ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

Παράλληλη Επιτάχυνση αλγορίθμων Dictionary Learning και Παραγοντοποίησης με εφαρμογή σε fMRI: k-SVD, αλγόριθμος MM, PARAFAC2

Χρήστος Δ. Πατσούρας
Δημήτριος Α. Παπαγεωργίου

Επιβλέπων: **Σέργιος Θεοδωρίδης,** Καθηγητής

ΑΘΗΝΑ

ΟΚΤΩΒΡΙΟΣ 2017

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Παράλληλη Επιτάχυνση αλγορίθμων Dictionary Learning και Παραγοντοποίησης με εφαρμογή σε fMRI: k-SVD, αλγόριθμος MM, PARAFAC2

Χρήστος Δ. Πατσούρας

Α.Μ.: 1115201100132

Δημήτριος Α. Παπαγεωργίου

Α.Μ.: 1115201100122

ΕΠΙΒΛΕΠΟΝΤΕΣ: **Σέργιος Θεοδωρίδης,** Καθηγητής

ΠΕΡΙΛΗΨΗ

Η αξιοσημείωτη πρόοδος που παρατηρείται στην Ιατρική με την πάροδο των χρόνων σχετίζεται άμεσα με την ταχεία και ευρεία εξάπλωση της χρήσης των υπολογιστών στις επιστήμες υγείας (Ιατρική, Βιολογία, Βιοτεχνολογία). Οι πολύπλοκοι υπολογισμοί που απαιτούνται σε ερευνητικό και πειραματικό στάδιο καθιστούν τον τομέα της Ιατρικής άρρηκτα συνδεδεμένο με εκείνον της Πληροφορικής.

Ένα από τα φλέγοντα και ενδιαφέροντα ερευνητικά ζητήματα στον τομέα της Ιατρικής – συγκεκριμένα της νευρολογίας - είναι η λειτουργία του ανθρώπινου εγκεφάλου. Μια διαδεδομένη μέθοδος ανάλυσης της εγκεφαλικής δραστηριότητας τού ανθρώπου είναι η Λειτουργική Απεικόνιση Μαγνητικού Συντονισμού (fMRI). Είναι μη επεμβατική μέθοδος μέσω της οποίας μπορούμε να μελετήσουμε τις περιοχές του εγκεφάλου που ενεργοποιούνται κατά τη διάρκεια εκτέλεσης μιας ενέργειας (π.χ. ομιλία, παρατήρηση μιας εικόνας).

Στην εργασία μας ασχολούμαστε με την εφαρμογή αλγορίθμων Dictionary Learning και Factorization σε fMRI. Πιο συγκεκριμένα επικεντρωνόμαστε στους αλγορίθμους k-SVD, MM και PARAFAC2. Στόχος μας είναι η επιτάχυνση των παραπάνω αλγορίθμων μέσω της χρήσης κάρτας γραφικών, η οποία παρέχει τη δυνατότητα πολλαπλών παράλληλων πράξεων πάνω σε ογκώδη δεδομένα. Τα πειράματα μας διενεργήθηκαν με χρήση CUDA και Matlab.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Επιτάχυνση αλγορίθμων με χρήση κάρτας γραφικών, Dictionary Learning, Παραγοντοποίηση, fMRI

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: fMRI, k-SVD, MM, PARAFAC2, GPU

ABSTRACT

The remarkable progress observed in medicine over the years is related to the rapid and widespread deployment of computer use in health sciences (Medicine, Biology, Biotechnology). The complex calculations required at the research and experimental stages make the field of medicine inextricably linked to that of computer technology.

One of the burning and interesting research issues in the medical field is the function of the human brain. A widespread method of analyzing human brain activity is functional Magnetic Resonance Imaging (fMRI). It is a non-invasive method through which we can study the brain regions that are activated during the performance of an activity (for example speech, observation of an image).

In our thesis we deal with the implementation of Dictionary Learning and Factorization algorithms in fMRI. Specifically, we focus on k-SVD, MM and PARAFAC2 algorithms. Our goal is to accelerate the above algorithms through the use of GPU, which offers the capability of multiple parallel operations over big data. Our experiments were conducted using CUDA and Matlab.

SUBJECT AREA: Algorithm acceleration using GPU, Dictionary Learning, Factorization, fMRI

KEYWORDS: fMRI, k-SVD, MM, PARAFAC2, GPU

Στις οικογένειές μας για την έμπρακτη στήριξή τους

ΕΥΧΑΡΙΣΤΙΕΣ

Για τη διεκπεραίωση της παρούσας Πτυχιακής Εργασίας, θα θέλαμε να ευχαριστήσουμε τον επιβλέποντα, καθ. Σέργιο Θεοδωρίδη, που μας βοήθησε να εντοπίσουμε τα ενδιαφέροντά μας και μας καθοδήγησε σε όλη τη διάρκεια της εργασίας με τη μεγάλη του εμπειρία και τις πολύπλευρες γνώσεις του.

Ιδιαίτερη μνεία θα πρέπει να κάνουμε στον διδάκτορα Ιωάννη Κοψίνη και στους υποψήφιους διδάκτορες Χρήστο Χατζηχρήστο και Manuel Moreno Morante, με τους οποίους βρισκόμασταν σε διαρκή επικοινωνία και μας καθοδήγησαν αποτελεσματικά καθόλη τη διάρκεια της Εργασίας.

ΠΕΡΙΕΧΟΜΕΝΑ

ΠΡΟΛΟΓΟΣ **13**

1. ΕΙΣΑΓΩΓΗ **14**

1.1 Dictionary Learning **14**

1.2 Παραγοντοποίση πινάκων **15**

1.2.1 Ανάλυση Ιδιόμορφων Τιμών (SVD) 16

1.2.2 Ανάλυση QR 16

1.2.3 Ανάλυση Cholesky 16

1.3 Παραγοντοποίση τανυστών **17**

1.3.1 Εισαγωγή στους τανυστές 17

1.3.2 Παραγοντοποίηση τανυστών fMRI 18

1.3.3 Κανονική Ανάλυση (CANDECOMP) 18

1.4 Λειτουργική Απεικόνιση Μαγνητικού Συντονισμού **19**

2. ΠΑΡΑΛΛΗΛΟΣ ΥΠΟΛΟΓΙΣΜΟΣ **23**

2.1 Το χρονικό του παράλληλου υπολογισμού **23**

2.2 Η ανάπτυξη των υπολογισμών γενικού σκοπού με κάρτα γραφικών **24**

2.3 Η αρχιτεκτονική CUDA **26**

2.4 Τα βασικά της γλώσσας CUDA C/C++ **27**

2.4.1 Συνάρτηση πυρήνα CUDA (CUDA kernel) 27

2.4.2 Ετερογενείς υπολογισμοί 28

2.4.3 Ιεραρχία threads και blocks 28

2.4.4 Ιεραρχία μνήμης 29

2.4.5 Υπολογιστική ικανότητα 30

2.5 Βιβλιοθήκες CUDA **31**

2.5.1 Η βιβλιοθήκη cuBLAS 31

2.5.2 Η βιβλιοθήκη cuRAND 32

2.5.3 Η βιβλιοθήκη cuSOLVER 33

2.6 Υπολογιστικοί πόροι συστήματος διεξαγωγής πειραμάτων **34**

3. ΑΛΓΟΡΙΘΜΟΣ MM **35**

3.1 Εισαγωγή στις μεθόδους ανάλυσης fMRI 35

3.2 Υποβοηθούμενη Μάθηση με Λεξικό 36
3.2.1 Εποπτευόμενη Μάθηση με Λεξικό 36
3.2.2 Υποβοηθούμενη από Άτομα Μάθηση με Λεξικό 36
3.2.3 Μέθοδος Βελτιστοποίησης 37
3.2.4 Ο αλγόριθμος 38

3.3 Υλοποίηση 40
3.3.1 Υλοποίηση σε MATLAB με πίνακες GPU 40
3.3.2 Υλοποίηση σε CUDA 42
3.3.3 Σύγκριση των τριών υλοποιήσεων 44

4. ΑΛΓΟΡΙΘΜΟΣ K-SVD 46

4.1 Εισαγωγή στη Μάθηση με Επίγνωση Αραιότητας (Sparsity-Aware Learning) 46
4.1.1 Εισαγωγή στο Υπερ-πλήρες λεξικό 46
4.1.2 Η Μάθηση με λεξικό (Dictionary Learning) 46

4.2 Ο αλγόριθμος k-SVD 47
4.2.1 Συνοπτική περιγραφή προβλήματος 47
4.2.2 Ο αλγόριθμος 47
4.2.3 Συσχέτιση με το πρόβλημα του fMRI 49

4.3 Η παραλλαγή του k-SVD και το Batch-OMP 50

4.4 Υλοποίηση στην αρχιτεκτονική CUDA 51
4.4.1 Επιτάχυνση του Batch-OMP 52
4.4.2 Επιτάχυνση του Σταδίου Ενημέρωσης Λεξικού (Dictionary Update) 53
4.4.3 Επιτάχυνση του k-SVD 55

4.5 Επιδόσεις υλοποίησης 55

5. ΑΛΓΟΡΙΘΜΟΣ PARAFAC2 59

5.1 Εισαγωγή σε PARAFAC και PARAFAC2 59

5.2 Συσχέτιση PARAFAC2 με fMRI 61

5.3 Υλοποίηση 61
5.3.1 Υλοποίηση σε MATLAB με πίνακες GPU 61
5.3.2 Υλοποίηση σε CUDA 63

6. ΣΥΜΠΕΡΑΣΜΑΤΑ 65

6.1 Συμπεράσματα αλγορίθμου MM 65

6.2 Συμπεράσματα αλγορίθμου k-SVD 65

6.3 Συμπεράσματα αλγορίθμου PARAFAC2 65

ΠΙΝΑΚΑΣ ΟΡΟΛΟΓΙΑΣ 67

ΣΥΝΤΜΗΣΕΙΣ – ΑΡΚΤΙΚΟΛΕΞΑ – ΑΚΡΩΝΥΜΙΑ 69

ΠΑΡΑΡΤΗΜΑ I 70

ΥΛΟΠΟΙΗΣΗ ΑΛΓΟΡΙΘΜΩΝ ΣΕ MATLAB ΜΕ ΧΡΗΣΗ GPU 70

1. Αλγόριθμος MM 70

2. Αλγόριθμος PARAFAC2 71

ΠΑΡΑΡΤΗΜΑ II 76

ΥΛΟΠΟΙΗΣΗ ΑΛΓΟΡΙΘΜΩΝ ΣΕ CUDA 76

1. Αλγόριθμος MM 76

2. Αλγόριθμος k-SVD 95

3. Αλγόριθμος PARAFAC2 164

ΑΝΑΦΟΡΕΣ 201

ΚΑΤΑΛΟΓΟΣ ΕΙΚΟΝΩΝ

Εικόνα 1: Ένας τανυστής τρίτης τάξης....................17

Εικόνα 2: Οι εικόνες του εγκεφάλου ξεδιπλώνονται σε πίνακες και σωρεύονται σε τανυστές....................18

Εικόνα 3: Ανάλυση CANDECOMP σε πίνακα τριών διαστάσεων....................18

Εικόνα 4: Εμπρόσθια κομμάτια τανυστή....................19

Εικόνα 5: Ανατομική εικόνα εγκεφάλου....................20

Εικόνα 6: Λειτουργική εικόνα εγκεφάλου....................20

Εικόνα 7: Γραφική παράσταση μιας τυπικής HRF....................21

Εικόνα 8: Σημεία ενεργοποίησης εγκεφάλου....................22

Εικόνα 9: Duke Nukem 3D (1996), από τα πρώτα παιχνίδια με τρισδιάστατα γραφικά.24

Εικόνα 10: Σύγκριση CPU και GPU σε GFLOPS....................26

Εικόνα 11: Τυπική κλήση συνάρτησης πυρήνα....................28

Εικόνα 12: CPU (Host) και GPU (Device)....................28

Εικόνα 13: Συνάρτηση πυρήνα που χρησιμοποιεί blocks και threads....................28

Εικόνα 14: Πλέγμα από blocks που περιέχουν threads....................29

Εικόνα 15: Ιεραρχία μνήμης....................30

Εικόνα 16: Σύγκριση cuBLAS με MKL....................32

Εικόνα 17: Σύγκριση cuRAND με MKL....................33

Εικόνα 18: Σύγκριση παραγοντοποιήσεων Cholesky, LU και QR ανάμεσα σε cuSOLVER και MKL....................33

Εικόνα 19: Χαρακτηριστικά υπολογιστή διεξαγωγής πειραμάτων....................34

Εικόνα 20: Ψευδοκώδικας αλγορίθμου MM....................40

Εικόνα 21: Διαμόρφωση επιτάχυνσης αλγορίθμου MM σε MATLAB με GPU....................42

Εικόνα 22: Διαμόρφωση επιτάχυνσης αλγορίθμου MM σε CUDA σε σχέση με τα δεδομένα εισόδου....................44

Εικόνα 23: Σύγκριση χρόνων τριών υλοποιήσεων MM....................45

Εικόνα 24: Συνοπτικός ψευδοκώδικας των βημάτων του αλγορίθμου k-SVD 49

Εικόνα 25: Ο ψευδοκώδικας της διαδικασίας OMP 50

Εικόνα 26: Ο ψευδοκώδικας του προσεγγιστικού (Approximate) k-SVD 52

Εικόνα 27: Χρόνος Εκτέλεσης του αλγορίθμου k-SVD 56

Εικόνα 28: Επιτάχυνση της υλοποίησης σε CUDA του k-SVD 56

Εικόνα 29: Ποσοστό Επιτάχυνσης της υλοποίησης σε CUDA του k-SVD 57

Εικόνα 30: Επιτάχυνση της υλοποίησης σε CUDA του k-SVD σε συνάρτηση με τον αριθμό επαναλήψεων 57

Εικόνα 31: Ψευδοκώδικας αλγορίθμου PARAFAC2 61

Εικόνα 32: Διαμόρφωση επιτάχυνση αλγορίθμου PARAFAC2 σε MATLAB με GPU 63

Εικόνα 33: Επιβράδυνση υλοποίησης PARAFAC2 σε CUDA για διαφορετικά δεδομένα εισόδου 64

ΚΑΤΑΛΟΓΟΣ ΠΙΝΑΚΩΝ

Πίνακας 1: Σύγκριση χρόνων εκτέλεσης ΜΜ σε MATLAB με χρήση CPU και GPU 41

Πίνακας 2: Σύγκριση χρόνων εκτέλεσης ΜΜ σε MATLAB με χρήση CPU και σε CUDA44

Πίνακας 3: Σύγκριση εκτελέσεων k-SVD μεταξύ CPU και GPU 55

Πίνακας 4: Σύγκριση χρόνων εκτέλεσης PARAFAC2 σε MATLAB με χρήση CPU και GPU..62

Πίνακας 5: Σύγκριση χρόνων εκτέλεσης PARAFAC2 σε MATLAB με χρήση CPU και CUDA 64

ΠΡΟΛΟΓΟΣ

Η παρούσα Πτυχιακή Εργασία εκπονήθηκε στην Αθήνα κατά το ακαδημαϊκό έτος 2016-2017. Αποτελεί απαραίτητη προϋπόθεση για τη λήψη του πτυχίου μας ως προπτυχιακοί φοιτητές στο Τμήμα Πληροφορικής και Τηλεπικοινωνιών του Εθνικού και Καποδιστριακού Πανεπιστημίου Αθηνών. Κατά τη διάρκεια φοίτησής μας και παρακολουθώντας πληθώρα μαθημάτων διαφορετικών μεταξύ τους, ένα από τα θέματα που μας κέντρισε σημαντικά το ενδιαφέρον ήταν ο τρόπος που λειτουργεί ο ανθρώπινος εγκέφαλος, ο τρόπος που αντιλαμβάνεται, επεξεργάζεται και αντιδρά στα ερεθίσματα. Παράλληλα, πάντα είχαμε μια ιδιαίτερη κλίση προς τον προγραμματισμό. Ο κ. Θεοδωρίδης μας καθοδήγησε δίνοντάς μας ένα θέμα για την πτυχιακή μας εργασία που συνδυάζει και τα δύο. Αφενός μεν διαβάσαμε και κατανοήσαμε τον τρόπο που συλλέγονται με τη μέθοδο του fMRI τα δεδομένα που παράγονται από τον εγκέφαλο και μελετήσαμε τους αλγορίθμους που τα χειρίζονται, αφετέρου δε ασχοληθήκαμε προγραμματιστικά με την επιτάχυνση των αλγορίθμων αυτών.

1. ΕΙΣΑΓΩΓΗ

Στην ενότητα της εισαγωγής θα παρουσιάσουμε τις βασικές έννοιες της πτυχιακής μας εργασίας. Όπως είναι φανερό και από τον τίτλο, θα μιλήσουμε για τις έννοιες του Dictionary Learning, της Λειτουργικής Απεικόνισης Μαγνητικού Συντονισμού (fMRI), της παραγοντοποίησης πινάκων και τανυστών. Στις επόμενες ενότητες θα δούμε πως αυτές οι έννοιες συνδυάζονται και σχετίζονται με τους αλγορίθμους που αναφέρονται στον τίτλο (k-SVD, MM, PARAFAC2).

1.1 Dictionary Learning

Τόσο οι φυσικοί αισθητήρες (μάτι, αυτί) όσο και οι τεχνητοί (κάμερες, μικρόφωνα) συλλέγουν κάθε δευτερόλεπτο τεράστιο όγκο από δεδομένα μεγάλων διαστάσεων. Οι αισθητήρες αυτοί δεν είναι σε θέση να επεξεργαστούν τα δεδομένα, οπότε τα δειγματοληπτούν με υψηλότερο ρυθμό από την εγγενή διάσταση των δεδομένων. Η εγγενής διάσταση (intrinsic dimensionality) ή ισχύουσα διάσταση (effective dimension) είναι το πλήθος των παραμέτρων που μας επιτρέπουν να προσεγγίσουμε με σημαντική ακρίβεια τα δεδομένα και είναι μικρότερη από την πραγματική διάσταση. Στην πραγματικότητα, όμως, η πληροφορία που χρειαζόμαστε είναι εκείνη που σχετίζεται με τα αίτια που προκαλούν τα φαινόμενα που καταγράφουν οι αισθητήρες. Με άλλα λόγια μπορούμε να μειώσουμε τη διάσταση των δειγματοληπτημένων δεδομένων στην εγγενή διάσταση της βασικής διαδικασίας χωρίς αισθητή απώλεια στην επίδοση.

Ένας διαισθητικός τρόπος να προσεγγίσουμε το πρόβλημα της μείωσης διαστάσεων (dimensionality reduction) είναι να εντοπίσουμε τι προκαλεί το χάσμα διαστάσεων μεταξύ του φυσικού φαινομένου και της παρατήρησης. Ο πιο συνηθισμένος λόγος είναι η διαφορά στην αναπαράσταση των δεδομένων στον αισθητήρα σε σχέση με το φυσικό χώρο. Σε μερικές περιπτώσεις είναι, για παράδειγμα, ένας απλός γραμμικός μετασχηματισμός του χώρου αναπαράστασης που μπορεί να υπολογιστεί μέσω της Ανάλυσης Κύριων Συνιστωσών (PCA). Σε άλλες περιπτώσεις, οι αισθητήρες καταγράφουν ταυτόχρονα δύο ή περισσότερες διαδικασίες με τα αίτια που τις προκαλούν να βρίσκονται σε διαφορετικούς υποχώρους και απαιτούνται άλλες μέθοδοι, όπως η Ανάλυση Ανεξάρτητων Συνιστωσών (ICA), για να προσδιορίσουμε τις διαφορετικές διαδικασίες πίσω από τα δεδομένα που παρατηρούμε. Η μέθοδος ICA μπορεί να διαχωρίσει διαφορετικά αίτια ή πηγές αναλύοντας τα στατιστικά χαρακτηριστικά των δεδομένων και ελαχιστοποιώντας την αμοιβαία πληροφορία (mutual information) ανάμεσα στα παρατηρούμενα δείγματα. Το ICA διαφέρει από το PCA στο γεγονός ότι μπορεί να διαχωρίσει πηγές όχι μόνο σε σχέση με δεύτερης τάξης συσχετίσεις μέσα σε ένα σύνολο δεδομένων, αλλά και σε σχέση και με υψηλότερης τάξης στατιστικά.

Όμως, η αναπαράσταση των δεδομένων μπορεί να είναι υπερπλήρης (overcomplete), δηλαδή ο αριθμός των πηγών ή ο αριθμός των υποχώρων που χρησιμοποιούνται για την περιγραφή των δεδομένων να είναι μεγαλύτερος από τη διάσταση των δεδομένων. Η χρησιμότητα της μείωσης διαστάσεων σε αυτήν την περιπτώση αφορά την αποδοτικότητα. Παρόλο που το πλήθος των πιθανών διεργασιών στον κόσμο είναι τεράστιο, το πλήθος των αιτίων που καταγράφουν οι αισθητήρες μας σε μια χρονική στιγμή είναι πολύ μικρότερο: οι παρατηρούμενες διεργασίες είναι αραιές (sparse) στο σύνολο όλων των πιθανών αιτίων. Με άλλα λόγια, παρόλο που το πλήθος των υποχώρων παρατήρησης είναι μεγάλο, μόνο λίγοι από αυτούς θα περιέχουν δείγματα δεδομένων από τις μετρήσεις των αισθητήρων. Προσδιορίζοντας αυτούς τους υποχώρους, βρίσκουμε μια αναπαράσταση του συνόλου των υποχώρων σε μειωμένο χώρο μικρότερων διαστάσεων από τον αρχικό.

Το σημαντικό ερώτημα που τίθεται είναι πως θα προσδιορίσουμε τους υποχώρους που βρίσκονται τα δεδομένα που καταγράφει ο αισθητήρας. Η επιλογή είναι αποφασιστικής σημασίας για την αποδοτική μείωση των διαστάσεων. Αυτή η ανάγκη πυροδότησε ένα νέο και υποσχόμενο πεδίο έρευνας που ονομάζεται μάθηση με λεξικό (Dictionary Learning). Το Dictionary Learning εστιάζει στην κατασκευή νέων αλγορίθμων από άτομα (atoms) ή υποχώρους που παρέχουν αποτελεσματικές αναπαραστάσεις κατηγοριών σημάτων. Τα άτομα είναι στοιχειώδη διανύσματα μοναδιαίας νόρμας και στο σύνολό τους συνθέτουν το λεξικό. Οι περιορισμοί σπανιότητας (sparsity constraints) είναι κλειδί στους περισσότερους αλγορίθμους που λύνουν το πρόβλημα του Dictionary Learning, διότι επιβάλλουν την αναγνώριση των κύριων αιτίων των παρατηρούμενων δεδομένων και ευνοούν την ακριβή αναπαράσταση της σχετικής πληροφορίας. Στόχος της αραιής αναπαράστασης (sparse representation) είναι να εκφράσει ένα σήμα ως γραμμικό συνδυασμό ενός μικρού αριθμού σημάτων που προέρχονται από μια πηγή που ονομάζεται λεξικό. Έστω K η διάσταση του λεξικού και k η διάσταση του σήματος με K>k. Το λεξικό είναι υπερπλήρες (K>k) όταν καλύπτει το χώρο του σήματος και τα άτομα είναι γραμμικώς εξαρτημένα. Σε αυτήν την περίπτωση, η αναπαράσταση του σήματος δεν είναι μοναδική.

Η έρευνα στον τομέα του Dictionary Learning ακολουθεί τρεις διαφορετικές κατευθύνσεις που αντιστοιχούν σε τρεις κατηγορίες αλγορίθμων: i) οι πιθανοτικές μέθοδοι, ii) οι μέθοδοι που βασίζονται στην ομαδοποίηση (clustering) ή στην κβάντωση διανυσμάτων, iii) οι μέθοδοι εκμάθησης λεξικών με συγκεκριμένη δομή. Ειδικότερα στη δεύτερη κατηγορία εντάσσονται οι αλγόριθμοι της κατηγορίας K-means, στους οποίους ανήκει ο αλγόριθμος k-SVD. Οι K-means αλγόριθμοι, όπως προτάθηκαν από τους Schmid-Saugeon και Zakhor, βελτιστοποιούν ένα λεξικό με βάση ένα σύνολο από τμήματα (patches), αρχικά ομαδοποιώντας πρότυπα (patterns) έτσι ώστε η απόσταση με κάποιο άτομο να είναι η ελάχιστη και στη συνέχεια ενημερώνοντας το άτομο, ώστε η συνολική απόσταση στην ομάδα των προτύπων να είναι ελάχιστη. Η έμμεση υπόθεση εδώ είναι ότι το κάθε τμήμα μπορεί να αναπαρασταθεί με ένα μόνο άτομο και με ένα συντελεστή ίσο με ένα, που περιορίζει τη διαδικασία μείωσης διαστάσεων σε K άτομα. Γενίκευση των αλγορίθμων K-means είναι ο k-SVD με τη διαφορά ότι κάθε τμήμα μπορεί να αναπαρασταθεί με περισσότερα άτομα και με διαφορετικά βάρη. Θα επανέλθουμε αναλυτικότερα στον k-SVD σε επόμενο κεφάλαιο.

Το Dictionary Learning έχει εφαρμογή και έχει προσφέρει σημαντική βελτίωση απόδοσης σε τομείς που χειρίζονται δεδομένα μεγάλων διαστάσεων, όπως εικόνα, ήχος, βίντεο, αλλά και στην Ιατρική. Όσον αφορά τον τομέα της Ιατρικής, που σχετίζεται με την παρούσα Πτυχιακή Εργασία, το Dictionary Learning έχει μια ενδιαφέρουσα δυνατότητα να φανερώνει στατιστικά που ήταν a priori άγνωστα σε σήματα που αποθηκεύονται από διαφορετικά συστήματα μετρήσεων. Τέτοια σήματα είναι για παράδειγμα το ηλεκτροεγκεφαλογράφημα, το ηλεκτροκαρδιογράφημα, η μαγνητική τομογραφία, η ψηφιακή μαγνητική τομογραφία και η τομογραφία υπερήχων, όπου το παρατηρούμενο σήμα προκαλείται από διαφορετικές φυσικές αιτίες.

1.2 Παραγοντοποίηση πινάκων

Η παραγοντοποίηση (factorization) ενός πίνακα είναι η ανάλυση του σε γινόμενο δύο ή περισσότερων πινάκων σύμφωνα με κάποια λογική που εξυπηρετεί την επίλυση ενός προβλήματος. Στους αλγορίθμους με τους οποίους ασχοληθήκαμε απαραίτητη ήταν η χρήση των SVD, QR και Cholesky.

1.2.1 Ανάλυση Ιδιόμορφων Τιμών (SVD)

Η Ανάλυση Ιδιόμορφων Τιμών (Singular Value Decomposition) εκφράζει την παραγοντοποίηση ενός $m \times n$ πίνακα A στη μορφή UDV^T, όπου U και V ορθογώνιοι $m \times m$ και $n \times n$ αντίστοιχα και ο D ένας $m \times n$ ορθογώνια διαγώνιος (rectangular diagonal) πίνακας, ενώ V^T ο ανάστροφος του πίνακα V. Εναλλακτικά, θα μπορούσαμε να θεωρήσουμε ότι ο D είναι διαγώνιος και η μορφή του μετασχηματισμού θα γινόταν $A = U\begin{pmatrix} D & 0 \\ 0 & 0 \end{pmatrix}V^T$, έτσι ώστε να είναι και σχηματικά ξεκάθαρος ο όρος «ορθογώνια διαγώνιος». Ο πίνακας D περιέχει τις ιδιόμορφες τιμές του που είναι μοναδικά καθορισμένες. Αντίθετα, οι U και V δε χρειάζεται να είναι μοναδικοί στη γενική περίπτωση.

Ορθογώνιος πίνακας είναι ο τετραγωνικός πίνακας του οποίου οι γραμμές και οι στήλες είναι ορθογώνια μοναδιαία διανύσματα. Αν Q ορθογώνιος πίνακας, τότε ισχύει $QQ^T = Q^T Q = I$, όπου I ο ταυτοτικός πίνακας.

Ένας πίνακας $m \times n$ με $m \neq n$ καλείται ορθογώνια διαγώνιος αν, για τη μικρότερη εκ των δύο διαστάσεων του $r = \min\{m, n\}$, το $r \times r$ άνω αριστερό τμήμα του είναι διαγώνιος πίνακας και όλα τα υπόλοιπα στοιχεία έξω από αυτό το τμήμα ισούνται με μηδέν. Τα μη μηδενικά στοιχεία καλούνται ιδιόμορφές ή ιδιάζουσες τιμές (singular values).

Διαγώνιος είναι ο πίνακας του οποίου όλα τα στοιχεία που δεν ανήκουν στην κύρια διαγώνιο είναι ίσα με το μηδέν.

1.2.2 Ανάλυση QR

Η παραγοντοποίηση QR είναι η ανάλυση ενός πίνακα A σε ένα γινόμενο QR, όπου Q είναι ορθογώνιος πίνακας και ο R είναι άνω τριγωνικός πίνακας (ή δεξιά τριγωνικός). Προσφέρει έναν εναλλακτικό τρόπο για την επίλυση συστημάτων της μορφής $Ax = b$ χωρίς την αντιστροφή του A. Αφού ο Q είναι ορθογώνιος, τότε ισχύει ότι $Q^T Q = I$, οπότε η $Ax = b$ μπορεί ισοδύναμα να μετατραπεί στη σχέση $Rx = Q^T b$, που επιλύεται ευκολότερα, αφού ο R είναι άνω τριγωνικός πίνακας.

Στη γενική περίπτωση, μπορούμε να παραγοντοποιήσουμε έναν οποιονδήποτε πίνακα A διαστάσεων $m \times n$ με $m \geq n$ ως γινόμενο ενός $m \times m$ ορθογώνιου πίνακα Q και ενός $m \times n$ άνω τριγωνικού πίνακα R, υπό την έννοια ότι το άνω $n \times n$ τμήμα του είναι ο συνήθης άνω τριγωνικός, ενώ τα υπόλοιπα στοιχεία του είναι μηδενικά. Ο πίνακας A δεν είναι εν γένει μοναδικός, αλλά αν ο A είναι μέγιστης τάξης (full rank), υπάρχει μοναδικός πίνακας R με όλα τα στοιχεία της διαγωνίου θετικά. Αν ο A είναι τετραγωνικός, και ο Q είναι μοναδικός.

1.2.3 Ανάλυση Cholesky

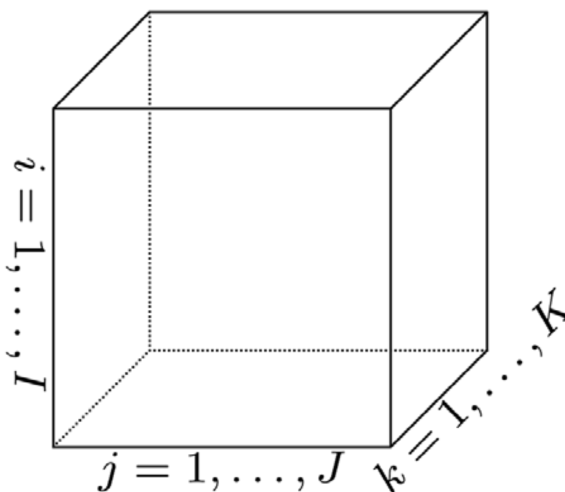
Η παραγοντοποίηση Cholesky είναι η ανάλυση ενός συμμετρικού, θετικά καθορισμένου πίνακα A σε ένα γινόμενο ενός κάτω τριγωνικού πίνακα L και του ανάστροφού του L^T στη μορφή $A = LL^T$. Ο L λέγεται παράγοντας Cholesky του A και μπορεί να ερμηνευτεί σαν μια γενικευμένη τετραγωνική ρίζα του A.

Ένας κάτω τριγωνικός πίνακας L είναι εκείνος του οποίου όλα τα στοιχεία του που βρίσκονται πάνω από την κύρια διαγώνιο του ισούνται με το μηδέν, δηλαδή έχει τη μορφή $L = \begin{pmatrix} a_{11} & 0 & \dots & 0 \\ a_{21} & a_{22} & \dots & 0 \\ \vdots & \vdots & \ddots & 0 \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}$.

1.3 Παραγοντοποίση τανυστών

1.3.1 Εισαγωγή στους τανυστές

Οι τανυστές είναι γεωμετρικά αντικείμενα που μπορούν να θεωρηθούν ως γενικευμένα διανύσματα. Ουσιαστικά πρόκειται για πολυδιάστατους πίνακες. Ειδικότερα, ένας τανυστής N-οστής τάξης είναι ένα στοιχείο του γινομένου τανυστών N διανυσματικών χώρων με καθέναν από αυτούς να έχει το δικό του σύστημα συντεταγμένων. Η έννοια του τανυστή, όπως την αναφέρουμε εδώ, δεν πρέπει να συγχέεται με την έννοια του τανυστή στη φυσική και τη μηχανική. Ένας πρώτης τάξης τανυστής είναι ένα διάνυσμα, ένας δεύτερης τάξης τανυστής είναι ένας πίνακας, ενώ οι τανυστές τρίτης τάξης και πάνω λέγονται υψηλότερης τάξεως τανυστές. Εν γένει, ένας n-οστής τάξης τανυστής σε m-διάστατο χώρο έχει n δείκτες και m^n στοιχεία. Κάθε δείκτης του τανυστή εκτείνεται στο πλήθος των διαστάσεων του χώρου

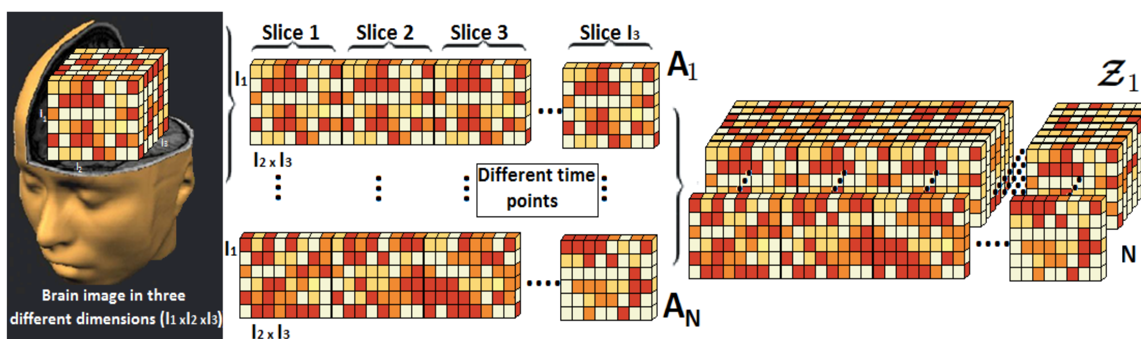


Εικόνα 1: Ένας τανυστής τρίτης τάξης

Υπάρχουν αρκετοί λόγοι που η χρήση τανυστών εντείνεται όλο και περισσότερο. Οι τανυστές συχνά προσφέρουν μια πιο φυσική αναπαράσταση μεγάλων δεδομένων, για παράδειγμα ένος βίντεο που αποτελείται από συσχετιζόμενες εικόνες με την πάροδο του χρόνου. Παράλληλα, η κατανόηση της πλειογραμμικής άλγεβρας (multilinear algebra) βελτιώνεται ραγδαία, ειδικότερα σε διάφορους τύπους παραγοντοποιήσεων, το οποίο με τη σειρά του μας βοηθάει να ταυτοποιήσουμε νέες πιθανές εφαρμογές, για παράδειγμα ανάλυση πολυδιάστατων συνιστωσών (multiway component analysis). Τέλος, η αύξηση της υπολογιστικής ισχύος επιτρέπει τη χρήση των τανυστών σε περισσότερα προβλήματα, παρά τη μεγάλη υπολογιστική τους πολυπλοκότητα.

1.3.2 Παραγοντοποίηση τανυστών fMRI

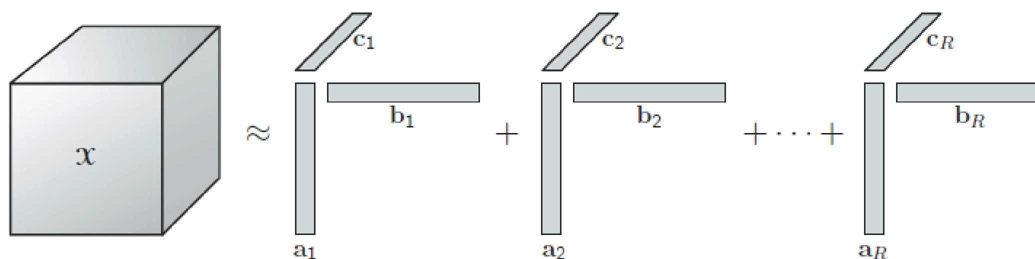
Στην αποθήκευση των δεδομένων του fMRI η χρησιμότητα των τανυστών είναι σημαντική. Ο εντοπισμός των ενεργοποιημένων περιοχών του εγκεφάλου είναι ένα απαιτητικό θέμα Τυφλού Διαχωρισμου Πηγής (Blind Source Separation), αφού οι πηγές αποτελούνται από έναν συνδυασμό χωρικών χαρτών (spatial maps) που δείχνουν τις ενεργοποιημένες περιοχές και χρονοδιαγράμματα (time courses) που δείχνουν το μοτίβο ενεργοποίησης τους. Μέχρι πρόσφατα οι πολυμεταβλητές διγραμμικές μέθοδοι (multivariate bi-linear methods), όπως για παράδειγμα αυτές που στηρίζονται σε πίνακες, που βασίζονται στη σύνδεση διαφορετικών τρόπων ήταν ό,τι πιο σύγχρονο στον Τυφλό Διαχωρισμό Πηγής σε fMRI. Ωστόσο, αυτές οι μέθοδοι εξ ορισμού αδυνατούν να εκμεταλλευτούν την πολυδιάστατη φύση των δεδομένων. Αντ' αυτών μπορούν να χρησιμοποιηθούν πολυγραμμικά (τανυστικά) μοντέλα, τα οποία, σε γενικές γραμμές, α) παράγουν μοναδικές αναπαραστάσεις, β) βελτιώνουν τη δυνατότητα εξαγωγής χωροχρονικών πεδίων ενδιαφέροντος, γ) διευκολύνουν τις με νόημα νευροφυσιολογικές ερμηνείες.



Εικόνα 2: Οι εικόνες του εγκεφάλου ξεδιπλώνονται σε πίνακες και σωρεύονται σε τανυστές

Η τανυστική μέθοδος που χρησιμοποιμοποιείται συνήθως για την ανάλυση των δεδομένων που προέρχονται από fMRI και αξιοποιείται και στον αλγόριθμο PARAFAC2 είναι η Κανονική Ανάλυση (Canonical Decomposition).

1.3.3 Κανονική Ανάλυση (CANDECOMP)

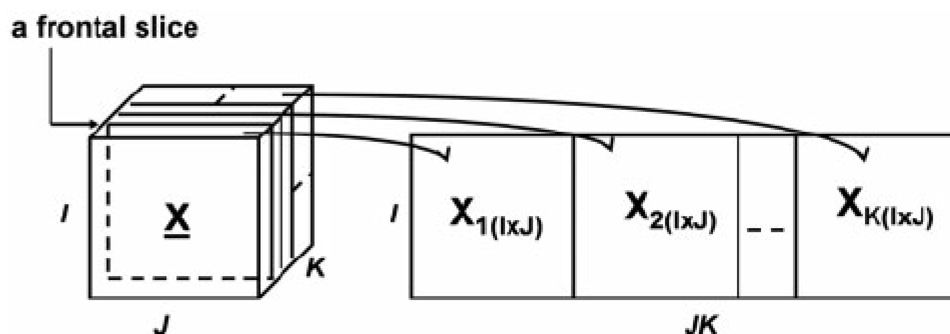


Εικόνα 3: Ανάλυση CANDECOMP σε πίνακα τριών διαστάσεων

Η Κανονική Ανάλυση, που εναλλακτικά ονομάζεται και Παράλληλη Παραγοντοποίηση (Parallel Factorization - PARAFAC), προσεγγίζει έναν τανυστή από δεδομένα fMRI, έστω τρίτης τάξης $T \in \mathbb{R}^{I_1 \times I_2 \times I_3}$, ως άθροισμα R πρώτης τάξης τανυστών, $\mathrm{T} = \sum_{r=1}^{R} a_r \circ b_r \circ c_r + \mathrm{E}$, όπου το $\circ$ συμβολίζει το εξωτερικό γινόμενο διανυσμάτων και το E αντιπροσωπεύει το σφάλμα του τανυστικού μοντέλου.

Ισοδύναμα, θεωρώντας πάλι το παράδειγμα του τανυστή τρίτης τάξης, προσεγγίζεται σε μορφή πινάκων και ως $\mathrm{T}_{(1)} = \mathrm{A}(C \odot B)^T + E_{(1)}$, $\mathrm{T}_{(2)} = B(C \odot A)^T + E_{(2)}$ και $\mathrm{T}_{(3)} = C(B \odot A)^T + E_{(3)}$, όπου με $\odot$ συμβολίζουμε το γινόμενο Khatri-Rao.

Το μοντέλο τριών δρόμων μερικές φορές γράφεται από την άποψη των εμπρόσθιων κομματιών (frontal slice) του τανυστή T στη μορφή $\mathrm{T}_k = BD_k A^T + E_k$, όπου $A = [a_1, a_2, \ldots, a_k]$ είναι ένας πίνακας που περιέχει τα βάρη των R συνιστωσών των I_1 voxels (χωρικοί χάρτες) και οι πίνακες B, C ορίζονται παρόμοια και περιέχουν τα χρονοδιαγράμματα και τα επίπεδα ενεργοποίησης αντίστοιχα. Ο D_k είναι διαγώνιος πίνακας που στην κύρια διαγώνιο του περιέχει τα στοιχεία της γραμμής k του πίνακα C. Αυτή η μέθοδος δύσκολα επεκτείνεται πάνω από τις τρεις διαστάσεις.



Εικόνα 4: Εμπρόσθια κομμάτια τανυστή

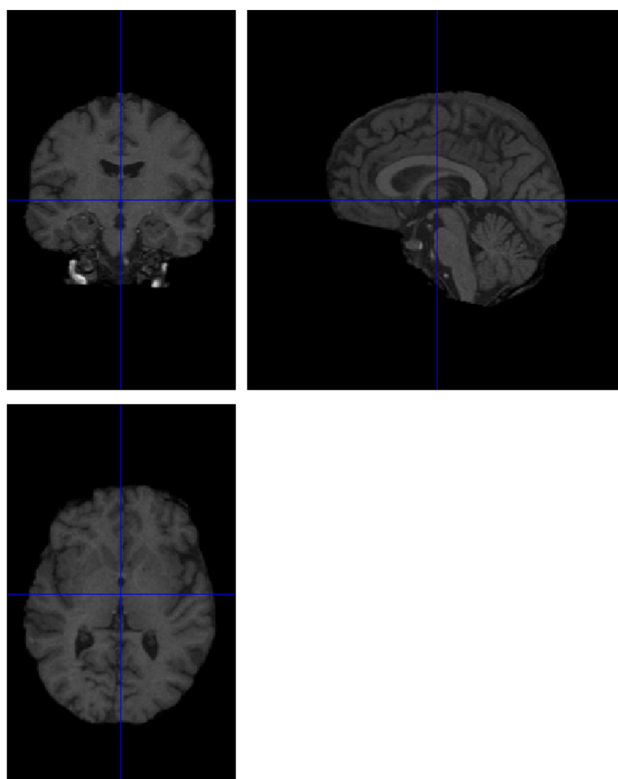
Το voxel είναι μονάδα γραφικής πληροφορίας που ορίζει ένα σημείο σε τρισδιάστατο χώρο. Είναι το αντίστοιχο του εικονοστοιχείου (pixel), αλλά έχει μία παραπάνω διάσταση. Ας σκεφτούμε ότι είναι ένας κύβος που κάθε σημείο στην εξωτερική πλευρά εκφράζεται με τις x,y συντεταγμένες και η συντεταγμένη z ορίζει τη θέση του μέσα στον κύβο.

Τα πλεονεκτήματα της ανάλυσης CANDECOMP, εκτός από την απλότητά της, είναι το γεγονός ότι είναι μοναδική (στη μετάθεση και την κλιμάκωση) κάτω από ήπιες συνθήκες. Έχει παρατηρηθεί ότι η CANDECOMP με fMRI δεδομένα είναι ισχυρή στις επικαλύψεις (χωρικές ή/και τοπικές). Από την άλλη πλευρά, το αποτέλεσμα της CANDECOMP συνδέεται σε μεγάλο βαθμό με τη σωστή εκτίμηση των R συνιστωσών.

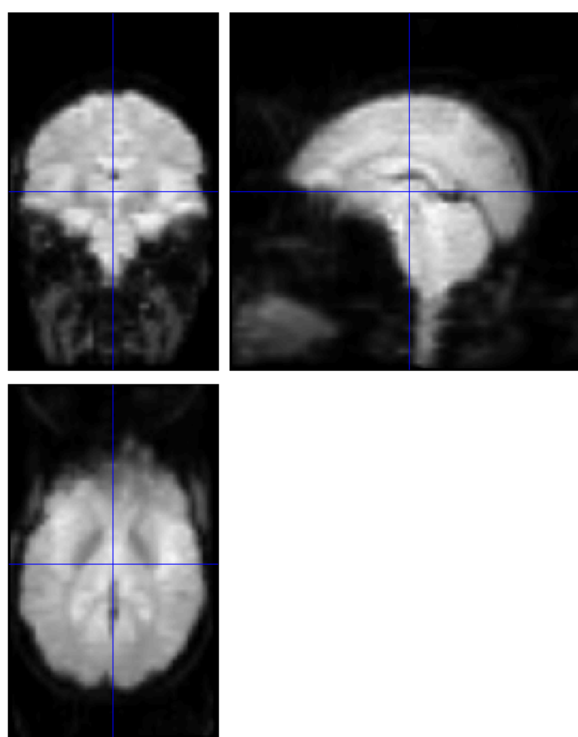
1.4 Λειτουργική Απεικόνιση Μαγνητικού Συντονισμού

Η Λειτουργική Απεικόνιση με Μαγνητικό Συντονισμό (fMRI) είναι μια από τις πιο γνωστές μη επεμβατικές μεθόδους εγκεφαλικής απεικόνισης. Χρησιμοποιείται τόσο για έρευνες σχετικά με τις λειτουργίες του εγκεφάλου, όσο για απεικόνιση εγκεφάλων ασθενών ή ατόμων με ψυχικές διαταραχές.

Ο όρος «λειτουργική απεικόνιση» αναφέρεται στη δυνατότητα εντοπισμού και απεικόνισης περιοχών του εγκεφάλου που ενεργοποιούνται από διάφορα είδη ερεθισμάτων ή δραστηριοτήτων. Οι ενεργοποιήσεις αυτές γίνονται αντιληπτές χάρη σε μικρές, αλλά ανιχνεύσιμες μεταβολές στην μαγνητική ευαισθησία του εγκεφάλου. Η απεικόνιση αυτή διαφέρει σημαντικά από την «ανατομική απεικόνιση» του εγκεφάλου, στην οποία προβάλλεται η δομή του εγκεφάλου και όχι τα επίπεδα ενεργοποίησης των περιοχών κάθε δεδομένη στιγμή. Στις εικόνες 5 και 6 φαίνεται η διαφορετική απεικόνιση του ίδιου εγκεφάλου με τις δύο διαφορετικές μεθόδους. Φυσικά, μπορούμε να χρησιμοποιήσουμε το fMRI τόσο για λειτουργική απεικόνιση, όσο και για ανατομική, κάνοντας κάθε φορά τις κατάλληλες μετρήσεις.



Εικόνα 5: Ανατομική εικόνα εγκεφάλου

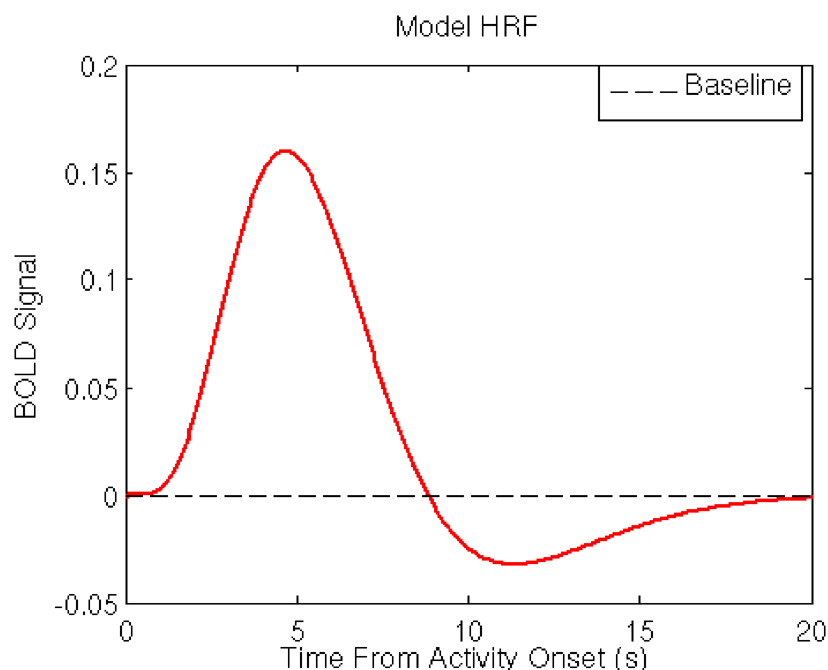


Εικόνα 6: Λειτουργική εικόνα εγκεφάλου

Η βασική μορφή του fMRI χρησιμοποιεί τις εξαρτώμενες από το επιπέδο οξυγόνωσης του αίματος (Blood-Oxygen-Level Dependent) διακυμάνσεις. Είναι μια μορφή εξειδικευμένης σάρωσης, που χρησιμοποιείται για να αντιστοιχίσει τη νευρωνική δραστηριότητα του εγκεφάλου (ή της σπονδυλικής στήλης) απεικονίζοντας τη μεταβολή της ροής του αίματος που σχετίζεται με την ενέργεια που κατανάλωσαν τα κύτταρα του εγκεφάλου (αιμοδυναμική απόκριση). Αξίζει να σημειωθεί ότι από το 1990 και έπειτα, το

fMRI είναι από τις πιο δημοφιλείς μεθόδους έρευνας και χαρτογράφησης του εγκεφάλου, καθώς δεν απαιτεί από το υπό εξέταση άτομο να υποβληθεί σε παρεμβάσεις όπως ένεση, χειρουργείο, κατάποση ουσιών, έκθεση σε ακτινοβολία κλπ.

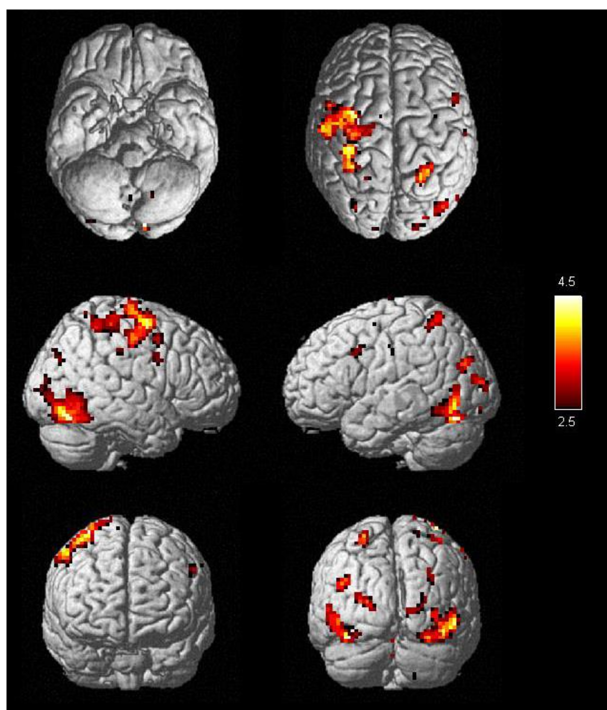
Οι νευρώνες χρειάζονται ενέργεια για να λειτουργήσουν. Γι' αυτό, όταν θέλουμε να κάνουμε κάποια συγκεκριμένη ενέργεια, όπως λόγου χάρη να μιλήσουμε, να πιάσουμε ένα αντικείμενο ή να ψάξουμε κάτι στο οπτικό μας πεδίο, οι νευρώνες που βρίσκονται στην περιοχή του εγκεφάλου που είναι υπεύθυνη για αυτού του είδους τις ενέργειες, ενεργοποιούνται καταναλώνοντας οξυγόνο. Συνέπεια αυτής της κατανάλωσης οξυγόνου είναι μια αύξηση στη ροή του αίματος στις περιοχές αυξημένης νευρωνικής δραστηριότητας, που συμβαίνει με καθυστέρηση περίπου 3-10 δευτερόλεπτα μετά τη διέγερση και οι συνέπειες της μοντελοποιούνται από τη συνάρτηση αιμοδυναμικής απόκρισης (haemodynamic response function). Η αιμοδυναμική απόκριση γίνεται μέγιστη μετά τα 4-5 δευτερόλεπτα και στη συνέχεια επιστρέφει στην φυσιολογική τιμή (συχνά ξεπερνώντας την ελαφρώς). Η απόκριση αυτή έχει ως συνέπεια τοπικές αλλαγές στις σχετικές συγκεντρώσεις οξυαιμοσφαιρίνης και δεοξυαιμοσφαιρίνης, και αλλαγή του όγκου του αίματος στην περιοχή μαζί με αυτή την αύξηση ροής. Η HRF ποικίλλει μεταξύ διαφορετικών ατόμων καθώς και μεταξύ διαφορετικών περιοχών του εγκεφάλου του ίδιου ατόμου.



Εικόνα 7: Γραφική παράσταση μιας τυπικής HRF

Η αιμοσφαιρίνη είναι διαμαγνητικό υλικό όταν είναι οξυγονωμένη, ενώ είναι παραμαγνητικό υλικό όταν αποξυγονωθεί. Ένα παραμαγνητικό υλικό έχει την ιδιότητα πως όταν τοποθετηθεί σε ένα ισχυρό μαγνητικό πεδίο, τα άτομα στο υλικό προσπαθούν να προσανατολιστούν με το πεδίο, αυξάνοντας την ισχύ του πεδίου. Με άλλα λόγια, το παραμαγνητικό υλικό γίνεται ένας μαγνήτης όσο το πεδίο είναι παρόν. Συνεπώς, υπάρχει μεταβολή στη μαγνήτιση ανάμεσα στο πλούσιο και στο φτωχό σε οξυγόνο αίμα. Το μη οξυγονωμένο αίμα είναι πιο παραμαγνητικό από το οξυγονωμένο. Η μεταβολή αυτή χρησιμοποιείται ως το βασικό κριτήριο στη διαδικασία του fMRI. Οι μετρήσεις που γίνονται σε αυτή τη μεταβολή είναι αρκετά ευάλωτες σε θόρυβο από διάφορες πηγές, και γι' αυτόν τον λόγο είναι αναγκαία η χρήση στατιστικών μεθόδων για την εύρεση του πραγματικού σήματος. Το τελικό αποτέλεσμα της ενεργοποίησης προβάλλεται πάνω σε ένα μοντέλο εγκεφάλου και απεικονίζεται συνήθως με έναν

κώδικα χρώματος όπου οι περισσότερο ενεργοποιημένες περιοχές είναι κοντά στο κίτρινο και οι λιγότερο κοντά στο κόκκινο.



Εικόνα 8: Σημεία ενεργοποίησης εγκεφάλου

2. ΠΑΡΑΛΛΗΛΟΣ ΥΠΟΛΟΓΙΣΜΟΣ

2.1 Το χρονικό του παράλληλου υπολογισμού

Υπήρξε μια περίοδος στο όχι και τόσο μακρινό παρελθόν όπου ο παράλληλος υπολογισμός (parallel computing) θεωρούταν μια εξωτική επιδίωξη. Η αντίληψη αυτή άλλαξε σε βάθος τα τελευταία χρόνια. Ο κόσμος των υπολογιστών έφτασε στο σημείο όπου σχεδόν κάθε φιλόδοξος προγραμματιστής χρειάζεται εκπαίδευση στον παράλληλο υπολογισμό για να είναι πλήρως αποδοτικός στην επιστήμη των υπολογιστών.

Τα τελευταία χρόνια πολλά προκάλεσαν τη στροφή της βιομηχανίας των υπολογιστών στον παράλληλο υπολογισμό. Από τις αρχές της τρέχουσας δεκαετίας σχεδόν όλοι οι καταναλωτές χρησιμοποιούν συσκευές με πολυπύρηνους επεξεργαστές. Από τους πρώτους διπύρηνους χαμηλών προδιαγραφών υπολογιστές μέχρι τους οκταπύρηνους και δεκαεξαπύρηνους σταθμούς εργασίας, η εξάπλωση του παράλληλου υπολογισμού είναι τόσο ευρεία που δεν μπορεί να περιορίζεται μόνο σε υπερυπολογιστές και μεγάλα υπολογιστικά συστήματα.

Για περισσότερα από 30 χρόνια, μία από τις βασικότερες μεθόδους βελτίωσης της απόδοσης των προσωπικών υπολογιστών που κυκλοφορούσαν στην αγορά ήταν η αύξηση της ταχύτητας στην οποίο λειτουργεί το ρολόι του επεξεργαστή. Ξεκινώντας από τους πρώτους προσωπικούς υπολογιστές στις αρχές της δεκαετίας του 1980, οι επεξεργαστές που βγήκαν στην αγορά χρησιμοποιούσαν ρολόγια με ταχύτητα περίπου 1 MHz. Σήμερα, οι περισσότεροι υπολογιστές χρησιμοποιούν επεξεργαστές των οποίων τα ρολόγια έχουν ταχύτητα μεταξύ 1GHz και 4GHz, που αντιστοιχεί σε αύξηση μεγαλύτερη των 1000 φορών. Παρόλο που η αύξηση της ταχύτητας των ρολογιών της CPU δεν είναι ο μόνος τρόπος αύξησης της υπολογιστικής ισχύος, ήταν πάντα ένας αξιόπιστος τρόπος για βελτίωση της απόδοσης.

Τα τελευταία χρόνια, ωστόσο, οι κατασκευαστές αναγκάστηκαν να ψάξουν για εναλλακτικές λύσεις για να αυξήσουν την υπολογιστική ισχύ. Εξαιτίας διάφορων βασικών περιορισμών στην κατασκευή ολοκληρωμένων κυκλωμάτων, δεν είναι πλέον εφικτό να βασιστούν στη διαρκή αύξηση της ταχύτητας του ρολογιού του επεξεργαστή ως μέσο για εξαγωγή επιπρόσθετης ενέργειας από υπάρχουσες αρχιτεκτονικές. Οι περιορισμοί ισχύος και θερμοκρασίας και η προσέγγιση του φυσικού ορίου του μεγέθους των τρανζίστορ οδήγησαν τους ερευνητές και τους κατασκευαστές να ψάξουν για άλλες λύσεις.

Αντίθετα με τους προσωπικούς υπολογιστές, στο χώρο των υπερυπολογιστών έχουν για δεκαετίες τεράστιο όφελος στην απόδοση με παρόμοιους τρόπους. Η απόδοση ενός επεξεργαστή που χρησιμοποιείται σε υπερυπολογιστές έχει εκτοξευθεί, παρόμοια με την απόδοση του επεξεργαστή σε προσωπικούς υπολογιστές. Παρόλα αυτά, επιπρόσθετα με τη δραματική βελτίωση στην απόδοση ενός μονού επεξεργαστή, οι κατασκευαστές υπερυπολογιστών έκαναν τεράστια άλματα στη βελτίωση της απόδοσης των υπερυπολογιστών αυξάνοντας διαρκώς το πλήθος των επεξεργαστών. Δεν είναι ασυνήθιστο για τους ταχύτερους υπερυπολογιστές να έχουν δεκάδες ή εκατοντάδες χιλιάδες πυρήνες υπολογιστών που δουλεύουν παράλληλα.

Φτάνοντας στο 2005, οι κατασκευαστές που είχαν να αντιμετωπίσουν μια πολύ ανταγωνιστική αγορά και λίγες εναλλακτικές λύσεις, ξεκίνησαν να υλοποιούν την ιδέα των πολυπύρηνων υπολογιστών στους προσωπικούς υπολογιστές, ξεκινώντας με διπύρηνους. Στα επόμενα χρόνια ξεκίνησαν να προσφέρονται υπολογιστές με τέσσερις, έξι και οκτώ πυρήνες, ενώ υπάρχουν και προσωπικοί υπολογιστές με ακόμα περισσότερους, όπως δώδεκα και δεκαέξι πυρήνες. Σήμερα είναι σχεδόν απίθανο να βρει κάποιος μονοπύρηνο υπολογιστή, επιβεβαιώνοντας ότι η επιλογή του παράλληλου υπολογισμού έχει έρθει για τα καλά.

2.2 Η ανάπτυξη των υπολογισμών γενικού σκοπού με κάρτα γραφικών

Στην προηγούμενη ενότητα αναφερθήκαμε στην πρόοδο των κεντρικών μονάδων επεξεργασίας σε σχέση τόσο με την ταχύτητα των ρολογιών όσο και με το πλήθος των πυρήνων. Παράλληλα, όμως, η επανάσταση στη χρήση των καρτών γραφικών μάς οδήγησε στην χρήση τους για υπολογισμούς γενικού σκοπού (general-purpose computations).

Στα τέλη της δεκαετίας του 1980 και στις αρχές της δεκαετίας του 1990, η ανάπτυξη της δημοτικότητας λειτουργικών συστημάτων που χρησιμοποιούν γραφικό περιβάλλον, όπως τα Microsoft Windows, βοήθησε στη δημιουργία μιας αγοράς για έναν νέο τύπο επεξεργαστή. Στις αρχές της δεκατίας του ’90, οι χρήστες ξεκίνησαν να αγοράζουν επιταχυντές δισδιάστατων εικόνων (2D display accelerators) για τους προσωπικούς τους υπολογιστές που χρησίμευαν στο να βοηθήσουν την εικόνα και την ευχρηστία των λειτουργικών συστημάτων με γραφικά.

Περίπου τον ίδιο καιρό, μια εταιρεία ονόματι Silicon Graphics, πέρασε τη δεκαετία του 1980 επιχειρώντας να καταστήσει δημοφιλή τη χρήση τρισδιάστατων γραφικών σε μια πληθώρα αγορών, συμπεριλαμβανομένων της κυβέρνησης, της εθνικής άμυνας, της επιστημονικής και της τεχνολογικής απεικόνισης, και να παρέχουν εργαλεία που δημιουργούν εντυπωσιακά εφφέ στο σινεμά. Το 1992, η Silicon Graphics έβγαλε στην αγορά την πρώτη διεπαφή προγραμματισμού με το υλικό (hardware) της, τη βιβλιοθήκη OpenGL. Σκοπός ήταν να χρησιμοποιηθεί σαν ανεξάρτητη από την πλατφόρμα μέθοδος δημιουργίας τρισδιάστατων γραφικών.

Στα μέσα της δεκαετίας του 1990, οι απαιτήσεις για να βγουν στην αγορά εφαρμογές με τρισδιάστατα γραφικά κλιμακώθηκαν ραγδαία, δημιουργώντας τις προϋποθέσεις για δύο σημαντικές εξελίξεις. Πρώτα, η κυκλοφορία επιβλητικών παιχνιδιών πρώτου προσώπου πυροδότησε την ανάζητηση για σταδιακά πιο ρεαλιστικά τρισδιάστατα περιβάλλοντα για παιχνίδια σε υπολογιστές. Η δημοτικότητα αυτών των παιχνιδιών θα επιταχύνει σημαντικά την εξάπλωση των τρισδιάστατων γραφικών στους προσωπικούς υπολογιστές. Το ίδιο διάστημα, εταιρείες όπως η NVIDIA, η ATI Technologies και η 3dfx Interactive ξεκίνησαν να κυκλοφορούν γραφικούς επιταχυντές αρκετά προσιτούς οικονομικά ώστε να προσελκύσουν την προσοχή.

Εικόνα 9: Duke Nukem 3D (1996), από τα πρώτα παιχνίδια με τρισδιάστατα γραφικά

Η κυκλοφορία της NVIDIA GeForce 256 στα τέλη του 1999 έφερε καινοτομίες. Για πρώτη φορά, οι υπολογισμοί του μετασχηματισμού και του φωτισμού μπορούσαν να

γίνονται κατευθείαν στον επεξεργαστή γραφικών, ενισχύοντας έτσι τη δυνατότητα για καλύτερο οπτικό αποτέλεσμα. Επειδή ο μετασχηματισμός και ο φωτισμός ήταν ήδη αναπόσπαστο κομμάτι της σωλήνωσης γραφικών (graphics pipeline) της OpenGL, η GeForce 256 σηματοδότησε την αρχή της φυσικής εξέλιξης όπου όλο και μεγαλύτερο μέρος της σωλήνωσης γραφικών γίνονται απευθείας στην κάρτα γραφικών.

Από την οπτική γωνία του παράλληλου προγραμματισμού, η NVIDIA κυκλοφόρησε το 2001 τη σειρά GeForce 3 που ήταν κατά πολλούς η μεγαλύτερη τομή στην τεχνολογία των GPU. Η σειρά αυτή ήταν το πρώτο ολοκληρωμένο κύκλωμα που εφάρμοζε το πρότυπο DirectX 8.0. Για πρώτη φορά, οι προγραμματιστές είχαν κάποιον έλεγχο στους υπολογισμούς που θα πραγματοποιούνταν στην GPU.

Η κυκλοφορία καρτών γραφικών που είχαν προγραμματίσιμες σωληνώσεις καλλιέργησε σε αρκετούς ερευνητές την ιδέα να χρησιμοποιήσουν τον επεξεργαστή γραφικών για περισσότερα από μια απλή απόδοση γραφικών με βάση την OpenGL ή το πρότυπο DirectX. Η γενική προσέγγιση στις απαρχές του υπολογισμού με GPU ήταν εξαιρετικά πολύπλοκη. Επειδή οι καθιερωμένες διεπαφές γραφικών, όπως οι OpenGL και DirectX, ήταν το μόνο μέσο αλληλεπίδρασης με την GPU, κάθε προσπάθεια εκτέλεσης αυθόρμητων υπολογισμών στην GPU θα τροποποιείται από τους περιορισμούς του προγραμματισμού στις διεπαφές αυτές. Εξαιτίας αυτού, οι ερευνητές εξευρένησαν τους υπολογισμούς γενικού σκοπού μέσω διεπαφών γραφικών προσπαθώντας να κάνουν τα προβλήματά τους να μοίαζουν με τη συνήθη απόδοση γραφικών.

Ουσιαστικά, οι κάρτες γραφικών στις αρχές της δεκαετίας του 2000 ήταν σχεδιασμένες να παράγουν ένα χρώμα για κάθε εικοστοιχείο χρησιμοποιώντας προγραμματίσιμες αριθμητικές μονάδες που είναι γνωστές ως σκιαστές εικονοστοιχείων (pixel shaders). Εν γένει, ο σκιαστής εικονοστοιχείου χρησιμοποιεί τις (x,y) συντεταγμένες του εικοστοιχείου μαζί με κάποιες επιπρόσθετες πληροφορίες ώστε να συνδυάσει διάφορες εισόδους για να υπολογίσει το τελικό χρώμα. Οι επιπρόσθετες πληροφορίες μπορεί να είναι τα συνεισφέροντα χρώματα, οι συντεταγμένες υφής ή άλλα χαρακτηριστικά. Ωστόσο, επειδή οι αριθμητικές πράξεις που πραγματοποιούνται πάνω στα χρώματα και τις υφές ελέγχονται από τον προγραμματιστή, οι ερευνητές παρατήρησαν ότι αντί για χρώματα, θα μπορούσαμε να έχουμε οποιουδήποτε είδους δεδομένα.

Έτσι, αν οι είσοδοι ήταν άλλα αριθμητικά δεδομένα που δηλώνουν κάτι διαφορετικό από χρώμα, οι προγραμματιστές θα μπορούσαν να προγραμματίσουν τους σκιαστές εικονοστοιχείων ώστε να πραγματοποιούν αυθαίρετους υπολογισμούς σε αυτά τα δεδομένα. Τα αποτελέσματα θα επιστρέφονταν στη GPU σαν το τελικό χρώμα του εικονοστοιχείου, παρόλο που τα χρώματα θα είναι απλά το αποτέλεσμα του οποιουδήποτε υπολογισμού κάνει ο προγραμματιστής. Στην ουσία, η GPU «παραπλανάται» να κάνει διαφορετικούς υπολογισμούς κάνοντας τους να φαίνονται σαν τυπική απόδοση εικόνας. Αυτό το κόλπο ήταν αρκετά ευφυές, αλλά και αρκετά περίπλοκο.

