32 threads, δηλαδή ένα warp. Όταν ολοκληρωθεί η εκτέλεση όλων των blocks, η εκτέλεση μεταφέρεται στον κεντρικό επεξεργαστή όπου πραγματοποιείται η μεταφορά της διάταξης που περιέχει τα τελικά σημεία στην μνήμη του συστήματος.

Ακολουθεί ο ψευδοκώδικας που εκτελεί κάθε block για την δημιουργία των αλυσίδων:

```
void  rainbowGenerate(int* SPoints,int* pwdLen,Chain* aChain,int* EPoints) {

        Chain cwc[blockDim.x]; //Struct with parameters for every thread of block
        int  tid=blockDim.x*blockIdx.x+threadIdx.x;
        int  totalThreads=gridDim.x*blockDim.x; //Total number of threads
        uint64  index;

        // Deep copy of parameters for every thread
        deepCopy(&cwc[threadIdx.x], aChain);

        //We cross all chains with parallel threads
        for( idx=tid; idx< aChain→rainbowNumChains; idx+=blockThreads )
        {
                //Set the input in chain
                index=SPoints[idx];
                cwc [threadIdx.x].ChainPlainLen=chainsPlainLen[idx];

                //Generation of chain
                for(int i=0;i< cwc.rainbowChainLen;i++)
                {
                        IndexToPlain(&cwc[threadIdx.x],index);   //Convert function
                        PlainToHashSHA1(&cwc[threadIdx.x]);     //Hash function
                        index=HashToIndex(&cwc[threadIdx.x], i);   //Reduction function
                }
                __syncthreads();
                //We keep output of chain
                EPoints[idx]=index;
        }
}
```

Υλοποίηση συνάρτησης ανακατασκευής

Η συνάρτηση ανακατασκευής μετατρέπει τα δύο LSBs της σύνοψης σε αριθμητική τιμή και στην συνέχεια προστίθεται ο αύξον αριθμός του κρίκου (nPos) από τον οποίον καλέστηκε η συνάρτηση.

Ακολουθεί ο ψευδοκώδικας για την συνάρτηση ανακατασκευής:

```
void HashToIndex(ChainWalkContext* aChain, int nPos)
{
        aChain->index  =( (uint64) aChain->hash[0] )<<0;
        aChain->index|=( (uint64) aChain->hash[1] )<<8;
        aChain->index|=( (uint64) aChain->hash[2] )<<16;
        aChain->index|=( (uint64) aChain->hash[3] )<<24;
        aChain->index|=( (uint64) aChain->hash[4] )<<32;
        aChain->index|=( (uint64) aChain->hash[5] )<<40;
        aChain->index|=( (uint64) aChain->hash[6] )<<48;
        aChain->index|=( (uint64) aChain->hash[7] )<<56;
        aChain->index+=nPos;
}
```

Υλοποίηση συνάρτησης μετατροπής

Η συνάρτηση μετατροπής χρησιμοποιεί ένα byte από την αριθμητική τιμή(index) που έχει σαν έξοδο η συνάρτηση ανακατασκευής για τον σχηματισμό κάθε χαρακτήρα του κωδικού. Καθώς κάθε byte έχει εύρος τιμής 0-255 και ενδεχομένως κάποιες τιμές αντιστοιχούν σε μη επιτρεπτούς χαρακτήρες χρησιμοποιούμε έναν πίνακα με όλους τους επιτρεπτούς χαρακτήρες. Για τον προσδιορισμό της θέσης μέσα στον πίνακα παίρνουμε το υπόλοιπο με το μέγεθος του πίνακα.

Ακολουθεί ο ψευδοκώδικας για την συνάρτηση μετατροπής:

```
Int  plainCharset[CHARSET_LEN]= {
                        'A','B','C','D','E','F','G','H','I','J','K',L','M','N', 'O','P','Q','R','S','T','U','V','W','X','Y','Z'
};

void IndexToPlain( uint64 index, Chain* aChain )
{
            position=(aChain->index>>0)% CHARSET_LEN; //Get position in array
            aChain->plain[0]= plainCharset[ position ];      //Get character

            position=(aChain->index>>8)% CHARSET_LEN;
            aChain->plain[1]= plainCharset[ position ];

            position=(aChain->index>>16)% CHARSET_LEN;
             aChain->plain[2]= plainCharset[ position ];

            position=(aChain->index>>24)% CHARSET_LEN;
            aChain->plain[3]=plainCharset[ position ];

            position=(aChain->index>>32)% CHARSET_LEN;
            aChain->plain[4]=plainCharset[ position ];

            position=(aChain->index>>40)% CHARSET_LEN;
            aChain->plain[5]=plainCharset[ position ];

            position=(aChain->index>>48)% CHARSET_LEN;
            aChain->plain[6]=plainCharset[ position ];

            position=(aChain->index>>56)% CHARSET_LEN;
            aChain->plain[7]=plainCharset[ position ];
}
```

Υλοποίηση Κρυπτογραφικών Αλγορίθμων

Για την υλοποίηση των κρυπτογραφικών αλγορίθμων κατακερματισμού MD5 και SHA-1 χρησιμοποίησα τον κώδικα από το OpenSSL. Καθώς όμως το OpenSSL είναι ένα project που προορίζεται για διάφορα λειτουργικά συστήματα και αρχιτεκτονικές επεξεργαστών πραγματοποίησα μια σειρά από παρεμβάσεις ώστε να απαλλαγώ από τα περιττά κομμάτια και να συγκεντρώσω τον κώδικα σε λίγες συναρτήσεις και αρχεία αντί για δεκάδες που χρησιμοποιεί η αρχική υλοποίηση.

Η υλοποίηση του αλγόριθμου LM hash δεν υπάρχει στο OpenSSL, καθώς όμως υπάρχει η υλοποίηση του DES χρειάστηκε να υλοποιήσω μόνο το τμήμα που διαμορφώνει τον κωδικό. Ομοίως με τους MD5 και SHA-1 συγκέντρωσα τον απαραίτητο κώδικα σε λίγες συναρτήσεις και αρχεία. Όσο αφορά το κομμάτι της δικής μου υλοποίησης χρησιμοποίησα μια παραλλαγή του LM hash η οποία εκμεταλλεύεται την μία από τις δύο αδυναμίες, που είχα περιγράψει στο κεφάλαιο 1.5.1, κατά την οποία οι κωδικοί με μήκος μεγαλύτερο των 7 χαρακτήρων χωρίζονται και κρυπτογραφούνται ξεχωριστά. Επομένως στην παραλλαγή αυτή χρησιμοποιώ μόνο μια κρυπτογράφηση DES και η σύνοψη που παράγεται έχει μήκος 64 bits.

4.3.4 Υλοποίηση τμήματος κατασκευής πίνακα ουράνιου τόξου

Στο τρίτο μέρος γίνεται η αποθήκευση των ζευγαριών με τα αρχικά και τα τελικά σημεία

κάθε αλυσίδας. Για τις αρχικές τιμές χρησιμοποιούμε την διάταξη που υπάρχει στην μνήμη του συστήματος, ενώ για τα τελικά σημεία χρησιμοποιούμε την διάταξη που μεταφέρθηκε από την συσκευή στην μνήμη του συστήματος στο προηγούμενο μέρος. Η αποθήκευση γίνεται αυξάνοντας σταδιακά την θέση από την οποία διαβάζουμε δεδομένα μέσα από τις διατάξεις.

Έλεγχος Ορθής Λειτουργίας

Για την ορθή λειτουργία του αλγορίθμου LM hash σύγκρινα τις συνόψεις για κοινές εισόδους στην υλοποίησή μου και σε μια γεννήτρια παραγωγής συνόψεων για LM hash στο διαδίκτυο[23]. Καθώς χρησιμοποιώ μια παραλλαγή του LM hash για τον έλεγχο χρησιμοποίησα μόνο το πρώτο μέρος της σύνοψης που παράγεται από την γεννήτρια. Για τους αλγόριθμους MD5 και SHA-1 χρησιμοποίησα τις ίδιες δοκιμές που χρησιμοποιούσε ο πρωτότυπος κώδικας στην υλοποίηση του OpenSSL. Για τον έλεγχο ολόκληρης της εφαρμογής υλοποίησα μια έκδοση γραμμένη σε C που εκτελείται εξ' ολοκλήρου στην CPU και επιπλέον χρησιμοποιεί τις κρυπτογραφικές συναρτήσεις από την βιβλιοθήκη του OpenSSL και χρησιμοποίησα ίδιες παραμέτρους και ίδια αρχικά σημεία για πολλά παραδείγματα.

Απόδοση

Οι χρόνοι κατασκευής πινάκων ουράνιου τόξου για την πρώτη έκδοση σε CUDA για κάθε κρυπτογραφικό αλγόριθμο με κωδικούς που περιέχουν 62 σύμβολα και μήκος 3 έως 7 χαρακτήρων φαίνονται στον παρακάτω πίνακα. Οι χρόνοι αυτοί θα χρησιμοποιηθούν για την εκτίμηση της βελτιστοποίησης που θα πραγματοποιήσω στην επόμενη ενότητα.

Αλγόριθμος Κρυπτογράφησης	Χρόνος Κατασκευής (sec)
LM hash	747,360
MD5	525,120
SHA-1	630,410

Πίνακας 4.1 *Χρόνοι κατασκευής των πινάκα με 37200 αλυσίδες των 100.000 κρίκων χωρίς βελτιστοποίηση*

4.4 Βελτιστοποιήσεις της υλοποίησης σε CUDA

Μετά τον αρχική έκδοση ακολουθούν μια σειρά από βελτιστοποιήσεις με σκοπό να αυξηθούν οι επιδόσεις την εφαρμογής. Όπως είχα αναφέρει και στην ενότητα 3.6 και υπενθυμίζω στο σημείο αυτό, οι βελτιστοποιήσεις χωρίζονται σε τρείς βασικές κατηγορίες που αφορούν την αύξηση του εύρους ζώνης των μνημών, την αύξηση του ρυθμού έκδοσης εντολών και την αύξηση του βαθμού παράλληλης επεξεργασίας. Για να μπορούμε να συγκρίνουμε κατά πόσο αυξάνονται οι επιδόσεις της εφαρμογής, σε όλη την φάση της βελτιστοποίησης θα χρησιμοποιήσω το ίδιο παράδειγμα κατασκευής πινάκων ουράνιου τόξου το οποίο είχα χρησιμοποιήσει και στην υλοποίηση χωρίς βελτιστοποίηση.

