# ΑΡΙΣΤΟΤΕΛΕΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΟΝΙΚΗΣ

# ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ
ΗΛΕΚΤΡΟΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΗΛΕΚΤΡΟΝΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ


## *«ΥΠΟΛΟΓΙΣΜΟΣ ΤΟΠΙΚΩΝ ΣΥΝΤΕΛΕΣΤΩΝ ΣΥΣΧΕΤΙΣΗΣ ΕΙΚΟΝΩΝ ΣΕ ΜΟΝΑΔΕΣ ΕΠΕΞΕΡΓΑΣΙΑΣ ΓΡΑΦΙΚΩΝ»*

## ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ


ΕΚΠΟΝΗΣΗ:

ΚΑΡΑΦΥΛΛΙΑΣ ΙΩΑΝΝΗΣ ΑΕΜ 5257

ΜΑΝΩΛΗΣ ΙΣΑΑΚ ΑΕΜ 3939

ΕΠΙΒΛΕΠΩΝ:

ΠΙΤΣΙΑΝΗΣ ΝΙΚΟΛΑΟΣ
ΕΠΙΚΟΥΡΟΣ ΚΑΘΗΓΗΤΗΣ



**ΘΕΣΣΑΛΟΝΙΚΗ ΑΠΡΙΛΗΣ 2009**

# ΠΕΡΙΛΗΨΗ

Το αντικείμενο της παρούσας διπλωματικής εργασίας εντάσσεται στο πεδίο της επεξεργασίας εικόνων. Πρόκειται για ένα ταχέως αναπτυσσόμενο πεδίο της σύγχρονης υπολογιστικής που είναι στενά συνδεδεμένο με μεθόδους παράλληλης επεξεργασίας και συγκεκριμένα με τις πιο πρόσφατες προσπάθειες στην ανάπτυξη υλικού υπολογιστών που είναι οι Γενικής Χρήσης Μονάδες Επεξεργασίας Γραφικών (GPGPUs).

Ειδικότερα παρουσιάζεται η μέθοδος της συσχέτισης εικόνων. Πρόκειται για μία μέθοδο που απαντά στο πρόβλημα της αντιπαραβολής εικόνων, της αναζήτησης και εύρεσης δηλαδή κάποιου μοτίβου σε μία μεγαλύτερη εικόνα. Πρόκειται για μία προσπάθεια που συναντά πολλαπλά πεδία εφαρμογής, όπως για παράδειγμα η ανίχνευση ενός αντικειμένου, σε πραγματικό χρόνο, στα πλαίσια ενός μεταβαλλόμενου τοπίου, σε ιατρικά διαγνωστικά μηχανήματα (MRI), στην πιστοποίηση αυθεντικότητας ψηφιακών εικόνων (πχ έργα τέχνης), στην αναγνώριση προτύπου και αλλού.

Με βάση το υφιστάμενο μαθηματικό και αλγοριθμικό υπόβαθρο της συσχέτισης εικόνων, το οποίο και αναπτύσσουμε, υλοποιήσαμε κώδικα σε γλώσσα προγραμματισμού CUDA. Πρόκειται για μία γλώσσα προγραμματισμού που επεκτείνει την C με έναν επιπλέον αριθμό ειδικών εντολών για την αξιοποίηση του μεγάλου εύρους παραλληλίας των GPUs. Ο κώδικας που συντάξαμε εστιάζει σχεδόν αποκλειστικά στην αξιοποίηση και βελτιστοποίηση των δυνατοτήτων των GPUs, προσπερνώντας μια αντίστοιχη προσπάθεια που μπορεί να γίνει και για τον αλγόριθμο που εκτελεί την πράξη της συσχέτισης στη CPU. Η επιλογή αυτή έγινε με συνείδηση του γεγονότος ότι απαιτείται πολύ μεγαλύτερη προγραμματιστική προσπάθεια για επίτευξη υψηλών αποδόσεων στη CPU, χωρίς όμως και πάλι αυτές να φτάνουν την αντίστοιχη απόδοση στη GPU. Χαρακτηριστικά εκτελώντας το πρόγραμμα στην Tesla C1060 GPU στον υπολογιστή «Διάδη» του ΤΗΜΜΥ με μέγιστη θεωρητική απόδοση 624 GFlops /device και συγκριτικά προς τη CPU του Intel(R) Xeon(R) E5420 2.50GHz , που έχει μέγιστη θεωρητική απόδοση 10 GFlops /core, επιτύχαμε έως και 94 φορές επιτάχυνση και έως και 143 GFlops στη GPU. Αποτελέσματα συγκρίσιμα και για ορισμένα μεγέθη προτύπου καλύτερα από την πλέον διαδεδομένη μέθοδο επεξεργασίας εικόνων του 2D FFT.

Η διπλωματική αυτή εργασία ξεκινάει στο πρώτο κεφάλαιο με εισαγωγικά στοιχεία που αναφέρονται στην μαθηματική περιγραφή των διαδικασιών της συσχέτισης και της συνέλιξης για μονοδιάστατη και δυσδιάστατη επεξεργασία πινάκων, παρουσιάζει τυπικά μεγέθη που εμφανίζονται σε εφαρμογές επεξεργασίας εικόνων και αναδεικνύει τη σημασία της παράλληλης υπολογιστικής στο συγκεκριμένο πεδίο. Στο δεύτερο κεφάλαιο παρουσιάζονται οι μονάδες επεξεργασίας γραφικών. Αρχικά γίνεται αναφορά στις κλάσεις και κατηγορίες των παράλληλων υπολογιστικών συστημάτων, στη συνέχεια παρουσιάζεται η αρχιτεκτονική της Μονάδας Επεξεργασίας Γραφικών για να μπορέσει έτσι να παρουσιαστεί με μεγαλύτερη σαφήνεια το προγραμματιστικό μοντέλο των σύγχρονων GPUs και να καταλήξει με μία σχετικά σύντομη ιστορική αναδρομή στην εξέλιξη της παράλληλης υπολογιστικής. Στο τρίτο κεφάλαιο γίνεται λόγος για την γλώσσα προγραμματισμού CUDA που αποτέλεσε και το βασικό προγραμματιστικό εργαλείο για τους κώδικες που συντάχθηκαν για την παρούσα εργασία. Στο τέταρτο και τελευταίο κεφάλαιο αναλύονται οι συγκεκριμένοι αλγόριθμοι υλοποίησης της συσχέτισης πινάκων βάσει των οποίων γράφτηκαν οι κώδικες στην CUDA για την αναγνώριση προτύπου. Παρουσιάζονται επίσης αναλυτικοί πίνακες και διαγράμματα απόδοσης καθώς και μία εφαρμογή σε πραγματικά δεδομένα. Η διπλωματική ολοκληρώνεται με την παράθεση των προγραμμάτων στο Παράρτημα και με την σχετική Βιβλιογραφία. Οι πηγές που αναφέρονται στη βιβλιογραφία συναντώνται κατά μήκος του κειμένου υπό την ένδειξη [] , όπου δίνεται η αρίθμηση της εκάστοτε αναφοράς στη βιβλιογραφία.

Θα θέλαμε τέλος να αναφέρουμε ότι ένα μεγάλο κομμάτι αυτής της διπλωματικής, κυρίως από το προγραμματιστικό της μέρος, εκπονήθηκε στο τμήμα Computer Science του πανεπιστήμιου Duke στο Durham της North Carolina στην Αμερική, με τη βοήθεια και συνεργασία του επιβλέποντα καθηγητή κ. Νίκου Πιτσιάνη και της επιστημονικής του συνεργάτιδας Xiaobai Sun.

# ΠΕΡΙΕΧΟΜΕΝΑ

# 1. ΕΙΣΑΓΩΓΗ

Η αντιπαραβολή εικόνων (registration) είναι μια βασική διαδικασία στο πεδίο της επεξεργασίας εικόνας που συχνά χρησιμοποιείται για την ταύτιση δύο ή περισσοτέρων εικόνων που έχουν για παράδειγμα αποκτηθεί σε διαφορετικούς χρόνους, από διαφορετικούς αισθητήρες ή από διαφορετική οπτική γωνία. Ειδικά παραδείγματα συστημάτων που χρησιμοποιούν την αντιπαραβολή εικόνας είναι η ανίχνευση στόχου σε μία μεταβαλλόμενη εικόνα σε πραγματικό χρόνο, η σύνθεση στερεομετρικών εικόνων για την αποκατάσταση ενός τοπίου για αυτόνομη πλοήγηση, καθώς και πολλά ιατρικά διαγνωστικά μηχανήματα. Υπάρχει σαφής σχέση ανάμεσα στην διαδικασία της ταυτοποίησης και στις αιτίες που μπορούν να προκαλούν απόκλιση μεταξύ των διαφόρων-διαφορετικών λήψεων μιας εικόνας. Αναδεικνύονται τρεις διαφορετικοί τέτοιοι τύποι αποκλίσεων. Πρώτος είναι η μη ευθυγράμμιση (misalignment) των εικόνων που για την αντιμετώπισή της απαιτείται η εφαρμογή χωρικών μετασχηματισμών πάνω στα στοιχεία της εικόνας. Ο δεύτερος τύπος απόκλισης έχει να κάνει με διαφορές φωτισμού και ατμοσφαιρικών συνθηκών και δεν μπορεί να μοντελοποιηθεί-αντιμετωπιστεί εύκολα. Ο τρίτος τύπος αναφέρεται σε διαφορές που οφείλονται στην μετακίνηση αντικειμένων μέσα στην εικόνα. {1}

Ένα συγκεκριμένο πρόβλημα στην αντιπαραβολή εικόνων, με το οποίο ασχολείται η παρούσα εργασία, είναι η ταύτιση μοτίβου (template matching). Αυτό σημαίνει αναζήτηση και εύρεση ενός μοτίβου αναφοράς μέσα σε κάποια μεγαλύτερη εικόνα. Χαρακτηριστικά της μεθόδου είναι η βασισμένη σε μοντέλο προσέγγιση, η προεπιλογή χαρακτηριστικών, η εκ των προτέρων γνώση των ιδιοτήτων ορισμένων αντικειμένων και η υψηλού επιπέδου ταυτοποίηση. Τυπικές εφαρμογές αυτής της μεθόδου βρίσκουμε στην επεξεργασία δεδομένων που έχουν ληφθεί εξ' αποστάσεως και μέσω αισθητήρων, καθώς και στην αναγνώριση προτύπου.

Η γενική ιδέα για την επίλυση του προβλήματος της αντιπαραβολής εικόνων έγκειται στην εύρεση κάποιου μετασχηματισμού που μπορεί να συσχετίζει ένα προς ένα τα σημεία-στοιχεία μίας εικόνας με τα αντίστοιχα κάποιας άλλης. Ειδικότερα η αναγνώριση της θέσης και του προσανατολισμού ενός μοτίβου μέσα σε μία εικόνα υλοποιείται με την μέθοδο του cross-correlation.

Πριν προχωρήσουμε παραθέτουμε ορισμένες αναγκαίες στοιχειώδεις έννοιες και μαθηματικά εργαλεία όπως αυτό της συνέλιξης στη βάση του οποίου αναπτύσσεται και η μέθοδος του cross correlation. Ακόμα εξηγούμε τον λόγο για τον οποίο είναι απαραίτητες κάποιες τροποποιήσεις των δεδομένων-πινάκων.

## 1.1. Περιγραφή των Διαδικασιών Συσχέτισης και Συνέλιξης Πινάκων

Η συνέλιξη (convolution) και η συσχέτιση (correlation) είναι βασικές λειτουργίες που χρησιμοποιούνται για την εξαγωγή πληροφοριών από εικόνες. Είναι υπό μία έννοια οι απλούστερες δυνατές λειτουργίες που μπορούν να εφαρμοστούν σε μία εικόνα αλλά παρόλα αυτά είναι εξαιρετικά χρήσιμες. Επιπλέον, ακριβώς επειδή είναι απλές μπορούν να αναλυθούν και να κατανοηθούν πολύ καλά και είναι επίσης εύκολο να εφαρμοστούν και να υπολογιστούν με υψηλό βαθμό απόδοσης. Στόχος μας σε αυτό το σημείο του κειμένου είναι να εξηγήσουμε ακριβώς τι είναι η συνέλιξη και η συσχέτιση.

Οι λειτουργίες αυτές έχουν δύο βασικά χαρακτηριστικά: είναι σταθερές κατά τη μετατόπιση και είναι γραμμικές. Σταθερές κατά τη μετατόπιση σημαίνει ότι εκτελούμε την ίδια λειτουργία σε κάθε σημείο της εικόνας. Γραμμικές σημαίνει ότι μπορούμε να αντικαταστήσουμε κάθε σημείο με κάποιο γραμμικό συνδυασμό των σημείων γύρω του.

### 1.1.1. Συσχέτιση

Θα θεωρήσουμε για αρχή την απλούστερη εκδοχή αυτών των διαδικασιών μιλώντας για μονοδιάστατες εικόνες. Μπορούμε να θεωρήσουμε μία μονοδιάστατη εικόνα ως μία μοναδική γραμμή σημείων όπως φαίνεται στην παρακάτω εικόνα.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 5 | 4 | 2 | 3 | 7 | 4 | 6 | 5 | 3 | 6 |

Μία από τις πιο απλές λειτουργίες που μπορούμε να εκτελέσουμε με την συσχέτιση είναι ο τοπικός μέσος όρος. Για παράδειγμα αντικαθιστούμε κάθε σημείο μίας μονοδιάστατης εικόνας με το μέσο όρο του σημείου αυτού και των δύο γειτόνων

του. Με αυτόν τον τρόπο έχουμε μία εικόνα ως είσοδο και παράγεται μία νέα εικόνα ως έξοδος. Οφείλουμε βέβαια στο σημείο αυτό να διευκρινίσουμε το τι ακριβώς συμβαίνει στα όρια της εικόνας όπως για παράδειγμα στο πρώτο σημείο που δεν υπάρχει γειτονικό σημείο στα αριστερά του. Υπάρχουν τέσσερις τρόποι για να αντιμετωπιστεί το συγκεκριμένο θέμα.

1) Η αρχική εικόνα επεκτείνεται με μηδενικά.

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 5 | 4 | 2 | 3 | 7 | 4 | 6 | 5 | 3 | 6 | 0 | 0 |

2) Η αρχική εικόνα επεκτείνεται επαναλαμβάνοντας τις πρώτες και τελευταίες τιμές.

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 5 | 5 | 5 | 4 | 2 | 3 | 7 | 4 | 6 | 5 | 3 | 6 | 6 | 6 |

3) Η αρχική εικόνα επεκτείνεται επεκτεινόμενη κυκλικά.

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 6 | 5 | 4 | 2 | 3 | 7 | 4 | 6 | 5 | 3 | 6 | 5 | 4 |

Τέλος, μπορεί να θεωρηθεί ότι η εικόνα δεν ορίζεται πέραν των σημείων που μας έχουν δοθεί αρχικά, και έτσι δεν μπορεί να υπολογιστούν οι μέσοι όροι των στοιχείων που χρησιμοποιούνε τις μη προσδιορισμένες αυτές τιμές. Κατά αυτόν τον τρόπο η εικόνα εξόδου θα είναι μικρότερη από την εικόνα εισόδου.

Με βάση και τα παραπάνω μπορούμε να προχωρήσουμε στον μαθηματικό ορισμό της συσχέτισης. Θεωρώντας ένα φίλτρο συσχέτισης T (Template) και μία εικόνα F (Frame), αυτός είναι:

$$T \circ F(x) = \sum_{n} T(i)F(x+i) \qquad \textit{Σχέση 1}$$

Μία εξαιρετικά χρήσιμη εφαρμογή της συσχέτισης είναι η εύρεση θέσεων σε μία εικόνα που είναι όμοιες με κάποιο μοτίβο αναζήτησης. Προκειμένου να γίνει αυτό μπορεί το φίλτρο T που αναφέρθηκε παραπάνω να θεωρηθεί ως το μοτίβο αυτό. Συγκεκριμένα, μετακινώντας το T πάνω στην εικόνα αναζητούμε τη θέση όπου οι τιμές του T έρχονται σε ένα προς ένα αντιστοιχία με παρόμοιες τιμές της εικόνας.

Αρχικά πρέπει να προσδιορίσουμε έναν τρόπο για να μετράμε την ομοιότητα μεταξύ του Template και της περιοχής της εικόνας με την οποία αντιστοιχίζεται. Ένας απλός και φυσικός τρόπος για να το κάνουμε αυτό είναι να μετρήσουμε το

άθροισμα των τετραγώνων των διαφορών μεταξύ των τιμών του Template και της εικόνας. Το άθροισμα αυτό αυξάνεται καθώς αυξάνεται και η διαφορά μεταξύ των δύο.

$$\sum_n (T(i) - F(x+i))^2 = \sum_n (T^2(i) + F^2(x+i) - 2T(i)F(x+i))$$

$$= \sum_n (T^2(i)) + \sum_n F^2(x+i) - 2\sum_n T(i)F(x+i) \qquad \textit{Σχέση 2}$$

Όπως φαίνεται μπορούμε να σπάσουμε την ευκλείδεια απόσταση σε τρία μέρη. Το πρώτο εξαρτάται μόνο από τον Template, και θα είναι το ίδιο για κάθε σημείο της εικόνας. Το δεύτερο είναι το άθροισμα των τετραγώνων των σημείων της εικόνας που επικαλύπτονται με τον Template, και το τρίτο είναι δύο φορές η αρνητική τιμή της συσχέτισης μεταξύ του T και του F. Φαίνεται ότι με σταθερούς όλους του υπόλοιπους όρους όσο αυξάνει η συσχέτιση μεταξύ του T και του F τόσο μειώνεται η ευκλείδεια απόσταση μεταξύ τους. Αυτό λοιπόν παρέχει έναν δείκτη για την χρήση συσχέτισης προκειμένου να βρεθεί ένα μοτίβο πάνω σε μία εικόνα. Περιοχές όπου η συσχέτιση μεταξύ των δύο είναι μεγάλη είναι περιοχές όπου το T και το F ταιριάζουν καλά. Αυτό βέβαια καταδεικνύει και μία αδυναμία στη χρήση της συσχέτισης προκειμένου να συγκριθούν ο T και ο F. Η συσχέτιση μπορεί να είναι μεγάλη και σε θέσεις όπου η ένταση της εικόνας είναι μεγάλη, ακόμα κι αν αυτές δεν ταιριάζουν με τον T. Αυτό φαίνεται στο παρακάτω παράδειγμα.

Έστω T:

| | | |
|---|---|---|
| **3** | **7** | **5** |

Και η εικόνα F:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 2 | 4 | 1 | 3 | 8 | 4 | 0 | 3 | 8 | 0 | 7 | 7 | 7 | 1 | 2 |

Θα πάρουμε το παρακάτω αποτέλεσμα:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 40 | 43 | 39 | 34 | 64 | **85** | 52 | 27 | 61 | 65 | 59 | 84 | **105** | 75 | 38 | 27 |

Παρατηρούμε ότι προκύπτει ένα υψηλό αποτέλεσμα (85) όταν κεντράρουμε το T στο 6ο σημείο, καθώς το (3, 7, 5) ταιριάζει πολύ με το (3, 8, 4). Αλλά η υψηλότερη συσχέτιση παρατηρείται στο 13ο σημείο όπου ο T αντιστοιχίζεται στο (7, 7, 7) παρόλο που αυτό δεν μοιάζει με το (3, 7, 5). Ένας τρόπος για να ξεπεράσουμε αυτήν την αδυναμία είναι το να χρησιμοποιήσουμε απλώς το άθροισμα των τετραγώνων των διαφορών μεταξύ των F και T. Κάτι τέτοιο θα μας έδινε το εξής αποτέλεσμα:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 25 | 26 | 26 | 41 | 29 | 25 | 59 | 54 | 34 | 26 | 78 | 13 | 20 | 32 | 61 | 38 |

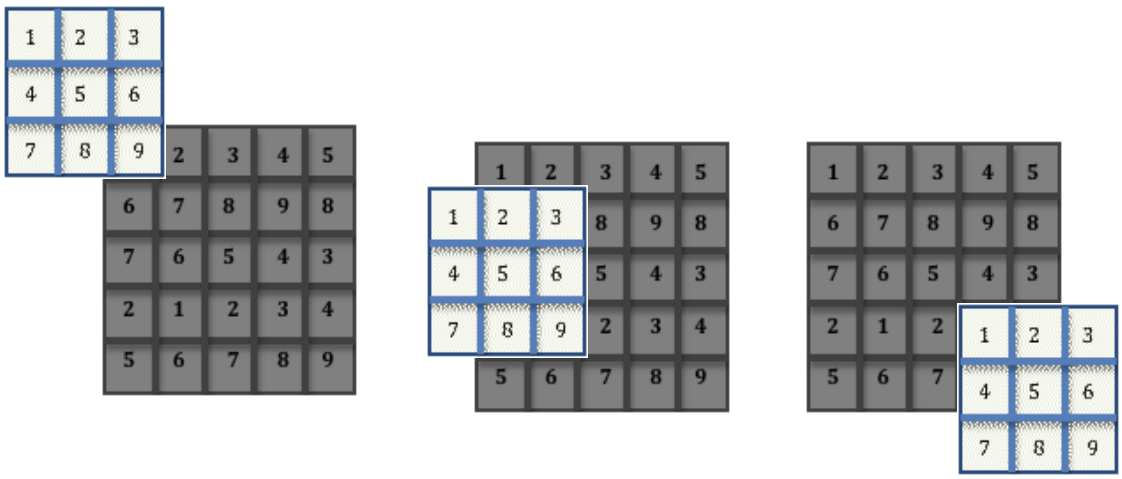
Βλέπουμε έτσι ότι χρησιμοποιώντας την ευκλείδεια απόσταση το (3, 7, 5) ταιριάζει καλύτερα με το τμήμα της εικόνας που έχει τιμές (3, 8, 4), ενώ το επόμενο καλύτερο σημείο ομοιότητας έχει τιμές (0, 7, 7) αλλά η συσχέτιση είναι σε σημαντικό βαθμό μικρότερη. Γίνεται επομένως φανερό ότι δεν αρκούν οι παραπάνω μέθοδοι για την βελτιστοποίηση των αποτελεσμάτων της συσχέτισης και παρουσιάζεται η ανάγκη μιας κανονικοποίησης των δεδομένων που εξηγείτε παρακάτω για την δυσδιάστατη συσχέτιση. [5]

#### 1.1.1.1. Δισδιάστατη Συσχέτιση

Οι εικόνες είναι δυσδιάστατοι πίνακες και ουσιαστικά στις πρακτικές εφαρμογές εφαρμόζουμε δυσδιάστατη συσχέτιση. Η βασική ιδέα είναι ίδια με αυτή του 1D. Ο μαθηματικός τύπος της δυσδιάστατης συσχέτισης είναι ο εξής:

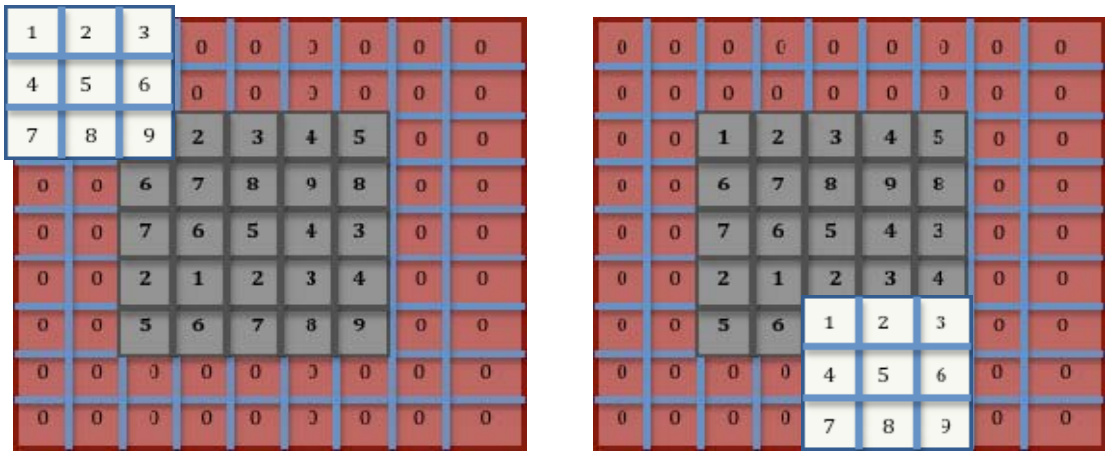
$$T \circ F(x,y) = \sum_{n} \sum_{m} T(i,j) F(x+i, y+j) \qquad \text{Σχέση 3}$$

Για να γίνει η πράξη της συσχέτισης των δύο πινάκων θα πρέπει ο πίνακας Template να μετακινηθεί σε όλες τις δυνατές θέσεις πάνω στον Frame, όπως φαίνεται ενδεικτικά στο παρακάτω σχήμα. Θεωρούμε τα εξής μεγέθη: Template 3X3 , Frame 5X5.



Σχήμα 1

Όπως παρατηρούμε ο υπολογισμός των στοιχείων στις γωνίες και συνολικά περιμετρικά του πίνακα Frame δεν μπορεί να γίνει και θα οδηγεί σε λάθη, ανάλογα με την περίπτωση του 1D. Για το λόγο αυτό καθίσταται αναγκαίο μία επέκταση του πίνακα Frame με μηδενικά στοιχεία για ένα πλήθος γραμμών και στηλών τέτοιο ώστε να είναι δυνατή η μετακίνηση του Template σε όλες τις δυνατές θέσεις.



*Σχήμα 2*

Όπως φαίνεται παραπάνω οι αναγκαίες γραμμές και στήλες που πρέπει να προστεθούν στον πίνακα Frame είναι 2 * ($M_T$ - 1) και 2 * ($N_T$-1) αντίστοιχα, μοιρασμένες εξίσου περιμετρικά του Frame (όπου $M_T$ το πλήθος των γραμμών και $N_T$ το πλήθος των στηλών του πίνακα Template).

Έτσι στη γενική περίπτωση το μέγεθος του πίνακα που προκύπτει (Pad_Frame) είναι ($M_F$ + 2 * ($M_T$-1)) γραμμές και ($N_F$ + 2 * ($N_T$-1)) στήλες (όπου $M_F$ το πλήθος των γραμμών και $N_F$ το πλήθος των στηλών του πίνακα Frame).

Ο πίνακας εξόδου της συσχέτισης (Outcome) που προκύπτει φαίνεται παρακάτω και στη γενική περίπτωση έχει ($M_F$ + $M_T$ - 1) γραμμές και ($N_F$+ $N_T$ - 1) στήλες.

| | | | | | | |
|---|---|---|---|---|---|---|
| 9 | 26 | 50 | 74 | 98 | 68 | 35 |
| 60 | 128 | 202 | 241 | 262 | 168 | 76 |
| 102 | 190 | 263 | 260 | 245 | 142 | 58 |
| 78 | 129 | 172 | 173 | 182 | 109 | 48 |
| 78 | 142 | 205 | 230 | 263 | 170 | 82 |
| 36 | 68 | 102 | 121 | 142 | 88 | 40 |
| 15 | 28 | 38 | 44 | 50 | 26 | 9 |

*Σχήμα 3*

### 1.1.1.2. Τοπικοί Συντελεστές Συσχέτισης Πινάκων

Εάν τα παραπάνω είναι αναγκαία για το σωστό υπολογισμό της συσχέτισης των πινάκων, τα όσα ακολουθούν είναι απαραίτητα για την βελτιστοποίηση και την μεγαλύτερη αποδοτικότητα του αλγορίθμου.

Η δυσδιάστατη LCC (Local Correlation Coefficient) συνάρτηση μεταξύ των πινάκων F και T (όπου ο T είναι αρκετά μικρότερος σε σχέση με τον F) δίνεται από τον τύπο:

$$\frac{\mathrm{cov}(F,T)}{\sigma_F \sigma_T} = \frac{\sum_x \sum_y (T(x,y) - \mu_T)(\mathrm{F(x - u, y - u)} - \mu_F)}{\sqrt{\sum_x \sum_y (F(x-u,y-u) - \mu_F)^2 \sum_x \sum_y (T(x,y) - \mu_T)^2}} \qquad \textit{Σχέση 4}$$

όπου $\mu_T$ και $\sigma_T$ είναι ο μέσος όρος και η τυπική απόκλιση αντίστοιχα για τον Template ενώ $\mu_F$ και $\sigma_F$ είναι ο μέσος όρος και η τυπική απόκλιση αντίστοιχα για τον Frame.

Το στατιστικό αυτό μέτρο έχει την ιδιότητα να υπολογίζει τη συσχέτιση σε μία απόλυτη κλίμακα που κυμαίνεται στο [-1, 1]. Υπό συγκεκριμένες θεωρήσεις η τιμή που μετράται από το correlation coefficient δίνει μία γραμμική ένδειξη της ομοιότητας μεταξύ του Frame και του Template.

Στην συγκεκριμένη εργασία η πολυπλοκότητα στον υπολογισμό του LCC μειώνεται μέσω ισοδύναμων μετασχηματισμών που οδηγούν σε αποδοτικές πράξεις μεταξύ πινάκων ή επιτρέπουν την χρήση παράλληλου computing.

### 1.1.2. Συνέλιξη

Η μέθοδος της συνέλιξης χρησιμοποιείται σε πάρα πολλές εφαρμογές των μαθηματικών και της μηχανικής. Ειδικότερα πολύ συχνά χρησιμοποιείται για την επεξεργασία εικόνων, όπως για παράδειγμα διαδικασίες φιλτραρίσματος (sharpening, blurring, smoothing). Επί της ουσίας η μόνη διαφορά της με την διαδικασία της συσχέτισης που περιγράφηκε παραπάνω έγκειται στην αναστροφή του φίλτρου (T).

Εκθέτουμε εν συντομία τον μαθηματικό ορισμό της συνέλιξης δύο συναρτήσεων:

$$r(i) = (s * k)(i) = \int s(i-n)k(n)dn \qquad \text{Σχέση 5}$$

Με διακριτούς όρους η παραπάνω μπορεί να γραφτεί ως εξής:

$$r(i) = (s * k)(i) = \sum_n s(i-n)k(n) \qquad \text{Σχέση 6}$$

Παρατηρούμε ότι η συνέλιξη και η συσχέτιση ταυτίζονται όταν το φίλτρο είναι συμμετρικό. Μαθηματικά η συνέλιξη στη γενική περίπτωση δύο συναρτήσεων υπολογίζει το ποσό της επικάλυψης μεταξύ των δύο αυτών συναρτήσεων. Μπορεί να θεωρηθεί ως μία λειτουργία συγκερασμού η οποία ενοποιεί τον στοιχείο προς στοιχείο πολλαπλασιασμό ενός συνόλου δεδομένων με κάποιο άλλο.

#### 1.1.2.1. Δυσδιάστατη Συνέλιξη

Η δυσδιάστατη συνέλιξη είναι ουσιαστικά μία επέκταση της μονοδιάστατης που αναφέρθηκε παραπάνω. Η πράξη της συνέλιξης αυτή τη φορά εφαρμόζεται τόσο στην οριζόντια όσο και στην κατακόρυφη διεύθυνση του δυσδιάστατου χώρου

$$. \; r(i) = (s * k)(i,j) = \sum_n \sum_m s(i-n, j-m)k(n,m)$$

Σχέση 7

Μία ειδική περίπτωση 2D convolution είναι αυτή στην οποία είναι δυνατός ο διαχωρισμός του Template σε δύο μονοδιάστατους πίνακες. Η πρακτική αυτή συναντάται κατά την εφαρμογή κάποιου φίλτρου πχ Gaussian σε μία εικόνα. Τα διαχωριζόμενα φίλτρα είναι μία ειδική κατηγορία όπου το φίλτρο μπορεί να εκφραστεί ως σύνθεση δύο μονοδιάστατων φίλτρων. Το ένα εφαρμόζεται στις γραμμές

και το άλλο στις στήλες της εικόνας. Αν διασπάσουμε τον πίνακα Template όπως φαίνεται παρακάτω έχουμε μια διαχωρίσιμη συνέλιξη.