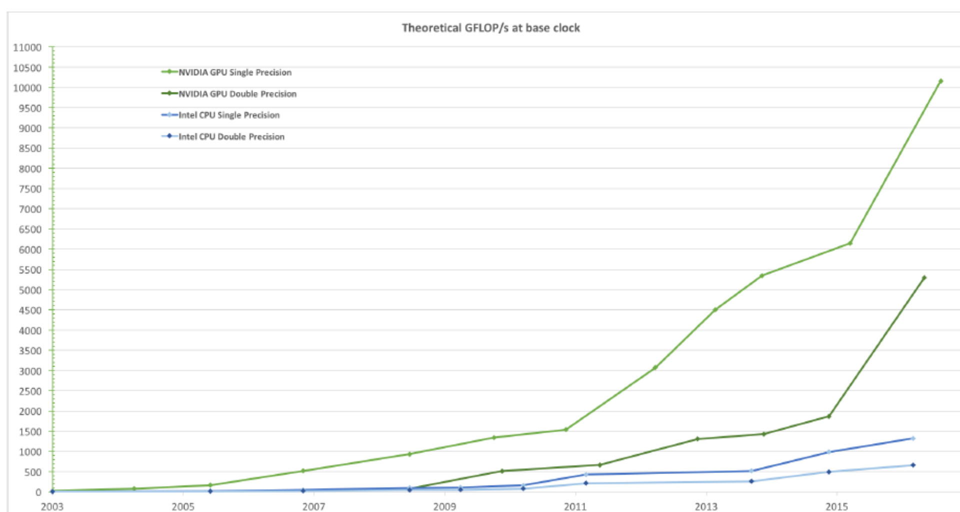
Λόγω της υψηλής διεκπεραιωτικής ικανότητας αριθμητικών πράξεων της GPU, τα αρχικά αποτελέσματα αυτών των πειραμάτων προμήνυαν ένα λαμπρό μέλλον για τους υπολογισμούς με GPU. Ωστόσο, το προγραμματιστικό μοντέλο ήταν ακόμα πολύ περιοριστικό, αφού τα προγράμματα μπορούσαν να πάρουν είσοδο μόνο από λίγα χρώματα και λίγες υφές. Επιπροσθέτως, ήταν σχεδόν αδύνατο να προβλέψει κάποιος πως μια συγκεκριμένα κάρτα γραφικών θα χειριζόταν δεκαδικά δεδομένα (αν μπορούσε να το κάνει αυτό), όποτε οι περισσότεροι επιστημονικοί υπολογισμοί δε μπορούσαν να γίνουν σε κάρτα γραφικών. Τέλος, όταν αναπόφευκτα ένα πρόγραμμα υπολόγιζε εσφαλμένα αποτελέσματα, αποτύγχανε να ολοκληρωθεί ή απλά «κρέμαγε» το σύστημα, δεν υπήρχε κάποια αρκετά καλή μέθοδος αποσφαλμάτωσης (debugging) του κώδικα που εκτελείται στη GPU.

Σαν να μην ήταν αρκετοί οι παραπάνω περιορισμοί, οποιοσδήποτε ήθελε ακόμα να χρησιμοποιήσει τη GPU για υπολογισμούς γενικού σκοπού, θα έπρεπε να μάθει είτε OpenGL είτε DirectX, μιας και παρέμεναν τα μόνα μέσα αλληλεπίδρασης με τη GPU. Αυτό δε σήμαινε μόνο ότι έπρεπε να εισάγουν τα δεδομένα σε κατάλληλη υφή και να εκτελέσουν υπολογισμούς μέσω της OpenGL ή του DirectX, αλλά και να γράψουν τις ίδιες τις υπολογιστικές πράξεις σε ειδικές γλώσσες προγραμματισμού μόνο για γραφικά που είναι γνωστές ως γλώσσες σκίασης (shading languages). Το να ζητάς από τους ερευνητές να αντεπεξέλθουν σε τόσο αυστηρούς περιορισμούς, όπως επίσης και να μάθουν γραφικά υπολογιστών και γλώσσες σκίασης πριν προσπαθήσουν να εκμεταλλευτούν την υπολογιστική ισχύ των καρτών γραφικών τους, αποδείχτηκε πολύ υψηλό εμπόδιο για την ευρεία αποδοχή της χρήσης GPU για γενικής χρήσης υπολογισμούς.

Δεν πέρασαν ούτε πέντε χρόνια μετά την κυκλοφορία της σειράς GeForce 3, όταν οι υπολογισμοί με κάρτα γραφικών ήταν έτοιμοι για το απόγειο τους. Το Νοέμβριο του 2006, η NVIDIA παρουσίασε την GeForce 8800 GTX, την πρώτη κάρτα γραφικών της εταιρείας που υποστήριζε DirectX10. Η GeForce 8800 GTX ήταν επίσης η πρώτη GPU που κατασκευάστηκε με την αρχιτεκτονική CUDA της NVIDIA. Αυτή η αρχιτεκτονική περιελάμβανε αρκετά νέα στοιχεία που σχεδιάστηκαν αυστηρά για υπολογισμούς με GPU και στόχευε να μειώσει τους περιορισμούς που απέτρεπαν προηγούμενες GPU να είναι αρκούντως χρήσιμες για γενικού σκοπού υπολογισμούς.

2.3 Η αρχιτεκτονική CUDA

Η CUDA είναι μια παράλληλη υπολογιστική πλατφόρμα και ένα προγραμματιστικό μοντέλο επινόησης της NVIDIA. Με την CUDA, οι μηχανικοί και οι επιστήμονες μπορούν να χρησιμοποιήσουν την τεράστια υπολογιστική ισχύ της GPU δημιουργώντας μαζικά παράλληλες εφαρμογές και αξιοποιώντας τα χιλιάδες παράλληλα νήματα (threads) της GPU μέσω γλωσσών προγραμματισμού ευρείας χρήσης, όπως οι C, C++, Fortran, Python και Matlab. Οι εφαρμογές που είχαν επιταχυνθεί με GPU έτρεχαν το σειριακό τους τμήμα σε CPU - που είναι βέλτιστη για εφαρμογές ενός νήματος – και το τμήμα τους απαιτούσε μεγάλη παραλληλία σε GPU, που είναι πολύ γρηγορότερη της CPU όταν υπάρχει μεγάλη παραλληλία.



Εικόνα 10: Σύγκριση CPU και GPU σε GFLOPS

Επιστρέφοντας στην ιστορική αναδρομή της προηγούμενης ενότητας, η διαφορά της αρχιτεκτονικής CUDA με τις προηγούμενες που είχαν κυκλοφορήσει είναι ότι περιελάμβανε μια ενιαία σωλήνωση σκίασης (unified shader pipeline), επιτρέποντας σε

καθεμία αριθμητική και λογική μονάδα (ALU) στο κύκλωμα να αξιοποιηθεί από ένα πρόγραμμα που σκοπεύει να πραγματοποιήσει υπολογισμούς γενικού σκοπού. Επειδή η NVIDIA προόριζε αυτή τη νέα οικογένεια GPU ώστε να χρησιμοποιηθούν για γενικού σκοπού υπολογισμούς, οι ALU τους ήταν κατασκευασμένες στα πρότυπα του IEEE για αριθμητικές πράξης κινητής υποδιαστολής μονής ή διπλής ακρίβειας και σχεδιασμένες να χρησιμοποιούν ένα σύνολο εντολών προσαρμοσμένο για υπολογισμούς γενικού σκοπού παρά για γραφικά. Επιπροσθέτως, οι μονάδες εκτέλεσης εντολών στη GPU είχαν αυθαίρετη πρόσβαση για ανάγνωση και εγγραφή στη μνήμη, καθώς επίσης και σε μια κρυφή μνήμη (cache memory) που διαχειρίζεται από το λογισμικό, τη διαμοιραζόμενη μνήμη (shared memory). Όλα αυτά τα χαρακτηριστικά της αρχιτεκτονικής CUDA προστέθηκαν για να δημιουργήσουν μια GPU που θα διακρίνεται στους υπολογισμούς, ενώ παράλληλα θα εκτελεί αποδοτικά τις εργασίες των γραφικών.

Παρόλα αυτά, η προσπάθεια της NVIDIA να παρέχει στους καταναλωτές ένα προϊόν κατάλληλο τόσο για υπολογισμούς όσο και για γραφικά δε θα μπορούσε να σταματήσει απλά στην παραγωγή GPU που ενσωματώνουν την αρχιτεκτονική CUDA. Ανεξάρτητα από το πόσα νέα χαρακτηριστικά προστέθηκαν στο κύκλωμα για να διευκολύνουν τους υπολογισμούς, συνέχιζε να μην υπάρχει τρόπος πρόσβασης σε αυτά τα χαρακτηριστικά χωρίς τη χρήση της OpenGL ή του DirectX. Έτσι, οι χρήστες θα έπρεπε να συνεχίσουν να μεταμφιέζουν τους υπολογισμούς σαν προβλήματα γραφικών και να γράφουν σε γλώσσες σκίασης.

Για να προσεγγίσει όσο το δυνατόν περισσότερους προγραμματιστές, η NVIDIA πρόσθεσε στη C έναν σχετικά μικρό αριθμό λέξεων-κλειδιών ώστε να εκμεταλλευτεί κάποια ειδικά χαρακτηριστικά της αρχιτεκτονικής CUDA. Λίγους μήνες μετά την κυκλοφορία της GeForce 8800 GTX, η NVIDIA δημοσιοποίησε έναν μεταγλωττιστή (compiler) για αυτήν τη γλώσσα, που ονομάστηκε CUDA C. Η CUDA C έγινε η πρώτη γλώσσα ειδικά σχεδιασμένη για να διευκολύνει τους υπολογισμούς γενικού σκοπού στη GPU. Εκτός από το σχεδιασμό γλώσσας για συγγραφή κώδικα στη GPU, η NVIDIA παρέχει έναν εξειδικευμένο οδηγό υλικού (hardware driver) για να αξιοποιήσει τη μαζική υπολογιστική ισχύ της αρχιτεκτονικής CUDA. Συνεπώς, οι χρήστες δεν απαιτείται πια να έχουν γνώσεις από OpenGL ή DirectX, ούτε να υποχρεώνουν τους υπολογισμούς τους να έχουν τη μορφή γραφικών προβλημάτων.

2.4 Τα βασικά της γλώσσας CUDA C/C++

2.4.1 Συνάρτηση πυρήνα CUDA (CUDA kernel)

Μια συνάρτηση πυρήνα ορίζεται χρησιμοποιώντας τον προσδιορισμό __global__ στη δήλωση της συνάρτησης. Στην κλήση της συνάρτησης χρησιμοποιούνται <<<...>>>, εντός των οποίων δίνεται το πλήθος των blocks και των threads ανά block, που προσδιορίζουν την παραλληλία της συνάρτησης. Στην Εικόνα 11 βλέπουμε ένα απλό παράδειγμα κλήσης μιας συνάρτησης πυρήνα που τρέχει με ένα μόνο thread.

```
#include <iostream>

__global__ void kernel( void ) {
}

int main( void ) {
    kernel<<<1,1>>>();
    printf( "Hello, World!\n" );
    return 0;
}
```

Εικόνα 11: Τυπική κλήση συνάρτησης πυρήνα

2.4.2 Ετερογενείς υπολογισμοί

Όπως βλέπουμε στην Εικόνα 11, τα προγράμματα CUDA έχουν τη δυνατότητα ετερογενών υπολογιστών, δηλαδή να συνυπάρχουν στο ίδιο πρόγραμμα κάποια τμήματα κώδικα που τρέχουν στη CPU και άλλα που τρέχουν στη GPU. Χρησιμοποιούμε τον όρο host για ό,τι αφορά τη CPU και τη μνήμη της και τον όρο device για τη GPU και τη μνήμη της.

Εικόνα 12: CPU (Host) και GPU (Device)

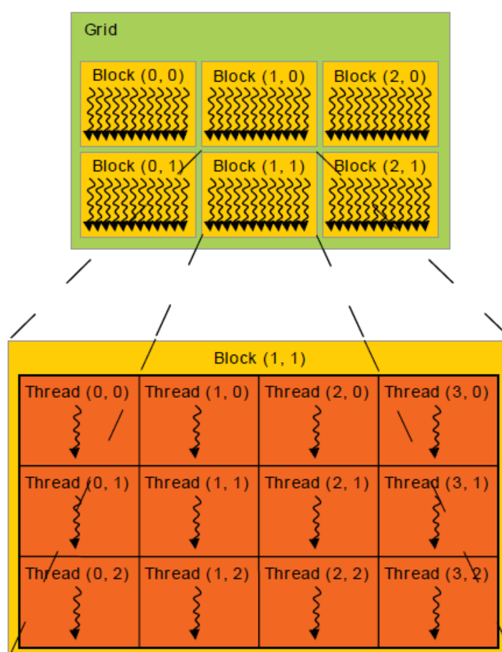
2.4.3 Ιεραρχία threads και blocks

```
__global__ void add(int *a, int *b, int *c) {
        int index = threadIdx.x + blockIdx.x * blockDim.x;
        c[index] = a[index] + b[index];
}
```

Εικόνα 13: Συνάρτηση πυρήνα που χρησιμοποιεί blocks και threads

Εκτελώντας μια συνάρτηση πυρήνα, όπως αυτή της Εικόνας 13, ανοίγει ένα πλέγμα (grid) από threads που εκτελούν παράλληλα τη συνάρτηση, το μέγεθος του οποίου καθορίζεται κατά την κλήση της. Επειδή όλα τα threads στο ίδιο πλέγμα εκτελούν την ίδια συνάρτηση πυρήνα, βασίζονται σε μοναδικές συντεταγμένες για να προσδιορίζουν τους εαυτούς του και να βρίσκουν το κατάλληλο τμήμα των δεδομένων της διεργασίας που θα χειριστούν. Το πλέγμα είναι δισδιάστατο ή τρισδιάστατο, χωρίζεται σε blocks και

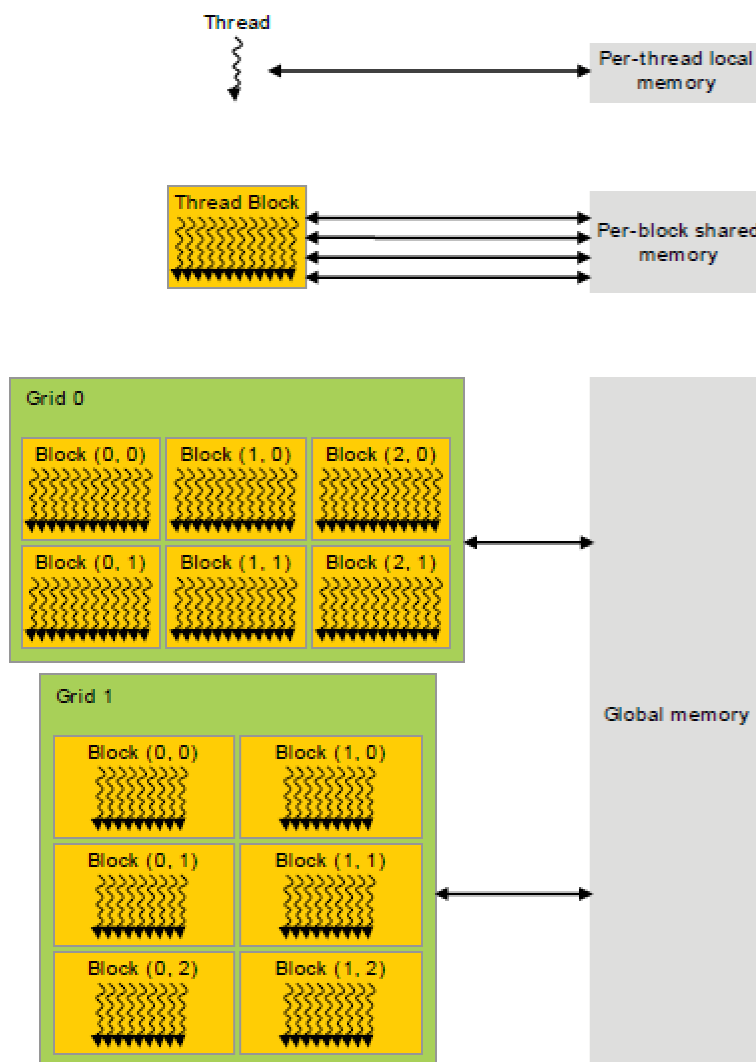
τα blocks περιέχουν threads, όπως φαίνεται και στην Εικόνα 14. Τα threads, δηλαδή, οργανώνονται σε μία ιεραρχία δύο επιπέδων έχοντας μοναδικές συντεταγμένες. Το σύστημα ορίζει δύο διανύσματα τριών διαστάσεων (στη C είναι structs) που μπορούν να προσπελαστούν μόνο στο εσωτερικό μιας συνάρτησης πυρήνα, τα blockIdx και threadIdx. Με αυτές τις δύο μεταβλητές μπορούμε να προσδιορίσουμε οποιοδήποτε thread και να του αναθέσουμε να εκτελέσει κάποια εντολή. Το πλήθος των threads και των blocks διαφέρει από GPU σε GPU. Για να προσδιορίσουμε μια τάξη μεγέθους της παραλληλίας που μπορεί να προσφέρει μία κάρτα γραφικών, θα αναφέρουμε ως παράδειγμα την NVIDIA GTX 480, η οποία μπορεί να υποστηρίξει 1024 threads ανά block, μέγιστο μέγεθος του block 1024x1024x64 στις διαστάσεις x, y, z αντίστοιχα και μέγιστο μέγεθος κάθε διάστασης του πλέγματος των blocks 65535 x 65535 x 65535. Αντιλαμβανόμαστε ότι μια κάρτα γραφικών αυτών των δυνατοτήτων μπορεί να υποστηρίξει μαζική παραλληλία εκατομμυρίων πράξεων.



Εικόνα 14: Πλέγμα από blocks που περιέχουν threads

2.4.4 Ιεραρχία μνήμης

Τα threads στην αρχιτεκτονική CUDA μπορούν να προσπελάσουν δεδομένα από διαφορετικούς χώρους μνήμης, όπως παρατηρούμε στην Εικόνα 15. Κάθε thread έχει την ιδιωτική τοπική μνήμη του. Κάθε block έχει διαμοιραζόμενη μνήμη που είναι ορατή σε όλα τα thread του ίδιου block και έχει την ίδια διάρκεια ζωής με το block. Όλα τα threads έχουν πρόσβαση στην ίδια καθολική μνήμη (global memory). Η πρόσβαση στην καθολική μνήμη είναι πολύ πιο αργή σε σχέση με τη διαμοιραζόμενη μνήμη.



Εικόνα 15: Ιεραρχία μνήμης

2.4.5 Υπολογιστική ικανότητα

Η υπολογιστική ικανότητα (compute capability) αντιπροσωπεύεται από έναν αριθμό έκδοσης, που ταυτοποιεί τα χαρακτηριστικά που υποστηρίζονται από το υλικό της κάρτας γραφικών και χρησιμοποιούνται από τις εφαρμογές κατά το χρόνο εκτέλεσής τους για να αποφασίσουν ποια χαρακτηριστικά του υλικού και ποιες εντολές είναι διαθέσιμες στην παρούσα GPU.

Η υπολογιστική ικανότητα καθορίζεται από έναν αριθμό της μορφής X.Y, όπου ο X είναι ο μείζων αριθμός αναθεώρησης (major revision number) και ο Y είναι ο ελάσσων αριθμός αναθεώρησης (minor revision number).

Οι κάρτες γραφικών με τον ίδιο μείζονα αριθμό αναθεώρησης είναι της ίδιας αρχιτεκτονικής πυρήνα (core architecture). Οι κάρτες με μείζονα αριθμό αναθεώρησης 1 βασίζονται στην αρχιτεκτονική Tesla, εκείνες με το 2 στην αρχιτεκτονική Fermi, εκείνες με το 3 στην αρχιτεκτονική Kepler, ενώ αυτές με το 5 στην αρχιτεκτονική Maxwell. Ο ελάσσων αριθμός αντιστοιχεί σε μια σταδιακά αυξανόμενη βελτίωση στην αρχιτεκτονική πυρήνα, πιθανώς με την προσθήκη κάποιων νέων χαρακτηριστικών.

Δεν πρέπει να συγχέουμε την υπολογιστική ικανότητα μιας GPU με την έκδοση CUDA (π.χ. CUDA 5.5, CUDA 6, CUDA 6.5), η οποία είναι η έκδοση της πλατφόρμας

λογισμικού. Η πλατφόρμα CUDA χρησιμοποιείται από τους προγραμματιστές για να δημιουργήσουν εφαρμογές που θα τρέχουν σε πολλές γενιές αρχιτεκτονικών GPU, συμπεριλαμβανομένων μελοντικών αρχιτεκτονικών GPU που ακόμα δεν έχουν εφευρεθεί.

2.5 Βιβλιοθήκες CUDA

Όπως αναφέρθηκε και στις προηγούμενες ενότητες, η αρχιτεκτονική CUDA δημιουργήθηκε για να εξυπηρετήσει τις ανάγκες των υπολογισμών γενικού σκοπού. Δε θα μπορούσαν, συνεπώς, να μην υπάρχουν βιβλιοθήκες που να διευκολύνουν αυτού του είδους τους υπολογισμούς και να αντικαθιστούν βιβλιοθήκες που είναι μόνο για CPU, όπως οι MKL, FFTW και IPP. Οι βιβλιοθήκες της CUDA παρέχουν εξαιρετικά βελτιστοποιημένες συναρτήσεις που μπορεί να είναι αρκετές φορές ταχύτερες από τις αντίστοιχες συναρτήσεις που είναι μόνο για CPU.

Υπάρχουν βιβλιοθήκες που έχουν βελτιστοποιηθεί για GPU με χρήση στους τομείς της γραμμικής άλγεβρας, της επεξεργασίας σήματος, στην επεξεργασία εικόνας και βίντεο. Για τις ανάγκες της παρούσας εργασίας, χρησιμοποιήθηκαν οι βιβλιοθήκες cuBLAS, cuRAND και cuSOLVER. Η επιτάχυνση που θα προσφέρουν σε ένα πρόγραμμα οι συναρτήσεις αυτών των βιβλιοθηκών εξαρτάται σε μεγάλο βαθμό από τον όγκο των δεδομένων.

2.5.1 Η βιβλιοθήκη cuBLAS

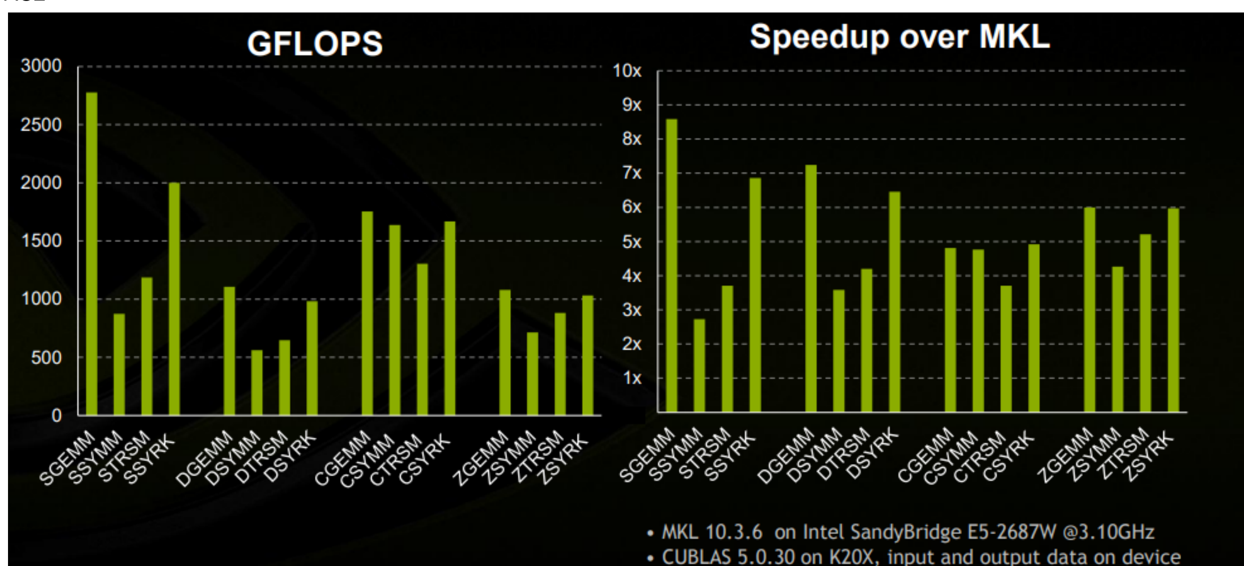
Η βιβλιοθήκη cuBLAS είναι μια υλοποίηση της βιβλιοθήκης BLAS κατάλληλη για την CUDA. Υλοποιεί βασικές πράξεις γραμμικής άλγεβρας. Επιτρέπει στο χρήστη να έχει πρόσβαση στους υπολογιστικούς πόρους μιας κάρτας γραφικών NVIDIA.

Για μέγιστη συμβατότητα με υπάρχοντα περιβάλλοντα Fortran, η βιβλιοθήκη cuBLAS χρησιμοποιεί αποθήκευση πινάκων κατά στήλες. Επειδή η C και η C++ αποθηκεύουν τα δεδομένα κατά σειρές, οι εφαρμογές που γράφονται σε αυτές τις γλώσσες πρέπει να καταφύγουν σε τεχνάσματα για να παρακάμψουν αυτές τις δυσκολίες. Σε αυτές τις γλώσσες, κρατώντας την αρίθμηση από το 0 στους πίνακες, για να βρούμε το στοιχείο (i,j) χρησιμοποιούμε τη μακροεντολή #define IDX2F (i,j,ld) (((j)*(ld)+(i)), όπου ld είναι η κυρίαρχη διάσταση. Στην περίπτωση πινάκων αποθηκευμένων κατά στήλες, στην ld δίνουμε τιμή ίση με το πλήθος των γραμμών του πίνακα.

Στην εργασία μας χρησιμοποιήσαμε το νέο API της βιβλιοθήκης που μπορεί να χρησιμοποιηθεί μέσω του header file “cublas_v2.h”, που περιέχει κάποια διαφορετικά χαρακτηριιστικά από το παλιότερο που μπορεί να χρησιμοποιηθεί μέσω του header file “cublas.h”.

Η βιβλιοθήκη χωρίζει τις πράξεις της σε 3 επίπεδα. Στο πρώτο επίπεδο τοποθετεί τις πράξεις με βαθμωτά μεγέθη και διανύσματα, στο δεύτερο επιπέδο κατατάσσει τις πράξεις μεταξύ πινάκων και διανυσμάτων, ενώ στο τρίτο τις πράξεις μεταξύ πινάκων.

Από τις παρεχόμενες συναρτήσεις, στα πλαίσια της συγγραφής κώδικα, χρησιμοποιήθηκαν η nrm2 για τον υπολογισμό της ευκλείδιας νόρμας ενός πίνακα, η gemm για πολλαπλασιασμούς πινάκων και η geam για αντιστροφή πίνακα.



Εικόνα 16: Σύγκριση cuBLAS με MKL

2.5.2 Η βιβλιοθήκη cuRAND

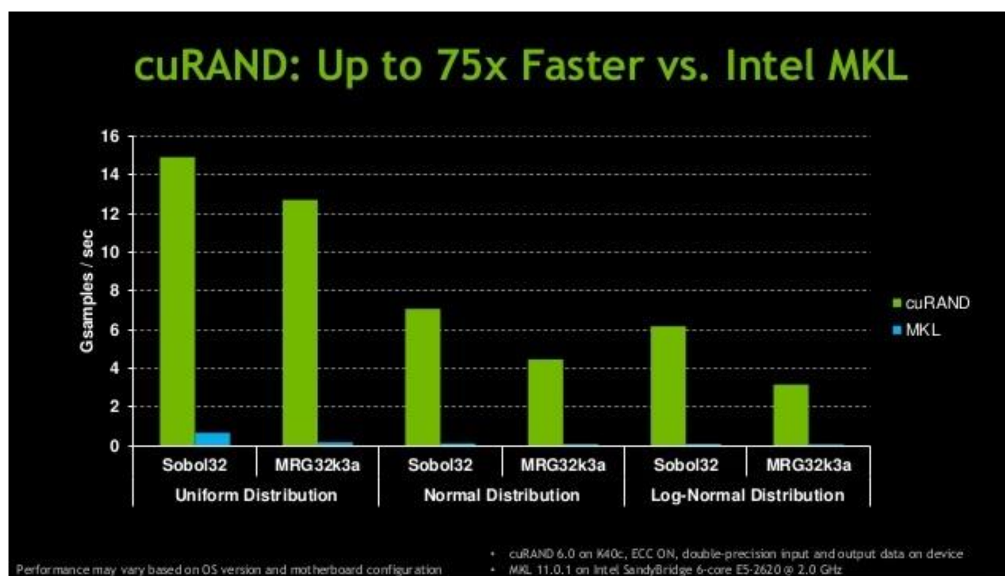
Η βιβλιοθήκη cuRAND προσφέρει ευκολίες που εστιάζουν στην απλή και αποδοτική παραγωγή υψηλής ποιότητας pseudorandom και quasirandom αριθμών, που είναι κατηγορίες ψευδοτυχαίων αριθμών. Μια ακολουθία pseudorandom αριθμών ικανοποιεί τις περισσότερες από τις στατιστικές ιδιότητες μιας πραγματικά τυχαίας ακολουθίας, αλλά παράγεται από έναν ντετερμινιστικό αλγόριθμο. Μια quasirandom ακολουθία από n-διάστατα σημεία παράγεται από έναν ντετερμινιστικό αλγόριθμο που είναι σχεδιασμένος να γεμίζει έναν n-διάστατο χώρο ομοιόμορφα.

Η βιβλιοθήκη cuRAND αποτελείται από δύο κομμάτια, μια host βιβλιοθήκη και μια device βιβλιοθήκη στη GPU. Υπενθυμίζουμε ότι χρησιμοποιούμε τον όρο host για ό,τι αφορά τη CPU και τη μνήμη της και τον όρο device για τη GPU και τη μνήμη της.

Το host τμήμα της βιβλιοθήκης χρησιμοποιείται μέσω του header file “curand.h” και μπορεί να παράξει τυχαίους αριθμούς τόσο στη CPU όσο και στη GPU. Για τη δημιουργία αριθμών στη GPU, η κλήση της συνάρτησης της βιβλιοθήκης γίνεται από τη CPU, αλλά ο υπολογισμός γίνεται στη GPU και το αποτέλεσμα αποθηκεύεται στη καθολική μνήμη.

Το device τμήμα της βιβλιοθήκης χρησιμοποιείται μέσω του header file “curand_kernel.h”. Σε αυτήν την περίπτωση οι αριθμοί μπορούν να παραχθούν και να χρησιμοποιηθούν απευθείας στις συναρτήσεις πυρήνα χωρίς τη μεσολάβηση της καθολικής μνήμης.

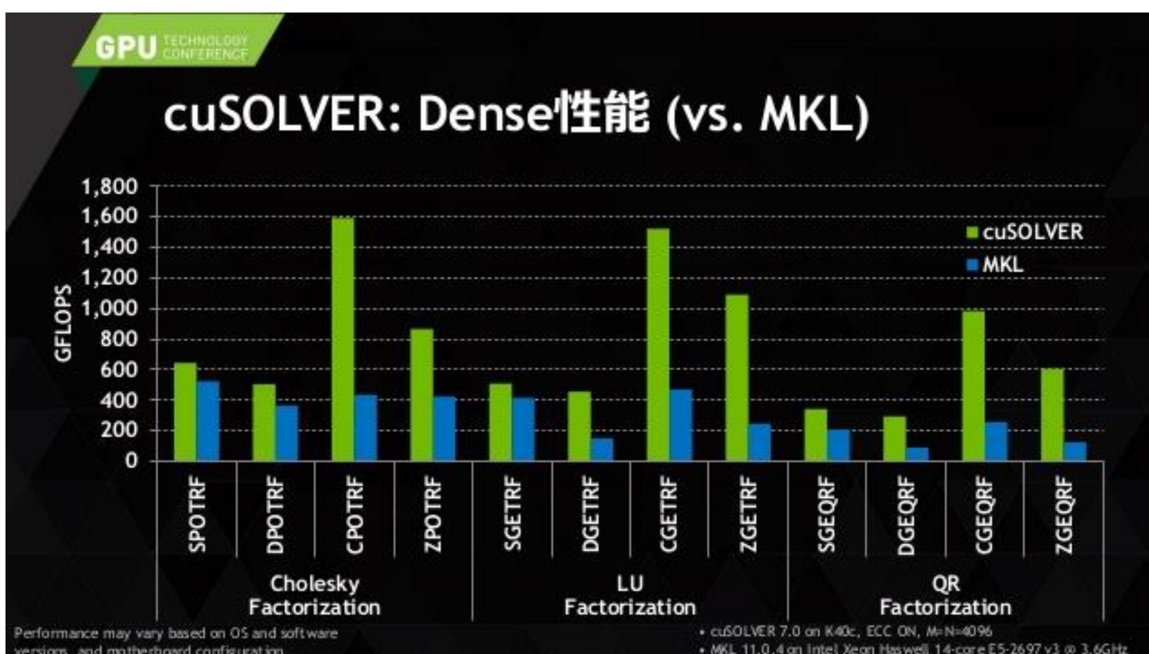
Στα πλαίσια της συγγραφής κώδικα χρησιμοποιήθηκε το host API της cuRAND. Συγκεκριμένα χρησιμοποιήθηκαν η curandGenerateUniform, που παράγει ομοιόμορφα κατανεμημένους αριθμούς κινητής υποδιαστολής μεταξύ 0 και 1, χωρίς να συμπεριλαμβάνεται το 0, και η curandGenerateNormal , που παράγει κανονικά κατανεμημένους αριθμούς κινητής υποδιαστολής με δεδομένη τη μέση τιμή και την τυπική απόκλιση.



Εικόνα 17: Σύγκριση cuRAND με MKL

2.5.3 Η βιβλιοθήκη cuSOLVER

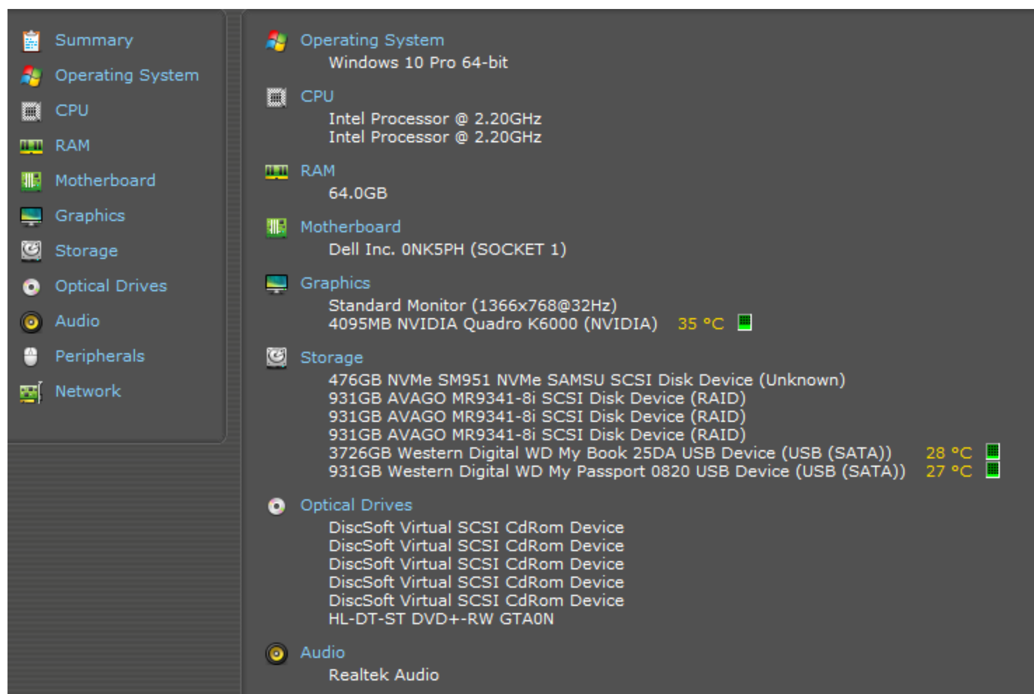
Η βιβλιοθήκη cuSolver είναι ένα υψηλού επιπέδου πακέτο που βασίζεται στις βιβλιοθήκες cuBLAS και cuSPARSE. Συνδυάζει τρεις ξεχωριστές βιβλιοθήκες (cuSolverDN, cuSolverSP, cuSolverRF) κάτω από την ίδια στέγη, κάθε μία από τις οποίες μπορεί να χρησιμοποιηθεί ανεξάρτητα ή σε συνδυασμό με άλλες βιβλιοθήκες. Χειρίζεται τόσο αραιούς όσο και πυκνούς (dense) πίνακες, αντιμετωπίζει θέματα παραγοντοποίησης πινάκων, επιλύει παραγοντοποιήσεις, όπως LU, QR, SVD, Cholesky και LDLT, ενώ μπορεί να υπολογίσει τις ιδιοτιμές ενός πίνακα. Για να χρησιμοποιηθούν οι συναρτήσεις της cuSOLVER πρέπει τα δεδομένα να βρίσκονται στη GPU και η GPU να έχει υπολογιστική ικανότητα 2.0 ή υψηλότερη.



Εικόνα 18: Σύγκριση παραγοντοποιήσεων Cholesky, LU και QR ανάμεσα σε cuSOLVER και MKL

2.6 Υπολογιστικοί πόροι συστήματος διεξαγωγής πειραμάτων

Η συγγραφή κώδικα, η διεξαγωγή πειραμάτων, η σύγκριση χρόνων και η βελτιστοποίηση πραγματοποιήθηκαν σε server του ΕΚΠΑ. Ο server έχει επεξεργαστή Intel Xeon E5-2630 v4 στα 2.2GHz με 10 πυρήνες και μνήμη RAM στα 64GB. Η κάρτα γραφικών που χρησιποιήθηκε για την εκτέλεση πειραμάτων σε CUDA είναι NVIDIA Quadro K6000 με υπολογιστική ικανότητα 3.5 και GPU μνήμη 12GB, που μπορεί να υποστηρίξει πράξεις με μεγάλου όγκου δεδομένα. Από άποψη λειτουργικού συστήματος, ο server χρησιμοποιεί Windows 10 Pro (64-bit). Το λογισμικό που χρησιμοποιήσαμε για τη συγγραφή κώδικα ήταν Visual Studio 2015 και Matlab R2015a.



Εικόνα 19: Χαρακτηριστικά υπολογιστή διεξαγωγής πειραμάτων

3. ΑΛΓΟΡΙΘΜΟΣ MM

3.1 Εισαγωγή στις μεθόδους ανάλυσης fMRI

Η εξαγωγή πληροφοριών από εικόνες fMRI υπήρξε ένα σημαντικό πεδίο έρευνας για πάνω από δύο δεκαετίες. Σκοπός αυτού του αλγορίθμου είναι να παρουσιάσει μια νέα μέθοδο για την ανάλυση των συνόλων δεδομένων από fMRI, η οποία είναι ικανή να ενσωματώσει εκ των προτέρων διαθέσιμη πληροφορία μέσω ενός αποδοτικού πλαισίου βελτιστοποίησης. Έρευνες πάνω σε συνθετικά δεδομένα επιδεικνύουν σημαντικό κέρδος στην απόδοση σε σχέση με υπάρχουσες μεθόδους αυτού του είδους. Στόχος της παρούσας εργασίας, είναι να μελετήσουμε τη βελτίωση που μπορεί να υπάρξει με τη χρήση GPU στο πρόβλημα που θα περιγράψουμε παρακάτω.

Η Λειτουργική Απεικόνιση με Μαγνητικό Συντονισμό (fMRI) είναι μια ισχυρή μη επεμβατική τεχνική κατάλληλη να παρέχει σημαντικές πληροφορίες σχετικά με τη δραστηριότητα του εγκεφάλου. Η μελέτη των διαφορετικών περιοχών του εγκεφάλου που σχετίζονται με σημαντικά καθήκοντα, όπως η όραση, η αντίληψη, η αναγνώριση κ.α., συνιστούν ένα τεράστιο ανοιχτό πεδίο έρευνας που απαιτεί στιβαρές και υψηλής ακρίβειας τεχνικές για την ανάλυση των δεδομένων του fMRI.

Τέτοια δεδομένα παράγονται σαν μια ακολουθία τρισδιάστατων εικόνων του εγκεφάλου που λαμβάνονται διαδοχικά στο χρόνο. Κάθε μία από αυτές τις εικόνες φτιάχνονται από μια αλληλουχία στοιχειωδών κύβων, τα voxels. Συνεπώς, η τιμή τού κάθε voxel αντικατοπτρίζει το βαθμό δραστηριότητας σε ένα συγκεκριμένο σημείο τού εγκεφάλου. Κάθε τρισδιάστατη εικόνα αποθηκεύεται σε ένα μεγάλο διάνυσμα σειράς, $x = [x_1, x_2, \ldots, x_N] \in \mathbb{R}^N$, όπου N είναι το συνολικό πλήθος των voxels. Στη συνέχεια, όλα αυτά τα διανύσματα δεδομένων συνενώνονται για να δημιουργήσουν έναν πίνακα δεδομένων, $\mathrm{X} \in \mathbb{R}^{\mathrm{T} \times N}$, όπου T είναι ο συνολικός αριθμός των διαδοχικών λήψεων στο χρόνο.

Στον εγκέφαλο, πολλές διαφορετικές δραστηριότητες πραγματοποιούνται ταυτόχρονα. Για αυτό, τα ληφθέντα δεδομένα αποτελούνται από μια μίξη διάφορων σημάτων ενεργοποίησης στα οποία αναφερόμαστε με τον όρο πηγές. Ο στόχος της ανάλυσης των δεδομένων fMRI είναι να διαχωρίσει αυτές τις πηγές ώστε να αποκαλύψει τόσο τα ματίβα ενεργοποίησης τους όσο και τις αντίστοιχες περιοχές του εγκεφάλου που σχετίζονται με κάθε μία από τις πηγές.

Από μαθηματικής άποψης, ο διαχωρισμός πηγών μπορεί να περιγραφεί σαν ένα πρόβλημα παραγοντοποίησης πίνακα στη μορφή $\mathrm{X} \approx DS$, όπου $D \in \mathbb{R}^{\mathrm{T} \times \mathrm{K}}$ είναι ένας πίνακας, του οποίου οι στήλες αναπαριστούν τα μοτίβα ενεργοποίησης ή χρονοδιαγράμματα που σχετίζονται με τις πηγές, $S \in \mathbb{R}^{K \times N}$ είναι ένας πίνακας, του οποίου οι γραμμές αναπαριστούν τις περιοχές του εγκεφάλου που ενεργοποιούνται από τις σχετικές πηγές και K το πλήθος των πηγών των οποίων οι τιμές ορίζονται από το χρήστη. Οι γραμμές του πίνακα S συχνά αναφέρονται ως χωρικοί χάρτες.

Ένα ευρέως χρησιμοποιούμενο εργαλείο στην ανάλυση fMRI είναι το Γενικό Γραμμικό Μοντέλο (General Linear Model), το οποίο βασίζεται στην υποθετική μορφή της HRF (ενότητα 1.4) για την κατασκευή του πίνακα D. Το συγκεκριμένο σχέδιο του κάθε πειράματος επιτρέπει να γίνει μια εικασία για τις χρονικές στιγμές που περιμένουμε να εμφανιστεί δραστηριότητα. Υιοθετώντας μια συναρτησιακή μορφή για την HRF και συσχετίζοντας τη με την αναμενόμενη ακολουθία ενεργοποίησης, το χρονοδιάγραμμα που αντιστοιχεί στην συγκεκριμένη ενέργεια μπορεί να προσεγγιστεί και να θεωρηθεί γνωστό. Τέτοια χρονοδιαγράμματα ονομάζονται σχετικά με την εργασία χρονοδιαγράμματα (task-related time courses).

Εναλλακτικά, κάποιος μπορεί να χρησιμοποιήσει μια BSS προσέγγιση, η οποία μπορεί ταυτόχρονα να υπολογίσει τους D και S χωρίς να καταφύγει σε καμία υπόθεση σχετικά με την HRF. Για το σκοπό αυτό, υιοθετόνται διαφορετικές εικασίες σχετικά είτε με τις στατιστικές είτε με τις δομικές ιδιότητες των εμπλεκόμενων πινάκων. Ειδικότερα, η ανάλυση ICA , που έχει χρησιμοποιηθεί εκτεταμένα από προβλήματα διαχωρισμού σε fMRI, υποθέτει ανεξαρτησία μεταξύ των πηγών, ενώ οι τεχνικές που βασίζονται σε Dictionary Learning, που γίνονται όλο και πιο δημοφιλείς τελευταία, εκμεταλλεύονται το γεγονός ότι ο πίνακας S αναμένεται να είναι αραιός.

Πρόσφατα, παρουσιάστηκε μια μέθοδος, η Εποπτευόμενη Μάθηση με Λεξικό (Supervised Dictionary Learning), που επιτρέπει την ενσωμάτωση των πληροφοριών που σχετίζονται με την HRF σε ένα πλαίσιο BSS, οδηγώντας σε βελτιωμένα αποτελέσματα. Παρόλα αυτά, τόσο η GLM όσο και η SDL υποφέρουν από το ίδιο μειονέκτημα: πρέπει να γίνει μια αρκετά ακριβής υπόθεση για τη συναρτησιακή μορφή της HRF.

Η μέθοδος που θα μελετήσουμε είναι μια νέα μέθοδος Dictionary Learning υποβοηθούμενη από άτομα (Atom-Assisted Dictionary Learning), η οποία, παρόλο που εκμεταλλεύεται τα πλεονεκτήματα της ενσωμάτωσης κάποιας εκ των προτέρων γνώσης σχετικά με την HRF, επιτρέπει σημαντική ανοχή στη μη ακριβή επιλογή αντίστοιχης μορφής.

3.2 Υποβοηθούμενη Μάθηση με Λεξικό

3.2.1 Εποπτευόμενη Μάθηση με Λεξικό

Στην πράξη, η χρήση εκτιμώμενων χρονοδιαγραμμάτων παρέχει γενικά ικανοποιητικά αποτελέσματα στο GLM. Από την άλλη πλευρά, η εικασία σχετικά με τη χωρική αραιότητα συμπυκνώνει πολύτιμες πληροφορίες, οι οποίες μαθηματικά μεταφράζονται σε ισοδύναμη χρήση των περιορισμών αραιότητας στις μεθόδους Dictionary Learning. Παρόλα αυτά, αποδεικνύεται ότι οι εκ των προτέρων εικασίες, που υιοθετούνται από αυτές τις μεθόδους, δεν είναι από μόνες τους αρκετά ισχυρές για να καταφέρουν επαρκή αποτελέσματα.

Η αφετηρία της διατύπωσης της SDL βρίσκεται στο διαχωρισμό του βασικού λεξικού σε δύο τμήματα: $D=[\Delta, D_F]\in\mathbb{R}^{T\times K}$, όπου το πρώτο τμήμα, $\Delta\in\mathbb{R}^{T\times M}$, περιορίζεται στο να περιέχει τα επιβαλλόμενα σχετικά με την εργασία χρονοδιαγράμματα και θεωρείται σταθερή. Αντίθετα, το δεύτερο τμήμα $D_F\in\mathbb{R}^{T\times(K-M)}$, είναι το μεταβλητό τμήμα που μπορεί να προσδοριστεί μέσω Dictionary Learning.

Το αποτέλεσμα είναι ακόμα ένα σχήμα Dictionary Learning, αλλά συμπεριλαμβάνει συγκεκριμένα χρονοδιαγράμματα. Αποδεικνύεται από τα αποτελέσματα ότι οδηγεί σε βελτιωμένα αποτελέσματα, σε σχέση με εκείνα των κλασικών τεχνικών Dictionary Learning. Εντούτοις, αυτή η προσέγγιση ακόμα κληρονομεί το ίδιο σοβαρό μειονέκτημα που σχετίζεται με το GLM. Αυτό είναι ότι τα άτομα του λεξικού στα οποία έχει επιβληθεί περιορισμός (στήλες του πίνακα Δ) βοηθούν μόνο αν η εκ των προτέρων επιβαλλόμενη πληροφορία είναι αρκετά ακριβής. Εάν δεν είναι, η συνεισφορά της μπορεί να έχει επιβλαβείς συνέπειες, οδηγώντας σε εσφαλμένα αποτελέσματα.

3.2.2 Υποβοηθούμενη από Άτομα Μάθηση με Λεξικό

Η υποβοηθούμενη από άτομα μάθηση με λεξικό είναι μια εναλλακτική προσέγγιση σε σχέση με την SDL. Παρέχει έναν πιο χαλαρό τρόπο ενσωμάτωσης των εκ των προτέρων υιοθετημένων μορφών των χρονοδιαγραμμάτων. Η κύρια ιδέα είναι ότι τα άτομα του λεξικού του τμήματος στο οποίο επιβάλλεται ο περιορισμός δεν είναι

υποχρεωτικά ίδια με τα επιλεγμένα. Αντίθετα, χρησιμοποιείται ένας χαλαρότερος περιορισμός που είναι ενσωματωμένος στη διαδικασία βελτιστοποίησης. Συνεπώς, η απαίτηση της ισχυρής ισότητας χαλαρώνει μέσω μιας πιο χαλαρής ομοιότητας που περιορίζει τη νόρμα μέτρησης της απόστασης.

Επομένως, εάν ένα μέρος της εκ των προτέρων πληροφορίας δεν είναι και τόσο ακριβές, επειδή τα άτομα στα οποία έχουν επιβληθεί περιορισμοί δε θεωρούνται σταθερά πλέον, η μέθοδος μπορεί να τα αναπροσαρμόσει, σε σχέση με την πληροφορία που υπάρχει στα δεδομένα, με βέλτιστο τρόπο. Αποδεικνύεται ότι μια τέτοια προσέγγιση κάνει τη διαδικασία πιο ισχυρή απέναντι στο βασικό μειονέκτημα των μεθόδων που βασίζονται στην HRF.

Και εδώ η αφετηρία είναι ο διαχωρισμός του λεξικού: $D=[D_c,D_F]\in\mathbb{R}^{T\times K}$. Σε αντίθεση με την προσέγγιση SDL, ωστόσο, το περιοσμένο τμήμα του λεξικού, $D_C\in\mathbb{R}^{T\times M}$, δε θεωρείται σταθερό πλέον, αλλά μπορεί να διαφοροποιείται σύμφωνα με το κόστος βελτιστοποίησης.

Η εργασία βελτιστοποίησης, που χρησιμοποιείται εδώ, διατυπώνεται ως εξής: $(D,S)=\underset{D,S}{\arg\min}\|X-DS\|_F^2+\lambda\|S\|_{1,1}$ έτσι ώστε $D\in\mathfrak{D}$, όπου $\|S\|_{1,1}=\sum_i^K\sum_j^N|s_{ij}|$ είναι ο όρος που προωθεί την αραιότητα στον πίνακα συντελεστών και $\mathfrak{D}$ είναι ένα αποδεκτό σύνολο λεξικών. Στην περίπτωσή μας, το $\mathfrak{D}$ συνιστά το σύνολο των λεξικών που μοιράζονται την ακόλουθη ιδιότητα: $\mathfrak{D}=\left\{D\in\mathbb{R}^{T\times K}:\begin{array}{l}\|d_i-\delta_i\|_2^2\le c_\delta,\ \forall i\in[1,M]\subset\mathbb{N}\\ \|d_i\|_2^2\le c_d,\ \forall i\in[M+1,K]\subset\mathbb{N}\end{array}\right.$, όπου $\mathbb{N}$ είναι το σύνολο των φυσικών αριθμών, $\|.\|_2$ δηλώνει τη νόρμα-2, d_i είναι η i-οστή στήλη του λεξικού D και δ_i είναι το i-οστό εκ των προτέρων επιλεγμένο σχετικό με την εργασία χρονοδιάγραμμα. Η σταθερά c_δ είναι μια ορισμένη από το χρήστη παράμετρος, η οποία ελέγχει το βαθμό της ομοιότητας μεταξύ των ατόμων που έχουν υποστεί περιορισμούς και των επιβαλλόμενων χρονοδιαγραμμάτων. Τα υπόλοιπα άτομα του λεξικού περιορίζονται στο να έχουν οριοθετημένη νόρμα, όχι μεγαλύτερη από μια προκαθορισμένη παράμετρο c_d.

3.2.3 Μέθοδος Βελτιστοποίησης

Για να λύσουμε το προηγούμενο πρόβλημα βελτιστοποίησης χρησιμοποιείται η μέθοδος μεγιστοποίησης (majorization method), η οποία έχει ήδη χρησιμοποιηθεί στο παρελθόν για να επιλύσει προβλήματα Dictionary Learning. Χωρίς αμφιβολία, οποιαδήποτε άλλη σχετική μέθοδος βελτιστοποίησης μπορεί να επιστρατευτεί και η καταλληλότητα της επιλογής αυτής είναι ακόμα υπό μελέτη. Παρόλο που η μέθοδος μεγιστοποίησης δεν απαιτεί χαλάρωση κατά Lagrange του υπό περιορισμό προβλήματος, αυτή χρησιμοποιείται για απλότητα. Συνεπώς, το ισοδύναμο πρόβλημα βελτιστοποίησης, μέσω της αντίστοιχης διατύπωσης Lagrange του παραπάνω προβλήματος ελαχιστοποίησης είναι $(D,S)=\underset{D,S}{\arg\min}\,\phi_{\lambda,\gamma}(\mathrm{D,S})$, όπου $\phi_{\lambda,\gamma}(\mathrm{D,S})=\|X-DS\|_F^2+\lambda\|S\|_{1,1}+P_\gamma(\mathrm{D})$. Το $P_\gamma(\mathrm{D})$ εξαρτάται από το λεξικό και ορίζεται ως $P_\gamma(\mathrm{D})=\sum_{i=1}^M\gamma_i[(\mathrm{d}_i-\delta_i)^T(\mathrm{d}_i-\delta_i)-c_\delta]+\sum_{i=M+1}^K\gamma_i(\mathrm{d}_i^T\mathrm{d}_i-c_d)$, όπου οι παράμετροι γ_i με $i=1,2,\ldots,\mathrm{K}$ αντιστοιχούν στους K συσχετισμένους πολλαπλασιαστές Lagrange. Η τελευταία ισότητα μπορεί συμπαγώς να εκφραστεί ως $P_\Gamma=tr\left[\Gamma(\mathrm{D}-\Delta\mathrm{M})^T(\mathrm{D}-\Delta\mathrm{M})-C\right]$, όπου $M\in\mathbb{R}^{M\times K}$, που έχει μηδενικά παντού εκτός από την κύρια διαγώνιο, κάθε στοιχείο

της οποίας ισούται με ένα, ο Γ είναι διαγώνιος πίνακας με γ_i, $i=1,2,\dots,\mathrm{K}$ το i-οστό στοιχείο της κύριας διαγωνίου και C είναι διαγώνιος πίνακας με τις αντίστοιχες παραμέτρους c_δ και c_d στη διαγώνιο του. Συνεπώς, η συνάρτηση κόστους μπορεί να ξαναγραφτεί ως $\phi_{\lambda,\Gamma}(\mathrm{D},\mathrm{S}) = \| X - DS \|_F^2 + \lambda \| S \|_{1,1} + P_\Gamma(\mathrm{D})$.

3.2.4 Ο αλγόριθμος

Η βελτιστοποίηση σε σχέση με τους πίνακες D και S είναι αρκετά απαιτητική και απλοποιείται σημαντικά υιοθετώντας μια εναλλασσόμενη επαναληπτική διαδικασία ελαχιστοποίησης δύο βημάτων. Πιο συγκεκριμένα, ξεκινώντας από τυχαίες εκτιμήσεις, $D_{(0)}$ και $S_{(0)}$, ο αλγόριθμος αποτελείται από τα ακόλουθα βήματα:

Βήμα I: $\min_S \phi_{\lambda,\Gamma}(\mathrm{D},\mathrm{S})$ με σταθερό D (Coefficient Update)

Βήμα II: $\min_D \phi_{\lambda,\Gamma}(\mathrm{D},\mathrm{S})$ με σταθερό S (Dictionary Update)

Ακολουθώντας την τεχνική μεγιστοποίησης, για κάθε βήμα, η συνάρτηση που αποτελεί αντικείμενο μεγιστοποίησης αντικαθίσταται από μια αντιπροσωπευτική συνάρτηση (surrogate function), που τη μεγιστοποιεί και είναι ευκολότερο να ελαχιστοποιηθεί επαναληπτικά, σε σχέση με την αυθεντική συνάρτηση. Η αντιπροσωπευτική συνάρτηση δεν είναι μοναδική, αλλά πρέπει να ικανοποιεί συγκεκριμένες συνθήκες.

3.2.4.1 Βήμα 1: Ενημέρωση Συντελεστών (Coefficient Update)

Στο βήμα t, η αυθεντική συνάρτηση ελαχιστοποιείται σε σχέση με το S κρατώντας σταθερό το D στην τρέχουσα διαθέσιμη εκτίμηση, $\mathrm{D} = \mathrm{D}_{(t)}$. Αυτή η ελαχιστοποίηση επιτυγχάνεται με επαναληπτικό τρόπο και μέσω της εισαγωγής της αντιπροσωπευτικής συνάρτησης. Ξεκινώντας τις επαναλήψεις από την υπάρχουσα εκτίμηση, $S^{[0]} = S_{(t)}$, η εκτίμηση, $S^{[\mathrm{n}]}$, της n-οστής επανάληψης, παράγεται από την προηγούμενη εκτίηση, $S^{[\mathrm{n}-1]}$, ελαχιστοποιώντας την αντιπροσωπευτική συνάρτηση $\psi_\lambda(\mathrm{S},\mathrm{S}^{[\mathrm{n}-1]}) = \phi_{\lambda,\Gamma}(\mathrm{D},\mathrm{S}) + \pi_S(\mathrm{S},\mathrm{S}^{[\mathrm{n}-1]})$, όπου $\pi_S(\mathrm{S},\mathrm{S}^{[\mathrm{n}-1]}) := c_S \left\| S - S^{[\mathrm{n}-1]} \right\|_F^2 - \left\| DS - DS^{[\mathrm{n}-1]} \right\|_F^2$ και $c_S > \left\| D^T D \right\|_2$ είναι μια σταθερά όπου $\left\| \bullet \right\|_2$ είναι η φασματική νόρμα. Συνεπώς, δύο διαφορετικές επαναλήψεις τρέχουν σε εμφωλευμένη τιμή, μία εξωτερική σε σχέση με το t και μία σε σχέση με το n.

Έστω $A := \frac{1}{c_S}(\mathrm{D}^T\mathrm{X} + (c_S I_K - \mathrm{D}^T\mathrm{D})\mathrm{S}^{[\mathrm{n}-1]})$. Μπορεί να δειχθεί ότι η βέλτιστη τιμή της παραπάνω αντιπροσωπευτικής συνάρτησης βρίσκεται συρρικνώνοντας τα στοιχεία του A, με άλλα λόγια, $S^{[\mathrm{n}]} = S_\lambda(\mathrm{A})$, όπου $S_\lambda(\mathrm{A}) : \mathrm{s}_{ij} = \begin{cases} a_{ij} - \frac{\lambda}{2}\mathrm{sign}(a_{ij}), & \alpha\nu \ \left| a_{ij} \right| > \frac{\lambda}{2} \\ 0, & \alpha\lambda\lambda\iota\acute{\omega}\varsigma \end{cases}$.

Η επαναληπτική ενημέρωση συνεχίζεται μέχρις ότου ικανοποιηθεί ένα κριτήριο τερματισμού. Ο ψευδοκώδικας για την ενημέρωση συντελεστών είναι διαθέσιμος στο πεδίο Algorithm1 - Step 1 της Εικόνας 20.

3.2.4.2 Βήμα 2: Ενημέρωση Λεξικού (Dictionary Update)

Στο δεύτερο βήμα της εναλλασσόμενης ελαχιστοποίησης, η αντικειμενική συνάρτηση ελαχιστοποιείται σε σχέση με το D, κρατώντας σταθερό το S στην τρέχουσα διαθέσιμη εκτίμηση, $S = S_{(t+1)}$. Σε αυτό το βήμα χρησιμοποιείται επίσης μια λογική μεγιστοποίησης.

Για το σκοπό αυτό, εισάγεται μια κατάλληλη αντιπροσωπευτική συνάρτηση που δίνεται από τον τύπο $\psi_{\Gamma}(\mathrm{D},\mathrm{R}) = \phi_{\lambda,\Gamma}(\mathrm{D},\mathrm{S}) + \pi_D(\mathrm{D},\mathrm{R})$, όπου $c_D > \left\| S^T S \right\|_2$ είναι σταθερά και $R = D^{[\mathrm{n}-1]}$ είναι η εκτίμηση του λεξικού στο προηγούμενο βήμα. Η ελαχιστοποίηση σε σχέση με το D γίνεται επίσης επαναληπτικά, ξεκινώντας από $D^{[0]} = D_{(t)}$. Η βέλτιστη τιμή της αντιπροσωπευτικής συνάρτησης βρίσκεται στο σημείο μηδενικής κλίσης: $\frac{d\psi_{\Gamma}}{dD} = -2XS^T + 2(\mathrm{D} - \Delta\mathrm{M})\Gamma + 2c_D(\mathrm{D} - \mathrm{R}) + 2\,\mathrm{RSS}^T$.

Θέτοντας την παραπάνω παράγωγο ίση με το μηδέν, λύνοντας ως προς D και δίνοντας στα γ_i τιμές που ικανοποιούν τις συνθήκες Karush-Kuhn-Tucker (KKT), ακολουθείται μια διαδικασία δύο βημάτων για την ενημέρωση του λεξικού, παρόμοια με το βήμα 1 με αντίστοιχες ποσότητες. Ειδικότερα, ορίζεται μια ενδιάμεση ποσότητα B και υπολογίζεται με τον τύπο $\mathrm{B} := \frac{1}{c_D}(\mathrm{XS}^T + R(c_D I - SS^T))$.

Οι προσεγγίσεις των ενημερωμένων ατόμων του λεξικού δίνονται ως εξής:

$$
d_i^{[n]} = \begin{cases} i \in [1,\mathrm{M}], \begin{cases} b_i, \ \alpha\nu \ \left\| b_i - \delta_i \right\|_2^2 \le c_\delta \\ \dfrac{c_\delta (b_i - \delta_i)}{\left\| b_i - \delta_i \right\|^2} + \delta_i, \ \alpha\lambda\lambda\iota\acute{\omega}\varsigma \end{cases} \\ i \in [\mathrm{M}+1,\mathrm{K}], \begin{cases} b_i, \ \alpha\nu \ \left\| b_i \right\|^2 \le c_\delta \\ \dfrac{c_d}{\left\| b_i \right\|^2} b_i, \ \alpha\lambda\lambda\iota\acute{\omega}\varsigma \end{cases} \end{cases}
$$

Δεν είναι δύσκολο να αποδειχθεί ότι μετά από αυτή την ενημέρωση, διατηρούνται οι συνθήκες KKT για όλες τις στήλες του λεξικού. Επιπροσθέτως, το σύνολο όλων των αποδεκτών λεξικών, $\mathfrak{D}$, απαρτίζουν ένα κυρτό μη κενό σύνολο. Μπορεί να δειχθεί ότι αυτό το γεγονός εγγυάται ότι ο προτεινόμενος αλγόριθμος συγκλίνει για τυχαία αρχικοποίηση. Ο ψευδοκώδικας για την ενημέρωση λεξικού παρουσιάζεται στο πεδίο Algorithm2 - Step 2 της Εικόνας 20.

Algorithm 1 - Step I

1: **Initialization:** $c_S > \|\mathbf{D}^T\mathbf{D}\|_2$, $\mathbf{S}^{[0]} = \mathbf{S}_{(t)}$, $n = 0$
2: **repeat**
3: $n = n + 1$
4: $\mathbf{A} = \frac{1}{c_S}\left(\mathbf{D}^T\mathbf{X} + \left(c_S\mathbf{I}_K - \mathbf{D}^T\mathbf{D}\right)\mathbf{S}^{[n-1]}\right)$
5: $\mathbf{S}^{[n]} = \mathcal{S}_\lambda(\mathbf{A})$
6: **until** stop criterion is met*
7: **output:** $\mathbf{S}_{(t+1)} = \mathbf{S}^{[n]}$

Algorithm 2 - Step II

1: **Initialization:** $c_D > \|\mathbf{S}^T\mathbf{S}\|_2$, $\mathbf{D}^{[0]} = \mathbf{D}_{(t)}$, $n = 0$
2: **repeat**
3: $n = n + 1$
4: $\mathbf{B} = \frac{1}{c_D}\left(\mathbf{X}\mathbf{S}^T + \mathbf{R}\left(c_D\mathbf{I}_K - \mathbf{S}\mathbf{S}^T\right)\right)$
5: $\mathbf{D}^{[n]} = \mathcal{U}(\mathbf{B})$
6: **until** stop criterion is met*
7: **output:** $\mathbf{D}_{(t+1)} = \mathbf{D}^{[n]}$

* In this paper, a fixed number of iterations is used.

Εικόνα 20: Ψευδοκώδικας αλγορίθμου MM

3.3 Υλοποίηση

Ο κώδικας MATLAB για τη μέθοδο που περιγράψαμε στην προηγούμενη ενότητα βρίσκεται στη διεύθυνση https://github.com/MorCTI/Attom-Assisted-DL. Σε αυτόν τον κώδικα βασίστηκε η βελτιστοποίηση μας. Σε όλες τις εκτελέσεις, τόσο του αρχικού προγράμματος όσο και των δύο υλοποιήσεων σε MATLAB με πίνακες GPU και σε CUDA, χρησιμοποιούμε κοινές παραμέτρους, ώστε να μπορούμε να συγκρίνουμε τους χρόνους εκτέλεσης και το σφάλμα που προκύπτει. Έτσι, τα προγράμματα εκτελούνται με διπλή επανάληψη, μια εξωτερική 200 φορές, και δύο εσωτερικές από 20 φορές που ενημερώνουν τους συντελεστές και το λεξικό αντίστοιχα. Το πλήθος των πηγών είναι 10, η παράμετρος λ παίρνει την τιμή 0.5 και η παράμετρος ccl που ελέγχει την κανονικοποίηση των ατόμων του λεξικού λαμβάνει την τιμή 1.

3.3.1 Υλοποίηση σε MATLAB με πίνακες GPU

Ο κώδικας MATLAB βρίσκεται στο Παράρτημα Ι στην ενότητα με τίτλο «Αλγόριθμος MM». Οι παράμετροι K, Tt, Ts, Es, Td, Ed, lambda και ccl διατηρήθηκαν όπως στον αρχικό κώδικα. Η διαφορά έγκειται στη χρήση gpuArrays στον υπάρχοντα κώδικα καθώς και στην προσαρμογή κάποιον πράξεων ώστε να υποστηρίζουν πράξεις σε GPU.

Ειδικότερα, οι βελτιστοποίησεις σε σχέση με τον αρχικό κώδικα είναι οι εξής:

- Αφαιρέσαμε την παράμετρο Verb που χρησιμοποιείται για την εμφάνιση γραμμής προόδου (progress bar). Στη θέση της εκτυπώνουμε το σφάλμα σε κάθε επανάληψη, ώστε να διαπιστώνουμε πόσο γρήγορα συγκλίνει ο αλγόριθμος. Για αυτό το λόγο δε χρησιμοποιούμε και τον πίνακα E που αποθηκεύει το σφάλμα σε κάθε επανάληψη.
- Αγνοήσαμε τις παραμέτρους του κανονικού λεξικού (canonical dictionary), δηλαδή τις cdl και Del, επειδή δε μας ζητήθηκε να υλοποιήσουμε το κανονικό λεξικό στα πλαίσια αυτής της εργασίας. Συνεπώς, αγνοήσαμε και τα αντίστοιχα κομμάτια κώδικα.

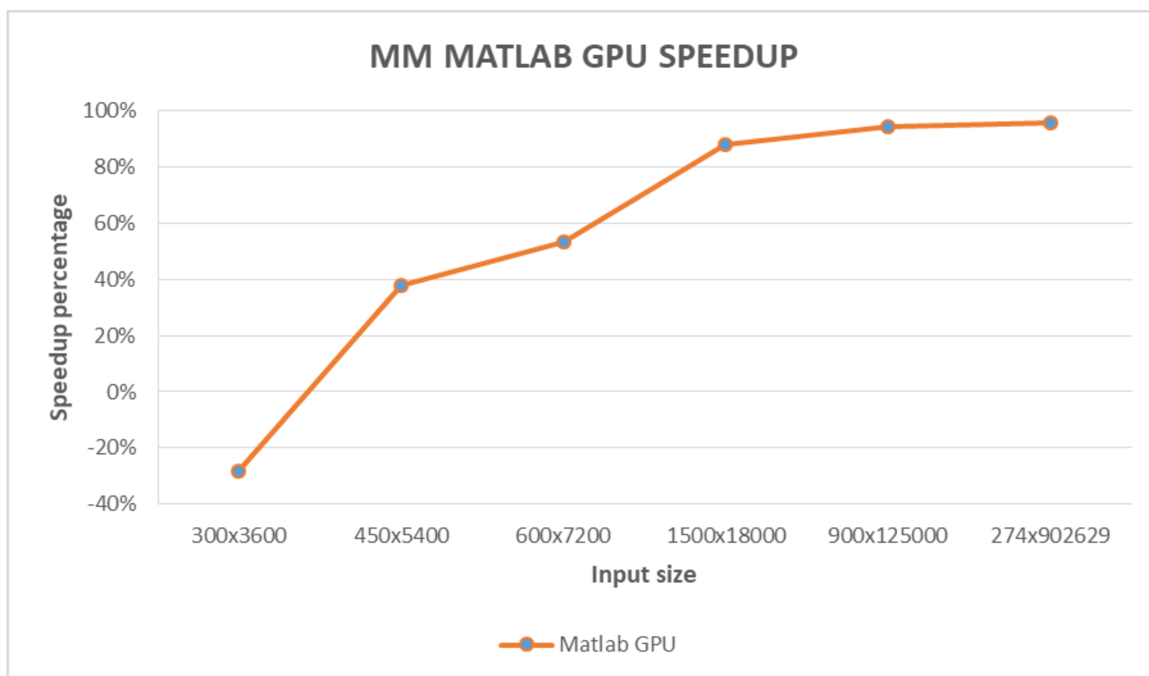
- Δημιουργήσαμε τους πίνακες Y, D, S και I ως πίνακες GPU. Όλοι οι υπόλοιποι πίνακες δημιουργήθηκαν και αυτοί στη GPU, επειδή προήλθαν από πράξεις πινάκων που βρίσκονται ήδη στη GPU.
- Προσπαθήσαμε να αποφύγουμε την επανάληψη πράξεων. Για αυτό, τον πίνακα I, που στο αρχικό πρόβλημα δημιουργούνταν τόσο στη συνάρτηση NewCoef όσο και στη NewDict, δηλαδή συνολικά $2\times Tt$ φορές, πλέον τον δημιουργούμε μόνο μία. Ακόμα, αποθηκεύσαμε το γινόμενο $D'*D$ στον πίνακα Dux και αντικαταστήσαμε την πράξη $Aq = I - D'*D / c_S$ με $Aq = I - Dux / c_S$, ώστε να αποφύγουμε να υπολογίσουμε ξανά το γινόμενο. Την ίδια λογική ακολουθούμε και με το γινόμενο $S*S'$ και την πράξη $Bq = I - S*S'/c_D$.

