ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Παραλληλοποίηση του Sirene: Υλοποίηση της κυβικής Β-καμπυλοειδής παρεμβολής στην μονάδα επεξεργασίας γραφικών για πολλές διαστάσεις

Γιώργος Ε. Βουλαρίνος

Επιβλέπων: **Ιωάννης Κοτρώνης,** Αναπληρωτής Καθηγητής

ΑΘΗΝΑ

ΟΚΤΩΒΡΙΟΣ 2016

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Παραλληλοποίηση του Sirene: Υλοποίηση κυβικής Β-καμπυλοειδής παρεμβολής στην μονάδα επεξεργασίας γραφικών για πολλές διαστάσεις

Γιώργος Ε. Βουλαρίνος

Α.Μ.: Μ1255

ΕΠΙΒΛΕΠΩΝ: **Ιωάννης Κοτρώνης,** Αναπληρωτής Καθηγητής

ΕΞΕΤΑΣΤΙΚΗ ΕΠΙΤΡΟΠΗ **Θεοχάρης Θεοχάρης,** Καθηγητής

Οκτώβριος 2016

ΠΕΡΙΛΗΨΗ

Σκοπός αυτής της εργασίας είναι η μελέτη και υλοποίηση της Κυβικής Β-Καμπυλοειδούς Παρεμβολής για πολλές διαστάσεις χρησιμοποιώντας και την μονάδα επεξεργασίας γραφικών. Ως κίνητρο υπήρξε η μελέτη που έγινε για την παραλληλοποίηση του SIRENE, ενός προσομοιωτή ανιχνευτών φωτονίων τα οποία παράγονται από νετρόνια. Παρουσιάζονται η έρευνα αυτή που έγινε στο SIRENE, τα ευρήματά της και μια ανάλυση πάνω στην υπάρχουσα υλοποίηση του. Δίνονται ορισμένα σημεία όπου η παραλληλοποίηση μπορεί να επιτευχθεί, χωρίς όμως να υπάρξει περαιτέρω ανάλυση και υλοποίηση. Μετά ακολουθεί μια παρουσίαση για την Κυβική Β-καμπυλοειδή Παρεμβολή. Εκεί δείχνεται γιατί έχει επιλεχθεί ως μέθοδος παρεμβολής και γίνεται μια μικρή σύγκριση με άλλους παρόμοιες μεθόδους. Η υλοποίησή της μέχρι τριών διαστάσεων για μονάδα επεξεργασίας γραφικών δίνεται από υπάρχουσα εργασία και όπου χρησιμοποιούνται οι υφές. Για περισσότερες διαστάσεις, χρησιμοποιείται αυτήν των τριών διαστάσεων και για τις επιπλέον διαστάσεις υλοποιείται ο αλγόριθμος της παρεμβολής καθαρά με χρήση υπολογισμών από την μονάδα επεξεργασίας γραφικών.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Παράλληλη Επεξεργασία

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: παραλληλοποίηση, γραφική μονάδα επεξεργασίας, κυβική Β-καμπυλοειδής παρεμβολή

ABSTRACT

Purpose of this thesis is the study and implementation of Cubic B-Spline Interpolation for many dimensions with the usage of Graphical Processing Unit. As motivation for this, it was the study which was carried out for the parallelization of SIRENE, a simulator of photo detector produced by neutrons. The research on SIRENE, the findings and the analysis on the current implementation are presented. A few facts are given wherever the parallelization was feasible, without any further analyzing and implementation. Afterwards a presentation of Cubic B-Spline Interpolation follows. There it is pointed why it has been chosen as interpolation method and a small comparison with other similar methods is carried out. The implementation up to three dimensions for graphical processing unit is given by an existing project where the textures are used. For more dimensions, that one of three dimensions is used and for the additional dimensions the algorithm of interpolation is purely implemented with usage of computations by the graphical processing unit.

SUBJECT AREA: Parallel Processing

KEYWORDS: parallelization, graphical processing unit, cubic b-spline interpolation

ΕΥΧΑΡΙΣΤΙΕΣ

Θα ήθελα να ευχαριστήσω τον καθηγητή Ιωάννη Κοτρώνη για την ευκαιρία που μου δόθηκε να παραβρεθώ στην έρευνα, τους Πέτρο Γιαννακόπουλο, Μιχαήλ Γκούμα και Ιωάννη Δίπλα για την έκρυθμη συνεργασία που είχαμε, καθώς και την ομάδα του Ινστιτούτου Πυρηνικής και Σωματιδιακής Φυσικής του Δημόκριτου για την υποστήριξή όπου αυτή χρειάστηκε.

ΠΕΡΙΕΧΟΜΕΝΑ

ΚΑΤΑΛΟΓΟΣ ΕΙΚΟΝΩΝ

ΚΑΤΑΛΟΓΟΣ ΠΙΝΑΚΩΝ

ΠΡΟΛΟΓΟΣ

Η υπάρχουσα διπλωματική εργασία εκπονήθηκε κάτω από την ανάγκη μελέτης και διερεύνησης παραλληλοποίησης του προσομοιωτή ανίχνευσης νετρονίων SIRENE. Σε συνεργασία με το Ινστιτούτο Πυρηνικής και Σωματιδιακής Φυσικής, το οποίο χρησιμοποιεί το SIRENE, έγινε προσπάθεια ανάλυσης και εύρεσης λύσεων που θα βοηθήσουν στην αύξηση της επεξεργαστικής του απόδοσης με χρήση παράλληλων μονάδων επεξεργασίας γενικού σκοπού ή/και με χρήση μονάδων γραφικής επεξεργασίας. Η ερεύνα αυτή πραγματοποιήθηκε σε συνεργασία με τους Πέτρο Γιαννακόπουλο, Μιχαήλ Γκούμα και Ιωάννη Δίπλα κάτω από την επίβλεψη και συμβουλή του Ιωάννη Κοτρώνη, αναπληρωτή καθηγητή του τμήματος Πληροφορικής και Τηλεπικοινωνιών του Καποδιστριακού Πανεπιστημίου Αθηνών [1]. Από αυτή την μελέτη, εντοπίσθηκαν ο τρόπος προσομοίωσης του SIRENE και τα σημεία εκείνα του κώδικά του που ενδέχεται να αλλαχτούν με παράλληλη υλοποίηση. Ένα από αυτά τα σημεία διερεύνησης ήταν και η χρήση της μεθόδου της παρεμβολής για την εύρεση των ριζών μιας συνεχούς συνάρτησης με χρήση ενός υποσυνόλου τιμών της. Για αυτή την λειτουργία ζητήθηκε η διερεύνηση εκτέλεσής της από μονάδα επεξεργασίας γραφικών ανεξαρτήτως της διάστασης του χώρου του πεδίου ορισμού της συνάρτησης.

1. ΕΙΣΑΓΩΓΗ

1.1 Σκοπός της εργασίας

Σκοπός της εργασίας είναι να παρουσιαστεί η μελέτη που έγινε για το SIRENE και κυρίως η έρευνα και υλοποίηση για την κυβική Β-καμπυλοειδή παρεμβολή για πολλές διαστάσεις. Η παρουσίαση για το SIRENE είναι ξεκάθαρα για να δείξει και την δουλειά που έγινε εκεί, αλλά και να δείξει τα κίνητρα που οδήγησαν στην μελέτη της παρεμβολής σε μονάδα επεξεργασίας γραφικών.

Στα κεφάλαια που ακολουθούν θα παρουσιαστεί το ίδιο το SIRENE, το πώς γίνεται η προσομοίωση σε λογικό επίπεδο, αλλά και πως έχει υλοποιηθεί σε επίπεδο κώδικα. Θα δοθούν μερικές αλλαγές που προτάθηκαν ώστε να είναι δυνατή η παραλληλοποίηση του κώδικα, καθώς και μερικές λύσεις παραλληλοποίησης που προτάθηκαν. Αργότερα θα ακολουθήσει το κύριο κομμάτι της εργασίας η οποία είναι η κυβική Β-καμπυλοειδή παρεμβολή.

Θα γίνει μια μικρή ανάλυση της παρεμβολής και θα παρουσιαστεί η προσπάθεια που έγινε για την υλοποίησή της σε γραφική μονάδα επεξεργασίας (Graphic Processing Unit – GPU). Η υλοποίηση βασίστηκε σε υπάρχουσα μελέτη, αλλά η χρήση της παρεμβολής που γίνεται από το SIRENE κατέστησε την ανάγκη για υλοποίηση σε πολλές διαστάσεις.

2. ΜΕΛΕΤΗ ΤΟΥ SIRENE

2.1 Λειτουργία του Sirene

Το SIRENE αποτελεί ένας προσομοιωτής ανίχνευσης νετρονίων. Σκοπό έχει δεδομένου μιας διάταξης ανιχνευτών και κάποιων αριθμών σωματιδίων, ανάμεσα στα οποία και μιόνια, να υπολογισθεί η ενέργεια που εντοπίζεται από τα φωτόνια που παράγουν αυτά τα σωματίδια. Έτσι με τη δοκιμή διαφορετικών διατάξεων, μπορεί να βρεθεί εκείνη που μπορεί να αποδώσει καλύτερα και να χρησιμοποιηθεί σε πραγματικό επίπεδο.

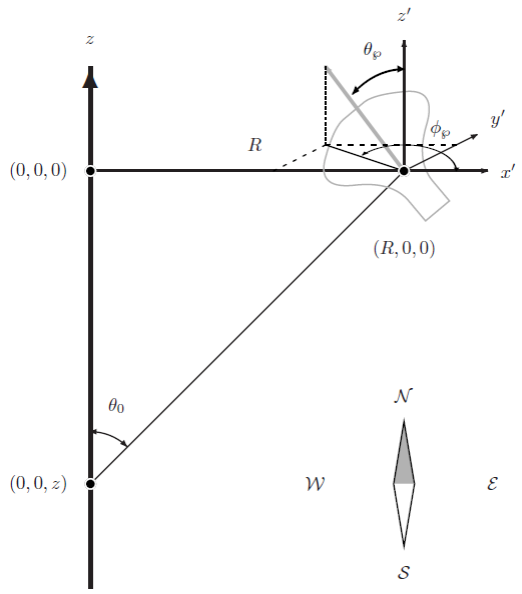
2.1.1 Φυσικό μοντέλο προσομοίωσης

Για την προσομοίωση το SIRENE ακολουθεί συγκεκριμένο μοντέλο προσομοίωσης. Οι ανιχνευτές θεωρείται ως δεδομένο ότι βρίσκονται μέσα σε νερό, στοιχείο το οποίο επηρεάζει την προσομοίωση. Ο χώρος της προσομοίωσης αποτελεί ένας κύλινδρος σταθερής διαμέτρου. Τα σωματίδια εισέρχονται από διάφορα σημεία της επιφάνειας του κυλίνδρου και τον διασχίζουν σε ευθείες τροχιές με διαφορετικές γωνίες εισόδου και διαφορετική γωνία σπιν.

