



ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

Σχολή Τεχνολογίας

Τμήμα Ψηφιακών Συστημάτων

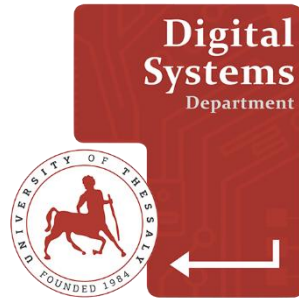
Μελέτη βιβλιοθηκών της CUDA και ανάπτυξη εφαρμογών σε αυτές

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Αλέξανδρος Σταβάρας (ΑΜ: 3120108)

Επιβλέπων: Αθανάσιος Μάργαρης, Συμβασιούχος Διδάσκων

ΛΑΡΙΣΑ, Φεβρουάριος 2025



UNIVERSITY OF THESSALY

School of Technology

Digital Systems Department

CUDA Libraries: Theory and Practice

BACHELOR THESIS

ALEXANDROS STAVARAS (RN: 3120108)

Supervisor: *Athanasios Margaris, Contractual Instructor*

LARISSA, February 2025

Υπεύθυνη Δήλωση περί Ακαδημαϊκής Δεοντολογίας και Πνευματικών Δικαιωμάτων

Με πλήρη επίγνωση των συνεπειών του νόμου περί πνευματικών δικαιωμάτων, δηλώνω ρητά ότι η παρούσα πτυχιακή εργασία, καθώς και τα ηλεκτρονικά αρχεία και πηγαίοι κώδικες που αναπτύχθηκαν ή τροποποιήθηκαν στα πλαίσια αυτής της εργασίας, αποτελεί αποκλειστικά προϊόν προσωπικής μου εργασίας, δεν προσβάλλει κάθε μορφής δικαιώματα διανοητικής ιδιοκτησίας, προσωπικότητας και προσωπικών δεδομένων τρίτων, δεν περιέχει έργα/εισφορές τρίτων για τα οποία απαιτείται άδεια των δημιουργών/δικαιούχων και δεν είναι προϊόν μερικής ή ολικής αντιγραφής, οι πηγές δε που χρησιμοποιήθηκαν περιορίζονται στις βιβλιογραφικές αναφορές και μόνον, και πληρούν τους κανόνες της επιστημονικής παράθεσης. Τα σημεία όπου έχω χρησιμοποιήσει ιδέες, κείμενο, αρχεία ή/και πηγές άλλων συγγραφέων, αναφέρονται ευδιάκριτα στο κείμενο με την κατάλληλη παραπομπή και η σχετική αναφορά περιλαμβάνεται στο τμήμα των βιβλιογραφικών αναφορών με πλήρη περιγραφή.

Αναλαμβάνω πλήρως, ατομικά και προσωπικά, όλες τις νομικές και διοικητικές συνέπειες που δύναται να προκύψουν στην περίπτωση κατά την οποία αποδειχθεί, διαχρονικά, ότι η εργασία αυτή ή τμήμα της δεν μου ανήκει διότι είναι προϊόν λογοκλοπής.

Ο/Η Δηλών/ούσα

(Υπογραφή)

Αλέξανδρος Σταβάρας

23/2/2025

Εγκρίνεται από την Επιτροπή Εξέτασης:

Επιβλέπων	Αθανάσιος Μάργαρης Συμβασιούχος Διδάσκων, Τμήμα Ψηφιακών Συστημάτων, Πανεπιστήμιο Θεσσαλίας
Μέλος	Γεώργιος Αδάμ Καθηγητής, Τμήμα Ψηφιακών Συστημάτων, Πανεπιστήμιο Θεσσαλίας
Μέλος	Όμηρος Ιατρέλλης Επίκουρος Καθηγητής, Τμήμα Ψηφιακών Συστημάτων, Πανεπιστήμιο Θεσσαλίας

Ημερομηνία έγκρισης: dd-mm-2025

Περίληψη

Η παρούσα πτυχιακή εργασία παρουσιάζει τις βιβλιοθήκες της NVIDIA οι οποίες εκμεταλλεύονται την παράλληλο επεξεργαστικό περιβάλλον της κάρτας γραφικών, με σκοπό την ανάδειξη της πρακτικής τους αξίας μέσα από παραδείγματα και υλοποιήσεις κώδικα. Ο κύριος στόχος δεν ήταν η εις βάθος θεωρητική ανάλυση, αλλά η επίδειξη της πρακτικότητας που προσφέρουν αυτές οι τεχνολογίες στους προγραμματιστές, επιτρέποντάς τους να αξιοποιήσουν την υψηλή ταχύτητα επεξεργασίας.

Στο τελικό αποτέλεσμα της εργασίας γίνεται η επίδειξη εφαρμογών που εκτελούνται σε πραγματικό χρόνο, επιβεβαιώνοντας την αποτελεσματικότητα και τη χρηστικότητα των εξεταζόμενων βιβλιοθηκών. Παράλληλα, τα θέματα των εφαρμογών που αναπτύχθηκαν υποδεικνύουν ότι υπάρχει σημαντικός χώρος για μελλοντικές βελτιώσεις και επεκτάσεις, προσφέροντας τη βάση για περαιτέρω έρευνα και εμβάθυνση σε τομείς όπου η επεξεργασία δεδομένων απαιτεί υψηλές επιδόσεις.

Η περίληψη αυτή λειτουργεί ως σύντομη διαφήμιση της ολοκληρωμένης πτυχιακής εργασίας, προσφέροντας στον αναγνώστη μια ξεκάθαρη εικόνα για το περιεχόμενο και τα αποτελέσματα που επιτεύχθηκαν, και τον ενθαρρύνει να εξερευνήσει περαιτέρω το έργο.

Abstract

This thesis presents NVIDIA libraries that leverage the parallel processing environment of the graphics card, aiming to highlight their practical value through examples and code implementations. The main goal was not an in-depth theoretical analysis, but to demonstrate the practicality these technologies offer to developers, enabling them to harness high-speed real-time processing.

The final result was particularly impressive, with applications running in real-time, confirming the effectiveness and usability of the examined libraries. Moreover, the developed applications indicate significant room for future improvements and extensions, providing a foundation for further research and exploration in high-performance data processing fields.

This summary serves as a brief advertisement for the completed thesis, giving the reader a clear idea of the content and results achieved, encouraging further exploration of the work

Ευχαριστίες

Θα ήθελα να ευχαριστήσω θερμά την οικογένεια μου για την στήριξη που μου προσέφερε καθώς επίσης και τον επιβλέποντα καθηγητή μου κ.Μάργαρη Αθανάσιο για την καθοδήγηση του κατά την διάρκεια της εκπόνησης της πτυχιακής μου εργασίας

Αλέξανδρος Σταβάρας

23/2/2025

Περιεχόμενα

ΠΕΡΙΛΗΨΗ	IX
ABSTRACT	XI
ΕΥΧΑΡΙΣΤΙΕΣ.....	XIII
ΠΕΡΙΕΧΟΜΕΝΑ.....	XV
1 ΕΙΣΑΓΩΓΗ	1
2 ΒΙΒΛΙΟΘΗΚΕΣ CUDA.....	3
2.1 ΒΙΒΛΙΟΘΗΚΗ CUBLAS	4
2.1.1 Εισαγωγή.....	5
2.1.2 Πως χρησιμοποιούμε την βιβλιοθήκη;.....	6
2.1.3 Συναρτήσεις – λειτουργίες :	7
2.1.4 cuBLAS Level-1 Functions	9
2.1.5 cuBLAS Level-2 Function	10
2.1.6 cuBLAS Level-3 Function	15
2.2 ΒΙΒΛΙΟΘΗΚΗ CUSPARSE	17
2.2.1 Εισαγωγή στη CUSPARSE	17
2.2.2 Αναπαράσταση δεδομένων.....	18
2.2.3 Πως χρησιμοποιείται η βιβλιοθήκη	20
2.2.4 Συναρτήσεις της CUSPARSE.....	21
2.3 ΒΙΒΛΙΟΘΗΚΗ CUFFT	27
2.3.1 Εισαγωγή.....	27
2.3.2 Πως χρησιμοποιούμε τη βιβλιοθήκη.....	29
2.3.3 Διάταξη δεδομένων	30
2.3.4 Αρχικοποίηση και Διαχείριση Μνήμης:.....	31
2.4 ΒΙΒΛΙΟΘΗΚΗ CURAND	32
2.4.1 Εισαγωγή.....	32
2.4.2 Περιγραφή των συναρτήσεων του Host API	33
2.4.3 Συναρτήσεις Device API.....	35

2.5	BIBΛΙΟΘΗΚΗ NPP(NVIDIA PERFORMANCE PRIMITIVES)	39
2.5.1	Λειτουργικότητα της NPP.....	40
2.5.2	Λειτουργίες τις βιβλιοθήκης για επεξεργασία εικόνας.....	41
2.5.3	Περιγραφή των λειτουργιών για επεξεργασία σήματος.....	46
2.5.4	Διαχείριση Μνήμης κατά τη Χρήση της Βιβλιοθήκης NPP.....	47
2.6	BIBΛΙΟΘΗΚΗ THURST	49
2.6.1	Πως χρησιμοποιείται η βιβλιοθήκη.....	50
2.6.2	Αναλυτική περιγραφή λειτουργικότητας της βιβλιοθήκης.....	51
2.7	ΑΝΑΦΟΡΕΣ.....	56
3	ΠΡΟΕΤΟΙΜΑΣΙΑ ΓΙΑ ΥΛΟΠΟΙΗΣΗ ΕΦΑΡΜΟΓΩΝ ΣΕ ΠΕΡΙΒΑΛΛΟΝ	
CUDA		57
3.1	CUDA TOOLKIT	58
3.1.1	Εγκατάσταση	58
3.1.2	Επιπλέον λειτουργίες και δυνατότητες.....	59
3.1.3	Υποστήριξη Διαλειτουργικότητας με Γραφικά.....	59
3.1.4	Microsoft Visual studio.....	60
3.2	ΕΠΙΠΛΕΟΝ ΒΟΗΘΗΤΙΚΕΣ ΒΙΒΛΙΟΘΗΚΕΣ.....	60
3.2.1	OpenCV.....	60
3.2.2	OpenGL.....	61
3.2.3	Libsndfile.....	61
4	ΥΛΟΠΟΙΗΣΗ ΕΦΑΡΜΟΓΩΝ.....	63
4.1	ΕΦΑΡΜΟΓΗ ΕΠΕΞΕΡΓΑΣΙΑΣ ΗΧΟΥ	63
4.1.1	Μετασχηματισμός Fourier και ψηφιακή επεξεργασία ήχου:	63
4.1.2	Απαιτήσεις και Ρυθμίσεις Εφαρμογής.....	65
4.1.3	Υλοποίηση:	65
4.1.4	Αποτελέσματα Εφαρμογής	67
4.2	ΕΠΙΤΑΧΥΝΣΗ 3D ΓΡΑΦΙΚΩΝ ΜΕ CUBLAS.....	68
4.2.1	Εισαγωγή στην OpenGL και τη Χρήση της στο Έργο.....	68
4.2.2	Απαιτήσεις και Ρυθμίσεις Περιβάλλοντος.....	69
4.2.3	Υλοποίηση της εφαρμογής.....	70
4.2.4	Σημαντικά σημεία κώδικα.....	74
4.2.5	Benchmarking.....	78

4.3	ΕΠΕΞΕΡΓΑΣΙΑ ΕΙΚΟΝΑΣ ΜΕ ΧΡΗΣΗ ΤΗΣ ΒΙΒΛΙΟΘΗΚΗΣ NPP	80
4.3.1	Απαιτήσεις και Ρυθμίσεις Περιβάλλοντος	80
4.3.2	<i>Gaussian blur</i> :.....	81
4.3.3	Υλοποίηση κωδικα	81
4.3.4	Αποτελέσματα προγράμματος και σύγκριση απόδοσης CPU- GPU 83	
5	ΤΕΧΝΙΚΕΣ ΛΕΠΤΟΜΕΡΕΙΕΣ ΚΑΙ ΟΔΗΓΙΕΣ ΧΡΗΣΗΣ.....	85
5.1	ΠΩΣ ΧΡΗΣΙΜΟΠΟΙΟΥΜΕ ΤΙΣ ΕΦΑΡΜΟΓΕΣ.....	85
5.1.1	Εφαρμογή επεξεργασίας ήχου με <i>fft</i> :	85
5.1.2	Πρόγραμμα 3D γραικών με <i>OpenGL</i> και <i>CUBLAS</i>	86
5.1.3	Εφαρμογή για την επεξεργασία εικόνας.....	87
5.2	ΛΕΙΤΟΥΡΓΙΕΣ ΚΛΕΙΔΙΑ	88
5.2.1	Προσομοίωση εφέ κύματος και περιστροφής	88
5.2.2	Δομές για την διαχείριση των <i>buffer</i> στην <i>OpenGL</i>	90
5.2.3	Εφαρμογή του <i>short time Fourier transform</i> για την ανάλυση και επεξεργασία δείγματα ήχου στο πεδίο της συχνότητας	92
6	ΣΥΜΠΕΡΑΣΜΑΤΑ.....	95
	ΒΙΒΛΙΟΓΡΑΦΙΑ	97

1 Εισαγωγή

Οι κάρτες γραφικών σχεδιάστηκαν αρχικά για την απόδοση τρισδιάστατων γραφικών, όπως αυτά που συναντάμε στα βιντεοπαιχνίδια και στα προγράμματα επεξεργασίας εικόνας. Οι κεντρικές μονάδες επεξεργασίας (CPU) δεν ήταν κατάλληλες για τέτοιες απαιτητικές εργασίες, καθώς έχουν σχεδιαστεί κυρίως για σειριακή εκτέλεση κώδικα και για την επίλυση γενικών υπολογιστικών προβλημάτων. Ωστόσο, με την πάροδο του χρόνου, έγινε αντιληπτό ότι οι GPUs, λόγω της υψηλής επεξεργαστικής τους ισχύος, μπορούσαν να αξιοποιηθούν και σε άλλους τομείς πέρα από την απεικόνιση γραφικών.

Η μεγάλη υπολογιστική δύναμη των GPUs οφείλεται κυρίως στον τρόπο σχεδιάσής τους. Στην ίδια επιφάνεια σιλικόνης διαθέτουν περισσότερες μονάδες υπολογισμών και επεξεργασίας δεδομένων σε σύγκριση με ένα επεξεργαστή ο οποίος φιλοξενεί μεγάλη ποσότητα μνήμη cache. Το γεγονός ότι υπάρχουν τόσοι πολλοί πυρήνες για μικρές πράξεις είναι αυτό που τις καθιστά εξαιρετικά καλή επιλογή για μαζική παραλληλία, σε εφαρμογές που απαιτούν την επεξεργασία μεγάλων συνόλων δεδομένων και επιτρέπουν την ταυτόχρονη εκτέλεση πολλών ανεξάρτητων υπολογισμών. Αντίθετα οι επεξεργαστές (CPUs) είναι σχεδιασμένοι να εκτελούν μικρό όγκο εργασιών με πολύ υψηλή ταχύτητα, γεγονός που τους καθιστά ιδανικούς για γενικές υπολογιστικές διεργασίες.

Αυτή η υπεροχή της παραλληλίας των GPUs έγινε αντιληπτή ως ένας ισχυρός μηχανισμός επιτάχυνσης που θα μπορούσε να αξιοποιηθεί και σε άλλους υπολογιστικούς τομείς, προσφέροντας σημαντικό πλεονέκτημα σε ταχύτητα. Για τον λόγο αυτό, σήμερα οι GPUs δεν χρησιμοποιούνται αποκλειστικά για την απόδοση γραφικών, αλλά και ως βασικά εργαλεία επιτάχυνσης σε ποικίλες υπολογιστικές εφαρμογές με την ικανότητά τους να εκτελούν χιλιάδες πράξεις ταυτόχρονα. Έχουν πλέον σημαντικό ρόλο σε τομείς όπως η επιστημονική έρευνα, η τεχνητή νοημοσύνη, η επεξεργασία δεδομένων και η ανάλυση μεγάλων δεδομένων (Big Data), χρησιμοποιούνται σε εφαρμογές όπως τα νευρωνικά δίκτυα, η εξόρυξη κρυπτονομισμάτων, οι φυσικές προσομοιώσεις και χρηματοοικονομικές αναλύσεις.

Στις σύγχρονες υπολογιστικές εφαρμογές, η ανάγκη για υψηλή απόδοση και επεξεργασία μεγάλων όγκων δεδομένων οδήγησε στην ανάγκη για αξιοποίηση των GPUs και γιαυτό δημιουργήθηκε το μοντέλο του ετερογενούς υπολογισμού (heterogeneous

computing) που επιτρέπει την κατανομή των υπολογιστικών φορτίων μεταξύ της CPU και της GPU, αξιοποιώντας τις δυνατότητες της κάθε μονάδας ανάλογα με το είδος της εργασίας. Στα τέλη του 2006, η NVIDIA έκανε ένα σημαντικό βήμα προς αυτή την κατεύθυνση, εισάγοντας την ιδέα ενός προγραμματιστικού περιβάλλοντος που θα επέτρεπε στις κάρτες γραφικών να χρησιμοποιηθούν για γενικής φύσεως υπολογιστικές εργασίες.

Αυτό οδήγησε στη δημιουργία της CUDA (Compute Unified Device Architecture), ενός από τα πρώτα αξιόπιστα συστήματα προγραμματισμού GPU, που απομακρύνθηκε από την παραδοσιακή προσέγγιση «code-it-as-graphics», η οποία απαιτούσε την προσομοίωση υπολογιστικών εργασιών μέσω γραφικών APIs. Η CUDA παρέχει δύο επίπεδα διεπαφής: το χαμηλού επιπέδου CUDA Driver API και το υψηλότερου επιπέδου CUDA Runtime API, προσφέροντας στους προγραμματιστές μεγαλύτερη ευελιξία στον τρόπο αξιοποίησης του υλικού. Η πρώτη δημόσια έκδοση του CUDA SDK κυκλοφόρησε στις 15 Φεβρουαρίου 2007 για Windows και Linux, ενώ η υποστήριξη για Mac OS X προστέθηκε στην έκδοση 2.0. Η CUDA είναι αποκλειστικά συμβατή με τις κάρτες γραφικών της NVIDIA και παρέχει ένα προγραμματιστικό περιβάλλον με δικό του compiler, επιτρέποντας τη χρήση γλωσσών όπως η C και η C++ για την ανάπτυξη εφαρμογών υψηλής απόδοσης.

Τα επόμενα κεφάλαια της παρούσας εργασίας εμβαθύνουν στις βασικές βιβλιοθήκες της NVIDIA και την πρακτική τους εφαρμογή στο περιβάλλον CUDA. Συγκεκριμένα, στο Κεφάλαιο 2 θα παρουσιαστεί αναλυτικά η θεωρητική θεμελίωση και οι λειτουργικές ιδιότητες των βασικών βιβλιοθηκών. Στο Κεφάλαιο 3 θα περιγραφούν οι απαραίτητες διαδικασίες προετοιμασίας του περιβάλλοντος ανάπτυξης, με έμφαση στην εγκατάσταση και διαμόρφωση του CUDA Toolkit καθώς και στις υποστηρικτικές τεχνολογίες. Στο Κεφάλαιο 4 θα παρουσιαστούν οι υλοποιήσεις εφαρμογών, με παραδείγματα που αποδεικνύουν την πρακτική αξία και τα οφέλη της παραλληλίας στην επεξεργασία δεδομένων. Τέλος, στο Κεφάλαιο 5 θα αναλυθούν τα αποτελέσματα των υλοποιήσεων, ενώ στο Κεφάλαιο 6 θα συνοψιστούν τα συμπεράσματα της εργασίας και θα προταθούν πιθανές προεκτάσεις για μελλοντική έρευνα.

2 Βιβλιοθήκες CUDA

Το παράλληλο μοντέλο προγραμματισμού CUDA έχει σχεδιαστεί για να διευκολύνει την ανάπτυξη λογισμικού που αξιοποιεί την επεξεργαστική ισχύ των GPUs, χωρίς να απαιτεί από τους προγραμματιστές να μάθουν εντελώς νέες γλώσσες προγραμματισμού. Αντίθετα, βασίζεται σε επεκτάσεις της γλώσσας C, επιτρέποντας μια πιο ομαλή μετάβαση στην παράλληλη επεξεργασία. Στον πυρήνα του CUDA βρίσκονται τρεις βασικές έννοιες που βοηθούν στην αποτελεσματική εκμετάλλευση της παραλληλίας. Η ιεραρχία ομάδων νημάτων, κοινόχρηστες μνήμες και συγχρονισμός φραγμού.

Ο κώδικας οργανώνεται σε μικρές ομάδες παράλληλων νημάτων (threads), οι οποίες ομαδοποιούνται σε μπλοκ νημάτων (thread blocks) και περαιτέρω σε πλέγματα μπλοκ (grids). Αυτή η δομή επιτρέπει την εύκολη κλιμάκωση των υπολογισμών για μεγάλο πλήθος δεδομένων. Οι ομάδες νημάτων μπορούν να μοιράζονται δεδομένα μέσω της κοινόχρηστης μνήμης (shared memory), η οποία είναι πολύ ταχύτερη από τη κύρια μνήμη της GPU (global memory) μειώνοντας την ανάγκη για αργές μεταφορές δεδομένων και απώλεια στην απόδοση. Τα νήματα μπορούν να συγχρονίζονται μεταξύ τους, ώστε να εκτελούν συγκεκριμένες λειτουργίες με συντονισμένο τρόπο, μοιράζοντας ομοιόμορφα το φορτίο επεξεργασίας σε όλους τους διαθέσιμους πυρήνες. Αυτές οι αρχές καθοδηγούν τον προγραμματιστή να χωρίσει ένα πρόβλημα σε μικρότερα υποπροβλήματα, τα οποία μπορούν να λυθούν ταυτόχρονα από διαφορετικά μπλοκ νημάτων. Στη συνέχεια, κάθε μπλοκ αναθέτει στα νήματά του να συνεργαστούν για την επίλυση του συγκεκριμένου υποπροβλήματος.

Αυτό το μοντέλο όχι μόνο διευκολύνει την ανάπτυξη παράλληλου κώδικα, αλλά επιτρέπει και την αυτόματη κλιμάκωση της εκτέλεσης. Τα μπλοκ νημάτων μπορούν να εκτελούνται σε οποιονδήποτε από τους διαθέσιμους πολυεπεξεργαστές μιας GPU, είτε ταυτόχρονα είτε διαδοχικά, ανάλογα με τους διαθέσιμους πόρους. Έτσι, ένα πρόγραμμα CUDA μπορεί να εκτελεστεί αποδοτικά ανεξάρτητα από τον αριθμό των πολυεπεξεργαστών της GPU, αφού το σύστημα χρόνου εκτέλεσης αναλαμβάνει να διαχειριστεί την κατανομή των υπολογισμών.

Αφού κατανοήσαμε τη βασική δομή της CUDA και τον τρόπο που επιτρέπει την αποδοτική εκμετάλλευση της υπολογιστικής ισχύος των GPUs, είναι σημαντικό να

εξετάσουμε τις μερικές βασικές βιβλιοθήκες που έχουν αναπτυχθεί με κατά κόρο αυτό το μοντέλο.

Η NVIDIA έχει αναπτύξει ένα σύνολο από εξειδικευμένες βιβλιοθήκες που επιτρέπουν τη βελτιστοποιημένη εκτέλεση συγκεκριμένων αλγορίθμων χωρίς να απαιτείται η άμεση υλοποίηση τους σε χαμηλό επίπεδο CUDA κώδικα. Αυτές οι βιβλιοθήκες επιταχύνουν σημαντικά διάφορες αριθμητικές πράξεις, επιτρέποντας την αξιοποίηση των GPUs σε ένα ευρύ φάσμα εφαρμογών με μεγάλη ευκολία ενσωμάτωσης σε ήδη υπάρχοντα project.

Στην παρούσα πτυχιακή εργασία, η εστίαση δίνεται στις ακόλουθες βιβλιοθήκες:

- CUBLAS (CUDA Basic Linear Algebra Subprograms): Μια βιβλιοθήκη βελτιστοποιημένης εκτέλεσης πράξεων γραμμικής άλγεβρας, όπως πίνακες και διανύσματα, η οποία χρησιμοποιείται εκτενώς σε επιστημονικούς υπολογισμούς.
- CUSPARSE (CUDA Sparse Matrix Library): Εξειδικευμένη για πράξεις με αραιούς πίνακες, επιτρέποντας σημαντική εξοικονόμηση μνήμης και βελτιωμένη απόδοση σε υπολογισμούς που βασίζονται σε αραιή γραμμική άλγεβρα.
- CURAND (CUDA Random Number Generation Library): Παρέχει γρήγορη παραγωγή τυχαίων αριθμών σε GPU.
- CUFFT (CUDA Fast Fourier Transform Library): Βιβλιοθήκη για αποδοτική εκτέλεση του γρήγορου μετασχηματισμού Fourier (FFT), χρήσιμη για επεξεργασία σήματος και ανάλυση δεδομένων.
- NPP (Nvidia Performance Primitives): Ένα σύνολο συναρτήσεων για εφαρμογές επεξεργασίας εικόνας και σήματος.
- Thrust: Επέκταση της γλώσσας C++ με υποστηρικτικές λειτουργίες και αλγορίθμους που ενεργοποιούν τις δυνατότητες της gpu.

2.1 Βιβλιοθήκη CUBLAS

Η βιβλιοθήκη cuBLAS αποτελεί μια βελτιστοποιημένη έκδοση της παραδοσιακής BLAS, σχεδιασμένη για την πλήρη εκμετάλλευση των πόρων της GPU μέσω του περιβάλλοντος CUDA. Διαθέτει τέσσερα διακριτά API, το καθένα από τα οποία εξυπηρετεί συγκεκριμένες ανάγκες:

Βασικό cuBLAS API:

Παρέχει ένα σύνολο συναρτήσεων για τις βασικές πράξεις γραμμικής άλγεβρας (π.χ. πολλαπλασιασμός πινάκων, πράξεις με διανύσματα). Αυτή η συλλογή αποτελεί τον κύριο πυρήνα της βιβλιοθήκης και θα αναλυθεί λεπτομερώς στην παρούσα εργασία.

cuBLASxt API:

Επεκτείνει τη λειτουργικότητα του βασικού API επιτρέποντας τη διαχείριση μνήμης τόσο στον host (CPU) όσο και σε πολλαπλές GPU. Μέσω της αυτόματης μεταφοράς των δεδομένων, εξασφαλίζεται ο ισορροπημένος διαμοιρασμός του υπολογιστικού φορτίου μεταξύ των διαφόρων συσκευών.

cuBLASLt API:

Σχεδιασμένο για εφαρμογές που απαιτούν υψηλή ευελιξία στον πολλαπλασιασμό πινάκων (GEMM), επιτρέπει την προσαρμογή της διάταξης των δεδομένων, των τύπων εισόδου/εξόδου και της επιλογής αλγορίθμων. Αυτή η επιλογή είναι ιδιαίτερα χρήσιμη για εξειδικευμένες εφαρμογές που απαιτούν λεπτομερή ρύθμιση.

cuBLASDx API:

Πρόκειται για μια εξειδικευμένη επέκταση, η οποία δεν περιλαμβάνεται στο τυπικό πακέτο του CUDA Toolkit, και επικεντρώνεται σε συγκεκριμένες αριθμητικές λειτουργίες που απαιτούν προσαρμοσμένη διαχείριση και βελτιστοποίηση.

Από τα παραπάνω, το βασικό cuBLAS API αποτελεί το κύριο εργαλείο της βιβλιοθήκης, το οποίο θα αναλυθεί και θα εφαρμοστεί στην εργασία μας για την υλοποίηση βασικών πράξεων γραμμικής άλγεβρας σε περιβάλλον GPU.

2.1.1 Εισαγωγή

Το βασικό API της cuBLAS παρέχει ένα σύνολο εύχρηστων συναρτήσεων για την αποδοτική εκτέλεση πράξεων γραμμικής άλγεβρας στην GPU, επιταχύνοντας σημαντικά τις υπολογιστικές πράξεις. Οι συναρτήσεις αυτές είναι οργανωμένες σε τρία επίπεδα. Στο πρώτο επίπεδο, περιλαμβάνονται βασικές λειτουργίες σε διανύσματα, όπως το εσωτερικό γινόμενο και η κλιμάκωση διανυσμάτων, στο δεύτερο περιλαμβάνει πράξεις μεταξύ διανυσμάτων και πινάκων, όπως ο πολλαπλασιασμός πίνακα-διανύσματος (matrix-vector multiplication) και η επίλυση τριγωνικών συστημάτων και τέλος στο τρίτο επίπεδο, περιλαμβάνονται πιο σύνθετες πράξεις πινάκων, με κύρια λειτουργία τον πολλαπλασιασμό πίνακα-πίνακα (GEMM), ο οποίος αξιοποιεί στο μέγιστο την υπολογιστική ισχύ της GPU. Όσο πιο σύνθετες γίνονται οι συναρτήσεις, τόσο πιο αισθητή γίνεται η βελτιστοποίηση των υπολογισμών, καθώς οι πράξεις ανώτερου

επιπέδου αξιοποιούν πιο αποτελεσματικά την παραλληλία της GPU, επιτυγχάνοντας σημαντική αύξηση της απόδοσης σε μαθηματικές πράξεις μεγάλης κλίμακας.

Η διαχείριση μνήμης παίζει καθοριστικό ρόλο στη βέλτιστη αξιοποίηση της cuBLAS, καθώς οι μεταφορές δεδομένων μεταξύ της μνήμης του host και της συσκευής (device) μπορούν να αποτελέσουν σημαντικό σημείο συμφόρησης. Ο προγραμματιστής πρέπει να διασφαλίζει τη σωστή κατανομή και απελευθέρωση της μνήμης στη GPU, αποφεύγοντας περιττές καθυστερήσεις και μειώνοντας την αλληλεπίδραση με την CPU. Επιπλέον, η cuBLAS χρησιμοποιεί διάταξη δεδομένων column-major και 1-based indexing, χαρακτηριστικά που εξυπηρετούν τη συμβατότητα με υπάρχοντα BLAS routines και κώδικα γραμμένο σε Fortran και πρέπει να λαμβάνονται υπόψη κατά την προετοιμασία των δεδομένων για την εκτέλεση πράξεων.

2.1.2 Πως χρησιμοποιούμε την βιβλιοθήκη;

Αρχικοποίηση της βιβλιοθήκης

Το πρώτο βήμα είναι η αρχικοποίηση της βιβλιοθήκης καλώντας την συνάρτηση `cublasCreate()` όπου και δημιουργεί μια λαβή (handle) στην εκάστοτε καθορισμένη συσκευή που έγινε από την κλήση της `cudaInitDevice()` ή της `cudaSetDevice()`. Η παράμετρος handle θα χρησιμοποιηθεί για όλες τις επόμενες κλήσεις της CUBLAS και είναι μοναδική για το thread που έκανε την κλήση. Αν σε ένα σύστημα έχουμε πολλές μονάδες συσκευών με πολλά νήματα μπορούμε ευκολά να αναθέσουμε τους πόρους για τον υπολογισμό σε διαφορετικές συσκευές δημιουργώντας διαφορετικά handles για κάθε συσκευή. Με αυτόν τον τρόπο ο προγραμματιστής έχει τον απολυτό έλεγχο ενώ ταυτόχρονα επιτρέπει την εκτέλεση πολλαπλών νημάτων.

Στην συνέχεια είναι ευθύνη του προγραμματιστή να δηλώσει σωστά την ποσότητα μνήμης που θα χρησιμοποιηθεί στη συσκευή. Αυτό γίνεται μέσω της χρήσης συναρτήσεων όπως η `cudaMalloc()` για την διάθεση μνήμης για πίνακες και δεδομένα που θα χρησιμοποιήσουν οι συναρτήσεις για πράξεις. Σημαντική προϋπόθεση για την αποφυγή σφαλμάτων είναι η σωστή διαχείριση της μνήμης στη GPU καθώς όλες οι συναρτήσεις της βιβλιοθήκης που δέχονται ως παράμετρο δείκτη σε πίνακα ή διάνυσμα, ο δείκτης αυτός πρέπει να δείχνει σε χώρο μνήμης που βρίσκεται στην συσκευή. Η σωστή δέσμευση και διαχείριση της μνήμης αποτελεί κρίσιμο σημείο στον προγραμματισμό και απαιτεί ιδιαίτερη προσοχή ώστε να εξασφαλιστεί η ομαλή εκτέλεση χωρίς σφάλματα.

Το επόμενο βήμα είναι να μεταφερθούν τα δεδομένα από τον Host στην GPU. Ο Κλασικός τρόπος θα ήταν να κληθεί η `cudaMemcpy` και να γίνει έτσι η μεταφορά ωστόσο η βιβλιοθήκη `cuBLAS` παρέχει ένα σύνολο βοηθητικών συναρτήσεων για τη μεταφορά δεδομένων, οι οποίες μπορούν να αντικαταστήσουν τη χρήση της όταν δουλεύουμε με διανύσματα ή πίνακες. Οι συναρτήσεις μεταφοράς είναι οι `cublasSetVector` και `cublasSetMatrix`. Η συνάρτηση `cublasSetVector` αντιγράφει n στοιχεία μεγέθους `elemSize` από το διάνυσμα x στην μνήμη του Host στο διάνυσμα y στην μνήμη της GPU με βήμα προσπέλασης του επομένου στοιχείου `incx` και `incy` για το διάνυσμα εισόδου και διάνυσμα προορισμού αντίστοιχα. Η συνάρτηση `cublasSetMatrix` χρησιμοποιείται για να αντιγράψει ένα μπλοκ δεδομένων μεγέθους `rows x cols` από έναν πίνακα A στη μνήμη του Host σε έναν πίνακα B στη μνήμη της GPU. Η αντιγραφή γίνεται με διάταξη κατά στήλης (`column-major format`). Οι παράμετροι `lda` και `ldb` μπορεί να είναι μεγαλύτεροι από τον αριθμό των γραμμών του υποπίνακα που θέλουμε να αντιγράψουμε. Αυτό βοηθά να αντιγράφονται τμήματα από μεγαλύτερους πίνακες ή να γίνονται μεταφορές δεδομένων σε συγκεκριμένα υποσύνολα των πινάκων. Επίσης υπάρχουν και οι εκδοχές `cublasSetVectorAsync` και `cublasSetMatrixAsync` που προσφέρουν την ίδια λειτουργία με την μόνη διαφορά ότι εκτελούνται ασύγχρονα (ως προς τον Host). Αντίστοιχα για την λήψη των δεδομένων πίσω στον host έχουμε τις συναρτήσεις : `cublasGetVector()` και `cublasGetMatrix()` που λειτουργούν με παρόμοιο τρόπο και η διάταξη των δεδομένων διατηρείται σε μορφή κατά στήλης (`column-major`). Υπάρχουν και ασύγχρονες εκδοχές (`cublasGetVectorAsync`, `cublasGetMatrixAsync`) που επιτρέπουν την ταχύτερη ανάκτηση δεδομένων με παράλληλη εκτέλεση εργασιών στο Host.

2.1.3 Συναρτήσεις – λειτουργίες :

Υποστηρίζει τη χρήση ροών (`streams`) για την εκτέλεση των συναρτήσεων, μια λειτουργικότητα που μπορεί να συμβάλει στην ταχύτερη και αποδοτικότερη εκτέλεση υπολογισμών. Οι ροές επιτρέπουν την παράλληλη εκτέλεση ανεξάρτητων εργασιών στην GPU, βελτιώνοντας την απόδοση, ειδικά όταν οι εργασίες δεν αξιοποιούν πλήρως τους διαθέσιμους πόρους της GPU. Η ενσωμάτωση των `streams` στις κλήσεις της `cuBLAS` γίνεται μέσω της χρήσης της συνάρτησης `cublasSetStream()`, η οποία ορίζει ποια ροή θα χρησιμοποιηθεί για τις επόμενες λειτουργίες.

Όλες οι συναρτήσεις ακολουθούν την διάταξη ονομασίας `cublas<t><operation>` για ευκολή αναγνώριση, όπου το `<t>` δηλώνει τον τύπο δεδομένων και το `<operation>` την

πράξη που εκτελείται. Επειδή οι συναρτήσεις αυτές επιτρέπουν τη χρήση διαφόρων τύπων δεδομένων, εδώ είναι τέσσερις κατηγοριοποιήσεις ανάλογα με τον ποιον χρησιμοποιούν.

Πίνακας 1: Ορισμοί για τους διαφορετικούς τύπους των παραμέτρων

<type>	<t>
float	‘s’ ή ‘S’
double	‘d’ ή ‘D’
cuComplex	‘c’ ή ‘C’
cuDoubleComplex	‘z’ ή ‘Z’

Για τους τύπους complex που αναπαριστούν τους μιγαδικούς αριθμούς έχουμε τα $\text{Re}()$ και $\text{Im}()$ μέλη που αντιπροσωπεύουν το πραγματικό μέρος και το φανταστικό μέρος. Επειδή στους πραγματικούς αριθμούς δεν έχουμε το φανταστικό μέρος απλά θα το θεωρούμε σαν 0 και δεν θα συμπεριλαμβάνεται στην υπολογιστική πράξη.

Στη cuBLAS οι παράμετροι κλίμακας (scalars), όπως θα συναντήσουμε στις περιγραφές των συναρτήσεων σαν *alpha* και *beta*, που χρησιμοποιούνται εκτενώς στις περισσότερες λειτουργίες, μεταβιβάζονται με αναφορά είτε από τη μνήμη του host είτε από τη μνήμη της συσκευής (device). Είναι σημαντικό να σημειωθεί ότι, εάν υπάρχουν πολλαπλές παράμετροι κλίμακας σε μια κλήση συνάρτησης, η σωστή διαχείρισή τους είναι κρίσιμη για την αποφυγή σφαλμάτων και την εξασφάλιση της ορθής λειτουργίας του προγράμματος και συστήνεται να βρίσκονται σε χώρο μνήμης συσκευής για μείωση καθυστερήσεων .

Ακόμη αξίζει να αναφερθεί η υποστήριξη των batched kernels που επιτρέπουν την ταυτόχρονη εκτέλεση πολλαπλών ανεξάρτητων πράξεων, όπως ο πολλαπλασιασμός πινάκων ή η λύση συστημάτων εξισώσεων. Αντί να εκτελούνται αυτές οι πράξεις ξεχωριστά, τα batched kernels επεξεργάζονται σύνολα δεδομένων σε μία κλήση, μειώνοντας το κόστος επικοινωνίας και βελτιώνοντας σημαντικά την αποδοτικότητα. Οι υπολογισμοί οργανώνονται σε παρτίδες (batches) και εκτελούνται παράλληλα, αξιοποιώντας πλήρως τους διαθέσιμους πόρους της GPU για μέγιστη απόδοση.

Όλες οι συναρτήσεις cublas επιστρέφουν την τιμή αναφοράς σφάλματος με τύπο cublasStatus_t για τον έλεγχο σφαλμάτων. Η τιμή αυτή υποδεικνύει εάν η λειτουργία ολοκληρώθηκε επιτυχώς ή αν προέκυψε κάποιο σφάλμα (π.χ. ανεπάρκεια μνήμης, μη έγκυρη παράμετρος). Ανάλογα με το αποτέλεσμα που επιστρέφεται, ο χρήστης μπορεί

να καθορίσει κατάλληλα τις ενέργειές του, διευκολύνοντας τον εντοπισμό και τη διαχείριση λαθών στο πρόγραμμα.

2.1.4 cuBLAS Level-1 Functions

Εδώ θα περιγράψουμε τις συναρτήσεις του επιπέδου 1 (BLAS1), οι οποίες εκτελούν λειτουργίες που βασίζονται σε βαθμωτά μεγέθη και διανύσματα.