Στον Πίνακα 1 συγκρίνονται τα δύο προγράμματα. Παρατηρούμε τους χρόνους εκτέλεσης των δύο προγραμμάτων για διαφορετικού μέγεθους πίνακες εισόδου και υπολογίζουμε το ποσοστό βελτίωσης του νέου προγράμματος χρησιμοποιώντας τον τύπο $\frac{t_{CPU} - t_{GPU}}{t_{CPU}} \times 100\%$, όπου t_{CPU} ο χρόνος εκτέλεσης του προγράμματος σε CPU και t_{GPU} ο χρόνος εκτέλεσης σε GPU. Στα αρχικά δεδομένα υπάρχει επιβράδυνση διότι δεν είναι αρκετά μεγάλα, ώστε να δικαιολογηθεί η κατασκευή πινάκων στη GPU, που επιβαρύνει το πρόγραμμα. Η βελτίωση είναι εμφανής όσο ο πίνακας εισόδου μεγαλώνει, διότι η επιβάρυνση είναι αμελητέα σε σχέση με την πολυπλοκότητα των πράξεων.

Πίνακας 1: Σύγκριση χρόνων εκτέλεσης MM σε MATLAB με χρήση CPU και GPU

ΜΕΓΕΘΟΣ	Χρόνος CPU (sec)	Χρόνος GPU (sec)	ΠΟΣΟΣΤΟ ΒΕΛΤΙΩΣΗΣ
300×3600	4,8782	6,2463	-28,05%
450×5400	10,1033	6,2969	37,67%
600×7200	14,8908	6,9444	53,36%
1500×18000	85,5759	10,5524	87,67%
900×125000	401,814	23,9282	94,05%
274×902629	3101,5717	143,862	95,36%

Στην Εικόνα 21 βλέπουμε πως διαμορφώνεται η επιτάχυνση της υλοποίησης του αλγορίθμου MM με χρήση MATLAB σε GPU σε σχέση με την υλοποίηση του με χρήση CPU, σύμφωνα με τους χρόνους του Πίνακα 1. Παρατηρούμε ότι στα αρχικά δεδομένα 300×3600 υπάρχει επιβράδυνση, όμως μεγαλώνοντας τα δεδομένα το πρόγραμμα επιταχύνεται σημαντικά. Τέλος, είναι φανερό ότι όσο μεγαλώνουν τα δεδομένα, δεν είναι τόσο ραγδαία η ποσοστιαία επιτάχυνση όσο είναι μεταξύ των αρχικών μικρότερων δεδομένων.



Εικόνα 21: Διαμόρφωση επιτάχυνσης αλγορίθμου MM σε MATLAB με GPU

3.3.2 Υλοποίηση σε CUDA

Ο κώδικας CUDA βρίσκεται στο Παράρτημα II στην ενότητα με τίτλο «Αλγόριθμος MM». Οι παράμετροι K, Tt, Ts, Es, Td, Ed, lambda και ccl διατηρήθηκαν όπως στον αρχικό κώδικα με αντίστοιχα ονόματα K, ITER, TS, ES, TD, ED, LAMBDA και CCL και προστέθηκαν η παράμετρος MODE και κάποιες ακόμα για διάβασμα εισόδου από αρχείο μορφής mat. Στην παράμετρο MODE δίνουμε την τιμή 0 ή 1 ανάλογα με το αν θέλουμε να εκτελέσουμε το πρόγραμμα με μονή ή διπλή ακρίβεια αντίστοιχα. Για το διάβασμα αρχείου mat, υπάρχουν δυό συμβάσεις που υιοθετούνται στο πρόγραμμα: πρώτον, ο πίνακας πρέπει να έχει το όνομα X και να αποθηκευτεί στο αρχείο X.mat και δεύτερον, πρέπει να βρεθεί ο ανάστροφος του, να γίνει reshape στις αρχικές διαστάσεις και μετά να αποθηκευτεί στο mat αρχείο. Έτσι, αν στο MATLAB έχουμε έναν πίνακα X μέγεθους $a \times b$ πρέπει να ακολουθήσουμε τα βήματα X=X' και X= reshape(X,a,b) πριν παράξουμε το X.mat. Αυτό συμβαίνει γιατί το MATLAB περιέχει πίνακες βασισμένους σε στήλες, ενώ η υλοποίηση μας για τον MM περιέχει πίνακες βασισμένους σε γραμμές. Τα ονόματα των ενδιάμεσων μεταβλητών Dux, Dy, Aq, A, Sux, YS, Bq και B έχουν αντικατασταθεί με κάποιες κοινές μεταβλητές που αξιοποιούνται τόσο στη συνάρτηση NewDict όσο και στη συνάρτηση NewCoef, ώστε να μειώσουμε όσο περισσότερο γίνεται τη μνήμη που δεσμεύουμε στη GPU.

Θα περιγράψουμε αναλυτικά πως υλοποιήσαμε κάθε πράξη σε CUDA συσχετίζοντας τη με τον αντίστοιχο κώδικα σε MATLAB:

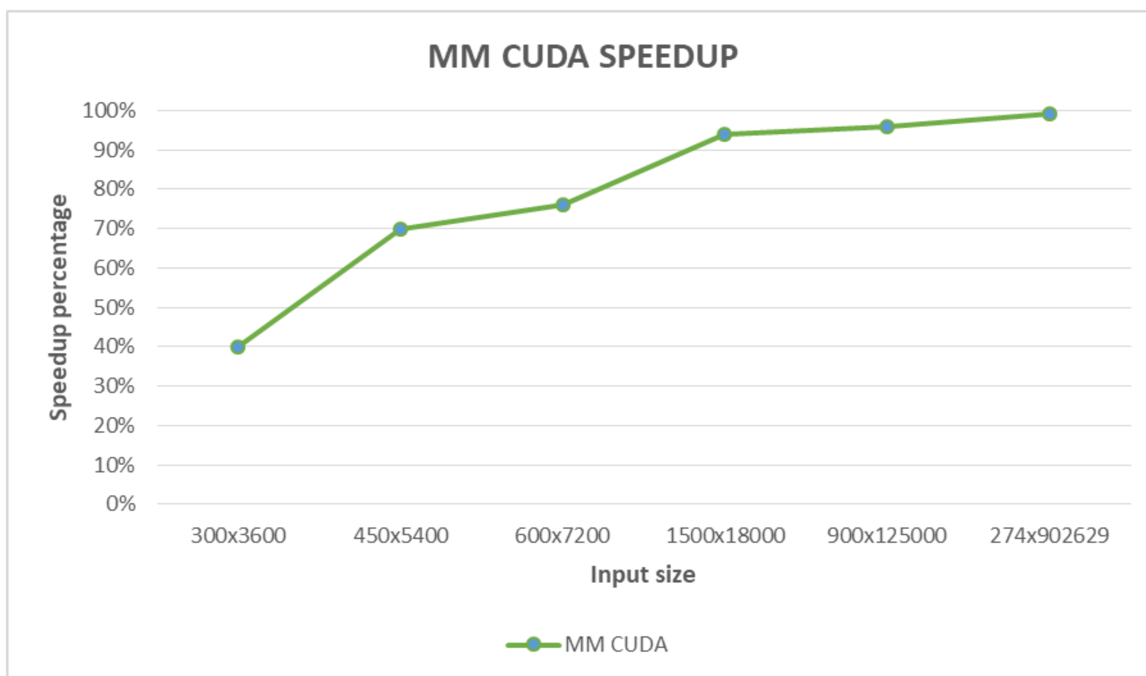
- Η νόρμα Frobenius υπολογίζεται με τη συνάρτηση cublas<t>nrm2 της βιβλιοθήκης cublas ($norm(\mathrm{Y}, \text{'frob'})$ στο MATLAB).
- Ο υπολογισμός τυχαίων τιμών που ακολουθούν κανονική κατανομή με μέση τιμή 0 και τυπική απόκλιση 1 γίνεται με τη συνάρτηση curandGenerateNormal και curandGenerateNormalDouble της βιβλιοθήκης curand για δεκαδικούς αριθμούς μονής και διπλής ακρίβειας αντίστοιχα ($D = rand(\mathrm{size1}, \mathrm{size2})$ στο MATLAB).

- Η δημιουργία του ταυτοτικού πίνακα I (1 σε όλα τα στοιχεία της διαγωνίου, 0 αλλού) γίνεται με συνάρτηση πυρήνα initIdentityGPU που ενημερώνει παράλληλα όλα τα στοιχεία ($I = diag(\mathrm{ones}(1,\mathrm{K}))$ στο MATLAB).
- Ο πολλαπλασιασμός πινάκων γίνεται με τη συνάρτηση cublas<t>gemm της βιβλιοθήκης cublas (στο MATLAB γίνεται με το τελεστή *). Είναι η καλύτερη για να γίνει ένας μεμονωμένος πολλαπλασιασμός. Η συνάρτηση gpu_blas_mmul που έχουμε κατασκευάσει και στηρίζεται στην cublas<t>gemm υποστηρίζει και πολλαπλασιασμό με τον ανάστροφο πίνακα, ανάλογα με τα ορίσματα που θα δοθούν.
- Η εύρεση ιδιοτιμών υπολογίζεται μέσω της συνάρτησης cusolverDn<t>syevd της βιβλιοθήκης cusolver ($eig(\mathrm{X})$ στο MATLAB).
- Η πρόσθεση και η αφαίρεση πινάκων γίνεται με απλές συναρτήσεις πυρήνα (addMatrices και subtractMatrices αντίστοιχα) που ενημερώνουν παράλληλα όλα τα στοιχεία του πίνακα.
- Η συνάρτηση του MATLAB $A = wthresh(\mathrm{A},\text{'s'},\mathrm{T})$ εφαρμόζει την τεχνική της ήπιας κατωφλίωσης (soft thresholding) στον πίνακα A σε σχέση με ένα όριο T εφαρμόζοντας τον τύπο $Y = sign(\mathrm{X})\cdot(|X| - T)_+$ όπου $(\mathrm{x})_+ = \begin{cases} 0, & \alpha\nu\ x < 0 \\ \mathrm{x}, & \alpha\nu\ x \geq 0 \end{cases}$. Σε CUDA έχουμε φτιάξει μια συνάρτηση πυρήνα wthresh η οποία εφαρμόζει την παραπάνω πράξη σε όλα τα στοιχεία του πίνακα παράλληλα.
- Οι εντολές $Kv = sqrt(\mathrm{sum}(\mathrm{B}.\hat{}\,2))$ και $Kv(\mathrm{Kv} < \mathrm{ccl}) = 1$ αποθηκεύουν σε κάθε θέση του πίνακα Kv την τετραγωνική ρίζα του αθροίσματος των τετραγώνων των στοιχείων της αντίστοιχης στήλης του πίνακα B. Κάθε τιμή του πίνακα Kv λαμβάνει την τιμή 1, αν είναι μικρότερη από ccl. Η συνάρτηση που έχει την αντίστοιχη λειτουργία στο πρόγραμμά μας είναι η normalizeReduction. Η συνάρτηση αυτή εφαρμόζει την τεχνική του reduction για να υπολογίσει το άθροισμα των τετραγώνων των στοιχείων των στηλών του B. Από τις τεχνικές εφαρμογής του reduction που δοκιμάσαμε, υλοποιήσαμε την shuffle on down που ήταν η ταχύτερη, ενώ δεν απαιτεί τη χρήση διαμοιραζόμενης μνήμης και συγχρονισμού ανάμεσα στα thread.
- Η συνάρτηση $B = bsxfun(@\mathrm{rdivide},\mathrm{B},\mathrm{Kv})$ του MATLAB κάνει διαίρεση στοιχείου με στοιχείο κάθε γραμμής του πίνακα B με τον πίνακα-γραμμή Kv και αποθηκεύει το αποτέλεσμα στον πίνακα B. Η συνάρτηση που έχει την αντίστοιχη λειτουργία στο πρόγραμμά μας είναι η rdivide, η οποία διαιρεί παράλληλα όλα τα στοιχεία του πίνακα B με το κατάλληλο στοιχείο του πίνακα Kv, ενώ κάνει χρήση και της διαμοιραζόμενης μνήμης ώστε να γίνουν ταχύτερα οι πράξεις.

Πίνακας 2: Σύγκριση χρόνων εκτέλεσης MM σε MATLAB με χρήση CPU και σε CUDA

ΜΕΓΕΘΟΣ	MATLAB CPU (sec)	CUDA (sec)	ΠΟΣΟΣΤΟ ΒΕΛΤΙΩΣΗΣ
300×3600	4,8782	2,9046	40,46%
450×5400	10,1033	3,019	70,12%
600×7200	14,8908	3,5299	76,29%
1500×18000	85,5759	5,4828	93,59%
900×125000	401,814	14,7895	96,32%
274×902629	3101,5717	43,2029	98,61%

Στην Εικόνα 22 βλέπουμε πως διαμορφώνεται η επιτάχυνση της υλοποίησης του αλγορίθμου MM με χρήση CUDA σε σχέση με την υλοποίηση του με χρήση MATLAB στη CPU, σύμφωνα με τους χρόνους του Πίνακα 2. Παρατηρούμε ότι η επιτάχυνση αυξάνεται όσο αυξάνεται το μέγεθος των δεδομένων εισόδου. Σε σχέση με την προηγούμενη υλοποίηση σε MATLAB με GPU βλέπουμε ότι δεν έχουμε σε κανένα σημείο επιβράδυνση, ενώ και ποσοστιαία οι επιταχύνσεις είναι μεγαλύτερες σε κάθε μέγεθος δεδομένων. Τέλος, και σε αυτήν την υλοποίηση παρατηρούμε ότι μεταξύ διαδοχικών μεγεθών δεδομένων εισόδου, η μεγαλύτερη είσοδος εκ των δύο προκαλεί όλο και μικρότερες μεταβολές στο ποσοστό επιτάχυνσης.

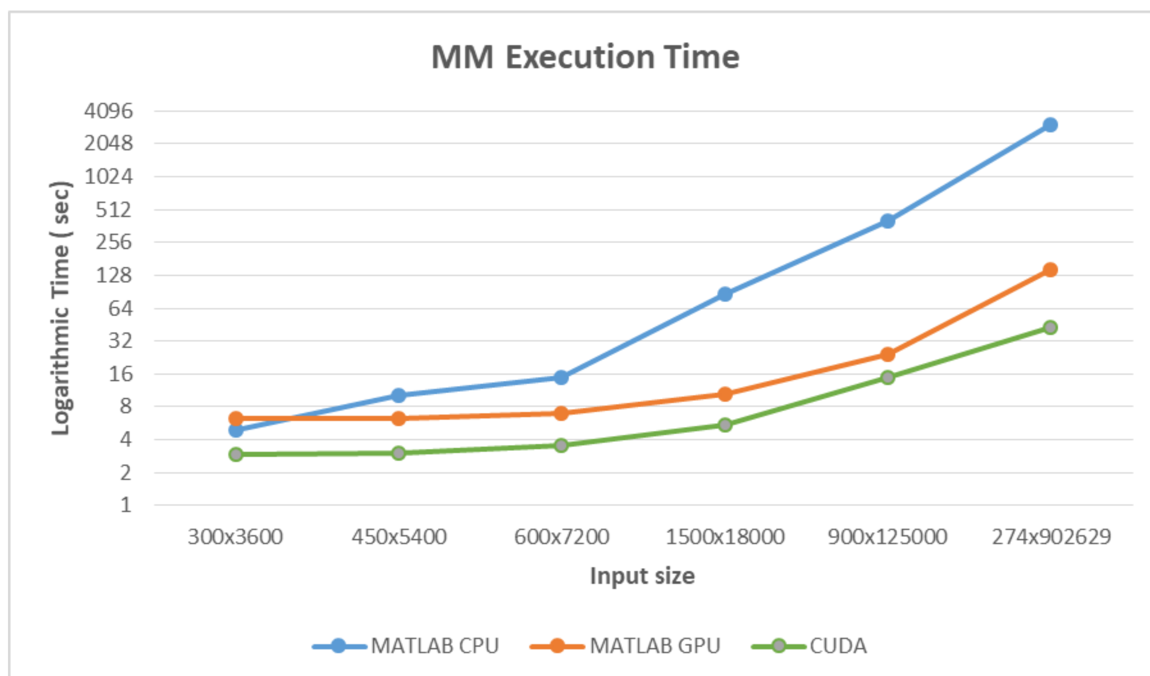


Εικόνα 22: Διαμόρφωση επιτάχυνσης αλγορίθμου MM σε CUDA σε σχέση με τα δεδομένα εισόδου

3.3.3 Σύγκριση των τριών υλοποιήσεων

Στην Εικόνα 23 φαίνονται οι συγκρίσεις των υλοποιήσεων του αλγορίθμου MM που παρουσιάστηκαν, σύμφωνα με τους χρόνους που υπολογίστηκαν στους Πίνακες 1 και 2. Λόγω της μεγάλης διαφοράς σε χρόνο εκτέλεσης μεταξύ της αρχικής υλοποίησης σε MATLAB με CPU και των άλλων δύο, επιλέχθηκε η λογαριθμική κλίμακα για να

αποτυπωθεί ο χρόνος, ώστε να υπάρχει μια ξεκάθαρη εικόνα των επιδόσεων των αλγορίθμων.



Εικόνα 23: Σύγκριση χρόνων τριών υλοποιήσεων MM

Συμπεραίνουμε ότι μεγαλώνοντας τα δεδομένα και οι δύο εκτελέσεις βελτιώνουν το χρόνο εκτέλεσης αυτού του αλγορίθμου. Παρόλα αυτά, η υλοποίηση σε CUDA είναι σταθερά καλύτερη για κάθε μέγεθος δεδομένων.

4. ΑΛΓΟΡΙΘΜΟΣ k-SVD

4.1 Εισαγωγή στη Μάθηση με Επίγνωση Αραιότητας (Sparsity-Aware Learning)

4.1.1 Εισαγωγή στο Υπερ-πλήρες λεξικό

Μια έννοια που έχουμε ήδη δει είναι αυτή του λεξικού (dictionary). Οι ιδιότητές του αξιοποιούνται κατά κόρον στο πεδίο της συμπιεστικής δειγματοληψίας (Compressive Sensing). Η χρήση του σε αυτό το πεδίο συνοψίζεται στην ιδέα πως ένα λεξικό αποτελεί μια συλλογή παραμετροποιημένων κυματομορφών, οι οποίες αποτελούν δείγματα σημάτων διακριτού χρόνου (για παράδειγμα θα μπορούσαν να είναι οι στήλες ενός DFT ή ενός DWT πίνακα μετασχηματισμού). Στην γενική περίπτωση, λοιπόν, θα αναφερόμαστε στο λεξικό ως μια συλλογή διανυσμάτων που δύνανται να χρησιμοποιηθούν για την περιγραφή ενός ή περισσοτέρων σημάτων του πραγματικού κόσμου.

Το Compressive Sensing αποτελεί μια μέθοδο επεξεργασίας σήματος που αποσκοπεί στην ανακατασκευή ενός σήματος μέσω της επίλυσης υπο-προσδιορισμένου συστήματος γραμμικών εξισώσεων (underdetermined system of linear equations). Με πιο απλά λόγια, αυτή η μέθοδος εκμεταλλεύεται το γεγονός πως τα σήματα στη φύση μπορούν να εκφραστούν πιο «συμπιεσμένα» σε μια κατάλληλη βάση που εξαρτάται από τη μορφή του εκάστοτε σήματος. Σε αντιστοιχία των παραπάνω, μια μαθηματική περιγραφή θα υπαγόρευε πως, εάν ένα σήμα S αποτελούμενο από L συνιστώσες κωδικοποιηθεί ως οι k πιο σημαντικές εξ' αυτών με την αποθήκευσή τους πίσω στο μετασχηματισμένο S (καθώς και των θέσεων αυτών), τότε κάποιος που διαθέτει αυτές τις k συνιστώσες (και θεωρώντας όλες τις άλλες μηδενικές) μπορεί να ανακτήσει (προσεγγιστικά) το αρχικό σήμα μέσω εξισώσεων ανάλυσης:

$$s = DS$$

όπου D είναι ένας πίνακας μετασχηματισμού (η διαδικασία μετασχηματισμού εδώ δεν είναι παρά μια προβολή ενός διανύσματος στο σύστημα συντεταγμένων των στηλών του D, γεγονός που φανερώνει τη σύνδεση με τις στήλες ενός λεξικού), S είναι ο παραπάνω πίνακας των μετασχηματισμένων συνιστωσών και s η προσέγγιση που λαμβάνουμε. Η έννοια του υπο-προσδιορισμένου συστήματος γραμμικών εξισώσεων εισάγεται όταν παρατηρήσουμε πως ο πίνακας S αποτελείται από πολλά μηδενικά στοιχεία, δηλαδή είναι αραιός (sparse). Μια προσέγγιση που εκμεταλλεύεται αυτό το γεγονός μάς είναι χρήσιμη όταν ανοίγει έναν διαφορετικό δρόμο ανάλυσης του ακατέργαστου σήματος μέσω του οποίου κάποιος χρειάζεται λιγότερες από L συνιστώσες προκειμένου να ανακτήσει ολόκληρη την πληροφορία. Αποδεικνύεται, όπως θα δούμε, πως η περίπτωση απόκτησης μόνο N σημάτων ($k < N << L$) είναι αξιοποιήσιμη και οδηγεί σε σύστημα με πολύ λίγα δεδομένα ως προς το πλήθος των αγνώστων στο σύστημα εξισώσεων και με τον περιορισμό ότι ο πίνακας στον οποίο στοχεύουμε είναι αραιός.

Καθώς η συσχέτιση του πίνακα D παραπάνω με το λεξικό είναι πλέον προφανής, μπορούμε να ορίσουμε 2 είδη λεξικών. Ένα λεξικό όπου έχει ίδιο αριθμό (ορθοκανονικών) διανυσμάτων L όσο είναι και το μήκος του σήματος εισόδου λέγεται πλήρες λεξικό. Βέβαια μια τέτοια θεώρηση είναι περιοριστική, καθώς τα σήματα του πραγματικού κόσμου δεν αποτελούνται εξ' ολοκλήρου από κομμάτια που θα μοντελοποιηθούν από το ίδιο σετ συνιστωσών (για παράδειγμα μια εικόνα έχει «άγρια» μέρη με έντονες ακμές ή πιο «ομαλά» χωρίς διαφορές). Εδώ εισάγουμε την έννοια του υπερ-πλήρους λεξικού το οποίο διαθέτει περισσότερους από L συντελεστές, τους οποίους αποκαλούμε άτομα (atoms). Με αυτόν τον τρόπο μπορούμε να επιτύχουμε ακριβέστερη μοντελοποίηση του σήματός μας.

Αξίζει να τονίσουμε σε αυτό το σημείο πως υπάρχουν περιπτώσεις που απλώς δεν είναι δυνατό να αποκτήσουμε μεγάλο αριθμό δειγμάτων (και να οδηγηθούμε σε τέτοια συστήματα) λόγω φυσικών ή τεχνικών περιορισμών. Μια τέτοια περίπτωση είναι το fMRI που εξετάζουμε στην παρούσα εργασία.

4.1.2 Η Μάθηση με λεξικό (Dictionary Learning)

Ένας τρόπος να βελτιώσουμε την απεικόνιση σημάτων με διαφορετικές ανάγκες μοντελοποίησης των επιμέρους μερών τους (όπως παραπάνω) μέσω λεξικού είναι μέσω της «εκπαίδευσής» του ώστε να περιγράψει καλύτερα τα δεδομένα σήματα του πραγματικού κόσμου. Το υπερ-πλήρες λεξικό υποθέσαμε πως διαθέτει έναν προκαθορισμένο αριθμό ατόμων που είναι και μεγαλύτερος που αριθμού L (π.χ. 2L) και τα άτομα αυτά είναι γνωστά. Τώρα, θα υιοθετήσουμε την περίπτωση που τα άτομα του λεξικού θα πρέπει να εκτιμηθούν από τα δεδομένα. Αυτή η προσέγγιση αποτελεί την «τυφλή» (blind) εκδοχή της διαδικασίας που περιγράψαμε έως τώρα.

4.2 Ο αλγόριθμος k-SVD

4.2.1 Συνοπτική περιγραφή προβλήματος

Ο αλγόριθμος k-SVD αποτελεί ένα εξαιρετικά διαδεδομένο παράδειγμα μάθησης με λεξικό. Προσπαθεί να επιλύσει το ακόλουθο πρόβλημα μάθησης με λεξικό (Dictionary Learning Task – DL task):

Έστω ότι l είναι οι τυχαίες μεταβλητές και εκφράζονται ως προς $m > l$ λανθάνουσες, σύμφωνα με το γραμμικό μοντέλο:

$$x = Az, \quad x \in \mathbb{R}^l, \; z \in \mathbb{R}^m$$

όπου A είναι ένας άγνωστος $l \times m$ πίνακας και $m >> l$. Είναι προφανές πως, ακόμα και να ήταν γνωστό το A, αυτό το σύστημα δεν έχει μοναδική λύση. Θα επιστρατεύσουμε, λοιπόν, περιορισμούς αραιότητας (sparsity constraints). Τότε, θα μετατραπεί στο ακόλουθο πρόβλημα βελτιστοποίησης:

Εάν A και Z οι πίνακες όπως παραπάνω για ένα δεδομένο σύνολο δεδομένων X τότε

όπως φαίνεται στο [1] έχουμε :

$$\begin{aligned} &\text{minimize with respect to } A, Z \quad \|X - AZ\|_F^2 \\ &\qquad\qquad\qquad \text{subject to } \|z_n\|_0 \le T_0, \quad n = 1, 2, ..., N, \end{aligned}$$

όπου T_0 μια τιμή κατωφλίου. Η επίλυση αυτού του προβλήματος γίνεται επαναληπτικά και σε 2 στάδια. Πρώτα κρατάμε σταθερό το λεξικό A και βελτιστοποιούμε τον αραιό πίνακα συνιστωσών Z. Έπειτα διατηρούμε σταθερές τις λανθάνουσες συνιστώσες Z και ανανεώνουμε το λεξικό A βελτιστοποιώντας ως προς τις στήλες του.

4.2.2 Ο αλγόριθμος

Στον αλγόριθμο k-SVD ακολουθούμε μια ελάχιστα διαφορετική προσέγγιση του παραπάνω προβλήματος. Η δεύτερη φάση, κατά την οποία διατηρούμε σταθερό το Z, δεν χρειάζεται κατ' ανάγκη να ανανεώνει μόνον το λεξικό, αλλά μπορεί να ενημερώνει και κάποιες τιμές του Z. Αυτή είναι μια πολύ σημαντική διαφορά ως προς τις υπόλοιπες

τεχνικές βελτιστοποίησης και φαίνεται να οδηγεί σε βελτίωση της απόδοσης του αλγορίθμου στην πράξη.

Τα στάδια που διαμορφώνουν τον k-SVD παρουσιάζονται παρακάτω:

- *Στάδιο 1*: Υποθέτουμε γνωστό το λεξικό A και έτσι το πρόβλημα βελτιστοποίησης γίνεται όπως παρακάτω:

$$\min_{z_n} \left\| x_n - A z_n \right\|^2, \quad \text{s.t.} \quad \left\| z_n \right\|_0 \le T_0, \quad n = 1, 2, ..., N$$

  Παρατηρούμε ότι αυτό είναι ένα πρόβλημα περιορισμένο από τον βαθμό αραιότητας που ορίζει το κατώφλι T_0 (Sparsity-constrained k-SVD) και το οποίο έχει υλοποιηθεί στα πλαίσια του fMRI προβλήματος. Αυτή η διαδικασία μπορεί να ανατεθεί σε διάφορους γνωστούς λύτες ελαχιστοποίησης (minimization solvers), όπως ο OMP (Orthogonal Matching Pursuit). Αυτό το στάδιο το ονομάζουμε *sparse coding*.

- *Στάδιο 2:* Σε αυτό το στάδιο έχουμε αποκτήσει από την προηγούμενη επανάληψη τον πίνακα Z και θέλουμε να ενημερώσουμε το λεξικό A καθώς και κάποια στοιχεία του Z. Το πρόβλημα αυτή τη φορά γίνεται η ελαχιστοποίηση της ποσότητας

$$\left\| E_k - a_k z_k^{rT} \right\|_F^2,$$

  όπου

$$E_k := X - \sum_{i=1, i \neq k}^{m} a_i z_i^{rT}.$$

  όπου a_k είναι η στήλη του λεξικού που ενημερώνουμε αυτή τη στιγμή και z_k^{rT} είναι μια γραμμή του πίνακα Z. Στην ουσία, η παραπάνω πράξη δηλώνει ότι αναζητούμε την καλύτερη, ως προς την Frobenious νόρμα, τάξης-1 προσέγγιση του E_k. Παρατηρούμε, επίσης, πως σε κάθε τέτοια πράξη υπολογισμού του E_k συμμετέχουν όλα τα προηγουμένως υπολογισμένα διανύσματα με δείκτες από i έως και k-1 και κατά συνέπεια χρησιμοποιούνται κάθε φορά οι πιο ενημερωμένες πληροφορίες. Επίσης, αξίζει να αναφέρουμε πως σε αυτό το πρόβλημα ελαχιστοποίησης λύνουμε και ως προς την στήλη του λεξικού και ως προς την γραμμή του Z. Αυτές οι δύο ποσότητες θα ενημερωθούν ταυτόχρονα στο ίδιο στάδιο. Η γνωστότερη μέθοδος επίλυσης της παραπάνω ποσότητας είναι η μέθοδος SVD (Singular Value Decomposition) και θα εφαρμοστεί στο E_k. Τέλος, το στάδιο αυτό ονομάζεται *codebook update* ή *dictionary update stage.*

Μια σημαντική τροποποίηση είναι αναγκαία, όμως, στο τελευταίο στάδιο. Η εκτέλεση SVD δεν μπορεί να μας εγγυηθεί ότι η αραιότητα που επιβάλαμε στο 1° στάδιο θα διατηρηθεί και στο 2° στάδιο. Εύκολη λύση σε αυτό το πρόβλημα είναι να εστιάσουμε μόνο στο σύνολο με τις συνιστώσες που είναι μη-μηδενικές. Η αλλαγή που πρέπει να εφαρμοστεί λοιπόν συνίσταται στην αναζήτηση και αποθήκευση των θέσεων των μη-μηδενικών στοιχείων κάθε γραμμής του Z σε ένα σύνολο Ω. Χρησιμοποιώντας αυτό το σύνολο μπορούμε να κατασκευάσουμε ένα νέο διάνυσμα γραμμής με μικρότερη διάσταση από το αρχικό που περιέχει μόνο τα στοιχεία που βρήκαμε παραπάνω. Όλες

οι πράξεις που παρουσιάστηκαν στο 2ο στάδιο θα χρησιμοποιήσουν τώρα διανύσματα γραμμής μικρότερου μεγέθους και θα παράξουν έναν επίσης μικρότερης τάξης πίνακα E_k. Έπειτα, θα εκτελέσουμε σε αυτόν SVD και σε κάθε επανάληψη το σφάλμα θα μικραίνει. Τελικά ο αλγόριθμος θα συγκλίνει σε ένα τοπικό ελάχιστο.

Συνοπτικά τα βήματα που εκτελεί ο αλγόριθμος όπως παρουσιάζονται στο [1] διαμορφώνονται όπως στην παρακάτω εικόνα:

- Initialize $A^{(0)}$ with columns normalized to unit ℓ_2 norm.
- Set $i = 1$.
- *Stage 1*: Solve the optimization task to obtain the sparse coding representation vectors, z_n, $n = 1, 2, \ldots, N$; use any algorithm developed for this task.
- *Stage 2*: For any column, $k = 1, 2, \ldots, m$, in $A^{(i-1)}$, update it according to the following:
  - Identify the locations of the nonzero elements in the kth row of the computed, from stage 1, matrix Z.
  - Select the columns in X, which correspond to the locations of the nonzero elements of the kth row of Z and form a reduced order error matrix, $\tilde{E}_k$.
  - Perform SVD on $\tilde{E}_k$: $\tilde{E}_k = UDV^T$.
  - Update the kth column of $A^{(i)}$ to be the eigenvector corresponding to the largest singular value, $\boldsymbol{a}_k^{(i)} = \boldsymbol{u}_1$.
  - Update Z, by embedding in the nonzero locations of its kth row, the values $D(1,1)\boldsymbol{v}_1^T$.
- Stop if a convergence criterion is met.
- If not, $i = i + 1$, and continue.

Εικόνα 24: Συνοπτικός ψευδοκώδικας των βημάτων του αλγορίθμου k-SVD

4.2.3 Συσχέτιση με το πρόβλημα του fMRI

Ήδη έχουμε παρουσιάσει αναλυτικά το πρόβλημα του fMRI και τον στόχο που προσπαθούμε να επιτύχουμε για την επίλυσή του. Εδώ θα εστιάσουμε περισσότερο στον τρόπο χρήσης του k-SVD για αυτό το σκοπό. Έστω πως έχουμε ένα σκανάρισμα fMRI που έχει ανάλυση 64×64×48. Κάθε τέτοια τιμή αντιστοιχεί σε «κύβο» από το υποθετικό πλέγμα που καλύπτει τον εγκέφαλο αν τον χωρίζαμε με βάση τις παραπάνω διαστάσεις. Αυτούς τους κύβους και κατά συνέπεια τις αντίστοιχες τιμές θα τα ονομάσουμε *voxels.* Εάν αυτές οι τιμές μετασχηματιστούν σε έναν μονοδιάστατο πίνακα και επαναλάβουμε αυτό το πείραμα για ένα αριθμό TIMECOMPS φορών (π.χ. δευτερόλεπτα ή λεπτά) τότε αυτά τα δεδομένα μπορούν να οργανωθούν σε έναν πίνακα X μεγέθους TIMECOMP × VOXELS. Με αυτό τον τρόπο κάθε γραμμή αντιπροσωπεύει τον πίνακα ενεργοποίησης ή χωρικό χάρτη όλων των voxels για μια δεδομένη χρονική στιγμή. Αντίστοιχα, μια στήλη του πίνακα X δηλώνει τη χρονική εξέλιξη της ενεργοποίησης (ή μη ενεργοποίησης) ενός συγκεκριμένου voxel.

Η καταλληλότητα του k-SVD για το συγκεκριμένο πρόβλημα διαφαίνεται στην παρατήρηση πως η διαδικασία του πειράματος μπορεί να μοντελοποιηθεί από την παρακάτω παραγοντοποίηση του πίνακα δεδομένων:

$$X = \sum_{j=1}^{m} a_j z_j^T := AZ$$

όπου A είναι πίνακας του οποίου οι στήλες a_j δηλώνουν τη χρονική ακολουθία ενεργοποίησης για μια ομάδα voxels που ονομάζουμε FBN (Functional Brain Network), ενώ ο Z είναι ένας αραιός πίνακας από λανθάνουσες μεταβλητές του οποίου κάθε γραμμή z_j δηλώνει τον χωρικό χάρτη του j-οστού FBN και έχει μη-μηδενικές τιμές μόνο στα voxels που σχετίζονται με το συγκεκριμένο FBN.

Μετά την εφαρμογή του αλγορίθμου στα δεδομένα X θα αποκτήσουμε μια προσέγγιση των παραπάνω πινάκων A και Z όπως έχει ήδη παρουσιαστεί προηγούμενα. Η ελπίδα μας, λοιπόν, είναι να ανιχνεύσουμε ποια FBN σχετίζονται μεταξύ τους ως προς την χωρική, αλλά και χρονική τους ενεργοποίηση, όταν όλη αυτή η δραστηριότητα προέρχεται από ένα ή περισσότερα ερεθίσματα και γίνεται με ελεγχόμενο και πειραματικό τρόπο. Τέλος, μας επιτρέπει να εκτιμήσουμε την σχέση νοητικών και φυσικών ερεθισμάτων με την εγκεφαλική δραστηριότητα που προκαλούν.

4.3 Η παραλλαγή του k-SVD και το Batch-OMP

Όλες μας οι μετρήσεις που έγιναν σε CPU περιλάμβαναν την προσεγγιστική έκδοση του k-SVD μαζί με την μέθοδο Batch-OMP στην περίπτωση του sparse coding, όπως έχουν παρουσιαστεί στο [17]. Ο αντίστοιχος κώδικας MATLAB® είναι διαθέσιμος προς όλους στη διεύθυνση http://www.cs.technion.ac.il/~ronrubin/software.html. Οι διαφορές της συγκεκριμένης υλοποίησης από τον κλασσικό αλγόριθμο εντοπίζονται κυρίως σε δύο σημεία: αφενός στο κομμάτι του *OMP* και αφετέρου στην ενημέρωση του λεξικού στα τελευταία βήματα του *codebook update* σταδίου.

Ο αρχικός αλγόριθμος του OMP συνοψίζεται στα εξής βήματα, όπως στην παρακάτω εικόνα:

Algorithm 1 ORTHOGONAL MATCHING PURSUIT

1: Input: Dictionary $\mathbf{D}$, signal $\underline{x}$, target sparsity K *or* target error ϵ
2: Output: Sparse representation $\underline{\gamma}$ such that $\underline{x} \approx \mathbf{D}\underline{\gamma}$
3: Init: Set $I := (\,)$, $\underline{r} := \underline{x}$, $\underline{\gamma} := \underline{0}$
4: **while** (*stopping criterion not met*) **do**
5: $\quad \hat{k} := \operatorname{Argmax}_k |\underline{d}_k^T \underline{r}|$
6: $\quad I := (I, \hat{k})$
7: $\quad \underline{\gamma}_I := (\mathbf{D}_I)^+ \underline{x}$
8: $\quad \underline{r} := \underline{x} - \mathbf{D}_I \underline{\gamma}_I$
9: **end while**

Εικόνα 25: Ο ψευδοκώδικας της διαδικασίας OMP

Η παραλλαγή του Batch-OMP βασίζεται στην παρατήρηση πως σε κάθε επανάληψη δεν χρειάζεται να γνωρίζουμε επ' ακριβώς το $\underline{r}$ ή το $\underline{\gamma}$ αλλά μόνον το $D^T\underline{r}$. Αντικαθιστούμε λοιπόν τον υπολογισμό αυτόν με μια ισοδύναμη σχέση που περιλαμβάνει έναν πίνακα $G = D^TD$ και τον πίνακα D^TX (πολλαπλασιασμός του λεξικού με τα δεδομένα). Αυτές οι δύο πράξεις παρουσιάζουν το βασικό πλεονέκτημα του προ-υπολογισμού τους έξω από την επανάληψη και κατά συνέπεια αφήνουν πολύ μικρότερου φόρτου εργασία προς εκτέλεση από την επανάληψη. Για μεγάλο όγκο δεδομένων αυτό αποτελεί σημαντικότατο όφελος στην απόδοση του OMP. Η ισοδυναμία και η τελική μορφή της πράξης φαίνεται παρακάτω:

Denoting $\underline{a} = D^T \underline{r}, \underline{a}^0 = D^T \underline{x}$, and $G = D^T D$, we can write:

$$\begin{aligned} \underline{a} &= D^T (\underline{x} - D_I (D_I)^+ \underline{x}) \\ &= \underline{a}^0 - G_I (D_I)^+ \underline{x} \\ &= \underline{a}^0 - G_I (D_I^T D_I)^{-1} D_I^T \underline{x} \\ &= \underline{a}^0 - G_I (G_{I,I})^{-1} \underline{a}^0{}_I \end{aligned}$$

Αξίζει να σημειώσουμε εδώ πως ο υπολογισμός του $(G_{I,I})^{-1}$ γίνεται μέσω προοδευτικής παραγοντοποίησης Cholesky, όπως αναλύεται στο [17]. Αυτή τη μέθοδο υλοποιούμε τελικά και εμείς στο πρόγραμμά μας.

Η δεύτερη διαφοροποίηση βρίσκεται στην αποφυγή εκτέλεσης της διαδικασίας SVD στο 2ο στάδιο του αλγορίθμου. Πρόκειται αναμφισβήτητα για μια εξαιρετικά αργή πράξη και με ελάχιστα περιθώρια βελτιστοποίησης στο μέλλον. Σε αυτό το κλίμα προτιμούμε να υπολογίζουμε στη θέση του τις εξής τιμές:

$$\begin{aligned} \underline{d} &:= E_{\underline{g}} / \left\| E_{\underline{g}} \right\|_2 \\ \underline{g} &:= E^T \underline{d} \end{aligned}$$

όπου $\underline{d}$ και $\underline{g}$ είναι τα διανύσματα που θα αντικαταστήσουν (ενημερώσουν) την τρέχουσα στήλη του πίνακα D και την τρέχουσα γραμμή του πίνακα Γ (ή πίνακα Z παραπάνω) αντίστοιχα για την δεδομένη επανάληψη. Είναι, δηλαδή, υποκατάστατα της εξόδου του SVD και έχει αποδειχθεί πως τελικά θα συγκλίνει σε μια βέλτιστη λύση ενώ συνεχίζει να μειώνει το σφάλμα με πολύ καλή προσέγγιση του αρχικού αλγορίθμου. Φυσικά αυτή την τεχνική υλοποιήσαμε στα πλαίσια του k-SVD. Συνοπτικά ο προσεγγιστικός αλγόριθμος συνοψίζεται στην Εικόνα 26.

4.4 Υλοποίηση στην αρχιτεκτονική CUDA

Όπως έχει αναφερθεί, ο κώδικας που χρησιμοποιήσαμε ως βάση προκειμένου να πραγματοποιήσουμε την επιτάχυνση του αλγορίθμου ανήκει στον Rubinstein και είναι διαθέσιμος διαδικτυακά ως toolbox για την πλατφόρμα MATLAB®. Ωστόσο, η αρχιτεκτονική CUDA προστάζει διαφορετικό χειρισμό των δεδομένων από την σκοπιά της παραλληλίας. Καθίσταται αναγκαία, επίσης, μια προσέγγιση χαμηλότερου επιπέδου στην χρήση αυτής σε συνδυασμό με τις διαθέσιμες λειτουργίες που προσφέρει το CUDA Runtime API σε σχέση με το MATLAB®. Τα επιμέρους «τεχνάσματα» που κρίθηκαν απαραίτητα για την επίτευξη ορθής παράλληλης υλοποίησης εξετάζονται ξεχωριστά στα επόμενα υπό-κεφάλαια. Θα αναφέρουμε σε αυτό το σημείο πως ο κώδικας CUDA για τον k-SVD βρίσκεται στο Παράρτημα ΙΙ, στην ενότητα με τίτλο «Αλγόριθμος KSVD». Οι παράμετροι iternum, Tdata και muthresh διατηρήθηκαν όπως στον αρχικό κώδικα με αντίστοιχα ονόματα NUMBERofITERATIONS, Tdata και muTHRESH καθώς επίσης προστέθηκε η δυνατότητα να οριστεί (με #define) η παράμετρος DOUBLE. Ο ορισμός (ή μη-ορισμός) αυτής της παραμέτρου δηλώνει εάν θέλουμε να εκτελέσουμε το πρόγραμμα με double precision (διπλή ακρίβεια) ή single precision (μονή ακρίβεια) αντίστοιχα.

Algorithm 4 APPROXIMATE K-SVD

1: Input: Signal set $\mathbf{X}$, initial dictionary $\mathbf{D}_0$, target sparsity K, number of iterations k.
2: Output: Dictionary $\mathbf{D}$ and sparse matrix $\mathbf{\Gamma}$ such that $\mathbf{X} \approx \mathbf{D\Gamma}$
3: Init: Set $\mathbf{D} := \mathbf{D}_0$
4: **for** $n = 1 \ldots k$ **do**
5: $\quad \forall i: \quad \mathbf{\Gamma}_i := \underset{\underline{\gamma}}{\text{Argmin}} \|\underline{x}_i - \mathbf{D}\underline{\gamma}\|_2^2 \quad$ Subject To $\quad \|\underline{\gamma}\|_0 \leq K$
6: $\quad$ **for** $j = 1 \ldots L$ **do**
7: $\quad\quad \mathbf{D}_j := \underline{0}$
8: $\quad\quad I := \{$*indices of the signals in* $\mathbf{X}$ *whose representations use* $\underline{d}_j\}$
9: $\quad\quad \underline{g} := \mathbf{\Gamma}_{j,I}^T$
10: $\quad\quad \underline{d} := \mathbf{X}_I\underline{g} - \mathbf{D\Gamma}_I\underline{g}$
11: $\quad\quad \underline{d} := \underline{d}/\|\underline{d}\|_2$
12: $\quad\quad \underline{g} := \mathbf{X}_I^T\underline{d} - (\mathbf{D\Gamma}_I)^T\underline{d}$
13: $\quad\quad \mathbf{D}_j := \underline{d}$
14: $\quad\quad \mathbf{\Gamma}_{j,I} := \underline{g}^T$
15: $\quad$ **end for**
16: **end for**

Εικόνα 26: Ο ψευδοκώδικας του προσεγγιστικού (Approximate) k-SVD

4.4.1 Επιτάχυνση του Batch-OMP

Όπως εύκολα θα διαπιστώσει κανείς με μια ματιά στον αρχικό αλγόριθμο του Batch-OMP, η διαδικασία που ακολουθείται δεν είναι παρά μια επαναληπτική εξέταση μίας κάθε φορά στήλης του αρχικού σήματος X, από την οποία εξάγουμε στο τέλος μια αραιή αναπαράστασή της. Δεδομένου ότι δεν υπάρχει εξάρτηση μεταξύ των επαναλήψεων, αναθέτουμε σε ένα CUDA Kernel το διαμοιρασμό της διαδικασίας του OMP σε πλήθος νημάτων (threads) όσο και το πλήθος των στηλών του πίνακα X. Έτσι επιτυγχάνουμε παράλληλη επεξεργασία κάθε στήλης και η απόδοση του OMP, σε μια κάρτα γραφικών με αρκετούς πόρους, μπορεί να βελτιωθεί ανάλογα του αριθμού των στηλών που εξετάζουμε. Η εσωτερική επανάληψη, φυσικά, δεν μπορεί να παραληφθεί, καθώς πρέπει να εξασφαλίσουμε τη σύγκλιση στο επιθυμητό επίπεδο αραιότητας.

Οι εσωτερικές πράξεις του αλγορίθμου επιδέχονται και αυτές περαιτέρω βελτιώσεις σε ένα πλαίσιο παραλληλίας όπως η CUDA. Διαπιστώσαμε, όμως, ότι η προσπάθεια παραλληλοποίησης εσωτερικών πράξεων είχε ως αποτέλεσμα τη χειρότερη απόδοση του προγράμματος. Αυτό το συμπέρασμα έρχεται σε συμφωνία με το πεπερασμένο των δυνατοτήτων της εκάστοτε κάρτας γραφικών και το πεπερασμένο των πόρων που μπορεί να διαθέσει ώστε να διατηρήσει την παραλληλία. Μια πρόχειρη ασυμπτωτική ανάλυση στο μυαλό μας θα καταστήσει προφανές το γεγονός ότι ο αριθμός των στηλών

του X αυξάνει σε πολύ μεγαλύτερο βαθμό σε σχέση με τον αριθμό των γραμμών στις πραγματικές μετρήσεις δεδομένων fMRI. Τελικά, λοιπόν, είναι συμφέρον να «επενδύσει» κανείς στην παραλληλοποίηση αυτής της ποσότητας εργασιών παρά στην παραλληλοποίηση σαφώς μικρότερων εσωτερικών διεργασιών.

Οι ενέργειες που τελικά αξίζει να αναφερθούν και αφορούν το εσωτερικό του OMP, ενώ ταυτόχρονα διαφοροποιούνται από την αρχική έκδοση είναι οι εξής:

- Δεδομένου ότι όλες οι στήλες του σήματος X εξετάζονται παράλληλα, θα πρέπει να δεσμεύσουμε ενδιάμεσες μεταβλητές που χρειάζεται αυτό το στάδιο τόσες φορές όσο και το πλήθος των στηλών. Για παράδειγμα, εάν έχουμε έναν προσωρινό πίνακα μεγέθους 100 και εκτελούμε 2.000 παράλληλα νήματα, τότε θα χρειαστούμε έναν buffer μεγέθους 100 × 2000 δεσμευμένο από την αρχή του προγράμματος. Εξασφαλίζουμε ταυτόχρονα την αποφυγή αλλοίωσης δεδομένων λόγω race condition (συναγωνισμού δεδομένων). Κάθε νήμα θα μπορεί πλεόν να χρησιμοποιεί την αποκλειστική μνήμη που του αντιστοιχεί μέσω κατάλληλης μετατόπισης (offset) που θα υπολογίσει μέσω του προσωπικού threadID του (βλ. Κεφάλαιο 3). Αυτό οδηγεί αναπόφευκτα σε σημαντικά αυξημένη χρήση μνήμης από το στάδιο αυτό.

- Στο τέλος του OMP θα παραχθεί ο αραιός πίνακα Z ή αραιός πίνακας Γ που περιέχει τις λανθάνουσες συνιστώσες, όπως παρήχθησαν από τον αλγόριθμο. Παρόλο που στην περίπτωση του σειριακού κώδικα είναι συμφέρον να αποθηκευτεί αυτός ο πίνακας ως sparse, δηλαδή να κρατηθούν μόνο τα μη-μηδενικά στοιχεία του, από πλευράς μνήμης αλλά και απόδοσης σε ορισμένες πράξεις, η CUDA έκδοση γίνεται εξαιρετικά πολύπλοκη και συνεπώς ασύμφορη, όταν καλείται να εκτελέσει πράξεις σε μνήμη που δεν είναι ενιαία και ιδανικά συνεχόμενη. Για το λόγο αυτό, αργά ή γρήγορα, ο πίνακας θα εκφυλιστεί σε πίνακα πλήρους τάξης (full-rank) πριν χρησιμοποιηθεί στο πρώτο κομμάτι του dictionary update σταδίου.

Όλες οι υπόλοιπες πράξεις υλοποιούνται σειριακά για τους λόγους που αναφέραμε παραπάνω συμπεριλαμβανομένου του πολλαπλασιασμού διανύσματος με πίνακα, της εύρεσης της θέσης της μέγιστης (απόλυτης) τιμής ενός πίνακα, του υπολογισμού του αθροίσματος των τετραγώνων των τιμών ενός πίνακα, καθώς και της παραγοντοποίησης Cholesky σε συμφωνία με τον κώδικα για CPU.

4.4.2 Επιτάχυνση του Σταδίου Ενημέρωσης Λεξικού (Dictionary Update)

Το στάδιο αυτό ενημερώνει την εκάστοτε στήλη του λεξικού, καθώς και την αντίστοιχη γραμμή του πίνακα αραιής αναπαράστασης (sparse representation) Z ή Γ. Αυτό επιτυγχάνεται επαναληπτικά για όλες τις στήλες του λεξικού και κάθε ενημερωμένο διάνυσμα από κάποια επανάληψη θα πρέπει να χρησιμοποιηθεί στην επόμενη. Βλέπουμε, λοιπόν, την εξάρτηση δεδομένων ανάμεσα στις επαναλήψεις και την αδυναμία πλήρους εκμετάλλευσης της διαθέσιμης παραλληλίας. Αφού η επανάληψη δεν μπορεί βεβαίως να εξαλειφθεί, δοκιμάζουμε τεχνικές βελτίωσης στο εσωτερικό της.

Οι ενέργειες που τελικά αξίζει να αναφερθούν και αφορούν το εσωτερικό του σταδίου αυτού ενώ ταυτόχρονα διαφοροποιούνται από την αρχική έκδοση είναι οι εξής:

- Η πρώτη ενέργεια του dictionary update είναι η εύρεση των μη-μηδενικών στοιχείων της γραμμής του αραιού πίνακα Z που εξετάζουμε κάθε φορά. Αυτή η πράξη είναι ανεξάρτητη των προηγούμενων επαναλήψεων και γενικά μπορεί να γίνει παράλληλα για όλες τις γραμμές και μάλιστα εκτός επανάληψης. Εκκινώντας ένα νήμα για κάθε θέση του πίνακα μπορούμε ταυτόχρονα, πολύ γρήγορα και αποδοτικά, να εξάγουμε το αποτέλεσμά μας. Έπειτα αρκεί σε κάθε επανάληψη να ανασύρουμε αυτό το αποτέλεσμα από την κατάλληλη θέση του πίνακα.

- Όλες οι πράξεις που περιλαμβάνουν πολλαπλασιασμό πίνακα με διάνυσμα έχουν υλοποιηθεί με τη χρήση της τεχνικής reduction, η οποία υπολογίζει αποδοτικά μια τιμή από ένα σύνολο δεδομένων (π.χ. νόρμα, άθροισμα, μέγιστο ή ελάχιστον κ.ά.). Η CUDA παρέχει εργαλεία για την ταχύτατη εκτέλεση τέτοιων εργασιών και, αφού ο πολλαπλασιασμός πίνακα-διανύσματος είναι παράλληλα αθροίσματα πολλαπλασιασμών ανά στοιχείο, επιτυγχάνουμε ακόμη μεγαλύτερη απόδοση χρησιμοποιώντας παράλληλα reductions.

- Όλες οι πράξεις που περιλαμβάνουν πολλαπλασιασμό πίνακα με πίνακα έχουν διεκπεραιωθεί με την βοήθεια της βιβλιοθήκης γραμμικής άλγεβρας CUBLAS από την Nvidia. Αυτή η επιλογή έγινε με γνώμονα την εξαιρετική βελτιστοποίηση της βιβλιοθήκης αυτής για τη συγκεκριμένη πράξη.

- Παράλληλα επίσης εκτελούμε την πράξη πολλαπλασιασμού του διανύσματος των μη-μηδενικών (της εκάστοτε γραμμής) στοιχείων με τον εαυτό τους, αφού γνωρίζουμε το αποτέλεσμα του 1ου βήματος, όπως περιγράψαμε παραπάνω και μάλιστα έξω από την επανάληψη. Επιπλέον η πράξη:

  $$X_I \underline{g}$$

  δεν εξαρτάται από την εκάστοτε επανάληψη και μπορεί να υπολογιστεί έξω από αυτή. Λόγω του ότι οι δοκιμές μας έγιναν σε κάρτα γραφικών με αυξημένη υπολογιστική ικανότητα, καταφέραμε να κάνουμε αυτές τις δυο πράξεις να εκτελούνται και μεταξύ τους παράλληλα ώστε να επιτύχουμε ακόμη υψηλότερο utilization της κάρτας.

- Το νέο άτομο που παράγει αυτό το στάδιο θα πρέπει να διαιρεθεί με τη νόρμα του πριν αντικαταστήσει το παλιό άτομο στο λεξικό. Αυτό αποτελεί πράξη που μπορεί να παραλληλοποιηθεί και να είναι αποδοτική σε CUDA με χρήση του reduction που ήδη αναφέραμε.

- Η ακόλουθη πράξη από την ενημέρωση της γραμμής του αραιού πίνακα Z:

  $$X_I^T \underline{d}$$

  δύναται να επιταχυνθεί με χρήση CUDA, εάν ανοίξουμε ένα νήμα για κάθε στοιχείο και όλα μαζί να κατασκευάσουν το αποτέλεσμα. Αυτή η εργασία απαιτεί επίσης την εκκίνηση δισδιάστατου πλέγματος μπλοκ και νημάτων και επιτυγχάνει τη χρήση μεγάλου ποσοστού των πόρων του συστήματος.

Γενικά είναι εύκολο να διαπιστώσει κανείς πως αυτό το στάδιο δεν επιδέχεται τόση παραλληλία όση το προηγούμενο στάδιο και είναι σαφώς το αργότερο εκ των δύο.

4.4.3 Επιτάχυνση του k-SVD

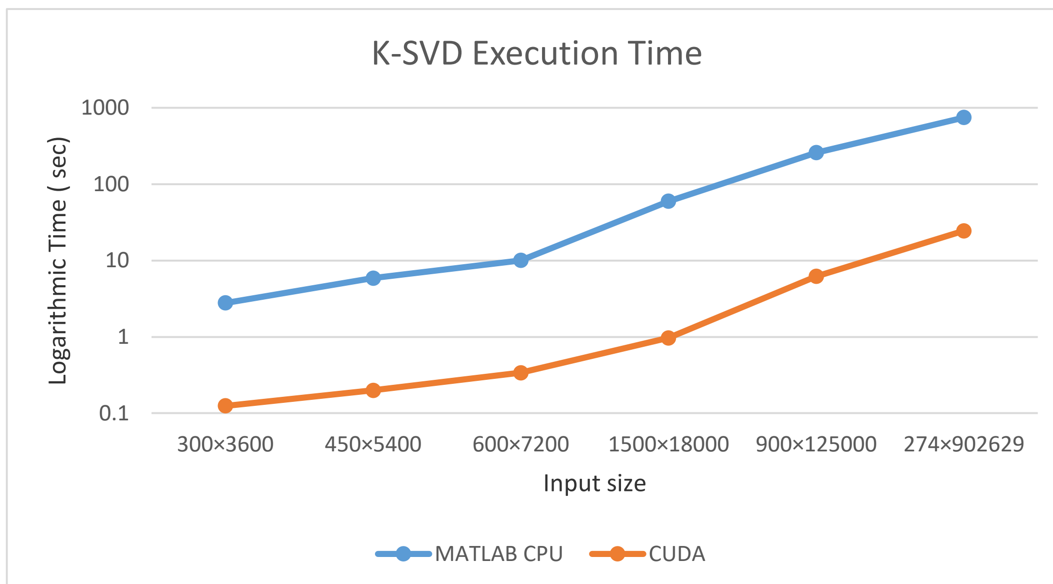
Στο σύνολό του ο αλγόριθμος για την επιτάχυνσή του θα επιστρατεύσει τα δύο παραπάνω ήδη βελτιστοποιημένα στάδια. Θα πρέπει να τονίσουμε σε αυτό το σημείο πως για τα επιμέρους αποτελέσματά του, ο k-SVD δεσμεύει χώρο εξ' ολοκλήρου στην μνήμη της συσκευής και όλα τα παραγόμενα δεδομένα δεν μεταφέρονται ανάμεσα στη CPU και την GPU ή αντίστροφα. Το πλεονέκτημα από αυτό το γεγονός είναι πως αποφεύγουμε τις μεταφορές από και προς τη συσκευή, οι οποίες είναι γνωστό πως είναι σημαντικά αργές και μπορούν να επηρεάσουν σε εξαιρετικό βαθμό την απόδοση ενός προγράμματος CUDA. Τέλος, μετά την ολοκλήρωση του τελευταίου σταδίου, ο κώδικας που διαθέτουμε σε CPU κάνει μια εκκαθάριση στο λεξικό προκειμένου να απορρίψει άτομα που δεν χρησιμοποιούνται ή έχουν μεγάλο σφάλμα ή δεν χρησιμοποιούνται από καμία γραμμή του αραιού πίνακα συνιστωσών Z. Ακολουθούμε ακριβώς την ίδια υλοποίηση μεταποιημένη για την αρχιτεκτονική CUDA.

4.5 Επιδόσεις υλοποίησης

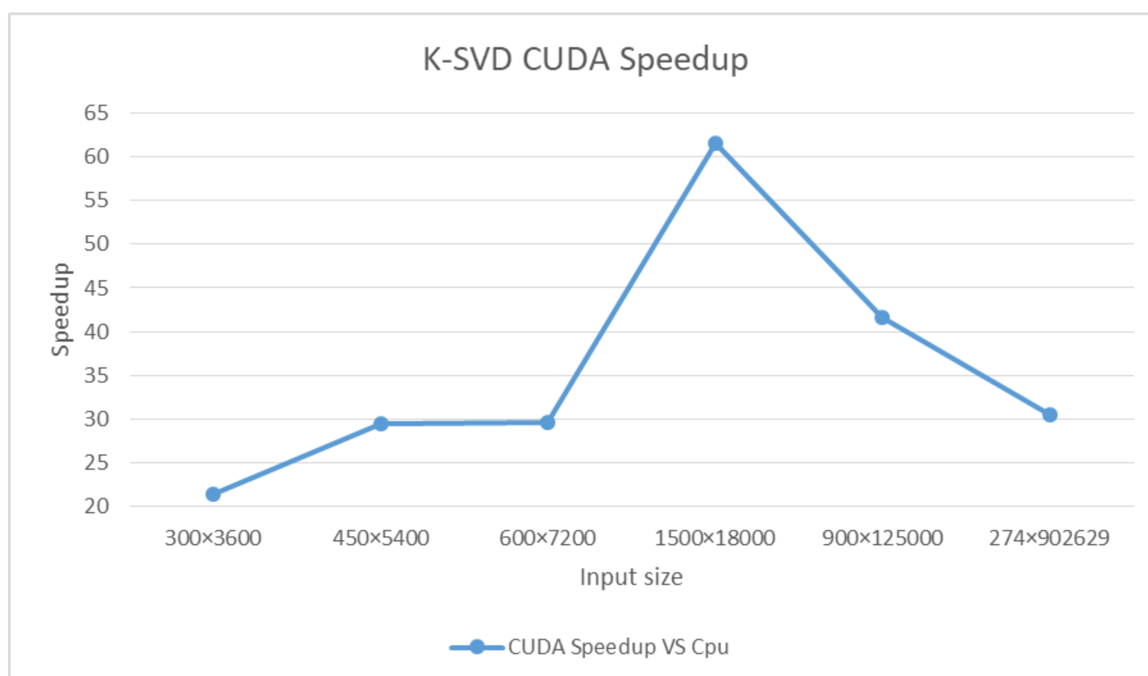
Στα παρακάτω διαγράμματα συνοψίζεται η απόδοση του προγράμματος για δεδομένα διαφορετικού μεγέθους. Χαρακτηριστικά εκτέλεσης: Iterations: 30, Sparsity level threshold: 6, Mutual incoherence threshold: 0.8, Sources: 9, Initial dictionary: τυχαία παραγόμενος πίνακας με τιμές που ακολουθούν την τυποποιημένη κανονική κατανομή.

Πίνακας 3: Σύγκριση εκτελέσεων k-SVD μεταξύ CPU και GPU

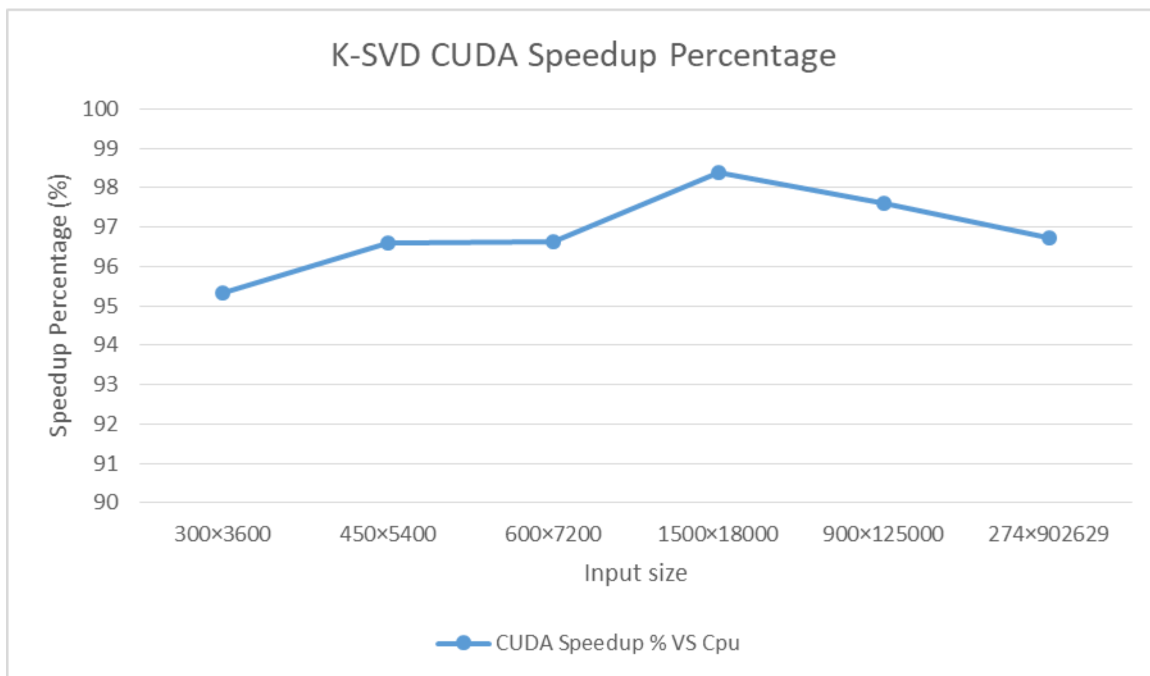
ΜΕΓΕΘΟΣ	MATLAB CPU (sec)	CUDA (sec)	ΕΠΙΤΑΧΥΝΣΗ	ΠΟΣΟΣΤΟ ΒΕΛΤΙΩΣΗΣ
300×3600	2,785	0,125	21,43	95,33%
450×5400	5,891	0,199	29,46	96,61%
600×7200	10,05	0,339	29,59	96,62 %
1500×18000	59,66	0,969	61,51	98,37%
900×125000	259,35	6,21	41,70	97,60%
274×902629	747,94	24,55	30,47	96,72%



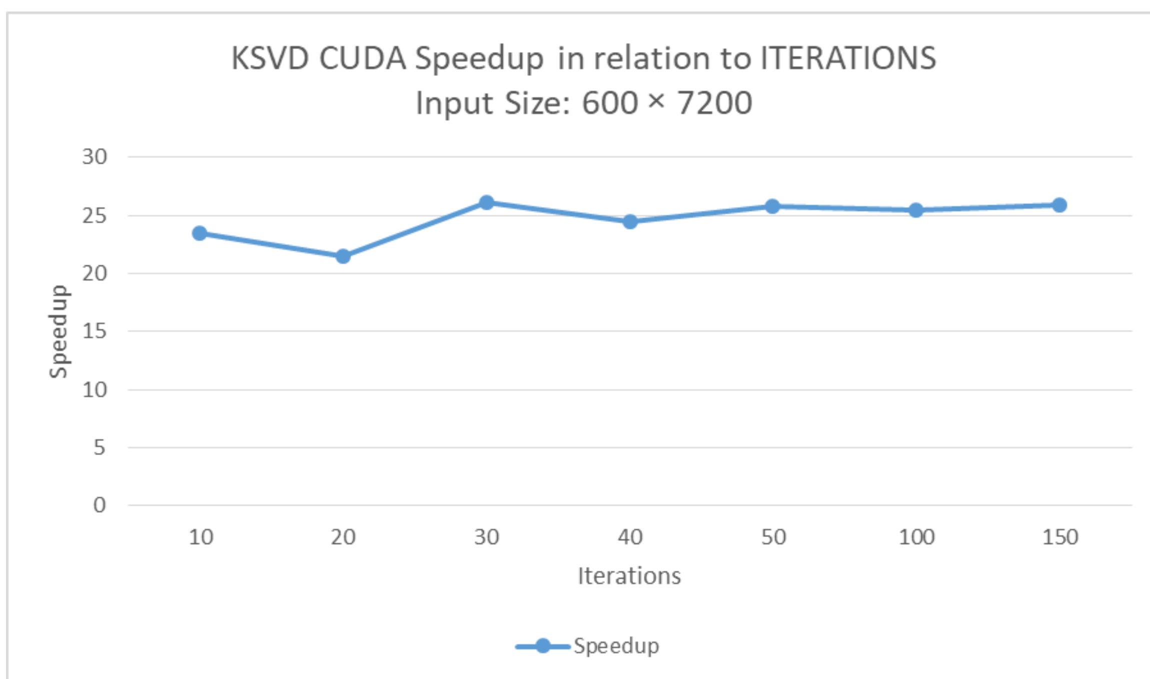
Εικόνα 27: Χρόνος Εκτέλεσης του αλγορίθμου k-SVD



Εικόνα 28: Επιτάχυνση της υλοποίησης σε CUDA του k-SVD



Εικόνα 29: Ποσοστό Επιτάχυνσης της υλοποίησης σε CUDA του k-SVD



Εικόνα 30: Επιτάχυνση της υλοποίησης σε CUDA του k-SVD σε συνάρτηση με τον αριθμό επαναλήψεων

Μπορούμε να παρατηρήσουμε την εκπληκτική επιτάχυνση του αλγορίθμου όσο μεγαλώνει η διάσταση των δεδομένων. Παρατηρούμε ακόμα πως το ποσοστό επιτάχυνσης είναι μεγάλο ακόμα και στα δεδομένα μικρότερης διάστασης. Το γεγονός αυτό δηλώνει την τεράστια παραλληλία που εκμεταλλεύεται η υλοποίηση μας. Ταυτόχρονα, όμως, διαφαίνεται και ο αντίκτυπος αυτής της τεράστιας ανάγκης για πόρους στα μεγαλύτερης διάστασης δεδομένα. Η επιτάχυνση (όπως και το αντίστοιχο ποσοστό) υποχωρούν και δεν κλιμακώνουν όπως θα θέλαμε. Αυτό είναι συνέπεια της πεπερασμένης υπολογιστικής ικανότητας της συγκεκριμένης κάρτας γραφικών που αδυνατεί να ανταπεξέλθει στις απαιτήσεις μιας τόσο μεγάλης παράλληλης εκτέλεσης.