4.4.1 Βελτιστοποίηση μέσα από μνήμες.

Αποδοτική μεταφορά δεδομένων

Για την αποδοτική μεταφορά των δεδομένων μεταξύ των μνημών του συστήματος και της μνήμης της συσκευής θα περιορίσω των αριθμό των μεταφορών και τον όγκο των δεδομένων με την μεταφορά μόνο ενός στιγμιότυπου της δομής δεδομένων. Για όλα τα υπόλοιπα στιγμιότυπα που απαιτούνται θα δημιουργήσω αντίγραφα μέσα στην συσκευή. Για

την μεταφορά από την συσκευή στο σύστημα απαιτείται μόνο μια μεταφορά, για την διάταξη με τα τελικά σημεία, και επομένως υπάρχει επαρκής βελτιστοποίηση και δεν χρειάζεται κάτι παραπάνω.

Αξιοποίηση Κοινόχρηστης μνήμης

Στην αρχική υλοποίηση γινόταν χρήση μόνο της global memory που είναι πολύ αργή μνήμη με αποτέλεσμα να μην μπορούν να επιτευχθούν υψηλές επιδόσεις. Σε αυτό το σημείο θα χρησιμοποιήσουμε την κοινόχρηστη μνήμη που είναι πολύ πιο γρήγορη και να την αξιοποιήσουμε με τον καλύτερο δυνατό τρόπο. Να διευκρινίσω εδώ ότι δεν αρκεί να «χωρέσει» ολόκληρη η εφαρμογή μέσα στην κοινόχρηστη μνήμη, αλλά θέλουμε να χρησιμοποιηθεί από τον μεγαλύτερο δυνατό αριθμό warps για να πετύχουμε υψηλή απασχόληση.

Στην αρχική υλοποίηση χρησιμοποιούσα μια δομή με τις παραμέτρους της αλυσίδας που πολλές από αυτές είναι κοινές για όλες τις αλυσίδες και προορίζονται μόνο για ανάγνωση. Αποτέλεσμα να σπαταλείται χώρος με την επανάληψη των παραμέτρων. Για την ελάττωση των απαιτήσεων σε χώρο, ώστε να χρησιμοποιηθεί πιο αποτελεσματικά η κοινόχρηστη μνήμη, θα χρησιμοποιήσω δύο δομές όπου η μια θα περιέχει τις κοινές παραμέτρους όλων των αλυσίδων και η άλλη τις παραμέτρους του κρίκου που είναι διαφορετικές για κάθε αλυσίδα. Για κάθε block θα χρησιμοποιώ μόνο ένα στιγμιότυπο με τις κοινές παραμέτρους το οποίο θα χρησιμοποιείται για ανάγνωση από όλες τις αλυσίδες του block, ενώ το πλήθος των στιγμιότυπων με τις παραμέτρους του κρίκου θα είναι ίσο με το πλήθος των αλυσίδων μέσα στο block. Τα τελικά σημεία των αλυσίδων δεν θα αποθηκεύονται στην κοινόχρηστη μνήμη, αλλά θα μεταφέρονται άμεσα στην global memory.

Για τον LM hash στην αρχική υλοποίηση σύμφωνα με το πρωτόκολλο ,η δημιουργία των 32 υποκλειδιών και η χρήση τους για κρυπτογράφηση γίνεται σε διαφορετικές φάσεις και επομένως απαιτείται η αποθήκευσή τους. Για να μειώσω τις μεγάλες απαιτήσεις σε αποθηκευτικό χώρο πραγματοποιώ την κρυπτογράφηση μαζί με την δημιουργία των κλειδιών. Ο ελάχιστος αριθμός υποκλειδιών που χρειάζονται για κρυπτογράφηση είναι τέσσερα και επομένως έχουμε μια πολύ σημαντική μείωση σε απαιτήσεις χώρου.

Ο MD5 κρυπτογραφεί το μήνυμα αυθαιρέτου μήκους που δέχεται σαν είσοδο σε ομάδες των 512 bits. Εξαίρεση αποτελεί η τελευταία ομάδα η οποία διαμορφώνεται με τέτοιο τρόπο ώστε να εμπεριέχει το πολύ 447 bits αρχικού μηνύματος και ακολουθεί ένα bit που περιέχει την τιμή ‘1’ για τον προσδιορισμό του τέλους του μηνύματος και 64 bits που αποθηκεύουν το μήκος του αρχικού μηνύματος. Αν το μήκος του αρχικού μηνύματος είναι λιγότερο από 447 bits μετά το bit με την τιμή ‘1’ ακολουθούν bits που περιέχουν ‘0’ ώστε η τελευταία ομάδα να αποκτά πάντα μήκος 512 bits. Στην δική μας περίπτωση που χρησιμοποιούμε μήνυμα με μέγιστο μήκος 8 bytes (καθώς το μέγιστο μήκος κωδικού είναι 8 χαρακτήρες) δηλαδή 64 bits η επεξεργασία του μηνύματος γίνεται από μία μόνο ομάδα που αποτελεί συγχρόνως την τελευταία. Καθώς η επεξεργασία του μηνύματος γίνεται με 32-bit ακέραιους και γνωρίζουμε ποια bits θα είναι μηδέν και επομένως και ποιοι ακέραιοι δεν χρειάζεται να δεσμεύουμε κοινόχρηστη μνήμη για την αποθήκευση τους. Ο αριθμός των ακεραίων για τον οποίο πρέπει να δεσμεύσουμε χώρο είναι 4 καθώς απαιτούνται 3 ακέραιοι για το μήνυμα και το bit με τιμή ‘1’ σύνολο 65 bits. Ο $4^{ο}$ ακέραιος χρησιμοποιείται για την αποθήκευση του μήκος του μηνύματος. Παρά το γεγονός ότι για τον προσδιορισμό του μήκους του μηνύματος χρειάζονται 64 bits δηλαδή 2 ακέραιοι εντούτοις καθώς το μήκος είναι πάντα μικρότερο από 64 bits μπορεί να αναπαρασταθεί με μόνο ένα ακέραιο.

Ο SHA-1 παρότι χωρίζει το αρχικό μήνυμα σε ομάδες ακριβώς με τον ίδιο τρόπο όπως ο αλγόριθμος MD5 δεν μπορεί να πραγματοποιηθεί ελάττωση των ακεραίων καθώς κάθε μία από τις 80 πράξεις χρησιμοποιεί μια 32-bit λέξη επέκταση της αρχικής ομάδας των 512 bits. Καθώς ο κώδικας που χρησιμοποιεί το OpenSSL περιέχει βελτιστοποιήσεις και δεν ακολουθεί πιστά το πρωτόκολλο οι επεκτάσεις πραγματοποιούνται μέσα σε κάθε μια από τις 80 πράξεις και επομένως απαιτούνται και τα 512 bits του αρχικού μηνύματος.

Αξιοποίηση της constant memory

Επόμενη βελτιστοποίηση είναι η χρήση της constant memory για την αποθήκευση διαφόρων πινάκων που χρησιμοποιεί ο αλγόριθμος LM hash, όπως ο πίνακας με τον αριθμό των μετατοπίσεων κατά την κατασκευή των υποκλειδιών και η διάταξη που αποθηκεύει το αρχικό κείμενο(plaintext) με την οποία γίνεται η κρυπτογράφηση του κλειδιού.

Αποδοτικές προσπελάσεις στις μνήμες

Επόμενο βήμα είναι η βελτιστοποίηση για να μπορούν να είναι αποδοτικές οι προσβάσεις στην μνήμη του συστήματος και στην κοινόχρηστη μνήμη. Οι προσβάσεις στην μνήμη του συστήματος, αφορούν τις δύο διατάξεις και την δομή που περιέχει τις κοινές παραμέτρους των αλυσίδων. Τόσο οι διατάξεις, όσο και η δομή δεν επιδέχονται βελτιστοποίηση, καθώς οι διατάξεις περιέχουν τύπους της CUDA (οι διατάξεις είναι ήδη βελτιστοποιημένες) ενώ η δομή έχει μέγεθος μεγαλύτερο από 128 bytes και απαιτούνται όπως είχα αναφέρει στην ενότητα 3.6.2 περισσότερες συναλλαγές ανά half-warp λόγω της αρχιτεκτονικής της μνήμης.

Για την κοινόχρηστη μνήμη όμως μπορούμε να πετύχουμε μεγάλο βαθμό βελτιστοποίησης με την χρήση διαδοχικών ακεραίων 32-bits αντί για διαδοχικούς χαρακτήρες 8-bits στην σύνοψη και τον κωδικό, ώστε κάθε ακέραιος να προσπελαύνεται από ένα bank της μνήμης. Σαν επακόλουθο μπορούν να πραγματοποιηθούν επιπλέον βελτιστοποιήσεις καθώς όλοι οι κρυπτογραφικοί αλγόριθμοι διαχειρίζονται τους κωδικούς σαν ακεραίους και επομένως δεν απαιτείται η μετατροπή από ακέραιους σε χαρακτήρες και αντίστροφα.