Η εφαρμογή του Template $\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$

στον Frame θα έχει το ίδιο αποτέλεσμα με την εφαρμογή του: $\begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}$

ακολουθούμενο από το: $\begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$

Ενώ δηλαδή μία δυσδιάστατη συνέλιξη απαιτεί n*m πολλαπλασιασμούς για κάθε στοιχείο εξόδου, όπου n και m είναι οι στήλες και οι γραμμές του φίλτρου, μία διαχωρίσιμη συνέλιξη απαιτεί μόλις n+m πολλαπλασιασμούς για κάθε στοιχείο εξόδου. Η τεχνική αυτή υλοποιείται από τον κώδικα convolution Separable στο sample της NVIDIA CUDA SDK. {6}

## 1.2. Τυπικά Μεγέθη στην Επεξεργασία Εικόνων

Για εφαρμογές πραγματικού χώρου και χρόνου η πολυπλοκότητα σε αριθμητικούς υπολογισμούς, στον προγραμματισμό και στο χρόνο πρόσβασης στη μνήμη είναι ένα καθοριστικό ζήτημα για την επιλογή του αλγορίθμου επίλυσης του προβλήματος. Ενδεικτικά για την επεξεργασία μίας ασπρόμαυρης εικόνας, προκειμένου αυτή να έχει μία σχετικά υψηλή ανάλυση, θα χρειαζόμασταν 512 X 512 = 262.144 Pixels. Δεδομένου μάλιστα ότι χρειάζονται 8 bits για την αναπαράσταση ενός ακεραίου εύρους [0 255] (grayscale), η εικόνα θα απαιτούσε 2.097.152 bits αποθηκευτικού χώρου. Την ίδια στιγμή μία ίσου μεγέθους έγχρωμη εικόνα απαιτεί 6.291.456 bits. Πολλές φορές μάλιστα το μέγεθος μεγάλων εικόνων ξεπερνάει το μέγεθος της L2 cache μνήμης της CPU με αποτέλεσμα να είναι αδύνατη η επεξεργασία τους σε αυτό το επίπεδο μνήμης το οποίο και θα μπορούσε να εξασφαλίζει υψηλές ταχύτητες επεξεργασίας.

Έτσι με βάση και τα όσα εκτέθηκαν στις παραπάνω ενότητες, η διαδικασία της επεξεργασίας εικόνας (2D LCC μεταξύ των Frame και Template) είναι μία διαδικασία επαναλαμβανόμενων ομοειδών ανεξάρτητων μεταξύ τους υπολογισμών και για το λόγο αυτό είναι δυνατή η παραλληλοποίηση τους και ενδείκνυται πλήρως η

αξιοποίηση παράλληλου υλικού και λογισμικού. Για το λόγο αυτό κάνουμε χρήση της CUDA για την επίλυση του προβλήματος σε GPUs.

## 1.3. Η σημασία της Παράλληλης Υπολογιστικής

Η παράλληλη υπολογιστική είναι μία μορφή υπολογισμού όπου πολλές πράξεις διεξάγονται ταυτόχρονα, στη βάση της αρχής που θέλει μεγάλα προβλήματα να διαιρούνται σε μικρότερα, τα οποία με τη σειρά τους επιλύονται ταυτόχρονα. Ο παραλληλισμός στους υπολογιστές χρησιμοποιούνταν εδώ και πολλά χρόνια, αλλά το ενδιαφέρον έχει αυξηθεί ιδιαίτερα τα τελευταία χρόνια, κυρίως στον τομέα των υπολογιστών υψηλών επιδόσεων, ακριβώς λόγω του φυσικού ορίου που αποτρέπει την αύξηση στη συχνότητα λειτουργίας των επεξεργαστών. Ακόμα, καθώς φαίνεται να δίνεται όλο και περισσότερη σημασία στην κατανάλωση ενέργειας των σύγχρονων υπολογιστών, τα συστήματα παράλληλου υπολογισμού έχουν γίνει κυρίαρχα παραδείγματα στην αρχιτεκτονική υπολογιστών, κυρίως υπό τη μορφή των πολύ-πύρινων επεξεργαστών.

Ο παραλληλισμός έχει αντιμετωπισθεί μερικές φορές ως σπάνια και εξωτική υπο-περιοχή της υπολογιστικής, που είναι μεν ενδιαφέρουσα αλλά μικρής πρόσβασης στο μέσο προγραμματιστή. Μια μελέτη των τάσεων στις εφαρμογές, την αρχιτεκτονική υπολογιστών, και τη δικτύωση δείχνει ότι αυτή η άποψη δεν είναι πλέον υποστηρίξιμη. Ο παραλληλισμός γίνεται πανταχού παρών, και ο παράλληλος προγραμματισμός γίνεται το κέντρο στην αυτοκρατορία του προγραμματισμού.

Τα προγράμματα – κώδικες παράλληλου υπολογισμού είναι δυσκολότερο να συνταχθούν σε σχέση με τα αντίστοιχα των σειριακών, καθώς ο συγχρονισμός και παραλληλισμός εισάγει αρκετούς νέους τύπους από software bugs, εκ των οποίων τα πιο συνηθισμένα είναι οι «συνθήκες ανταγωνισμού» (race conditions). Η επικοινωνία και ο συγχρονισμός μεταξύ διαφορετικών υπό-προβλημάτων είναι ένα από τα μεγαλύτερα εμπόδια προκειμένου να καταφέρει κανείς καλή απόδοση σε παράλληλο πρόγραμμα. Η επιτάχυνση ενός προγράμματος ως αποτέλεσμα παραλληλισμού δίνεται από το νόμο του Amdahl:

$$S = \frac{1}{(1 - P)} \qquad \textit{Σχέση 8}$$

όπου S είναι η επιτάχυνση του προγράμματος (ως συντελεστής του αρχικού σειριακού χρόνου εκτέλεσης) και P είναι το ποσοστό κατά το οποίο έχει παραλληλοποιηθεί.

Παραδοσιακά το λογισμικό υπολογιστών γραφόταν για σειριακή εκτέλεση. Για την επίλυση κάποιου προβλήματος κατασκευαζόταν και εκτελούνταν κάποιος αλγόριθμος ως μία σειριακή ακολουθία εντολών. Αυτές οι εντολές εκτελούνταν σε μία κεντρική μονάδα επεξεργασίας (CPU) σε έναν υπολογιστή.

Τα παράλληλα υπολογιστικά συστήματα από την άλλη, χρησιμοποιούν ταυτόχρονα πολλαπλά υπολογιστικά στοιχεία για την επίλυση ενός προβλήματος. Αυτό γίνεται κατορθωτό με τη διάσπαση του προβλήματος σε επιμέρους ανεξάρτητα τμήματα, έτσι ώστε κάθε στοιχείο επεξεργασίας να μπορεί να εκτελεί το δικό του κομμάτι του αλγορίθμου ταυτόχρονα με τα υπόλοιπα. Τα στοιχεία επεξεργασίας μπορεί να ποικίλουν και να χρησιμοποιούν πόρους όπως είναι ένας μοναδικός υπολογιστής με πολλαπλούς επεξεργαστές, πολλοί υπολογιστές σε δίκτυο, ειδικό υλικό, ή και οποιοσδήποτε συνδυασμός των παραπάνω.

Παραδοσιακά, οι εξελίξεις στην υψηλή υπολογιστική έχουν παρακινηθεί από τις αριθμητικές προσομοιώσεις πολυσύνθετων συστημάτων όπως το κλίμα, οι μηχανικές συσκευές, τα ηλεκτρονικά κυκλώματα, οι κατασκευαστικές διαδικασίες και οι χημικές αντιδράσεις. Εντούτοις, οι σημαντικότερες δυνάμεις που οδηγούν στην ανάπτυξη γρηγορότερων υπολογιστών είναι σήμερα οι αναδυόμενες εμπορικές εφαρμογές που απαιτούν από έναν υπολογιστή να είναι σε θέση να επεξεργαστεί μεγάλο πλήθος στοιχείων με περίπλοκους τρόπους. Αυτές οι εφαρμογές περιλαμβάνουν τα συνεργάσιμα περιβάλλοντα εργασίας, τη με τη βοήθεια υπολογιστή διάγνωση στην ιατρική, τις παράλληλες βάσεις δεδομένων οι οποίες χρησιμοποιούνται για υποστήριξη απόφασης, και τα προηγμένα γραφικά και την εικονική πραγματικότητα  ιδιαίτερα στη βιομηχανία διασκέδασης. Παραδείγματος χάριν, η ολοκλήρωση του παράλληλου υπολογισμού, της υψηλής απόδοσης δικτύωσης, και των τεχνολογιών πολυμέσων οδηγεί στην ανάπτυξη των video-servers, υπολογιστών οι οποίοι σχεδιάστηκαν με σκοπό να εξυπηρετήσουν τα εκατοντάδες ή χιλιάδες διαφορετικά αιτήματα για ένα βίντεο πραγματικού χρόνου.

# 2. Μονάδες Επεξεργασίας Γραφικών (GPUs)

## 2.1. Κατηγοριοποίηση Παράλληλων Υπολογιστικών Συστημάτων

Οι παράλληλοι υπολογιστές θα μπορούσαν να κατηγοριοποιηθούν ανάλογα με το επίπεδο στο οποίο το υλικό υποστηρίζει τον παραλληλισμό. Οι πολυ-πύρινοι καθώς και οι υπολογιστές πολλών επεξεργαστών έχουν πολλαπλά στοιχεία επεξεργασίας μέσα στο ίδιο μηχάνημα, ενώ τα clusters, τα MPP's (Massively Parallel Processors) και τα Grids χρησιμοποιούν πολλούς υπολογιστές για το ίδιο πρόβλημα. Ειδικές αρχιτεκτονικές παράλληλου υπολογισμού χρησιμοποιούνται καμιά φορά σε συνδυασμό με παραδοσιακές αρχιτεκτονικές για την επιτάχυνση συγκεκριμένων προβλημάτων.

### 2.1.1. Κατηγορίες Παραλληλισμού

#### 2.1.1.1. Επιπέδου Bit

Από την κατασκευή των ολοκληρωμένων πολύ υψηλής κλίμακας (very large scale integration –VLSI computer chip) στις αρχές της δεκαετίας του '70 μέχρι και περίπου το 1986, η επιτάχυνση στην αρχιτεκτονική υπολογιστών επιτυγχάνονταν διπλασιάζοντας το μέγεθος λέξης υπολογιστή, το ποσό δηλαδή πληροφορίας που μπορεί να εκτελέσει ένας επεξεργαστής σε κάθε κύκλο μηχανής. Αυξάνοντας το μέγεθος λέξης μειώνεται ο αριθμός εντολών που πρέπει να εκτελέσει ο επεξεργαστής για να κάνει πράξεις μεταξύ μεταβλητών το μέγεθος των οποίων είναι μεγαλύτερο του μήκους λέξης.

#### 2.1.1.2. Επιπέδου Εντολών

Ένα πρόγραμμα υπολογιστή είναι στην ουσία μία αλληλουχία εντολών που εκτελούνται από έναν επεξεργαστή. Αυτές οι εντολές μπορούν να αναδιατάσσονται και να συνδυάζονται σε ομάδες οι οποίες εν συνεχεία εκτελούνται παράλληλα χωρίς να αλλάζει το αποτέλεσμα του προγράμματος. Ο παραλληλισμός επιπέδου εντολών κυριάρχησε στην αρχιτεκτονική υπολογιστών από τα μέσα της δεκαετίας του '80 έως τα μέσα του '90.

#### 2.1.1.3. Παραλληλισμός Δεδομένων

Ο παραλληλισμός δεδομένων εφαρμόζεται στις επαναλήψεις του προγράμματος (program loops), και εστιάζει στο διαμερισμό των δεδομένων κατά μήκος διαφορετικών υπολογιστικών κόμβων για παράλληλη επεξεργασία. Ο παραλληλισμός των επαναλήψεων συχνά οδηγεί σε όμοιες (όχι απαραίτητα ταυτόσημες) ακολουθίες διαδικασιών και συναρτήσεων που εκτελούνται πάνω σε στοιχεία μεγάλων δομών δεδομένων. Πολλές επιστημονικές και μηχανικές εφαρμογές χρησιμοποιούν τον παραλληλισμό δεδομένων.

#### 2.1.1.4. Παραλληλισμός Διαδικασιών (Tasks)

Ο παραλληλισμός διαδικασιών είναι το χαρακτηριστικό ενός παράλληλου προγράμματος όπου εντελώς διαφορετικοί υπολογισμοί μπορούν να εκτελεστούν είτε σε ίδια είτε σε διαφορετικά σύνολα δεδομένων. Αυτό διαφέρει από τον παραλληλισμό δεδομένων, όπου ο ίδιος υπολογισμός εκτελείται στο ίδιο ή σε διαφορετικό σύνολο δεδομένων. Ο παραλληλισμός διαδικασιών πάντως, συνήθως δεν αυξάνεται ανάλογα με το μέγεθος του προβλήματος.

### 2.1.2. Κλάσεις Παράλληλων Υπολογιστών

Οι παράλληλοι υπολογιστές μπορούν να ταξινομηθούν κατά προσέγγιση σύμφωνα με το επίπεδο στο οποίο το υλικό υποστηρίζει τον παραλληλισμό. Αυτή η ταξινόμηση είναι σε γενικές γραμμές ανάλογη της απόστασης μεταξύ των βασικών υπολογιστικών κόμβων.

#### 2.1.2.1. Πολυπύρηνοι επεξεργαστές

Ένας πολύ-πύρηνος επεξεργαστής, είναι ένας επεξεργαστής που περιλαμβάνει πολλές μονάδες εκτέλεσης και επεξεργασίας. Αυτοί οι επεξεργαστές διαφέρουν από τους υπερβαθμωτούς (superscalar) επεξεργαστές, οι οποίοι μπορούν να εκκινήσουν πολλαπλές εντολές ανά κύκλο μηχανής, από μία μόνο ακολουθία εντολών. Αντίθετα ένας πολυπύρηνος επεξεργαστής μπορεί να εκκινεί πολλαπλές εντολές ανά κύκλο μηχανής από πολλές ακολουθίες πολλαπλών εντολών. Κάθε πυρήνας σε έναν πολυπύρηνο επεξεργαστή έχει προοπτική να γίνει υπερβαθμωτός. Δηλαδή, σε κάθε κύκλο, κάθε πυρήνας εκκινεί πολλαπλές εντολές από μία ακολουθία εντολών.

#### 2.1.2.2. Συμμετρικοί Επεξεργαστές

Ένας συμμετρικός πολυεπεξεργαστής (symmetric multiprocessor - SMP) είναι ένα υπολογιστικό σύστημα με πολλαπλούς πανομοιότυπους επεξεργαστές οι οποίοι μοιράζονται την μνήμη και συνδέονται με δίαυλο επικοινωνίας. Οι συγκρούσεις όμως διαύλου αποτρέπουν τους σχεδιαστές από το να τους επιταχύνουνε. Σαν αποτέλεσμα, οι συμμετρικοί πολυεπεξεργαστές συνήθως δεν αποτελούνται από περισσότερους από 32 επεξεργαστές.

#### 2.1.2.3. Διανεμημένοι Υπολογιστές

Ένας διανεμημένος υπολογιστής ( γνωστός και ως πολυεπεξεργαστής διανεμημένης μνήμης) είναι ένα σύστημα υπολογιστή διανεμημένης μνήμης στο οποίο τα στοιχεία επεξεργασίας είναι συνδεδεμένα σε δίκτυο. Οι διανεμημένοι υπολογιστές έχουν υψηλό βαθμό κλιμακοποίησης.

##### 2.1.2.3.1. Clusters

Το cluster είναι ένα σύνολο από χαλαρά συνδεδεμένους υπολογιστές που λειτουργούν τόσο στενά μεταξύ τους που σε κάποιες περιπτώσεις λογίζονται ως ένας. Τα clusters αποτελούνται από πολλές μεμονωμένες μηχανές συνδεδεμένες σε δίκτυο.

##### 2.1.2.3.2. Μαζικά Παράλληλοι Επεξεργαστές

Ένας μαζικά παράλληλος επεξεργαστής (massively parallel processor - MPP) είναι ένας μεμονωμένος υπολογιστής με πολλούς δικτυωμένους επεξεργαστές. Οι MPPs έχουν πολλά παρόμοια χαρακτηριστικά με τα clusters, ωστόσο οι MPPs έχουν εξειδικευμένα εσωτερικά διασυνδεδεμένα δίκτυα ενώ τα clusters χρησιμοποιούν εμπορικό υλικό για δικτύωση. Επίσης οι MPPs τείνουν να είναι μεγαλύτεροι από τα clusters, αφού έχουν συνήθως πάνω από εκατό επεξεργαστές. Σε έναν MPP κάθε CPU περιέχει την δική του μνήμη και αντίγραφο του λειτουργικού συστήματος και του λογισμικού εφαρμογής. Κάθε υποσύστημα επικοινωνεί με όλα τα άλλα μέσω διασύνδεσης υψηλής ταχύτητας.

##### 2.1.2.3.3. Grid

Η Grid υπολογιστική είναι η πιο διαδεδομένη μορφή παράλληλων υπολογισμών. Κάνει χρήση υπολογιστών που επικοινωνούν μέσω internet ώστε να δουλέψουν όλοι μαζί για ένα συγκεκριμένο πρόβλημα. Λόγω του χαμηλού εύρους ζώνης και της εξαιρετικά υψηλής καθυστέρησης που υπάρχει στο internet, η grid υπολογιστική αντιμετωπίζει μόνο στενά σχετιζόμενα παράλληλα προβλήματα.

#### 2.1.2.4. Εξειδικευμένοι Παράλληλοι Υπολογιστές

Μεταξύ των παράλληλων υπολογιστών υπάρχουν ορισμένες εξειδικευμένες συσκευές που αποτελούν νέα πεδία ενδιαφέροντος. Ενώ ακόμα δεν κυριαρχούν σε συγκεκριμένους τομείς εμφανίζουν την τάση να χρησιμοποιούνται σε ορισμένες εφαρμογές παράλληλων προβλημάτων.

##### 2.1.2.4.1. FPGAs

Τα FPGAs είναι συνεπεξεργαστές σε κάποιον υπολογιστή γενικής χρήσης.

##### 2.1.2.4.2. GPGPUs

Οι GPGPUs είναι ένα σχετικά πρόσφατο πεδίο έρευνας για τους μηχανικούς υπολογιστών. Οι GPUs (**Graphics Processing Units**) είναι συν-επεξεργαστές εξαιρετικά βελτιστοποιημένοι κυρίαρχα για επεξεργασία γραφικών. Η επεξεργασία γραφικών είναι ένα πεδίο που κυριαρχείται από λειτουργίες παραλληλισμού δεδομένων και ειδικότερα λειτουργίες γραμμικής άλγεβρας πινάκων.

Μέχρι πρόσφατα οι GPGPUs χρησιμοποιούσαν τις συνηθισμένες εφαρμογές γραφικών (APIs) για την εκτέλεση ορισμένων προγραμμάτων. Τελευταία όμως αναπτύσσονται νέες γλώσσες προγραμματισμού και πλατφόρμες για υπολογισμούς και επίλυση προβλημάτων σε GPGPUs. Τόσο η NVIDIA όσο και η AMD έχουν εκδώσει περιβάλλοντα προγραμματισμού όπως η CUDA και η CMT αντίστοιχα. Άλλες γλώσσες προγραμματισμού των GPGPUs είναι οι BrookGPU, PeakStream και RapidMind. Η NVIDIA έχει μάλιστα εκδώσει ειδικά πακέτα λογισμικού για εφαρμογή στις κάρτες Tesla.

##### 2.1.2.4.3. ASICs

Ακριβώς επειδή τα ASIC εξ' ορισμού ειδικεύονται σε κάποια δεδομένη εφαρμογή, μπορούν να είναι πλήρως βελτιστοποιημένα για την εφαρμογή αυτή. Ως αποτέλεσμα, δεδομένης της εφαρμογής, μπορεί τα ASIC να ξεπερνούν κατά πολύ σε απόδοση τις GPGPUs.

## 2.2. Αρχιτεκτονικές Μνήμης Παράλληλων Υπολογιστικών Συστημάτων

Η κύρια μνήμη σε έναν παράλληλο υπολογιστή είναι είτε διαμοιραζόμενη μνήμη (shared memory) (μοιρασμένη ανάμεσα σε όλα τα επεξεργαζόμενα στοιχεία σε ένα μοναδικό χώρο διευθύνσεων) είτε διανεμημένη μνήμη (distributed memory) (στην οποία κάθε επεξεργαζόμενο στοιχείο έχει τη δικό του τοπικό χώρο διευθύνσεων). Η διανεμημένη μνήμη αναφέρεται στο γεγονός ότι η μνήμη είναι λογικώς διανεμημένη, αλλά συχνά υπονοείται ότι είναι επίσης και φυσικά διανεμημένη. Η διανεμημένη διαμοιραζόμενη μνήμη (distributed shared memory) είναι ένας συνδυασμός των δύο προσεγγίσεων, όπου το επεξεργαζόμενο στοιχείο έχει τη δική του τοπική μνήμη και ταυτόχρονα πρόσβαση στη μνήμη των μη-τοπικών επεξεργαστών. Οι προσβάσεις στην τοπική μνήμη είναι συνήθως ταχύτερες από τις προσβάσεις στη μη-τοπική μνήμη.

## 2.3. Αρχιτεκτονική Μονάδας Επεξεργασίας Γραφικών

Η μονάδα επεξεργασίας γραφικών (GPU) έχει γίνει βασικό μέρος των σημερινών κοινώς διαδεδομένων μονάδων επεξεργασίας . Τα τελευταία 7 χρόνια, έχει σημειωθεί μια αξιόλογη άνοδος στην απόδοση και τις ικανότητες των GPUs. Η σύγχρονη GPU δεν είναι μόνο μια ισχυρή μηχανή γραφικών αλλά και ένας επεξεργαστής που μπορεί να προγραμματιστεί σε υψηλό βαθμό παραλληλίας και υποστηρίζει μεγαλύτερο αριθμητικό εύρος και εύρος μνήμης και ουσιαστικά εκτοπίζει την CPU. Η ραγδαία ανάπτυξη της GPU σε αμφότερες την ικανότητα προγραμματισμού και την επιδεκτικότητα, έχει δημιουργήσει μια ερευνητική κοινότητα που έχει επιτυχώς χαρτογραφήσει μια ευρεία κλίμακα απαιτητικών υπολογιστικά και πολύπλοκων προβλημάτων σχετικά με τη GPU. Αυτή η προσπάθεια στους υπολογισμούς γενικής χρήσης πάνω στη GPU, που είναι γνωστή ως GPU computing, έχει τοποθετήσει τη GPU σε θέση αναγκαστικής εναλλακτικής λύσης σε σχέση με τους παραδοσιακούς μικροεπεξεργαστές όσον αφορά στα υψηλής απόδοσης υπολογιστικά συστήματα του μέλλοντος.

Η GPU είναι σχεδιασμένη για μια ειδική κατηγορία εφαρμογών με τα ακόλουθα χαρακτηριστικά.

• Οι υπολογιστικές απαιτήσεις είναι μεγάλες. Η επεξεργασία σε πραγματικό χρόνο απαιτεί δισεκατομμύρια pixels το δευτερόλεπτο και κάθε pixel απαιτεί εκατοντάδες ή και περισσότερες πράξεις. Οι GPUs πρέπει να έχουν μια τεράστια υπολογιστική απόδοση για να ικανοποιούν τις απαιτήσεις περίπλοκων εφαρμογών πραγματικού χρόνου.

•Η παραλληλότητα είναι βασική. Ο αγωγός διοχέτευσης γραφικών (graphics pipeline) είναι κατάλληλος για παραλληλισμό. Οι εφαρμογές σε vertices και fragments ταιριάζουν καλά σε καλά ορισμένες, στενά συνδεδεμένες προγραμματιστικές παράλληλες υπολογιστικές μονάδες, οι οποίες εν συνεχεία είναι προσαρμόσιμες σε πολλά ακόμα υπολογιστικά πεδία.

•Ωστόσο η ρυθμαπόδοση (throughput) είναι πιο σημαντική από τη λανθάνουσα περιόδο (latency). Οι εφαρμογές της GPU στον αγωγό διοχέτευσης γραφικών παίρνουν σαν προτεραιότητα την ρυθμαπόδοση και όχι τη λανθάνουσα περίοδο. Το ανθρώπινο οπτικό σύστημα λειτουργεί σε κλίμακα χρόνου τάξης msec, την ώρα που οι λειτουργίες σε σύγχρονες εφαρμογές παίρνουν nsec. Αυτό το κενό της τάξης του $10^6$ σημαίνει ότι η λανθάνουσα περίοδος οποιασδήποτε μεμονωμένης εφαρμογής είναι άνευ σημασίας. Σαν αποτέλεσμα αυτού, ο αγωγός διοχέτευσης γραφικών έχει τη δυνατότητα εκατοντάδων ή χιλιάδων κύκλων μηχανής, με χιλιάδες πρωταρχικές λειτουργίες σε εκτέλεση κάθε δεδομένη στιγμή. Ο αγωγός είναι επίσης προς το εμπρός τροφοδοτούμενος (feed forward), αφαιρώντας το μειονέκτημα των αντικρούσεων (conflicts), επιτρέποντας περαιτέρω ευνοϊκότερη ρυθμαπόδοση των πρωταρχικών λειτουργιών μέσω της γραμμής διοχέτευσης. Αυτή η έμφαση στη ρυθμαπόδοση είναι χαρακτηριστικό εφαρμογών και σε άλλα πεδία.

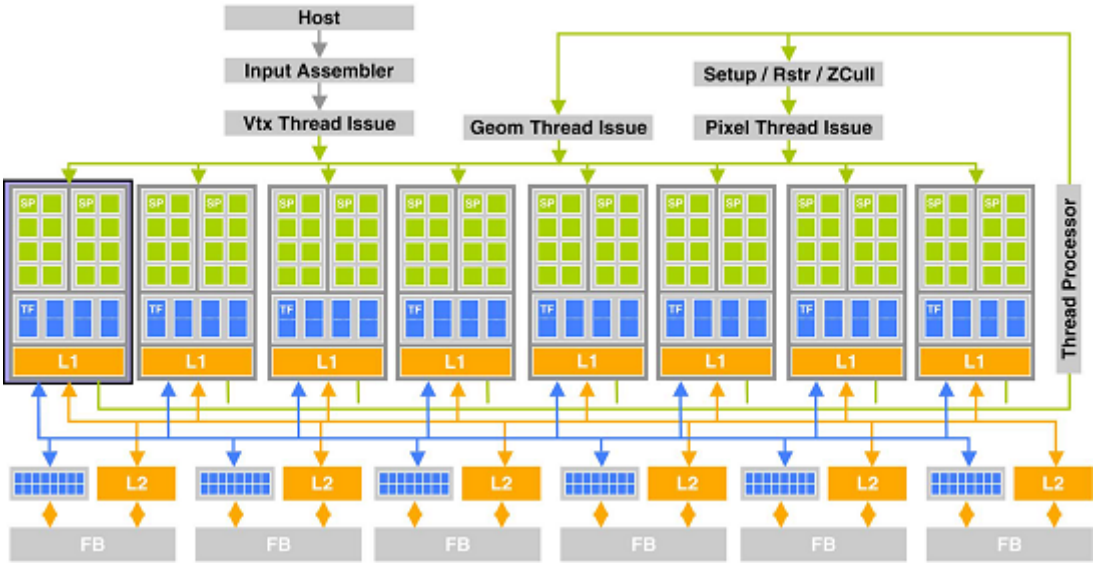
### 2.3.1. Γενικά Χαρακτηριστικά

Πιο πάνω, σημειώσαμε πως η GPU σχεδιάστηκε για απαιτήσεις εφαρμογών άλλου είδους από τη CPU: υψηλές, παράλληλες υπολογιστικές απαιτήσεις με μια έμφαση στην ρυθμαπόδοση αντί για τη λανθάνουσα περίοδο. Συνεπώς η αρχιτεκτονική της έχει εξελιχθεί προς διαφορετική κατεύθυνση. Σκεφτείτε έναν αγωγό διοχέτευσης εργασιών, όπως αυτόν που βλέπουμε στις περισσότερες εφαρμογές (APIs) γραφικών (και σε άλλες εφαρμογές επίσης), που πρέπει να επεξεργαστεί ένα

μεγάλο όγκο εισερχόμενων στοιχείων. Σε έναν τέτοιο αγωγό, το αποτέλεσμα κάθε επιτυχούς εργασίας τροφοδοτεί την είσοδο της επόμενης εργασίας. Ο αγωγός εξασφαλίζει την παραλληλοποίηση των εργασιών, αφού τα δεδομένα σε πολλαπλά στάδια του αγωγού μπορούν να υπολογιστούν την ίδια στιγμή, μέσα σε κάθε στάδιο. Ο υπολογισμός περισσότερων από ένα στοιχείων την ίδια στιγμή είναι η παραλληλότητα των δεδομένων. Για να εκτελέσει έναν τέτοιο αγωγό, μια CPU θα έπαιρνε κάθε απλό στοιχείο (ή ομάδα στοιχείων) και θα επεξεργαζόταν το πρώτο στάδιο στον αγωγό, μετά το επόμενο και ούτω καθεξής. Η CPU διαιρεί τον αγωγό εγκαίρως, εφαρμόζοντας όλους τους πόρους σε κάθε στάδιο με τη σειρά.

Οι GPUs ιστορικά πήραν μια διαφορετική προσέγγιση. Η GPU διανέμει τους πόρους του επεξεργαστή ανάμεσα στα διάφορα στάδια, έτσι ώστε ο αγωγός να διαιρείται στο χώρο και όχι στο χρόνο. Το τμήμα του επεξεργαστή που λειτουργεί σε κάθε στάδιο τροφοδοτεί με την έξοδό του κατευθείαν ένα διαφορετικό τμήμα που λειτουργεί στο επόμενο στάδιο. Αυτή η οργάνωση είναι πολύ επιτυχημένη σε GPUs καθορισμένης λειτουργίας (fixed-function) για δυο λόγους. Πρώτον, το υλικό σε οποιοδήποτε στάδιο μπορεί να εκμεταλλευτεί την παραλληλότητα των δεδομένων μέσα στο στάδιο, επεξεργαζόμενο πολλά στοιχεία την ίδια στιγμή. Επειδή πολλά στάδια παράλληλων εργασιών «τρέχουν» κάθε στιγμή, η GPU μπορεί να έχει υψηλή υπολογιστική απόδοση στον αγωγό των γραφικών. Δεύτερον, το υλικό κάθε σταδίου μπορεί να είναι προσαρμοζόμενο με υλικό ειδικού σκοπού για κάθε εργασία, επιτρέποντας ουσιαστικά μεγαλύτερους υπολογισμούς και απόδοση, εν συγκρίσει με μια λύση γενικής χρήσης. Για παράδειγμα το στάδιο της rasterization , που υπολογίζει πληροφορίες κάλυψης pixel για κάθε τρίγωνο εισόδου, είναι πιο αποτελεσματικό όταν υλοποιείται σε υλικό ειδικού σκοπού. Καθώς τα προγραμματίσιμα στάδια (όπως τα προγράμματα vertex και fragment) αντικατέστησαν τα καθορισμένης λειτουργίας στάδια, τα καθορισμένης λειτουργίας - ειδικού σκοπού στοιχεία απλά αντικαταστάθηκαν από προγραμματίσιμα στοιχεία, αλλά η οργάνωση παράλληλων εργασιών δεν άλλαξε. Το αποτέλεσμα ήταν ένας feed forward αγωγός GPU με πολλά στάδια, το καθένα επιταχυνόμενο από ειδικού-σκοπού παράλληλο υλικό. Σε μια CPU, κάθε δεδομένη λειτουργία, μπορεί να πάρει ως και 20 κύκλους από την είσοδο ως την έξοδο από τον αγωγό CPU. Σε μια GPU, μια λειτουργία γραφικών μπορεί να πάρει χιλιάδες κύκλους από την αρχή ως το τέλος. Η λανθάνουσα περίοδος κάθε δεδομένης λειτουργίας είναι μεγάλος. Ωστόσο η παραλληλότητα των δεδομένων και των εργασιών κατά μήκος και ανάμεσα στα στάδια επιφέρει υψηλή ρυθμαπόδοση.