5. ΑΛΓΟΡΙΘΜΟΣ PARAFAC2

5.1 Εισαγωγή σε PARAFAC και PARAFAC2

Ο τανυστής είναι ένας πολυδιάστατος πίνακας που αντιπροσωπεύει ένα σύνολο δεδομένων διατηρώντας την πολυτροπική του δομή. Η έννοια της παραγοντοποίησης τανυστών αναφέρεται στις μεθόδους που μπορούν να αναλύσουν έναν δεδομένο τανυστή ως άθροισμα πολυγραμμικών όρων με τρόπο ανάλογο με την ανάλυση διγραμμικών πινάκων. Μια από αυτές τις μεθόδους είναι η PARAFAC που χρησιμοποιείται για ανάλυση τριγραμμικών δεδομένων. Το μοντέλο PARAFAC για έναν τριών διαστάσεων πίνακα δεδομένων διατυπώνεται ως εξής: δεδομένου ενός τανυστή δεδομένων $\underline{X} \in \mathbb{R}^{I\times J\times K}$ και ενός θετικού ακεραίου L που δηλώνει το πλήθος των συνιστωσών, χρειάζεται να βρεθούν τρεις συντελεστές $\mathbf{A} \in \mathbb{R}^{I\times L}$, $\mathbf{F} \in \mathbb{R}^{J\times L}$ και $\mathbf{C} \in \mathbb{R}^{Q\times L}$, οι οποίοι προκύπτουν από την ακόλουθη προσεγγιστική ανάλυση:

$$\underline{X} = \sum_{l=1}^{L} \mathbf{a}_l \circ \mathbf{f}_l \circ \mathbf{c}_l + \underline{\mathrm{E}} = [\mathbf{A}, \mathbf{F}, \mathbf{C}] + \underline{\mathrm{E}},$$

όπου $\hat{\underline{X}} = [\mathbf{A}, \mathbf{F}, \mathbf{C}]$ είναι ο συμβολισμός της συντομογραφίας της ανάλυσης PARAFAC, $\mathbf{a}_l = [\mathrm{a}_{il}] \in \mathbb{R}^I$, $\mathbf{f}_l = [\mathrm{f}_{jl}] \in \mathbb{R}^J$, $\mathbf{c}_l = [\mathrm{c}_{ql}] \in \mathbb{R}^Q$ τα διανύσματα συνιστωσών των αντίστοιχων πινάκων-συντελεστών $\mathbf{A}$, $\mathbf{F}$, $\mathbf{C}$ αντίστοιχα, $\underline{\mathrm{E}} \in \mathbb{R}^{I\times J\times K}$ είναι το σφάλμα της ανάλυσης και $\circ$ είναι ο τελεστής εξωτερικού γινομένου. Η παραπάνω ισότητα μπορεί να γραφτεί σε μορφή στοιχείων ως ακολούθως:

$$x_{ijq} = \sum_{l=1}^{L} a_{il} f_{jl} c_{ql} + e_{ijq}\ .$$

Το βασικό μοντέλο της ανάλυσης PARAFAC μπορεί να γραφτεί σε μορφή πινάκων με ξεδίπλωμα του πίνακα στοιχείων $\underline{X}$ ως εξής:

$$\begin{cases} \mathbf{X}_{(1)} = \mathbf{A}(\mathbf{C} \odot \mathbf{F})^T \\ \mathbf{X}_{(2)} = \mathbf{F}(\mathbf{C} \odot \mathbf{A})^T \\ \mathbf{X}_{(3)} = \mathbf{C}(\mathbf{F} \odot \mathbf{A})^T \end{cases}$$

όπου $\odot$ είναι ο τελεστής του γινομένου Khatri-Rao. Οι συντελεστές $\mathbf{A}$, $\mathbf{F}$ και $\mathbf{C}$ μπορούν να υπολογιστούν ελαχιστοποιώντας τη συνάρτηση κόστους $J_1(\mathbf{A}, \mathbf{F}, \mathbf{C}) = \left\| \underline{X} - \hat{\underline{X}} \right\|_F^2$.

Στον PARAFAC2, τα κομμάτια μπορούν να παραγοντοποιηθούν ανεξάρτητα, αλλά έχουν κοινή μια συνιστώσα (εδώ την $\mathbf{A}$). Με άλλα λόγια, ο τανυστής δεδομένων μπορεί να διαφέρει σε μια διάσταση, για παράδειγμα, όταν οι πίνακες δεδομένων σε κάθε κομμάτι του τανυστή έχουν τον ίδιο αριθμό στηλών, αλλά διαφορετικό αριθμό γραμμών. Σε αυτήν την περίπτωση, το J θα διαφέρει ανάμεσα στα διαφορετικά κομμάτια του τανυστή. Το μοντέλο του PARAFAC2 δίνεται από τον τύπο: $\mathbf{X}_q = \mathbf{F}_q \mathbf{D}_q \mathbf{A}^T + \mathbf{E}_q$, όπου $\mathbf{X}_q$ είναι το q-οστό μπροστινό κομμάτι του τανυστή για $q = 1, \ldots, \mathrm{Q}$. Ο πίνακας $\mathbf{A}$ είναι η συνιστώσα της πρώτης διάστασης, ο πίνακας $\mathbf{F}_q$ είναι η συνιστώσα της δεύτερης διάστασης που σχετίζεται με το q-οστό μποστινό κομμάτι του $\underline{X}$, ο πίνακας $\mathbf{D}_q$ είναι διαγώνιος που περιέχει στην κύρια διαγώνιό του περιέχει τη γραμμή q του πίνακα C, που είναι η συνιστώσα της τρίτης διάστασης και το $\mathbf{E}_q$ αναπαριστά το σφάλμα που σχετίζεται με το $\mathbf{X}_q$.

Όπως μπορεί εύκολα να επαληθευτεί, το μοντέλο που παρουσιάζεται στον PARAFAC2 δεν παρέχει την ιδιότητα της μοναδικότητας του μοντέλου PARAFAC. Ο κύριος λόγος της μη ύπαρξης μοναδικής λύσης στο PARAFAC2 είναι ότι έχει διαφορετικές συνιστώσες $\mathbf{F}_q$ για διαφορετικούς πίνακες δεδομένων $\mathbf{X}_q$. Για αυτό προτάθηκε ένας συγκεκριμένος περιορισμός για την εξασφάλιση μοναδικών αποτελεσμάτων. Συγκεκριμένα, προτάθηκε η εκμετάλλευση του γεγονότος ότι το γινόμενο $\mathbf{F}_q^{\mathrm{T}}\mathbf{F}_q$ πρέπει να είναι σταθερό πάνω στο q. Έτσι, παρουσιάστηκε μια μέθοδος που θα ταιριάζει στο προαναφερθέν μοντέλο ελαχιστοποιώντας μια νέα συνάρτηση κόστους πάνω σε όλα τα ορίσματατα: $J_2(\mathbf{F}_q, \mathbf{A}, \mathbf{D}_1, ..., \mathbf{D}_q) = \sum_{q=1}^{Q} \left\| \mathbf{X}_q - \mathbf{F}_q \mathbf{D}_q \mathbf{A}^T \right\|_F^2$ σε σχέση με τον περιορισμό $\mathbf{F}_q^{\mathrm{T}}\mathbf{F}_q = \mathbf{F}_p^{\mathrm{T}}\mathbf{F}_p$ για όλα τα ζευγάρια p, q.

Για να επιβληθεί αυτός ο περιορισμός, είναι απαραίτητο και επαρκές να ισχύει $\mathbf{F}_q = \mathbf{P}_q\mathbf{F}$ για έναν ορθογώνιο ως προς τις στήλες πίνακα $\mathbf{P}_q \in \mathbb{R}^{J_q \times L}$ και πίνακα $\mathbf{F} \in \mathbb{R}^{L \times L}$. Επομένως, η παραπάνω συνάρτηση κόστους J_2 διαμορφώνεται ως ακολούθως: $J_2(\mathbf{P}_1, ..., \mathbf{P}_q, \mathbf{F}, \mathbf{A}, \mathbf{D}_1, ..., \mathbf{D}_q) = \sum_{q=1}^{Q} \left\| \mathbf{X}_q - \mathbf{P}_q \mathbf{F} \mathbf{D}_q \mathbf{A}^T \right\|_F^2$ σε σχέση με τον περιορισμό $\mathbf{P}_q^{\mathrm{T}}\mathbf{P}_q = I_L$.

Στη μέθοδο που προτάθηκε στο [14], χρησιμοποιείται εναλλακτικά ένας αλγόριθμος ελαχίστων τετραγώνων για ελαχιστοποίηση της J_2 πάνω στο $\mathbf{P}_q$ για σταθερά $\mathbf{F}$, $\mathbf{D}_q$, $\mathbf{A}$ με $q = 1, ..., \mathrm{Q}$ και πάνω στα $\mathbf{F}$, $\mathbf{D}_1, ..., \mathbf{D}_q$, $\mathbf{A}$ για σταθερά $\mathbf{P}_1, ..., \mathbf{P}_q$.

Ελαχιστοποιώντας πάνω στο $\mathbf{P}_q$ τη J_2 σε σχέση με τον περιορισμό $\mathbf{P}_q^{\mathrm{T}}\mathbf{P}_q = I_L$ οδηγούμαστε σε μεγιστοποίηση της ακόλουθης συνάρτησης: $f(\mathbf{P}_q) = \mathrm{tr}(\mathbf{F}\mathbf{D}_q \mathbf{A}^T \mathbf{X}_q^{\ T} \mathbf{P}_q)$ με $q = 1, ..., \mathrm{Q}$, όπου $\mathrm{tr}(\cdot)$ δηλώνει το ίχνος του πίνακα. Αν $\mathbf{F}\mathbf{D}_q \mathbf{A}^T \mathbf{X}_q^{\ T} = \mathbf{U}_q \mathbf{\Sigma} \mathbf{V}_q^T$ η ανάλυση SVD, τότε η μέγιστη τιμή της $f(\mathbf{P}_q)$ πάνω στο $\mathbf{P}_q$ λαμβάνεται από $\mathbf{P}_q = \mathbf{V}_q \mathbf{U}_q^T$ με $q = 1, ..., \mathrm{Q}$.

Μετά τον υπολογισμό του $\mathbf{P}_q$, το πρόβλημα της ελαχιστοποίησης της J_2 πάνω στα $\mathbf{F}$, $\mathbf{D}_1, ..., \mathbf{D}_q$, $\mathbf{A}$ περιορίζεται στην ελαχιστοποίηση:

$$J_2(\mathbf{F}, \mathbf{A}, \mathbf{D}_1, ..., \mathbf{D}_q) = \sum_{q=1}^{Q} \left\| \mathbf{P}_q^{\mathrm{T}} \mathbf{X}_q - \mathbf{F} \mathbf{D}_q \mathbf{A}^T \right\|_F^2$$

Όπως είναι ξεκάθαρο, η τελευταία ελαχιστοποίηση είναι ισοδύναμη με το πρόβλημα του PARAFAC, όταν το $\mathbf{X}_q$ αντικατασταθεί με το $\mathbf{Y}$, για το οποίο ισχύει $\mathbf{Y} = \mathbf{P}_q^{\mathrm{T}}\mathbf{X}_q$.

Ο ψευδοκώδικας για τον PARAFAC2 είναι βασισμένος στο [5] χωρίς να λαμβάνονται υπόψη κάποιες παράμετροι για τα πλαίσια αυτής της εργασίας. Συγκεκριμένα, αγνοούνται οι M που περιέχει προηγούμενη πληροφορία για τα χρονικά χαρακτηριστικά του Y, R που είναι πίνακας μετάθεσης (permutation matrix) και λ που είναι παράμετρος κανονικοποίησης που μειώνεται με τις επαναλήψεις.

Input: $\underline{\mathbf{X}}$: three-way data tensor
Output: $\mathbf{A}$, $\mathbf{F}$, $\mathbf{C}$: seperated factors for temporal, spatial and slice domain
begin
  Calculate the best initial values of $\mathbf{A}$, $\mathbf{F}$, $\mathbf{C}$ using multi initialization technique.
  repeat
  - Compute the SVD $\mathbf{F}\mathbf{D}_q\mathbf{A}^T\mathbf{X}_q^T$ and update $\mathbf{P}_q$ as $\mathbf{V}_q\mathbf{U}_q^T$, q = 1,...,Q
  - Compute $\mathbf{Y}_q = \mathbf{P}_q^T\mathbf{X}_q$, q = 1,...,Q

  repeat
    $\mathbf{A} \leftarrow \mathbf{Y}_{(1)}(\mathbf{C} \odot \mathbf{F}) + ((\mathbf{C}^T\mathbf{C}) \otimes (\mathbf{F}^T\mathbf{F}))^\dagger$
    $\mathbf{F} \leftarrow \mathbf{Y}_{(2)}(\mathbf{C} \odot \mathbf{A}) + ((\mathbf{C}^T\mathbf{C}) \otimes (\mathbf{A}^T\mathbf{A}))^\dagger$
    $\mathbf{C} \leftarrow \mathbf{Y}_{(3)}(\mathbf{F} \odot \mathbf{A}) + ((\mathbf{F}^T\mathbf{F}) \otimes (\mathbf{A}^T\mathbf{A}))^\dagger$
    $J_3 = ||\mathbf{Y} - \hat{\mathbf{Y}}||_F^2$
  until $(J_3^{old} - J_3^{new} > \epsilon J_3^{old})$
  $J_2 = \sum_{q=1}^{Q} ||\mathbf{X}_q - \mathbf{P}_q\mathbf{F}\mathbf{D}_q\mathbf{A}^T||$
  until $(J_2^{old} - J_2^{new} > \epsilon J_2^{old})$
end

Εικόνα 31: Ψευδοκώδικας αλγορίθμου PARAFAC2

5.2 Συσχέτιση PARAFAC2 με fMRI

Ας υποθέσουμε ότι ο τανυστής δεδομένων $\underline{X}$ περιέχει καταγεγραμμένες εικόνες fMRI. Κάθε ποσότητα fMRI δεδομένων που καταγράφεται σε κάθε μαγνητική τομογραφία αποτελείται από ένα πλήθος από κομμάτια (slices). Για να διατάξουμε τα fMRI δεδομένα σε έναν τανυστή πολλών διαστάσεων, πρώτα από όλα, τα κομμάτια μετατρέπονται σε διανύσματα και εισάγονται σαν γραμμές στον τανυστή $\underline{X}$. Συνεπώς, το $\underline{X}(\mathrm{i},:,:)$ κρατάει τα δεδομένα που καταγράφηκαν στην i-οστή τομογραφία, το $\underline{X}(:,:,\mathrm{q})$ κρατάει το q-οστό κομμάτι (βλ. Εικόνα 4) όλων των καταγεγραμμένων δεδομένων όλων των τομογραφιών και το $\underline{X}(:,\mathrm{j},:)$ κρατάει το voxel στη j-οστή χωρική τοποθεσία. Αν ο τανυστής $\underline{X}$ είναι φτιαγμένος όπως παραπάνω, οι πίνακες $\mathbf{A}$, $\mathbf{F}$, $\mathbf{C}$ δηλώνουν τους συντελεστές επιβάρυνσης στους τομείς του χρόνου, του χώρου και των κομματιών αντίστοιχα.

5.3 Υλοποίηση

Ο κώδικας για τη μέθοδο της ενότητας 5.1 βασίζεται σε κώδικα MATLAB που προήλθε από τα [5], [14] με μερικές απλοποιήσεις για τα πλαίσια της πτυχιακής μας εργασίας. Σε όλες τις εκτελέσεις, τόσο του αρχικού προγράμματος όσο και των δύο υλοποιήσεων σε MATLAB με πίνακες GPU και σε CUDA, χρησιμοποιούμε κοινές παραμέτρους, ώστε να μπορούμε να συγκρίνουμε τους χρόνους εκτέλεσης και το σφάλμα που προκύπτει. Έτσι, τα προγράμματα καλούν τον PARAFAC2 για 1000 επαναλήψεις, 8 πηγές, εκτυπώνουν έξοδο κάθε 100 επαναλήψεις και έχουν κριτήριο σύγκλισης 10^{-7}.

5.3.1 Υλοποίηση σε MATLAB με πίνακες GPU

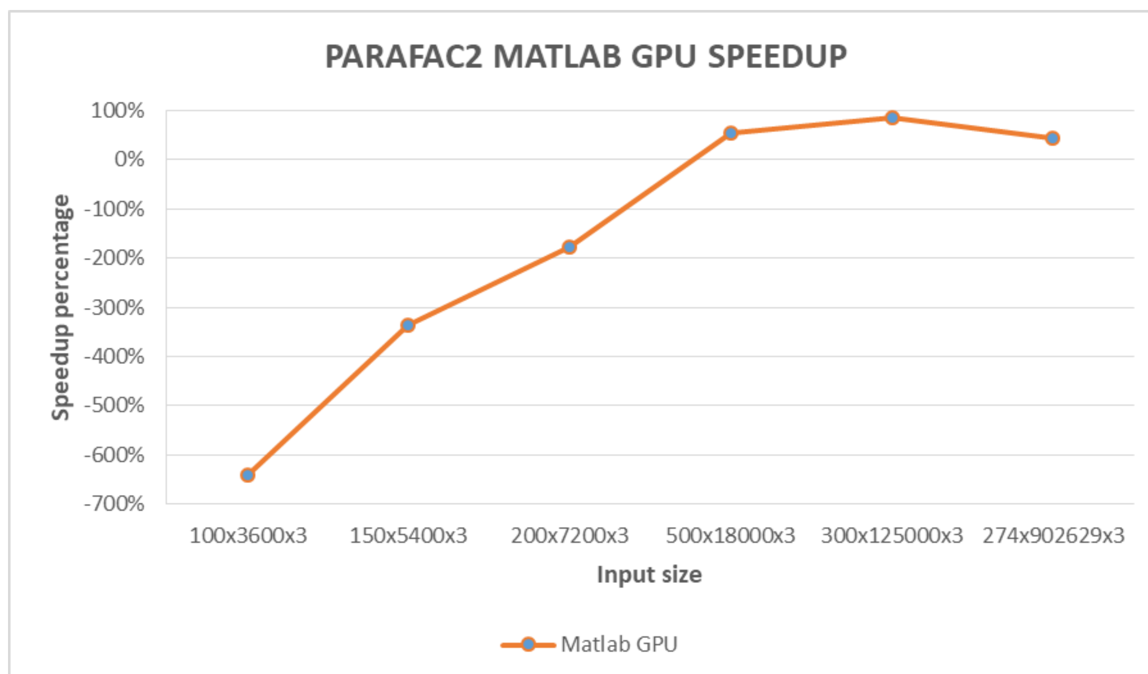
Ο κώδικας MATLAB βρίσκεται στο Παράρτημα I στην ενότητα με τίτλο «Αλγόριθμος PARAFAC2». Όλες οι εσωτερικές πράξεις στον PARAFAC2 χρησιμοποιούν gpuArrays. Τα δεδομένα που δίνονται ως είσοδος στον PARAFAC2 είναι σε μορφή gpuArray, εκτός αν η μνήμη της κάρτας γραφικών δεν επαρκεί, όπως στην περίπτωση των δεδομένων μεγέθους 274×902629×3. Στον Πίνακα 4 παρουσιάζονται οι χρόνοι εκτέλεσης σε

MATLAB με CPU και συγκρίνονται με τους χρόνους εκτέλεσης σε GPU. Πρόκειται για μέσους χρόνους δέκα εκτελέσεων.

Πίνακας 4: Σύγκριση χρόνων εκτέλεσης PARAFAC2 σε MATLAB με χρήση CPU και GPU

ΜΕΓΕΘΟΣ	MATLAB CPU (sec)	MATLAB GPU (sec)	ΠΟΣΟΣΤΟ ΒΕΛΤΙΩΣΗΣ
100×3600×3	15,5883	115,642	-641.85%
150×5400×3	27,766	121,4709	-337.48%
200×7200×3	45,4389	125,6939	-176,62%
500×18000×3	323,5443	153,3647	52,60%
300×125000×3	1396,0452	194,74	86,05%
274×902629×3	12505,7265	7135,5781	42,94%

Παρατηρούμε ότι στις πρώτες τρεις περιπτώσεις έχουμε σημαντική επιβράδυνση, διότι το μέγεθος των δεδομένων δεν επαρκεί για να αναδείξει τις δυνατότητες της GPU. Στις τρεις επόμενες περιπτώσεις, τα δεδομένα είναι επαρκώς μεγάλα ώστε να έχουμε φανερή επιτάχυνση. Συνεπώς, διαμορφώνεται το διάγραμμα της Εικόνας 32 που ξεκινά με αρνητικές τιμές επιτάχυνσης (επιβράδυνση) και καταλήγει σε θετικές. Αξιοσημείωτο είναι το γεγονός ότι στα μεγαλύτερα δεδομένα που δοκιμάσαμε, υπάρχει μεν επιτάχυνση, αλλά μικρότερη σε σχέση με τις δύο προηγούμενες. Αυτό συμβαίνει διότι, όπως προαναφέρθηκε, δε μπορούμε να δεσμεύσουμε όλους τους πίνακες που χρησιμοποιούνται στην εκτέλεση του αλγορίθμου και τα δεδομένα αυτού του μεγέθους στη GPU ταυτόχρονα. Οπότε, εκτελούμε τη συνάρτηση με τους πίνακες GPU χωρίς να έχουμε τα δεδομένα εισόδου στη GPU. Είναι φανερό ότι δε μπορούμε να προσεγγίσουμε τα ίδια ποσοστά επιτάχυνσης σε σχέση με τις προηγούμενες εκτελέσεις που είχαμε και τα δεδομένα εισόδου στη GPU, καθώς είναι αδύνατο να γίνουν πράξεις μεταξύ δεδομένων που βρίσκονται στη CPU και δεδομένων που βρίσκονται στη GPU, χωρίς να γίνει κάποιου είδους μεταφορά που επιβαρύνει το πρόγραμμα.



Εικόνα 32: Διαμόρφωση επιτάχυνση αλγορίθμου PARAFAC2 σε MATLAB με GPU

5.3.2 Υλοποίηση σε CUDA

Ο κώδικας CUDA βρίσκεται στο Παράρτημα II στην ενότητα με τίτλο «Αλγόριθμος PARAFAC2». Οι παράμετροι που χρησιμοποιήθηκαν είναι οι show_fit, conv_crit, max_iter, sources, MODE, sizeX, sizeZ. Η παράμετρος show_fit χρησιμοποιείται για να δείχνει κάθε πόσες επαναλήψεις θα εμφανίζονται στοιχεία εκτέλεσης του προγράμματος και τίθεται ίση με 100. Η παράμετρος conv_crit είναι το κριτήριο σύγκλισης του αλγορίθμου και τίθεται ίση με 10^{-7}. Η παράμετρος sources είναι το πλήθος των πηγών και ισούται με 8. Αυτές είναι οι κοινές παράμετροι και των τριών προγραμμάτων που καθορίζουν τη ροή εκτέλεσης τους. Επιπροσθέτως, η παράμετρος MODE λαμβάνει την τιμή 0 ή 1 ανάλογα με το αν θέλουμε να εκτελέσουμε το πρόγραμμα για μονή ή διπλή ακρίβεια αντίστοιχα. Τέλος, οι παράμετροι sizeX και sizeZ είναι οι δύο σταθερές διαστάσεις του κάθε κομματιού του τανυστή. Η τρίτη διάσταση ενδέχεται να μεταβάλλεται μεταξύ των διαφορετικών τμημάτων του τανυστή.

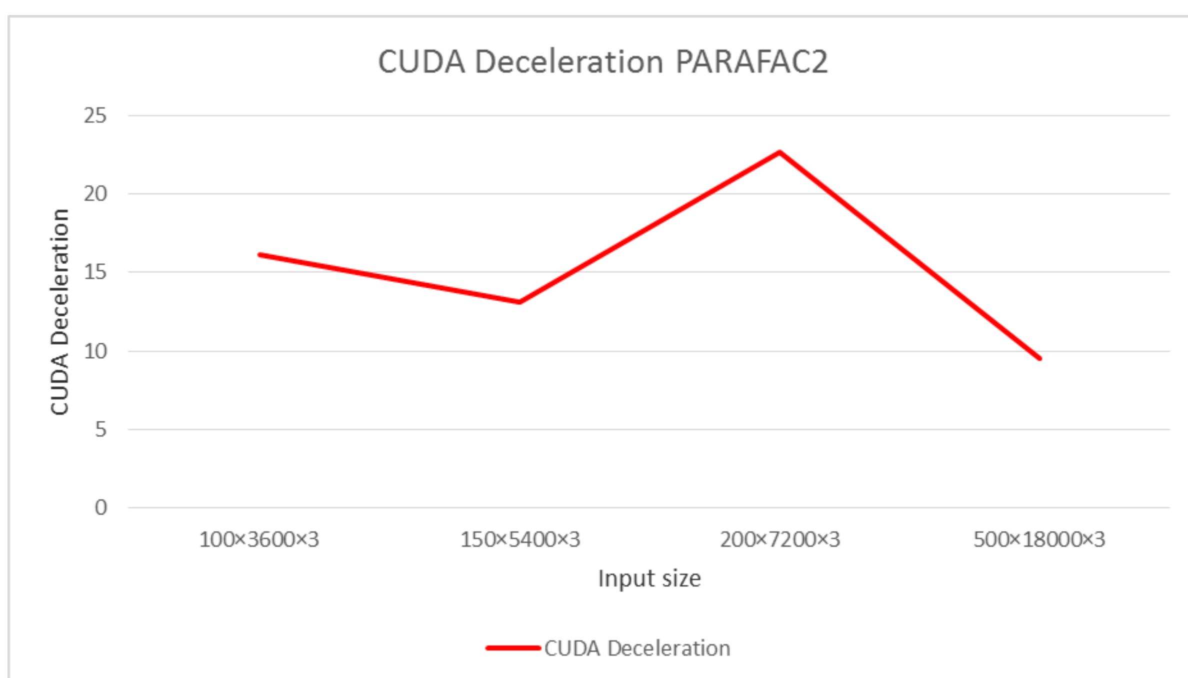
Όσον αφορά την είσοδο του προγράμματος, ο τανυστής μπορεί να έχει πολλά τμήματα, καθένα εκ των οποίων θα διαφέρει σε μια διάσταση. Παίρνουμε κάθε τμήμα του τανυστή από ένα αρχείο με όνομα input1.txt, input2.txt κ.ο.κ., στην πρώτη γραμή του οποίου αναγράφουμε το μεταβλητό μέγεθος μεταξύ των τμημάτων των τανυστών.

Από άποψη επιδόσεων, η υλοποίηση σε CUDA είναι σημαντικά πιο αργή σε σχέση τόσο με την αρχική όσο και με την υλοποίηση σε MATLAB με χρήση GPU. Στον Πίνακα 5 παρατίθενται οι χρόνοι εκτέλεσης της υλοποίησης σε CUDA και συγκρίνονται με αυτούς της αρχικής υλοποίησης, υπολογίζοντας και την επιβράδυνση των χρόνων εκτέλεσης της υλοποίησης σε CUDA.

Πίνακας 5: Σύγκριση χρόνων εκτέλεσης PARAFAC2 σε MATLAB με χρήση CPU και CUDA

ΜΕΓΕΘΟΣ	MATLAB CPU (sec)	CUDA(sec)	ΕΠΙΒΡΑΔΥΝΣΗ
100×3600×3	15,59	251,65	16,14
150×5400×3	27,77	363,40	13,09
200×7200×3	45,44	1029,90	22,67
500×18000×3	323,54	3072,92	9,50

Στην Εικόνα 33 βλέπουμε την επιβράδυνση που είναι σταθερά (πολύ) μεγάλη για όλα τα μέγεθη δεδομένων εισόδου, γεγονός που μας οδηγεί στο συμπέρασμα ότι δεν υπάρχουν μεγάλες δυνατότητες βελτίωσης του.



Εικόνα 33: Επιβράδυνση υλοποίησης PARAFAC2 σε CUDA για διαφορετικά δεδομένα εισόδου

Οι βασικοί λόγοι που προκαλούν αυτά τα μη-ικανοποιητικά αποτελέσματα της υλοποίησης σε CUDA είναι η μικρή δυνατότητα παραλληλοποίησης πράξεων και η χρήση μεθόδων παραγοντοποίησης πινάκων με πολύ μικρά δεδομένα. Όσον αφορά τον πρώτο λόγο, παρατηρούμε ότι υπάρχει εξάρτηση των υπολογισμών στα περισσότερα σημεία του αλγορίθμου από πράξεις που βρίσκονται αμέσως πριν, γεγονός που δε μας επιτρέπει να εκτελέσουμε παράλληλα πολλές πράξεις. Τέλος, όσον αφορά την παραγοντοποίηση πινάκων, χρησιμοποιούνται σε πολλά σημεία του προγράμματος οι SVD και QR με λίγα δεδομένα. Οι συναρτήσεις της βιβλιοθήκης cuSOLVER δεν ήταν καθόλου αποδοτικές για την εκτέλεση αυτών των παραγοντοποιήσεων. Συγκεκριμένα, η υλοποίηση της cuSOLVER για την QR όχι μόνο δεν ήταν αποδοτική, αλλά μας προσέθεσε επιπλέον πράξεις για να ξεχωρίσουμε τον πίνακα Q από τον πίνακα R, αφού επιστρέφει αποτέλεσμα σε έναν μόνον πίνακα. Όλα τα παραπάνω μας οδήγησαν στο συμπέρασμα ότι δε γίνεται να κατασκευάσουμε σε CUDA μια υλοποίηση ταχύτερη από το MATLAB για αυτόν τον αλγόριθμο.

6. ΣΥΜΠΕΡΑΣΜΑΤΑ

6.1 Συμπεράσματα αλγορίθμου MM

Ο αλγόριθμος MM είναι ένας αλγόριθμος με αρκετά απλές πράξεις που είναι εύκολο να επιταχυνθούν με χρήση GPU. Αν και υπάρχει μεγάλη εξάρτηση των δεδομένων και κάθε υπολογισμός είναι απαραίτητος στον ακριβώς επόμενο, οι πράξεις από μόνες τους κρύβουν εσωτερική παραλληλία, γεγονός που μας οδήγησε στο να επιτύχουμε βελτιωμένα αποτελέσματα σε σχέση με το αρχικό πρόγραμμα.

Οι δύο βασικές συναρτήσεις, η NewCoef και η NewDict, ενημερώνουν τον πίνακα συντελεστών και το λεξικό αντίστοιχα, παίρνοντας η καθεμία ως είσοδο την τιμή που παράγει η άλλη. Αποδεικνύεται ότι, ακόμα και αν οι δύο συναρτήσεις παίρνουν ως είσοδο τιμή προγενέστερη από την τελευταία ενημέρωση - όχι κατ' ανάγκη την αμέσως προηγούμενη τιμή - ο αλγόριθμος συγκλίνει. Συνεπώς, θα είχε ενδιαφέρον μελλοντικά η προσπάθεια εκπόνησης τέτοιας υλοποίησης με γνώμονα πάντα ότι αυτές οι δύο συναρτήσεις δεν έχουν τον ίδιο χρόνο εκτέλεσης και σε μια εκτέλεση της μίας μπορούν να χωρέσουν περισσότερες εκτελέσεις της άλλης.

6.2 Συμπεράσματα αλγορίθμου k-SVD

Ο αλγόριθμος k-SVD, όπως φάνηκε από τις μετρήσεις, αποτέλεσε πρόσφορο έδαφος για την πλήρη εκμετάλλευση της παραλληλίας που μας προσφέρει η CUDA. Οι μέθοδοι του OMP και του Batch-OMP αποτέλεσαν μια σχετικά εύκολη υλοποίηση και ταυτόχρονα εξαιρετικά αποδοτική. Αντίθετα, το στάδιο ενημέρωσης του λεξικού με την επανάληψη που δεν μπορεί να παραλληλοποιηθεί, αποτέλεσε το βασικό σημείο καθυστερήσεων και δεν μοιάζει να είναι πρόσφορο για περαιτέρω βελτίωση από πλευράς παραλληλίας.

Μελλοντική έρευνα στην επιτάχυνση του αλγορίθμου θα μπορούσε να εστιάσει στην βελτίωση των μεγάλων πολλαπλασιασμών πίνακα με πίνακα, ώστε να βελτιωθεί ο υπολογισμός του σφάλματος. Ακόμα και έτσι όμως πιστεύουμε ότι λόγω της υπάρχουσας παραλληλίας η πιθανότερη επίτευξη περαιτέρω επιτάχυνσης συνίσταται στην χρήση μιας ακόμη δυνατότερης κάρτας γραφικών που θα επιτρέψει ακόμα περισσότερους παράλληλους υπολογισμούς.

6.3 Συμπεράσματα αλγορίθμου PARAFAC2

Όπως και ο αλγόριθμος MM, έτσι και ο PARAFAC2 έχει το μειονέκτημα της μεγάλης εξάρτησης που παρουσιάζουν διαδοχικές πράξεις, δηλαδή κάθε πράξη χρειάζεται το αποτέλεσμα τής αμέσως προηγούμενης της. Σε αντίθεση, όμως, με τον MM έχει αρκετά πιο πολύπλοκες πράξεις, αρκετές εκ των οποίων επιδέχονται μηδαμινή ως ελάχιστη εσωτερική παραλληλία. Ο συνδυασμός της πολυπλοκότητας των πράξεων και της έλλειψης παραλληλίας, καθιστούν δύσκολο πρόβλημα την υλοποίηση μιας καλύτερης λύσης για τον PARAFAC2.

Παρόλο που η υλοποίηση με gpuArrays σε MATLAB είναι μια εν μέρει αποδοτική λύση, η αποτυχία υλοποίησης σε CUDA μιας συνολικά καλύτερης λύσης ανεξάρτητα από το μέγεθος των δεδομένων, αφήνει ως ανοιχτό ερευνητικό θέμα την προσπάθεια αυτή, παρά τις δυσκολίες που περιγράψαμε. Ένα σημείο στο οποίο θα έπρεπε να εστιάσει μια μελλοντική έρευνα θα ήταν το να βρει αποδοτικότερους τρόπους υλοποίησης μεθόδων παραγοντοποίησης πινάκων, όπως QR, SVD, κ.α., για μικρά δεδομένα. Ένα καλό

αποτέλεσμα σε αυτό το κομμάτι, θα ήταν μια πολλά υποσχόμενη αρχή για μια ταχύτερη επίλυση του PARAFAC2.

ΠΙΝΑΚΑΣ ΟΡΟΛΟΓΙΑΣ

Ξενόγλωσσος όρος	Ελληνικός Όρος
2D display accelerators	Επιταχυντές δισδιάστατων εικόνων
Atom-Assisted Dictionary Learning	Υποβοηθούμενη από άτομα μάθηση με λεξικό
Atoms	Άτομα
Blind Source Separation	Τυφλός Διαχωρισμός Πηγής
Blood-Oxygen-Level Dependent	Εξαρτώμενη από το επιπέδο οξυγόνωσης του αίματος
Cache memory	Κρυφή μνήμη
Canonical Decomposition	Κανονική Ανάλυση
Canonical Dictionary	Κανονικό Λεξικό
Clustering	Ομαδοποίηση
Coefficient update	Ενημέρωση συντελεστών
Compiler	Μεταγλωττιστής
Compressive Sensing	Συμπιεστική δειγματοληψία
Compute capability	Υπολογιστική δυνατότητα
Core architecture	Αρχιτεκτονική πυρήνα
CUDA kernel	Συνάρηση πυρήνα CUDA
Debugging	Αποσφαλμάτωση
Dense	Πυκνός
Dictionary Learning	Μάθηση με λεξικό
Dictionary Update	Ενημέρωση λεξικού
Dimensionality reduction	Μείωση διαστάσεων
Effective dimension	Ισχύουσα διάσταση
Factorization	Παραγοντοποίηση
Frontal slice	Εμπρόσθιο κομμάτι
Full rank	Μέγιστη τάξη
functional Magnetic Resonance Imaging	Λειτουργική Απεικόνιση Μαγνητικού Συντονισμού
General Linear Model	Γενικό Γραμμικό Μοντέλο
General-purpose computations	Υπολογισμοί γενικού σκοπού
Global memory	Καθολική μνήμη
Graphics pipeline	Σωλήνωση γραφικών
Grid	Πλέγμα
Haemodynamic response function	Συνάρτηση αιμοδυναμικής απόκρισης
Hardware driver	Οδηγός υλικού
Independent Component Analysis	Ανάλυση Ανεξάρτητων Συνιστωσών
Intrinsic Dimensionality	Εγγενής Διάσταση
Major revision number	Μείζων αριθμός αναθεώρησης
Majorization method	Μέθοδος μεγιστοποίησης
Minor revision number	Ελάσσων αριθμός αναθεώρησης
Multilinear algebra	Πλειογραμμική άλγεβρα
Multivariate bi-linear methods	Πολυμεταβλητές διγραμμικές μέθοδοι
Multiway component analysis	Ανάλυση πολυδιάστατων συνιστωσών
Mutual Information	Αμοιβαία Πληροφορία
Offset	Μετατόπιση
Overcomplete	Υπερπλήρης
Parallel computing	Παράλληλος υπολογισμός
Patch	Τμήμα

Pattern	Πρότυπο
Permutation matrix	Πίνακας μετάθεσης
Pixel	Εικονοστοιχείο
Pixel shaders	Σκιαστές εικονοστοιχείων
Principal Component Analysis	Ανάλυση Κύριων Συνιστωσών
Progress bar	Γραμμλη προόδου
Race condition	Συναγωνισμός δεδομένων
Rectangular diagonal	Ορθογώνια διαγώνιος
Redundant Dictionary	Πλεονάζον Λεξικό
Shading languages	Γλώσσες προγραμματισμού
Shared memory	Διαμοιραζόμενη μνήμη
Singular Value Decomposition	Παραγοντοποίηση Ιδιόμορφων Τιμών
Slice	Κομμάτι
Soft thresholding	Ήπια κατωφλίωση
Sparse	Αραιός
Sparse representation	Αραιή αναπαράσταση
Sparsity constraints	Περιορισμοί αραιότητας
Spatial map	Χωρικός χάρτης
Surrogate function	Αντιπροσωπευτική συνάρτηση
Task-related time courses	Σχετικά με την εργασία χρονοδιαγράμματα
Tensor	Τανυστής
Tensor Probabilistic Component Analysis	Τανυστική Πιθανολογική Ανάλυση Ανεξάρτητων Συνιστωσών
Thread	Νήμα
Time course	Χρονοδιάγραμμα
Unified shader pipeline	Ενιαία σωλήνωση σκίασης

ΣΥΝΤΜΗΣΕΙΣ – ΑΡΚΤΙΚΟΛΕΞΑ – ΑΚΡΩΝΥΜΙΑ

ALU	Arithmetic Logic Unit
API	Application Programmming Interface
BLAS	Basic Linear Algebra Subprograms
BOLD	Blood-Oxygen-Level Dependent
BSS	Blind Source Separation
CANDECOMP	Canonical Decomposition
CPU	Central Processing Unit
cuBLAS	cuda Basic Linear Algebra Subprograms
CUDA	Compute Unified Device Architecture
DFT	Discrete Fourier Transform
DWT	Discrete Wavelet Transform
FBN	Functional Brain Networks
FFTW	Fast Fourier Tranform in the West
fMRI	functional Magnetic Resonance Imaging
GB	Giga Byte
GFLOPS	Giga Floating-Point Operations per Second
GHz	Giga Hertz
GLM	General Linear Model
GPU	Graphics Processing Unit
HRF	Haemodynamic Response Function
IEEE	Institute of Electrical and Electronics Engineers
ICA	Independent Component Analysis
IPP	Integrated Performance Primitives
KKT	Karush-Kuhn-Tucker
LAPACK	Linear Algebra PACKage
LU	Lower Upper
Matlab	Matrix laboratory
MHz	Mega Hertz
MKL	Math Kernel Library
MM	Majorized Minimization
OMP	Orthogonal Matching Pursuit
OpenGL	Open Graphics Library
PARAFAC	Parallel Factorization
PCA	Principal Component Analysis
Pixel	Picture element
SDL	Supervised Dictionary Learning
SVD	Singular Value Decomposition
Voxel	Volume picture element

ΠΑΡΑΡΤΗΜΑ I

Υλοποίηση αλγορίθμων σε MATLAB με χρήση GPU

Στο παράρτημα Ι παραθέτουμε τις αντίστοιχες υλοποιήσεις σε MATLAB με χρήση GPU των αλγορίθμων που μας δόθηκαν και με τους οποίους ασχοληθήκαμε στα πλαίσια της παρούσας εργασίας. Για τον k-SVD δεν παρέχουμε τέτοια υλοποίηση, καθώς υπήρχαν κομμάτια που ήταν υλοποιημένα σε C και δεν ήταν συμβατά με τα gpuArrays του MATLAB.

1. Αλγόριθμος MM

```
%--------------------------------------------------------------------------
%                    Majorized Minimization (MoM)                          |
%-------------------------------------------------------------------------+
% Designed to solve the optimization task of the Assisted Dictionary       |
% Learning (ADL) via Majorization Method also known as the Majorized       |
% Minimization algorithm. Using GPU, based on :                            |
% https://github.com/MorCTI/Attom-Assisted-DL/blob/master/Algorithm/MoM.m |
%--------------------------------------------------------------------------
clear;
X = load('X.mat'); % load data from mat file
[a, b, c] = size(X);
X = reshape(X,a, b*c);
Y = gpuArray(X); % store data into gpu
clearvars -except Y;
K = 10; % Number of components
Tt = 200; % Total number of iterations
Ts = 20; % Number of iteration to compute the spatial maps
Es = 0;  % (optional parameter) not used in our implementation
Td = 20; % Number of iteration to compute the dictionary
Ed = 0; % (optional parameter) not used in our implementation
lambda =0.5; % The ë of the problem
ccl=1; % Value of the normalization of each atom
% Initialization
tic;
[T,N] = size(Y);
lambda = lambda*sqrt(norm(Y,'fro')/(T*N));%Normalization of the parameter ë
D = randn(T,K,'gpuArray');
S = randn(K,N,'gpuArray');
I = gpuArray(diag(ones(1,K)));
fprintf('Initial error : %.6f \n',sqrt(norm(Y-D*S,'fro')/(T*N)));
for i=1:Tt
    % Update Coefficients
    Dux = D'*D;
    cS  = real(max(sqrt(complex(eig(Dux.'*Dux)))));
    DY = D'*Y/cS;
    Aq = I-Dux/cS;
    Err = 1;
    t = 1;
    bound = 0.5*lambda/cS;
    while (t<=Ts && Err>Es)
        A = DY+Aq*S;
        A = wthresh(A,'s', bound);
        S = A;
        t = t+1;
    end
    % Update Dictionary
    Sux = S*S';
    cD  = real(max(sqrt(complex(eig(Sux.'*Sux)))));
    YS = Y*S'/cD;
    Bq = I-Sux/cD;
    Err = 1;
```

```
    t = 1;
    while (t<=Td && Err > Ed)
        B = YS + D*Bq;
        Kv = sqrt(sum(B.^2));
        Kv(Kv < ccl) = 1;
        B = bsxfun(@rdivide,B,Kv);
        D = B;
        t = t+1;
    end
    % Error Computation
    fprintf('Iteration %d : %.6f \n',i,sqrt(norm(Y-D*S,'fro')/(T*N)));

end
disp(['mom cpu time:   ' num2str(toc)]);
```

2. Αλγόριθμος PARAFAC2

```
function [A,H,C,P,fit] = parafac2(X,F)
    %     ___________________________________________________
    %
    %                   THE PARAFAC2 MODEL
    %
    %     ___________________________________________________
    %
    % Algorithm to fit the PARAFAC2 model which is an advanced variant of the
    % normal PARAFAC1 model. It handles slab-wise deviations between
components
    % in one mode as long as the cross-product of the components stays
    % reasonably fixed. This can be utilized for modeling chromatographic
    % data with retention time shifts, modeling certain batch data of
    % varying length etc. See Bro, Kiers & Andersson, Journal of
Chemometrics,
    % 1999, 13, 295-309 for details on application and Kiers, ten Berge &
    % Bro, Journal of Chemometrics, 1999, 13, 275-294, for details on the
algorithm
    %
    %
    % The PARAFAC2 model is given
    %
    % Xk = A*Dk*(Pk*H)' + Ek, k = 1, .., K
    %
    % Xk is a slab of data (I x J) in which J may actually vary with K. K
    % is the number of slabs. A (I x F) are the scores or first-mode
loadings. Dk
    % is a diagonal matrix that holds the k'th row of C in its diagonal. C
    % (K x F) is the third mode loadings, H is an F x F matrix, and Pk is a
    % J x F orthogonal matrix (J may actually vary from k to k. The output
here
    % is given as a cell array of size J x F x K. Thus, to get e.g. the
second P
    % write P(:,:,2), and to get the estimate of the second mode loadings at
this
    % second frontal slab (k = 2), write P(:,:,2)*H. The matrix Ek holds the
residuals.
    %
    % INPUT
    %
    % X
    %   Holds the data.
    %   If all slabs have similar size, X is an array:
    %      X(:,:,1) = X1; X(:,:,2) = X2; etc.
```

```
    %   If the slabs have different size X is a cell array (type <<help
cell>>)
    %      X{1} = X1; X{2} = X2; etc.
    %   If you have your data in an 'unfolded' two-way array of size
    %   I x JK (the three-way array is I x J x K), then simply type
    %   X = reshape(X,[I J K]); to convert it to an array.
    %
    % F
    %   The number of components (sources) to extract
    %
    %
    % OUTPUT
    % See right above INPUT

    ShowFit  = 100; % Show fit every 'ShowFit' iteration
    ConvCrit = 1e-7; % Convergence criterion
    MaxIt = 1000; % Maximal number of iterations
    fprintf('\n\nConvergence criterion        : %s\n',num2str(ConvCrit));
    fprintf('Maximal number of iterations : %s\n',num2str(MaxIt));
    fprintf('Number of factors            : %s\n\n\n',num2str(F));
    % Make X a cell array if it isn't
    if (~iscell(X))
      fprintf('Converting input into a cell array\n\n');
      i = size(X,3);
      x{i} = ones(i);
      for k = 1:size(X,3)
        x{k} = X(:,:,k);
      end
      X = x;
      clear x
    end
    I = size(X{1},1);
    K = max(size(X));
    % Initialize randomly
    disp(' Random initialization of A, C, H');
    A = rand(I,F,'gpuArray');
    C = rand(K,F,'gpuArray');
    H = eye(F,'gpuArray'); % returns an F-by-F identity matrix with ones on
the main diagonal and zeros elsewhere.
    XtX=X{1}*X{1}'; % Calculate for evaluating fit (but if initi = 1 it has
been calculated)
    for k = 2:K
        XtX = XtX + X{k}*X{k}';
    end
    fit    = sum(diag(XtX));% athroisma stoixeiwn diagwniou tetrsg
    oldfit = fit*2;
    fit0   = fit;
    it     = 0;
    disp(' ')
    disp(' Fitting model ...')
    disp(' Loss-value       Iteration      VariationExpl')
    P = cell(1);
    Y = gpuArray(zeros(I,F,K));
    % Iterative part
    while (abs(fit-oldfit)>oldfit*ConvCrit && it<MaxIt && fit>1000*eps)
        oldfit = fit;
        it   = it + 1;
        % Update P
        for k = 1:K
            Qk = X{k}'*(A*diag(C(k,:))*H');
            P{k}     = Qk*psqrt(Qk'*Qk);
            Y(:,:,k)  = X{k}*P{k};
        end
        %%
```

```
        % Update A,H,C using PARAFAC-ALS
        [A,H,C]=parafac(reshape(Y,I,F*K),[I F K],F,1e-4,A,H,C,5);
        [fit,X] = pf2fit(X,A,H,C,P,K);
        % Print interim result
        if (rem(it,ShowFit)==0 || it == 1)
            fprintf(' %12.10f       %g        %3.4f \n',fit,it,100*(1-
fit/fit0));
        end
    end
     if (rem(it,ShowFit)~=0) %Show final fit if not just shown
        fprintf(' %12.10f       %g        %3.4f \n',fit,it,100*(1-fit/fit0));
     end
end



function [A,B,C,fit,it] = parafac(X,DimX,Fac,crit,A,B,C,maxit)

    % Complex PARAFAC-ALS
    % Fits the PARAFAC model Xk = A*Dk*B.' + E
    % where Dk is a diagonal matrix holding the k'th
    % row of C.
    %
    % Uses on-the-fly projection-compression to speed up
    % the computations. This requires that the first mode
    % is the largest to be effective
    %
    % INPUT
    % X          : Data
    % DimX       : Dimension of X
    % Fac        : Number of factors
    % OPTIONAL INPUT
    % crit       : Convergence criterion (default 1e-6)
    % Constraints: [a b c], if e.g. a=0 => A unconstrained, a=1 => A
nonnegative
    % A,B,C      : Initial parameter values
    %
    % I/O
    % [A,B,C,fit,it]=parafac(X,DimX,Fac,crit,A,B,C);
    %
    % Copyright 1998
    % Rasmus Bro
    % KVL, Denmark, rb@kvl.dk

    % Initialization
    if nargin<8
      maxit    = 2500;       % Maximal number of iterations
    end
    showfit = pi;          % Show fit every 'showfit'th iteration (set to pi
to avoid)
    if nargin<4
      crit=1e-6;
    end
    if crit==0
      crit=1e-6;
    end

    J = DimX(2);
    K = DimX(3);

    SumSqX = sum(sum(abs(X).^2));
    fit    = SumSqX;
    fit0   = fit;
```

```
    fitold = 2*fit;
    it     = 0;

    while (abs((fit-fitold)/fitold)>crit && it<maxit && fit>10*eps)
       it=it+1;
       fitold=fit;

       % Update A
       Xbc=0;
       for k=1:K
         Xbc = Xbc + X(:,(k-1)*J+1:k*J)*conj(B*diag(C(k,:)));
       end
       A = Xbc*pinv((B'*B).*(C'*C)).';

       % Project X down on orth(A) - saves time if first mode is large
       [Qa,Ra]=qr(A,0);
       x=Qa'*X;

       % Update B
       Xac=0;
       for k=1:K
          Xac = Xac + x(:,(k-1)*J+1:k*J).'*conj(Ra*diag(C(k,:)));
       end
       B = Xac*pinv((Ra'*Ra).*(C'*C)).';

       % Update C
       ab=pinv((Ra'*Ra).*(B'*B));
       for k=1:K
          C(k,:) = (ab*diag(Ra'* x(:,(k-1)*J+1:k*J)*conj(B))).';
       end

        % Calculating fit. Using orthogonalization instead
       %fit=0;for k=1:K,residual=X(:,(k-1)*J+1:k*J)-
A*diag(C(k,:))*B.';fit=fit+sum(sum((abs(residual).^2)));end

       [~,Rb]=qr(B,0);
       [~,Rc]=qr(C,0);
       fit=SumSqX-sum(sum(abs(Ra*ppp(Rb,Rc).').^2));

       if rem(it,showfit)==0
          fprintf(' %12.10f       %g        %3.4f \n',fit,it,100*(1-
fit/fit0));
       end
    end

    % ORDER ACCORDING TO VARIANCE
    Tuck     = diag((A'*A).*(B'*B).*(C'*C));
    [~,ID] = sort(Tuck);
    A = A(:,ID);
    B = B(:,ID);
    C = C(:,ID);

    % NORMALIZE A AND C (variance in B)
    for f=1:Fac,normC(f) = gpuArray(norm(C(:,f)));end
    for f=1:Fac,normA(f) = gpuArray(norm(A(:,f)));end
    B = B*diag(normC)*diag(normA);
    A = A*diag(normA.^(-1));
    C = C*diag(normC.^(-1));

    %APPLY SIGN CONVENTION
    SignA = sign(sum(sign(A))+eps);
    SignC = sign(sum(sign(C))+eps);
```

```
        A = A*diag(SignA);
        C = C*diag(SignC);
        B = B*diag(SignA)*diag(SignC);
end



function [fit,X]=pf2fit(X,A,H,C,P,K)
    fit = 0;
    for k = 1:K
      M  = A*diag(C(k,:))*(P{k}*H)';
      fit = fit + sum(sum(abs (X{k} - M ).^2));
    end
end


function AB=ppp(A,B)
% The parallel proportional profiles product - triple-P product
% For two matrices with similar column dimension the triple-P product
% is ppp(A,B) = [kron(B(:,1),A(:,1) .... kron(B(:,F),A(:,F)]
%
% AB = ppp(A,B);
     [I,F]=size(A);
     [J,F1]=size(B);
     if F~=F1
        error(' Error in ppp.m - The matrices must have the same number of
columns')
     end
     AB=zeros(I*J,F, 'gpuArray');
     for f=1:F
        ab=A(:,f)*B(:,f).';
        AB(:,f)=ab(:);
     end
end


function X = psqrt(A,tol)
   % Produces A^(-.5) even if rank-problems
   [U,S,V] = svd(A,0);
   if (min(size(S)) == 1)
     S = S(1);
   else
     S = diag(S);
   end
   if (nargin == 1)
     tol = max(size(A)) * S(1) * eps;
   end
   r = sum(S > tol);
   if (r == 0)
     X = zeros(size(A'));
   else
     S = diag(ones(gather(r),1)./sqrt(S(1:r)));
     X = V(:,1:r)*S*U(:,1:r)';
   end
end
```

ΠΑΡΑΡΤΗΜΑ II

Υλοποίηση αλγορίθμων σε CUDA

Στο παράρτημα II παραθέτουμε τις αντίστοιχες υλοποιήσεις σε CUDA των τριών αλγορίθμων με τους οποίους ασχοληθήκαμε.