Σύγκριση Αποδόσεων

Αλγόριθμος	**Χρόνος Χωρίς Βελτιστοποίηση (sec)**	**Χρόνος με βελτιστοποίηση στις μνήμες (sec)**	**Βελτιστοποίηση Βήματος (φορές)**
LM hash	747,360	161,74	4,6
MD5	525,120	52,240	10,0
SHA-1	630,410	85,960	7,3

Πίνακας 4.2 *Χρόνος κατασκευής των πινάκων με βελτιστοποίηση στις μνήμες*

4.4.2 Βελτιστοποίηση μέσα από αύξηση του ρυθμού έκδοσης εντολών

Γνωρίζοντας την σπουδαιότητα που έχει για τις επιδόσεις ο ρυθμός έκδοσης εντολών, όλη μου η σχεδίαση βασίστηκε στην εκπλήρωση αυτού του στόχου. Βασικές αποφάσεις κατά την διάρκεια της σχεδίασής μου, όπως η δημιουργία συγκεκριμένου μήκους κωδικών σε κάθε αλυσίδα και η ομαδοποίηση τους, επιτρέπει στο μεγαλύτερο μέρος της εκτέλεσης να ακολουθεί ένα κοινό μονοπάτι. Επιπλέον με τον σχεδιασμό είναι δυνατό να μην χρησιμοποιήσω καμία διαίρεση όταν ο αριθμός των συμβόλων είναι δύναμη του δύο ενώ σε αντίθετη περίπτωση χρησιμοποιώ μόνο μία διαίρεση για κάθε χαρακτήρα κωδικού.

Σε αυτήν φάση μπορώ να πραγματοποιήσω κάποιες μικρές αλλαγές που θα αυξήσουν τον ρυθμό έκδοσης εντολών. Η πρώτη αλλαγή είναι η χρήση πινάκων αναζήτησης που περιέχουν τα υπόλοιπα της διαίρεσης των αριθμών 0-255 με το πλήθους των συμβόλων του κωδικού. Με αυτόν τον τρόπο αποφεύγεται η χρονοβόρα πράξη του υπολοίπου. Υπενθυμίζω ότι οι τιμές 0-255 είναι οι πιθανές τιμές ενός byte της σύνοψης που χρησιμοποιείται για την επιλογή ενός συμβόλου στον κωδικό.

Ακολουθεί ο ψευδοκώδικας για την βελτιστοποιημένη συνάρτηση μετατροπής.

```
 __constant__ int LookUp256mod36[256]={
            0, 1, 2, 3, 4, 5, 6, 7, 8, 9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,
            30,31,32,33,34,35, 0, 1, 2, 3,4, 5, 6, 7, 8, 9,10,11,12,13, 14,15,16,17,18,19,20,21,22,23,
            24,25,26,27,28,29,30,31,32,33,34,35, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,10,11,12,13,14,15,16,17,
            18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,10,11,
            12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35, 0, 1, 2, 3, 4, 5,
             6, 7, 8, 9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,
             0, 1, 2, 3, 4, 5, 6, 7, 8, 9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,
            30,31,32,33,34,35, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,
            24,25,26,27,28,29,30,31,32,33,34,35, 0, 1, 2, 3
            };

Int  plainCharset[36]= {
                        'A','B','C','D','E','F','G','H','I','J','K',L','M','N', 'O','P','Q',
                        'R','S','T','U','V','W','X','Y','Z,'0','1','2','3','4','5','6','7','8','9'
};

void IndexToPlain( uint64 index,Chain* aChain )
{
            position=LookUp256mod36[((index>> 0)&0xFF)]; //Get position using the lookup table
            aChain ->plain[0] =plainCharset[ position ]; //Get valid character

            position=LookUp256mod36[((index>> 8)&0xFF)];
            aChain ->plain[1]= plainCharset[ position ];

            position=LookUp256mod36[((index>>16)&0xFF)];
            aChain ->plain[2]=plainCharset[ position ];

            position=LookUp256mod36[((index>>24)&0xFF)];
            aChain ->plain[3]=plainCharset[ position ];

            position=LookUp256mod36[((index>>32)&0xFF)];
            aChain ->plain[4] =plainCharset[ position ];

            position=LookUp256mod36[((index>>40)&0xFF)];
            aChain ->plain[5]= plainCharset[ position];

            position=LookUp256mod36[((index>>48)&0xFF)];
            aChain ->plain[6]= plainCharset[ position ];

            position=LookUp256mod36[((index>>56)&0xFF)];
            aChain ->plain[7]= plainCharset[ position ];
}
```

Η δεύτερη αλλαγή πραγματοποιείται με την εκμετάλλευση της δεύτερης αδυναμίας του LM hash στην οποία τα μικρά γράμματα μετατρέπονται σε κεφαλαία. Για την αποφυγή της παραπάνω μετατροπής χρησιμοποιώ πίνακες που δεν περιέχουν ως σύμβολα τα μικρά γράμματα ακόμα και στην περίπτωση που ο χρήστης ζητήσει την χρήση τους.

Σύγκριση Αποδόσεων

Αλγόριθμος	Χρόνος Χωρίς Βελτιστοποιήση (sec)	Χρόνος με προηγούμενες βελτιστοποιήσεις (sec)	Χρόνος με βελτιστοποίηση στον ρυθμό έκδοσης εντολών (sec)	Βελτιστοποίηση με αύξηση του ρυθμού έκδοσης εντολών (φορές)	Συνολική Βελτιστοποίηση (φορές)
LM hash	747,360	161,74	139,64	1,16	5,4
MD5	525,120	52,240	43,95	1,19	12,0
SHA-1	630,410	85,960	77,77	1,11	8,1

Πίνακας 4.3 *Χρόνος κατασκευής του πινάκων με βελτιστοποίηση του ρυθμού έκδοσης εντολών*

Βελτιστοποίηση μέσα από παράλληλη επεξεργασία

Όπως έχω αναφέρει και νωρίτερα η χρήση μεγαλύτερου βαθμού παραλληλισμού από τον φυσικό παραλληλισμό που επιτρέπει η αρχιτεκτονική της συσκευής μπορεί να βελτιώσει τις αποδόσεις. Αυτό είναι εφικτό καθώς επικαλύπτονται οι καθυστερήσεις με εναλλαγή των δεδομένων που επεξεργάζονται οι λειτουργικές μονάδες.

Για την επίτευξη του μέγιστου δυνατού παραλληλισμού χρησιμοποιούμε ένα εργαλείο που λέγεται «CUDA Occupancy calculator». Σε αυτό το εργαλείο εισάγουμε τους περιοριστικούς παράγοντες της απασχόλησης που είναι οι απαιτήσεις σε κοινόχρηστη μνήμη και καταχωρητές ανά block. Με βάση τους παραπάνω παράγοντες το εργαλείο υπολογίζει τον βαθμό απασχόλησης και εμφανίζει τον αριθμό των ενεργών warps σε κάθε πολυεπεξεργαστή και το πλήθος των blocks από τον οποίον προέρχονται. Καθώς όμως η υψηλότερη απασχόληση δεν σημαίνει απαραίτητα καλύτερη απόδοση θα ελέγξω ένα μεγάλο πλήθος συνδυασμών από threads ανά block και blocks ανά πολυεπεξεργαστή στην πράξη πώς επιτυγχάνεται η καλύτερη απόδοση. Υπενθυμίζω ότι για βέλτιστη απόδοση ο αριθμός των threads πρέπει να είναι πολλαπλάσιο του μεγέθους των warps.

Για τον αλγόριθμο LM hash οι σταθερές απαιτήσεις στην κοινόχρηστη μνήμη είναι 4096 bytes για την αποθήκευση δύο διατάξεων με τιμές που χρειάζονται στην κρυπτογράφηση και 108 bytes για την αποθήκευση της δομής που περιέχει παραμέτρους του κρίκου κάθε αλυσίδας. Οι απαιτήσεις σε καταχωρητές είναι 28 ανά thread. Επιπλέον κάθε ενεργό warp χρειάζεται 768 bytes κοινόχρηστης μνήμης. Με την βοήθεια του εργαλείου η μεγαλύτερη απασχόληση που μπορεί να επιτευχθεί είναι 47% όταν χρησιμοποιηθεί ένα block ανά πολυεπεξεργαστή και 480 threads ανά block. Όμως οι μετρήσεις έδειξαν ότι ο καλύτερη επίδοση επιτυγχάνεται για ένα block ανά πολυεπεξεργαστή και 256 threads ανά block με απασχόληση 28%.

Για τον αλγόριθμο MD5 οι σταθερές απαιτήσεις στην κοινόχρηστη μνήμη είναι 108 bytes για την αποθήκευση της δομής που περιέχει παραμέτρους του κρίκου κάθε αλυσίδας, ενώ οι απαιτήσεις σε καταχωρητές είναι 27 ανά thread. Επιπλέον κάθε ενεργό warp χρειάζεται 1920 bytes κοινόχρηστης μνήμης. Με την βοήθεια του εργαλείου η μεγαλύτερη απασχόληση που μπορεί να επιτευχθεί είναι 25% για διαφόρους συνδυασμούς. Μετά από μετρήσεις κατέληξα ότι οι καλύτερες επιδόσεις επιτυγχάνονται χρησιμοποιώντας δύο blocks ανά πολυεπεξεργαστή και 128 threads ανά block με απασχόληση 25%.

Για τον αλγόριθμο SHA-1 οι σταθερές απαιτήσεις στην κοινόχρηστη μνήμη είναι 108 bytes για την αποθήκευση της δομής που περιέχει παραμέτρους του κρίκου κάθε αλυσίδας. Επιπλέον κάθε ενεργό warp χρειάζεται 1920 bytes κοινόχρηστης μνήμης ενώ οι απαιτήσεις σε καταχωρητές είναι 32 ανά thread. Με την βοήθεια του εργαλείου η μεγαλύτερη απασχόληση που μπορεί να επιτευχθεί είναι 50% για διαφόρους συνδυασμούς. Μετά από μετρήσεις κατέληξα ότι οι καλύτερες επιδόσεις επιτυγχάνονται για 4 blocks ανά πολυεπεξεργαστή και 128 threads ανά block με απασχόληση 50%.