Οι συναρτήσεις που παρατίθενται εξυπηρετούν βασικές λειτουργίες γραμμικής άλγεβρας για διανύσματα. Οι συναρτήσεις *amax* και *amin* επιστρέφουν τον δείκτη του στοιχείου με τη μέγιστη ή τη μικρότερη απόλυτη τιμή αντίστοιχα, ενώ η *asum* υπολογίζει το άθροισμα των απόλυτων τιμών όλων των στοιχείων του διανύσματος. Η *copy* επιτρέπει την αντιγραφή των στοιχείων ενός διανύσματος σε ένα άλλο, ενώ η *swap* διευκολύνει την ανταλλαγή των στοιχείων μεταξύ δύο διανυσμάτων.

```
//Συνάρτηση amax
//Επιστρέφει τον δείκτη του στοιχείου με τη μέγιστη απόλυτη τιμή σε ένα διάνυσμα.
cublasStatus_t cublasIsamax(cublasHandle_t handle, int n, const float *x, int incx, int *result);
cublasStatus_t cublasIdamax(cublasHandle_t handle, int n, const double *x, int incx, int *result);
cublasStatus_t cublasIcamax(cublasHandle_t handle, int n, const cuComplex *x, int incx, int *result);
cublasStatus_t cublasIzamax(cublasHandle_t handle, int n, const cuDoubleComplex *x, int incx, int *result);

//Συνάρτηση amin
//Επιστρέφει τον δείκτη του στοιχείου με τη μικρότερη απόλυτη τιμή σε ένα διάνυσμα.
cublasStatus_t cublasIsamin(cublasHandle_t handle, int n, const float *x, int incx, int *result);
cublasStatus_t cublasIdamin(cublasHandle_t handle, int n, const double *x, int incx, int *result);
cublasStatus_t cublasIcamin(cublasHandle_t handle, int n, const cuComplex *x, int incx, int *result);
cublasStatus_t cublasIzamin(cublasHandle_t handle, int n, const cuDoubleComplex *x, int incx, int *result);

//Συνάρτηση asum
//Υπολογίζει το άθροισμα των απόλυτων τιμών των στοιχείων ενός διανύσματος.
cublasStatus_t cublasSasum(cublasHandle_t handle, int n, const float *x, int incx, float *result);
cublasStatus_t cublasDasum(cublasHandle_t handle, int n, const double *x, int incx, double *result);
cublasStatus_t cublasScasum(cublasHandle_t handle, int n, const cuComplex *x, int incx, float *result);
cublasStatus_t cublasDzasum(cublasHandle_t handle, int n, const cuDoubleComplex *x, int incx, double *result);

//Συνάρτηση copy
//Αντιγράφει τα στοιχεία ενός διανύσματος σε ένα άλλο διάνυσμα.
cublasStatus_t cublasScopy(cublasHandle_t handle, int n, const float *x, int incx, float *y, int incy);
cublasStatus_t cublasDcopy(cublasHandle_t handle, int n, const double *x, int incx, double *y, int incy);
cublasStatus_t cublasCcopy(cublasHandle_t handle, int n, const cuComplex *x, int incx, cuComplex *y, int incy);
cublasStatus_t cublasZcopy(cublasHandle_t handle, int n, const cuDoubleComplex *x, int incx, cuDoubleComplex *y, int incy);

//Συνάρτηση swap
//Ανταλλάσσει τα στοιχεία δυο διανυσμάτων x και y.
cublasStatus_t cublasSswap(cublasHandle_t handle, int n, float *x, int incx, float *y, int incy);
cublasStatus_t cublasDswap(cublasHandle_t handle, int n, double *x, int incx, double *y, int incy);
cublasStatus_t cublasCswap(cublasHandle_t handle, int n, cuComplex *x, int incx, cuComplex *y, int incy);
cublasStatus_t cublasZswap(cublasHandle_t handle, int n, cuDoubleComplex *x, int incx, cuDoubleComplex *y, int incy);
```

Εικόνα 1: Ορισμός Συναρτήσεων για απλές λειτουργίες με διανύσματα.

Εκτός από τις βασικές λειτουργίες αναζήτησης, αντιγραφής και κλιμάκωσης διανυσμάτων, σε αυτό το επίπεδο έχουμε συναρτήσεις για αριθμητικές πράξεις, όπως το εσωτερικό γινόμενο, τον υπολογισμό του Ευκλείδειου κανόνα και τον συνδυασμό διανυσμάτων μέσω γραμμικών πράξεων. Η *scal* πολλαπλασιάζει κάθε στοιχείο ενός

διανύσματος με έναν σταθερό αριθμό, καθιστώντας εύκολη την κλιμάκωσή του. Η *dot* υπολογίζει το εσωτερικό γινόμενο δύο διανυσμάτων, χρήσιμο για ελέγχους ορθογωνιότητας και υπολογισμούς προβολών. Η *nrm2* επιστρέφει τον Ευκλείδειο κανόνα (norm) ενός διανύσματος, ο οποίος μετρά το μέγεθός του. Η *axpy* υπολογίζει το άθροισμα δύο διανυσμάτων με τη μορφή $y = a * x + y$

```
//Συνάρτηση scal
//Πολλαπλασιάζει κάθε στοιχείο ενός διανύσματος x με έναν σταθερό αριθμό alpha.
cublasStatus_t cublasSscal(cublasHandle_t handle, int n, const float *alpha, float *x, int incx);
cublasStatus_t cublasDscal(cublasHandle_t handle, int n, const double *alpha, double *x, int incx);
cublasStatus_t cublasCscal(cublasHandle_t handle, int n, const cuComplex *alpha, cuComplex *x, int incx);
cublasStatus_t cublasCzscal(cublasHandle_t handle, int n, const float *alpha, cuComplex *x, int incx);
cublasStatus_t cublasZscal(cublasHandle_t handle, int n, const cuDoubleComplex *alpha, cuDoubleComplex *x, int incx);
cublasStatus_t cublasZdscal(cublasHandle_t handle, int n, const double *alpha, cuDoubleComplex *x, int incx);

//Συνάρτηση dot
cublasStatus_t cublasSdot (cublasHandle_t handle, int n, const float *x, int incx, const float *y, int incy,
    float *result);
cublasStatus_t cublasDdot (cublasHandle_t handle, int n, const double *x, int incx, const double *y, int incy,
    double *result);
cublasStatus_t cublasCdotu(cublasHandle_t handle, int n, const cuComplex *x, int incx, const cuComplex *y, int incy,
    cuComplex *result);
cublasStatus_t cublasCdotc(cublasHandle_t handle, int n, const cuComplex *x, int incx, const cuComplex *y, int incy,
    cuComplex *result);
cublasStatus_t cublasZdotu(cublasHandle_t handle, int n, const cuDoubleComplex *x, int incx, const cuDoubleComplex *y,
    int incy, cuDoubleComplex *result);
cublasStatus_t cublasZdotc(cublasHandle_t handle, int n, const cuDoubleComplex *x, int incx, const cuDoubleComplex *y,
    int incy, cuDoubleComplex *result);

//Συνάρτηση nrm2
cublasStatus_t cublasSnrm2(cublasHandle_t handle, int n, const float *x, int incx, float *result);
cublasStatus_t cublasDnrm2(cublasHandle_t handle, int n, const double *x, int incx, double *result);
cublasStatus_t cublasScnrm2(cublasHandle_t handle, int n, const cuComplex *x, int incx, float *result);
cublasStatus_t cublasDznrm2(cublasHandle_t handle, int n, const cuDoubleComplex *x, int incx, double *result);

//Συνάρτηση axpy
//y = alpha * x + y
cublasStatus_t cublasSaxpy(cublasHandle_t handle, int n, const float *alpha, const float *x, int incx, float *y, int incy);
cublasStatus_t cublasDaxpy(cublasHandle_t handle, int n, const double *alpha, const double *x, int incx, double *y, int incy);
cublasStatus_t cublasCaxpy(cublasHandle_t handle, int n, const cuComplex *alpha, const cuComplex *x, int incx, cuComplex *y,
    int incy);
cublasStatus_t cublasZaxpy(cublasHandle_t handle, int n, const cuDoubleComplex *alpha, const cuDoubleComplex *x, int incx,
    cuDoubleComplex *y, int incy);
```

Εικόνα 2: Ορισμός Συναρτήσεων που εκτελούν πράξεις σε διανύσματα.

2.1.5 cuBLAS Level-2 Function

Οι συναρτήσεις επιπέδου 2 (BLAS2), που εκτελούν πράξεις μεταξύ πίνακα και διανύσματος, έχουν σχεδιαστεί για να επιταχύνουν την επεξεργασία πινάκων και διανυσμάτων. Υπάρχουν διάφορες εκδοχές αυτών των συναρτήσεων, με εξειδικεύσεις για διαφορετικές διατάξεις και κατηγορίες πινάκων. Συγκεκριμένα, υπάρχουν ξεχωριστές συναρτήσεις για συμμετρικούς, ερμιτιανούς, τριγωνικούς και ταινιωτούς πίνακες, καθώς και για τις "packed" εκδοχές τους, όπου οι πίνακες αυτοί αποθηκεύονται σε πιο συνοπτική μορφή.

gbmv()

Η συνάρτηση `gbmv` υλοποιεί τον πολλαπλασιασμό ενός διανύσματος με έναν ταινιωτό πίνακα, εκτελώντας την πράξη $y = \alpha * \text{op}(A) * x + \beta * \text{op}(A)$ όπου $\text{op}(A)$ μπορεί να είναι ο αρχικός πίνακας ή ο μεταθετός του, ανάλογα με τις ρυθμίσεις του χρήστη. Ο πίνακας ορίζεται από τις διαστάσεις του (m και n) καθώς και από τον αριθμό των κάτω (kl) και πάνω (ku) διαγωνίων που περιέχει. Αυτή η λειτουργία είναι ιδιαίτερα χρήσιμη σε εφαρμογές όπου οι πίνακες παρουσιάζουν ειδική δομή, μειώνοντας τον αριθμό των απαραίτητων υπολογισμών και την κατανάλωση μνήμης. Για κάθε τύπο δεδομένων υπάρχουν αντίστοιχες υλοποιήσεις: για πραγματικές τιμές με μονή ακρίβεια (`cublasSgbmv`), για πραγματικές τιμές με διπλή ακρίβεια (`cublasDgbmv`) και για μιγαδικούς αριθμούς (`cublasCgbmv` για μονή ακρίβεια και `cublasZgbmv` για διπλή ακρίβεια).

```
//gbmv - banded matrix-vector multiplication
cublasStatus_t cublasSgbmv(cublasHandle_t handle, cublasOperation_t trans, int m, int n, int kl, int ku,
    const float *alpha, const float *A, int lda,
    const float *x, int incx, const float *beta, float *y, int incy);
cublasStatus_t cublasDgbmv(cublasHandle_t handle, cublasOperation_t trans, int m, int n, int kl, int ku,
    const double *alpha, const double *A, int lda,
    const double *x, int incx, const double *beta, double *y, int incy);
cublasStatus_t cublasCgbmv(cublasHandle_t handle, cublasOperation_t trans, int m, int n, int kl, int ku,
    const cuComplex *alpha, const cuComplex *A, int lda,
    const cuComplex *x, int incx, const cuComplex *beta, cuComplex *y, int incy);
cublasStatus_t cublasZgbmv(cublasHandle_t handle, cublasOperation_t trans, int m, int n, int kl, int ku,
    const cuDoubleComplex *alpha, const cuDoubleComplex *A, int lda,
    const cuDoubleComplex *x, int incx, const cuDoubleComplex *beta, cuDoubleComplex *y, int incy);
```

Εικόνα 3: Ορισμός για την συνάρτηση `gbmv`.

Εκτός από τη `gbmv`, η cuBLAS παρέχει συναρτήσεις για τον πολλαπλασιασμό διανυσμάτων με άλλους ειδικούς τύπους πινάκων:

- *sbmv* (Symmetric Band Matrix-Vector multiplication): Εκτελεί τον πολλαπλασιασμό διανύσματος με συμμετρικό ταινιωτό πίνακα.
- *tbmv* (Triangular Band Matrix-Vector multiplication): Εκτελεί τον πολλαπλασιασμό διανύσματος με τριγωνικό ταινιωτό πίνακα.
- *hbmv* (Hermitian Band Matrix-Vector multiplication): Εκτελεί τον πολλαπλασιασμό διανύσματος με ερμιτιανό ταινιωτό πίνακα.

gemv()

Η συνάρτηση εκτελεί πολλαπλασιασμό πίνακα με διάνυσμα. Από τον ορισμό της συνάρτησης έχουμε τον πίνακα A μεγέθους $m \times n$ που είναι αποθηκευμένος σε column-major μορφή στην μνήμη της GPU.

Η πράξη που εκτελεί είναι η $Y = \alpha \times \text{op}(A) \times x + \beta \times y$ όπου το $\text{op}(A)$ μπορεί να είναι ο ίδιος ο πίνακας A ή ο μεταθετός του, ανάλογα με τις ανάγκες της εφαρμογής. Συγκεκριμένα, η `cublasSgemv` χρησιμοποιείται για δεδομένα πραγματικού τύπου με μονή ακρίβεια, ενώ η `cublasDgemv` εξυπηρετεί δεδομένα με διπλή ακρίβεια. Αντίστοιχα, οι `cublasCgemv` και `cublasZgemv` επεκτείνουν αυτή τη λειτουργία για μιγαδικούς αριθμούς, καλύπτοντας μονή και διπλή ακρίβεια αντίστοιχα.

```
//gemv - matrix-vector multiplication
cublasStatus_t cublasSgemv(cublasHandle_t handle, cublasOperation_t trans, int m, int n,
    const float *alpha, const float *A, int lda,
    const float *x, int incx, const float *beta, float *y, int incy);

cublasStatus_t cublasDgemv(cublasHandle_t handle, cublasOperation_t trans, int m, int n,
    const double *alpha, const double *A, int lda,
    const double *x, int incx, const double *beta, double *y, int incy);

cublasStatus_t cublasCgemv(cublasHandle_t handle, cublasOperation_t trans, int m, int n,
    const cuComplex *alpha, const cuComplex *A, int lda,
    const cuComplex *x, int incx, const cuComplex *beta, cuComplex *y, int incy);

cublasStatus_t cublasZgemv(cublasHandle_t handle, cublasOperation_t trans, int m, int n,
    const cuDoubleComplex *alpha, const cuDoubleComplex *A, int lda,
    const cuDoubleComplex *x, int incx, const cuDoubleComplex *beta, cuDoubleComplex *y, int incy);
```

Εικόνα 4: Ορισμός συνάρτησης `gemv`.

Εκτός από τη `gemv`, η cuBLAS παρέχει συναρτήσεις για τον πολλαπλασιασμό διανυσμάτων με άλλους ειδικούς τύπους πινάκων. Η `hemv` (Hermitian Matrix-Vector multiplication) που εκτελεί τον πολλαπλασιασμό ενός διανύσματος με έναν ερμιτιανό πίνακα, αξιοποιώντας τη συμμετρία του ως προς τη συζυγώς μεταθεμένη του. Η `gemvBatched` (Batched General Matrix-Vector multiplication) είναι μια παραλλαγή της `gemv` που επιτρέπει την ταυτόχρονη επεξεργασία πολλαπλών ανεξάρτητων πράξεων πολλαπλασιασμού πίνακα-διανύσματος σε μία μόνο κλήση, βελτιώνοντας έτσι την αποδοτικότητα στις περιπτώσεις όπου πρέπει να εκτελεστούν πολλές παρόμοιες πράξεις.

ger()

Η συνάρτηση εκτελεί την ενημέρωση 1ου βαθμού ενός πίνακα δηλαδή προσθέτει το γινόμενο ενός διανύσματος με το διάνυσμα μεταφοράς του (transpose), σε έναν πίνακα.

Για έναν πίνακα A , έναν βαθμωτό συντελεστή α , και δυο διανύσματα x, y , η συνάρτηση

υπολογίζει τον τύπο:
$$A = \begin{cases} \alpha xy^T + A & \text{if ger(), geru() is called} \\ \alpha xy^H + A & \text{if gerc() is called} \end{cases}$$

```

//ger - rank-1 update
cublasStatus_t cublasSger(cublasHandle_t handle, int m, int n, const float *alpha, const float *x, int incx,
    const float *y, int incy, float *A, int lda);

cublasStatus_t cublasDger(cublasHandle_t handle, int m, int n, const double *alpha, const double *x, int incx,
    const double *y, int incy, double *A, int lda);

cublasStatus_t cublasCgeru(cublasHandle_t handle, int m, int n, const cuComplex *alpha, const cuComplex *x, int incx,
    const cuComplex *y, int incy, cuComplex *A, int lda);

cublasStatus_t cublasCgerc(cublasHandle_t handle, int m, int n, const cuComplex *alpha, const cuComplex *x, int incx,
    const cuComplex *y, int incy, cuComplex *A, int lda);

cublasStatus_t cublasZgeru(cublasHandle_t handle, int m, int n, const cuDoubleComplex *alpha, const cuDoubleComplex *x, int incx,
    const cuDoubleComplex *y, int incy, cuDoubleComplex *A, int lda);

cublasStatus_t cublasZgerc(cublasHandle_t handle, int m, int n, const cuDoubleComplex *alpha, const cuDoubleComplex *x, int incx,
    const cuDoubleComplex *y, int incy, cuDoubleComplex *A, int lda);

```

Εικόνα 5: Ορισμός συνάντησης ger.

spr2()

Η συνάρτηση εκτελεί ενημέρωση δευτέρου βαθμού συμμετρικού πίνακα

$$A = \alpha(xy^T + yx^T) + A$$

οπού A είναι συμμετρικός πίνακας $n \times n$ διαστάσεων και x διάνυσμα με α συντελεστή.

```

//spr2 - packed symmetric rank-2 update
cublasStatus_t cublasSspr2(cublasHandle_t handle, cublasFillMode_t uplo, int n, const float *alpha,
    const float *x, int incx, const float *y, int incy, float *AP);
cublasStatus_t cublasDspr2(cublasHandle_t handle, cublasFillMode_t uplo, int n, const double *alpha,
    const double *x, int incx, const double *y, int incy, double *AP);

```

Εικόνα 6: Ορισμός συνάρτησης spr2.

tbsv()

Η συνάρτηση λύνει το σύστημα $op(A)x=b$ με A τριγωνικό ταινιωτό πίνακα και x, b διανύσματα. Όταν η διαδικασία επίλυσης ολοκληρωθεί, η λύση x αποθηκεύεται στον ίδιο χώρο μνήμης που χρησιμοποιούσε προηγουμένως το διάνυσμα (ή ο πίνακας) των δεξιών μελών b . Μέσω των παραμέτρων της, όπως την *uplo* που καθορίζει αν ο πίνακας είναι ανώ ή κάτω τριγωνικός, την *trans* που επιλέγει αν θα χρησιμοποιηθεί ο αρχικός ή ο μεταθετός (ή συζυγώς μεταθετός) πίνακας, και την *diag* που υποδεικνύει αν ο πίνακας είναι μονόδιαγώνιος, ο χρήστης μπορεί να προσαρμόσει την επίλυση ανάλογα με τις απαιτήσεις της εφαρμογής. Επιπλέον, ορίζονται οι διαστάσεις του πίνακα (n) και ο αριθμός των διαγωνίων στοιχείων που αποτελούν τη ζώνη (k), ενώ το διάνυσμα x που αρχικά περιέχει το δεξί μέλος της εξίσωσης τροποποιείται in-place ώστε να περιέχει τη λύση.

```

//tbsv - solves the triangular banded linear system with a single right-hand-side
cublasStatus_t cublasStbsv(cublasHandle_t handle, cublasFillMode_t uplo, cublasOperation_t trans, cublasDiagType_t diag,
|   int n, int k, const float *A, int lda, float *x, int incx);
cublasStatus_t cublasDtbsv(cublasHandle_t handle, cublasFillMode_t uplo, cublasOperation_t trans, cublasDiagType_t diag,
|   int n, int k, const double *A, int lda, double *x, int incx);
cublasStatus_t cublasCtbsv(cublasHandle_t handle, cublasFillMode_t uplo, cublasOperation_t trans, cublasDiagType_t diag,
|   int n, int k, const cuComplex *A, int lda, cuComplex *x, int incx);
cublasStatus_t cublasZtbsv(cublasHandle_t handle, cublasFillMode_t uplo, cublasOperation_t trans, cublasDiagType_t diag,
|   int n, int k, const cuDoubleComplex *A, int lda, cuDoubleComplex *x, int incx);

```

Εικόνα 7: Ορισμός συνάρτησης tbsv.

Παρόμοιες συναρτήσεις υπάρχουν για διαφορετικές αναπαραστάσεις τριγωνικών πινάκων, όπως η trsv(), η οποία επιλύει το σύστημα όταν ο τριγωνικός πίνακας αποθηκεύεται σε packed format, και η trsn(), η οποία λειτουργεί με πλήρεις τριγωνικούς πίνακες. Αυτές οι συναρτήσεις επιτρέπουν την επιλογή της κατάλληλης μορφής αποθήκευσης του πίνακα, προσφέροντας μεγαλύτερη ευελιξία και αποδοτικότητα σε εφαρμογές που εκμεταλλεύονται τις ιδιότητες των τριγωνικών συστημάτων γραμμικών εξισώσεων.

gemvBatched()

Η συνάρτηση gemvBatched επιτρέπει την εκτέλεση πράξεων πολλαπλασιασμού πίνακα-διανύσματος σε ομάδες (batches) ανεξάρτητων συνόλων δεδομένων, μειώνοντας έτσι το overhead που προκύπτει από την εκτέλεση κάθε πράξης ξεχωριστά. Με αυτήν τη λειτουργία, επιτυγχάνεται η ταυτόχρονη επεξεργασία πολλών μικρών έως μεσαίων πράξεων γραμμικής άλγεβρας, αξιοποιώντας πλήρως την παραλληλία της GPU και προσφέροντας υψηλές επιδόσεις σε εφαρμογές που απαιτούν επεξεργασία μεγάλου αριθμού ανεξάρτητων πράξεων.

```

//gemvBatched - matrix-vector multiplication of a batch of matrices and vectors
cublasStatus_t cublasSgemvBatched(cublasHandle_t handle, cublasOperation_t trans, int m, int n,
    const float *alpha, const float *const Aarray[], int lda, const float *const xarray[], int incx,
    const float *beta, float *const yarray[], int incy, int batchSize);

cublasStatus_t cublasDgemvBatched(cublasHandle_t handle, cublasOperation_t trans, int m, int n,
    const double *alpha, const double *const Aarray[], int lda, const double *const xarray[], int incx,
    const double *beta, double *const yarray[], int incy, int batchSize);

cublasStatus_t cublasCgemvBatched(cublasHandle_t handle, cublasOperation_t trans, int m, int n,
    const cuComplex *alpha, const cuComplex *const Aarray[], int lda, const cuComplex *const xarray[], int incx,
    const cuComplex *beta, cuComplex *const yarray[], int incy, int batchSize);

cublasStatus_t cublasZgemvBatched(cublasHandle_t handle, cublasOperation_t trans, int m, int n,
    const cuDoubleComplex *alpha, const cuDoubleComplex *const Aarray[], int lda, const cuDoubleComplex *const xarray[], int incx,
    const cuDoubleComplex *beta, cuDoubleComplex *const yarray[], int incy, int batchSize);

#ifdef __cplusplus
cublasStatus_t cublasHSHgemvBatched(cublasHandle_t handle, cublasOperation_t trans, int m, int n,
    const float *alpha, const __half *const Aarray[], int lda, const __half *const xarray[], int incx,
    const float *beta, __half *const yarray[], int incy, int batchSize);

cublasStatus_t cublasHSSgemvBatched(cublasHandle_t handle, cublasOperation_t trans, int m, int n,
    const float *alpha, const __half *const Aarray[], int lda, const __half *const xarray[], int incx,
    const float *beta, float *const yarray[], int incy, int batchSize);

cublasStatus_t cublasTSTgemvBatched(cublasHandle_t handle, cublasOperation_t trans, int m, int n,
    const float *alpha, const __nv_bfloat16 *const Aarray[], int lda, const __nv_bfloat16 *const xarray[], int incx,
    const float *beta, __nv_bfloat16 *const yarray[], int incy, int batchSize);

cublasStatus_t cublasTSSgemvBatched(cublasHandle_t handle, cublasOperation_t trans, int m, int n,
    const float *alpha, const __nv_bfloat16 *const Aarray[], int lda, const __nv_bfloat16 *const xarray[], int incx,
    const float *beta, float *const yarray[], int incy, int batchSize);
#endif

```

Εικόνα 8: Ορισμός της συνάρτησης gemvBatched.

2.1.6 cuBLAS Level-3 Function

Η κατηγορία αυτή περιλαμβάνει συναρτήσεις που εκτελούν πράξεις μεταξύ πινάκων, όπως ο πολλαπλασιασμός (GEMM), η επίλυση συστημάτων γραμμικών εξισώσεων και άλλες πράξεις που αφορούν μεγάλες ομάδες δεδομένων.

gemm()

Η συνάρτηση εκτελεί πολλαπλασιασμό πίνακα με πίνακα όπου alpha και beta είναι βαθμωτά και οι πίνακες A, B, C είναι αποθηκευμένοι σε column-major μορφή στη μνήμη της GPU με διαστάσεις op(A) (m x k), op(B) (k x n) και C (m x n).

$$C = \alpha \times \text{op}(A) \times \text{op}(B) + \beta C.$$

```

//gemm - matrix-matrix multiplication
cublasStatus_t cublasSgemm(cublasHandle_t handle, cublasOperation_t transa, cublasOperation_t transb, int m, int n, int k,
    const float *alpha, const float *A, int lda, const float *B, int ldb, const float *beta, float *C, int ldc);

cublasStatus_t cublasDgemm(cublasHandle_t handle, cublasOperation_t transa, cublasOperation_t transb, int m, int n, int k,
    const double *alpha, const double *A, int lda, const double *B, int ldb, const double *beta, double *C, int ldc);

cublasStatus_t cublasCgemm(cublasHandle_t handle, cublasOperation_t transa, cublasOperation_t transb, int m, int n, int k,
    const cuComplex *alpha, const cuComplex *A, int lda, const cuComplex *B, int ldb, const cuComplex *beta,
    cuComplex *C, int ldc);

cublasStatus_t cublasZgemm(cublasHandle_t handle, cublasOperation_t transa, cublasOperation_t transb, int m, int n, int k,
    const cuDoubleComplex *alpha, const cuDoubleComplex *A, int lda, const cuDoubleComplex *B, int ldb,
    const cuDoubleComplex *beta, cuDoubleComplex *C, int ldc);

cublasStatus_t cublasHgemm(cublasHandle_t handle, cublasOperation_t transa, cublasOperation_t transb, int m, int n, int k,
    const __half *alpha, const __half *A, int lda, const __half *B, int ldb, const __half *beta, __half *C, int ldc);

```

Εικόνα 9: Ορισμός της συνάρτησης gemm.

Εκτός από τη βασική λειτουργία `gemm`, η βιβλιοθήκη παρέχει παραλλαγές που προσαρμόζονται σε διαφορετικές ανάγκες εφαρμογών. Έχουμε την `gemm3m` υλοποιεί τον πολλαπλασιασμό μιγαδικών μητρών, χρησιμοποιώντας τον αλγόριθμο μείωσης πολυπλοκότητας του Gauss. Για επεξεργασία πολλαπλών δεδομένων σε ομοιόμορφη ομάδες η `gemmBatched` επιτρέπει την ταυτόχρονη εκτέλεση ανεξάρτητων πράξεων `gemm` σε μία κλήση, μειώνοντας έτσι το overhead των επαναλαμβανόμενων κλήσεων και αξιοποιώντας πλήρως την παραλληλία της GPU.

Η ομάδα θεωρείται "ομοιόμορφη", όταν όλοι οι πίνακες του batch έχουν τις ίδιες διαστάσεις (m, n, k), τις ίδιες κύριες διαστάσεις (lda, ldb, ldc) και τους ίδιους τύπους μεταθέσεων (`transa, transb`) για τις αντίστοιχες μητρικές A, B και C .

Σε παρόμοιο πνεύμα, η `gemmStridedBatched` διαχειρίζεται ομάδες πράξεων `gemm` όπου τα δεδομένα κάθε πράξης βρίσκονται σε συνεχόμενες αλλά απομακρυσμένες περιοχές μνήμης, με σταθερό βήμα (`stride`) μεταξύ τους, ενώ η `gemmGroupBatched` επιτρέπει την ομαδοποίηση και ταυτόχρονη επεξεργασία πράξεων που ανήκουν σε διαφορετικές ομάδες, προσφέροντας επιπλέον ευελιξία σε πολύπλοκα σενάρια.

Επιπρόσθετα, στο ίδιο επίπεδο της cuBLAS υπάρχουν συναρτήσεις που επεκτείνουν τη λειτουργικότητα του `gemm` για ειδικούς τύπους πινάκων. Η `symm` και η `hemm` υλοποιούν τον πολλαπλασιασμό διανύσματος με συμμετρικό ή ερμιτιανό πίνακα, αντίστοιχα, αξιοποιώντας τις ιδιότητες συμμετρίας ή ερμιτιανότητας για βελτιστοποίηση των υπολογισμών και τέλος, η `trmm` (triangular matrix-matrix multiplication) πραγματοποιεί τον πολλαπλασιασμό πίνακα με τριγωνικό πίνακα.

trsm()

Η συνάρτηση `trsm` χρησιμοποιείται για την επίλυση τριγωνικών συστημάτων γραμμικών εξισώσεων, όπου ένας τριγωνικός πίνακας A παίζει το ρόλο του συντελεστή. Συγκεκριμένα, αυτή η συνάρτηση λύνει το σύστημα της μορφής:

$$\text{op}(A) * X = \alpha * B \text{ if side == CUBLAS_SIDE_LEFT}$$

$$X * \text{op}(A) = \alpha * B \text{ if side == CUBLAS_SIDE_RIGHT}$$

όπου A είναι τριγωνικός πίνακας, X και B είναι πίνακες $m \times n$ διαστάσεων και α βαθμωτό πολλαπλασιασμού.

```

//trsm - solves the triangular linear system with multiple right-hand-sides
cublasStatus_t cublasStrsm(cublasHandle_t handle, cublasSideMode_t side, cublasFillMode_t uplo,
    cublasOperation_t trans, cublasDiagType_t diag, int m, int n,
    const float *alpha, const float *A, int lda, float *B, int ldb);

cublasStatus_t cublasDtrsm(cublasHandle_t handle, cublasSideMode_t side, cublasFillMode_t uplo,
    cublasOperation_t trans, cublasDiagType_t diag, int m, int n,
    const double *alpha, const double *A, int lda, double *B, int ldb);

cublasStatus_t cublasCtrsm(cublasHandle_t handle, cublasSideMode_t side, cublasFillMode_t uplo,
    cublasOperation_t trans, cublasDiagType_t diag, int m, int n,
    const cuComplex *alpha, const cuComplex *A, int lda, cuComplex *B, int ldb);

cublasStatus_t cublasZtrsm(cublasHandle_t handle, cublasSideMode_t side, cublasFillMode_t uplo,
    cublasOperation_t trans, cublasDiagType_t diag, int m, int n, const cuDoubleComplex *alpha,
    const cuDoubleComplex *A, int lda, cuDoubleComplex *B, int ldb);

```

Εικόνα 10: Ορισμός της συνάρτησης trsm.

Για περιπτώσεις όπου πρέπει να επιλυθούν πολλά ανεξάρτητα τριγωνικά συστήματα ταυτόχρονα, η cuBLAS παρέχει την παραλλαγή trsmBatched(). Με τη χρήση αυτής της συνάρτησης, επιτρέπεται η ταυτόχρονη επίλυση μιας ομάδας συστημάτων, κάτι που μειώνει το overhead των πολλαπλών ξεχωριστών κλήσεων.

2.2 Βιβλιοθήκη CUSPARSE

Η βιβλιοθήκη cuSPARSE της NVIDIA είναι μια εξειδικευμένη βιβλιοθήκη για την εκτέλεση λειτουργιών γραμμικής άλγεβρας σε αραιούς πίνακες, αξιοποιώντας την υπολογιστική ισχύ των GPU μέσω του μοντέλου CUDA. Είναι μέρος του CUDA Toolkit και προσβάσιμη μέσω των γλωσσών C και C++.

2.2.1 Εισαγωγή στη CUSPARSE

Η cuSPARSE περιλαμβάνει δύο σετ API, *Legacy* και *Generic*. Το *Legacy* API που βασίζεται στο πρότυπο Sparse BLAS και παρέχει ένα περιορισμένο σύνολο λειτουργιών. Αν και δεν αναπτύσσεται πλέον, εξακολουθεί να διατηρεί ορισμένες χρήσιμες λειτουργίες για εφαρμογές που απαιτούν συγκεκριμένες πράξεις γραμμικής άλγεβρας. Από την άλλη, το *Generic* API αποτελεί τη σύγχρονη και πιο ευέλικτη εκδοχή της βιβλιοθήκης, προσφέροντας εκτεταμένη υποστήριξη για πιο σύνθετες λειτουργίες, επιτρέποντας την αποτελεσματική διαχείριση και επεξεργασία αραιών πινάκων. Συγκεκριμένα περιλαμβάνει πλήθος συναρτήσεων για πράξεις όπως:

- Πράξεις μεταξύ αραιού διανύσματος και πυκνού διανύσματος.
- Πράξεις μεταξύ πυκνού πίνακα και αραιού διανύσματος.
- Πράξεις μεταξύ αραιού πίνακα και πυκνού διανύσματος.

- Πράξεις μεταξύ αραιού πίνακα και πυκνού πίνακα.
- Πράξεις μεταξύ αραιών πινάκων.
- Πολλαπλασιασμό πυκνών πινάκων με έξοδο έναν αραιό πίνακα.

2.2.2 Αναπαράσταση δεδομένων

Η cuSPARSE παρέχει μια σειρά από ισχυρές δυνατότητες για τη διαχείριση και την επεξεργασία αραιών πινάκων, επιτρέποντας την αποδοτική αποθήκευση και εκτέλεση μαθηματικών πράξεων σε αυτούς. Υποστηρίζει διάφορες μορφές αναπαράστασης αραιών πινάκων, όπως CSR (Compressed Sparse Row), CSC (Compressed Sparse Column), BSR (Block Sparse Row) και COO (Coordinate Format), επιτρέποντας την επιλογή της κατάλληλης δομής ανάλογα με τις απαιτήσεις της εφαρμογής. Επιπλέον, διαθέτει συναρτήσεις που περιλαμβάνουν μετατροπές μεταξύ διαφορετικών μορφών, όπως από CSR σε COO, καθώς και βασικές πράξεις όπως άθροισμα, πολλαπλασιασμό και μετασχηματισμούς αραιών πινάκων.

Διανύσματα

Η cuSPARSE για την αναπαράσταση αραιών διανυσμάτων χρησιμοποιεί δυο πίνακες, ένας για τις θέσεις αυτών των μη μηδενικών τιμών στο αρχικό πυκνό διάνυσμα και ένας για τις τιμές τους στις αντίστοιχες θέσεις.

Πίνακες

CSR (Compressed Sparse Row): Είναι ένας αποδοτικός τρόπος αναπαράστασης αραιών πινάκων, όπου αποθηκεύονται μόνο οι μη μηδενικές τιμές και οι θέσεις τους κατά γραμμές, μειώνοντας σημαντικά την ανάγκη χώρου αποθήκευσης. Η αναπαράσταση CSR χρησιμοποιεί τρεις πίνακες. *Values*: Περιέχει όλες τις μη μηδενικές τιμές του πίνακα, αποθηκευμένες κατά σειρά εμφάνισης από την πρώτη έως την τελευταία γραμμή. *Column Indices*: Αποθηκεύει τις αντίστοιχες στήλες για κάθε μη μηδενική τιμή στον πίνακα *Values*, δείχνοντας τη θέση της τιμής αυτής μέσα στη γραμμή. *Row Offsets*: Αυτός ο πίνακας κρατά το σημείο έναρξης κάθε γραμμής στον πίνακα *Values*. Το στοιχείο *Row Offsets[i]* δείχνει τη θέση του πρώτου στοιχείου της γραμμής *i* στον πίνακα *Values*, ενώ το *Row Offsets[i+1]* δείχνει το τέλος αυτής της γραμμής.

CSC (Compressed Sparse Column): Η CSC είναι παρόμοια με τη μορφή CSR, αλλά αποθηκεύει τον πίνακα κατά στήλες αντί για κατά γραμμές. Η κύρια διαφορά είναι ότι στη μορφή CSC ο πίνακας *Column Offsets* δείχνει το σημείο έναρξης κάθε στήλης στον

πίνακα Values και ότι ο πίνακας Row Indices κρατά τις γραμμές στις οποίες βρίσκονται οι αντίστοιχες μη μηδενικές τιμές.

COO (Coordinate): Η μορφή COO (Coordinate) είναι μια μέθοδος αναπαράστασης αραιών πινάκων, όπου αποθηκεύονται μόνο οι μη μηδενικές τιμές μαζί με τις συντεταγμένες τους. Σε αυτήν τη μορφή, ο πίνακας δεν αποθηκεύει όλα τα στοιχεία αλλά μόνο τις απαραίτητες πληροφορίες για τις θέσεις και τις τιμές των μη μηδενικών στοιχείων. Κατάλληλη για πίνακες με τυχαία διασπορά μηδενικών και μη μηδενικών τιμών.

Sliced ELLPACK: Η μορφή Sliced ELLPACK (SELL) είναι μια βελτιστοποιημένη μέθοδος αναπαράστασης αραιών πινάκων, που επιτρέπει σημαντική αύξηση της απόδοσης σε περιπτώσεις όπου οι γραμμές του πίνακα έχουν παρόμοιο αριθμό μηδενικών στοιχείων. Διαιρεί τον πίνακα σε κομμάτια (slices) με συγκεκριμένο αριθμό γραμμών (sliceSize), όπως ορίζεται από τον χρήστη. Για κάθε κομμάτι, υπολογίζεται το μέγιστο πλήθος μη μηδενικών στοιχείων ανά γραμμή, και κάθε γραμμή μέσα στο slice συμπληρώνεται με την τιμή -1 ως padding, ώστε όλες οι γραμμές να έχουν το ίδιο μήκος.

Block Sparse Row (BSR): Είναι παρόμοια με τη μορφή CSR με τη διαφορά ότι αντί να αποθηκεύει μεμονωμένες τιμές για κάθε στοιχείο του πίνακα, αποθηκεύει δισδιάστατα μπλοκ στοιχείων. Κάθε "κελί" αντιπροσωπεύει ένα block του αρχικού πίνακα και η διάταξη και τα μεγέθη αυτών των blocks ορίζονται από τον χρήστη.

Blocked ELLPACK (Blocked ELL): Είναι μια εξελιγμένη εκδοχή της κλασικής μορφής ELLPACK, που οργανώνει τα μη μηδενικά στοιχεία ενός αραιού πίνακα σε μπλοκ, διευκολύνοντας την επεξεργασία δεδομένων με σταθερό ή παρόμοιο αριθμό μηδενικών στοιχείων ανά γραμμή. Σε αυτή τη μορφή, ο πίνακας χωρίζεται σε δισδιάστατα μπλοκ συγκεκριμένου μεγέθους (όπως 2x2 ή 4x4), τα οποία ορίζονται από τον χρήστη. Κάθε γραμμή έχει σταθερό μέγιστο αριθμό μηδενικών μπλοκ, και σε περιπτώσεις όπου οι γραμμές δεν διαθέτουν αρκετά μηδενικά στοιχεία για να καλύψουν αυτό το μέγιστο, χρησιμοποιείται padding για να διατηρηθεί η συμμετρία της δομής. Αυτή η σταθερότητα στον αριθμό των μπλοκ ανά γραμμή μειώνει την απόκλιση του φόρτου εργασίας, βελτιώνοντας την αποδοτικότητα στη χρήση της μνήμης και επιτρέποντας καλύτερη παράλληλη επεξεργασία σε GPUs. Η αποθήκευση των δεδομένων των μπλοκ μπορεί να είναι είτε σε διάταξη κατά στήλης (column-major) είτε κατά γραμμής (row-major), ανάλογα με τις ανάγκες πρόσβασης της GPU, γεγονός που

επιτρέπει αποτελεσματικότερη χρήση των threads και ταχύτερη επεξεργασία για προβλήματα με σχετικά ομοιόμορφη πυκνότητα μη μηδενικών στοιχείων.

