Εθνικο Μετσοβιο Πολυτεχνειο

Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων

Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων

Βελτιστοποίηση διαχείρισης μνήμης στις GPU

Διπλωματικη Εργασια

του

ΑΡΙΣΤΟΜΕΝΗ ΘΕΟΔΩΡΙΔΗ

Επιβλέπων: Γεώργιος Γκούμας
Αναπληρωτής Καθηγητής Ε.Μ.Π.

Εργαστηριο Υπολογιστικων Συστηματων

Αθήνα, Νοέμβριος 2022

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Υπολογιστικών Συστημάτων

Βελτιστοποίηση διαχείρισης μνήμης στις GPU

Διπλωματικη Εργασια

του

ΑΡΙΣΤΟΜΕΝΗ ΘΕΟΔΩΡΙΔΗ

Επιβλέπων: Γεώργιος Γκούμας
Αναπληρωτής Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 3η Νοεμβρίου 2022.

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
.....	.....	.....
Γεώργιος Γκούμας	Νεκτάριος Κοζύρης	Διονύσιος Πνευματικάτος
Αναπλ. Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.

Αθήνα, Νοέμβριος 2022

(Υπογραφή)

.....

Αριστομενης Θεοδωριδης | AM:03115632
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Η κατασκευή προγραμμάτων που εκτελούνται με καλή απόδοση στις μονάδες επεξεργασίας γραφικών (GPUs) είναι μια διαδικασία που απαιτεί αρκετό χρόνο και προσπάθεια από τον προγραμματιστή. Ο πολλαπλασιασμός πινάκων είναι ένα συνηθισμένο κομμάτι προγραμμάτων που μπορεί να επιταχυνθεί πολύ αν εκτελεστεί σε GPU. Στα επιστημονικά προγράμματα οι πίνακες που πολλαπλασιάζονται είναι συχνά μεγάλων διαστάσεων δυσχεραίνοντας περισσότερο τον υπολογισμό τους, καθώς απαιτείται καλή διαχείρηση της μνήμης των GPUs και κατάλληλη χρονοδρομολόγηση των παράλληλων διεργασιών. Για την βελτίωση της απόδοσης των επιστημονικών προγραμμάτων δημιουργήθηκε η BLAS και στη συνέχεια αναπτύχθηκαν διάφορες βιβλιοθήκες για παραλληλοποίηση των BLAS με πολλές GPUs. Οι περισσότερες από αυτές μελετούν την βελτιστοποίηση αυτών των προγραμμάτων εστιάζοντας κυρίως στην αποδοτική χρονοδρομολόγηση.

Στα πλαίσια της συγκεκριμένης διπλωματικής εργασίας αναπτύχθηκε ένα εργαλείο που στοχεύει να βοηθήσει στη διαχείρηση της μνήμης κατά τον πολλαπλασιασμό πινάκων σε συστήματα με περισσότερες από μία GPUs, μειώνοντας την μνήμη που απαιτείται για αποδοτική εκτέλεση. Το εργαλείο αυτό ονομάζεται Software-assisted Memory Buffer και παρέχει δυο αντικείμενα στον προγραμματιστή, με τη χρήση των οποίων παύει να χρειάζεται να ελευθερώνει μνήμη ο προγραμματιστής και μπορεί να εστιάσει σε καλύτερη χρονοδρομολόγηση των διεργασιών. Επίσης, κατασκευάστηκε μια συνάρτηση με σκοπό την πρόβλεψη των μεταβλητών του Software-assisted Memory Buffer που μειώνουν την απαραίτητη μνήμη κρατώντας σταθερή την απόδοση του προγράμματος. Στο τέλος έγινε αξιολόγηση του Software-assisted Memory Buffer μέσω μιας σειράς από μετρήσεις. Επιβεβαιώθηκε ότι με τη χρήση του μπορεί να μειωθεί η μνήμη που χρησιμοποιείται διατηρώντας σταθερή την απόδοση. Οι διαφορετικές πολιτικές αντικατάστασης μπλοκ που υλοποιήθηκαν στον Software-assisted Memory Buffer παρουσίασαν καλύτερη απόδοση σε περιπτώσεις που χρησιμοποιείται περιορισμένη μνήμη.

Λέξεις Κλειδιά

Μονάδες επεξεργασίας γραφικών, BLAS, Διαχείρηση μνήμης, Υπολογιστικά συστήματα υψηλων επιδόσεων

Abstract

For a developer, writing an efficient program running on GPUs is time and effort consuming process. Matrix multiplication is a common part of many programs which can be sped up by running on a GPU. Matrices being multiplied in scientific code usually have large dimensions, making it difficult to calculate, because they require good GPU memory management and proper scheduling of the parallel tasks. In order to improve the performance of scientific programs, BLAS was created and then many libraries followed which parallelize BLAS for many GPUs. Most of them focus on an efficient scheduler to optimize these programs.

During this thesis a tool was developed which aims to help with memory management while performing matrix multiplication on multi-GPU systems, decreasing the memory needed for efficient execution. This tool is called Software-assisted Memory Buffer and provides the developer with two objects. While using these objects, the developer does not need to care about freeing memory and he can focus on optimizing task scheduling. Also a function was developed to predict the variables with which the Software-assisted Memory Buffer reduces the memory required while having the same performance. In the end, Software-assisted Memory Buffer was evaluated through a series of measurements. It was confirmed that, while using the tool, the memory used can be reduced and the performance remains stable. Also, the different schedule out policies implemented in the Software-assisted Memory Buffer showed better performance in cases of reduced memory.

Keywords

Graphics processing units, BLAS, Memory managment, High performance computing

Ευχαριστίες

Η συγκεκριμένη διπλωματική εργασία ολοκληρώνει ένα πολύ σημαντικό κύκλο σπουδών αλλά και μια σημαντική περίοδο της ζωής μου που ξεκίνησε τον Ιανουάριο του 2016. Κατά τη διάρκεια αυτής της χρονικής περιόδου αλλάξαν πολλά και θα ήθελα να ευχαριστήσω όλα τα άτομα που συντέλεσαν στην ολοκλήρωσή του.

Η εργασία εκπονήθηκε στο Εργαστήριο Υπολογιστικών Συστημάτων. Θα ήθελα να ευχαριστήσω πρώτα τον υπεύθυνό μου, Αναπληρωτή Καθηγητή Γεώργιο Γκούμα, τόσο για την ευκαιρία που μου έδωσε να ασχοληθώ με το συγκεκριμένο θέμα όσο και για τις γνώσεις που μου μετέφερε σε όλα τα μαθήματά του. Θα ήθελα να ευχαριστήσω ιδιαιτέρως τον Υποψήφιο Διδάκτορα Πέτρο Αναστασιάδη, ο οποίος με καθοδήγησε και με βοήθησε σε όλη τη διάρκεια και σε όλα τα στάδια αυτής της εργασίας.

Τέλος, δε γίνεται να μην ευχαριστήσω την οικογένεια και τους φίλους μου, οι οποίοι ήταν δίπλα μου καθόλη τη διάρκεια της περιοόδου αυτής, για τη βοήθειά τους, τη συμπαράστασή τους και την πάντα ευγενική τους παρότρυνση.

Περιεχόμενα

Κατάλογος Σχημάτων

Κατάλογος Πινάκων

Κεφάλαιο 1

Εισαγωγή

1.1 Αντικείμενο της διπλωματικής εργασίας

Οι βασικές υπολογιστικές υπορουτίνες γραμμικής άλγεβρας(**BLAS** - Basic Linear Algebra Subprograms) είναι ένα σύνολο από χαμηλού επιπέδου ρουτίνες για την εκτέλεση βασικών πράξεων γραμμικής άλγεβρας που είναι αρκετά διαδεδομένες σε εφαρμογές μηχανικής μάθησης, προσωμοιώσεων και άλλων επιστημονικών εφαρμογών. Η BLAS βιβλιοθήκη συνδυάζει καλές αριθμητικές μεθόδους για τον ακριβή υπολογισμό προβλημάτων σε συγκεκριμένους τομείς. Προγράμματα που εκτελούνται σε υπολογιστικά συστήματα υψηλών επιδόσεων (**HPC** - High Performance Computing) συχνά περιέχουν πολλούς και πυκνούς υπολογισμούς γραμμικής άλγεβρας, γι΄ αυτό και η BLAS είναι αρκετά διαδεδομένη στον επιστημονικό κώδικα. Η μεγάλη και οικονομική υπολογιστική ικανότητα των GPUs, που τα τελευταία χρόνια χρησιμοποιούνται ως επιταχυντές στα HPC συστήματα, μπορεί να αξιοποιήσει τον παραλληλισμό των BLAS ρουτινών και ως αποτέλεσμα έχουν δημιουργήθεί αρκετές GPU-BLAS βιβλιοθήκες. Η παρούσα διπλωματική εργασία στοχεύει στο να παρουσιάσει ένα εργαλείο που θα βοηθήσει τους προγραμματιστές να αξιοποιήσουν καλύτερα τη μνήμη των GPUs σε συστήματα με πολλές GPU. [2][5][13]

1.2 Οργάνωση του τόμου

Ο τόμος οργανώνεται ως εξής. Στο κεφάλαιο 2 παρουσιάζεται η BLAS . Στο κεφάλαιο 3 παρουσιάζεται ο Software-assisted Memory Buffer, το εργαλείο που υλοποιήθηκε στα πλαίσια της διπλωματικής αυτής εργασίας με σκοπό να βοηθήσει τους προγραμματιστές να αξιοποιήσουν καλύτερα τη μνήμη των GPUs. Στο κεφάλαιο 4 αναλύονται τα πειράματα που έγιναν για την απόδοση ενός προγράμματος που χρησιμοποιεί το εργαλείο. Τέλος στο κεφάλαιο 5 συνοψίζεται η εργασία και προτείνονται πιθανές επεκτάσεις του εργαλείου.

Κεφάλαιο 2

Θεωρητικό Υπόβαθρο

2.1 Εισαγωγή

Πριν παρουσιαστεί το εργαλείο, είναι χρήσιμο να τεθεί μια θεωρητική βάση πάνω στην οποία χτίστηκε όλη η εργασία. Αρχικά θα παρουσιαστεί η BLAS, οι υπορουτίνες της οποίας αποτελούν μέορς πολλών επιστημονικών προγραμμάτων. Το εργαλείο που θα παρουσιαστεί παρακάτω στοχεύει στην επιτάχυνση των υπορουτινών της. Επίσης, ο αναγνώστης είναι απαραίτητο να έχει βασικές γνώσεις της δομής και λειτουργίας μιας κάρτας γραφικών, καθώς και στον τρόπο προγραμματισμού τους. Ακόμα, θα παρουσιαστούν οι πολιτικές αντικατάστασης μπλοκ στη μνήμη που χρησιμοποιήθηκαν. Τέλος, πριν την παρουσίαση του εργαλείου θα γίνει αναφορά σε κάποιες multi-GPU βιβλιοθήκες, οι οποίες μελετήθηκαν για τη δημιουργία του εργαλείου.

2.2 BLAS

Η αριθμητική γραμμική άλγεβρα, και ιδίως η επίλυση γραμμικών συστημάτων, γραμμικών προβλημάτων ελαχίστων τετραγώνων και προβλημάτων ιδιοτιμών, είναι θεμελιώδης για την επίλυση των περισσοτέρων επιστημονικών υπολογισμών και συχνά αποτελεί το βαρύτερο υπολογιστικά τμήμα των υπολογισμών. Επειδή ένα μεγάλο μέρος του χρόνου εκτέλεσης σε περίπλοκα προγράμματα γραμμικής άλγεβρας μπορεί να καταναλωθεί σε μερικές χαμηλού επιπέδου λειτουργίες, η μείωση του χρόνου εκτέλεσης αυτών των λειτουργιών οδηγεί σε μείωση του συνολικού χρόνου εκτέλεσης του προγράμματος. Η πρώτη σημαντική μαζική προσπάθεια για μια γενική συμφωνία για τον προσδιορισμό ενός συνόλου υπορουτινών γραμμικής άλγεβρας οδήγησε στη δημιουργία των βασικών υπολογιστικών υπορουτινών γραμμικής άλγεβρας(**BLAS**-**B**asic **L**inear **A**lgebra **S**ubprograms).[3]

Υπάρχουν τρία επίπεδα BLAS. Το πρώτο επίπεδο αφορά πράξεις που γίνονται μεταξύ διανυσμάτων και ήταν και το πρώτο για το οποίο υλοποιήθηκαν προδιαγραφές. Το δεύτερο επίπεδο είναι πράξεις μεταξύ πίνακα και διανύσματος και το τρίτο είναι μεταξύ πινάκων. Οι προδιαγραφές αυτών βοήθησαν στην κατασκευή νέων λογισμικών που αξιοποιούν αποδοτικότερα τις ιεραρχίες μνήμης των σημερινών υπολογιστών. Η BLAS βοήθησε επίσης και στη δημιουργία

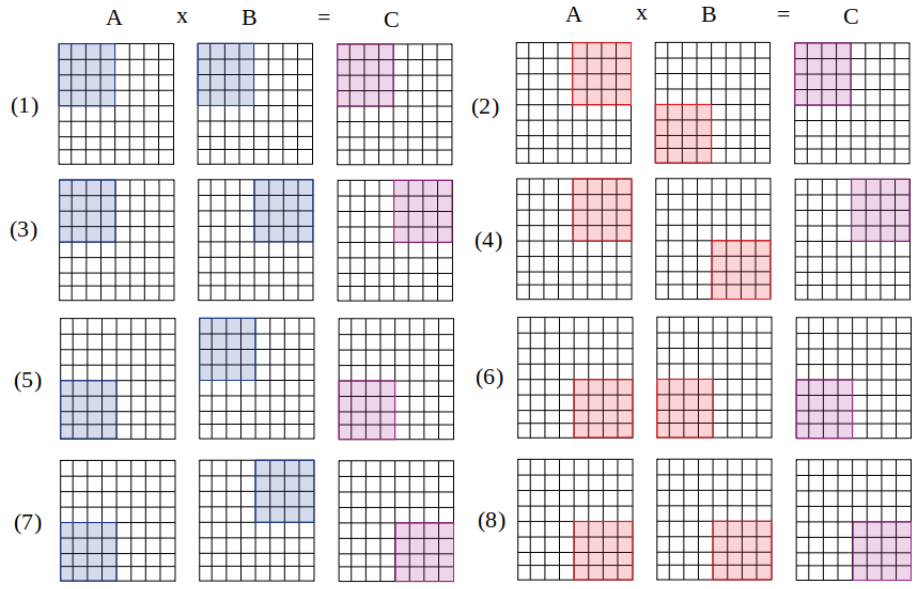
φορητών προγραμμάτων, καθώς είναι υλοποιημένη για υπολογιστές με διαφορετική αρχιτεκτονική. Ο συνδυασμός αυτών έδωσε την ικανότητα στους επιστήμονες να κατασκευάσουν πολύπλοκες προσομοιώσεις μεγάλης κλίμακας.

Η συχνή παραλληλοποιησιμότητα των BLAS ρουτινών τις κάνει ιδανικές για εκτέλεση σε GPU. Η παραλληλοποιησιμότητα είναι δυνατή καθώς η BLAS μπορεί να διασπαστεί σε μικρότερες ανεξάρτητες διεργασίες. Η πιο συχνή υπορουτίνα της BLAS-3 είναι ο πολλαπλασιασμός πινάκων GEMM - General Matrix Multiplication, η οποία χρησιμποιείται και από άλλες υπορουτίνες και σε αυτή θα εστιάσει η παρούσα διπλωματική. Συγκεκριμένα θα χρησιμοποιηθεί η DGEMM που είναι ο πολλαπλασιασμός πινάκων διπλής ακρίβειας.