Σύγκριση Αποδόσεων

Αλγόριθμος	Χρόνος Χωρίς Βελτιστοποιήση (sec)	Χρόνος με προηγούμενες βελτιστοποιήσεις (sec)	Χρόνος με βελτιστοποίηση στον ρυθμό έκδοσης εντολών (sec)	Βελτιστοποίηση με αύξηση του ρυθμού έκδοσης εντολών (φορές)	Συνολική Βελτιστοποίηση (φορές)
LM hash	747,360	139,64	35,460	3,9	21,0
MD5	525,120	43,95	9,012	4,9	58,3
SHA-1	630,410	77,77	17,284	4,5	36,5

Πίνακας 4.6 *Χρόνος κατασκευής του πινάκων με βελτιστοποίηση του βαθμού παραλληλισμού.*

5. Επιδόσεις

Στο κεφάλαιο αυτό παρουσιάζω ένα μεγάλο πλήθος μετρήσεων που αφορούν την κατασκευή πινάκων ουράνιου τόξου με την υλοποίηση σε CUDA. Στην συνέχεια ακολουθούν μετρήσεις για τους ίδιους πίνακες για διάφορες υλοποιήσεις ώστε να μπορούμε να συγκρίνουμε τις επιδόσεις όσο πιο αντικειμενικά γίνεται. Οι υλοποιήσεις που θα χρησιμοποιήσω είναι η σειριακή έκδοση που εκτελείται από τον κεντρικό επεξεργαστή της έκδοσης CUDA, η σειριακή έκδοση που εκτελείται από τον κεντρικό επεξεργαστή της έκδοσης CUDA αλλά με χρήση της βιβλιοθήκης OpenSSL για τους κρυπτογραφικούς αλγορίθμους κατακερματισμού, η έκδοση C++ του RainbowCrack και τέλος η υλοποίηση του Κ.Θεοχαρούλη με χρήση της πλατφόρμα FPGA. Στη συνέχεια συγκρίνω τον αριθμό των κρυπτογραφήσεων στην μονάδα του χρόνου για κάθε κρυπτογραφικό αλγόριθμο για όλες τις υλοποιήσεις. Η σύγκριση αυτή έχει νόημα καθώς ο χρόνος κατασκευής των πινάκων είναι ανεξάρτητος από τις παραμέτρους των αλυσίδων.

5.1 Μεθοδολογία Μέτρησης Επιδόσεων

Όλες οι μετρήσεις σε CUDA έγιναν με την κάρτα γραφικών της NVIDIA, GTX280, μια κάρτα που διαθέτει 30 πολυεπεξεργαστές εγκατεστημένη στον server του εργαστηρίου Μικροεπεξεργαστών και Υλικού του Πολυτεχνείου Κρήτης. Ο compiler που χρησιμοποιήθηκε ήταν ο nvcc στην έκδοση 2.3.

Όλες οι μετρήσεις σε CPU έγιναν σε PC με επεξεργαστή AMD Athlon 64 3800++ Venice Core που διαθέτει 1.25 GB μνήμης RAM στο λειτουργικό σύστημα Ubuntu 9.04. Ο compiler που χρησιμοποιήθηκε ήταν ο g++ στην έκδοση 4.3.3. Η βιβλιοθήκη που χρησιμοποιήθηκε στο OpenSSL ήταν στην έκδοση 0.9.8.

Για τις μετρήσεις θα χρησιμοποιήσω δύο ομάδες κωδικών. Η πρώτη ομάδα θα έχει μήκος 3 έως 7 χαρακτήρες αποτελούμενη από 36 σύμβολα και η δεύτερη θα έχει μήκος 4 έως 7 χαρακτήρες αποτελούμενη από 64 σύμβολα. Οι πίνακες που θα κατασκευάσουν αυτούς τους κωδικούς θα ομαδοποιούνται από την πιθανότητα επιτυχίας που προσφέρουν για κάθε ομάδα. Καθώς η πιθανότητα επιτυχίας για έναν πίνακα με δεδομένα χαρακτηριστικά κωδικών εξαρτάται από δύο παραμέτρους, τον αριθμό των αλυσίδων και το μήκος τους, μπορούμε να κατασκευάσουμε περισσότερους από έναν πίνακες με την ίδια πιθανότητα επιτυχίας με διαφορετικές παραμέτρους. Για να δούμε τι επίπτωση στον χρόνο κατασκευής των πινάκων έχουν οι διάφοροι παράμετροι, θα κατασκευάζω δύο πίνακες για κάθε πιθανότητα επιτυχίας. Είναι προφανές ότι ο χρόνος κατασκευής δεν επηρεάζεται από το είδος των συμβόλων αλλά ενδεχομένως να επηρεάζεται από το πλήθος τους. Για τον λόγο αυτόν δεν αναφέρω ποια σύμβολα χρησιμοποιώ αλλά επισημαίνω ότι πρέπει να είναι διαφορετικά μεταξύ τους για να ισχύουν οι πιθανότητες επιτυχίας. Επιπλέον για να έχουμε όμως την ίδια πιθανότητα επιτυχίας για όλες τις κρυπτογραφήσεις ώστε να είναι ευκολότερη η σύγκριση θεωρώ ότι τα σύμβολα που εισάγουμε στον αλγόριθμο LM hash είναι επιλεγμένα κατάλληλα ώστε κανένα γράμμα να μην υπάρχει συγχρόνως σαν κεφαλαίο και σαν μικρό. Επιτρέπεται να είναι μόνο κεφαλαίο ή μόνο μικρό.

5.2 Χρόνοι κατασκευής πινάκων ουράνιου τόξου με CUDA

Για πίνακες που περιέχουν κωδικούς με 36 διαφορετικά σύμβολα εύρους 3 έως 7 χαρακτήρων οι χρόνοι κατασκευής είναι οι ακόλουθοι:

m (Αριθμός αλυσίδων)	**t (Αριθμός κρίκων ανά αλυσίδα)**	**N (Πλήθος κωδικών)**	**P (Πιθανότητα επιτυχίας)**	**Μέγεθος πίνακα (MB)**	**Χρόνος κατασκευής πίνακα (ώρες)**		
					LM hash	**MD5**	**SHA1**
2.000.000	75.000	$8,06X10^{10}$	0.73	32	0,48	0,12	0,23
10.000.000	15.000	$8,06X10^{10}$	0.73	160	0,48	0,12	0,23
5.000.000	50.000	$8,06X10^{10}$	0.85	80	0,80	0,20	0,39
20.000.000	12.500	$8,06X10^{10}$	0.85	320	0,80	0,20	0,39
10.000.000	150.000	$8,06X10^{10}$	0.99	160	4,81	1,22	2,35
50.000.000	30.000	$8,06X10^{10}$	0.99	800	4,81	1,22	2,35

Πίνακας 5.1 *Χρόνος κατασκευής πινάκων με κωδικούς 36 σύμβολων μήκους 3 έως 7 με CUDA*

Για πίνακες που περιέχουν κωδικούς με 64 διαφορετικά σύμβολα εύρους 4 έως 7 χαρακτήρων οι χρόνοι κατασκευής είναι οι ακόλουθοι:

m (Αριθμός αλυσίδων)	**t (Αριθμός κρίκων ανά αλυσίδα)**	**N (Πλήθος κωδικών)**	**P (Πιθανότητα επιτυχίας)**	**Μέγεθος πίνακα (MB)**	**Χρόνος κατασκευής πίνακα (ώρες)**		
					LM hash	**MD5**	**SHA1**
20.000.000	300.000	$4,47X10^{12}$	0.64	320	19,2	4,9	9,4
100.000.000	60.000	$4,47X10^{12}$	0.64	1600	19,2	4,9	9,4
50.000.000	180.000	$4,47X10^{12}$	0.75	800	28,8	7,3	14,0
150.000.000	60.000	$4,47X10^{12}$	0.75	2400	28,8	7,3	14,0
100.000.000	875.000	$4,47X10^{12}$	0.99	1600	280,5	70,9	136,5
250.000.000	350.000	$4,47X10^{12}$	0.99	4000	280,5	70,9	136,5

Πίνακας 5.2 *Χρόνος κατασκευής πινάκων για 64 σύμβολα με μήκος χαρακτήρων 3-7 με CUDA*

Ο μέσος όρων κρυπτογραφήσεων ανά δευτερόλεπτο για κάθε κρυπτογραφικό αλγόριθμο δίνεται στον ακόλουθο πίνακα:

Αλγόριθμος	**Κρυπτογραφήσεις ανά δευτερόλεπτο**
LM hash	86.575.000
MD5	341.766.000
SHA-1	177.906.000

Πίνακας 5.3 *Ρυθμός κρυπτογράφησης για CUDA*

5.3 Χρόνοι κατασκευής πινάκων στον κεντρικό επεξεργαστή της σειριακής έκδοσης CUDA

Για να μπορέσουμε να συγκρίνουμε τις επιδόσεις της υλοποίησης σε CUDA κατασκεύασα ένα πρόγραμμα που εκτελείται στον κεντρικό επεξεργαστή με τον αντίστοιχο κώδικα που χρησιμοποίησα στην έκδοση CUDA.