2.2.3 Πως χρησιμοποιείται η βιβλιοθήκη

Το πρώτο βήμα είναι η αρχικοποίηση της βιβλιοθήκης cuSPARSE μέσω της συνάρτησης `cusparsCreate()`, η οποία δημιουργεί μια λαβή (handle) στην εκάστοτε συσκευή. Αυτή η λαβή είναι μοναδική για το thread που την καλεί και θα χρησιμοποιηθεί για όλες τις επόμενες κλήσεις της βιβλιοθήκης.

Αφού αρχικοποιηθεί η βιβλιοθήκη, ο προγραμματιστής πρέπει να προετοιμάσει τα δεδομένα. Για τις πράξεις με αραιούς πίνακες, είναι σημαντικό να οριστούν οι πίνακες στη μνήμη της GPU, χρησιμοποιώντας τη `cudaMalloc()` και επιπλέον να χρησιμοποιηθούν οι συναρτήσεις για τις διάφορες αναπαραστάσεις αραιών πινάκων, όπως *Compressed Sparse Row (CSR)*, *Compressed Sparse Column (CSC)* και *Coordinate (COO)*, επιτρέποντας αποδοτική αποθήκευση και επεξεργασία δεδομένων. Μετά την αποθήκευση των πινάκων στη μνήμη της GPU, ο προγραμματιστής μπορεί να χρησιμοποιήσει τις συναρτήσεις της cuSPARSE για πράξεις όπως ο πολλαπλασιασμός πίνακα-διανύσματος (SpMV), πίνακα-πίνακα (SpMM), η μετατροπή μεταξύ διαφορετικών μορφών αραιών πινάκων, καθώς και η ανάλυση και επίλυση γραμμικών συστημάτων.

Αποδέσμευση Πόρων στην cuSPARSE

Τέλος μόλις ολοκληρωθούν οι απαραίτητες πράξεις κατά τη χρήση της βιβλιοθήκης cuSPARSE, είναι σημαντικό να αποδεσμεύονται όλοι οι πόροι που χρησιμοποιούνται κατά την εκτέλεση των πράξεων. Κάθε descriptor που έχει δημιουργηθεί για αραιούς ή πυκνούς πίνακες πρέπει να καταστραφεί, καθώς και η μνήμη που χρησιμοποιείται από τις συναρτήσεις της βιβλιοθήκης. Αυτό εξασφαλίζει την αποφυγή διαρροών μνήμης και βελτιώνει την αποδοτικότητα.

Η βιβλιοθήκη cuSPARSE μπορεί να τερματιστεί πλήρως και να απελευθερώσει όλους τους δεσμευμένους πόρους καλώντας τη συνάρτηση `cusparsDestroy()` με μονή παράμετρο το αναγνωριστικό που είχαμε δημιουργήσει (handle). Αυτή η συνάρτηση καταστρέφει τον χειριστή (handle) της cuSPARSE και αποδεσμεύει όλους τους πόρους που χρησιμοποιούνται από τη βιβλιοθήκη. Είναι κρίσιμο να καλείται αυτή η συνάρτηση στο τέλος κάθε εφαρμογής που χρησιμοποιεί cuSPARSE, προκειμένου να εξασφαλιστεί η σωστή αποδέσμευση των πόρων και η αποφυγή προβλημάτων με τη διαχείριση της μνήμης.

2.2.4 Συναρτήσεις της CUSPARSE

Απλές λειτουργίες για διανύσματα

Οι συναρτήσεις *cusparseGather* και *cusparseScatter* εκτελούν βασικές πράξεις με διανύσματα για διαχείριση αραιών δεδομένων. Αυτές οι λειτουργίες επιταχύνουν την επεξεργασία και τη μετακίνηση στοιχείων διανυσμάτων, επιτρέποντας γρήγορη εκτέλεση σε GPU.

Sparse Matrix-Vector Multiplication (SpMV)

Η συνάρτηση *cusparseSpMV* παρέχει έναν αποδοτικό τρόπο για τον πολλαπλασιασμό ενός αραιού πίνακα με ένα διάνυσμα, μια πράξη κρίσιμη για πολλές εφαρμογές επιστημονικών υπολογισμών και μηχανικής μάθησης. Η υλοποίηση αυτής της λειτουργίας χωρίζεται σε τρία βασικά βήματα: Αρχικά, η *cusparseSpMV_bufferSize()* υπολογίζει το απαιτούμενο μέγεθος μνήμης για έναν εξωτερικό buffer, ο οποίος θα χρησιμοποιηθεί κατά τη διάρκεια της επεξεργασίας. Στη συνέχεια, η *cusparseSpMV_preprocess()* προετοιμάζει τα δεδομένα εφαρμόζοντας βελτιστοποιημένους αλγορίθμους για την αποτελεσματική διαχείριση των αραιών δομών δεδομένων. Τέλος, η *cusparseSpMV* εκτελεί την πράξη πολλαπλασιασμού σύμφωνα με τη σχέση $Y = \alpha \times \text{op}(A) \times X + \beta \times Y$.

```
cusparseStatus_t cusparseSpMV_bufferSize(cusparseHandle_t handle, cusparseOperation_t opA,
    const void* alpha, cusparseConstSpMatDescr_t matA, // non-const descriptor supported
    cusparseConstDnVecDescr_t vecX, // non-const descriptor supported
    const void* beta, cusparseDnVecDescr_t vecY,
    cudaDataType computeType, cusparseSpMValg_t alg, size_t* bufferSize);

cusparseStatus_t cusparseSpMV_preprocess(cusparseHandle_t handle, cusparseOperation_t opA,
    const void* alpha, cusparseConstSpMatDescr_t matA, // non-const descriptor supported
    cusparseConstDnVecDescr_t vecX, // non-const descriptor supported
    const void* beta, cusparseDnVecDescr_t vecY,
    cudaDataType computeType, cusparseSpMValg_t alg, void* externalBuffer);

cusparseStatus_t cusparseSpMV(cusparseHandle_t handle, cusparseOperation_t opA,
    const void* alpha, cusparseConstSpMatDescr_t matA, // non-const descriptor supported
    cusparseConstDnVecDescr_t vecX, // non-const descriptor supported
    const void* beta, cusparseDnVecDescr_t vecY,
    cudaDataType computeType, cusparseSpMValg_t alg, void* externalBuffer);
```

Εικόνα 11: Ορισμός τη συνάρτησης spmv

Sparse Solve (SpSV)

Η ομάδα συναρτήσεων *cusparseSpSV* της βιβλιοθήκης cuSPARSE παρέχει ένα ολοκληρωμένο σετ εργαλείων για την επίλυση αραιών τριγωνικών συστημάτων γραμμικών εξισώσεων. Συγκεκριμένα, η *cusparseSpSV_createDescr()* δημιουργεί έναν περιγραφέα (descriptor) ο οποίος χρησιμοποιείται σε όλες τις επόμενες κλήσεις για την

επίλυση του συστήματος, ενώ η *cusparseSpSV_destroyDescr()* διαγράφει αυτόν τον περιγραφέα μόλις ολοκληρωθεί η επεξεργασία, ελευθερώνοντας τους πόρους που είχε διαθέσει. Η *cusparseSpSV_bufferSize()* υπολογίζει το απαιτούμενο μέγεθος ενός εξωτερικού buffer, το οποίο είναι απαραίτητο για την αποδοτική εκτέλεση της πράξης, λαμβάνοντας υπόψη τις συγκεκριμένες παραμέτρους του συστήματος όπως ο τύπος του πίνακα, το διάνυσμα εισόδου και οι παράμετροι υπολογισμού. Στη συνέχεια, η *cusparseSpSV_analysis()* πραγματοποιεί προκαταρκτική ανάλυση του συστήματος, προετοιμάζοντας τα δεδομένα ώστε να επιτευχθεί βέλτιστη απόδοση κατά την επίλυση. Η *cusparseSpSV_solve()* εκτελεί την πράξη επίλυσης του τριγωνικού συστήματος χρησιμοποιώντας τις πληροφορίες που προέκυψαν από την ανάλυση και τον προεπεξεργασμένο περιγραφέα, ενώ η *cusparseSpSV_updateMatrix()* επιτρέπει την ενημέρωση των τιμών του πίνακα όταν απαιτούνται αλλαγές, διασφαλίζοντας ότι ο περιγραφέας παραμένει συμβατός με το νέο σύνολο δεδομένων. Αυτή η σειρά συναρτήσεων συνεργάζεται ώστε να εξασφαλίζει την αποτελεσματική και υψηλής απόδοσης επίλυση αραιών τριγωνικών συστημάτων σε περιβάλλον GPU.

```
cusparseStatus_t cusparseSpSV_createDescr(cusparseSpSVDescr_t* spsvDescr);
cusparseStatus_t cusparseSpSV_destroyDescr(cusparseSpSVDescr_t spsvDescr);
cusparseStatus_t cusparseSpSV_bufferSize(cusparseHandle_t handle, cusparseOperation_t opA,
    const void* alpha, cusparseConstSpMatDescr_t matA, // non-const descriptor supported
    cusparseConstDnVecDescr_t vecX, // non-const descriptor supported
    cusparseDnVecDescr_t vecY, cudaDataType computeType, cusparseSpSValg_t alg,
    cusparseSpSVDescr_t spsvDescr, size_t* bufferSize);
cusparseStatus_t cusparseSpSV_analysis(cusparseHandle_t handle, cusparseOperation_t opA,
    const void* alpha, cusparseConstSpMatDescr_t matA, // non-const descriptor supported
    cusparseConstDnVecDescr_t vecX, // non-const descriptor supported
    cusparseDnVecDescr_t vecY, cudaDataType computeType, cusparseSpSValg_t alg,
    cusparseSpSVDescr_t spsvDescr, void* externalBuffer);

cusparseStatus_t cusparseSpSV_solve(cusparseHandle_t handle, cusparseOperation_t opA,
    const void* alpha, cusparseConstSpMatDescr_t matA, // non-const descriptor supported
    cusparseConstDnVecDescr_t vecX, // non-const descriptor supported
    cusparseDnVecDescr_t vecY, cudaDataType computeType, cusparseSpSValg_t alg,
    cusparseSpSVDescr_t spsvDescr);

cusparseStatus_t cusparseSpSV_updateMatrix(cusparseHandle_t handle,
    cusparseSpSVDescr_t spsvDescr, void* newValues, cusparseSpSVUpdate_t updatePart);
```

Εικόνα 12: Ορισμός των συναρτήσεων SpSV.

Sparse General Matrix-Matrix Multiplication (SpMM)

Η συνάρτηση *cusparseSpMM* του cuSPARSE εκτελεί τον πολλαπλασιασμό ενός αραιού πίνακα με έναν πυκνό πίνακα, υπολογίζοντας την πράξη $C = \alpha \times \text{op}(A) \times \text{op}(B) + \beta \times C$. Εδώ, οι παράμετροι α και β είναι βαθμωτά, ενώ οι πίνακες A , B και C αντιπροσωπεύουν τον αραιό πίνακα, τον πυκνό πίνακα εισόδου και τον πυκνό πίνακα εξόδου αντίστοιχα. Η

συνάρτηση υποστηρίζει διάφορες μορφές αποθήκευσης του αραιού πίνακα, όπως το CSR (Compressed Sparse Row) και το COO (Coordinate), παρέχοντας ευελιξία στην αναπαράσταση των δεδομένων ανάλογα με το συγκεκριμένο πρόβλημα και την κατανομή των μη μηδενικών στοιχείων.

Πριν από την εκτέλεση της κύριας πράξης, απαιτείται ο καθορισμός του απαιτούμενου μεγέθους ενός προσωρινού buffer που βοηθάει στην ροή του αλγορίθμου, μέσω της συνάρτησης *cusparseSpMM_bufferSize()*. Αυτό το βήμα είναι κρίσιμο για τη σωστή διαχείριση της μνήμης και την επίτευξη βέλτιστης απόδοσης.

Σε ορισμένες περιπτώσεις μπορεί να χρειαστεί προεπεξεργασία των δεδομένων (π.χ. βελτιστοποίηση της δομής του πίνακα ή υπολογισμός βοηθητικών δεδομένων), η οποία υλοποιείται με τη συνάρτηση *cusparseSpMM_preprocess()*.

Η κύρια πράξη του πολλαπλασιασμού εκτελείται με την κλήση της *cusparseSpMM()*, ολοκληρώνοντας έτσι την υπολογιστική διαδικασία σύμφωνα με τις παραπάνω παραμέτρους και την επιλεγμένη μορφή αποθήκευσης του πίνακα.

```
cusparseStatus_t cusparseSpMM_bufferSize(cusparseHandle_t handle,
    cusparseOperation_t opA, cusparseOperation_t opB, const void* alpha,
    cusparseConstSpMatDescr_t matA, // non-const descriptor supported
    cusparseConstDnMatDescr_t matB, // non-const descriptor supported
    const void* beta, cusparseDnMatDescr_t matC, cudaDataType computeType,
    cusparseSpMMAlg_t alg, size_t* bufferSize);

cusparseStatus_t cusparseSpMM_preprocess(cusparseHandle_t handle,
    cusparseOperation_t opA, cusparseOperation_t opB, const void* alpha,
    cusparseConstSpMatDescr_t matA, // non-const descriptor supported
    cusparseConstDnMatDescr_t matB, // non-const descriptor supported
    const void* beta, cusparseDnMatDescr_t matC, cudaDataType computeType,
    cusparseSpMMAlg_t alg, void* externalBuffer);

cusparseStatus_t cusparseSpMM(cusparseHandle_t handle,
    cusparseOperation_t opA, cusparseOperation_t opB, const void* alpha,
    cusparseConstSpMatDescr_t matA, // non-const descriptor supported
    cusparseConstDnMatDescr_t matB, // non-const descriptor supported
    const void* beta, cusparseDnMatDescr_t matC, cudaDataType computeType,
    cusparseSpMMAlg_t alg, void* externalBuffer);
```

Εικόνα 13: Ορισμός της συνάρτησης SpMM.

Sparse Matrix triangular solve(SpSM)

Η συνάρτηση *cusparseSpSM* της βιβλιοθήκης, χρησιμοποιείται για την επίλυση αραιών τριγωνικών συστημάτων γραμμικών εξισώσεων της μορφής $op(A) \times X = \alpha \times B$, όπου A είναι ένας αραιός τριγωνικός πίνακας, B και X είναι πυκνοί πίνακες, και α είναι ένας βαθμωτός πολλαπλασιαστής. Η σειρά κλήσεων περιλαμβάνει τη δημιουργία ενός περιγραφέα με τη συνάρτηση *cusparseSpSM_createDescr()*, τον υπολογισμό του

απαιτούμενου μεγέθους μνήμης με τη `cusparseSpSM_bufferSize()`, την ανάλυση του συστήματος με τη `cusparseSpSM_analysis()`, και τέλος την επίλυση του συστήματος με τη `cusparseSpSM_solve()`. Μετά την ολοκλήρωση, ο περιγραφέας απελευθερώνεται χρησιμοποιώντας τη `cusparseSpSM_destroyDescr()`.

Ο `spsmDescr` λειτουργεί ως περιγραφέας στην συνάρτηση επίλυσης αραιών τριγωνικών συστημάτων (`cusparseSpSM`), αποθηκεύοντας όλα τα απαραίτητα μεταδεδομένα που αφορούν τη δομή και τις ιδιότητες του τριγωνικού πίνακα που εμπλέκεται στο σύστημα. Συγκεκριμένα, αυτός ο περιγραφέας περιέχει πληροφορίες για το αν ο πίνακας είναι άνω ή κάτω τριγωνικός, για το αν είναι μοναδιαίος (δηλαδή αν οι κύριες διαγώνιοι αποτελούν μονάδες) και άλλες παραμέτρους που καθορίζουν τον τρόπο με τον οποίο θα επιλυθεί το σύστημα. Επιπλέον, ο `spsmDescr` χρησιμοποιείται για την αποθήκευση προεπεξεργασμένων δεδομένων και ενδιάμεσων αποτελεσμάτων από την ανάλυση του συστήματος, επιτρέποντας έτσι την αποδοτικότερη και πιο επαναχρησιμοποιήσιμη επίλυση σε επόμενες κλήσεις της συνάρτησης,

```
cusparseStatus_t cusparseSpSM_createDescr(cusparseSpSMDescr_t* spsmDescr);
cusparseStatus_t cusparseSpSM_destroyDescr(cusparseSpSMDescr_t spsmDescr);
cusparseStatus_t cusparseSpSM_bufferSize(cusparseHandle_t handle,
    cusparseOperation_t opA, cusparseOperation_t opB, const void* alpha,
    cusparseConstSpMatDescr_t matA, // non-const descriptor supported
    cusparseConstDnMatDescr_t matB, // non-const descriptor supported
    cusparseDnMatDescr_t matC, cudaDataType computeType, cusparseSpSMAlg_t alg,
    cusparseSpSMDescr_t spsmDescr, size_t* bufferSize);

cusparseStatus_t cusparseSpSM_analysis(cusparseHandle_t handle,
    cusparseOperation_t opA, cusparseOperation_t opB, const void* alpha,
    cusparseConstSpMatDescr_t matA, // non-const descriptor supported
    cusparseConstDnMatDescr_t matB, // non-const descriptor supported
    cusparseDnMatDescr_t matC, cudaDataType computeType, cusparseSpSMAlg_t alg,
    cusparseSpSMDescr_t spsmDescr, void* externalBuffer);

cusparseStatus_t cusparseSpSM_solve(cusparseHandle_t handle,
    cusparseOperation_t opA, cusparseOperation_t opB, const void* alpha,
    cusparseConstSpMatDescr_t matA, // non-const descriptor supported
    cusparseConstDnMatDescr_t matB, // non-const descriptor supported
    cusparseDnMatDescr_t matC, cudaDataType computeType,
    cusparseSpSMAlg_t alg, cusparseSpSMDescr_t spsmDescr);

cusparseStatus_t cusparseSpSM_updateMatrix(cusparseHandle_t handle,
    cusparseSpSMDescr_t spsmDescr, void* newValues, cusparseSpSMUpdate_t updatePart);
```

Εικόνα 14: Ορισμός της συνάρτησης SpSM.

Sparse Matrix triangular solve(SpGEMM)

Οι συναρτήσεις της ομάδας `cusparseSpGEMM` παρέχουν ένα ολοκληρωμένο πλαίσιο για την υλοποίηση του πολλαπλασιασμού αραιών πινάκων με πίνακες, δηλαδή για την πράξη

$C = \alpha \text{op}(A) \times \text{op}(B) + \beta C$, όπου A και B είναι αραιοί πίνακες και C ο πίνακας εξόδου. Η διαδικασία ξεκινά με τη δημιουργία ενός περιγραφέα μέσω της συνάρτησης *cusparseSpGEMM_createDescr()*, ο οποίος αποθηκεύει όλα τα απαραίτητα μεταδεδομένα και παραμέτρους για την εκτέλεση της πράξης. Στη συνέχεια, η *cusparseSpGEMM_workEstimation()* υπολογίζει το απαιτούμενο μέγεθος μνήμης (*buffer*) που χρειάζεται για την εκτέλεση του πολλαπλασιασμού, λαμβάνοντας υπόψη τις ρυθμίσεις των παραμέτρων, τον επιλεγμένο αλγόριθμο και τις μορφές των λειτουργιών που θα εφαρμοστούν στους πίνακες.

Περαιτέρω, με τη συνάρτηση *cusparseSpGEMM_getNumProducts()* μπορεί να ανακτηθεί ο αριθμός των μη μηδενικών στοιχείων που θα παραχθούν από τον πολλαπλασιασμό, γεγονός που βοηθά στον καθορισμό του τελικού μεγέθους του πίνακα εξόδου. Η *cusparseSpGEMM_estimateMemory()* πραγματοποιεί μια πιο λεπτομερή εκτίμηση των απαιτήσεων μνήμης, λαμβάνοντας υπόψη επιπλέον παράγοντες όπως το *chunk_fraction*, και επιστρέφει τα απαραίτητα μεγέθη *buffer* για την επόμενη φάση. Η κύρια πράξη του πολλαπλασιασμού πραγματοποιείται με τη *cusparseSpGEMM_compute()*, η οποία εκτελεί τους υπολογισμούς στον εξωτερικό *buffer*, και ολοκληρώνεται με τη συνάρτηση *cusparseSpGEMM_copy()*, που αντιγράφει τα αποτελέσματα στον τελικό πίνακα C . Αυτή η ολοκληρωμένη διαδικασία επιτρέπει την αποδοτική εκτέλεση του πολλαπλασιασμού αραιών πινάκων, αξιοποιώντας πλήρως την υπολογιστική ισχύ της GPU για εφαρμογές που απαιτούν υψηλή απόδοση και βελτιστοποιημένες λύσεις σε προβλήματα γραμμικής άλγεβρας.

```

cusparsesStatus_t cusparsesSpGEMM_createDescr(cusparsesSpGEMMDescr_t* descr);
cusparsesStatus_t cusparsesSpGEMM_destroyDescr(cusparsesSpGEMMDescr_t descr);
cusparsesStatus_t cusparsesSpGEMM_workEstimation(cusparsesHandle_t handle,
    cusparsesOperation_t opA, cusparsesOperation_t opB, const void* alpha,
    cusparsesConstSpMatDescr_t matA, // non-const descriptor supported
    cusparsesConstSpMatDescr_t matB, // non-const descriptor supported
    const void* beta, cusparsesSpMatDescr_t matC, cudaDataType computeType, cusparsesSpGEMMAlg_t alg,
    cusparsesSpGEMMDescr_t spgemmDescr, size_t* bufferSize1, void* externalBuffer1);

cusparsesStatus_t cusparsesSpGEMM_getNumProducts(cusparsesSpGEMMDescr_t spgemmDescr, int64_t* num_prods);

cusparsesStatus_t cusparsesSpGEMM_estimateMemory(cusparsesHandle_t handle,
    cusparsesOperation_t opA, cusparsesOperation_t opB, const void* alpha,
    cusparsesConstSpMatDescr_t matA, // non-const descriptor supported
    cusparsesConstSpMatDescr_t matB, // non-const descriptor supported
    const void* beta, cusparsesSpMatDescr_t matC, cudaDataType computeType,
    cusparsesSpGEMMAlg_t alg, cusparsesSpGEMMDescr_t spgemmDescr,
    float chunk_fraction, size_t* bufferSize3, void* externalBuffer3, size_t* bufferSize2);

cusparsesStatus_t cusparsesSpGEMM_compute(cusparsesHandle_t handle,
    cusparsesOperation_t opA, cusparsesOperation_t opB, const void* alpha,
    cusparsesConstSpMatDescr_t matA, // non-const descriptor supported
    cusparsesConstSpMatDescr_t matB, // non-const descriptor supported
    const void* beta, cusparsesSpMatDescr_t matC, cudaDataType computeType,
    cusparsesSpGEMMAlg_t alg, cusparsesSpGEMMDescr_t spgemmDescr, size_t* bufferSize2, void* externalBuffer2);

cusparsesStatus_t cusparsesSpGEMM_copy(cusparsesHandle_t handle,
    cusparsesOperation_t opA, cusparsesOperation_t opB, const void* alpha,
    cusparsesConstSpMatDescr_t matA, // non-const descriptor supported
    cusparsesConstSpMatDescr_t matB, // non-const descriptor supported
    const void* beta, cusparsesSpMatDescr_t matC, cudaDataType computeType, cusparsesSpGEMMAlg_t alg,
    cusparsesSpGEMMDescr_t spgemmDescr);

```

Εικόνα 15: Ορισμός της συνάρτησης SpGEMM.

Sparse Matrix Conversion

Παρέχει μετατροπή μεταξύ διαφορετικών μορφών αποθήκευσης αραιών πινάκων (όπως CSR, COO, BSR), επιτρέποντας την ευελιξία και την προσαρμοστικότητα ανάλογα με την εφαρμογή.

cusparsesSparseToDense()

Η λειτουργία μετατρέπει τον αραιό πίνακα *matA* από μορφή CSR (Compressed Sparse Row), CSC (Compressed Sparse Column) ή COO (Coordinate List) σε πυκνή αναπαράσταση *matB*. Αυτό σημαίνει ότι τα μη μηδενικά στοιχεία του αραιού πίνακα μεταφέρονται σε έναν πίνακα πυκνού τύπου, όπου όλα τα στοιχεία αποθηκεύονται, συμπεριλαμβανομένων των μηδενικών.

cusparsesDenseToSparse()

Η συνάρτηση *cusparsesDenseToSparse()* μετατρέπει έναν πυκνό πίνακα σε αραιή μορφή. Αυτή η διαδικασία περιλαμβάνει την αποθήκευση μόνο των μη μηδενικών στοιχείων, μειώνοντας τη χρήση μνήμης για πίνακες με πολλά μηδενικά στοιχεία. Η συνάρτηση αυτή είναι χρήσιμη για βελτιστοποιημένες εφαρμογές που χειρίζονται μεγάλα σύνολα δεδομένων με αραιά χαρακτηριστικά.

2.3 Βιβλιοθήκη CUFFT

Η βιβλιοθήκη cuFFT της Nvidia είναι μια εξειδικευμένη βιβλιοθήκη για τον υπολογισμό μετασχηματισμών Fourier (Fast Fourier Transforms - FFTs) με τη χρήση GPU, αξιοποιώντας την υπολογιστική ισχύ της αρχιτεκτονικής CUDA. Παρέχει εξαιρετικά αποδοτικές υλοποιήσεις FFT, καθιστώντας την ιδανική για εφαρμογές που απαιτούν υψηλή απόδοση, όπως η επεξεργασία σήματος, η ανάλυση δεδομένων, η προσομοίωση φυσικών φαινομένων, και άλλες επιστημονικές εφαρμογές.

Η ανάπτυξη της cuFFT έχει επηρεαστεί από τη διάσημη βιβλιοθήκη FFTW (Fastest Fourier Transform in the West), η οποία θεωρείται μία από τις κορυφαίες λύσεις για FFT σε CPUs. Η cuFFT μοιράζεται πολλές ομοιότητες με την FFTW, καθιστώντας τη μετάβαση από την βιβλιοθήκη Host σε βιβλιοθήκη συσκευής σχετικά εύκολη. Αυτό είναι ιδιαίτερα χρήσιμο για προγραμματιστές που ήδη γνωρίζουν και χρησιμοποιούν την FFTW..

2.3.1 Εισαγωγή

Η cuFFT παρέχει βελτιστοποιήσεις υψηλού επιπέδου, προσαρμοσμένες στον υποκείμενο υλικό εξοπλισμό της GPU, εξασφαλίζοντας μέγιστη απόδοση κατά την εκτέλεση μετασχηματισμών. Επιτρέπει την οργάνωση των μετασχηματισμών για σενάριο πολλαπλών εκτελέσεων με την υποστήριξη CUDA streams, γεγονός που βελτιώνει την απόδοση στις εφαρμογές και επιπλέον, η βιβλιοθήκη προσφέρει εξατομίκευση με βάση την οργάνωση δεδομένων, μέσω strided layouts, διευκολύνοντας την επεξεργασία πολύπλοκων δεδομένων και την ενσωμάτωσή της σε εφαρμογές με ειδικές απαιτήσεις μνήμης και προσπέλασης.

Το API της cuFFT αποτελείται από δύο βασικά μέρη:

1. *cuFFT*: περιλαμβάνει λειτουργίες για την εκτέλεση μετασχηματισμών Fourier στην GPU, επιτρέποντας γρήγορη και αποδοτική επεξεργασία σημάτων και δεδομένων. Υποστηρίζει διάφορους τύπους μετασχηματισμών, όπως Complex-to-Complex (C2C) για μιγαδικά δεδομένα, Real-to-Complex (R2C) για μετατροπή πραγματικών δεδομένων σε μιγαδικά κατά την εκτέλεση του προς τα εμπρός μετασχηματισμού FFT και Complex-to-Real (C2R) για την αντίστροφη διαδικασία. Επιπλέον, παρέχει δυνατότητα εκτέλεσης μετασχηματισμών σε μία (1D), δύο (2D) ή τρεις (3D) διαστάσεις, καθώς και επεξεργασία πολλαπλών μετασχηματισμών ταυτόχρονα μέσω batch processing.

2. *cuFFT*: Ένα εργαλείο μετατροπής κώδικα, το οποίο διευκολύνει την προσαρμογή κώδικα που έχει γραφτεί για την FFTW, ώστε να μπορεί να εκτελεστεί στην GPU μέσω της cuFFT. Αυτό επιτρέπει τη γρήγορη μετάβαση σε CUDA χωρίς να απαιτείται εκτεταμένη αναδιάρθρωση του κώδικα.

Λειτουργικότητα της βιβλιοθήκης

Μια βασική αρχή λειτουργίας της cuFFT είναι η χρήση ενός μηχανισμού που ονομάζεται Plan ή στα ελληνικά *σχέδιο*. Ένα σχέδιο είναι ένας προκαθορισμένος διαχειριστής που αποθηκεύει όλες τις παραμέτρους και ρυθμίσεις για την εκτέλεση ενός μετασχηματισμού. Οι παράμετροι αυτοί περιλαμβάνουν το μέγεθος και τη διάσταση του μετασχηματισμού, τον τύπο δεδομένων (π.χ., πραγματικά, μιγαδικά), και τις ρυθμίσεις για την οργάνωση και την προσπέλαση των δεδομένων (π.χ., strided layout). Η δημιουργία ενός σχεδίου εξασφαλίζει ότι ο μετασχηματισμός θα είναι βελτιστοποιημένος για την εκάστοτε GPU και τα χαρακτηριστικά των δεδομένων. Κατά τη διαδικασία δημιουργίας ενός σχεδίου, η βιβλιοθήκη αφιερώνει ένα μεγάλο ποσό μνήμης, ανάλογα πάντα και με το μέγεθος του μετασχηματισμού, και καλή τεχνική θα ήταν να έχει προτεραιότητα στην εκτέλεση καθώς επίσης εκτελεί προκαταρκτικούς υπολογισμούς και προσαρμογές, ώστε να διασφαλιστεί η βέλτιστη χρήση των διαθέσιμων πόρων της GPU.

Με την cuFFT, οι χρήστες μπορούν να εκτελούν:

- 1D, 2D και 3D FFTs για δεδομένα διαφορετικών διαστάσεων.
- Μετασχηματισμούς μεταξύ complex-to-complex (C2C), real-to-complex (R2C) και complex-to-real (C2R).
- Παράλληλη εκτέλεση πολλαπλών FFTs σε παρτίδες (batch processing), κάτι που είναι ιδιαίτερα χρήσιμο για εφαρμογές με μεγάλο όγκο δεδομένων.

Η βιβλιοθήκη cuFFT υποστηρίζει επίσης streamed execution, μια δυνατότητα που επιτρέπει την παράλληλη επεξεργασία δεδομένων μέσω CUDA streams. Τα streams είναι ανεξάρτητες σειρές εντολών που εκτελούνται ταυτόχρονα στην GPU, επιτρέποντας την κατανομή μεγάλων όγκων δεδομένων σε μικρότερα τμήματα για ταχύτερη επεξεργασία. Αυτό προσφέρει αυξημένη απόδοση, μειωμένο χρόνο επεξεργασίας και δυνατότητα επικάλυψης υπολογισμών με μεταφορές δεδομένων. Κάθε σχέδιο μπορεί να συσχετιστεί με μια συγκεκριμένη ροή χρησιμοποιώντας τη συνάρτηση `cufftSetStream`, μέσω της οποίας όλες οι εσωτερικές διαδικασίες του σχεδίου εκτελούνται εντός της καθορισμένης ροής. Από την έκδοση CUDA 11.2, η χρήση ροών υποστηρίζεται και σε περιβάλλον πολλαπλών GPU.

Επιπλέον, η υποδομή της βιβλιοθήκης υποστηρίζει τη χρήση πολλαπλών GPU. Μπορεί να αξιοποιήσει έως και 16 GPU ταυτόχρονα (ανάλογα με την εγκατάσταση και την αρχιτεκτονική του συστήματος), επιτρέποντας την εκτέλεση εξαιρετικά απαιτητικών εφαρμογών σε περιβάλλοντα υψηλής απόδοσης.

Για την υποστήριξη αυτής της λειτουργικότητας, παρέχεται το cuFFTX API. Το cuFFTX API υποστηρίζει τη λειτουργικότητα πολλαπλών GPU μέσω συναρτήσεων που διαχειρίζονται την κατανομή των δεδομένων στη μνήμη κάθε GPU, εξασφαλίζοντας ότι τα δεδομένα κατανέμονται σωστά μεταξύ των συσκευών. Ο συντονισμός των υπολογισμών μεταξύ των GPU υλοποιείται με μηχανισμούς που αποτρέπουν την αλληλοεπικάλυψη ή τις καθυστερήσεις. Η διασύνδεση με άλλες βιβλιοθήκες CUDA πραγματοποιείται μέσω κοινών διεπαφών που επιτρέπουν τη συνδυαστική χρήση τους για την υποστήριξη πιο σύνθετων υπολογιστικών εφαρμογών.

2.3.2 Πώς χρησιμοποιούμε τη βιβλιοθήκη

Το πρώτο βήμα για τη χρήση της cuFFT είναι η αρχικοποίηση της βιβλιοθήκης καλώντας τη συνάρτηση `cufftCreate`, η οποία δημιουργεί ένα "σχέδιο" (plan) για τον επιθυμητό μετασχηματισμό. Αυτό το σχέδιο περιλαμβάνει τη δομή των δεδομένων, τη διάσταση του μετασχηματισμού και επίσης ρυθμίζει τη διαχείριση επιπλέον μνήμης που απαιτείται για τον μετασχηματισμό. Το σχέδιο είναι μοναδικό για την εκάστοτε διαδικασία, και οι πόροι του μπορούν να χρησιμοποιηθούν μόνο από το thread που εκτέλεσε τη δημιουργία του σχεδίου.

Δημιουργία Plans

Για τη ρύθμιση ενός μετασχηματισμού Fourier, η cuFFT παρέχει διάφορες συναρτήσεις που δημιουργούν plans για συγκεκριμένους τύπους και διατάξεις δεδομένων:

`cufftPlan1D()` / `cufftPlan2D()` / `cufftPlan3D()`:

Αυτές οι συναρτήσεις χρησιμοποιούνται για τη δημιουργία ενός απλού plan για μετασχηματισμούς 1D, 2D ή 3D αντίστοιχα. Είναι κατάλληλες για περιπτώσεις όπου τα δεδομένα έχουν συγκεκριμένη διάσταση χωρίς ιδιαίτερες απαιτήσεις στη διάταξη.

`cufftPlanMany()`:

Χρησιμοποιείται για τη δημιουργία plans που υποστηρίζουν batch επεξεργασία (δηλαδή πολλούς FFTs ταυτόχρονα) και δεδομένα με strided layouts. Είναι ιδανική για περιπτώσεις όπου υπάρχει ανάγκη για μαζική επεξεργασία παρόμοιων datasets.

cufftXtMakePlanMany():

Μια πιο εξελιγμένη συνάρτηση, η οποία υποστηρίζει batch επεξεργασία και strided δεδομένα για όλους τους υποστηριζόμενους τύπους ακρίβειας (single, double, half precision). Είναι ιδιαίτερα χρήσιμη σε εφαρμογές με πολυδιάστατα δεδομένα και διαφορετικές ακριβείς αναπαραστάσεις.

Εκτέλεση του Μετασχηματισμού

Αφού δημιουργηθεί το plan, η εκτέλεση του μετασχηματισμού πραγματοποιείται μέσω των συναρτήσεων cufftExec*, όπου το * εξαρτάται από τον τύπο των δεδομένων:

- cufftExecC2C: Complex-to-complex μετασχηματισμοί.
- cufftExecR2C: Real-to-complex μετασχηματισμοί.
- cufftExecC2R: Complex-to-real μετασχηματισμοί.

Οι συναρτήσεις αυτές εκτελούν τον μετασχηματισμό σύμφωνα με τις ρυθμίσεις του plan που δημιουργήθηκε προηγουμένως, εφαρμόζοντας την απαιτούμενη διαδικασία στα δεδομένα εισόδου.

Όλες οι συναρτήσεις της cuFFT επιστρέφουν τύπο αναφοράς σφάλματος cufftResult, ο οποίος περιλαμβάνει τιμές που βοηθούν τον προγραμματιστή να διαγνώσει πιθανά λάθη κατά την εκτέλεση. Αυτή η δυνατότητα παρέχει σημαντική υποστήριξη στην ανίχνευση και αποτροπή προβλημάτων, διασφαλίζοντας τη σωστή λειτουργία της βιβλιοθήκης κατά την εκτέλεση των μετασχηματισμών.

2.3.3 Διάταξη δεδομένων

Για την εκτέλεση μετασχηματισμών Fourier με τη βιβλιοθήκη cuFFT, απαιτείται η τήρηση συγκεκριμένων κανόνων σχετικά με τη διάταξη των δεδομένων στη μνήμη, ώστε να διασφαλιστεί η ορθή λειτουργία και η αποδοτική εκτέλεση στον επεξεργαστή GPU.

Η διαχείριση των δεδομένων διαφοροποιείται ανάλογα με τον τύπο του μετασχηματισμού. Για παράδειγμα, σε έναν μετασχηματισμό από μιγαδικά σε πραγματικά δεδομένα ή το αντίστροφο, λόγω της συμμετρίας των μιγαδικών συνιστωσών, δημιουργούνται πλεονάζοντα δεδομένα κατά τη διαδικασία. Αυτό απαιτεί τη χρήση buffers με κατάλληλο μέγεθος, όπως $N/2 + 1$ για την είσοδο και N για την έξοδο ή το αντίστροφο, ώστε να εξυπηρετηθεί η αποθήκευση των δεδομένων. Για παράδειγμα,

εάν ο αριθμός των σημείων του μετασχηματισμού είναι 1024, τότε το buffer εισόδου μπορεί να έχει μέγεθος $512 + 1$, ενώ το buffer εξόδου θα είναι 1024, όταν πραγματοποιείται μετασχηματισμός μιγαδικού σε πραγματικό.

Επιπλέον, η cuFFT προσφέρει βελτιστοποιημένη απόδοση όταν το μέγεθος των δεδομένων είναι γινόμενο πρώτων αριθμών της μορφής $2^a \times 3^b \times 5^c \times 7^d$, με τους συντελεστές a, b, c, d να είναι ακέραιοι μη αρνητικοί αριθμοί. Για παράδειγμα, διαστάσεις όπως 512, 768 ή 1920 επιτρέπουν την εκτέλεση του μετασχηματισμού με τη μέγιστη αποδοτικότητα, ενώ για άλλες διαστάσεις η απόδοση μπορεί να μειωθεί. Η τήρηση αυτών των κανόνων και προδιαγραφών βοηθά στην πλήρη αξιοποίηση της υπολογιστικής ισχύος της GPU, επιτυγχάνοντας ταχύτερους και αποδοτικότερους υπολογισμούς.

Τα σχέδια της βιβλιοθήκης cuFFT συχνά απαιτούν επιπλέον μνήμη για την αποθήκευση ενδιάμεσων αποτελεσμάτων. Από προεπιλογή, κάθε σχέδιο εκχωρεί αυτόνομα τη δική του περιοχή εργασίας στη μνήμη GPU μέσω της συνάρτησης `cufftSetAutoAllocation()`. Ωστόσο, είναι δυνατή η κοινή χρήση μιας περιοχής εργασίας μεταξύ πολλών σχεδίων, μειώνοντας έτσι την κατανάλωση μνήμης, κάτι που ρυθμίζεται με τη συνάρτηση `cufftXtSetWorkAreaPolicy()`. Για την εκτίμηση των απαιτήσεων μνήμης, οι συναρτήσεις `cufftEstimate1d()`, `cufftEstimate2d()`, `cufftEstimate3d()`, και `cufftEstimateMany()` παρέχουν υπολογισμούς που βοηθούν τον προγραμματιστή να εκχωρήσει τη σωστή ποσότητα μνήμης. Επιπλέον, μέσω του CUDA Unified Virtual Addressing, η βιβλιοθήκη μπορεί να χρησιμοποιεί διαφορετικούς τύπους μνήμης, όπως καρφιτσωμένη μνήμη host, διαχειριζόμενη μνήμη, ή μνήμη GPU διαφορετική από εκείνη που εκτελεί τους υπολογισμούς, παρέχοντας μεγαλύτερη ευελιξία αλλά με πιθανή επίπτωση στην απόδοση. Η ακριβής μνήμη που απαιτείται μπορεί να καθοριστεί με τη συνάρτηση `cufftGetSize()`, διασφαλίζοντας τη σωστή κατανομή πριν από την εκτέλεση.