Το βασικότερο μειονέκτημα του αγωγού παράλληλων-εργασιών της GPU είναι η εξισορρόπηση της φόρτωσης δεδομένων (load balancing). Όπως κάθε αγωγός, η απόδοση του αγωγού της GPU είναι εξαρτημένη από το αργότερο του στάδιο. Αν το πρόγραμμα vertex είναι πολύπλοκο και το πρόγραμμα fragment είναι απλό, η συνολική ρυθμαπόδοση είναι εξαρτημένη από την απόδοση του vertex. Στα πρώτα χρόνια των προγραμματίσιμων σταδίων, η δομή εντολών των vertex και fragment προγραμμάτων ήταν αρκετά διαφορετική, οπότε τα στάδια ήταν ξεχωριστά. Ωστόσο όσο τα προγράμματα (τόσο το vertex όσο και το fragment) αποκτούσαν ολοκληρωμένα χαρακτηριστικά, και όσο οι δομές εντολών συνέκλιναν, οι αρχιτέκτονες των GPUs άρχισαν να αναθεωρούν υπέρ ενός αυστηρά παράλληλων-εργασιών αγωγού, προς όφελος μιας ενοποιημένης αρχιτεκτονικής, στην οποία όλες οι προγραμματίσιμες μονάδες στον αγωγό μοιράζονται μια και μόνο προγραμματίσιμη μονάδα υλικού. Ενώ πολλοί από τους αγωγούς είναι ακόμα παράλληλων-εργασιών, οι προγραμματίσιμες μονάδες τώρα μοιράζουν το χρόνο τους μεταξύ της δουλειάς vertex, της δουλειάς fragment, και της δουλειάς geometry ( με τους νέους DirectX 10 geometry shaders). Αυτές οι μονάδες μπορούν να εκμεταλλευτούν τόσο τον παραλληλισμό δεδομένων όσο και τον παραλληλισμό εργασιών. Όσο τα προγραμματίσιμα μέρη του αγωγού είναι υπεύθυνα για όλο και περισσότερους υπολογισμούς εντός του αγωγού γραφικών, η αρχιτεκτονική των GPUs μεταναστεύει από μια αυστηρά παράλληλων-εργασιών αρχιτεκτονική σε μια που όλο και περισσότερο χτίζεται γύρω από μια μοναδική ενοποιημένη παράλληλων-δεδομένων προγραμματίσιμη μονάδα.{10}



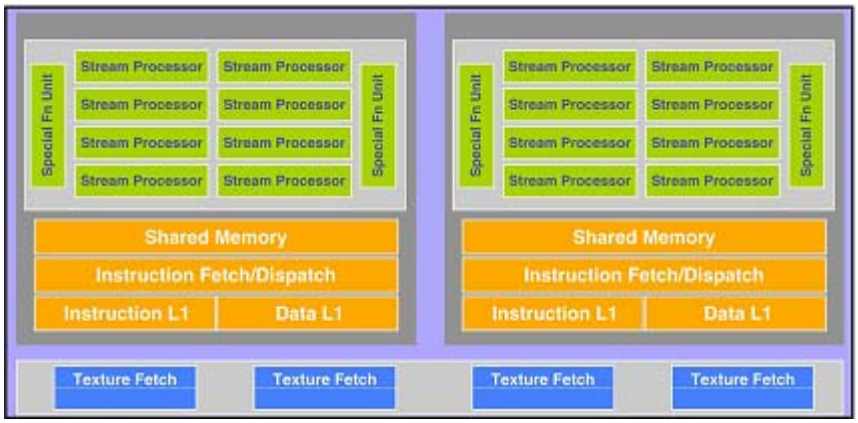

*Σχήμα 4* {10}

Στο σχήμα 4 φαίνονται 16 streaming πολυεπεξεργαστές με 8 επεξεργαστές νημάτων ο καθένας. Ένα ζευγάρι streaming πολυεπεξεργαστών φαίνεται στο σχήμα 5, ο καθένας περιέχει shared instruction και cache δεδομένα, control logic, μια shared μνήμη 16 kB, 8 επεξεργαστές νημάτων, και 2 μονάδες ειδικής λειτουργίας.

### 2.3.2. Αρχιτεκτονική Tesla

Η εισηγμένη από την NVIDIA το Νοέμβριο του 2006 ενοποιημένη - όσον αφορά γραφικές εφαρμογές (graphics) και υπολογιστική (computing) - αρχιτεκτονική Tesla επέκτεινε σημαντικά τις GPU πέρα από τις γραφικές εφαρμογές. Η μαζικά πολυνηματική σειρά επεξεργαστών της γίνεται μία ιδιαίτερα αποδοτική ενοποιημένη πλατφόρμα τόσο για γραφικές όσο και για γενικής χρήσης παράλληλες εφαρμογές υπολογισμού. Με τον κλιμακωτό αριθμό επεξεργαστών και κατατμήσεων της μνήμης, η αρχιτεκτονική Tesla εκτείνεται σε μία ευρεία αγορά. Τα υπολογιστικά χαρακτηριστικά της αρχιτεκτονικής αυτής επιτρέπουν τον απευθείας προγραμματισμό των πυρήνων της GPU σε γλώσσα C με την CUDA.

Η αρχιτεκτονική Tesla είναι χτισμένη γύρω από μία κλιμακωτή σειρά από πολυνηματικούς streaming πολυεπεξεργαστές (Streaming Multiprocessors - SMs). Οι τρέχουσες εφαρμογές GPU κυμαίνονται από 768 έως 12.288 ταυτόχρονα εκτελούμενα νήματα. Η ξεκάθαρη διαβάθμιση κατά μήκος αυτού του πλατύ φάσματος του διαθέσιμου παραλληλισμού είναι ένας βασικός στόχος της σχεδίασης τόσο της αρχιτεκτονικής GPU όσο και του μοντέλου προγραμματισμού της CUDA. Στο σχήμα 4 φαίνεται μια GPU με 16 SMs – συνολικά 128 πυρήνες επεξεργαστών streaming (SPs) – που συνδέεται με 6 εξωτερικά DRAM διαμερίσματα. Όταν ένα πρόγραμμα CUDA στον οικοδεσπότη (host) επικαλείται ένα πλέγμα (grid) πυρήνων (kernel), τότε το Κέντρο Διανομής Υπολογισμού (Compute Work Distribution –CWD) απαριθμεί τα block (blocks) του πλέγματος (grid) και αρχίζει να τα διανέμει στους SMs με τη διαθέσιμη ικανότητα εκτέλεσης. Τα νήματα ενός block εκτελούνται ταυτόχρονα σε έναν SM. Όταν τα block των νημάτων ολοκληρώνονται, το CWD ξεκινά νέα block στους ελεύθερους πολυεπεξεργαστές.




*Σχήμα 5* {10}

Ένας SM αποτελείται από οκτώ κλιμακωτούς πυρήνες SP, δύο ειδικές μονάδες λειτουργίας (Special Function Units – SFUs), μία πολυνηματική μονάδα εντολών (Multithreaded Instruction Unit – MT IU), και ένα chip κοινής μνήμης. Ο SM δημιουργεί, διαχειρίζεται και εκτελεί μέχρι 768 ταυτόχρονα νήματα στο υλικό με μηδενική δαπάνη χρονοπρογραμματισμού. Μπορεί να εκτελέσει το πολύ 8 block από νήματα ταυτόχρονα, που περιορίζονται από του πόρους νημάτων και μνήμης. Ο SM εκτελεί τον φυσικό συγχρονισμό των block με μία μόνο εντολή, την __syncthreads(). Ο γρήγορος συγχρονισμός των block μαζί με την δημιουργία lightweight διαδικασιών, των νημάτων, και τη μηδενική δαπάνη αποτελεσματικού χρονοπρογραμματισμού τους, υποστηρίζει πολύ λεπτομερή παραλληλισμό, επιτρέποντας σε ένα νέο νήμα να δημιουργηθεί για να υπολογίσει κάθε vertex, pixel και στοιχείο των δεδομένων.

Για να διαχειριστεί τα εκατοντάδες νήματα που τρέχουν αρκετά διαφορετικά προγράμματα, ο Tesla SM υιοθετεί μια νέα αρχιτεκτονική που καλείται αρχιτεκτονική Μίας Εντολής, Πολλαπλών Νημάτων (Single-Instruction, Multiple-Thread , SIMT). Ο SM αντιστοιχεί κάθε νήμα σε έναν SP βαθμωτό πυρήνα, και κάθε βαθμωτό νήμα εκτελείται ανεξάρτητα με τη δική του διεύθυνση εντολής και καταχωρητή κατάστασης. Η μονάδα SIMT του SM δημιουργεί, διαχειρίζεται, χρονοπρογραμματίζει και εκτελεί νήματα σε ομάδες των 32 παράλληλων νημάτων που καλούνται warps ( ο όρος αυτός προέρχεται από το weaving (ύφανση), η πρώτη τεχνολογία παράλληλων νημάτων). Τα μεμονωμένα νήματα που συνθέτουν ένα SIMT warp ξεκινούν μαζί στην ίδια διεύθυνση προγράμματος αλλά είναι παρόλα αυτά ελεύθερα να διακλαδιστούν και να εκτελεστούν ανεξάρτητα. Κάθε SM διαχειρίζεται μία δεξαμενή από 24 warps των 32 νημάτων σε κάθε warp, δηλαδή συνολικά 768 νήματα.

Κάθε φορά που είναι να διανεμηθεί μία εντολή, η μονάδα SIMT διαλέγει ένα warp που πρόκειται να εκτελεσθεί και διανέμει την επόμενη εντολή στα ενεργά νήματα του warp. Ένα warp εκτελεί μία κοινή εντολή την φορά, κι έτσι πλήρης αποδοτικότητα επιτυγχάνεται όταν και τα 32 νήματα ενός warp συμφωνούν στην πορεία εκτέλεσής τους. Αν τα νήματα ενός warp αποκλίνουν μέσω μίας συνθήκης διακλάδωσης που εξαρτάται από τα δεδομένα, τότε το warp εκτελεί σειριακά κάθε μονοπάτι διακλάδωσης που έχει ακολουθηθεί, απενεργοποιώντας τα νήματα εκείνα που δεν είναι σε εκείνο το μονοπάτι, και όταν όλα τα μονοπάτια ολοκληρωθούν, τα νήματα συγκλίνουν πίσω στο ίδιο μονοπάτι εκτέλεσης. Η απόκλιση μέσω διακλάδωσης συμβαίνει μόνο μέσα σε ένα warp. Διαφορετικά warps εκτελούνται ανεξάρτητα, χωρίς να έχει σημασία το αν εκτελούν κοινά ή διαχωρισμένα μονοπάτια κώδικα. Ως αποτέλεσμα, η αρχιτεκτονική Tesla των GPU είναι εντυπωσιακά αποδοτικότερη και ευέλικτη σε κώδικες με διακλαδώσεις, απ' ότι προηγούμενης γενιάς GPUs, δεδομένου ότι η τεχνολογία των warps των 32 νημάτων είναι πιο στενή και λεπτομερής από το εύρος της τεχνολογίας Μίας Εντολής – Πολλαπλών Δεδομένων, SIMD (single-instruction multiple-data), των προγενέστερων GPUs.

Η αρχιτεκτονική SIMT είναι συγγενής με SIMD τύπου αρχιτεκτονικές στο ότι μία εντολή ελέγχει πολλαπλά στοιχεία υπό επεξεργασία. Μία διαφορά κλειδί είναι ότι η δεύτερη αρχιτεκτονική εκθέτει το εύρος SIMD στο λογισμικό, ενώ οι εντολές SIMT καθορίζουν την συμπεριφορά εκτέλεσης και διακλάδωσης ενός απλού νήματος. Σε αντίθεση με τις SIMD μηχανές, η αρχιτεκτονική SIMT επιτρέπει στους προγραμματιστές να γράφουν παράλληλο κώδικα στο επίπεδο του νήματος (thread-level), για ανεξάρτητα, βαθμωτά νήματα, όπως επίσης και παράλληλο κώδικα στο επίπεδο δεδομένων (data-parallel) για νήματα με συγκεκριμένες συντεταγμένες. Για σκοπούς ακρίβειας, ο προγραμματιστής μπορεί ουσιαστικά να αγνοήσει τη συμπεριφορά της SIMT. Παρόλα αυτά ουσιαστικές βελτιώσεις απόδοσης μπορούν να πραγματοποιηθούν λαμβάνοντας υπόψη ότι ο κώδικας σπάνια απαιτεί από τα νήματα ενός warp να αποκλίνουν. Στη πράξη, αυτό είναι ανάλογο με το ρόλο των γραμμών cache σε παραδοσιακό κώδικα. Το μέγεθος της γραμμής cache μπορεί να αγνοηθεί άφοβα για λόγους ορθότητας, αλλά πρέπει να ληφθεί υπόψη για μέγιστη απόδοση. Η αρχιτεκτονικές ανυσμάτων, από την άλλη πλευρά, απαιτούν από το λογισμικό να ενώσει κομμάτια σε ανύσματα, και να διαχειριστεί τις αποκλίσεις χειροκίνητα.

Οι μεταβλητές ενός νήματος τυπικά εδρεύουν σε εν ενεργεία καταχωρητές. Η κοινή SM μνήμη των 16KB έχει πολύ μικρό χρόνο προσπέλασης και υψηλό εύρος

ζώνης, παρόμοιο με αυτό μιας L1 cache μνήμης. Διατηρεί τις κοινές μεταβλητές κάθε block της CUDA (per-block __shared__ variables) για τα ενεργά block νημάτων. Ο SM παρέχει εντολές load/store για να προσπελαύνει τις μεταβλητές συσκευής της CUDA (__device__ variables) στην εξωτερική DRAM της GPU. Συγχωνεύει ξεχωριστές προσπελάσεις παράλληλων νημάτων του ίδιου warp σε λιγότερες προσπελάσεις block μνήμης (memory-block) όταν οι διευθύνεις «πέφτουν» στο ίδιο block και πληρούν κριτήρια ευθυγράμμισης. Επειδή ο χρόνος προσπέλασης της γενικής μνήμης (global memory) μπορεί να είναι εκατοντάδες κύκλοι μηχανής του επεξεργαστή, τα προγράμματα σε CUDA αντιγράφουν δεδομένα στη κοινή μνήμη (shared) όταν πρέπει να προσπελαστούν αρκετές φορές από ένα block νημάτων. Οι Tesla load/store εντολές μνήμης χρησιμοποιούν διευθυνσιοδότηση ακέραιου byte για να διευκολύνει τις βελτιστοποιήσεις κώδικα από συμβατικούς μεταγλωτιστές. Ο μεγάλος αριθμός των νημάτων σε κάθε SM, μαζί με την υποστήριξη για πολλές σημαντικές αιτήσεις φόρτωσης (load) από την εξωτερική DRAM, βοηθά να καλυφθεί ο χρόνος μεταξύ φόρτωσης και χρήσης. Η πιο πρόσφατη αρχιτεκτονική Tesla των GPU παρέχει επίσης ατομικές εντολές για ανάγνωση-τροποποίηση-εγγραφή μνήμης, διευκολύνοντας παράλληλες απλοποιήσεις και διαχείριση δομών παράλληλων δεδομένων.

Οι εφαρμογές της CUDA αποδίδουν καλά σε GPUs με αρχιτεκτονική Tesla επειδή ο παραλληλισμός της CUDA, ο συγχρονισμός, οι κοινές μνήμες, και η ιεραρχία των ομάδων νημάτων ταιριάζουν αποτελεσματικά σε χαρακτηριστικά της αρχιτεκτονικής των GPU, και επειδή η CUDA εκφράζει καλά των παραλληλισμό εφαρμογών.[9]

## 2.4. Προγραμματιστικό Μοντέλο μίας GPU

Τώρα που είδαμε την αρχιτεκτονική μιας GPU, πάμε στο προγραμματιστικό μοντέλο της.

Οι προγραμματίσιμες μονάδες της GPU ακολουθούν ένα προγραμματιστικό μοντέλο μοναδικού προγράμματος-πολλαπλών δεδομένων (single program multiple data - SPMD ). Για αποτελεσματικότητα, η GPU επεξεργάζεται πολλά στοιχεία ( vertices ή fragments ) παράλληλα χρησιμοποιώντας το ίδιο πρόγραμμα. Κάθε στοιχείο είναι ανεξάρτητο από το άλλο , και στο βασικό μοντέλο προγραμματισμού τα

στοιχεία δεν επικοινωνούν μεταξύ τους. Όλα τα προγράμματα GPU πρέπει να είναι δομημένα τοιουτοτρόπως: πολλά παράλληλα στοιχεία, το καθένα επεξεργαζόμενο παράλληλα από ένα μοναδικό πρόγραμμα. Κάθε στοιχείο μπορεί να λειτουργεί σε ένα 32-bit ακέραιο ή κινητής υποδιαστολής ( integer or floating point) δεδομένο με ένα ολοκληρωμένο σετ εντολών γενικού-σκοπού. Τα στοιχεία μπορούν να διαβάζουν δεδομένα από μια κοινή global μνήμη και με τις καινούριες GPUs επίσης να γράφουν πίσω σε οποιεσδήποτε τοποθεσίες στην κοινή global μνήμη.

### 2.4.1. Μονάδα Επεξεργασίας Γραφικών Γενικής Χρήσης (GPGPU)

Η αντιστοίχιση της υπολογιστικής γενικής χρήσης στην GPU χρησιμοποιεί υλικό γραφικών με τον ίδιο περίπου τρόπο όπως και κάθε εφαρμογή γραφικών. Εξαιτίας αυτής της ομοιότητας, είναι ταυτόχρονα και πιο εύκολο και πιο δύσκολο να εξηγηθεί η διαδικασία. Απ' τη μια, οι διαδικασίες είναι ίδιες και εύκολες στην κατανόηση, από την άλλη η ορολογία είναι διαφορετική ανάμεσα στα γραφικά και στη γενική χρήση. Ξεκινάμε περιγράφοντας τον προγραμματισμό της GPU χρησιμοποιώντας την ορολογία των γραφικών και μετά χρησιμοποιούμε τα ίδια βήματα για να δείξουμε τον πιο απλό και άμεσο τρόπο με τον οποίο έχουν γραφτεί οι σημερινές υπολογιστικές εφαρμογές στις GPGPUs.

Προγραμματίζοντας μια GPU για γραφικά:

Ξεκινάμε με τον ίδιο GPU αγωγό διοχέτευσης που περιγράφτηκε πιο πάνω εστιάζοντας στα προγραμματίσιμα στοιχεία αυτού του αγωγού.

Ο προγραμματιστής καθορίζει τη γεωμετρία που καλύπτει μια περιοχή στην οθόνη. Ο rasterizer δημιουργεί ένα fragment σε κάθε θέση pixel που καλύπτεται από αυτή τη γεωμετρία.

Κάθε fragment τμήμα φωτοσκιάζεται (shaded) από το πρόγραμμα fragment.

Το πρόγραμμα fragment υπολογίζει την τιμή του fragment με ένα συνδυασμό μαθηματικών εφαρμογών και διαβάσματος global μνήμης από μια global texture μνήμη.

Το αποτέλεσμα, είναι μια εικόνα που μπορεί να χρησιμοποιηθεί ως υπόστρωμα σε μελλοντικά περάσματα από τον αγωγό γραφικών.

Προγραμματίζοντας μια GPU για προγράμματα γενικής χρήσης :

Μια από τις ιστορικές δυσκολίες στο προγραμματισμό GPGPU εφαρμογών είναι το ότι παρά το ότι οι γενικού-σκοπού εργασίες τους δεν έχουν τίποτα να κάνουν με γραφικά, οι εφαρμογές έπρεπε ακόμα να προγραμματίζονται με την χρήση APIs γραφικών. Επιπροσθέτως το πρόγραμμα έπρεπε να δομηθεί με όρους αγωγού γραφικών, με τις προγραμματίσιμες μονάδες προσβάσιμες μόνο ως ένα ενδιάμεσο βήμα σε αυτόν τον αγωγό, τη στιγμή που ο προγραμματιστής σχεδόν σίγουρα θα προτιμούσε να έχει άμεση πρόσβαση στις προγραμματίσιμες μονάδες. Τα σύγχρονα περιβάλλοντα προγραμματισμού, που περιγράφονται διεξοδικά αργότερα, επιλύουν αυτή τη δυσκολία παρέχοντας ένα πιο φυσικό, άμεσο, μη γραφικό interface στο υλικό, και ειδικά στις προγραμματιστικές μονάδες. Σήμερα, οι εφαρμογές των GPU υπολογισμών δομούνται με τον ακόλουθο τρόπο:

Ο προγραμματιστής ορίζει άμεσα το υπολογιστικό πεδίο ενδιαφέροντος σαν ένα δομημένο πλέγμα από νήματα.

Ένα πρόγραμμα SPMD γενικού-σκοπού υπολογίζει την τιμή του κάθε νήματος

Η τιμή για κάθε νήμα υπολογίζεται από ένα συνδυασμό μαθηματικών λειτουργιών και αμφότερες προσβάσεις για διάβασμα ή γράψιμο στην global μνήμη. Αντίθετα με τις προηγούμενες δυο μεθόδους, το ίδιο buffer μπορεί να χρησιμοποιηθεί και για την ανάγνωση και για το γράψιμο, επιτρέποντας πιο ευέλικτους αλγορίθμους (για παράδειγμα, in-place αλγορίθμους που χρησιμοποιούν λιγότερη μνήμη)

Το τελικό buffer στην global μνήμη μπορεί μετά να χρησιμοποιηθεί σαν είσοδος για μελλοντικούς υπολογισμούς.

Αυτό το μοντέλο προγραμματισμού είναι αποδοτικό για πολλούς λόγους. Πρώτον επιτρέπει στο υλικό να εκμεταλλευτεί πλήρως την παραλληλία δεδομένων της εφαρμογής καθορίζοντας ρητά αυτή την παραλληλία στο πρόγραμμα. Στη συνέχεια, βρίσκει μία προσεχτική ισορροπία ανάμεσα στη γενικότητα (μια πλήρως προγραμματίσιμη ρουτίνα σε κάθε στοιχείο) και τους περιορισμούς για να εξασφαλίσει καλή απόδοση (το μοντέλο SPMD, οι περιορισμοί στη διακλάδωση για αποτελεσματικότητα, περιορισμοί στην επικοινωνία δεδομένων ανάμεσα σε στοιχεία και ανάμεσα σε kernels/προσπελάσεις, και ούτω καθεξής). Τελικά, η άμεση πρόσβαση στις προγραμματίσιμες μονάδες εξαλείφει πολλή από την πολυπλοκότητα που αντιμετωπιζόταν από προηγούμενους GPGPU προγραμματιστές, που εγκλωβίζονταν στο interface γραφικών, για προγραμματισμό γενικής χρήσης. Σαν αποτέλεσμα, τα

προγράμματα εκφράζονται πιο συχνά σε μια οικεία γλώσσα προγραμματισμού (όπως στο C-like περιβάλλον προγραμματισμού CUDA της NVIDIA ) και είναι πιο απλά και πιο εύκολα να υλοποιηθούν και να απασφαλματοποιηθούν (και γίνονται ακόμη πιο εύκολα όσο τα εργαλεία προγραμματισμού εξελίσονται). Το αποτέλεσμα είναι ένα μοντέλο προγραμματισμού που επιτρέπει στους χρήστες του να εκμεταλλεύονται πλήρως το ισχυρό υλικό της GPU αλλά επίσης επιτρέπει και ένα όλο και περισσότερο υψηλού επιπέδου μοντέλο προγραμματισμού που καθιστά δυνατή την παραγωγική συγγραφή πολύπλοκων εφαρμογών.{10}

## 2.5. Ιστορική Εξέλιξη Παράλληλων Υπολογιστικών Συστημάτων

Την παράλληλη υπολογιστική τη συναντούμε για πρώτη φορά σε μια πλακέτα περίπου το 100 π.Χ. Η πλακέτα εκείνη ήλεγχε τρεις υπολογιστικές θέσεις, και είχε την ικανότητα να κάνει τον έλεγχο ταυτόχρονα. Οι πολλαπλές αυτές θέσεις χρησιμοποιούνταν είτε για να προσφέρουν μεγαλύτερη αξιοπιστία είτε υψηλότερες ταχύτητες υπολογισμού μέσω της χρήσης του παραλληλισμού. Με τον ίδιο τρόπο που μάθαμε να πετάμε, όχι κατασκευάζοντας μια μηχανή που κουνάει τα φτερά της σαν πτηνό, αλλά εφαρμόζοντας τους νόμους της αεροδυναμικής που προέκυψαν από την παρατήρηση των φυσικών φαινομένων, διαμορφώσαμε τον παράλληλο υπολογισμό στηριζόμενοι στη παρατήρηση βιολογικών διαδικασιών. Η λειτουργία του παράλληλου υπολογισμού μπορεί να γίνει σαφής από τη λειτουργία των νευρώνων του εγκεφάλου. Η ολική ταχύτητα με την οποία εκτελούνται πολύπλοκοι υπολογισμοί από τους νευρώνες είναι εξαιρετικά υψηλή, παρόλο που η απόκριση του κάθε νευρώνα ξεχωριστά είναι πολύ αργή (της τάξης των msec).

Οι αρχές της παράλληλης υπολογιστικής πολλαπλών εντολών πολλαπλών δεδομένων, γνωστό ως MIMD parallelism (Multiple Instruction Multiple Data) εντοπίζονται στη δουλειά των Federico Luigi και Conte Menabrea “Σχεδιασμός της αναλυτικής μηχανής που εφευρεθεί project του οποίου ηγούνταν ο Gene Amdahl. Ο 704 ήταν ο πρώτος εμπορικά διαθέσιμος υπολογιστής που χρησιμοποιούσε πλήρως αυτοματοποιημένα τις εντολές κινητής υποδιαστολής. {3}

Τον Απρίλιο του 1958, ο S. Gill (Ferranti) πρότεινε τον παράλληλο προγραμματισμό και την ανάγκη για διακλάδωση και αναμονή (branching and waiting). Την ίδια χρονιά οι ερευνητές της IBM John Cocke and Daniel Slotnick

σκέφτηκαν για πρώτη φορά να χρησιμοποιήσουν τον παραλληλισμό στους αριθμητικούς υπολογισμούς. Το 1962 η Burroughs Corporation εισήγαγε τον D825, έναν υπολογιστή τεσσάρων επεξεργαστών, ο οποίος είχε δυνατότητα πρόσβασης σε 16 modules μνήμης μέσω ενός διακόπτη. Το 1967 οι Amdahl και Slotnick παρουσίασαν στο συνέδριο του American Federation of Information Processing Societies την εργασία τους πάνω στην σκοπιμότητα της παράλληλης υπολογιστικής. Κατά τη διάρκεια αυτού του συνεδρίου προέκυψε ο νόμος του Amdahl που όριζε τη μέγιστη δυνατή επιτάχυνση που προκύπτει μέσω του παραλληλισμού.

Το 1969, η αμερικάνικη εταιρία Honeywell εισήγαγε το πρώτο της παράλληλο σύστημα. Έναν συμμετρικό πολυεπεξεργαστή ικανό να τρέχει παράλληλα μέχρι και σε οκτώ επεξεργαστές. Ο C.mmp ήταν ένας από τους πρώτους πολυεπεξεργαστές ο οποίος προέκυψε από έρευνες στο Canergie Mellon University το 1970.

Τους παράλληλους υπολογιστές SIMD(Single Instruction Multiple Data) τους συναντούμε πρώτη φορά το 1970. Το κίνητρο πίσω από τους πρώιμους υπολογιστές SIMD στάθηκε η προσπάθεια αντιρρόπησης της καθυστέρησης εισόδου πολλαπλών εντολών στη μονάδα ελέγχου του επεξεργαστή. Το 1964, ο Slotnick πρότεινε την κατασκευή ενός υπολογιστή υψηλής δυνατότητας παραλληλισμού για λογαριασμό του Lawrence Livermore National Laboratory. Ο σχεδιασμός του χρηματοδοτήθηκε από την US Air Force, κι έτσι δημιουργήθηκε ο ILIAC IV. Η καινοτομία στο σχεδιασμό του οφειλόταν στον συγκριτικά υψηλό παραλληλισμό, χρησιμοποιώντας 256 επεξεργαστές, που επέτρεπαν στον υπολογιστή να δουλεύει με μεγάλα σύνολα δεδομένων, διαδικασία που αργότερα έγινε γνωστή με τον όρο διανυσματικός επεξεργαστής (vector processing). Παρόλα αυτά ο ILLIAC IV ήταν ο πιο άσημος υπερ-υπολογιστής , λόγω του ότι ολοκληρώθηκε μόλις κατά το ένα τέταρτο σε διάστημα 11 χρόνων και κόστισε τέσσερις φορές περισσότερο από τον αρχικό προϋπολογισμό. Όταν τελικά το 1976 ήταν έτοιμος να εκτελέσει την πρώτη του πραγματική εφαρμογή, είχε ήδη παραγκωνιστεί από νεώτερους εμπορικούς υπερ-υπολογιστές όπως για παράδειγμα τον Cray-1.

Οι προσπάθειες για την ανάπτυξη των παράλληλων υπολογιστών συνεχίστηκαν τις δεκαετίες του ’80 και ’90 με βασικό στόχο και πεδία εφαρμογών την βελτιστοποίηση της παράλληλης επικοινωνίας μεταξύ επεξεργαστών και διαφορετικών επιπέδων μνήμης. Τομή στην ιστορική αυτή εξέλιξη αποτέλεσαν οι GPUs.[2]

### 2.5.1. Ιστορικό των GPUs

*ΔΕΚΑΕΤΙΑ ΤΟΥ 1970*

Τα ολοκληρωμένα κυκλώματα της ANTIC και της CTIA παράχθηκαν ώστε στους οκτάμπιτους υπολογιστές Atari να γίνεται ο έλεγχος σε γραφικά μαζί με κείμενα από το υλικό. Το ολοκληρωμένο κύκλωμα της ANTIC ήταν ένας ειδικής χρήσης επεξεργαστής για αντιστοίχιση (σε μια προγραμματίσιμη έκδοση) κειμένων και γραφικών στην έξοδο video. Ο σχεδιαστής του ολοκληρωμένου κυκλώματος της ANTIC, Jay Miner, σχεδίασε στη συνέχεια το ολοκληρωμένο κύκλωμα γραφικών για τον Commodore Amiga.

*ΔΕΚΑΕΤΙΑ ΤΟΥ 1980*

Ο Commodore Amiga ήταν ο πρώτος υπολογιστής ευρείας κυκλοφορίας που περιελάμβανε ένα blitter στο video υλικό του, και το σύστημα γραφικών 8514 της IBM ήταν μια από τις πρώτες κάρτες γραφικών υπολογιστή που πρωτοεφάρμοσε δισδιάστατες πρωταρχικές λειτουργίες στο υλικό.

Το Amiga ήταν μοναδικό, για την εποχή του, δεδομένου ότι χαρακτήρισε αυτό που σήμερα αναγνωρίζεται ως πλήρης επιταχυντής γραφικών, αναθέτοντας σχεδόν όλες τις λειτουργίες video του υλικού, συμπεριλαμβανομένου του σχεδιασμού γραμμών, του γεμίσματος περιοχής, της μεταφοράς block εικόνας, και ενός συνεπεξεργαστή γραφικών με το δικό του (αν και πρωτόγονο) σετ εντολών. Προγενέστερα (και αρκετό χρονικό διάστημα μετά στα περισσότερα συστήματα) μία CPU γενικής χρήσης έπρεπε να μπορεί να χειριστεί κάθε πτυχή του σχεδιασμού της απεικόνισης.

*ΔΕΚΑΕΤΙΑ ΤΟΥ 1990*

Από της αρχές της δεκαετίας του '90, η άνοδος της Microsoft Windows προκάλεσε ένα κύμα ενδιαφέροντος στα υψηλών επιδόσεων, υψηλής ευκρίνειας δισδιάστατα bitmapped γραφικά (που ήταν προηγουμένως η περιοχή σταθμών εργασίας της Unix και της Apple Macintosh). Για την αγορά υπολογιστών, η κυριαρχία των Windows σήμαινε ότι οι προμηθευτές γραφικών θα μπορούσαν τώρα να στρέψουν την προσπάθεια ανάπτυξης σε ένα ενιαίο interface προγραμματισμού, Graphics Device Interface (GDI).