Για πίνακες που περιέχουν κωδικούς με 36 διαφορετικά σύμβολα εύρους 3 έως 7 χαρακτήρων οι χρόνοι κατασκευής είναι οι ακόλουθοι:

m (Αριθμός αλυσίδων)	**t (Αριθμός κρίκων ανά αλυσίδα)**	**N (Πλήθος κωδικών)**	**P (Πιθανότητα επιτυχίας)**	**Μέγεθος πίνακα (MB)**	**Χρόνος κατασκευής πίνακα (ώρες)**		
					LM hash	**MD5**	**SHA1**
2.000.000	75.000	$8{,}06X10^{10}$	0.73	32	24,4	14,4	21,6
10.000.000	15.000	$8{,}06X10^{10}$	0.73	160	24,4	14,4	21,6
5.000.000	50.000	$8{,}06X10^{10}$	0.85	80	40,7	23,9	36,0
20.000.000	12.500	$8{,}06X10^{10}$	0.85	320	40,7	23,9	36,0
10.000.000	150.000	$8{,}06X10^{10}$	0.99	160	244,1	143,7	216,1
50.000.000	30.000	$8{,}06X10^{10}$	0.99	800	244,1	143,7	216,1

Πίνακας 5.4 *Χρόνος κατασκευής πινάκων με κωδικούς 36 σύμβολων μήκους 3 έως 7 για την υλοποίηση σε CPU*

Για πίνακες που περιέχουν κωδικούς με 64 διαφορετικά σύμβολα εύρους 4 έως 7 χαρακτήρων οι χρόνοι κατασκευής είναι οι ακόλουθοι:

m (Αριθμός αλυσίδων)	**t (Αριθμός κρίκων ανά αλυσίδα)**	**N (Πλήθος κωδικών)**	**P (Πιθανότητα επιτυχίας)**	**Μέγεθος πίνακα (MB)**	**Χρόνος κατασκευής πίνακα**		
					LM hash	**MD5**	**SHA1**
20.000.000	300.000	$4{,}47X10^{12}$	0.64	320	965,6	563,6	854,7
100.000.000	60.000	$4{,}47X10^{12}$	0.64	1600	965,6	563,6	854,7
50.000.000	180.000	$4{,}47X10^{12}$	0.75	800	1448,4	845,4	1282,1
150.000.000	60.000	$4{,}47X10^{12}$	0.75	2400	1448,4	845,4	1282,1
100.000.000	875.000	$4{,}47X10^{12}$	0.99	1600	14082,0	8219,0	12464,0
250.000.000	350.000	$4{,}47X10^{12}$	0.99	4000	14082,0	8219,0	12464,0

Πίνακας 5.5 *Χρόνος κατασκευής πινάκων για 64 σύμβολα με μήκος χαρακτήρων 3-7 για την υλοποίηση σε CPU*

Ο μέσος όρων κρυπτογραφήσεων ανά δευτερόλεπτο για κάθε κρυπτογραφικό αλγόριθμο στους παραπάνω πίνακες είναι:

Αλγόριθμος	**Κρυπτογραφήσεις (δευτερόλεπτο)**
LM hash	1.716.000
MD5	2.898.000
SHA-1	1.939.000

Πίνακας 5.6 *Ρυθμός κρυπτογράφησης για την υλοποίηση σε CPU*

5.4 Χρόνοι κατασκευής πινάκων στον κεντρικό επεξεργαστή της σειριακής έκδοσης CUDA με βιβλιοθήκες

Για να μπορέσουμε να συγκρίνουμε τις επιδόσεις της υλοποίησης σε CUDA κατασκεύασα ένα πρόγραμμα που εκτελείται σε CPU με τον αντίστοιχο κώδικα που χρησιμοποίησα στην έκδοση CUDA αλλά με χρήση της βιβλιοθήκης τους OpenSSL για τους κρυπτογραφικούς αλγορίθμους κατακερματισμού.

Για πίνακες που περιέχουν κωδικούς με 36 διαφορετικά σύμβολα εύρους 3 έως 7 χαρακτήρων οι χρόνοι κατασκευής είναι οι ακόλουθοι:

m (Αριθμός αλυσίδων)	**t (Αριθμός κρίκων ανά αλυσίδα)**	**N (Πλήθος κωδικών)**	**P (Πιθανότητα επιτυχίας)**	**Μέγεθος πίνακα (MB)**	**Χρόνος κατασκευής πίνακα (ώρες)**		
					LM hash	**MD5**	**SHA1**
2.000.000	75.000	$8{,}06X10^{10}$	0.73	32	26,9	18,7	21,8
10.000.000	15.000	$8{,}06X10^{10}$	0.73	160	26,9	18,7	21,8
5.000.000	50.000	$8{,}06X10^{10}$	0.85	80	44,8	31,1	36,4
20.000.000	12.500	$8{,}06X10^{10}$	0.85	320	44,8	31,1	36,4
10.000.000	150.000	$8{,}06X10^{10}$	0.99	160	269,1	187,0	218,7
50.000.000	30.000	$8{,}06X10^{10}$	0.99	800	269,1	187,0	218,7

Πίνακας 5.7 *Χρόνος κατασκευής πινάκων με κωδικούς 36 σύμβολων μήκους 3 έως 7 για την υλοποίηση σε CPU με χρήση της βιβλιοθήκης του OpenSSL*

Για πίνακες που περιέχουν κωδικούς με 64 διαφορετικά σύμβολα εύρους 4 έως 7 χαρακτήρων οι χρόνοι κατασκευής είναι οι ακόλουθοι:

m (Αριθμός αλυσίδων)	**t (Αριθμός κρίκων ανά αλυσίδα)**	**N (Πλήθος κωδικών)**	**P (Πιθανότητα επιτυχίας)**	**Μέγεθος πίνακα (MB)**	**Χρόνος κατασκευής πίνακα (ώρες)**		
					LM hash	**MD5**	**SHA1**
20.000.000	300.000	$4{,}47X10^{12}$	0.64	320	1067,7	739,7	864,9
100.000.000	60.000	$4{,}47X10^{12}$	0.64	1600	1067,7	739,7	864,9
50.000.000	180.000	$4{,}47X10^{12}$	0.75	800	1610,5	1109,6	1297,4
150.000.000	60.000	$4{,}47X10^{12}$	0.75	2400	1610,5	1109,6	1297,4
100.000.000	875.000	$4{,}47X10^{12}$	0.99	1600	15571,0	10788,0	12613,0
250.000.000	350.000	$4{,}47X10^{12}$	0.99	4000	15571,0	10788,0	12613,0

Πίνακας 5.8 *Χρόνος κατασκευής πινάκων για 64 σύμβολα με μήκος χαρακτήρων 3-7 για την υλοποίηση σε CPU με χρήση της βιβλιοθήκης του OpenSSL*

Ο μέσος όρων κρυπτογραφήσεων ανά δευτερόλεπτο για κάθε κρυπτογραφικό αλγόριθμο στους παραπάνω πίνακες είναι:

Αλγόριθμος	**Κρυπτογραφήσεις (δευτερόλεπτο)**
LM hash	1.554.000
MD5	2.240.000
SHA-1	1.916.000

Πίνακας 5.9 *Ρυθμός κρυπτογράφησης για την υλοποίηση σε CPU με χρήση της βιβλιοθήκης του OpenSSL*

5.5 Χρόνοι κατασκευής πινάκων στον κεντρικό επεξεργαστή του RainbowCrack

Για να συγκρίνουμε τις επιδόσεις πιο αντικειμενικά χρησιμοποιήσαμε το project RainbowCrack στην έκδοση σε C++. Οι επιδόσεις της έκδοσης του RainbowCrack σε CUDA δεν είναι δημοσιοποιημένες ώστε να μπορέσω να συγκρίνω.

Για πίνακες που περιέχουν κωδικούς με 36 διαφορετικά σύμβολα εύρους 3 έως 7 χαρακτήρων οι χρόνοι κατασκευής είναι οι ακόλουθοι:

m (Αριθμός αλυσίδων)	**t (Αριθμός κρίκων ανά αλυσίδα)**	**N (Πλήθος κωδικών)**	**P (Πιθανότητα επιτυχίας)**	**Μέγεθος πίνακα (MB)**	**Χρόνος κατασκευής πίνακα (ώρες)**		
					LM hash	**MD5**	**SHA1**
2.000.000	75.000	$8{,}06X10^{10}$	0.73	32	33,9	27,4	30,6
10.000.000	15.000	$8{,}06X10^{10}$	0.73	160	33,9	27,4	30,6
5.000.000	50.000	$8{,}06X10^{10}$	0.85	80	56,5	45,7	51,0
20.000.000	12.500	$8{,}06X10^{10}$	0.85	320	56,5	45,7	51,0
10.000.000	150.000	$8{,}06X10^{10}$	0.99	160	339,3	274,3	305,9
50.000.000	30.000	$8{,}06X10^{10}$	0.99	800	339,3	274,3	305,9

Πίνακας 5.10 *Χρόνος κατασκευής πινάκων με κωδικούς 36 σύμβολων μήκους 3 έως 7 για την υλοποίηση του RainbowCrack σε CPU*

Για πίνακες που περιέχουν κωδικούς με 64 διαφορετικά σύμβολα εύρους 4 έως 7 χαρακτήρων οι χρόνοι κατασκευής είναι οι ακόλουθοι:

m (Αριθμός αλυσίδων)	**t (Αριθμός κρίκων ανά αλυσίδα)**	**N (Πλήθος κωδικών)**	**P (Πιθανότητα επιτυχίας)**	**Μέγεθος πίνακα (MB)**	**Χρόνος κατασκευής πίνακα (ώρες)**		
					LM hash	**MD5**	**SHA1**
20.000.000	300.000	$4{,}47X10^{12}$	0.64	320	1357,2	1097,2	1223,7
100.000.000	60.000	$4{,}47X10^{12}$	0.64	1600	1357,2	1097,2	1223,7
50.000.000	180.000	$4{,}47X10^{12}$	0.75	800	2035,8	1645,8	1835,5
150.000.000	60.000	$4{,}47X10^{12}$	0.75	2400	2035,8	1645,8	1835,5
100.000.000	875.000	$4{,}47X10^{12}$	0.99	1600	19793	16001	17845
250.000.000	350.000	$4{,}47X10^{12}$	0.99	4000	19793	16001	17845

Ο μέσος όρων κρυπτογραφήσεων ανά δευτερόλεπτο για κάθε κρυπτογραφικό αλγόριθμο στους παραπάνω πίνακες είναι:

Αλγόριθμος	**Κρυπτογραφήσεις (δευτερόλεπτο)**
LM hash	1.228.000
MD5	1.519.000
SHA-1	1.362.000

Πίνακας 5.12 *Ρυθμός κρυπτογράφησης για την υλοποίηση του RainbowCrack σε CPU*

5.6 Χρόνοι κατασκευής πινάκων με την πλατφόρμα FPGA

Επίσης μετράμε τους χρόνους κατασκευής με την υλοποίηση του Κ.Θεοχαρούλη για τον αλγόριθμο LM hash που είναι διαθέσιμος. Στους παρακάτω χρόνους δεν συνυπολογίζεται ο χρόνος για την μεταφορά των δεδομένων στην FPGA καθώς επίσης και η μεταφορά των αποτελεσμάτων από την FPGA για την αποθήκευσή τους σε αρχείο.