Για κάθε ένα από τα διερχόμενα σωματίδια εξετάζονται, ανά χρονική μονάδα, το πλήθος των φωτονίων τα οποία παράγονται και εντοπίζονται από τους φωτοανιχνευτές. Οι φωτοανιχνευτές χωρίζονται σε ομάδες οι οποίες έχουν συγκεκριμένες γωνίες εντοπισμού των φωτονίων. Ανάλογα με την γωνία εισόδου του σωματιδίου, το σπιν του και την γωνία ανίχνευσης της ομάδας υπολογίζεται η ενέργεια που εντοπίζεται πρώτα από την ομάδα και έπειτα σε κάθε φωτοανιχνευτή ξεχωριστά.

Κάθε σωματίδιο φέρει μια συγκεκριμένη ενέργεια με την είσοδο του στον κύλινδρο. Με βάση αυτήν, αλλά και τον χρόνο παραμονής του σωματιδίου στον χώρο, υπολογίζονται τα φωτόνια τα οποία παράγονται. Τα φωτόνια ανά στιγμή του χρόνου έχουν και συγκεκριμένες κατευθύνσεις στις οποίες διαχέονται. Αυτές οι κατευθύνσεις μπορούν να περιοριστούν σε μια κωνική διατομή του χώρου με κορυφή το σημείο το οποίο βρίσκεται εκείνη τη χρονική στιγμή το σωματίδιο. Οι ομάδες των φωτοανιχνευτών και οι φωτοανιχνευτές εξετάζονται μόνο αν βρίσκονται μέσα σε αυτή την διατομή. Επίσης ανά ομάδα φωτοανιχνευτών υπάρχει μια πιθανότητα για το αν θα ανιχνευθούν ή όχι τα φωτόνια που προσκρούουν πάνω τους.

Η ανίχνευση των φωτονίων δεν περιορίζεται μόνο στα άμεσα φωτόνια τα οποία βρίσκονται μέσα στην αναμενόμενη κατεύθυνση, αλλά λαμβάνονται υπ’ όψη και τα φωτόνια που αλλάζουν κατεύθυνση λόγω της διάχυσης. Επίσης, εκτός από τα διαφορετικά σωματίδια τα οποία διασχίζουν το χώρο, φωτόνια παράγονται και από ηλεκτρομαγνητικούς καταιονισμούς. Και σε αυτές τις περιπτώσεις εξετάζονται τα φωτόνια που ανιχνεύονται είτε από άμεσο είτε από έμμεσο φως.



Εικόνα 1: Αναπαράσταση πρόσπτωσης ενός φωτονίου σε έναν φωτοανιχνευτή

2.2 Σειριακή εκτέλεση

Αναγκαία για την βελτιστοποίηση του κώδικα ήταν η ανάλυση της σειριακής εκτέλεσης. Μέσα από αυτήν την μπορούν να εντοπισθούν οι εξαρτήσεις που εμφανίζονται μεταξύ των υπολογισμών, αλλά και τα σημεία εκείνα τα οποία μπορεί να γίνει παρέμβαση για παραλληλοποίηση. Μέσα από αυτήν προέκυψαν τα εξής στάδια του κώδικα:

1. Η προσομοίωση χωρίζεται σε ανεξάρτητα «Γεγονότα» τα οποία περιλαμβάνουν πληροφορίες για διαφορετικά σωματίδια.
2. Για κάθε ένα «Γεγονός» εξετάζονται οι τροχιές κίνησης των σωματιδίων που περιλαμβάνουν.
3. Ελέγχεται για κάθε σωματίδιο που μελετάται αν είναι μιόνιο ή άλλου τύπου σωματίδιο καθώς προκύπτουν διαφορετικοί υπολογισμοί για κάθε περίπτωση.
4. Στην περίπτωση ενός μιονίου, εφόσον αυτό διέρχεται εντός του κυλινδρικού χώρου εκτελούνται τα παρακάτω:
    a. Λαμβάνονται υπ’ όψη οι επιπτώσεις που έχει αν το σωματίδιο διέρχεται από το έδαφος και από το νερό.
    b. Υπολογίζεται η ενέργεια που θα έχουν οι καταιονισμοί με βάση την ενέργεια του σωματιδίου.
    c. Γίνονται οι υπολογισμοί όσο το σωματίδιο διαθέτει την κατάλληλη ενέργεια και βρίσκεται μέσα στον κυλινδρικό χώρο.
    d. Εφόσον οι κατευθύνσεις των φωτονίων τα οποία παράγονται προσπίπτουν με την ομάδα φωτοανιχνευτών, τότε μόνο γίνεται υπολογισμούς σε αυτή.
    e. Υπολογίζεται ο συνολικός αριθμός φωτονίων που θα διέλθουν από την ομάδα.
    f. Για κάθε φωτοανιχευτή υπολογίζονται ο αριθμός των φωτονίων που τελικά θα ανιχνευθούν και πως αυτά κατανέμονται με την πάροδο του χρόνου.

g. Μετά γίνονται οι υπολογισμοί και για τα φωτόνια που παράγονται από καταιονισμούς όπως στα βήματα δ-στ.

2.3 Σημεία παραλληλίας

Μετά από την ανάλυση του υπάρχοντος σειριακού κώδικα, έγινε μελέτη ώστε να βρεθούν τα σημεία εκείνα τα οποία με διάφορες μεθόδους μπορεί να υποστηριχθεί παραλληλία. Οι βρόγχοι είναι σημεία παραλληλίας, αλλά χρειάζεται να εξετασθούν για τυχόν σημεία εξάρτησης που μπορεί να υπάρχουν μεταξύ των επαναλήψεων. Τέτοιες εξαρτήσεις μπορούν να εμποδίσουν οποιαδήποτε προσπάθεια παραλληλοποίησης ή να την περιορίσουν εξαιτίας της ανάγκης συγχρονισμού των παράλληλων διεργασιών.

Οι πιθανοί βρόγχοι στους οποίους θα μπορούσε να χρησιμοποιηθεί παραλληλία είναι αυτοί των γεγονότων, των σωματιδίων, των ομάδων των φωτοανιχνευτών και των φωτοανιχνευτών σε μια ομάδα. Η παραλληλία ως προς τα γεγονότα θεωρήθηκε ότι δεν είχε νόημα να γίνει μέσα από τον κώδικα, καθώς αυτά είναι εντελώς ανεξάρτητα μεταξύ τους και μπορεί να επιτευχθεί με εκτέλεση παράλληλων ανεξάρτητων διεργασιών. Το ίδιο συμβαίνει και για τα σωματίδια καθώς δεν υπάρχει οποιαδήποτε αλληλεπίδραση ή εξάρτηση μεταξύ τους. Έτσι μόνο οι διαδικασίες που αφορούν τους υπολογισμούς πάνω στους φωτοανιχνευτές και τις ομάδες τους ανά σωματίδιο θεωρήθηκε άξιο περεταίρω διερεύνησης.

Για κάθε σωματίδιο έχουμε υπολογισμούς που αφορούν τον υπολογισμό για φωτόνια τα οποία παράγονται από τα μιόνια άμεσου και έμμεσου φωτός και υπολογισμούς από τους καταιονισμούς. Αυτοί οι υπολογισμοί για τα μιόνια και τους καταιονισμούς παρουσιάζουν ομοιότητες και δεν έχουν εξαρτήσεις οπότε μπορούν να εκτελεστούν με τις ίδιες μεθόδους παραλληλίας και ανεξάρτητα.

Επίσης παρατηρήθηκε είναι η ότι υπάρχει υπολογισμός ανά βήμα καθώς το σωματίδιο διέρχεται μέσα από τον κύλινδρο. Για την επίτευξη αυτού χρησιμοποιείται ένας βρόγχος που εκτελείται εφόσον υπάρχει η κατάλληλη ενέργεια και το σωματίδιο βρίσκεται εντός του κυλίνδρου. Σε κάθε επανάληψη του βρόγχου υπολογίζεται με τυχαίο τρόπο το μήκος που θα εξεταστεί, ο ανάλογος χρόνος μετάβασης και η ανάλογη απώλεια ενέργειας του σωματιδίου. Αυτοί οι υπολογισμοί μπορούν να γίνουν νωρίτερα, έτσι ώστε εν τέλει ο βρόγχος να μπορεί να σπάσει και η εκτέλεση για κάθε βήμα να γίνει ανεξάρτητο. Μετά από αυτή την τροποποίηση πλέον η παραλληλία μπορεί να γίνει πλέον ανά βήμα του σωματιδίου.

Εν τέλει αυτό που θα υπολογίζει ένα νήμα μας είναι οι χρόνοι εντοπισμού και ο αριθμός των φωτονίων που ανιχνεύει ένας φωτοανιχνευτής, για μια συγκεκριμένη μετάβαση του σωματιδίου, μια συγκεκριμένη απώλεια ενέργειας και θα προκύπτει είτε από άμεσο είτε από έμμεσο φως.

3. ΚΥΒΙΚΗ Β-ΚΑΜΠΥΛΟΕΙΔΗ ΠΑΡΕΜΒΟΛΗ

3.1 Ορισμός της παρεμβολής

Η παρεμβολή είναι μια μέθοδος που χρησιμοποιείται προκειμένου να κατασκευαστούν νέα σημεία πάνω σε ένα χώρο έχοντας ως πληροφορία ορισμένα στοιχεία του χώρου. Έτσι για παράδειγμα γνωρίζοντας τις ρίζες μιας συνάρτησης για ορισμένα σημεία της, μπορούν να υπολογισθούν οι αντίστοιχες ρίζες για τις υπόλοιπες τιμές της. Χρησιμοποιείται ευρέως στην επιστήμη καθώς επιτρέπει υπολογισμούς γνωρίζοντας ένα δείγμα δεδομένων και όχι την συνάρτηση καθ' αυτή.

3.2 Χρήση της παρεμβολής από το SIRENE

Στο SIRENE η παρεμβολή χρησιμοποιείται για τις συναρτήσεις που έχουν να κάνουν με τον υπολογισμό του αριθμού των φωτονίων που ανιχνεύονται καθώς και των αντίστοιχων χρόνων. Έτσι αντί για να υπολογίζονται πολύπλοκες συναρτήσεις για κάθε σύνολο διαφορετικών παραμέτρων, χρησιμοποιείται ένα δείγμα έτοιμων σημείων πάνω στο χώρο. Με χρήση της παρεμβολής υπολογίζονται όσες ρίζες δεν είναι μέσα στα γνωστά σημεία.

Το δείγμα με τα δεδομένα των συνόλων τιμών – ριζών έχουν προϋπολογιστεί και δίνονται στο SIRENE ως είσοδο κατά την εκκίνηση. Οι τιμές είναι μέσα σε ένα συγκεκριμένο πεδίο ορισμού και για την ανεύρεση οποιαδήποτε τιμής από την παρεμβολή χρησιμοποιούνται μετασχηματισμοί σε και από αυτό το πεδίο ορισμού. Επίσης επειδή αφορούν συναρτήσεις που έχουν πολλές παραμέτρους, η παρεμβολή εφαρμόζεται σε πολυδιάστατους χώρους. Κάθε διάσταση του χώρου αντιπροσωπεύεται από ένα πίνακα. Για κάθε γνωστή τιμή της διάστασης αντιστοιχεί ένας άλλος πίνακας που αντιπροσωπεύει την επόμενη διάσταση.