1. Αλγόριθμος MM

```
/*********************************************************************/
/*  This function is designed to solve the optimization task of the  */
/*  Assisted Dictionary Learning via Majorization Method also known  */
/*  as the Majorized Minimization algorithm                          */
/*  Based on https://github.com/MorCTI/Attom-Assisted-DL/find/master */
/*  Written in CUDA C using Microsoft Visual Studio 2015              */
/*  Implemented by C. Patsouras, D. Papageorgiou                      */
/*********************************************************************/


//////////////////////////////////////////////
//             File: main.cu               //
//////////////////////////////////////////////
#include <iostream>
#include <ctime>
#include <windows.h>
#include <curand.h>
#include <cusolverDn.h>
#include "cublas_v2.h"
#include "errorTypes.h"
#include "declarations.h"
#include "Globals.h"
#include "timer.h"
#include "mem.h"
#include INCLUDE_FILE(MATLAB_include,mat.h)
#include INCLUDE_FILE(MATLAB_include,matrix.h)
#include "matCode.h"
#include "primaryFunctions.h"

using namespace std;

//////////////////////////////////////////////////////////////////////////////
/* Global variables */
int *devInfo;
cublasHandle_t handle;
curandGenerator_t generator;
cusolverDnHandle_t cusolverH;
datatype *I, *KK, *KK2, *KVOXELS, *KVOXELS2, *TIMECOMPK, *TIMECOMPK2, *Kv,
*TIMECOMPVOXELS, *X, *D, *S, *W, *d_W;
//////////////////////////////////////////////////////////////////////////////


int main(void) {
      cudaError_t err;
      datatype *dataInput;
      mxArray *mx_ptr = NULL;

      printf("MoM algorithm\n");

      /* Reading input from file */
      printf("Reading input from file\nPlease wait ...\n");
      {
            Timer t;
            if (!t.isValid()) {
```

```
                printf("Timer initialization error! Aborting...\n");
                return -1;
            }
            t.tic();
            readingInput(&dataInput, &mx_ptr);
            printf("Reading Time: %f sec\n", t.toc());
            printf("Input size : %d x %d\n", Globals::rowsX, Globals::colsX);
        }
        /* Allocate all the necessary memory */
        {
            Timer t;
            if (!t.isValid()) {
                cerr << "Timer initialization error! Aborting..." << endl;
                return -1;
            }
            t.GPUtic();
            allocate();
            printf("Time to allocate all the necessary memory: %f sec\n", t.GPUtoc() /
1000.0);
        }
        if ((err = cudaMemcpy(X, dataInput, Globals::rowsX * Globals::colsX *
sizeof(datatype), cudaMemcpyHostToDevice)) != cudaSuccess) {
            printf("cudaMemcpy X failed: %s\n", cudaGetErrorString(err));
            cleanup();
            return -1;
        }
        /* Calling MoM */
        {
            Timer t;
            if (!t.isValid()) {
                cerr << "Timer initialization error! Aborting..." << endl;
                return -1;
            }
            t.GPUtic();
            mom();
            printf("MoM Time: %f sec\n", t.GPUtoc() / 1000.0);
        }
        /* Check for possible errors */
        while ((err = cudaGetLastError()) != cudaSuccess)
            printf("Error: An error occured during the execution of the program: %s\n",
cudaGetErrorString(err));
        /* Free allocated memory */
        if (mx_ptr) mxDestroyArray(mx_ptr);
        cleanup();
        return 0;
    }

///////////////////////////////////////////////
//            File: Timer.h                 //
///////////////////////////////////////////////
#ifndef _TIMER_
#define _TIMER_


// Functions used for calculating
// elapsed time on CPU or GPU
class Timer {

private:
        // Private Members
        cudaEvent_t GPUstart, GPUstop;
        LARGE_INTEGER CPUstart, CPUstop, frequency;

public:
        // Constructor
```

```
	Timer();
	~Timer();

	// Public Methods
	void tic();
	double	toc();
	double	tocmilli();
	void GPUtic();
	float GPUtoc();
	bool isValid();

};

#endif // !_TIMER_

////////////////////////////////////////////////
//            File: Timer.cu                  //
////////////////////////////////////////////////
#include <iostream>
#include <windows.h>
#include "declarations.h"
#include "timer.h"

using namespace std;


/*---------------*/
/* Constructor   */
/*---------------*/
Timer::Timer() {
	this->GPUstart = NULL;
	this->GPUstop = NULL;
	cudaError_t cuda_error;
	if ((cuda_error = cudaEventCreate(&this->GPUstart)) != cudaSuccess) {
		cerr << "Cuda event create for GPUstart failed: " <<
cudaGetErrorString(cuda_error) << endl;
	}
	if ((cuda_error = cudaEventCreate(&this->GPUstop)) != cudaSuccess) {
		cerr << "Cuda event create for GPUstop failed: " <<
cudaGetErrorString(cuda_error) << endl;
	}
}


/*--------------*/
/* Destructor   */
/*--------------*/
Timer::~Timer() {
	if (this->GPUstart) {
		cudaEventDestroy(this->GPUstart);
	}
	if (this->GPUstop) {
		cudaEventDestroy(this->GPUstop);
	}
}


/*--------------------------*/
/* Check if timer is valid  */
/*--------------------------*/
bool Timer::isValid() {
	return (this->GPUstart != NULL && this->GPUstop != NULL);
}
```

```
/*-------------------*/
/* Start CPU timer   */
/*-------------------*/
void Timer::tic() {
	QueryPerformanceFrequency(&this->frequency);
	QueryPerformanceCounter(&this->CPUstart);
}


/*--------------------*/
/* Stop CPU timer and */
/* return in sec!     */
/*--------------------*/
double Timer::toc() {
	QueryPerformanceCounter(&this->CPUstop);
	return ((double)(this->CPUstop.QuadPart - this->CPUstart.QuadPart) / this-
>frequency.QuadPart);
}


/*--------------------*/
/* Stop CPU timer and */
/* return in ms!      */
/*--------------------*/
double Timer::tocmilli() {
	QueryPerformanceCounter(&this->CPUstop);
	return ((double)(this->CPUstop.QuadPart - this->CPUstart.QuadPart) / this-
>frequency.QuadPart * 1000.0);
}


/*-------------------*/
/* Start GPU timer   */
/*-------------------*/
void Timer::GPUtic() {
	cudaEventRecord(this->GPUstart);
}


/*--------------------*/
/* Stop GPU timer and */
/* return in ms!      */
/*--------------------*/
float Timer::GPUtoc() {
	cudaEventRecord(this->GPUstop);
	cudaEventSynchronize(this->GPUstop);
	float milliseconds = 0;
	cudaEventElapsedTime(&milliseconds, this->GPUstart, this->GPUstop);
	return milliseconds;

}

///////////////////////////////////////////////
//           File: reduction.h               //
///////////////////////////////////////////////
#ifndef __reduction__
#define __reduction__

void normalizeReduction(datatype *, datatype*);


#endif // !__reduction__

///////////////////////////////////////////////
//           File: reduction.cu              //
```

```
////////////////////////////////////////////////
#include <math.h>
#include "declarations.h"
#include "reduction.h"
#include "Globals.h"


// Macro for finding the minimum of 2 numbers
#define MINIMUM(a,b) ( a < b ? a : b )

// Temporary reduction storage buffer
__device__ datatype experimental_array[K * 1024];


/* Action performed on data before reduction of sum of elements! 'SUM OF SQUARE VALUES'
*/
__inline__ __device__ datatype sqval(datatype val) {
	return val*val;
}


/* Reduction within a single warp */
__inline__ __device__ datatype warpReduceSum(datatype val) {
	for (int offset = warpSize / 2; offset > 0; offset /= 2)
		val += __shfl_down(val, offset);
	return val;
}


/* Reduction within a single block */
__inline__ __device__ datatype blockReduceSum(datatype val) {
	// Max. block size = 1024 (32 x 32) = 32 warps of 32 threads
	static __shared__ datatype shared[32];      // Shared mem for 32 partial sums
	int lane = threadIdx.x % warpSize;
	int wid = threadIdx.x / warpSize;
	val = warpReduceSum(val);  // Each warp performs partial reduction
	if (lane == 0) shared[wid] = val; // Write reduced value to shared memory
	__syncthreads(); // Wait for all partial reductions
	//read from shared memory only if that warp existed
	val = (threadIdx.x < blockDim.x / warpSize) ? shared[lane] : 0;

	if (wid == 0) val = warpReduceSum(val);     //Final reduce within first warp
	return val;
}


/* Reduction across multiple blocks -> STAGE 1: Blocks to buffer */
__global__ void deviceReduceKernel(datatype *in, int N, int cols) {
	datatype sum = 0;
	int i = blockIdx.x * blockDim.x + threadIdx.x;
	if ( i < N){
		// Retrieve value filtered through function
		sum = sqval(in[i*cols + blockIdx.y]);
	}
	sum = blockReduceSum(sum);
	if (threadIdx.x == 0) {
		// Write to first stage buffer
		experimental_array[blockIdx.y * 1024 + blockIdx.x] = sum;
	}
}


/* Reduction across multiple blocks -> STAGE 2: Partial sums */
__global__ void deviceReduceKernel2(datatype* out, int N) {
	datatype sum = 0;
```

```
	int i = blockIdx.x * blockDim.x + threadIdx.x;
	if (i < N){
		// Retrieve partial sum
		sum = *(experimental_array + blockIdx.y * 1024 + i);
	}
	sum = blockReduceSum(sum);
	if (threadIdx.x == 0) {
		// Write to output...
		datatype s = sqrt(sum);
		// ...after Soft Thresholding
		out[blockIdx.y] = (s < CCL ? 1.0 : s);
	}
}


/* Normalization of columns of A by 2 stage square sum reductions */
void normalizeReduction(datatype *A, datatype* B) {
	dim3 block(512);
	dim3 grid(MINIMUM((Globals::rowsX + block.x - 1) / block.x, 1024), K);
	// 1st stage
	deviceReduceKernel << <grid, block >> > (A, Globals::rowsX, K);
	dim3 grid2(1, K);
	// 2nd stage
	deviceReduceKernel2 << <grid2, 1024 >> > (B, grid.x);

}

///////////////////////////////////////////////
//       File: primaryFunctions.h            //
///////////////////////////////////////////////
#ifndef  __primaryFunctions__
#define __primaryFunctions__

void mom();


#endif

///////////////////////////////////////////////
//       File: primaryFunctions.cu           //
///////////////////////////////////////////////
#include <iostream>
#include <curand.h>
#include <cusolverDn.h>
#include "declarations.h"
#include "cublas_v2.h"
#include "kernel.h"
#include "cuLibs.h"
#include "reduction.h"
#include "errorTypes.h"
#include "Globals.h"
#include "mem.h"

using namespace std;

////////////////////////////////////////////////////////////////////////////////
/* Global variables */
extern int *devInfo;
extern cublasHandle_t handle;
extern curandGenerator_t generator;
extern cusolverDnHandle_t cusolverH;
extern datatype *I, *KK, *KK2, *KVOXELS, *KVOXELS2, *TIMECOMPK, *TIMECOMPK2, *Kv,
*TIMECOMPVOXELS, *X, *D, *S, *W, *d_W;
////////////////////////////////////////////////////////////////////////////////
```

```
/* Computes the error of the approximation */
datatype ErrorComp() {
	gpu_blas_mmul(handle, D, S, TIMECOMPVOXELS, 0, Globals::rowsX, K, K,
Globals::colsX); // D*S
	subtractMatrices << <(Globals::rowsX*Globals::colsX) / 1024 + 1, 1024 >> >
(TIMECOMPVOXELS, X, TIMECOMPVOXELS, Globals::rowsX*Globals::colsX); // Y - D*S

	return sqrt(frobeniusNorm(handle, TIMECOMPVOXELS, Globals::rowsX*Globals::colsX) /
(Globals::rowsX*Globals::colsX)); // sqrt(norm(Y - D*S, 'fro') / (TIMECOMP*VOXELS))
}


/* Calculates the new coefficients according to the MM algorithm */
void NewCoef(datatype lamb) {
	cudaError_t err;
	datatype cS, Err, bound;
	int t, Ts, Es;
	if (ES != -1) {
		Ts = 500;
		Es = ES;
	}
	else {
		Ts = TS;
		Es = 0;
	}
	gpu_blas_mmul(handle, D, D, KK, 1, Globals::rowsX, K, Globals::rowsX, K); // Dux =
D'*D;
	gpu_blas_mmul(handle, KK, KK, KK2, 1, K, K, K, K); // Dux.'*Dux
	cS = findMaxSqrtEigenvalue(cusolverH, KK2, W, d_W, devInfo, K); // cS  =
max(sqrt(eig(Dux.'*Dux))
	gpu_blas_mmul(handle, D, X, KVOXELS, 1, Globals::rowsX, K, Globals::rowsX,
Globals::colsX); // D'*Y
	matrixDivNum << <(K*Globals::colsX) / 1024 + 1, 1024 >> > (KVOXELS,
K*Globals::colsX, cS); // DY = D'*Y/cS
	matrixDivNum << < (K*K) / 1024 + 1, 1024 >> > (KK, K*K, cS); // Dux/cS
	subtractMatrices << <(K*K) / 1024 + 1, 1024 >> > (KK2, I, KK, K*K); //Aq = I - Dux /
cS;
	Err = 1.0;
	t = 1;
	bound = (datatype)0.5*lamb / cS;
	while (t <= Ts && Err>Es) {
		gpu_blas_mmul(handle, KK2, S, KVOXELS2, 0, K, K, K, Globals::colsX); // Aq*S
		addMatrices << <(K*Globals::colsX) / 1024 + 1, 1024 >> > (KVOXELS2, KVOXELS,
KVOXELS2, K*Globals::colsX); // A = DY+Aq*S
		wthresh << <(K*Globals::colsX) / 1024 + 1, 1024 >> > (KVOXELS2,
K*Globals::colsX, bound); // A = wthresh(A, 's', 0.5*lam / cS)
		if ((err = cudaMemcpy(S, KVOXELS2, K * Globals::colsX * sizeof(datatype),
cudaMemcpyDeviceToDevice)) != cudaSuccess) {
			printf("cudaMemcpy S failed: %s\n", cudaGetErrorString(err));
			cleanup();
			exit(-1);
		}	// S=A
		t = t + 1;
	}
}


/* Calculates the new Dictionary atoms according to the MM algorithm */
void NewDict() {
	int t, Ed, Td;
	cudaError_t err;
	datatype cD, Err;
	if (ED != -1) {
		Ed = ED;
		Td = 100;
```

```
	}
	else {
		Td = TD;
		Ed = 0;
	}
	gpu_blas_mmul(handle, S, S, KK, 2, K, Globals::colsX, K, Globals::colsX); // Sux = S*S'
	gpu_blas_mmul(handle, KK, KK, KK2, 1, K, K, K, K); // Sux.'*Sux
	cD = findMaxSqrtEigenvalue(cusolverH, KK2, W, d_W, devInfo, K); //  cD  = max(sqrt(eig(Sux.'*Sux)))
	gpu_blas_mmul(handle, X, S, TIMECOMPK, 2, Globals::rowsX, Globals::colsX, K, Globals::colsX); // Y*S'
	matrixDivNum << <(Globals::rowsX*K) / 1024 + 1, 1024 >> > (TIMECOMPK, Globals::rowsX*K, cD); // YS = Y*S'/cD
	matrixDivNum << <(K*K) / 1024 + 1, 1024 >> > (KK, K*K, cD); // Sux/cD
	subtractMatrices << <(K*K) / 1024 + 1, 1024 >> > (KK2, I, KK, K*K); // Bq = I-Sux/cD
	Err = 1.0;
	t = 1;
	while (t <= Td && Err > Ed) {
		gpu_blas_mmul(handle, D, KK2, TIMECOMPK2, 0, Globals::rowsX, K, K, K); // D*Bq
		addMatrices << <(Globals::rowsX*K) / 1024 + 1, 1024 >> > (TIMECOMPK2, TIMECOMPK, TIMECOMPK2, Globals::rowsX*K); // B = YS + D*Bq
		normalizeReduction(TIMECOMPK2, Kv); // Kv = sqrt(sum(B.^2)); Kv(Kv < ccl) = 1;
		rdivide << <(Globals::rowsX*K) / 1024 + 1, 1024 >> > (TIMECOMPK2, Globals::rowsX, K, Kv); //B = bsxfun(@rdivide,B,Kv)
		if ((err = cudaMemcpy(D, TIMECOMPK2, Globals::rowsX * K * sizeof(datatype), cudaMemcpyDeviceToDevice)) != cudaSuccess) {
			printf("cudaMemcpy failed D: %s\n", cudaGetErrorString(err));
			cleanup();
			exit(-1);
		}  //D = B
		t = t + 1;
	}
}


/* Majorized minimization */
void mom() {
	int i;
	datatype lamb;
	lamb = (datatype)LAMBDA*sqrt(frobeniusNorm(handle, X, Globals::rowsX*Globals::colsX) / (Globals::rowsX*Globals::colsX)); //lamb = lamb*sqrt(norm(Y,'fro')/(T*N));
	randn(generator, D, Globals::rowsX * K); // D = randn(T,K)
	randn(generator, S, K * Globals::colsX); // S = randn(K,N)
	dim3 grid(1, 1);
	dim3 threads(K, K);
	initIdentityGPU << <grid, threads >> > (I, K, K);
	printf("Initial error\t:  %f\n\n", ErrorComp());
	for (i = 0; i < ITER; i++) {
		NewCoef(lamb);
		NewDict();
		printf("Iteration %d\t:  %f \n", i + 1, ErrorComp());
	}
	printf("\nFinal error\t:  %f \n", ErrorComp());

}

////////////////////////////////////////////////
//                File: mem.h                 //
////////////////////////////////////////////////
#ifndef __mem__
#define __mem__

void allocate();
void cleanup();
```

```
#endif // !__clean__
//////////////////////////////////////////////
//              File: mem.cu                //
//////////////////////////////////////////////
#include <iostream>
#include <ctime>
#include <curand.h>
#include <cusolverDn.h>
#include "cublas_v2.h"
#include "declarations.h"
#include "errorTypes.h"
#include "Globals.h"

//////////////////////////////////////////////////////////////////////////////
/* Global variables */
extern int *devInfo;
extern cublasHandle_t handle;
extern curandGenerator_t generator;
extern cusolverDnHandle_t cusolverH;
extern datatype *I, *KK, *KK2, *KVOXELS, *KVOXELS2, *TIMECOMPK, *TIMECOMPK2, *Kv,
*TIMECOMPVOXELS, *X, *D, *S, *W, *d_W;
//////////////////////////////////////////////////////////////////////////////


/* Free all the previously allocated memory */
void cleanup() {
      cudaError_t err;
      curandStatus_t curstat;
      cublasStatus_t cubstat;
      cusolverStatus_t cusstat;
      if (W) free(W);
      if (generator && (curstat = curandDestroyGenerator(generator)) !=
CURAND_STATUS_SUCCESS)
            printf("curandDestroyGenerator failed: %s\n", curandGetErrorString(curstat));
      if (handle && (cubstat = cublasDestroy(handle)) != CUBLAS_STATUS_SUCCESS)
            printf("cublasDestroy failed: %s\n", cublasGetErrorString(cubstat));
      if (X && (err = cudaFree(X)) != cudaSuccess) printf("cudaFree X failed: %s\n",
cudaGetErrorString(err));
      if (D && (err = cudaFree(D)) != cudaSuccess) printf("cudaFree D failed: %s\n",
cudaGetErrorString(err));
      if (S && (err = cudaFree(S)) != cudaSuccess) printf("cudaFree S failed: %s\n",
cudaGetErrorString(err));
      if (I && (err = cudaFree(I)) != cudaSuccess) printf("cudaFree cuI failed: %s\n",
cudaGetErrorString(err));
      if (KK && (err = cudaFree(KK)) != cudaSuccess) printf("cudaFree KK failed: %s\n",
cudaGetErrorString(err));
      if (KK2 && (err = cudaFree(KK2)) != cudaSuccess) printf("cudaFree KK2 failed: %s\n",
cudaGetErrorString(err));
      if (KVOXELS && (err = cudaFree(KVOXELS)) != cudaSuccess) printf("cudaFree KVOXELS
failed: %s\n", cudaGetErrorString(err));
      if (KVOXELS2 && (err = cudaFree(KVOXELS2)) != cudaSuccess) printf("cudaFree KVOXELS2
failed: %s\n", cudaGetErrorString(err));
      if (TIMECOMPK && (err = cudaFree(TIMECOMPK)) != cudaSuccess) printf("cudaFree
TIMECOMPK failed: %s\n", cudaGetErrorString(err));
      if (TIMECOMPK2 && (err = cudaFree(TIMECOMPK2)) != cudaSuccess) printf("cudaFree
TIMECOMPK2 failed: %s\n", cudaGetErrorString(err));
      if (Kv && (err = cudaFree(Kv)) != cudaSuccess) printf("cudaFree Kv failed: %s\n",
cudaGetErrorString(err));
      if (TIMECOMPVOXELS && (err = cudaFree(TIMECOMPVOXELS)) != cudaSuccess)
printf("cudaFree TIMECOMPVOXELS failed: %s\n", cudaGetErrorString(err));
      if (devInfo && (err = cudaFree(devInfo)) != cudaSuccess) printf("cudaFree devInfo
failed: %s\n", cudaGetErrorString(err));
      if (cusolverH && (cusstat = cusolverDnDestroy(cusolverH)) !=
CUSOLVER_STATUS_SUCCESS)
```

```
		printf("cusolverDnDestroy failed: %s\n", cusolverGetErrorString(cusstat));
	if (d_W && (err = cudaFree(d_W)) != cudaSuccess) printf("cudaFree d_W failed: %s\n", 
cudaGetErrorString(err));
	if (cudaDeviceReset() != cudaSuccess) printf("cudaDeviceReset failed: %s\n", 
cudaGetErrorString(err));
}


/* Allocate all necessary memory for global variables */
void allocate() {
	cudaError_t err;
	cublasStatus_t cubStat;
	curandStatus_t curStat;
	cusolverStatus_t cusStat;
	if ((W = (datatype *)malloc(K * sizeof(datatype))) == NULL) {
		perror("malloc failed W");
		cleanup();
		exit(-1);
	}
	if ((cubStat = cublasCreate(&handle)) != CUBLAS_STATUS_SUCCESS) {
		printf("cublasCreate failed: %s\n", cublasGetErrorString(cubStat));
		cleanup();
		exit(-1);
	}
	if ((cusStat = cusolverDnCreate(&cusolverH)) != CUSOLVER_STATUS_SUCCESS) {
		printf("cusolverDnCreate failed: %s\n", cusolverGetErrorString(cusStat));
		cleanup();
		exit(-1);
	}
	if ((curStat = curandCreateGenerator(&generator, CURAND_RNG_PSEUDO_DEFAULT)) != 
CURAND_STATUS_SUCCESS) {
		printf("curandCreateGenerator failed: %s\n", curandGetErrorString(curStat));
		cleanup();
		exit(-1);
	}
	if ((curStat = curandSetPseudoRandomGeneratorSeed(generator, (unsigned long long) 
clock())) != CURAND_STATUS_SUCCESS) {
		printf("curandSetPseudoRandomGeneratorSeed failed: %s\n", 
curandGetErrorString(curStat));
		cleanup();
		exit(-1);
	}
	if ((err = cudaMalloc((void **)&X, Globals::rowsX * Globals::colsX * 
sizeof(datatype))) != cudaSuccess) {
		printf("cudaMalloc failed cuX: %s\n", cudaGetErrorString(err));
		cleanup();
		exit(-1);
	}
	if ((err = cudaMalloc((void **)&D, Globals::rowsX * K * sizeof(datatype))) != 
cudaSuccess) {
		printf("cudaMalloc failed D: %s\n", cudaGetErrorString(err));
		cleanup();
		exit(-1);
	}
	if ((err = cudaMalloc((void **)&S, K * Globals::colsX * sizeof(datatype))) != 
cudaSuccess) {
		printf("cudaMalloc failed S: %s\n", cudaGetErrorString(err));
		cleanup();
		exit(-1);
	}
	if ((err = cudaMalloc((void **)&I, K * K * sizeof(datatype))) != cudaSuccess) {
		printf("cudaMalloc failed cuI: %s\n", cudaGetErrorString(err));
		cleanup();
		exit(-1);
	}
```

```
	if ((err = cudaMalloc((void **)&KK, K * K * sizeof(datatype))) != cudaSuccess) {
		printf("cudaMalloc failed KK: %s\n", cudaGetErrorString(err));
		cleanup();
		exit(-1);
	}
	if ((err = cudaMalloc((void **)&KK2, K * K * sizeof(datatype))) != cudaSuccess) {
		printf("cudaMalloc failed KK2: %s\n", cudaGetErrorString(err));
		cleanup();
		exit(-1);
	}
	if ((err = cudaMalloc((void **)&KVOXELS, K * Globals::colsX * sizeof(datatype))) !=
cudaSuccess) {
		printf("cudaMalloc failed KVOXELS: %s\n", cudaGetErrorString(err));
		cleanup();
		exit(-1);
	}
	if ((err = cudaMalloc((void **)&KVOXELS2, K * Globals::colsX * sizeof(datatype))) !=
cudaSuccess) {
		printf("cudaMalloc failed KVOXELS: %s\n", cudaGetErrorString(err));
		cleanup();
		exit(-1);
	}
	if ((err = cudaMalloc((void **)&TIMECOMPK, Globals::rowsX * K * sizeof(datatype)))
!= cudaSuccess) {
		printf("cudaMalloc failed TIMECOMPK: %s\n", cudaGetErrorString(err));
		cleanup();
		exit(-1);
	}
	if ((err = cudaMalloc((void **)&TIMECOMPK2, Globals::rowsX * K * sizeof(datatype)))
!= cudaSuccess) {
		printf("cudaMalloc failed TIMECOMPK2: %s\n", cudaGetErrorString(err));
		cleanup();
		exit(-1);
	}
	if ((err = cudaMalloc((void **)&Kv, K * sizeof(datatype))) != cudaSuccess) {
		printf("cudaMalloc failed Kv: %s\n", cudaGetErrorString(err));
		cleanup();
		exit(-1);
	}
	if ((err = cudaMalloc((void **)&TIMECOMPVOXELS, Globals::rowsX * Globals::colsX *
sizeof(datatype))) != cudaSuccess) {
		printf("cudaMalloc failed TIMECOMPVOXELS: %s\n", cudaGetErrorString(err));
		cleanup();
		exit(-1);
	}
	if ((err = cudaMalloc((void**)&d_W, K * sizeof(datatype))) != cudaSuccess) {
		printf("cudaMalloc failed d_W: %s\n", cudaGetErrorString(err));
		cleanup();
		exit(-1);
	}
	if ((err = cudaMalloc((void**)&devInfo, sizeof(int))) != cudaSuccess) {
		printf("cudaMalloc failed devInfo: %s\n", cudaGetErrorString(err));
		cleanup();
		exit(-1);
	}

}

//////////////////////////////////////////////
//              File: matCode.h            //
//////////////////////////////////////////////
#ifndef __matCode__
#define __matCode__

bool readMat (datatype**, mxArray **, unsigned int*, unsigned int*,char*, char*);
```

```
void readingInput(datatype **, mxArray**);

#endif

////////////////////////////////////////////////
//            File: matCode.cu                //
////////////////////////////////////////////////
#include <iostream>
#include "mem.h"
#include "Globals.h"
#include "declarations.h"
#include INCLUDE_FILE(MATLAB_include,mat.h)
#include INCLUDE_FILE(MATLAB_include,matrix.h)


using namespace std;


/* Read MATLAB file format */
bool readMat(datatype** array_dbl, mxArray **mArray, unsigned int* rows_ptr, unsigned int* cols_ptr, char* file, char* array_name) {
      MATFile *pmat;
      if ((pmat = matOpen(file, "r") ) == NULL) {
            cout << "Error opening file " << file << " ..." << endl;
            return false;
      }
      if ((*mArray = matGetVariable(pmat, array_name)) == NULL) {
            cerr << "Error reading existing matrix X..." << endl;
            matClose(pmat);
            return false;
      }
      #ifdef MODE
            *array_dbl = mxGetPr(*mArray);
      #else
            *array_dbl = (datatype*)mxGetData(*mArray);
      #endif
      *rows_ptr = (unsigned int)mxGetM(*mArray);
      *cols_ptr = (unsigned int)mxGetN(*mArray);
      if (matClose(pmat) != 0) {
            cerr << "Error closing file " << file << "..." << endl;
            return false;
      }
      return true;
}


/* Uses readMat function to read input from mat file and stores it to given array */
void readingInput(datatype **array, mxArray** mx) {
      if (!readMat(array, mx, &Globals::rowsX, &Globals::colsX, Xarray_in_PATH, "X")) {
            printf("An error occured while reading input file!");
            cleanup();
            exit(-1);
      }

}

////////////////////////////////////////////////
//             File: kernel.h                 //
////////////////////////////////////////////////
#ifndef __kernel__
#define __kernel__

__global__ void initIdentityGPU(datatype *, int, int);
__global__ void matrixDivNum(datatype *, int, datatype);
__global__ void subtractMatrices(datatype *, datatype *, datatype *, int);
```

```
__global__ void addMatrices(datatype *, datatype *, datatype *, int);
__global__ void wthresh(datatype *, int, datatype);
__global__ void rdivide(datatype *, int, int, datatype *);


#endif // !__kernel__

///////////////////////////////////////////////
//           File: kernel.cu                 //
///////////////////////////////////////////////
#include "declarations.h"

/* Creates an identity matrix with size row x col (needs 2D block) */
__global__ void initIdentityGPU(datatype *matrix, int row, int col) {
      int x = blockDim.x*blockIdx.x + threadIdx.x;
      int y = blockDim.y*blockIdx.y + threadIdx.y;
      if (x < row && y < col) {
            if (x == y) matrix[x*col + y] = 1.0;
            else matrix[x*col + y] = 0.0;
      }
}


/* Divides every element of the given matrix by a specific value */
__global__ void matrixDivNum(datatype *a, int size, datatype val) {
      int index = blockIdx.x *blockDim.x + threadIdx.x;
      if (index < size) a[index] = a[index] / val;
}


/* Subtracts two matrices a,b and stores the result in matrix c */
__global__ void subtractMatrices(datatype *c, datatype *a, datatype *b, int size) {
      int index = blockIdx.x *blockDim.x + threadIdx.x;
      if (index < size) c[index] = a[index] - b[index];
}


/* Adds two matrices a,b and stores the result in matrix c */
__global__ void addMatrices(datatype *c, datatype *a, datatype *b, int size) {
      int index = blockIdx.x *blockDim.x + threadIdx.x;
      if (index < size) c[index] = a[index] + b[index];
}


/* Performs soft thresholding to the input matrix arr
Y = sign(X)*(|X|-T)+ , soft thresholding is wavelet shrinkage ( (x)+ = 0 if x<0; (x)+ =
x, if x >=0 ) */
__global__ void wthresh(datatype *arr, int size, datatype thres) {
      int index = blockIdx.x *blockDim.x + threadIdx.x;
      if (index < size) {
            datatype sign = 1.0;
            if (arr[index] < 0) {
                  arr[index] = -arr[index];
                  sign = -1.0;
            }
            arr[index] = arr[index] - thres;
            if (arr[index] < 0) arr[index] = 0;
            else arr[index] = sign *  arr[index];
      }
}


/* Divides each row of matrix b by the row matrix kv and stores result in matrix b */
__global__ void rdivide(datatype *b, int rowb, int colb, datatype *kv) {
      __shared__  datatype shm[K];
```

```
	int index = blockIdx.x *blockDim.x + threadIdx.x;
	if (threadIdx.x < K) shm[threadIdx.x] = kv[threadIdx.x];
	__syncthreads();
	if (index < rowb*colb) b[index] = b[index] / shm[index % colb];


}

////////////////////////////////////////////////
//             File: Globals.h                //
////////////////////////////////////////////////
#ifndef __Globals__
#define __Globals__

/* Namespace used for global variables */
namespace Globals {
	/* Global values (size of input array) */
	extern unsigned int rowsX;
	extern unsigned int colsX;
};


#endif // !__Globals__

////////////////////////////////////////////////
//            File: Globals.cu                //
////////////////////////////////////////////////
#include "Globals.h"

unsigned int Globals::rowsX = 0; // Rows of input array X

unsigned int Globals::colsX = 0; // Columns of inout array X

////////////////////////////////////////////////
//           File: errorTypes.h              //
////////////////////////////////////////////////
#ifndef __errorTypes__
#define __errorTypes__

const char *curandGetErrorString(curandStatus_t);
const char* cublasGetErrorString(cublasStatus_t);
const char* cusolverGetErrorString(cusolverStatus_t);


#endif // !__errorTypes__

////////////////////////////////////////////////
//           File: errorTypes.cu              //
////////////////////////////////////////////////
#include <curand.h>
#include <cusolverDn.h>
#include "cublas_v2.h"


/* Finds the proper error string based on the error code of a library's curand function
*/
const char *curandGetErrorString(curandStatus_t error) {
	switch (error) {
		case CURAND_STATUS_SUCCESS: return "CURAND_STATUS_SUCCESS";
		case CURAND_STATUS_VERSION_MISMATCH: return "CURAND_STATUS_VERSION_MISMATCH";
		case CURAND_STATUS_NOT_INITIALIZED: return "CURAND_STATUS_NOT_INITIALIZED";
		case CURAND_STATUS_ALLOCATION_FAILED: return
"CURAND_STATUS_ALLOCATION_FAILED";
		case CURAND_STATUS_TYPE_ERROR: return "CURAND_STATUS_TYPE_ERROR";
		case CURAND_STATUS_OUT_OF_RANGE: return "CURAND_STATUS_OUT_OF_RANGE";
		case CURAND_STATUS_LENGTH_NOT_MULTIPLE: return
"CURAND_STATUS_LENGTH_NOT_MULTIPLE";
```

```
		case CURAND_STATUS_DOUBLE_PRECISION_REQUIRED: return 
"CURAND_STATUS_DOUBLE_PRECISION_REQUIRED";
		case CURAND_STATUS_LAUNCH_FAILURE: return "CURAND_STATUS_LAUNCH_FAILURE";
		case CURAND_STATUS_PREEXISTING_FAILURE: return 
"CURAND_STATUS_PREEXISTING_FAILURE";
		case CURAND_STATUS_INITIALIZATION_FAILED: return 
"CURAND_STATUS_INITIALIZATION_FAILED";
		case CURAND_STATUS_ARCH_MISMATCH: return "CURAND_STATUS_ARCH_MISMATCH";
		case CURAND_STATUS_INTERNAL_ERROR: return "CURAND_STATUS_INTERNAL_ERROR";
	}
	return "UNKNOWN ERROR";
}


/* Finds the proper error string based on the error code of a library's cublas function 
*/
const char *cublasGetErrorString(cublasStatus_t status) {
	switch (status) {
		case CUBLAS_STATUS_SUCCESS: return "CUBLAS_STATUS_SUCCESS";
		case CUBLAS_STATUS_NOT_INITIALIZED: return "CUBLAS_STATUS_NOT_INITIALIZED";
		case CUBLAS_STATUS_ALLOC_FAILED: return "CUBLAS_STATUS_ALLOC_FAILED";
		case CUBLAS_STATUS_INVALID_VALUE: return "CUBLAS_STATUS_INVALID_VALUE";
		case CUBLAS_STATUS_ARCH_MISMATCH: return "CUBLAS_STATUS_ARCH_MISMATCH";
		case CUBLAS_STATUS_MAPPING_ERROR: return "CUBLAS_STATUS_MAPPING_ERROR";
		case CUBLAS_STATUS_EXECUTION_FAILED: return "CUBLAS_STATUS_EXECUTION_FAILED";
		case CUBLAS_STATUS_INTERNAL_ERROR: return "CUBLAS_STATUS_INTERNAL_ERROR";
	}
	return "UNKNOWN ERROR";
}


/* Finds the proper error string based on the error code of a library's cusolver function 
*/
const char *cusolverGetErrorString(cusolverStatus_t status) {
	switch (status) {
		case CUSOLVER_STATUS_SUCCESS: return "CUSOLVER_STATUS_SUCCESS";
		case CUSOLVER_STATUS_NOT_INITIALIZED: return 
"CUSOLVER_STATUS_NOT_INITIALIZED";
		case CUSOLVER_STATUS_ALLOC_FAILED: return "CUSOLVER_STATUS_ALLOC_FAILED";
		case CUSOLVER_STATUS_INVALID_VALUE: return "CUSOLVER_STATUS_INVALID_VALUE";
		case CUSOLVER_STATUS_ARCH_MISMATCH: return "CUSOLVER_STATUS_ARCH_MISMATCH";
		case CUSOLVER_STATUS_EXECUTION_FAILED: return "CUSOLVER_STATUS_execution 
failed";
		case CUSOLVER_STATUS_MATRIX_TYPE_NOT_SUPPORTED: return "CUSOLVER_STATUS_Matrix 
not supported";
		case CUSOLVER_STATUS_INTERNAL_ERROR: return "CUSOLVER_STATUS_INTERNAL_ERROR";
	}
	return "UNKNOWN ERROR";

}

///////////////////////////////////////////////
//         File: declarations.h              //
///////////////////////////////////////////////
#ifndef __declarations__
#define __declarations__

#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#define MODE 1   // 0 for float, 1 for double

#if (MODE)
	typedef double datatype;
#else
```

```
	typedef float datatype;
#endif

#define K 10 // The number of sources
#define ITER 200 // The total number of iterations
#define TS 20 // The number of internal iterations of NewCoef
#define TD 20 // The number of internal iterations of NewDict
#define LAMBDA 0.5 // The λ of the problem (depends on the dataset)
#define CCL 1 // Value of the normalization of each atom
#define ES -1 //(optional) If it is selected, the stop criteria of the spatial maps is
changed to the error Es between iterations given by ||S^[n]-S^[n-1]||_{F} < Es
#define ED -1 //(optional) The stop criteria is changed to use an  error check step
||D^[n]-D^[n-1]||_{F} < Ed

/* In order to use read mat files */
#define	Xarray_in_PATH		"./input/X.mat"
#define	MATLAB_include		C:\Program Files\MATLAB\MATLAB Production
Server\R2015a\extern\include\\

#define QUOTEME(x)		QUOTEME_1(x)
#define QUOTEME_1(x)		#x
#define INCLUDE_F(x,y)		QUOTEME(x ## y)
#define INCLUDE_FILE(x,y)	INCLUDE_F(x,y)


#endif

///////////////////////////////////////////////
//		File: cuLibs.h			//
///////////////////////////////////////////////
#ifndef __cuLibs__
#define __cuLibs__

datatype frobeniusNorm(cublasHandle_t, datatype *, int);
void randn(curandGenerator_t, datatype *, int);
datatype findMaxSqrtEigenvalue(cusolverDnHandle_t, datatype *, datatype *, datatype *,
int *, const int);
void gpu_blas_mmul(cublasHandle_t, const datatype *, const datatype *, datatype *, int,
const int, const int, const int, const int);


#endif

///////////////////////////////////////////////
//		File: cuLibs.cu			//
///////////////////////////////////////////////
#include <iostream>
#include <cusolverDn.h>
#include <curand.h>
#include "cublas_v2.h"
#include "declarations.h"
#include "errorTypes.h"
#include "mem.h"
#include INCLUDE_FILE(MATLAB_include,mat.h)
#include INCLUDE_FILE(MATLAB_include,matrix.h)
#include "primaryFunctions.h"


/* Calculates the frobenius norm of the given array */
datatype frobeniusNorm(cublasHandle_t handle, datatype *Array, int size) {
	datatype retval;
	cublasStatus_t stat;
	#if (MODE)
		if ((stat = cublasDnrm2(handle, size, Array, 1, &retval)) !=
CUBLAS_STATUS_SUCCESS) {
			printf("cublasDnmr2 failed: %s\n", cublasGetErrorString(stat));
```

```
                cleanup();
                exit(-1);
            }
        #else
            if ((stat = cublasSnrm2(handle, size, Array, 1, &retval)) != 
CUBLAS_STATUS_SUCCESS) {
                printf("cublasSnmr2 failed: %s\n", cublasGetErrorString(stat));
                cleanup();
                exit(-1);
            }
        #endif
        return retval;
}


/* Generate numbers using normal distribution
   http://docs.nvidia.com/cuda/curand/host-api-overview.html#axzz4SrFExYZh */
void randn(curandGenerator_t generator, datatype *Arr, int size) {
        curandStatus_t stat;
        #if (MODE)
            if ((stat = curandGenerateNormalDouble(generator, Arr, size, (datatype)0, 
(datatype)1)) != CURAND_STATUS_SUCCESS) {
                printf("curandGenerateNormalDouble failed: %s\n", 
curandGetErrorString(stat));
                cleanup();
                exit(-1);
            }
        #else
            if ((stat = curandGenerateNormal(generator, Arr, size, (datatype)0, 
(datatype)1)) != CURAND_STATUS_SUCCESS) {
                printf("curandGenerateNormal failed: %s\n", curandGetErrorString(stat));
                cleanup();
                exit(-1);
            }
        #endif
}


/* Returns the square root of the maximum eigenvalue of the given array using power 
method */
datatype findMaxSqrtEigenvalue(cusolverDnHandle_t cusolverH, datatype *dA, datatype *W, 
datatype *d_W, int *devInfo, const int m) {
        int lwork = 0;
        cudaError_t err;
        const int lda = m;
        datatype *d_work = 0;
        cusolverStatus_t stat;
        #if (MODE)
            if ((stat = cusolverDnDsyevd_bufferSize(cusolverH, CUSOLVER_EIG_MODE_NOVECTOR, 
CUBLAS_FILL_MODE_LOWER, m, dA, lda, d_W, &lwork)) != CUSOLVER_STATUS_SUCCESS) {
                printf("cusolverDnDsyevd_bufferSize failed: %s\n", 
cusolverGetErrorString(stat));
                cleanup();
                exit(-1);
            }
        #else
            if ((stat = cusolverDnSsyevd_bufferSize(cusolverH, CUSOLVER_EIG_MODE_NOVECTOR, 
CUBLAS_FILL_MODE_LOWER, m, dA, lda, d_W, &lwork)) != CUSOLVER_STATUS_SUCCESS) {
                printf("cusolverDnSsyevd_bufferSize failed: %s\n", 
cusolverGetErrorString(stat));
                cleanup();
                exit(-1);
            }
        #endif
        if ((err = cudaMalloc((void**)&d_work, lwork * sizeof(datatype))) != cudaSuccess) {
```

```
		printf("cudaMalloc d_work failed: %s\n", cudaGetErrorString(err));
		if (d_work && (err = cudaFree(d_work)) != cudaSuccess) printf("cudaFree
d_work: %s\n", cudaGetErrorString(err));
		cleanup();
		exit(-1);
	}
	#if (MODE)
		if ((stat = cusolverDnDsyevd(cusolverH, CUSOLVER_EIG_MODE_NOVECTOR,
CUBLAS_FILL_MODE_UPPER, m, dA, lda, d_W, d_work, lwork, devInfo)) !=
CUSOLVER_STATUS_SUCCESS) {
			printf("cusolverDnDsyevd failed: %s\n", cusolverGetErrorString(stat));
			if (d_work && (err = cudaFree(d_work)) != cudaSuccess) printf("cudaFree
d_work: %s\n", cudaGetErrorString(err));
			cleanup();
			exit(-1);
		}
	#else
		if ((stat = cusolverDnSsyevd(cusolverH, CUSOLVER_EIG_MODE_NOVECTOR,
CUBLAS_FILL_MODE_UPPER, m, dA, lda, d_W, d_work, lwork, devInfo)) !=
CUSOLVER_STATUS_SUCCESS) {
			printf("cusolverDnSsyevd failed: %s\n", cusolverGetErrorString(stat));
			if (d_work && (err = cudaFree(d_work)) != cudaSuccess) printf("cudaFree
d_work: %s\n", cudaGetErrorString(err));
			cleanup();
			exit(-1);
		}
	#endif
	if ((err = cudaMemcpy(W, d_W, m * sizeof(datatype), cudaMemcpyDeviceToHost)) !=
cudaSuccess) {
		printf("cudaMemcpy W failed: %s\n", cudaGetErrorString(err));
		if (d_work && (err = cudaFree(d_work)) != cudaSuccess) printf("cudaFree
d_work: %s\n", cudaGetErrorString(err));
		cleanup();
		exit(-1);
	}
	if (d_work && (err = cudaFree(d_work)) != cudaSuccess) printf("cudaFree d_work:
%s\n", cudaGetErrorString(err));
	return sqrt(W[m - 1]);
}


/* Multiplies two matrices (category=0:C=A*B, category=1:C=A'*B, category=2:C=A*B') */
void gpu_blas_mmul(cublasHandle_t handle, const datatype *A, const datatype *B, datatype
*C,
	int category, const int rowA, const int colA, const int rowB, const int colB) {

	int lda, ldb, ldc;
	const datatype alf = 1;
	const datatype bet = 0;
	const datatype *alpha = &alf;
	const datatype *beta = &bet;
	cublasStatus_t status;
	if (category == 0) { // C=A*B ( http://peterwittek.com/cublas-matrix-c-style.html )
		lda = colB; ldb = colA; ldc = colB;
		if (colA != rowB) {
			printf("colA != rowB. Cannot multiply the matrices\n");
			cleanup();
			exit(-1);
		}
		#if (MODE)
			status = cublasDgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, colB, rowA, colA,
alpha, B, lda, A, ldb, beta, C, ldc);
		#else
			status = cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, colB, rowA, colA,
alpha, B, lda, A, ldb, beta, C, ldc);
```

```
		#endif
	}
	else if (category == 1) { // C = A'*B ( 
http://stackoverflow.com/questions/14595750/transpose-matrix-multiplication-in-cublas-howto )
		lda = colA; ldb = colB; ldc = colB;
		if (rowA != rowB) {
			printf("rowA != rowB. Cannot multiply the matrices\n");
			cleanup();
			exit(-1);
		}
		#if (MODE)
			status = cublasDgemm(handle, CUBLAS_OP_N, CUBLAS_OP_T, colB, colA, rowB, 
alpha, B, ldb, A, lda, beta, C, ldc);
		#else
			status = cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_T, colB, colA, rowB, 
alpha, B, ldb, A, lda, beta, C, ldc);
		#endif
	}
	else if (category == 2) { // C=A*B'
		lda = colA; ldb = colB; ldc = rowB;
		if (colA != colB) {
			printf("colA != colB. Cannot multiply the matrices\n");
			cleanup();
			exit(-1);
		}
		#if (MODE)
			status = cublasDgemm(handle, CUBLAS_OP_T, CUBLAS_OP_N, rowB, rowA, colB, 
alpha, B, ldb, A, lda, beta, C, ldc);
		#else
			status = cublasSgemm(handle, CUBLAS_OP_T, CUBLAS_OP_N, rowB, rowA, colB, 
alpha, B, ldb, A, lda, beta, C, ldc);
		#endif
	}
	else {
		printf("gpu_blas_mmul : not valid status %d \n", category);
		cleanup();
		exit(-1);
	}
	if (status != cudaSuccess) {
		printf("Cublas multiplication failed: %s\n", cublasGetErrorString(status));
		cleanup();
		exit(-1);
	}

}
```