Στον πολλαπλασιασμό δυο πινάκων, έστω $A_{N\times K} \cdot B_{K\times M} = C_{N\times M}$, κάθε κελί του πίνακα C_{xy} υπολογίζεται από την εξίσωση:

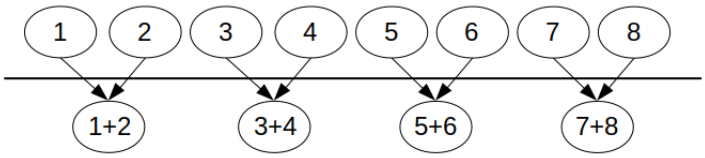
$$C_{xy} = A_{x1} \cdot B_{1y} + A_{x2} \cdot B_{2y} + ... + A_{xK} \cdot B_{Ky} \tag{2.1}$$

Το άθροισμα αυτό μπορεί να διασπαστεί σε κομμάτια και κάθε κομμάτι να υπολογιστεί ξεχωριστά. Στο τέλος αθροίζονται τα τμήματα και έχουμε το τελικό αποτέλεσμα. Επίσης κάθε κελί C_{xy} είναι ανεξάρτητο από τα υπόλοιπα, επομένως όλα τα κελιά μπορούν να υπολογιστούν παράλληλα μεταξύ τους. Με βάση αυτές τις δυο ιδιότητες, οι πίνακες μπορούν να χωριστούν σε πλακίδια και να υπολογιστεί παράλληλα ο πολλαπλασιασμός των πινάκων. Στο σχήμα 2.1 παρουσιάζεται ο πολλαπλασιασμός δυο πινάκων 8×8 με 4 πλακίδια. Οι πολλαπλασιασμοί 1,3,5,7 υπολογίζουν το πρώτο ημιάθροισμα κάθε πλακιδίου του πίνακα C, ενώ οι 2,4,6,8 υπολογίζουν το δεύτερο ημιάθροισμα.



Σχήμα 2.1: Πολλαπλασιασμός πινάκων με πλακίδια.

Στο σχήμα 2.2 παρουσιάζονται οι πολλαπλασιασμοί του σχήματος 2.1 ως διεργασίες. Σε κάθε επίπεδο όλες οι διεργασίες μπορούν να γίνουν παράλληλα. Η παραλληλοποιησιμότητα αυτή επεκτείνεται σε μεγαλύτερους πίνακες με περισσότερα πλακίδια. Συνεπώς, οι υπορουτίνες



Σχήμα 2.2: Πολλαπλασιασμός πινάκων με πλακίδια.

της BLAS-3 μπορούν να εκμεταλευτούν αυτή την παραλληλοποιησιμότητα για να εκτελέσουν, σε συστήματα με περισσότερους από έναν επεξεργαστές, τον πολλαπλασιασμό σε πολύ μικρότερο χρόνο.

2.3 Επιταχυντές για την επεξεργασία γραφικών

Οι μονάδες επεξεργασίας γραφικών ή αλλιώς **GPU**s (**G**raphics **P**rocessing **U**nits) έχουν εξελιχθεί από καλωδιωμένοι, περιορισμένων δυνατοτήτων ελεγκτές VGA σε πολυεπεξεργαστές υψηλής παραλληλίας και υψηλής πολυνημάτωσης βελτιστοποιημένοι για οπτική υπολογιστική (visual computing). Στη σύγχρονη εποχή βρίσκονται σε κάθε προσωπικό, φορητό, και επιτραπέζιο υπολογιστή και σταθμό εργασίας το οποίο τις κάνει διαθέσιμες σε πολλούς. Όταν οι προοπτικές τους αυτές συνδυάστηκαν με μια γλώσσα προγραμματισμού που κάνει τον προγραμματισμό τους εύκολο, τράβηξαν το ενδιαφέρον πολλών προγραμματιστών. Έτσι άρχισαν να συνδυάζονται με τις CPUs δημιουργώντας ετερογεννή συστήματα, επιταχύνοντας την εκτέλεση πολλών εφαρμογών επιστημονικού σκοπού.[12]

2.3.1 GPU και CPU

Ένας από τους λόγους που οι GPUs βοηθούν στην επιτάχυνση εφαρμογών όταν συνεργάζοντα με τις CPUs είναι η διαφορετική αρχιτεκτονική της GPU από τη CPU. Η CPU με λίγους πυρήνες και μεγάλη κρυφή μνήμη, είναι γρήγορη και προσαρμώσιμη (versatile). Εκτελεί πολλές και διαφορετικές διεργασίες, εναλλάσοντας συνεχώς από τη μια στην άλλη, οι οποίες συνήθως αλληλεπιδρούν συχνά με τα υπόλοιπα συστήματα του υπολογιστή. Για παράδειγμα μπορεί να χρειάζονται πρόσβαση στο σκληρό δίσκο για να παίρνουν τα δεδομένα που ο χρήστης πληκτρολογεί. Αντίθετα, η GPU με εκατοντάδες πυρήνες και λίγη μνήμη, χωρίζει ένα σύνθετο πρόγραμμα σε χιλιάδες διεργασίες, οι οποίες εκτελούνται ταυτόχρονα. Πολλές από αυτές εκτελούν τον ίδιο κώδικα αλλά σε διαφορετικά δεδομένα. Η ικάνότητα αυτή της GPU την καθιστά ιδανική για τον υπολογισμό γραφικών, υφών (textures), φωτισμού (lighting) και άλλων παρόμοιων διεργασιών που πρέπει να υπολογίζονται στιγμιαία για να συνεχίζεται η συνεχής ροή εικόνων στην οθόνη. [11][12]

Συνδυάζοντας τις GPUs με τις CPUs προκύπτουν ετερογεννή συστήματα τα οποία έχουν καλύτερη απόδοση στην εκτέλεση παραλληλοποιήσιμων εφαρμογών όπως η μηχανική μάθηση, προσωμοιώσεις κ.λ.π.. Γι΄ αυτό το λόγο τα τελευταία χρόνια οι GPUs άρχισαν να τοποθετούνται και σε υπολογιστικά συστήματα υψηλών επιδόσεων (HPC- High Performance Comput-

CPU	GPU
Λίγοι πυρήνες	Εκατοντάδες πυρήνες
Μεγάλη χωρητικότητα κύριας μνήμης	Περιορισμένη χωρητικότητα κύριας μνήμης
Καλή για σειριακή εκτέλεση	Καλή για υψηλή παραλληλοποίηση

Πίνακας 2.1: Διαφορές των CPUs από τις GPUs.

ing) ως επιταχυντές. Η ανάπτυξη της χρήσης των GPUs οφείλεται επίσης στην ανάπτυξη της Αρχιτεκτονικής Συσκευών Ενοποιημένων Υπολογισμών (Compute Unified Device Architecture - **CUDA**) της NVIDIA που είναι ένα επεκτάσιμο μοντέλο παράλληλου προγραμματισμού και πλατφόρμα λογισμικού για την GPU και άλλους παράλληλους επεξεργαστές, το οποίο επιτρέπει στον προγραμματιστή να παρακάμψει το API και τις διασυνδέσεις γραφικών της GPU και να προγραμματίζει απλώς σε C ή C++.

2.3.2 Προγραμματισμός σε GPU

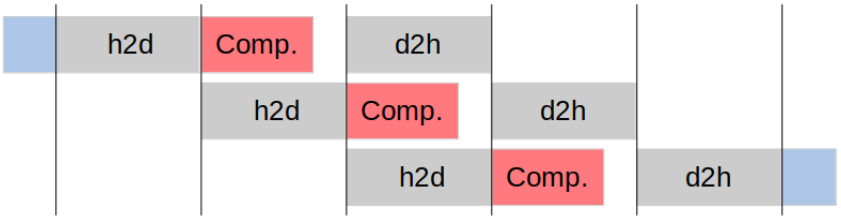
Τα τελευταία χρόνια έχουν αναπτυχθεί διάφορα framework για τον προγραμματισμό των GPUs, όπως το OpenCL και η CUDA. Η εργασία αυτή εστιάζει στα χαρακτηριστικά που έχουν οι υλοποιήσεις με CUDA καθώς στα HPC συστήματα χρησιμοποιούνται κυρίως NVIDIA GPUs. Παρόλα αυτά, τα υπόλοιπα framework δεν διαφέρουν σημαντικά.

Κατά την εκτέλεση ενός προγράμματος σε ένα ετερογενές σύστημα, συμμετέχουν η CPU (ως host) και η GPU (ως device). Η εκτέλεση του προγράμματος ξεκινάει στο host, ο οποίος κάνει τις απαραίτητες ενέργειες (δέσμευση μνήμης, μεταφορά δεδομένων) για να ξεκινήσει η εκτέλεση στη GPU και στη συνέχεια καλεί μια συνάρτηση, που είναι ορισμένη για να τρέχει σε GPU, η οποία καλείται συνήθως kernel. Ο κώδικας του kernel είναι σειριακός και κάθε νήμα εκτελεί αυτόν τον σειρακό κώδικα. Συνεπώς, ένα απλό πρόγραμμα που τρέχει σε GPU ακολουθεί τα εξής βήματα:

1. Δέσμευση αναγκαίας μνήμης στην μνήμη της GPU.
2. Μεταφορά δεδομένων από τη μνήμη του host(CPU) στη μνήμη του device(GPU) (h2d).
3. Κλήση του kernel.
4. Μεταφορά δεδομένων από τη μνήμη του device στη μνήμη του host (d2h).
5. Αποδέσμευση μνήμης στο device.

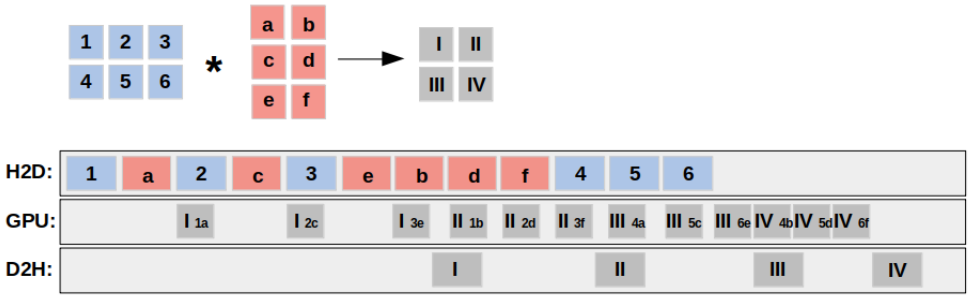
Σχήμα 2.3: Απλό πρόγραμμα εκτέλεσης σε GPU.

Στο σχήμα 2.3 φαίνεται σχηματικά η εκτέλεση. Είναι εμφανές από το σχήμα ότι η μεταφορά δεδομένων από και προς τη GPU παίρνει αρκετό χρόνο συγκριτικά με το χρόνο εκτέλεσης του kernel. Επίσης μπορεί τα δεδομένα που χρειάζονται για να την εκτέλεση να πιάνουν περισσότερο χώρο από την διαθέσιμη μνήμη στη μνήμη της GPU. Για να αντιμετωπιστούν αυτές οι περιπτώσεις, ο kernel διασπάται σε μικρότερους, οι οποίοι εκτελούνται σειριακά. Η διάσπαση αυτή επιτρέπει να ξεκινήσει η εκτέλεση του πρώτου kernel και ταυτόχρονα να μεταφέρονται τα δεδομένα που χρειάζονται για τον δεύτερο kernel. Αυτός ο τρόπος εκτέλεσης είναι ο πιο διαδεδομένος και φαίνεται στο σχήμα 2.4.



Σχήμα 2.4: Σύνηθες τρόπος εκτέλεσης σε GPU.

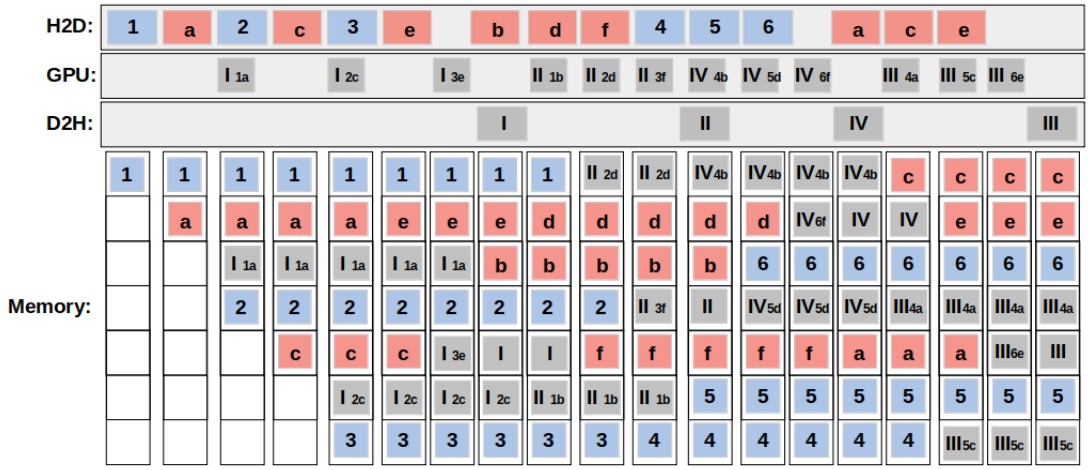
Ο τρόπος εκτέλεσης αυτός λέγεται 3-way concurrency overlap καθώς επικαλύπτονται οι h2d και d2h μεταφορές δεδομένων με την εκτέλεση. Το σχήμα 2.5 παρουσιάζει πώς θα πολλαπλασιάζονταν δυο πίνακες με την παραπάνω λογική. Στη συγκεκριμένη εκτέλεση θεωρείται ότι ολοι οι πίνακες χωράνε στη μνήμη της GPU. Στην H2D γραμμή φαίνονται οι μεταφορές των τμημάτων των πινάκων που πολλαπλασιάζονται από την μνήμη της CPU στη μνήμη της GPU, ενώ στην D2H γραμμή είναι οι μεταφορές των αποτελεσμάτων πίσω στην CPU. Η γραμμή GPU δείχνει πότε εκτελείται κάποιος kernel στη GPU. Σε κάθε εκτέλεση στη GPU, οι δείκτες δείχνουν ποια τμήματα των πινάκων πολλαπλασιάζονται μεταξύ τους. Θεωρείται ότι στο τέλος κάθε εκτέλεσης γίνεται και ο συνολικός υπολογισμός ενός τμήματος του πίνακα αποτελεσμάτων. Δηλαδή, κατά την εκτέλεση του I_{3e} γίνεται και η τελική άθροιση $I = I_{1a} + I_{2c} + I_{3e}$.



Σχήμα 2.5: 3-way concurrency overlap σε μια GPU.

Στο σχήμα 2.6 γίνεται ακριβώς η ίδια πράξη αλλά η μνήμη της GPU έχει μέγεθος όσο 7 στοιχεία του πίνακα και παρουσιάζονται οι αλλαγές που γίνονται στη μνήμη σε κάθε βήμα εκτέλεσης. Οι πρώτες τρεις γραμμές (H2D, GPU, D2H) δείχνουν ότι και στο σχήμα 2.5,

ενώ από κάτω παρουσιάζεται το περιεχόμενο της μνήμης της GPU σε κάθε αλλαγή. Κάθε στήλη δείχνει το περιεχόμενο της μνήμης εκείνη τη χρονική στιγμή. Αρχικά η εκτέλεση δεν έχει καμία διαφορά από το προηγούμενο παράδειγμα, μέχρι που ξεκινά να εκτελείται ο kernel I_{3e}. Σε αυτό το σημείο, η μνήμη έχει γεμίσει και όλα τα δεδομένα είναι απαραίτητα για τις επόμενες εκτελέσεις.Οι μεταφορές δεδομένων σταματάνε έως ότου τελειώσει η εκτέλεση του kernel και ελευθερούν οι θέσεις μνήμης των αποτελεσμάτων I_{1a} και I_{2c}. Μετά η εκτέλεση συνεχίζεται κανονικά μέχρι τον kernel IV_{6f} που συμβαίνει το ίδιο πρόβλημα. Τα τμήματα a, c, e του πίνακα πρέπει να ξαναμεταφερθούν, καθώς λόγω του περιορισμένου χώρου αφαιρέθηκαν. Συνεπώς, το πρόγραμμα χρειάζεται περισσότερο χρόνο για να εκτελεστεί και αρκετή δουλειά από τον προγραμματιστή ο οποίος πρέπει να διαχειριστεί αποδοτικά τη μνήμη. Σε πολλές επιστημονικές εφαρμογές είναι σύνηθες τα δεδομένα να υπερβαίνουν σε μέγεθος το διαθέσιμο χώρο στην GPU.



Σχήμα 2.6: 3-way concurrency overlap σε μια GPU με 7 στοιχεία μνήμη.