2.3.4 Αρχικοποίηση και Διαχείριση Μνήμης:

Για τη χρήση της cuFFT, ο προγραμματιστής πρέπει να αναλάβει τη διαχείριση της μνήμης στη GPU, ξεκινώντας από τη δέσμευση χώρου για τα δεδομένα. Η δέσμευση μνήμης γίνεται μέσω της συνάρτησης `cudaMalloc()`, όπως συμβαίνει και σε άλλες βιβλιοθήκες CUDA. Αφού δεσμευτεί η μνήμη στη GPU, το επόμενο βήμα είναι η μεταφορά των δεδομένων από τον host στη GPU. Αυτό επιτυγχάνεται χρησιμοποιώντας τη συνάρτηση `cudaMemcpy()`, η οποία διασφαλίζει ότι τα δεδομένα είναι σωστά

τοποθετημένα στη μνήμη της GPU για χρήση από τη βιβλιοθήκη. Μετά την ολοκλήρωση της δέσμευσης μνήμης και τη μεταφορά των δεδομένων στον χώρο μνήμης της GPU, μπορούν να πραγματοποιηθούν οι υπολογισμοί και οι μετασχηματισμοί της βιβλιοθήκης cuFFT.

Αποδέσμευση Πόρων

Η αποδέσμευση των πόρων που δεσμεύονται από ένα σχέδιο στη βιβλιοθήκη cuFFT είναι απαραίτητη για την αποφυγή διαρροών μνήμης. Αυτό επιτυγχάνεται μέσω της συνάρτησης `cufftDestroy`, η οποία διαγράφει το σχέδιο και απελευθερώνει τη μνήμη που είχε δεσμευτεί για τη λειτουργία του.

2.4 Βιβλιοθήκη CURAND

Η CURAND είναι μια βιβλιοθήκη που παρέχει την δυνατότητα παραγωγής ψευδοτυχαίων (pseudorandom) και χαοτικά τυχαίων (quasirandom) αριθμών με την χρήση αποδοτικών και γρήγορων αλγορίθμων που ικανοποιούν τα περισσότερα στατιστικά μοντέλα και προδιαγραφές.

Οι ψευδοτυχαίοι αριθμοί παράγονται με ντετερμινιστικούς αλγορίθμους που χρησιμοποιούν αρχικές τιμές (seeds). Μεταξύ αυτών, ο αλγόριθμος XORWOW, που ανήκει στην οικογένεια XOR-shift, και είναι ο πιο συνηθισμένος για γενική χρήση, ο MRG32K3A ανήκει στην οικογένεια Combined Multiple Recursive και προσφέρει μεγαλύτερη ακρίβεια και μακρύτερες ακολουθίες, οι MT19937 και MTGP32 βασίζονται στην οικογένεια Mersenne Twister, με τον MT19937 να υποστηρίζεται μόνο στο host API και ο PHILOX4_32_10 που ανήκει στην οικογένεια Philox, που χρησιμοποιεί μια counter-based προσέγγιση. Για την παραγωγή χαοτικά τυχαίων αριθμών, η βιβλιοθήκη παρέχει τέσσερις παραλλαγές του αλγόριθμου Sobol', οι οποίες μπορούν να παράγουν ακολουθίες σε έως και 20.000 διαστάσεις. Οι SOBOL32 και SOBOL64 παράγουν 32-bit και 64-bit αριθμούς αντίστοιχα, ενώ οι *SCRAMBLED_SOBOL32* και *SCRAMBLED_SOBOL64* αποτελούν παραλλαγές του αλγορίθμου Sobol'.

2.4.1 Εισαγωγή

Το API της βιβλιοθήκης cuRAND χωρίζεται σε δύο κύρια μέρη, τα οποία διακρίνονται σε συναρτήσεις που καλούνται από το περιβάλλον του host και σε Kernels που εκτελούν την παραγωγή και επεξεργασία αριθμών. Ένα από αυτά είναι μέρος του Host, το οποίο περιλαμβάνεται στο δικό του αρχείο κεφαλίδας "*curand.h*" και παρέχει συναρτήσεις που

καλούνται από κώδικα εκτελούμενο στον host. Η παραγωγή των τυχαίων αριθμών μπορεί να πραγματοποιηθεί είτε απευθείας στον host είτε στο device, ανάλογα με τις ανάγκες της εφαρμογής, προσφέροντας μεγάλη ευελιξία στον προγραμματιστή. Στην περίπτωση της παραγωγής αριθμών στον host, χρησιμοποιούνται ειδικές συναρτήσεις για την αρχικοποίηση της γεννήτριας και την παραγωγή των αριθμών. Η διαδικασία αυτή είναι απλή, καθώς δεν απαιτείται κάποια ιδιαίτερη διαχείριση μνήμης στη GPU. Αντίθετα, όταν η παραγωγή πραγματοποιείται στο device, απαιτείται η δέσμευση μνήμης στον χώρο της GPU, όπου αποθηκεύονται τα παραγόμενα δεδομένα. Αυτά μπορούν είτε να μεταφερθούν πίσω στον χώρο μνήμης του host είτε να παραμείνουν στη GPU για περαιτέρω επεξεργασία από kernels, επιτρέποντας τη βέλτιστη αξιοποίηση των δυνατοτήτων του device για περαιτέρω υπολογισμούς υψηλής απόδοσης.

Το δεύτερο μέρος της βιβλιοθήκης, που περιλαμβάνεται στο αρχείο “*curand_kernel.h*”, παρέχει συναρτήσεις για την παραγωγή τυχαίων αριθμών απευθείας μέσα σε kernels. Η εκτέλεση αυτών των συναρτήσεων από την GPU επιτρέπει την άμεση επεξεργασία δεδομένων, εξαλείφοντας την ανάγκη για πρόσθετες μεταφορές δεδομένων μεταξύ host και device, ενώ αξιοποιεί στο μέγιστο την παραλληλία της GPU. Ωστόσο, η διαχείριση της κατάστασης της γεννήτριας ανατίθεται στον προγραμματιστή, γεγονός που απαιτεί προσεκτικό χειρισμό για την αποφυγή καθυστερήσεων.

Παρακάτω θα αναλύσουμε τον τρόπο ενσωμάτωσης και χρήσης της βιβλιοθήκης cuRAND μέσα σε ένα πρόγραμμα CUDA C++. Η διαδικασία περιλαμβάνει τη δημιουργία και διαχείριση γεννητριών τυχαίων αριθμών καθώς επίσης και την επιλογή κατάλληλων αλγορίθμων.

2.4.2 Περιγραφή των συναρτήσεων του Host API

Η λειτουργία της βιβλιοθήκης cuRAND ξεκινά με τη δημιουργία μιας γεννήτριας τυχαίων αριθμών, η οποία γίνεται μέσω της συνάρτησης *curandCreateGenerator()*. Η συνάρτηση αυτή δέχεται ως παράμετρο τον τύπο του αλγορίθμου παραγωγής που θα χρησιμοποιηθεί, για ψευδοτυχαίους ή χαστικούς τυχαίους αριθμούς. Αφού δημιουργηθεί η γεννήτρια, οι παράμετροι όπως αρχική τιμή, σειρά και μετατόπιση παίρνουν κάποιες προεπιλεγμένες τιμές αλλά, μπορούν να κληθούν πρόσθετες συναρτήσεις για τον καθορισμό των παραμέτρων της, ώστε να προσαρμοστεί στις ανάγκες της εφαρμογής. Για τη δημιουργία μιας γεννήτριας που λειτουργεί αποκλειστικά στον host, μπορούμε να

χρησιμοποιήσουμε τη συνάρτηση *curandCreateGeneratorHost()*, η οποία λειτουργεί με τον ίδιο τρόπο όπως αναλύσαμε παραπάνω.

Έχοντας φτιάξει μια γεννήτρια μπορούμε να ορίσουμε τις παραμέτρους της με συναρτήσεις που προσφέρει η βιβλιοθήκη. Η συνάρτηση *curandSetGeneratorOffset()*, η οποία ορίζει την μετατόπιση της γεννήτριας, δηλαδή το σημείο έναρξης στην ακολουθία τυχαίων αριθμών που παράγεται είναι χρήσιμη όταν θέλουμε διαφορετικά τμήματα της ίδιας ακολουθίας, εξασφαλίζοντας μη επικαλυπτόμενες ακολουθίες. Η *curandSetGeneratorOrdering()* επιτρέπει τον καθορισμό της σειράς με την οποία παράγονται οι τυχαίοι αριθμοί, κάτι που μπορεί να επηρεάσει την απόδοση ή την ακρίβεια της εφαρμογής. Η *curandSetPseudoRandomGeneratorSeed()* χρησιμοποιείται για να οριστεί η αρχική τιμή της γεννήτριας, εξασφαλίζοντας ότι οι παραγόμενες ακολουθίες είναι αναπαράξιμες.

Μόλις έχουμε την αρχικοποιημένη γεννήτρια, μπορούμε να προχωρήσουμε στην παραγωγή τυχαίων αριθμών με τις συναρτήσεις της βιβλιοθήκης cuRAND. Οι βασικές συναρτήσεις παραγωγής περιλαμβάνουν τις *curandGenerate()* και *curandGenerateLongLong()*, οι οποίες δημιουργούν τυχαίους αριθμούς σε μορφή 32-bit και 64-bit αντίστοιχα. Ωστόσο, η βιβλιοθήκη παρέχει και εξειδικευμένες συναρτήσεις για την παραγωγή τυχαίων αριθμών που ακολουθούν συγκεκριμένες στατιστικές κατανομές γεγονός που προσφέρει σημαντική ευελιξία.

Για παράδειγμα, η *curandGenerateUniform()* δημιουργεί αριθμούς που ακολουθούν την ομοιόμορφη κατανομή στο διάστημα $[0,1][0,1][0,1]$, ενώ η *curandGenerateNormal()* παράγει αριθμούς σύμφωνα με τη φυσιολογική (κανονική) κατανομή με καθορισμένα μέσα και διασπορές. Αντίστοιχα, η *curandGenerateLogNormal()* χρησιμοποιείται για αριθμούς που ακολουθούν τη λογαριθμοκανονική κατανομή, και η *curandGeneratePoisson()* δημιουργεί αριθμούς που ακολουθούν την κατανομή Poisson, η οποία είναι χρήσιμη για μελέτες διακριτών φαινομένων. Οι συναρτήσεις αυτές απλοποιούν την παραγωγή εξειδικευμένων στατιστικών δειγμάτων, καθιστώντας τη βιβλιοθήκη cuRAND ιδανική για εφαρμογές όπως μοντελοποίηση, προσομοιώσεις Monte Carlo και στατιστικές αναλύσεις.

Όταν ολοκληρωθεί η χρήση της γεννήτριας, είναι απαραίτητο να απελευθερωθούν οι πόροι που δεσμεύτηκαν για τη λειτουργία της. Αυτό γίνεται με τη συνάρτηση «*curandDestroyGenerator()*», η οποία διαγράφει τη γεννήτρια και διασφαλίζει τη σωστή διαχείριση μνήμης και πόρων.

Κάθε συνάρτηση του Host API επιστρέφει μια τιμή τύπου **curandStatus_t**, η οποία χρησιμοποιείται για την αναφορά τυχόν σφαλμάτων και τη διευκόλυνση της διαχείρισης λαθών. Σε περίπτωση επιτυχούς εκτέλεσης, επιστρέφεται η τιμή *CURAND_STATUS_SUCCESS*, επιτρέποντας την άμεση επιβεβαίωση ότι η λειτουργία ολοκληρώθηκε σωστά.

```
//Create new random number generator.
curandStatus_t curandCreateGenerator ( curandGenerator_t* generator, curandRngType_t rng_type );
//Create new host CPU random number generator.
curandStatus_t curandCreateGeneratorHost ( curandGenerator_t* generator, curandRngType_t rng_type);
//Construct the histogram array for a Poisson distribution.
curandStatus_t curandCreatePoissonDistribution ( double lambda,
| curandDiscreteDistribution_t* discrete_distribution );
//Destroy the histogram array for a discrete distribution (e.g. Poisson).
curandStatus_t curandDestroyDistribution ( curandDiscreteDistribution_t discrete_distribution);
//Destroy an existing generator.
curandStatus_t curandDestroyGenerator ( curandGenerator_t generator );
//Generate 32-bit pseudo or quasirandom numbers.
curandStatus_t curandGenerate ( curandGenerator_t generator, unsigned int* outputPtr, size_t num);
//Generate 64-bit quasirandom numbers.
curandStatus_t curandGenerateLongLong ( curandGenerator_t generator,
| unsigned long long* outputPtr, size_t num);
//Generate normally distributed doubles.
curandStatus_t curandGenerateNormal ( curandGenerator_t generator, float* outputPtr, size_t n,
| float mean, float stddev );
//Set the absolute offset of the pseudo or quasirandom number generator.
curandStatus_t curandSetGeneratorOffset ( curandGenerator_t generator, unsigned long long offset);
//Set the ordering of results of the pseudo or quasirandom number generator.
curandStatus_t curandSetGeneratorOrdering ( curandGenerator_t generator, curandOrdering_t order);
//Set the seed value of the pseudo-random number generator.
curandStatus_t curandSetPseudoRandomGeneratorSeed ( curandGenerator_t generator,
| unsigned long long seed );
//Generate Poisson-distributed unsigned ints.
curandStatus_t curandGeneratePoisson ( curandGenerator_t generator, unsigned int* outputPtr,
| size_t n, double lambda );
//Setup starting states.
curandStatus_t curandGenerateSeeds ( curandGenerator_t generator );
//Generate uniformly distributed floats.
curandStatus_t curandGenerateUniform ( curandGenerator_t generator, float* outputPtr, size_t num);
```

Εικόνα 16: Συναρτήσεις της CURAND.

2.4.3 Συναρτήσεις Device API

Στο Device API της βιβλιοθήκης cuRAND παρέχονται συναρτήσεις ειδικά σχεδιασμένες για εκτέλεση μέσα σε kernels, επιτρέποντας την άμεση παραγωγή τυχαίων αριθμών στη GPU. Υποστηρίζεται παραγωγή τόσο ψευδοτυχαίων (pseudorandom) όσο και χαοτικών (quasirandom) αριθμών, καλύπτοντας τις ανάγκες για ευρύ φάσμα εφαρμογών όπως μοντελοποίηση και προσομοιώσεις.

Η αρχικοποίηση της γεννήτριας στη GPU γίνεται μέσω της συνάρτησης *curand_init()*, η οποία ρυθμίζει την κατάσταση της γεννήτριας ανά νήμα. Η κλήση αυτής της συνάρτησης διαφοροποιείται ανάλογα με τον αλγόριθμο παραγωγής που

χρησιμοποιείται, καθώς και με τον τύπο της δομής που αντιπροσωπεύει τη γεννήτρια, ώστε να υποστηρίζονται όλοι οι διαθέσιμοι τύποι γεννητριών. Οι παράμετροι της `curand_init` για `pseudorandom RNG` περιλαμβάνουν την τιμή `seed`, τον αριθμό ακολουθίας (`sequence number`) και την μετατόπιση (`offset`), δίνοντας στον προγραμματιστή τη δυνατότητα να ελέγξει την αναπαραγωγικότητα και την ανεξαρτησία των τυχαίων αριθμών ανά νήμα ενώ για `quasirandom RNG` απαιτείται ο καθορισμός των `direction_vectors`, που καθορίζουν την κατεύθυνση της ακολουθίας, η αρχική θέση (`offset`) μέσα στην ακολουθία και ακόμα αν χρησιμοποιούμε την εκδοχή `scrambled`, επιπλέον απαιτείται ο ορισμός της παραμέτρου `scramble_c`, η οποία προσθέτει τυχαιότητα μέσω τεχνικών ανακατέματος.

Για την παραγωγή αριθμών, το Device API παρέχει την συνάρτηση `curand`, η οποία παράγει μια τυχαία ακολουθία 32 bit από μια γεννήτρια ενώ παράλληλα προσφέρονται και άλλες συναρτήσεις για παραγωγή αριθμών από στατιστικές κατανομές όπως η `curand_uniform`, που επιστρέφει αριθμούς στο διάστημα $[0,1]$ σύμφωνα με την ομοιόμορφη κατανομή, η `curand_normal`, που παράγει αριθμούς βάσει κανονικής κατανομής (με μέσο όρο 0 και τυπική απόκλιση 1), η `curand_log_normal`, που δημιουργεί αριθμούς από τη λογαριθμοκανονική κατανομή και η `curand_poisson()` παράγει αριθμούς που ακολουθούν την κατανομή Poisson.

Για `quasirandom` αριθμούς, παρέχονται εξειδικευμένες γεννήτριες που βασίζονται στον αλγόριθμο Sobol', οι οποίες είναι ιδανικές για εφαρμογές που απαιτούν χαμηλή απόκλιση. Η χρήση αυτών των γεννητριών περιλαμβάνει την επιλογή του αριθμού των διαστάσεων και την κλήση αντίστοιχων συναρτήσεων για την παραγωγή ακολουθιών.

Ας εξετάσουμε συγκεκριμένα τον ορισμό των συναρτήσεων που αντιστοιχούν σε κάθε τύπο αλγορίθμου γεννήτριας.

Γεννήτριες XORWOW and MRG32k3a

Η cuRAND παρέχει διαφορετικούς τύπους γεννητριών τυχαίων αριθμών που μπορούν να χρησιμοποιηθούν ανάλογα με τις ανάγκες της εφαρμογής.

XORWOW: Γεννήτρια που βασίζεται σε XOR-based τεχνικές, γνωστή για την ταχύτητα και την καλή ισορροπία ανάμεσα σε απόδοση και ποιότητα.

MRG32k3a: Μια πιο εξελιγμένη γεννήτρια που χρησιμοποιεί μεθόδους γραμμικής σύγκλισης, κατάλληλη για εφαρμογές που απαιτούν υψηλή ακρίβεια και μεγάλη περίοδο.

Γεννήτρια MTGP32

Η MTGP32 (Mersenne Twister for Graphic Processors) είναι μια παραλλαγή της Mersenne Twister που είναι προσαρμοσμένη για εκτέλεση σε GPU. Χαρακτηρίζεται ιδανική για περιπτώσεις που απαιτούν υψηλή ακρίβεια και μεγάλες περιόδους και είναι κατάλληλη για εφαρμογές που εκτελούν υπολογιστικά εντατική τυχαιότητα.

Γεννήτρια Philox_4x32_10

Η Philox_4x32_10 είναι μια counter-based γεννήτρια που παρέχει μεγάλη ταχύτητα και καλό επίπεδο τυχαιότητας. Είναι εξαιρετικά αποδοτική για παράλληλες εφαρμογές και προορίζεται κυρίως για εφαρμογές που απαιτούν ανεξάρτητες ακολουθίες.

```
void curand_init ( curandDirectionVectors64_t direction_vectors, unsigned long long scramble_c,
| unsigned long long offset, curandStateScrambledSobol64_t* state );//Initialize Sobol64 state.
void curand_init ( curandDirectionVectors64_t direction_vectors,
| unsigned long long offset,curandStateSobol64_t* state ); //Initialize Sobol64 state.
void curand_init ( curandDirectionVectors32_t direction_vectors, unsigned int scramble_c,
| unsigned int offset, curandStateScrambledSobol32_t* state );//Initialize Scrambled Sobol32 state.
void curand_init ( curandDirectionVectors32_t direction_vectors,
| unsigned int offset, curandStateSobol32_t* state );//Initialize Sobol32 state.
void curand_init ( unsigned long long seed, unsigned long long subsequence,
| unsigned long long offset, curandStateMRG32k3a_t* state );//Initialize MRG32k3a state.
void curand_init ( unsigned long long seed, unsigned long long subsequence,
| unsigned long long offset, curandStatePhilox4_32_10_t* state );//Initialize Philox4_32_10 state.
void curand_init ( unsigned long long seed, unsigned long long subsequence,
| unsigned long long offset, curandStateXORWOW_t* state );//Initialize XORWOW state.
```

Εικόνα 17: Συναρτήσεις του Device API για αρχικοποίηση των γεννητριών.

```

//Return 32-bits of pseudorandomness from a mtgp32 generator.
unsigned int curand(curandStateMtg32_t* state);
//Return 64-bits of quasirandomness from a scrambled Sobol64 generator.
unsigned long long curand(curandStateScrambledSobol64_t* state);
//Return 64-bits of quasirandomness from a Sobol64 generator.
unsigned long long curand(curandStateSobol64_t* state);
//Return 32-bits of quasirandomness from a scrambled Sobol32 generator.
unsigned int curand(curandStateScrambledSobol32_t* state);
//Return 32-bits of quasirandomness from a Sobol32 generator.
unsigned int curand(curandStateSobol32_t* state);
//Return 32-bits of pseudorandomness from an MRG32k3a generator.
unsigned int curand(curandStateMRG32k3a_t* state);
//Return 32-bits of pseudorandomness from an Philox4_32_10 generator.
unsigned int curand(curandStatePhilox4_32_10_t* state);
//Return 32-bits of pseudorandomness from an XORWOW generator.
unsigned int curand(curandStateXORWOW_t* state);
//Return tuple of 4 32-bit pseudorandoms from a Philox4_32_10 generator.
uint4 curand4 (curandStatePhilox4_32_10_t* state);

```

Εικόνα 18: Συναρτήσεις για την παραγωγή τυχαίων αριθμών.

Κατανομές

Η βιβλιοθήκη υποστηρίζει την παραγωγή τυχαίων αριθμών από κατανομές που καλύπτουν ποικίλες ανάγκες σε τυχαία δεδομένα. Περιλαμβάνει την ομοιόμορφη κατανομή (uniform), η οποία δημιουργεί αριθμούς ομοιόμορφα κατανεμημένους στο διάστημα $[0,1)$, την κανονική κατανομή, που παράγει αριθμούς με μέση τιμή 0 και τυπική απόκλιση 1, καθώς και την log-normal κατανομή, ιδανική για δεδομένα που ακολουθούν εκθετική αύξηση, και ακόμα, παρέχεται η Poisson κατανομή, κατάλληλη για την προσομοίωση γεγονότων που εμφανίζονται με σταθερό μέσο ρυθμό.

Οι παρακάτω συναρτήσεις επιτρέπουν τη δημιουργία τυχαίων αριθμών από διάφορες στατιστικές κατανομές απευθείας στη GPU, χρησιμοποιώντας διαφορετικούς τύπους γεννητριών. Οι συναρτήσεις αυτές είναι βελτιστοποιημένες για παράλληλη εκτέλεση και χρησιμοποιούνται σε προσομοιώσεις, μοντελοποίηση δεδομένων και μηχανική μάθηση.

```

// Επιστρέφει έναν ομοιόμορφα κατανομημένο float από έναν XORWOW generator.
__device__ float curand_uniform(curandStateXORWOW_t* state);

// Επιστρέφει έναν κανονικά κατανομημένο float από έναν Philox4x32-10 generator.
__device__ float curand_normal(curandStatePhilox4_32_10_t* state);

// Επιστρέφει έναν ακέραιο αριθμό από κατανομή Poisson από έναν Sobol32 generator.
__device__ unsigned int curand_poisson(curandStateSobol32_t* state, double lambda);

// Επιστρέφει έναν log-normally κατανομημένο double από έναν MRG32k3a generator.
__device__ double curand_log_normal_double(curandStateMRG32k3a_t* state,
double mean, double stddev);

// Επιστρέφει έναν κανονικά κατανομημένο double από έναν Sobol64 generator.
__device__ double curand_normal_double(curandStateSobol64_t* state);

```

Εικόνα 19: Συναρτήσεις για παραγωγή τυχαίων αριθμών από διαφορετικές κατανομές.

2.5 Βιβλιοθήκη NPP(Nvidia Performance Primitives)

Η βιβλιοθήκη NVIDIA Performance Primitives (NPP) της NVIDIA είναι μια συλλογή εργαλείων για την επεξεργασία εικόνας και σήματος, βελτιστοποιημένη για χρήση σε GPUs μέσω της πλατφόρμας CUDA. Κυρίως στόχος της είναι η μεταφορά υπολογιστικά βαριών εργασιών από την CPU στην GPU, διευκολύνοντας τους προγραμματιστές να δημιουργήσουν εφαρμογές υψηλών επιδόσεων με ευκολία. Η NPP υποστηρίζει τόσο βασικές όσο και σύνθετες λειτουργίες, προσφέροντας εργαλεία που καλύπτουν τις ανάγκες πολλών πεδίων, όπως η επεξεργασία βίντεο σε πραγματικό χρόνο, η βιομηχανική αυτοματοποίηση και η τεχνητή νοημοσύνη.

Η βιβλιοθήκη NPP μπορεί να αξιοποιηθεί σε διάφορα σενάρια χρήσης, παρέχοντας σημαντικά πλεονεκτήματα στην ανάπτυξη και τη βελτίωση εφαρμογών. Ένα βασικό σενάριο είναι η επιτάχυνση υπαρχόντων προγραμμάτων, όπου οι λειτουργίες της βιβλιοθήκης ενσωματώνονται για να αξιοποιήσουν την υπολογιστική ισχύ της GPU. Αυτό οδηγεί σε σημαντική αύξηση της απόδοσης, μειώνοντας τον χρόνο εκτέλεσης της υπάρχουσας λογικής. Παράλληλα, η NPP διευκολύνει τη διαχείριση σύνθετων προγραμμάτων, προσφέροντας ένα ευρύ φάσμα έτοιμων λειτουργιών που μειώνουν την πολυπλοκότητα της ανάπτυξης. Με αυτόν τον τρόπο, οι προγραμματιστές μπορούν να επικεντρωθούν στις βασικές πτυχές του έργου τους, μειώνοντας τον απαιτούμενο χρόνο προγραμματισμού και επιτυγχάνοντας αποδοτικότερα αποτελέσματα.

Η NPP αποτελεί ένα κρίσιμο εργαλείο για προγραμματιστές που δραστηριοποιούνται στον τομέα της ανάλυσης και επεξεργασίας εικόνας, καθώς και σε εφαρμογές που απαιτούν εντατική υπολογιστική ισχύ. Μέσω της πλούσιας συλλογής έτοιμων λειτουργιών και της δυνατότητας ενσωμάτωσης σε σύνθετα συστήματα, η NPP καθιστά την ανάπτυξη αποδοτικών εφαρμογών πιο προσιτή και αποτελεσματική για τους ειδικούς του χώρου διευκολύνοντας την επεξεργασία μεγάλου όγκου δεδομένων.

2.5.1 Λειτουργικότητα της NPP

Η NPP προσφέρει πάνω από 5.000 προκαθορισμένες λειτουργίες, ειδικά σχεδιασμένες για επεξεργασία εικόνας και σήματος. Στον τομέα της επεξεργασίας εικόνας, οι χρήστες μπορούν να επωφεληθούν από αριθμητικές και λογικές πράξεις, μετατροπές χρώματος, γεωμετρικούς μετασχηματισμούς και φιλτράρισμα. Επιπλέον, υποστηρίζονται στατιστικές μέθοδοι, μορφολογικές πράξεις, καθώς και λειτουργίες για κατωφλίωση και διαχείριση μνήμης εικόνας.

Στην επεξεργασία σήματος, η βιβλιοθήκη περιλαμβάνει λειτουργίες όπως αριθμητικές πράξεις, φιλτράρισμα και μετατροπές σήματος. Επίσης, προσφέρει εργαλεία για την αρχικοποίηση και τη διαχείριση μνήμης, καθώς και στατιστικές αναλύσεις, επιτρέποντας στους χρήστες να αναπτύξουν αποδοτικές και ευέλικτες εφαρμογές επεξεργασίας σήματος.

Δομή και Αρχιτεκτονική

Η αρχιτεκτονική της NPP έχει σχεδιαστεί με γνώμονα την ευκολία ενσωμάτωσής της σε προγράμματα που αξιοποιούν τους πόρους της GPU, παρέχοντας στους προγραμματιστές ένα ευέλικτο εργαλείο για τη γρήγορη ανάπτυξη και εκτέλεση εφαρμογών υψηλών απαιτήσεων. Ένα από τα σημαντικότερα χαρακτηριστικά της είναι ότι αποτελεί μια stateless βιβλιοθήκη, γεγονός που σημαίνει ότι δεν διατηρεί εσωτερική κατάσταση ή δεδομένα μεταξύ διαδοχικών κλήσεων. Αυτή η προσέγγιση επιτρέπει τη βέλτιστη αξιοποίηση των διαθέσιμων πόρων της GPU και διευκολύνει τη διαχείριση της παράλληλης επεξεργασίας.

Η βιβλιοθήκη NPP είναι οργανωμένη σε ξεχωριστές υποκατηγορίες, κάθε μία από τις οποίες εξυπηρετεί συγκεκριμένες λειτουργίες επεξεργασίας εικόνας και σήματος. Αυτή η διαίρεση συνδυάζεται με τη δομή της βιβλιοθήκης σε ανεξάρτητα αρχεία lib, επιτρέποντας στους προγραμματιστές να κάνουν link μόνο τα απαραίτητα στοιχεία για το πρόγραμμά τους. Αυτός ο σχεδιασμός συμβάλλει στη μείωση του χρόνου φόρτωσης

και ανάπτυξης, καθώς διευκολύνει την εστίαση μόνο στις λειτουργίες που είναι απαραίτητες για την εκάστοτε εφαρμογή. Οι διάφορες υποκατηγορίες της βιβλιοθήκης, οι οποίες εξειδικεύονται σε διαφορετικά είδη λειτουργιών, αναλύονται διεξοδικά στα επόμενα τμήματα.

Αρχεία κεφαλίδας

Για να αποκτήσει ο προγραμματιστής πρόσβαση στις δυνατότητες της βιβλιοθήκης, αρκεί να ενσωματώσει το αρχείο «npp.h», το οποίο περιλαμβάνει ολόκληρη τη λειτουργικότητα που προσφέρεται. Η δήλωση του αρχείου «npp.h» αρκεί για πλήρη πρόσβαση στην λειτουργικότητα της βιβλιοθήκης

Η αρχιτεκτονική της NPP είναι οργανωμένη σε τρεις κύριες ομάδες βιβλιοθηκών, καθεμία από τις οποίες εξυπηρετεί διαφορετικούς σκοπούς και περιλαμβάνει συγκεκριμένες κατηγορίες λειτουργιών. Η Core Library (NPPC) αποτελεί τη βασική βιβλιοθήκη, παρέχοντας θεμελιώδεις λειτουργίες που είναι κοινές και απαραίτητες για τη λειτουργία των υπόλοιπων βιβλιοθηκών. Η Image Processing Library (NPPI) εστιάζει στην επεξεργασία εικόνας και προσφέρει ένα ευρύ φάσμα λειτουργιών, όπως αριθμητικές και λογικές πράξεις, μετατροπές χρώματος, γεωμετρικούς μετασχηματισμούς, φιλτράρισμα και στατιστικές μεθόδους. Οι συναρτήσεις της βιβλιοθήκης NPPI ορίζονται στο αρχείο «npipi.h» και στα σχετικά αρχεία κεφαλίδας με την ονομασία «npipi_XXX.h». Από την άλλη πλευρά, η Signal Processing Library (NPPS) είναι σχεδιασμένη για την επεξεργασία σήματος, παρέχοντας λειτουργίες όπως αριθμητικές πράξεις, φιλτράρισμα και μετατροπές σήματος. Όλες οι συναρτήσεις που αφορούν την επεξεργασία σήματος ορίζονται στο αρχείο «npps.h» και στα σχετικά αρχεία κεφαλίδας. Αυτή η δομή διασφαλίζει αποδοτικότητα και ευελιξία, καθώς επιτρέπει τη φόρτωση μόνο των απαραίτητων λειτουργιών, μειώνοντας την κατανάλωση πόρων.

2.5.2 Λειτουργίες της βιβλιοθήκης για επεξεργασία εικόνας NPPC (Core NPP)

Η NPPC αποτελεί τον πυρήνα για τη διαχείριση της βιβλιοθήκης και παρέχει βοηθητικές λειτουργίες καθώς και συναρτήσεις ανάκτησης πληροφοριών. Σκοπός της είναι να διευκολύνει την πρόσβαση σε πληροφορίες σχετικά με τη βιβλιοθήκη και τη συσκευή GPU, καθώς και να επιτρέπει τη διαχείριση του περιβάλλοντος εκτέλεσης. Μέσα από τη NPPC, παρέχονται συναρτήσεις όπως η *nppGetLibVersion*, που επιστρέφει την έκδοση της βιβλιοθήκης, και η *nppGetGpuNumSMs*, η οποία παρέχει πληροφορίες για τον αριθμό

των πολυεπεξεργαστών της GPU. Επιπλέον, λειτουργίες όπως οι *nppGetMaxThreadsPerBlock* και *nppGetMaxThreadsPerSM* επιτρέπουν την ανίχνευση των μέγιστων διαθέσιμων threads ανά block ή ανά πολυεπεξεργαστή.

NPPIAL (Arithmetic and Logical Operations Library):

Η κατηγορία NPPIAL (Arithmetic and Logical Operations Library) περιέχει συναρτήσεις για την εκτέλεση αριθμητικών και λογικών πράξεων για την επεξεργασία εικόνας, παρέχοντας λειτουργίες που διευκολύνουν την αποδοτική διαχείριση δεδομένων. Οι αριθμητικές πράξεις περιλαμβάνουν όλες τις γνωστές βασικές λειτουργίες, όπως η πρόσθεση, η αφαίρεση, ο πολλαπλασιασμός και η διαίρεση, αλλά και άλλες για πράξεις με τα δεδομένα των πίξελ. Στις λογικές λειτουργίες περιλαμβάνονται οι πράξεις AND, OR, XOR και NOT, όπως και λειτουργίες μετατόπισης bits, όπως Rshift και Lshift. Επιπλέον, παρέχονται 2 λειτουργίες για σύνθεση εικόνων οι οποίες επιτρέπουν τον συνδυασμό δύο εικόνων με χρήση τιμών διαφάνειας.

NPPICC (Color Conversion Library):

Η κατηγορία NPPICC (Color Conversion Library) της βιβλιοθήκης NPP επικεντρώνεται στις λειτουργίες μετατροπής χρωμάτων και δειγματοληψίας, επιτρέποντας την επεξεργασία εικόνων με διαφορετικά χρωματικά μοντέλα και μορφές δειγματοληψίας. Μέσω των συναρτήσεων της, μπορεί να μετατραπούν έγχρωμες εικόνες σε κλίμακα του γκρι με υψηλή ακρίβεια, ενώ η ανακατασκευή εικόνων από δεδομένα RAW, με χρήση του μηχανισμού Color Debayer που υποστηρίζει το Bayer pattern, εξασφαλίζει την ορθή επεξεργασία των εικόνων. Επιπλέον, λειτουργίες όπως η GammaFwd εφαρμόζουν γάμμα διόρθωση για την προσαρμογή της φωτεινότητας και του contrast, ενώ οι Complement Color Key και ColorTwist (με δυνατότητα batch μέσω της ColorTwistBatch) επιτρέπουν την δημιουργία συμπληρωματικών χρωμάτων και την εφαρμογή προσαρμοσμένων μετασχηματισμών στον χρωματικό χώρο.

Παρέχονται επίσης συναρτήσεις για τη μετατροπή μεταξύ διαφορετικών μορφών δειγματοληψίας (π.χ. από YUV σε RGB ή από υποδειγματοληπτημένη σε πλήρη μορφή) καθώς και μεταξύ διαφόρων χρωματικών μοντέλων, όπως RGB, CMYK και HSV, εξασφαλίζοντας την απαιτούμενη ευελιξία σε εφαρμογές επεξεργασίας εικόνων, συμπεριλαμβανομένων και των JPEG για υψηλή ποιότητα κατά την κωδικοποίηση και αποκωδικοποίηση.

NPPIDEI (Data Exchange and Initialization Library):

Η κατηγορία NPPIDEI (Data Exchange and Initialization Library) της βιβλιοθήκης NPP παρέχει ένα πλήρες σύνολο εργαλείων για την αρχικοποίηση, αντιγραφή και διαχείριση δεδομένων εικόνας, διευκολύνοντας την επεξεργασία και τη μετατροπή τους με υψηλή ακρίβεια. Μέσω των λειτουργιών της, επιτυγχάνεται ο ορισμός τιμών σε επιλεγμένες περιοχές (ROI) με εντολές όπως η Set ή η Masked Set, ενώ παρέχεται και η δυνατότητα επιλεκτικής επεξεργασίας καναλιών, όπως με τη λειτουργία Channel Set.

Η βιβλιοθήκη διευκολύνει την εξαγωγή και εισαγωγή δεδομένων σε συγκεκριμένα κανάλια εικόνων πολλαπλών καναλιών, καθιστώντας την αναδιάταξη των χρωματικών δεδομένων απλή και αποδοτική. Παράλληλα, υποστηρίζει προηγμένες μετατροπές που επιτρέπουν την προσαρμογή του βάθους bit (π.χ. από 8-bit σε 16-bit) και λειτουργίες όπως η Transpose, η Channel Swap και η Sub-Pixel Copy συμβάλλουν στην περαιτέρω επεξεργασία εικόνων.

NPPIF (Filtering Functions Library):

Η κατηγορία NPPIF (Filtering Functions Library) παρέχει ένα ευρύ φάσμα εργαλείων για το φιλτράρισμα εικόνων, καθιστώντας την αναπόσπαστο μέρος της επεξεργασίας οπτικών δεδομένων. Μέσω αυτών, επιτυγχάνεται η εφαρμογή τόσο γραμμικών όσο και μη γραμμικών φίλτρων που βασίζονται σε τεχνικές επεξεργασίας γειτονικών pixel, επιτρέποντας την αναβάθμιση, τη διόρθωση και την ανάλυση των χαρακτηριστικών της εικόνας.

Επιπλέον, η βιβλιοθήκη υποστηρίζει διαδικασίες τμηματοποίησης και ανάλυσης, διευκολύνοντας την εξαγωγή σημαντικών περιοχών και την προετοιμασία των δεδομένων για περαιτέρω επεξεργασία.

NPPIG (Geometry Transformations Library):

Η κατηγορία NPPIG (Geometry Transformations Library) περιλαμβάνει ένα σύνολο συναρτήσεων που επιτρέπουν την εκτέλεση γεωμετρικών μετασχηματισμών σε εικόνες. Αυτές οι συναρτήσεις λειτουργούν σε ορθογώνιες περιοχές ενδιαφέροντος (ROI) στις εικόνες παραμέτρους, με την επεξεργασία να περιορίζεται στα εικονοστοιχεία που βρίσκονται στη διασταύρωση αυτών των περιοχών. Η τυπική διαδικασία περιλαμβάνει τη μετατροπή του ROI της πηγής, στο χώρο του προορισμού, δημιουργώντας ένα τετράπλευρο, και την εγγραφή μόνο των εικονοστοιχείων που βρίσκονται στη διασταύρωση της μετασχηματισμένης ROI πηγής και του ROI του προορισμού.

Η βιβλιοθήκη υποστηρίζει βασικές λειτουργίες όπως μετασχηματισμούς προσαρμογής (Affine Transforms), μετασχηματισμούς Perspective Transforms, και λειτουργίες για περιστροφή (Rotate, Rotate Utility Functions), αναστροφή (Mirror), και προσαρμογή μεγέθους εικόνων (Resize, ResizeBatch, ResizeSqrPixel). Παρέχει επίσης τη δυνατότητα επαναχαρτογράφησης (Remap), επιτρέποντας την ανακατανομή των εικονοστοιχείων μιας εικόνας σύμφωνα με έναν ορισμένο χάρτη.

Σημαντική είναι η χρήση παρεμβολής (Pixel Interpolation) για την ακρίβεια στους μετασχηματισμούς, εξασφαλίζοντας ομαλές και ποιοτικές αναπαραστάσεις των εικόνων μετά τις γεωμετρικές αλλαγές.