Για τον υπολογισμό της ρίζας μιας συνάρτησης, γίνεται πρώτα μια αναζήτηση στις αλλεπάλληλες εμφωλεύσεις με βάση τις τιμές των δοθέντων παραμέτρων μέχρι να φτάνει στον δισδιάστατο χώρο. Εκεί μπορεί να γίνει εύκολα με χρήση μιας παρεμβολής ο υπολογισμού της ρίζας του δοθέντος σημείου της διάστασης παρεμβολής. Αυτό εφαρμόζεται σ' ένα σύνολο δυσδιάστατων χώρων και πάνω στις ρίζες οι οποίες βρίσκονται, εφαρμόζεται αντίστοιχα η παρεμβολή για την παραπάνω διάσταση. Έτσι βρίσκεται η ρίζα η οποία αναζητείται στον τρισδιάστατο χώρο. Αυτή η μέθοδος υπολογισμού συνεχίζεται μέχρι να ληφθούν υπόψιν όλες οι διαστάσεις.

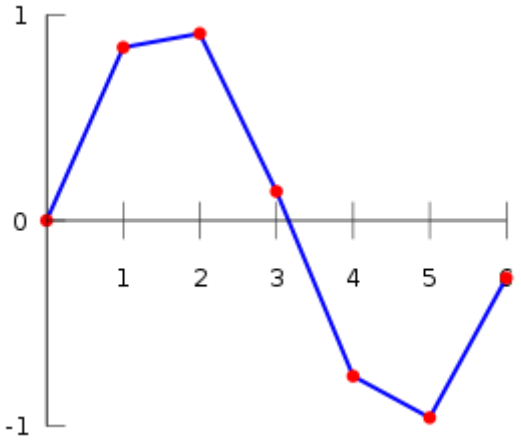
Για κάθε όμως αναζήτηση της ρίζας μιας τιμής σε μια διάσταση, πρέπει πρώτα να βρεθούν οι πλησιέστερες γνωστές τιμές, σε κάθε μια απ' αυτές να υπολογισθούν αναδρομικά η ρίζα τους όπως έχει περιγραφθεί στην προηγούμενη παράγραφο και έπειτα να υπολογισθεί η ρίζα που αφορά τη συγκεκριμένη διάσταση με τη χρήση της παρεμβολής. Οι τιμές – ρίζες αποθηκεύονται σε ένα ισορροπημένο δυαδικό δέντρο, όπως δηλαδή γίνεται η υλοποίηση από την STL% της C++. Άρα η αναζήτηση των πλησιέστερων τιμών γίνεται με χρήση δυαδικής αναζήτησης. Οι υπολογισμοί επίσης γίνονται εξ' ολοκλήρου από την κεντρική μονάδα επεξεργασίας χωρίς κανένα είδος παραλληλίας.

Οι υπολογισμοί αυτοί έχουν μεγάλο υπολογιστικό κόστος καθώς γίνονται ξανά και ξανά για κάθε σωματίδιο, βήμα σωματιδίου και φωτοανιχνευτή. Αποτελούν μαζί με κάποιους υπολογισμούς για ανεύρεση τυχαίων αριθμών, το μεγαλύτερο κομμάτι υπολογισμών που γίνεται από το SIRENE. Για αυτό τον σκοπό θα γίνει μια προσπάθεια να βρεθεί μια γρήγορη μέθοδος υπολογισμού της κυβικής παρεμβολής με τη χρήση μονάδας επεξεργασίας γραφικών.

Στο SIRENE έχει υλοποιηθεί η κυβική καμπυλοειδής παρεμβολή, η οποία είναι ένας τύπος παρεμβολής με μικρότερα σφάλματα στον υπολογισμό των τιμών αυτών. Επίσης δεν εμφανίζει το φαινόμενο του Ρούνγκε, δηλαδή εμφάνιση μεγάλου σφάλματος σε περιπτώσεις που χρησιμοποιούνται πολλά σημεία για την εύρεση των τιμών. Ως μέθοδος όμως που θα χρησιμοποιηθεί για πιο γρήγορους υπολογισμούς είναι αυτή της κυβικής Β-καμπυλοειδής παρεμβολής.

3.3 Υλοποίηση της κυβικής Β-καμπυλοειδής παρεμβολής για μονάδα επεξεργασίας γραφικών

Η εξαγωγή τιμών με βάση ορισμένο δείγμα δεδομένων δεν είναι κάτι καινούργιο για στο χώρο της επεξεργασίας των γραφικών. Χρησιμοποιείται ευρεία εδώ και πολλά χρόνια για την οπτικοποίηση των υφών. Ακριβώς εξαιτίας διαφορετικών αναλύσεων και διαφορετικών σημείων θέασης ενός τρισδιάστατου γραφικού αντικειμένου, δεν ήταν δυνατή η χρήση ένα προς ένα απεικόνισης των υφών τους. Ιδιαίτερα σε σημεία θέασης τα οποία προσεγγίζουν πολύ το αντικείμενο απαιτούνται υφές μεγάλης οπτικής ανάλυσης. Για το σκοπό αυτό χρησιμοποιούνται υφές με συγκεκριμένη ανάλυση και αν για οποιαδήποτε άλλο λόγω χρειαστεί να γίνει χρήση μεγαλύτερης ανάλυσης, τότε οι τιμές οι οποίες δεν υπάρχουν υπολογίζονται με διάφορες προσεγγιστικές μεθόδους. Η συχνή χρήση και το υπολογιστικό κόστος αυτών, συνέβαλε στην δημιουργία ειδικού υλικού για τον γρήγορο υπολογισμό τους. Έτσι για πολλά τέτοια σημεία στον τρισδιάστατο χώρο μπορούν ταυτόχρονα να υπολογισθούν οι αντίστοιχες τιμές.



Εικόνα 2: Παράδειγμα ενός σετ τιμών δεδομένων και των αντίστοιχων γραμμικών παρεμβολών τους

Η μέθοδος όμως που χρησιμοποιούν οι κάρτες γραφικών, δεν είναι άλλη από αυτή της γραμμικής παρεμβολής. Η γραμμική παρεμβολή δεν είναι τίποτε άλλο από τον εντοπισμό των δυο πλησιέστερων γνωστών σημείων και ο υπολογισμός της εν ζητούμενης τιμής με βάση την ευθύγραμμο τμήμα που ορίζουν αυτά τα δυο σημεία. Η γραμμική παρεμβολή είναι υποκατηγορία μια μεγαλύτερης οικογένειας, των πολυωνυμικών παρεμβολών. Είναι ιδιαίτερα απλή στη λογική και στην υλοποίηση, αλλά δεν είναι τόσο ακριβής ώστε να αναπαραστήσει πολύπλοκες συναρτήσεις που εμφανίζουν μεγάλες μεταβολές ή καμπύλες στις γραφικές αναπαραστάσεις τους.

Μια καλύτερη μέθοδος που υλοποιήθηκε πάνω στις κάρτες γραφικών, είναι η μέθοδος της κυβικής Β-καμπυλοειδής παρεμβολής. Αυτή η μέθοδος προσφέρει μεγαλύτερη ακρίβεια σε σχέση με τη γραμμική παρεμβολή, αλλά μπορεί να υλοποιηθεί χρησιμοποιώντας την. Έτσι ενώ η γραμμική παρεμβολή είναι αρκετά πιο γρήγορη στις κάρτες γραφικών αφού υπάρχει έτοιμο υλικό, η κυβική Β-καμπυλοειδής παρεμβολή

μπορεί να χρησιμοποιήσει αυτό το υλικό προκειμένου να βγάλει πιο ακριβή αποτελέσματα με εύλογο κόστος. Η ιδέα αυτή αναπτύχθηκε από τους Daniel Ruijters, Bart M. ter Haar Romeny και Paul Suetens στην εργασία τους με τίτλο Efficient GPU-Based Texture Interpolation using Uniform B-Splines [2]. Μέσα σε αυτήν παρουσιάζουν αναλυτικά τον αλγόριθμο που ακολούθησαν προκειμένου να κάνουν την υλοποίηση. Η ίδια η υλοποίηση χρησιμοποιεί το υλικό που υπάρχει στις κάρτες γραφικών για την επεξεργασία υφών, αποφεύγοντας ακόμα και τη χρήση συνθηκών ή την αναζήτηση σε πίνακα προκειμένου να υπολογισθούν οι τιμές που δεν υπάρχουν σύμφωνα με το υπάρχον δείγμα τιμών. Παρακάτω ακολουθεί μια μικρή παρουσίαση στην παρεμβολή ώστε να γίνει αντιληπτός ο τρόπος υλοποίησης.

Η Β-καμπυλοειδής παρεμβολή ορίζεται από τα παρακάτω:

- Ως Β-καμπυλοειδή βάση μηδενικού βαθμού ($n=0$) ορίζεται η συνάρτηση γνωστή και ως συνάρτηση «κουτιού» (box function):

$$\beta_0(x)=\begin{cases}1, |x|<\frac{1}{2}\\ \frac{1}{2}, |x|=\frac{1}{2}\\ 0, |x|>\frac{1}{2}\end{cases}$$

- Κάθε Β-καμπυλοειδή βάση βαθμού $n\geq 1$ ορίζεται από την παρακάτω συνέλιξη:

$$\beta_n(x)=\beta_0(x)*\beta_{n-1}(x)$$

- Ο τύπος υπολογισμού της τιμής ενός δοθέντος σημείου $x\in\mathbf{R}$ ορίζεται από το άθροισμα των ολισθημένων κεντρικών Β-καμπυλοειδών β_n, σταθμισμένοι από τους Β-καμπυλοειδείς συντελεστές $c(k)$:

$$s_n(x)=\sum_{k\in\mathbf{Z}} c(k)\beta_n(x-k)$$

Από τα παραπάνω για την κυβική Β-καμπυλοειδή παρεμβολή ($n=3$) μπορεί να ορισθεί η Β-καμπυλοειδή βάση ως:

$$\beta_3(x)=\begin{cases}\frac{2}{3}-\frac{1}{2}|x|^2(2-|x|), |x|<1\\ \frac{1}{6}(2-|x|)^3, 1\leq|x|<2\\ 0, |x|\geq 2\end{cases}$$

Ενώ ο τύπος υπολογισμού των τιμών για μια τιμή $x=i+a$ όπου $i\in\mathbf{Z}$ και $x,a\in\mathbf{R}:0\leq\alpha<1$ γίνεται:

$$s_3(i+a)=\sum_{k\in\mathbf{Z}} c(k)\beta_3(i+a-k)=\sum_{k=i-1}^{i+2} c(k)\beta_3(i+a-k)=$$
$$=c(i-1)\beta_3(a+1)+c(i)\beta_n(a)+c(i+1)\beta_n(a-1)+c(i+2)\beta_n(a-2)$$