Γενικά, εκτός από το 3-way concurrency overlap που αναφέρθηκε, εφόσον η GPU που χρησιμοποιείται μπορεί να τις υποστηρίξει, η CUDA επιτρέπει τις εξής διαδικάσίες να γίνονται ταυτόχρονα:

- Εκτέλεση στο host.
- Εκτέλεση στο device.
- Μεταφορά δεδομένων από το host στο device.
- Μεταφορά δεδομένων από το device στο host.
- Μεταφορά δεδομένων μεταξύ διαφορετικών devices.
- Μεταφορά δεδομένων στο εσωτερικό ενός device.[1]

2.4 Επικάλυψη υπολογισμών και μεταφοράς δεδομένων

Οι μεταφορές μέσω PCIe μπορεί να έχουν μεγάλη επίπτωση στην απόδοση, ιδίως αν ληφθεί υπόψιν ότι το εύρος ζώνης είναι πολύ μικρότερο από το εύρος ζώνης της μνήμης των GPUs. Οι Gregg και Hazelwood [6] δηλώνουν ο χρόνος εκτέλεσης ενός GPU kernel μπορεί να αυξηθεί από 2 έως 50 φορές σε σχέση με τον αρχικό, όταν συμπεριλαμβάνονται και οι μεταφορές μέσω PCIe. Ο απλός κλασσικός τρόπος είναι να αναβάλεται η εκτέλεση όλων των υπολογισμών στην GPU έως ότου να τελειώσει η μεταφορά όλων των δεδομένων. Ένας σημαντικός λόγος που συμβαίνει αυτό είναι διότι η επικάλυψη μεταφοράς δεδομένων με εκτέλεση απαιτεί δουλεια από τον προγραμματιστή και οδηγεί σε σημαντική αύξηση του κώδικα [7], [8], [9], [15]. Όμως, όπως ήδη αναφέρθηκε, συχνά τα δεδομένα δε χωράνε στη μνήμη της GPU ή για να μειωθεί ο χρόνος μεταφοράς δεδομένων διασπάται ο kernel σε μικρότερους ώστε να ξεκινήσει νωρίτερα η εκτέλεση. Υπάρχουν διάφορες τεχνικές για την μεταφορά δεδομένων από και προς τη μνήμη της GPU και για την επικάλυψή των μεταφορών αυτών με εκτέλεση υπολογισμών όπως αναλύεται από τους Werkhoven, Maassen, Seinstra και Bal.[14]

Στη μελέτη που παρουσιάζεται σε αυτή την εργασία, δε χρησιμοποιείται κάποιο μοντέλο για τις μεταφορές δεδομένων. Το εργαλείο που υλοποιήθηκε έχει ως στόχο να βοηθήσει τον προγραμματιστή στη διαχείρηση της μνήμης και αυτό έχει σαν αποτέλεσμα να γίνονται και οι αντίστοιχες μεταφορές δεδομένων. Αν και δε μελετάται απευθείας, ο χρόνος που χάνεται για τη μεταφορά δεδομένων συμπεριλαμβάνεται στο συνολικό χρόνο εκτέλεσης ρίχνοντας την απόδοσοση σε περίπτωση που γίνονται υπερβολικά πολλές μεταφορές.

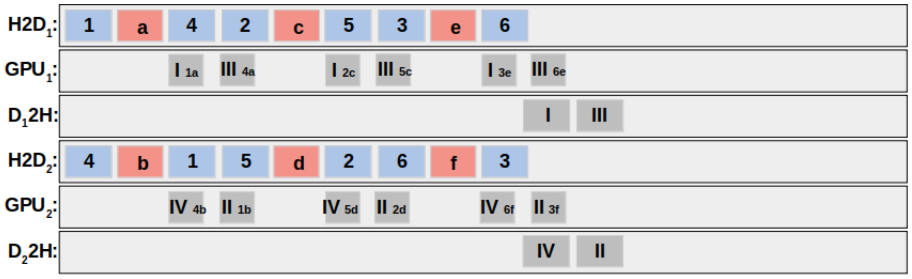
2.5 Πολιτικές αφαίρεσης μπλοκ

Το εργαλείο που παρούσιάζεται στην παρούσα διπλωματική μελετά τρεις διαφορετικές πολιτικές για την αφαίρεση μπλοκ από τη μνήμη των GPUs. Η πρώτη χρησιμοποιεί μια δομή πρώτης εισαγωγής - πρώτης αφαίρεσης (**FIFO** - First In First Out). Η FIFO είναι μια ουρά στην οποία εισάγονται στοιχεία από τη μία πλευρά και αφαιρούνται από την άλλη. Συνεπώς το πρώτο στοιχείο το οποίο εισάγεται στην ουρά είναι και το πρώτο που αφαιρείται. Η πολιτική που χρησιμοποιεί αυτή τη δομή εισάγει στη δομή τα μπλοκ όταν έρχονται στη μνήμη, ώστε αν γεμίσει να επιλεγεί το αρχαιότερο για να ελευθρωθεί μνήμη.[10]

Η δεύτερη πολιτική που χρησιμποιείται είναι η μέθοδος του λιγότερο πρόσφατα χρησιμοποιημένου (**LRU** - Least Recently Used). Στην μέθοδο αυτή, το μπλοκ που αντικαθίσταται είναι αυτό το οποίο έχει μείνει αχρησιμοποίητο για το μεγαλύτερο χρονικό διάστημα. Η αντικατάσταση αυτή υλοποιείται με την παρακολούθηση του πότε χρησιμποιήθηκε κάθε στοιχείο του συνόλου σε σχέση με τα υπόλοιπα στοιχεία του. [12] Η τρίτη πολιτική που χρησιμποιείται είναι η μέθοδος του πιο πρόσφατα χρησιμοποιημένου (**MRU** - Most Recently Used). Η μέθοδος αυτή διαφέρει από την LRU μόνο στο μπλοκ που επιλέγεται να αφαιρεθεί. Αντί για αυτό που έχει μείνει αχρησιμποιήτο, η MRU αφαιρεί αυτό που χρησιμποιήθηκε πιο πρόσφατα.

2.6 Multi-GPU Libraries

Πολλά συστήματα και ιδίως τα HPC συστήματα δεν έχουν μόνο μια GPU. Στις μεγάλες επιστημονικές εφαρμογές λόγω του όγκου των δεδομένων γίνεται χρήση πολλών GPU που δρουν ως επιταχυντές και τα συστήματα αποκαλούνται multi-GPU. Στο σχήμα 2.7 παρουσιάζεται ενδεικτική εκτέλεση αντίστοιχη με αυτή του σχήματος 2.5 αλλά με δυο GPUs. Εδώ με τη λογική του 3-way concurrency overlap υπάρχει κέρδος σε χρόνο σε σχέση με την εκτέλεση σε έναν επεξεργαστή.

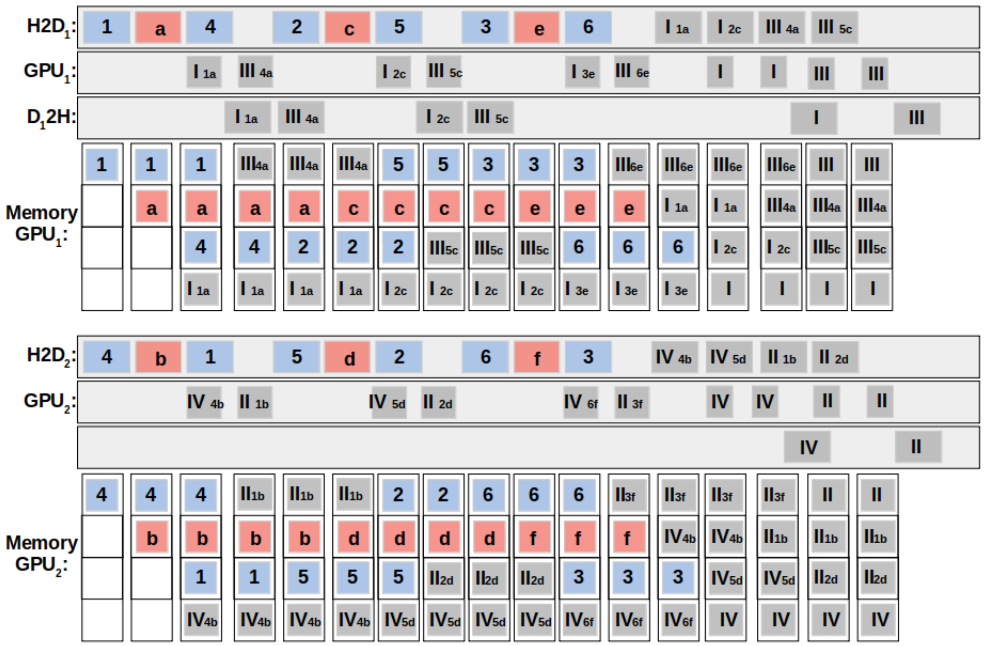


Σχήμα 2.7: 3-way concurrency overlap σε δυο GPU.

Αν μειωθεί η διαθέσιμη μνήμη των GPU στα τέσσερα στοιχεία, το κέρδος σε χρόνο δεν είναι πια τόσο πολύ και απαιτεί πολύ περισσότερη δουλειά από τον προγραμματιστή για τη σωστη διαχείρηση της μνήμης. Η εκτέλεση αυτή παρουσιάζεται ενδεικτικά στο σχήμα 2.8. Όπως και στο παράδειγμα του σχήματος 2.6, όταν η μνήμη γεμίσει, η μεταφορά δεδομένων σταματάει και περιμένει να τελειώσει η εκτέλεση για να ελευθερωθεί κάποια θέση μνήμης. Αυτό συμβάινει για παράδειγμα κατά την εκτέλεση του III_{4a} στη GPU1. Επίσης, στο τέλος της εκτέλεσης η GPU1 φέρνει πάλι τα I_{1a}, I_{2c}, III_{4a} και III_{5c} για να υπολογίσει το τελικό αποτέλεσμα κάθε κελιού. Ακριβώς με τον ίδιο τρόπο γίνεται και η εκτέλεση στην GPU2.

Παρόλα αυτά, και στα δυο σχήματα φαίνεται να υπάρχει αρκετή ώρα αρδάνειας των GPUs καθώς οι μεταφορές δεδομένων καθυστερούν και τα δεδομένα που αποστέλονται είναι σε μεγάλο βαθμό ίδια και στις δυο GPUs. Όπως αναφέρθηκε και στο κεφάλαιο 2.3.2, οι σημερινές GPUs είναι ικανές να μεταφέρουν δεδομένα μεταξύ τους παράλληλα με τις υπόλοιπες διεργασίες που εκτελούν. Η απόστολή δεδομένων μεταξύ GPUs είναι πολύ πιο γρήγορη από την h2d. Σε προβλήματα με πολύ περισσότερα δεδομένα αυτές οι αποστολές μπορούν να κάνουν τη διαφορά. Για αυτό το λόγο, η διαχείρηση της μνήμης αποτελεί ένα σημαντικό κομμάτι της επιτάχυνσης της εκτέλεσης των προγραμμάτων σε GPU.

Για να αντιμετώπιστούν τα παραπάνω προβλήματα και να βοηθηθούν οι προγραμματιστές, έχουν αναπτυχθεί διάφορες multi-GPU BLAS βιβλιοθήκες για την καλύτερη αξιοποίηση της μνήμης. Σε αυτή την εργασία αναλύονται τρεις διαφορετικές BLAS βιβλιοθήκες, η BLASX, η XKBlas και το CoCoPeLia. Οι τρεις αυτές βιβλιοθήκες εστιάζουν στην βελτιστοποίηση 3ου επιπέδου BLAS προβλημάτων. Τα προβλήματα που ανήκουν σε αυτή την κατηγορία περιλαμβάνουν πολλαπλασιασμό δισδιάστατων πινάκων.



Σχήμα 2.8: 3-way concurrency overlap σε δυο GPU με 4 στοιχεία μνήμη.

2.6.1 BLASX

Η BLASX είναι μια βελτιστοποιημένη multi-GPU level-3 BLAS. Η βιβλιοθήκη αυτή χρησιμοποιεί αλγορίθμους πλακιδίων, θεωρώντας το πλακίδιο των πινάκων ως τη θεμελιώδη μονάδα δεδομένων και τις λειτουργίες που γίνονται πάνω σε ένα πλακίδιο ως θεμελιώδεις διεργασίες. Με τη βιβλιοθήκη αυτή, οι συγγραφείς στοχεύουν να μειώσουν τις μεταφορές δεδομένων μεταξύ CPU-GPU με τη χρήση ενός ασύγχρονου και δυναμικού χρονοδρομολογητή, που εκμεταλεύεται την προσωρινή ύπαρξη δεδομένων στην κρυφή μνήμη της GPU (cache and locality aware). Επίσης, υλοποιεί μια 2-επιπέδων ιεραρχία κρυφής μνήμης για πλακίδια, εκμεταλευόμενη τις μεταφορές δεδομένων μεταξύ GPU. Ως πρώτο επίπεδο (L1) κρυφής μνήμης θεωρείται η μνήμη που είναι ενσωματωμένη σε κάθε GPU, ενώ το δεύτερο επίπεδο (L2) κρυφής μνήμης είναι ο συνδυασμός όλων των μνημών των GPUs. Η L1 κρυφή μνήμη στοχεύει στην ελαχιστοποίηση όλων των μεταφορών δεδομένων, ενώ η L2 κρυφή μνήμη στη μείωση των CPU-GPU μεταφορών, μετατρέποντάς τες σε GPU-GPU. Για την διαχείρηση της ιεραρχίας των κρυφών μνημών, χρησιμοποιείται ένας νέος LRU αλγόριθμος για να βοηθήσει στην ασύγχρονη εκτέλεση των διεργασιών, καθώς και ένα νέο πρωτόκολλο συνοχής (coherence) της κρυφής μνήμης για τη διασφάλιση της συνέπειας (consistency) των δεδομένων στις GPUs. Η BLASX για να μειώσει την αναμονή στις GPUs δρομολογεί 4 τέσσερις διεργασίες σε κάθε GPU κάθε φορά ώστε οι μεταφορές δεδομένων να επικαλύπτονται με την εκτέλεση. Για να μειώσει περαιτέρω τις μεταφορές δεδομένων, θέτει προτεραιότητα στις διεργασίες με ευστοχία κρυφής μνήμης L1 (cache hit), μετά στις διεργασίες με ευστοχία κρυφής μνήμης L2 και τέλος σε αυτές με αστοχία κρυφής μνήμης.[13]

2.6.2 XKBlas

Η XKBlas είναι άλλη μια βιβλιοθήκη που στοχεύει στην βελτίωση της εκτέλεσης των BLAS-3 kernels σε multi-GPU συστήματα. Χρησιμοποιεί το πρωτόκολλο της BLASX που προτείνει ένα δύο-επιπέδων μηχανισμό κρυφής μνήμης που βελτιώνει την τοπικότητα της πρόσβασης στα δεδομένα ευνοώντας την επικοινωνία μεταξύ GPU και το XKaapi που η πολιτική αφαίρεσης του δίνει προτεραιότητα στην αφαίρεση των μπλοκ που είναι για διάβασμα. Στο XKaapi έχουν γίνει αλλαγές ώστε να επιλέγει την προέλευση των δεδομένων κατά τις μεταφορές, ευνοώντας την κοντινότερη GPU-to-GPU μεταφορά.[4][5]

Η XKBlas ενώνει ένα σύνολο από BLAS-3 αλγορίθμους γραμμικής άλγεβρας για πλακίδια με το XKaapi σύστημα χρόνου εκτέλεσης για την χρονοδρομολόγηση διεργασιών με εξαρτήσεις δεδομένων σε πολλές GPUs. Η έκδοση για πλακίδια του πυκνού σε γραμμική άλγεβρα κώδικα που χρησιμοποιείται, δημιουργεί ένα γράφο ροής δεδομένων με βαθμό παραλληλισμού που εξαρτάται από τη διανομή των πινάκων. Στη συνέχεια, ο αλγόριθμος χρονοδρομολόγησης χρησιμοποιεί το γράφο αυτό για να αντιστοιχίσει τις διεργασίες στους πόρους του συστήματος. Μια διεργασία αντιστοιχίζεται στον πόρο που του ανήκει ο πίνακας εισόδου της. Αν δεν υπάρχει, επιλέγεται τυχαία. Στη συνέχεια το XKaapi διανέμει τις διεργασίες και τα δεδομένα στους αντίστοιχους πόρους του συστήματος. Κατά τη διάρκεια της εκτέλεσης ελέγχει την κατανεμημένη εκτέλεση των διεργασιών, προγραμματίζει μεταφορές δεδομένων και προσπαθεί να επικαλύψει τη μεταφορά δεδομένων με την εκτέλεση των kernels.[4][5]

2.6.3 CoCoPeLia

Το CoCoPeLia είναι ένα framework που στοχεύει στη βελτιστοποίηση του 3-way- concurrency στις GPUs που υποστηρίζουν σχεδόν βέλτιστη απόδοση για BLAS. Αυτό επιτυχγάνεται μέσω τριών συνεισφορών:[2]

1. Εισάγονται δυο αναλυτικά μοντέλα για 3-way-concurrency στις GPUs κατά την εκτέλεση BLAS ρουτινών, ένα για περιπτώσεις με επαναχρησιμοποίηση δεδομένων και ένα για περιπτώσεις χωρίς.