Για πίνακες που περιέχουν κωδικούς με 36 διαφορετικά σύμβολα εύρους 3 έως 7 χαρακτήρων οι χρόνοι κατασκευής είναι οι ακόλουθοι:

m (Αριθμός αλυσίδων)	**t (Αριθμός κρίκων ανά αλυσίδα)**	**N (Πλήθος κωδικών)**	**P (Πιθανότητα επιτυχίας)**	**Μέγεθος πίνακα (MB)**	**Χρόνος κατασκευής πίνακα (ώρες)**
					LM hash
2.000.000	75.000	$8,06X10^{10}$	0.73	32	0,06
10.000.000	15.000	$8,06X10^{10}$	0.73	160	0,06
5.000.000	50.000	$8,06X10^{10}$	0.85	80	0,1
20.000.000	12.500	$8,06X10^{10}$	0.85	320	0,1
10.000.000	150.000	$8,06X10^{10}$	0.99	160	0,6
50.000.000	30.000	$8,06X10^{10}$	0.99	800	0,6

Πίνακας 5.13 *Χρόνος κατασκευής πινάκων με κωδικούς 36 σύμβολων μήκους 3 έως 7 για την υλοποίηση σε FPGA*

Για πίνακες που περιέχουν κωδικούς με 64 διαφορετικά σύμβολα εύρους 4 έως 7 χαρακτήρων οι χρόνοι κατασκευής είναι οι ακόλουθοι:

m (Αριθμός αλυσίδων)	**t (Αριθμός κρίκων ανά αλυσίδα)**	**N (Πλήθος κωδικών)**	**P (Πιθανότητα επιτυχίας)**	**Μέγεθος πίνακα (MB)**	**Χρόνος κατασκευής πίνακα (ώρες)**
					LM hash
20.000.000	300.000	$4,47X10^{12}$	0.64	320	2,4
100.000.000	60.000	$4,47X10^{12}$	0.64	1600	2,4
50.000.000	180.000	$4,47X10^{12}$	0.75	800	3,6
150.000.000	60.000	$4,47X10^{12}$	0.75	2400	3,6
100.000.000	875.000	$4,47X10^{12}$	0.99	1600	34,5
250.000.000	350.000	$4,47X10^{12}$	0.99	4000	34,5

Οι κρυπτογραφήσεις ανά δευτερόλεπτο για τον αλγόριθμο LM hash είναι:

Αλγόριθμος	**Κρυπτογραφήσεις (δευτερόλεπτο)**
LM hash	683.521.000

Πίνακας 5.14 *Ρυθμός κρυπτογράφησης για την υλοποίηση σε FPGA*

5.7 Συγκρίσεις των υλοποιήσεων

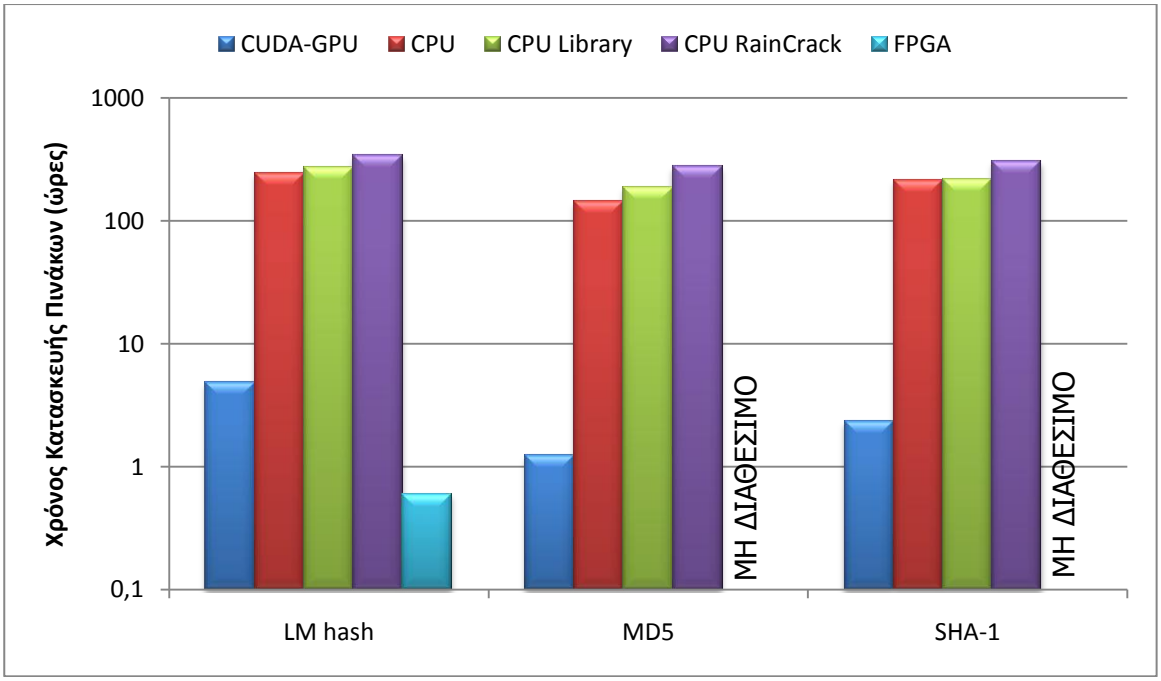
Αρχικά θα συγκρίνω τον χρόνο κατασκευής που απαιτείται για όλες τις υλοποιήσεις για τους πίνακες με 36 και 64 σύμβολα. Στην συνέχεια θα συγκρίνω τον ρυθμό κρυπτογραφήσεων και τέλος την απόδοση με βάση την υλοποίηση σε CUDA.

Πίνακες με 36 σύμβολα

Ενδεικτικά συγκρίνουμε τον χρόνο που χρειάζεται η κατασκευή ενός πίνακα με πιθανότητα επιτυχίας 0.99 με 50 εκατομμύρια αλυσίδες και 30 χιλιάδες κρίκους για 36 σύμβολα με μήκος που έχει εύρος 3-7 χαρακτήρες. Τα σύμβολα αυτά θα μπορούσαν να αναπαριστούν όλα τα κεφαλαία γράμματα και όλους τους αριθμούς. Επιπλέον ο αλγόριθμος LM hash θα μπορούσε να παραβιάζει κωδικούς που περιέχουν όλα τα γράμματα, κεφαλαία και μικρά, καθώς και όλους τους αριθμούς λόγω της αδυναμίας που παρουσιάζει.

Αλγόριθμος	GPU	CPU	CPU Library	RainbowCrack	FPGA
LM hash	4,81	244,1	269,1	339,3	0,6
MD5	1,22	143,7	187,0	274,3	-
SHA-1	2,35	216,1	218,7	305,9	-

Πίνακας 5.15 *Χρόνος κατασκευής πινάκων σε ώρες με πιθανότητα επιτυχίας 0.99 για 36 σύμβολα.*



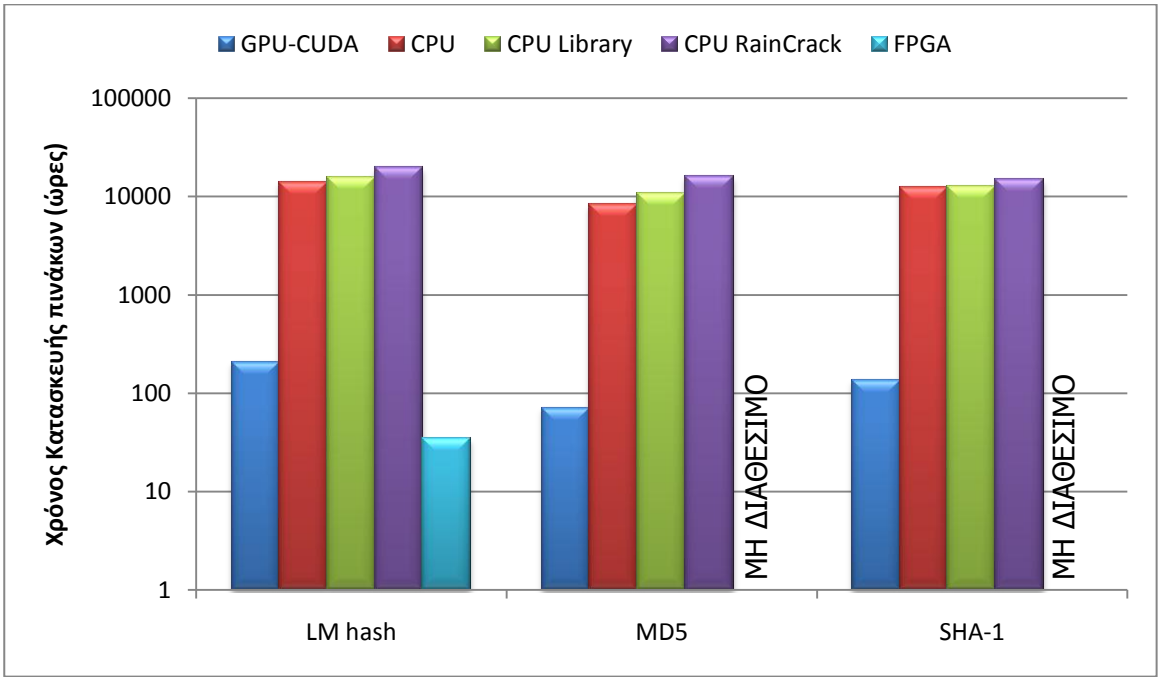
Γραφική Παράσταση 5.1 *Χρόνος κατασκευής πινάκων με πιθανότητα επιτυχίας 0.99 για 36 σύμβολα.*