Εάν ορίσουμε επίσης ότι:

$$w_0(a)=\beta_3(a+1)=\beta_3(-a-1)$$

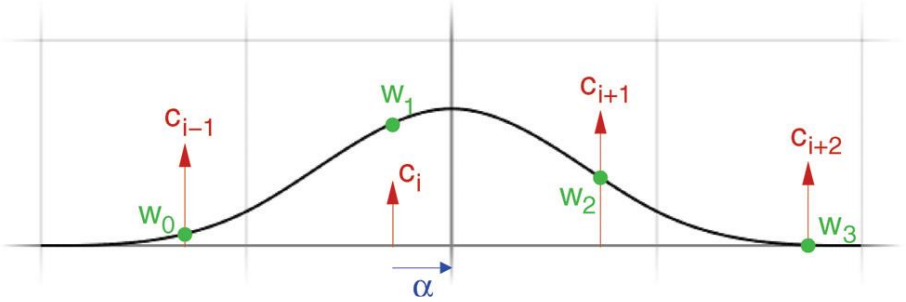
$$w_1(a)=\beta_3(a)=\beta_3(-a)$$

$$w_2(a)=\beta_3(a-1)=\beta_3(1-a)$$

$$w_3(a)=\beta_3(a-2)=\beta_3(2-a)$$

Τότε έχουμε:

$$s_3(i+a)=w_0(a)c(i-1)+w_1(a)c(i)+w_2(a)c(i+1)+w_3(a)c(i+2)$$



Εικόνα 3: Παράδειγμα δείγμα τιμών και αντίστοιχων βαρών σε μια τρίτου βαθμού κυβική Β-καμπυλοειδή παρεμβολή

Η Β-καμπυλοειδή παρεμβολή μπορεί να υπολογισθεί με χρήση της γραμμικής παρεμβολής με βάση τις παρακάτω παρατηρήσεις. Μια γραμμική παρεμβολή δυο διαδοχικών σημείων μπορεί να γραφτεί ως εξής για μια τιμή $x=i+a$ όπου $i\in\mathbf{Z}$ και $x,\alpha\in\mathbf{R}:0\le\alpha<1$:

$$s_1(i+\alpha)=(1-\alpha)s_0(i)+\alpha s_0(i+1)$$

Με βάση αυτό το άθροισμα δυο σταθμευμένων διαδοχικών τιμών μπορεί να είναι:

$$as_0(i)+bs_0(i+1)=(a+b)\left(\frac{a}{a+b}s_0(i)+\frac{b}{a+b}s_0(i+1)\right)=(a+b)\left(\left(1-\frac{b}{a+b}\right)s_0(i)+\frac{b}{a+b}s_0(i+1)\right)=$$

$$=(a+b)s_1\left(i+\frac{b}{a+b}\right)$$

Άρα με βάση αυτό η κυβική Β-καμπυλοειδή παρεμβολή γίνεται:

$$s_3(i+\alpha)=(w_0(\alpha)+w_1(\alpha))c_1\left(i-1+\frac{w_1(\alpha)}{w_0(\alpha)+w_1(\alpha)}\right)+(w_2(\alpha)+w_3(\alpha))c_1\left(i+1+\frac{w_3(\alpha)}{w_2(\alpha)+w_1(\alpha)}\right)$$

Όπου η c_1 είναι γραμμική παρεμβολή. Επίσης εάν ορίσουμε τα εξής:

$$g_0(\alpha)=w_0(\alpha)+w_1(\alpha)$$

$$g_1(\alpha)=w_2(\alpha)+w_3(\alpha)$$

$$h_0(\alpha)=\frac{w_1(\alpha)}{g_0(\alpha)}-1$$

$$h_1(\alpha)=\frac{w_3(\alpha)}{g_1(\alpha)}+1$$

Τότε μπορούμε να γράψουμε τον παραπάνω τύπο:

$$s_3(i+\alpha) = g_0(\alpha)c_1(i+h_0(\alpha)) + g_1(\alpha)c_1(i+h_1(\alpha))$$

Με βάση αυτόν τον τελευταίο τύπο γίνεται και η υλοποίηση. Οι δυο γραμμικές παρεμβολές μπορούν και υλοποιούνται από το ίδιο το υλικό που υπάρχει στην μονάδα επεξεργασίας γραφικών επιταχύνοντας την διαδικασία.

Η κυβική Β-καμπυλοειδής παρεμβολή όμως παρουσιάζει το φαινόμενο ότι οι τιμές που υπολογίζονται για τα ίδια τα δοθέντα σημεία, είναι προσεγγιστικές. Έτσι ενώ υπάρχει μια πιο ομαλή και ακριβής υπολογισμός των τιμών γενικά, δεν ισχύει τόσο για το ίδιο το δείγμα των σημείων. Λίγα χρόνια μετά οι Daniel Ruijters και Philippe Thevenaz παρουσιάζουν μια επιπλέον εργασία η οποία εφαρμόζει ένα προ-φίλτρο πάνω στα δείγματα με στόχο να διορθώσει αυτή την αδυναμία της Β-καμπυλοειδούς κυβικής παρεμβολής [3]. Αυτό το φίλτρο τροποποιεί τα δεδομένα ώστε να γίνουν συντελεστές της παρεμβολής. Παρουσίαση του φίλτρου δεν θα γίνει καθώς δεν έγινε κάποια τροποποίηση πάνω σε αυτό, ώστε να χρειασθεί κάποια παραπάνω αναφορά.

3.4 Κυβική Β-καμπυλοειδή παρεμβολή σε πολλές διαστάσεις

Η παρεμβολή μπορεί αντιστοίχως να χρησιμοποιηθεί και σε χώρους με περισσότερες από μια διαστάσεις. Αρκεί να εφαρμοστεί διαδοχικά για κάθε μια διάσταση η παρεμβολή πάνω στα γνωστά σημεία, καθώς οι τιμές οι οποίες δίνονται είναι ανεξάρτητες. Όσο όμως αυξάνονται οι διαστάσεις, διπλασιάζεται και το πλήθος των σημείων που μετέχουν στην παρεμβολή. Έτσι για παράδειγμα στον δυσδιάστατο χώρο θα πρέπει να εφαρμοστεί η παρεμβολή πρώτα στη μια διάσταση, τα αποτελέσματα αυτής να χρησιμοποιηθούν για την εφαρμογή της στην δεύτερης και συνολικά να χρησιμοποιηθούν οκτώ γνωστά σημεία. Έτσι θα έχουμε τα εξής για μια τιμή $x = (i+a, j+b)$ όπου $i, j \in \mathbf{Z}$ και $a,b \in \mathbf{R} : 0 \le a < 1, 0 \le b < 1$:

$$w_0(a) = \beta(1+a),\ w_1(a) = \beta(a),\ w_2(a) = \beta(a-1),\ w_3(a) = \beta(a-2)$$

$$w_0(b) = \beta(1+b),\ w_1(b) = \beta(b),\ w_2(b) = \beta(b-1),\ w_3(b) = \beta(b-2)$$

$$g_0(a) = w_0(a) + w_1(a),\ g_1(a) = w_2(a) + w_3(a)$$

$$g_0(b) = w_0(b) + w_1(b),\ g_1(b) = w_2(b) + w_3(b)$$

$$h_0(a) = \frac{w_1(a)}{g_0(a)} - 1,\ h_1(a) = \frac{w_3(a)}{g_1(a)} + 1$$

$$h_0(b) = \frac{w_1(b)}{g_0(b)} - 1,\ h_1(b) = \frac{w_3(b)}{g_1(b)} + 1$$

$$s_3^{j+h_0(b)}(i+a) = g_0(a)c_1((i+h_0(a),\ j+h_0(b))) + g_1(a)c_1((i+h_1(a),\ j+h_0(b)))$$

$$s_3^{j+h_1(b)}(i+a) = g_0(a)c_1((i+h_0(a),\ j+h_1(b))) + g_1(a)c_1((i+h_0(a),\ j+h_1(b)))$$

$$s_3((i+a,\ j+b)) = g_0(b)s_3^{j+h_0(b)}(i+a) + g_1(b)s_3^{j+h_1(b)}(i+a)$$

Γενικά για ένα σημείο $x=\{x_0,x_1,\ldots,x_{n-1}\}$ με $n>0$ και $x_k=i_k+\alpha_k$ όπου $i_k\in\mathbf{Z},\alpha_k\in\mathbf{R}:0\le a_k<1$ έχουμε:

$$s_3(x)=g_0(\alpha_{n-1})s_3^{i_{n-1}+h_0(x_{n-1})}(\{x_0,x_1,\ldots,x_{n-2}\})+g_1(\alpha_{n-1})s_3^{i_{n-1}+h_1(x_{n-1})}(\{x_0,x_1,\ldots,x_{n-2}\})$$