2. Αναπτύσεται μια αυτόματη εμπειρική μεθοδολογία για την προσαρμογή των μοντέλων αυτών σε ένα σύστημα.

3. Συνδυάζονται τα δύο προηγούμενα με έναν χρονοδρομολογητή πλακιδίων για τη δημιουργία του CoCoPeLia, ενός ολοκληρωμένου GPU BLAS framework που αξιοποιεί την αυτοματοποιημένη επιλογή διάστασης πλακιδίου.[2]

Το CoCoPeLia framework διαχειρίζεται όλα τα απαραίτητα βήματα συμπεριλαμβανομένης της αυτόματης προσαρμογής του μοντέλου σε ένα συγκεκριμένο μηχάνημα και της ανάπτυξης μιας proof-of-concept βιβλιοθήκης που αξιοποιεί το μοντέλο στο χρόνο εκτέλεσης. Στη βάση του CoCoPeLia βρίσκεται το σύστημα χρόνου εκτέλεσης επιλογής πλακιδίου, το οποίο χρησιμοποιεί τα μοντέλα πρόβλεψης που αναφέρθηκαν. Η μονάδα ανάπτυξης τροφοδοτεί το σύστημα χρόνου εκτέλεσης με τα κατάλληλα υπομοντέλα μεταφορών και υπολογισμών, ενώ

η βιβλιοθήκη υλοποιεί ένα βελτιστοποιημένο υποσύνολο του BLAS πρωτοτύπου επάνω στους βασικούς BLAS kernels. Κατά την διάρκεια της εκτέλεσης μιας εφαρμογής, όταν καλείται μια BLAS υπορουτίνα με ένα συγκεκριμένο σύνολο απο παραμέτρους για πρώτη φορά, συμβουλεύεται το μοντέλο του CoCoPeLia ώστε να επιλεγεί το κατάλληλο μέγεθος πλακιδίου.[2]

Κεφάλαιο 3

Software-assisted Memory Buffer

3.1 Εισαγωγή

Προηγούμενες μελέτες όπως αυτές που αναλύθηκαν στα κεφάλαια 2.6.1 και 2.6.2, εστιάζουν κυρίως στην κατάλληλη δρομολήγηση διεργασιών ώστε να μειώσουν το χρόνο αναμονής της GPU και υλοποιούν κάποιο σύστημα software caching για πλακίδια με την μνήμη της GPU να θεωρείται κρυφή μνήμη. Η μελέτη που αναλύθηκε στο 2.6.3 εστιάζει στην χρήση μοντέλων για την αυτόματη επιλογή του βέλτιστου μεγέθους πλακιδίου σύμφωνα με το σύστημα συνδυάζοντας την με την κατάλληλη δρομολόγηση διεργασιών. Επίσης, εστιάζουν μόνο στις GPUs και δε μελετούν το host, ενώ χρησιμοποιούν όλη την ελεύθερη μνήμη που έχει η GPU. Τέλος, δε μελετούν σε βάθος το μέγεθος των πλακιδίων. Η μελέτη που παρουσιάζεται παρακάτω, εστιάζει σε αυτά τα σημεία που δεν έχουν ερευνηθεί και χρησιμποιεί μια διαφορετική οπτική από τις προηγούμενες, θεωρεί δηλαδή τις CPUs και τις GPUs ως συσκευές χωρίς να τις διαχωρίζει και κάθε συσκευή έχει τη δική της μνήμη. Το σύστημα που υλοποιείται λειτουργεί ως κομμάτι του CoCoPeLia. Η κατασκευή του Software-assisted Memory Buffer ξεκίνησε από μια ήδη υπάρχουσα απλή υλοποίηση, η οποία βελτιώθηκε και προστέθηκαν νέες λειτουργίες. [2]

3.2 Προϋπάρχουσα υλοποιήση

Στα πλαίσια της μελέτης, ανακατασκευάστηκε το υπάρχον σύστημα ώστε να έχει σχεδόν την ίδια λειτουργικότητα αλλά υλοποιήθηκε με αντικειμενοστραφή λογική. Κατασκευάστηκαν δυο διαφορετικές κλάσεις, μία που υλοποιεί τις λειτουργικότητες της μνήμης κάθε συσκευής και μία που αποτελεί έναν wrapper για τα μπλοκ μνήμης. Τα ονόματα των κλάσεων είναι Buffer και BufferBlock αντίστοιχα.

Η κλάση BufferBlock περιλαμβάνει όλες τις πληροφορίες που χρειάζεται ένα μπλοκ. Ένα από τα σημαντικότερα πεδία των μπλοκ είναι το `state State`. Το πεδίο αυτό παίρνει τις τιμές που φαίνονται στον πίνακα 3.1. Η INVALID κατάσταση χρησιμποιείται όταν το μπλοκ αυτό δεν έχει ανατεθεί σε κάποια διεργασία και το περιεχόμενό της μνήμης του δεν είναι έγκυρο. Για τις υπόλοιπες καταστάσεις, κάθε γραμμή του πίνακα έχει τις ιδιότητες των προηγούμενων.

Δηλαδή, AVAILABLE είναι ένα μπλοκ στου οποίου τη μνήμη δεν γίνεται κάποια ανάγνωση και τα δεδομένα της μνήμης δεν έχουν υποστεί επεξεργασία, SHAREABLE όταν έχει προγραμματιστεί να γίνει ή γίνεται ανάγνωση της μνήμης και EXCLUSIVE όταν έχει γίνει, γίνεται ή θα γίνει επεξεργασία της μνήμης και δεν μπορούν να αντικατασταθούν όταν αναζητείται μνήμη από το σύστημα.

State	**Επεξήγηση**
INVALID	Δεν είναι δεσμευμένο από κάποια διεργασία.
AVAILABLE	Έχει δεσμευθεί αλλά δεν γίνεται κάποια διαδικασία πάνω του.
SHAREABLE	Έχει δεσμευτεί και γίνεται ανάγνωση του περιεχομένου του.
EXCLUSIVE	Έχει δεσμευτεί και γίνεται επεξεργασία του περιεχομένου του.

Πίνακας 3.1: Οι διαφορετικές καταστάσεις (state) ενός BufferBlock.

Δύο εξίσου σημαντικά πεδία με τα οποία συνδέεται άμεσα η κατάσταση του μπλοκ είναι οι `PendingReaders` και `PendingWriters`. Αυτοί οι δυο μετρητές κρατούν το άθροισμα των διεργασιών που διαβάζουν και γράφουν στο μπλοκ αντίστοιχα. Ο προγραμματιστής που χρησιμοποιεί το σύστημα αυτό είναι υπεύθυνος να ενημερώνει τους μετρητές χρησιμοποιώντας τις κατάλληλες μεθόδους (`add_reader, add_writer, remove_reader, remove_writer`). Οι μέθοδοι αυτοί ενημερώνουν και την κατάσταση του μπλοκ ανάλογα με τους PendingReaders. Η κατάσταση δεν αλλάζει σε περίπτωση που το μπλοκ είναι EXCLUSIVE. Συνεπώς, όταν ένα μπλοκ έχει $PendingReaders > 0$ πρέπει η κατάστασή του να είναι SHAREABLE ή EXCLUSIVE και αν έχει $PendingWriters > 0$ πρέπει η κατάστασή του να είναι EXCLUSIVE.

Κάθε μπλοκ ανήκει σε ένα αντικείμενο Buffer. Ο προγραμματιστής καλείται να φτιάξει ένα αντικείμενο Buffer για κάθε CPU και GPU που θέλει να χρησιμοποιήσει, δίνοντας το πλήθος των μπλοκ καθώς και το μέγεθός τους. Στη συνέχεια δεσμεύεται η αναγκαία μνήμη και δημιουργούνται τα μπλοκ. Από κει και πέρα, όταν μια διεργασία χρειάζεται μνήμη, καλεί τη μέθοδο `assign_Cblock` της κατάλληλης Buffer, η οποία μόλις βρει ένα ελεύθερο μπλοκ μνήμης, το επαναφέρει στην αρχική του κατάσταση και το επιστρέφει στην διεργασία που αναζητά μνήμη. Η συνάρτηση επιστρέφει το πρώτο INVALID ή AVAILABLE μπλοκ που θα συναντήσει. Ο αλγόριθμος 1 δείχνει τη λειτουργία της `assign_Cblock` με ψευδοκώδικα.

3.3 Προσθήκη νέων λειτουργιών

Το σύστημα που μόλις περιγράφηκε, είναι απλό σαν λειτουργία και περιορίζει τον προγραμματιστή καθώς απαιτεί αρκετή ελάχιστη μνήμη για να λειτουργήσει και η πολιτική αντικατάστασης μπλοκ είναι πολύ απλή. Για αυτό το λόγο έγιναν οι αλλαγές που παρουσιάζονται παρακάτω. Στο παράρτημα Α′ βρίσκεται ο ολοκληρωμένος κώδικας.

Algorithm 1 `assign_Cblock()` function

```
1: if buffer full then
2:     x = first block in buffer
3:     while x.State = INVALID OR x.State = AVAILABLE do
4:         if x.PendingReaders > 0 then
5:             break
6:         end if
7:         x = next block
8:     end while
9:     reset x
10: else
11:     x = next unused block
12:     reset x
13: end if
14: return x
```

3.3.1 Πολιτικές αντικατάστασης των μπλοκ της μνήμης

Στο νέο αντικειμενοστραφές σύστημα για διαχείρηση της μνήμης, πραγματοποιήθηκε μελέτη για την επιρροή της πολιτικής αφαίρεσης μπλοκ (caching out) στην ταχύτητα εκτέλεσης και κατασκευή τριών διαφορετικών πολιτικών εκτός της Naive, οι οποίες αναλύονται παρακάτω. Η επιλογή της πολιτκής γίνεται με σημαία (flag) κατά την εκτέλεση του cmake. Η σημαία ονομάζεται BUFFER_VAL και παίρνει τις εξής τιμές:

- 'N': Naive
- 'F': First In First Out (FIFO)
- 'M': Most Recently Used (MRU)
- 'L': Least Recently Used (LRU)

Η τιμή αυτής της σημαίας μεταφέρεται έως ότου ορίζεται ένα από τα από τα εξής preprocessor macros: `NAIVE, FIFO, MRU, LRU`. Η εναλλαγή αυτή γίνεται για να είναι πιο ξεκάθαρο στον προγραμματιστή ποιο κομμάτι κώδικα μεταγλωτίζεται ανάλογα με την πολιτική που έχει επιλέξει.

Για την υλοποίηση των πολιτκών FIFO, LRU και MRU, χρειάστηκε η κατασκευή μιας ουράς. Αυτή υλοποιείται ως μια διπλά διασυνδεδεμένη λίστα και περιλαμβάνει μια δομή για τους κόμβους και μια κλάση που υλοποιεί την ουρά. Η δομή των κόμβων ονομάζεται `Node_LL` και περιλαμβάνει το μοναδικό αναγνωριστικό ενός μπλοκ καθώς και μια δυαδική σημαία για την εγκυρότητα του αντίστοιχου μπλοκ. Κάθε κόμβος αντιστοιχεί σε ένα μόνο μπλοκ και αντίστροφα. Με τη χρήση αυτής της δομής για τους κόμβους, υλοποιείται η κλάση που θα χρησιμοποιηθεί ως ουρά. Η κλάση αυτή ονομάζεται `LinkedList`.

Με τη χρήση της LinkedList δομής υλοποιούνται δύο διαφορετικές ουρές μέσα στην Buffer ανεξαρτήτως πολιτικής αντικατάστασης, η InvalidQueue και η Queue. Η πρώτη περιλαμβάνει τους κόμβους των μπλοκ που δεν έχουν δοθεί και δε χρησιμοποιούνται. Εδώ ξεκινάνε όλοι οι

κόμβοι αρχικά κατά την αρχικοποίηση της μνήμης και εφαρμόζεται πάντα FIFO πολιτική για την επιλογή μπλοκ για χρήση. Στη δεύτερη ανήκουν όλα τα μπλοκ που βρίσκονται σε χρήση και σε αυτήν εφαρμόζονται οι διαφορετικές πολιτικές αντικατάστασης.

Πρώτα υλοποιήθηκε η FIFO πολιτική. Σύμφωνα με αυτή την πολιτική, όταν γεμίσει η μνήμη, το μπλοκ που θα αντικατασταθεί θα είναι το αρχαιότερο. Για να υλοποιηθεί αυτή, τα μπλοκ τοποθετούνται πάντα στο τέλος της ουράς όταν επιλέγονται για χρήση στην Queue ουρά ή όταν γίνονται invalidated στην περίπτωση της InvalidQueue ουράς. Το μπλοκ που επιλέγεται για αντικατάσταση είναι όποιο είναι πρώτο και δεν έχει PendingReaders.

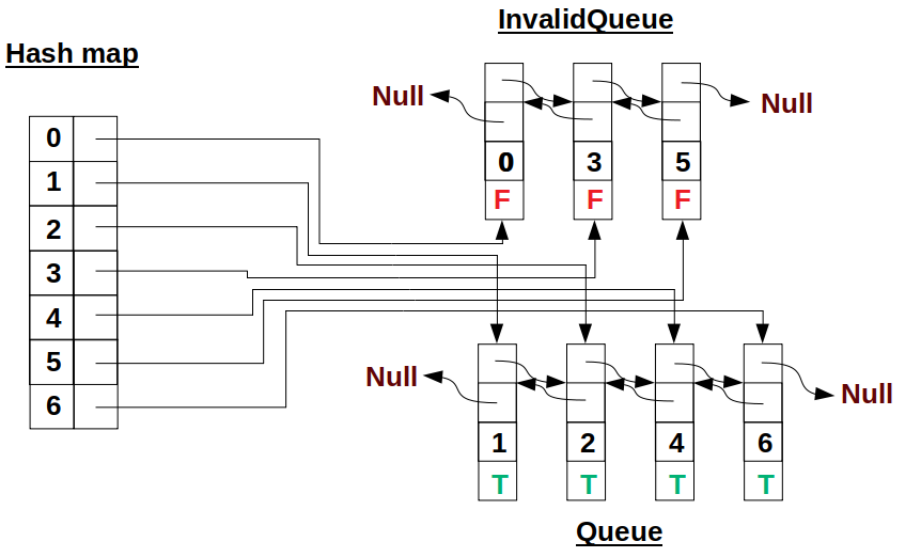
Οι επόμενες δυο πολιτικές αποτελούν παραλλαγές των LRU και MRU. Το σύστημα δεν είναι εφικτό να γνωρίζει τη στιγμή που γίνεται πρόσβαση σε κάποιο μπλοκ της μνήμης για ανάγνωση ή γραφή, καθώς η CUDA δεν παρέχει κάποια τέτοια λειτουργία. Για να λυθεί αυτό, όταν μια διεργασία σκοπεύει να διαβάσει ή να γράψει ένα μπλοκ πρέπει να ενημερώσει τον μετρητή (PendingReader ή PendingWriter) του αντίστοιχου μπλοκ, πριν την πρόσβαση στη μνήμη και αφού τελειώσει. Έτσι εξασφαλίζεται ότι το μπλοκ δε θα αντικατασταθεί ενώ χρησιμποιείται.

Έτσι, οι εναλλακτικοί αλγόριθμοι για LRU και MRU που χρησιμποιούνται ενημερώνονται κάθε φορά που παύει να γίνεται κάποια διαδικασία πάνω στη μνήμη κάποιου μπλοκ, δηλαδή όταν τρέξουν οι συναρτήσεις remove_reader και remove_writer. Επίσης, οι δυο αυτές πολιτικές ενημερώνονται όταν το μπλοκ ανατίθεται σε κάποια διεργασία. Τελικά καταλλήγουμε να έχουμε μια παραλλαγή των δυο πολιτικών, οι οποίες διαφέρουν μόνο στο πως ενημερώνονται. Και οι δυο πολιτικές χρησιμοποιούν την Queue για να γνωρίζουν τη σειρά αντικατάστασης. Τα μπλοκ που θα αντικατασταθούν είναι πάντα αυτά που οι κόμβοι τους βρίσκονται στην αρχή της ουράς συνεπώς οι δυο πολιτικές διαφέρουν στη θέση που τοποθετούν τον κόμβο του μπλοκ που μόλις χρησιμποιήθηκε. Η LRU τοποθετεί τον κόμβο στο τέλος της ουράς, ενώ η MRU στην αρχή της ουράς. Στον πίνακα 3.2 παρουσιάζεται συγκεντρωτικά πότε ενημερώνεται κάθε πολιτική και σε ποιο σημείο της Queue μεταφέρει τον κόμβο του μπλοκ που ενημερώθηκε.