NPPIIM (Morphological Operations Library):

Η κατηγορία NPPIIM (Morphological Operations Library) της βιβλιοθήκης NPP παρέχει ένα ευρύ φάσμα λειτουργιών μορφολογικής επεξεργασίας που επιτρέπουν την ανάλυση και τροποποίηση της δομής των εικόνων, τόσο για binary όσο και για grayscale δεδομένα. Συγκεκριμένα, οι λειτουργίες αυτές περιλαμβάνουν:

- Διάβρωση και Διεύρυνση: Εφαρμογή τεχνικών διάβρωσης (erosion) για τη μείωση των λεπτομερειών και διεύρυνσης (dilation) για την έμφαση σε συγκεκριμένες περιοχές.
- Συνδυαστικές λειτουργίες που βοηθούν στην εξάλειψη του θορύβου και την εξομάλυνση των περιγραμμάτων των αντικειμένων, διατηρώντας ταυτόχρονα τα σημαντικά χαρακτηριστικά της εικόνας.
- Εξειδικευμένοι Μετασχηματισμοί: Δυνατότητα προσαρμογής των μορφολογικών λειτουργιών για την ανάδειξη και ανάλυση συγκεκριμένων γεωμετρικών χαρακτηριστικών, βελτιώνοντας την ποιότητα της επεξεργασίας εικόνας.

Με αυτές τις λειτουργίες, η NPPIIM επιτρέπει την αποτελεσματική μεταβολή της γεωμετρίας των εικόνων, καθιστώντας την ιδανική για εφαρμογές που απαιτούν ακριβή μορφολογική ανάλυση και επεξεργασία.

NPPIST (Statistics and Linear Transform Library):

Η βιβλιοθήκη NPPIST (Statistics and Linear Transform Library) παρέχει ένα σύνολο βασικών λειτουργιών για την στατιστική ανάλυση και τους γραμμικούς μετασχηματισμούς εικόνων, διευκολύνοντας την εξαγωγή πληροφοριών από τα οπτικά δεδομένα και την προεπεξεργασία τους σε εφαρμογές υπολογιστικής όρασης. Μέσα από τις συναρτήσεις της, μπορεί να υπολογιστεί το ελάχιστο και το μέγιστο pixel μιας εικόνας μαζί με τις θέσεις τους (μέσω των λειτουργιών minIndex και maxIndex), καθώς και το

συνολικό άθροισμα των τιμών των pixel, το οποίο χρησιμεύει στην εκτίμηση της συνολικής φωτεινότητας. Επιπλέον, η βιβλιοθήκη υποστηρίζει πιο σύνθετες στατιστικές μετρήσεις όπως ο υπολογισμός της μέσης τιμής, της διακύμανσης, της τυπικής απόκλισης και της ενέργειας, παρέχοντας έτσι λεπτομερείς πληροφορίες για την κατανομή των τιμών της εικόνας είτε σε ολόκληρο το πλαίσιο είτε σε συγκεκριμένες περιοχές ενδιαφέροντος (ROI).

Όσον αφορά τους γραμμικούς μετασχηματισμούς εικόνας, η NPPIST ενσωματώνει λειτουργίες που επιτρέπουν την εφαρμογή μαθηματικών μετασχηματισμών στα δεδομένα των εικόνων, διευκολύνοντας εργασίες όπως η κανονικοποίηση και η ενίσχυση της αντίθεσης. Ένας βασικός τύπος μετασχηματισμού αφορά τη γραμμική προσαρμογή των τιμών των pixel, όπου οι εισερχόμενες τιμές τροποποιούνται σύμφωνα με μια γραμμική σχέση για την παραγωγή των τελικών αποτελεσμάτων. Επιπλέον, υποστηρίζονται μετασχηματισμοί Fourier, οι οποίοι είναι κρίσιμοι για την ανάλυση συχνοτήτων σε εικόνες και βρίσκουν εφαρμογή σε λειτουργίες φιλτραρίσματος, ανίχνευσης μοτίβων και συμπίεσης, μέσω της μετατροπής του χώρου των τιμών σε ένα φάσμα συχνοτήτων που επιτρέπει την αποδοτικότερη επεξεργασία.

NPPIST (Memory Support Functions Library):

Η κατανομή και η διαχείριση αποθηκευτικού χώρου για εικόνες αποτελεί ένα βασικό στοιχείο της απόδοσης στις εφαρμογές που χρησιμοποιούν τη βιβλιοθήκη NPP. Οι ρουτίνες που περιέχονται εδώ διευκολύνουν την εκχώρηση και αποδέσμευση μνήμης για δεδομένα εικόνας, ενώ ενσωματώνουν δυνατότητες που βελτιώνουν την απόδοση μέσω ευθυγράμμισης της μνήμης, λαμβάνοντας υπόψη παραμέτρους όπως το πλάτος και το ύψος της εικόνας, και επιστρέφουν τον αριθμό των byte μεταξύ διαδοχικών γραμμών (line pitch).

Η χρήση του padding, δηλαδή της προσθήκης επιπλέον byte στο τέλος κάθε γραμμής pixel, αν και δεν είναι υποχρεωτική, μπορεί να αποτρέψει σημαντική υποβάθμιση της απόδοσης, ιδίως σε συστήματα όπου η ευθυγράμμιση μνήμης παίζει κρίσιμο ρόλο.

Να σημειωθεί ότι η χρήση των κατανομών αυτών είναι προαιρετική, καθώς οποιοσδήποτε έγκυρος δείκτης μνήμης συσκευής CUDA μπορεί να χρησιμοποιηθεί από τις βασικές λειτουργίες της NPP, χωρίς περιορισμούς στα βήματα γραμμής.

NPPITC (Threshold and Compare Operations Library):

Η βιβλιοθήκη NPPITC (Threshold and Compare Operations Library) εξειδικεύεται στις λειτουργίες κατωφλίωσης και συγκρίσεων, παρέχοντας εργαλεία για την ανάλυση και

την τροποποίηση εικόνων σε επίπεδο pixel. Οι μέθοδοι κατωφλίου επιτρέπουν τον καθορισμό ορίων που εφαρμόζονται στα pixels μιας εικόνας, ενώ οι μέθοδοι σύγκρισης επιτρέπουν τη δημιουργία δυαδικών εικόνων που βασίζονται σε συγκρίσεις μεταξύ εικόνων ή μεταξύ εικόνας και σταθερών τιμών.

2.5.3 Περιγραφή των λειτουργιών για επεξεργασία σήματος

Signal Arithmetic And Logical Operations

Περιλαμβάνουν λειτουργίες που εφαρμόζουν αριθμητικές και λογικές πράξεις στα δεδομένα σήματος. Αυτές οι λειτουργίες επιτρέπουν την εκτέλεση βασικών αριθμητικών πράξεων, όπως προσθέσεις, αφαιρέσεις, πολλαπλασιασμούς και διαιρέσεις μεταξύ σημάτων ή μεταξύ σήματος και σταθερής τιμής. Επιπλέον, παρέχουν τη δυνατότητα εκτέλεσης λογικών πράξεων, όπως AND, OR, XOR και NOT, για την επεξεργασία δεδομένων σε δυαδική μορφή.

Signal Conversion Functions

Εδώ μας δίνονται λειτουργίες που επιτρέπουν τη μετατροπή των δεδομένων σήματος από έναν τύπο σε άλλο ή την εφαρμογή κατωφλίου. Η υποκατηγορία "Signal Convert" χρησιμοποιείται για την προσαρμογή του εύρους ή του τύπου των δεδομένων, ενώ η "Signal Threshold" εφαρμόζει λειτουργίες κατωφλίου για την ταξινόμηση των τιμών ενός σήματος, θέτοντας τιμές κάτω ή πάνω από ένα όριο σε καθορισμένες τιμές.

Signal Filtering Functions

Προσφέρει εργαλεία για την εφαρμογή φίλτρων σε σήματα, με στόχο τη βελτίωση ή την εξαγωγή χρήσιμων χαρακτηριστικών. Η λειτουργία "Integral" υπολογίζει την ολοκλήρωση του σήματος, μια μέθοδος που μπορεί να χρησιμοποιηθεί για την εξαγωγή συσσωρευτικών τιμών ή την ανάλυση χαρακτηριστικών σε διαφορετικά τμήματα του σήματος.

Signal Initialization Functions

Περιλαμβάνει λειτουργίες για την αρχικοποίηση σήματος. Η "Signal set" επιτρέπει την εκχώρηση σταθερών τιμών σε ένα σήμα, ενώ η "Signal zero" μηδενίζει τα δεδομένα του. Η "Signal copy" χρησιμοποιείται για τη μεταφορά των δεδομένων σήματος από μια θέση μνήμης σε μια άλλη, παρέχοντας χρήσιμες λειτουργίες για τη διαχείριση και την επεξεργασία δεδομένων.

Signal Statistical Functions

Το Signal Statistical Functions περιλαμβάνει λειτουργίες που παρέχουν στατιστικά στοιχεία για σήματα, απαραίτητες για την ανάλυση δεδομένων και την εξαγωγή στατιστικών χαρακτηριστικών.

```
//Signal Statistical Functions
// nppsStdDev_32f_Ctx: Υπολογίζει την τυπική απόκλιση ενός πίνακα 32-bit float.
NppStatus nppsStdDev_32f_Ctx(const Npp32f *pSrc, size_t nLength, Npp32f *pStdDev, Npp8u *pDeviceBuffer,
    NppStreamContext nppStreamCtx);

// nppsMax_32f_Ctx: Υπολογίζει τη μέγιστη τιμή σε έναν πίνακα 32-bit float.
NppStatus nppsMax_32f_Ctx(const Npp32f *pSrc, size_t nLength, Npp32f *pMax, NppStreamContext nppStreamCtx);

// nppsMin_32f_Ctx: Υπολογίζει τη μικρότερη τιμή σε έναν πίνακα 32-bit float.
NppStatus nppsMin_32f_Ctx(const Npp32f *pSrc, size_t nLength, Npp32f *pMin, NppStreamContext nppStreamCtx);

// nppsMeanStdDev_32f_Ctx: Υπολογίζει ταυτόχρονα τη μέση τιμή και την τυπική απόκλιση ενός πίνακα 32-bit float.
NppStatus nppsMeanStdDev_32f_Ctx(const Npp32f *pSrc, size_t nLength, Npp32f *pMean, Npp32f *pStdDev,
    Npp8u *pDeviceBuffer, NppStreamContext nppStreamCtx);
```

Εικόνα 20: Στατιστικές Συναρτήσεις.

Signal Memory Management Functions

Μερικά εργαλεία για τη διαχείριση της μνήμης σήματος. Αυτές οι λειτουργίες περιλαμβάνουν την εκχώρηση, την απελευθέρωση και την αντιγραφή δεδομένων στη μνήμη, επιτρέποντας την αποτελεσματική χρήση πόρων στη συσκευή CUDA.

2.5.4 Διαχείριση Μνήμης κατά τη Χρήση της Βιβλιοθήκης NPP.

Η διαχείριση μνήμης κατά τη χρήση της βιβλιοθήκης NPP βασίζεται στη λογική ότι κάθε παράμετρος που αποτελεί δείκτη προς δεδομένα πρέπει να αναφέρεται αποκλειστικά στον χώρο μνήμης της GPU. Η μνήμη αυτή μπορεί να κατανεμηθεί και να αρχικοποιηθεί με οποιαδήποτε από τις γενικές μεθόδους διαχείρισης μνήμης που προσφέρει το CUDA περιβάλλον, όπως οι συναρτήσεις `cudaMalloc` και `cudaMemcpy`, καθώς και άλλες συναρτήσεις που σχετίζονται με τη διαχείριση της μνήμης. Αυτή η μέθοδος θεωρείται βέλτιστος, καθώς προσφέρει στον χρήστη πλήρη έλεγχο πάνω στον όγκο της μνήμης που απαιτείται και τη χρονική στιγμή που αυτή θα κατανεμηθεί. Με αυτόν τον τρόπο αποφεύγονται οι σπατάλες στη μεταφορά δεδομένων και ο χρήστης μπορεί να προσαρμόσει το πρόγραμμα στις ανάγκες της εφαρμογής του.

Σε ορισμένες περιπτώσεις, κατά την κλήση συγκεκριμένων συναρτήσεων της NPP, απαιτείται η χρήση ενός προσωρινού χώρου μνήμης για την εκτέλεση της λειτουργίας. Η ανάθεση αυτού του προσωρινού χώρου μνήμης παραμένει ευθύνη του χρήστη, δίνοντάς του τη δυνατότητα να διαχειριστεί καλύτερα τις κατανομές μνήμης στο πρόγραμμα. Αυτή

η προσέγγιση επιτρέπει την αποφυγή περιττών δεσμεύσεων μνήμης και τη χρήση ενός προκαθορισμένου χώρου για πολλαπλές κλήσεις και λειτουργίες, ενισχύοντας περαιτέρω την αποδοτικότητα της εφαρμογής.

Διαχειρισμός δεδομένων εικόνας

Στο κομμάτι NPPI της βιβλιοθήκης NPP, η διαχείριση δεδομένων εικόνας βασίζεται σε έναν απλοποιημένο και ευέλικτο τρόπο αναπαράστασης. Κάθε συνάρτηση που επεξεργάζεται εικόνες λαμβάνει ως παραμέτρους: (α) έναν δείκτη στον υποκείμενο τύπο δεδομένων της εικόνας και (β) ένα βήμα γραμμής (line stride) σε bytes. Αυτός ο σχεδιασμός προσφέρει σημαντικά πλεονεκτήματα, καθώς επιτρέπει τη χρήση της βιβλιοθήκης σε ποικίλες περιπτώσεις χωρίς την ανάγκη πολύπλοκων μετατροπών δεδομένων. Η χρήση ενός ακατέργαστου δείκτη στον υποκείμενο τύπο δεδομένων pixel αποφεύγει τη δημιουργία δομημένων δεδομένων ανά χρώμα, μειώνοντας την ανάγκη για επικίνδυνους τύπους cast ή δαπανηρά αντίγραφα δεδομένων. Επιπλέον, η διαχείριση του δείκτη δεδομένων και του βήματος γραμμής ως ανεξάρτητες παραμέτρους, αντί για μια υψηλότερου επιπέδου δομή εικόνας, εξασφαλίζει ευελιξία και ευκολία υιοθέτησης.

Αυτό σημαίνει ότι η βιβλιοθήκη δεν επιβάλλει κάποια συγκεκριμένη αναπαράσταση εικόνας, επιτρέποντας στις εφαρμογές να χρησιμοποιούν τις δικές τους δομές δεδομένων. Έτσι, αποφεύγεται η ανάγκη ανακατασκευής δεδομένων μια ειδική μορφή εικόνας NPP, γεγονός που καθιστά τη διαδικασία πιο αποδοτική και συμβατή με διάφορα σενάρια χρήσης.

Ευθυγράμμιση Δεδομένων Εικόνας

Η ευθυγράμμιση των δεδομένων μιας εικόνας είναι καθοριστική για την απόδοση των λειτουργιών, ιδιαίτερα στις παλαιότερες γενιές GPUs. Όταν οι εικόνες έχουν μη τυποποιημένα, ασύμμετρα μεγέθη, η απουσία ευθυγράμμισης μπορεί να οδηγήσει σε σημαντική μείωση της ταχύτητας επεξεργασίας. Για αυτό η βιβλιοθήκη κατέχει την ιδιότητα Line Stride όπου αναφέρεται στο μέγεθος του πλάτους μιας γραμμής δεδομένων, με την προσθήκη κενών (padding) όταν απαιτείται. Καθορίζει την απόσταση, σε bytes, μεταξύ διαδοχικών γραμμών εικόνας, δηλαδή το μήκος μιας γραμμής σε bytes. Αυτή η προσέγγιση διασφαλίζει ότι τα δεδομένα της εικόνας είναι σωστά ευθυγραμμισμένα στη μνήμη, γεγονός που βελτιώνει την απόδοση των υπολογισμών και επιτρέπει την καλύτερη αξιοποίηση της μνήμης της GPU. Η ιδιότητα αυτή είναι ιδιαίτερα σημαντική όταν επεξεργαζόμαστε στήλες δεδομένων, καθώς εξασφαλίζει την προβλέψιμη απόσταση μεταξύ διαδοχικών pixels μιας στήλης, ακόμα και σε εικόνες με padding. Ως

αποτέλεσμα, οι λειτουργίες επεξεργασίας εικόνας εκτελούνται με μεγαλύτερη ταχύτητα και αποδοτικότητα, εξαλείφοντας πιθανά προβλήματα ασυμμετρίας ή κακής τοποθέτησης δεδομένων.

Περιοχή Ενδιαφέροντος (ROI) στην Επεξεργασία Εικόνας

Η Περιοχή Ενδιαφέροντος (Region-Of-Interest) στην NPP αναφέρεται σε ένα ορθογώνιο τμήμα της εικόνας όπου εκτελούνται οι λειτουργίες επεξεργασίας. Ορίζεται μέσω μιας δομής που περιλαμβάνει τις συντεταγμένες του πάνω αριστερού σημείου και τις διαστάσεις (πλάτος και ύψος) της περιοχής. Είναι ιδιαίτερα βοηθητικό καθώς στις περισσότερες εφαρμογές είναι πιο σύνθηες η επεξεργασία ενός μέρους μια εικόνας και όχι ολόκληρη. Επίσης σημαντικό είναι για εφαρμογές που απαιτούν τοπική ανάλυση ή επεξεργασία, όπως φιλτράρισμα, μετασχηματισμοί ή στατιστικές λειτουργίες σε συγκεκριμένες περιοχές της εικόνας. Οι συναρτήσεις που υποστηρίζουν ROI εκτελούν υπολογισμούς μόνο εντός αυτής της περιοχής, αφήνοντας ανέπαφα τα υπόλοιπα δεδομένα της εικόνας. Αυτό προσφέρει αυξημένη αποδοτικότητα, καθώς περιορίζει τους υπολογισμούς μόνο στα ενδιαφέροντα τμήματα, μειώνοντας έτσι τον απαιτούμενο χρόνο επεξεργασίας.

Συμβάσεις Επεξεργασίας Σημάτων

Στο τμήμα NPPS της βιβλιοθήκης, η επεξεργασία σημάτων βασίζεται στην απευθείας μετάδοση των δεδομένων μέσω ενός δείκτη στον τύπο του σήματος. Αυτή η προσέγγιση χαμηλού επιπέδου επιτρέπει την ευέλικτη ενσωμάτωση της βιβλιοθήκης σε υπάρχοντα έργα λογισμικού, καθώς δεν απαιτείται η χρήση σύνθετων δομών (π.χ. που περιέχουν πληροφορίες για μέγεθος, μετατόπιση ή πρόσθετα δεδομένα) για την αναπαράσταση των σημάτων.

2.6 Βιβλιοθήκη Thrust

Η Thrust είναι μια βιβλιοθήκη της NVIDIA η οποία παρέχει ένα σύνολο εργαλείων ενσωματωμένα στην γλώσσα c++, προσφέροντας μια διεσπάρη υψηλού επιπέδου στους προγραμματιστές. Ένας από τους κύριους στόχους της είναι να επιτρέπει τη συνεργασία μεταξύ του host (CPU) και του device (GPU) χωρίς την ανάγκη περίπλοκου κώδικα, ενώ η διαισθητική της σύνταξη, παρόμοια με αυτή της βιβλιοθήκης STL (Standard Template Library) της C++, την καθιστά ιδανική για την υλοποίηση αποδοτικών εφαρμογών με ελάχιστη προσπάθεια. Η Thrust αποτελεί μέρος του CUDA Toolkit της NVIDIA, εγκαθίσταται μαζί του, και είναι ένα έργο ανοιχτού κώδικα, επιτρέποντας στους

προγραμματιστές να συνεισφέρουν και να προσαρμόζουν τη λειτουργικότητά της. Συνολικά, είναι χτισμένη πάνω σε υπάρχουσες δομές παραλληλισμού, όπως η OpenMP και η TBB (Threading Building Blocks), προσφέροντας ευελιξία και αποδοτικότητα τόσο σε GPU όσο και σε πολυπύρρηνα περιβάλλοντα CPU.

Για να χρησιμοποιήσει κάποιος τη βιβλιοθήκη Thrust, απαιτούνται συγκεκριμένες προϋποθέσεις σε λογισμικό και υλικό. Η βιβλιοθήκη απαιτεί την εγκατάσταση είτε του NVIDIA HPC SDK είτε του CUDA Toolkit, με τις τελευταίες εκδόσεις να είναι οι συνιστώμενες και επίσημα υποστηριζόμενες. Όσον αφορά τη γλώσσα προγραμματισμού, υποστηρίζονται C++14 και C++17, με την C++11 να θεωρείται πλέον απαρχαιωμένη. Οι υποστηριζόμενοι μεταγλωττιστές περιλαμβάνουν το NVCC, το NVCC++, το GCC 5+, το Clang 7+, και το MSVC 2019+.

Σε επίπεδο υλικού, η Thrust υποστηρίζει όλες τις αρχιτεκτονικές GPU της NVIDIA από την SM 35 και μεταγενέστερες. Όσον αφορά τις αρχιτεκτονικές επεξεργαστών, υποστηρίζονται τα συστήματα x86-64 και aarch64 σε τα Linux και Windows. Για την εκτέλεση ή τη δοκιμή κώδικα Thrust, είναι απαραίτητο το υλικό και το λογισμικό να ικανοποιούν τις παραπάνω απαιτήσεις, διασφαλίζοντας έτσι τη βέλτιστη λειτουργία της βιβλιοθήκης.

2.6.1 Πως χρησιμοποιείται η βιβλιοθήκη

Η βιβλιοθήκη Thrust περιλαμβάνει μια σειρά από εξειδικευμένα include files που διευκολύνουν την υλοποίηση κοινών αλγορίθμων επεξεργασίας δεδομένων. Για παράδειγμα, τα αρχεία όπως το «*thrust/host_vector.h*» και το «*thrust/device_vector.h*» παρέχουν εύχρηστες δομές για τη διαχείριση δεδομένων τόσο στον host όσο και στην GPU, ενώ με αρχεία όπως το «*thrust/sort.h*» και το «*thrust/transform.h*» δίδεται η δυνατότητα χρήσης προκατασκευασμένων αλγορίθμων για ταξινόμηση και μετασχηματισμό, χωρίς να απαιτείται η ανάπτυξη εξειδικευμένου CUDA κώδικα. Αυτή η οργάνωση των include files συμβάλλει στο να διευκολύνει την ενσωμάτωσή της σε υπάρχοντα έργα, είτε αυτά προορίζονται για CPU ή GPU.

Ένα από τα μεγαλύτερα πλεονεκτήματά της είναι η φορητότητα που προσφέρει. Η χρήση της Thrust απαιτεί μόνο την προσθήκη ενός include αρχείου στον κώδικα ανάλογα με την λειτουργία που χρειαζόμαστε, χωρίς την ανάγκη περαιτέρω εξειδικευμένων βημάτων. Η απλότητα αυτή, σε συνδυασμό με τη δυνατότητα αξιοποίησης της υποδομής του CUDA Toolkit, την καθιστά ισχυρό εργαλείο για προγραμματιστές που επιθυμούν

να προσθέσουν παράλληλη επεξεργασία στις εφαρμογές με ευκολία και μέγιστη αποδοτικότητα.

2.6.2 Αναλυτική περιγραφή λειτουργικότητας της βιβλιοθήκης

Η βιβλιοθήκη Thrust παρέχει μια σειρά από δυνατότητες και λειτουργίες που επιτρέπουν την υλοποίηση παραλληλισμένων εφαρμογών με ευκολία και αποτελεσματικότητα. Οι βασικές λειτουργίες της μπορούν να κατηγοριοποιηθούν ως εξής:

Λειτουργίες Αλγορίθμων της Thrust

Η βιβλιοθήκη Thrust παρέχει ένα ευρύ φάσμα αλγορίθμων που επιτρέπουν την αποδοτική και απλή υλοποίηση παραλληλισμένων λειτουργιών. Οι κύριες κατηγορίες λειτουργιών περιλαμβάνουν:

Copying (Αντιγραφή)

Αλγόριθμοι αντιγραφής που επιτρέπουν την αποδοτική μεταφορά δεδομένων μεταξύ διαφορετικών περιοχών μνήμης, είτε συνεχόμενων είτε μη συνεχόμενων.

Merging (Συγχώνευση)

Οι λειτουργίες συγχώνευσης επιτρέπουν τη συνένωση ταξινομημένων δεδομένων, διατηρώντας τη σειρά τους.

Πρόθεμα Αθροισμάτων (Prefix Sums)

Αλγόριθμοι αθροισμάτων προθέματος που επιτρέπουν τον υπολογισμό σωρευτικών αθροισμάτων, χρήσιμα για σάρωση δεδομένων και παραλληλισμένες αριθμητικές πράξεις.

Reductions

Οι αλγόριθμοι reduction χρησιμοποιούνται για την εξαγωγή συνοπτικών αποτελεσμάτων, όπως το άθροισμα, το γινόμενο ή η εύρεση ελαχίστων/μέγιστων από ένα σύνολο δεδομένων.

Reordering (Αναδιάταξη)

Οι λειτουργίες αναδιάταξης επιτρέπουν την αναδιοργάνωση δεδομένων, βελτιώνοντας τη διάταξή τους στη μνήμη για αποδοτικότερη επεξεργασία. Περιλαμβάνουν μεθόδους ταξινόμησης και τυχαίας ανακατάταξης, καθώς και την τεχνική stream compaction, που αφαιρεί ανεπιθύμητα στοιχεία, βελτιώνοντας τη διαχείριση της μνήμης.

Searching (Αναζήτηση)

Η Thrust παρέχει αποδοτικούς αλγορίθμους αναζήτησης, όπως δυαδική αναζήτηση για γρήγορη εύρεση στοιχείων σε ταξινομημένες ακολουθίες.

Set Operations (Λειτουργίες Συνόλων)

Οι αλγόριθμοι συνόλων επιτρέπουν την αποδοτική επεξεργασία και σύγκριση συνόλων δεδομένων, όπως η εύρεση ενώσεων, τομών και διαφορών. Μέσω των λειτουργιών όπως ``thrust::set_union``, ``thrust::set_intersection`` και ``thrust::set_difference``, επιτυγχάνεται γρήγορη και αποτελεσματική διαχείριση μεγάλων όγκων δεδομένων.

Sorting (Ταξινόμηση)

Η Thrust προσφέρει αλγορίθμους ταξινόμησης υψηλής απόδοσης, όπως οι `thrust::sort` και `thrust::stable_sort`, που ταξινομούν δεδομένα με ή χωρίς διατήρηση της σχετικής σειράς.

Transformations (Μετασχηματισμοί)

Οι μετασχηματισμοί επιτρέπουν την εφαρμογή προσαρμοσμένων συναρτήσεων σε στοιχεία δεδομένων, επιταχύνοντας αριθμητικές και λογικές πράξεις. Ο `thrust::transform` εφαρμόζει συναρτήσεις στα δεδομένα, ενώ οι `thrust::sequence` και `thrust::generate` δημιουργούν ακολουθίες αριθμών και τυχαία δεδομένα.

Δομές Δεδομένων (Containers) της Thrust

Η βιβλιοθήκη Thrust παρέχει δύο βασικές δομές δεδομένων, τις `thrust::host_vector` και `thrust::device_vector`, οι οποίες είναι αντίστοιχες των `std::vector` της C++, αλλά σχεδιασμένες για να υποστηρίζουν την αποθήκευση και επεξεργασία δεδομένων τόσο στη μνήμη της CPU (host) όσο και της GPU (device). Αυτές οι δομές εξασφαλίζουν ευκολία και αποδοτικότητα στον παραλληλισμό, με δυνατότητα εναλλαγής μεταξύ host και device περιβάλλοντος.

thrust::host_vector

Η `thrust::host_vector` είναι μια δομή δεδομένων που αποθηκεύει δεδομένα στη μνήμη του host (CPU) και παρέχει λειτουργικότητα παρόμοια με αυτήν της `std::vector` στη C++. Χρησιμοποιείται για την αποθήκευση και επεξεργασία δεδομένων που βρίσκονται στη μνήμη του host, ενώ ταυτόχρονα διευκολύνει τη μεταφορά τους στη GPU όταν αυτό είναι απαραίτητο. Η δομή αυτή επιτρέπει εύκολη ανάγνωση και εγγραφή δεδομένων από και προς τη μνήμη του συστήματος, καθιστώντας την ιδανική επιλογή για εφαρμογές που απαιτούν διαχείριση δεδομένων σε περιβάλλον CPU.

thrust::device_vector

Η `thrust::device_vector` είναι και αυτή παρόμοια με την δομή δεδομένων `host_vector` αλλά σχεδιασμένη για την αποθήκευση και επεξεργασία δεδομένων στη μνήμη του device (GPU). Παρέχει αυτοματοποιημένη διαχείριση μνήμης, αναλαμβάνοντας τη διαδικασία κατανομής και αποδέσμευσης χωρίς να απαιτείται πρόσθετη παρέμβαση από τον προγραμματιστή. Επιπλέον, δίνει τη δυνατότητα αξιοποίησης της υπολογιστικής

ισχύος της GPU για παραλληλισμένες λειτουργίες, χωρίς την ανάγκη χειροκίνητης μεταφοράς δεδομένων από και προς τη μνήμη της συσκευής, καθιστώντας τη μια εύχρηστη και αποδοτική λύση για τον παραλληλισμό.

Function Objects

Τα functors (ή function objects) είναι αντικείμενα που έχουν οριστεί ώστε να συμπεριφέρονται όπως συναρτήσεις. Αυτό επιτυγχάνεται μέσω της υπερφόρτωσης του τελεστή κλήσης, δηλαδή της συνάρτησης operator(). Με άλλα λόγια, ένα functor είναι ένας τύπος αντικειμένου που μπορεί να κληθεί σαν συνάρτηση, επιτρέποντας τον ενσωματωμένο χειρισμό λογικής και παραμέτρων. Επιπλέον, σε αντίθεση με τους δείκτες σε συναρτήσεις, τα functors μπορούν να διατηρούν εσωτερική κατάσταση, προσφέροντας μεγαλύτερη ευελιξία και δυνατότητα παραμετροποίησης όταν χρησιμοποιούνται ως παράμετροι σε αλγοριθμικές συναρτήσεις ή σε παραλληλισμένες εφαρμογές, όπως με την `thrust::transform` στη βιβλιοθήκη Thrust.

Iterators

Στη βιβλιοθήκη Thrust, οι Iterators διαδραματίζουν κρίσιμο ρόλο, επιτρέποντας την αφηρημένη πρόσβαση και διαχείριση δεδομένων τόσο στη μνήμη του host (CPU) όσο και του device (GPU). Η Thrust προσφέρει μια ποικιλία από "fancy iterators", όπως:

- `thrust::make_counting_iterator`: Δημιουργεί έναν iterator που παράγει μια ακολουθία αυξανόμενων τιμών, χρήσιμο για την παραγωγή σειρών αριθμών.
- `thrust::make_transform_iterator`: Συνδυάζει έναν iterator με μια συνάρτηση μετασχηματισμού, επιτρέποντας την εφαρμογή της συνάρτησης σε κάθε στοιχείο κατά την πρόσβαση.
- `thrust::make_zip_iterator`: Επιτρέπει την ταυτόχρονη πρόσβαση σε πολλαπλά containers, διευκολύνοντας τη συγχρονισμένη επεξεργασία δεδομένων.
- `thrust::make_permutation_iterator`: Παρέχει πρόσβαση σε στοιχεία με βάση έναν πίνακα δεικτών, επιτρέποντας την αναδιάταξη των δεδομένων χωρίς αντιγραφή.

Αυτοί οι iterators προσφέρουν σημαντικά πλεονεκτήματα, όπως:

- Ευελιξία: Επιτρέπουν την εύκολη προσαρμογή και σύνθεση λειτουργιών, διευκολύνοντας την ανάπτυξη σύνθετων αλγορίθμων.
- Αποδοτικότητα: Μειώνουν την ανάγκη για περιττές αντιγραφές δεδομένων, βελτιώνοντας την απόδοση των εφαρμογών.

- **Φορητότητα:** Διευκολύνουν τη συγγραφή κώδικα που μπορεί να εκτελεστεί τόσο σε CPU όσο και σε GPU, προάγοντας την επαναχρησιμοποίηση και τη συντηρησιμότητα του κώδικα.

Επιπλέον, η Thrust παρέχει διάφορα iterator tags, όπως:

- `thrust::random_access_device_iterator_tag`: Υποδηλώνει ότι ένας iterator υποστηρίζει τυχαία πρόσβαση σε δεδομένα στη μνήμη της συσκευής.
- `thrust::input_host_iterator_tag`: Υποδηλώνει ότι ένας iterator υποστηρίζει σειριακή ανάγνωση δεδομένων από τη μνήμη του host.

Αυτά τα tags βοηθούν στη βελτιστοποίηση των αλγορίθμων, επιτρέποντας την προσαρμογή τους ανάλογα με τις δυνατότητες των iterators που χρησιμοποιούνται.

Memory Management

Παρέχει ένα σύνολο εργαλείων για αποτελεσματική διαχείριση μνήμης μεταξύ του host (CPU) και του device (GPU). Αυτά τα εργαλεία επιτρέπουν στους προγραμματιστές να διαχειρίζονται τη μνήμη με τρόπο παρόμοιο με τις τυπικές βιβλιοθήκες της C++, διευκολύνοντας την ανάπτυξη παραλληλισμένων εφαρμογών. Μερικές από τις βασικές λειτουργίες διαχείρισης μνήμης που προσφέρει η Thrust περιλαμβάνουν:

- `thrust::device_malloc` και `thrust::device_free`: Χρησιμοποιούνται για την κατανομή και αποδέσμευση μνήμης στη συσκευή (GPU), αντίστοιχα.
- `thrust::device_new` και `thrust::device_delete`: Επιτρέπουν την κατανομή και αποδέσμευση αντικειμένων στη μνήμη της συσκευής, παρέχοντας λειτουργικότητα παρόμοια με τους τελεστές `new` και `delete` της C++.
- `thrust::raw_pointer_cast`: Διευκολύνει τη μετατροπή έξυπνων δεικτών της Thrust σε ακατέργαστους δείκτες, επιτρέποντας τη διαλειτουργικότητα με κώδικα CUDA που απαιτεί ακατέργαστους δείκτες.
- `thrust::get_temporary_buffer` και `thrust::return_temporary_buffer`: Παρέχουν μηχανισμούς για την προσωρινή κατανομή και αποδέσμευση μνήμης, χρήσιμους σε περιπτώσεις όπου απαιτείται προσωρινός χώρος αποθήκευσης κατά την εκτέλεση αλγορίθμων.

Parallel Execution Policies

Στη βιβλιοθήκη Thrust, οι πολιτικές παράλληλης εκτέλεσης καθορίζουν πού και πώς θα εκτελούνται οι αλγόριθμοι—στη CPU (host) ή στη GPU (device). Η `thrust::host_execution_policy` είναι η βασική κλάση για όλες τις πολιτικές εκτέλεσης που

σχετίζονται με το σύστημα του host, επιτρέποντας την παράλληλη εκτέλεση αλγορίθμων στην CPU. Αντίστοιχα, η `thrust::device_execution_policy` είναι η βασική κλάση για τις πολιτικές που σχετίζονται με το σύστημα του device, διευκολύνοντας την εκτέλεση αλγορίθμων στην GPU.

Οι προκαθορισμένες πολιτικές `thrust::host` και `thrust::device` επιτρέπουν στους προγραμματιστές να καθορίζουν ρητά αν ένας αλγόριθμος θα εκτελεστεί στην CPU ή στην GPU, αντί να βασίζονται στην αυτόματη ανάθεση βάσει των τύπων των δεικτοδοτών. Αυτή η ρητή ανάθεση μπορεί να αποτρέψει περιττές αντιγραφές δεδομένων και να βελτιώσει την απόδοση. Είναι σημαντικό να διασφαλίζεται ότι οι δεικτοδοτές που παρέχονται σε έναν αλγόριθμο είναι συμβατοί με το επιλεγμένο σύστημα εκτέλεσης, ώστε να αποφεύγονται σφάλματα κατά την εκτέλεση.

Random Number Generators

Η βιβλιοθήκη Thrust της NVIDIA παρέχει επιπλέον ένα API για την παραγωγή τυχαίων αριθμών που περιλαμβάνει διάφορες γεννήτριες τυχαίων αριθμών και κατανομές, επιτρέποντας στους προγραμματιστές να δημιουργούν τυχαίες ακολουθίες με συγκεκριμένες στατιστικές ιδιότητες.

Μεταξύ των γεννητριών που προσφέρει είναι οι `minstd_rand`, `ranlux24_base`, `ranlux48`, `default_random_engine`, `ranlux24`, `taus88`, `ranlux48_base` και `minstd_rand0`. Κάθε μία από αυτές έχει διαφορετικά χαρακτηριστικά όσον αφορά την ταχύτητα, την ποιότητα και την περίοδο των παραγόμενων τυχαίων αριθμών, επιτρέποντας την επιλογή της κατάλληλης γεννήτριας ανάλογα με τις απαιτήσεις της εφαρμογής.

Επιπλέον, το API της Thrust υποστηρίζει διάφορες κατανομές τυχαίων αριθμών, όπως την κανονική (normal distribution), την ομοιόμορφη (uniform distribution) και άλλες, επιτρέποντας την παραγωγή τυχαίων αριθμών που ακολουθούν συγκεκριμένες στατιστικές κατανομές.

Η ενσωμάτωση αυτών των λειτουργιών στο περιβάλλον της GPU μέσω της Thrust επιτρέπει την αποδοτική παραγωγή τυχαίων αριθμών σε παράλληλα υπολογιστικά περιβάλλοντα, βελτιώνοντας την απόδοση και την κλιμάκωση των εφαρμογών που βασίζονται σε τυχαίες διαδικασίες. Η χρήση των γεννητριών και των διανομών της Thrust είναι απλή και συνεπής με το πρότυπο της C++, διευκολύνοντας την ενσωμάτωσή τους σε υπάρχοντα έργα.

2.7 Αναφορές

J. Dongarra, κ.α, "An updated set of Basic Linear Algebra Subprograms (BLAS)", ACM Transactions on Mathematical Software, 1990.

D. E. Knuth, "The Art of Computer Programming, Volume 2: Seminumerical Algorithms," 3rd ed., Addison-Wesley Professional, 1997.

3 Προετοιμασία για Υλοποίηση Εφαρμογών σε Περιβάλλον CUDA

Για την ανάπτυξη εφαρμογών σε CUDA, απαιτείται ένα πλήρες και σωστά ρυθμισμένο προγραμματιστικό περιβάλλον που να επιτρέπει τη βέλτιστη αξιοποίηση των δυνατοτήτων της GPU. Στο παρόν κεφάλαιο, θα παρουσιαστούν οι βασικές συνιστώσες που είναι απαραίτητες για την προετοιμασία και διαμόρφωση ενός τέτοιου περιβάλλοντος. Ειδικότερα, θα αναλυθούν τα εργαλεία και οι βιβλιοθήκες που θα χρησιμοποιηθούν στις εφαρμογές του επόμενου κεφαλαίου, καθώς και η διαδικασία εγκατάστασής τους.

Το CUDA Toolkit αποτελεί το βασικό λογισμικό που απαιτείται για την δημιουργία και εκτέλεση προγραμμάτων, παρέχοντας τον απαραίτητο compiler, εργαλεία αποσφαλμάτωσης και διάφορες βιβλιοθήκες όπως αυτές που προαναφέραμε στο προηγούμενο κεφάλαιο. Θα περιγράψουμε τη διαδικασία λήψης και εγκατάστασης του λογισμικού, καθώς και την υποστήριξη που προσφέρει για διαφορετικές εκδόσεις λειτουργικών συστημάτων και αρχιτεκτονικών GPU. Παράλληλα, θα εξετάσουμε τις διαθέσιμες επιλογές για IDE, με έμφαση στη χρήση του Visual Studio, το οποίο προσφέρει ενσωματωμένη υποστήριξη για CUDA και επιτρέπει την ανάπτυξη και αποσφαλμάτωση κώδικα μέσα από ένα φιλικό περιβάλλον εργασίας.