3.4.1 Υλοποίηση

Η γενίκευση που βρέθηκε, χρησιμοποιήθηκε ώστε να υλοποιηθεί ο κώδικας για την εφαρμογή της κυβικής B-καμπυλοειδής παρεμβολής σε πολλές διαστάσεις. Παρακάτω ακολουθεί ο κώδικας:

```
// Inline calculation of the bspline convolution weights, without conditional statements
template <typename key_type>
inline __host__ __device__ void bspline_weights(key_type fraction, key_type& w0,
key_type& w1, key_type& w2, key_type& w3)
{
	const key_type one_frac = 1.0f - fraction;
	const key_type squared = fraction * fraction;
	const key_type one_sqd = one_frac * one_frac;

	w0 = 1.0f / 6.0f * one_sqd * one_frac;
	w1 = 2.0f / 3.0f - 0.5f * squared * (2.0f - fraction);
	w2 = 2.0f / 3.0f - 0.5f * one_sqd * (2.0f - one_frac);
	w3 = 1.0f / 6.0f * squared * fraction;
};

template <unsigned int N>
__global__ void weights_and_mask(unsigned int idx, float_key<N>* coords, unsigned int*
mask, unsigned int* submask, float* weights, unsigned int size, unsigned int size_shift,
unsigned int max_index)
{
	int tid = threadIdx.x + blockDim.x * blockIdx.x;
	// One coordination per core
	if (tid < size)
	{
		// Check the relevant mask on one position or four depending the on the size
of results
		int i = tid << size_shift;
		if ((!size_shift && mask[i] == idx) || (size_shift && (mask[i] == idx ||
mask[i + 1] == idx || mask[i + 2] == idx || mask[i + 3] == idx)))
		{
			int j = tid * 4;
			// Transform the coordinate from [0,extent] to [-0.5, extent-0.5]
			const float last_coord = coords[tid][N - 1] - 0.5f;
			int index = static_cast<int>(floor(last_coord));

			// Calculate weights
			bspline_weights<float>(last_coord - static_cast<float>(index),
weights[j], weights[j + 1], weights[j + 2], weights[j + 3]);

			// Limit index into available values
			index = clamp(index, 1, static_cast<int>(max_index) - 3);

			// Give to the submask the necessary index positions
			submask[j] = index - 1;
			submask[j + 1] = index;
			submask[j + 2] = index + 1;
			submask[j + 3] = index + 2;
		}
	}
```

```
}

__global__ void dot_product(unsigned int idx, float* results, unsigned int* mask, float*
subresults, float* weights, unsigned int size, unsigned int size_shift)
{
	int tid = threadIdx.x + blockDim.x * blockIdx.x;
	// One coordination per core
	if (tid < size)
	{
		// Check the relevant mask on one position or four depending the on the size
of results
		int i = tid << size_shift;
		if ((!size_shift && mask[i] == idx) || (size_shift && (mask[i] == idx ||
mask[++i] == idx || mask[++i] == idx || mask[++i] == idx)))
		{
			int j = tid * 4;
			results[i] = weights[j] * subresults[j++] + weights[j] * subresults[j++]
+ weights[j] * subresults[j++] + weights[j] * subresults[j];
		}
	}
}

// Implementation of Cubic B-Spline Interpolation for 1D
template <unsigned int N, typename value_type, typename return_type>
__global__ void cubicTex1D(unsigned int idx, cudaTextureObject_t tex, float_key<N>*
coords, unsigned int* mask, return_type* results, unsigned int size, unsigned int
size_shift)
{
	int tid = threadIdx.x + blockDim.x * blockIdx.x;
	// One coordination per core
	if (tid < size)
	{
		// Check the relevant mask on one position or four depending the on the size
of results
		int i = tid << size_shift;
		if ((!size_shift && mask[i] == idx) || (size_shift && (mask[i] == idx ||
mask[++i] == idx || mask[++i] == idx || mask[++i] == idx)))
		{
			float coord = coords[tid][0];

			// Transform the coordinate from [0,extent] to [-0.5, extent-0.5]
			const float coord_grid = coord - 0.5;
			const float index = floor(coord_grid);
			const float fraction = coord_grid - index;
			float w0, w1, w2, w3;
			bspline_weights<float>(fraction, w0, w1, w2, w3);

			const float g0 = w0 + w1;
			const float g1 = w2 + w3;
			// h0 = w1/g0 - 1, move from [-0.5, extent-0.5] to [0, extent]
			const float h0 = (w1 / g0) - 0.5f + index;
			// h1 = w3/g1 + 1, move from [-0.5, extent-0.5] to [0, extent]
			const float h1 = (w3 / g1) + 1.5f + index;

			// Fetch the two linear interpolations
			return_type tex0 = tex1D<return_type>(tex, h0);
			return_type tex1 = tex1D<return_type>(tex, h1);

			// Weigh along the x-direction
			results[i] = (g0 * tex0 + g1 * tex1);
		}
	}
};
```

```
// Kernel of calling Prefilter for 1D
template <typename value_type>
__global__ void prefilter1D(value_type* coeffs, uint DataLength, int step)
{
	ConvertToInterpolationCoefficients<value_type>(coeffs, DataLength, step);
}

// Implementation of Cubic B-Spline Interpolation for 2D
template <unsigned int N, typename value_type, typename return_type>
__global__ void cubicTex2D(unsigned int idx, cudaTextureObject_t tex, float_key<N>*
coords, unsigned int* mask, return_type* results, unsigned int size, unsigned int
size_shift)
{
	int tid = threadIdx.x + blockDim.x * blockIdx.x;
	// One coordination per core
	if (tid < size)
	{
		// Check the relevant mask on one position or four depending the on the size
of results
		int i = tid << size_shift;
		if ((!size_shift && mask[i] == idx) || (size_shift && (mask[i] == idx ||
mask[++i] == idx || mask[++i] == idx || mask[++i] == idx)))
		{
			float2 coord = make_float2(coords[tid][0], coords[tid][1]);

			// Transform the coordinate from [0,extent] to [-0.5, extent-0.5]
			const float2 coord_grid = coord - 0.5f;
			const float2 index = floor(coord_grid);
			const float2 fraction = coord_grid - index;
			float2 w0, w1, w2, w3;
			bspline_weights<float2>(fraction, w0, w1, w2, w3);

			const float2 g0 = w0 + w1;
			const float2 g1 = w2 + w3;
			//h0 = w1/g0 - 1, move from [-0.5, extent-0.5] to [0, extent]
			const float2 h0 = (w1 / g0) - 0.5f + index;
			//h1 = w3/g1 + 1, move from [-0.5, extent-0.5] to [0, extent]
			const float2 h1 = (w3 / g1) + 1.5f + index;

			// Fetch the four linear interpolations
			return_type tex00 = tex2D<return_type>(tex, h0.x, h0.y);
			return_type tex10 = tex2D<return_type>(tex, h1.x, h0.y);
			return_type tex01 = tex2D<return_type>(tex, h0.x, h1.y);
			return_type tex11 = tex2D<return_type>(tex, h1.x, h1.y);

			// Weigh along the y-direction
			tex00 = g0.y * tex00 + g1.y * tex01;
			tex10 = g0.y * tex10 + g1.y * tex11;

			// Weigh along the x-direction
			results[i] = (g0.x * tex00 + g1.x * tex10);
		}
	}
};

// Implementation of Cubic B-Spline Interpolation for 3D
template <unsigned int N, typename value_type, typename return_type>
__global__ void cubicTex3D(unsigned int idx, cudaTextureObject_t tex, float_key<N>*
coords, unsigned int* mask, return_type* results, unsigned int size, unsigned int
size_shift)
{
	int tid = threadIdx.x + blockDim.x * blockIdx.x;
```

```
	// One coordination per core
	if (tid < size)
	{
		// Check the relevant mask on one position or four depending the on the size
of results
		int i = tid << size_shift;
		if ((!size_shift && mask[i] == idx) || (size_shift && (mask[i] == idx ||
mask[++i] == idx || mask[++i] == idx || mask[++i] == idx)))
		{
			float3 coord = make_float3(coords[tid][0], coords[tid][1],
coords[tid][2]);

			// Transform the coordinate from [0,extent] to [-0.5, extent-0.5]
			const float3 coord_grid = coord - 0.5;
			const float3 index = floor(coord_grid);
			const float3 fraction = coord_grid - index;
			float3 w0, w1, w2, w3;
			bspline_weights<float3>(fraction, w0, w1, w2, w3);

			const float3 g0 = w0 + w1;
			const float3 g1 = w2 + w3;
			const float3 h0 = (w1 / g0) - 0.5f + index;  //h0 = w1/g0 - 1, move from
[-0.5, extent-0.5] to [0, extent]
			const float3 h1 = (w3 / g1) + 1.5f + index;  //h1 = w3/g1 + 1, move from
[-0.5, extent-0.5] to [0, extent]

			// Fetch the eight linear interpolations
			// Weighting and fetching is interleaved for performance and stability
reasons
			return_type tex000 = tex3D<return_type>(tex, h0.x, h0.y, h0.z);
			return_type tex100 = tex3D<return_type>(tex, h1.x, h0.y, h0.z);
			// Weigh along the x-direction
			tex000 = g0.x * tex000 + g1.x * tex100;
			return_type tex010 = tex3D<return_type>(tex, h0.x, h1.y, h0.z);
			return_type tex110 = tex3D<return_type>(tex, h1.x, h1.y, h0.z);
			// Weigh along the x-direction
			tex010 = g0.x * tex010 + g1.x * tex110;
			// Weigh along the y-direction
			tex000 = g0.y * tex000 + g1.y * tex010;
			return_type tex001 = tex3D<return_type>(tex, h0.x, h0.y, h1.z);
			return_type tex101 = tex3D<return_type>(tex, h1.x, h0.y, h1.z);
			// Weigh along the x-direction
			tex001 = g0.x * tex001 + g1.x * tex101;
			return_type tex011 = tex3D<return_type>(tex, h0.x, h1.y, h1.z);
			return_type tex111 = tex3D<return_type>(tex, h1.x, h1.y, h1.z);
			// Weigh along the x-direction
			tex011 = g0.x * tex011 + g1.x * tex111;
			// Weigh along the y-direction
			tex001 = g0.y * tex001 + g1.y * tex011;

			// Weigh along the z-direction
			results[i] = (g0.z * tex000 + g1.z * tex001);
		}
	}
};

// Cubic B-Spline Interplation of D Dimensions for Nth Dimension
template <unsigned int D, unsigned int N, typename value_type, typename return_type>
class CubicBSplineInterpolation
{
	typedef typename float_key<D> key_type;
	typedef typename CubicBSplineInterpolation<D, N-1, value_type, return_type>
mapped_type;
```

```
	static const unsigned int size_shift = (D == N) ? 0 : 2;

public:
	// Constructor
	__host__ CubicBSplineInterpolation()
	{
		idx_ = 0;
		mapped_ = NULL;
		size_ = 0;
	}

	__host__ void set_idx(unsigned int idx)
	{
		idx_ = idx;
	}

	// Destructor
	__host__ ~CubicBSplineInterpolation()
	{
		if (mapped_ != NULL)
			delete[] mapped_;
	}

	// Loading of data
	__host__ unsigned int load(void* data)
	{
		char* cur_data = static_cast<char*>(data);

		// Retrieve size of this dimension
		size_ = *(static_cast<unsigned int*>(data));
		cur_data += sizeof(unsigned int);

		// Allocate space for subspaces
		mapped_ = new mapped_type[size_];

		for (unsigned int i = 0; i < size_; i++)
		{
			// Load data for a subspace and give its index
			mapped_[i].set_idx(i);
			unsigned int size = mapped_[i].load(cur_data);
			cur_data += size;
		}

		// Return the number of bytes read totally
		return static_cast<unsigned int>(cur_data - static_cast<char*>(data));
	}

	// Get an interpolated value with the usage of a key x with usage of mask
	__host__ int get_value(key_type* x, return_type* results, unsigned int* mask,
unsigned int size)
	{
		cudaError cuda_result;
		dim3 dimBlock(min(size, 1024));
		dim3 dimGrid(static_cast<unsigned int>(ceil(static_cast<float>(size) /
1024)));
		key_type* dev_x;
		return_type* dev_results;
		float* subresults;
		unsigned int* submask;
		float* weights;

		// Allocate space which can be used by the GPU
		cuda_result = cudaMalloc(&subresults, sizeof(return_type) * size * 4);
```

```
            if (cuda_result != cudaSuccess)
            {
                  print_cuda_error("Failed allocating memory space for results",
cuda_result);
                  return -1;
            }

            cuda_result = cudaMalloc(&submask, sizeof(return_type) * size * 4);
            if (cuda_result != cudaSuccess)
            {
                  print_cuda_error("Failed allocating memory space for mask", cuda_result);
                  cudaFree(subresults);
                  return -1;
            }

            cuda_result = cudaMalloc(&weights, sizeof(float) * size * 4);
            if (cuda_result != cudaSuccess)
            {
                  print_cuda_error("Failed allocating memory space for weights",
cuda_result);
                  cudaFree(subresults);
                  cudaFree(submask);
                  return -1;
            }

            // Find the device pointers
            cuda_result = cudaHostGetDevicePointer(&dev_x, x, 0);
            if (cuda_result != cudaSuccess)
            {
                  print_cuda_error("Failed retrieving pointer to Device space for x",
cuda_result);
                  cudaFree(subresults);
                  cudaFree(submask);
                  cudaFree(weights);
                  return -1;
            }

            // Use directly the result pointer or get the device pointer
            if (D == N)
            {
                  cuda_result = cudaHostGetDevicePointer(&dev_results, results, 0);
                  if (cuda_result != cudaSuccess)
                  {
                        print_cuda_error("Failed retrieving pointer to Device space for
results", cuda_result);
                        cudaFree(subresults);
                        cudaFree(submask);
                        cudaFree(weights);
                        return -1;
                  }
            }
            else
            {
                  dev_results = results;
            }

            // Initialize submask with the maximum unsigned value
            cudaMemset(submask, UINT_MAX, size * 4);

            // Calculate weights ans mask
            weights_and_mask<D> << <dimGrid, dimBlock >> >(idx_, dev_x, mask, submask,
weights, size, size_shift, size_);
            cudaThreadSynchronize();
```

```
            // Calculate values on each subspace
            for (unsigned int i = 0; i < size_; i++)
            {
                  if (mapped_[i].get_value(x, subresults, submask, size) == -1)
                  {
                        cudaFree(subresults);
                        cudaFree(submask);
                        cudaFree(weights);
                        return -1;
                  }
            }

            // Calculate the dot product of results found from subspaces
            dot_product << <dimGrid, dimBlock >> >(idx_, dev_results, mask, subresults,
weights, size, size_shift);
            cudaThreadSynchronize();

            cudaFree(subresults);
            cudaFree(submask);
            cudaFree(weights);

            return 0;
      }

protected:
      unsigned int idx_;

private:
      mapped_type* mapped_;
      unsigned int size_;
};

// Cubic B-Spline Interplation of D Dimensions for the 1-Dimension
template <unsigned int D, typename value_type, typename return_type>
class CubicBSplineInterpolation <D, 1, value_type, return_type>
{
      typedef typename float_key<D> key_type;
      static const unsigned int size_shift = (D == 1) ? 0 : 2;

public:
      // Constructor
      __host__ CubicBSplineInterpolation()
      {
            idx_ = 0;
            extent_ = make_cudaExtent(0, 0, 0);
            dev_volume_ = NULL;
            tex_array_ = NULL;
            tex_ = 0;
      }

      // Destructor
      __host__ ~CubicBSplineInterpolation()
      {

            if (tex_ != 0)
                  cudaDestroyTextureObject(tex_);

            if (tex_array_ != NULL)
                  cudaFreeArray(tex_array_);

            if (dev_volume_ != NULL)
                  cudaFree(dev_volume_);
```

```
	}

	__host__ void set_idx(unsigned int idx)
	{
		idx_ = idx;
	}

	// Loading of data
	__host__ unsigned int load(void* data)
	{
		void* raw_data = static_cast<char*>(data) + sizeof(cudaExtent);

		// Extent retrieval
		extent_ = *static_cast<cudaExtent*>(data);
		extent_.height = 0;
		extent_.depth = 0;

		// Prefilter data
		prefilter(raw_data);

		// Create texture
		if (create_texture() != 0)
		{
			perror("Creating texture failed!\n");
			return 0;
		}

		cudaFree(dev_volume_);
		dev_volume_ = NULL;

		// Return the number of bytes read totally
		return static_cast<unsigned int>(sizeof(cudaExtent) + extent_.width *
sizeof(value_type));
	}

	// Get an interpolated value with the usage of a key x with usage of mask
	__host__ int get_value(key_type* x, return_type* results, unsigned int* mask,
unsigned int size)
	{
		cudaError cuda_result;
		key_type* dev_x;
		return_type* dev_results;

		// Find the device pointers
		cuda_result = cudaHostGetDevicePointer(&dev_x, x, 0);
		if (cuda_result != cudaSuccess)
		{
			print_cuda_error("Failed retrieving pointer to Device space for x",
cuda_result);
			return -1;
		}