	`assign_Cblock`	`remove_reader`	`remove_writer`
NAIVE	-	-	-
FIFO	Τέλος	-	-
LRU	Τελος	Τελος	Τελος
MRU	Αρχή	Αρχή	Αρχή

Πίνακας 3.2: Τοποθέτηση κόμβου στην ουρά Queue κατά την ενημέρωση ανάλογα με την πολιτική.

Σημαντικό είναι να αναφερθεί ότι για να μην υπάρχει καθυστέρηση στην ενημέρωση των ουρών και άλλες λειτουργίες, κάθε Buffer έχει έναν πίνακα κατακερματισμού που με το ειδικό αναγνωριστικό κάθε μπλοκ επιστρέφεται ο αντίστοιχος κόμβος. Έτσι όλες οι λειτουργίες για τις πολιτικές, εκτός από την αντικατάσταση γίνονται σε $O(1)$. Στο σχήμα 3.1 φαίνεται πως με το αναγνωριστικό κάθε μπλοκ επιστρέφεται ο αντίστοιχος κόμβος μέσω του πίνακα.



Σχήμα 3.1: Πίνακας κατακερματισμού.

3.3.2 Αντικατάσταση exclusive μπλοκ

Από την περιγραφή των λειτουργιών του αρχικού συστήματος, μπορεί κανείς να εκμαιεύσει ότι τα EXCLUSIVE μπλοκ δε μπορούν να αφαιρεθούν από τη μνήμη αν δε το κάνει η διεργασία που της ανήκουν. Σε προγράμματα με πολλά EXCLUSIVE μπλοκ και λίγη σχετικά μνήμη, η λειτουργία αυτή μπορεί να δημιουργήσει μεγάλες καθυστερήσεις έως και αδυναμία εκτέλεσης. Για να λυθεί αυτό το πρόβλημα, δημιουργήθηκε μια νέα κατάσταση μπλοκ και έγιναν αλλαγές στην assign_Cblock ώστε να αντικαθιστά και τα EXCLUSIVE.

Όπως αναφέρθηκε στην εισαγωγή, για το σύστημα αυτό οι CPUs και οι GPUs θεωρούνται συσκευές και δεν διαχωρίζονται. Κατά την εκκίνηση του συστήματος πρέπει να δημιουργηθεί από το χρήστη ένα αντικείμενο μνήμης για κάθε CPU ή GPU του συστήματος. Στη συνέχεια, τα δεδομένα που χρειάζονται για τους υπολογισμούς πρέπει να μεταφερθούν σε κάποιες από αυτές τις μνήμες από όπου και δε θα μπορούν να αφαιρεθούν. Επίσης τα ενδιάμεσα και τελικά αποτελέσματα πρέπει να έχουν κάποιο μόνιμο μέρος αποθήκευσης σε κάποιες από τις μνήμες. Για αυτούς τους λόγους, δημιουργήθηκε μια νέα κατάσταση (State) για τα μπλοκ, η NATIVE. Τα μπλοκ με την κατάσταση αυτή μπορούν να επεξεργαστούν όπως και τα EXCLUSIVE αλλά δεν μπορούν να αφαιρεθούν. Έτσι, οι ιδιότητες των μπλοκ διαμορφώνονται όπως στον πίνακα 3.3.

Για να μπορεί να αντικατασταθεί ένα EXCLUSIVE μπλοκ πρέπει να υπάρχει ένας μηχανισμός για να εξασφαλίζεται ότι κατά την αντικατάσταση δε θα χάνονται δεδομένα. Ο μηχανισμός που υλοποιήθηκε είναι η ετερόχρονη εγγραφή (write-back), σύμφωνα με την οποία όταν γίνεται κάποια εγγραφή η νέα τιμή γράφεται μόνο στο EXCLUSIVE μπλοκ. Το τροποποιημένο μπλοκ θα εγγραφεί στο NATIVE όταν αντικατασταθεί. Για αυτό το λόγο υλοποιήθηκε μια μέθοδος στην κλάση των μπλοκ, που όταν γίνεται αντικατάσταση του μπλοκ

State	Valid	Read	Write	Can be scheduled out
INVALID	X	X	X	-
AVAILABLE	✓	X	X	✓
SHAREABLE	✓	✓	X	✓
EXCLUSIVE	✓	✓	✓	✓
NATIVE	✓	✓	✓	X

Πίνακας 3.3: Χαρακτηριστικά καταστάσεων (State) των μπλοκ.

και η κατάστασή του είναι EXCLUSIVE, τα δεδομένα μεταφέρονται στο αντίστοιχο NATIVE μπλοκ. Κάθε EXCLUSIVE μπλοκ έχει ένα προς ένα σχέση με ένα NATIVE στο οποίο θα αποθηκευτούν οι αλλαγές σε περίπτωση που το EXCLUSIVE αντικατασταθεί. Ο λόγος που γίνεται αυτό είναι για να μην μπορούν δυο διεργασίες να γράψουν τα ίδια δεδομένα, εξασφαλίζοντας τη συνέπεια και τη συνοχή της μνήμης. Η ετερόχρονη εγγραφη επιλέχθηκε έναντι άλλων διότι έχει τις λιγότερες μεταφορές δεδομένων.[12]

3.3.3 Κλειδώματα

Οι δομές που περιγράφονται παραπάνω επεξεργάζονται από πολλά νήματα. Για να εξασφαλιστεί η σωστή χρήση των δομών και να μην υπάρχουν σφάλματα, χρησιμοποιούνται κλειδώματα (Locks). Κάθε μια από τις κλάσεις, υλοποιεί ένα κλείδωμα. Έτσι κάθε ουρά, κάθε μπλοκ και κάθε μνήμη μπορεί να κλειδωθεί ώστε να γίνουν αλλαγές πάνω στο αντικείμενο. Όλες οι συναρτήσεις παίρνουν εσωτερικά όλα τα κλειδώματα που χρειάζονται. Παρόλα αυτά, αν ο χρήστης θέλει να εκτελέσει κάποιες διεργασίες παίρνοντας εξωτερικά τα κλειδώματα, μπορεί να χρησιμοποιήσει το όρισμα `lockfree` των συναρτήσεων. Όταν καλείται μια συνάρτηση με `lockfree=True` τότε στο εσωτερικό της συνάρτησης δε λαμβάνεται κανένα κλείδωμα. Για την αποφυγή αδιεξόδων (deadlock), τα κλειδώματα πρέπει να παίρνονται πάντα με την ίδια σειρά. Φυσικά πρέπει να λαμβάνονται μόνο τα αναγκαία κλειδώματα για την αποφυγή περιττών καθυστερήσεων. Η σειρά λήψης των κλειδωμάτων είναι:

1. κλείδωμα της Buffer
2. κλείδωμα της InvalidQueue
3. κλείδωμα της Queue
4. κλείδωμα του μπλοκ
5. κλείδωμα του Native μπλοκ

3.3.4 Τελική μορφή των κλάσεων

Σε αυτό το σημείο καλό είναι να γίνει μια τελική παρουσίαση των βασικών περιεχομένων των κλάσεων Buffer καιBufferBlock για να είναι πιο ξεκάθαρο. Στους πίνακες 3.4 και 3.5

δίνεται μια σύντομη περιγραφή για τις πιο σημαντικές μεταβλητές και συναρτήσεις που περιέχει η κλάση Buffer, ενώ στους πίνακες 3.6 και 3.7 μια σύντομη περιγραφή για τις πιο σημαντικές μεταβλητές και συναρτήσεις που περιέχει η κλάση BufferBlock.

Μεταβλητή	**Επεξήγηση**
`int id`	Μοναδικό αναγνωριστικό για κάθε buffer.
`int dev_id`	Αναγνωριστικό για τη συσκευή στην οποία ανήκει.
`long long Size`	Η μνήμη προς χρήση που περιλαμβάνουν συνολικά όλα τα μπλοκ του buffer.
`int Lock`	Κλείδωμα του buffer.
`int BlockNum`	Το πλήθος των μπλοκ που μπορεί να χωρέσει ο buffer.
`long long BlockSize`	Το μέγεθος της μνήμης που δεσμεύεται για κάθε μπλοκ.
`CBlock_p* Blocks`	Δείκτης στα μπλοκ του buffer.
`Node_LL_p* Hash`	Πίνακας κατακερματισμού των κόμβων που αντιστοιχούν στα μπλοκ.
`LinkedList_p InvalidQueue`	Ουρά αχρησιμοποίητων μπλοκ.
`LinkedList_p Queue`	Ουρά μπλοκ που βρίσκονται σε χρήση.

Πίνακας 3.4: Μεταβλητές της κλάσης Buffer.

Μέθοδοι	**Επεξήγηση**
`void allocate(bool lockfree)`	Δεσμεύει μνήμη για την τοποθέτηση των μπλοκ.
`void reset(bool lockfree, bool forceReset)`	Επαναφέρει τον buffer στην αρχική του κατάσταση.
`CBlock_p assign_Cblock(state start_state, bool lockfree)`	Δεσμεύει ένα ελεύθερο μπλοκ για χρήση ή αν δεν υπάρχει, αποδεσμεύει κάποιο που δε χρησιμοποιείται σύμφωνα με την επιλεγμένη πολιτική.
`void lock()`	Κλειδώνει τον Buffer.
`void unlock()`	Ξεκλειδώνει τον Buffer.

Πίνακας 3.5: Μέθοδοι της κλάσης Buffer.

3.3.5 Λειτουργία τελικού συστήματος

Ανακεφαλαιώνοντας, έγιναν δύο σημαντικές αλλαγές. Προστέθηκαν νέες πολιτικές αντικατάστασης και δόθηκε η δυνατότητα να αντικαθιστούνται και τα EXCLUSIVE μπλοκ. Οι αλλαγές αυτές δίνουν τη δυνατότητα για περισσότερη μείωση στη μνήμη που χρησιμοποιείται για τους υπολογισμούς σε κάθε συσκευή. Με λιγότερη μνήμη όμως αυξάνονται οι κλήσεις της

Μεταβλητή	**Επεξήγηση**
`int id`	Μοναδικό αναγνωριστικό για κάθε μπλοκ.
`Buffer_p Parent`	Δείκτης στον buffer που ανήκει το μπλοκ.
`writeback_info_p WritebackData_p`	Δείκτης στα δεδομένα που χρειάζονται για το write back.
`long long Size`	Η μνήμη του μπλοκ.
`std::atomic<int> PendingReaders`	Πλήθος αιτημάτων για διάβασμα.
`std::atomic<int> PendingWriters`	Πλήθος αιτημάτων για εγγραφή.
`void* Adrs`	Δείκτης στη θέση μνήμης για αποθήκευση δεδομένων.
`state State`	Η κατάσταση του μπλοκ.
`int Lock`	Κλείδωμα του μπλοκ.

Πίνακας 3.6: Μεταβλητές της κλάσης BufferBlock.

Μέθοδοι	**Επεξήγηση**
`void allocate(bool lockfree)`	Δεσμεύει μνήμη για χρήση.
`void add_reader(bool lockfree)`	Αυξάνει κατά ένα το πλήθος αιτημάτων για διάβασμα.
`void add_writer(bool lockfree)`	Αυξάνει κατά ένα το πλήθος αιτημάτων για εγγραφή.
`void remove_reader(bool lockfree)`	Μειώνει κατά ένα το πλήθος αιτημάτων για διάβασμα.
`void remove_writer(bool lockfree)`	Μειώνει κατά ένα το πλήθος αιτημάτων για εγγραφή.
`void reset(bool lockfree, bool forceReset)`	Επαναφορά του μπλοκ στην αρχική κατάσταση που δε χρησιμοποιείται.
`void write_back(bool lockfree)`	Μεταφορά των δεδομένων του μπλοκ στη μνήμη του Native μπλοκ.
`state get_state()`	Επιστρέφει την κατάσταση του μπλοκ.
`state set_state(state new_state, bool lockfree)`	Αλλάζει την κατάσταση του μπλοκ.
`int update_state(bool lockfree)`	Ενημερώνει την κατάσταση του μπλοκ.
`void lock()`	Κλειδώνει το μπλοκ.
`void unlock()`	Ξεκλειδώνει το μπλοκ.

Πίνακας 3.7: Μέθοδοι της κλάσης BufferBlock.

`assign_Cblock`. Αυτό έχει ως αποτέλεσμα να αντικαθιστούνται τα μπλοκ πριν χρησιμοποιηθούν. Για να αντιμετωπιστεί αυτό όταν καλείται η `assign_Cblock`, ο χρήστης δίνει και την κατάσταση που θα έχει το νέο μπλοκ. Αν είναι SHAREABLE τότε η `assign_Cblock` προσθέτει και έναν `PendingReader`, ενώ αν είναι EXCLUSIVE τότε η προσθέτει έναν `PendingWriter`. Έτσι εξασφαλίζεται ότι θα γίνει τουλάχιστον μια διεργασία πάνω στο μπλοκ.

Για να χρησιμοποιηθεί ο Software-assisted memory buffer πρέπει αρχικά να μεταγλωτιστεί ο κώδικας επιλέγοντας την επιθυμητή πολιτική. Στην συνέχεια, στην αρχή της εκτέλεσης πρέπει να κατασκευαστεί ένα αντικείμενο Buffer για κάθε επεξεργαστή CPU ή GPU. Κατά την αρχικοποίηση κάθε μνήμης πρέπει να δοθεί το ειδικό αναγνωριστικό του επεξεργαστή, το πλήθος των μπλοκ που θα περιέχει και το μέγεθος κάθε μπλοκ. Από δω και πέρα, κάθε φορά που κάποια διεργασία χρειάζεται κάποιο μπλοκ μνήμης, αφού ελέγξει ότι δεν είναι ήδη στη μνήμη του επεξεργαστή που τρέχει, καλεί την assign_Cblock μέθοδο της Buffer, δηλώνοντας και την κατάσταση (State) που χρειάζεται να είναι το μπλοκ. Η συνάρτηση αυτή θα επιστρέψει ένα δείκτη στο πρώτο διαθέσιμο μπλοκ μνήμης που θα βρει. Στην περίπτωση που το μπλοκ ζττήται στην EXCLUSIVE κατάσταση, πρέπει να δοθεί και ένας δείκτης στο NATIVE μπλοκ που θα εγγραφούν τα δεδομένα σε περίπτωση αντικατάστασης του. Αν μια διεργασία έχει ήδη το μπλοκ στη μνήμη, τότε πριν από κάθε ανάγνωση ή εγγραφή του μπλοκ πρέπει να αυξήσει τον κατάλληλο μετρητή `PendingReader` ή `PendingWriter` και να τον μειώσει όταν τελειώσει η χρήση του μπλοκ. Έτσι θα εξασφαλίσει ότι δεν θα αφαιρεθεί το μπλοκ όσο το επεξεργάζεται.

Ο αλγόριθμος 2 αποτελεί ψευδοκώδικα της εκτέλεσης της βελτιωμένης `assign_Cblock` όταν αυτή έχει συμβολομεταφραστεί (compile) για τις πολιτικές FIFO, LRU και MRU. Η λειτουργία για τη NAIVE πολιτική είναι παρόμοια με την αρχική εκτέλεση απλώς έχει προστεθεί η επιλογή EXCLUSIVE μπλοκ για αντικατάσταση σε περίπτωση που δεν υπάρχουν άλλα διαθέσιμα μπλοκ. Για τις νέες πολιτικές δίνεται η μορφή της συνάρτησης `assign_Cblock()` στον ψευδοκώδικα 2 καθώς και των συναρτήσεων επιλογής μπλοκ που χρησιμοποιεί στους ψευδοκώδικες 3 και 4.

Στην εικόνα 3.2 παρουσιάζεται η ιδανική εκτέλεση του πολλαπλασιασμού πινάκων σε δυο GPUs αλλά σε αντίθεση με την εικόνα 2.7, αυτή τη φορά αξιοποιείται η ικανότητα των GPUs να αποστέλουν δεδομένα μεταξύ τους παράλληλα με την εκτέλεση του kernel και τις αποστολές hd2 (host-to-device) και hd2 (device-to-host). Οι σειρές $D_1 2 D_2$ και $D_2 2 D_1$ δείχνουν τις αποστολές από την GPU_1 στην GPU_2 και από την GPU_2 στην GPU_1 αντίστοιχα. Συγκρίνοντας με το σχήμα 2.7 φαίνεται ότι η αξιοποίηση των $D_1 2 D_2$ και $D_2 2 D_1$ βοηθάει να μειωθεί σημαντικά ο χρόνος που δεν γίνεται εκτέλεση στις GPU.