Εκτός από την CUDA, η υλοποίηση των εφαρμογών θα απαιτήσει και τη χρήση επιπλέον βιβλιοθηκών, οι οποίες θα καλύψουν συγκεκριμένες ανάγκες. Θα αναφερθούμε στη GLFW και τη GLEW, που χρησιμοποιούνται για την δημιουργία παραθύρων και rendering με OpenGL, καθώς και στη libSndFile, η οποία επιτρέπει την ανάγνωση και εγγραφή αρχείων ήχου. Επιπλέον, θα εξετάσουμε τη OpenCV, που θα χρησιμοποιηθεί για την επεξεργασία εικόνας.

Θα γίνει αναλυτική περιγραφή της εγκατάστασης και ρύθμισης κάθε εργαλείου ξεχωριστά, καθώς και των απαιτούμενων ρυθμίσεων για τη σωστή σύνδεση των βιβλιοθηκών στο προγραμματιστικό περιβάλλον. Αυτή η προετοιμασία θα μας επιτρέψει,

στο επόμενο κεφάλαιο, να προχωρήσουμε στην ανάπτυξη των εφαρμογών CUDA, αξιοποιώντας πλήρως τις δυνατότητες του υλικού και των διαθέσιμων εργαλείων.

3.1 Cuda Toolkit

Το CUDA Toolkit αποτελεί το βασικό πακέτο εργαλείων που απαιτείται για την ανάπτυξη εφαρμογών που εκτελούνται σε GPU μέσω της αρχιτεκτονικής CUDA. Παρέχει όλα τα απαραίτητα στοιχεία για τη συγγραφή, μεταγλώττιση και εκτέλεση CUDA προγραμμάτων, συμπεριλαμβανομένου του NVIDIA CUDA Compiler (nvcc), βιβλιοθηκών όπως cuBLAS, cuSPARSE, cuFFT και εργαλείων για την ανάλυση και αποσφαλμάτωση εφαρμογών GPU.

3.1.1 Εγκατάσταση

Απαιτήσεις συστήματος

Το CUDA Toolkit υποστηρίζεται σε λειτουργικά συστήματα Windows και Linux, με τους αντίστοιχους συνδέσμους εγκατάστασης να είναι διαθέσιμοι στην επίσημη ιστοσελίδα της NVIDIA. Εκεί, ο χρήστης μπορεί να επιλέξει την κατάλληλη έκδοση, ανάλογα με το λειτουργικό του σύστημα, και να προχωρήσει στην εγκατάστασή της μέσω των παρεχόμενων πακέτων. Στα Windows, η εγκατάσταση γίνεται με αυτόματο εγκαταστάτη, ενώ στα Linux παρέχονται πακέτα .deb ή .rpm, ανάλογα με τη διανομή.

Πριν την εγκατάσταση, είναι σημαντικό να επιβεβαιωθεί ότι η κάρτα γραφικών υποστηρίζει την τεχνολογία CUDA. Αυτό μπορεί να γίνει στα Windows αν ανοίγουμε τη Διαχείριση Συσκευών και εντοπίζουμε την ενότητα Προσαρμογείς οθόνης. Εκεί εμφανίζεται το όνομα του κατασκευαστή και το μοντέλο της κάρτας γραφικών. Στα Linux μπορεί να γίνει ο ίδιος έλεγχος με συγκεκριμένη εντολή. Μπορούμε να ελέγξουμε αν υποστηρίζει CUDA στη λίστα διαθέσιμων μοντέλων που παρέχεται στη διεύθυνση:

<https://developer.nvidia.com/cuda-gpus>.

Μετά την ολοκλήρωση της εγκατάστασης, το σύστημα διαθέτει όλα τα απαραίτητα αρχεία και εργαλεία για την ανάπτυξη εφαρμογών με CUDA. Συγκεκριμένα, περιλαμβάνονται οι βιβλιοθήκες, οι μεταγλωττιστές, τα εργαλεία ανάλυσης απόδοσης και τα παραδείγματα κώδικα, που επιτρέπουν τη δημιουργία και βελτιστοποίηση προγραμμάτων που αξιοποιούν την επεξεργαστική ισχύ της GPU. Επιπλέον, εγκαθίστανται βασικές εντολές και βοηθητικά προγράμματα, όπως το nvcc (CUDA Compiler), που χρησιμοποιείται για τη μεταγλώττιση κώδικα CUDA C/C++, καθώς και

το CUDA Driver API, το οποίο εξασφαλίζει την επικοινωνία μεταξύ του λογισμικού και της GPU. Συνιστάται να επαληθευτεί η ορθή λειτουργία του CUDA Toolkit εκτελώντας ένα από τα παρεχόμενα παραδείγματα (deviceQuery, bandwidthTest), ώστε να επιβεβαιωθεί ότι η GPU αναγνωρίζεται και είναι έτοιμη προς χρήση.

3.1.2 Επιπλέον λειτουργίες και δυνατότητες

Το CUDA Toolkit προσφέρει μια πληθώρα βιβλιοθηκών που επιταχύνουν διάφορες εφαρμογές μέσω της χρήσης της GPU. Αυτές οι βιβλιοθήκες κατηγοριοποιούνται σε διάφορες ομάδες, καθεμία από τις οποίες στοχεύει σε συγκεκριμένους τομείς υπολογιστικών αναγκών. Στον ιστότοπο της NVIDIA, οι προγραμματιστές μπορούν να βρουν μια ευρεία γκάμα βιβλιοθηκών που επιταχύνουν εφαρμογές σε διάφορους τομείς, όπως μαθηματικούς υπολογισμούς, ανάλυση δεδομένων, επεξεργασία εικόνας και βίντεο, επικοινωνία μεταξύ πολλαπλών GPU, ακόμα και κβαντικούς υπολογισμούς. Αυτές οι βιβλιοθήκες είναι σχεδιασμένες ώστε να αξιοποιούν πλήρως τις δυνατότητες των GPU της NVIDIA, παρέχοντας εργαλεία για ανάπτυξη αποδοτικών, παράλληλων εφαρμογών. Περισσότερες πληροφορίες, οδηγίες χρήσης και τεκμηρίωση μπορούν να βρεθούν στην επίσημη σελίδα της NVIDIA, όπου διατίθενται επίσης παραδείγματα κώδικα και εκπαιδευτικοί πόροι για διευκόλυνση της ενσωμάτωσης των τεχνολογιών αυτών στις εφαρμογές.

3.1.3 Υποστήριξη Διαλειτουργικότητας με Γραφικά

Μέσα σε ολόκληρο αυτό το σύνολο των λειτουργιών θα βρούμε επίσης υποστήριξη για συνδυασμό CUDA και γραφικών που επιτρέπει τη διασύνδεση με γραφικές εφαρμογές μέσω της CUDA-OpenGL και CUDA-Direct3D Graphics Interoperability διεπαφής. Αυτή η δυνατότητα επιτρέπει την απευθείας επεξεργασία γραφικών δεδομένων στη GPU, μειώνοντας τις μεταφορές δεδομένων μεταξύ GPU και CPU, γεγονός που βελτιώνει την απόδοση εφαρμογών που συνδυάζουν υπολογισμούς και απεικόνιση.

Με τη λειτουργία Graphics Interoperability, ο προγραμματιστής μπορεί να αντιστοιχίσει δεδομένα από γραφικές διεπαφές, όπως η OpenGL, απευθείας στη μνήμη της GPU. Αυτό περιλαμβάνει στοιχεία όπως buffers και textures, τα οποία διαφορετικά θα απαιτούσαν συνεχή μεταφορά μεταξύ CPU και GPU, επιβαρύνοντας την απόδοση. Αντί να μεταφέρονται επανειλημμένα τα δεδομένα μέσω του συστήματος, η CUDA επιτρέπει την απευθείας πρόσβαση στη μνήμη της GPU, μειώνοντας δραστικά τον χρόνο επικοινωνίας.

Η υλοποίηση αυτής της δυνατότητας γίνεται μέσω ειδικών συναρτήσεων που επιτρέπουν την καταχώριση και διαχείριση αυτών των περιοχών μνήμης. Εκτός από υποστήριξη με την OpenGL, η NVIDIA παρέχει υποστήριξη και για άλλες διεπαφές γραφικών, όπως το DirectX. Αυτό επιτρέπει την ενσωμάτωση της CUDA σε μια ευρύτερη γκάμα εφαρμογών που απαιτούν υψηλή απόδοση, όπως μηχανές παιχνιδιών, επιστημονική απεικόνιση και απόδοση γραφικών σε πραγματικό χρόνο.

3.1.4 Microsoft Visual studio

Το Visual Studio είναι ιδιαίτερα χρήσιμο, καθώς παρέχει υποστήριξη για προγραμματισμό CUDA και GPU μέσω της ενσωμάτωσης του και είναι το ολοκληρωμένο περιβάλλον ανάπτυξης που θα χρησιμοποιήσουμε. Όταν χρησιμοποιούμε το Visual Studio, επωφελούμαστε από εξαιρετική υποστήριξη ανάλυσης κώδικα, αυξημένη διαχείριση των έργων, καθώς και ταχύτατη σύνδεση με βιβλιοθήκες και εργαλεία ανάπτυξης για CUDA.

Το ολοκληρωμένο IDE παρέχει μια πλήρη και εύχρηστη αναπτυξιακή εμπειρία για προγραμματισμό με CUDA. Η υποστήριξη για αρχεία CUDA εξασφαλίζει απρόσκοπτη αλληλεπίδραση με τις απαιτήσεις του CUDA, ενώ η ενσωμάτωση λειτουργιών επέκτασης όπως το IntelliSense σημαίνει πιο γρήγορη και εύκολη ανάπτυξη κώδικα με αυτόματες προτάσεις.

3.2 Επιπλέον Βοηθητικές Βιβλιοθήκες

3.2.1 OpenCV

Η OpenCV (Open Source Computer Vision Library) είναι μια δημοφιλής βιβλιοθήκη ανοιχτού κώδικα που παρέχει εργαλεία για επεξεργασία εικόνας και υπολογιστική όραση. Υποστηρίζει μια πληθώρα λειτουργιών, όπως ανίχνευση αντικειμένων, επεξεργασία βίντεο, ανάλυση εικόνας και εφαρμογή φίλτρων.

Χρήση στα παραδείγματα

Στις εφαρμογές της πτυχιακής, η OpenCV χρησιμοποιείται για την προεπεξεργασία εικόνων και τη διευκόλυνση της διαχείρισης δεδομένων. Για παράδειγμα, στην εφαρμογή επεξεργασίας εικόνας, αξιοποιείται για τη φόρτωση, και την εφαρμογή φίλτρων.

3.2.2 OpenGL

Για τη χρήση της OpenGL, έχουμε επιλέξει να αξιοποιήσουμε δύο βασικές βιβλιοθήκες: τη GLFW και τη GLEW. Η GLFW αναλαμβάνει τη δημιουργία και διαχείριση παραθύρων, την επεξεργασία εισόδου από πληκτρολόγιο και ποντίκι, καθώς και την υλοποίηση του περιβάλλοντος (context) για την OpenGL, εξασφαλίζοντας ότι το γραφικό περιβάλλον είναι σωστά ρυθμισμένο. Από την άλλη πλευρά, η GLEW λειτουργεί ως μηχανισμός φόρτωσης των επεκτάσεων της OpenGL, επιτρέποντας την πρόσβαση σε σύγχρονες λειτουργίες της βιβλιοθήκης χωρίς να απαιτείται χειροκίνητη διαχείριση των δεικτών στις συναρτήσεις. Με αυτή την ολοκληρωμένη διαμόρφωση, οι δύο βιβλιοθήκες συνεργάζονται για να προσφέρουν ένα σταθερό και πλήρες περιβάλλον ανάπτυξης γραφικών εφαρμογών.

Στην εφαρμογή που θα υλοποιήσουμε, θα εστιάσουμε στις βασικές λειτουργίες της OpenGL, χωρίς να εμβαθύνουμε πολύ στο σύνολο του pipeline. Θα χρησιμοποιήσουμε κυρίως τα buffers για την αποθήκευση δεδομένων, όπως τα vertex και index buffers, καθώς και απλά shader programs για την απόδοση των τρισδιάστατων γραφικών. Με αυτόν τον τρόπο, επιτυγχάνουμε μια απλή και αποτελεσματική υλοποίηση, μειώνοντας τις περιττές μεταφορές δεδομένων μεταξύ CPU και GPU και εξασφαλίζοντας υψηλή απόδοση στην απεικόνιση. Αυτή η προσέγγιση επιτρέπει τη δημιουργία εφαρμογών που εκμεταλλεύονται πλήρως τις δυνατότητες της GPU, ενώ παραμένουν εύκολες στην ανάπτυξη και συντήρηση.

3.2.3 Libsndfile

Η libsndfile είναι μια βιβλιοθήκη ανοιχτού κώδικα που χρησιμοποιείται για την ανάγνωση και εγγραφή αρχείων ήχου σε μορφές όπως WAV, AIFF και FLAC. Διευκολύνει την επεξεργασία δεδομένων ήχου, ανεξαρτήτως μορφής, και είναι ιδανική για εφαρμογές που χρειάζονται απλότητα και ακρίβεια στη διαχείριση αρχείων ήχου.

Η libsndfile είναι διαθέσιμη στον επίσημο ιστότοπό της :

<http://www.mega-nerd.com/libsndfile/>

Εκεί θα βρείτε πλήρεις πληροφορίες σχετικά με την εγκατάσταση, το documentation, παραδείγματα χρήσης και οδηγίες για την ενσωμάτωσή της σε εφαρμογές. Επιπλέον, η βιβλιοθήκη διατίθεται σε διάφορες πλατφόρμες και μπορεί να εγκατασταθεί μέσω των αντίστοιχων package managers, διευκολύνοντας έτσι την ενσωμάτωσή της στα έργα σας.

Για να ενσωματώσετε τη `libsndfile` στον κώδικά σας, αρκεί να συμπεριλάβετε το αρχείο κεφαλίδας (header file) `"sndfile.h"` στο πρόγραμμα σας.

4 Υλοποίηση Εφαρμογών

Σε αυτό το μέρος της πτυχιακής, που αποτελεί το μεγαλύτερο και πιο ουσιώδες τμήμα της, στοχεύω στην ανάπτυξη εφαρμογών που αξιοποιούν τις λειτουργίες των βιβλιοθηκών CUDA, παρέχοντας άμεσα αποτελέσματα που μπορούν να παρατηρηθούν μετά την ολοκλήρωσή τους. Οι εφαρμογές αυτές θα επικεντρωθούν στη χρήση προηγμένων λειτουργιών των βιβλιοθηκών, επιλύοντας προβλήματα που απαιτούν γρήγορες και αποδοτικές υπολογιστικές λύσεις. Με αυτόν τον τρόπο, θα αναδειχθεί η πρακτική τους χρησιμότητα σε επιστημονικούς και τεχνολογικούς τομείς.

Οι εφαρμογές που θα αναπτύξω χρησιμοποιώντας το CUDA Toolkit θα αναδείξουν πόσο εύκολα μπορεί να ενσωματωθεί η GPU επιτάχυνση σε ένα ήδη υπάρχον πρόγραμμα. Μέσω μικρών τροποποιήσεων στον κώδικα, οι υπολογιστικά απαιτητικές διεργασίες μπορούν να μεταφερθούν στην GPU, προσφέροντας σημαντική βελτίωση στην απόδοση. Παράλληλα, θα δοθεί έμφαση στη σημασία της σωστής διαχείρισης μνήμης, καθώς η αποτελεσματική χρήση των πόρων της GPU είναι κρίσιμη για την επίτευξη υψηλής απόδοσης και τη μείωση της καθυστέρησης κατά την εκτέλεση.

4.1 Εφαρμογή Επεξεργασίας ήχου

Στην παρούσα εργασία αναπτύχθηκε εφαρμογή ψηφιακής επεξεργασίας ήχου που αξιοποιεί τη βιβλιοθήκη cuFFT για την υλοποίηση του μετασχηματισμού Fourier, επιτρέποντας την ανάλυση και τροποποίηση του ήχου στο πεδίο της συχνότητας. Ο κύριος στόχος της εφαρμογής είναι όχι μόνο η ανάδειξη της χρησιμότητας της cuFFT, αλλά κυρίως η επίδειξη της υψηλής απόδοσης που προσφέρει σε σχέση με την υλοποίηση της βιβλιοθήκης FFTW, η οποία εκτελείται στην CPU.

4.1.1 Μετασχηματισμός Fourier και ψηφιακή επεξεργασία ήχου:

Ο μετασχηματισμός Fourier αποτελεί μια θεμελιώδη μαθηματική διαδικασία για την ανάλυση ενός σήματος στο πεδίο της συχνότητας, μετασχηματίζοντάς το σε μια σειρά συχνοτικών συνιστωσών. Συγκεκριμένα, ο Discrete Fourier Transform (DFT) και ο Fast Fourier Transform (FFT) είναι από τους πιο σημαντικούς και ευρέως

χρησιμοποιούμενους αριθμητικούς αλγόριθμους στην υπολογιστική φυσική και τη γενική επεξεργασία σήματος. Στην ψηφιακή επεξεργασία ήχου, μέσω του Fourier μετασχηματισμού μπορούμε να απομονώσουμε ή να ενισχύσουμε συγκεκριμένες συχνότητες, να εφαρμόσουμε φίλτρα (π.χ. φίλτρα χαμηλής ή υψηλής διέλευσης), να εξαλείψουμε τον θόρυβο ή ακόμη να δημιουργήσουμε νέους ήχους με βάση συνδυασμούς συχνοτήτων.

Η cuFFT επιτρέπει την αποδοτική εκτέλεση FFT σε μεγάλα δεδομένα, αξιοποιώντας την παράλληλη επεξεργασία που παρέχει η GPU. Αυτό εξαλείφει την ανάγκη για βαριούς υπολογισμούς στον επεξεργαστή και μειώνει σημαντικά το χρόνο επεξεργασίας, καθιστώντας την ιδανική για επιτάχυνση εφαρμογές ήχου όπου απαιτείται άμεση ανάλυση και επεξεργασία μεγάλων δεδομένων.

STFT, Windowing και Overlap

Για την ανάλυση σημάτων που εξελίσσονται με τον χρόνο, όπως ο ήχος, χρησιμοποιείται η τεχνική της Short-Time Fourier Transform (STFT). Η STFT χωρίζει το σήμα σε διαδοχικά μικρά χρονικά παράθυρα (blocks), στα οποία εφαρμόζεται FFT. Αυτή η διαδικασία επιτρέπει την παρακολούθηση των συχνοτήτων σε κάθε χρονική στιγμή.

Ένα κρίσιμο βήμα σε αυτήν την ανάλυση είναι η εφαρμογή ενός παραθύρου (window function), όπως το Hann window, το οποίο βοηθά στη μείωση των παρεμβολών (artifacts) και βελτιώνει την ακρίβεια της ανάλυσης. Η εφαρμογή του Hann window γίνεται σύμφωνα με την εξίσωση:

$$w[n] = 0.5(1 - \cos(\frac{2\pi n}{N-1})) \quad \text{Εξίσωση 1}$$

όπου n είναι το δείγμα του σήματος και N το μέγεθος του παραθύρου. Επιπλέον, η τεχνική Overlap ή αλλιώς επικάλυψη παραθύρων, αποτελεί ένα κρίσιμο στοιχείο στην ανάλυση σήματος μέσω STFT, καθώς εξασφαλίζει την πλήρη κάλυψη των δεδομένων χωρίς απώλεια πληροφορίας στα σύνορα των παραθύρων. Συγκεκριμένα, με την εφαρμογή επικάλυψης κάθε νέο παράθυρο μοιράζεται ένα ποσοστό του περιεχομένου του με το προηγούμενο, διασφαλίζοντας έτσι ότι οι μεταβάσεις μεταξύ των παραθύρων είναι ομαλές και δεν υπάρχουν κενά που να προκαλούν artifacts. Αυτή η στρατηγική συμβάλλει στην καλύτερη διατήρηση των χαρακτηριστικών του αρχικού σήματος κατά την ανάλυση και ανακατασκευή μέσω του IFFT, μέσω της μεθόδου Overlap-Add, επιτυγχάνοντας ένα ορθό τελικό αποτέλεσμα.

4.1.2 Απαιτήσεις και Ρυθμίσεις Εφαρμογής

Η διαδικασία για την εφαρμογή που έφτιαξα περιλαμβάνει τη φόρτωση ενός δείγματος ήχου στη μνήμη, τη μεταφορά του στη GPU μέσω CUDA, και τη χρήση της cuFFT για την εκτέλεση του FFT. Στη συνέχεια, εφαρμόζονται τα φίλτρα μέσω προσαρμοσμένων συναρτήσεων επεξεργασίας. Το αποτέλεσμα επιστρέφεται στη μορφή αντίστροφου FFT (Inverse FFT) και μεταφέρεται πίσω στη CPU για αναπαραγωγή ή αποθήκευση.

Παρακάτω περιγράφονται οι βασικές απαιτήσεις και οι απαραίτητες ρυθμίσεις που πρέπει να ολοκληρωθούν.

CUFFT

Η cuFFT χρησιμοποιείται για την εκτέλεση FFT σε δεδομένα που βρίσκονται στην GPU, επιτρέποντας την ταχεία μετατροπή δεδομένων από το πεδίο του χρόνου στο πεδίο της συχνότητας. Η ενσωμάτωσή της στο project απαιτεί μια απλή ρύθμιση του Linker, όπου προστίθεται το αρχείο cufft.lib στην επιλογή input. Στον πηγαίο κώδικα, αρκεί η προσθήκη του `#include <cufft.h>`, καθιστώντας τη βιβλιοθήκη άμεσα διαθέσιμη για χρήση.

Βοηθητικές συναρτήσεις

Για την υλοποίηση του προγράμματος, χρησιμοποιούνται οι βιβλιοθήκες FFTW (fftwf) για την εκτέλεση FFT (floating point) στο CPU και libsndfile για την ανάγνωση/εγγραφή αρχείων ήχου WAV. Η σύνδεσή τους απαιτεί τη σωστή προσθήκη των αντίστοιχων header files (fftw3.h, sndfile.h) και τη ρύθμιση των paths στις επιλογές Include Directories και Library Directories του Visual Studio. Στις Ρυθμίσεις του Linker στην επιλογή input προσθέτουμε τα αρχεία sndfile.lib και libfftw3f-3.lib για την FFTW (single-precision). Επιπλέον, η χρήση της C++ Standard Library (C++17) είναι απαραίτητη για δυνατότητες όπως `std::filesystem` και γι'αυτό θα προσθεσουμε σαν flag στον compiler την `-std=c++17`.

Επίσης θα προσθέσουμε μερικά ακόμη compiler flags όπως `--extended-lambda` και το flag `(-D_USE_MATH_DEFINES)` για να μπορούμε να χρησιμοποιήσουμε σωστά την σταθερά `π` σε κομμάτι κώδικα που εκτελείται στην GPU.

4.1.3 Υλοποίηση:

Ας ξεκινήσουμε πρώτα από την συνάρτηση `main`, όπου αρχικά γίνεται σάρωση του φακέλου "Samples" για αρχεία WAV χρησιμοποιώντας την `std::filesystem`. Μόλις εντοπιστούν διαθέσιμα αρχεία, αυτά εμφανίζονται στην κονσόλα και ο χρήστης καλείται

να επιλέξει το επιθυμητό αρχείο για επεξεργασία. Στη συνέχεια, ζητείται από τον χρήστη να εισαγάγει τις τιμές για το χαμηλό και το υψηλό cutoff frequency, οι οποίες καθορίζουν το εύρος συχνοτήτων που θα διατηρηθεί στο τελικό αποτέλεσμα. Επιπλέον, καταγράφεται ο τρέχων χρόνος, ο οποίος χρησιμοποιείται για τη δημιουργία μοναδικών ονομάτων στα παραγόμενα αρχεία εξόδου στο τέλος.

Αφότου επιλεγούν το αρχείο και οι τιμές παράμετροι για το φίλτρο, το πρόγραμμα προχωρά στην επεξεργασία του ήχου μέσω δύο παραλλαγών: μία υλοποίηση που αξιοποιεί την υπολογιστική ισχύ της GPU και μία υλοποίηση που τρέχει στην CPU. Και οι δυο κάνουν ακριβώς την ίδια επεξεργασία στο κομμάτι ήχου ωστόσο για να μπορέσει να πραγματοποιηθεί επεξεργασία μέσω της cuFFT απαιτεί τον κατακερματισμό του σήματος σε μικρά τμήματα (chunks) προκειμένου να καταστεί εφικτή η μετατροπή του στο συχνοτικό πεδίο. Ο λόγος είναι ότι η δημιουργία του πλάνου εκτέλεσης της FFT (cuFFT plan) καταναλώνει σημαντική ποσότητα μνήμης, η οποία μπορεί να ξεπεράσει τη διαθέσιμη στη GPU εάν επιχειρηθεί η επεξεργασία ενός πολύ μεγάλου σήματος ως ενιαίο σύνολο.

Τα δεδομένα ήχου φορτώνονται μέσω της βιβλιοθήκης libsndfile, η οποία επιτρέπει την ανάγνωση και εγγραφή αρχείων ήχου διαφόρων μορφών. Στη συνέχεια, μεταφέρονται στη συνάρτηση επεξεργασίας applySTFTOverlapAdd()

Αυτό το τμήμα κώδικα υλοποιεί την επεξεργασία ενός ήχου με τεχνική STFT (Short-Time Fourier Transform) σε συνδυασμό με Overlap-Add, χρησιμοποιώντας την cuFFT και CUDA kernels. Συγκεκριμένα, το σήμα διαιρείται σε επικάλυπτα chunks με καθορισμένο μέγεθος και hop size, και δημιουργείται ένα Hann window μέσω του kernel generateHannWindow. Κάθε chunk πολλαπλασιάζεται με το Hann window (μέσω του kernel applyWindow), στη συνέχεια μετατρέπεται στο πεδίο της συχνότητας με forward FFT, όπου εφαρμόζεται ένα band-pass φίλτρο (μέσω του kernel applyBandPassFilter) που μηδενίζει τις συχνότητες εκτός των καθορισμένων ορίων. Μετά, ο αντίστροφος FFT επιστρέφει τα δειγματα του chunk στο πεδίο του χρόνου, όπου μέσω του kernel overlapAddChunk πραγματοποιείται κανονικοποίηση και τα δειγματα προστίθενται (overlap-add) στο τελικό αποτέλεσμα.

Τελικά, τα επεξεργασμένα δεδομένα αντιγράφονται πίσω στον host και αποθηκεύονται ως νέο αρχείο WAV με όνομα που περιέχει τις παραμέτρους του φίλτρου και το timestamp.

Παράλληλα, η υλοποίηση στην CPU ακολουθεί μια παρόμοια λογική με συναρτήσεις της FFTW. Η ανάγνωση και εγγραφή του αρχείου ήχου γίνεται επίσης με τη χρήση της `libsndfile`, ενώ η εκτέλεση της FFT επιτυγχάνεται με χρήση της συνάρτησης `fftwf_plan_dft_r2c_1d` για την εμπρός μετατροπή και `fftwf_plan_dft_c2r_1d` για την αντίστροφη). Το φίλτρο εφαρμόζεται με έναν απλό βρόχο που μηδενίζει τις τιμές των συχνοτήτων εκτός του επιλεγμένου εύρους, και μετά από την αντίστροφη FFT, τα δεδομένα κανονικοποιούνται ώστε να διασφαλιστεί η ορθή κλίμακα τιμών. Το τελικό αποτέλεσμα αποθηκεύεται επίσης ως αρχείο WAV, δίνοντας τη δυνατότητα σύγκρισης της απόδοσης μεταξύ των δύο υλοποιήσεων.

4.1.4 Αποτελέσματα Εφαρμογής

Μέσω της χρήσης και των δύο μεθοδολογιών, το πρόγραμμα επιτρέπει μια άμεση σύγκριση της απόδοσης και της αποτελεσματικότητας μεταξύ της GPU και της CPU, καταγράφοντας ταυτόχρονα το χρόνο εκτέλεσης κάθε προσέγγισης.

Για την αξιολόγηση της επιτάχυνσης που επιτυγχάνεται με την GPU, θα πραγματοποιηθεί σύγκριση στους χρόνους εκτέλεσης για τρία αρχεία ήχου μορφής WAV διαφορετικού μεγέθους, με μικρό, μεσαίο και μεγάλο πλήθος δειγμάτων.

Πίνακας 2: Πίνακας χρόνων εκτέλεσης.

Πλήθος δειγμάτων	Χρόνος CPU	Χρόνος GPU	Αύξηση απόδοσης
1.079.294	0.043563 sec	0.008407 sec	×5,3
42.340.352	1.118175 sec	0.053993 sec	×20,7
112.726.492	5.2520425 sec	0.144378 sec	×36,3

Τα αποτελέσματα δείχνουν σαφώς ότι η χρήση της GPU για την επεξεργασία FFT προσφέρει σημαντική αύξηση απόδοσης σε σύγκριση με την CPU. Ακόμα και για ένα μικρό αρχείο, ο χρόνος εκτέλεσης μειώνεται από 0,0436 sec σε 0,00841 sec, δηλαδή περίπου 5,3 φορές. Καθώς αυξάνεται το μέγεθος του αρχείου, η διαφορά γίνεται ακόμη πιο εντυπωσιακή: για 42,34 εκατομμύρια δείγματα ο χρόνος επεξεργασίας πέφτει από 1,118 sec σε 0,054 sec (αύξηση περίπου 20,7 φορές), και για 112,73 εκατομμύρια δείγματα από 5,252 sec σε 0,144 sec (αύξηση περίπου 36,3 φορές). Αυτή η τάση υποδεικνύει ότι η GPU κλιμακώνεται καλύτερα με το μέγεθος των δεδομένων, καθώς οι

επιπλέον δαπάνες χρόνου που σχετίζονται με τη μεταφορά δεδομένων και την εκκίνηση kernels έχουν μικρότερη επίδραση σε μεγάλα σύνολα δεδομένων.

4.2 Επιτάχυνση 3D γραφικών με CUBLAS

Αφού εξετάσαμε την επιτάχυνση της επεξεργασίας ήχου με την cuFFT, είναι εξίσου ενδιαφέρον να δούμε πώς η επιτάχυνση μπορεί να εφαρμοστεί και σε έναν εντελώς διαφορετικό τομέα: τα 3D γραφικά.

Ο κώδικας που έχω αναπτύξει υλοποιεί μια εφαρμογή 3D γραφικών με χρήση OpenGL για την απόδοση και cuBLAS για την επιτάχυνση μαθηματικών υπολογισμών, που είναι οι μετασχηματισμοί και η περιστροφή των αντικειμένων με τη χρήση GEMM (General Matrix-matrix multiplication) σε μια 3D σκηνή. Συγκεκριμένα δημιουργεί ένα πλέγμα από ένα μεγάλο πλήθος τριγώνων με τις θέσεις του να ανανεώνονται σε κάθε καρέ. Η συνεργασία με το CUDA Graphics interop επιτρέπει την σύνδεση μεταξύ του API της OpenGL και CUDA, επιτρέποντας την άμεση μεταφορά δεδομένων και τον υπολογισμό των μετασχηματισμών χωρίς την ανάγκη για συνεχείς μεταφορές μεταξύ CPU και GPU.

Αυτός ο συνδυασμός επιτρέπει την αποδοτικότερη διαχείριση της σκηνής και τη μείωση του φόρτου υπολογισμών στον κεντρικό επεξεργαστή, προσφέροντας υψηλή απόδοση σε πραγματικό χρόνο.

4.2.1 Εισαγωγή στην OpenGL και τη Χρήση της στο Έργο

Η OpenGL (Open Graphics Library) είναι ένα από τα πιο διαδεδομένα API για την ανάπτυξη γραφικών εφαρμογών, προσφέροντας εργαλεία για την απόδοση τρισδιάστατων σκηνών σε πραγματικό χρόνο. Είναι ένα cross-platform API που επιτρέπει την απευθείας επικοινωνία με το hardware της κάρτας γραφικών, δίνοντας τη δυνατότητα δημιουργίας πολύπλοκων γραφικών μέσω shaders και προγραμματιζόμενης pipeline.

Στην πτυχιακή μου εργασία, η OpenGL χρησιμοποιείται για την απόδοση της τρισδιάστατης σκηνής. Ο σχεδιασμός της σκηνής γίνεται με Vertex Array Objects (VAOs) και Vertex Buffer Objects (VBOs), χρησιμοποιούνται Index Buffer Objects (IBOs) για την καλύτερη διαχείριση των γεωμετρικών δεδομένων, μειώνοντας τον αριθμό των αποθηκευμένων και επεξεργασμένων κορυφών μέσω της χρήσης δεικτών,

ενώ τα shaders χειρίζονται τον φωτισμό, τις υφές και τα γεωμετρικά μετασχηματιστικά εφέ.

Για να εκμεταλλευτούμε την OpenGL συνδυάζουμε τις βιβλιοθήκες GLFW και GLEW. Η GLFW χρησιμοποιείται για τη διαχείριση των παραθύρων και των εισόδων του χρήστη (π.χ. κίνηση της κάμερας), ενώ η GLEW επιτρέπει τη φόρτωση των σύγχρονων OpenGL extensions, διασφαλίζοντας τη συμβατότητα με πιο προηγμένα χαρακτηριστικά του API.

Ένα κρίσιμο στοιχείο της υλοποίησης είναι η διαλειτουργικότητα CUDA-OpenGL, που επιτρέπει τη διαχείριση των buffers απευθείας στη GPU, εξαλείφοντας τις περιττές μεταφορές δεδομένων μεταξύ CPU και GPU. Με την αξιοποίηση του CUDA Graphics Resource API, οι τροποποιήσεις στα buffers πραγματοποιούνται σε πραγματικό χρόνο, προσφέροντας βελτιωμένη απόδοση στην απεικόνιση της σκηνής.

4.2.2 Απαιτήσεις και Ρυθμίσεις Περιβάλλοντος

Για τη σωστή ρύθμιση και ανάπτυξη του κώδικα αυτής της εφαρμογής, είναι απαραίτητο να ληφθούν υπόψη ορισμένα προαπαιτούμενα. Αυτά περιλαμβάνουν το υλικό (hardware) που υποστηρίζει CUDA, τις απαραίτητες βιβλιοθήκες και εργαλεία ανάπτυξης, καθώς και τις βασικές ρυθμίσεις του περιβάλλοντος. Παρακάτω αναλύονται οι βιβλιοθήκες και οι ρυθμίσεις που απαιτούνται για την εκτέλεση της εφαρμογής.

OpenGL

Η απόδοση των 3D γραφικών πραγματοποιείται μέσω OpenGL, το οποίο επιτρέπει τη σχεδίαση γεωμετρικών σχημάτων στην οθόνη και τη διαχείριση του rendering pipeline της GPU. Στην παρούσα εφαρμογή, η υλοποίηση του OpenGL γίνεται με τη χρήση των βιβλιοθηκών GLFW και GLEW

GLFW και GLEW

Το GLFW χρησιμοποιείται για τη δημιουργία και διαχείριση παραθύρων, καθώς και για τον χειρισμό εισόδων από πληκτρολόγιο και ποντίκι και το GLEW επιτρέπει τη φόρτωση επεκτάσεων OpenGL, διασφαλίζοντας ότι οι σύγχρονες δυνατότητες των GPUs είναι διαθέσιμες.

GLM (OpenGL Mathematics)

Το GLM είναι μια βιβλιοθήκη μαθηματικών συναρτήσεων ειδικά σχεδιασμένη για OpenGL, παρέχοντας υλοποιήσεις για διανύσματα, πίνακες μετασχηματισμών, και άλλες συναρτήσεις που είναι απαραίτητες σε 3D γραφικά και γραμμική άλγεβρα.

Χρησιμοποιείται στην παρούσα εφαρμογή για τη διαχείριση πινάκων περιστροφής, μεταφορών και κλιμάκωσης των 3D αντικειμένων.

Ρύθμιση του Visual Studio για CUDA και OpenGL

Για τη σωστή μεταγλώττιση του προγράμματος, πρέπει να γίνει η ρύθμιση του Linker για τη σύνδεση των απαραίτητων βιβλιοθηκών και ο καθορισμός των διαδρομών των include και lib αρχείων για OpenGL, GLFW, GLEW και GLM. Μερικές από τις ρυθμίσεις που προστέθηκαν είναι :

- Defines: GLEW_STATIC;_DEBUG; (για Debug) και GLEW_STATIC; (για Release).
- cublas.lib → Χρησιμοποιείται η βιβλιοθήκη cuBLAS.
- glfw3.lib → Σύνδεση με τη βιβλιοθήκη GLFW.
- glew32s.lib → Χρήση της στατικής έκδοσης του GLEW (λόγω του ορισμού GLEW_STATIC).

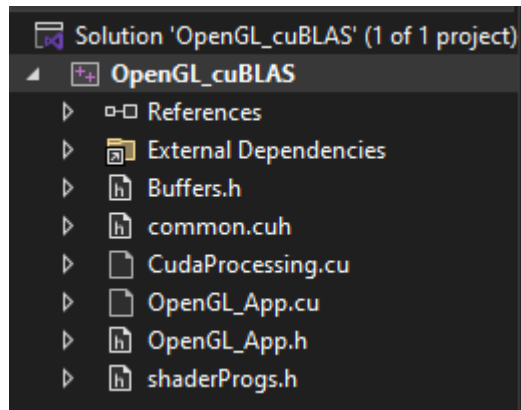
Οι ρυθμίσεις του linker περιλαμβάνουν τις βιβλιοθήκες της CUDA (όπως η cudart_static.lib και η cublas.lib), καθώς και τις βιβλιοθήκες του OpenGL, εξασφαλίζοντας ότι η εφαρμογή μπορεί να αξιοποιήσει τόσο τις GPU-επιταχυνόμενες λειτουργίες όσο και την απόδοση 3D γραφικών.

Το έργο έχει ρυθμιστεί ώστε να αναζητά τις απαιτούμενες κεφαλίδες και βιβλιοθήκες από τις κατάλληλες τοποθεσίες, οι οποίες έχουν προστεθεί στις διαδρομές συμπερίληψης και βιβλιοθηκών του Visual Studio.

4.2.3 Υλοποίηση της εφαρμογής

Η εφαρμογή ακολουθεί μια δομημένη αρχιτεκτονική, χωρίζοντας τις βασικές λειτουργίες σε διαφορετικά αρχεία, ώστε να είναι πιο οργανωμένη και επεκτάσιμη. Ο κώδικας είναι δομημένος σε διακριτά μέρη που αναλαμβάνουν:

- Την αρχικοποίηση των γραφικών (OpenGL context, window creation, render setup).
- Την προετοιμασία της σκηνής (φόρτωση αντικειμένων, shaders).
- Τη διαχείριση των δεδομένων για την GPU (buffers, CUDA).
- Την κύρια διαδικασία rendering και τον βρόχο εκτέλεσης (rendering loop)



Εικόνα 21: Τα αρχεία του project

Ο κώδικας αποτελείται από πολλαπλά αρχεία, όπου κάθε ένα διαχειρίζεται διαφορετικές πτυχές της λειτουργικότητας, όπως το rendering, την επεξεργασία δεδομένων και τη διαχείριση shaders και buffers.

Πίνακας 3: Δομή των αρχείων της εφαρμογής.

Αρχείο	Ρόλος
Buffers.h	Περιέχει έτοιμες δομές για την εύκολη προσθήκη και διαχείριση των buffers των VAO/VBO/IBO για τα γραφικά.
Common.cuh	Μερικές κοινόχρηστες δηλώσεις για τα 2 .cu αρχεία
CudaProcessing.cu	Βασική λειτουργικότητα για την επεξεργασία με CUDA. Εδώ έχουμε και τις συναρτήσεις CUBLAS.
OpenGL_App.cu	Κυρίο αρχείο του κώδικα, εδώ βρίσκεται και η συνάρτηση main και αποτελεί το μεγαλύτερο κομμάτι της ροής του κώδικα.
OpenGL_App.h	Βοηθητικές συναρτήσεις, όπως διαχείριση εισόδου, κάμερας και δημιουργία πλέγματος
shaderProgs.h	Κώδικας των shader προγραμμάτων για την εμφάνιση των αντικειμένων στη σκηνή.