		// Use directly the result pointer or get the device pointer
		if (D == 1)
		{
			cuda_result = cudaHostGetDevicePointer(&dev_results, results, 0);
			if (cuda_result != cudaSuccess)
			{
				print_cuda_error("Failed retrieving pointer to Device space for
results", cuda_result);
				return -1;
			}
```

```
            }
            else
            {
                  dev_results = results;
            }

            unsigned int results_size = size << size_shift;
            dim3 dimBlock(min(results_size, 1024));
            dim3 dimGrid(static_cast<unsigned int>(ceil(static_cast<float>(results_size) /
1024)));
            cubicTex1D<D, value_type, return_type> << <dimGrid, dimBlock >> >(idx_, tex_,
dev_x, mask, dev_results, size, size_shift);
            cudaThreadSynchronize();

            return 0;
      }

protected:
      unsigned int idx_;

private:
      // Prefiltering of data
      __host__ int prefilter(void* volume)
      {
            cudaError cuda_result;

            // Copy data to GPU
            cuda_result = cudaMalloc(&dev_volume_, extent_.width * sizeof(value_type));
            if (cuda_result != cudaSuccess)
            {
                  print_cuda_error("Failed allocating memory space on Device",
cuda_result);
                  return -1;
            }

            cuda_result = cudaMemcpy(dev_volume_, volume, extent_.width *
sizeof(value_type), cudaMemcpyHostToDevice);
            if (cuda_result != cudaSuccess)
            {
                  print_cuda_error("Failed coping data of volume to Device", cuda_result);
                  cudaFree(dev_volume_);
                  return -1;
            }

            // Call kernel in order to prefilter data
            prefilter1D<value_type> << <1, 1 >> >(dev_volume_, static_cast<unsigned
int>(extent_.width), sizeof(value_type));
            cudaThreadSynchronize();

            return 0;
      }

      // Creating of texture
      __host__ int create_texture()
      {
            cudaError cuda_result;
            cudaChannelFormatDesc channel_desc = cudaCreateChannelDesc<value_type>();
            cudaResourceDesc res_desc;
            cudaTextureDesc tex_desc;

            channel_desc.x = 32;
            channel_desc.f = cudaChannelFormatKindFloat;
```

```
            // Allocate the memory space on Device
            cuda_result = cudaMallocArray(&tex_array_, &channel_desc, extent_.width);
            if (cuda_result != cudaSuccess)
            {
                  print_cuda_error("Failed allocating memory space on Device",
cuda_result);
                  return -1;
            }

            // Copy data of volume to Device
            cuda_result = cudaMemcpyToArray(tex_array_, 0, 0, dev_volume_, extent_.width *
sizeof(value_type), cudaMemcpyDeviceToDevice);
            if (cuda_result != cudaSuccess)
            {
                  print_cuda_error("Failed coping data of volume to Device", cuda_result);
                  cudaFreeArray(tex_array_);
                  tex_array_ = NULL;
                  return -1;
            }

            // Give the resource, texture descriptions and create texture object
            memset(&res_desc, 0, sizeof(cudaResourceDesc));
            res_desc.resType = cudaResourceTypeArray;
            res_desc.res.array.array = tex_array_;

            memset(&tex_desc, 0, sizeof(cudaTextureDesc));
            tex_desc.readMode = cudaReadModeElementType;
            tex_desc.normalizedCoords = false;
            tex_desc.filterMode = cudaFilterModeLinear;
            tex_desc.addressMode[0] = cudaAddressModeClamp;

            cuda_result = cudaCreateTextureObject(&tex_, &res_desc, &tex_desc, NULL);
            if (cuda_result != cudaSuccess)
            {
                  print_cuda_error("Failed creating texture object", cuda_result);
                  cudaFreeArray(tex_array_);
                  tex_array_ = NULL;
                  return -1;
            }

            return 0;
      }

      cudaExtent extent_;
      value_type* dev_volume_;
      cudaArray_t tex_array_;
      cudaTextureObject_t tex_;
};

// Cubic B-Spline Interplation of 2-Dimensions
template <unsigned int D, typename value_type, typename return_type>
class CubicBSplineInterpolation <D, 2, typename value_type, typename return_type>
{
      typedef typename float_key<2> key_type;
      static const unsigned int size_shift = (D == 2) ? 0 : 2;

public:
      // Constructor
      __host__ CubicBSplineInterpolation()
      {
            idx_ = 0;
            extent_ = make_cudaExtent(0, 0, 0);
            dev_volume_ = NULL;
```

```
            tex_array_ = NULL;
            tex_ = 0;
      }

      // Destructor
      __host__ ~CubicBSplineInterpolation()
      {
            if (tex_ != 0)
                  cudaDestroyTextureObject(tex_);

            if (tex_array_ != NULL)
                  cudaFreeArray(tex_array_);

            if (dev_volume_ != NULL)
                  cudaFree(dev_volume_);
      }

      __host__ void set_idx(unsigned int idx)
      {
            idx_ = idx;
      }

      // Loading of data
      __host__ unsigned int load(void* data)
      {
            void* raw_data = static_cast<char*>(data) + sizeof(cudaExtent);

            // Extent retrieval
            extent_ = *static_cast<cudaExtent*>(data);
            extent_.depth = 0;

            // Prefilter data
            prefilter(raw_data);

            // Create texture
            if (create_texture() != 0)
            {
                  perror("Creating texture failed!\n");
                  return 0;
            }

            cudaFree(dev_volume_);
            dev_volume_ = NULL;

            // Return the number of bytes read totally
            return static_cast<unsigned int>(sizeof(cudaExtent) + extent_.width *
extent_.height * sizeof(value_type));
      }

      // Get an interpolated value with the usage of a key x with usage of mask
      __host__ int get_value(key_type* x, return_type* results, unsigned int* mask,
unsigned int size)
      {
            cudaError cuda_result;
            key_type* dev_x;
            return_type* dev_results;

            // Find the device pointers
            cuda_result = cudaHostGetDevicePointer(&dev_x, x, 0);
            if (cuda_result != cudaSuccess)
            {
                  print_cuda_error("Failed retrieving pointer to Device space for x",
cuda_result);
```

```
                return -1;
            }

            // Use directly the result pointer or get the device pointer
            if (D == 2)
            {
                cuda_result = cudaHostGetDevicePointer(&dev_results, results, 0);
                if (cuda_result != cudaSuccess)
                {
                    print_cuda_error("Failed retrieving pointer to Device space for
results", cuda_result);
                    return -1;
                }
            }
            else
            {
                dev_results = results;
            }

            dim3 dimBlock(min(size, 1024));
            dim3 dimGrid(static_cast<unsigned int>(ceil(static_cast<float>(size) /
1024)));
            cubicTex2D<D, value_type, return_type> << <dimGrid, dimBlock >> >(idx_, tex_,
dev_x, mask, dev_results, size, size_shift);
            cudaThreadSynchronize();

            return 0;
        }

protected:
        unsigned int idx_;

private:
        // Prefiltering of data
        __host__ int prefilter(void* volume)
        {
            cudaError cuda_result;

            // Copy data to GPU
            cuda_result = cudaMalloc(&dev_volume_, extent_.width * extent_.height *
sizeof(value_type));
            if (cuda_result != cudaSuccess)
            {
                print_cuda_error("Failed allocating memory space on Device",
cuda_result);
                return -1;
            }

            cuda_result = cudaMemcpy(dev_volume_, volume, extent_.width * extent_.height *
sizeof(value_type), cudaMemcpyHostToDevice);
            if (cuda_result != cudaSuccess)
            {
                print_cuda_error("Failed coping data of volume to Device", cuda_result);
                cudaFree(dev_volume_);
                return -1;
            }

            // Call function in order to prefilter data
            CubicBSplinePrefilter2D<value_type>(dev_volume_, static_cast<unsigned
int>(extent_.width * sizeof(value_type)), static_cast<unsigned int>(extent_.width),
static_cast<unsigned int>(extent_.height));

            return 0;
```

```
	}

	// Creating of texture
	__host__ int create_texture()
	{
		cudaError cuda_result;
		cudaChannelFormatDesc channel_desc = cudaCreateChannelDesc<value_type>();
		cudaResourceDesc res_desc;
		cudaTextureDesc tex_desc;

		channel_desc.x = 32;
		channel_desc.f = cudaChannelFormatKindFloat;

		// Allocate the memory space on Device
		cuda_result = cudaMallocArray(&tex_array_, &channel_desc, extent_.width,
extent_.height);
		if (cuda_result != cudaSuccess)
		{
			print_cuda_error("Failed allocating memory space on Device",
cuda_result);
			return -1;
		}

		// Copy data of volume to Device
		cuda_result = cudaMemcpy2DToArray(tex_array_, 0, 0, dev_volume_, extent_.width
* sizeof(value_type), extent_.width * sizeof(value_type), extent_.height,
cudaMemcpyDeviceToDevice);
		if (cuda_result != cudaSuccess)
		{
			print_cuda_error("Failed coping data of volume to Device", cuda_result);
			cudaFreeArray(tex_array_);
			tex_array_ = NULL;
			return -1;
		}

		// Give the resource, texture descriptions and create texture object
		memset(&res_desc, 0, sizeof(cudaResourceDesc));
		res_desc.resType = cudaResourceTypeArray;
		res_desc.res.array.array = tex_array_;

		memset(&tex_desc, 0, sizeof(cudaTextureDesc));
		tex_desc.readMode = cudaReadModeElementType;
		tex_desc.normalizedCoords = false;
		tex_desc.filterMode = cudaFilterModeLinear;
		tex_desc.addressMode[0] = cudaAddressModeBorder;
		tex_desc.addressMode[1] = cudaAddressModeBorder;

		cuda_result = cudaCreateTextureObject(&tex_, &res_desc, &tex_desc, NULL);
		if (cuda_result != cudaSuccess)
		{
			print_cuda_error("Failed creating texture object", cuda_result);
			cudaFreeArray(tex_array_);
			tex_array_ = NULL;
			return -1;
		}

		return 0;
	}

	cudaExtent extent_;
	value_type* dev_volume_;
	cudaArray_t tex_array_;
	cudaTextureObject_t tex_;
```

```
};

// Cubic B-Spline Interplation of 3-Dimensions
template <unsigned int D, typename value_type, typename return_type>
class CubicBSplineInterpolation <D, 3, typename value_type, typename return_type>
{
	typedef typename float_key<D> key_type;
	static const unsigned int size_shift = (D == 3) ? 0 : 2;

public:
	// Constructor
	__host__ CubicBSplineInterpolation()
	{
		idx_ = 0;
		extent_ = make_cudaExtent(0, 0, 0);
		dev_volume_ = NULL;
		tex_array_ = NULL;
	}

	// Destructor
	__host__ ~CubicBSplineInterpolation()
	{
		if (tex_ != 0)
			cudaDestroyTextureObject(tex_);

		if (tex_array_ != NULL)
			cudaFreeArray(tex_array_);

		if (dev_volume_ != NULL)
			cudaFree(dev_volume_);
	}

	__host__ void set_idx(unsigned int idx)
	{
		idx_ = idx;
	}

	// Loading of data
	__host__ unsigned int load(void* data)
	{
		void* raw_data = static_cast<char*>(data) + sizeof(cudaExtent);

		// Extent retrieval
		extent_ = *static_cast<cudaExtent*>(data);

		// Prefilter data
		prefilter(raw_data);

		// Create texture
		if (create_texture() != 0)
		{
			perror("Creating texture failed!\n");
			return 0;
		}

		cudaFree(dev_volume_);
		dev_volume_ = NULL;

		// Return the number of bytes read totally
		return static_cast<unsigned int>(sizeof(cudaExtent) + extent_.width *
extent_.height * extent_.depth * sizeof(value_type));
	}
```

```
	// Get an interpolated value with the usage of a key x
	__host__ int get_value(key_type* x, return_type* results, unsigned int* mask,
unsigned int size)
	{
		cudaError cuda_result;
		key_type* dev_x;
		return_type* dev_results;

		// Find the device pointers
		cuda_result = cudaHostGetDevicePointer(&dev_x, x, 0);
		if (cuda_result != cudaSuccess)
		{
			print_cuda_error("Failed retrieving pointer to Device space for x",
cuda_result);
			return -1;
		}

		// Use directly the result pointer or get the device pointer
		if (D == 3)
		{
			cuda_result = cudaHostGetDevicePointer(&dev_results, results, 0);
			if (cuda_result != cudaSuccess)
			{
				print_cuda_error("Failed retrieving pointer to Device space for
results", cuda_result);
				return- 1;
			}
		}
		else
		{
			dev_results = results;
		}

		dim3 dimBlock(min(size, 1024));
		dim3 dimGrid(static_cast<unsigned int>(ceil(static_cast<float>(size) /
1024)));
		cubicTex3D<D, value_type, return_type> << <dimGrid, dimBlock >> >(idx_, tex_,
dev_x, mask, dev_results, size, size_shift);
		cudaThreadSynchronize();

		return 0;
	}

protected:
	unsigned int idx_;

private:
	// Prefiltering of data
	__host__ int prefilter(void* volume)
	{
		cudaError cuda_result;

		// Copy data to GPU
		cuda_result = cudaMalloc(&dev_volume_, extent_.width * extent_.height *
extent_.depth * sizeof(value_type));
		if (cuda_result != cudaSuccess)
		{
			print_cuda_error("Failed allocating memory space on Device",
cuda_result);
			return -1;
		}

		cuda_result = cudaMemcpy(dev_volume_, volume, extent_.width * extent_.height *
```

```
extent_.depth * sizeof(value_type), cudaMemcpyHostToDevice);
            if (cuda_result != cudaSuccess)
            {
                  print_cuda_error("Failed coping data of volume to Device", cuda_result);
                  cudaFree(dev_volume_);
                  return -1;
            }

            // Call function in order to prefilter data
            CubicBSplinePrefilter3D<value_type>(dev_volume_, static_cast<unsigned
int>(extent_.width * sizeof(value_type)), static_cast<unsigned int>(extent_.width),
static_cast<unsigned int>(extent_.height), static_cast<unsigned int>(extent_.depth));

            return 0;
      }

      // Creating of texture
      __host__ int create_texture()
      {
            cudaError cuda_result;
            cudaChannelFormatDesc channel_desc = cudaCreateChannelDesc<value_type>();
            cudaPitchedPtr ptr;
            cudaMemcpy3DParms params = { 0 };
            cudaResourceDesc res_desc;
            cudaTextureDesc tex_desc;

            channel_desc.x = 32;
            channel_desc.f = cudaChannelFormatKindFloat;

            // Create a pitched pointer on volume using extent parameters
            ptr = make_cudaPitchedPtr(dev_volume_, extent_.width * sizeof(value_type),
extent_.width, extent_.height);

            // Allocate the memory space on Device
            cuda_result = cudaMalloc3DArray(&tex_array_, &channel_desc, extent_);
            if (cuda_result != cudaSuccess)
            {
                  print_cuda_error("Failed allocating memory space on Device",
cuda_result);
                  return -1;
            }

            // Give the parameters and copy data of volume to Device
            params.extent = extent_;
            params.srcPtr = ptr;
            params.dstArray = tex_array_;
            params.kind = cudaMemcpyDeviceToDevice;
            cuda_result = cudaMemcpy3D(&params);
            if (cuda_result != cudaSuccess)
            {
                  print_cuda_error("Failed coping data of volume to Device", cuda_result);
                  cudaFreeArray(tex_array_);
                  tex_array_ = NULL;
                  return -1;
            }

            // Give the resource, texture descriptions and create texture object
            memset(&res_desc, 0, sizeof(cudaResourceDesc));
            res_desc.resType = cudaResourceTypeArray;
            res_desc.res.array.array = tex_array_;

            memset(&tex_desc, 0, sizeof(cudaTextureDesc));
            tex_desc.readMode = cudaReadModeElementType;
```