Το μέγεθος όμως των δεδομένων μπορεί να είναι μεγαλύτερο απο την μνήμη των GPUs. Στο σχήμα 3.3 παρουσιάζεται η εκτέλεση που μπορεί να πετύχει ένας προγραμματιστής με τη χρήση του Software-assisted memory buffer και αξιοποιόντας και τις τέσσερις διεργασίες που μπορεί να κάνει παράλληλα μια GPU. Στο συγκεκριμένο παράδειγμα οι GPUs έχουν μόνο τέσσερα στοιχεία μνήμη και ο Software-assisted memory buffer χρησιμοποιεί την εναλλακτική LRU που περιγράφεται στο κεφάλαιο 3.3.1. Οι γραμμές Memory LRU1 και Memory LRU2 δείχνουν το περιεχόμενο της μνήμης τις χρονικές στιγμές 1-17 με τη σειρά που βρίσκονται

Algorithm 2 `assign_Cblock(state)` function

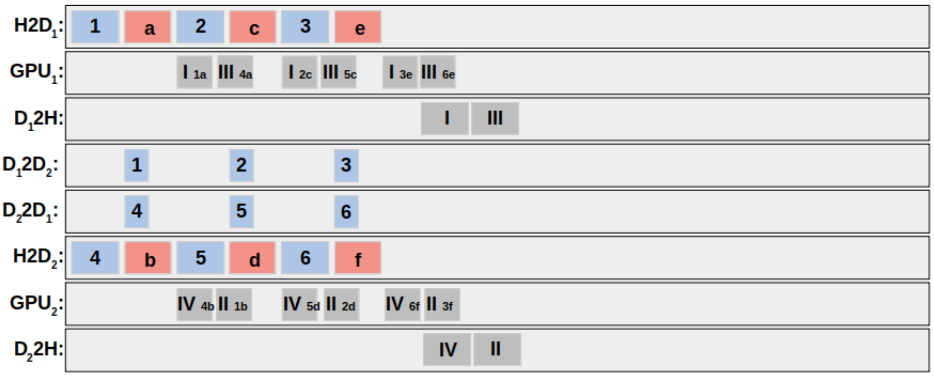
```
1: x = select_block_to_remove()
2: if x is NULL then
3:     x = select_block_to_remove()
4:     if x is NULL then
5:         x = select_block_to_remove_plus_exclusives()
6:     end if
7: end if
8: if x is not NULL then
9:     put x in Queue
10:    reset x
11:    x.set_state(state)
12:    if x.state = EXCLUSIVE then
13:        x.add_writer()
14:    end if
15:    if x.state = SHAREABLE then
16:        x.add_reader()
17:    end if
18: else
19:    x = assign_Cblock(state)
20: end if
21: return x
```

Algorithm 3 `select_block_to_remove()` function

```
1: x = NULL
2: if InvalidQueue not empty then
3:     x = remove first block in InvalidQueue
4:     x.set_state(INVALID)
5: else
6:     for b in Queue do
7:         if b.state==AVAILABLE then
8:             x = b
9:             break
10:        end if
11:    end for
12:    if x!=NULL and x.state==AVAILABLE then
13:        x.set_state(INVALID)
14:        remove x from InvalidQueue
15:    end if
16: end if
17: return x
```

Algorithm 4 `select_block_to_remove_plus_exclusives()` function

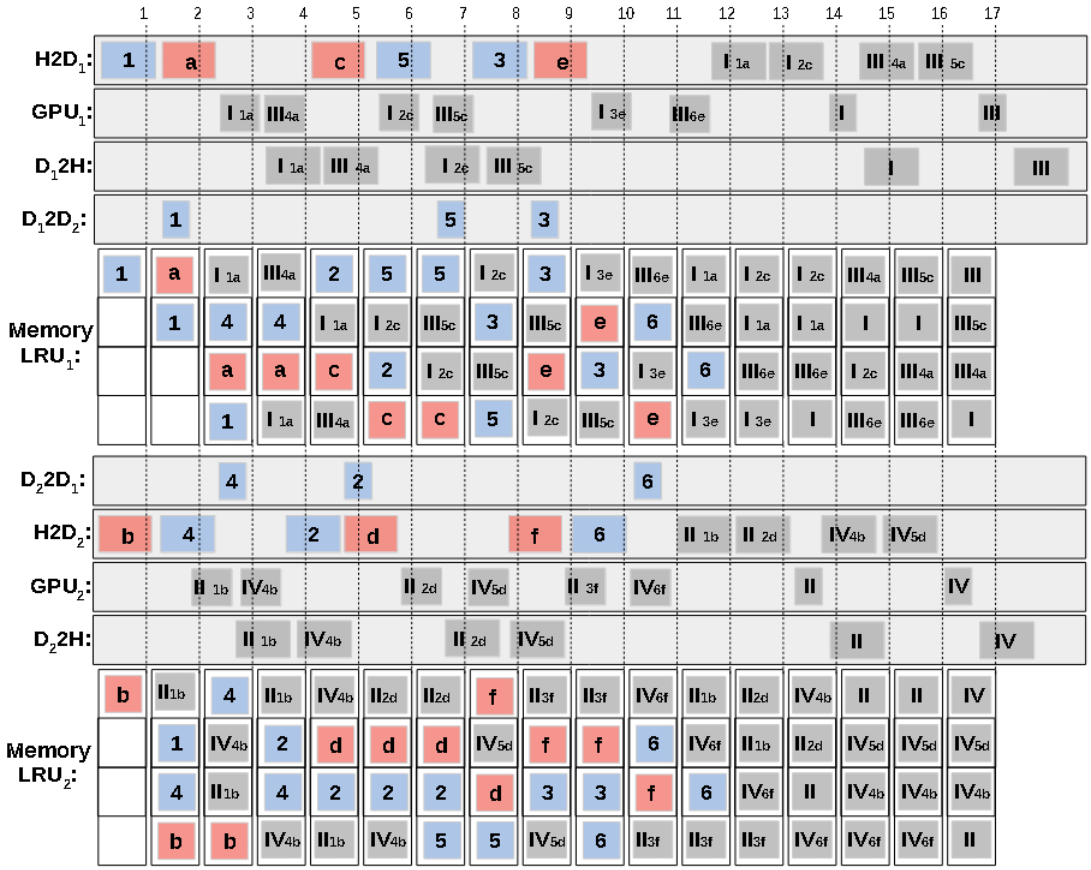
```
1:  x = NULL
2:  if InvalidQueue not empty then
3:      x = remove first block in InvalidQueue
4:      x.set_state(INVALID)
5:  else
6:      for b in Queue do
7:          if b.state==EXCLUSIVE and PendingReaders==PendingWriters==0 then
8:              x = b
9:              break
10:         end if
11:     end for
12:     if x!=NULL and b.state==EXCLUSIVE and PendingReaders==PendingWriters==0 then
13:         write back from block x to its native block
14:         x.set_state(INVALID)
15:         remove x from InvalidQueue
16:     end if
17: end if
18: return x
```



Σχήμα 3.2: Ιδανική εκτέλεση σε δυο GPUs

στην ουρά της LRU. Τα μπλοκ που βρίσκονται πιο πάνω στη μνήμη έχουν χρησιμοποιηθεί πιο πρόσφατα. Σε αυτό το σημείο πρέπει να σημειωθεί ότι ένα μπλοκ ενημερώνεται ότι μολις χρησιμοποιήθηκε όταν δεσμεύεται μνήμη για να τοποθετηθεί και κάθε φορά που τελειώνει κάποια διαδικασία που το χρησιμποιούσε.

Η εκτέλεση στο παράδειγμα του σχήματος 3.3 ξεκινάει με την αποστολή των απαραίτητων δεδομένων για την εκτέλεση του πρώτου kernel. Αξιοποιώντας την $D_1 2 D_2$ αποστολή, η GPU_2 φορτώνει παράλληλα στη μνήμη της τα μπλοκ 1 και 4 μειώνοντας το χρόνο που χρειάζεται για να ξεκινήσει η εκτέλεση του πρώτου kernel (ξεκινάει πριν την χρονική στιγμή 2, ενώ στην GPU_1 ξεκινάει μετά τη χρονική στιγμή 2) και εξαφανίζοντας την καθυστέρηση (stall) που υπήρχε στο παράδειγμα του σχήματος 2.8 από το τέλος της εκτέλεσης του πρώτου kernel μέχρι την εκκίνηση του δεύτερου. Στην GPU_1 ο πρώτος kernel ξεκινάει πιο αργά αλλά και πάλι δεν υπάρχει καθυστέρηση μεταξύ των εκτελέσεων των δυο kernels. Όσον αφορά τη λειτουργία



Σχήμα 3.3: Παράδειγμα εκτέλεσης του Software-assisted memory buffer σε δυο GPUs.

της LRU ουράς, θα εξηγηθεί πως προκύπτει η μνήμη της χρόνικής στιγμής 3 από τη μνήμη της χρονικής στιγμής 2 στην GPU_2. Αρχικά, μόλις τελειώσει η $H2D_2$ αποστολή του μπλοκ 4, μεταφέρεται το μπλοκ στην αρχή της ουράς ($[4, II_{1b}, 1, b]$). Στη συνέχεια, μόλις τελειώσει η εκτέλεση του kernel II_{1b} ενημερώνονται και μπαίνουν στην αρχή της ουράς τα τρία μπλοκ που χρησιμοποιήθηκαν, το II_{1b} που έγινε η εγγραφή και τα 1, b που διαβάστηκαν ($[II_{1b}, 1, b, 4]$). Η σειρά με την οποία μπαίνουν στην αρχή της ουράς είναι τυχαία. Ακριβώς μετά, ξεκινάει η εκτέλεση του kernel IV_{4b}. Κατά την εκκίνηση αυτή δεσμεύεται χώρος για να τοποθετηθεί το αποτέλεσμα, επομένως θα αντικατασταθεί το πρώτο μπλοκ που είναι διαθέσιμο. Στη μνήμη δεν υπάρχουν INVALID μπλοκ οπότε επιλέγεται αναζητείται ένα μπλοκ που είναι AVAILABLE ή SHAREABLE χωρίς PendingReaders. Η σειρά αυτή τη στιγμή στην LRU_2 είναι $[II_{1b}, 1, b, 4]$ και τα μπλοκ II_{1b}, b, 4 έχουν PendingReaders και PendingWriters καθώς χρειάζονται για την εκτέλεση του kernel επομένως το μπλοκ που αφαιρείται είναι το 1 ($[IV_{4b}, II_{1b}, b, 4]$). Αφού ξεκινήσει η εκτέλεση του kernel IV_{4b}, τελειώνει η $D_2 2D_1$ αποστολή του μπλοκ 4 και τοποθετείται στην αρχή της ουράς. Τελικά η σειρά των μπλοκ στην ουρά τη χρονική στιγμή 3 είναι [4, IV_{4b}, II_{1b}, b].

Κεφάλαιο 4

Πειραματική αξιολόγηση

4.1 Εισαγωγή

Σκοπός της δημιουργίας του Software-assisted Memory Buffer είναι η μείωση της απραίτητης μνήμης για προγράμματα που αξιοποιούν τις GPUs και η αποδοτικότερη χρήση της περιορισμένης αυτής μνήμης. Έτσι, στα πλαίσια της μελέτης έγιναν πειράματα στα οποία μελετήθηκε η επιρροή του μεγέθους του πλακιδίου και των διαφορετικών πολιτικών αντικατάστασης μπλοκ στην απόδοση του προγράμματος. Επίσης τα πειράματα εστιάστηκαν και στην ικανότητα μείωσης της διαθέσιμης μνήμης στις συσκευές (CPUs και GPUs) χωρίς να υπάρχει σημαντική πτώση στην απόδοση. Όλα τα πειράματα διεξήχθηκαν με την χρήση του CoCoPeLia, εκτελώντας το Dgemm benchmark που περιλαμβάνει, καθώς ο Software-assisted Memory Buffer υλοποιήθηκε ως υποσύστημα του CoCoPeLia. Όλες οι μετρήσεις έγιναν στο μηχάνημα silver1 το οποίο περιλαμβάνει σαράντα (40) Intel Xeon CPUs και τρεις (3) GPUs, μία NVIDIA GTX και δύο NVIDIA Tesla V100.

4.2 Πειραματική Αξιολόγηση

Στον πίνακα 4.1 φαίνονται τα διαφορετικά μεγέθη προβλήματος για το οποία πάρθηκαν μετρήσεις. Για κάθε διάσταση M των τετραγωνικών πινάκων έγιναν μετρήσεις με διαφορετικές διαστάσεις T των τετραγωνικών πλακιδίων όπως φαίνεται στην τρίτη στήλη του πίνακα. Κάθε συνδυασμός διάστασης πίνακα και πλακιδίου μετρήθηκε για δεκαπέντε διαφορετικά μεγέθη διαθέσιμης μνήμης. Το ελάχιστο μέγεθος μνήμης που μετρήθηκε υπολογίζεται από τον τύπο $min\_buffer = 3 \cdot T \cdot T \cdot size\_of\_elements$ και χωράει ακριβώς τρία πλακίδια, που είναι και ο ελάχιστος αριθμός πλακιδίων που πρέπει να χωράει η μνήμη για να μπορεί να πραγματοποιηθεί ο πολλαπλασιασμός των πινάκων. Το μέγιστο μέγεθος μνήμης που μετρήθηκε για κάθε συνδυασμό διάστασης πίνακα και πλακιδίου είναι όταν μπορούν να αποθηκευτούν όλα τα πλακίδια και των τριών πινάκων που χρειάζονται για τον πολλαπλασιασμό. Τα υπόλοιπα μεγέθη της διαθέσιμης μνήμης μοιράζονται με ίσες αποστάσεις στο διάστημα αυτό. Τέλος, σε κάθε συνδυασμό μεγέθους διαθέσιμης μνήμης, διάστασης πίνακα και διάστασης πλακιδίου αξιολογήθηκε κάθε μία από τις τέσσερις πολιτικές αντικατάστασης μπλοκ (Naive, Fifo, Lru,

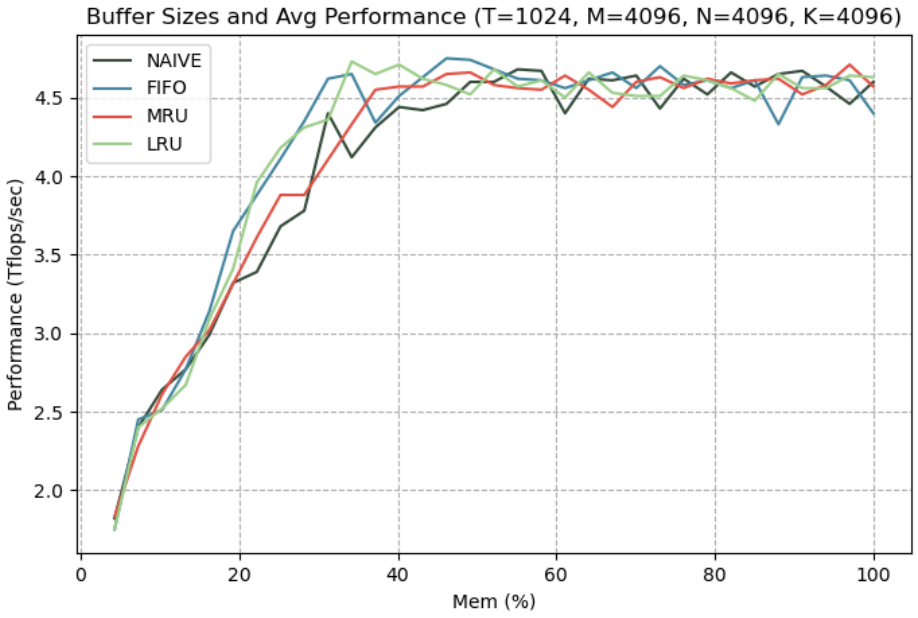
Mru) που είναι υλοποιημένες.

Table Dim. (M)	Step of M	Table Dim. (T)	Step of T
[2048,24567]	2048	$[1024, \frac{M}{2}]$	1024
32768	-	$[2048, \frac{M}{2}]$	2048

Πίνακας 4.1: Διαστάσεις πινάκων και πλακιδίων για τις μετρήσεις.