Το 1991, η S3 Grafics εισήγαγε το πρώτο απλό ολοκληρωμένο κύκλωμα δισδιάστατου επιταχυντή, το S3 86C911 (το οποίο οι σχεδιαστές του ονόμασαν μετά Porsche 911 ως ένδειξη της αύξησης απόδοσης που υπόσχονταν). Το 86C911

πυροδότησε ένα πλήθος μιμητών: μέχρι το 1995, όλοι οι σημαντικοί κατασκευαστές ολοκληρωμένων κυκλωμάτων γραφικών για PC είχαν προσθέσει τη δισδιάστατη υποστήριξη επιτάχυνσης στα κυκλώματα τους. Σε εκείνο το χρονικό διάστημα, οι καθορισμένης λειτουργίας επιταχυντές των Windows είχαν ξεπεράσει σε απόδοση τους ακριβούς γενικής χρήσης συνεπεξεργαστές γραφικών των Windows, και έτσι οι τελευταίοι σταδιακά αποσύρθηκαν από την αγορά υπολογιστών.

Καθ' όλη τη διάρκεια της δεκαετίας του '90, η δισδιάστατη επιτάχυνση GUI συνέχισε να εξελίσσεται. Καθώς οι κατασκευαστικές δυνατότητες βελτιώνονταν, το ίδιο συνέβαινε με το επίπεδο ολοκλήρωσης των ολοκληρωμένων κυκλωμάτων γραφικών. Επιπλέον APIs δημιουργήθηκαν για ποικίλες εφαρμογές, όπως η βιβλιοθήκη γραφικών WinG της Microsoft για τα Windows 3.x, και αργότερα το DirectDraw interface για την επιτάχυνση υλικού των δισδιάστατων παιχνιδιών μέσα στα WINDOWS 95 και αργότερα.

Στις αρχές και τα μέσα της δεκαετίας του '90, τα βοηθητικά στη CPU τρισδιάστατα γραφικά πραγματικού χρόνου γινόταν όλο και περισσότερο κοινά στα παιχνίδια υπολογιστών και κονσόλων, τα οποία οδήγησαν σε μια αυξανόμενη δημόσια ζήτηση για 3D γραφικά επιταχυνόμενα μέσω υλικού . Τα αρχικά παραδείγματα του ευρείας κυκλοφορίας υλικού τρισδιάστατων γραφικών μπορούν να βρεθούν σε κονσόλες παιχνιδιών πέμπτης γενεάς όπως PlayStation και η Nintendo 64. Στον κόσμο των υπολογιστών, αξιοσημείωτες αποτυχημένες πρώτες προσπάθειες δημιουργίας χαμηλού κόστους κυκλωμάτων τρισδιάστατων γραφικών ήταν το S3 ViRGE, ATI Rage, και Matrox Mystique. Αυτά τα κυκλώματα ήταν ουσιαστικά δισδιάστατοι επιταχυντές προηγούμενης γενιάς με ενσωματωμένα τρισδιάστατα χαρακτηριστικά. Πολλά ήταν ακόμα και pin-compatible με τα κυκλώματα προηγούμενης γενιάς για ευκολία εφαρμογής και ελαχιστοποίηση του κόστους. Αρχικά, η απόδοση των τρισδιάστατων γραφικών ήταν δυνατή μόνο με τους ιδιαίτερους πίνακες που χρησιμοποιούνταν για την επιτάχυνση των τρισδιάστατων λειτουργιών (και την έλλειψη της δισδιάστατης επιτάχυνσης GUI εξ ολοκλήρου) όπως ο 3dfx Voodoo. Εντούτοις, καθώς η τεχνολογία κατασκευής προχωρούσε, το βίντεο, η δισδιάστατη επιτάχυνση GUI, και η τρισδιάστατη λειτουργικότητα ενσωματώθηκαν σε ένα τσιπ. Τα πρώτα σετ κυκλωμάτων που το πέτυχαν αυτό αξιομνημόνευτα καλά ήταν το Verite της Rendition.

ΔΕΚΑΕΤΙΑ 2000 ΩΣ ΣΗΜΕΡΑ

Με την εμφάνιση του OpenGL API και της παρόμοιας λειτουργίας σε DirectX, οι GPUs πρόσθεσαν το προγραμματίσιμο shading στις ικανότητές τους. Κάθε pixel θα μπορούσε τώρα να υποβληθεί σε επεξεργασία από ένα σύντομο πρόγραμμα που θα μπορούσε να περιλάβει επιπρόσθετες image textures ως εισόδους, και κάθε γεωμετρικό vertex θα μπορούσε επιπλέον να υποβληθεί σε επεξεργασία από ένα σύντομο πρόγραμμα προτού προβληθεί στην οθόνη. Η NVIDIA ήταν η πρώτη που παρήγαγε ένα ολοκληρωμένο κύκλωμα ικανό για προγραμματίσιμο shading, την GeForce 3 (κωδικά ονομαζόμενο NV20). Μέχρι τον Οκτώβριο του 2002, με την εισαγωγή του ATI Radeon 9700 (που είναι γνωστό επίσης ως R300), του πρώτου επιταχυντή Direct3D 9.0 στον κόσμο, pixel και vertex shaders μπορούσαν να εφαρμόσουν looping και lengthy floating point υπολογισμούς, και γενικά γίνονταν γρήγορα τόσο ευέλικτα όσο οι CPUs, και τα μεγέθη γρηγορότερα για τις διαδικασίες εικόνας-πινάκων.Το Pixel shading χρησιμοποιείται συχνά για εφαρμογές όπως η αντιστοίχιση προσκρούσεων, που προσθέτει υφή, για να κάνει ένα αντικείμενο να φανεί λαμπερό, θαμπό, τραχύ, ή ακόμα και καμπύλο ή με προεξοχές.

Δεδομένου ότι η δύναμη επεξεργασίας των GPUs έχει αυξηθεί, το ίδιο συμβαίνει με την απαίτησή τους για ηλεκτρική ενέργεια. Τα GPUs υψηλής απόδοσης συχνά καταναλώνουν περισσότερη ενέργεια από τις τρέχουσες CPUs.

Σήμερα , οι παράλληλες GPUs έχουν αρχίσει να κάνουν υπολογιστικά άλματα σε σχέση με τις CPUs, και μία ερευνητική υπο-περιοχή, οι GPGPUs για γενικής χρήσης υπολογισμούς σε GPUs έχει βρει εφαρμογές σε διάφορα πεδία τόσο διαφορετικά όπως η εξερεύνηση πετρελαίου, η επιστημονική επεξεργασία εικόνας, η γραμμική άλγεβρα, η τρισδιάστατη αναδόμηση και ακόμα και ο προσδιορισμός τιμών μετοχών.

Υπάρχει αυξανόμενη πίεση στους κατασκευαστές GPU από χρήστες GPGPU για βελτίωση του σχεδιασμού του υλικού, που συνήθως εστιάζει στην προσθήκη περισσότερης ευελιξίας στο μοντέλο προγραμματισμού.

# 3. Η Γλώσσα Προγραμματισμού CUDA

## 3.1. Λογισμικό παράλληλης Υπολογιστικής

Διάφορες καλά σχεδιασμένες τεχνολογίες λογισμικού είναι διαθέσιμες για τη χρησιμοποίηση υλικού παράλληλου υπολογισμού. Παρακάτω παρουσιάζονται οι σημαντικότερες εξ αυτών. Σημειώστε ότι κάθε μία από αυτές είναι κατάλληλη για μία συγκεκριμένη αρχιτεκτονική υλικού.

### 3.1.1. Threads

Το νήμα (thread) είναι μία σειρά από εντολές που μπορεί να προγραμματιστούν ως μία ανεξάρτητη οντότητα. Είναι σημαντικό να κατανοήσουμε τη διαφορά ανάμεσα σε ένα thread και σε μία διεργασία. Η διεργασία περιλαμβάνει δύο είδη πληροφοριών: πόρους που είναι διαθέσιμοι για το σύνολο της διεργασίας, όπως οι εντολές του προγράμματος, δεδομένα τύπου global και καταλόγους αρχείων εργασίας, οντότητες με δυνατότητα χρονοπρογραμματισμού, οι οποίες περιλαμβάνουν μετρητές προγράμματος και στοίβες. Το νήμα είναι μία οντότητα εντός μία διεργασίας που αποτελείται από εκείνο το μέρος της διεργασίας που έχει δυνατότητα χρονοπρογραμματισμού.

Σε ένα σύστημα με έναν επεξεργαστή, τα νήματα εκτελούνται τεμαχίζοντας τον χρόνο, αλλά τα συστήματα παράλληλων υπολογιστών κοινής μνήμης μπορούν να αναθέτουν νήματα σε διαφορετικούς επεξεργαστές.

### 3.1.2. Pthreads

Υπήρχαν πολλές βιβλιοθήκες στη γλώσσα C για συγκεκριμένα συστήματα που κοινής μνήμης. Τώρα, ωστόσο, η βιβλιοθήκη Pthread είναι ένα πρότυπο βιβλιοθήκης νημάτων για πολλά συστήματα.

Σημειώνουμε ότι δεν είναι εύκολο να γράψουμε εφαρμογές multi-threaded στην γλώσσα C, ακόμα και αν χρησιμοποιήσουμε την βιβλιοθήκη Pthread. Δεδομένου ότι η βιβλιοθήκη αυτή προστέθηκε αργότερα στη γλώσσα C δεν υπάρχουν εγγυήσεις ότι οι αρχικές θεμελιώδεις βιβλιοθήκες είναι συμβατές με τα νήματα (thread-safe). Ο όρος thread-safe σημαίνει ότι μία δεδομένη εντολή βιβλιοθήκης μπορεί να εφαρμόζεται με τέτοιο τρόπο ώστε να μπορεί να εκτελεστεί σωστά από πολλαπλά ταυτόχρονα νήματα εκτέλεσης. Πρέπει να είμαστε προσεκτικοί στη χρήση thread-safe εντολών σε πολυνηματικό (multi-thread) προγραμματισμό. Η βιβλιοθήκη Pthread

χρησιμοποιείται κυρίως από επαγγελματίες προγραμματιστές συστημάτων για την υποστήριξη προηγμένων τεχνολογιών παράλληλου υπολογισμού όπως το OpenMP.

### 3.1.3. JavaThreads

Η γλώσσα Java υποστηρίζει τα νήματα ως ένα από τα βασικά της χαρακτηριστικά. Η βιβλιοθήκη Java παρέχει μία κλάση *Thread* η οποία υποστηρίζει μία πλούσια συλλογή από μεθόδους.

Παρόλο που η γλώσσα Java έχει σχεδιαστεί για να είναι thread-safe και παρέχει διάφορα μέσα για προγραμματισμό νημάτων, εξακολουθεί να είναι δύσκολο το να γράψεις αποτελεσματικά προγράμματα εφαρμογών σε Java. Τα εργαλεία της Java γενικά είναι κατάλληλα στα συστήματα προγραμματισμού εφαρμογών, όπως τα interfaces γραφικών-χρηστών και τα κατανεμημένα συστήματα, γιατί παρέχουν λειτουργίες συγχρονισμού οι οποίες είναι λεπτομερείς και αποτελεσματικές, αλλά μη δομημένες και πολύπλοκες. Μπορούν να θεωρηθούν μια συμβολική γλώσσα για προγραμματισμό νημάτων. Για το λόγο αυτό δεν είναι εύκολη η χρήση τους για στατιστικό προγραμματισμό.

### 3.1.4. MPI

Το MPI (Interface Μεταφοράς Μηνυμάτων - Message Passing Interface) είναι από τις πιο συνηθισμένες τεχνικές παράλληλου υπολογισμού. Προσδιορίζει μία βιβλιοθήκη για την προσθήκη μηχανισμών μεταφοράς μηνυμάτων σε ήδη υπάρχουσες γλώσσες όπως η Fortran ή η C.

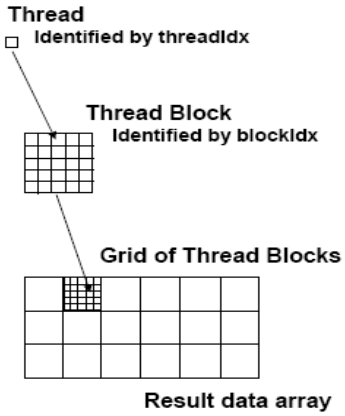
### 3.1.5. CUDA

Η CUDA (Compute Unified Device Architecture) είναι μία νέα τεχνολογία για γενικής χρήσης υπολογισμό στη GPU, η οποία καθιστά εύκολο στους χρήστες να αναπτύξουν γενικά προγράμματα GPU (Graphics Processing Unit).

## 3.2. Τα Βασικά Προγραμματιστικά Χαρακτηριστικά της CUDA

Η γλώσσα προγραμματισμού CUDA είναι μία επέκταση των γλωσσών C και C++. Ο προγραμματιστής γράφει ένα σειριακό κώδικα και καλεί παράλληλους "kernels" που μπορεί να είναι απλές συναρτήσεις ή και ολόκληρα προγράμματα. Ο kernel εκτελείται παράλληλα από παράλληλα νήματα. Ο προγραμματιστής οργανώνει αυτά τα νήματα σε μία ιεραρχική δομή από πλέγματα και blocks. Ένα block νημάτων είναι ένα σύνολο από συγχρονισμένα νήματα που μπορούν να συνεργάζονται μεταξύ

τους δια μέσου περιορισμών συγχρονισμού και μοιραζόμενη πρόσβαση σε πεδίο μνήμης που αντιστοιχεί στο εκάστοτε block. Το πλέγμα είναι ένα σύνολο από blocks νημάτων που καθένα από αυτά (τα block) μπορεί να εκτελείται ανεξάρτητα αλλά εκτελούνται παράλληλα.




*Σχήμα 6* {8}

Όταν καλείται ο kernel, ο προγραμματιστής καθορίζει το πλήθος των νημάτων σε κάθε block και το πλήθος των blocks σε κάθε πλέγμα. Σε κάθε νήμα δίνεται μία μοναδική ταυτότητα νήματος (threadIdx) μέσα στο block στο οποίο ανήκει, αριθμούμενη από 0, 1, 2, ... έως blockDim-1, όπου blockDim είναι η διάσταση – μέγεθος του block. Αντίστοιχα, σε κάθε block νημάτων δίνεται μία μοναδική ταυτότητα block (blockIdx) μέσα στο πλέγμα στο οποίο ανήκει. Η CUDA υποστηρίζει blocks νημάτων που περιέχουν μέχρι και 512 νήματα. Για ευκολία –ανάλογα και με τη φύση του προβλήματος- τα blocks και τα πλέγματα μπορούν να έχουν 1, 2 ή και 3 διαστάσεις ώστε να είναι δυνατή η πρόσβαση κατά την x, y ή και z διάσταση.

Ως ένα εξαιρετικά απλό παράδειγμα παράλληλου προγραμματισμού ας υποθέσουμε ότι μας δίνονται δύο διανύσματα x και y από n το πλήθος αριθμούς κινητής υποδιαστολής το καθένα και ότι θέλουμε να υπολογίσουμε το αποτέλεσμα του $y \leftarrow ax + y$ για κάποιο βαθμωτό μέγεθος a. Αυτός είναι ο λεγόμενος saxpy kernel οριζόμενος από τη βιβλιοθήκη βασικών γραμμικών υποπρογραμμάτων της άλγεβρας BLAS (basic linear algebra subprograms) library. Ο κώδικας για την εκτέλεση αυτού του υπολογισμού τόσο για σειριακή όσο και παράλληλη επεξεργασία με χρήση CUDA φαίνεται στο σχήμα 7.

**Computing y ← ax + y with a Serial Loop**

```
void saxpy_serial(int n, float alpha, float *x, float *y)
{
    for(int i = 0; i<n; ++i)
        y[i] = alpha*x[i] + y[i];
}

// Invoke serial SAXPY kernel
saxpy_serial(n, 2.0, x, y);
```

**Computing y ← ax + y in parallel using CUDA**

```
__global__
void saxpy_parallel(int n, float alpha, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;

    if( i<n )  y[i] = alpha*x[i] + y[i];
}

// Invoke parallel SAXPY kernel (256 threads per block)
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```

*Σχήμα 7* {9}

Στα προγράμματα γραμμένα σε CUDA η κλήση των παράλληλων συναρτήσεων kernel γίνεται με τη χρήση της εντολής

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

όπου dimGrid και dimBlock είναι δείκτες εώς και τριών στοιχείων -τύπου δεδομένων dim3- που προσδιορίζουν τη διάσταση των πλεγμάτων σε blocks και τη διάσταση των blocks σε νήματα αντίστοιχα. Όταν δεν προσδιορίζεται κάποια διάσταση τότε λαμβάνεται ως 1.

Στο συγκεκριμένο παράδειγμα φτιάχνουμε ένα πλέγμα που αντιστοιχεί ένα thread σε κάθε στοιχείο των διανυσμάτων και βάζει 256 νήματα σε κάθε block. Κάθε Thread υπολογίζει ένα δείκτη στοιχείου με βάση την thread ID και block ID και έπειτα κάνει τον επιθυμητό υπολογισμό στα αντίστοιχα στοιχεία του διανύσματος. Η παράλληλη και σειριακή εκδοχή αυτού του κώδικα μοιάζουν εξαιρετικά μεταξύ τους. Ο σειριακός κώδικας αποτελείται από ένα βρόχο όπου κάθε επανάληψη είναι ανεξάρτητη από όλες τις υπόλοιπες. Τέτοιοι βρόχοι μπορούν μηχανιστικά να μετασχηματιστούν σε παράλληλους kernels. Κάθε επανάληψη μετασχηματίζεται σε ένα ανεξάρτητο thread. Αντιστοιχίζοντας κάθε thread σε κάθε στοιχείο εξόδου αποφεύγουμε την ανάγκη για οποιονδήποτε συγχρονισμό μεταξύ των νημάτων όταν γράφουμε τα αποτελέσματα στη μνήμη.

Ο κώδικας για έναν CUDA kernel είναι απλούστατα ένας κώδικας–συνάρτηση σε γλώσσα C για ένα νήμα. Κατ' αυτόν τον τρόπο η CUDA είναι πολύ σαφής και αρκετά ευκολότερη στη γραφή του κώδικα απ' ότι ένας παράλληλος κώδικας με διανυσματικές λειτουργίες. Ο παραλληλισμός προσδιορίζεται με σαφήνεια απλά και μόνο καθορίζοντας τις διαστάσεις των πλεγμάτων και blocks τη στιγμή φόρτωσης – εκκίνησης του kernel.

Η παράλληλη εκτέλεση και η διαχείριση των νημάτων γίνονται αυτόματα. Η δημιουργία των threads, ο χρόνο-προγραμματισμός τους και ο τερματισμός τους ελέγχονται από το ίδιο το σύστημα. Ο προγραμματιστής δε χρειάζεται να καθορίσει άμεσα κάτι από όλα αυτά. Για την ακρίβεια στην αρχιτεκτονική Tesla των GPUs της NVIDIA όλη η διαχείριση των threads εκτελείται από το υλικό. Τα νήματα ενός block εκτελούνται ταυτόχρονα και συγχρονίζονται σε συγκεκριμένα όρια (barriers) (σημεία του κώδικα) καλώντας την εσωτερική συνάρτηση __sync threads(). Αυτή η εντολή εξασφαλίζει ότι κανένα thread που συμμετέχει σε μία ορισμένη διαδικασία δε θα συνεχίσει πέραν του ορίου, εάν δεν έχουν φτάσει στο συγκεκριμένο σημείο – όριο όλα τα νήματα. Εξασφαλίζεται ακόμα ότι όλα τα threads αφού περάσουν το συγκεκριμένο φράγμα μπορούν να δουν όλες τις εγγραφές που έγιναν στη μνήμη από αυτά τα νήματα πριν από το φράγμα. Έτσι τα threads σε ένα block μπορούν να επικοινωνούν μεταξύ τους γράφοντας και διαβάζοντας στην ανά block διαμοιραζόμενη μνήμη (per-block shared memory) στα όρια συγχρονισμού.

Καθώς, επομένως, τα νήματα σε ένα block μπορούν να μοιράζονται τη ίδια τοπική μνήμη και να συγχρονίζονται μεταξύ τους σε συγκεκριμένα φράγματα – όρια, θα βρίσκονται ουσιαστικά και στον ίδιο φυσικό επεξεργαστή. Παρόλα αυτά το πλήθος των blocks μπορεί να ξεπερνά κατά πολύ το πλήθος των επεξεργαστών. Κάτι τέτοιο εικονικοποιεί τα στοιχεία επεξεργασίας και δίνει στον προγραμματιστή την ευελιξία να παραλληλοποιήσει το πρόβλημα σε όποιο βαθμό του είναι βολικότερο. Αυτό επιτρέπει τη διαισθητική ανάλυση του προβλήματος καθώς το πλήθος των blocks μπορεί να προσδιοριστεί από το μέγεθος των δεδομένων που πρόκειται να επεξεργαστούν, παρά από το πλήθος των επεξεργαστών στο εκάστοτε σύστημα όπου προγραμματίζουμε. Κατ' αυτόν τον τρόπο το ίδιο πρόγραμμα CUDA μπορεί να τρέξει σε ευρέως κυμαινόμενο πλήθος πυρήνων επεξεργασίας.

Για να είναι δυνατή η διαχείριση της εικονικής αυτής επεξεργασίας δεδομένων, η CUDA απαιτεί τα blocks των νημάτων να εκτελούνται ανεξάρτητα. Πρέπει να είναι δυνατή η εκτέλεση των blocks με οποιαδήποτε σειρά. Τα διαφορετικά blocks δεν

έχουν τρόπο απευθείας επικοινωνίας μεταξύ τους, αλλά μπορούν να συντονίζουν τις δράσεις τους χρησιμοποιώντας λειτουργίες ατομικής μνήμης (atomic memory operations) πάνω στην global μνήμη η οποία και είναι ορατή από όλα τα threads.

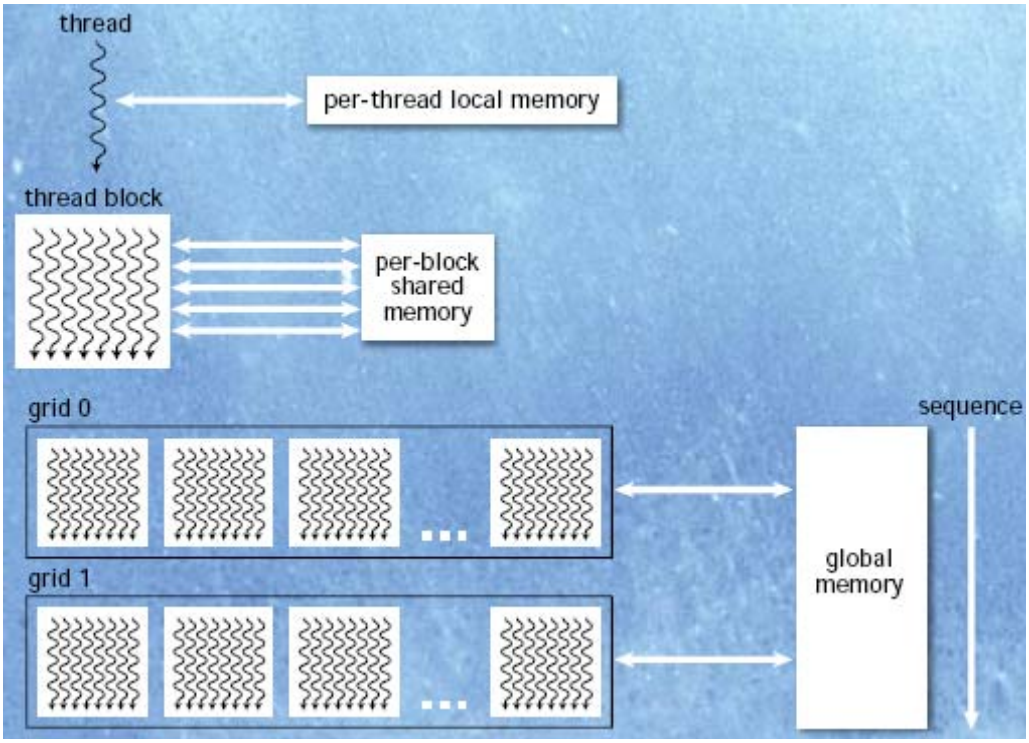
Αυτή η ιδιότητα ανεξαρτησίας των blocks, επιτρέπει τον προγραμματισμό τους με οποιαδήποτε σειρά κατά μήκος οποιουδήποτε αριθμού πυρήνων και έτσι το μοντέλο προγραμματισμού της CUDA εμφανίζει πρακτική ισχύ τόσο σε αυθαίρετο πλήθος πυρήνων όσο και σε διαφορετικές παράλληλες αρχιτεκτονικές. Βοηθάει επίσης στην αποφυγή πιθανών αδιεξόδων (deadlock).

Μία εφαρμογή μπορεί να εκτελεί πολλαπλά πλέγματα είτε ανεξάρτητα είτε όχι μεταξύ τους. Τα ανεξάρτητα πλέγματα μπορούν να εκτελούνται ταυτόχρονα δεδομένου επαρκών υπολογιστικώ πόρων του υλικού. Τα αλληλοεξαρτώμενα πλέγματα εκτελούνται σειριακά υπό τον όρο ύπαρξης σαφούς φράγματος (barrier) μεταξύ τους που ορίζεται στον kernel, εξασφαλίζοντας έτσι ότι όλα τα blocks του πρώτου για παράδειγμα πλέγματος θα ολοκληρώσουν τους υπολογισμούς τους προτού εκκινήσει το δεύτερο πλέγμα.

Τα νήματα μπορούν να έχουν πρόσβαση στα δεδομένα από πολλαπλές θέσεις μνήμης κατά τη διάρκεια της εκτέλεσης τους. Κάθε νήμα έχει μία ιδιωτική τοπική μνήμη (private local memory). Η CUDA χρησιμοποιεί αυτή τη μνήμη για μεταβλητές ιδιωτικές σε κάθε thread (thread-private variables) που δε χωράνε στους registers του νήματος, καθώς επίσης και για δομές στοίβας (stack frames) ή για υπερχείλιση των registers (register spilling). Κάθε block από νήματα έχει και τη διαμοιραζόμενη μνήμη (shared memory) που είναι ορατή από όλα τα νήματα του block και έχει την ίδια διάρκεια ζωής με εκείνη του αντίστοιχου block. Τέλος όλα τα threads έχουν πρόσβαση στην ίδια global μνήμη. Στο πρόγραμμα – κώδικα δηλώνονται μεταβλητές στη διαμοιραζόμενη ή στη global μνήμη με το χαρακτηριστικό __shared__ ή __global__. Στην αρχιτεκτονική Tesla αυτές οι διαφορετικές περιοχές μνήμης διαχωρίζονται και φυσικά. Η μνήμη ανά block (per-block shared memory) είναι μία χαμηλής καθυστέρησης on-chip μνήμη RAM, ενώ η μνήμη global βρίσκεται στην DRAM της κάρτας γραφικών.

Η διαμοιραζόμενη μνήμη (Shared memory) αναμένεται να είναι μία γρήγορη μνήμη κοντά σε κάθε επεξεργαστή, περίπου σαν στην μνήμη L1 cache. Για το λόγο αυτό παρέχει υψηλής απόδοση επικοινωνία μεταξύ των δεδομένων που μοιράζονται τα threads σε κάθε block. Καθώς, μάλιστα, έχει και την ίδια διάρκεια ζωής με το block στο οποίο αντιστοιχεί, ο kernel κώδικας κατά κανόνα θα αρχικοποιήσει τα

δεδομένα στις μεταβλητές στη shared μνήμη, θα κάνει τους υπολογισμούς, και θα αντιγράψει τα αποτελέσματα της μνήμης shared πίσω στη global μνήμη. Τα blocks νημάτων των σειριακά εξαρτώμενων πλεγμάτων επικοινωνούν δια μέσου της global μνήμης από την οποία διαβάζουν δεδομένα και στην οποία γράφουν αποτελέσματα. {9}




*Σχήμα 8* {9}

Στο σχήμα 8 φαίνονται τα φωλιασμένα επίπεδα των threads, blocks και grids. Καθώς και τα αντίστοιχα επίπεδα μνήμης που τα επίπεδα αυτά μοιράζονται μεταξύ τους: local, shared, και global μνήμη για κάθε thread, thread-block, thread-grid και για κάθε εφαρμογή.

Ένα πρόγραμμα διαχειρίζεται τον προσβάσιμο στους kernels χώρο μνήμης global μέσω κλήσεων στην CUDA, όπως η cudaMalloc() και η cudaFree().Οι kernels μπορούν να εκτελούνται σε φυσικά ξεχωριστές συσκευές, όπως στην περίπτωση όπου εκτελούμε kernels στη GPU. Συνεπώς, η εφαρμογή πρέπει να χρησιμοποιεί την cudaMemcpy() για να αντιγράφει δεδομένα μεταξύ του δεσμευμένου χώρου και του συστήματος μνήμης του οικοδεσπότη (host).

Το μοντέλο προγραμματισμού της CUDA είναι παρόμοιο σε τεχνοτροπία με το οικείο μοντέλο Μίας Εντολής – Πολλαπλών Δεδομένων (SPMD). Εκφράζει την παραλληλία ρητά, και κάθε kernel εκτελεί ένα σταθερό αριθμό από νήματα. Η CUDA,

ωστόσο, είναι πιο ευέλικτη από τις περισσότερες από τις υλοποιήσεις του SPMD μοντέλου, επειδή κάθε κλήση του πυρήνα δημιουργεί δυναμικά ένα νέο πλέγμα με το σωστό αριθμό από block νημάτων και το σωστό αριθμό νημάτων για το εκάστοτε βήμα της εφαρμογής. Ο προγραμματιστής μπορεί να χρησιμοποιήσει ένα βολικό βαθμό παραλληλισμού για κάθε πυρήνα, αντί να χρειάζεται να σχεδιάσει όλες τις φάσεις του υπολογισμού για να χρησιμοποιήσει τον ίδιο αριθμό από νήματα.

Τα παράλληλα νήματα ενός block δείχνουν τον υψηλού βαθμού παραλληλισμό νημάτων και δεδομένων. Τα ανεξάρτητα block νημάτων ενός πλέγματος φανερώνουν τον χαμηλότερου βαθμού παραλληλισμό νημάτων και δεδομένων. Ένας kernel είναι απλά κώδικας C για ένα νήμα της ιεραρχίας.

## 3.3. Το Προγραμματιστικό Μοντέλο της CUDA

Η CUDA κρύβει τις λεπτομέρειες χαμηλού επιπέδου υλικό. Τα βασικά βήματα του γενικού μοντέλου παράλληλου προγραμματισμού είναι τα εξής:

- Αντιγραφή δεδομένων από τη host μνήμη (CPU) στην device μνήμη (GPU). Ακριβώς λόγο του φυσικού ορίου στην ταχύτητα προσπέλασης της μνήμης, η μεταφορά δεδομένων από την host memory στην device memory (bottleneck) καθυστερεί τη συνολική ταχύτητα. Έτσι μία αποτελεσματική μέθοδος είναι η οργάνωση των δεδομένων σε texture, που μπορούμε να επεξεργαστούμε με texture functions.

Η CPU προγραμματίζει και προσδιορίζει την εκτέλεση του kernel. Αυτό το στάδιο περιέχει τα εξής βήματα:

- Προσδιορισμός των ρυθμίσεων εκτέλεσης του kernel. Οργάνωση των δεδομένων εισόδου και δέσμευση των blocks.
- Μεταφορά δεδομένων από την global μνήμη στην shared. Κάτι τέτοιο δεν είναι απαραίτητο από άποψη λειτουργικότητας, αλλά είναι εξαιρετικά αποδοτικό εφόσον γίνει σωστά. Για παράδειγμα η σάρωση και αντιγραφή της global μνήμης να γίνει με τρόπο που να μην υπάρχουν μετά συγκρούσεις (memory access conflicts) κατά την πρόσβαση της shared μνήμης.
- Εκτέλεση των υπολογισμών του kernel
- Εγγραφή των αποτελεσμάτων πίσω στη host μνήμη.

### 3.3.1. Περιορισμοί στο Προγραμματιστικό Μοντέλο της CUDA

Όταν κάποιος αναπτύσσει προγράμματα CUDA, είναι σημαντικό να καταλαβαίνει τα σημεία στα οποία το μοντέλο της CUDA είναι περιορισμένο, κυρίως για λόγους αποδοτικότητας. Τα νήματα και τα blocks νημάτων μπορούν να δημιουργηθούν μόνο επικαλούμενοι έναν παράλληλο πυρήνα και όχι μέσα από έναν παράλληλο πυρήνα. Μαζί με την απαιτούμενη ανεξαρτησία των blocks, αυτό εφιστά δυνατό να εκτελούνται τα προγράμματα CUDA με έναν απλό χρονοπρογραμματιστή που εισάγει ελάχιστη καθυστέρηση χρόνου στην εκτέλεση. Στην πραγματικότητα, η αρχιτεκτονική Tesla εφαρμόζει διαχείριση και χρονοπρογραμματισμό των νημάτων και των blocks μέσω υλικού.

Ο παραλληλισμός διαδικασιών μπορεί να εκφραστεί στο επίπεδο του block νημάτων, αλλά οι barriers συγχρονισμού των blocks δεν είναι κατάλληλοι για υποστήριξη παράλληλων διαδικασιών μεταξύ των νημάτων ενός block. Για να κατασταθεί δυνατό στα προγράμματα CUDA να εκτελούνται σε οποιονδήποτε αριθμό από επεξεργαστές, η επικοινωνία μεταξύ των blocks από νήματα του ίδιου πλέγματος ενός πυρήνα δεν επιτρέπεται – πρέπει να εκτελούνται ανεξάρτητα. Αφού η CUDA απαιτεί να είναι ανεξάρτητα τα blocks και επιτρέπει σε αυτά να εκτελούνται με οποιαδήποτε σειρά, ο συνδυασμός των αποτελεσμάτων που προέρχονται από πολλαπλά blocks πρέπει εν γένει να γίνεται εκκινώντας έναν δεύτερο πυρήνα σε ένα νέο πλέγμα από blocks νημάτων. Ωστόσο, πολλαπλά blocks μπορούν να συντονίσουν την δουλειά τους χρησιμοποιώντας ατομικές λειτουργίες στη μνήμη global (για παράδειγμα να χειριστούν μια δομή δεδομένων)

Αναδρομικές λειτουργίες (recursive) δεν επιτρέπονται στους kernels της CUDA. Οι διαδικασίες τύπου recursive είναι μη ελκυστικές σε έναν μαζικά παράλληλο kernel επειδή η εξασφάλιση χώρου στοίβας για τις δεκάδες χιλιάδες των νημάτων που μπορεί να είναι ενεργά θα απαιτούσε σημαντικές ποσότητες μνήμης. Σειριακοί αλγόριθμοι που χρησιμοποιούν recursive διαδικασίες, όπως ο quicksort, είναι τυπικά καλύτερα εφαρμόσιμοι χρησιμοποιώντας παραλληλισμό δεδομένων, παρά χρησιμοποιώντας αναδρομικές λειτουργίες.

Για την υποστήριξη ενός ετερογενούς συστήματος που συνδυάζει μία CPU με μία GPU, κάθε μια με το δικό της σύστημα μνήμης, τα προγράμματα CUDA πρέπει να αντιγράφουν δεδομένα και αποτελέσματα μεταξύ της μνήμης host (CPU) και της μνήμης device (GPU). Το κόστος της αλληλεπίδρασης μεταξύ CPU και GPU και των μεταφορών των δεδομένων ελαχιστοποιείται χρησιμοποιώντας μηχανές DMA block-

transfer και γρήγορων αλληλοσυνδέσεων. Βέβαια, προβλήματα με ικανά μεγάλο μέγεθος για να χρειάζονται την απόδοση μιας GPU αντισταθμίζουν καλύτερα το κόστος, από ότι μικρότερα προβλήματα.

## 3.4. Παραδείγματα Εφαρμογών σε CUDA

Το μοντέλο προγραμματισμού της CUDA επεκτείνει τη γλώσσα προγραμματισμού C με έναν σχετικά μικρό αριθμό επιπλέον παράλληλων εντολών. Οι προγραμματιστές που είναι εξοικειωμένοι με τον προγραμματισμό στη C μπορούν πολύ εύκολα να γράψουν κώδικα σε CUDA.

Στο σχετικά σύντομο χρονικό διάστημα κατά το οποίο υπάρχει η CUDA, έχει αναπτυχθεί ένας σημαντικός αριθμός παράλληλων εφαρμογών για πραγματικά προβλήματα. Παραδείγματα τέτοιων εφαρμογών βρίσκουμε στα MRIs, στη μοριακή δυναμική, στην προσομοίωση αστροφυσικών προβλημάτων. Εκτελώντας τις εφαρμογές αυτές σε GPUs αρχιτεκτονικής Tesla, έχει γίνει κατορθωτό να έχουμε σημαντικές επιταχύνσεις σε σχέση με εναλλακτικές εφαρμογές που εκτελούνται σειριακά σε CPUs. Η αναδόμηση MRI ήταν μέχρι και 263 φορές γρηγορότερη, κώδικας μοριακής δυναμικής 10 έως 100 φορές πιο γρήγορος, η n-βαθμού προσομοίωση 50 με 250 φορές ταχύτερη. Αυτές οι μεγάλες επιταχύνσεις είναι αποτέλεσμα του υψηλού βαθμού παραλληλοποίησης και του μεγάλου εύρους μνήμης που μπορούμε να πετύχουμε με την αρχιτεκτονική Tesla.[9]

# 4. Επεξεργασία Εικόνων με χρήση GPUs

## 4.1. Αλγόριθμοι Υλοποίησης Συσχέτισης Πινάκων

### 4.1.1. "Frame – Centric"

Σε αυτή τη μέθοδο θεωρούμε ότι ο πίνακας Frame παραμένει «σταθερός» και πάνω σε αυτόν γίνονται όλες οι δυνατές και απαραίτητες μετακινήσεις του Template. Αυτή η διαδικασία απαιτεί τη χρήση του Pad Frame όπως εξηγήθηκε στην ενότητα *1.1.1.1.* Δηλαδή πολλαπλασιάζονται με κάθε στοιχείο του Pad Frame όλα τα στοιχεία του Template και υπολογίζεται το άθροισμά τους που αποτελεί το εκάστοτε στοιχείο εξόδου. Σε μορφή αλγόριθμου αυτό φαίνεται στον παρακάτω κώδικα Matlab. Αυτή τη μέθοδο αναπτύσσουμε στον παράλληλο αλγόριθμο που έχουμε υλοποιήσει στην CUDA για την εκτέλεση σε GPU.