2. Αλγόριθμος k-SVD

```
%% fMRI & k-SVD test file
% This file automatically calculates the speedup,
% the speedup percentage as well as the difference in
% the final error between CPU and GPU execution of the
% k-SVD algorithm.
%
%% Initializations %%
% load data
load('300_3600.mat');
clearvars -except X

% FMRI data dimensions
% Rows(data) = number_of_time_components
% Columns(data) = number_of_signals
params.data = X;

% Dictionary dimensions
% Rows(Dictionary) = Rows(data)
% Columns(Dictionary) = number_of_sources
sources = 9;
D = randn(size(X,1),sources);

% Sparsity level target
k = 6;

%% Run k-SVD training %%
disp('=============================================');
disp('================= MATLAB ====================');
disp('=============================================');

params.data = X;
params.Tdata = k;
params.dictsize = sources;
params.iternum = 30;
params.muthresh = 0.8;
params.memusage = 'high';
params.codemode = 'sparsity';
params.initdict = D;

tic;
[Dksvd,g,err] = ksvd(params,'it');
time = toc;
fprintf('\n>Time for k-SVD training in CPU: %.6f\n',time);


%% Write input to folder that CUDA project will use %%
save('input\X.mat','X');
save('input\D.mat','D');

%% Run CUDA
disp('=============================================');
disp('================== CUDA =====================');
disp('=============================================');
path = 'C:\Users\Thesis\Dropbox\Visual Studio 2015\Projects\K-
SVD\x64\Release\';
[status,cmdout] = system(strcat(path,'K-SVD.exe'));
```

```
disp(cmdout);

%% Display statistics

disp('============================================');
disp('================ RESULTS ===================');
disp('============================================');

k = strfind(cmdout,'RMSE = ');
cuda_err = str2double(cmdout(k(end) + 7:k(end) + 14));
fprintf('>Final errors\n  MATLAB = %.5f, CUDA = %.5f\n',err(end),cuda_err);
fprintf('>Difference = %.5f\n',abs(err(end) - cuda_err));
k = strfind(cmdout,'(');
speedup = time / str2double(cmdout(k(1) + 1:k(1) + 5));
fprintf('>Speedup = %.2f (%.2f %%)\n\n', speedup, (1 - 1/speedup)*100);

%% END

function [newVectors] = remmean(vectors)
%REMMEAN - remove the mean from vectors
%
% [newVectors, meanValue] = remmean(vectors);
%
% Removes the mean of row vectors.
% Returns the new vectors and the mean.
%
% This function is needed by FASTICA and FASTICAG

% @(#)$Id: remmean.m,v 1.2 2003/04/05 14:23:58 jarmo Exp $

newVectors = zeros (size (vectors));
meanValue = mean (vectors')';
newVectors = vectors - meanValue * ones (1,size (vectors, 2));
end
```

```
/******************************/
/* CUDA IMPLEMENTATION - KSVD */
/******************************/
#ifdef _DEBUG
#define _CRTDBG_MAP_ALLOC
#include <stdlib.h>
#include <crtdbg.h>
#endif
#include <vector>
#include <iostream>
#include <iomanip>
#include <windows.h>
#include <thrust/device_vector.h>
#include <thrust/version.h>
#include "GlobalDeclarations.cuh"
#include "Globals.cuh"
#include "FileManager.cuh"
#include "HostMemory.cuh"
#include "DeviceMemory.cuh"
#include "Timer.cuh"
#include "Utilities.cuh"
#include "CudaAlgorithm.cuh"
#include "OMP.cuh"
#include "DictionaryUpdate.cuh"
#include "KSVD.cuh"
#include "Algorithms.cuh"
```

```
/****************
*   NAMESPACES   *
*   __________    *
****************/
using namespace       std;
using DeviceSpace     = DeviceMemory<datatype>;
using HostSpace       = HostMemory<datatype>;
using kSVD            = KSVD;

/**********************
*   FUNCTIONS - main   *
*   _______________     *
**********************/
int main(int argc, char** argv)
{
      #ifdef _DEBUG
            {
      #endif
      cout << "Initializing! Please wait..." << endl;{
      /*------------------*/
      /* Required objects */
      /*------------------*/
      // Files' manipulator
      FileManager fileManager;
      // Host memory space
      HostSpace hostMemory(fileManager);
      if (hostMemory.get(HostSpace::MAX_ELEMENTS) == NULL) {
            cerr << "Host Memory initialization error! Aborting..." << endl;
            return -1;
      }
      // Device memory space
      DeviceSpace deviceMemory(hostMemory);
      if (deviceMemory.get(DeviceSpace::MAX_ELEMENTS) == NULL) {
            cerr << "Device Memory initialization error! Aborting..." << endl;
            return -1;
      }
      // Timer for profiling
      Timer timer;
      if (!timer.isValid()) {
            cerr << "Timer initialization error! Aborting..." << endl;
            return -1;
      }
      // Class wrapper for the K-SVD execution
      kSVD KSVD(&deviceMemory, &hostMemory);
      if (!KSVD.isValid()) {
            cerr << "KSVD initialization error! Aborting..." << endl;
            return -1;
      }
      /*----------------*/
      /* Initialization */
      /*----------------*/
      // Set reduction buffer now
      // that we have device memory initialized
      if (Algorithms::Reduction::setReductionBuffer(
                  deviceMemory.get(DeviceSpace::REDUCTION_BUFFER)) == false){

            cerr << "Reduction buffer initialization error! Aborting..." << endl;
            return -1;
      }
      // Set also the random generator buffer now...
      if (Algorithms::RandomPermutation::setGeneratorBuffer(
            hostMemory.get_RND_GNRT_buffer(), Globals::colsD ) == false) {

            cerr << "Random Generator buffer initialization error! Aborting..." << endl;
            return -1;
```

```
	}
	// Print mode of operation
	#ifdef DOUBLE
		cout << "Double precision mode." << endl;
	#else
		cout << "Single precision mode." << endl;
	#endif
	// Print sizes
	cout << "Dictionary size: " << Globals::rowsD << " x " << Globals::colsD << endl;
	cout << "Input data size: " << Globals::rowsX << " x " << Globals::colsX << endl;
	/*-----------*/
	/* Execution */
	/*-----------*/
	cout << "Running..." << endl << endl;
	timer.GPUtic();
	if (KSVD.ExecuteIterations(NUMBERofITERATIONS) == false) {
		cerr << "An error occured while executing KSVD!...Aborting" << endl;
		return -1;
	}
	float time = timer.GPUtoc();
	/*-----------*/
	/* Printing  */
	/*-----------*/
	cout << endl << "Final Execution Time =>\t\t" << setprecision(3) << std::fixed
		<< time << " ms (" << setprecision(2) << std::fixed
		<< time / 1000.0 << " sec)" << endl;
	cout << "Total Memory Used (program) =>\t" << setprecision(2) << std::fixed
		<< deviceMemory.getConsumed() / 1024.0 / 1024.0 << " MB" << endl;
	cout << "Total Available Memory =>\t" << setprecision(2) << std::fixed
		<< deviceMemory.getTotal() / 1024.0 / 1024.0 << " MB" << endl;
	cout << "Total Memory Used (OS) =>\t" << setprecision(2) << std::fixed
		<< deviceMemory.getUsed() / 1024.0 / 1024.0 << " MB" << endl;
	cout << "Total Free Memory =>\t\t" << setprecision(2) << std::fixed
		<< deviceMemory.getFree() / 1024.0 / 1024.0 << " MB" << endl;
	cout << "Memory Utilization =>\t\t" << setprecision(1) << std::fixed
		<< 100 * (deviceMemory.getConsumed() / deviceMemory.getFree()) << " %" <<
endl;
	/*------------*/
	/* Write back */
	/*------------*/
	thrust::host_vector<datatype> temp(Globals::colsD*Globals::colsX);
	thrust::copy(
		deviceMemory.getThrust(DeviceSpace::SELECTED_ATOMS).begin(),
		deviceMemory.getThrust(DeviceSpace::SELECTED_ATOMS).end(),
		temp.begin());
	datatype* ptr = thrust::raw_pointer_cast(&(temp[0]));
	if (fileManager.writeTemporary(ptr,   Globals::colsD, Globals::colsX, "CUDA_Gamma")
== false) {
		cerr << "Cannot write file for temporary array! Aborting..." << endl;
		return -1;
	}
	/*---------*/
	/* Cleanup */
	/*---------*/
	cudaDeviceSynchronize();
	cout << "-----------------------------------------------------" << endl;
	cudaError_t err;
	while( (err = cudaGetLastError()) != cudaSuccess){
		cerr << "Error: An error occured during the execution of the program: " <<
			cudaGetErrorString(err) << endl;
	}}
	cout << "# Exiting!" << endl;
	#ifdef _DEBUG
		}
		_CrtDumpMemoryLeaks();
```

```
#endif
return 0;
}


/*------------------------------*/
/*  REDUCTION ONLY HEADER FILE  */
/*------------------------------*/
#pragma once
#ifndef _REDHF_
#define _REDHF_


////////////////////////////////////////////////
// This file serves as a separator between  //
// reduction kernel functions and reduction //
// high-level API ( in Reduction.cu )       //
////////////////////////////////////////////////

/***************************
*         MACROS          *
*         ______            *
**************************/
// Macro for finding the minimum of 2 numbers
#define MINIMUM(a,b) ( a < b ? a : b )
#define MAXIMUM(a,b) ( a > b ? a : b )

/*********************************
*         DECLARATIONS          *
*         ____________            *
********************************/
// K-SVD operations
__global__
void deviceReduceKernel_2D_square_values(datatype *, int, datatype*);
// Dict. Upd. operations
__global__
void multi_reduction_squared_sum_STAGE1(datatype*, datatype*, datatype*, int);
__global__
void multi_reduction_squared_sum_STAGE2_together_collincomb(datatype*, datatype*,
datatype*, int, datatype*,

      datatype*, datatype*, unsigned int);
__global__
void multi_reduction_smallGamma_times_gammaJ_STAGE1(datatype *, datatype*, datatype*,
datatype*, datatype*, unsigned int);
__global__
void multi_reduction_smallGamma_times_gammaJ_STAGE2(datatype*, datatype*, unsigned int);
__global__
void rowlincomb(datatype*, datatype*, datatype*, datatype*, unsigned int, datatype*,
datatype*, datatype*,
                      datatype*, unsigned int);
__global__
void SCase_err_stage1(datatype*, datatype*, unsigned int, unsigned int, datatype*,
datatype*, datatype*, datatype*);
__global__
void SCase_err_stage2(datatype*, int, datatype*, unsigned int, datatype*, datatype*);
__global__
void SCase_err_stage3(int, datatype*, datatype*);
__global__
void SCase_norm(datatype*, datatype*,unsigned int, datatype*, datatype*, unsigned int);
// Error computation operations
__global__
void RMSE_stage1(datatype*, datatype*, unsigned int, datatype*);
__global__
void RMSE_stage2(datatype*, unsigned int, unsigned int, unsigned int, datatype*);
```

```
__global__
void RMSE_stage3(unsigned int, unsigned int, unsigned int);
// Clear Dict. operations
__global__
void device_max_err_STAGE1(datatype*, int, datatype*);
__global__
void device_max_err_STAGE2(datatype*, int, datatype*);
__global__
void device_max(datatype*, unsigned int, datatype*, datatype*, unsigned int);
__global__
void EUnorm(datatype*, datatype*, unsigned int);

/**********************
*         END         *
*         ___          *
**********************/


#endif // !REDHF

/*------------------------------*/
/*  REDUCTION ONLY HEADER FILE  */
/*------------------------------*/
#pragma once
#ifndef _REDBASE_
#define _REDBASE_


/////////////////////////////////////////////
// This file serves as a separator between //
// reduction base functions and reduction  //
// higher-level kernels.                   //
/////////////////////////////////////////////

/*************************************
*   REDUCTION-SPECIFIC USER TYPES    *
*     _____________________________   *
*************************************/

/*------------------------------------*/
/* Finding the index of the maximum    */
/* or minimum value of a set of values */
/* requires interleaved, multiple      */
/* reduction units per operation (one  */
/* for max and one for index).         */
/*------------------------------------*/
struct DoubleReductionType {
     datatype    value;
     int               index;
};

/*************************************
*     REDUCTION BASE FUNCTIONS       *
*       ________________________      *
*************************************/

////////////////////////////////////
// Reduction within a single warp //
////////////////////////////////////
__inline__ __device__
DoubleReductionType warpReduceMaximumIndex(DoubleReductionType val) {
     for (int offset = warpSize / 2; offset > 0; offset /= 2) {
          datatype shfl_val = __shfl_down(val.value, offset);
          int shfl_ind = __shfl_down(val.index, offset);
          if (shfl_val > val.value) {
```

```
                val.value = shfl_val;
                val.index = shfl_ind;
            }
        }
        return val;
}

////////////////////////////////////
// Reduction within a single warp //
////////////////////////////////////
__inline__ __device__
datatype warpReduceMax(datatype val) {
        for (int offset = warpSize / 2; offset > 0; offset /= 2) {
            datatype shfl_val = __shfl_down(val, offset);
            if (shfl_val > val)
                val = shfl_val;
        }
        return val;
}

////////////////////////////////////
// Reduction within a single warp //
////////////////////////////////////
__inline__ __device__
datatype warpReduceSum(datatype val) {
        for (int offset = warpSize / 2; offset > 0; offset /= 2)
            val += __shfl_down(val, offset);
        return val;
}

/////////////////////////////////////
// Reduction within a single block //
/////////////////////////////////////
__inline__ __device__
datatype blockReduceSum(datatype val) {

        // Max. block size = 1024 (32 x 32) = 32 warps of 32 threads
        static __shared__
            datatype shared[32];  // Shared mem for 32 partial sums
        int lane = threadIdx.x % warpSize;
        int wid = threadIdx.x / warpSize;

        val = warpReduceSum(val);  // Each warp performs partial reduction

        if (lane == 0)                  // Write reduced value to shared memory
            shared[wid] = val;

        __syncthreads();            // Wait for all partial reductions

                                        //read from shared memory only if that warp
existed
        val = (threadIdx.x < blockDim.x / warpSize) ? shared[lane] : 0;

        if (wid == 0)                   //Final reduce within first warp
            val = warpReduceSum(val);

        return val;
}

/////////////////////////////////////
// Reduction within a single block //
// (maximum version)               //
/////////////////////////////////////
__inline__ __device__
DoubleReductionType blockReduceMaximumIndex(DoubleReductionType val) {
```

```
	// Max. block size = 1024 (32 x 32) = 32 warps of 32 threads
	static __shared__
		DoubleReductionType shared[32];  // Shared mem for 32 partial sums
	int lane = threadIdx.x % warpSize;
	int wid = threadIdx.x / warpSize;

	val = warpReduceMaximumIndex(val);     // Each warp performs partial reduction

	if (lane == 0)                     // Write reduced value to shared memory
		shared[wid] = val;

	__syncthreads();            // Wait for all partial reductions

	                                    //read from shared memory only if that warp
existed
	if (threadIdx.x < blockDim.x / warpSize) {
		val = shared[lane];
	}
	else {
		val.value = 0;
	}

	if (wid == 0)                      //Final reduce within first warp
		val = warpReduceMaximumIndex(val);

	return val;
}

/************************************
*          UNARY FUNCTIONS          *
*          _______________            *
************************************/

/*---------------------------------*/
/* Action performed on data before */
/*  reduction of sum of elements!  */
/*      'SUM OF SQUARE VALUES'     */
/*---------------------------------*/
__inline__ __device__
datatype sqval(datatype val) {
	return val*val;
}

/*---------------------------------*/
/* Action performed on data after  */
/*  reduction of sum of elements!  */
/*      'INVERTED SQUARE ROOT'     */
/*---------------------------------*/
__inline__ __device__
datatype invSQRT(datatype val) {
	return 1.0 / sqrt(val);
}

/******************************
*          GENERATORS          *
*          __________            *
******************************/
// This functions acts as a random shuffle generator
// on the device by taking an index and then returning
// the new index that the specific position maps to.

/*-----------------------------------------------------*/
/* This function acts as a random shuffle generator    */
/* on the device by taking an index and then returning */
```

```
/* the new index that the specific position maps to.   */
/*----------------------------------------------------*/
__inline__ __device__
unsigned int myGenerator(unsigned int pos, unsigned int N) {
      // We return the permutation: (1:length) so that every position maps to its self.
      return pos;
}

/**********************
*         END         *
*          ___          *
**********************/


#endif // !REDBASE

/*---------------------------*/
/*   ALGORITHMIC TOOLS CLASS  */
/*   - REDUCTIONS SUBCLASS -  */
/*       IMPLEMENTATION       */
/*---------------------------*/
#include <iostream>
#include "GlobalDeclarations.cuh"
#include "Algorithms.cuh"
#include "ReductionKernels.cuh"

using namespace std;


/*********************************
*         DECLARATIONS          *
*          ____________          *
*********************************/
extern __device__
datatype* experimental_array;

/*************************************
*         MEMBER FUNCTIONS          *
*          ________________          *
*************************************/
/*--------------------------------------------*/
/* Reduction in 1D layout (i.e. vector )      */
/* of __M__A__X__I__M__U__M__ squared value.  */
/*--------------------------------------------*/
__host__
bool Algorithms::Reduction::Maximum_squared(
      datatype* in, unsigned int colsD, datatype* usecount,
      datatype* replaced, unsigned int j) {

      // We again assume colsD <= 32 to utilize a single warp
      device_max << <1, 32 >> > (in, colsD, usecount, replaced, j);
      return true;
}

/*-------------------------------------------*/
/* Reduction in 1D layout (i.e. vector) of   */
/* __M__A__X__I__M__U__M__ value of its      */
/* elements as well as their indices.        */
/*-------------------------------------------*/
__host__
bool Algorithms::Reduction::Maximum(datatype* in, int size, datatype* out) {
      dim3 block(1024);
      dim3 grid((size + block.x - 1) / block.x);
      device_max_err_STAGE1 << < grid, block >> > (in, size, out);
      int N;
```

```
	while (grid.x > 1) {
		N = grid.x;
		grid.x = (N + block.x - 1) / block.x;
		device_max_err_STAGE2 << < grid, block >> > (in, N, out);
	}
	return true;
}

/*-----------------------------------------------*/
/* Reduction in 2D layout (i.e. matrix columns)  */
/* of __S__U__M__ of squared values of elements. */
/* Result is then square rooted and inverted.    */
/*-----------------------------------------------*/
__host__
bool Algorithms::Reduction::reduce2D_Sum_SQUAREDelements(
	datatype* in, int rows, int cols, datatype* out,
	unsigned int recommended) {

	dim3 block(recommended);
	dim3 grid(cols);
	deviceReduceKernel_2D_square_values << <grid, block >> > (in, rows, out);
	return true;
}

/*--------------------------------------*/
/* Reduction in 1D layout (i.e. vector) */
/* of __S__U__M__ of squared values of  */
/* elements (more than 1024).           */
/*--------------------------------------*/
__host__
bool Algorithms::Reduction::reduce1D_Batched_Sum_TOGETHER_collincomb(
	datatype* in, datatype* out, datatype* counters, int colsD, int colsX,
	datatype *A, datatype* columns, datatype* out2,  unsigned int rowsX) {

	dim3 block(1024);
	dim3 grid((colsX + block.x - 1) / block.x, colsD);
	multi_reduction_squared_sum_STAGE1 << <grid, block >> > (in, out, counters, colsX);
	grid.x = colsD;
	grid.y = 2;
	multi_reduction_squared_sum_STAGE2_together_collincomb << <grid, block >> >(
		in, out, counters, colsX,
		A, columns, out2, rowsX);
	return true;
}

/*------------------------------------------------*/
/* Reduction in 2D layout (i.e. matrix columns)   */
/* of linear combination of input matrix and      */
/* given columns (filtered by the specified rows) */
/*------------------------------------------------*/
__host__
bool Algorithms::Reduction::reduce2D_rowlincomb_plus_nrm2_plus_mul(
	datatype *A, datatype* x, datatype* out, datatype* cols,
	unsigned int rowsX, unsigned int colsX,     datatype* counters,
	unsigned int recommended, datatype* out2, datatype* D,
	datatype* out3, unsigned int colsD) {

	rowlincomb << <colsX + colsD + 1, recommended >> > (
		A, x, out, cols, rowsX, counters,
		out2, D, out3, colsX);
	return true;
}

/*-------------------------------------------*/
/* Reduction in 2D layout (i.e. matrix rows) */
```

```
/* of dot products with given matrix         */
/* (filtered by the specified cols).          */
/*-------------------------------------------*/
__host__
bool Algorithms::Reduction::reduce2D_dot_products_modified(
	datatype* Gamma, datatype* columns, datatype* gammaJ, datatype* out, datatype*
counters,
	unsigned int pJ, unsigned int colsD, unsigned int colsX) {

	// We also assume that colsD <= 32
	dim3 block(1024);
	dim3 grid((colsX + block.x - 1) / block.x, colsD);
	multi_reduction_smallGamma_times_gammaJ_STAGE1 << <grid, block >> > (
		Gamma, columns, gammaJ, out,
		counters, pJ);
	multi_reduction_smallGamma_times_gammaJ_STAGE2 << <colsD, block >> > (
		out, counters, pJ);
	return true;
}

/*-------------------------------*/
/* Euclidean norm of the vector. */
/*-------------------------------*/
__host__
bool Algorithms::Reduction::euclidean_norm(
	datatype* in, datatype* out,
	unsigned int length, unsigned int recommended) {

	// We assume length i.e. rowsX <= 1024
	EUnorm << <1, recommended >> > (in, out, length);
	return true;
}

/*-----------------------------------------------*/
/* Compute the Round Mean Square Error between a */
/* matrix X and its approximation matrix X~      */
/*-----------------------------------------------*/
__host__
bool Algorithms::Reduction::reduce_RMSE(
	datatype* X, datatype* Xappr, datatype* out,
	unsigned int rows, unsigned int cols, unsigned int iter,
	unsigned int recommended_threads, datatype* bitmap) {

	dim3 block(recommended_threads);
	dim3 grid(cols);
	RMSE_stage1 << <grid, block >> > (X, Xappr, rows, out);
	/********************/
	block.x = 512;
	grid.x = (cols + block.x - 1) / block.x;
	RMSE_stage2 << < grid, block >> > (out, cols, rows*cols, iter, bitmap);
	/********************/
	int N;
	while (grid.x > 1) {
		N = grid.x;
		grid.x = (N + block.x - 1) / block.x;
		RMSE_stage3 << < grid, block >> > (N, rows*cols, iter);
	}
	return true;
}

/*------------------------------------------------*/
/* Compute the Round Mean Square Error between a  */
/* matrix X and its approximation matrix X~       */
/* ( modified for use only in Dictionary Update's */
/* special case )                                 */
```

```
/*-------------------------------------------------*/
__host__
bool Algorithms::Reduction::reduce_ERROR_for_special_case(
      datatype* X, datatype* Xappr, datatype* out,
      unsigned int rows, unsigned int cols, datatype* counters,
      datatype* unused, datatype* unsig_counter,
      unsigned int recommended_threads) {

      dim3 block(recommended_threads);
      dim3 grid(MIN(cols,MAX_SIGNALS));
      SCase_err_stage1 << <grid, block >> > (X, Xappr, rows, cols, out, counters, unused,
unsig_counter);
      /*******************/
      int N = grid.x;
      block.x = 1024;
      grid.x = (N + block.x - 1) / block.x;
      SCase_err_stage2 << < grid, block >> > (out, N, Xappr, cols, counters,
unsig_counter);
      /*******************/
      while (grid.x > 1) {
            N = grid.x;
            grid.x = (N + block.x - 1) / block.x;
            SCase_err_stage3 << < grid, block >> > (N, Xappr, counters);
      }
      return true;
}

/*-------------------------------*/
/* Euclidean norm of the vector. */
/* ONLY for the special case!    */
/*-------------------------------*/
__host__
bool Algorithms::Reduction::SCase_EU_norm(
      datatype* in, datatype* out, unsigned int length,
      unsigned int recommended, datatype* counters, datatype* unused,
      unsigned int colsX) {

      // We assume length <= 1024
      SCase_norm << <1, recommended >> > (in, out, length, counters, unused, MIN(colsX,
MAX_SIGNALS));
      return true;
}

/*--------------------------------------*/
/* Set the size of the reduction buffer */
/* to be used in the calculations.      */
/*--------------------------------------*/
bool Algorithms::Reduction::setReductionBuffer(datatype* b) {
      // Implicit Synchronization
      cudaError_t cet;
      if ((cet = cudaMemcpyToSymbol(
                  experimental_array, &b, sizeof(datatype*), 0, cudaMemcpyHostToDevice)
            ) != cudaSuccess) {

            cerr << "CudaMemcpyToSymbol (reduction buffer) failed: " <<
cudaGetErrorString(cet) << endl;
            return false;
      }
      return true;

}

/*---------------------------*/
/*   ALGORITHMIC TOOLS CLASS  */
/*   - RAND.PERM. SUBCLASS -  */
```

```
/*        IMPLEMENTATION       */
/*---------------------------*/
#include <iostream>
#include <algorithm>
#include "GlobalDeclarations.cuh"
#include "Algorithms.cuh"

using namespace std;


/**********************************
*          DECLARATIONS          *
*           ____________          *
*********************************/
unsigned int* output_array = NULL;

/**********************************
*         CONSTANT FIELDS        *
*          _______________        *
*********************************/
unsigned int static_values[9] = {
      1, 2, 6, 5, 0, 4, 7, 3, 8
};

/**************************************
*          MEMBER FUNCTIONS          *
*           ________________          *
*************************************/
/*---------------------------------------*/
/* Generate a set of random permutations */
/*---------------------------------------*/
unsigned int* Algorithms::RandomPermutation::generateRandPerm(unsigned int perm_size) {
      #if STATIC_GENERATION
            return static_values;
      #else
            random_shuffle(output_array, output_array + perm_size);
            return output_array;
      #endif
}

/*--------------------------------------*/
/* Set the size of the generator buffer */
/* to be used in the calculations.      */
/*--------------------------------------*/
bool Algorithms::RandomPermutation::setGeneratorBuffer(unsigned int* b, int permsize) {
      output_array = b;
      // Initialize the given buffer space
      for (int j = 0; j < permsize; j++) {
            b[j] = j;
      }
      cudaDeviceSynchronize();
      return true;

}

/*------------------------------*/
/*     ALGORITHMIC TOOLS CLASS    */
/*   - MULTIPLICATION SUBCLASS -  */
/*         IMPLEMENTATION         */
/*------------------------------*/
#include <iostream>
#include "GlobalDeclarations.cuh"
#include "Algorithms.cuh"
#include "ErrorHandler.cuh"
#include "ReductionBase.cuh"
```

```
using namespace std;


/*********************************
*         DECLARATIONS          *
*          ____________          *
*********************************/
__global__
void device_A_times_x_modified(datatype*, datatype*, datatype*, unsigned int, unsigned
int,  datatype*,
                                            datatype*, datatype*);
__global__
void device_normal_x_times_A_modified(datatype*, datatype*, datatype*, unsigned int,
datatype*, datatype*,
                                                       datatype*, datatype*, unsigned int,
     datatype*, datatype*, datatype*,
                                                       unsigned int, unsigned int);


/*************************************
*         MEMBER FUNCTIONS          *
*          ________________          *
*************************************/
/*----------------------------------*/
/* Matrix-Matrix Multiplication Set  */
/* _________________________________*/
/* Library: Cublas                  */
/*        { using cublas<T>GEMM }      */
/* Operation performed format       */
/*        ==> C = A'*B or AT*B         */
/*----------------------------------*/
bool Algorithms::Multiplication::AT_times_B(cublasHandle_t handle,
      datatype* A, int rowsA, int colsA,
      datatype* B, int rowsB, int colsB,
      datatype* C) {

      cublasStatus_t status;
      const datatype alpha = 1.0;
      const datatype beta = 0.0;
      if ((
            #ifdef DOUBLE
                  status = cublasDgemm(handle, CUBLAS_OP_T, CUBLAS_OP_N,
                        colsA, colsB, rowsB, &alpha, A, rowsA, B, rowsB, &beta, C, colsA)
            #else
                  status = cublasSgemm(handle, CUBLAS_OP_T, CUBLAS_OP_N,
                        colsA, colsB, rowsB, &alpha, A, rowsA, B, rowsB, &beta, C, colsA)
            #endif
      ) != cudaSuccess) {

            cerr << "Cublas multiplication (AT*B) failed: " <<
ErrorHandler::cublasGetErrorString(status) << endl;
            return false;
      }

      return true;
}

/*----------------------------------*/
/* Matrix-Matrix Multiplication Set  */
/* _________________________________*/
/* Library: Cublas                  */
/*        { using cublas<T>GEMM }      */
/* Operation performed format       */
/*        ==>   C = A*B                */
```

```
/*-----------------------------------*/
bool Algorithms::Multiplication::A_times_B(cublasHandle_t handle,
      datatype* A, int rowsA, int colsA,
      datatype* B, int rowsB, int colsB,
      datatype* C) {

      cublasStatus_t status;
      const datatype alpha = 1.0;
      const datatype beta = 0.0;
      if ((
            #ifdef DOUBLE
                  status = cublasDgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N,
                        rowsA, colsB, rowsB, &alpha, A, rowsA, B, rowsB, &beta, C, rowsA)
            #else
                  status = cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N,
                        rowsA, colsB, rowsB, &alpha, A, rowsA, B, rowsB, &beta, C, rowsA)
            #endif
      ) != cudaSuccess) {

            cerr << "Cublas multiplication (A*B) failed: " <<
ErrorHandler::cublasGetErrorString(status) << endl;
            return false;
      }

      return true;
}

/*-----------------------------------*/
/* Matrix-Vector Multiplication Set  */
/* _________________________________*/
/* Operation performed format        */
/*        ==>   C -= A*x              */
/*-----------------------------------*/
bool Algorithms::Multiplication::A_times_x_SUBTRACT(
      datatype* A, int rowsA,
      datatype* x, int length,
      datatype* C,
      datatype* D, datatype* gamma_J, datatype* counter,
      unsigned int recommended) {

      device_A_times_x_modified << <1,recommended >> > (
            A, x, C, rowsA, length, D, gamma_J, counter);

      return true;
}

/*-----------------------------------*/
/* Matrix-Vector Multiplication Set  */
/* _________________________________*/
/* Operation performed format        */
/*        ==>   C = AT*x              */
/*-----------------------------------*/
bool Algorithms::Multiplication::AT_times_x(cublasHandle_t handle,
      datatype* A, int rowsA,
      datatype* x, int length,
      datatype* C) {

      cublasStatus_t status;
      const datatype alpha = 1.0;
      const datatype beta = 0.0;
      if ((
            #ifdef DOUBLE
                  status = cublasDgemv(handle, CUBLAS_OP_T,
                              rowsA, length, &alpha, A, rowsA, x, 1, &beta, C, 1)
            #else
```

```
                status = cublasSgemv(handle, CUBLAS_OP_T,
                           rowsA, length, &alpha, A, rowsA, x, 1, &beta, C, 1)
           #endif
      ) != cudaSuccess) {

           cerr << "Cublas multiplication (AT*x) failed: " <<
ErrorHandler::cublasGetErrorString(status) << endl;
           return false;
      }

      return true;
}

/*----------------------------------*/
/* Matrix-Vector Multiplication Set  */
/* __________________________________*/
/* Operation performed format        */
/*        ==>   C = x*A                */
/*----------------------------------*/
bool Algorithms::Multiplication::normal_x_times_A_modified(
      datatype* A, datatype* Gamma, datatype* columns, unsigned int colsD,
      datatype* gammaJ, datatype* rowlincomb, datatype* constant, datatype* norm, unsigned
int pJ,
      datatype* counter, unsigned int colsX, datatype* D, datatype* in, unsigned int N) {

      // block.x => No. of cols direction
      dim3 block(512);
      dim3 grid((colsX + N + block.x - 1) / block.x);
      device_normal_x_times_A_modified << <grid, block >> > (
           A, Gamma, columns, colsD, gammaJ, rowlincomb,
           constant, norm, pJ, counter, D, in, N, colsX);
      return true;
}


/*=================================================================*/
/*========================        KERNELS       ========================*/
/*=================================================================*/


/*--------------------------------*/
/* Operation: C = C - A*x         */
/*--------------------------------*/
__global__ void device_A_times_x_modified(
      datatype* A, datatype* x, datatype* out,
      unsigned int rowsA, unsigned int N, datatype* D,
      datatype* gammaJ, datatype* counters) {

      datatype sum;
      if (*((int*)counters) != 0) {
           if (threadIdx.x < rowsA) {
                sum = 0;
                for (int i = 0; i < N; i++) {
                     sum += A[i*rowsA + threadIdx.x] * x[i];
                }
                out[threadIdx.x] = out[threadIdx.x] - sum +
                     D[threadIdx.x] * (*gammaJ);
           }
      }
}

/*--------------------*/
/* Operation: C = x*A */
/*--------------------*/
__global__ void device_normal_x_times_A_modified(
```

```
	datatype* A, datatype* Gamma, datatype* columns, unsigned int colsD,
	datatype* gammaJ, datatype* rowlincomb, datatype* constant,
	datatype* norm, unsigned int pJ, datatype* counter,
	datatype* D, datatype* in, unsigned int N, unsigned int colsX) {

	unsigned int No_cols;
	if ((No_cols = *((int*)counter)) != 0) {
		unsigned int myCol = blockIdx.x*blockDim.x + threadIdx.x;
		if (myCol < No_cols) {
			datatype sum = 0;
			for (int i = 0; i < colsD; i++) {
				sum += Gamma[(int)columns[myCol] * colsD + i] * A[i];
			}
			Gamma[(int)columns[myCol] * colsD + pJ] = (rowlincomb[myCol] - sum +
(*constant)*gammaJ[myCol]) / (*norm);
		}
		else if (myCol >= colsX) {
			// Now update the Dictionary!
			// OPERATIONS:
			//         D(:,p(j)) = atom
			//
			myCol -= colsX;
			if (myCol < N) {
				D[myCol] = in[myCol] / (*norm);
			}
		}
	}
}

/*-------------------------------*/
/*   ALGORITHMIC TOOLS CLASS     */
/*      (REDUCTIONS etc.)        */
/*-------------------------------*/
#pragma once
#ifndef _ALGOS_
#define _ALGOS_


// Collection of all abstract algorithmic
// tools we'll need in the GPU algorithms
class Algorithms {

	public:
		// Family of all reduction operations
		// based on optimized warp reductions
		class Reduction {

			public:
				// Reduction in 1D layout (i.e. vector) of
				// __M__A__X__I__M__U__M__ value of its
				// elements as well as its index.
				__host__ static
				bool Maximum(datatype*, int, datatype*);

				// Reduction in 1D layout (i.e. vector)
				// of __M__A__X__I__M__U__M__ squared value.
				__host__ static
				bool Maximum_squared(datatype*, unsigned int, datatype*, datatype*,
unsigned int);

				// Reduction in 2D layout (i.e. matrix columns)
				// of __S__U__M__  of squared values of elements.
				// Result is then square rooted and inverted.
				__host__ static
```

```
                        bool reduce2D_Sum_SQUAREDelements(datatype*, int, int, datatype*,
unsigned int);

                        // Reduction in 1D layout (i.e. vector)
                        // of __S__U__M__ of squared values of elements
                        // (Mutliblock).
                        __host__ static
                        bool reduce1D_Batched_Sum_TOGETHER_collincomb(datatype*, datatype*,
datatype*, int, int, datatype*,

      datatype*, datatype*, unsigned int);

                        // Reduction in 2D layout (i.e. matrix rows)
                        // of dot products with given matrix
                        // (filtered by the specified cols)
                        __host__ static
                        bool reduce2D_dot_products_modified(datatype*, datatype*,
datatype*, datatype*, datatype*,
                                                                      unsigned int,
unsigned int, unsigned int);

                        // Euclidean norm of the vector.
                        __host__ static
                        bool euclidean_norm(datatype*, datatype*, unsigned int, unsigned
int);

                        // Reduction in 2D layout (i.e. matrix columns)
                        // of linear combination of input matrix and
                        // given columns (filtered by the specified rows)
                        __host__ static
                        bool reduce2D_rowlincomb_plus_nrm2_plus_mul(datatype*, datatype*,
datatype*, datatype*, unsigned int,
                                                                      unsigned int,
datatype*, unsigned int, datatype*, datatype*,
                                                                      datatype*,
unsigned int);

                        // Compute the Round Mean Square Error between a
                        // matrix X and its approximation matrix X~
                        __host__ static
                        bool reduce_RMSE(datatype*, datatype*, datatype*, unsigned int,
unsigned int, unsigned int,
                                                     unsigned int, datatype*);

                        // Compute the Round Mean Square Error between a
                        // matrix X and its approximation matrix X~
                        // ( modified for use only in Dictionary Update's
                        // special case )
                        __host__ static
                        bool reduce_ERROR_for_special_case(datatype*, datatype*, datatype*,
unsigned int, unsigned int,
                                                                      datatype*,
datatype*, datatype*, unsigned int);

                        // Euclidean norm of the vector
                        // ( used only in the special case)
                        __host__ static
                        bool SCase_EU_norm(datatype*, datatype*, unsigned int, unsigned
int, datatype*, datatype*, unsigned int);

                        // Set the size of the reduction buffer
                        // to be used in the calculations
                        static bool setReductionBuffer(datatype*);

                private:
```

```
                    Reduction(){}

          };

          // Family of all matrix-matrix and
          // vector-matrix multiplication variants
          class Multiplication {

               public:
                    // Matrix-Matrix Multiplication
                    // using cublas<T>GEMM
                    // Operation format ==> C = A'*B
                    static bool AT_times_B(cublasHandle_t, datatype*, int, int,
datatype*, int, int, datatype*);

                    // Matrix-Matrix Multiplication
                    // using cublas<T>GEMM
                    // Operation format ==> C = A*B
                    static bool A_times_B(cublasHandle_t, datatype*, int, int,
datatype*, int, int, datatype*);

                    // Matrix-Vector Multiplication
                    // Operation format ==> C -= A*x
                    static bool A_times_x_SUBTRACT(datatype*, int, datatype*, int,
datatype*, datatype*, datatype*,
                                                            datatype*, unsigned
int);

                    // Matrix-Vector Multiplication
                    // Operation format ==> C = AT*x
                    static bool AT_times_x(cublasHandle_t, datatype*, int, datatype*,
int, datatype*);

                    // Matrix-Vector Multiplication
                    // Operation format ==> C = x*A
                    static bool normal_x_times_A_modified(datatype*, datatype*,
datatype*, unsigned int, datatype*,
                                                                      datatype*,
datatype*, datatype*, unsigned int, datatype*,
                                                                      unsigned int,
datatype*, datatype*, unsigned int);

               private:
                    Multiplication() {}

          };

          // Family of all random permutation
          // generators' algorithmic utilities
          class RandomPermutation {

               public:
                    // Generate a random permutation
                    // of indices in the specified range.
                    static unsigned int* generateRandPerm(unsigned int);

                    // Set the size of the generator buffer
                    // to be used in the calculations
                    static bool setGeneratorBuffer(unsigned int*, int);

               private:
                    RandomPermutation() {}

          };
```

```
	private:
		Algorithms(){}

};


#endif // !_ALGOS_

/*---------------------------*/
/* REDUCTION KERNEL FUNCTIONS */
/*      IMPLEMENTATION       */
/*---------------------------*/
#include "GlobalDeclarations.cuh"
#include "ReductionBase.cuh"
#include "ReductionKernels.cuh"
#include "Algorithms.cuh"
#include <cstdio>

using namespace std;


/*********************************
*         DECLARATIONS          *
*           ____________          *
*********************************/
__device__
datatype* experimental_array;


/*************************************
*    BATCHED REDUCTION WRAPPERS     *
*     __________________________      *
*************************************/

/*--------------------------------*/
/*  2-STAGE MULTIBLOCK REDUCTION  */
/*          *STAGE 1*             */
/*--------------------------------*/
__global__ void multi_reduction_squared_sum_STAGE1(
	datatype* in, datatype* out, datatype* counter, int stride) {

	// blockIdx.y = row = reduction id
	// blockIdx.x = reduction axis
	int N = *((int*)(counter + blockIdx.y));
	datatype sum = 0;
	int index = blockIdx.x * blockDim.x + threadIdx.x;
	if (index < N) {
		// Retrieve value filtered through function
		sum = sqval(in[blockIdx.y*stride + index]);
	}
	sum = blockReduceSum(sum);
	if (threadIdx.x == 0) {
		// Write to output
		if (gridDim.x > 1) {
			experimental_array[blockIdx.y * 1024 + blockIdx.x] = sum;
		}
		else {
			*(out + blockIdx.y) = sum;
		}
	}
}

/*--------------------------------*/
/*  2-STAGE MULTIBLOCK REDUCTION  */
/*           *STAGE 2*            */
```

```
/* ALSO COLLINCOMB FUSED IN KERNEL*/
/* TO INCREASE CONCURRENCY.       */
/*--------------------------------*/
__global__ void multi_reduction_squared_sum_STAGE2_together_collincomb(
      datatype* in, datatype* out, datatype* counter, int stride,
      datatype *A, datatype* columns, datatype* out2,
      unsigned int rowsX) {

      if (blockIdx.y == 0) {
            /**/
            /*             *STAGE 2*            */
            /**/
            int N = (*((int*)(counter + blockIdx.x)) + 1024 - 1) / 1024;
            datatype sum = 0;
            if (threadIdx.x < N) {
                  sum = experimental_array[blockIdx.x * 1024 + threadIdx.x];
            }
            sum = blockReduceSum(sum);
            if (threadIdx.x == 0) {
                  // Write to output
                  *(out + blockIdx.x) = sum;
            }
      }
      else {
            /**/
            /*           *COLLINCOMB*            */
            /**/
            if (threadIdx.x < rowsX) {
                  // Multi-Reduction sub-id: blockIdx.x
                  columns += blockIdx.x * stride;
                  in += blockIdx.x * stride;
                  out2 += blockIdx.x * rowsX;
                  unsigned int N = *((int*)(counter + blockIdx.x));
                  datatype sum = 0;
                  for (int i = 0; i < N; i++) {
                        sum += A[((int)columns[i])*rowsX + threadIdx.x] * in[i];
                  }
                  *(out2 + threadIdx.x) = sum;
            }
      }
}

/*------------------------------------*/
/* 2D + 2-STAGE REDUCTION FOR MATRIX- */
/* VECTOR PRODUCT FILTERED BY COLUMNS */
/*             *STAGE 1*              */
/*------------------------------------*/
__global__ void multi_reduction_smallGamma_times_gammaJ_STAGE1(
      datatype *Gamma, datatype* columns, datatype* gammaJ, datatype* out,
      datatype* counters, unsigned int pJ) {

      // blockIdx.x - reduction axis
      // blockIdx.y - reduction ID axis
      int N = *((int*)counters);
      if (N != 0) {
            if (pJ == blockIdx.y) {
                  if (blockIdx.x == 0 && threadIdx.x == 0) {
                        *(out + pJ) = *(out + pJ - gridDim.y);
                  }
                  return;
            }
            datatype sum = 0;
            int index = blockIdx.x * blockDim.x + threadIdx.x;
            if (index < N) {
                  // Retrieve value filtered through columns
```

```
                sum = Gamma[((int)columns[index])*gridDim.y + blockIdx.y] *
gammaJ[index];
            }
            sum = blockReduceSum(sum);
            if (threadIdx.x == 0) {
                // Write to output
                N = (N + 1024 - 1) / 1024;
                if (N > 1) {
                    experimental_array[blockIdx.y * 1024 + blockIdx.x] = sum;
                }
                else {
                    if (blockIdx.x == 0) {
                        *(out + blockIdx.y) = sum;
                    }
                }
            }
        }
}

/*------------------------------------*/
/* 2D + 2-STAGE REDUCTION FOR MATRIX- */
/* VECTOR PRODUCT FILTERED BY COLUMNS */
/*              *STAGE 2*             */
/*------------------------------------*/
__global__ void multi_reduction_smallGamma_times_gammaJ_STAGE2(
    datatype* out, datatype* counters, unsigned int pJ) {

    int N = *((int*)counters);
    if (N != 0) {
        if (pJ == blockIdx.x) {
            return;
        }
        N = (N + 1024 - 1) / 1024;
        if (N > 1) {
            datatype sum = 0;
            if (threadIdx.x < N) {
                // Retrieve partial sum
                sum = experimental_array[blockIdx.x * 1024 + threadIdx.x];
            }
            sum = blockReduceSum(sum);
            if (threadIdx.x == 0) {
                // Write to output
                *(out + blockIdx.x) = sum;
            }
        }
    }
}


/***********************************
*    REDUCTION STAGES FUNCTIONS     *
*     ___________________________    *
***********************************/

//////////////////////////////////////
// Reduction across multiple blocks //
//////////////////////////////////////
__global__ void deviceReduceKernel_2D_square_values(datatype *in, int N, datatype* out) {
    datatype sum = 0;
    if (threadIdx.x < N) {
        // Retrieve value filtered through function
        sum = sqval(in[blockIdx.x*N + threadIdx.x]);
    }
    sum = blockReduceSum(sum);
    if (threadIdx.x == 0) {
```

```
			// Write to output
			out[blockIdx.x] = invSQRT(sum);
		}
}

//////////////////////////////////////
// - 1 STAGE MULTIBLOCK REDUCTION - //
// -  LINEAR COMBINATION OF ROWS  - //
//////////////////////////////////////
__global__ void rowlincomb(
	datatype *A, datatype* x, datatype* out, datatype* cols,
	unsigned int rowsX, datatype* counters, datatype* out2,
	datatype* D, datatype* out3, unsigned int colsX) {

	unsigned int N;
	if ( (N = *((int*)counters)) != 0) {
		if (blockIdx.x < N) {
			datatype sum = 0;
			if (threadIdx.x < rowsX) {
				// Retrieve value filtered through columns and rows
				sum = A[(int)cols[blockIdx.x] * rowsX + threadIdx.x] *
x[threadIdx.x];
			}
			sum = blockReduceSum(sum);
			if (threadIdx.x == 0) {
				// Write to output
				out[blockIdx.x] = sum;
			}
		}
		else if (blockIdx.x >= colsX) {
			if (blockIdx.x == gridDim.x - 1) {
				// Calculate the norm of the new atom
				// by combining the previous results.
				//
				datatype sum = 0;
				if (threadIdx.x < rowsX) {
					// Retrieve value filtered through sqval
					sum = sqval(x[threadIdx.x]);
				}
				sum = blockReduceSum(sum);
				if (threadIdx.x == 0) {
					// Write to output
					*out2 = sqrt(sum);
				}
			}
			else {
				// We now calculate the following
				// vector-matrix multiplication in
				// parallel:
				//             (atom'*D)
				//
				datatype sum = 0;
				if (threadIdx.x < rowsX) {
					// Retrieve column
					sum = D[(blockIdx.x - colsX)*rowsX + threadIdx.x] *
x[threadIdx.x];
				}
				sum = blockReduceSum(sum);
				if (threadIdx.x == 0) {
					// Write to output
					out3[blockIdx.x - colsX] = sum;
				}
			}
		}
	}
```

```
}

//////////////////////////////////////
// - 3 STAGE MULTIBLOCK REDUCTION - //
// -       RMSE CALCULATION       - //
// -         *STAGE 1*            - //
//////////////////////////////////////
__global__ void RMSE_stage1(datatype* X, datatype* Xappr, unsigned int N, datatype* out)
{

      datatype sum = 0;
      if (threadIdx.x < N) {
            // Retrieve value and compute difference
            sum = sqval(X[blockIdx.x*N + threadIdx.x] - Xappr[blockIdx.x*N +
threadIdx.x]);
      }
      sum = blockReduceSum(sum);
      if (threadIdx.x == 0) {
            // Write to output
            out[blockIdx.x] = sum;
      }

}

//////////////////////////////////////
// - 3 STAGE MULTIBLOCK REDUCTION - //
// -       RMSE CALCULATION       - //
// -          *STAGE 2*           - //
//////////////////////////////////////
__global__ void RMSE_stage2(
      datatype* in, unsigned int N, unsigned int size, unsigned int iter,
      datatype* bitmap) {

      datatype sum = 0;
      int index = blockIdx.x * blockDim.x + threadIdx.x;
      if (index < N) {
            // Retrieve value
            sum = in[index];
            in[index] = sum * bitmap[index];
      }
      sum = blockReduceSum(sum);
      if (threadIdx.x == 0) {
             //Write to output
            if (gridDim.x > 1) {
                  experimental_array[blockIdx.x] = sum;
            }
            else {
                  printf("Iteration %02d / %d complete, RMSE = %.5f\n", iter,
NUMBERofITERATIONS, sqrt(sum / size));
            }
      }
}

//////////////////////////////////////
// - 3 STAGE MULTIBLOCK REDUCTION - //
// -       RMSE CALCULATION       - //
// -          *STAGE 3*           - //
//////////////////////////////////////
__global__ void RMSE_stage3(unsigned int N, unsigned int size, unsigned int iter) {

      datatype sum = 0;
      int index = blockIdx.x * blockDim.x + threadIdx.x;
      if (index < N) {
            // Retrieve value
            sum = experimental_array[index];
```

```
	}
	sum = blockReduceSum(sum);
	if (threadIdx.x == 0) {
		//Write to output
		if (gridDim.x > 1) {
			experimental_array[blockIdx.x] = sum;
		}
		else {
			printf("Iteration %02d / %d complete, RMSE = %.5f\n", iter,
NUMBERofITERATIONS, sqrt(sum / size));
		}
	}
}

///////////////////////////////////////////
// -  MULTI-STAGE REDUCTION TO BUFFER  -//
// - BLOCK-WIDE MAXIMUM VALUE & INDEX  -//
// -           *STAGE 1*             - //
///////////////////////////////////////////
__global__ void device_max_err_STAGE1(datatype* in, int N, datatype* out) {

	if (!(*out)) {
		// Our atom does not need replacement
		return;
	}
	DoubleReductionType myElement;
	myElement.value = 0;
	int index = blockIdx.x*blockDim.x + threadIdx.x;
	if (index < N) {
		// Retrieve value
		myElement.value = in[index];
		// Store index
		myElement.index = index;
	}
	myElement = blockReduceMaximumIndex(myElement);
	if (threadIdx.x == 0) {
		// Write to output
		if (gridDim.x > 1) {
			((DoubleReductionType*)experimental_array)[blockIdx.x] = myElement;
		}
		else {
			*(out + 1) = myElement.index;
			in[myElement.index] = 0;
		}
	}
}

///////////////////////////////////////////
// -  MULTI-STAGE REDUCTION TO BUFFER  -//
// - BLOCK-WIDE MAXIMUM VALUE & INDEX  -//
// -           *STAGE 2*             - //
///////////////////////////////////////////
__global__ void device_max_err_STAGE2(datatype* in, int N, datatype* out) {

	if (!(*out)) {
		// Our atom does not need replacement
		return;
	}
	DoubleReductionType myElement;
	myElement.value = 0;
	int index = blockIdx.x*blockDim.x + threadIdx.x;
	if (index < N) {
		// Retrieve value
		myElement = ((DoubleReductionType*)experimental_array)[index];
	}
```

```
	myElement = blockReduceMaximumIndex(myElement);
	if (threadIdx.x == 0) {
		// Write to output
		if (gridDim.x > 1) {
			((DoubleReductionType*)experimental_array)[blockIdx.x] = myElement;
		}
		else {
			*(out + 1) = myElement.index;
			in[myElement.index] = 0;
		}
	}
}

////////////////////////////////////////////
// -  SINGLE-STAGE IN-PLACE REDUCTION   -//
// -  COLUMN-WISE MAXIMUM SQUARE VALUE  -//
// -           SINGLE WARP              -//
////////////////////////////////////////////
__global__ void device_max(
	datatype* G, unsigned int colsD, datatype* usecount, datatype* replaced,
	unsigned int j) {

	datatype max = 0;
	if (threadIdx.x < colsD && threadIdx.x != j) {
		// Retrieve value
		max = sqval( G[threadIdx.x] );
	}
	max = warpReduceMax(max);
	if (threadIdx.x == 0) {
		// Write to output
		*G = ( ( max > SQR_muTHRESH ) || ( *((unsigned int*)usecount) < USE_THRESH ) )
				&& (*replaced == 0);
	}
}

///////////////////////////////////////////
// - 1 STAGE EUCLEDIAN NORM REDUCTION - //
///////////////////////////////////////////
__global__ void EUnorm(datatype* in, datatype* out, unsigned int N) {

	if (!(*out)) {
		// Our atom does not need replacement
		return;
	}
	in += (unsigned int)(*(out + 1))*N;
	datatype sum = 0;
	if (threadIdx.x < N) {
		// Retrieve value filtered through sqval
		sum = sqval(in[threadIdx.x]);
	}
	sum = blockReduceSum(sum);
	if (threadIdx.x == 0) {
		// Write to output
		*(out + 2) = sqrt(sum);
	}
}

/////////////////////////////////////////
// - 1 STAGE SPECIAL CASE REDUCTION - //
// -       ERROR CALCULATION       - //
/////////////////////////////////////////
__global__ void SCase_err_stage1(
	datatype* X, datatype* Xappr, unsigned int N, unsigned int colsX,
	datatype* out, datatype* counters, datatype* unused,
	datatype* UScounter) {
```

```
	if (*((int*)counters) == 0) {
		if (blockIdx.x < (colsX - *UScounter)) {
			unsigned int column = unused[myGenerator(blockIdx.x, blockDim.x)];
			datatype sum = 0;
			if (threadIdx.x < N) {
				// Retrieve value and compute difference
				sum = sqval(X[column*N + threadIdx.x] - Xappr[column*N + 
threadIdx.x]);
			}
			sum = blockReduceSum(sum);
			if (threadIdx.x == 0) {
				// Write to output
				out[blockIdx.x] = sum;
			}
		}
	}
}

////////////////////////////////////////
// - 2 STAGE SPECIAL CASE REDUCTION - //
// -      MAXIMUM CALCULATION       - //
////////////////////////////////////////
__global__ void SCase_err_stage2(
	datatype* in, int N, datatype* out, unsigned int colsX,
	datatype* counters, datatype* UScounter) {

	if (*((int*)counters) == 0) {

		DoubleReductionType myElement;
		myElement.value = 0;
		int index = blockIdx.x*blockDim.x + threadIdx.x;
		if (index < N && index < (colsX - *UScounter) ) {
			// Retrieve value
			myElement.value = in[index];
			// Store index
			myElement.index = index;
		}
		myElement = blockReduceMaximumIndex(myElement);
		if (threadIdx.x == 0) {
			// Write to output
			if (gridDim.x > 1) {
				((DoubleReductionType*)experimental_array)[blockIdx.x] = myElement;
			}
			else {
				*out = myElement.index;
			}
		}
	}
}

////////////////////////////////////////
// - 2 STAGE SPECIAL CASE REDUCTION - //
// -      MAXIMUM CALCULATION       - //
////////////////////////////////////////
__global__ void SCase_err_stage3(
	int N, datatype* out, datatype* counters) {

	if (*((int*)counters) == 0) {

		DoubleReductionType myElement;
		myElement.value = 0;
		int index = blockIdx.x*blockDim.x + threadIdx.x;
		if (index < N) {
			// Retrieve value
```

```
                myElement = ((DoubleReductionType*)experimental_array)[index];
            }
            myElement = blockReduceMaximumIndex(myElement);
            if (threadIdx.x == 0) {
                // Write to output
                if (gridDim.x > 1) {
                    ((DoubleReductionType*)experimental_array)[blockIdx.x] = myElement;
                }
                else {
                    *out = myElement.index;
                }
            }
        }
}

//////////////////////////////////////////
// - 1 STAGE EUCLEDIAN NORM REDUCTION - //
//////////////////////////////////////////
__global__ void SCase_norm(
        datatype* in, datatype* out, unsigned int N,
        datatype* counters, datatype* unused, unsigned int dim) {

        if(*((int*)counters) == 0) {
            in += (unsigned int)unused[myGenerator((unsigned int)(*out), dim)] * N;
            datatype sum = 0;
            if (threadIdx.x < N) {
                // Retrieve value filtered through sqval
                sum = sqval(in[threadIdx.x]);
            }
            sum = blockReduceSum(sum);
            if (threadIdx.x == 0) {
                // Write to output
                *(out + 1) = sqrt(sum);
            }
        }
}

/*-------------------------*/
/*   UTILITIES NAMESPACE   */
/*-------------------------*/
#pragma once
#ifndef _UTIL_
#define _UTIL_


// Functions used for various
// secondary tasks
namespace Utilities {

        // Public Methods
        void print(datatype*, unsigned int, unsigned int, unsigned int, unsigned int);
        void printTransposed(datatype*,unsigned int,unsigned int,unsigned int,unsigned int);
        void printDevice(datatype*, unsigned int, unsigned int, unsigned int, unsigned int);
        void printDeviceTransposed(datatype*, unsigned int, unsigned int, unsigned int,
unsigned int);

};

#endif // !_UTIL_

/*------------------------------*/
/*     UTILITIES  NAMESPACE     */
/*       IMPLEMENTATION         */
/*------------------------------*/
```

```
#include <iostream>
#include <iomanip>
#include "GlobalDeclarations.cuh"
#include "Utilities.cuh"

using namespace std;


// Declarations - functions
__global__ void kernelPrint(datatype*, unsigned int, unsigned int, unsigned int, unsigned
int);
__global__ void kernelPrintTransposed(datatype*, unsigned int, unsigned int, unsigned
int, unsigned int);

/*----------------*/
/* Print an array */
/*----------------*/
void Utilities::print(datatype* array, unsigned int rows, unsigned int cols,
                                  unsigned int limit_rows, unsigned int limit_cols) {

      cout << "______________________________________________________" << endl;
      for (size_t i = 0; i < rows && i < limit_rows; i++)
      {
            cout << "# " << setfill('0') << setw(3) << i + 1 << " #   ";
            for (size_t j = 0; j < cols && j < limit_cols; j++)
            {
                  cout << setprecision(10) << std::fixed << array[i*cols + j] << "\t";
            }
            cout << endl;
      }
      cout << "______________________________________________________" << endl;
}

/*-------------------------------*/
/* Print an array (transposed)   */
/*-------------------------------*/
void Utilities::printTransposed(datatype* array, unsigned int rows, unsigned int cols,
                                             unsigned int limit_rows, unsigned int
limit_cols) {

      cout << "______________________________________________________" << endl;
      for (size_t i = 0; i < rows && i < limit_rows; i++)
      {
            cout << "# " << setfill('0') << setw(3) << i + 1 << " #   ";
            for (size_t j = 0; j < cols && j < limit_cols; j++)
            {
                  cout << setprecision(10) << std::fixed << array[i + j*rows] << "\t";
            }
            cout << endl;
      }
      cout << "______________________________________________________" << endl;
}

/*----------------------*/
/* Print a device array */
/*----------------------*/
void Utilities::printDevice(datatype* array, unsigned int rows, unsigned int cols,
                                        unsigned int limit_rows, unsigned int limit_cols) {

      SINGLE_THREAD(kernelPrint, (array, rows, cols, limit_rows, limit_cols) );
}

/*-----------------------------------*/
/* Print a device array (transposed) */
/*-----------------------------------*/
```

```
void Utilities::printDeviceTransposed(datatype* array, unsigned int rows, unsigned int
cols,
                                                    unsigned int limit_rows, unsigned
int limit_cols) {

      SINGLE_THREAD(kernelPrintTransposed,(array, rows, cols, limit_rows, limit_cols));
}

/*-------------------------*/
/* Print a device array in */
/* a kernel on the device  */
/*-------------------------*/
__global__
void kernelPrint(datatype* array, unsigned int rows, unsigned int cols,
                      unsigned int limit_rows, unsigned int limit_cols) {

      printf("_____________________________________________________\n");
      for (size_t i = 0; i < rows && i < limit_rows; i++)
      {
            printf("#  %03d #   ",i + 1);
            for (size_t j = 0; j < cols && j < limit_cols; j++)
            {
                  printf("%.10f\t",array[i*cols + j]);
            }
            printf("\n");
      }
      printf("_____________________________________________________\n");
}

/*-----------------------------------*/
/* Print a device array (transposed) */
/* in a kernel on the device         */
/*-----------------------------------*/
__global__
void kernelPrintTransposed(datatype* array, unsigned int rows, unsigned int cols,
                                       unsigned int limit_rows, unsigned int limit_cols) {

      printf("_____________________________________________________\n");
      for (size_t i = 0; i < rows && i < limit_rows; i++)
      {
            printf("#  %03d #   ", i + 1);
            for (size_t j = 0; j < cols && j < limit_cols; j++)
            {
                  printf("%.10f\t", array[i + j*rows]);
            }
            printf("\n");
      }
      printf("_____________________________________________________\n");
}
/*--------------------------*/
/* TIMING OPERATIONS CLASS  */
/*--------------------------*/
#pragma once
#ifndef _TIMER_
#define _TIMER_


// Functions used for calculating
// elapsed time on CPU or GPU
class Timer {

      private:
            // Private Members
            cudaEvent_t GPUstart, GPUstop;
            LARGE_INTEGER CPUstart, CPUstop, frequency;
```

```
	public:
		// Constructor
		Timer();
		~Timer();

		// Public Methods
		void	tic();
		double	toc();
		double	tocmilli();
		void	GPUtic();
		float	GPUtoc();
		bool	isValid();

};

#endif // !_TIMER_

/*------------------------------*/
/*      TIMING OPERATIONS       */
/*       IMPLEMENTATION         */
/*------------------------------*/
#include <iostream>
#include <windows.h>
#include "GlobalDeclarations.cuh"
#include "Timer.cuh"

using namespace std;


/*---------------*/
/* Constructor   */
/*---------------*/
Timer::Timer() {
	this->GPUstart = NULL;
	this->GPUstop = NULL;
	cudaError_t cuda_error;
	if ( (cuda_error = cudaEventCreate(&this->GPUstart)) != cudaSuccess) {
		cerr << "Cuda event create for GPUstart failed: " << 
cudaGetErrorString(cuda_error) << endl;
	}
	if ( (cuda_error = cudaEventCreate(&this->GPUstop)) != cudaSuccess) {
		cerr << "Cuda event create for GPUstop failed: " << 
cudaGetErrorString(cuda_error) << endl;
	}
}

/*--------------*/
/* Destructor   */
/*--------------*/
Timer::~Timer() {
	if (this->GPUstart) {
		cudaEventDestroy(this->GPUstart);
	}
	if (this->GPUstop) {
		cudaEventDestroy(this->GPUstop);
	}
}

/*--------------------------*/
/* Check if timer is valid   */
/*--------------------------*/
bool Timer::isValid() {
	return (this->GPUstart != NULL && this->GPUstop != NULL);
```

```
}

/*-------------------*/
/* Start CPU timer   */
/*-------------------*/
void Timer::tic() {
	QueryPerformanceFrequency(&this->frequency);
	QueryPerformanceCounter(&this->CPUstart);
}

/*--------------------*/
/* Stop CPU timer and */
/* return in sec!     */
/*--------------------*/
double Timer::toc() {
	QueryPerformanceCounter(&this->CPUstop);
	return ((double)(this->CPUstop.QuadPart - this->CPUstart.QuadPart) / this-
>frequency.QuadPart);
}

/*--------------------*/
/* Stop CPU timer and */
/* return in ms!      */
/*--------------------*/
double Timer::tocmilli() {
	QueryPerformanceCounter(&this->CPUstop);
	return ((double)(this->CPUstop.QuadPart - this->CPUstart.QuadPart) / this-
>frequency.QuadPart * 1000.0);
}

/*-------------------*/
/* Start GPU timer   */
/*-------------------*/
void Timer::GPUtic() {
	cudaEventRecord(this->GPUstart);
}

/*--------------------*/
/* Stop GPU timer and */
/* return in ms!      */
/*--------------------*/
float Timer::GPUtoc() {
	cudaEventRecord(this->GPUstop);
	cudaEventSynchronize(this->GPUstop);
	float milliseconds = 0;
	cudaEventElapsedTime(&milliseconds, this->GPUstart, this->GPUstop);
	return milliseconds;

}

/*--------------------------------*/
/* FILE OPERATIONS RELATED CLASS  */
/*--------------------------------*/
#pragma once
#ifndef _FILEMNGR_
#define _FILEMNGR_


// MatLab Libraries for MAT-files and
// Mx matrices
#include INCLUDE_FILE(MATLAB_include,mat.h)
#include INCLUDE_FILE(MATLAB_include,matrix.h)

// Functions used for reading from/writing to
// specific files
```

```
class FileManager {

	private:
		// Private Members
		mxArray *aX, *aD;

	public:
		// Constructor
		FileManager();
		// Destructor
		~FileManager();

		// Public Methods
		bool  readXarray(datatype**);
		bool  readDarray(datatype**);
		bool  writeDarray(datatype*);
		bool  writeTemporary(datatype*,unsigned int,unsigned int,char*);

};

#endif // !_FILEMNGR_

/*------------------------------*/
/*	FILE OPERATIONS		*/
/*	IMPLEMENTATION		*/
/*------------------------------*/
#include <iostream>
#include "GlobalDeclarations.cuh"
#include "Globals.cuh"
#include "FileManager.cuh"

using namespace std;


// Declarations
bool readMat(datatype**,mxArray**,unsigned int*,unsigned int*,char*,char*);
bool writeMat(datatype*,unsigned int,unsigned int, char*, char*);
unsigned int next_pow_2(unsigned int);

// Macros
#define MIN2(a,b) (a < b ? a : b)

/*---------------*/
/* Constructor   */
/*---------------*/
FileManager::FileManager() {
	this->aX = NULL;
	this->aD = NULL;
}

/*--------------*/
/* Destructor   */
/*--------------*/
FileManager::~FileManager() {
	if (this->aX) {
		mxDestroyArray(this->aX);
	}
	if (this->aD) {
		mxDestroyArray(this->aD);
	}
}

/*-------------------------------*/
/* Read X array stored in file   */
```

```
/*-------------------------------*/
bool FileManager::readXarray(datatype** array) {
	if (readMat(
		array, &(this->aX),
		&Globals::rowsX, &Globals::colsX,
		Xarray_in_PATH, "X") == false) {

		cerr << "An error occured while reading file!..." << endl;
		return false;
	}
	Globals::TPB_rowsX = MIN2(next_pow_2(Globals::rowsX), 1024);
	return true;
}

/*-------------------------------*/
/* Read D array stored in file   */
/*-------------------------------*/
bool FileManager::readDarray(datatype** array) {
	if (readMat(
		array, &(this->aD),
		&Globals::rowsD, &Globals::colsD,
		Darray_in_PATH, "D") == false) {

		cerr << "An error occured while reading file!..." << endl;
		return false;
	}
	Globals::TPB_colsD = MIN2(next_pow_2(Globals::colsD), 32);
	return true;
}

/*-------------------------*/
/* Write D array to file   */
/*-------------------------*/
bool FileManager::writeDarray(datatype* array) {
	if ( writeMat(array, Globals::rowsD, Globals::colsD, Darray_out_PATH,"FinalD") ==
false) {
		cerr << "An error occured while writing file!..." << endl;
		return false;
	}
	return true;
}

/*-----------------------------*/
/* Write given array to file   */
/*-----------------------------*/
bool FileManager::writeTemporary(datatype* array, unsigned int rows, unsigned int
cols,char* array_name) {
	if (writeMat(array, rows, cols, temp_out_PATH, array_name) == false) {
		cerr << "An error occured while writing file!..." << endl;
		return false;
	}
	return true;
}

/*---------------------------*/
/* Read MATLAB file format   */
/*---------------------------*/
bool readMat(datatype** array_dbl,mxArray **mArray,
		unsigned int* rows_ptr,unsigned int* cols_ptr,
		char* file,char* array_name) {

	MATFile *pmat;
	pmat = matOpen(file, "r");
	if (pmat == NULL) {
		cerr << "Error opening file " << file << " ..." << endl;
```

```
		return(false);
	}
	*mArray = matGetVariable(pmat, array_name);
	if (*mArray == NULL) {
		cerr << "Error reading existing matrix X..." << endl;
		matClose(pmat);
		return(false);
	}
	#ifdef DOUBLE
		*array_dbl = mxGetPr(*mArray);
	#else
		*array_dbl = (datatype*)mxGetData(*mArray);
	#endif
	*rows_ptr = (unsigned int) mxGetM(*mArray);
	*cols_ptr = (unsigned int) mxGetN(*mArray);
	if (matClose(pmat) != 0) {
		cerr << "Error closing file " << file << "..." << endl;
		return(false);
	}
	return true;
}

/*---------------------------*/
/* Write MATLAB file format   */
/*---------------------------*/
bool writeMat(datatype* array,unsigned int rows,unsigned int cols, char* file, char*
array_name) {
	MATFile *pmat;
	pmat = matOpen(file, "w");
	if (pmat == NULL) {
		cerr << "Error opening file " << file << " ..." << endl;
		return(false);
	}

	mxArray *pa2 = mxCreateDoubleMatrix(rows, cols, mxREAL);
	if (pa2 == NULL) {
		cerr << "Error: Out of memory while writing existing matrix X..." << endl;
		matClose(pmat);
		return(false);
	}
	memcpy((void *)(mxGetPr(pa2)), (void *)array, rows * cols * sizeof(datatype));

	int status = matPutVariableAsGlobal(pmat, array_name, pa2);
	if (status != 0) {
		cerr << "Error using matPutVariableAsGlobal..." << endl;
		matClose(pmat);
		return(false);
	}
	/* clean up */
	mxDestroyArray(pa2);
	if (matClose(pmat) != 0) {
		cerr << "Error closing file " << file << "..." << endl;
		return(false);
	}
	return true;
}

/*------------------------------*/
/* Helper function to find the  */
/* next power of 2 for the given */
/* number.                      */
/*------------------------------*/
unsigned int next_pow_2(unsigned int v) {
	v--;
	v |= v >> 1;
```

```
	v |= v >> 2;
	v |= v >> 4;
	v |= v >> 8;
	v |= v >> 16;
	v++;
	return v;

}

/*------------------------------*/
/*    ERROR HANDLING NAMESPACE    */
/*------------------------------*/
#pragma once
#ifndef _ERROR_
#define _ERROR_


// Functions used for error handling
namespace ErrorHandler {

	// Public Methods
	const char* cublasGetErrorString(cublasStatus_t);

};

#endif // !_ERROR_

/*------------------------------*/
/*    ERROR HANDLING NAMESPACE    */
/*        IMPLEMENTATION          */
/*------------------------------*/
#include "GlobalDeclarations.cuh"
#include "curand.h"
#include "cusolverDn.h"
#include "ErrorHandler.cuh"


/*-------------------------------*/
/* Error string from cublas call */
/*-------------------------------*/
const char* ErrorHandler::cublasGetErrorString(cublasStatus_t status) {
	switch (status) {
		case CUBLAS_STATUS_SUCCESS: return "CUBLAS_STATUS_SUCCESS";
		case CUBLAS_STATUS_NOT_INITIALIZED: return "CUBLAS_STATUS_NOT_INITIALIZED";
		case CUBLAS_STATUS_ALLOC_FAILED: return "CUBLAS_STATUS_ALLOC_FAILED";
		case CUBLAS_STATUS_INVALID_VALUE: return "CUBLAS_STATUS_INVALID_VALUE";
		case CUBLAS_STATUS_ARCH_MISMATCH: return "CUBLAS_STATUS_ARCH_MISMATCH";
		case CUBLAS_STATUS_MAPPING_ERROR: return "CUBLAS_STATUS_MAPPING_ERROR";
		case CUBLAS_STATUS_EXECUTION_FAILED: return "CUBLAS_STATUS_EXECUTION_FAILED";
		case CUBLAS_STATUS_INTERNAL_ERROR: return "CUBLAS_STATUS_INTERNAL_ERROR";
	}
	return "UNKNOWN ERROR";
}

/*---------------------------------*/
/* Error string from cusolver call */
/*---------------------------------*/
const char* cusolverGetErrorString(cusolverStatus_t status) {
	switch (status) {
		case CUSOLVER_STATUS_SUCCESS: return "CUSOLVER_STATUS_SUCCESS";
		case CUSOLVER_STATUS_NOT_INITIALIZED: return
"CUSOLVER_STATUS_NOT_INITIALIZED";
		case CUSOLVER_STATUS_ALLOC_FAILED: return "CUSOLVER_STATUS_ALLOC_FAILED";
		case CUSOLVER_STATUS_INVALID_VALUE: return "CUSOLVER_STATUS_INVALID_VALUE";
		case CUSOLVER_STATUS_ARCH_MISMATCH: return "CUSOLVER_STATUS_ARCH_MISMATCH";
```

```
		case CUSOLVER_STATUS_EXECUTION_FAILED: return "CUSOLVER_STATUS_execution failed";
		case CUSOLVER_STATUS_MATRIX_TYPE_NOT_SUPPORTED: return "CUSOLVER_STATUS_Matrix not supported";
		case CUSOLVER_STATUS_INTERNAL_ERROR: return "CUSOLVER_STATUS_INTERNAL_ERROR";
	}
	return "UNKNOWN ERROR";
}

/*------------------------------*/
/* Error string from curand call */
/*------------------------------*/
const char* curandGetErrorString(curandStatus_t error) {
	switch (error) {
		case CURAND_STATUS_SUCCESS: return "CURAND_STATUS_SUCCESS";
		case CURAND_STATUS_VERSION_MISMATCH: return "CURAND_STATUS_VERSION_MISMATCH";
		case CURAND_STATUS_NOT_INITIALIZED: return "CURAND_STATUS_NOT_INITIALIZED";
		case CURAND_STATUS_ALLOCATION_FAILED: return "CURAND_STATUS_ALLOCATION_FAILED";
		case CURAND_STATUS_TYPE_ERROR: return "CURAND_STATUS_TYPE_ERROR";
		case CURAND_STATUS_OUT_OF_RANGE: return "CURAND_STATUS_OUT_OF_RANGE";
		case CURAND_STATUS_LENGTH_NOT_MULTIPLE: return "CURAND_STATUS_LENGTH_NOT_MULTIPLE";
		case CURAND_STATUS_DOUBLE_PRECISION_REQUIRED: return "CURAND_STATUS_DOUBLE_PRECISION_REQUIRED";
		case CURAND_STATUS_LAUNCH_FAILURE: return "CURAND_STATUS_LAUNCH_FAILURE";
		case CURAND_STATUS_PREEXISTING_FAILURE: return "CURAND_STATUS_PREEXISTING_FAILURE";
		case CURAND_STATUS_INITIALIZATION_FAILED: return "CURAND_STATUS_INITIALIZATION_FAILED";
		case CURAND_STATUS_ARCH_MISMATCH: return "CURAND_STATUS_ARCH_MISMATCH";
		case CURAND_STATUS_INTERNAL_ERROR: return "CURAND_STATUS_INTERNAL_ERROR";
	}
	return "UNKNOWN ERROR";
}
/*---------------------------------*/
/* PROGRAM PARAMETERS - EQUIVALENT */
/* OF MATLAB SCRIPT PARAMETERS FOR */
/* THE EXECUTION OF KSVD ALGORITHM */
/*---------------------------------*/
#pragma once
#ifndef __PROGPARAMS__
#define __PROGPARAMS__


/*****************************/
/*    Program parameters     */
/*****************************/
// Number of iterations of the K-SVD algorithm
// MATLAB variable: params.iternum = 30
// Implementation maximum: 1024
#define NUMBERofITERATIONS 30
// Sparsity level threshold
// MATLAB variable: params.Tdata = 6
#define Tdata                    6
// Mutual incoherence threshold
// MATLAB variable: params.muthresh = 0.8
#define muTHRESH           0.8


#endif // !__PROGPARAMS__

/*---------------------------*/
/*   GLOBAL OBJECTS STORAGE   */
```

```
/*----------------------------*/
#pragma once
#ifndef _GLOBALS_
#define _GLOBALS_


// Namespace used for global variables
namespace Globals {

	// Global values
	extern unsigned int rowsX;
	extern unsigned int colsX;
	extern unsigned int rowsD;
	extern unsigned int colsD;
	extern unsigned int TPB_rowsX;
	extern unsigned int TPB_colsD;

};

#endif // !_GLOBALS_

/*------------------------------*/
/* GLOBAL VARIABLES DEFINITIONS */
/*------------------------------*/
#include "Globals.cuh"


/*-----------------------*/
/* Rows of input array X */
/*-----------------------*/
unsigned int Globals::rowsX = 0;

/*--------------------------*/
/* Columns of input array X */
/*--------------------------*/
unsigned int Globals::colsX = 0;

/*-----------------------*/
/* Rows of input array D */
/*-----------------------*/
unsigned int Globals::rowsD = 0;

/*--------------------------*/
/* Columns of input array D */
/*--------------------------*/
unsigned int Globals::colsD = 0;

/*-------------------------------*/
/* Recommended threads per block */
/* for a kernel that uses colsX. */
/*-------------------------------*/
unsigned int Globals::TPB_rowsX = 0;

/*-------------------------------*/
/* Recommended threads per block */
/* for a kernel that uses colsD. */
/*-------------------------------*/

unsigned int Globals::TPB_colsD = 0;

/*-------------------------------*/
/* GLOBAL CONSTANTS - STRUCTURES */
/*-------------------------------*/
#pragma once
#ifndef __DECLARATIONS__
```

```
#define __DECLARATIONS__


/***************
*   INCLUDES   *
*    ________    *
***************/
// These should be included from every
// compilation unit
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <cublas_v2.h>
#include "ProgramParameters.cuh"

/*************************
*   TYPEDEFS - SWITCHES   *
*    ___________________     *
*************************/
// Comment out the following #define
// for single-precision operations
// or leave as-is for double precision
#define DOUBLE
// The type of data used in the
// program ( can be either float
// or double)
#ifdef     DOUBLE
typedef double datatype;
#else
typedef float datatype;
#endif // !DOUBLE

/*****************************
*   DEFINITIONS - CONSTANTS   *
*    _______________________      *
*****************************/
// Number of active threads per reduction.
// (Half of the number of elements per single reduction,
// since only half of them are needed).
// Current: 512 threads/1024 elements
#define REDUCTION_BLCK_SIZE512
// Random generator mode of operation:
// fully random ( 0 ) or pre-defined
// numbers for comparison to MATLAB (1).
#define STATIC_GENERATION  0
// At least this number of samples must use the
// atom to be kept and not be cleared. This
// threshold is used when clearing the dictionary
// in the last step of the KSVD alogorithm.
#define USE_THRESH              4
// We use the square value of the Mutual incoherence
// threshold in the program so we #define it explicitly
// as well
#define SQR_muTHRESH       muTHRESH * muTHRESH
// Maximum number of signals used in the special case that
// sprow() did not find any non-zero element in a row of
// Gamma. Consequently we use at most that many signals
// to construct a new atom.
#define MAX_SIGNALS             5000

/************
*   PATHS   *
*    _____ *
***********/
#define    Xarray_in_PATH       "./input/X.mat"
#define    Darray_in_PATH       "./input/D.mat"
```

```
#define    Darray_out_PATH       "./output/D.mat"
#define    temp_out_PATH         "./output/temp.mat"
#define    MATLAB_include        C:\Program Files\MATLAB\MATLAB Production
Server\R2015a\extern\include\\

/************
*   MACROS   *
*     ______      *
************/
#define QUOTEME(x)              QUOTEME_1(x)
#define QUOTEME_1(x)       #x
#define INCLUDE_F(x,y)          QUOTEME(x ## y)
#define INCLUDE_FILE(x,y)  INCLUDE_F(x,y)
#define SINGLE_THREAD(x,y) x ## <<<1,1>>> ## y
#define    SQR(x)                     (x)*(x)
#define MIN(a,b)           ( a < b ? a : b)

#endif // !__DECLARATIONS__

/*----------------------------------------*/
/*  BATCH-OMP CUDA ALGORITHM BASE CLASS   */
/*----------------------------------------*/
#pragma once
#ifndef _BOMP_
#define _BOMP_


// Base class implementing the
// Batch-OMP (Orthogonal Matching Pursuit)
// algorithm
class OMP : public virtual CudaAlgorithm{

     private:
          // Private Functions
          bool  parallel_signals_operations();

     protected:
          // Default Constructor
          OMP();

          // Protected Members
          bool  BatchOMP();

};

#endif // !_BOMP_

/*------------------------------*/
/*   BATCH-OMP CUDA ALGORITHM   */
/*       IMPLEMENTATION         */
/*------------------------------*/
#include <vector>
#include <thrust/device_vector.h>
#include "GlobalDeclarations.cuh"
#include "Globals.cuh"
#include "FileManager.cuh"
#include "HostMemory.cuh"
#include "DeviceMemory.cuh"
#include "CudaAlgorithm.cuh"
#include "OMP.cuh"
#include "Algorithms.cuh"
#include "Utilities.cuh"

using namespace std;
```

```
/*****************
*    NAMESPACES    *
*    __________     *
*****************/
using DeviceSpace      = DeviceMemory<datatype>;
using HostSpace        = HostMemory<datatype>;

/*******************
*    DECLARATIONS    *
*    ____________     *
*******************/
inline void parallelOMP(datatype*, datatype*, datatype*, datatype*, int, int);

/*---------------*/
/*  Constructor  */
/*---------------*/
OMP::OMP() : CudaAlgorithm(NULL,NULL){
}

/*---------------*/
/* OMP function  */
/*---------------*/
bool OMP::BatchOMP() {
      // DtX = D'*data  ==>  DtX = D'*X  ==>  DtX = D_temp'*X_ARRAY
      if (Algorithms::Multiplication::AT_times_B(this->CBhandle,
                  this->deviceMemory.get(DeviceSpace::TEMP_ROWSD_BY_COLSD),
                  Globals::rowsD, Globals::colsD,
                  this->deviceMemory.get(DeviceSpace::X_ARRAY),
                  Globals::rowsX, Globals::colsX,
                  this->deviceMemory.get(DeviceSpace::DtX)
            ) == false) {

            return false;
      }
      // Start Batch-OMP in parallel fashion now!
      if (this->parallel_signals_operations() == false) {
            return false;
      }
      return true;
}

/*-------------------------------*/
/* Implementation of the parallel */
/* iterations of the OMP alg.     */
/*-------------------------------*/
bool OMP::parallel_signals_operations() {
      parallelOMP(
            this->deviceMemory.get(DeviceSpace::DtX),
            this->deviceMemory.get(DeviceSpace::SELECTED_ATOMS),
            this->deviceMemory.get(DeviceSpace::G),
            this->deviceMemory.get(DeviceSpace::c),
            Globals::colsD, Globals::colsX);
      return true;
}

///////////////////////////////////////////////////
///////////     HELPER FUNCTIONS    ///////////////
///////////////////////////////////////////////////

/*************************************
*           DECLARATIONS            *
*           ____________             *
*************************************/
```

```
__global__
void parOMP(datatype*, datatype*, datatype*, datatype*, int, int);

/***********************************
*            WRAPPERS             *
*              ________             *
***********************************/

/*-----------------------------------*/
/* OPERATION:                        */
/* ----------                        */
/* for (signum=0; signum<L; ++signum) */
/* {                                 */
/*     ...                             */
/* }                                 */
/* *in parallel threads*             */
/*-----------------------------------*/
inline void parallelOMP(
            datatype* DtX, datatype* selected_atoms,
            datatype* G, datatype* c,
            int colsD, int colsX) {

      // We set 512 threads per block as
      // it was experimentally found that
      // there are not enough resources on the
      // GPU to utilize this kernel.
      parOMP << < (colsX + 512 - 1) / 512, 512 >> > (
            DtX, selected_atoms, G,
            c, colsD, colsX);

}

/***********************************
*            KERNELS              *
*              _______              *
***********************************/

/*--------------------------------*/
/* This function computes the back */
/* substitution of a linear system */
/* L'*x = b where L is product of  */
/* a Cholesky factorization.       */
/*--------------------------------*/
__inline__ __device__
void backsubst_Lt(datatype* L, datatype* b, datatype* x, int n, int k) {
      unsigned int i, j;
      datatype rhs;

      for (i = k; i >= 1; --i) {
            rhs = b[i - 1];
            for (j = i; j<k; ++j) {
                  rhs -= L[(i - 1)*n + j] * x[j];
            }
            x[i - 1] = rhs / L[(i - 1)*n + i - 1];
      }
}

/*--------------------------------*/
/* This function computes the back */
/* substitution of a linear system */
/* L*x = b where L is product of a */
/* Cholesky factorization. It also */
/* uses indexed input.             */
/*--------------------------------*/
__inline__ __device__
```

```
void backsubt_indexed_L(datatype* L, datatype* b, datatype* x, unsigned int* ind, int n,
int k) {
	unsigned int i, j;
	datatype rhs;

	for (i = 0; i<k; ++i) {
		rhs = b[ind[i]];
		for (j = 0; j<i; ++j) {
			rhs -= L[j*n + i] * x[j];
		}
		x[i] = rhs / L[i*n + i];
	}
}

/*---------------------------------*/
/* This function computes the back */
/* substitution of a linear system */
/* L*x = b where L is product of a */
/* Cholesky factorization. It also */
/* uses indexed input and in-place */
/* output.                         */
/*---------------------------------*/
__inline__ __device__
void backsubt_indexed_append_L(datatype* L, datatype* b, unsigned int* ind, int n, int k,
datatype* temp_storage) {
	unsigned int i, j;
	datatype rhs;

	for (i = 0; i<k; ++i) {
		rhs = b[ind[i]];
		for (j = 0; j<i; ++j) {
			rhs -= L[j*n + i] * temp_storage[j];
		}
		temp_storage[i] = rhs / L[i*n + i];
		L[i*n + k] = temp_storage[i];
	}
}

/*---------------------------------*/
/* Helper function to zero-out the */
/* specified array.                */
/*---------------------------------*/
__inline__ __device__
void zeroOut(datatype* array, int colsD) {
	for (int k = 0; k < colsD; k++) {
		array[k] = 0.0;
	}

}

/*-------------------------------*/
/* Helper function to return the */
/* index of the element of the   */
/* maximum absolute value inside */
/* the specified array.          */
/*-------------------------------*/
__inline__ __device__
unsigned int maxabs(datatype* array, int size) {
	datatype max = SQR(*array), value;
	unsigned int index = 0;
	for (int k = 1; k < size; k++) {
		value = SQR(array[k]);
		if (value > max) {
			max = value;
			index = k;
```

```
			}
		}
		return index;
}

/*-------------------------------*/
/* Helper function to return the */
/* sum of the square values of   */
/* the elements in the specified */
/* array.                        */
/*-------------------------------*/
__inline__ __device__
datatype sumSquareValues(datatype* array, int size, int stride) {
		datatype sum = 0;
		for (int k = 0; k < size; k++) {
			sum += SQR(array[k*stride + size]);
		}
		return (1 - sum);
}

/*---------------------------*/
/* Helper function for:      */
/*     alpha = D'*residual   */
/*---------------------------*/
__inline__ __device__
void addMul(
		datatype* A, int rowsA, int colsA, datatype* x,
		unsigned int* ind, datatype* alpha, datatype* alpha_original) {

		datatype sum;
		for (int i = 0; i < rowsA; i++) {
			sum = 0;
			for (int j = 0; j < colsA; j++) {
				sum += A[ind[j] * rowsA + i] * x[j];
			}
			alpha[i] = alpha_original[i] - sum;
		}
}

/*-----------------------------------*/
/* This kernel executes the parallel */
/* loops for calculating the sparse  */
/* representation of each signal.    */
/*-----------------------------------*/
__global__
void parOMP(
			datatype* DtX, datatype* selected_atoms, datatype* G, datatype* c,
			int colsD, int colsX){
		// >
		// Start Batch-OMP...
		//...................................
		//  Parallel operations from now on!
		//...................................
		// Index of the current signal
		unsigned int signum = blockIdx.x * blockDim.x + threadIdx.x;
		if (signum < colsX) {
			// Local Batch-O.M.P. matrices
			datatype cholBuffer[Tdata];
			unsigned int ind[Tdata];
			datatype Lchol[Tdata*Tdata];
			datatype alpha[32];		// colsD <= 32 as in the rest of the program
			// Increasing factors
			const unsigned int inc_colsD = signum*colsD;
			const unsigned int inc_Tdata = signum*Tdata;
			// We reset the 'selected atoms' array to zero because
```

```
            // of remnants from previous iterations of K-SVD.
            zeroOut(selected_atoms + inc_colsD, colsD);
            // Stack variables
            unsigned int i = 0, pos;
            datatype sum;

            /* >>> O.M.P. - Main Iteration <<< */
            while (i < Tdata) {

                  if (i == 0) {
                        /* Avoid copy and use DtX directly*/

                        // Index of next atom
                        // OPERATION :=> pos = maxabs(alpha, m)
                        pos = maxabs(DtX + inc_colsD, colsD);

                        // Stop criterion: selected same atom twice, or inner product too
small
                        if ((selected_atoms + inc_colsD)[pos] || SQR((DtX +
inc_colsD)[pos]) < 1e-14) {
                              break;
                        }
                  }
                  else {
                        /* Use alpha array for any subsequent iteration */

                        // Index of next atom
                        // OPERATION :=> pos = maxabs(alpha, m)
                        pos = maxabs(alpha, colsD);

                        // Stop criterion: selected same atom twice, or inner product too
small
                        if ((selected_atoms + inc_colsD)[pos] || SQR(alpha[pos]) < 1e-14) {
                              break;
                        }
                  }

                  // Mark selected atom
                  ind[i] = pos;
                  //////////////// MODIFIED ////////////////
                  /* (see note at the end of this function) */
                  (selected_atoms + inc_colsD)[pos] = i + 1;

                  // Cholesky update
                  if (i == 0) {
                        *Lchol = 1;
                  }
                  else {
                        /* incremental Cholesky decomposition: compute next row of Lchol */
                        backsubt_indexed_append_L(
                              Lchol,
                              G + pos*colsD,
                              ind,
                              Tdata, i,
                              cholBuffer);

                        /* compute Lchol(i,i) */
                        sum = sumSquareValues(Lchol, i, Tdata);

                        if (sum <= 1e-14) {
                              break;
                        }

                        Lchol[i*Tdata + i] = sqrt(sum);
                  }
```

```
            i++;

            /* perform orthogonal projection and compute sparse coefficients */

            // The two following functions together
            // compute the Cholesky factorization
            // of a given array and solve the
            // corresponding set of linear systems
            // by executing types of back substitution.
            backsubt_indexed_L(
                  Lchol,
                  DtX + inc_colsD,
                  cholBuffer,
                  ind,
                  Tdata, i);
            backsubst_Lt(
                  Lchol,
                  cholBuffer,
                  c + inc_Tdata,
                  Tdata, i);


            /* update alpha = D'*residual */

            // Here we make the assumption that 'colsD' cannot be more than 32 atoms
            // (i.e. signal sources ). Also 'i' cannot be more than colsD because the
            // loop would have 'breaked' in line 316
            addMul(
                  G, colsD, i,
                  c + inc_Tdata,
                  ind,
                  alpha,
                  DtX + inc_colsD
            );
      }
      /* >>> O.M.P. - Post Iteration Finalization <<< */
      /*
        Note: At this point the CPU equivalent of OMP produces a
                  sparse array Gamma using array 'c' and indices array
                  'ind'. However the next step ( i.e the first step of
                  Dictionary update phase) commands that only non-zero
                  elements of Gamma can be used and their respective
                  indices. That can be easily implemented by launching
                  a new grid to recover the data and as such the need
                  for a temporary storage is no longer neccessary. For
                  this to happen, however, a slight modification has to
                  be made so that the 'selected atoms' array informs us
                  not only whether a non-zero element is present at a
                  specific position but also its index in the output
                  array ('c').
      */
      /*
        Modification summary:
                  ORIGINAL selected_atoms ARRAY => stores 0 or 1 to indicate
                                                          presence/absence of
element
                  MODIFIED VERSION => stores index of element in range {1...Tdata}
                                                 or 0 to indicate zero-element

        A non-zero element can later be found through row and column indices
        of Gamma:
                  For a *non-zero* element at Full_Gamma[i][j] we get its value at:
                        (c + j*Tdata) [ (int) ( (selected_atoms + j*colsD) [ i ] - 1 )
]
```

```
			*/
		}
		//..................................
		//      END OF PARALLEL SECTION
		//..................................
}

/*--------------------------------*/
/*          KSVD CLASS            */
/*--------------------------------*/
#pragma once
#ifndef _KSVD_
#define _KSVD_


// Class used for executing the
// K-SVD algorithm
class KSVD : public OMP, public DictionaryUpdate {

	private:
		// Private Functions
		bool normcols();
		bool errorComputation(unsigned int);
		bool clearDict();

	public:
		// Constructor
		KSVD(DeviceMemory<datatype>*, HostMemory<datatype>*);
		// Destructor
		~KSVD();

		// Public Methods
		bool ExecuteIterations(unsigned int);
		bool isValid();

};

#endif // !_KSVD_

/*------------------------------*/
/*       KSVD-OPERATION         */
/*       IMPLEMENTATION         */
/*------------------------------*/
#include <vector>
#include <thrust/device_vector.h>
#include <thrust/sort.h>
#include "GlobalDeclarations.cuh"
#include "Globals.cuh"
#include "FileManager.cuh"
#include "HostMemory.cuh"
#include "DeviceMemory.cuh"
#include "CudaAlgorithm.cuh"
#include "OMP.cuh"
#include "DictionaryUpdate.cuh"
#include "KSVD.cuh"
#include "ErrorHandler.cuh"
#include "ThrustOperators.cuh"
#include "Utilities.cuh"
#include "Algorithms.cuh"


/****************
*   NAMESPACES   *
*    __________   *
```

```
*****************/
using DeviceSpace     = DeviceMemory<datatype>;
using HostSpace       = HostMemory<datatype>;

/******************
*   DECLARATIONS   *
*    ____________   *
******************/
inline void diagX_Mul(datatype*,datatype*,datatype*,int,int);
inline bool use_count(datatype*, datatype*, unsigned int, unsigned int);
inline void replace(datatype*, datatype*, datatype*, unsigned int, unsigned int);
inline void initialize(datatype*, datatype*, datatype*, datatype*, unsigned int, unsigned
int);

/*---------------*/
/* Constructor   */
/*---------------*/
KSVD::KSVD(DeviceSpace* devptr, HostSpace* hostptr)
      : CudaAlgorithm(devptr, hostptr) {
}

/*--------------*/
/* Destructor   */
/*--------------*/
KSVD::~KSVD() {
}

/*-------------------------------*/
/*  Perform the K-SVD algorithm  */
/*-------------------------------*/
bool KSVD::ExecuteIterations(unsigned int iternum) {
      // D_temp = normcols(D);
      if (normcols() == false) {
            return false;
      }
      // Main Loop
      unsigned int* all_perms;
      for (unsigned int iter = 0; iter < iternum; iter++) {
            // G = D'*D  ==>  G = D_temp'*D_temp
            if (Algorithms::Multiplication::AT_times_B(this->CBhandle,
                        this->deviceMemory.get(DeviceSpace::TEMP_ROWSD_BY_COLSD),
                        Globals::rowsD, Globals::colsD,
                        this->deviceMemory.get(DeviceSpace::TEMP_ROWSD_BY_COLSD),
                        Globals::rowsD, Globals::colsD,
                        this->deviceMemory.get(DeviceSpace::G)
                  ) == false) {

                  return false;
            }

            //////////////////////////
            // Sparse coding stage! //
            //////////////////////////
            // Execute Batch-OMP stage now!
            // Equivalent: Gamma = sparsecode(data,D,XtX,G,thresh)
            if (this->BatchOMP() == false) {
                  return false;
            }
            /*
                  Note:
                        Now array Gamma is not present but its elements are accessed
                        using the following formula:

                   For a *non-zero* element at (i,j) we get its value at:
                        (c + j*Tdata) [ (int) ( (selected_atoms + j*colsD) [ i ] - 1 ) ]
```

```
		*/

		//////////////////////////////////
		// Reset variables for the new //
		// iteration before Dictionary //
		// Update stage.               //
		//////////////////////////////////
		initialize(
			this->deviceMemory.get(DeviceSpace::UNUSED_SIGS),
			this->deviceMemory.get(DeviceSpace::REPLACED_ATOMS),
			this->deviceMemory.get(DeviceSpace::UNUSED_SIGS_COUNTER),
			this->deviceMemory.get(DeviceSpace::UNUSED_SIGS_BITMAP),
			Globals::colsX, Globals::colsD
		);

		///////////////////////////////
		// Dictionary update stage! //
		///////////////////////////////
		// Generate a random permutation of indices in the
		// range 1...colsD. Equivalent: p = randperm(dictsize);
		all_perms = Algorithms::RandomPermutation::generateRandPerm(Globals::colsD);
		// Execute Dictionary Update stage now!
		if (this->updateDictionary(all_perms) == false) {
			return false;
		}

		/////////////////////////
		// Compute error now! //
		/////////////////////////
		if (this->errorComputation(iter + 1) == false) {
			return false;
		}

		/////////////////////////
		//  Clear Dictionary  //
		/////////////////////////
		if (this->clearDict() == false) {
			return false;
		}
	}
	return true;
}

/*-----------------------------------------*/
/* Clear the dictionary of unused atoms    */
/* or atoms having error above a threshold */
/* or atoms that few samples use them!     */
/*-----------------------------------------*/
bool KSVD::clearDict() {
	// Count how many elements in every row of Gamma
	// have absolute value above 1e-7
	if (use_count(
			this->deviceMemory.get(DeviceSpace::SELECTED_ATOMS),
			this->deviceMemory.get(DeviceSpace::TEMP_1_BY_COLSD),
			Globals::colsD, Globals::colsX
		)
		== false) {

		return false;
	}
	// Iteration to clear every atom that satisfies
	// a certain condition implemented in maximum
	// function below. Matlab equivalent:
	//
	// for j = 1:dictsize
```

```
      // | % compute G(:, j)
      // | % replace atom
      // end
      //
      for (unsigned int j = 0; j < Globals::colsD; j++) {
            // Now we compute:
            //
            //      Gj = D'*D(:,j);
            //
            if (Algorithms::Multiplication::AT_times_x(this->CBhandle,
                        this->deviceMemory.get(DeviceSpace::TEMP_ROWSD_BY_COLSD),
                        Globals::rowsD,
                        this->deviceMemory.get(DeviceSpace::TEMP_ROWSD_BY_COLSD) +
j*Globals::rowsD,
                        Globals::colsD,
                        this->deviceMemory.get(DeviceSpace::G)
                  )
                  == false) {

                  return false;
            }
            // Now we compute the maximum (square) value
            // of Gj. Operation performed:
            //
            //            (max(Gj.^2))
            //
            // We also apply the condition.
            if (Algorithms::Reduction::Maximum_squared(
                        this->deviceMemory.get(DeviceSpace::G),
                        Globals::colsD,
                        this->deviceMemory.get(DeviceSpace::TEMP_1_BY_COLSD) + j,
                        this->deviceMemory.get(DeviceSpace::REPLACED_ATOMS) + j,
                        j
                  )
                  == false) {

                  return false;
            }
            // We now find the signal with the maximum error.
            // Matlab equivalent:
            //
            //      [~,i] = max(err);
            //
            if (Algorithms::Reduction::Maximum(
                        this->deviceMemory.get(DeviceSpace::ERR),
                        Globals::colsX,
                        this->deviceMemory.get(DeviceSpace::G)
                  )
                  == false) {

                  return false;
            }
            // We should now calculate the norm of the
            // selected signal in our data.
            if (Algorithms::Reduction::euclidean_norm(
                        this->deviceMemory.get(DeviceSpace::X_ARRAY),
                        this->deviceMemory.get(DeviceSpace::G),
                        Globals::rowsX, Globals::TPB_rowsX
                  )
                  == false) {

                  return false;
            }
            // Finally replace atom.
            replace(
```

```
                this->deviceMemory.get(DeviceSpace::TEMP_ROWSD_BY_COLSD) + j *
Globals::rowsD,
                this->deviceMemory.get(DeviceSpace::X_ARRAY),
                this->deviceMemory.get(DeviceSpace::G),
                Globals::rowsX, Globals::TPB_rowsX
           );
      }
      return true;
}

/*-----------------------------------*/
/* Compute the residual error        */
/* using the formula:                */
/*                                   */
/*  sqrt(sum[(X-D*Gamma)^2]/numel(X)) */
/*                                   */
/*-----------------------------------*/
bool KSVD::errorComputation(unsigned int iter) {
      // Compute (D*Gamma) using CUBLAS
      // for performance
      if (Algorithms::Multiplication::A_times_B(this->CBhandle,
                this->deviceMemory.get(DeviceSpace::TEMP_ROWSD_BY_COLSD), Globals::rowsD,
Globals::colsD,
                this->deviceMemory.get(DeviceSpace::SELECTED_ATOMS), Globals::colsD,
Globals::colsX,
                this->deviceMemory.get(DeviceSpace::TEMP_ROWSX_BY_COLSX))
           == false) {

           return false;
      }
      // Now reduce over the result and calculate
      // the error:
      //         sum{ (X - X~)^2 }
      //
      if (Algorithms::Reduction::reduce_RMSE(
                this->deviceMemory.get(DeviceSpace::X_ARRAY),
                this->deviceMemory.get(DeviceSpace::TEMP_ROWSX_BY_COLSX),
                this->deviceMemory.get(DeviceSpace::ERR),
                Globals::rowsX , Globals::colsX, iter,
                Globals::TPB_rowsX,
                this->deviceMemory.get(DeviceSpace::UNUSED_SIGS_BITMAP))
           == false) {

           return false;
      }
      return true;

}


/*------------------------*/
/*  Normalize columns of D */
/*------------------------*/
bool KSVD::normcols() {
      // Calculating ==> 1./sqrt(sum(D.*D))
      if (Algorithms::Reduction::reduce2D_Sum_SQUAREDelements(
                this->deviceMemory.get(DeviceSpace::D_ARRAY),
                Globals::rowsD, Globals::colsD,
                this->deviceMemory.get(DeviceSpace::TEMP_1_BY_COLSD),
                Globals::TPB_rowsX
           ) == false) {

           return false;
      }
      // D = D*spdiag( 1./sqrt(sum(D.*D)) ) ==> D = D*spdiag(TEMP_1_BY_COLSD)
```

```
	diagX_Mul(
		this->deviceMemory.get(DeviceSpace::D_ARRAY),
		this->deviceMemory.get(DeviceSpace::TEMP_ROWSD_BY_COLSD),
		this->deviceMemory.get(DeviceSpace::TEMP_1_BY_COLSD),
		Globals::rowsD * Globals::colsD, Globals::rowsD);

	return true;
}

/*-------------------------*/
/* Check if error occured  */
/*-------------------------*/
bool KSVD::isValid() {
	return(CudaAlgorithm::isValid());
}

////////////////////////////////////////////////////
///////////    HELPER FUNCTIONS   ////////////////
////////////////////////////////////////////////////

/*************************************
*           DECLARATIONS            *
*            ____________             *
*************************************/
__global__
void diagMul(datatype*,datatype*, datatype*, int, int);
__global__
void use_count_kernel(datatype*, datatype*, unsigned int, unsigned int);
__global__
void replaceDj(datatype*, datatype*, datatype*, unsigned int);
__global__
void init(datatype*, datatype*, datatype*, datatype*, unsigned int, unsigned int);

/*************************************
*            WRAPPERS               *
*             ________              *
*************************************/

/*---------------------------------*/
/* OPERATION:                      */
/*      out = in*diag(coeff)       */
/*---------------------------------*/
inline void diagX_Mul(datatype* in, datatype* out, datatype* coeff, int size, int rows) {
	dim3 block(1024);
	dim3 grid((size + block.x - 1) / block.x);
	diagMul <<< grid, block >>> (in, out, coeff, size, rows);
}

/*------------------------------*/
/* OPERATION:                   */
/*                              */
/*  usecount =                  */
/*   sum(abs(Gamma)>1e-7, 2);   */
/*                              */
/*------------------------------*/
inline bool use_count(
	datatype* gamma, datatype* counters,
	unsigned int colsD, unsigned int colsX) {

	// Initialize counters to zero
	if (cudaMemset(counters, 0, colsD * sizeof(datatype)) != cudaSuccess) {
		return false;
	}

	// Once again we assume colsD <= 32
```

```
	dim3 block(16, colsD); // => max. 512 threads per block
	dim3 grid((colsX + block.x - 1) / block.x);
	use_count_kernel << < grid, block >> > (gamma, counters, colsD, colsX);

	return true;
}

/*---------------------------------*/
/* OPERATION:                      */
/*   D(:,j) = X(:,unused_sigs(i))  */
/*     / norm(X(:,unused_sigs(i))) */
/*---------------------------------*/
inline void replace(
	datatype* Dj, datatype* X, datatype* G,
	unsigned int size, unsigned int recommended) {

	replaceDj << < 1, recommended >> > (Dj, X, G, size);
}

/*-----------------------------------*/
/* OPERATIONS:                       */
/*  replaced_atoms = zeros(dictsize) */
/*  unused_sigs = 1:size(data,2);    */
/*-----------------------------------*/
inline void initialize(
	datatype* un, datatype* rep, datatype* counter, datatype* bitmap,
	unsigned int N, unsigned int colsD) {

	dim3 block(1024);
	dim3 grid((N + block.x - 1) / block.x);
	init << <grid, block>> > (un, rep, counter, bitmap, N, colsD);
}

/************************************
*            KERNELS               *
*             _______               *
************************************/

/*---------------------------*/
/* This kernel multiplies a  */
/* vector transformed into a */
/* diagonal matrix with some */
/* other matrix.             */
/*---------------------------*/
__global__
void diagMul(datatype* in, datatype* out, datatype* coeff, int size, int rows) {
	int index = blockIdx.x * blockDim.x + threadIdx.x;
	if (index < size) {
		int col = index / rows;
		out[index] = in[index] * coeff[col];
	}
}

/*-----------------------------------*/
/* This kernel uses an array of      */
/* counters to simultaneously count  */
/* elements of every row of input    */
/* that satisfy a certain condition. */
/*-----------------------------------*/
__global__ void use_count_kernel(
	datatype* gamma, datatype* counters,
	unsigned int colsD, unsigned int colsX) {

	// threadIdx.x = my column
	// threadIdx.y = my row
```

```
	unsigned int column = blockIdx.x * blockDim.x + threadIdx.x;
	if (column < colsX) {
		if (SQR((gamma + column*colsD)[threadIdx.y]) > 1e-7) {
			atomicAdd((unsigned int*)(counters + threadIdx.y), 1);
		}
	}
}

/*---------------------------------*/
/* This kernel performs the simple */
/* task of:                        */
/*  D(:,j) = X(:,i) / norm(X(:,i)) */
/*---------------------------------*/
__global__ void replaceDj(
	datatype* Dj, datatype* X, datatype* G,
	unsigned int size) {

	if (!(*G)) {
		// Our atom does not need replacement
		return;
	}
	if (threadIdx.x < size) {
		X += (unsigned int)(*(G + 1))*size;
		Dj[threadIdx.x] = X[threadIdx.x] / (*(G + 2));
	}
}

/*-----------------------------------*/
/* This kernel is the equivalent of: */
/*                                   */
/*  replaced_atoms = zeros(dictsize) */
/*  unused_sigs = 1:size(data,2)     */
/*-----------------------------------*/
__global__ void init(
	datatype* un, datatype* rep, datatype* counter, datatype* bitmap,
	unsigned int N, unsigned int N2) {

	int index = blockIdx.x * blockDim.x + threadIdx.x;
	if (index < N) {
		un[index] = index;
		bitmap[index] = 1.0;
		if (index < N2) {
			rep[index] = 0.0;
		}
		if (index == 0) {
			*counter = 0.0;
		}
	}

}

/*-----------------------------------------------*/
/*  k-MEANS DICTIONARY UPDATE STAGE BASE CLASS   */
/*-----------------------------------------------*/
#pragma once
#ifndef _DICUPD_
#define _DICUPD_


// Base class implementing the
// Dictionary Update Stage
class DictionaryUpdate : public virtual CudaAlgorithm {

	private:
		// Private Functions
```

```
		bool  Sprow();
		bool  specialCase(unsigned int);

	protected:
		// Default Constructor
		DictionaryUpdate();

		// Protected Members
		bool  updateDictionary(unsigned int*);

};

#endif // !_DICUPD_

/*---------------------------------*/
/* k-MEANS DICTIONARY UPDATE STAGE */
/*          IMPLEMENTATION         */
/*---------------------------------*/
#include <vector>
#include <thrust/device_vector.h>
#include "GlobalDeclarations.cuh"
#include "Globals.cuh"
#include "FileManager.cuh"
#include "HostMemory.cuh"
#include "DeviceMemory.cuh"
#include "CudaAlgorithm.cuh"
#include "DictionaryUpdate.cuh"
#include "Utilities.cuh"
#include "Algorithms.cuh"
#include "ReductionBase.cuh"

using namespace std;


/****************
*   NAMESPACES   *
*     ________    *
****************/
using DeviceSpace      = DeviceMemory<datatype>;
using HostSpace        = HostMemory<datatype>;

/******************
*   DECLARATIONS   *
*     ____________    *
******************/
inline void sprow(datatype*, datatype*, datatype*, datatype*, datatype*, int, int);
inline void special_case_update(datatype*, datatype*, datatype*, unsigned int, datatype*,
datatype*, datatype*,
						datatype*, datatype*, unsigned int, unsigned
int);

/*--------------*/
/* Constructor  */
/*--------------*/
DictionaryUpdate::DictionaryUpdate() : CudaAlgorithm(NULL, NULL) {
}

/*--------------------*/
/* Dictionary Update  */
/*--------------------*/
bool DictionaryUpdate::updateDictionary(unsigned int* p) {
	// We retrieve all the non-zero elements
	// along with their indices for every row
	// of the O.M.P. output Gamma. That way we
```

```
// have eliminated the operation:
//    [gamma_j, data_indices] =
//            sprow(Gamma, j)
//
// in each iteration.
// Output stored in 'DtX' and 'alpha'.
if (this->Sprow() == false) {
     return false;
}
/**/
/* Analyze the data obtained by sprow in parallel */
/**/
// We pre-calculate the sum of squares for
// each gamma_j because it remains constant
// for every iteration. Thus we eliminate the
// calculation of:
//          (gamma_j*gamma_j')
//
// for each row's loop. Output stored in 'G'.
// Simultaneously, we pre-calculate the linear
// combination of the columns of X using gamma_j
// as the coefficients matrix. Essentially this
// operation is:
//              Y = X(:, COLS)*gamma_j'
//
// But each gamma_j remains constant
// for every iteration. Thus we eliminate the
// calculation of:
//    collincomb(X,data_indices,gamma_j')
//
// for each row's loop.
// Output stored in 'TEMP_COLSD_BY_ROWSX'.
if (Algorithms::Reduction::reduce1D_Batched_Sum_TOGETHER_collincomb(
          this->deviceMemory.get(DeviceSpace::DtX),
          this->deviceMemory.get(DeviceSpace::G),
          this->deviceMemory.get(DeviceSpace::TEMP_1_BY_COLSD),
          Globals::colsD, Globals::colsX,
          this->deviceMemory.get(DeviceSpace::X_ARRAY),
          this->deviceMemory.get(DeviceSpace::alpha),
          this->deviceMemory.get(DeviceSpace::TEMP_COLSD_BY_ROWSX),
          Globals::rowsX)
     == false){

     return false;
}

//////////////////////////////////////
// Dictionary Update - Main Iteration //
//////////////////////////////////////
unsigned int pJ;
for (unsigned int j = 0; j < Globals::colsD; j++) {
     /*-- Optimize atom --*/
     pJ = p[j];
     // First we handle the special case where sprow()
     // returned zero for this row of Gamma i.e. no
     // non-zero elements were found
     if (specialCase(pJ) == false) {
          return false;
     }
     // Matrix-vector multiplication
     // Operation:
     //          (smallGamma*gamma_j')
     //
     if (Algorithms::Reduction::reduce2D_dot_products_modified(
               this->deviceMemory.get(DeviceSpace::SELECTED_ATOMS),
```

```
                    this->deviceMemory.get(DeviceSpace::alpha) + pJ*Globals::colsX,
                    this->deviceMemory.get(DeviceSpace::DtX) + pJ*Globals::colsX,
                    this->deviceMemory.get(DeviceSpace::G) + Globals::colsD,
                    this->deviceMemory.get(DeviceSpace::TEMP_1_BY_COLSD) + pJ,
                    pJ,
                    Globals::colsD, Globals::colsX)
               == false) {

               return false;
          }
          // Matrix-vector multiplication
          // Operation:
          //         D*(smallGamma*gamma_j')
          //
          if (Algorithms::Multiplication::A_times_x_SUBTRACT(
                    this->deviceMemory.get(DeviceSpace::TEMP_ROWSD_BY_COLSD),
                    Globals::rowsD,
                    this->deviceMemory.get(DeviceSpace::G) + Globals::colsD ,
                    Globals::colsD,
                    this->deviceMemory.get(DeviceSpace::TEMP_COLSD_BY_ROWSX) + pJ *
Globals::rowsD,
                    this->deviceMemory.get(DeviceSpace::TEMP_ROWSD_BY_COLSD) + pJ *
Globals::rowsD,
                    this->deviceMemory.get(DeviceSpace::G) + pJ,
                    this->deviceMemory.get(DeviceSpace::TEMP_1_BY_COLSD) + pJ,
                    Globals::TPB_rowsX)
               == false) {

               return false;
          }
          // We calculate the linear combination
          // of the rows of X using 'atom' as the
          // coefficients matrix. Essentially this
          // operation is:
          //              Y = X'*A(ROWS,:)
          //
          // We also calculate the norm of the new
          // atom in parallel as well as execute the
          // multiplication: (atom'*D).
          if (Algorithms::Reduction::reduce2D_rowlincomb_plus_nrm2_plus_mul(
                    this->deviceMemory.get(DeviceSpace::X_ARRAY),
                    this->deviceMemory.get(DeviceSpace::TEMP_COLSD_BY_ROWSX) + pJ *
Globals::rowsX,
                    this->deviceMemory.get(DeviceSpace::TEMP_SINGLE_VALUE),
                    this->deviceMemory.get(DeviceSpace::alpha) + pJ * Globals::colsX,
                    Globals::rowsX, Globals::colsX,
                    this->deviceMemory.get(DeviceSpace::TEMP_1_BY_COLSD) + pJ,
                    Globals::TPB_rowsX,
                    this->deviceMemory.get(DeviceSpace::c),
                    this->deviceMemory.get(DeviceSpace::TEMP_ROWSD_BY_COLSD),
                    this->deviceMemory.get(DeviceSpace::G) + Globals::colsD,
                    Globals::colsD
               )
               == false) {

               return false;
          }
          //
          // We finally calculate:
          //
          //         (atom'*D)*smallGamma
          //
          // using reduction and simultaneously
          // update D and Gamma with the new values.
          // These two operations can be performed
```

```
			// in parallel!
			if (Algorithms::Multiplication::normal_x_times_A_modified(
					this->deviceMemory.get(DeviceSpace::G) + Globals::colsD,
					this->deviceMemory.get(DeviceSpace::SELECTED_ATOMS),
					this->deviceMemory.get(DeviceSpace::alpha) + pJ * Globals::colsX,
					Globals::colsD,
					this->deviceMemory.get(DeviceSpace::DtX) + pJ * Globals::colsX,
					this->deviceMemory.get(DeviceSpace::TEMP_SINGLE_VALUE),
					this->deviceMemory.get(DeviceSpace::G) + Globals::colsD + pJ,
					this->deviceMemory.get(DeviceSpace::c),
					pJ,
					this->deviceMemory.get(DeviceSpace::TEMP_1_BY_COLSD) + pJ,
					Globals::colsX,
					this->deviceMemory.get(DeviceSpace::TEMP_ROWSD_BY_COLSD) +
pJ*Globals::rowsX,
					this->deviceMemory.get(DeviceSpace::TEMP_COLSD_BY_ROWSX) +
pJ*Globals::rowsX,
					Globals::rowsX
				)
				== false) {

				return false;
			}
		}
		////////////////////////////////////////
		//       End Of Main Iteration        //
		////////////////////////////////////////

		return true;
}

/*------------------------------------*/
/* SPROW: Store all non-zero elements, */
/* then store their indices ( source:  */
/* array Gamma [ i.e. 'c' ] )          */
/*------------------------------------*/
bool DictionaryUpdate::Sprow() {
		// Initialize counters to zero
		if (cudaMemset(
				this->deviceMemory.get(DeviceSpace::TEMP_1_BY_COLSD),
				0,
				Globals::colsD * sizeof(datatype)
			) != cudaSuccess) {

			return false;
		}
		// Execute sprow in a single pass
		sprow(
			this->deviceMemory.get(DeviceSpace::c),
			this->deviceMemory.get(DeviceSpace::SELECTED_ATOMS),
			this->deviceMemory.get(DeviceSpace::TEMP_1_BY_COLSD),
			this->deviceMemory.get(DeviceSpace::DtX),
			this->deviceMemory.get(DeviceSpace::alpha),
			Globals::colsD, Globals::colsX
		);
		return true;
}

/*----------------------------------------*/
/* In this function we handle the special */
/* case when a row of Gamma is not used   */
/* by any atom and thus sprow() returned  */
/* an empty set.                          */
/*----------------------------------------*/
bool DictionaryUpdate::specialCase(unsigned int pJ) {
```

```
	// Compute (D*Gamma) using CUBLAS
	// for performance
	if (Algorithms::Multiplication::A_times_B(this->CBhandle,
				this->deviceMemory.get(DeviceSpace::TEMP_ROWSD_BY_COLSD), Globals::rowsD,
Globals::colsD,
				this->deviceMemory.get(DeviceSpace::SELECTED_ATOMS), Globals::colsD,
Globals::colsX,
				this->deviceMemory.get(DeviceSpace::TEMP_ROWSX_BY_COLSX)
		)
		== false) {

		return false;
	}
	// Now reduce over the result and calculate
	// the error:
	//		E = sum{ (X - X~)^2 }
	//
	// Then we calculate:
	//		[d,i] = max(E);
	//
	// Output 'i' stored in *TEMP_ROWSX_BY_COLSX.
	if (Algorithms::Reduction::reduce_ERROR_for_special_case(
				this->deviceMemory.get(DeviceSpace::X_ARRAY),
				this->deviceMemory.get(DeviceSpace::TEMP_ROWSX_BY_COLSX),
				this->deviceMemory.get(DeviceSpace::ERR),
				Globals::rowsX, Globals::colsX,
				this->deviceMemory.get(DeviceSpace::TEMP_1_BY_COLSD) + pJ,
				this->deviceMemory.get(DeviceSpace::UNUSED_SIGS),
				this->deviceMemory.get(DeviceSpace::UNUSED_SIGS_COUNTER),
				Globals::TPB_rowsX
		)
		== false) {

		return false;
	}
	// We should now calculate the norm of the
	// selected signal in our data.
	if (Algorithms::Reduction::SCase_EU_norm(
				this->deviceMemory.get(DeviceSpace::X_ARRAY),
				this->deviceMemory.get(DeviceSpace::TEMP_ROWSX_BY_COLSX),
				Globals::rowsX, Globals::TPB_rowsX,
				this->deviceMemory.get(DeviceSpace::TEMP_1_BY_COLSD) + pJ,
				this->deviceMemory.get(DeviceSpace::UNUSED_SIGS),
				Globals::colsX
		)
		== false) {

		return false;
	}
	// We finally update the dictionary with the new
	// atom divided by its norm
	special_case_update(
		this->deviceMemory.get(DeviceSpace::X_ARRAY),
		this->deviceMemory.get(DeviceSpace::TEMP_ROWSD_BY_COLSD) + pJ *
Globals::rowsD,
		this->deviceMemory.get(DeviceSpace::TEMP_ROWSX_BY_COLSX),
		Globals::rowsX,
		this->deviceMemory.get(DeviceSpace::TEMP_1_BY_COLSD) + pJ,
		this->deviceMemory.get(DeviceSpace::UNUSED_SIGS),
		this->deviceMemory.get(DeviceSpace::UNUSED_SIGS_COUNTER),
		this->deviceMemory.get(DeviceSpace::UNUSED_SIGS_BITMAP),
		this->deviceMemory.get(DeviceSpace::REPLACED_ATOMS) + pJ,
		Globals::colsX,
		Globals::TPB_rowsX
	);
```

```
	return true;
}

//////////////////////////////////////////////////////
////////////    HELPER FUNCTIONS    ////////////////
//////////////////////////////////////////////////////

/*************************************
*            DECLARATIONS            *
*              ____________            *
*************************************/
__global__
void sprow_kernel(datatype*, datatype*, datatype*, datatype*, datatype*, int, int);
__global__
void SCupdate(datatype*, datatype*, datatype*, unsigned int, datatype*, datatype*,
datatype*, datatype*,
                      datatype*, unsigned int, unsigned int);

/*************************************
*             WRAPPERS               *
*               ________               *
*************************************/

/*------------------------------*/
/* OPERATION:                   */
/*  [gamma_j, data_indices] =   */
/*       sprow(Gamma, j);       */
/*                              */
/* { for every j in 1...colsD } */
/*------------------------------*/
inline void sprow(
	datatype* c, datatype* selected_atoms, datatype* counters,
	datatype* out_values, datatype* out_indices,
	int colsD, int colsX) {

	// Once again we assume colsD <= 32 in
	// order to fit in a single warp
	dim3 block(16,32); // = 512 threads per block
	dim3 grid((colsX + block.x - 1) / block.x);
	sprow_kernel << < grid, block >> > (
		c, selected_atoms, counters,
		out_values, out_indices,
		colsD, colsX);
}

/*------------------------------------*/
/* OPERATIONS:                        */
/*    atom = X(:,unused_sigs(perm(i))) */
/* and                                */
/*     atom = atom./norm(atom)        */
/* and                                */
/*          D(:,p(j)) = atom             */
/*                                    */
/*------------------------------------*/
inline void special_case_update(
	datatype* X, datatype* Dj, datatype* temp, unsigned int N,
	datatype* counters, datatype* unused, datatype* UScounter,
	datatype* bitmap, datatype* replaced, unsigned int colsX,
	unsigned int recommended) {

	// We assume length i.e. rowsD <= 1024
	SCupdate << <1, recommended >> > (X, Dj, temp, N, counters,
		unused, UScounter, bitmap, replaced, colsX, MIN(colsX, MAX_SIGNALS));
}
```

```
/***********************************
*            KERNELS               *
*             _______               *
***********************************/

/*----------------------------------*/
/* This kernel uses an array of      */
/* counters to simultaneously count, */
/* index and store the non-zero      */
/* elements of input, multiplexed by */
/* the selected atoms array, to two  */
/* output arrays.                    */
/*----------------------------------*/
__global__ void sprow_kernel(
      datatype* c, datatype* selected_atoms, datatype* counters,
      datatype* out_values, datatype* out_indices,
      int colsD, int colsX) {

      // threadIdx.x = my column
      // threadIdx.y = my row
      unsigned int column = blockIdx.x * blockDim.x + threadIdx.x, ind, pos;
      if (column < colsX && threadIdx.y < colsD) {
            ind = (selected_atoms + column*colsD)[threadIdx.y];
            if (ind) {
                  pos = atomicAdd((int*)(counters + threadIdx.y), 1);
                  *(out_values + threadIdx.y*colsX + pos) = (c + column*Tdata)[ind - 1];
                  *(out_indices + threadIdx.y*colsX + pos) = column;
                  (selected_atoms + column*colsD)[threadIdx.y] = (c + column*Tdata)[ind -
1];
            }
      }
}

/*-----------------------------------*/
/* OPERATIONS:                        */
/*    atom = X(:,unused_sigs(perm(i))) */
/* and                                */
/*     atom = atom./norm(atom)        */
/* and                                */
/*          D(:,p(j)) = atom             */
/*                                    */
/*-----------------------------------*/
__global__ void SCupdate(
      datatype* X, datatype* Dj, datatype* temp, unsigned int N,
      datatype* counters, datatype* unused, datatype* UScounter,
      datatype* bitmap, datatype* replacedJ, unsigned int colsX,
      unsigned int dim) {

      unsigned int offset, counter;
      if (*((int*)counters) == 0) {
            if (threadIdx.x < N) {
                  offset = (unsigned int)unused[myGenerator((unsigned int)(*temp), dim)];
                  X += offset*N;
                  Dj[threadIdx.x] = X[threadIdx.x] / (*(temp + 1));
                  if (threadIdx.x == 0) {
                        counter = (*UScounter);
                        unused[myGenerator((unsigned int)(*temp), dim)] = unused[colsX -
counter - 1];
                        (*UScounter) = counter + 1;
                        *replacedJ = 1;
                        bitmap[offset] = 0;
                  }
            }
      }
```

```
}

/*------------------------------*/
/*  CUDA ALGORITHM BASE CLASS   */
/*------------------------------*/
#pragma once
#ifndef _CUALG_
#define _CUALG_


// Base class inherited from all
// algorithms implemented in the GPU
class CudaAlgorithm {

	protected:
		// Constructor
		CudaAlgorithm(DeviceMemory<datatype>*, HostMemory<datatype>*);
		// Destructor
		~CudaAlgorithm();

		/*-------------------*/
		/* Protected Members */
		/*-------------------*/
		// References to the device and
		// host memory objects so that all
		// functions have access
		DeviceMemory<datatype>&    deviceMemory;
		HostMemory<datatype>& hostMemory;
		// Cublas Handle
		cublasHandle_t             CBhandle;
		// Error flag
		bool                       valid;

		/*-------------------*/
		/* Protected Methods */
		/*-------------------*/
		// Evaluates the flag for
		// normal operation
		bool  isValid();

};

#endif // !_CUALG_

/*------------------------------*/
/*   CUDA ALGORITHM BASE CLASS  */
/*       IMPLEMENTATION         */
/*------------------------------*/
#include <vector>
#include <thrust/device_vector.h>
#include "Globals.cuh"
#include "GlobalDeclarations.cuh"
#include "FileManager.cuh"
#include "HostMemory.cuh"
#include "DeviceMemory.cuh"
#include "CudaAlgorithm.cuh"
#include "curand.h"
#include "cusolverDn.h"
#include "ErrorHandler.cuh"

using namespace std;


/*---------------*/
```

```
/* Constructor    */
/*---------------*/
CudaAlgorithm::CudaAlgorithm(DeviceMemory<datatype>* devptr, HostMemory<datatype>* 
hostptr)
      : deviceMemory(*devptr), hostMemory(*hostptr) {

      valid = true;
      // Create handle
      cublasStatus_t cubstat;
      if ((cubstat = cublasCreate(&(this->CBhandle))) != CUBLAS_STATUS_SUCCESS) {
            cerr << "Cublas Create failed: " << 
ErrorHandler::cublasGetErrorString(cubstat) << endl;
            valid = false;
      }
}

/*--------------*/
/* Destructor   */
/*--------------*/
CudaAlgorithm::~CudaAlgorithm() {
      // Destroy handle
      cublasStatus_t cubstat;
      if ((cubstat = cublasDestroy(this->CBhandle)) != CUBLAS_STATUS_SUCCESS) {
            cerr << "Cublas Destroy failed: " << 
ErrorHandler::cublasGetErrorString(cubstat) << endl;
      }
 }

/*-------------------------*/
/* Check if error occured   */
/*-------------------------*/
bool CudaAlgorithm::isValid() {
      return(valid);

}

/*---------------------------*/
/* ALL HOST MEMORY OPERATIONS */
/*           (HIGH LEVEL)         */
/*---------------------------*/
#pragma once
#ifndef _HOSTMEM_
#define _HOSTMEM_


// Class wrapper for memory allocations
template < typename T>
class HostMemory {

      private:
            // Private members
            std::vector< T* >     array;
            unsigned int*         rand_buffer;

      public:

            // Constructor/Destructor
            HostMemory(FileManager&);
            ~HostMemory();

            // Public Methods
            T*          get(unsigned int);
            unsigned
            int*  get_RND_GNRT_buffer();
```

```
		// Public Types
		enum {
			X_ARRAY = 0,
			D_ARRAY,
			MAX_ELEMENTS = D_ARRAY
		};

};

// Include the implementation because
// separate templated methods won't link
#include "HostMemory.cu"


#endif // !_HOSTMEM_

/*------------------------------*/
/* ALL HOST MEMORY OPERATIONS   */
/*       IMPLEMENTATION         */
/*           (HIGH LEVEL)             */
/*------------------------------*/
#include <iostream>

using namespace std;


/*-------------*/
/* Constructor */
/*-------------*/
template < typename T>
HostMemory<T>::HostMemory(FileManager& fm) : array(HostMemory<T>::MAX_ELEMENTS + 1, 0) {
	/* Arrays in vector */
	if (fm.readXarray(&this->array[X_ARRAY]) == false) {
		cerr << "Cannot read file for X! Aborting..." << endl;
		return;
	}
	if (fm.readDarray(&this->array[D_ARRAY]) == false) {
		cerr << "Cannot read file for D! Aborting..." << endl;
		return;
	}
	/* End of arrays in vector */

	if ((this->rand_buffer = new(nothrow) unsigned int[Globals::colsD]) == NULL) {
		cerr << "Allocation for rand_gen_buffer failed!..." << endl;
		return;
	}
}

/*------------*/
/* Destructor */
/*------------*/
template < typename T>
HostMemory<T>::~HostMemory() {
	this->array[X_ARRAY] = NULL;
	this->array[D_ARRAY] = NULL;
	for (int i = 0; i <= HostMemory<T>::MAX_ELEMENTS; i++) {
		if (this->array[i]){
			delete[] this->array[i];
		}
	}
	if (this->rand_buffer) {
		delete[] this->rand_buffer;
	}
	cout << "# Host Memory Cleared!" << endl;
}
```

```
/*----------------*/
/* Public Methods */
/*----------------*/
// Return an array pointer
template < typename T> inline
T* HostMemory<T>::get(unsigned int index) {
      return this->array.at(index);
}

//Return the random generator buffer
template < typename T> inline
unsigned int* HostMemory<T>::get_RND_GNRT_buffer() {
      return this->rand_buffer;

}

/*-----------------------------*/
/* ALL DEVICE MEMORY OPERATIONS */
/*           (HIGH LEVEL)             */
/*-----------------------------*/
#pragma once
#ifndef _DEVMEM_
#define _DEVMEM_


// Class wrapper for device memory
// allocations
template < typename T>
class DeviceMemory {

      private:
            // Private members
            std::vector< T* >                                   POINTERarray;
            std::vector< thrust::device_vector<T> >          THRUSTarray;
            double                                                    total, free, used,
consumed;

      public:

            // Constructor/Destructor
            DeviceMemory(HostMemory<datatype>&);
            ~DeviceMemory();

            // Public Methods
            T*                                      get(unsigned int);
            thrust::device_vector<T>&   getThrust(unsigned int);
            double                                  getTotal();
            double                                  getFree();
            double                                  getUsed();
            double                                  getConsumed();

            // Public Types
            enum {
                  X_ARRAY = 0,
                  /*_______________*/
                  /*  More arrays  */
                  /*_______________*/
                  // TEMPORARY
                  TEMP_ROWSD_BY_COLSD,
                  TEMP_1_BY_COLSD,
                  TEMP_SINGLE_VALUE,
                  // CONSTANT
                  REDUCTION_BUFFER,
                  // INTERMEDIATE RESULTS - batch.OMP
```

```
                 G,DtX,SELECTED_ATOMS,
                 c,alpha,
                 // INTERMEDIATE RESULTS - Dictionary Update
                 TEMP_COLSD_BY_ROWSX,
                 UNUSED_SIGS,
                 UNUSED_SIGS_COUNTER,
                 UNUSED_SIGS_BITMAP,
                 REPLACED_ATOMS,
                 // INTERMEDIATE RESULTS - Error computation
                 TEMP_ROWSX_BY_COLSX,
                 ERR,
                 /*_______________*/
                 /* End of arrays */
                 /*_______________*/
                 D_ARRAY,
                 MAX_ELEMENTS = D_ARRAY
           };

};

// Include the implementation because
// separate templated methods won't link
#include "DeviceMemory.cu"


#endif // !_DEVMEM_

/*------------------------------*/
/* ALL DEVICE MEMORY OPERATIONS */
/*       IMPLEMENTATION         */
/*           (HIGH LEVEL)             */
/*------------------------------*/
#include <iostream>


/****************
*   NAMESPACES    *
*     __________    *
****************/
using namespace       std;
using HostSpace       = HostMemory<datatype>;

/*-------------*/
/* Constructor */
/*-------------*/
template < typename T>
DeviceMemory<T>::DeviceMemory(HostSpace& hostMemory)
      : POINTERarray(DeviceMemory<T>::MAX_ELEMENTS + 1, 0),
THRUSTarray(DeviceMemory<T>::MAX_ELEMENTS + 1) {

      // Set up memory metrics first!
      // Total used memory
      this->consumed =
            (Globals::rowsX * Globals::colsX) +
            (Globals::rowsD * Globals::colsD) +
            (Globals::colsD) +
            (Globals::colsX) +
            (Globals::rowsD * Globals::colsD * 1024) +
            (Globals::colsD * Globals::colsD) +
            (Globals::colsD * Globals::colsX) +
            (Globals::colsD * Globals::colsX) +
            (Tdata * Globals::colsX) +
            (Globals::colsD * Globals::colsX) +
            (Globals::colsD * Globals::rowsX) +
            (Globals::rowsD * Globals::colsD) +
```

```
           (Globals::rowsX * Globals::colsX) +
           (Globals::colsX) +
           (Globals::colsD) +
           (1) +
           (Globals::colsX);

     this->consumed = this->consumed * sizeof(datatype);
     // Total available global memory
     size_t free_byte;
     size_t total_byte;
     cudaMemGetInfo(&free_byte, &total_byte);
     this->free = (double)free_byte;
     this->total = (double)total_byte;
     this->used = this->total - this->free;

     // Now the arrays!
     // X_ARRAY (rowsX * colsX)
     this->THRUSTarray[X_ARRAY] = thrust::device_vector<T>(
           hostMemory.get(HostSpace::X_ARRAY),
           hostMemory.get(HostSpace::X_ARRAY) + Globals::rowsX*Globals::colsX);
     this->POINTERarray[X_ARRAY] = thrust::raw_pointer_cast(&(this-
>THRUSTarray[X_ARRAY])[0]);
     /*_______________*/
     /*  More arrays  */
     /*_______________*/
     // TEMP_ROWSD_BY_COLSD
     this->THRUSTarray[TEMP_ROWSD_BY_COLSD] =
thrust::device_vector<T>(Globals::rowsD*Globals::colsD);
     this->POINTERarray[TEMP_ROWSD_BY_COLSD] = thrust::raw_pointer_cast(
           &(this->THRUSTarray[TEMP_ROWSD_BY_COLSD])[0]);
     // TEMP_1_BY_COLSD
     this->THRUSTarray[TEMP_1_BY_COLSD] = thrust::device_vector<T>(Globals::colsD);
     this->POINTERarray[TEMP_1_BY_COLSD] = thrust::raw_pointer_cast(
           &(this->THRUSTarray[TEMP_1_BY_COLSD])[0]);
     // TEMP_SINGLE_VALUE ( 1 !by colsX threads )
     this->THRUSTarray[TEMP_SINGLE_VALUE] = thrust::device_vector<T>(Globals::colsX);
     this->POINTERarray[TEMP_SINGLE_VALUE] = thrust::raw_pointer_cast(
           &(this->THRUSTarray[TEMP_SINGLE_VALUE])[0]);
     // REDUCTION_BUFFER ( rowsD * colsD * 1024 )
     // The maximum buffer needed! ( in collincomb )
     this->THRUSTarray[REDUCTION_BUFFER] =
thrust::device_vector<T>(Globals::colsD*Globals::rowsD*1024);
     this->POINTERarray[REDUCTION_BUFFER] = thrust::raw_pointer_cast(
           &(this->THRUSTarray[REDUCTION_BUFFER])[0]);
     // G ( colsD * colsD)
     this->THRUSTarray[G] = thrust::device_vector<T>(Globals::colsD*Globals::colsD);
     this->POINTERarray[G] = thrust::raw_pointer_cast(
           &(this->THRUSTarray[G])[0]);
     // DtX ( colsD * colsX)
     this->THRUSTarray[DtX] = thrust::device_vector<T>(Globals::colsD*Globals::colsX);
     this->POINTERarray[DtX] = thrust::raw_pointer_cast(
           &(this->THRUSTarray[DtX])[0]);
     cudaMemset(this->POINTERarray[DtX], 0, Globals::colsD*Globals::colsX);
     // SELECTED_ATOMS ( colsD !by colsX threads )
     this->THRUSTarray[SELECTED_ATOMS] =
thrust::device_vector<T>(Globals::colsD*Globals::colsX);
     this->POINTERarray[SELECTED_ATOMS] = thrust::raw_pointer_cast(
           &(this->THRUSTarray[SELECTED_ATOMS])[0]);
     cudaMemset(this->POINTERarray[SELECTED_ATOMS], 0, Globals::colsD*Globals::colsX);
     // c ( Tdata !by colsX threads )
     this->THRUSTarray[c] = thrust::device_vector<T>(Tdata*Globals::colsX);
     this->POINTERarray[c] = thrust::raw_pointer_cast(
           &(this->THRUSTarray[c])[0]);
     cudaMemset(this->POINTERarray[c], 0, Tdata*Globals::colsX);
     // alpha ( colsD !by colsX threads )
```

```
	this->THRUSTarray[alpha] = thrust::device_vector<T>(Globals::colsD*Globals::colsX);
	this->POINTERarray[alpha] = thrust::raw_pointer_cast(
		&(this->THRUSTarray[alpha])[0]);
	// TEMP_COLSD_BY_ROWSX ( colsD * rowsX )
	this->THRUSTarray[TEMP_COLSD_BY_ROWSX] =
thrust::device_vector<T>(Globals::colsD*Globals::rowsX);
	this->POINTERarray[TEMP_COLSD_BY_ROWSX] = thrust::raw_pointer_cast(
		&(this->THRUSTarray[TEMP_COLSD_BY_ROWSX])[0]);
	// TEMP_ROWSX_BY_COLSX ( rowsX * colsX )
	this->THRUSTarray[TEMP_ROWSX_BY_COLSX] =
thrust::device_vector<T>(Globals::rowsX*Globals::colsX);
	this->POINTERarray[TEMP_ROWSX_BY_COLSX] = thrust::raw_pointer_cast(
		&(this->THRUSTarray[TEMP_ROWSX_BY_COLSX])[0]);
	// ERR ( 1 * colsX )
	this->THRUSTarray[ERR] = thrust::device_vector<T>(Globals::colsX);
	this->POINTERarray[ERR] = thrust::raw_pointer_cast(
		&(this->THRUSTarray[ERR])[0]);
	// UNUSED_SIGS ( 1 * colsX )
	this->THRUSTarray[UNUSED_SIGS] = thrust::device_vector<T>(Globals::colsX);
	this->POINTERarray[UNUSED_SIGS] = thrust::raw_pointer_cast(
		&(this->THRUSTarray[UNUSED_SIGS])[0]);
	// REPLACED_ATOMS ( 1 * colsD)
	this->THRUSTarray[REPLACED_ATOMS] = thrust::device_vector<T>(Globals::colsD);
	this->POINTERarray[REPLACED_ATOMS] = thrust::raw_pointer_cast(
		&(this->THRUSTarray[REPLACED_ATOMS])[0]);
	// UNUSED_SIGS_COUNTER ( 1 )
	this->THRUSTarray[UNUSED_SIGS_COUNTER] = thrust::device_vector<T>(1);
	this->POINTERarray[UNUSED_SIGS_COUNTER] = thrust::raw_pointer_cast(
		&(this->THRUSTarray[UNUSED_SIGS_COUNTER])[0]);
	// UNUSED_SIGS_BITMAP ( 1 * colsX )
	this->THRUSTarray[UNUSED_SIGS_BITMAP] = thrust::device_vector<T>(Globals::colsX);
	this->POINTERarray[UNUSED_SIGS_BITMAP] = thrust::raw_pointer_cast(
		&(this->THRUSTarray[UNUSED_SIGS_BITMAP])[0]);
	/*_______________*/
	/* End of arrays */
	/*_______________*/
	// D_ARRAY
	this->THRUSTarray[D_ARRAY] = thrust::device_vector<T>(
		hostMemory.get(HostSpace::D_ARRAY),
		hostMemory.get(HostSpace::D_ARRAY) + Globals::rowsD*Globals::colsD);
	this->POINTERarray[D_ARRAY] = thrust::raw_pointer_cast(&(this-
>THRUSTarray[D_ARRAY])[0]);
}

/*------------*/
/* Destructor */
/*------------*/
template < typename T>
DeviceMemory<T>::~DeviceMemory() {
	cout << "# Device Memory Cleared!" << endl;
}

/*----------------*/
/* Public Methods */
/*----------------*/

// Return an array pointer
template < typename T> inline
T* DeviceMemory<T>::get(unsigned int index) {
	return this->POINTERarray.at(index);
}

// Return a thrust device array pointer
template < typename T> inline
thrust::device_vector<T>&
```

```
DeviceMemory<T>::getThrust(unsigned int index) {
	return this->THRUSTarray.at(index);
}

// Return total available device memory
template < typename T> inline
double DeviceMemory<T>::getTotal() {
	return this->total;
}

// Return total memory used by the OS
template < typename T> inline
double DeviceMemory<T>::getUsed() {
	return this->used;
}

// Return free global memory
template < typename T> inline
double DeviceMemory<T>::getFree() {
	return this->free;
}

// Return total memory used by our program
template < typename T> inline
double DeviceMemory<T>::getConsumed() {
	return this->consumed;

}

/*------------------------------*/
/*  THRUST OPERATORS NAMESPACE  */
/*------------------------------*/
#pragma once
#ifndef _THR_
#define _THR_


// Methods used as unary functions
// in Thrust family calls
namespace ThrustOperators {

	/*----------------*/
	/* Public Methods */
	/*----------------*/
	// Unary functions
	template < typename T >
	struct Exp;
	template < typename T >
	struct Inv;

};

// Include the implementation because
// separate templated methods won't link
#include "ThrustOperators.cu"


#endif // !_THR_

/*------------------------------*/
/*  THRUST OPERATORS NAMESPACE  */
/*       IMPLEMENTATION         */
/*------------------------------*/

/*========================================================================*/
/*================================= BEGIN ================================*/
```

```
/*=========================================================================*/

/*--------------------------------*/
/* Unary function for Thrust that */
/* calculates the exponential of  */
/* each element.                  */
/*--------------------------------*/
template < typename T >
struct ThrustOperators::Exp : public thrust::unary_function<T, T>
{
	__host__ __device__ T operator()(T x)
	{
		return x*x;
	}
};

/*--------------------------------*/
/* Unary function for Thrust that */
/* calculates the inverse of the  */
/* square root of each element.   */
/*--------------------------------*/
template < typename T >
struct ThrustOperators::Inv : public thrust::unary_function<T, T>
{
	__host__ __device__ T operator()(T x)
	{
		return (T) 1.0 / sqrt(x);
	}
};

/*=========================================================================*/
/*=================================  END  =================================*/

/*=========================================================================*/
```