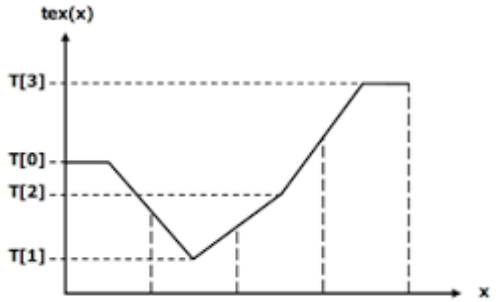
```
            tex_desc.normalizedCoords = false;
            tex_desc.filterMode = cudaFilterModeLinear;
            tex_desc.addressMode[0] = cudaAddressModeClamp;
            tex_desc.addressMode[1] = cudaAddressModeClamp;
            tex_desc.addressMode[2] = cudaAddressModeClamp;

            cuda_result = cudaCreateTextureObject(&tex_, &res_desc, &tex_desc, NULL);
            if (cuda_result != cudaSuccess)
            {
                  print_cuda_error("Failed creating texture object", cuda_result);
                  cudaFreeArray(tex_array_);
                  tex_array_ = NULL;
                  return -1;
            }

            return 0;
      }

      cudaExtent extent_;
      value_type* dev_volume_;
      cudaArray_t tex_array_;
      cudaTextureObject_t tex_;
};

// Cubic B-Spline Interpolation of D Dimensions
template <unsigned int D, typename value_type, typename return_type>
class CubicBSplineInterpolationBase : public CubicBSplineInterpolation<D, D, value_type,
return_type>
{
      typedef typename float_key<D> key_type;
      typedef typename CubicBSplineInterpolation<D, D, value_type, return_type>
parent_class;
public:
      // Get an interpolated value with the usage of a key x
      __host__ void get_value(key_type* x, return_type* results, unsigned int size)
      {
            cudaError cuda_result;
            unsigned int* mask;

            // Allocate space which can be used even by the GPU for the submask
            cuda_result = cudaMalloc(&mask, sizeof(unsigned int) * size);
            if (cuda_result != cudaSuccess)
            {
                  print_cuda_error("Failed allocating memory space for mask", cuda_result);
                  return;
            }

            cuda_result = cudaMemset(mask, parent_class::idx_, size);
            if (cuda_result != cudaSuccess)
            {
                  print_cuda_error("Failed setting memory of mask", cuda_result);
                  cudaFree(mask);
                  return;
            }

            parent_class::get_value(x, results, mask, size);

            cudaFree(mask);
      }
};
```

Για να επιταχυνθεί όμως η διεργασία για τις περιπτώσεις της μιας, των δύο και των τριών διαστάσεων, η υλοποίηση έγινε με τη χρήση του κώδικα που έχει αναπτυχθεί από τους Ruijters, Romeny και Suetens [2]. Κατ' αυτό τον τρόπο γίνεται χρήση του υλικού της μονάδας επεξεργασίας γραφικών για τον γρήγορο υπολογισμό των γραμμικών παρεμβολών όπου χρειάζονται από τον αλγόριθμο.