```
function M=Frame_Centric_Correlation(F, T)

[mF, nF] = size(F);
[mT,nT] = size(T);

P = padarray(F, [mT-1, nT-1], 'both');
M = zeros(mF+mT-1, nF+nT-1);

  for i = 1:mF+mT-1
     ii = i-1+(1:mT);
     for j = 1: nF+nT-1
        jj = j-1+(1:nT);
        M(i,j) = sum(sum(P(ii,jj).*T));
     end
   end

return
```

*Πίνακας* 1

### 4.1.2. "Template – Centric"

Στη μέθοδο αυτή θεωρούμε ο πίνακας Template παραμένει σταθερός και πάνω σε αυτόν γίνονται οι απαραίτητες μετακινήσεις του Frame. Συγκεκριμένα κάθε στοιχείο του Frame πολλαπλασιάζεται με ένα στοιχείο του Template. Τα γινόμενα αυτά καταχωρούνται στις εκάστοτε θέσεις της εξόδου. Στη συνέχεια σε κάθε μία από αυτές προστίθεται το αποτέλεσμα της αντίστοιχης πράξης για το επόμενο στοιχείο του Template κ.ο.κ. Σε μορφή αλγόριθμου αυτό φαίνεται στον παρακάτω κώδικα Matlab.

```
function M=Template_Centric_Correlation(F,T)

[mF, nF] = size(F);
[mT,nT] = size(T);

M = zeros(mF+mT-1, nF+nT-1);

    for j = 1:nT
       for i = 1:mT
          ii = i-1+(1:mF);
          jj = j-1+(1:nF);
          M(ii,jj) =  M(ii,jj) + F.*T(i,j);
       end
  end

return
```

*Πίνακας 2*

### 4.1.3. Local Correlation

Σε αυτή τη μέθοδο εφαρμόζουμε πάνω στην πρώτη έκδοση από τις παραπάνω την εύρεση των τοπικών συντελεστών συσχέτισης. Ο μαθηματικός τύπος υπολογισμού του Local Correlation έχει παρουσιαστεί στο πρώτο κεφάλαιο.

$$C(\mathrm{F},\mathrm{T}) = \frac{\sum\sum(F(i,j)-\mu_{\mathrm{F}})(T(i,j)-\mu_{\mathrm{T}})}{\sqrt{\sum\sum(\mathrm{F(i,j)}-\mu_{\mathrm{F}})^2(T(i,j)-\mu_{\mathrm{T}})^2}} \qquad \textit{Σχέση 9}$$

αλλά προκειμένου να μπορέσουμε να τον εφαρμόσουμε με μεγαλύτερη αποδοτικότητα σε επίπεδο αλγορίθμου είναι να αναγκαίο να προχωρήσουμε σε συγκεκριμένες βελτιστοποιήσεις. Βλέπει κανείς ότι με βάση την σχέση 9 απαιτείται κατ' αρχήν ο υπολογισμός του μέσου όρου των σημείων τόσο του Frame όσο και του Template, και έπειτα σε δεύτερο βήμα η αφαίρεση του από κάθε στοιχείο και ο element-wise πολλαπλασιασμός τους σε κάθε επανάληψη. Για το λόγο αυτό κανονικοποιούμε κατ' αρχήν τον Template ανεξάρτητα από τον Frame καθ' ότι πρόκειται για μία «φτηνή» διαδικασία δεδομένου ότι το μέγεθος του Template είναι μικρό. Έτσι ο παραπάνω τύπος μορφοποιείται στον εξής:

$$C(F,T) = \frac{\sum\sum(F(i,j)-\mu_{\mathrm{F}})}{\sqrt{\sum\sum(F(i,j)-\mu_{\mathrm{F}})^2}}\hat{T} \qquad \textit{Σχέση 10}$$

Αναλύοντας τους όρους έχουμε:

$$C(\mathrm{F},\mathrm{T}) = \frac{\sum\sum\mathrm{F(i,j)}\hat{\mathrm{T}} - ?_{\mathrm{F}}\sum\sum\hat{T}}{\sqrt{\sum\sum F^2(\mathrm{i,j}) - 2\mu_{\mathrm{F}}\sum\sum\mathrm{F(i,j)} + \sum\sum\mu_{\mathrm{F}}^2}} \qquad \textit{Σχέση 11}$$

$$C(F,T) = \frac{\sum\sum F(i,j)\hat{T}}{\sqrt{\sum\sum F^2(i,j) - \frac{1}{n}(\sum\sum F(i,j))^2}} \qquad \textit{Σχέση 12}$$

Όπου από τον τελευταίο τύπο παρατηρούμε ότι είναι δυνατόν ο υπολογισμός της κανονικοποιημένης συσχέτισης να γίνει με ενιαίο τρόπο σε έναν βρόχο επανάληψης που υπολογίζει τα τρία αθροίσματα. Σε μορφή αλγορίθμου αυτό φαίνεται στον παρακάτω κώδικα matlab (οι πίνακες έχουν μετασχηματιστεί σε μονοδιάστατους). {11}, {16}

```
function M=Lcc(F,T)

[mF, nF] = size(F);
[mT,nT] = size(T);


Tmean = mean(T(:));
Tsh = T – Tmean;
nrm = sqrt(sum(Tsh(:).^2);

T = T / nrm;

P=padarray(F, [mT-1, nT-1], 'both');
M=zeros(mF+mT-1, nF+nT-1);
mm = zeros((mF+mT-1)*(nF+nT-1), 1);
ff = P(:);
tt = T(:);

   for i=1:mF+mT-1
      for j=1: nF+nT-1
         pf_corr=0;
         pf_sum=0;
         pf_sqr=0;

         for k = 1 : mT
            for l = 1:nT
               pf_elem= ff(i-1+k +(j-1+l-1) * ( mF+2*(mT-1)));
               pf_corr= pf_corr + pf_elem * tt(k+(l-1)*mT);
               pf_sum= pf_sum + pf_elem;
               pf_sqr= pf_sqr + pf_elem*pf_elem;
            end
         end

         mm(i+(j-1)*(mF+mT-1)) = pf_corr / sqrt(pf_sqr - pf_sum*pf_sum/(mT*nT));
      end
   end

for i=1:mF+mT-1
   for j=1:nF+nT-1
      M(i,j)=mm(i + (j-1)*(mF+mT-1));
   end
end
```

*Πίνακας 3*

## 4.2. Υλοποίηση των Αλγορίθμων σε Γλώσσα Προγραμματισμού CUDA

Οι κώδικες, των οποίων ορισμένα κρίσιμα σημεία θα αναλυθούν παρακάτω, υπάρχουν στο σύνολό τους στο παράρτημα. Το προγράμματα που έχουμε γράψει διαρθρώνονται σε τέσσερα επίπεδα. Ένα αρχείο που περιέχει τη συνάρτηση main, ένα άλλο που περιέχει τον kernel (τη συνάρτηση δηλαδή που εκτελείται στη gpu), ένα τρίτο που περιέχει τις cpu συναρτήσεις και τέλος ένα αρχείο βιβλιοθήκης που περιέχει τις σταθερές των μεγεθών του εκάστοτε προβλήματος.

Η βασική ροή εκτέλεσης του κώδικα σε κάθε μία από τις παρακάτω εκδοχές είναι η εξής:

1. Προσδιορίζονται τα δεδομένα εισόδου και τα παραγόμενα εξ αυτών μεγέθη.
2. Γίνονται οι απαραίτητες δεσμεύσεις μνήμης στη CPU και στη GPU.
3. Αρχικοποιούνται τα δεδομένα.
4. Γίνεται η μεταφορά των πινάκων στη GPU.
5. Εκτελείται η συνάρτηση kernel που υπολογίζει τη συσχέτιση δύο πινάκων.
6. Μεταφέρεται το αποτέλεσμα πίσω στη CPU.
7. Εκτελείται η αντίστοιχη πράξη στη CPU.
8. Γίνεται σύγκριση των αποτελεσμάτων και εκτιμάται η επιτυχία και η απόδοση των υπολογισμών.
9. Τέλος γίνεται η αποδέσμευση των χώρων στη μνήμη που είχαν δεσμευτεί.

### 4.2.1. Correlation

Αρχικά θα αναφερθούμε στην περίπτωση του χωρίς LCC κώδικα ο οποίος περιέχει στοιχεία κώδικα-προγραμματισμού που είναι κοινά και στις άλλες δύο εκδοχές.

**Στη συνάρτηση main:**

Με την εντολή cudaSetDevice(0) ορίζουμε την συσκευή η οποία θα χρησιμοποιηθεί για την εκτέλεση του κώδικα kernel δηλαδή του κώδικα που αφορά στη GPU. Στην προκειμένη περίπτωση χρησιμοποιούμε την device 0.

Με τις εντολές τύπου

*CUDA_SAFE_CALL( cudaMalloc( (void **)&d_Outcome ,OUTCOME_SIZE) );*

δεσμεύουμε χώρο στη μνήμη επιπέδου global της GPU, κατ' αντιστοιχία με την εντολή malloc της γλώσσας C. Η δεύτερη παράμετρος αποτελεί τον δείκτη στη θέση

μνήμης και η τρίτη το πλήθος των bytes συνεχόμενης μνήμης που θέλουμε να δεσμεύσουμε. Η μνήμη που δεσμεύεται με αυτόν τον τρόπο, αποδεσμεύεται με εντολές του τύπου CUDA_SAFE_CALL( cudaFree(d_Outcome) ) που βρίσκονται στο τέλος του προγράμματος της main.

Με τις εντολές

*CUDA_SAFE_CALL(cudaMemcpyToSymbol(d_Template,h_Template, TEMPLATE_SIZE));*

*CUDA_SAFE_CALL(cudaMemcpy(d_Frame,h_Frame,FRAME_SIZE, cudaMemcpyHostToDevice) );*

μεταφέρουμε τα δεδομένα από την μνήμη της CPU (host) στη μνήμη της GPU (device). Η πρώτη παράμετρος δίνει την θέση μνήμης της GPU, η δεύτερη τη θέση μνήμης της CPU, η τρίτη το μέγεθος σε πλήθος bytes των δεδομένων που θα μετακινηθούν, και η τέταρτη τη φορά μετακίνησης, που στην προκειμένη περίπτωση είναι από τον host (CPU) στο device (GPU). Η εντολή cudaMemcpyToSymbol έχει ορισμένα επιπλέον χαρακτηριστικά που θα εξηγηθούν παρακάτω.

Για την αντίστροφη πορεία, για την αντιγραφή δηλαδή δεδομένων πίσω στη μνήμη της CPU, σε αντιστοιχία με τα παραπάνω, χρησιμοποιούμε τον ακόλουθο τύπο εντολών:

*CUDA_SAFE_CALL(cudaMemcpy(h_OutcomeGPU,d_Outcome,OUTCOME_SIZE, cudaMemcpyDeviceToHost) );*

Με τις εντολές

*dim3 dimBlock (BLOCKDIM_X, BLOCKDIM_Y);*

*dim3 dimGrid (GRID_X, GRID_Y);*

προοδιορίζουμε το πλήθος των threads σε κάθε block και το πλήθος των blocks σε κάθε πλέγμα, για το εκάστοτε πρόβλημα, με τέτοιο τρόπο ώστε (χωρίς αυτό γενικά να είναι απαραίτητο) το μέγεθος του πλέγματος να είναι ίσο με το μέγεθος της εξόδου του προβλήματος.

Με το σετ εντολών

*CUT_SAFE_CALL( cutResetTimer(hTimer) );*

*CUT_SAFE_CALL( cutStartTimer(hTimer) );*

*…*

*CUT_SAFE_CALL(cutStopTimer(hTimer));*

*GPU_time = cutGetTimerValue(hTimer);*

Χρησιμοποιούμε το ρολόι της gpu προκειμένου να χρονομετρήσουμε τη διάρκεια εκτέλεσης του kernel.

Με την εντολή

*CUDA_SAFE_CALL( cudaThreadSynchronize() );*

θέτουμε ένα φράγμα στην ροή εκτέλεσης του κώδικα που προϋποθέτει την ολοκλήρωση όλων των διεργασιών που εκτελούνται στη συσκευή για να συνεχίσει το πρόγραμμα.

**Στη συνάρτηση kernel:**

Με την εντολή

*__device__ __constant__ float d_Template[TEMPLATE_M*TEMPLATE_N];*

ορίζουμε τον πίνακα d_Template που υπό τον προσδιορισμό __device__ τοποθετείται στην συσκευή. Ενώ με τον επιπλέον προσδιορισμό __constant__ η μεταβλητή αυτή τοποθετείται στην κοινή μνήμη του πλέγματος και έχει διάρκεια ζωής ίδια με αυτήν της εφαρμογής.

Με την εντολή

*__shared__ float s_Pad_Frame[S_PAD_M * S_PAD_N];*

ορίζουμε τον πίνακα s_Pad_Frame (όπου αποθηκεύεται κάθε φορά κάποιο τμήμα του Pad_Frame) ο οποίος τοποθετείται στη μνήμη επιπέδου shared η οποία αντιστοιχεί σε κάθε block και είναι προσβάσιμη μόνο από τα threads του εκάστοτε block και έχει διάρκεια ζωής ίση με αυτή του block. Μόνο μετά απ' την εκτέλεση __syncthreads() μπορούν οι εγγραφές που έχουν γίνει στις μνήμες shared να είναι ορατές απ' όλα τα threads.

Με τις εντολές

*const int i = blockDim.y * blockIdx.y + threadIdx.y;*

*const int j = blockDim.x * blockIdx.x + threadIdx.x;*

*const int k0 = blockIdx.y * blockDim.y;*

*const int l0 = blockIdx.x * blockDim.x;*

προσδιορίζουμε δείκτες τέτοιους ώστε να αντιστοιχίζονται στην ταυτότητα του κάθε νήματος σε επίπεδο block (threadIdx) και πλέγματος (blockIdx). Με αυτόν τον τρόπο έχουμε την δυνατότητα να προσδιορίσουμε στον υπόλοιπο κώδικα τις λειτουργίες ενός μόνο νήματος οι οποίες όμως θα εκτελεστούν ταυτόχρονα και παράλληλα από όλα τα νήματα σε διαφορετικά πεδία δεδομένων με βάση την σχετική μεταξύ του θέση-ταυτότητα.

### 4.2.2. Με Local Correlation

Στην εκδοχή αυτή δεν χρησιμοποιούνται νέα προγραμματιστικά στοιχεία αλλά υπάρχουν αλγοριθμικές μεταβολές σε σχέση με την προηγούμενη. Ειδικότερα αυτές έγκεινται στα εξής τρία σημεία:

1. Στην συνάρτηση main κατά την αρχικοποίηση των δεδομένων ο πίνακας Template κανονικοποιείται στη λογική που αποτυπώνεται στη σχέση 10.
2. Στην συνάρτηση kernel κάθε τμήμα του Pad_Frame, που μεταφέρεται στη μνήμη shared όπως αναφέρθηκε παραπάνω, κανονικοποιείται στον ίδιο βρόχο επανάληψης που υπολογίζεται και η συσχέτισή του με τον ήδη κανονικοποιημένο Template στη λογική που αποτυπώνεται στη σχέση 12.
3. Εφαρμόζοντας στα δεδομένα του προβλήματος το LCC παίρνουμε έναν κανονικοποιημένο πίνακα εξόδου και έχουμε έτσι τη δυνατότητα άμεσου εντοπισμού της περιοχής ταύτισης-μέγιστης ομοιότητας μεταξύ των συνελισσόμενων πινάκων. Το κέντρο της περιοχής αυτής αντιστοιχεί στον πίνακα εξόδου στο σημείο με τιμή 1.

#### 4.2.2.1. Εκτέλεση σε Πολλαπλές GPUs

Οι μόνες διαφορές σε αυτή την εκδοχή σε σχέση με την αμέσως προηγούμενη εντοπίζονται στην συνάρτηση main. Η βασική φιλοσοφία έγκειται στο να υποδιαιρέσουμε ένα πρόβλημα σε μικρότερα και να εκτελέσουμε παράλληλα καθένα από αυτά σε διαφορετικές συσκευές. Για το λόγο αυτό στόχος μας είναι να διαμοιράσουμε τον Frame σε περισσότερες από μία devices, σε κάθε μία εξ' αυτών να πραγματοποιηθεί συσχέτιση του αντίστοιχου κομματιού με τον Template που είναι κοινός σε όλες τις devices και να ανασυντεθούν τα επιμέρους αποτελέσματα σε ένα πίνακα, ο οποίος θα είναι ίδιος με αυτόν της προηγούμενης εκδοχής. Για να

υλοποιήσουμε τα παραπάνω σε επίπεδο κώδικα κάνουμε χρήση μίας δομής (structure) και εκκινούμε ισάριθμα με το πλήθος των συσκευών νήματα. Για την βέλτιστη και απρόσκοπτη λειτουργία του κώδικα θα πρέπει οι συσκευές στις οποίες θα διαμοιραστεί το πρόβλημα να έχουν ακριβώς τα ίδια χαρακτηριστικά.

Οφείλουμε στο σημείο αυτό να ξεκαθαρίσουμε ότι η συγκεκριμένη υλοποίηση παρουσιάζεται ενδεικτικά προκειμένου να καταδείξει τη δυνατότητα ακόμα υψηλότερων αποδόσεων του αλγορίθμου LCC σε πολλαπλές GPUs. Η χρήση του παρακάτω κώδικα επειβεβαιώνει τη δυνατότητα λήψης σωστών αποτελεσμάτων, εμφανίζει όμως κάποιες πολύ συγκεκριμένες αδυναμίες που έχουν να κάνουν με την ταυτόχρονη και παράλληλη εκτέλεση του στις επιμέρους συσκευές. Συγκεκριμένα, τα νήματα που «ξυπνούν» τις κάρτες εμφανίζουν υψηλές (της τάξης των msec) χρονικές καθυστερήσεις μεταξύ τους, με αποτέλεσμα να καθίσταται προβληματική η εξαγωγή ασφαλών συμπερασμάτων για το συνολικό χρόνο επίλυσης του προβλήματος. Φαίνεται όμως από την άλλη πλευρά η δυνατότητα πολλές κάρτες να δουλεύουν ταυτόχρονα πάνω στο ίδιο πρόβλημα (η κάθε μία σε διαφορετικό τμήμα δεδομένων), κάτι που εξασφαλίζει ότι για μεγάλα τουλάχιστον προβλήματα θα ξεπερνιούνται κατά πολύ οι αποδόσεις του αντίστοιχου κώδικα για μία συσκευή.

Αναλύοντας τον κώδικα θέλουμε να σταθούμε στα εξής: Η χρήση της δομής γίνεται για την αποφυγή χρήσης πολλών μεταβλητών και επιπλέον για τη δυνατότητα δυναμικού ορισμού μεταβλητών ανάλογα και με το πλήθος των συσκευών στις οποίες θα διαμοιραστεί το πρόβλημα.

Με την εντολή