3. Αλγόριθμος PARAFAC2

```
//////////////////////////////////////////////
//             File: main.cu                //
//////////////////////////////////////////////
#include <iostream>
#include <windows.h>
#include "declarations.h"
#include "customvector.h"
#include "read.h"
#include "timer.h"
#include "parafac.h"
#include "memalloc.h"

using namespace std;

extern CustomVector *X;

int main(void) {
	int maximum_variable_dimension;

	cout << "Parafac2 algorithm" << endl;
	cout << "Convergence criterion : " << conv_crit << endl;
	cout << "Maximal number of iterations : " << max_iter << endl;
	cout << "Number of sources : " << sources << endl;

	cout << "Reading input from file" << endl;
	cout << "Please, wait a moment ..." << endl;
	/* Reading input from file */
	{
		Timer t;
```

```
		if (!t.isValid()) {
			cerr << "Timer initialization error! Aborting..." << endl;
			return -1;
		}
		t.tic();
		if (readingInputFromFile(&X,&maximum_variable_dimension) != 0) {
			cout << "An error occurred on reading input" << endl;
			return -1;
		}
		cout << "Reading time : " << t.toc() << " sec" << endl;
	}
	/* Allocate all the necessary memory */
	{
		Timer t;
		if (!t.isValid()) {
			cerr << "Timer initialization error! Aborting..." << endl;
			return -1;
		}
		t.GPUtic();
		allocate(maximum_variable_dimension);
		cout << "Time to allocate all the necessary memory " << t.GPUtoc() / 1000.0 <<
" sec" << endl;
	}
	/* Executing algorithm parafac2 */
	{
		Timer t;
		if (!t.isValid()) {
			cerr << "Timer initialization error! Aborting..." << endl;
			return -1;
		}
		t.GPUtic();
		parafac2();
		cout << "Parafac2 time : " << t.GPUtoc() / 1000.0 << " sec" << endl;
	}
	/* Cleaning all of the previously allocated memory */
	cleanup();

	return 0;
}

///////////////////////////////////////////////
//            File: Timer.h                 //
///////////////////////////////////////////////
#ifndef _TIMER_
#define _TIMER_


// Functions used for calculating
// elapsed time on CPU or GPU
class Timer {

	private:
		// Private Members
		cudaEvent_t GPUstart, GPUstop;
		LARGE_INTEGER CPUstart, CPUstop, frequency;

	public:
		// Constructor
		Timer();
		~Timer();

		// Public Methods
		void tic();
		double	toc();
		double	tocmilli();
```

```
		void  GPUtic();
		float GPUtoc();
		bool  isValid();

};

#endif // !_TIMER_

/////////////////////////////////////////////////
//              File: Timer.cu                 //
/////////////////////////////////////////////////
#include <iostream>
#include <windows.h>
#include "declarations.h"
#include "timer.h"

using namespace std;


/*---------------*/
/* Constructor   */
/*---------------*/
Timer::Timer() {
	this->GPUstart = NULL;
	this->GPUstop = NULL;
	cudaError_t cuda_error;
	if ((cuda_error = cudaEventCreate(&this->GPUstart)) != cudaSuccess) {
		cerr << "Cuda event create for GPUstart failed: " <<
cudaGetErrorString(cuda_error) << endl;
	}
	if ((cuda_error = cudaEventCreate(&this->GPUstop)) != cudaSuccess) {
		cerr << "Cuda event create for GPUstop failed: " <<
cudaGetErrorString(cuda_error) << endl;
	}
}


/*--------------*/
/* Destructor   */
/*--------------*/
Timer::~Timer() {
	if (this->GPUstart) {
		cudaEventDestroy(this->GPUstart);
	}
	if (this->GPUstop) {
		cudaEventDestroy(this->GPUstop);
	}
}


/*--------------------------*/
/* Check if timer is valid  */
/*--------------------------*/
bool Timer::isValid() {
	return (this->GPUstart != NULL && this->GPUstop != NULL);
}


/*------------------*/
/* Start CPU timer  */
/*------------------*/
void Timer::tic() {
	QueryPerformanceFrequency(&this->frequency);
	QueryPerformanceCounter(&this->CPUstart);
}
```

```
/*--------------------*/
/* Stop CPU timer and */
/* return in sec!     */
/*--------------------*/
double Timer::toc() {
	QueryPerformanceCounter(&this->CPUstop);
	return ((double)(this->CPUstop.QuadPart - this->CPUstart.QuadPart) / this-
>frequency.QuadPart);
}


/*--------------------*/
/* Stop CPU timer and */
/* return in ms!      */
/*--------------------*/
double Timer::tocmilli() {
	QueryPerformanceCounter(&this->CPUstop);
	return ((double)(this->CPUstop.QuadPart - this->CPUstart.QuadPart) / this-
>frequency.QuadPart * 1000.0);
}


/*-------------------*/
/* Start GPU timer   */
/*-------------------*/
void Timer::GPUtic() {
	cudaEventRecord(this->GPUstart);
}


/*--------------------*/
/* Stop GPU timer and */
/* return in ms!      */
/*--------------------*/
float Timer::GPUtoc() {
	cudaEventRecord(this->GPUstop);
	cudaEventSynchronize(this->GPUstop);
	float milliseconds = 0;
	cudaEventElapsedTime(&milliseconds, this->GPUstart, this->GPUstop);
	return milliseconds;
}
////////////////////////////////////////////////
//               File: rest.h                 //
////////////////////////////////////////////////
#ifndef __rest__
#define __rest__

void rand(curandGenerator_t, datatype *, int);
void gpu_blas_mmul (cublasHandle_t, const datatype *, const datatype *, datatype *, int,
const int, const int, const int, const int);
void SVD(cusolverDnHandle_t, datatype *, datatype *, datatype *, datatype *, int *, int
);
void psqrt(cusolverDnHandle_t, cublasHandle_t, datatype *, datatype *, datatype *,
datatype *, datatype *, datatype *, int *);
void pinv(cusolverDnHandle_t, cublasHandle_t, datatype*, datatype* , datatype* ,
datatype*, datatype*, datatype*, int*, int);
datatype pf2fit (cublasHandle_t, CustomVector*, CustomVector *, datatype *, datatype *,
datatype *, CustomVector *, CustomVector *, CustomVector *, datatype *, datatype *,
datatype *, CustomVector *);
void QRonly(cusolverDnHandle_t, cublasHandle_t, datatype*, datatype*, datatype *,
datatype *, int, int, int*);
```

```
void QRsolve(cusolverDnHandle_t, cublasHandle_t, datatype*, datatype*, datatype*,
datatype*, int, int, CustomVector*, int,int,CustomVector*, int*);
void QRdecomposition(cusolverDnHandle_t, cublasHandle_t, datatype*, datatype*, datatype*,
int, int,datatype**,int*, int*);

#endif // !__rest__

//////////////////////////////////////////////
//              File: rest.cu               //
//////////////////////////////////////////////
#include <iostream>
#include <cusolverDn.h>
#include <curand.h>
#include "cublas_v2.h"
#include "declarations.h"
#include "customvector.h"
#include "reductions.h"
#include "memalloc.h"
#include "kernel.h"
#include "errorTypes.h"

/*    Generate numbers using uniform distribution
      (http://docs.nvidia.com/cuda/curand/host-api-overview.html#axzz4SrFExYZh) */
void rand(curandGenerator_t generator, datatype *Arr, int size) {
      curandStatus_t stat;
      #if (MODE)
            if ((stat = curandGenerateUniformDouble(generator, Arr, size)) !=
CURAND_STATUS_SUCCESS) {
                  printf("curandGenerateUniformDouble failed: %s\n",
curandGetErrorString(stat));
                  cleanup();
                  exit(-1);
            }
      #else
            if ((stat = curandGenerateUniform(generator, Arr, size)) !=
CURAND_STATUS_SUCCESS) {
                  printf("curandGenerateUniform failed: %s\n", curandGetErrorString(stat));
                  cleanup();
                  exit(-1);
            }
      #endif
}



/* Matrix Multiplication (category=0 : C=A*B, category=1 : C=A'*B, category 2 : C=A*B')
*/
void gpu_blas_mmul(cublasHandle_t handle, const datatype *A, const datatype *B, datatype
*C,
      int category, const int rowA, const int colA, const int rowB, const int colB) {

      int lda, ldb, ldc;
      const datatype alf = 1;
      const datatype bet = 0;
      const datatype *alpha = &alf;
      const datatype *beta = &bet;
      cublasStatus_t status;
      // http://peterwittek.com/cublas-matrix-c-style.html
      if (category == 0) { // C=A*B
            lda = colB; ldb = colA; ldc = colB;
            if (colA != rowB) {
                  printf("colA != rowB. Cannot multiply the matrices\n");
                  cleanup();
                  exit(-1);
            }
```

```
		#if (MODE)
			status = cublasDgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, colB, rowA, colA, alpha, B, lda, A, ldb, beta, C, ldc);
		#else
			status = cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, colB, rowA, colA, alpha, B, lda, A, ldb, beta, C, ldc);
		#endif
	}
	// http://stackoverflow.com/questions/14595750/transpose-matrix-multiplication-in-cublas-howto
	else if (category == 1) { // C = A'*B
		lda = colA; ldb = colB; ldc = colB;
		if (rowA != rowB) {
			printf("rowA != rowB. Cannot multiply the matrices\n");
			cleanup();
			exit(-1);
		}
		#if (MODE)
			status = cublasDgemm(handle, CUBLAS_OP_N, CUBLAS_OP_T, colB, colA, rowB, alpha, B, ldb, A, lda, beta, C, ldc);
		#else
			status = cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_T, colB, colA, rowB, alpha, B, ldb, A, lda, beta, C, ldc);
		#endif
	}
	else if (category == 2) { // C=A*B'
		lda = colA; ldb = colB; ldc = rowB;
		if (colA != colB) {
			printf("colA != colB. Cannot multiply the matrices\n");
			cleanup();
			exit(-1);
		}
		#if (MODE)
			status = cublasDgemm(handle, CUBLAS_OP_T, CUBLAS_OP_N, rowB, rowA, colB, alpha, B, ldb, A, lda, beta, C, ldc);
		#else
			status = cublasSgemm(handle, CUBLAS_OP_T, CUBLAS_OP_N, rowB, rowA, colB, alpha, B, ldb, A, lda, beta, C, ldc);
		#endif
	}
	else {
		printf("gpu_blas_mmul : not valid status %d \n", category);
	}
	if (status != cudaSuccess) {
		printf("Error in gpu_blas_mmul : %s\n", cublasGetErrorString(status));
		cleanup();
		exit(-1);
	}
}

/* Computes the Singular Value Decomposition of the given array (computes U, S, VT)*/
void SVD(cusolverDnHandle_t cusolverH, datatype *QkT_Qk, datatype *U, datatype *S, datatype *VT, int *devInfo, int size) {
	int lwork = 0;
	cudaError_t err;
	cusolverStatus_t stat;
	datatype *d_work = 0, *d_rwork = 0;
	const int m = size, n = size, lda = m;
	unsigned char jobu = 'A', jobvt = 'A';
	#if (MODE)
		if ((stat = cusolverDnDgesvd_bufferSize(cusolverH, m, n, &lwork)) != CUSOLVER_STATUS_SUCCESS) {
			printf("cusolverDnDgesvd_bufferSize failed: %s\n", cusolverGetErrorString(stat));
			cleanup();
```

```
                exit(-1);
            }
      #else
            if ((stat = cusolverDnSgesvd_bufferSize(cusolverH, m, n, &lwork)) !=
CUSOLVER_STATUS_SUCCESS) {
                printf("cusolverDnDgesvd_bufferSize failed: %s\n",
cusolverGetErrorString(stat));
                cleanup();
                exit(-1);
            }
      #endif
      if ((err = cudaMalloc((void**)&d_work, lwork * sizeof(datatype))) != cudaSuccess) {
            printf("cudaMalloc d_work failed: %s\n", cudaGetErrorString(err));
            cleanup();
            exit(-1);
      }
      #if (MODE)
            if ((stat = cusolverDnDgesvd(cusolverH, jobu, jobvt, m, n, QkT_Qk, lda, S, U,
lda, VT, lda, d_work, lwork, d_rwork, devInfo)) != CUSOLVER_STATUS_SUCCESS) {
                printf("cusolverDnDgesvd failed : %s\n", cusolverGetErrorString(stat));
                cleanup();
                exit(-1);
            }
      #else
            if ((stat = cusolverDnSgesvd(cusolverH, jobu, jobvt, m, n, QkT_Qk, lda, S, U,
lda, VT, lda, d_work, lwork, d_rwork, devInfo)) != CUSOLVER_STATUS_SUCCESS) {
                printf("cusolverDnSgesvd failed: %s\n", cusolversGetErrorString(stat));
                cleanup();
                exit(-1);
            }
      #endif
      if ((err = cudaFree(d_work)) != cudaSuccess) {
            printf("cudaFree d_work(SVD) failed: %s\n", cudaGetErrorString(err));
      }
}


void psqrt(cusolverDnHandle_t cusolverH, cublasHandle_t handle, datatype *QkT_Qk,
datatype *U, datatype *S, datatype *VT, datatype *Sdiag, datatype *VS, int *devInfo) {
      bool r_condition;
      cudaError_t cudaError;
      SVD(cusolverH, QkT_Qk, U, S, VT, devInfo, sources);
      filterColumns << < 1, sources >> > (S);
      get_r(&r_condition);
      if (!r_condition) {
            if ((cudaError = cudaMemset(QkT_Qk, 0, sources*sources * sizeof(datatype))) !=
cudaSuccess) {
                printf("cudaMemset QkT_Qk failed: %s\n", cudaGetErrorString(cudaError));
                cleanup();
                exit(-1);
            }
      }
      else {
            diagonalize << <1, sources >> >(S, Sdiag, sources, 0);
            gpu_blas_mmul(handle, VT, Sdiag, VS, 0, sources, sources, sources, sources);
            gpu_blas_mmul(handle, VS, U, QkT_Qk, 0, sources, sources, sources, sources);
      }
}

/*    Computes the pseudoinverse of the given array
      (https://www.mathworks.com/help/matlab/ref/pinv.html)  */
void pinv(cusolverDnHandle_t cusolverH, cublasHandle_t handle, datatype *A, datatype *U,
      datatype *S, datatype *VT, datatype *Sdiag, datatype *Temp, int *devInfo, int sizeA)
{
      SVD(cusolverH, A, U, S, VT, devInfo, sizeA);
```

```
	pinvDiagonalMatrix << <1, sizeA >> >(S, Sdiag, sizeA, 0);
	gpu_blas_mmul(handle, VT, Sdiag, Temp, 0, sizeA, sizeA, sizeA, sizeA);
	gpu_blas_mmul(handle, Temp, U, A, 0, sizeA, sizeA, sizeA, sizeA);
}

/* Computes the fit of an execution */
datatype pf2fit(cublasHandle_t handle, CustomVector *CuX, CustomVector *X, datatype *A,
datatype *H, datatype *C,
	CustomVector *P, CustomVector *M, CustomVector *cuM, datatype *diagC, datatype
*AdotdiagC,
	datatype *AdiagCdotH, CustomVector * fs) {
	int i;
	datatype fit = 0.0;
	for (i = 0; i < minSize; i++) {
		diagonalize << <1, sources >> >(C, diagC, sources, i);
		gpu_blas_mmul(handle, A, diagC, AdotdiagC, 0, sizeX, sources, sources,
sources);
		gpu_blas_mmul(handle, AdotdiagC, H, AdiagCdotH, 2, sizeX, sources, sources,
sources);
		gpu_blas_mmul(handle, AdiagCdotH, P[i].array, M[i].array, 2, sizeX, sources,
M[i].size, sources);
	}
	dim3 block(1024);
	dim3 grid(fs->size / minSize, minSize);
	pf2fitSquareSum_STAGE1 << <grid, block >> > (CuX, cuM, fs->array);
	grid.y = 1;
	int N = fs->size;
	grid.x = (N + block.x - 1) / block.x;
	pf2fitSquareSum_STAGE2 << <grid, block >> > (N, fs->array);
	while (grid.x > 1) {
		N = grid.x;
		grid.x = (N + block.x - 1) / block.x;
		pf2fitSquareSum_STAGE2 << <grid, block >> > (N, fs->array);
	}
	get_fit(&fit);
	return fit;
}


/* Implements QR decomposition of the given array */
void QRdecomposition(cusolverDnHandle_t cusH, cublasHandle_t cubH, datatype *A,
	datatype *Xbc, datatype *d_tau, int rowsA, int colsA, datatype **d_work_ptr,
	int* l_ptr, int *devInfo) {
	int  lwork = 0;
	datatype *d_work;
	cudaError_t err;
	cublasStatus_t cubErr;
	cusolverStatus_t cusErr;
	const double a = 1.0;
	const double b = 0.0;
	#if (MODE)
		if ((cubErr = cublasDgeam(cubH, CUBLAS_OP_T, CUBLAS_OP_N, rowsA, colsA, &a, A,
colsA, &b, A, rowsA, Xbc, rowsA)) != cudaSuccess) {
			printf("cublasDgeam failed: %s\n", cublasGetErrorString(cubErr));
			cleanup();
			exit(-1);
		}
		if ((cusErr = cusolverDnDgeqrf_bufferSize(cusH, rowsA, colsA, Xbc, rowsA,
&lwork)) != CUSOLVER_STATUS_SUCCESS) {
			printf("cusolverDnDgeqrf_bufferSize failed: %s\n",
cusolverGetErrorString(cusErr));
			cleanup();
			exit(-1);
		}
```

```
	#else
		if ((cubErr = cublasSgeam(cubH, CUBLAS_OP_T, CUBLAS_OP_N, rowsA, colsA, &a, A,
colsA, &b, A, rowsA, Xbc, rowsA)) != cudaSuccess) {
			printf("cublasSgeam failed: %s\n", cublasGetErrorString(cubErr));
			cleanup();
			exit(-1);
		}
		if ((cusErr = cusolverDnSgeqrf_bufferSize(cusH, rowsA, colsA, Xbc, rowsA,
&lwork)) != CUSOLVER_STATUS_SUCCESS) {
			printf("cusolverDnSgeqrf_bufferSize failed: %s\n",
cusolverGetErrorString(cusErr));
			cleanup();
			exit(-1);
		}

	#endif
	if ((err = cudaMalloc((void**)&d_work, lwork * sizeof(datatype))) != cudaSuccess) {
		printf("cudaMalloc d_work failed: %s\n", cudaGetErrorString(err));
		cleanup();
		exit(-1);
	}
	#if (MODE)
		if ((cusErr = cusolverDnDgeqrf(cusH, rowsA, colsA, Xbc, rowsA, d_tau, d_work,
lwork, devInfo)) != CUSOLVER_STATUS_SUCCESS) {
			printf("cusolverDnDgeqrf failed: %s\n", cusolverGetErrorString(cusErr));
			cleanup();
			exit(-1);
		}
	#else
		if ((cusErr = cusolverDnSgeqrf(cusH, rowsA, colsA, Xbc, rowsA, d_tau, d_work,
lwork, devInfo)) != CUSOLVER_STATUS_SUCCESS) {
			printf("cusolverDnSgeqrf failed: %s\n", cusolverGetErrorString(cusErr));
			cleanup();
			exit(-1);
		}
	#endif
	*d_work_ptr = d_work;
	*l_ptr = lwork;
}

/*	Project X down on orth(A) - saves time if first mode is large
	[Qa, Ra] = qr(A, 0);  x = Qa'*X; */
void QRsolve(cusolverDnHandle_t cusH, cublasHandle_t cubH, datatype *A, datatype *Xbc,
	datatype *d_tau, datatype *R, int rowsA, int colsA, CustomVector *Y,
	int rowsY, int colsY, CustomVector* temp_buffer,int *devInfo) {
	int lwork;
	dim3 grid(rowsA / 32 + 1, colsA / 32 + 1);
	dim3 threads(32, 32);
	datatype* d_work;
	cusolverStatus_t cusErr;
	cudaError_t	cet;
	QRdecomposition(cusH, cubH, A, Xbc, d_tau, rowsA, colsA, &d_work, &lwork, devInfo);
	for (size_t i = 0; i < minSize; i++) {
		//compute Q^T*Y === Xbc*Y[i].array
		if ((cet = cudaMemcpy(temp_buffer[i].array, Y[i].array, rowsY * colsY *
sizeof(datatype), cudaMemcpyDeviceToDevice)) != cudaSuccess) {
			printf("cudaMemcpy temp_buffer failed: %s\n", cudaGetErrorString(cet));
			cleanup();
			exit(-1);
		}
		#if (MODE)
			cusErr = cusolverDnDormqr(cusH, CUBLAS_SIDE_RIGHT, CUBLAS_OP_N, colsY,
				rowsY, colsA, Xbc, rowsA, d_tau, temp_buffer[i].array, colsY,
d_work,
				lwork, devInfo);
```

```
			if (cusErr != CUSOLVER_STATUS_SUCCESS) {
				printf("cusolverDnDormqr failed: %s\n",
cusolverGetErrorString(cusErr));
				cudaFree(d_work);
				cleanup();
				exit(-1);
			}
		#else
			cusErr = cusolverDnSormqr(cusH, CUBLAS_SIDE_RIGHT, CUBLAS_OP_N, colsY,
rowsY,
				colsA, Xbc, rowsA, d_tau, Y[i].array, colsY, d_work, lwork,
devInfo);
			if (cusErr == CUSOLVER_STATUS_SUCCESS) {
				printf("cusolverDnSormqr failed: %s\n",
cusolverGetErrorString(cusErr));
				cudaFree(d_work);
				cleanup();
				exit(-1);
			}
		#endif
	}
	stripR << <grid, threads >> >(Xbc, R, colsA, rowsA, colsA);
	cudaFree(d_work);
}

/* Computes only the R array og QR decomposition */
void QRonly(cusolverDnHandle_t cusH, cublasHandle_t cubH, datatype *A, datatype *Xbc,
	datatype *d_tau, datatype *R, int rowsA, int colsA, int *devInfo) {

	datatype* d_work;
	int lwork;
	dim3 grid(rowsA / 32 + 1, colsA / 32 + 1);
	dim3 threads(32, 32);
	QRdecomposition(cusH, cubH, A, Xbc, d_tau, rowsA, colsA, &d_work, &lwork, devInfo);
	stripR << <grid, threads >> >(Xbc, R, colsA, rowsA, colsA);
	cudaFree(d_work);
}

/* Computes the frobenius norm of the given array */
datatype frobeniusNorm(cublasHandle_t handle, datatype *Array, int size) {
	datatype retval;
	cublasStatus_t stat;
	#if (MODE)
		if ((stat = cublasDnrm2(handle, size, Array, 1, &retval)) !=
CUBLAS_STATUS_SUCCESS) {
			printf("cublasDnmr2 failed: %s\n", cublasGetErrorString(stat));
			cleanup();
			exit(-1);
		}
	#else
		if ((stat = cublasSnrm2(handle, size, Array, 1, &retval)) !=
CUBLAS_STATUS_SUCCESS) {
			printf("cublasSnmr2 failed: %s\n", cublasGetErrorString(stat));
			cleanup();
			exit(-1);
		}
	#endif
	return retval;
}
///////////////////////////////////////////////
//            File: reductions.h             //
///////////////////////////////////////////////
#ifndef __reductions__
#define __reductions__
```

```
__global__ void reduceDiagonal_STAGE1(datatype *, unsigned int);
__global__ void reduceDiagonal_STAGE2(unsigned int);
__global__ void filterColumns (datatype *);
__global__ void reduceSquareSum_STAGE1(CustomVector *, unsigned int);
__global__ void reduceSquareSum_STAGE2(unsigned int);
__global__ void pf2fitSquareSum_STAGE1(CustomVector*,CustomVector*, datatype*);
__global__ void pf2fitSquareSum_STAGE2(unsigned int, datatype*);
__global__ void reduceSquareSum_block(datatype *, unsigned int);
__global__ void ReduceSunSign_A_STAGE1(datatype*, datatype*, int, int);
__global__ void ReduceSunSign_A_STAGE2(datatype*, datatype*, int);
__global__ void ReduceSunSign_C(datatype*, int, datatype*);
void get_diag(datatype *);
void get_r(bool *);
void get_sumsum (datatype *);
void get_fit(datatype *);
void get_block_fit(datatype*);

#endif // !__reductions__

////////////////////////////////////////////////
//           File: reductions.cu             //
////////////////////////////////////////////////
#include <iostream>
#include <cmath>
#include "declarations.h"
#include "customvector.h"
#include "memalloc.h"
#include "ReductionBase.cuh"


// Temporary storage
__device__ datatype sums_array[(sizeX + 1024 - 1) / 1024];
// Return value reduceDiagonal_STAGE1 and reduceDiagonal_STAGE2
__device__ datatype diagonal_sum;


/* Reduction for fit = sum(diag(XtX)) in parafac2 */
__global__ void reduceDiagonal_STAGE1(datatype* in, unsigned int N) {
      datatype sum = 0;
      int index = blockIdx.x * blockDim.x + threadIdx.x;
      if (index < N) {
            sum = in[index*N + index];
      }
      sum = blockReduceSum(sum);
      if (threadIdx.x == 0) {
            if (gridDim.x > 1) {
                  sums_array[blockIdx.x] = sum;
            }
            else {
                  diagonal_sum = sum;
            }
      }
}

__global__ void reduceDiagonal_STAGE2(unsigned int N) {
      datatype sum = 0;
      int index = blockIdx.x * blockDim.x + threadIdx.x;
      if (index < N) {
            sum = sums_array[index];
      }
      sum = blockReduceSum(sum);
      if (threadIdx.x == 0) {
            if (gridDim.x > 1) {
                  sums_array[blockIdx.x] = sum;
```

```
            }
            else {
                  diagonal_sum = sum;
            }
      }
}



// Return value ofo filterColumns
__device__ bool r;

/* For pinv function : Reduction count of thresholded elements */
__global__ void filterColumns (datatype *g_idata) {
      unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
      datatype tol = sources * eps * g_idata[0];
      datatype value = g_idata[i];
      if (value > tol) {
            g_idata[i] = 1 / sqrt(value);
            if (i == 0) r = true;
      }
      else {
            g_idata[i] = 0;
            if (i == 0) r = false;
      }
}



// Temporary storage for reduceSquareSum_STAGE1, reduceSquareSum_STAGE2
__device__ datatype squares_array[(((sizeX*sources)/1024) + 1)*minSize];
// Return value for reduceSquareSum_STAGE1, reduceSquareSum_STAGE2
__device__ datatype sumsumX2;


__global__ void reduceSquareSum_STAGE1(CustomVector* cvArray, unsigned int N) {
      datatype sum = 0;
      datatype* in = cvArray[blockIdx.y].array;
      if (in == NULL) {
            return;
      }
      int index = blockIdx.x * blockDim.x + threadIdx.x;
      if (index < N) {
            sum = sqval( in[index] );
      }
      sum = blockReduceSum(sum);
      if (threadIdx.x == 0) {
            squares_array[blockIdx.y * gridDim.x + blockIdx.x] = sum;
      }
}

__global__ void reduceSquareSum_STAGE2(unsigned int N) {
      datatype sum = 0;
      int index = blockIdx.x * blockDim.x + threadIdx.x;
      if (index < N) {
            sum = squares_array[index];
      }
      sum = blockReduceSum(sum);
      if (threadIdx.x == 0) {
            if (gridDim.x > 1) {
                  squares_array[blockIdx.x] = sum;
            }
            else {
                  sumsumX2 = sum;
            }
```

```
	}
}



// Return value
__device__ datatype pf2fit_fit;

__global__ void pf2fitSquareSum_STAGE1(CustomVector* X, CustomVector* M, datatype* buffer) {
	datatype sum = 0;
	datatype* Xin = X[blockIdx.y].array;
	datatype* Min = M[blockIdx.y].array;
	unsigned int N = sizeX*M[blockIdx.y].size;
	int index = blockIdx.x * blockDim.x + threadIdx.x;
	if (index < N) {
		sum = sqval(Xin[index]-Min[index]);
	}
	sum = blockReduceSum(sum);
	if (threadIdx.x == 0) {
		buffer[blockIdx.y * gridDim.x + blockIdx.x] = sum;
	}
}

__global__ void pf2fitSquareSum_STAGE2(unsigned int N, datatype* buffer) {
	datatype sum = 0;
	int index = blockIdx.x * blockDim.x + threadIdx.x;
	if (index < N) {
		sum = buffer[index];
	}
	sum = blockReduceSum(sum);
	if (threadIdx.x == 0) {
		if (gridDim.x > 1) {
			buffer[blockIdx.x] = sum;
		}
		else {
			pf2fit_fit = sum;
		}
	}
}



// Return value of function reduceSquareSum_block
__device__ datatype block_fit;

/* Reduction sum of the square of elements */
__global__ void reduceSquareSum_block(datatype* in, unsigned int N) {
	datatype sum = 0;
	if (threadIdx.x < N) {
		sum = sqval(in[threadIdx.x]);
	}
	sum = blockReduceSum(sum);
	if (threadIdx.x == 0) {
		block_fit = sum;
	}
}

__inline__ __device__
int sgn(datatype val) {
	return (datatype(0) < val) - (val < datatype(0));
}

__global__ void ReduceSunSign_A_STAGE1(datatype *in, datatype* out, int N, int cols) {
```

```
	datatype sum = 0;
	int index = blockIdx.x * blockDim.x + threadIdx.x;
	if (index < N) {
		sum = sgn(in[index*cols + blockIdx.y]);
	}
	sum = blockReduceSum(sum);
	if (threadIdx.x == 0) {
		out[blockIdx.y * 1024 + blockIdx.x] = sum;
	}
}

__global__ void ReduceSunSign_A_STAGE2(datatype *in, datatype* out, int N) {
	datatype sum = 0;
	in = in + blockIdx.x * 1024;
	if (threadIdx.x < N) {
		sum = in[threadIdx.x];
	}
	sum = blockReduceSum(sum);
	if (threadIdx.x == 0) {
		out[blockIdx.x] = sgn(sum + eps);
	}
}

__global__ void ReduceSunSign_C(datatype *in, int N, datatype* out) {
	datatype sum = 0;
	for (int i = 0; i < N; i++) {
		sum += sgn(in[i*blockDim.x + threadIdx.x]);
	}
	out[threadIdx.x] = sgn(sum + eps);
}


/* Copy returned values fron device to host*/

void get_diag (datatype* fit) {
	cudaError_t cet;
	if ((cet = cudaMemcpyFromSymbol(fit, diagonal_sum, sizeof(datatype), 0,
cudaMemcpyDeviceToHost)) != cudaSuccess) {
		printf("cudaMemcpyFromSymbol diagonal_sum failed: %s\n",
cudaGetErrorString(cet));
		cleanup();
		exit(-1);
	}
}

void get_r (bool* fit) {
	cudaError_t cet;
	if ((cet = cudaMemcpyFromSymbol(fit, r, sizeof(bool), 0, cudaMemcpyDeviceToHost)) !=
cudaSuccess) {
		printf("cudaMemcpyFromSymbol r failed: %s\n", cudaGetErrorString(cet));
		cleanup();
		exit(-1);
	}
}

void get_sumsum (datatype* s) {
	cudaError_t cet;
	if ((cet = cudaMemcpyFromSymbol(s, sumsumX2, sizeof(datatype), 0,
cudaMemcpyDeviceToHost)) != cudaSuccess) {
		printf("cudaMemcpyFromSymbol sumsumX2 failed: %s\n", cudaGetErrorString(cet));
		cleanup();
		exit(-1);
	}
}
```

```
void get_fit(datatype *fit) {
	cudaError_t cet;
	if ((cet = cudaMemcpyFromSymbol(fit, pf2fit_fit, sizeof(datatype), 0,
cudaMemcpyDeviceToHost)) != cudaSuccess) {
		printf("cudaMemcpyFromSymbol pf2fit_fit failed: %s\n",
cudaGetErrorString(cet));
		cleanup();
		exit(-1);
	}
}

void get_block_fit(datatype *fit) {
	cudaError_t cet;
	if ((cet = cudaMemcpyFromSymbol(fit, block_fit , sizeof(datatype), 0,
cudaMemcpyDeviceToHost)) != cudaSuccess) {
		printf("cudaMemcpyFromSymbol block_fit failed: %s\n",
cudaGetErrorString(cet));
		cleanup();
		exit(-1);
	}
}
//////////////////////////////////////////////
//        File: ReductionBase.cuh           //
//////////////////////////////////////////////
#ifndef _REDBASE_
#define _REDBASE_


////////////////////////////////////////////
// This file serves as a separator between //
// reduction base functions and reduction  //
// higher-level kernels.                   //
////////////////////////////////////////////

/*************************************
*   REDUCTION-SPECIFIC USER TYPES    *
*    _____________________________    *
*************************************/

/*------------------------------------*/
/* Finding the index of the maximum    */
/* or minimum value of a set of values */
/* requires interleaved, multiple      */
/* reduction units per operation (one  */
/* for max and one for index).         */
/*------------------------------------*/
struct DoubleReductionType {
	datatype	value;
	int		index;
};

/*************************************
*     REDUCTION BASE FUNCTIONS       *
*      _________________________      *
*************************************/

//////////////////////////////////
// Reduction within a single warp //
//////////////////////////////////
__inline__ __device__
DoubleReductionType warpReduceMaximumIndex(DoubleReductionType val) {
	for (int offset = warpSize / 2; offset > 0; offset /= 2) {
		datatype shfl_val = __shfl_down(val.value, offset);
		int shfl_ind = __shfl_down(val.index, offset);
```

```
			if (shfl_val > val.value) {
				val.value = shfl_val;
				val.index = shfl_ind;
			}
		}
		return val;
}

////////////////////////////////////
// Reduction within a single warp //
////////////////////////////////////
__inline__ __device__
datatype warpReduceMax(datatype val) {
		for (int offset = warpSize / 2; offset > 0; offset /= 2) {
			datatype shfl_val = __shfl_down(val, offset);
			if (shfl_val > val)
				val = shfl_val;
		}
		return val;
}

////////////////////////////////////
// Reduction within a single warp //
////////////////////////////////////
__inline__ __device__
datatype warpReduceSum(datatype val) {
		for (int offset = warpSize / 2; offset > 0; offset /= 2)
			val += __shfl_down(val, offset);
		return val;
}

/////////////////////////////////////
// Reduction within a single block //
/////////////////////////////////////
__inline__ __device__
datatype blockReduceSum(datatype val) {

		// Max. block size = 1024 (32 x 32) = 32 warps of 32 threads
		static __shared__
			datatype shared[32];  // Shared mem for 32 partial sums
		int lane = threadIdx.x % warpSize;
		int wid = threadIdx.x / warpSize;

		val = warpReduceSum(val);  // Each warp performs partial reduction

		if (lane == 0)                  // Write reduced value to shared memory
			shared[wid] = val;

		__syncthreads();            // Wait for all partial reductions

						//read from shared memory only if that warp
existed
		val = (threadIdx.x < blockDim.x / warpSize) ? shared[lane] : 0;

		if (wid == 0)                   //Final reduce within first warp
			val = warpReduceSum(val);

		return val;
}

/////////////////////////////////////
// Reduction within a single block //
// (maximum version)               //
/////////////////////////////////////
__inline__ __device__
```

```
DoubleReductionType blockReduceMaximumIndex(DoubleReductionType val) {

      // Max. block size = 1024 (32 x 32) = 32 warps of 32 threads
      static __shared__
            DoubleReductionType shared[32];  // Shared mem for 32 partial sums
      int lane = threadIdx.x % warpSize;
      int wid = threadIdx.x / warpSize;

      val = warpReduceMaximumIndex(val);    // Each warp performs partial reduction

      if (lane == 0)                     // Write reduced value to shared memory
            shared[wid] = val;

      __syncthreads();            // Wait for all partial reductions

                                          //read from shared memory only if that warp
existed
      if (threadIdx.x < blockDim.x / warpSize) {
            val = shared[lane];
      }
      else {
            val.value = 0;
      }

      if (wid == 0)                      //Final reduce within first warp
            val = warpReduceMaximumIndex(val);

      return val;
}

/************************************
*         UNARY FUNCTIONS          *
*          _______________          *
************************************/

/*---------------------------------*/
/* Action performed on data before */
/*  reduction of sum of elements!  */
/*     'SUM OF SQUARE VALUES'      */
/*---------------------------------*/
__inline__ __device__
datatype sqval(datatype val) {
      return val*val;
}

/*---------------------------------*/
/* Action performed on data after  */
/*  reduction of sum of elements!  */
/*     'INVERTED SQUARE ROOT'      */
/*---------------------------------*/
__inline__ __device__
datatype invSQRT(datatype val) {
      return 1.0 / sqrt(val);
}

/******************************
*         GENERATORS         *
*          __________          *
******************************/
// This functions acts as a random shuffle generator
// on the device by taking an index and then returning
// the new index that the specific position maps to.

/*---------------------------------------------------*/
/* This function acts as a random shuffle generator    */
```

```
/* on the device by taking an index and then returning */
/* the new index that the specific position maps to.   */
/*----------------------------------------------------*/
__inline__ __device__
unsigned int myGenerator(unsigned int pos, unsigned int N) {
      // We return the permutation: (1:length) so that every position maps to its self.
      return pos;
}

/**********************
*         END         *
*           ___         *
**********************/

#endif // !REDBASE

//////////////////////////////////////////////
//              File: read.h               //
//////////////////////////////////////////////
#ifndef _read_
#define _read_

int readingInputFromFile(CustomVector**, int*);

#endif // !_read_

//////////////////////////////////////////////
//             File: read.cu               //
//////////////////////////////////////////////
#include <string>
#include <fstream>
#include <iostream>
#include "declarations.h"
#include "customvector.h"

using namespace std;

/* Read input from txt files */
int readingInputFromFile(CustomVector** x, int *max_dim) {
      string line;
      CustomVector temp;
      datatype* lineArray;
      cudaError_t cudaError;
      int i, counter, linesize, max = -1;
      char num[5], filename[25];
      *x = new CustomVector[minSize];
      for (i = 0; i < minSize; i++) {
            strcpy(filename, prefix);
            itoa(i + 1, num, 10);
            strcat(filename, num);
            strcat(filename, ".txt");
            ifstream is(filename, std::ifstream::binary);
            if (is) {
                  linesize = 0;
                  counter = 0;
                  while (getline(is, line)) {
                        if (line[0] == '\n') continue;
                        if (linesize == 0) {
                              linesize = stoi(line);
                              lineArray = new datatype[linesize*sizeX];
                              continue;
                        }
                        #if (MODE)
                              lineArray[counter] = stod(line);
                        #else
```

```
                         lineArray[counter] = stof(line);
                    #endif
                    counter++;
               }
               is.close();
               if ((cudaError = cudaMalloc((void**)&temp.array, linesize * sizeX *
sizeof(datatype))) != cudaSuccess) {
                    printf("cudaMalloc temp array failed: %s\n",
cudaGetErrorString(cudaError));
                    return 1;
               }
               temp.size = linesize;
               if (linesize > max) {
                    max = linesize;
               }
               if ((cudaError = cudaMemcpy(temp.array, lineArray, linesize * sizeX *
sizeof(datatype), cudaMemcpyHostToDevice)) != cudaSuccess) {
                    printf("cudaMemcpy temp array failed: %s\n",
cudaGetErrorString(cudaError));
                    return 2;
               }
               delete[] lineArray;
               (*x)[i] = temp;
          }
          else {
               printf("Cannot open input file\n");
               return 3;
          }
     }
     *max_dim = max;
     return 0;
}
///////////////////////////////////////////////
//            File: parafac.h                //
///////////////////////////////////////////////
#ifndef _parafac_
#define _parafac_

void parafac2 (void);
void parafac (CustomVector *, CustomVector *, int , datatype, int);

#endif // !_parafac_
///////////////////////////////////////////////
//           File: parafac.cu                //
///////////////////////////////////////////////
#include <iostream>
#include <cusolverDn.h>
#include <curand.h>
#include "cublas_v2.h"
#include "declarations.h"
#include "customvector.h"
#include "memalloc.h"
#include "rest.h"
#include "kernel.h"
#include "errorTypes.h"
#include "reductions.h"


/* Extrern global variables */
extern curandGenerator_t generator;
extern cusolverDnHandle_t cusolverH;
extern cublasHandle_t handle;
extern datatype *A, *C, *H, *XtX, *XtXhelp, *QkT_Qk, *diagC, *AdotdiagC, *AdiagCdotH, *U,
*S;
```

```
extern datatype *VT, *Sdiag, *VS, *HCdiag, *Xbc, *XbcTemp, *CTC, *HTH, *HTHdotCTC, *Qa,
*Ra, *ReductionBuffer;
extern datatype *Rb, *Rc, *d_tauA, *d_tauH, *d_tauC, *Xac, *RaTRa, *RbpppRc, *RcT,
*RaRbRc, *S2;
extern CustomVector *X, *CuX, *M, *cuM, firstStageSums, *deviceX, *Q, *P, *Y,
*QRsolveBUFFER;
extern int *devInfo, *ID;


void parafac(CustomVector *Y, CustomVector *QRsolveBUFFER, int F, datatype crit, int
maxit) {
	int i, it;
	cudaError_t cet;
	datatype SumSqX, fit, fit0, fitold;
	if (crit == 0.0) crit = 1e-6;

	dim3 block3(1024);
	dim3 grid3((sizeX*sources + block3.x - 1) / block3.x, minSize);
	reduceSquareSum_STAGE1 << <grid3, block3 >> > (CuX, sizeX*sources);
	grid3.y = 1;
	int N = grid3.x*minSize;
	grid3.x = (N + block3.x - 1) / block3.x;
	reduceSquareSum_STAGE2 << <grid3, block3 >> > (N);
	while (grid3.x > 1) {
		N = grid3.x;
		grid3.x = (N + block3.x - 1) / block3.x;
		reduceSquareSum_STAGE2 << <grid3, block3 >> > (N);
	}
	get_sumsum(&SumSqX); // SumSqX = sum(sum(abs(X).^2));

	fit = SumSqX;
	fit0 = fit;
	fitold = 2 * fit;
	it = 0;
	while (abs((fit - fitold) / fitold)> crit && it < maxit && fit > 10 * eps) {
		it = it + 1;
		fitold = fit;


	//__________________________________________________________________________________
___
		// Update A
		if ((cet = cudaMemset(Xbc, 0, sizeX * sources * sizeof(datatype))) !=
cudaSuccess) {
			printf("cudaMemset Xbc failed: %s\n", cudaGetErrorString(cet));
			cleanup();
			exit(-1);
		} // Xbc = 0
		for (i = 0; i < minSize; i++) {
			arrayMultDiag << <sources*sources / 1024 + 1, 1024 >> > (HCdiag, H, C +
i*sources, sources, sources); // B*diag(C(k,:))
			gpu_blas_mmul(handle, Y[i].array, HCdiag, XbcTemp, 0, sizeX, sources,
sources, sources); // X(:,(k-1)*J+1:k*J)*(B*diag(C(k,:)))
			addMatrices << <(sizeX*sources) / 1024 + 1, 1024 >> > (Xbc, Xbc, XbcTemp,
sizeX*sources); // Xbc = Xbc + X(:,(k-1)*J+1:k*J)*(B*diag(C(k,:)));
		}
		gpu_blas_mmul(handle, H, H, HTH, 1, sources, sources, sources, sources); //
B'*B
		gpu_blas_mmul(handle, C, C, CTC, 1, dimensions, sources, dimensions, sources);
// C'*C
		elementwiseProduct << <1, sources*sources >> > (HTHdotCTC, HTH, CTC,
sources*sources); // (B'*B).*(C'*C)
		pinv(cusolverH, handle, HTHdotCTC, U, S, VT, Sdiag, VS, devInfo, sources); //
pinv((B'*B).*(C'*C))
```

```
            gpu_blas_mmul(handle, Xbc, HTHdotCTC, A, 2, sizeX, sources, sources, sources);
//  A = Xbc*pinv((B'*B).*(C'*C)).';

            // Project X down on orth(A) - saves time if first mode is large
            QRsolve(cusolverH, handle, A, Xbc, d_tauA, Ra, sizeX, sources, Y, sizeX,
sources, QRsolveBUFFER, devInfo); // [Qa,Ra]=qr(A,0); x = Qa'*X;



      //____________________________________________________________________________________
___
            // Update B
            if ((cet = cudaMemset(Xac, 0, sources * sources * sizeof(datatype))) !=
cudaSuccess) {
                  printf("cudaMemset Xac failed: %s\n", cudaGetErrorString(cet));
                  cleanup();
                  exit(-1);
            } // Xac = 0
            for (i = 0; i < minSize; i++) {
                  diagonalize << <1, sources >> >(C, diagC, sources, i); // diag(C(k,:))
                  gpu_blas_mmul(handle, Ra, diagC, HCdiag, 0, sources, sources, sources,
sources); // Ra*diag(C(k, :))
                  gpu_blas_mmul(handle, QRsolveBUFFER[i].array, HCdiag, QkT_Qk, 1, sources,
sources, sources, sources); // x(:,(k-1)*J+1:k*J).'*(Ra*diag(C(k,:)))
                  addMatrices << <1, sources * sources >> > (Xac, Xac, QkT_Qk,
sources*sources); //  Xac = Xac + x(:,(k-1)*J+1:k*J).'*(Ra*diag(C(k,:)));
            }
            gpu_blas_mmul(handle, Ra, Ra, RaTRa, 1, sources, sources, sources, sources);
// Ra'*Ra
            elementwiseProduct << <1, sources*sources >> > (QkT_Qk, RaTRa, CTC,
sources*sources); // B'*B
            pinv(cusolverH, handle, QkT_Qk, U, S, VT, Sdiag, VS, devInfo, sources); //
pinv((Ra'*Ra).*(C'*C))
            gpu_blas_mmul(handle, Xac, QkT_Qk, H, 2, sources, sources, sources, sources);
// B = Xac*pinv((Ra'*Ra).*(C'*C)).';



      //____________________________________________________________________________________
___
            // Update C
            gpu_blas_mmul(handle, H, H, HTH, 1, sources, sources, sources, sources); //
B'*B
            elementwiseProduct << <1, sources*sources >> > (QkT_Qk, RaTRa, HTH,
sources*sources); // (Ra'*Ra).*(B'*B)
            pinv(cusolverH, handle, QkT_Qk, U, S, VT, Sdiag, VS, devInfo, sources); //
ab=pinv((Ra'*Ra).*(B'*B));
            for (i = 0; i < minSize; i++) {
                  gpu_blas_mmul(handle, Ra, QRsolveBUFFER[i].array, U, 1, sources, sources,
sources, sources); // (Ra'* x(:,(k-1)*J+1:k*J)
                  gpu_blas_mmul(handle, U, H, VT, 0, sources, sources, sources, sources);
// Ra'* x(:,(k-1)*J+1:k*J)*B
                  getDiagonal << <1, sources >> > (S, VT, sources); // diag(Ra'* x(:,(k-
1)*J+1:k*J)*B)
                  gpu_blas_mmul(handle, QkT_Qk, S, C + i*sources, 0, sources, sources,
sources, 1); // C(k,:) = (ab*diag(Ra'* x(:,(k-1)*J+1:k*J)*B)).';
            }



      //____________________________________________________________________________________
___
            // Calculating fit
            QRonly(cusolverH, handle, H, HCdiag, d_tauA, Rb, sources, sources, devInfo);
// [~,Rb]=qr(B,0);
```

```
            QRonly(cusolverH, handle, C, HCdiag, d_tauA, Rc, dimensions, sources,
devInfo); // [~,Rc]=qr(C,0);
            ppp << < (minSize*sources*sources) / 1024 + 1, 1024 >> > (Rb, Rc, RbpppRc,
minSize*sources, sources); // ppp(Rb,Rc)
            gpu_blas_mmul(handle, Ra, RbpppRc, RaRbRc, 2, sources, sources,
sources*dimensions, sources); // Ra*ppp(Rb,Rc)
            reduceSquareSum_block << <1, 1024 >> >(RaRbRc, sources*sources*dimensions);
            get_block_fit(&fit); // sum(sum(abs(Ra*ppp(Rb,Rc).').^2))
            fit = SumSqX - fit; // fit=SumSqX-sum(sum(abs(Ra*ppp(Rb,Rc).').^2));
            if (it % show_fit == 0)
                  printf("%12.10f\t\t   %d\t\t\t   %3.4f\n", fit, it, 100 * (1 - fit /
fit0));
      }
      // ORDER ACCORDING TO VARIANCE
      gpu_blas_mmul(handle, A, A, Xac, 1, sizeX, sources, sizeX, sources); // A'*A
      gpu_blas_mmul(handle, C, C, CTC, 1, dimensions, sources, dimensions, sources); //
B'*B
      elementwiseProduct3Arrays << <1, sources*sources >> > (QkT_Qk, Xac, HTH, CTC,
sources*sources); // (A'*A).*(B'*B).*(C'*C)
      getDiagonal << <1, sources >> > (S, QkT_Qk, sources); // Tuck =
diag((A'*A).*(B'*B).*(C'*C));
      bubbleSort << <1, sources >> > (S, ID); // [~,ID] = sort(Tuck);

      // NORMALIZE A AND C(variance in B)
      // A = A(:, ID); B = B(:, ID); C = C(:, ID);
      // for f = 1:Fac, normC(f) = norm(C(:, f)); end
     // for f = 1:Fac, normA(f) = norm(A(:, f)); end
      // the columns of A,C change in normes
      normes << <1, 2 * sources >> > (A, C, S, S2, ID, sizeX, dimensions);
      normDiag << <(sources*sources) / 1024 + 1, 1024 >> > (H, sources, sources, S, S2,
ID); // B = B*diag(normC)*diag(normA);
      updateAC << <(sizeX*sources) / 1024 + 1, 1024 >> > (A, S, ID, sizeX, sources); // A
= A*diag(normA. ^ (-1));
      updateAC << <(dimensions*sources) / 1024 + 1, 1024 >> > (C, S2, ID, dimensions,
sources); // C = C*diag(normC. ^ (-1));

      // APPLY SIGN CONVENTION
      dim3 block(1024);
      dim3 grid(MINIMUM((sizeX + block.x - 1) / block.x, 1024), sources);
      ReduceSunSign_A_STAGE1 << <grid, block >> > (A, ReductionBuffer, sizeX, sources);
      ReduceSunSign_A_STAGE2 << <sources, block >> > (ReductionBuffer, S, grid.x); //
SignA = sign(sum(sign(A))+eps);
      ReduceSunSign_C << <1, sources >> > (C, dimensions, S2); // SignC =
sign(sum(sign(C))+eps);
      updateSignAC << <(sizeX*sources) / 1024 + 1, 1024 >> > (A, S, sizeX, sources); // A
= A*diag(SignA);
      updateSignAC << <(dimensions*sources) / 1024 + 1, 1024 >> > (C, S2, dimensions,
sources); // C = C*diag(SignC);
      signDiag << <(sources*sources) / 1024 + 1, 1024 >> > (H, sources, sources, S, S2);
//  B = B*diag(SignA)*diag(SignC);
}


void parafac2() {
      int i, it = 0;
      datatype fit = 0.0, oldfit = 0.0, fit0 = 0.0;
      // random initialization for A,C (Gaussian destribution)
      rand(generator, A, sizeX * sources); // A = rand(I,F);
      rand(generator, C, dimensions * sources); // C = rand(K,F);
      dim3 grid(1, 1);
      dim3 threads(sources, sources);
      eye << <grid, threads >> > (H, sources); // H = eye(F);
      // Calculate for evaluating fit
      gpu_blas_mmul(handle, X[0].array, X[0].array, XtX, 2, sizeX, X[0].size, sizeX,
X[0].size); // XtX=X{1}*X{1}';
```

```
	for (i = 1; i < minSize; i++) {
		gpu_blas_mmul(handle, X[i].array, X[i].array, XtXhelp, 2, sizeX, X[i].size, 
sizeX, X[i].size); // X{k}*X{k}'
		addMatrices << < sizeX*sizeX / 1024 + 1, 1024 >> > (XtX, XtX, XtXhelp, 
sizeX*sizeX); // XtX = XtX + X{k}*X{k}';
	}
	dim3 block(1024);
	dim3 grid2((sizeX + block.x - 1) / block.x);
	reduceDiagonal_STAGE1 << <grid2, block >> > (XtX, sizeX);
	while (grid2.x > 1) {
		int N = grid2.x;
		grid2.x = (N + block.x - 1) / block.x;
		reduceDiagonal_STAGE2 << <grid2, block >> > (N);
	}
	get_diag(&fit); // fit = sum(diag(XtX));

	oldfit = fit * 2;
	fit0 = fit;
	printf("Fitting model...\n");
	printf("  Loss-value\t\t\tIteration\t\tVariationExpl\n");

	while (abs(fit - oldfit) > oldfit*conv_crit && it < max_iter && fit > 1000 * eps) {
		oldfit = fit;
		it++;
		for (i = 0; i < minSize; i++) {
			arrayMultDiag << <sizeX*sources / 1024 + 1, 1024 >> > (AdotdiagC, A, C + 
i*sources, sizeX, sources); // A*diag(C(k,:))
			gpu_blas_mmul(handle, AdotdiagC, H, AdiagCdotH, 2, sizeX, sources, 
sources, sources); // A*diag(C(k,:))*H'
			gpu_blas_mmul(handle, X[i].array, AdiagCdotH, Q[i].array, 1, sizeX, 
X[i].size, sizeX, sources); // Qk = X{k}'*(A*diag(C(k,:))*H');
			gpu_blas_mmul(handle, Q[i].array, Q[i].array, QkT_Qk, 1, Q[i].size, 
sources, Q[i].size, sources); // Qk'*Qk
			psqrt(cusolverH, handle, QkT_Qk, U, S, VT, Sdiag, VS, devInfo); // 
psqrt(Qk'*Qk)
			gpu_blas_mmul(handle, Q[i].array, QkT_Qk, P[i].array, 0, Q[i].size, 
sources, sources, sources); // P{k} = Qk*psqrt(Qk'*Qk);
			gpu_blas_mmul(handle, X[i].array, P[i].array, Y[i].array, 0, sizeX, 
X[i].size, P[i].size, sources); // Y(:,:,k) = X{k}*P{k};
		}
		parafac(Y, QRsolveBUFFER, sources, 1e-4, 5); // 
[A,H,C]=parafac(reshape(Y,I,F*K),[I F K],F,1e-4,A,H,C,5);
		fit = pf2fit(handle, deviceX, X, A, H, C, P, M, cuM, diagC, AdotdiagC, 
AdiagCdotH,&firstStageSums); // [fit,X] = pf2fit(X,A,H,C,P,K);
		if (!(it % show_fit) || it == 1)
			printf("%12.10f\t\t   %d\t\t\t   %3.4f\n", fit, it, 100 * (1 - fit / 
fit0));
	}
	if (it % show_fit)
		printf("%12.10f\t\t   %d\t\t\t   %3.4f\n", fit, it, 100 * (1 - fit / fit0));
}

//////////////////////////////////////////////
//            File: memalloc.h              //
//////////////////////////////////////////////
#ifndef _memalloc_
#define _memalloc_

void cleanup(void);
void allocate(int);

#endif // !_memalloc_

//////////////////////////////////////////////
//           File: memalloc.cu              //
//////////////////////////////////////////////
```

```
#include <ctime>
#include <iostream>
#include <curand.h>
#include <cusolverDn.h>
#include "cublas_v2.h"
#include "declarations.h"
#include "customvector.h"
#include "kernel.h"
#include "errorTypes.h"


/* Global variables */
curandGenerator_t generator;
cusolverDnHandle_t cusolverH;
cublasHandle_t handle;
datatype *A, *C, *H, *XtX, *XtXhelp, *QkT_Qk, *diagC, *AdotdiagC, *AdiagCdotH, *U, *S,
*VT;
datatype *Sdiag, *VS, *HCdiag, *Xbc, *XbcTemp, *CTC, *HTH, *HTHdotCTC, *Qa, *Ra, *d_tauA;
datatype *d_tauH, *d_tauC, *Rb, *Rc, *Xac, *RaTRa, *RbpppRc, *RcT, *RaRbRc, *S2,
*ReductionBuffer;
CustomVector *X, *CuX, *M, *cuM, firstStageSums, *deviceX, *Q, *P, *Y, *QRsolveBUFFER;
int *devInfo, *ID;


/* Cleaning GPU memory */
void cleanup() {
      int i;
      cudaError_t cerror;
      cublasStatus_t cubstat;
      curandStatus_t curstat;
      cusolverStatus_t cusstat;
      if (A && (cerror = cudaFree(A)) != cudaSuccess)
            printf("cudaFree A failed: %s\n", cudaGetErrorString(cerror));
      if (C && (cerror = cudaFree(C)) != cudaSuccess)
            printf("cudaFree C failed: %s\n", cudaGetErrorString(cerror));
      if (H && (cerror = cudaFree(H)) != cudaSuccess)
            printf("cudaFree H failed: %s\n", cudaGetErrorString(cerror));
      if (XtX && (cerror = cudaFree(XtX)) != cudaSuccess)
            printf("cudaFree XtX failed: %s\n", cudaGetErrorString(cerror));
      if (XtXhelp && (cerror = cudaFree(XtXhelp)) != cudaSuccess)
            printf("cudaFree XtXhelp failed: %s\n", cudaGetErrorString(cerror));
      if (QkT_Qk && (cerror = cudaFree(QkT_Qk)) != cudaSuccess)
            printf("cudaFree QkT_Qk failed: %s\n", cudaGetErrorString(cerror));
      if (diagC && (cerror = cudaFree(diagC)) != cudaSuccess)
            printf("cudaFree diagC failed: %s\n", cudaGetErrorString(cerror));
      if (AdotdiagC && (cerror = cudaFree(AdotdiagC)) != cudaSuccess)
            printf("cudaFree AdotdiagC failed: %s\n", cudaGetErrorString(cerror));
      if (AdiagCdotH && (cerror = cudaFree(AdiagCdotH)) != cudaSuccess)
            printf("cudaFree AdiagCdotH failed: %s\n", cudaGetErrorString(cerror));
      if (U && (cerror = cudaFree(U)) != cudaSuccess)
            printf("cudaFree U failed: %s\n", cudaGetErrorString(cerror));
      if (S && (cerror = cudaFree(S)) != cudaSuccess)
            printf("cudaFree S failed: %s\n", cudaGetErrorString(cerror));
      if (S2 && (cerror = cudaFree(S2)) != cudaSuccess)
            printf("cudaFree S2 failed: %s\n", cudaGetErrorString(cerror));
      if (VT && (cerror = cudaFree(VT)) != cudaSuccess)
            printf("cudaFree VT failed: %s\n", cudaGetErrorString(cerror));
      if (Sdiag && (cerror = cudaFree(Sdiag)) != cudaSuccess)
            printf("cudaFree Sdiag failed: %s\n", cudaGetErrorString(cerror));
      if (VS && (cerror = cudaFree(VS)) != cudaSuccess)
            printf("cudaFree VS failed: %s\n", cudaGetErrorString(cerror));
      if (HCdiag && (cerror = cudaFree(HCdiag)) != cudaSuccess)
            printf("cudaFree HCdiag failed: %s\n", cudaGetErrorString(cerror));
      if (Xbc && (cerror = cudaFree(Xbc)) != cudaSuccess)
            printf("cudaFree Xbc failed: %s\n", cudaGetErrorString(cerror));
```

```
	if (XbcTemp && (cerror = cudaFree(XbcTemp)) != cudaSuccess)
		printf("cudaFree XbcTemp failed: %s\n", cudaGetErrorString(cerror));
	if (CTC && (cerror = cudaFree(CTC)) != cudaSuccess)
		printf("cudaFree CTC failed: %s\n", cudaGetErrorString(cerror));
	if (HTH && (cerror = cudaFree(HTH)) != cudaSuccess)
		printf("cudaFree HTH failed: %s\n", cudaGetErrorString(cerror));
	if (HTHdotCTC && (cerror = cudaFree(HTHdotCTC)) != cudaSuccess)
		printf("cudaFree HTHdotCTC failed: %s\n", cudaGetErrorString(cerror));
	if (CuX && (cerror = cudaFree(CuX)) != cudaSuccess)
		printf("cudaFree cuX failed: %s\n", cudaGetErrorString(cerror));
	if (deviceX && (cerror = cudaFree(deviceX)) != cudaSuccess)
		printf("cudaFree cuX failed: %s\n", cudaGetErrorString(cerror));
	if (Qa && (cerror = cudaFree(Qa)) != cudaSuccess)
		printf("cudaFree Qa failed: %s\n", cudaGetErrorString(cerror));
	if (Ra && (cerror = cudaFree(Ra)) != cudaSuccess)
		printf("cudaFree Ra failed: %s\n", cudaGetErrorString(cerror));
	if (Rb && (cerror = cudaFree(Rb)) != cudaSuccess)
		printf("cudaFree Ra failed: %s\n", cudaGetErrorString(cerror));
	if (Rc && (cerror = cudaFree(Rc)) != cudaSuccess)
		printf("cudaFree Ra failed: %s\n", cudaGetErrorString(cerror));
	if (firstStageSums.array && (cerror = cudaFree(firstStageSums.array)) != 
cudaSuccess)
		printf("cudaFree Ra failed: %s\n", cudaGetErrorString(cerror));
	if (devInfo && (cerror = cudaFree(devInfo)) != cudaSuccess)
		printf("cudaFree devInfo failed: %s\n", cudaGetErrorString(cerror));
	if (Xac && (cerror = cudaFree(Xac)) != cudaSuccess)
		printf("cudaFree Xac failed: %s\n", cudaGetErrorString(cerror));
	if (RaTRa && (cerror = cudaFree(RaTRa)) != cudaSuccess)
		printf("cudaFree RaTRa failed: %s\n", cudaGetErrorString(cerror));
	if (RbpppRc && (cerror = cudaFree(RbpppRc)) != cudaSuccess)
		printf("cudaFree RbpppRc failed: %s\n", cudaGetErrorString(cerror));
	if (RcT && (cerror = cudaFree(RcT)) != cudaSuccess)
		printf("cudaFree RcT failed: %s\n", cudaGetErrorString(cerror));
	if (RaRbRc && (cerror = cudaFree(RaRbRc)) != cudaSuccess)
		printf("cudaFree RaRbRc failed: %s\n", cudaGetErrorString(cerror));
	if (X) {
		for (i = 0; i < minSize; i++) {
			if (X[i].array && (cerror = cudaFree(X[i].array)) != cudaSuccess)
				printf("cudaFree array of X failed: %s\n", 
cudaGetErrorString(cerror));
		}
		delete[] X;
	}
	if (M) {
		for (i = 0; i < minSize; i++) {
			if (M[i].array && (cerror = cudaFree(M[i].array)) != cudaSuccess)
				printf("cudaFree array of M failed: %s\n", 
cudaGetErrorString(cerror));
		}
		delete[] M;
	}
	if (cuM && (cerror = cudaFree(cuM)) != cudaSuccess)
		printf("cudaFree cuM failed: %s\n", cudaGetErrorString(cerror));
	if (ReductionBuffer && (cerror = cudaFree(ReductionBuffer)) != cudaSuccess)
		printf("cudaFree ReductionBuffer failed: %s\n", cudaGetErrorString(cerror));
	if (d_tauA && (cerror = cudaFree(d_tauA)) != cudaSuccess)
		printf("cudaFree d_tauA failed: %s\n", cudaGetErrorString(cerror));
	if (d_tauH && (cerror = cudaFree(d_tauH)) != cudaSuccess)
		printf("cudaFree d_tauH failed: %s\n", cudaGetErrorString(cerror));
	if (d_tauC && (cerror = cudaFree(d_tauC)) != cudaSuccess)
		printf("cudaFree d_tauC failed: %s\n", cudaGetErrorString(cerror));
	if (ID && (cerror = cudaFree(ID)) != cudaSuccess)
		printf("cudaFree ID failed: %s\n", cudaGetErrorString(cerror));
	if (generator && (curstat = curandDestroyGenerator(generator)) != 
CURAND_STATUS_SUCCESS)
```

```
		printf("curandDestroyGenerator failed: %s\n", curandGetErrorString(curstat));
	if (handle && (cubstat = cublasDestroy(handle)) != CUBLAS_STATUS_SUCCESS)
		printf("cublasDestroy failed : %s\n", cublasGetErrorString(cubstat));
	if (cusolverH && (cusstat = cusolverDnDestroy(cusolverH)) !=
CUSOLVER_STATUS_SUCCESS)
		printf("cusolverDnDestroy failed: %s\n", cusolverGetErrorString(cusstat));
	for (i = 0; i < minSize; i++) {
		if (Q[i].array && (cerror = cudaFree(Q[i].array)) != cudaSuccess)
			printf("cudaFree Q array failed: %s\n", cudaGetErrorString(cerror));
		if (P[i].array && (cerror = cudaFree(P[i].array)) != cudaSuccess)
			printf("cudaFree P array failed: %s\n", cudaGetErrorString(cerror));
		if (Y[i].array && (cerror = cudaFree(Y[i].array)) != cudaSuccess)
			printf("cudaFree Y array failed: %s\n", cudaGetErrorString(cerror));
		if (QRsolveBUFFER[i].array && (cerror = cudaFree(QRsolveBUFFER[i].array)) !=
cudaSuccess)
			printf("cudaFree Y array failed: %s\n", cudaGetErrorString(cerror));
	}
	if (Q) delete[] Q;
	if (P) delete[] P;
	if (Y) delete[] Y;
	if (QRsolveBUFFER) delete[] QRsolveBUFFER;
	udest << < 1, 1 >> > ();
	if ((cerror = cudaDeviceReset()) != cudaSuccess)
		printf("cudaDeviceReset failed: %s\n", cudaGetErrorString(cerror));
}

/* Allocating all the necessary memory */
void allocate(int maximum_variable_dimension) {
	int i;
	cudaError_t cuerror;
	cublasStatus_t cubstat;
	curandStatus_t curstat;
	cusolverStatus_t cusstat;
	if ((M = new CustomVector[minSize]) == NULL) {
		printf("Cannot allocate memory for M\n");
		cleanup();
		exit(-1);
	}
	if ((Q = new CustomVector[minSize]) == NULL) {
		printf("Cannot allocate memory for Q\n");
		cleanup();
		exit(-1);
	}
	if ((P = new CustomVector[minSize]) == NULL) {
		printf("Cannot allocate memory for P\n");
		cleanup();
		exit(-1);
	}
	if ((Y = new CustomVector[minSize]) == NULL) {
		printf("Cannot allocate memory for Y\n");
		cleanup();
		exit(-1);
	}
	if ((QRsolveBUFFER = new CustomVector[minSize]) == NULL) {
		printf("Cannot allocate memory for QRsolveBUFFER\n");
		cleanup();
		exit(-1);
	}
	if ((cubstat = cublasCreate(&handle)) != CUBLAS_STATUS_SUCCESS) {
		printf("cublasCreate failed: %s\n", cublasGetErrorString(cubstat));
		cleanup();
		exit(-1);
	}
	for (i = 0; i < minSize; i++) {
```

```
            if ((cuerror = cudaMalloc((void**)&Y[i].array, sizeX * sources *
sizeof(datatype))) != cudaSuccess) {
                  printf("cudaMalloc failed Y array: %s\n", cudaGetErrorString(cuerror));
                  cleanup();
                  exit(-1);
            }
            if ((cuerror = cudaMalloc((void**)&QRsolveBUFFER[i].array, sizeX * sources *
sizeof(datatype))) != cudaSuccess) {
                  printf("cudaMalloc failed Y array: %s\n", cudaGetErrorString(cuerror));
                  cleanup();
                  exit(-1);
            }
      }
      if ((cuerror = cudaMalloc((void **)&A, sizeX * sources * sizeof(datatype))) !=
cudaSuccess) {
            printf("cudaMalloc failed A : %s\n", cudaGetErrorString(cuerror));
            cleanup();
            exit(-1);
      }
      if ((cuerror = cudaMalloc((void **)&C, dimensions * sources * sizeof(datatype))) !=
cudaSuccess) {
            printf("cudaMalloc failed C : %s\n", cudaGetErrorString(cuerror));
            cleanup();
            exit(-1);
      }
      if ((cuerror = cudaMalloc((void **)&H, sources * sources * sizeof(datatype))) !=
cudaSuccess) {
            printf("cudaMalloc failed H : %s\n", cudaGetErrorString(cuerror));
            cleanup();
            exit(-1);
      }
      if ((cuerror = cudaMalloc((void **)&XtX, sizeX * sizeX * sizeof(datatype))) !=
cudaSuccess) {
            printf("cudaMalloc failed XtX : %s\n", cudaGetErrorString(cuerror));
            cleanup();
            exit(-1);
      }
      if ((cuerror = cudaMalloc((void **)&XtXhelp, sizeX * sizeX * sizeof(datatype))) !=
cudaSuccess) {
            printf("cudaMalloc failed XtXhelp : %s\n", cudaGetErrorString(cuerror));
            cleanup();
            exit(-1);
      }
      if ((cuerror = cudaMalloc((void **)&QkT_Qk, sources * sources * sizeof(datatype)))
!= cudaSuccess) {
            printf("cudaMalloc failed QkT_Qk : %s\n", cudaGetErrorString(cuerror));
            cleanup();
            exit(-1);
      }
      if ((cuerror = cudaMalloc((void **)&diagC, sources * sources * sizeof(datatype))) !=
cudaSuccess) {
            printf("cudaMalloc failed diagC : %s\n", cudaGetErrorString(cuerror));
            cleanup();
            exit(-1);
      }
      if ((cuerror = cudaMalloc((void **)&AdotdiagC, sizeX * sources * sizeof(datatype)))
!= cudaSuccess) {
            printf("cudaMalloc failed AdotdiagC : %s\n", cudaGetErrorString(cuerror));
            cleanup();
            exit(-1);
      }
      if ((cuerror = cudaMalloc((void **)&AdiagCdotH, sizeX * sources * sizeof(datatype)))
!= cudaSuccess) {
            printf("cudaMalloc failed AdiagCdotH : %s\n", cudaGetErrorString(cuerror));
            cleanup();
```

```
		exit(-1);
	}
	if ((cuerror = cudaMalloc((void **)&U, sources * sources * sizeof(datatype))) != 
cudaSuccess) {
		printf("cudaMalloc failed U: %s\n", cudaGetErrorString(cuerror));
		cleanup();
		exit(-1);
	}
	if ((cuerror = cudaMalloc((void **)&S, sources * sizeof(datatype))) != cudaSuccess) 
{
		printf("cudaMalloc failed S: %s\n", cudaGetErrorString(cuerror));
		cleanup();
		exit(-1);
	}
	if ((cuerror = cudaMalloc((void **)&S2, sources * sizeof(datatype))) != cudaSuccess) 
{
		printf("cudaMalloc failed S2: %s\n", cudaGetErrorString(cuerror));
		cleanup();
		exit(-1);
	}
	if ((cuerror = cudaMalloc((void **)&ID, sources * sizeof(int))) != cudaSuccess) {
		printf("cudaMalloc failed ID : %s\n", cudaGetErrorString(cuerror));
		cleanup();
		exit(-1);
	}
	if ((cuerror = cudaMalloc((void **)&VT, sources * sources * sizeof(datatype))) != 
cudaSuccess) {
		printf("cudaMalloc failed VT: %s\n", cudaGetErrorString(cuerror));
		cleanup();
		exit(-1);
	}
	if ((cuerror = cudaMalloc((void **)&Sdiag, sources * sources * sizeof(datatype))) != 
cudaSuccess) {
		printf("cudaMalloc failed Sdiag: %s\n", cudaGetErrorString(cuerror));
		cleanup();
		exit(-1);
	}
	if ((cuerror = cudaMalloc((void **)&VS, sources * sources * sizeof(datatype))) != 
cudaSuccess) {
		printf("cudaMalloc failed VS: %s\n", cudaGetErrorString(cuerror));
		cleanup();
		exit(-1);
	}
	if ((cuerror = cudaMalloc((void **)&HCdiag, sources * sources * sizeof(datatype))) 
!= cudaSuccess) {
		printf("cudaMalloc failed HCdiag: %s\n", cudaGetErrorString(cuerror));
		cleanup();
		exit(-1);
	}
	if ((cuerror = cudaMalloc((void **)&Xbc, sizeX * sources * sizeof(datatype))) != 
cudaSuccess) {
		printf("cudaMalloc failed Xbc: %s\n", cudaGetErrorString(cuerror));
		cleanup();
		exit(-1);
	}
	if ((cuerror = cudaMalloc((void **)&XbcTemp, sizeX * sources * sizeof(datatype))) != 
cudaSuccess) {
		printf("cudaMalloc failed XbcTemp: %s\n", cudaGetErrorString(cuerror));
		cleanup();
		exit(-1);
	}
	if ((cuerror = cudaMalloc((void **)&CTC, sources * sources * sizeof(datatype))) != 
cudaSuccess) {
		printf("cudaMalloc failed CTC: %s\n", cudaGetErrorString(cuerror));
		cleanup();
```

```
		exit(-1);
	}
if ((cuerror = cudaMalloc((void **)&HTH, sources * sources * sizeof(datatype))) != 
cudaSuccess) {
		printf("cudaMalloc failed HTH: %s\n", cudaGetErrorString(cuerror));
		cleanup();
		exit(-1);
	}
	if ((cuerror = cudaMalloc((void **)&HTHdotCTC, sources * sources * 
sizeof(datatype))) != cudaSuccess) {
		printf("cudaMalloc failed HTHdotCTC: %s\n", cudaGetErrorString(cuerror));
		cleanup();
		exit(-1);
	}
	if ((cuerror = cudaMalloc((void **)&Qa, sizeX * sources * sizeof(datatype))) != 
cudaSuccess) {
		printf("cudaMalloc failed Qa: %s\n", cudaGetErrorString(cuerror));
		cleanup();
		exit(-1);
	}
	if ((cuerror = cudaMalloc((void **)&Ra, sources * sources * sizeof(datatype))) != 
cudaSuccess) {
		printf("cudaMalloc failed Ra: %s\n", cudaGetErrorString(cuerror));
		cleanup();
		exit(-1);
	}
	if ((cuerror = cudaMalloc((void **)&Rb, sources * sources * sizeof(datatype))) != 
cudaSuccess) {
		printf("cudaMalloc failed Rb: %s\n", cudaGetErrorString(cuerror));
		cleanup();
		exit(-1);
	}
	if ((cuerror = cudaMalloc((void **)&Rc, dimensions * sources * sizeof(datatype))) != 
cudaSuccess) {
		printf("cudaMalloc failed Rc: %s\n", cudaGetErrorString(cuerror));
		cleanup();
		exit(-1);
	}
	if ((cuerror = cudaMalloc((void **)&CuX, minSize * sizeof(CustomVector))) != 
cudaSuccess) {
		printf("cudaMalloc failed CuX: %s\n", cudaGetErrorString(cuerror));
		cleanup();
		exit(-1);
	}
	if ((cuerror = cudaMalloc((void **)&deviceX, minSize * sizeof(CustomVector))) != 
cudaSuccess) {
		printf("cudaMalloc failed deviceX: %s\n", cudaGetErrorString(cuerror));
		cleanup();
		exit(-1);
	}
	if ((cuerror = cudaMalloc((void **)&cuM, minSize * sizeof(CustomVector))) != 
cudaSuccess) {
		printf("cudaMalloc failed cuM: %s\n", cudaGetErrorString(cuerror));
		cleanup();
		exit(-1);
	}
	if ((cuerror = cudaMalloc((void **)&d_tauA, sizeX * sizeof(datatype))) != 
cudaSuccess) {
		printf("cudaMalloc failed sizeX: %s\n", cudaGetErrorString(cuerror));
		cleanup();
		exit(-1);
	}
	if ((cuerror = cudaMalloc((void **)&d_tauH, sources * sizeof(datatype))) != 
cudaSuccess) {
```

```
		printf("cudaMalloc failed sizeX: %s\n", cudaGetErrorString(cuerror));
		cleanup();
		exit(-1);
	}
	if ((cuerror = cudaMalloc((void **)&d_tauC, minSize * sizeof(datatype))) != 
cudaSuccess) {
		printf("cudaMalloc failed sizeX: %s\n", cudaGetErrorString(cuerror));
		cleanup();
		exit(-1);
	}
	if ((cuerror = cudaMalloc((void**)&devInfo, sizeof(int))) != cudaSuccess) {
		printf("cudaMalloc failed devInfo: %s\n", cudaGetErrorString(cuerror));
		cleanup();
		exit(-1);
	}
	if ((cuerror = cudaMalloc((void **)&Xac, sources * sources * sizeof(datatype))) != 
cudaSuccess) {
		printf("cudaMalloc failed Xac: %s\n", cudaGetErrorString(cuerror));
		cleanup();
		exit(-1);
	}
	if ((cuerror = cudaMalloc((void **)&RaTRa, sources * sources * sizeof(datatype))) != 
cudaSuccess) {
		printf("cudaMalloc failed RaTRa: %s\n", cudaGetErrorString(cuerror));
		cleanup();
		exit(-1);
	}
	if ((cuerror = cudaMalloc((void **)&RbpppRc, (dimensions * sources) * sources * 
sizeof(datatype))) != cudaSuccess) {
		printf("cudaMalloc failed RbpppRc: %s\n", cudaGetErrorString(cuerror));
		cleanup();
		exit(-1);
	}
	if ((cuerror = cudaMalloc((void **)&RaRbRc, (dimensions * sources) * sources * 
sizeof(datatype))) != cudaSuccess) {
		printf("cudaMalloc failed RaRbRc: %s\n", cudaGetErrorString(cuerror));
		cleanup();
		exit(-1);
	}
	if ((cuerror = cudaMalloc((void **)&RcT, dimensions * sources * sizeof(datatype))) 
!= cudaSuccess) {
		printf("cudaMalloc failed RcT: %s\n", cudaGetErrorString(cuerror));
		cleanup();
		exit(-1);
	}
	if ((cuerror = cudaMalloc((void **)&ReductionBuffer, sources * 1024 * 
sizeof(datatype))) != cudaSuccess) {
		printf("cudaMalloc failed ReductionBuffer: %s\n", 
cudaGetErrorString(cuerror));
		cleanup();
		exit(-1);
	}
	if ((curstat = curandCreateGenerator(&generator, CURAND_RNG_PSEUDO_DEFAULT)) != 
CURAND_STATUS_SUCCESS) {
		printf("curandCreateGenerator failed: %s\n", curandGetErrorString(curstat));
		cleanup();
		exit(-1);
	}
	if ((curstat = curandSetPseudoRandomGeneratorSeed(generator, (unsigned long long)
clock())) != CURAND_STATUS_SUCCESS) {
		printf("curandSetPseudoRandomGeneratorSeed failed: %s\n", 
curandGetErrorString(curstat));
		cleanup();
		exit(-1);
	}
```

```
	if ((cusstat = cusolverDnCreate(&cusolverH)) != CUSOLVER_STATUS_SUCCESS) {
		printf("cusolverDnCreate failed: %s\n", cusolverGetErrorString(cusstat));
		cleanup();
		exit(-1);
	}
	uinit << < 1, 1 >> > ();


	firstStageSums.size = minSize * (((sizeX*maximum_variable_dimension) + 1024 - 1) / 
1024);
	if ((cuerror = cudaMalloc((void **)&firstStageSums.array, firstStageSums.size * 
sizeof(datatype))) != cudaSuccess) {
		printf("cudaMalloc failed fs: %s\n", cudaGetErrorString(cuerror));
		cleanup();
		exit(-1);
	}
	for (i = 0; i < minSize; i++) {
		if ((cuerror = cudaMalloc((void**)&Q[i].array, X[i].size * sources * 
sizeof(datatype))) != cudaSuccess) {
			printf("cudaMalloc failed Q array: %s\n", cudaGetErrorString(cuerror));
			cleanup();
			exit(-1);
		}
		Q[i].size = X[i].size;
		if ((cuerror = cudaMalloc((void**)&P[i].array, X[i].size * sources * 
sizeof(datatype))) != cudaSuccess) {
			printf("cudaMalloc failed P array: %s\n", cudaGetErrorString(cuerror));
			cleanup();
			exit(-1);
		}
		P[i].size = X[i].size;
		if ((cuerror = cudaMalloc((void**)&M[i].array, sizeX * X[i].size * 
sizeof(datatype))) != cudaSuccess) {
			printf("cudaMalloc failed M array: %s\n", cudaGetErrorString(cuerror));
			cleanup();
			exit(-1);
		}
		M[i].size = X[i].size;
	}
	if ((cuerror = cudaMemcpy(CuX, Y, minSize * sizeof(CustomVector), 
cudaMemcpyHostToDevice)) != cudaSuccess) {
		printf("cudaMemcpy cux failed: %s\n", cudaGetErrorString(cuerror));
		cleanup();
		exit(-1);
	}
	if ((cuerror = cudaMemcpy(cuM, M, minSize * sizeof(CustomVector), 
cudaMemcpyHostToDevice)) != cudaSuccess) {
		printf("cudaMemcpy cuM failed: %s\n", cudaGetErrorString(cuerror));
		cleanup();
		exit(-1);
	}
	if ((cuerror = cudaMemcpy(deviceX, X, minSize * sizeof(CustomVector), 
cudaMemcpyHostToDevice)) != cudaSuccess) {
		printf("cudaMemcpy cuM failed: %s\n", cudaGetErrorString(cuerror));
		cleanup();
		exit(-1);
	}
}
///////////////////////////////////////////////
//             File: kernel.h               //
///////////////////////////////////////////////
#ifndef _kernel_
#define _kernel_
```

```
__global__ void uinit();
__global__ void udest();
__global__ void eye(datatype *, int);
__global__ void addMatrices (datatype *, datatype *, datatype *, int);
__global__ void subtractMatrices(datatype *, datatype *, datatype *, int);
__global__ void diagonalize (datatype *, datatype*, int, int);
__global__ void elementwiseProduct (datatype *, datatype *, datatype *, int);
__global__ void elementwiseProduct3Arrays (datatype *, datatype *, datatype *, datatype
*, int);
__global__ void pinvDiagonalMatrix (datatype *, datatype*, int, int);
__global__ void stripR(datatype*, datatype*, int, int, int);
__global__ void getDiagonal(datatype *, datatype *, int);
__global__ void ppp (datatype*, datatype*, datatype*,int,int);
__global__ void normes (datatype*, datatype *, datatype*, datatype*, int *, unsigned int,
unsigned int);
__global__ void normDiag (datatype*, unsigned int, unsigned int, datatype*, datatype*,
int*);
__global__ void bubbleSort (datatype*, int*);
__global__ void updateAC (datatype *, datatype *, int *, unsigned int, unsigned int);
__global__ void arrayMultDiag(datatype *, datatype *, datatype *, unsigned int, unsigned
int);
__global__ void updateSignAC(datatype*, datatype *, unsigned int, unsigned int);
__global__ void signDiag(datatype*, unsigned int, unsigned int, datatype*, datatype*);

#endif // !__kernel__

///////////////////////////////////////////////
//            File: kernel.cu                //
///////////////////////////////////////////////
#include <iostream>
#include <ctime>
#include <cublas_v2.h>
#include "declarations.h"
#include "customvector.h"
#include "kernel.h"


/****************
*   VARIABLES   *
*    _________   *
****************/
// Handle of cublas context allocated statically
// on device memory.
__device__ cublasHandle_t handle_gpu;


/***********************
*   FUNCTIONS - init   *
*    ________________   *
***********************/
// Initialize cublas context. It's a separate kernel
// so as to be used as implicit synchronization point.
__global__ void uinit() {
      cublasCreate(&handle_gpu);
}

/***********************
*   FUNCTIONS - dest   *
*    ________________   *
***********************/
// Initialize cublas context. It's a separate kernel so as
// to be used as implicit synchronization point before destruction.
__global__ void udest() {
      cublasDestroy(handle_gpu);
}
```

```
/* Creates an identity matrix with size row x col (needs 2D block) */
__global__ void eye (datatype *matrix, int size) {
	int x = blockDim.x*blockIdx.x + threadIdx.x;
	int y = blockDim.y*blockIdx.y + threadIdx.y;
	if (x < size && y < size) {
		if (x == y) matrix[x*size + y] = 1.0;
		else matrix[x*size + y] = 0.0;
	}
}


/* Adds two matrices a,b and stores the result in matrix c */
__global__ void addMatrices (datatype *c, datatype *a, datatype *b, int size) {
	int index = blockIdx.x *blockDim.x + threadIdx.x;
	if (index < size) c[index] = a[index] + b[index];
}


/* Adds two matrices a,b and stores the result in matrix c */
__global__ void subtractMatrices (datatype *c, datatype *a, datatype *b, int size) {
	int index = blockIdx.x *blockDim.x + threadIdx.x;
	if (index < size) c[index] = a[index] - b[index];
}

/* Takes a row from an array and creates a square array */
__global__ void diagonalize (datatype *C, datatype* out, int size, int row) {
	int i = threadIdx.x;
	out[i*size + i] = C[row*size + i];
}


/* Multiplies two matrices element by element a,b and stores the result in matrix c */
__global__ void elementwiseProduct (datatype *c, datatype *a, datatype *b, int size) {
	int index = blockIdx.x *blockDim.x + threadIdx.x;
	if (index < size) c[index] = a[index] * b[index];
}

/* Multiplies three matrices element by element a,b,c and stores the result in matrix d
*/
__global__ void elementwiseProduct3Arrays(datatype *d, datatype *a, datatype *b, datatype
*c, int size) {
	int index = blockIdx.x *blockDim.x + threadIdx.x;
	if (index < size) d[index] = a[index] * b[index]* c[index];
}

/* Pseudoinverse of diagonal matrix */
__global__ void pinvDiagonalMatrix (datatype *C, datatype* out, int size, int row) {
	int i = threadIdx.x;
	out[i*size + i] = 1.0 / C[row*size + i];
}

/* Takes only the R array from the QR decomposition */
__global__ void stripR(datatype* A, datatype* out,int rows, int cols,int common_dim) {
	int x = blockDim.x*blockIdx.x + threadIdx.x;
	int y = blockDim.y*blockIdx.y + threadIdx.y;
	if (x < rows && y < cols) {
		if (x < y) {
			out[y*common_dim + x] = 0.0;
		}
		else {
			out[y*common_dim + x] = A[x*cols + y];
		}
	}
```

```
}


__global__ void getDiagonal (datatype *out, datatype *in, int size) {
      int i = threadIdx.x;
      out[i] = in[i*size + i];
}


/* Element wise product of two matrices*/
__global__ void ppp(datatype* in1, datatype* in2, datatype* out,int rows,int cols) {
      int pos = blockDim.x*blockIdx.x + threadIdx.x;
      if (pos < rows*cols) {
            int x = pos / cols;
            int y = pos % cols;
            int in_x = x % cols;
            int in_y = y;
            int who = x / cols;
            out[pos] = in1[in_x*cols + in_y] * in2[y + who*cols];
      }
}


__global__ void normes (datatype* A, datatype *C, datatype* outA, datatype* outC, int
*ID,
                                    unsigned int colsizeA, unsigned int colsizeC) {

      int i = threadIdx.x;
      if (i < (blockDim.x / 2)) {
            #if (MODE)
                  cublasDnrm2(handle_gpu, colsizeA, A + ID[i], sources, outA + i);
            #else
                  cublasSnrm2(handle_gpu, colsizeA, A + ID[i], sources, outA + i);
            #endif
      }
      else {
            #if (MODE)
                  cublasDnrm2(handle_gpu, colsizeC, C + ID[i - sources], sources, outC + i
- sources);
            #else
                  cublasSnrm2(handle_gpu, colsizeC, C + ID[i], sources, outC + i);
            #endif
      }
}

__global__ void normDiag (datatype *array, unsigned int rows, unsigned int cols, datatype
*S, datatype *S2, int *ID) {
      int pos = blockDim.x*blockIdx.x + threadIdx.x;
      if (pos >= rows*cols) return;
      int i = pos / cols;
      int y = pos % cols;
      int source = ID[y];
      datatype result = array[i*cols + source] * S[y] * S2[y];
      __syncthreads();
      array[pos] = result;
}


__global__ void bubbleSort (datatype* a, int* ID) {

      int size = blockDim.x;
      ID[threadIdx.x] = threadIdx.x;
      __syncthreads();
      if (threadIdx.x >= (size / 2)) return;
```

```
	int i = blockDim.x * blockIdx.x + threadIdx.x * 2;
	int cacheFirst, cacheSecond, cacheThird;

	for (int j = 0; j < size / 2 + 1; j++) {
		if (i + 1 < size) {
			cacheFirst = ID[i];
			cacheSecond = ID[i + 1];
			if (a[cacheFirst] > a[cacheSecond]) {
				int temp = cacheFirst;
				ID[i] = cacheSecond;
				cacheSecond = ID[i + 1] = temp;
			}
		}
		if (i + 2 < size) {
			cacheThird = ID[i + 2];
			if (a[cacheSecond] > a[cacheThird]) {
				int temp = cacheSecond;
				ID[i + 1] = cacheThird;
				ID[i + 2] = temp;
			}
		}
		__syncthreads();
	}
}


__global__ void updateAC (datatype *AC, datatype *normAC, int *ID, unsigned int row,
unsigned int col) {
	int index = blockDim.x*blockIdx.x + threadIdx.x;
	if (index < row*col) {
		int x = index / col;
		int y = index % col;
		int source = ID[y];
		datatype result = AC[x*col + source] / normAC[y];
		__syncthreads();
		AC[index] = result;
	}
}


__global__ void arrayMultDiag (datatype *res, datatype *a, datatype *diag, unsigned int
rows, unsigned int cols) {
	int index = blockDim.x*blockIdx.x + threadIdx.x;
	if (index < rows*cols) res[index] = a[index] * diag[index%cols];
}

__global__ void updateSignAC(datatype *AC, datatype *normAC, unsigned int row, unsigned
int col) {
	int index = blockDim.x*blockIdx.x + threadIdx.x;
	if (index < row*col) {
		int y = index % col;
		AC[index] = AC[index] * normAC[y];
	}
}

__global__ void signDiag(datatype *array, unsigned int rows, unsigned int cols, datatype
*S, datatype *S2) {
	int pos = blockDim.x*blockIdx.x + threadIdx.x;
	if (pos >= rows*cols) return;
	int y = pos % cols;
	array[pos] = array[pos] * S[y] * S2[y];
}

////////////////////////////////////////////////
//            File: errorTypes.h              //
```

```
//////////////////////////////////////////////
#ifndef _errorTypes_
#define _errorTypes_

const char *curandGetErrorString (curandStatus_t);
const char* cublasGetErrorString (cublasStatus_t);
const char* cusolverGetErrorString (cusolverStatus_t);

#endif // !_errorTypes_

//////////////////////////////////////////////
//           File: errorTypes.cu             //
//////////////////////////////////////////////
#include <cusolverDn.h>
#include <curand.h>
#include "cublas_v2.h"

const char *curandGetErrorString (curandStatus_t error) {
      switch (error) {
            case CURAND_STATUS_SUCCESS: return "CURAND_STATUS_SUCCESS";
            case CURAND_STATUS_VERSION_MISMATCH: return "CURAND_STATUS_VERSION_MISMATCH";
            case CURAND_STATUS_NOT_INITIALIZED: return "CURAND_STATUS_NOT_INITIALIZED";
            case CURAND_STATUS_ALLOCATION_FAILED: return
"CURAND_STATUS_ALLOCATION_FAILED";
            case CURAND_STATUS_TYPE_ERROR: return "CURAND_STATUS_TYPE_ERROR";
            case CURAND_STATUS_OUT_OF_RANGE: return "CURAND_STATUS_OUT_OF_RANGE";
            case CURAND_STATUS_LENGTH_NOT_MULTIPLE: return
"CURAND_STATUS_LENGTH_NOT_MULTIPLE";
            case CURAND_STATUS_DOUBLE_PRECISION_REQUIRED: return
"CURAND_STATUS_DOUBLE_PRECISION_REQUIRED";
            case CURAND_STATUS_LAUNCH_FAILURE: return "CURAND_STATUS_LAUNCH_FAILURE";
            case CURAND_STATUS_PREEXISTING_FAILURE: return
"CURAND_STATUS_PREEXISTING_FAILURE";
            case CURAND_STATUS_INITIALIZATION_FAILED: return
"CURAND_STATUS_INITIALIZATION_FAILED";
            case CURAND_STATUS_ARCH_MISMATCH: return "CURAND_STATUS_ARCH_MISMATCH";
            case CURAND_STATUS_INTERNAL_ERROR: return "CURAND_STATUS_INTERNAL_ERROR";
      }
      return "UNKNOWN ERROR";
}

const char* cublasGetErrorString (cublasStatus_t status) {
      switch (status) {
            case CUBLAS_STATUS_SUCCESS: return "CUBLAS_STATUS_SUCCESS";
            case CUBLAS_STATUS_NOT_INITIALIZED: return "CUBLAS_STATUS_NOT_INITIALIZED";
            case CUBLAS_STATUS_ALLOC_FAILED: return "CUBLAS_STATUS_ALLOC_FAILED";
            case CUBLAS_STATUS_INVALID_VALUE: return "CUBLAS_STATUS_INVALID_VALUE";
            case CUBLAS_STATUS_ARCH_MISMATCH: return "CUBLAS_STATUS_ARCH_MISMATCH";
            case CUBLAS_STATUS_MAPPING_ERROR: return "CUBLAS_STATUS_MAPPING_ERROR";
            case CUBLAS_STATUS_EXECUTION_FAILED: return "CUBLAS_STATUS_EXECUTION_FAILED";
            case CUBLAS_STATUS_INTERNAL_ERROR: return "CUBLAS_STATUS_INTERNAL_ERROR";
      }
      return "UNKNOWN ERROR";
}


const char* cusolverGetErrorString (cusolverStatus_t status) {
      switch (status) {
            case CUSOLVER_STATUS_SUCCESS: return "CUSOLVER_STATUS_SUCCESS";
            case CUSOLVER_STATUS_NOT_INITIALIZED: return
"CUSOLVER_STATUS_NOT_INITIALIZED";
            case CUSOLVER_STATUS_ALLOC_FAILED: return "CUSOLVER_STATUS_ALLOC_FAILED";
            case CUSOLVER_STATUS_INVALID_VALUE: return "CUSOLVER_STATUS_INVALID_VALUE";
            case CUSOLVER_STATUS_ARCH_MISMATCH: return "CUSOLVER_STATUS_ARCH_MISMATCH";
```

```
		case CUSOLVER_STATUS_EXECUTION_FAILED: return "CUSOLVER_STATUS_execution failed";
		case CUSOLVER_STATUS_MATRIX_TYPE_NOT_SUPPORTED: return "CUSOLVER_STATUS_Matrix not supported";
		case CUSOLVER_STATUS_INTERNAL_ERROR: return "CUSOLVER_STATUS_INTERNAL_ERROR";
	}
	return "UNKNOWN ERROR";
}
////////////////////////////////////////////////
//          File: declarations.h              //
////////////////////////////////////////////////
#ifndef _declarations_
#define _declarations_

	#include "cuda_runtime.h"
	#include "device_launch_parameters.h"

	#define MODE 1  // 0 for float, 1 for double

	#if (MODE)
		typedef double datatype;
	#else
		typedef float datatype;
	#endif

	#define prefix "./input/input" // input should be placed in the folder input with name inputk.txt for the kth slab, k=1,2,3,...
	#define show_fit 100  // Show fit every 'show_fit' iterations
	#define conv_crit 1e-7      // Convergence criterion
	#define max_iter 1000 // Maximal number of iterations
	#define sources 8           // The number of components (sources) to extract
	#define sizeX 125000        // Number of rows of each slab of tensor
	#define sizeZ 3             // Number of slabs
	// The other dimension may actually vary in different slabs
	#define MINIMUM(a,b) (a < b ? a:b) // Macro for minimum of two numbers
	#define minSize MINIMUM(sizeX, sizeZ) // Minimum of two fixed dimensions
	#define dimensions 3  // Amount of dimensions
	#define eps 2.2204e-16      //  Floating-point relative accuracy : the distance from 1.0 to the next largest double-precision number

#endif

////////////////////////////////////////////////
//          File: customvector.h              //
////////////////////////////////////////////////
#ifndef _customvector_
#define _customvector_

struct CustomVector { // Struct to hold a slab because the third dimension is uknown at compile time
	int   size;            // size of each row of 2D slab
	datatype *array; // serialized slab
};

#endif // !__customvector__
```