Το πρόγραμμα ξεκινάει την εκτέλεση από τη συνάρτηση main(), η οποία έχει ως κύριο στόχο τη δημιουργία του παραθύρου, την αρχικοποίηση των βιβλιοθηκών (GLFW, GLEW, CUDA) και την έναρξη του κύριου βρόχου rendering.

Αρχή του προγράμματος

Στη συνάρτηση `main()`, αρχικά καλείται η `glfwInit()` για την αρχικοποίηση της βιβλιοθήκης GLFW, η οποία χρησιμοποιείται για τη δημιουργία παραθύρων και τη διαχείριση εισόδων από το πληκτρολόγιο και το ποντίκι. Αν η αρχικοποίηση αποτύχει, το πρόγραμμα τερματίζεται. Στη συνέχεια, δημιουργείται ένα παράθυρο με προκαθορισμένες διαστάσεις και τίτλο μέσω της `glfwCreateWindow()`, και σε περίπτωση αποτυχίας, το πρόγραμμα τερματίζεται. Το παράθυρο ορίζεται ως το τρέχον context για την απόδοση, ενώ παράλληλα ορίζεται ένα callback (`framebuffer_size_callback()`) για την προσαρμογή του viewport σε αλλαγές μεγέθους του παραθύρου. Ακολουθεί η αρχικοποίηση της GLEW μέσω της `glewInit()`, επιτρέποντας τη χρήση σύγχρονων επεκτάσεων του OpenGL. Επιπλέον, ορίζεται ένα callback (`mouse_callback()`) για την παρακολούθηση και επεξεργασία της κίνησης του ποντικιού.

Προετοιμασία της εφαρμογής

Μετά την αρχικοποίηση του παραθύρου και των βασικών βιβλιοθηκών, καλείται η συνάρτηση `app_init()`, η οποία προετοιμάζει τα δεδομένα της εφαρμογής. Αρχικά, υπολογίζονται οι συνολικοί αριθμοί κορυφών (vertices) και indices με βάση το προκαθορισμένο μέγεθος του terrain. Στη συνέχεια, δεσμεύεται μνήμη στη CPU για τα δεδομένα των vertices και των indices, και δημιουργείται ένα high-triangle count plane μέσω της συνάρτησης `createHighTriCountPlane()`. Ακολουθεί η δέσμευση μνήμης στη GPU τόσο για τα vertices όσο και για έναν πίνακα περιστροφής, ενώ τα αρχικά δεδομένα μεταφέρονται από τη CPU στη GPU. Αν η εφαρμογή χρησιμοποιεί CUDA streams, δημιουργείται ένα stream για την εκτέλεση των υπολογισμών. Τέλος, πραγματοποιείται η αρχικοποίηση της βιβλιοθήκης cuBLAS, η οποία θα αξιοποιηθεί για επιταχυνόμενες γραμμικές πράξεις στη GPU.

Αφού ολοκληρωθεί η εκτέλεση της `app_init()` καλείται η συνάρτηση `drawFrame` όπου περιέχει τον κώδικα για την δημιουργία και ενημέρωση των καρέ. Δημιουργούνται τα απαραίτητα OpenGL objects, όπως τα Vertex Array Objects (VAO), Vertex Buffer Objects (VBO) και Index Buffer Objects (IBO). Αν η CUDA interoperability είναι ενεργή, το vertex buffer καταχωρείται στη CUDA ώστε να επιτρέπεται η ενημέρωσή του απευθείας από τη GPU. Στη συνέχεια, ρυθμίζεται το shader πρόγραμμα που θα χρησιμοποιηθεί για την απόδοση του terrain.

Δημιουργία καρέ στον βρόχο απόδοσης (render loop):

Η συνάρτηση `drawFrame(GLFWwindow* window)` είναι υπεύθυνη για το rendering της σκηνής και την επικοινωνία μεταξύ OpenGL και CUDA. Πριν μπούμε στον κύριο βρόχο ενεργοποιείται η λειτουργία βάθους (depth test) και ορίζεται το χρώμα του φόντου. Ο

κύριος βρόχος απόδοσης ξεκινά και εκτελείται συνεχώς μέχρι το κλείσιμο του παραθύρου. Κατά τη διάρκεια του βρόχου, υπολογίζεται ο χρόνος μεταξύ των frames για την εξασφάλιση ομαλής κίνησης και animation. Αν η λειτουργία animation (runAnimation) είναι ενεργή, ενημερώνονται τα δεδομένα του mesh. Αν η επεξεργασία γίνεται στη GPU μέσω CUDA, καλείται η UpdateWaveSimulation(), ενώ αν γίνεται στην CPU, οι κορυφές ενημερώνονται με τις ApplyWaveEffectCPU() και ApplyRotationEffectCPU().

Στο αρχείο OpenGL_App.cu αποτελεί την καρδιά της εφαρμογής, καθώς περιλαμβάνει τον βασικό έλεγχο της ροής του προγράμματος, την αρχικοποίηση των υποσυστημάτων (CUDA, OpenGL), τη δημιουργία και διαχείριση της μνήμης, καθώς και το βασικό rendering loop.

Σε συνδυασμό με το OpenGL_App.cu, το αρχείο CudaProcessing.cu παρέχει την υλοποίηση των GPU-επιταχυνόμενων συναρτήσεων που καλούνται μέσα στη ροή του rendering loop για την εφαρμογή δυναμικών εφέ στα vertices μέσω CUDA kernels και τη βιβλιοθήκη cuBLAS. Σε αυτό το αρχείο, ορίζεται η συνάρτηση sineWaveKernel, η οποία αναλαμβάνει την κίνηση του πλέγματος τροποποιώντας την τιμή του ύψους (y) κάθε σημείου με βάσει μιας ημιτονοειδούς συνάρτησης που εξαρτάται από το χρόνο, καθώς και τις x και z συντεταγμένες για να δημιουργήσουμε το εφέ του ρευστού και κυματιστού πλέγματος. Η συνάρτηση ApplyWaveEffect() ορίζει τους παραμέτρους και πραγματοποιεί την εκτέλεση του kernel. Η λειτουργία του αρχείου επεκτείνεται με την υλοποίηση της συνάρτησης ApplyRotationEffect η οποία δημιουργεί έναν πίνακα περιστροφής, ο οποίος αντιγράφεται στη μνήμη της GPU, και στη συνέχεια χρησιμοποιεί τη συνάρτηση cublasSgemm της βιβλιοθήκης cuBLAS για να πραγματοποιήσει matrix multiplication. Με αυτόν τον τρόπο, πραγματοποιείται ο περιστροφικός μετασχηματισμός των κορυφών της σκηνής.

Σε περίπτωση χρήσης CUDA interoperability, τα ενημερωμένα δεδομένα μεταφέρονται απευθείας στο OpenGL buffer. Παράλληλα, ελέγχονται οι είσοδοι του χρήστη, επιτρέποντας αλληλεπίδραση με τη σκηνή, όπως η κίνηση της κάμερας. Οι view και projection matrices δημιουργούνται με τις glm::lookAt() και glm::perspective(), καθορίζοντας την οπτική γωνία και την προοπτική προβολή. Το framebuffer καθαρίζεται και, αν είναι ενεργοποιημένο το wireframe mode, αλλάζει ο τρόπος απόδοσης των πολυγώνων. Το shader πρόγραμμα ενεργοποιείται και λαμβάνει τις αντίστοιχες μεταβλητές για τις view και projection matrices. Στη συνέχεια, δεσμεύονται τα vertex και

index buffers, ενώ αν χρησιμοποιείται CUDA interoperability, ενημερώνεται το OpenGL buffer με τα επεξεργασμένα δεδομένα της GPU. Τέλος, το terrain αποδίδεται μέσω της `glDrawElements()`, το παράθυρο ανανεώνεται με `glfwSwapBuffers()`, και ελέγχονται νέα γεγονότα εισόδου μέσω της `glfwPollEvents()`.

Τερματισμός της εφαρμογής:

Όταν ο χρήστης κλείσει το παράθυρο, ο βρόχος rendering σταματάει και ακολουθούν οι διαδικασίες τερματισμού που είναι για την απελευθέρωση των δεσμευμένων πόρων. Αρχικά, καλείται η συνάρτηση `app_cleanup()`, η οποία αναλαμβάνει να αποδεσμεύσει τη μνήμη που χρησιμοποιήθηκε τόσο στη GPU όσο και στη CPU, καθώς και να διαγράψει τις εγγραφές των buffers που χρησιμοποιήθηκαν για την CUDA interoperability. Μετά την ολοκλήρωση αυτών των ενεργειών, το παράθυρο καταστρέφεται και η βιβλιοθήκη GLFW τερματίζεται μέσω της `glfwTerminate()`, ολοκληρώνοντας έτσι ομαλά την εκτέλεση του προγράμματος.

Οι συναρτήσεις `cublas_init` και `cublas_term` είναι υπεύθυνες για την αρχικοποίηση και την καταστροφή του cuBLAS handle, εξασφαλίζοντας ότι οι πόροι θα διαχειριστούν σωστά για την cublas κατά τη διάρκεια της ζωής της εφαρμογής.

4.2.4 Σημαντικά σημεία κώδικα

Εκτέλεση FFT με την βιβλιοθήκη cuFFT

Η συνάρτηση `cufftExecR2C` εκτελεί την Forward FFT μετατροπή, δηλαδή μετατρέπει δεδομένα του από το πεδίο του χρόνου (που αποθηκεύονται ως πραγματικοί αριθμοί στον πίνακα `d_ffftInput`) σε δεδομένα του πεδίου συχνότητας (ως σύνθετοι αριθμοί στον πίνακα `d_ffftResults`), χρησιμοποιώντας το προκαθορισμένο FFT plan (`planForward`). Αντίστοιχα, η `cufftExecC2R` πραγματοποιεί την αντίστροφη FFT μετατροπή, μετατρέποντας τα μιγαδικά αποτελέσματα του φάσματος (στον `d_ffftResults`) πίσω σε πραγματικά δεδομένα στον πίνακα `d_ffftInput`, σύμφωνα με το FFT plan (`planInverse`). Αυτές οι δύο συναρτήσεις είναι θεμελιώδεις στην εφαρμογή μας για την ανάλυση και επεξεργασία σημάτων διευκολύνοντας την εφαρμογή τεχνικών όπως η STFT (Short-Time Fourier Transform).

```

// Επεξεργασία κάθε chunk
for (int i = 0; i < numChunks; i++) {
    int offset = i * hopSize;
    int currentChunkSize = chunkSize;
    if (offset + chunkSize > numSamples)
        currentChunkSize = numSamples - offset;

    // Αντιγραφή του chunk από το fftInput στη μνήμη GPU
    cudaMemcpy(d_fftInput, fftInput + offset, currentChunkSize * sizeof(cufftReal), cudaMemcpyDeviceToDevice);

    if (currentChunkSize < chunkSize) // Μηδενισμός του padding αν το chunk είναι μικρότερο από το chunkSize
        cudaMemcpy(d_fftInput + currentChunkSize, 0, (chunkSize - currentChunkSize) * sizeof(cufftReal));

    // Εφαρμογή του hann window
    blocks = (currentChunkSize + threads - 1) / threads;
    applyWindow<<<blocks, threads>>>(d_fftInput, d_window, currentChunkSize);

    cufftExecR2C(planForward, d_fftInput, d_fftResults); // Εκτέλεση forward FFT

    // Εφαρμογή φίλτρου
    float spectrumRange = sampleRate * 2.0f;
    int lowCutoffBinIndex = (lowCutoffFreq / spectrumRange) * currentChunkSize;
    int highCutoffBinIndex = (highCutoffFreq / spectrumRange) * currentChunkSize;
    blocks = ((currentChunkSize / 2) + 1 + threads - 1) / threads;
    applyBandPassFilter<<<blocks, threads>>>(d_fftResults, (currentChunkSize / 2) + 1, lowCutoffBinIndex, highCutoffBinIndex);

    // Εκτέλεση inverse FFT για επιστροφή στο χρονικό domain
    cufftExecC2R(planInverse, d_fftResults, d_fftInput);

    // Κανονικοποίηση (διαιρούμε με το chunkSize)
    blocks = (currentChunkSize + threads - 1) / threads;
    overlapAddChunk <<<blocks, threads>>>(d_output, d_fftInput, d_window, currentChunkSize, offset, numSamples, hopSize);
}

```

Εικόνα 22: επεξεργασία κάθε chunk για την εφαρμογή του συχνοτικού φίλτρου.

Διαχείριση των δεδομένων της επεξεργασίας

Η δομή Vertex ορίζει ένα σημείο στον χώρο με τις συντεταγμένες x, y, z. Η δομή VerticesData χρησιμοποιείται για την αποθήκευση των δεδομένων γεωμετρίας του πλέγματος που θα απεικονιστεί. Συγκεκριμένα:

- **vertices:** Δείκτης προς το χώρο μνήμης στο CPU που περιέχει τα δεδομένα των κορυφών. Το μέγεθος αυτού του πίνακα υπολογίζεται ως $(width+1) \times (height+1)$ για να καλυφθεί ο πλήρης αριθμός κορυφών ενός πλέγματος.
- **device_vertices:** Αντίστοιχα, δείκτης στον χώρο μνήμης της GPU, όπου μεταφέρονται τα δεδομένα για χρήση από τους πυρήνες CUDA.
- **d_rotationMatrix:** Δείκτης σε έναν πίνακα 9 float που αποθηκεύει τον πίνακα περιστροφής και βρίσκεται στη μνήμη της GPU.
- **indices:** Δείκτης προς τον πίνακα δεικτών (indices) στο CPU, ο οποίος χρησιμοποιείται για την κατασκευή των τριγώνων. Το μέγεθος του πίνακα υπολογίζεται ως $width \times height \times 6$, αφού κάθε κελί του πλέγματος αντιστοιχεί σε δύο τρίγωνα (κάθε τρίγωνο απαιτεί 3 δείκτες).
- **v_n και i_n:** Μεταβλητές που αποθηκεύουν τον συνολικό αριθμό κορυφών και δεικτών αντίστοιχα.

```

struct Vertex {
    float x, y, z;
};

struct VerticesData
{
    Vertex* vertices;//(width + 1) * (height + 1)
    Vertex* device_vertices;//(width + 1) * (height + 1)
    float* d_rotationMatrix;
    unsigned int* indices;//(width * height * 6)
    unsigned int v_n;
    unsigned int i_n;
};

```

- Εικόνα 23: Οι Δομές Vertex και VerticesData.

Η συνάρτηση για την δημιουργία του πλέγματος

Η συνάρτηση `createHighTriCountPlane` δημιουργεί ένα πλέγμα (plane) με μεγάλο αριθμό τριγώνων βάση εισόδου ενός `'width'` και `'height'`, που καθορίζουν τις διαστάσεις του πλάνου σε τετραγωνικά μπλοκ, το οποίο χρησιμοποιείται για την αναπαράσταση ενός εδάφους με υψηλή πυκνότητα γεωμετρίας. Αρχικά, υπολογίζει τις αρχικές συντεταγμένες ώστε το πλέγμα να τοποθετείται κεντρικά στη σκηνή, ορίζοντας το μέγεθος κάθε κελιού (παραλληλογράμμου) και τις αρχικές τιμές για τις κορυφές. Μετά την τοποθέτηση όλων των κορυφών, η συνάρτηση δημιουργεί τον πίνακα δεικτών (indices) που καθορίζει πως οι κορυφές συνδέονται για να σχηματίσουν τρίγωνα. Ξεκινώντας με τα δύο πρώτα τρίγωνα στην κάτω αριστερή γωνία, χρησιμοποιεί επαναληπτικούς βρόχους για να συνδέσει ομαλά τις κορυφές τόσο οριζόντια όσο και κάθετα, ώστε να σχηματιστεί ένα συνεκτικό πλέγμα τριγώνων. Το αποτέλεσμα είναι ένα υψηλής πυκνότητας mesh, έτοιμο για περαιτέρω επεξεργασία και απόδοση σε περιβάλλοντα 3D γραφικών.

Προετοιμασία των Buffers για το Rendering

```
void drawFrame(GLFWwindow* window)
{
    VertexArray mesh_va;
    //Δυναμική ενημέρωση των σημείων στο buffer
    VertexBuffer mesh_vb(GL_DYNAMIC_DRAW, data.v_n * sizeof(Vertex));

#ifdef UseCudaInterop
    //register the mesh vertex buffer to cuda
    cudaGraphicsGLRegisterBuffer(&cudaVBufResource, mesh_vb.GetID(), cudaGraphicsMapFlagsWriteDiscard);
#endif

    VertexBufferLayout mesh_layout;
    mesh_layout.push<float>(3); // x, y, z
    mesh_va.AddBuffer(mesh_vb, mesh_layout);
    IndexBuffer mesh_ib(data.indices, data.i_n);
    Shader mesh_Shader(hightrishader::vertexProg, hightrishader::fragmentProg);

    // Ενεργοποίηση βάθους (Depth Test)
    glEnable(GL_DEPTH_TEST);
    glClearColor(0.1f, 0.1f, 0.1f, 1.0f);

    glfwSwapInterval(0); // uncap fps
}
```

Εικόνα 24: Κώδικας της συνάρτησης drawFrame, πριν τον επαναληπτικό βρόγχο.

Δημιουργία Vertex Array και Vertex Buffer:

Δημιουργείται ένα Vertex Array Object (VAO) για να συγκεντρώσει τη διαμόρφωση των δεδομένων των κορυφών. Παράλληλα, ορίζεται ένας Vertex Buffer (VBO) με δυναμική διαχείριση (GL_DYNAMIC_DRAW) που σηματοδοτεί την ενημέρωση των δεδομένων των κορυφών σε πραγματικό χρόνο.

Ενσωμάτωση CUDA Interop (εφόσον είναι ενεργό):

Εάν έχει ενεργοποιηθεί η υποστήριξη CUDA interoperability, ο vertex buffer καταχωρείται στη CUDA μέσω της συνάρτησης cudaGraphicsGLRegisterBuffer, επιτρέποντας την σύνδεση του με την CUDA για να τροποποιούμε απευθείας τα δεδομένα χωρίς επιπλέον μεταφορές μεταξύ CPU και GPU.

Ορισμός Διάταξης Δεδομένων (Layout):

Δημιουργείται ένας Vertex Buffer Layout που καθορίζει πως θα διαβαστούν τα δεδομένα του buffer. Εδώ, ορίζεται ότι κάθε κορυφή αποτελείται από 3 τιμές τύπου float (για τις συντεταγμένες x, y και z) και προστίθεται στο VAO.

Δημιουργία Index Buffer και Shader:

Τέλος, δημιουργείται ένας Index Buffer που θα χρησιμοποιηθεί για την απόδοση των τριγώνων, και φορτώνονται οι shader κώδικες (vertex και fragment) που θα εφαρμοστούν κατά τη διαδικασία rendering.

Ενημέρωση Δεδομένων και Vertex Buffer κατά τη Διάρκεια του Rendering Loop του και απόδοση της Σκηνής

Στο rendering loop υπολογίζεται η τρέχουσα γωνία βάσει του χρόνου, και στη συνέχεια το πρόγραμμα επιλέγει μεταξύ GPU- και CPU-based προσομοίωσης. Εάν ενεργοποιείται η επεξεργασία μέσω CUDA, καλείται η συνάρτηση `UpdateWaveSimulation`, η οποία εκτελεί στο GPU το sine wave και τον περιστροφικό μετασχηματισμό. Σε περίπτωση που δεν χρησιμοποιείται η CUDA-OpenGL interoperability, πραγματοποιείται μεταφορά των ενημερωμένων δεδομένων από τη μνήμη της GPU πίσω στη μνήμη της CPU μέσω της `cudaMemcpy`. Αντίθετα, όταν επιλέγεται η CPU-based επεξεργασία με την `UpdateWaveSimulationCPU`, και αν έχει ενεργοποιηθεί η CUDA interoperability, τα δεδομένα μεταφέρονται από τη CPU στη GPU, διασφαλίζοντας την ομοιόμορφη ενημέρωση του vertex buffer.

Μετά την ενημέρωση των δεδομένων, το πρόγραμμα διαχειρίζεται την επικοινωνία μεταξύ GPU και OpenGL για την απόδοση της σκηνής. Εάν είναι ενεργή η CUDA interoperability, καλείται η συνάρτηση `cudaInterop_Update()`, η οποία χαρτογραφεί το vertex buffer και ενημερώνει απευθείας τα δεδομένα του από τη GPU, εξαλείφοντας την ανάγκη μεταφοράς μέσω της CPU. Διαφορετικά, η ενημέρωση του buffer πραγματοποιείται με την `glBufferSubData`, η οποία αντιγράφει τα δεδομένα από τη μνήμη της CPU στο OpenGL buffer. Τέλος, δεσμεύεται ο index buffer και καλείται η `glDrawElements` για την απόδοση των τριγώνων, ολοκληρώνοντας έτσι το frame rendering με συνεχή και αποδοτική ενημέρωση της γεωμετρίας της σκηνής.

4.2.5 Benchmarking

Στη διαδικασία αυτή, θα αξιολογηθούν διαφορετικά σενάρια εκτέλεσης της εφαρμογής με στόχο να μετρηθεί η επίδραση της επιλογής επεξεργασίας μέσω GPU έναντι CPU και η περαιτέρω βελτιώσει όταν γίνεται και χρήση της CUDA interoperability. Η χρήση της CUDA Interop επιτρέπει την άμεση ενημέρωση των buffers της OpenGL από τη GPU, μειώνοντας την ανάγκη για μεταφορές δεδομένων από και προς την κατά την απόδοση της σκηνής. Αντίθετα, στα σενάρια χωρίς CUDA Interop, οι τα αποτελέσματα των υπολογισμών που γίνονται στη GPU με την `cublas` πρέπει να μεταφερθούν από τη GPU στη CPU και μετά να δρομολογηθούν ξανά πίσω στην GPU από κλήση της OpenGL για περαιτέρω επεξεργασία ή απόδοση. Στο benchmarking θα αναδειχθούν οι διαφορές όταν η εκτέλεση κώδικα γίνεται αποκλειστικά στην CPU, καθώς μπορούμε κατά την εκτέλεση του προγράμματος να κάνουμε άμεση εναλλαγή του τόπου εκτέλεσης, για να καταδειχθεί

η απόδοση της εφαρμογής κατά την εκτέλεση των διάφορων υπολογιστικών φορτίων, καθώς και πώς η GPU επιταχύνει τις υπολογιστικές διαδικασίες σε σχέση με την CPU. Τα αποτελέσματα αυτής της ανάλυσης θα παρουσιαστούν στον επόμενο πίνακα, επιτρέποντας μια συγκριτική αξιολόγηση της απόδοσης ανάλογα με το περιβάλλον εκτέλεσης.

Τα αποτελέσματα προέκυψαν από στατιστικές μετρήσεις της εφαρμογής, καταγράφοντας τον ρυθμό καρέ (Frames Per Second - FPS). Για μια πιο ολοκληρωμένη αξιολόγηση της απόδοσης, μετρήθηκε επίσης ο χρόνος μεταξύ διαδοχικών καρέ (Frame Time), ο οποίος δίνει μια πιο ακριβή εικόνα της απόκρισης της εφαρμογής. Κάθε μέτρηση πραγματοποιήθηκε με την εκτέλεση του προγράμματος για 1 λεπτό από 5 φορές στο σύστημα μου με Ryzen 5 7600 και GTX 1660 Super, υπό τις ίδιες συνθήκες, ώστε να διασφαλιστεί η εγκυρότητα των αποτελεσμάτων.

Πίνακας 4 Αποτελέσματα μετά από 5 δοκιμές

Test run	Average FPS	Average Frametime
CPU ONLY	157,33 FPS	6,355 ms
CUBLAS	193,48 FPS	5,16 ms
CUBLAS + CUDA Interop	456,87 FPS	2,188 ms

Τα αποτελέσματα δείχνουν πως η χρήση της CUBLAS οδηγεί σε σαφή βελτίωση των επιδόσεων σε σχέση με την σειριακή επεξεργασία της CPU, με αύξηση από περίπου 157 FPS σε 193 FPS και με σταθερό frametime 5,16 ms. Όταν προστίθεται και η τεχνική CUDA Interop, οι επιδόσεις βελτιώνονται δραματικά, φτάνοντας τα 456,87 FPS και μειώνοντας περαιτέρω το frametime σε 2,188 ms. Αυτό υποδηλώνει ότι οι τεχνικές μεταφοράς δεδομένων μέσω CUDA Interop είναι ιδιαίτερα αποδοτικές όταν χρησιμοποιούμε παραλληλοποίηση με βιβλιοθήκες της CUDA, μειώνοντας σημαντικά το overhead.

Συμπερασματικά, η σωστή αξιοποίηση των βιβλιοθηκών CUDA και CUBLAS μπορεί να αυξήσει δραματικά την απόδοση, καθιστώντας τις εφαρμογές πιο ομαλές και γρήγορες. Οι μετρήσεις αυτές είναι ενδεικτικές για εφαρμογές με υψηλές απαιτήσεις σε γραφικά και υπολογισμούς, όπου η GPU εκμεταλλεύεται πλήρως τους πόρους της.

4.3 Επεξεργασία Εικόνας με Χρήση της Βιβλιοθήκης NPP

Η παρούσα εφαρμογή επικεντρώνεται στην επεξεργασία εικόνας, αξιοποιώντας τη βιβλιοθήκη NPP (NVIDIA Performance Primitives) για την επιτάχυνση των υπολογισμών. Η NPP παρέχει βελτιστοποιημένες συναρτήσεις για επεξεργασία εικόνας, όπως φιλτράρισμα, μετασχηματισμούς και άλλες λειτουργίες υψηλής απόδοσης. Στην παρούσα ενότητα, θα παρουσιαστεί η χρήση της βιβλιοθήκης NPP για την εφαρμογή φίλτρου θόλωσης Gaussian και θα συγκριθεί η απόδοσή της με μια αντίστοιχη υλοποίηση σε CPU μέσω OpenCV.

4.3.1 Απαιτήσεις και Ρυθμίσεις Περιβάλλοντος

Στην παρούσα εφαρμογή, θα αναλυθεί ο τρόπος χρήσης της NPP μέσα στον κώδικα, καθώς και τα πλεονεκτήματα που παρέχει σε σύγκριση με μια σειριακή υλοποίηση σε CPU.

Η εφαρμογή υλοποιεί ένα Gaussian φίλτρο για την εξομάλυνση (blur) μιας εικόνας, χρησιμοποιώντας την OpenCV για τη διαχείριση των αρχείων εικόνας και την εκτέλεση της CPU-based επεξεργασίας. Για να επιτευχθεί η εκτέλεση του προγράμματος, απαιτείται η κατάλληλη ρύθμιση του Visual Studio, ώστε να υποστηρίζει τον μεταγλωττιστή CUDA και να επιτρέπει τη σωστή σύνδεση των βιβλιοθηκών.

Η βιβλιοθήκη NPP περιλαμβάνεται στο περιβάλλον ανάπτυξης της CUDA και για τη χρήση της απαιτείται η σωστή διαμόρφωση του linker στο Visual Studio. Συγκεκριμένα, απαιτείται η σύνδεση (linking) του αρχείου «nppif.lib», το οποίο ανήκει στην κατηγορία των γραμμικών και μη γραμμικών λειτουργιών φιλτραρίσματος εικόνας που παρέχει η βιβλιοθήκη NPP.

Για τη βιβλιοθήκη OpenCV, είναι απαραίτητη η σωστή ρύθμιση της μεταβλητής περιβάλλοντος 'PATH' στα Windows, ώστε το σύστημα να μπορεί να εντοπίσει τις απαραίτητες δυναμικές βιβλιοθήκες .dll και πρέπει να καθοριστεί η τοποθεσία των αρχείων 'include' για την ανάπτυξη του κώδικα, μέσω των ρυθμίσεων «VC++ Directories».

Όπως και με την NPP, για τη σωστή λειτουργία της OpenCV στη συγκεκριμένη εφαρμογή, απαιτείται η σύνδεση του αρχείου 'opencv_world4100.lib', το οποίο περιλαμβάνει όλες τις βασικές λειτουργίες της OpenCV σε μία ενιαία βιβλιοθήκη.

4.3.2 Gaussian blur:

Το Gaussian Blur είναι μια τεχνική εξομάλυνσης εικόνας που χρησιμοποιείται συχνά στην επεξεργασία εικόνας και στην υπολογιστή όραση. Αποτελεί μία από τις πιο διαδεδομένες τεχνικές θολώματος εικόνας, χρησιμοποιούμενη τόσο για την εξομάλυνση του θορύβου όσο και για την απαλότητα των λεπτομερειών με την μείωση των λεπτομερειών υψηλής συχνότητας. Σε αντίθεση με άλλες μεθόδους θολώματος, το Gaussian Blur προσφέρει μεγαλύτερο έλεγχο, διατηρώντας τις άκρες της εικόνας ενώ παράλληλα επιτυγχάνει ομαλές μεταβάσεις μεταξύ των pixel.

Η μέθοδος βασίζεται στη χρήση ενός πυρήνα που προκύπτει από την κανονική κατανομή (Gaussian distribution), η οποία χαρακτηρίζεται από τη γνωστή "καμπάνα" της. Η τυπική απόκλιση της κατανομής (standard deviation) καθορίζει το πλάτος της καμπάνας και, επομένως, το εύρος των βαρών που εφαρμόζονται στα γειτονικά pixel. Ο πυρήνας αυτός εφαρμόζει υψηλότερα βάρη στο κέντρο του και σταδιακά μειώνει τα βάρη προς τις άκρες, επιτρέποντας έτσι μια σταθμισμένη συνάρτηση μέσου όρου.

Κατά την εκτέλεση του φίλτρου, ο αλγόριθμος διατρέχει κάθε εικονοστοιχείο της εικόνας και, για κάθε pixel, υπολογίζει τον σταθμισμένο μέσο όρο των τιμών των γειτονικών pixel με βάση τα βάρη που καθορίζονται από τον πυρήνα. Το αποτέλεσμα της συνάρτησης συνδυασμού αποθηκεύεται σε ένα buffer εξόδου, παράγοντας έτσι μια τελικά θολωμένη εικόνα με ομαλές και φυσικές μεταβάσεις.

Η βιβλιοθήκη NPP παρέχει μια ολοκληρωμένη σειρά συναρτήσεων για την εφαρμογή του Gaussian Blur. Μέσω της βασικής λειτουργίας για Gaussian filtering, όπως αυτή που υλοποιείται με την `npFilterGauss_8u_C3R()`, επιτυγχάνεται η εξομάλυνση της εικόνας με το παραδοσιακό Gaussian φίλτρο, ενώ οι προχωρημένες εκδόσεις της λειτουργίας επιτρέπουν λεπτομερή ρύθμιση παραμέτρων όπως το μέγεθος του πυρήνα και τις τιμές των συντελεστών, προσαρμόζοντας έτσι το φίλτρο στις συγκεκριμένες ανάγκες της εφαρμογής. Επιπλέον υπάρχουν συναρτήσεις που ενσωματώνουν διαχείριση ορίων και εξασφαλίζουν ομαλή εφαρμογή του φίλτρου ακόμη και στα άκρα της εικόνας, για τις ειδικές περιπτώσεις.

4.3.3 Υλοποίηση κωδικα

Γενική Περιγραφή

Ο κώδικας αυτός υλοποιεί την εφαρμογή του Gaussian Blur τόσο στη CPU μέσω OpenCV όσο και στη GPU χρησιμοποιώντας τη βιβλιοθήκη NPP. Ο χρήστης μπορεί να

επιλέξει το μέγεθος του kernel μέσω του μενού επιλογών και στη συνέχεια ο θόρυβος εφαρμόζεται στην εικόνα τόσο με χρήση της CPU όσο και της GPU. Ο κώδικας συγκρίνει την απόδοση των δύο μεθόδων, μετρώντας τον χρόνο εκτέλεσης και αποθηκεύοντας τις παραγόμενες εικόνες.

Αρχικά, η εικόνα φορτώνεται στη μνήμη μέσω της OpenCV και ο χρήστης επιλέγει το επιθυμητό μέγεθος μάσκας από ένα σύνολο προκαθορισμένων τιμών. Ο κώδικας αντιστοιχίζει την επιλογή του χρήστη με τις διαθέσιμες σταθερές της NPP μέσω της συνάρτησης `getMaskSize()`. Στη συνέχεια, γίνεται δέσμευση μνήμης στη GPU και αντιγραφή των δεδομένων της εικόνας από τη CPU στη GPU για την εκτέλεση του Gaussian Blur μέσω της συνάρτησης `applyGaussianBlurCUDA()`.

Η συνάρτηση `applyGaussianBlurCUDA()` ετοιμάζει τους παραμέτρους που χρησιμοποιεί η `nppiFilterGauss\_8u\_C3R()` για την εφαρμογή του Gaussian Blur και εκτελεί το την εφαρμογή του φίλτρου. Σε αυτό το σημείο μετράμε και τον χρόνο εκτέλεσης για να γίνει σύγκριση μετά με την ταχύτητα της CPU. Εφόσον η επεξεργασία ολοκληρωθεί επιτυχώς, τα δεδομένα αντιγράφονται πίσω στη CPU και η τροποποιημένη εικόνα αποθηκεύεται. Παράλληλα, το Gaussian Blur εφαρμόζεται και στην CPU μέσω της OpenCV με τη χρήση της `cv::GaussianBlur()`, καταγράφοντας επίσης τον χρόνο εκτέλεσης για σύγκριση.

```
//Gaussian blur using NPP
NppStatus GaussianBlurNPP(const Npp8u* d_image, int width, int height, int channels, NppiMaskSize mask_size, Npp8u* d_out)
{
    auto start_t = std::chrono::high_resolution_clock::now(); // Έναρξη χρονομέτρησης

    NppiSize roiSize = { width, height };
    int step = width * channels;

    // Εφαρμογή Gaussian Blur
    NppStatus status = nppiFilterGauss_8u_C3R(d_image, step, d_out, step, roiSize, mask_size);
    auto end_t = std::chrono::high_resolution_clock::now(); // Λήξη χρονομέτρησης

    if (status)
    {
        std::cout << "npp error " << status << "\n";
    }

    std::chrono::duration<double> elapsed = end_t - start_t; // Υπολογισμός διάρκειας
    std::cout << "Execution time GPU: " << elapsed.count() << " seconds" << std::endl;

    return status;
}
```

Εικόνα 25: Η συνάρτηση GaussianBlurNPP().

Τέλος, το πρόγραμμα αποθηκεύει τις παραγόμενες εικόνες σε αρχεία ('out/image_CUDA_<timestamp>.jpg' και 'out/image_CPU_<timestamp>.jpg') και εμφανίζει τα αποτελέσματα στην κονσόλα.

Η υλοποίηση του Gaussian Blur στη CPU πραγματοποιείται μέσω της βιβλιοθήκης OpenCV με σειριακή εκτέλεση, γεγονός που οδηγεί σε σημαντικές επιβραδύνσεις, ειδικά για εικόνες μεγάλης ανάλυσης. Αντίθετα, στην υλοποίηση με τη βιβλιοθήκη NPP, εκμεταλλευόμαστε πλήρως την αρχιτεκτονική της GPU, όπου οι υπολογισμοί διεξάγονται ταυτόχρονα σε πολλά pixels μέσω του παραλληλισμού που προσφέρει η CUDA.

4.3.4 Αποτελέσματα προγράμματος και σύγκριση απόδοσης CPU-GPU

Συνθήκες Δοκιμής:

Για κάθε συνδυασμό ανάλυσης εικόνας και επιλογής πυρήνα για το φίλτρο, το πρόγραμμα εκτελέστηκε τρεις φορές ώστε να υπολογιστεί ο μέσος όρος του χρόνου εκτέλεσης. Οι εικόνες που χρησιμοποιήθηκαν είχαν τρεις διαφορετικές αναλύσεις: 3264x2448 (μικρή), 3472x4624 (μεσαία) και 6944x9248(μεγάλη). Αυτή η προσέγγιση εξασφάλισε αξιόπιστα αποτελέσματα που επιτρέπουν την ακριβή σύγκριση των επιδόσεων μεταξύ της σειριακής υλοποίησης στη CPU και της παραλληλισμένης υλοποίησης στη GPU.

Απόδοση

Για τις μικρές εικόνες, παρατηρείται ότι με έναν πυρήνα 3×3 ο χρόνος εκτέλεσης στη GPU ήταν ελαφρώς υψηλότερος από αυτόν στη CPU, με αποτέλεσμα μια επιτάχυνση (CPU/GPU) περίπου 0,57, γεγονός που υποδεικνύει ότι ο επιπρόσθετος χρόνος για την προετοιμασία της παράλληλης εκτέλεσης υπερβαίνει τα οφέλη για τόσο μικρά δεδομένα. Αντίθετα, όταν χρησιμοποιήθηκε ένας μεγαλύτερος πυρήνας 15×15 , ο χρόνος στη GPU παρέμεινε περίπου σταθερός, ενώ ο χρόνος στη CPU αυξήθηκε, οδηγώντας σε επιτάχυνση περίπου 2,67, κάτι που δείχνει πως ακόμη και για μικρές εικόνες, η GPU μπορεί να αποδώσει καλύτερα σε πιο απαιτητικές λειτουργίες. Για την εικόνα μεσαίου μεγέθους, η διαφορά στην απόδοση γίνεται πιο εμφανής, καθώς με πυρήνα 3×3 , ο χρόνος εκτέλεσης μεταξύ CPU και GPU ήταν σχεδόν ίσος, ενώ για πυρήνα 15×15 ο χρόνος στην CPU ήταν σχεδόν 5,21 φορές μεγαλύτερος από αυτόν στη GPU. Το πλεονέκτημα της GPU γίνεται ακόμη πιο προφανές, με επιτάχυνση περίπου 4,20 για τον πυρήνα 3×3 και εντυπωσιακή επιτάχυνση 18,91 για τον πυρήνα 15×15 για μεγάλες εικόνες.

Συμπεράσματα και Εφαρμογές:

Τα αποτελέσματα καταδεικνύουν ότι η παραλληλισμένη υλοποίηση του Gaussian Blur με τη χρήση της GPU και της βιβλιοθήκης NPP είναι ιδιαίτερα αποτελεσματική για

εικόνες υψηλής ανάλυσης ή όταν χρησιμοποιούνται μεγαλύτερα φίλτρα. Όσο αυξάνεται το μέγεθος της εικόνας και το μέγεθος του πυρήνα, τόσο πιο σημαντική γίνεται η διαφορά απόδοσης, με την GPU να επιτυγχάνει επιτάχυνση που φτάνει σχεδόν τα 20 φορές σε πολύ μεγάλες εικόνες.

Πίνακας 5: Χρόνοι εκτέλεσης

Ανάλυση Εικόνας	Κέρνελ	Χρόνος GPU (sec)	Χρόνος CPU (sec)	Επιτάχυνση (CPU/GPU)
Μικρή	3_X_3	0,00858	0,00486	0,57
Μικρή	15_X_15	0,00847	0,02260	2,67
Μεσαία	3_X_3	0,00869	0,00976	1,12
Μεσαία	15_X_15	0,00890	0,04627	5,21
Μεγάλη	3_X_3	0,00873	0,03671	4,20
Μεγάλη	15_X_15	0,00876	0,16575	18,91

5 Τεχνικές Λεπτομέρειες και Οδηγίες Χρήσης

Στο παρόν κεφάλαιο παρουσιάζονται οι τεχνικές λεπτομέρειες των εφαρμογών που αναπτύχθηκαν στο πλαίσιο αυτής της πτυχιακής εργασίας. Δίνεται μια αναλυτική περιγραφή των βασικών αλγορίθμων και τεχνολογιών που χρησιμοποιήθηκαν, καθώς και των βελτιστοποιήσεων που εφαρμόστηκαν για τη βελτίωση της απόδοσης.

Επιπλέον, παρέχονται οδηγίες χρήσης για τις εφαρμογές, με έμφαση στη διαδικασία εκτέλεσής τους. Περιλαμβάνονται χρήσιμες συμβουλές για τον χρήστη, τόσο για τη σωστή παραμετροποίηση των εφαρμογών όσο και για την αντιμετώπιση πιθανών προβλημάτων που μπορεί να προκύψουν κατά την εκτέλεσή τους.