```
for(i = 0; i < GPU_N; i++)
    threadID[i] = cutStartThread((CUT_THREADROUTINE)GPU_Thread,
    (void *)(str + i));
cutWaitForThreads(threadID, GPU_N);
```

Εκκινούμε τα νήματα που θα διαμοιράσουν το πρόβλημα (μέγεθος και δεδομένα), θα καλέσουν τον kernel για κάθε υπο-πρόβλημα, και θα ανασυνθέσουν το αποτέλεσμα σε ενιαίο πίνακα.

Με την εντολή

*CUDA_SAFE_CALL(cudaMemcpy(&h_OutcomeGPU[*

*(str->device)*(FRAME_M+TEMPLATE_M-1)*(FRAME_N/(str->GPU_NUMBER)) +*
*(FRAME_M+TEMPLATE_M-1)*(str->k)*(TEMPLATE_N-1)], d_OutcomeGPU,*

*str->OUTCOME_SIZE_PER_DEVICE, cudaMemcpyDeviceToHost) );*

γίνεται η επιστροφή των αποτελεσμάτων του εκάστοτε υπο-προβλήματος και η εγγραφή τους σε έναν ενιαίο πίνακα. Για κάθε νήμα-διαφορετική συσκευή προσδιορίζεται αρχικά ο δείκτης διεύθυνσης του ενιαίου πίνακα από όπου θα εκκινήσει η εγγραφή, και επιπλέον το μέγεθος της εκάστοτε εξόδου.

## 4.3. Συγκριτικοί Πίνακες και Διαγράμματα Αποτελεσμάτων

1ος ΠΙΝΑΚΑΣ ΣΥΓΚΡΙΤΙΚΩΝ ΑΠΟΤΕΛΕΣΜΑΤΩΝ

ΥΛΟΠΟΙΗΣΗ ΑΛΓΟΡΙΘΜΟΥ ΣΥΣΧΕΤΙΣΗΣ ΧΩΡΙΣ LCC

| TEMPLATE SIZE | | | FRAME SIZE | | | OUTCOME SIZE | | | GPU values | | | | CPU values | | COMPARING values | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | DATA IN time(msec) | DATA OUT time(msec) | KERNEL time(msec) | GFLOPS | time(msec) | GFLOPS | RELATIVE error | MAX ABS error | ACCELARATION |
| **5** | x | 5 | 106 | x | 106 | 110 | x | 110 | 0,080 | 0,084 | 0,036 | 16,80 | 0,41 | 1,45 | 0 | 0 | 11,55 |
| **5** | x | 5 | 616 | x | 616 | 620 | x | 620 | 0,906 | 2,066 | 0,482 | 39,87 | 12,29 | 1,56 | 0 | 0 | 25,49 |
| **5** | x | 5 | 2056 | x | 2056 | 2060 | x | 2060 | 15,969 | 20,413 | 6,234 | 34,03 | 147,63 | 1,44 | 0 | 0 | 23,68 |
| **5** | x | 5 | 5011 | x | 5011 | 5015 | x | 5015 | 92,036 | 118,256 | 71,711 | 17,54 | 956,68 | 1,31 | 0 | 0 | 13,34 |
| **5** | x | 5 | 11571 | x | 11571 | 11575 | x | 11575 | 489,577 | 626,218 | 788,367 | 8,49 | 11.495,77 | 0,58 | 0 | 0 | 14,58 |
| **16** | x | 16 | 113 | x | 113 | 128 | x | 128 | 0,114 | 0,095 | 0,154 | 54,47 | 5,21 | 1,61 | 0 | 0 | 33,84 |
| **16** | x | 16 | 609 | x | 609 | 624 | x | 624 | 0,928 | 2,069 | 2,381 | 83,74 | 124,91 | 1,59 | 0 | 0 | 52,46 |
| **16** | x | 16 | 2049 | x | 2049 | 2064 | x | 2064 | 16,022 | 20,740 | 26,028 | 83,80 | 1.393,19 | 1,56 | 0 | 0 | 53,52 |
| **16** | x | 16 | 5009 | x | 5009 | 5024 | x | 5024 | 92,769 | 118,582 | 172,450 | 74,94 | 8.301,48 | 1,55 | 0 | 0 | 48,14 |
| **16** | x | 16 | 11553 | x | 11553 | 11568 | x | 11568 | 489,034 | 624,119 | 1.032,568 | 66,35 | 61.629,35 | 1,11 | 0 | 0 | 59,68 |
| **21** | x | 21 | 106 | x | 106 | 126 | x | 126 | 0,108 | 0,106 | 0,186 | 75,28 | 9,85 | 1,42 | 0 | 0 | 52,99 |
| **21** | x | 21 | 610 | x | 610 | 630 | x | 630 | 0,933 | 2,126 | 2,677 | 130,77 | 254,10 | 1,38 | 0 | 0 | 94,92 |
| **21** | x | 21 | 2080 | x | 2080 | 2100 | x | 2100 | 16,489 | 21,179 | 32,608 | 119,28 | 2.827,05 | 1,37 | 0 | 0 | 86,69 |
| **21** | x | 21 | 5020 | x | 5020 | 5040 | x | 5040 | 92,366 | 117,960 | 199,630 | 112,22 | 16.160,09 | 1,38 | 0 | 0 | 80,95 |
| **21** | x | 21 | 11572 | x | 11572 | 11592 | x | 11592 | 486,032 | 620,584 | 1.224,660 | 96,77 | 88.609,73 | 1,33 | 0 | 0 | 72,35 |

2ος ΠΙΝΑΚΑΣ ΣΥΓΚΡΤΙΚΩΝ ΑΠΟΤΕΛΕΣΜΑΤΩΝ

ΥΛΟΠΟΙΗΣΗ ΑΛΓΟΡΙΘΜΟΥ ΣΥΣΧΕΤΙΣΗΣ ΧΩΡΙΣ LCC

| TEMPLATE SIZE | | | FRAME SIZE | | | OUTCOME SIZE | | | GPU values | | | | CPU values | | COMPARING values | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | DATA IN time(msec) | DATA OUT time(msec) | KERNEL time(msec) | GFLOPS | time(msec) | GFLOPS | RELATIVE error | MAX ABS error | ACCELAR. |
| **4** | x | 4 | 125 | x | 125 | 128 | x | 128 | 0,118 | 0,097 | 0,040 | 13,11 | 0,33 | 1,56 | 0 | 0 | 8,37 |
| **8** | x | 8 | 121 | x | 121 | 128 | x | 128 | 0,119 | 0,097 | 0,050 | 41,94 | 1,25 | 1,67 | 0 | 0 | 25,12 |
| **16** | x | 16 | 113 | x | 113 | 128 | x | 128 | 0,116 | 0,092 | 0,154 | 54,47 | 5,18 | 1,62 | 0 | 0 | 33,65 |
| **4** | x | 4 | 253 | x | 253 | 256 | x | 256 | 0,232 | 0,347 | 0,110 | 19,07 | 1,29 | 1,62 | 0 | 0 | 11,79 |
| **8** | x | 8 | 249 | x | 249 | 256 | x | 256 | 0,207 | 0,343 | 0,147 | 57,06 | 5,23 | 1,60 | 0 | 0 | 35,57 |
| **16** | x | 16 | 241 | x | 241 | 256 | x | 256 | 0,205 | 0,358 | 0,483 | 69,47 | 20,90 | 1,61 | 0 | 0 | 43,27 |
| **4** | x | 4 | 509 | x | 509 | 512 | x | 512 | 1,088 | 1,422 | 0,446 | 18,81 | 5,69 | 1,47 | 0 | 0 | 12,78 |
| **8** | x | 8 | 505 | x | 505 | 512 | x | 512 | 0,731 | 1,410 | 0,707 | 47,46 | 21,42 | 1,56 | 0 | 0 | 30,29 |
| **16** | x | 16 | 497 | x | 497 | 512 | x | 512 | 0,715 | 1,414 | 2,181 | 61,54 | 84,32 | 1,59 | 0 | 0 | 38,66 |
| **4** | x | 4 | 1021 | x | 1021 | 1024 | x | 1024 | 3,912 | 5,408 | 1,781 | 18,84 | 26,65 | 1,26 | 0 | 0 | 14,96 |
| **8** | x | 8 | 1017 | x | 1017 | 1024 | x | 1024 | 4,189 | 5,371 | 3,087 | 43,48 | 90,20 | 1,49 | 0 | 0 | 29,22 |
| **16** | x | 16 | 1009 | x | 1009 | 1024 | x | 1024 | 3,511 | 5,382 | 8,958 | 59,93 | 349,25 | 1,54 | 0 | 0 | 38,99 |
| **4** | x | 4 | 2045 | x | 2045 | 2048 | x | 2048 | 15,482 | 20,176 | 7,902 | 16,98 | 465,22 | 0,28 | 0 | 0 | 58,87 |
| **8** | x | 8 | 2041 | x | 2041 | 2048 | x | 2048 | 15,227 | 20,155 | 13,140 | 40,86 | 545,19 | 0,98 | 0 | 0 | 41,49 |
| **16** | x | 16 | 2033 | x | 2033 | 2048 | x | 2048 | 15,723 | 20,049 | 36,110 | 59,47 | 1.507,92 | 1,42 | 0 | 0 | 41,76 |
| **4** | x | 4 | 4093 | x | 4093 | 4096 | x | 4096 | 61,429 | 79,222 | 61,144 | 8,78 | 2.196,87 | 0,24 | 0 | 0 | 35,93 |
| **8** | x | 8 | 4089 | x | 4089 | 4096 | x | 4096 | 61,243 | 78,373 | 59,087 | 36,34 | 2.406,79 | 0,89 | 0 | 0 | 40,73 |
| **16** | x | 16 | 4081 | x | 4081 | 4096 | x | 4096 | 61,123 | 78,316 | 148,324 | 57,91 | 6.169,16 | 1,39 | 0 | 0 | 41,59 |
| **4** | x | 4 | 8189 | x | 8189 | 8192 | x | 8192 | 243,401 | 309,844 | 262,585 | 8,17 | 8.803,25 | 0,24 | 0 | 0 | 33,53 |
| **8** | x | 8 | 8185 | x | 8185 | 8192 | x | 8192 | 243,045 | 309,702 | 33,505 | 25,76 | 9.609,54 | 0,89 | 0 | 0 | 28,81 |
| **16** | x | 16 | 8177 | x | 8177 | 8192 | x | 8192 | 243,453 | 309,226 | 726,818 | 47,27 | 24.892,05 | 1,38 | 0 | 0 | 34,25 |
| **8** | x | 8 | 16377 | x | 16377 | 16384 | x | 16384 | 968,838 | 1.237,197 | 2.439,489 | 14,08 | 26.790,11 | 0,49 | 0 | 0 | 28,61 |
| **16** | x | 16 | 16369 | x | 16369 | 16384 | x | 16384 | 966,992 | 1.236,239 | 4.369,268 | 31,46 | 190.301,34 | 0,72 | 0 | 0 | 43,55 |

3ος ΠΙΝΑΚΑΣ ΣΥΓΚΡΙΤΙΚΩΝ ΑΠΟΤΕΛΕΣΜΑΤΩΝ

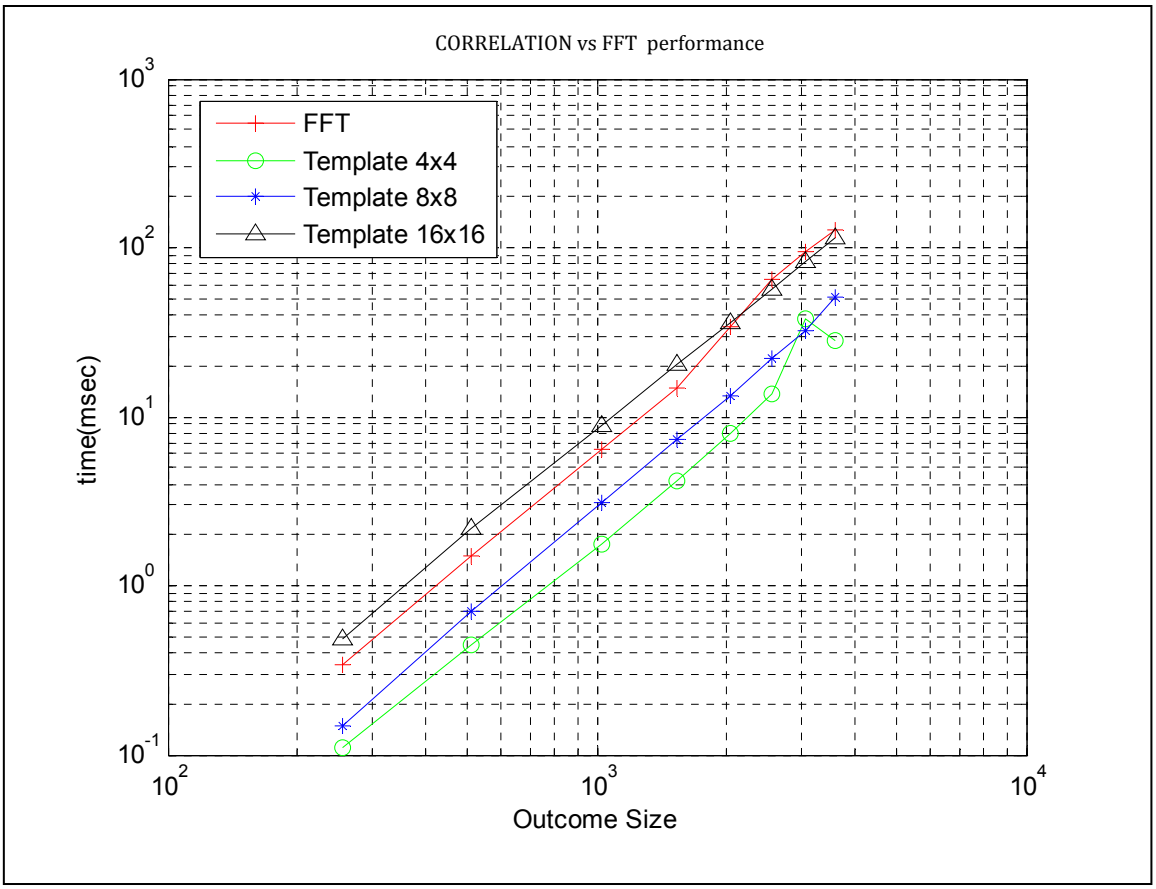
ΥΛΟΠΟΙΗΣΗ ΑΛΓΟΡΙΘΜΟΥ ΣΥΣΧΕΤΙΣΗΣ ΜΕ LCC

| TEMPLATE SIZE | | | FRAME SIZE | | | OUTCOME SIZE | | | GPU values | | | | CPU values | | COMPARING values | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | DATA IN time(msec) | DATA OUT time(msec) | KERNEL time(msec) | GFLOPS | time(msec) | GFLOPS | RELATIVE error | MAX ABS error | ACCELER. |
| **5** | x | 5 | 106 | x | 106 | 110 | x | 110 | 0,091 | 0,075 | 0,042 | 37,45 | 0,59 | 2,66 | 2,14E-07 | 2,98E-07 | 14,07 |
| **5** | x | 5 | 616 | x | 616 | 620 | x | 620 | 0,914 | 2,027 | 0,655 | 76,29 | 18,96 | 2,63 | 2,29E-07 | 4,77E-07 | 28,94 |
| **5** | x | 5 | 2056 | x | 2056 | 2060 | x | 2060 | 15,951 | 20,051 | 6,242 | 88,38 | 217,17 | 2,54 | 2,14E-07 | 5,96E-07 | 34,78 |
| **5** | x | 5 | 5011 | x | 5011 | 5015 | x | 5015 | 92,598 | 117,408 | 76,779 | 42,58 | 1.338,10 | 2,44 | 2,50E-07 | 6,56E-07 | 17,43 |
| **5** | x | 5 | 11571 | x | 11571 | 11575 | x | 11575 | 485,398 | 617,835 | 756,540 | 23,02 | 13.429,87 | 1,29 | 2,38E-07 | 6,55E-07 | 17,75 |
| **16** | x | 16 | 113 | x | 113 | 128 | x | 128 | 0,111 | 0,089 | 0,345 | 61,02 | 7,23 | 2,91 | 2,89E-07 | 1,19E-07 | 20,96 |
| **16** | x | 16 | 609 | x | 609 | 624 | x | 624 | 0,952 | 2,079 | 5,560 | 89,99 | 173,83 | 2,87 | 2,69E-07 | 1,34E-07 | 31,26 |
| **16** | x | 16 | 2049 | x | 2049 | 2064 | x | 2064 | 15,829 | 20,035 | 60,040 | 91,18 | 1.997,42 | 2,74 | 2,79E-07 | 3,58E-07 | 33,27 |
| **16** | x | 16 | 5009 | x | 5009 | 5024 | x | 5024 | 91,809 | 116,410 | 380,308 | 85,28 | 11.618,92 | 2,79 | 2,76E-07 | 2,08E-07 | 30,55 |
| **16** | x | 16 | 11553 | x | 11553 | 11568 | x | 11568 | 484,904 | 615,706 | 2.161,639 | 79,55 | 80.582,59 | 2,13 | 3,02E-07 | 2,08E-07 | 37,28 |
| **21** | x | 21 | 106 | x | 106 | 126 | x | 126 | 0,121 | 0,092 | 0,427 | 82,17 | 13,75 | 2,55 | 2,47E-07 | 2,38E-07 | 32,20 |
| **21** | x | 21 | 610 | x | 610 | 630 | x | 630 | 0,931 | 2,091 | 6,123 | 143,25 | 359,00 | 2,44 | 3,18E-07 | 1,12E-07 | 58,63 |
| **21** | x | 21 | 2080 | x | 2080 | 2100 | x | 2100 | 16,465 | 20,987 | 71,002 | 137,27 | 4.114,16 | 2,37 | 3,29E-07 | 1,64E-07 | 57,94 |
| **21** | x | 21 | 5020 | x | 5020 | 5040 | x | 5040 | 93,407 | 118,274 | 425,088 | 132,06 | 23.671,84 | 2,37 | 3,16E-07 | 1,49E-07 | 55,69 |
| **21** | x | 21 | 11572 | x | 11572 | 11592 | x | 11592 | 490,523 | 624,661 | 2.298,340 | 129,21 | 133.222,33 | 2,23 | 3,23E-07 | 1,64E-07 | 57,96 |

# 4$^{Ος}$ ΠΙΝΑΚΑΣ ΣΥΓΚΡΙΤΙΚΩΝ ΑΠΟΤΕΛΕΣΜΑΤΩΝ

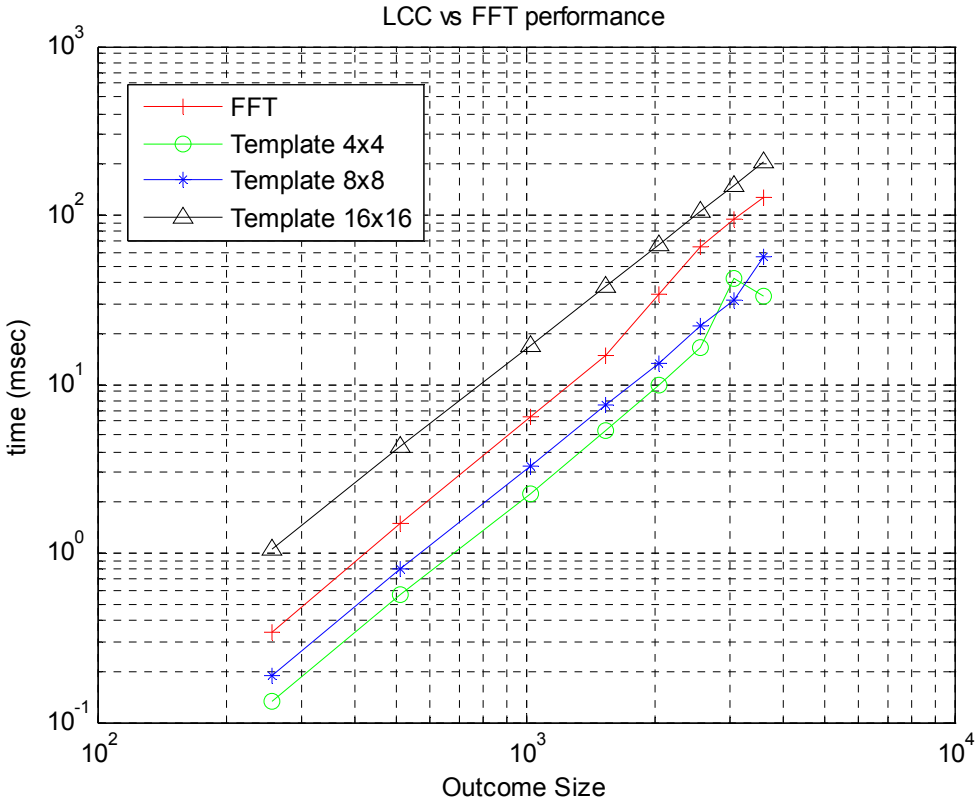
## ΥΛΟΠΟΙΗΣΗ ΑΛΓΟΡΙΘΜΟΥ ΣΥΣΧΕΤΙΣΗΣ ΜΕ LCC

| TEMPLATE SIZE | | | FRAME SIZE | | | OUTCOME SIZE | | | GPU values | | | | CPU values | | COMPARING values | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | DATA IN time(msec) | DATA OUT time(msec) | KERNEL time(msec) | GFLOPS | time(msec) | GFLOPS | RELATIVE error | MAX ABS error | ACCELARATION |
| **4** | x | 4 | 125 | x | 125 | 128 | x | 128 | 0,113 | 0,103 | 0,047 | 29,63 | 0,56 | 2,48 | 1,52E-07 | 2,68E-07 | 11,95 |
| **8** | x | 8 | 121 | x | 121 | 128 | x | 128 | 0,114 | 0,094 | 0,064 | 83,20 | 1,61 | 3,30 | 2,07E-07 | 1,49E-07 | 25,22 |
| **16** | x | 16 | 113 | x | 113 | 128 | x | 128 | 0,115 | 0,096 | 0,344 | 61,20 | 7,24 | 2,91 | 2,69E-07 | 1,19E-07 | 21,03 |
| **4** | x | 4 | 253 | x | 253 | 256 | x | 256 | 0,216 | 0,339 | 0,133 | 41,88 | 2,20 | 2,53 | 1,48E-07 | 2,38E-07 | 16,56 |
| **8** | x | 8 | 249 | x | 249 | 256 | x | 256 | 0,204 | 0,353 | 0,190 | 112,10 | 6,34 | 3,36 | 2,24E-07 | 2,09E-07 | 33,39 |
| **16** | x | 16 | 241 | x | 241 | 256 | x | 256 | 0,213 | 0,348 | 1,060 | 79,45 | 29,03 | 2,90 | 2,66E-07 | 1,19E-07 | 27,38 |
| **4** | x | 4 | 509 | x | 509 | 512 | x | 512 | 0,720 | 1,349 | 0,571 | 39,02 | 8,97 | 2,48 | 1,44E-07 | 3,57E-07 | 15,71 |
| **8** | x | 8 | 505 | x | 505 | 512 | x | 512 | 0,720 | 1,433 | 0,799 | 106,63 | 25,88 | 3,29 | 2,52E-07 | 2,53E-07 | 32,39 |
| **16** | x | 16 | 497 | x | 497 | 512 | x | 512 | 0,704 | 1,426 | 4,243 | 79,39 | 115,97 | 2,90 | 2,68E-07 | 1,64E-07 | 27,33 |
| **4** | x | 4 | 1021 | x | 1021 | 1024 | x | 1024 | 3,375 | 5,549 | 2,259 | 39,45 | 40,68 | 2,19 | 1,59E-07 | 3,57E-07 | 17,98 |
| **8** | x | 8 | 1017 | x | 1017 | 1024 | x | 1024 | 3,382 | 5,532 | 3,305 | 103,11 | 109,87 | 3,10 | 1,97E-07 | 2,38E-07 | 33,24 |
| **16** | x | 16 | 1009 | x | 1009 | 1024 | x | 1024 | 3,281 | 5,484 | 16,698 | 80,68 | 484,84 | 2,78 | 3,22E-07 | 1,56E-07 | 29,03 |
| **4** | x | 4 | 2045 | x | 2045 | 2048 | x | 2048 | 15,802 | 20,445 | 9,842 | 36,22 | 510,12 | 0,69 | 1,74E-07 | 4,17E-07 | 51,83 |
| **8** | x | 8 | 2041 | x | 2041 | 2048 | x | 2048 | 15,635 | 20,309 | 13,410 | 101,65 | 567,11 | 2,40 | 2,35E-07 | 2,68E-07 | 42,29 |
| **16** | x | 16 | 2033 | x | 2033 | 2048 | x | 2048 | 15,726 | 20.507 | 66,999 | 80,44 | 2.128,65 | 2,53 | 3,11E-07 | 1,64E-07 | 31,77 |
| **4** | x | 4 | 4093 | x | 4093 | 4096 | x | 4096 | 61,374 | 78,304 | 69,754 | 20,44 | 2.300,06 | 0,62 | 1,61E-07 | 4,77E-07 | 32,97 |
| **8** | x | 8 | 4089 | x | 4089 | 4096 | x | 4096 | 61,488 | 79,137 | 58,639 | 92,99 | 2.468,51 | 2,21 | 2,14E-07 | 2,98E-07 | 42,09 |
| **16** | x | 16 | 4081 | x | 4081 | 4096 | x | 4096 | 61,126 | 78,447 | 270,730 | 79,63 | 8.498,52 | 2,53 | 3,17E-07 | 2,08E-07 | 31,39 |
| **4** | x | 4 | 8189 | x | 8189 | 8192 | x | 8192 | 245,970 | 313,029 | 297,581 | 19,16 | 9.342,16 | 0,61 | 1,56E-07 | 4,77E-07 | 31,39 |
| **8** | x | 8 | 8185 | x | 8185 | 8192 | x | 8192 | 245,366 | 313,032 | 328,003 | 66,49 | 9.929,23 | 2,19 | 1,92E-07 | 2,98E-07 | 30,27 |
| **16** | x | 16 | 8177 | x | 8177 | 8192 | x | 8192 | 244,193 | 310,945 | 1.282,016 | 67,26 | 34.382,97 | 2,51 | 2,93E-07 | 2,08E-07 | 26,82 |
| **8** | x | 8 | 16377 | x | 16377 | 16384 | x | 16384 | 966,692 | 1.236,567 | 2.343,192 | 37,23 | 56.580,44 | 1,54 | 2,28E-07 | 3,57E-07 | 24,14 |
| **16** | x | 16 | 16369 | x | 16369 | 16384 | x | 16384 | 967,819 | 1.240,703 | 6.548,346 | 52,67 | 229.366,99 | 1,50 | 3,15E-07 | 2,38E-07 | 35,02 |



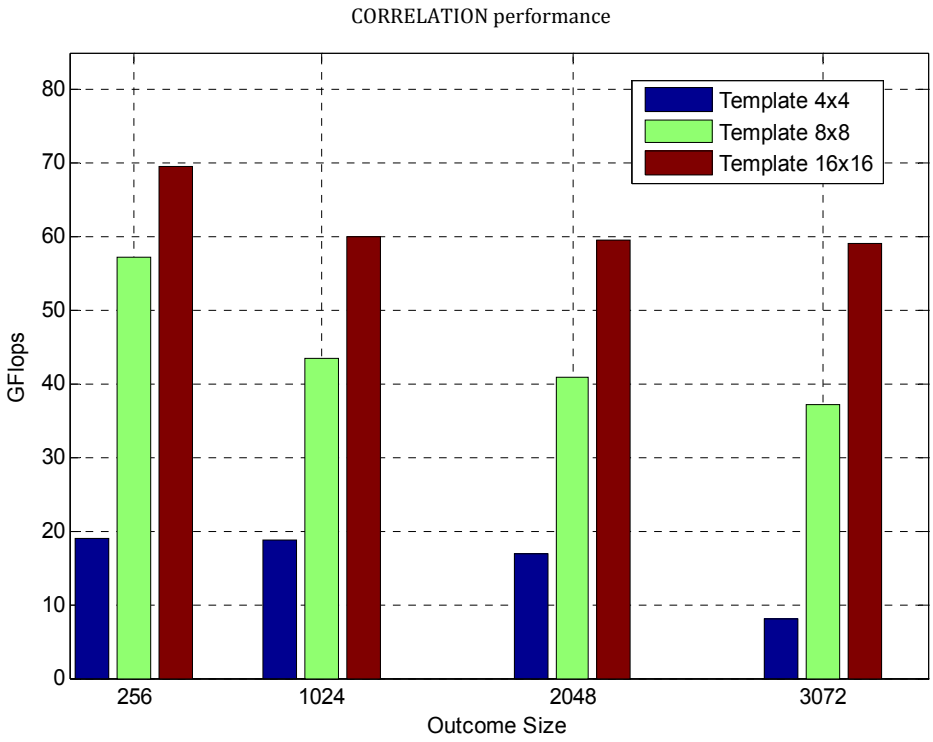

*Διάγραμμα Νο 1*

Στο διάγραμμα Νο 1 παρουσιάζουμε την απόδοση του κώδικα συσχέτισης συκριτικά με τον κώδικα FFT2D από το CUDA SDK sample. Στον οριζόντιο άξονα δίνουμε το μέγεθος του πίνακα εξόδου ως προς τη μία του διάσταση δεδομένου ότι οι συγκεκριμένες μετρήσεις έγιναν για τετραγωνικούς πίνακες. Στον κάθετο άξονα δίνουμε τον χρόνο εκτέλεσης της συνάρτησης kernel, δηλαδή του ενεργού χρόνου εκτέλεσης στη GPU. Για λόγους καλύτερης οπτικής απεικόνισης οι άξονες είναι σε λογαριθμική κλίμακα. Συμπεραίνουμε ότι ο κώδικας correlation αποδίδει καλύτερα για μικρά μοτίβα και σε μεγάλους πίνακες αναζήτησης.



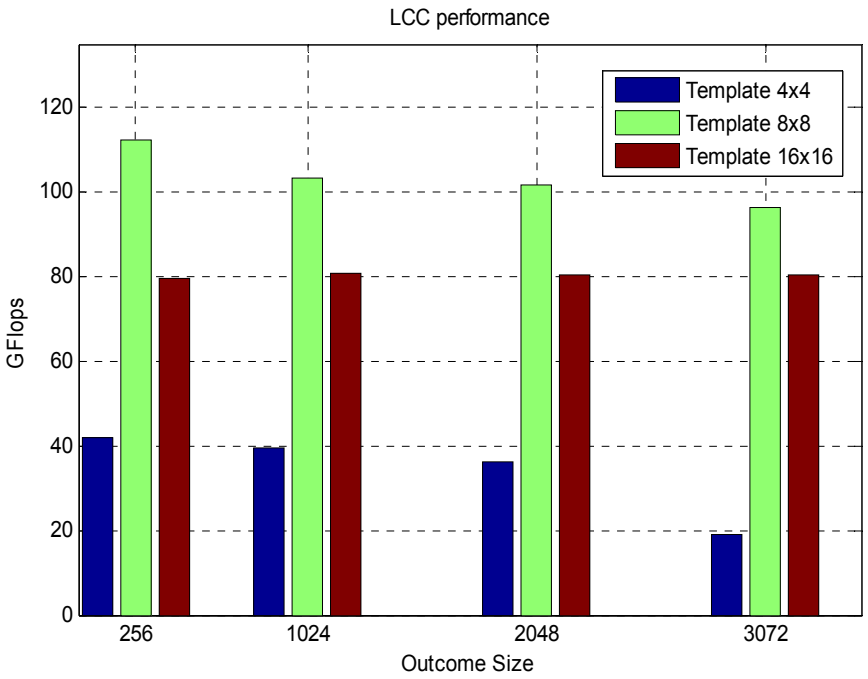

*Διάγραμμα Νο 2*

Στο διάγραμμα Νο 2 παρουσιάζουμε την απόδοση του κώδικα LCC συκριτικά με τον κώδικα FFT2D. Τα βασικά συμπεράσματα από το διάγραμμα αυτό σε ότι αφορά στη σύγκριση του LCC με τον FFT2D παραμένουν ίδια με εκείνα που εξηγήσαμε στο διάγραμμα 1, έχει αξία όμως να παρατηρήσουμε τις μεγαλύτερες χρονικές καθυστερήσεις που οφείλονται στις περισσότερες πράξεις του LCC.



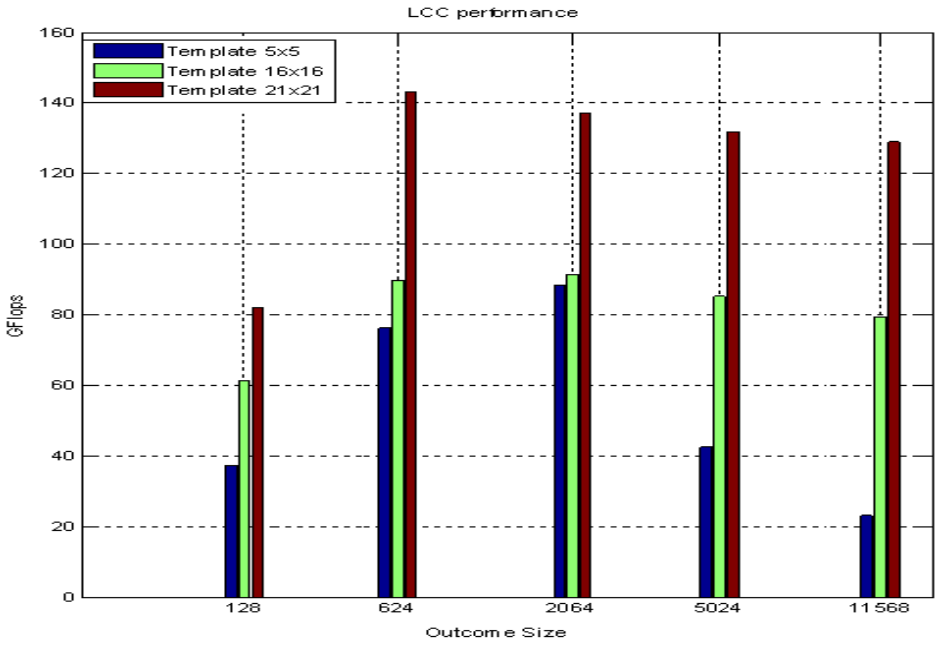

*Διάγραμμα Νο 3*

Στο διάγραμμα Νο 3 παρουσιάζουμε την απόδοση σε GFlops του κώδικα Correlation (χωρίς LCC) για τρία διαφορετικά μεγέθη μοτίβου. Παρατηρούμε ότι η άυξηση του πλήθους των υπολογισμών δεν συνοδεύεται με ανάλογη αύξηση του χρόνου, γι αυτό και τα GFlops μεγαλύτερων Templates είναι περισσότερα.




*Διάγραμμα Νο 4*

Στο διάγραμμα Νο 4 παρουσιάζουμε την απόδοση σε GFlops του κώδικα LCC για τρία διαφορετικά μεγέθη μοτίβου. Η σύνθετη υπολογιστική δομή του LCC αντιστρέφει σε ένα βαθμό τα συμπεράσματα που προέκυψαν από το διάγραμμα 3, χωρίς όμως αυτό να γενικεύεται όπως φαίνεται χαρακτηριστικά και από το διάγραμμα 5.




*Διάγραμμα Νο 5*

## 4.4. Εφαρμογή

Στο κεφάλαιο αυτό παρουσιάζουμε δύο πραγματικά παραδείγματα με εισαγωγή δεδομένων από δύο εικόνες. Μία ασπρόμαυρη εικόνα μετασχηματίζεται σε δυοδιάστατο πίνακα δεδομένων, εισέρχεται ως είσοδος από ένα αρχείο txt στο κώδικα. Στη συνέχεια επιλέγεται με τυχαίο τρόπο ένα τμήμα αυτής το οποίο και αποτελεί το μοτίβο αναζήτησης. Μετά την επεξεργασία των δεδομένων το πρόγραμμα επιστρέφει το σημείο εκείνο στο οποίο εντοπίζεται το κέντρο του μοτίβου πάνω στον πίνακα εξόδου. Το σημείο αυτό σημειώνεται με x, ενώ για βέλτιστη οπτικοποίηση των αποτελεσμάτων η εικόνα εσόδου εμφανίζεται στις διαστάσεις του πίνακα εξόδου.

Πρώτη περίπτωση:

Frame Size: 253x358

Template Size: 21x21

Outcome Size: 273x378

Best match located at (90, 220)

Δεύτερη περίπτωση:

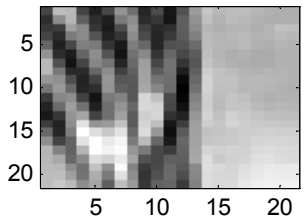
Frame: 400 x 610

Template: 21x21

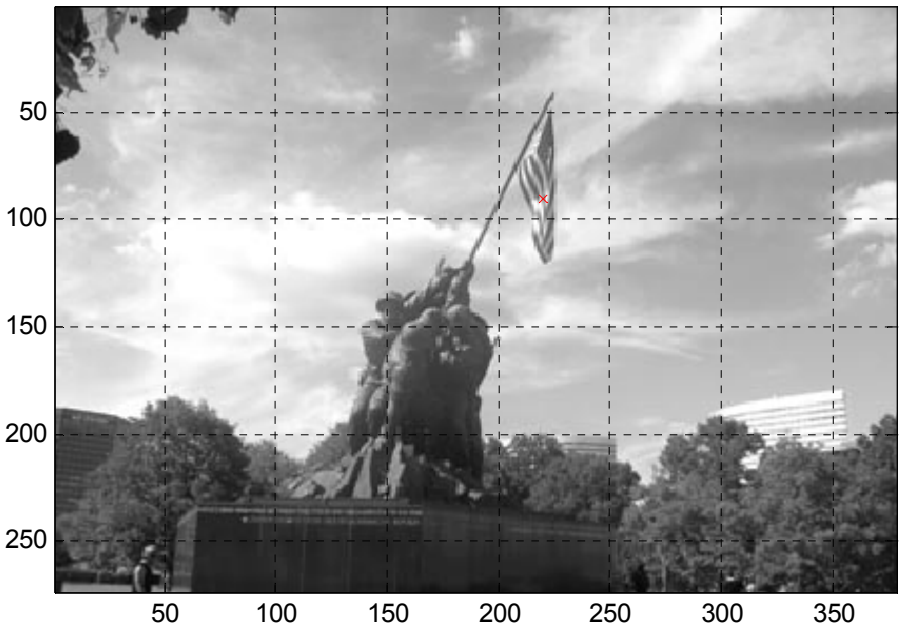
Outcome: 420 x 630

Best match located at (210, 505)

Πρώτη περίπτωση:

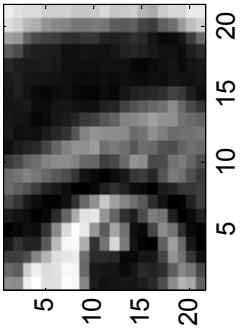



*Σχήμα 9*




*Σχήμα 10*

Δεύτερη περίπτωση:


*Σχήμα 12*

Salvador Dali. *Self Portrait as Mona Lisa*. 1954
Photographic elements by Philippe Halsman
from: *Marcel Duchamp*
[the catalogue of an exhibition at the
Museum of Modern Art and the
Philadelphia Museum of Art] 1973, p. 195.

# 5. ΕΠΙΛΟΓΟΣ

Στη διπλωματική αυτή εργασία ασχοληθήκαμε με το πεδίο της επεξεργασίας εικόνων και ειδικότερα με την αναζήτηση προτύπου. Εφαρμόσαμε το μαθηματικό τύπο της συσχέτισης δισδιάστατων πινάκων μεγέθους αντίστοιχου μιας εικόνας ο ένας και σαφώς μικρότερος ο δεύτερος. Χρησιμοποιήσαμε τον αλγόριθμο κεντραρίσματος των πινάκων στο μέσο όρο τους και κανονικοποίησης τους στη μοναδιαία τυπική απόκλιση ώστε να μπορέσουμε να υπολογίσουμε τους τοπικούς συντελεστές συσχέτισης.

Παρουσιάσαμε την χρηστική αξία της παράλληλης υπολογιστικής για την επίλυση του συγκεκριμένου προβλήματος. Αναδείξαμε βασικά στοιχεία αρχιτεκτονικής των Μονάδων Επεξεργασίας Γραφικών Γενικής Χρήσης καθώς και το προγραμματιστικό μοντέλο της γλώσσας προγραμματισμού CUDA.

Το ουσιαστικότερο τμήμα αυτής της διπλωματικής ήταν η σύνταξη κώδικα σε γλώσσα CUDA που υλοποιεί το Correlation και το Local Correlation και που επιτυγχάνει χρόνους έως και 95 φορές ταχύτερους από τη CPU και που αγγίζει απόδοση έως και 150 GFlops στη GPU με θεωρητικό peak τα 624 GFlops. Ο κώδικας παρέχει ανταγωνιστικά αποτελέσματα με εκείνα του 2DFFT που κατά βάση χρησιμοποιείται στην επεξεργασία εικόνων.

Τέλος, η διπλωματική αυτή εργασία μπορεί να αποτελέσει βάση για περεταίρω μελέτη και έρευνα στο πεδίο επεξεργασίας εικόνων με χρήση GPGPUs, ειδικότερα σε ότι αφορά την αξιοποίηση πολλαπλών GPUs για την επίλυση του ίδιου προβλήματος (ήδη παρέχεται ένα ανεπτυγμένο υπόβαθρο σε αυτή την κατεύθυνση από αυτή την εργασία). Επίσης μπορεί να αξιοποιηθεί για την επεξεργασία 3D εικόνων, την αναζήτηση προτύπου σε εικόνες που έχουν υποστεί παραμόρφωση (μετατόπιση, στροφή, αλλοίωση) κ.α.

# ΠΑΡΑΡΤΗΜΑ

## 5.1. Correlation