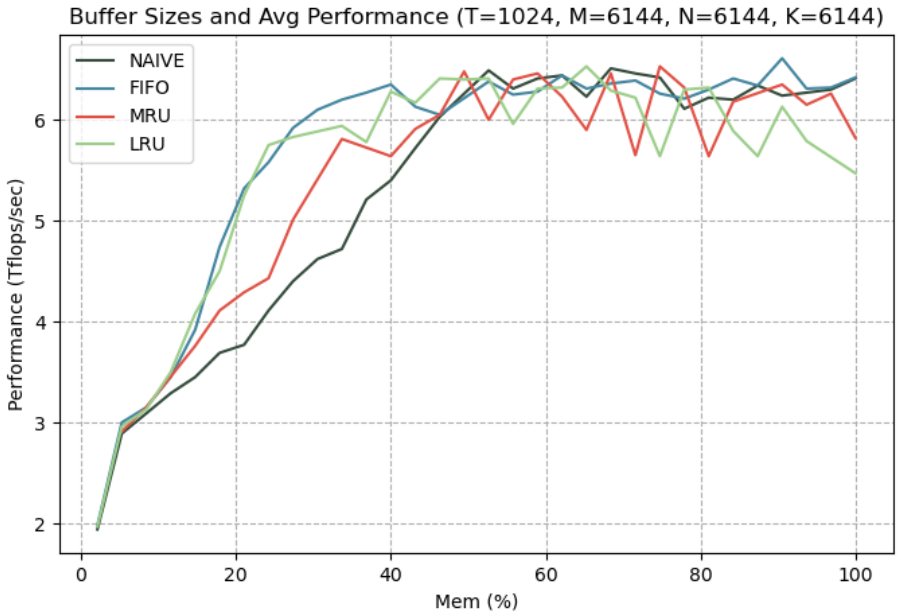
4.2.1 Σύγκριση διαφορετικών πολιτικών

Αρχικά έγιναν μετρήσεις για να διαπιστωθεί αν υπάρχει κέρδος με τη χρήση διαφορετικών πολιτικών αντικατάστασης μπλοκ. Η μελέτη ξεκίνησε από τα μικρά μεγέθη προβήματος, στα οποία διαπιστώθηκε ότι αρχίζει να υπάρχει λίγο καλύτερη απόδοση (performance) των πολιτικών LRU και FIFO όταν οι GPUs έχουν περιορισμένη μνήμη και τα πλακίδια που χρησιμοποιούνται έχουν διάσταση το πολύ το ένα τέταρτο της διάστασης των πινάκων του προβλήματος. Στο σχήμα 4.1 μπορεί να παρατηρηθεί η αρχή αυτής της διαφοράς όταν στη μνήμη μπορεί να αποθηκευτεί το 20-30% των δεδομένων του προβληματος. Αν μεγαλώσει η διαφορά διάστασης πλακιδίου με διάσταση προβλήματος, όπως φαίνεται στο σχήμα 4.2, όλες οι πολικές έχουν καλύτερη απόδοση συγκριτικά με την NAIVE όταν χρησιμποείται το 15-40% της αναγκαίας μνήμης, με καλύτερες τη FIFO και την LRU.



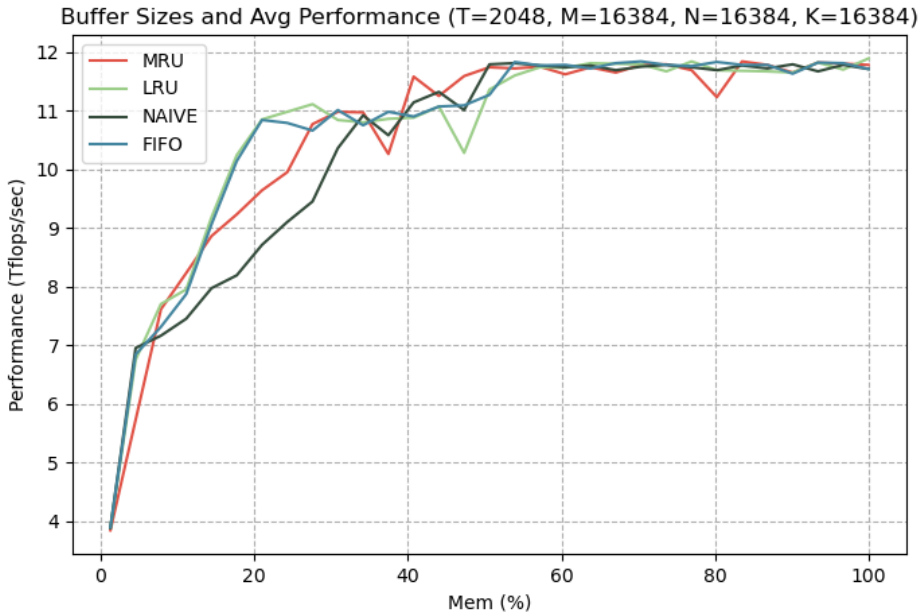
Σχήμα 4.1: Απόδοση διαφορετικών πολιτικών για T=1024 και M=N=K=4096

Σε ακόμα μεγαλύτερα προβλήματα, όπως αυτό του σχήματος 4.3 μπορεί και εκεί να παρατηρηθεί διαφορά στην απόδοση με τη χρήση διαφορετικών πολιτικών. Σε αυτό το παράδειγμα, που είναι αρκετά μεγαλύτεροι οι πίνακες, μπορεί να παρατηρηθεί έως και 20% υψηλότερη α-



Σχήμα 4.2: Απόδοση διαφορετικών πολιτικών για T=1024 και M=N=K=6144

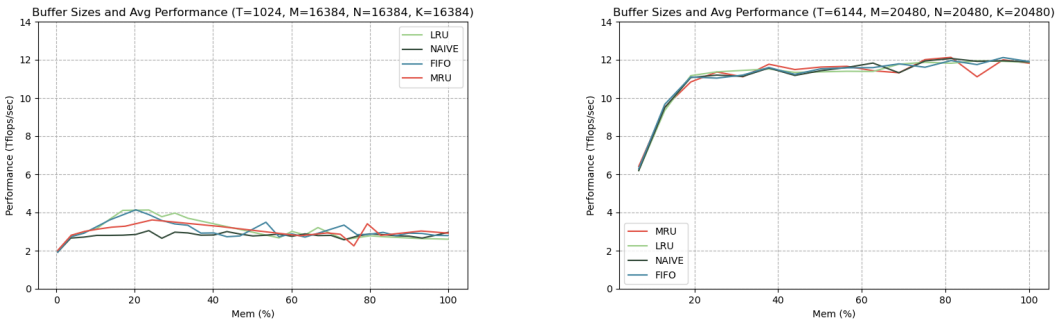
πόδοση των πολιτικών FIFO και LRU σε σχέση με τη NAIVE, ενώ η MRU δίνει έως και 10% καλύτερη απόδοση. Συνεπώς η χρήση διαφορετικών πολιτικών αντικατάστασης μπλοκ μπορεί να αυξήσει την απόδοση άρα και να μειώσει το χρόνο εκτέλεσης των προγραμμάτων.



Σχήμα 4.3: Απόδοση διαφορετικών πολιτικών για T=2048 και M=N=K=16384

Στο σχήμα 4.4 παρουσιάζονται δύο περιπτώσεις που ο Software-assisted Memory Buffer

αρχίζει να μην έχει τις επιθυμητές αποδόσεις. Στο αριστερό σχήμα φαίνεται η εκτέλεση με διάσταση πλακιδίου 1024 και διάσταση πίνακα 16384. Παρόλο που φαίνεται να υπάρχει κάποιο κέρδος για μικρά ποσοστα μνήμης (<40%) με τη χρήση διαφορετικών πολιτικών, η απόδοση της εκτέλεσης δεν ξεπερνάει ποτέ τα 4Tflops/sec σε αντίθεση με το σχήμα 4.3 που η απόδοση για το ίδιο πρόβλημα με μεγαλύτερο πλακίδιο ξεπερνάει τα 11Tflops/sec. Η πτώση της απόδοσης οφείλεται λογικά στις αυξημένες μεταφορές δεδομένων λόγω του πολύ μικρού πλακιδίου. Αντίθετα, στο δεξί διάγραμμα του σχήματος 4.4 όπου οι πίνακες έχουν διάσταση 20480 και τα πλακίδιο 6144, οι διαφορετικές πολιτικές παύουν να δίνουν καλύτερη απόδοση από την Naive καθώς τα πλακίδια είναι μεγάλα σε σχέση με το μέγεθος των πινάκων και η σωστή επιλογή αντικατάστασης μπλοκ δεν δίνει πια καλύτερη απόδοση αφού τα πλακίδια που χρησιμοποιούνται είναι λίγα.



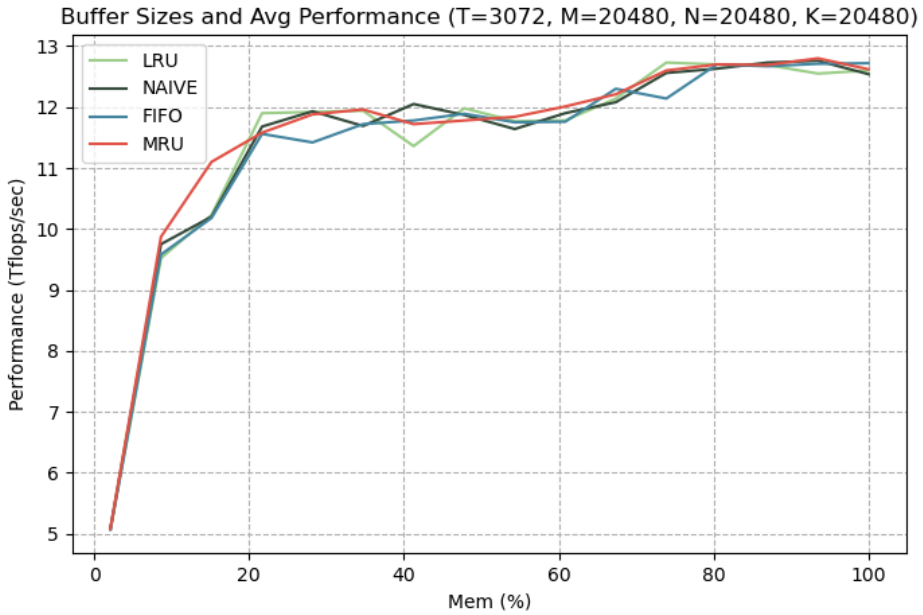
Σχήμα 4.4: Απόδοση διαφορετικών πολιτικών για ακραίες περιπτώσεις

4.2.2 Μείωση αναγκαίας μνήμης

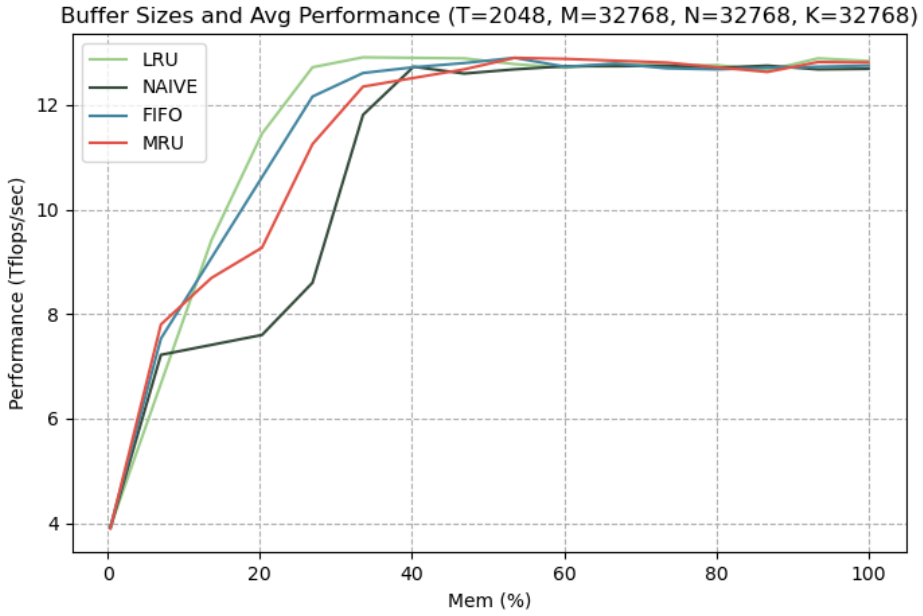
Το δεύτερο σημαντικό σημείο στο οποίο εστίασαν η μετρήσεις είναι η δυνατότητα μείωσης της μνήμης που χρησιμοποιείται από το πρόγραμμα σε κάθε συσκευή. Η μελέτη εδώ έχει νόημα κυρίως για μεγαλύτερα μεγέθη πίνακα, αφού σε αυτά είναι πιο σημαντικό να μειωθεί η μνήμη που χρησιμοποιείται.

Στο σχήμα 4.5 παρουσιάζεται η απόδοση όταν οι πίνακες έχουν διάστασεις ίσες με 20480 και το πλακίδιο έχει διάσταση 3072. Η απόδοση, όταν υπάρχει διαθέσιμη μνήμη για όλα τα δεδομένα του προβλήματος, είναι λίγο παραπάνω από το 12.5. Παρατηρώντας το σχήμα, όσο η διαθέσιμη μνήμη είναι πάνω από το 20% του χώρου που θα καταλάμβανε το πρόβλημα, η απόδοση δεν πέφτει κάτω από 11.5, δηλαδή κάτω από το 90% της βέλτιστης απόδοσης που θα μπορούσε να έχει. Αντίστοιχα στο σχήμα 4.6, που παρουσιάζεται η απόδοση όταν οι πίνακες έχουν διάστασεις ίσες με 32768 και το πλακίδιο έχει διάσταση 2048, φαίνεται ότι η απόδοση παραμένει περίπου η ίδια ακόμα και αν η μνήμη μειωθεί έως και στο 40% της μέγιστης με οποιαδήποτε πολιτική αντικατάστασης μπλοκ, ενώ με τη χρήση της LRU μπορεί να μειωθεί η διαθέσιμη μνήμη έως και στο 30% της μνήμης που θα καταλάμβαναν όλα τα δεδομένα.

Στον πίνακα 4.2 παρουσιάζονται χαρακτηριστικές εκτελέσεις για κάθε μέγεθος πίνακα. Σε αυτό το σημείο, για να είναι κατανοητές οι τιμές της απόδοσης (performance), πρέπει να σημειωθεί ότι με τις GPUs που γίνονται οι μετρήσεις, η απόδοση δε γίνεται να ξεπεράσει τα



Σχήμα 4.5: Απόδοση για T=3072 και M=N=K=20480



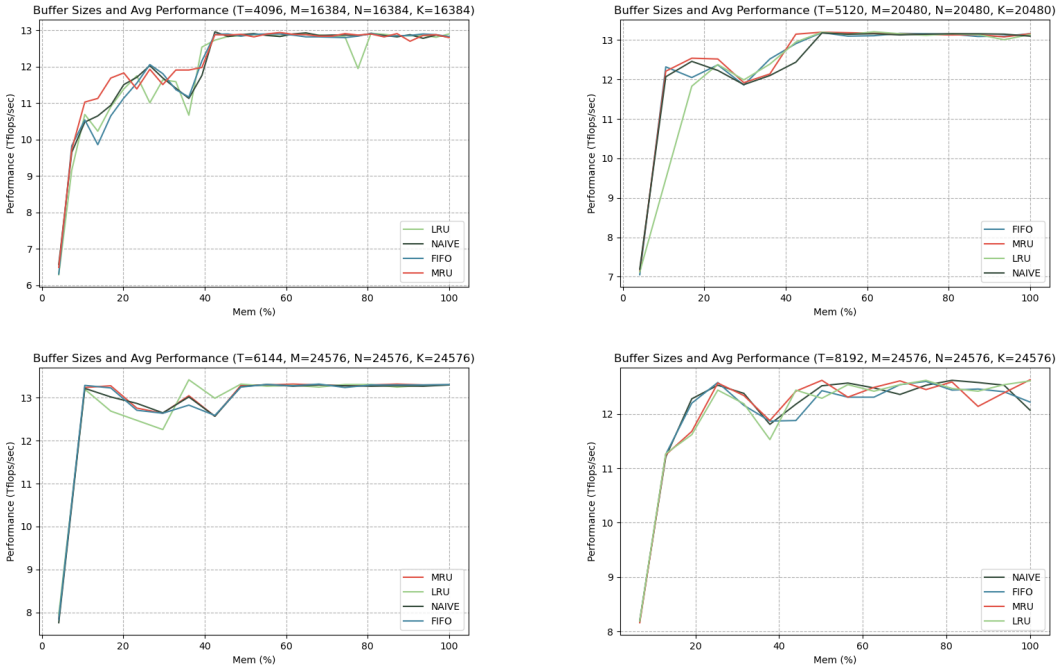
Σχήμα 4.6: Απόδοση για T=2048 και M=N=K=32768

14Tflops/sec. Έτσι, οι μετρήσεις που περιλαμβάνονται χρησιμοποιούν όσο το δυνατό λιγότερη μνήμη με ικανοποιητική απόδοση. Η στήλη Memory Usage δείχνει ότι για όλα τα μεγέθη πίνακα είναι εφικτό να μειωθεί παραπάνω από το μισό η μνήμη που απαιτείται για την εκτέλεση και σε συγκεκριμένες περιπτώσεις μπορεί να φτάσει και στο ένα τέταρτο της απαιτούμενης μνήμης. Από τη στήλη Performance επιβεβαιώνεται ότι μειώνοντας τη μνήμη η εκτέλεση

εξακολουθεί να έχει ικανοποιητική απόδοση. Τα αποτελέσματα του πίνακα 4.2 επιβεβαιώνονται και από τα διαγράμματα του σχήματος 4.7. Συνεπώς, με τη χρήση του Software-assisted Memory Buffer και την κατάλληλη επιλογή διάστασης πλακιδίου και πολιτικής αντικατάστασης μπλοκ, είναι εφικτό να γίνει σημαντική μείωση της μνήμης που απαιτείται για την εκτέλεση πολλαπλασιασμού πινάκων (Gemm) χωρίς να υπάρξει κάποια σημαντική πτώση στην απόδοση του προγράμματος.