3.4.2 Δυσκολίες στην υλοποίηση

Κατά τη διάρκεια της υλοποίησης προέκυψαν κάποια προβλήματα. Ένα απ' αυτά είναι η χρήση της διεπαφής προγραμματισμού εφαρμογών με αναφορές υφών. Το πρόβλημα που ανακύπτει είναι ότι με αυτή την διεπαφή δεν μπορούν να δημιουργηθούν δυναμικά οι υφές οι οποίες χρειάζονται. Η δυναμικότητα είναι χρήσιμη καθώς είναι άγνωστος εξ' αρχής ο αριθμός των υφών που θα χρειαστούν, ενώ πρέπει να έχουν δηλωθεί εξ' αρχής με την εκκίνηση της εφαρμογής. Ως λύση είναι η χρήση της διεπαφή προγραμματισμού εφαρμογών με αντικείμενα υφών. Η διεπαφή αυτή όμως είναι διαθέσιμη από την έκδοση 3 της Υπολογιστικής Ικανότητας που έχει μια μονάδα επεξεργασίας γραφικών. Αυτό σημαίνει ότι ο κώδικας μπορεί να χρησιμοποιηθεί μόνο από την γενιά μονάδας επεξεργασίας γραφικών Kepler της Nvidia και μετέπειτα.

Επίσης αυτό που παρατηρήθηκε ήταν ότι χρειάζονται τα δεδομένα να έχουν συγκεκριμένη μορφή. Καταρχάς οι τιμές των δεδομένων που δίνονται αφορούν σε θέσεις που είναι ακέραιες. Δηλαδή θα πρέπει να δοθούν όλες οι τιμές που αντιστοιχούν στις ακέραιες θέσεις ενός πεδίο ορισμού που είναι μεγαλύτερο ή ίσο του μηδενός. Εκτός αυτού επειδή το υλικό τον υφών για μια ακέραιη θέση θεωρεί ότι στην ουσία έχει την τιμή του μέσου όρου που αντιστοιχεί σε αυτή, οι τιμές των δεδομένων πρέπει να θεωρούνται ότι αναφέρονται σε μια ολίσθηση του πεδίου ορισμού κατά +0.5.



Εικόνα 4: Αναπαράσταση των δεδομένων στις διάφορες θέσεις των υφών και της γραμμικής παρεμβολής όπως εφαρμόζεται σε αυτά

Για την υλοποίηση επίσης τέθηκε το πρόβλημα της υλοποίησης της γενίκευσης για τον αριθμό των διαστάσεων. Δηλαδή πως το πλήθος των διαστάσεων μπορεί να μπει δυναμικά χωρίς να χρειάζεται να γίνει συγκεκριμένη υλοποίηση για αυτό. Για την λύση αυτού του προβλήματος, οι τιμές για τις ανευρέσεις και τα ίδια τα αντικείμενα που χειρίζονται τον υπολογισμό με χρήση της παρεμβολής χρησιμοποιούν τα πρότυπα της C++. Έτσι όταν φτιάχνεται ένα αντικείμενο, δυναμικά κατασκευάζεται έτσι ώστε να χειρίζεται έναν συγκεκριμένο αριθμό διαστάσεων όπως έχει σε αυτό ορισθεί.

Τέλος από θέμα της επίδοσης διαπιστώθηκε ότι η υλοποίηση είναι σχετικά αργή για την ανεύρεση μιας τιμής. Ο κώδικας τροποποιήθηκε ακολούθως ώστε να γίνεται αναζήτηση πολλαπλών τιμών αντί για μια. Η επιλογή αυτή έγινε καθώς η ίδια η φύση της επεξεργασίας στην μονάδα επεξεργασίας των γραφικών, επιτρέπει πολλαπλές παράλληλες εκτελέσεις αυξάνοντας την αποδοτικότητα.

3.4.3 Συγκριτικά αποτελέσματα εκτέλεσης

Προκειμένου να γίνει διαπίστωση για την αποτελεσματικότητα της αυτής της υλοποίησης, χρησιμοποιήθηκε υλοποίηση πάνω σε κεντρική μονάδα επεξεργασίας. Σε αυτή την υλοποίηση χρησιμοποιήθηκαν εντολές SSE ώστε να επιταχυνθεί η διαδικασία και για να γίνει και πιο σωστή σύγκριση. Η παρεμβολή εφαρμόστηκε στις τέσσερις διαστάσεις όπου δοκιμάζεται από εκεί και πλέον η επιπλέον υλοποίηση που έχει γίνει στην κάρτα γραφικών. Μέχρι εκεί επίσης η υλοποίηση για την κεντρική μονάδα επεξεργασίας είναι απλή αφού οι εντολές SSE τρέχουν για σύνολα μέχρι τεσσάρων αριθμών κινητής υποδιαστολής απλής ακρίβειας.

Πίνακας 1: Αποτελέσματα σύγκρισης

Μέγεθος χώρου	Πλήθος τιμών αναζήτησης	CPU(ms)	GPU(ms)	Επιτάχυνση
16^4	16^4	13.4187	6.84025	1.96
32^4	32^4	324.932	68.7728	4.72
64^4	64^4	5323.88	1226.92	4.34
64^4	64^3	82.6943	26.9824	3.06
64^4	64^2	1.36261	6.59928	0.20

Από τα παραπάνω μπορούμε να συμπεράνουμε ότι για αρκετά μεγάλους χώρους και για μεγάλο πλήθος τιμών αναζήτησης έχουμε επιτάχυνση που ξεπερνάει το 3 και αγγίζει σχεδόν το 5. Μικρό πλήθος τιμών αναζήτησης αντίθετα παρουσιάζει αρνητικά αποτελέσματα.

4. ΣΥΜΠΕΡΑΣΜΑΤΑ

Από την εργασία αυτή έγινε δυνατόν να επεκταθεί η χρήση της κυβικής Β-καμπυλοειδής παρεμβολή από τη μονάδα επεξεργασίας γραφικών σε περισσότερες από μια διαστάσεις. Αυτό ανοίγει τον δρόμο να χρησιμοποιηθεί από εφαρμογές όπως το SIRENE για την εύρεση τιμών με τη χρήση της παρεμβολής πιο αποδοτικά. Η μεθοδολογία που χρησιμοποιήθηκε επίσης της υλοποίησης με πρότυπα βοηθάει ώστε να φτιάχνονται αυτόματα αντικείμενα που αναλογούν στο χειρισμό της παρεμβολής σε ένα συγκεκριμένο πλήθος διαστάσεων.

Αυτό το οποίο όμως που μένει να μελετηθεί είναι κατά πόσο μπορεί αυτή να εφαρμοστεί από το ίδιο το SIRENE με βάσει τα υπάρχοντα δεδομένα που χρησιμοποιούσε προκειμένου να κάνει τους υπολογισμούς, καθώς αυτά μπορεί να μην αναφέρονται σε σταθερές θέσεις, αλλά απλά να δίνονταν γειτονικά σημεία μη σταθερής απόστασης. Επίσης επειδή οι υφές μέχρι στιγμής περιορίζονται σε χρήση αριθμούς κινητής υποδιαστολή απλής ακρίβειας, πρέπει να γίνει μελέτη ως προς την ακρίβεια των τιμών που εξάγονται και πως μπορεί να βελτιωθεί. Στο SIRENE η ακρίβεια είναι σημαντική καθώς αφορούν υπολογισμούς δεδομένων που ένα μικρό σφάλμα μπορεί να αλλάξει σημαντικά τα αποτελέσματα της προσομοίωσης.

ΠΙΝΑΚΑΣ ΟΡΟΛΟΓΙΑΣ

Ξενόγλωσσος όρος	**Ελληνικός Όρος**
Application Programming Interface	Διεπαφή προγραμματισμού εφαρμογών
Box function	Συνάρτηση κουτιού
Cubic B-Spline Interpolation	Κυβική Β-Καμπυλοειδή Παρεμβολή
Central Processing Unit	Κεντρική Μονάδα Επεξεργασίας
Coefficient	Συντελεστής
Graphics Processing Unit	Μονάδα Επεξεργασίας Γραφικών
General-Purpose computing on Graphics Processing Unit	Υπολογισμοί Γενικού Σκοπού σε Μονάδες Επεξεργασίας Γραφικών
Object Texture API	Διεπαφή προγραμματισμού εφαρμογών με αντικείμενα υφών
Reference Texture API	Διεπαφή προγραμματισμού εφαρμογών με αναφορές υφών
Shower	Καταιονισμός
Spin	Σπιν
Template	Πρότυπο
Texture	Υφή

ΣΥΝΤΜΗΣΕΙΣ – ΑΡΚΤΙΚΟΛΕΞΑ – ΑΚΡΩΝΥΜΙΑ

API	Application Programming Interface
CPU	Central Processing Unit
GPU	Graphical Processing Unit
SIMD	Single Instruction – Multiple Data
SSE	Streaming SIMD Extensions
STL	Standard Template Library

ΑΝΑΦΟΡΕΣ

[1] P. Giannakopoulos, M. Gkoumas, I. Diplas, G. Voularinos, T. Vlachos, K. Balasi, E. Tzamariudaki, C. Filippidis, Y. Cotronis and C. Markou, "A study on implementing a multithreaded version of the SIRENE detector simulation software for high energy neutrinos," *EPJ Web of Conferences,* vol. 116, 2016.

[2] D. Ruijters, B. M. t. H. Romeny και P. Suetens, «Efficient GPU-Based Texture Interpolation using Uniform B-Splines,» *Journal of Graphics Tools,* τόμ. 13, αρ. 4, pp. 61-69, 2008.

[3] D. Ruijters και P. Thevenaz, «GPU Prefilter for Accurate Cubic B-Spline Interpolation,» *The Computer Journal,* τόμ. 55, αρ. 1, pp. 15-20, Ιανουάριος 2012.