Πίνακες με 64 σύμβολα

Ενδεικτικά συγκρίνουμε τον χρόνο που χρειάζεται η κατασκευή ενός πίνακα με πιθανότητα επιτυχίας 0.99, 250 εκατομμύριων αλυσίδων και 350 χιλιάδων κρίκων για 64 σύμβολα με μήκος που έχει εύρος 3-7 χαρακτήρες. Τα σύμβολα αυτά θα μπορούσαν να αναπαριστούν όλα τα γράμματα, όλους τους αριθμούς και δύο ειδικούς χαρακτήρες για τους αλγορίθμους MD5 και SHA-1 ενώ για τον αλγόριθμο LM hash θα μπορούσε να παραβιάζει όλα τα προηγούμενα και επιπλέον 26 ειδικούς χαρακτήρες.

Αλγόριθμος	GPU	CPU	CPU Library	RainbowCrack	FPGA
LM hash	208,5	14082,0	15571,0	19793,0	34,5
MD5	70,9	8219,7	10788,0	16001,0	-
SHA-1	136,5	12464,0	12613,0	17845,0	-

Πίνακας 5.16 *Χρόνος κατασκευής πινάκων σε ώρες με πιθανότητα επιτυχίας 0.99 για 64 σύμβολα.*



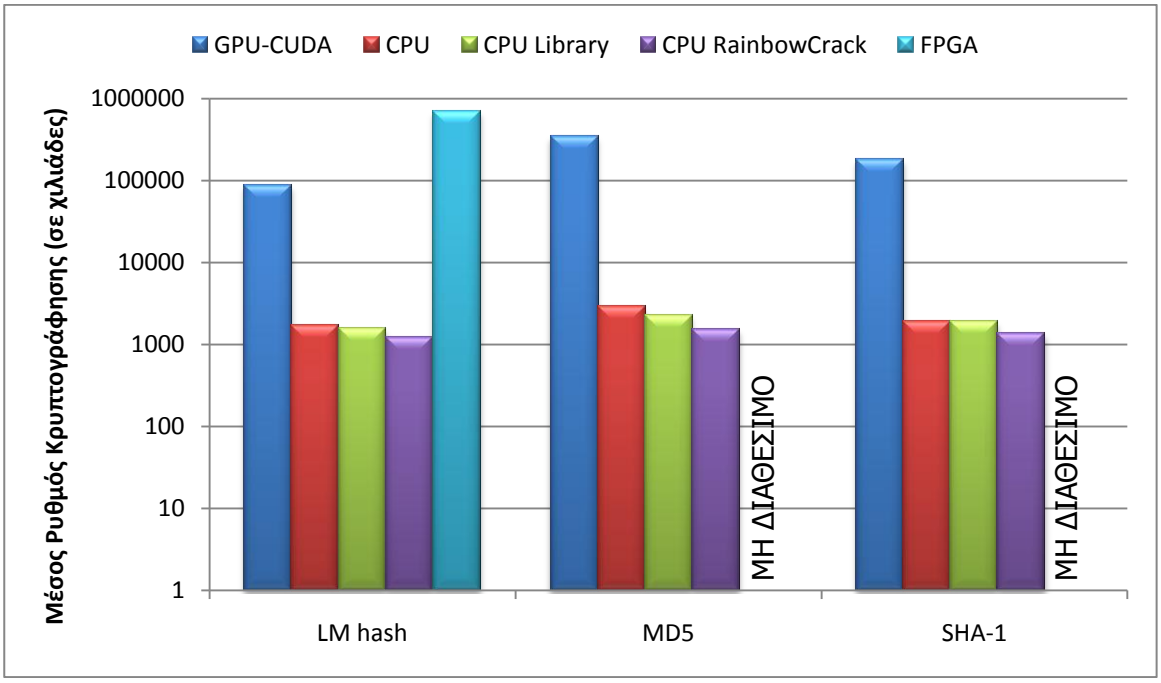
Γραφική Παράσταση 5.2 *Χρόνος κατασκευής πινάκων με πιθανότητα επιτυχίας 0.99 για 64 σύμβολα.*

5.8 Ρυθμός κρυπτογραφήσεων

Για να έχουμε μια πιο ολοκληρωμένη εικόνα συγκρίνουμε τον μέσο ρυθμό κρυπτογραφήσεων, για πίνακες 36 συμβόλων με εύρος 3-7 χαρακτήρες και για πίνακες 64 συμβόλων με εύρος 4-7 χαρακτήρες, σε ένα δευτερόλεπτο.

Αλγόριθμος	GPU	CPU	CPU Library	RainbowCrack	FPGA
LM hash	86.575.000	1.716.000	1.554.000	1.228.000	683.521.000
MD5	341.766.000	2.898.000	2.240.000	1.519.000	-
SHA-1	177.906.000	1.939.000	1.916.000	1.362.000	-

Πίνακας 5.17 *Μέσος ρυθμός κρυπτογράφησης*



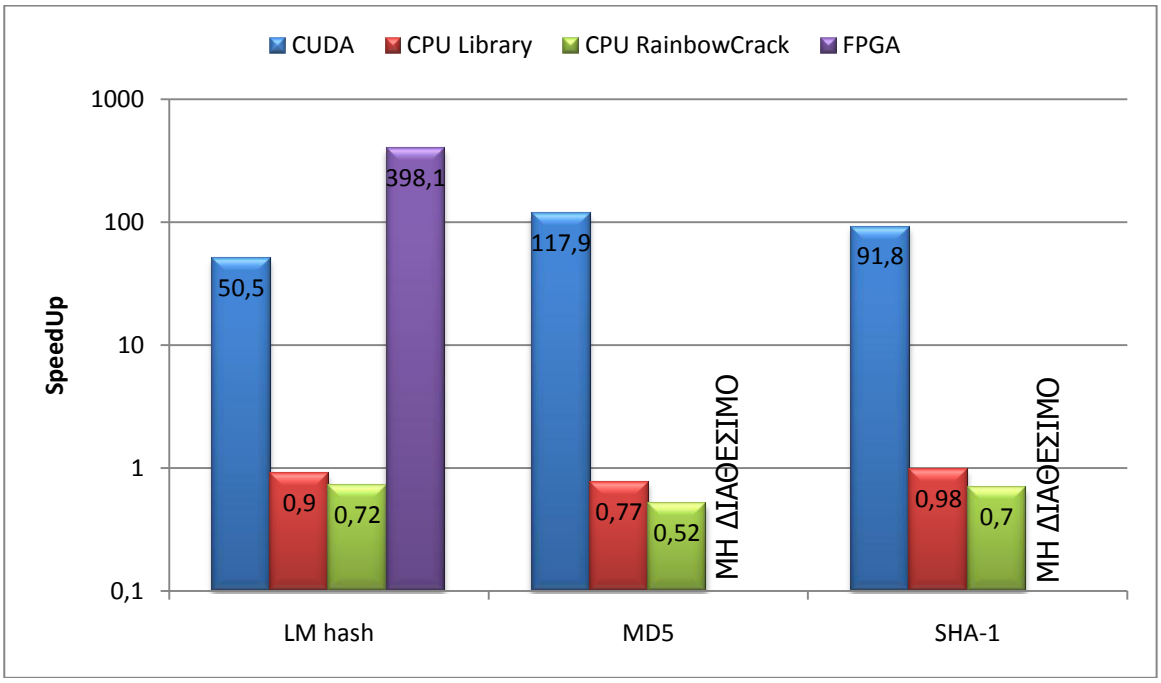
Γραφική Παράσταση *5.3* *Μέσος ρυθμός κρυπτογράφησης ανά δευτερόλεπτο*

5.9 SpeedUp

Για την σύγκριση των επιδόσεων συγκρίνω το speedup με βάση την σειριακή υλοποίηση της έκδοσης CUDA που εκτελείται στον κεντρικό επεξεργαστή.