ΑΝΑΦΟΡΕΣ

[1] S. Theodoridis, Machine Learning – A Bayesian and Optimization Perspective, Academic Press – Elsevier, 2015, pp. 233-272, 403-444, 937-1003

[2] A. Smilde, R. Bro and P.Geladi, Multi-way Analysis in Chemistry and Related Field, 2004 John Wiley & Sons, Ltd

[3] M. M. Moreno, Y. Kopsinis, E. Kofidis, C. Chatzichristos and S. Theodoridis, Assisted Dictionary Learning For FMRI Data Analysis, Oct. 2016

[4] J. Sanders and E. Kandrot, CUDA by example: an introduction to general purpose GPU programming, 2011, Nvidia Corporation, Addison – Wesley, pp. 1-114

[5] S. Ferdowsi, V. Abolghasemi and S. Sanei, A new informed tensor factorization approach to EEG-fMRI fusion, JOURNAL OF NEUROSCIENCE METHODS, vol. 254, pp. 27-35

[6] J. Mairal et al, “Sparse learned representations for image restoration”, 4th World Conf. Comp. Stat. (LASC-ARS 2008), pp. 1-10

[7] R. Bro, PARAFAC. Tutorial and applications. Chemometrics and Intelligent Laboratory Systems, vol.38, 1997, pp. 149 – 171

[8] C. Chatzichristos, E. Kofidis, S. Theodoridis, “PARAFAC2 and its Block Term Decomposition Analog for Blind fMRI Source unmixing”, European Signal Processing Conference (EUSIPSCO), Aug 2017

[9] Δ. Αγρανιώτης και Α. Σβίγκος, «Στατιστική ανάλυση σε fMRI δεδομένα: εφαρμογή και ανάλυση της μεθόδου ICA», Πτυχιακή Εργασία, Τμήμα Πληροφορικής & Τηλεπικοινωνιών, Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών, 2015

[10] I. Tošić and P. Frossard, “Dictionary Learning”, Signal Processing Magazine, IEEE, vol. 28, pp. 27-38

[11] A. Leclaire and M. Lebrun, An implementation and detailed analysis of the K-SVD image denoising algorithm, Image Processing On Line, 2012, pp. 96-133

[12] M. Elad, Sparse and Redundant Representation: From Theory to Applications in Signal Processing, 2010, Springer Publishing Company, pp. 36-39, 48-51, 55-71, 228-236, 254-262

[13] Y. Kopsinis, H. Georgiou, S. Theodoridis, “FMRI unmixing via properly adjusted dictionary learning”. European Signal Processing Conference (EUSIPCO), Nov 2014

[14] H. Kiers, J.M.F. Ten Berge, R. Bro, “PARAFAC2 – PART. I A DIRECT FITTING ALGORITHM FOR PARAFAC MODEL”, Journal of Chemometrics, vol. 13, 1999, pp. 275-294

[15] T.G. Kolda, B.W. Bader, Tensor Decompositions and Applications, Journal SIAM Review, vol. 51, Aug 2009, pp. 455-500

[16] J. Li, J. Sun, Y. Song, Y. Xu and J. Zhao, “Accelarating the reconstruction of magnetic resonance imaging by three-dimensional dual-dictionary learning using CUDA”, 2014, 36th Annual International Conference of the IEEE Engineering in Medicine and Biology Society, Chicago, IL, 2014, pp. 2412-2415.

[17] M. Elad, R. Rubinstein and M. Zibulevsky, “Efficient Implementation of the K-SVD Algorithm and the Batch-OMP Method”, CS Technion, 40, 2008.

[18] Nvidia, CUDA C Programming Guide – Design Guide, version 8, 2017, Nvidia Corporation, Addison – Wesley, pp. 1-14, 17-29, 89-91

[19] cuBLAS Library – User Guide, no. DU-06702-001_v8.0, Nvidia Corporation, 2016

[20] Nvidia, CURAND LIBRARY, Programming Guide, version 8, Feb 2016, Nvidia Corporation, Addison-Wesley, pp. 2-11, 37-62

[21] Nvidia, CUSOLVER LIBRARY, Jun 2017, Nvidia Corporation, Addison-Wesley, pp. 1-12, 16-69, 189-210

[22] S.S. Cho, GPU Programming Lecture Notes, Lecture 5, CUDA Thread Basics, 2011, Departments of Computer Science and Physics, Wake Forest University. http://users.wfu.edu/choss/CUDA/docs/Lecture%205.pdf

[23]]T.Rowland and E.W. Weisstein, "Tensor." From MathWorld--A Wolfram Web Resource”. http://mathworld.wolfram.com/Tensor.html [Προσπελάστηκε 10/8/2017]

[24] Quora, Question and Answers: What is a tensor. https://www.quora.com/What-is-a-tensor [Προσπελάστηκε 13/8/2017]

[25] L. Albera, Introduction to Blind Source Seperation, Universite de Rennes, https://perso.univ-rennes1.fr/laurent.albera/alberasiteweb/bss.html [Προσπελάστηκε 13/8/2017]

[26] Δ. Καραγιαννάκης, «Αριθμητική Γραμμική Άλγεβρα», Ιανουάριος 2015 http://www.disigma.gr/media/blfa_files/chapter_ARITHMITIKH_GRAMMIKH_ALGEBRA.pdf [Προσπελάστηκε 18/8/2017]

[27] S. Gilbert, Γραμμική Άλγεβρα και Εφαρμογές, Πανεπιστημιακές Εκδόσεις Κρήτης, Οκτώβριος 2016 http://ph113.edu.physics.uoc.gr/files/113-lexiko.pdf [Προσπελάστηκε 21/8/2017]

[28] Definition of Voxel, May 2007, http://whatis.techtarget.com/definition/voxel [Προσπελάστηκε 23/8/2017]

[29] Sophia Learning Tutorials, Matrix Decomposition, https://www.sophia.org/tutorials/matrix-decomposition [Προσπελάστηκε 24/8/2017]
[30] I. Yanovsky QR Decomposition with Gram-Schmidt, UCLA, Department of Mathematics, http://www.math.ucla.edu/~yanovsky/Teaching/Math151B/handouts/GramSchmidt.pdf [Προσπελάστηκε 25/8/2017]
[31] L. Vandenberghe, QR Factorization, UCLA Engineering, http://www.seas.ucla.edu/~vandenbe/133A/lectures/qr.pdf [Προσπελάστηκε 25/8/2017]
[32] E.W. Weisstein, “Singular Value Decomposition.” From MathWorld – A Wolfram Web Resource http://mathworld.wolfram.com/SingularValueDecomposition.html [Προσπελάστηκε 26/8/2017]
[33] Rosetta Code, Cholesky decomposition, https://rosettacode.org/wiki/Cholesky_decomposition [Προσπελάστηκε 26/8/2017]
[34] J. Larkin, Fast GPU Development with CUDA Libraries, Nvidia Corporation, 2012. https://www.olcf.ornl.gov/wp-content/uploads/2013/02/GPU_libraries-JL.pdf [Προσπελάστηκε 3/9/2017]
[35] D. Kirk and W.M. Hwu, CUDA Threading Model, Summer School 2008: Accelarators for Science and Engineering Applications: GPUs and Multicores (GLCPC), 2008. http://www.greatlakesconsortium.org/events/GPUMulticore/Chapter3-CudaThreadingModel.pdf [Προσπελάστηκε 4/9/2017]
[36] A. Chrzeszczyk and J. Chrzeszczyk, Matrix Computations on the GPU, Cublas and Magma by example, NVIDIA Corporation, Aug 2013, https://developer.nvidia.com/sites/default/files/akamai/cuda/files/Misc/mygpu.pdf [Προσπελάστηκε 6/9/2017]
[37] J. Demouth, Shuffle: Tips and Tricks, GPU Technology Conference (GTC), 2013, http://on-demand.gputechconf.com/gtc/2013/presentations/S3174-Kepler-Shuffle-Tips-Tricks.pdf [Προσπελάστηκε 8/9/2017]
[38] J.Luitjens, Faster Paraller Reductions on Keppler, Feb. 2014, https://devblogs.nvidia.com/parallelforall/faster-parallel-reductions-kepler/ [Προσπελάστηκε 8/9/2017]