```
//*****************************************************************************
//************************************MAIN FILE********************************
//*****************************************************************************

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <cutil.h>
#include <assert.h>
#include <time.h>
#include <sys/time.h>

////////////////////////////////////////////////////////////////////////////////
// Common host and device functions
////////////////////////////////////////////////////////////////////////////////
//Round a / b to nearest higher integer value
int iDivUp(int a, int b){
  return (a % b != 0) ? (a / b + 1) : (a / b);
}

////////////////////////////////////////////////////////////////////////////////
// Including
////////////////////////////////////////////////////////////////////////////////
#include <gpu_functions_Final_America.cu>
#include <ourConstants_Final_America.h>

////////////////////////////////////////////////////////////////////////////////
// CPU functions
////////////////////////////////////////////////////////////////////////////////
extern "C" void Printing_Matrix(
                                        int Rows,
                                        int Columns,
                                        float *Matrix
                                        );

extern "C" void correlationCPU(
                                       float *h_OutcomeCPU,
                                       float *h_Pad_Frame,
                                       float *h_Template
                                       );

extern "C" void GetSubMatrix (float *h_Frame,
                                      float *h_Template,
                                int startRow,
                                      int startCol
                                      );

////////////////////////////////////////////////////////////////////////////////
// Data configuration
////////////////////////////////////////////////////////////////////////////////
const  int FRAME_SIZE    = FRAME_M * FRAME_N * sizeof(float);
const  int TEMPLATE_SIZE = TEMPLATE_M * TEMPLATE_N * sizeof(float);

const   int OUTCOME_SIZE  = (FRAME_M + TEMPLATE_M -1) * (FRAME_N + TEMPLATE_N -1)*
sizeof(float);
```

```
const  int PAD_FRAME_SIZE= PAD_FRAME_M * PAD_FRAME_N * sizeof(float);

const int GRID_Y = iDivUp((FRAME_M + TEMPLATE_M - 1), BLOCKDIM_Y);
const int GRID_X = iDivUp((FRAME_N + TEMPLATE_N - 1), BLOCKDIM_X);
////////////////////////////////////////////////////////////////////////////
// Defining
////////////////////////////////////////////////////////////////////////////
#define WARMUP

////////////////////////////////////////////////////////////////////////////
// Main program
////////////////////////////////////////////////////////////////////////////
int main(int argc, char **argv){

  float
    *h_Template,
    *h_Frame,
    *h_OutcomeCPU,
    *h_OutcomeGPU,
    *h_Pad_Frame;

  float
    *d_Frame,
    *d_Outcome,
    *d_Pad_Frame;

  timeval randome_time, start_in, end_in, start_out, end_out, start_CPU, end_CPU;
  double in_time, out_time, GPU_time, CPU_time, flop, GPU_gflops, CPU_gflops,
    speedup, L1norm, sum_delta, max_abs_err, sum_ref;
  int i, k, f, startRow, startCol;
  unsigned int hTimer;


  CUT_DEVICE_INIT(argc, argv);

  cudaSetDevice(0);


  CUT_SAFE_CALL(cutCreateTimer(&hTimer));

  printf("\nFrame Size is %i x %i\n", FRAME_M, FRAME_N);
  printf("Template size is %i x %i\n", TEMPLATE_M, TEMPLATE_N);
  printf("Outcome size is %i x %i\n\n", (FRAME_M + TEMPLATE_M -1),    (FRAME_N +
TEMPLATE_N -1));
  printf("Initializing data...\n");

  h_Template  = (float *)malloc(TEMPLATE_SIZE);
  h_Frame     = (float *)malloc(FRAME_SIZE);
  h_OutcomeGPU= (float *)malloc(OUTCOME_SIZE);
  h_OutcomeCPU= (float *)malloc(OUTCOME_SIZE);
  h_Pad_Frame = (float *)malloc(PAD_FRAME_SIZE);


  CUDA_SAFE_CALL( cudaMalloc( (void **)&d_Outcome ,OUTCOME_SIZE) );
  CUDA_SAFE_CALL( cudaMalloc( (void **)&d_Frame , FRAME_SIZE) );
  CUDA_SAFE_CALL( cudaMalloc( (void **)&d_Pad_Frame ,PAD_FRAME_SIZE) );

////////////////////////////////////////////////////////////////////////////
//  Initialising Data
////////////////////////////////////////////////////////////////////////////

//Initializing the Frame with random values
  gettimeofday(&randome_time, NULL);
  srand(randome_time.tv_sec);
```

```
  for(i = 0; i < FRAME_M *  FRAME_N; i++){
    h_Frame[i] = (float) (rand() % 1000 + 1);
  }

  if (PRINT_INPUT_MATRICES == TRUE){
    printf("The Frame Matrix is printed here \n");
    Printing_Matrix(FRAME_M, FRAME_N, h_Frame);
  }

//Initializing the Template as part of the Frame
  startRow = rand() % (FRAME_M - TEMPLATE_M);
  startCol = rand() % (FRAME_N - TEMPLATE_N);

  GetSubMatrix(
                h_Frame,
                h_Template,
                startRow,
                startCol
                );

  printf("Template is located startng at the (%d %d) point in the Frame\n\n", startRow, startCol);

  if (PRINT_INPUT_MATRICES == TRUE){
    printf("The Template Matrix is printed here \n");
    Printing_Matrix(TEMPLATE_M, TEMPLATE_N, h_Template);
  }


////PADDING THE FRAME////
  for (i =0; i<PAD_FRAME_M*PAD_FRAME_N; i++)
    h_Pad_Frame[i] = 0;

  for (i=0 ; i<PAD_FRAME_M*PAD_FRAME_N; i++){
    if (i==((TEMPLATE_N+f)*PAD_FRAME_M-(TEMPLATE_M-2)))
      f=f+1;
    else            if          ((i>(TEMPLATE_N-1+f)*PAD_FRAME_M+(TEMPLATE_M-1))          &&
(i<(TEMPLATE_N+f)*PAD_FRAME_M-(TEMPLATE_M-2))){
      h_Pad_Frame[i-1]=h_Frame[k];
      k=k+1;
      if (k>FRAME_M*FRAME_N)
            break;
    }

  }

  if (PRINT_INPUT_MATRICES == TRUE){
    printf("The Padded Matrix is printed here \n");
    Printing_Matrix(PAD_FRAME_M, PAD_FRAME_N, h_Pad_Frame);
  }

////////////////////////////////////////////////////////////////////////////
//  Transporting Data to GPU
////////////////////////////////////////////////////////////////////////////
  gettimeofday(&start_in, NULL);
  CUDA_SAFE_CALL( cudaMemcpyToSymbol(d_Template, h_Template, TEMPLATE_SIZE));
  CUDA_SAFE_CALL( cudaMemcpy(d_Frame, h_Frame, FRAME_SIZE, cudaMemcpyHostToDevice) );
  CUDA_SAFE_CALL(       cudaMemcpy(d_Pad_Frame,       h_Pad_Frame,       PAD_FRAME_SIZE,
cudaMemcpyHostToDevice) );
  gettimeofday(&end_in, NULL);
////////////////////////////////////////////////////////////////////////////
//  Setting the block & grid dimensions
////////////////////////////////////////////////////////////////////////////
  dim3 dimBlock (BLOCKDIM_X, BLOCKDIM_Y);
```

```
  dim3 dimGrid (GRID_X, GRID_Y);
////////////////////////////////////////////////////////////////////////////
//  WARMING UP
////////////////////////////////////////////////////////////////////////////
#ifdef WARMUP

  matrix_correlationGPU<<<dimGrid, dimBlock>>>(
                                                d_Outcome,
                                                d_Pad_Frame
                                                      );

  CUT_CHECK_ERROR("matrix_correlationGPU() execution failed\n");
  CUDA_SAFE_CALL( cudaThreadSynchronize() );
#endif


////////////////////////////////////////////////////////////////////////////
//  GPU FUNCTIONS
////////////////////////////////////////////////////////////////////////////
  CUDA_SAFE_CALL(cudaMemset(d_Outcome, 0, OUTCOME_SIZE));

  printf("\nGPU  matrix_correlation...\n");

  CUDA_SAFE_CALL( cudaThreadSynchronize() );
  CUT_SAFE_CALL( cutResetTimer(hTimer) );
  CUT_SAFE_CALL( cutStartTimer(hTimer) );

  matrix_correlationGPU<<<dimGrid, dimBlock>>>(
                                                d_Outcome,
                                                d_Pad_Frame
                                                );

  CUT_CHECK_ERROR("matrix_correlationGPU() execution failed\n");
  CUDA_SAFE_CALL( cudaThreadSynchronize() );
  CUT_SAFE_CALL(cutStopTimer(hTimer));
  GPU_time = cutGetTimerValue(hTimer);

  gettimeofday(&start_out, NULL);
  CUDA_SAFE_CALL(          cudaMemcpy(h_OutcomeGPU,          d_Outcome,          OUTCOME_SIZE,
cudaMemcpyDeviceToHost) );
  gettimeofday(&end_out, NULL);

  in_time = (double)((end_in.tv_usec - start_in.tv_usec)/1.0e6 + end_in.tv_sec - start_in.tv_sec);
  out_time = (double)((end_out.tv_usec - start_out.tv_usec)/1.0e6 + end_out.tv_sec - start_out.tv_sec);
  flop      =      1e-6      *      (FRAME_M      +      TEMPLATE_M      -      1)*(FRAME_N      +      TEMPLATE_N      -
1)*2*(TEMPLATE_M*TEMPLATE_N);
  GPU_gflops =   flop / GPU_time;

  printf("Transfer Data  TO   GPU time is: %f msec\n", in_time*1000);
  printf("Transfer Data FROM  GPU time is: %f msec\n", out_time*1000);
  printf("GPU matrix_correlation time is : %f msec\n", GPU_time);
  printf("GPU matrix_correlation GFLOPS  : %f \n", GPU_gflops);

////////////////////////////////////////////////////////////////////////////
//  CPU FUNCTIONS
////////////////////////////////////////////////////////////////////////////
  printf("\nCPU matrix correlation..\n");

  gettimeofday(&start_CPU, NULL);
  correlationCPU(
                        h_OutcomeCPU,
                        h_Pad_Frame,
                        h_Template
                        );
```

```
  gettimeofday(&end_CPU, NULL);

  CPU_time = (double)(((end_CPU.tv_usec - start_CPU.tv_usec)/1.0e6 + end_CPU.tv_sec - start_CPU.tv_sec)*1000);
  CPU_gflops =  flop / CPU_time;

  printf("CPU matrix correlation time is: %f msec\n", CPU_time);
  printf("CPU matrix_correlation GFLOPS : %f \n", CPU_gflops);

//Printing Output Data
  if (PRINT_OUTCOME_MATRICES == TRUE){
    printf("The GPU Correlation Matrix is printed here: \n");
    Printing_Matrix((FRAME_M + TEMPLATE_M -1),(FRAME_N + TEMPLATE_N -1), h_OutcomeGPU);

    printf("The CPU correlation Matrix is printed here: \n");
    Printing_Matrix((FRAME_M + TEMPLATE_M -1),(FRAME_N + TEMPLATE_N -1), h_OutcomeCPU);
  }
////////////////////////////////////////////////////////////////////////////
//  COMPERING THE GPU AND CPU RESULTS
////////////////////////////////////////////////////////////////////////////

  printf("...comparing GPU VS CPU \n");

  sum_delta   = 0.0;
  max_abs_err = 0.0;
  sum_ref     = 0.0;

  for(i = 0; i < OUTPUT_M * OUTPUT_N; i++){
    sum_delta += fabs((double)h_OutcomeCPU[i] - (double)h_OutcomeGPU[i]);
    if(fabs((double)h_OutcomeCPU[i] - (double)h_OutcomeGPU[i]) > max_abs_err){
      max_abs_err = fabs((double)h_OutcomeCPU[i] - (double)h_OutcomeGPU[i]);}
    sum_ref   += fabs((double)h_OutcomeCPU[i]);
  }
  L1norm = sum_delta / sum_ref;

  speedup = CPU_time/GPU_time ;

  printf("GPU is %f times faster than CPU\n",speedup);
  printf("Max. absolute error is %E\n",max_abs_err);
  printf("Relative error: %E\n", L1norm);

  printf((L1norm < 1e-6) ? "TEST PASSED\n\n" : "TEST FAILED\n\n");

////////////////////////////////////////////////////////////////////////////
//  SHUTTING DOWN
////////////////////////////////////////////////////////////////////////////

  printf("Shutting down...\n");

  CUDA_SAFE_CALL( cudaFree(d_Frame ) );
  CUDA_SAFE_CALL( cudaFree(d_Outcome) );
  CUDA_SAFE_CALL( cudaFree(d_Pad_Frame) );

  free(h_OutcomeGPU);
  free(h_OutcomeCPU);
  free(h_Frame);
  free(h_Template);
  free(h_Pad_Frame);

  CUT_SAFE_CALL(cutDeleteTimer(hTimer));

  CUT_EXIT(argc, argv);
```

```
}
```

```
//************************************************************************
//*********************************GPU FUNCTIONS**************************
//************************************************************************

////////////////////////////////////////////////////////////////////////
//  GPU FUNCTIONS
////////////////////////////////////////////////////////////////////////
#include <stdio.h>
#include <ourConstants_Final_America.h>
#define IMUL(a, b) __mul24(a, b)

__device__ __constant__ float d_Template[TEMPLATE_M*TEMPLATE_N];

////////////////////////////////////////////////////////////////////////
//  FRAME CENTRIC ON GPU (Optimized)
////////////////////////////////////////////////////////////////////////
__global__ void matrix_correlationGPU(
                                        float *d_Outcome,
                                        float *d_Pad_Frame
                                        ){


  const int i = blockDim.y * blockIdx.y + threadIdx.y;
  const int j = blockDim.x * blockIdx.x + threadIdx.x;

  const int k0 = blockIdx.y * blockDim.y;
  const int l0 = blockIdx.x * blockDim.x;

  __shared__ float s_Pad_Frame[S_PAD_M * S_PAD_N];

//Every thread copies a part of the frame.
  for(int l = threadIdx.x; l<= blockDim.x+TEMPLATE_N-2; l+=blockDim.x){
    for(int k = threadIdx.y; k <= blockDim.y+TEMPLATE_M-2; k+=blockDim.y){
      access(s_Pad_Frame, k, l, S_PAD_M ) =
            access(d_Pad_Frame, k+k0, l+l0, PAD_FRAME_M);
    }
  }

  __syncthreads();

  float s = 0.0;

//By changing the order of the loops it failed.
  for(int k=0; k<TEMPLATE_M; k++){
    for(int l=0; l<TEMPLATE_N; l++){
      s +=
            access(s_Pad_Frame, threadIdx.y + k, threadIdx.x + l, BLOCKDIM_Y + 2*TEMPLATE_M-2) *
            access(d_Template, k, l, TEMPLATE_M);
    }
  }
  access(d_Outcome, i, j, FRAME_M+TEMPLATE_M-1) = s;
}




//************************************************************************
//*********************************CPU FUNCTIONS**************************
//************************************************************************

////////////////////////////////////////////////////////////////////////
// CPU FUNCTIONS
```

```
////////////////////////////////////////////////////////////////////////////

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <ourConstants_Final_America.h>


////////////////////////////////////////////////////////////////////////////
// FRAME CENTRIC ON CPU
////////////////////////////////////////////////////////////////////////////

extern "C" void correlationCPU(
                                        float *h_OutcomeCPU,
                                        float *h_Pad_Frame,
                                        float *h_Template
                                        ){

  int i,j,k,l;

  for(i=0; i< FRAME_M + TEMPLATE_M -1; i++){
    for(j=0; j< FRAME_N + TEMPLATE_N -1; j++){
      float s=0.0;
      for (k=0; k < TEMPLATE_M; k++){
            for (l=0; l < TEMPLATE_N; l++){
              s += h_Pad_Frame[i+k+ (j+l) * (FRAME_M + 2*TEMPLATE_M-2)]
                * h_Template[k+l*TEMPLATE_M];
            }
      }
      h_OutcomeCPU[i+j*(FRAME_M+TEMPLATE_M-1)] = s;
    }
  }
}



////////////////////////////////////////////////////////////////////////////
// PRINTING FUNCTION
////////////////////////////////////////////////////////////////////////////

extern "C"
void Printing_Matrix(

                            int Rows,
                            int Columns,
                            float *Matrix
                            )
{
  int i,j;

  for ( i=0; i< Rows ; i++) {

    for (j=0; j< Columns ; j++){

      printf(" %4.1f " , Matrix[i+j*Rows]);

    }
    printf("\n");

  }
}


////////////////////////////////////////////////////////////////////////////
```

```
// GETTING A SUBMATRIX
//////////////////////////////////////////////////////////////////////////////

extern "C" void GetSubMatrix(float *h_Frame,
                             float *h_Template,
                             int startRow,
                             int startCol
                             ){

  if (startRow + TEMPLATE_M > FRAME_M || startCol + TEMPLATE_N > FRAME_N){
    printf("ERROR: incorrect dimensions for submatrix!\n");
    return;
  }

  int i, j;
  for (i = 0; i < TEMPLATE_M; i++){
    for (j = 0; j < TEMPLATE_N; j++){
      h_Template[i+j*TEMPLATE_M] = h_Frame[startRow+i+(startCol+j)*FRAME_M];
    }
  }
}




//****************************************************************************
//************************OUR CONSTANTS LIBRARY*******************************
//****************************************************************************

/*////////////////////// SIZE RESTRICTIONS ////////////////////////////////
1. Output dimensions devided by Template must result in an integer
*//////////////////////////////////////////////////////////////////////////

#define FRAME_M 2080
#define FRAME_N 2080

#define  TEMPLATE_M 21
#define  TEMPLATE_N 21

#define BLOCKDIM_X TEMPLATE_N
#define BLOCKDIM_Y TEMPLATE_M

#define OUTPUT_M (FRAME_M + TEMPLATE_M - 1)
#define OUTPUT_N (FRAME_M + TEMPLATE_N - 1)

#define PAD_FRAME_M  (FRAME_M + 2*(TEMPLATE_M - 1))
#define PAD_FRAME_N  (FRAME_N + 2*(TEMPLATE_N - 1))

#define S_PAD_M  (BLOCKDIM_Y + 2*(TEMPLATE_M - 1))
#define S_PAD_N  (BLOCKDIM_X + 2*(TEMPLATE_N - 1))

#define TRUE 1
#define FALSE 0
#define PRINT_INPUT_MATRICES FALSE
#define PRINT_OUTCOME_MATRICES FALSE



#define access(a,i,j,ldim) a[(i) + (j) * (ldim)]
```

## 5.2. Local Correlation

```
//****************************************************************************
//*********************************MAIN FILE**********************************
//****************************************************************************

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <cutil.h>
#include <assert.h>
#include <time.h>
#include <sys/time.h>
#include <math.h>

////////////////////////////////////////////////////////////////////////////////
// Common host and device functions
////////////////////////////////////////////////////////////////////////////////
//Round a / b to nearest higher integer value
int iDivUp(int a, int b){
  return (a % b != 0) ? (a / b + 1) : (a / b);
}

////////////////////////////////////////////////////////////////////////////////
// Including
////////////////////////////////////////////////////////////////////////////////
#include <gpu_correlation.cu>
#include <ourConstants_correlation.h>

////////////////////////////////////////////////////////////////////////////////
// CPU functions
////////////////////////////////////////////////////////////////////////////////
extern "C" void Printing_Matrix(
                                        int Rows,
                                        int Columns,
                                        float *Matrix
                                        );

extern "C" void correlationCPU(
                                       float *h_OutcomeCPU,
                                       float *h_Pad_Frame,
                                       float *h_Template
                                       );

extern "C" void GetSubMatrix (float *h_Frame,
                                      float *h_Template,
                                  int startRow,
                                      int startCol
                                      );

extern "C" float MatrixMean (float *matrix,
                                      int m,
                                      int n
                                      );

extern "C" float MatrixNorm (float *matrix,
                                      int m,
                                      int n
                                      );

extern "C" void FindMaxElement (float *matrix,
```

```
                                        float *max,
                                        int m,
                                        int n
                                        );

extern "C" void Get_Frame_From_File(float *h_Frame);

extern "C" void Sent_Template_to_File(float *h_Template);

////////////////////////////////////////////////////////////////////////////
// Data configuration
////////////////////////////////////////////////////////////////////////////
const  int FRAME_SIZE    = FRAME_M * FRAME_N * sizeof(float);
const  int TEMPLATE_SIZE = TEMPLATE_M * TEMPLATE_N * sizeof(float);

const   int OUTCOME_SIZE  = (FRAME_M + TEMPLATE_M -1) * (FRAME_N + TEMPLATE_N -1)*
sizeof(float);
const  int PAD_FRAME_SIZE= PAD_FRAME_M * PAD_FRAME_N * sizeof(float);

const int GRID_Y = iDivUp((FRAME_M + TEMPLATE_M - 1), BLOCKDIM_Y);
const int GRID_X = iDivUp((FRAME_N + TEMPLATE_N - 1), BLOCKDIM_X);
////////////////////////////////////////////////////////////////////////////
// Defining
////////////////////////////////////////////////////////////////////////////
#define WARMUP

////////////////////////////////////////////////////////////////////////////
// Main program
////////////////////////////////////////////////////////////////////////////
int main(int argc, char **argv){

 float
  *h_Template,
  *h_Frame,
  *h_OutcomeCPU,
  *h_OutcomeGPU,
  *h_Pad_Frame;

 float
  *d_Frame,
  *d_Outcome,
  *d_Pad_Frame;

 timeval randome_time, start_in, end_in, start_out, end_out, start_CPU, end_CPU;
 double in_time, out_time, GPU_time, CPU_time, flop, GPU_gflops, CPU_gflops,
  speedup, L1norm, sum_delta, max_abs_err, sum_ref;
 int i, j, startRow, startCol;
 float max_GPU[3], max_CPU[3], Template_Mean, Template_Norm;
 unsigned int hTimer;


 CUT_DEVICE_INIT(argc, argv);

 cudaSetDevice(0);

 CUT_SAFE_CALL(cutCreateTimer(&hTimer));

 printf("\nFrame Size is %i x %i\n", FRAME_M, FRAME_N);
 printf("Template size is %i x %i\n", TEMPLATE_M, TEMPLATE_N);
 printf("Outcome  size  is  %i  x  %i\n\n", (FRAME_M + TEMPLATE_M -1),   (FRAME_N +
TEMPLATE_N -1));

 printf("Initializing data...\n");
```

```
  h_Template  = (float *)malloc(TEMPLATE_SIZE);
  h_Frame     = (float *)malloc(FRAME_SIZE);
  h_OutcomeGPU= (float *)malloc(OUTCOME_SIZE);
  h_OutcomeCPU= (float *)malloc(OUTCOME_SIZE);
  h_Pad_Frame = (float *)malloc(PAD_FRAME_SIZE);

  CUDA_SAFE_CALL( cudaMalloc( (void **)&d_Outcome ,OUTCOME_SIZE) );
  CUDA_SAFE_CALL( cudaMalloc( (void **)&d_Frame , FRAME_SIZE) );
  CUDA_SAFE_CALL( cudaMalloc( (void **)&d_Pad_Frame ,PAD_FRAME_SIZE) );

////////////////////////////////////////////////////////////////////////////
//  Initialising Data
////////////////////////////////////////////////////////////////////////////

//Initializing the Frame with random values
  gettimeofday(&randome_time, NULL);
  srand(randome_time.tv_sec);

  if (GET_FILE == TRUE)
    Get_Frame_From_File(h_Frame);
  else{
    for(i = 0; i < FRAME_M *  FRAME_N; i++){
      h_Frame[i] = (float) (rand() % 1000 + 1);
    }
  }

  if (PRINT_INPUT_MATRICES == TRUE){
    printf("The Frame Matrix is printed here \n");
    Printing_Matrix(FRAME_M, FRAME_N, h_Frame);
  }

//Initializing the Template as part of the Frame
  startRow = rand() % (FRAME_M - TEMPLATE_M);
  startCol = rand() % (FRAME_N - TEMPLATE_N);

  GetSubMatrix(
                h_Frame,
                h_Template,
                startRow,
                startCol
                );

  printf("Template is located startng at the (%d %d) point in the Frame\n\n", startRow, startCol);

  if (GET_FILE == TRUE)
    Sent_Template_to_File(h_Template);

  if (PRINT_INPUT_MATRICES == TRUE){
    printf("The Template Matrix is printed here \n");
    Printing_Matrix(TEMPLATE_M, TEMPLATE_N, h_Template);
  }

//Normilizing the Template
  Template_Mean = 0.0;
  Template_Norm = 0.0;

  Template_Mean =  MatrixMean(h_Template, TEMPLATE_M, TEMPLATE_N);
  for(i=0; i < TEMPLATE_M*TEMPLATE_N; i++){
    h_Template[i] = h_Template[i] - Template_Mean;
  }

  Template_Norm = MatrixNorm(h_Template, TEMPLATE_M, TEMPLATE_N);
  Template_Norm = (Template_Norm == 0) ? 1 : Template_Norm;
  for(i=0; i < TEMPLATE_M*TEMPLATE_N; i++){
```

```
    h_Template[i] = h_Template[i] / Template_Norm;
  }

  if (PRINT_INPUT_MATRICES == TRUE){
    printf("The Normalized Template is printed here: \n");
    Printing_Matrix(TEMPLATE_M, TEMPLATE_N, h_Template);
  }

//Pading the frame
  for (i =0; i<PAD_FRAME_M*PAD_FRAME_N; i++)
    h_Pad_Frame[i] = 0;

  for (i=0; i < FRAME_M; i++) {
    for (j=0; j < FRAME_N; j++) {
      h_Pad_Frame[i + TEMPLATE_M - 1 + (j + TEMPLATE_N - 1) * PAD_FRAME_M]=
            h_Frame[i + j * FRAME_M];
    }
  }

  if (PRINT_INPUT_MATRICES == TRUE){
    printf("The Padded Matrix is printed here \n");
    Printing_Matrix(PAD_FRAME_M, PAD_FRAME_N, h_Pad_Frame);
  }

////////////////////////////////////////////////////////////////////////////
//  Transporting Data to GPU
////////////////////////////////////////////////////////////////////////////

  gettimeofday(&start_in, NULL);
  CUDA_SAFE_CALL( cudaMemcpyToSymbol(d_Template, h_Template, TEMPLATE_SIZE));
  CUDA_SAFE_CALL( cudaMemcpy(d_Frame, h_Frame, FRAME_SIZE, cudaMemcpyHostToDevice) );
  CUDA_SAFE_CALL(       cudaMemcpy(d_Pad_Frame,       h_Pad_Frame,       PAD_FRAME_SIZE,
cudaMemcpyHostToDevice) );
  gettimeofday(&end_in, NULL);
////////////////////////////////////////////////////////////////////////////
//  Setting the block & grid dimensions
////////////////////////////////////////////////////////////////////////////
  dim3 dimBlock (BLOCKDIM_X, BLOCKDIM_Y);
  dim3 dimGrid (GRID_X, GRID_Y);

////////////////////////////////////////////////////////////////////////////
//  WARMING UP
////////////////////////////////////////////////////////////////////////////
#ifdef WARMUP

  printf("Warming Up... \n");

  matrix_correlationGPU<<<dimGrid, dimBlock>>>(
                                                d_Outcome,
                                                d_Pad_Frame
                                                );

  CUT_CHECK_ERROR("matrix_correlationGPU() execution failed\n");
  CUDA_SAFE_CALL( cudaThreadSynchronize() );
#endif

////////////////////////////////////////////////////////////////////////////
//  GPU FUNCTIONS
////////////////////////////////////////////////////////////////////////////
  CUDA_SAFE_CALL(cudaMemset(d_Outcome, 0, OUTCOME_SIZE));

  printf("\nGPU  matrix_correlation...\n");

  CUDA_SAFE_CALL( cudaThreadSynchronize() );
```

```
  CUT_SAFE_CALL( cutResetTimer(hTimer) );
  CUT_SAFE_CALL( cutStartTimer(hTimer) );

  matrix_correlationGPU<<<dimGrid, dimBlock>>>(
                                                d_Outcome,
                                                d_Pad_Frame
                                                );

  CUT_CHECK_ERROR("matrix_correlationGPU() execution failed\n");
  CUDA_SAFE_CALL( cudaThreadSynchronize() );
  CUT_SAFE_CALL(cutStopTimer(hTimer));
  GPU_time = cutGetTimerValue(hTimer);

  gettimeofday(&start_out, NULL);
  CUDA_SAFE_CALL( cudaMemcpy(h_OutcomeGPU, d_Outcome, OUTCOME_SIZE,
cudaMemcpyDeviceToHost) );
  gettimeofday(&end_out, NULL);

  in_time = (double)((end_in.tv_usec - start_in.tv_usec)/1.0e6 + end_in.tv_sec - start_in.tv_sec);
  out_time = (double)((end_out.tv_usec - start_out.tv_usec)/1.0e6 + end_out.tv_sec - start_out.tv_sec);
  flop = 1e-6 * (FRAME_M + TEMPLATE_M - 1)*(FRAME_N + TEMPLATE_N -
1)*(5*(TEMPLATE_M*TEMPLATE_N)+5);
  GPU_gflops = flop / GPU_time;

  printf("Transfer Data TO GPU time is: %f msec\n", in_time*1000);
  printf("Transfer Data FROM GPU time is: %f msec\n", out_time*1000);
  printf("GPU matrix_correlation time is : %f msec\n", GPU_time);
  printf("GPU matrix_correlation GFLOPS : %f \n", GPU_gflops);

  FindMaxElement(h_OutcomeGPU,
                 max_GPU,
                 OUTPUT_M,
                 OUTPUT_N);

  printf("Maximum Element on GPU is %3.2f and located at (%5.0f, %5.0f)\n",max_GPU[0],
max_GPU[1], max_GPU[2]);


////////////////////////////////////////////////////////////////////////////
//  CPU FUNCTIONS
////////////////////////////////////////////////////////////////////////////
  printf("\nCPU matrix correlation..\n");

  gettimeofday(&start_CPU, NULL);
  correlationCPU(
                 h_OutcomeCPU,
                 h_Pad_Frame,
                 h_Template
                 );
  gettimeofday(&end_CPU, NULL);