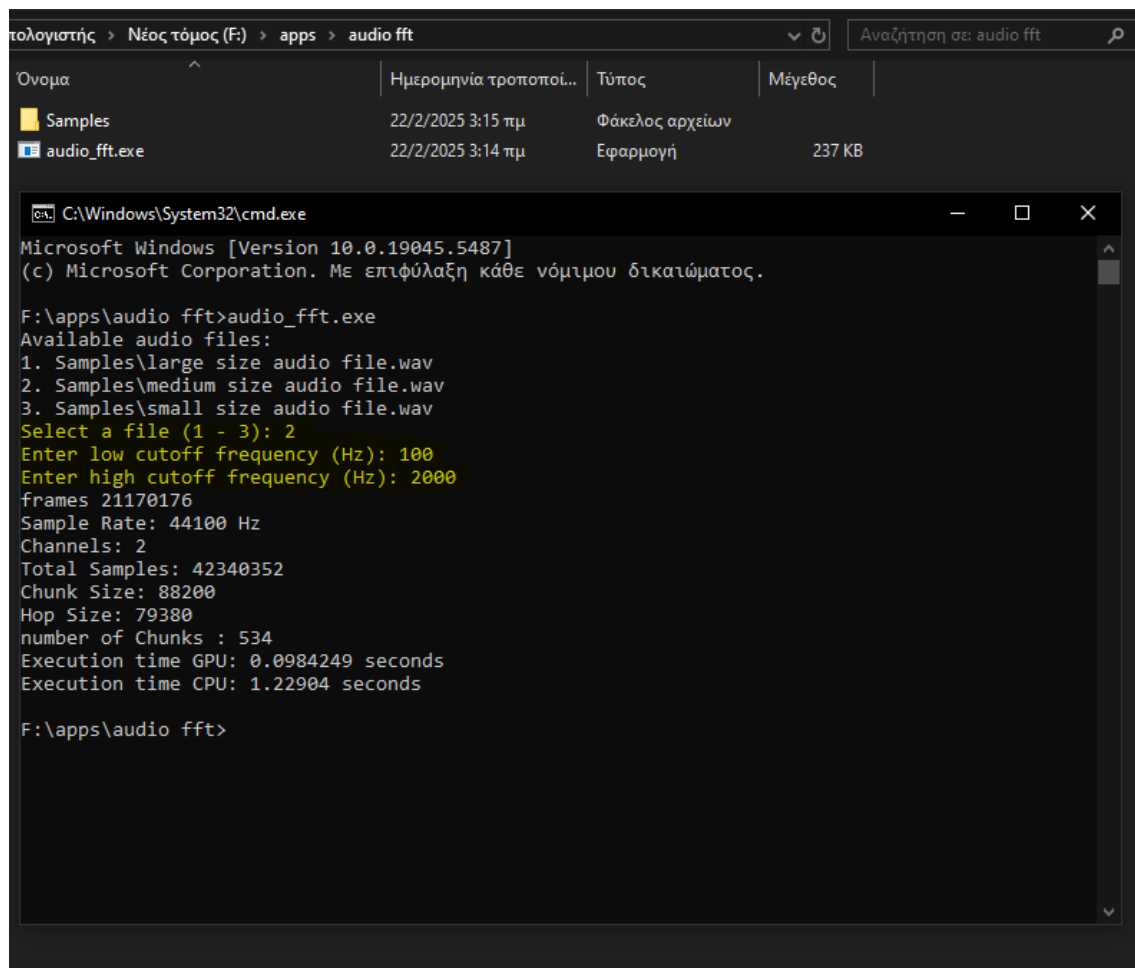
5.1 Πως χρησιμοποιούμε τις εφαρμογές

5.1.1 Εφαρμογή επεξεργασίας ήχου με fft:

Η εφαρμογή ακολουθεί μια καλά οργανωμένη δομή, διασφαλίζοντας μια ομαλή και καθοδηγούμενη εμπειρία χρήστη. Κατά την εκκίνηση, ελέγχεται αν υπάρχει ο φάκελος "Samples" στον ίδιο κατάλογο με το εκτελέσιμο αρχείο (`.exe`). Εάν ο φάκελος δεν βρεθεί ή είναι άδειος, εμφανίζεται ένα κατάλληλο μήνυμα σφάλματος και το πρόγραμμα τερματίζεται. Στη συνέχεια, ο χρήστης επιλέγει το αρχείο ήχου που θέλει να επεξεργαστεί από τα διαθέσιμα στον φάκελο. Αφού γίνει η επιλογή, καλείται να ορίσει το εύρος συχνοτήτων (0-20.000 Hz) που επιθυμεί να διατηρήσει. Το πρόγραμμα διαβάζει το αρχείο, εξάγει πληροφορίες όπως ο ρυθμός δειγματοληψίας και το πλήθος δειγμάτων, και εφαρμόζει το επιλεγμένο φίλτρο. Μετά την επεξεργασία, δημιουργείται ένα νέο αρχείο ήχου με τα φιλτραρισμένα δείγματα, το οποίο αποθηκεύεται στον ίδιο φάκελο με διαφορετικό όνομα ώστε να μην αντικατασταθεί το αρχικό. Τέλος, ο χρήστης ενημερώνεται για την ολοκλήρωση της διαδικασίας και μπορεί είτε να επεξεργαστεί νέο αρχείο είτε να τερματίσει την εφαρμογή.

Παρακάτω παρουσιάζεται ένα στιγμιότυπο οθόνης που απεικονίζει τη διαδικασία που περιγράφηκε προηγουμένως. Συγκεκριμένα, θα δούμε την πλήρη εκτέλεση της

εφαρμογής για ένα αρχείο ήχου, από την επιλογή του αρχείου έως τη δημιουργία του φιλτραρισμένου αποτελέσματος. Αυτό το στιγμιότυπο βοηθά στην καλύτερη κατανόηση της λειτουργίας του προγράμματος και της ροής εργασίας που ακολουθεί ο χρήστης.



Εικόνα 26: Εκτέλεση της εφαρμογής audio_fft.exe

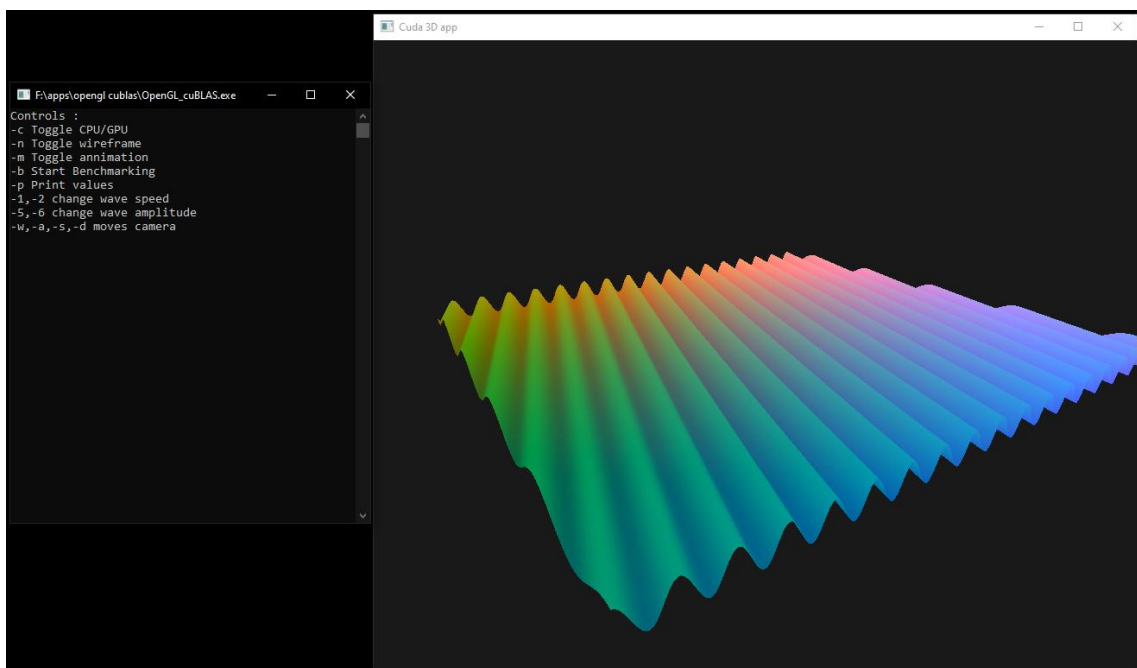
Στην παραπάνω εικόνα έχουν επισημανθεί με κίτρινο χρώμα τα σημεία όπου το πρόγραμμα ζητά είσοδο από τον χρήστη. Όπως φαίνεται, η εκτέλεση της εφαρμογής είναι απλή και δεν απαιτεί καμία επιπλέον παράμετρο από τη γραμμή εντολών.

5.1.2 Πρόγραμμα 3D γραικών με OpenGL και CUBLAS

Για να εκτελέσει την εφαρμογή, ο χρήστης απλώς καλεί το εκτελέσιμο αρχείο, είτε μέσω της γραμμής εντολών είτε με διπλό κλικ από την Εξερεύνηση των Windows. Δεν απαιτείται καμία επιπλέον ρύθμιση ή παράμετρος, καθιστώντας τη διαδικασία εκκίνησης γρήγορη και απλή.

Κατά την εκκίνηση της εφαρμογής ανοίγουν δύο παράθυρα, ένα παράθυρο κονσόλας και ένα παράθυρο στο οποίο γίνεται το rendering της σκηνής. Στην

κονσόλα εμφανίζονται χρήσιμες πληροφορίες και οδηγίες για τον έλεγχο του προγράμματος, ενώ ο χρήστης μπορεί να τροποποιήσει τις παραμέτρους της σκηνής μέσω των διαθέσιμων εντολών. Συγκεκριμένα, με το πλήκτρο 'c' γίνεται εναλλαγή του φόρτου εργασίας μεταξύ CPU και GPU, με το 'n' εναλλάσσεται η απόδοση σε wireframe mode, και με το 'm' ενεργοποιείται ή απενεργοποιείται η κίνηση του πλέγματος. Επιπλέον, με το 'b' ξεκινά η λειτουργία benchmarking, με το 'p' εμφανίζονται οι τιμές του πλάτους και της ταχύτητας του κύματος, και με τις εντολές -1 και -2 αλλάζει η ταχύτητα των κυμάτων ενώ οι -5 και -6 το πλάτος τους. Τέλος, με τα πλήκτρα -w, -a, -s, -d επιτρέπουν τη μετακίνηση της κάμερας. Η συνολική διαδικασία είναι σχεδιασμένη ώστε να είναι απλή και διαισθητική, επιτρέποντας στον χρήστη να ελέγχει άμεσα την εφαρμογή και να τροποποιεί τη σκηνή με βάση τις προτιμήσεις του.

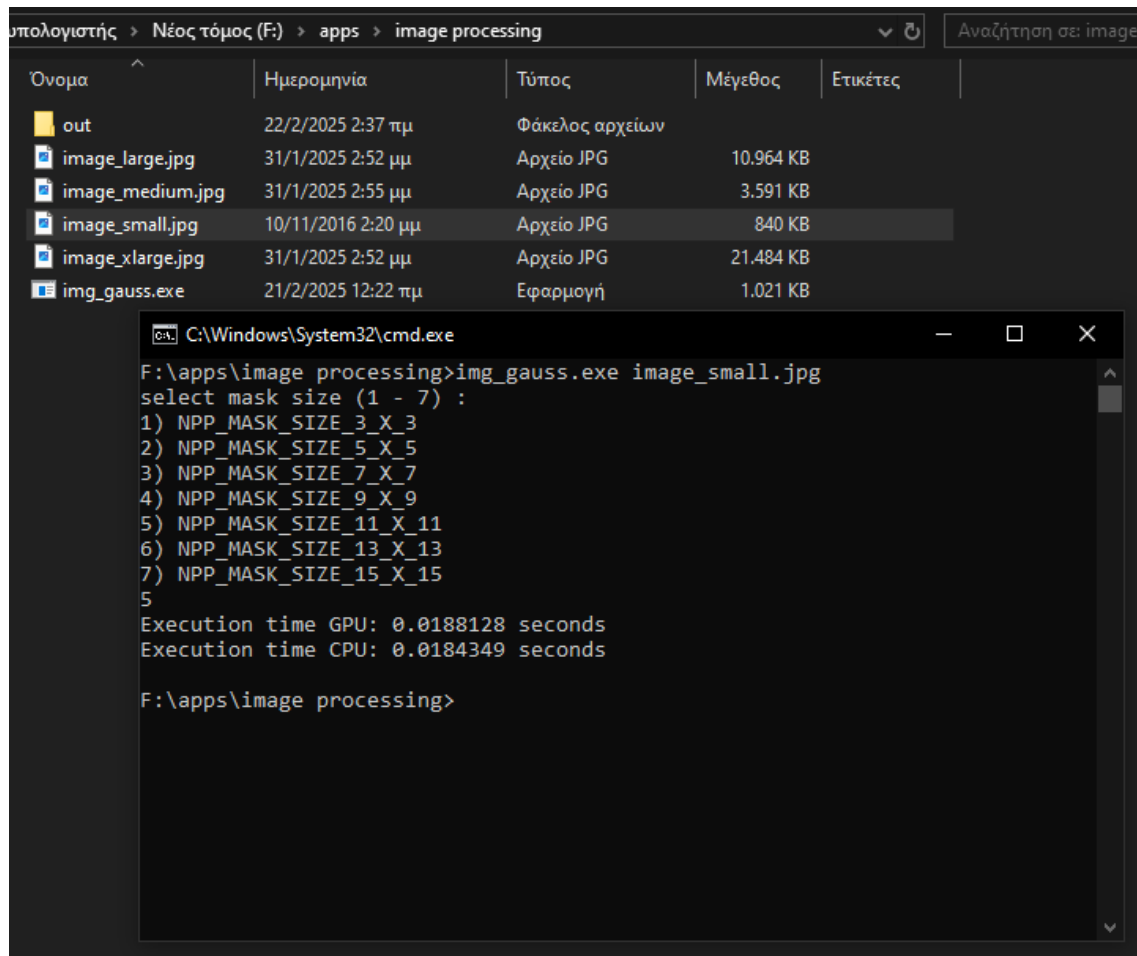


Εικόνα 27: Η Εφαρμογή 3D Γραφικών σε λειτουργία σε λειτουργία.

5.1.3 Εφαρμογή για την επεξεργασία εικόνας

Η εφαρμογή εκτελείται μέσω γραμμής εντολών, όπου ως παράμετρος λαμβάνει το αρχείο εικόνας που πρόκειται να υποβληθεί σε επεξεργασία. Μόλις ξεκινήσει και διαβαστεί σωστά το αρχείο, εμφανίζεται στον χρήστη μια λίστα επιλογών που αφορά το μέγεθος του kernel για το Gaussian φίλτρο. Ο χρήστης καλείται να επιλέξει μία από τις διαθέσιμες επιλογές, η οποία καθορίζει το επίπεδο εξομάλυνσης που θα εφαρμοστεί στην εικόνα. Αμέσως μετά την επιλογή, η εφαρμογή προχωρά στην επεξεργασία της εικόνας,

εφαρμόζοντας το φίλτρο, και ολοκληρώνει την εκτέλεση, αποδίδοντας το τελικό επεξεργασμένο αποτέλεσμα.



Εικόνα 28: Εκτέλεση της εφαρμογής img_gauss.exe.

5.2 Λειτουργίες κλειδιά

Σε αυτό το σημείο θα περιγράψουμε ορισμένες βοηθητικές λειτουργίες που έχουν υλοποιηθεί στον κώδικα των εφαρμογών και παίζουν σημαντικό ρόλο στη συνολική δομή της εφαρμογής. Αυτές οι λειτουργίες συμβάλλουν στην αποδοτικότητα, την οργάνωση και τη διευκόλυνση της διαχείρισης των δεδομένων και των πόρων, επιτρέποντας την ομαλή εκτέλεση και την επεκτασιμότητα του προγράμματος.

5.2.1 Προσομοίωση εφέ κύματος και περιστροφής

Στην εφαρμογή, η ενημέρωση του πλέγματος πραγματοποιείται με συνδυασμό ενός CUDA kernel, που καθορίζει το ύψος κάθε σημείου μέσω του sine wave effect, και μιας κλήσης της cuBLAS, που εφαρμόζει περιστροφή μέσω πράξεων γραμμικής άλγεβρας.

Η συνάρτηση kernel `sineWaveKernel` εφαρμόζει το `sinewave` εφέ σε κάθε κορυφή ταυτόχρονα. Κάθε νήμα υπολογίζει τον δείκτη της κορυφής που θα επεξεργαστεί. Για κάθε κορυφή, τα `x` και `z` συντεταγμένα ανακτώνται, και το `y` καθορίζεται βάσει της ημιτονοειδούς συνάρτησης που εξαρτάται από τον χρόνο, τις συντεταγμένες και τους παράγοντες πλάτους και ταχύτητας του κύματος. Η συνάρτηση `ApplyWaveEffect` ορίζει τον αριθμό των νήματος ανά block και τον συνολικό αριθμό blocks για την εκτέλεση του kernel .

Εφαρμογή του Rotation Effect με cuBLAS:

Η λειτουργία `ApplyRotationEffect` χρησιμοποιεί το cuBLAS για να πραγματοποιήσει τον πολλαπλασιασμό πινάκων που αντιστοιχεί σε περιστροφή των κορυφών. Αρχικά, δημιουργείται ένας πίνακας 3x3 (rotation matrix) που ορίζει την περιστροφή γύρω από τον κατακόρυφο άξονα (με χρήση των συναρτήσεων `cosf` και `sinf`). Ο πίνακας αυτός μεταφέρεται στη μνήμη της GPU με τη χρήση της `cudaMemcpy`. Στη συνέχεια, με την κλήση της συνάρτησης `cublasSgemm` πραγματοποιείται ο πολλαπλασιασμός του πίνακα περιστροφής με τον πίνακα κορυφών:

- *handle*: Το handle της βιβλιοθήκης cuBLAS.
- *CUBLAS_OP_N*: Καθορίζει ότι ούτε ο πίνακας περιστροφής ούτε ο πίνακας κορυφών δεν υφίστανται μετασχηματισμό (no transposition).
- *Διαστάσεις*: Οι παράμετροι 3, `data.v_n`, 3 καθορίζουν το μέγεθος των πινάκων, όπου το 3 είναι ο αριθμός γραμμών του πίνακα περιστροφής, το `data.v_n` ο αριθμός των κορυφών και το τελευταίο 3 ο αριθμός στηλών του πίνακα περιστροφής.
- *alpha* και *beta*: Οι παράγοντες κλίμακας για το αποτέλεσμα ($\alpha = 1.0$) και για την αρχική τιμή της εξόδου ($\beta = 0.0$).
- *Πίνακες*: Ο πίνακας περιστροφής και ο πίνακας κορυφών, όπου το αποτέλεσμα του πολλαπλασιασμού αποθηκεύεται στον ίδιο πίνακα κορυφών, επιτρέποντας έτσι την ταχεία ενημέρωση των συντεταγμένων τους.

```

// CUDA Kernel για την εφαρμογή του sinewave effect
__global__ void sineWaveKernel(Vertex* d_vertices, float time, int vertNums, float wave_amp, float wave_speed) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i >= vertNums) return;

    float x = d_vertices[i].x; // X
    float z = d_vertices[i].z; // Z
    d_vertices[i].y = wave_amp * sinf(wave_speed * time + x * 0.05f + z * 0.1f);
}

void ApplyWaveEffect(float time) { // Μέθοδος που εφαρμόζει το sine wave effect
    int threadsPerBlock = 512;
    int numBlocks = (data.v_n + threadsPerBlock - 1) / threadsPerBlock;
    sineWaveKernel << numBlocks, threadsPerBlock >> (data.device_vertices, time, data.v_n, WAVE_AMPLITUDE, WAVE_SPEED);
}

void ApplyRotationEffect(float angle) { // Matrix Multiplication με cuBLAS για περιστροφή
    float rotationMatrix[] = {
        cosf(angle), 0, sinf(angle),
        0, 1, 0,
        -sinf(angle), 0, cosf(angle)
    };

    cudaMemcpy(data.d_rotationMatrix, rotationMatrix, 9 * sizeof(float), cudaMemcpyHostToDevice);

    // Υπολογισμός νέων συντεταγμένων με cuBLAS
    float alpha = 1.0f, beta = 0.0f;
    cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, // Χωρίς μετασχηματισμό
        3, data.v_n, 3, // Διαστάσεις: 3xN * 3xN
        &alpha, data.d_rotationMatrix, 3, // Μήτρα περιστροφής 3x3
        (float*)data.device_vertices, 3, &beta, // Πίνακας κορυφών 3xN
        (float*)data.device_vertices, 3 // Αποθήκευση του αποτελέσματος στον ίδιο πίνακα
    );
}

void UpdateWaveSimulation(float time, float angle)
{
    ApplyWaveEffect(time);
    ApplyRotationEffect(angle);
    cudaDeviceSynchronize();
}

```

Εικόνα 29: Λειτουργίες για ενημέρωση του πλέγματος στην σκηνή.

5.2.2 Δομές για την διαχείριση των buffer στην OpenGL

Η κλάση *VertexBuffer* διαχειρίζεται ένα OpenGL Vertex Buffer Object (VBO) μέσω ενός ιδιωτικού μέλους, του *m_RendererID*, που αποθηκεύει το αναγνωριστικό του buffer. Διαθέτει δύο κατασκευαστές: ο πρώτος δέχεται δεδομένα και μέγεθος για αρχική φόρτωση του buffer, ενώ ο δεύτερος δημιουργεί ένα κενό buffer με καθορισμένη χρήση (usage) και μέγεθος. Μέσω του destructor, το buffer διαγράφεται σωστά, εξασφαλίζοντας την απελευθέρωση των πόρων. Επιπλέον, παρέχει μεθόδους *Bind* και *Unbind* για τη σύνδεση και αποσύνδεση του buffer, καθώς και μια μέθοδο για την απόκτηση του αναγνωριστικού του.

```

class VertexBuffer
{
private:
    unsigned int m_RendererID;
public:
    VertexBuffer(const void* data, unsigned int size) {
        GLCall(glGenBuffers(1, &m_RendererID));
        GLCall(glBindBuffer(GL_ARRAY_BUFFER, m_RendererID));
        GLCall(glBufferData(GL_ARRAY_BUFFER, size, data, GL_STATIC_DRAW));
    }

    VertexBuffer(unsigned int usage, unsigned int size) {
        GLCall(glGenBuffers(1, &m_RendererID));
        GLCall(glBindBuffer(GL_ARRAY_BUFFER, m_RendererID));
        GLCall(glBufferData(GL_ARRAY_BUFFER, size, nullptr, usage));
    }

    ~VertexBuffer() {
        GLCall(glDeleteBuffers(1, &m_RendererID));
    }

    void Bind() const {
        GLCall(glBindBuffer(GL_ARRAY_BUFFER, m_RendererID));
    }

    void Unbind() const {
        GLCall(glBindBuffer(GL_ARRAY_BUFFER, 0));
    }

    unsigned int GetID() const { return m_RendererID; }
};

```

Εικόνα 30:VertexBufffer Class.

Σε παρόμοιο πνεύμα με την κλάση VertexBuffer, οι υπόλοιπες κλάσεις του συστήματος παρέχουν μια ολοκληρωμένη και οργανωμένη υποδομή για την ανάπτυξη εφαρμογών OpenGL. Η IndexBuffer διαχειρίζεται τους δείκτες που χρησιμοποιούνται για την αποτελεσματική αναπαράσταση και απόδοση των κορυφών, εξασφαλίζοντας παράλληλα τη σωστή απελευθέρωση των πόρων μέσω του destructor. Η VertexBufferLayout και ο VertexArray συνεργάζονται για να οργανώσουν τα δεδομένα κορυφών: η πρώτη καθορίζει τη δομή και το μέγεθος των στοιχείων που αποτελούν κάθε κορυφή (όπως τύποι δεδομένων και αριθμός στοιχείων), ενώ ο δεύτερος συγκεντρώνει όλα τα buffers που απαιτούνται για την απόδοση, επιτρέποντας την OpenGL να αναγνωρίσει τα δεδομένα. Επιπλέον έχουμε την κλάση Shader που αναλαμβάνει το ρόλο της δημιουργίας, σύνταξης και διαχείρισης των shader προγραμμάτων με δυνατότητα για αποτελεσματική διαχείριση των μεταβλητών uniform μέσω του caching.

Η συνδυασμένη χρήση αυτών των κλάσεων παρέχει ένα ευέλικτο και αποδοτικό περιβάλλον για την μελλοντική ανάπτυξη της εφαρμογής.

5.2.3 Εφαρμογή του short time Fourier transform για την ανάλυση και επεξεργασία δείγματα ήχου στο πεδίο της συχνότητας

Η απλή εφαρμογή του FFT σε ολόκληρο το δείγμα ήχου οδηγούσε σε υπερβολική χρήση μνήμης, γεγονός που συχνά οδηγούσε σε εξάντληση των διαθέσιμων πόρων και μεγάλες καθυστερήσεις στην εκτέλεση. Για να αντιμετωπιστεί αυτό το ζήτημα, υιοθετήθηκε η τεχνική του Short Time Fourier Transform (STFT), η οποία επιτρέπει την ανάλυση και επεξεργασία του ήχου σε μικρότερα κομμάτια (chunks). Με αυτόν τον τρόπο, γίνεται δυνατή η εφαρμογή του FFT σε κάθε chunk ξεχωριστά, μειώνοντας σημαντικά τις απαιτήσεις μνήμης και επιταχύνοντας τη διαδικασία. Παρά τα πλεονεκτήματα, αυτή η προσέγγιση εισήγαγε και νέες προκλήσεις, καθώς η διαχωρισμένη ανάλυση οδήγησε στην εμφάνιση artifacts που απαιτούσαν περαιτέρω διόρθωση για την επίτευξη ενός ομαλού και συνεκτικού τελικού αποτελέσματος.

Η χρήση επικάλυψης (overlap) κατά 20% στα chunks επιτρέπει την ομαλή σύνδεση των επεξεργασμένων τμημάτων του ήχου κατά την ανασύνθεση του σήματος. Όταν ένα σήμα διαιρείται σε μικρότερα κομμάτια για ανάλυση μέσω του STFT, τα άκρα κάθε τμήματος μπορεί να παρουσιάζουν απότομες μεταβάσεις, οι οποίες δημιουργούν ανεπιθύμητα artifacts μετά την επεξεργασία. Με την εφαρμογή επικάλυψης, διασφαλίζεται ότι τα κομμάτια αυτά αλληλοκαλύπτονται, έτσι ώστε οι διαφορές στα άκρα να εξομαλύνονται και το τελικό αποτέλεσμα να είναι ομαλό και συνεκτικό.

Παράλληλα, η χρήση παραθύρου τύπου Hanning έχει ως στόχο τη μείωση της φασματικής διαρροής (spectral leakage) κατά τη διάρκεια της εφαρμογής της FFT. Το Hanning παράθυρο, χάρη στην ομαλή του διαμόρφωση που καταλήγει σε μηδενική τιμή στα άκρα, μειώνει τις απότομες μεταβάσεις στα όρια κάθε chunk. Αυτό βοηθά στο να διατηρηθεί η ακρίβεια της ανάλυσης του ήχου, επιτρέποντας την καλύτερη απομόνωση των συχνοτικών συστατικών και αποφεύγοντας παραμορφώσεις που θα μπορούσαν να προκύψουν από τις απότομες μεταβάσεις στα άκρα των τμημάτων.

Παρακάτω θα δούμε όλο το κώδικα για την επεξεργασία αυτή, υπόψιν ότι ο κώδικας των κέρνελ είναι αρκετά αυτονόητος και δεν χρειάζεται πολύ ανάλυση.

```

void applySTFTOverlapAdd(cufftReal* fftinput, int numSamples, int sampleRate, int lowCutoffFreq, int highCutoffFreq) {
    // Ορισμός παραμέτρων
    int threads = 256;
    int chunkSize = sampleRate * 2;
    float overlapFactor = 0.2f; // % overlap
    int hopSize = static_cast<int>(chunkSize * (1.0f - overlapFactor));
    int numChunks = ceil((float)(numSamples - chunkSize) / hopSize) + 1;

    std::cout << "Chunk Size: " << chunkSize << "\nHop Size: " << hopSize
        << "\nnumber of Chunks : " << numChunks << std::endl;

    // Δημιουργία FFT plans για forward και inverse FFT
    cufftHandle planForward, planInverse;
    CUFFT_CALL(cufftPlan1d(&planForward, chunkSize, CUFFT_R2C, 1));
    CUFFT_CALL(cufftPlan1d(&planInverse, chunkSize, CUFFT_C2R, 1));

    // Δέσμευση μνήμης για το chunk που θα επεξεργαστούμε
    cufftReal* d_fftInput;
    CUDA_CHECK(cudaMalloc((void**)&d_fftInput, chunkSize * sizeof(cufftReal)));
    cufftComplex* d_fftResults;
    CUDA_CHECK(cudaMalloc((void**)&d_fftResults, ((chunkSize / 2) + 1) * sizeof(cufftComplex)));

    // Δέσμευση μνήμης για το τελικό αποτέλεσμα (αρχικοποιημένο σε 0)
    cufftReal* d_output;
    CUDA_CHECK(cudaMalloc((void**)&d_output, numSamples * sizeof(cufftReal)));
    CUDA_CHECK(cudaMemset(d_output, 0, numSamples * sizeof(cufftReal)));

    cufftReal* d_window;
    CUDA_CHECK(cudaMalloc((void**)&d_window, chunkSize * sizeof(cufftReal)));

    int blocks = (chunkSize + threads - 1) / threads;
    generateHannWindow << <blocks, threads >> > (d_window, chunkSize, overlapFactor);

    // Επεξεργασία κάθε chunk
    for (int i = 0; i < numChunks; i++) {
        int offset = i * hopSize;
        int currentChunkSize = chunkSize;
        if (offset + chunkSize > numSamples)
            currentChunkSize = numSamples - offset;

        // Αντιγραφή του chunk από το fftinput στη μνήμη GPU
        cudaMemcpy(d_fftInput, fftinput + offset, currentChunkSize * sizeof(cufftReal), cudaMemcpyDeviceToDevice);

        if (currentChunkSize < chunkSize) // Μηδενισμός του padding αν το chunk είναι μικρότερο από το chunkSize
            cudaMemcpy(d_fftInput + currentChunkSize, 0, (chunkSize - currentChunkSize) * sizeof(cufftReal));

        // Εφαρμογή του hann window
        blocks = (currentChunkSize + threads - 1) / threads;
        applyWindow<<<blocks, threads>>>(d_fftInput, d_window, currentChunkSize);

        cufftExecR2C(planForward, d_fftInput, d_fftResults); // Εκτέλεση forward FFT

        // Εφαρμογή φίλτρου
        float spectrumRange = sampleRate * 2.0f;
        int lowCutoffBinIndex = (lowCutoffFreq / spectrumRange) * currentChunkSize;
        int highCutoffBinIndex = (highCutoffFreq / spectrumRange) * currentChunkSize;
        blocks = ((currentChunkSize / 2) + 1 + threads - 1) / threads;
        applyBandPassFilter<<<blocks, threads>>>(d_fftResults, (currentChunkSize / 2) + 1, lowCutoffBinIndex, highCutoffBinIndex);

        // Εκτέλεση inverse FFT για επιστροφή στο χρονικό domain
        cufftExecC2R(planInverse, d_fftResults, d_fftInput);

        // Κανονικοποίηση (διαιρούμε με το chunkSize)
        blocks = (currentChunkSize + threads - 1) / threads;
        overlapAddChunk <<<blocks, threads >>>(d_output, d_fftInput, d_window, currentChunkSize, offset, numSamples, hopSize);
    }

    // Αντιγραφή του τελικού αποτελέσματος πίσω στο fftinput ή σε άλλο buffer εξόδου
    cudaMemcpy(fftinput, d_output, numSamples * sizeof(cufftReal), cudaMemcpyDeviceToDevice);

    // Καθαρισμός
    cufftDestroy(planForward);
}

```

Εικόνα 31: Η συνάρτηση applySTFTOverlapAdd.

6 Συμπεράσματα

Περιγραφή και Δραστηριότητες

Κατά τη διάρκεια της εκπόνησης της πτυχιακής μου εργασίας, απέκτησα νέες θεωρητικές και πρακτικές γνώσεις που με βοήθησαν να εμβαθύνω στον τομέα της πληροφορικής. Επεκτείνοντας τις αρχικές μου γνώσεις, ασχολήθηκα με το Visual Studio και τις βιβλιοθήκες CUDA, οι οποίες, μέσω της ευρείας τους εφαρμογής, με βοήθησαν να μετατρέψω την κάρτα γραφικών σε ένα πραγματικά ισχυρό εργαλείο. Η μελέτη αυτών των βιβλιοθηκών μου επέτρεψε να κατανοήσω καλύτερα τις διαφορές και τη σχέση μεταξύ παράλληλης και σειριακής επεξεργασίας, ενώ ταυτόχρονα μου άνοιξε νέες προοπτικές στο πεδίο της ανάπτυξης εφαρμογών, στον χώρο της επεξεργασίας ήχου, των 3D γραφικών και της επεξεργασίας εικόνας

Δυσκολίες και Αντιμετώπιση Προκλήσεων

Κατά την πορεία της εργασίας, αντιμετώπισα αρκετές προκλήσεις που απαιτούσαν επιμονή και δημιουργικές λύσεις. Συγκεκριμένα, το debugging σε περιβάλλον GPU αποτέλεσε μια ιδιαίτερη δυσκολία, καθώς ήταν κάτι που δεν είχα αντιμετωπίσει στο παρελθόν και με ανάγκασε να εξερευνήσω νέες προσεγγίσεις για την ανίχνευση και επίλυση σφαλμάτων. Επιπλέον, η εφαρμογή του FFT παρουσίασε προβλήματα λόγω της μεγάλης κατανάλωσης μνήμης και της πρόσθετης επεξεργασίας που απαιτούνταν για τον ορθό χειρισμό των παραθύρων (hann window).

Για την αντιμετώπιση αυτών των προβλημάτων, επέλεξα να οργανώσω καλύτερα τα project μου, να δοκιμάζω διάφορες προσεγγίσεις και να αξιοποιώ την εμπειρία που απέκτησα μέσα από την πρακτική εφαρμογή και τη συνεχή επαφή με το πρόβλημα. Αυτή η διαδικασία και η διαχείριση του κώδικα με δίδαξαν τη σημασία της συστηματικότητας και της υπομονής σε κάθε τεχνική πρόκληση.

Αναστοχασμός και Κριτική της Διαδικασίας

Αν και το έργο που πραγματοποίησα με ικανοποίησε και με δίδαξε πολλά, αναλογιζόμενος τα γεγονότα, διαπίστωσα ότι μερικές από τις προσεγγίσεις θα μπορούσαν να βελτιωθούν με τις σημερινές τεχνολογικές εξελίξεις. Αν θα ξαναέκανα την πτυχιακή, πιθανότατα θα ενσωμάτωνα περισσότερες σύγχρονες τεχνικές και εργαλεία,

αξιοποιώντας νέες πλατφόρμες και βελτιστοποιημένες μεθόδους για την ανάπτυξη εφαρμογών. Ωστόσο, οι βασικές μεθοδολογίες που ακολούθησα παραμένουν αξιόλογες και αποτέλεσαν θεμέλιο για την κατανόησή μου του αντικειμένου.

Η εργασία αυτή απέφερε ουσιαστικά οφέλη και γνώσεις που θα μπορούσα να ενσωματώσω στη μελλοντική μου επαγγελματική πορεία και δημιούργησε ένα ισχυρό ενδιαφέρον για περαιτέρω έρευνα στα πεδία.

Τελική Σκέψη

Η εκπόνηση αυτής της πτυχιακής εργασίας, παρά τις δυσκολίες που συνάντησα, ήταν μια πολύτιμη εμπειρία. Μου προσέφερε γνώσεις, εμπειρίες και ενέπνευσε την επιθυμία για περαιτέρω εμβάθυνση σε σύνθετα και πολυδιάστατα πεδία. Στο μέλλον, τόσο εγώ όσο και άλλοι φοιτητές θα μπορούσαμε να συνεχίσουμε την έρευνα σε πιο εξειδικευμένα θέματα, εξερευνώντας νέες εφαρμογές και βελτιώνοντας τις υπάρχουσες μεθοδολογίες για την ανάπτυξη παράλληλων προγραμμάτων και την αξιοποίηση προηγμένων υπολογιστικών πόρων.

Βιβλιογραφία

- [1] Μάργαρης, Αθ., "ΓΡΑΜΜΙΚΗ ΑΛΓΕΒΡΑ", 1η Έκδοση, ΕΚΔΟΣΕΙΣ Α. ΤΖΙΟΛΑ & ΥΙΟΙ Α.Ε., 2015. ISBN: 978-960-418-471-2.
- [2] Μάργαρης, Αθ., "Εισαγωγή στην Παράλληλη Επεξεργασία", 1η Έκδοση, Εκδόσεις Τζιόλα, 2024. ISBN: 978-618-221-094-9.
- [3] Μάργαρης, Αθ., "Σήματα και Συστήματα: Συνεχούς και Διακριτού Χρόνου", 1η Έκδοση, Εκδόσεις Τζιόλα, 2011. ISBN: 978-960-418-366-1.
- [4] Γραμματή Πάντζιου, Βασίλειος Μάμαλης, Αλέξανδρος Τομαράς, "Εισαγωγή στον Παράλληλο Υπολογισμό", 2η έκδοση, ΕΚΔΟΣΕΙΣ ΝΕΩΝ ΤΕΧΝΟΛΟΓΙΩΝ ΙΔΙΩΤΙΚΗ ΚΕΦΑΛΑΙΟΥΧΙΚΗ ΕΤΑΙΡΕΙΑ, 2023. ISBN: 978-960-578-106-4.
- [5] Hearn, D., Baker, P., & Warren, C., "Γραφικά Υπολογιστών με OpenGL (4η Έκδοση)", Εκδόσεις Τζιόλα, 2021, ISBN: 978-960-418-895-6.
- [6] Burger, W., & Burge, M. J., "Ψηφιακή Επεξεργασία Εικόνων", 3η Έκδοση, Εκδόσεις Φούντας, 2024, ISBN: 9789603308355.
- [7] <https://www.jstatsoft.org/article/view/v008i14>
Marsaglia, G., "Xorshift RNGs", Journal of Statistical Software, 8(14), [21/2/2022].
- [8] <https://docs.nvidia.com/cuda>
NVIDIA Corporation, "CUDA Toolkit Documentation", [21/2/2022].
- [9] <https://docs.nvidia.com/cuda/cublas>
NVIDIA Corporation, "cuBLAS Library User Guide", [21/2/2022].
- [10] <https://docs.nvidia.com/cuda/cusparse>
NVIDIA Corporation, "cuSPARSE Library User Guide", [21/2/2022].
- [11] <https://docs.nvidia.com/cuda/cufft>
NVIDIA Corporation, "cuFFT Library User Guide", [21/2/2022].
- [12] <https://nvidia.github.io/cccl/thrust/index.html>
NVIDIA Corporation, "NPP Library User Guide", [22/2/2025].
- [13] <https://learnopengl.com/> J. De Vries, "Learn OpenGL", [22/2/2025].

- [14] <https://www.songho.ca/opengl/index.html>
Song Ho Ahn, "OpenGL Tutorials", [22/2/2025].
- [15] https://www.youtube.com/watch?v=C_zFhWdM4ic
Computerphile, "How Blurs & Filters Work", YouTube, [22/2/2025].
- [16] <https://github.com/NVIDIA/CUDALibrarySamples>
NVIDIA, "CUDA Library Samples", GitHub, [22/2/2025].
- [17] R. Cristi, "Short-Time Fourier Transform (STFT)", Naval Postgraduate School,
Διαθέσιμο: <https://faculty.nps.edu/rcristi/EO3404/B-Discrete-Fourier-Transform/text/3-STFT.pdf>
- [18] Libsndfile, "A C library for reading and writing sound files", 2024. Διαθέσιμο:
<https://libsndfile.github.io/libsndfile/>