Αλγόριθμος	CUDA	CPU Library	RainbowCrack	FPGA
LM hash	50,45	0,90	0,72	398,1
MD5	117,9	0,77	0,52	-
SHA-1	91,8	0,98	0,70	-

Πίνακας 5.18 SpeedUp με *βάση την σειριακή υλοποίηση στην CPU.*



Γραφική Παράσταση 5.4 SpeedUp με *βάση την σειριακή υλοποίηση στην CPU.*

6. Συμπεράσματα και μελλοντική δουλειά

6.1 Συμπεράσματα

Στην παρούσα εργασία προτάθηκε μια λύση για την δημιουργία των πινάκων ουράνιου τόξου με την αξιοποίηση της τεράστιας υπολογιστικής ισχύος των σύγχρονων καρτών γραφικών. Με την χρήση μιας κάρτας γραφικών η δημιουργία των πινάκων επιταχύνθηκε 50 με 118 φορές συγκρινόμενη με τους μικροεπεξεργαστές γενικού σκοπού. Η μεγάλη απόκλιση της απόδοσης οφείλεται στην διαφορετική υλοποίηση των κρυπτογραφικών αλγορίθμων κατακερματισμού. Στους αλγορίθμους όπως ο MD5 και ο SHA-1 που δίνεται μεγάλο βάρος στην επεξεργασία δεδομένων χωρίς την χρήση μεγάλων διατάξεων επετεύχθησαν οι καλύτερες επιδόσεις. Αντίθετα στον αλγόριθμο LM hash στον οποίον η επεξεργασία των δεδομένων απαιτεί την χρήση μεγάλων διατάξεων οι επιδόσεις ήταν περιορισμένες. Επιπλέον οι κάρτες γραφικών έχουν μικρό κόστος, στα ίδια επίπεδα με τους μικροεπεξεργαστές γενικού σκοπού και όλοι οι χρήστες είναι εξοικειωμένοι ώστε να μπορούν να τις χρησιμοποιήσουν. Στα αρνητικά της χρήσης καρτών γραφικών είναι οι χαμηλές επιδόσεις με σχέση τις πλατφόρμες FPGA και η ιδιαίτερες γνώσεις που απαιτούνται για την υλοποίηση εφαρμογών που μπορούν να εκτελεστούν στις κάρτες γραφικών.

Η υλοποίηση εφαρμογών σε CUDA απαιτεί αρκετό καιρό για την απόκτηση εμπειρίας ιδιαίτερα αν ο προγραμματιστής δεν διαθέτει εμπειρία σε παράλληλη επεξεργασία παρότι χρησιμοποιείται μια επέκταση της δημοφιλούς γλώσσας C. Όμως μετά την απόκτηση των βασικών γνώσεων η βελτιστοποίηση των εφαρμογών είναι σχετικά εύκολη διαδικασία καθώς ακολουθούνται συγκεκριμένα βήματα. Επίσης σε κάθε γενιά καρτών η βελτιστοποίηση γίνεται πιο εύκολη διαδικασία καθώς το υλικό τείνει να χαλαρώνει τους περιορισμούς σχετικά με την προσπέλαση των μνημών.

Επιγραμματικά μπορώ να καταλήξω στο συμπέρασμα ότι η αξιοποίηση της υπολογιστική ισχύος των καρτών είναι μια προσιτή διαδικασία για όλους τους χρήστες που μπορεί να επιφέρει μεγάλη αύξηση των επιδόσεων με πολύ μικρό κόστος.

6.2 Μελλοντικές επεκτάσεις

Στην ενότητα αυτή θα παρουσιάσω κάποιες προτάσεις για μελλοντική εργασία. Η βασικότερη πρόταση για μελλοντική δουλειά είναι η υλοποίηση δύο εφαρμογών ώστε να μπορούν να χρησιμοποιηθούν οι πίνακες ουράνιου τόξου για παραβίαση κωδικών.
Η πρώτη εφαρμογή θα πραγματοποιεί ταξινόμηση των πινάκων για την αύξηση της ταχύτητας αναζήτησης και η δεύτερη για την χρήση των ταξινομημένων πινάκων για την παραβίαση κωδικών.

Μια δεύτερη πρόταση είναι η συγχώνευση των αλυσίδων που περιέχουν κοινές τιμές και η αντικατάσταση τους από νέες αλυσίδες. Με αυτόν τον τρόπο οι πίνακες περιέχουν τον μέγιστο δυνατό αριθμό αλυσίδων με αποτέλεσμα στον ίδιο χώρο να υπάρχει μεγαλύτερη πιθανότητα επιτυχίας. Βέβαια με αυτόν τον τρόπο ο χρόνος κατασκευής που απαιτείται είναι μεγαλύτερος.

Στην παρούσα εργασία το εύρος μήκους των κωδικών με το οποίο μπορούν να κατασκευαστούν πίνακες είναι 1 μέχρι 14 χαρακτήρες για τον LM hash και 1 μέχρι 8 χαρακτήρες για τους MD5 και SHA-1. Το μέγιστο μήκος των 14 χαρακτήρων για τους κωδικούς για τον LM

hash είναι το μέγιστο δυνατό και δεν δέχεται επέκταση όμως σαν τρίτη πρόταση για μελλοντική εργασία προτείνω την αύξηση του εύρους μήκους των κωδικών στους αλγόριθμους MD5 και SHA-1.

Τέλος προτείνω την μέτρηση των επιδόσεων με χρήση της τελευταίας γενιάς καρτών γραφικών που διαθέτουν μέχρι και 480 επεξεργαστές ροής.

6.3 Επίλογος

Καταλήγοντας θέλω να αναφέρω ότι με αυτήν την εργασία πήρα πολύ σημαντικές γνώσεις για την υλοποίηση εφαρμογών που κάνουν χρήση παράλληλης επεξεργασίας δεδομένων. Θεωρώ ότι οι γνώσεις αυτές θα είναι πολύ χρήσιμες στο μέλλον καθώς η σύγχρονη τάση επιτάσσει η αύξηση των επιδέσεων σε όλων των ειδών επεξεργαστικές μονάδες να γίνεται με χρήση πολλαπλών πυρήνων.

ΠΑΡΑΡΤΗΜΑ

Οι πίνακες S-Boxes που χρησιμοποιούνται από τον κρυπτογραφικό αλγόριθμο DES.

14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
4	1	8	8	13	6	2	11	15	12	9	7	3	10	5	0
15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

Ο πίνακας S0

15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15
13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9

Ο πίνακας S1

10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8
13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1
13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7
1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12

Ο πίνακας S2

7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15
13	8	11	5	6	15	0	3	9	7	2	12	1	10	14	9
10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4
3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14

Ο πίνακας S3

},

2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9
14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6
4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14
11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3

Ο πίνακας S4

12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	11
10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8
9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6
4	3	2	12	9	5	15	10	11	14	1	7	6	0	8	13

Ο πίνακας S5

4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1
13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6
1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2
6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12

Ο πίνακας S6

13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7
1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2
7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8
2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11

Ο πίνακας S7

ΑΝΑΦΟΡΕΣ

1. Martin Hellman. A cryptanalytic time-memory trade-off. IEEE Transactions on Information Theory, 1980.

2. Philip Oechslin, Making a Faster Cryptanalytic Time-Memory Trade-Off, CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, 2003

3. Project RainbowCrack "The Time-Memory Tradeoff Hash Cracker",
http://project-rainbowcrack.com/

4. Κώστας Θεοχαρούλης, "Υλοποίηση σε αναδιατασσόμενη λογική των πινάκων ουρανίου τόξου (rainbow tables) για την αποκρυπτογράφηση των κρυπταλγόριθμων Lmhash, MD5, SHA1" ,Διπλωματική Εργασία, Πολυτεχνείο Κρήτης, 2008

5. David B. Kirk, Wen-mei W. Hwu "Programming Massively Parallel Processors: A Hands-on Approach" ISBN:978-0-12-381472-2 Elsevier, 2010

6. Graphics Processing Unit from Wikipedia, the free encyclopedia
http://en.wikipedia.org/wiki/Graphics_processing_unit

7. Rob Farber, "CUDA, Supercomputing for the Masses: Part 1"
http://www.drdobbs.com/architecture-and-design/207200659

8. "CUDA Programming Guide 2.3.1 for Windows", NVIDIA™ ,2009

9. Bruce Schneier "Applied Cryptography: Protocols, Algorithms, and Source Code in C, 2nd Edition", ISBN: 0471128457, Publisher: John Wiley & Sons, Inc., 1996

10. **Α.Πομπορτζής και Α.Τσούλφας "Εισαγωγή στο ηλεκτρονικό εμπόριο"**, ISBN: 960-8050-82-0, Εκδόσεις Τζιόλα, 2002

11. Phil Zimmermann, "An Introduction to Cryptography", PDF Edition, Network Associates, Inc. and its Affiliated Companies

12. Data Encryption Standard from Wikipedia, the free encyclopedia
http://en.wikipedia.org/wiki/Data_Encryption_Standard

13. Triple DES from Wikipedia, the free encyclopedia
http://en.wikipedia.org/wiki/Triple_DES

14. Advanced Encryption Standard from Wikipedia, the free encyclopedia
http://en.wikipedia.org/wiki/Advanced_Encryption_Standard

15. LM hash from Wikipedia, the free encyclopedia
http://en.wikipedia.org/wiki/LM_hash

16. Jonathan Knudsen, "Java cryptography", ISBN: 1-56592-402-9, O'Reilly Media, 1998

17. Dictionary Attack from Wikipedia, the free encyclopedia
http://en.wikipedia.org/wiki/Dictionary_attack

18. Dictionary Attack, TopBits.com Tech Community
http://www.topbits.com/dictionary-attack.html

19. Brute Force Attack from Wikipedia, the free encyclopedia
http://en.wikipedia.org/wiki/Brute_force_attack

20. Brute Force Attack, TopBits.com Tech Community
http://www.topbits.com/brute-force-attack.html

21. Space-time Tradeoff from Wikipedia, the free encyclopedia
http://en.wikipedia.org/wiki/Space-time_tradeoff

22. The Open Source toolkit for SSL/TLS
http://www.openssl.org/

23. NEDIAM.COM.XN
http://nediam.com.mx/winhashes/search_lm_hash.php

24. NVIDIA Corporation
http://www.nvidia.com/object/what_is_cuda_new.html

25. Custom Hardware Attack from Wikipedia, the free encyclopedia
http://en.wikipedia.org/wiki/Custom_hardware_attack

26. Cryptanalytic Computer from Wikipedia, the free encyclopedia
http://en.wikipedia.org/wiki/Cryptanalytic_computer

27. Salt Cryptography from Wikipedia, the free encyclopedia
http://en.wikipedia.org/wiki/Salt_cryptography

28. Parallel Computing from Wikipedia, the free encyclopedia
http://en.wikipedia.org/wiki/Parallel_computing