  CPU_time = (double)(((end_CPU.tv_usec - start_CPU.tv_usec)/1.0e6 + end_CPU.tv_sec -
start_CPU.tv_sec)*1000);
  CPU_gflops = flop / CPU_time;

  printf("CPU matrix correlation time is: %f msec\n", CPU_time);
  printf("CPU matrix_correlation GFLOPS : %f \n", CPU_gflops);

  FindMaxElement(h_OutcomeCPU,
                 max_CPU,
                 OUTPUT_M,
                 OUTPUT_N);
```

```
  printf("Maximum Element on CPU is %3.2f and located at (%5.0f, %5.0f)\n",max_CPU[0], max_CPU[1], max_CPU[2]);

//Printing Output Data
  if (PRINT_OUTCOME_MATRICES == TRUE){
    printf("The GPU Correlation Matrix is printed here: \n");
    Printing_Matrix((FRAME_M + TEMPLATE_M -1),(FRAME_N + TEMPLATE_N -1), h_OutcomeGPU);

    printf("The CPU correlation Matrix is printed here: \n");
    Printing_Matrix((FRAME_M + TEMPLATE_M -1),(FRAME_N + TEMPLATE_N -1), h_OutcomeCPU);
  }

////////////////////////////////////////////////////////////////////////////
//  COMPERING THE GPU AND CPU RESULTS
////////////////////////////////////////////////////////////////////////////
  printf("\n...comparing GPU VS CPU \n");

  sum_delta = 0.0;
  max_abs_err = 0.0;
  sum_ref   = 0.0;

  for(i = 0; i < OUTPUT_M * OUTPUT_N; i++){
    sum_delta += fabs((double)h_OutcomeCPU[i] - (double)h_OutcomeGPU[i]);
    if(fabs((double)h_OutcomeCPU[i] - (double)h_OutcomeGPU[i]) > max_abs_err){
      max_abs_err = fabs((double)h_OutcomeCPU[i] - (double)h_OutcomeGPU[i]);}
    sum_ref   += fabs((double)h_OutcomeCPU[i]);
  }
  L1norm = sum_delta / sum_ref;

  speedup = CPU_time/GPU_time ;

  printf("GPU is %f times faster than CPU\n",speedup);
  printf("Max. absolute error is %E\n",max_abs_err);
  printf("Relative error: %E\n", L1norm);

  printf((L1norm < 1e-6) ? "TEST PASSED\n\n" : "TEST FAILED\n\n");

////////////////////////////////////////////////////////////////////////////
//  SHUTTING DOWN
////////////////////////////////////////////////////////////////////////////

  printf("Shutting down...\n");

  CUDA_SAFE_CALL( cudaFree(d_Frame ) );
  CUDA_SAFE_CALL( cudaFree(d_Outcome) );
  CUDA_SAFE_CALL( cudaFree(d_Pad_Frame) );

  free(h_OutcomeGPU);
  free(h_OutcomeCPU);
  free(h_Frame);
  free(h_Template);
  free(h_Pad_Frame);

  CUT_SAFE_CALL(cutDeleteTimer(hTimer));

  CUT_EXIT(argc, argv);
}
```

```
//*****************************************************************************
//*****************************GPU FUNCTIONS*************************************
//*****************************************************************************

////////////////////////////////////////////////////////////////////////////
//  GPU FUNCTIONS
////////////////////////////////////////////////////////////////////////////

#include <stdio.h>
#include <ourConstants_correlation.h>
#define IMUL(a, b) __mul24(a, b)

__device__ __constant__ float d_Template[TEMPLATE_M*TEMPLATE_N];

////////////////////////////////////////////////////////////////////////////
//  FRAME CENTRIC ON GPU (Optimized)
////////////////////////////////////////////////////////////////////////////


__global__ void matrix_correlationGPU(
                                        float *d_Outcome,
                                        float *d_Pad_Frame
                                        ){

  const int i = blockDim.y * blockIdx.y + threadIdx.y;
  const int j = blockDim.x * blockIdx.x + threadIdx.x;

  const int k0 = blockIdx.y * blockDim.y;
  const int l0 = blockIdx.x * blockDim.x;

  __shared__ float s_Pad_Frame[S_PAD_M * S_PAD_N];

//Every thread copies a part of the frame.
  for(int l = threadIdx.x; l <= blockDim.x + TEMPLATE_N - 2; l += blockDim.x){
    for(int k = threadIdx.y; k <=  blockDim.y + TEMPLATE_M - 2; k += blockDim.y){
      access(s_Pad_Frame, k, l, S_PAD_M ) =
            access(d_Pad_Frame, k+k0, l+l0, PAD_FRAME_M);
    }
  }

  __syncthreads();

  float PF_element = 0.0, PF_sum = 0.0, PF_corr = 0.0, PF_square = 0.0;

  for (int k = 0; k < TEMPLATE_M; k++){
    for (int l = 0; l < TEMPLATE_N; l++){
      PF_element = access(s_Pad_Frame, threadIdx.y + k, threadIdx.x + l, S_PAD_M);
      PF_corr += PF_element * access(d_Template, k, l, TEMPLATE_M);
      PF_sum += PF_element;
      PF_square += PF_element * PF_element;
    }
  }

  access(d_Outcome, i, j, OUTPUT_M) = PF_corr / sqrtf(PF_square - PF_sum*PF_sum/(TEMPLATE_M
* TEMPLATE_N));
}
```

```
//*************************************************************************
//*****************************CPU FUNCTIONS*******************************
//*************************************************************************

////////////////////////////////////////////////////////////////////////////
// CPU FUNCTIONS
////////////////////////////////////////////////////////////////////////////

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <ourConstants_correlation.h>

////////////////////////////////////////////////////////////////////////////
// FRAME CENTRIC ON CPU
////////////////////////////////////////////////////////////////////////////

extern "C" void correlationCPU(
                                    float *h_OutcomeCPU,
                                    float *h_Pad_Frame,
                                    float *h_Template
                                    ){

 int i,j,k,l;
 float PF_element, PF_corr, PF_sum, PF_square;

 for(i=0; i< FRAME_M + TEMPLATE_M -1; i++){
   for(j=0; j< FRAME_N + TEMPLATE_N -1; j++){
     PF_corr = 0.0;
     PF_sum = 0.0;
     PF_square = 0.0;
     for (k=0; k < TEMPLATE_M; k++){
           for (l=0; l < TEMPLATE_N; l++){
             PF_element = h_Pad_Frame[i+k + (j+l)*(FRAME_M + 2*(TEMPLATE_M-1))];
             PF_corr += PF_element * h_Template[k + l*TEMPLATE_M];
             PF_sum += PF_element;
             PF_square += PF_element * PF_element;
           }
     }
 h_OutcomeCPU[i + j*(FRAME_M+TEMPLATE_M-1)] = PF_corr / sqrtf(PF_square -
PF_sum*PF_sum/(TEMPLATE_M * TEMPLATE_N));
   }
 }
}

////////////////////////////////////////////////////////////////////////////
// PRINTING FUNCTION
////////////////////////////////////////////////////////////////////////////

extern "C" void Printing_Matrix(
                                    int Rows,
                                    int Columns,
                                    float *Matrix
                                    ){
 int i,j;

 for ( i=0; i< Rows ; i++) {
   for (j=0; j< Columns ; j++){
     printf(" %5.3f " , Matrix[i+j*Rows]);
   }
   printf("\n");
 }
}
```

```
////////////////////////////////////////////////////////////////////////////
// GETTING A SUBMATRIX
////////////////////////////////////////////////////////////////////////////

extern "C" void GetSubMatrix(float *h_Frame,
                             float *h_Template,
                             int startRow,
                             int startCol
                             ){

  if (startRow + TEMPLATE_M > FRAME_M || startCol + TEMPLATE_N > FRAME_N){
    printf("ERROR: incorrect dimensions for submatrix!\n");
    return;
  }

  int i, j;
  for (i = 0; i < TEMPLATE_M; i++){
    for (j = 0; j < TEMPLATE_N; j++){
      h_Template[i+j*TEMPLATE_M] = h_Frame[startRow+i+(startCol+j)*FRAME_M];
    }
  }
}

////////////////////////////////////////////////////////////////////////////
// Calculates the mean value of the matrix
////////////////////////////////////////////////////////////////////////////
extern "C" float MatrixMean (float *matrix, int m, int n)
{
  int i;
  float mean = 0;

  for (i=0; i< m*n; i++){
     mean += matrix[i];
    }

  return mean / (m*n);
}


////////////////////////////////////////////////////////////////////////////
// Calculates the Frobenius norm: sqrt(sum(diag(M'*M)))
////////////////////////////////////////////////////////////////////////////
extern "C" float MatrixNorm (float *matrix, int m, int n)
{
  int i;
  float norm = 0;

  for (i=0; i< m*n; i++){
     norm += matrix[i] * matrix[i];
  }
  return sqrt(norm);
}


////////////////////////////////////////////////////////////////////////////
// Find Maximum Element
////////////////////////////////////////////////////////////////////////////
extern "C" void FindMaxElement (float *matrix, float *max, int m, int n)
{
  for(int i = 0; i < m; i++){
    for(int j = 0; j < n; j++){
      if(matrix[i + j*m] > max[0]){
            max[0] = matrix[i + j*m];
```

```
            max[1] = i;
            max[2] = j;
      }
    }
  }
}



////////////////////////////////////////////////////////////////////////
// Getting Matrix from File
////////////////////////////////////////////////////////////////////////
extern "C" void Get_Frame_From_File(float *h_Frame){

  FILE *In_file;

  In_file = fopen("Input_Frame.txt", "r");

  if(In_file==NULL) {
    printf("Error: can't open Input_File.\n");
  }
  else {
    printf("Input_File opened.\n");

    for(int i = 0; i < FRAME_M; i++){
      for(int j = 0; j < FRAME_N; j++){
        fscanf(In_file, "%f", &h_Frame[i+j*FRAME_M]);
      }
    }
  }

  fclose(In_file);
  printf("Input_File closed.\n");
}

////////////////////////////////////////////////////////////////////////
// Writting Matrix To File
////////////////////////////////////////////////////////////////////////
extern "C" void Sent_Template_to_File(float *h_Template){

  FILE *Out_file;

  Out_file = fopen("Output_Template.txt", "w");

  if(Out_file==NULL) {
    printf("Error writting out\n");
  }
  else {
    printf("Output_File opened.\n");

    for(int i = 0; i < TEMPLATE_M; i++){
      if (i!=0){
            fprintf(Out_file, "\n ");}
      for(int j = 0; j < TEMPLATE_N; j++){
        fprintf(Out_file, "%3.0f ", h_Template[i+j*TEMPLATE_M]);
      }
    }
  }

  fclose(Out_file);
  printf("Output_File closed.\n");
}
```

```
//**************************************************************************
//*****************************OUR CONSTANTS LIBRARY************************
//**************************************************************************

/*////////////////////// SIZE RESTRICTIONS ////////////////////////////////
1. Output dimensions devided by Template must result in an integer
*//////////////////////////////////////////////////////////////////////////

#define FRAME_M 610
#define FRAME_N 610

#define  TEMPLATE_M 21
#define  TEMPLATE_N 21

#define BLOCKDIM_X TEMPLATE_N
#define BLOCKDIM_Y TEMPLATE_M

#define OUTPUT_M (FRAME_M + TEMPLATE_M - 1)
#define OUTPUT_N (FRAME_M + TEMPLATE_N - 1)

#define PAD_FRAME_M  (FRAME_M + 2*(TEMPLATE_M - 1))
#define PAD_FRAME_N  (FRAME_N + 2*(TEMPLATE_N - 1))

#define S_PAD_M  (BLOCKDIM_Y + 2*(TEMPLATE_M - 1))
#define S_PAD_N  (BLOCKDIM_X + 2*(TEMPLATE_N - 1))

#define TRUE 1
#define FALSE 0
#define GET_FILE TRUE
#define PRINT_INPUT_MATRICES FALSE
#define PRINT_OUTCOME_MATRICES FALSE



#define access(a,i,j,ldim) a[(i) + (j) * (ldim)]
```

## 5.3. Multi – GPU

```
//**************************************************************************
//*********************************MAIN FILE********************************
//**************************************************************************


#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <cutil.h>
#include <assert.h>
#include <time.h>
#include <sys/time.h>
#include <math.h>
#include <multithreading.h>
```

```
////////////////////////////////////////////////////////////////////////////
// Common host and device functions
////////////////////////////////////////////////////////////////////////////
//Round a / b to nearest higher integer value
int iDivUp(int a, int b){
  return (a % b != 0) ? (a / b + 1) : (a / b);
}

////////////////////////////////////////////////////////////////////////////
// Including Files
////////////////////////////////////////////////////////////////////////////
#include <ourConstants_correlation_M_GPU.h>
#include <gpu_correlation_M_GPU.cu>

////////////////////////////////////////////////////////////////////////////
// CPU functions
////////////////////////////////////////////////////////////////////////////
extern "C" void Printing_Matrix(
                                        int Rows,
                                        int Columns,
                                        float *Matrix
                                        );

extern "C" void correlationCPU(
                                        float *h_OutcomeCPU,
                                        float *h_Pad_Frame,
                                        float *h_Template
                                        );

extern "C" void GetSubMatrix (float *h_Frame,
                                        float *h_Template,
                                    int startRow,
                                        int startCol
                                        );

extern "C" float MatrixMean (float *matrix,
                                        int m,
                                        int n
                                        );

extern "C" float MatrixNorm (float *matrix,
                                        int m,
                                        int n
                                        );

extern "C" void FindMaxElement (float *matrix,
                                        float *max,
                                        int m,
                                        int n
                                        );

extern "C" void Get_Frame_From_File(float *h_Frame);
////////////////////////////////////////////////////////////////////////////
// Data configuration
////////////////////////////////////////////////////////////////////////////
const  int FRAME_SIZE    = FRAME_M * FRAME_N * sizeof(float);
const  int TEMPLATE_SIZE = TEMPLATE_M * TEMPLATE_N * sizeof(float);
const  int OUTCOME_SIZE  = OUTPUT_M * OUTPUT_N * sizeof(float);
const  int PAD_FRAME_SIZE= PAD_FRAME_M * PAD_FRAME_N * sizeof(float);

float h_OutcomeGPU[OUTCOME_SIZE];
float h_Template[TEMPLATE_SIZE];
////////////////////////////////////////////////////////////////////////////
//  GPU Thread
```

```
////////////////////////////////////////////////////////////////////////////
typedef struct {
  int device, k, GPU_NUMBER, PAD_FRAME_SIZE_PER_DEVICE, OUTCOME_SIZE_PER_DEVICE;
  float *h_Pad_Frame;
  timeval start_in, end_in, start_GPU, end_GPU, start_out, end_out, starting_Thread;
} GPU_Struct;

static CUT_THREADPROC GPU_Thread(GPU_Struct *str){

  CUDA_SAFE_CALL(cudaSetDevice(str->real_device));

  float *d_Pad_Frame, *d_OutcomeGPU;

  CUDA_SAFE_CALL( cudaMalloc((void**)&d_Pad_Frame, str->PAD_FRAME_SIZE_PER_DEVICE));
  CUDA_SAFE_CALL( cudaMalloc((void**)&d_OutcomeGPU, str->OUTCOME_SIZE_PER_DEVICE));

  gettimeofday (&str->start_in, NULL);
  CUDA_SAFE_CALL( cudaMemcpyToSymbol(d_Template, h_Template, TEMPLATE_SIZE));
  CUDA_SAFE_CALL( cudaMemcpy(d_Pad_Frame, str->h_Pad_Frame, str-
>PAD_FRAME_SIZE_PER_DEVICE, cudaMemcpyHostToDevice));
  gettimeofday (&str->end_in, NULL);


   int GRID_Y = iDivUp(OUTPUT_M, BLOCKDIM_Y);
  int GRID_X = iDivUp((str->OUTCOME_SIZE_PER_DEVICE),
BLOCKDIM_X*OUTPUT_M*sizeof(float));

  dim3 dimGrid (GRID_X, GRID_Y);
  dim3 dimBlock (BLOCKDIM_X, BLOCKDIM_Y);

//Perform GPU computations
  gettimeofday (&str->start_GPU, NULL);
  matrix_correlationGPU<<<dimGrid, dimBlock>>>(
                                                d_OutcomeGPU,
                                                d_Pad_Frame
                                                );
  CUT_CHECK_ERROR("matrix_correlationGPU() execution failed\n");
  CUDA_SAFE_CALL( cudaThreadSynchronize() );
  gettimeofday (&str->end_GPU, NULL);


//Read back GPU results
  gettimeofday (&str->start_out, NULL);
  CUDA_SAFE_CALL(  cudaMemcpy(&h_OutcomeGPU[(str->device)*(FRAME_M + TEMPLATE_M -
1)*(FRAME_N/(str->GPU_NUMBER)) + (FRAME_M + TEMPLATE_M - 1)*(str->k)*(TEMPLATE_N-
1)], d_OutcomeGPU, str->OUTCOME_SIZE_PER_DEVICE, cudaMemcpyDeviceToHost) );
  gettimeofday (&str->end_out, NULL);

  if (PRINT_PER_GPU_MATRICES == TRUE){
    printf("%d's GPUs h_Outcome:\n", str->device);
    Printing_Matrix((FRAME_M + TEMPLATE_M-1), (FRAME_N + TEMPLATE_N -1),
h_OutcomeGPU);
  }


//Shut down this GPU
  CUDA_SAFE_CALL( cudaFree(d_OutcomeGPU));
  CUDA_SAFE_CALL( cudaFree(d_Pad_Frame));

  CUT_THREADEND;
}

////////////////////////////////////////////////////////////////////////////
// Main program
```

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

  float
    *h_Frame,
    *h_OutcomeCPU,
    *h_Pad_Frame;

  double L1norm, sum_delta, max_abs_err, sum_ref;
  int i, j, d, r, GPU_N;
  float max_GPU[3], max_CPU[3];
  timeval start_kernel, end_kernel;
  double kernel_time = 0.0, in_time = 0.0, out_time = 0.0, GPU_time = 0.0;

  CUDA_SAFE_CALL(cudaGetDeviceCount(&GPU_N));

  GPU_Struct str[GPU_N];
  CUTThread threadID[GPU_N];

  printf("\nFrame Size is %i x %i\n", FRAME_M, FRAME_N);
  printf("Template size is %i x %i\n", TEMPLATE_M, TEMPLATE_N);
  printf("Outcome size is %i x %i\n\n", (FRAME_M + TEMPLATE_M -1), (FRAME_N +
TEMPLATE_N -1));

  printf("Initializing data...\n\n");

  h_Frame     = (float *)malloc(FRAME_SIZE);
  h_OutcomeCPU= (float *)malloc(OUTCOME_SIZE);
  h_Pad_Frame = (float *)malloc(PAD_FRAME_SIZE);
////////////////////////////////////////////////////////////////////////////
//  Initialising Data
////////////////////////////////////////////////////////////////////////////

//Initializing the Frame with random values
 timeval randome_time;
 gettimeofday(&randome_time, NULL);
 srand(randome_time.tv_sec);

 if (GET_FILE == TRUE)
   Get_Frame_From_File(h_Frame);
 else{
   for(i = 0; i < FRAME_M *  FRAME_N; i++){
     h_Frame[i] = (float) (rand() % 1000 + 1);
   }
 }

 if (PRINT_INPUT_MATRICES == TRUE){
   printf("The Frame Matrix is printed here \n");
   Printing_Matrix(FRAME_M, FRAME_N, h_Frame);
 }

//Initializing the Template as part of the Frame
 int startRow = rand() % (FRAME_M - TEMPLATE_M);
 int startCol = rand() % (FRAME_N - TEMPLATE_N);

  GetSubMatrix(
                h_Frame,
                h_Template,
                startRow,
                startCol
                );

  printf("Template is located startng at the (%d %d) point in the Frame\n\n", startRow, startCol);
```

```
  if (PRINT_INPUT_MATRICES == TRUE){
    printf("The Template Matrix is printed here \n");
    Printing_Matrix(TEMPLATE_M, TEMPLATE_N, h_Template);
  }

//Normilizing the Tamplate
  float Template_Mean = 0.0;
  float Template_Norm = 0.0;

  Template_Mean =  MatrixMean(h_Template, TEMPLATE_M, TEMPLATE_N);
  for(i=0; i < TEMPLATE_M*TEMPLATE_N; i++){
    h_Template[i] = h_Template[i] - Template_Mean;
  }

  Template_Norm = MatrixNorm(h_Template, TEMPLATE_M, TEMPLATE_N);
  Template_Norm = (Template_Norm == 0) ? 1 : Template_Norm;
  for(i=0; i < TEMPLATE_M*TEMPLATE_N; i++){
    h_Template[i] = h_Template[i] / Template_Norm;
  }

  if (PRINT_INPUT_MATRICES == TRUE){
    printf("The Normalized Template is printed here: \n");
    Printing_Matrix(TEMPLATE_M, TEMPLATE_N, h_Template);
  }

//Pading the frame
  for (i =0; i<PAD_FRAME_M*PAD_FRAME_N; i++)
    h_Pad_Frame[i] = 0;

  for (i=0; i < FRAME_M; i++) {
    for (j=0; j < FRAME_N; j++) {
      h_Pad_Frame[i + TEMPLATE_M - 1 + (j + TEMPLATE_N - 1) * PAD_FRAME_M]=
            h_Frame[i + j * FRAME_M];
    }
  }

  if (PRINT_INPUT_MATRICES == TRUE){
    printf("The Padded Matrix is printed here \n");
    Printing_Matrix(PAD_FRAME_M, PAD_FRAME_N, h_Pad_Frame);
  }

////////////////////////////////////////////////////////////////////////////
//  GPU FUNCTIONS
////////////////////////////////////////////////////////////////////////////

//Subdividing input data across GPUs
//Get data sizes for each GPU
  for(i = 0; i < GPU_N; i++){
    if (i==0){
      r=2;
      str[i].k=0;
    }
    else{
      r=1;
      str[i].k=1;
    }
    str[i].PAD_FRAME_SIZE_PER_DEVICE   =   (FRAME_N/GPU_N   +   r*(TEMPLATE_N-1))   *
PAD_FRAME_M * sizeof(float);
    str[i].OUTCOME_SIZE_PER_DEVICE = (FRAME_M + TEMPLATE_M-1) * (FRAME_N/GPU_N
+(r-1)*(TEMPLATE_N -1))* sizeof(float);
  }

  for(i = 0; i<GPU_N; i++){
    str[i].GPU_NUMBER = GPU_N;
```

```
  }


//Assign data ranges to GPUs
  for(i = 0; i < GPU_N; i++){
    str[i].device = i;
    if (i==0)
      d=0;
    else d=1;
    str[i].h_Pad_Frame = h_Pad_Frame + i * PAD_FRAME_M * (FRAME_N/GPU_N) +
d*(TEMPLATE_N-1)*PAD_FRAME_M ;
  }


  printf("main(): waiting for GPU results...\n");

  gettimeofday(&start_kernel, NULL);
  for(i = 0; i < GPU_N; i++)
    threadID[i] = cutStartThread((CUT_THREADROUTINE)GPU_Thread, (void *)(str + i));
  cutWaitForThreads(threadID, GPU_N);
  gettimeofday(&end_kernel, NULL);

  kernel_time = (double)((end_kernel.tv_usec - start_kernel.tv_usec)/1.0e6 + end_kernel.tv_sec -
start_kernel.tv_sec);

  printf("\n Returned from GPU's... \n\n");

  double
    min_start_in = (double)(str[0].start_in.tv_usec/1.0e6 + str[0].start_in.tv_sec),
    max_end_in = (double)(str[0].end_in.tv_usec/1.0e6 + str[0].end_in.tv_sec),
    min_start_GPU =  (double)(str[0].start_GPU.tv_usec/1.0e6 + str[0].start_GPU.tv_sec),
    max_end_GPU = (double)(str[0].end_GPU.tv_usec/1.0e6 + str[0].end_GPU.tv_sec),
    min_start_out = (double)(str[0].start_out.tv_usec/1.0e6 + str[0].start_out.tv_sec),
    max_end_out = (double)(str[0].end_out.tv_usec/1.0e6 + str[0].end_out.tv_sec);


  for(i = 0; i < GPU_N; i++){
    if ((double)(str[i].start_in.tv_usec/1.0e6 + str[i].start_in.tv_sec) < min_start_in)
      min_start_in = (double)(str[i].start_in.tv_usec/1.0e6 + str[i].start_in.tv_sec);
    if((double)(str[i].end_in.tv_usec/1.0e6 + str[i].end_in.tv_sec)> max_end_in)
      max_end_in = (double)(str[i].end_in.tv_usec/1.0e6 + str[i].end_in.tv_sec);

    if ((double)(str[i].start_GPU.tv_usec/1.0e6 + str[i].start_GPU.tv_sec)< min_start_GPU)
      min_start_GPU = (double)(str[i].start_GPU.tv_usec/1.0e6 + str[i].start_GPU.tv_sec);
    if((double)(str[i].end_GPU.tv_usec/1.0e6 + str[i].end_GPU.tv_sec)> max_end_GPU)
      max_end_GPU = (double)(str[i].end_GPU.tv_usec/1.0e6 + str[i].end_GPU.tv_sec);

    if ((double)(str[i].start_out.tv_usec/1.0e6 + str[i].start_out.tv_sec)< min_start_out)
      min_start_out = (double)(str[i].start_out.tv_usec/1.0e6 + str[i].start_out.tv_sec);
    if((double)(str[i].end_out.tv_usec/1.0e6 + str[i].end_out.tv_sec)> max_end_out)
      max_end_out = (double)(str[i].end_out.tv_usec/1.0e6 + str[i].end_out.tv_sec);
  }

  in_time  = max_end_in - min_start_in;
  GPU_time = max_end_GPU - min_start_GPU;
  out_time = max_end_out - min_start_out;


  printf("Transporting DATA into GPUs time is: %f sec\n", in_time);
  printf("Prossesing DATA in GPUs time is: %f sec\n", GPU_time);
  printf("Transporting DATA out of GPUs time is: %f sec\n", out_time);
  printf("Total thread-routine execution time is: %f sec\n\n", kernel_time);
```

```
 double flop =   1.0e-9 * (FRAME_M + TEMPLATE_M - 1)*(FRAME_N + TEMPLATE_N -
1)*(5*(TEMPLATE_M*TEMPLATE_N)+5);
 double gflops =   flop / GPU_time;

 printf("\nGPU matrix_correlation GFLOPS : %f \n", gflops);

 FindMaxElement(h_OutcomeGPU,
                max_GPU,
                OUTPUT_M,
                OUTPUT_N);

 printf("Maximum Element on GPU is %3.2f and located at (%5.0f, %5.0f)\n",max_GPU[0],
max_GPU[1], max_GPU[2]);
//////////////////////////////////////////////////////////////////////////
//  CPU FUNCTIONS
//////////////////////////////////////////////////////////////////////////
 clock_t start1, stop1;
 double t1 = 0.0;

 assert((start1 = clock())!=-1);

 printf("\nCPU matrix correlation..\n");

 correlationCPU(
                h_OutcomeCPU,
                h_Pad_Frame,
                h_Template
                );

 stop1 = clock();

 t1 = (double) (stop1-start1)/CLOCKS_PER_SEC;
 printf("CPU matrix correlation time: %f msec\n", t1*1000);

 FindMaxElement(h_OutcomeCPU,
                max_CPU,
                OUTPUT_M,
                OUTPUT_N);

 printf("Maximum Element on CPU is %3.2f and located at (%5.0f, %5.0f)\n",max_CPU[0],
max_CPU[1], max_CPU[2]);


//Printing Output Data
 if (PRINT_OUTCOME_MATRICES == TRUE){
   printf("The GPU Correlation Matrix is printed here: \n");
   Printing_Matrix((FRAME_M + TEMPLATE_M -1),(FRAME_N + TEMPLATE_N -1),
h_OutcomeGPU);

   printf("The CPU correlation Matrix is printed here: \n");
   Printing_Matrix((FRAME_M + TEMPLATE_M -1),(FRAME_N + TEMPLATE_N -1),
h_OutcomeCPU);
 }

//////////////////////////////////////////////////////////////////////////
//  COMPERING THE GPU AND CPU RESULTS
//////////////////////////////////////////////////////////////////////////
 printf("\n...comparing GPU VS CPU \n");

 sum_delta = 0.0;
 max_abs_err = 0.0;
 sum_ref   = 0.0;
 for(i = 0; i < OUTPUT_M * OUTPUT_N; i++){
   sum_delta += fabs((double)h_OutcomeCPU[i] -(double) h_OutcomeGPU[i]);
```

```
    if(fabs((double)h_OutcomeCPU[i] - (double)h_OutcomeGPU[i]) > max_abs_err){
      max_abs_err = fabs((double)h_OutcomeCPU[i] - (double)h_OutcomeGPU[i]);}
    sum_ref   += fabs((double)h_OutcomeCPU[i]);
  }
  L1norm = sum_delta / sum_ref;

  printf("Max. Absolute error is %E\n",max_abs_err);
  printf("Relative error: %E\n", L1norm);

  printf((L1norm < 1e-6) ? "TEST PASSED\n\n" : "TEST FAILED\n\n");

////////////////////////////////////////////////////////////////////////////
//  SHUTTING DOWN
////////////////////////////////////////////////////////////////////////////

  printf("Shutting down...\n");

  free(h_OutcomeCPU);
  free(h_Frame);
  free(h_Pad_Frame);

  CUT_EXIT(argc, argv);
}
```

# 6. Βιβλιογραφία – Αναφορές

[1] Lisa Gottesfeld Brown, "A Survey of Image Registration Techniques", Department of Computer Science Columbia University New York (Jan 1992).

[2] Mark Pacifico and Mike Merrill, "The History of Parallel Processing", Department of Computer Science University of Maryland (Fall 1998).

[3] Rajkumar Buyya, "The Design of Paras Microkernel" ch.1 "Parallel Computing at a Glance", C-DAC India (2000).

[4] Bradley Sanford, "Integrated Graphics Solutions for Graphics-Intensive Applications", ADVANCED MICRO DEVICES INC (Sep 2002)

[5] David Jacobs, "Correlation and Convolution", Class Notes for CMSC 426 (Fall 2005).

[6] Victor Podlozhnyuk, "Image Convolution with CUDA", NVIDIA Corporation (Jun 2007).

[7] Sha'Kia Boggan and Daniel M. Pressel, "GPUs: An Emerging Platform for General-Purpose Computation", ARL, (Aug 2007).

[8] Massimiliano Fatica and David Luebke, "SUPERCOMPUTING 2007 Tutorial: High Performance Computing with CUDA", NVIDIA Corporation (Nov 2007).

[9] John Nickolls, Ian Buck, Michael Garland and Kevin Skadron, "Scalable Parallel Programming with CUDA", Queue vol. 6 no. 2 – ACM Digital Library (Mar 2008).

[10] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips, "GPU Computing", Proceedings of the IEEE Vol. 96 No. 5 (May 2008).

[11] Xiaobai Sun, Nikos Pitsianis and Paolo Bientinesi, "Fast Computation of Local Correlation Coefficients", Duke University USA, AUTH University Greece and RWTH-Aachen University Germany (Sep 2008).

[12] Shuai Che, Jiayuan Meng, Jeremy W. Sheaffer and Kevin Skadron, "A Performance Study of General Purpose Applications on Graphics Processors", The University of Virginia, Department of Computer Science (Oct 2008).

[13] Song Ho Ann, “Convolution”, (Nov 2008).

[14] Zhiyi Yang, Yating Zhu, Yong Pu, “Parallel Image Processing Based on CUDA”, Dept. Computer of Northwestern Polytechnical University Xi’ an Shaanxi China (2008).

[15] Greg Ruetsch and Brent Oster, “Getting Started with CUDA”, NVIDIA Corporation (2008).

[16] Patrick Eibl, “3D Local Correlation Coefficient Computation with CUDA” , Duke University (2009).

[17] Blaise Barney, “Introduction to Parallel Computing”, Lawrence Livermore National Laboratory (2009)

[18] NVIDIA CUDA Compute Unified Device Architecture Reference Manual version 2.0 (Jun 2008).

[19] NVIDIA CUDA Compute Unified Device Architecture Programming Guide version 2.0 (Jul 2008).