Scheduling Policy	**Table Dimension**	**Tile Dimension**	**Performance (Tflops/sec)**	**Memory Usage**
NAIVE	12288	3072	11.13	32.92%
FIFO	14336	2048	11.06	44.23%
FIFO	16384	4096	12.89	42.5%
LRU	18432	6144	12.15	25.33%
LRU	20480	5120	12.15	25.33%
MRU	22528	2048	12.1	40.42%
LRU	24576	6144	13.42	36.11%
LRU	32768	2048	12.72	26.92%

Πίνακας 4.2: Χαρακτηριστικές εκτελέσεις για κάθε μέγεθος πίνακα.



Σχήμα 4.7: Απόδοση για μεγάλα μεγέθη προβλήματος.

4.3 Πρόβλεψη μεταβλητών

Τα αποτελέσματα των μετρήσεων έδειξαν ότι με την επιλογή της κατάλληλης διάστασης για τα πλακίδια και την κατάλληλη πολιτική αφαίρεσης μπλοκ μπορεί να μειωθεί η απαιτούμενη μνήμη. Ένας προγραμματιστής όμως που χρησιμοποιεί τον Software-assisted memory buffer χρειάζεται να γνωρίζει από πριν τις κατάλληλες μεταβλητές χωρίς να πρέπει να κάνει μετρήσεις ή να συμβουλευτεί παλαιότερες μετρήσεις. Για αυτό το λόγο υλοποιήθηκε συνάρτηση με τη χρήση της οποίας μπορεί ο προγραμματιστής να βρει τις κατάλληλες μεταβλητές για να εκτελέσει το πρόγραμμά του.

Η συνάρτηση που κατασκευάστηκε ονομάζεται `samb_variables` και παίρνει τρία υποχρεωτικά ορίσματα, τις τρεις διαστάσεις M, N και K, και δυο προεραιτικά που αφορούν τη μέγιστη μνήμη που θέλει ο προγραμματιστής να χρησιμοποιηθεί. Το πρώτο προεραιτικό (`mem_percentage`) είναι η μνήμη που θα χρησιμοποιηθεί ως ποσοστό της μνήμης που θα καταλάμβαναν όλα τα δεδομένα του προβλήματος, ενώ το δεύτερο (`memory`) είναι η μέγιστη μνήμη που θα χρησιμοποιηθεί σε byte. Αν δοθεί το όρισμα memory τότε το mem_percentage αγνοείται. Η συνάρτηση φορτώνει ένα προεκπαιδευμένο μοντέλο από το αρχείο sambPrediction.txt και βρίσκει τις μεταβλητές που μεγιστοποιούν την απόδοση. Στον αλγόριθμο 5 φαίνεται ο τρόπος λειτουργίας της `samb_variables`.

Algorithm 5 `samb_variables(M, N, K, mem_percentage=100, memory=0)`

```
1: model=load("sambPrediction.txt")
2: X = Create all the different combinations of policy, tile, memory
3: y = model.predict(X)
4: variables = max_performance(y)
5: return variables
```

Το προεκπαιδευμένο μοντέλο που χρησιμοποιεί η `samb_variables` είναι ένα μοντέλο εκπαιδευμένο πάνω στα αποτελέσματα των μετρήσεων που έγιναν στα πλαίσια αυτής της μελέτης και παρουσιάστηκαν νωρίτερα. Για την κατασκευή του χρησιμοποιήθηκε η βιβλιοθήκη schikit-learn της python. Το μοντέλο που κάνει την πρόβλεψη είναι μια δομή σωλήνωσης (pipeline) που αποτελείται από δυο επίπεδα, έναν μετασχηματιστή (transformer) `PolynomialFeatures` και έναν ταξινομητή γραμμικής παλινδρόμησης (Linear Regression classifier). Το μοντέλο εκπαιδεύεται με χρήση 5-fold cross validation και αποθηκεύεται στο αρχείο sambPrediction.txt για να μπορεί να χρησιμοποιηθεί χωρίς να χρειάζονται τα δεδομένα των μετρήσεων.

Στην εικόνα 4.8 παρουσιάζεται ένα παράδειγμα χρήσης της `samb_variables`. Ανάλογα με την είσοδο, η συνάρτηση επιστρέφει τις μεταβλητές που εκτιμάει το μοντέλο ως καλύτερες για βέλτιστη απόδοση σύμφωνα με την μέγιστη μνήμη που δόθηκε. Στον πίνακα 4.3 παρουσιάζονται έξοδοι της συνάρτησης με είσοδο τα μεγέθη πινάκων που φαίνονται στη στήλη Table Dimension και ποσοστό χρήσης μνήμης που φαίνεται στην στήλη Memory Usage(In). Σε σύγκριση και με τα αποτελέσματα του πίνακα 4.2 φαίνεται τα αποτελέσματα να μην είναι σωστά αφού υπάρχει μια επαναληψιμότητα που δεν εμφανίζεται στις μετρήσεις και η MRU συνήθως δεν έδινε την καλύτερη απόδοση. Η απόκλιση αυτή της πρόβλεψης οφείλεται στο ότι τα δεδομένα δεν είναι αρκετά για να γίνει σωστή εκπαίδευση του μοντέλου.

```
from samb_lib import sambVariables

sambVariables(32768, 32768, 32768, mem_percentage=30)
```

```
['LRU', 4096, 29, 32768, 32768, 32768]
```

```
sambVariables(14336, 14336, 14336, memory=4046664486)
```

```
['MRU', 3072, 41, 14336, 14336, 14336]
```

Σχήμα 4.8: Παράδειγμα χρήσης της συνάρτησης πρόβλεψης `samb_variables`

Table Dimension	Memory Usage(In)	Memory Usage(Out)	Scheduling Policy	Tile Dimension	Performance (Tflops/sec)
12288	30%	29%	MRU	3072	10.08
13312	30%	29%	MRU	3072	10.45
14336	30%	29%	MRU	3072	10.76
15360	30%	29%	MRU	4096	11.08
16384	30%	29%	MRU	4096	11.40
12288	50%	49%	MRU	3072	10.95
13312	50%	49%	MRU	3072	11.32
14336	50%	49%	MRU	4096	11.68
15360	50%	49%	MRU	4096	12.04
16384	50%	49%	MRU	4096	12.36

Πίνακας 4.3: Χαρακτηριστικές εκτελέσεις για κάθε μέγεθος πίνακα.

Κεφάλαιο 5

Σύνοψη

Η αριθμητική γραμμική άλγεβρα είναι θεμελιώδης για την επίλυση πολλών επιστημονικών υπολογισμών και συχνά αποτελεί ένα βαρύ υπολογιστικά τμήμα της εκτέλεσης επιστημονικών εφαρμογών. Για να επιταχυνθεί η εκτέλεση αυτών των υπολογισμών δημιουργήθηκε η BLAS. Λόγω του χαρακτήρα των υπολογισμών, δηλαδή της ικανότητας τους να γίνουν σε μεγάλο βαθμό παράλληλα, είναι ιδανικοί για εκτέλεση σε GPU. Η παραλληλία που μπορούν να παρέχουν οι GPUs, με χιλιάδες διεργασίες που εκτελούνται ταυτόχρονα, δημιούργησε μια τάση για κατασκευή βιβλιοθηκών με σκοπό την επιτάχυνση της εκτέλεσης των BLAS σε πολλές GPUs. Στο κεφάλαιο 2.6 παρουσιάστηκαν τρεις multi-GPU βιβλιοθήκες, η BLASX, η XKBlas και το CoCoPeLia framework οι οποίες όμως δε μελετούν ιδιαίτερα τη διαχείρηση της μνήμης και θεωρούν ότι έχουν όλη τη μνήμη των GPUs.

Ο Software-assisted Memory Buffer δημιουργήθηκε με σκοπό να καλύψει αυτό το κενό και να βοηθήσει τους προγραμματιστές να διαχειριστούν αποδοτικά τη μνήμη των GPUs. Επίσης βοήθησε να μελετηθεί αν με καλύτερη διαχείριση της μνήμης μπορεί να μειωθεί ο χρόνος εκτέλεσης ή/και να μειωθεί η μνήμη που χρησιμοποιείται. Με τη χρήση του Software-assisted Memory Buffer παρουσιάστηκε βελτίωση στην απόδοση της εκτέλεσης της GEMM μέσω του CoCoPeLia όταν χρησιμοποιούνται οι διαφορετικές πολιτικές αντικατάστασης μπλοκ που υλοποιούνται. Επίσης, παρουσιάστηκε ικανότητα μείωσης της μνήμης που χρησιμοποιείται ιδίως σε πίνακες μεγάλων διαστάσεων, με χρήση σε πολλές περιπτώσεις λιγότερης μνήμης από το 50% του μεγέθους του προβλήματος χωρίς να μειωθεί η απόδοση. Για να αξιοποιηθούν καλύτερα αυτά τα αποτελέσματα, κατασκευάστηκε η συνάρτηση samb_variables που βασίζεται σε ένα προεκπαιδευμένο μοντέλο για να προτείνει στον προγραμματιστή τις κατάλληλες μεταβλητές ώστε να επιτύχει μέγιστη απόδοση. Όπως αναφέρθηκε και στο κεφάλαιο 4.3, η συνάρτηση δεν δίνει τα επιθυμητά αποτελέσματα καθώς το μοντέλο στο οποίο βασίζεται χρειάζεται περισσότερα δεδομένα για να εκπαιδευτεί κατάλληλα.

Κλείνοντας, παρατίθονται συνοπτικά μερικά θέματα τα οποία θεωρούνται πιθανή επέκταση του συγκεκριμένου εργαλείου. Αρχικά, όπως αναφέρθηκε στην προηγούμενη παράγραφο, η συνάρτηση που κατασκευάστηκε για την πρόβλεψη χρειάζεται περισσότερα δεδομένα για να βρίσκει με καλύτερη ακρίβεια τις κατάλληλες μεταβλητές. Μια καλή επέκταση θα είναι να γίνουν περισσότερες μετρήσεις, πιθανότατα και σε μεγαλύτερα μεγέθη πινάκων ώστε να

υπάρχουν περισσότερα δεδομένα για την εκπαίδευση. Φυσικά αυτό πρέπει να συνδυαστεί και με βελτιστοποίηση του μοντέλου που κάνει την πρόβλεψη, ίσως και με αντικατάσταση του μοντέλου γραμμικής πρόβλεψης με κάποιο άλλο. Όσον αφορά συγκεκριμένα τον Software-assisted Memory Buffer, θα έχει ενδιαφέρον να μελετηθεί κατά πόσον η χρήση του μπορεί να προσφέρει και σε άλλα βαριά υπολογιστικά προγράμματα που τρέχουν σε GPUs εκτός της Dgemm που χρησιμοποιείται εδώ.

Bibliography

[1] *CUDA Toolkit Documentation*. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html\#asynchronous-concurrent-execution`.

[2] P. Anastasiadis, N. Papadopoulou, G. Goumas, and N. Koziris. Cocopelia: Communication-computation overlap prediction for efficient linear algebra on gpus. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 36–47, 2021.

[3] J. Dongarra. Basic linear algebra subprograms technical (blast) forum standard. *Int. J. High Perform. Comput. Appl.*, pages 1–111, 2002.

[4] T. Gautier, J. V. Lima, N. Maillard, and B. Raffin. Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 1299–1308, 2013.

[5] T. Gautier and J. V. F. Lima. Xkblas: a high performance implementation of blas-3 kernels on multi-gpu server. In *2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 1–8, 2020.

[6] C. Gregg and K. Hazelwood. Where is the data? why you cannot debate cpu vs. gpu performance without the answer. pages 134–144, 04 2011.

[7] J. Gómez-Luna, J. M. González-Linares, J. I. Benavides, and N. Guil. Performance models for asynchronous data transfers on consumer graphics processing units. *Journal of Parallel and Distributed Computing*, 72(9):1117–1126, 2012.

[8] S. Hong and H. Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. *SIGARCH Comput. Archit. News*, 37(3):152–163, jun 2009.

[9] B. Liu, W. Qiu, L. Jiang, and Z. Gong. Software pipelining for graphic processing unit acceleration: Partition, scheduling and granularity. *The International Journal of High Performance Computing Applications*, 30(2):169–185, 2016.

[10] B. Preiss. *Data Structures and Algorithms with Object-Oriented Design Patterns in C++*. John Wiley & Sons, 1999.

[11] Brian Caulfield. *What's the Difference Between a CPU and a GPU?* `https://blogs.nvidia.com/blog/2009/12/16/whats-the-difference-between-a-cpu-and-a-gpu/`.

[12] John L. Hennessy, David A. Patterson. Data-Level Parallelism in Vector, SIMD and GPU Architectures. In *Computer Architecture: A Quantitative Approach, Sixth Edition*, page 310. Morgan Kaufmann, 2019.

[13] L. Wang, W. Wu, Z. Xu, J. Xiao, and Y. Yang. Blasx: A high performance level-3 blas library for heterogeneous multi-gpu computing. In *Proceedings of the 2016 International Conference on Supercomputing*, page 20. ACM, 2016.

[14] B. v. Werkhoven, J. Maassen, F. Seinstra, and H. Bal. Performance models for cpu-gpu data transfers. In *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 11–20, 2014.

[15] J. White III and J. Dongarra. Overlapping computation and communication for advection on hybrid parallel computers. In *2011 IEEE International Parallel & Distributed Processing Symposium*, pages 59–67, 2011.

Παράρτημα Α΄

Κώδικας

Σε αυτό το παράρτημα παρουσιάζεται το header αρχείο του Software-assisted Memory Buffer. Λόγω μεγέθους, ο υπόλοιπος κώδικας είναι διαθέσιμος στο repository στον ακόλουθο σύνδεσμο "https://github.com/aristomenisTheod/Software-assisted_Memory_Buffer".

```
#ifndef DATACACHNING_H
#define DATACACHNING_H

#if BUFFER_SCHEDULING_POLICY=='N'
  #define NAIVE
#elif BUFFER_SCHEDULING_POLICY=='F'
  #define FIFO
#elif BUFFER_SCHEDULING_POLICY=='M'
  #define MRU
#elif BUFFER_SCHEDULING_POLICY=='L'
  #define LRU
#endif

#include<iostream>
#include <string>
#include <mutex>
#include <atomic>
#include "unihelpers.hpp"

enum state{
  INVALID = 0,
  NATIVE = 1,
  EXCLUSIVE = 2,
  SHARABLE = 3,
  AVAILABLE = 4,
```

```
};
const char* print_state(state in_state);

typedef class Buffer* Buffer_p;
typedef class BufferBlock* CBlock_p;
typedef struct Node_LL* Node_LL_p;
typedef class LinkedList* LinkedList_p;

typedef struct writeback_info{
    CBlock_p Native_block;
    int* WB_master_p;
    int dim1, dim2;
    int ldim, ldim_wb;
    int dtype_sz;
    CQueue_p wb_queue;
}* writeback_info_p;

typedef class BufferBlock{
  private:
  public:
    int id;
    std::string Name;
    Buffer_p Parent;
    void** Owner_p;
    writeback_info_p WritebackData_p;
    long long Size;
    std::atomic<int> PendingReaders, PendingWriters;
    void* Adrs;
    state State;
    Event_p Available;
    int Lock;

    BufferBlock(int id, Buffer_p Parent, long long Size);
    ~BufferBlock();
    // Functions
    void draw_block(bool lockfree=false);
    void allocate(bool lockfree=false);
    void add_reader(bool lockfree=false);
    void add_writer(bool lockfree=false);
    void remove_reader(bool lockfree=false);
    void remove_writer(bool lockfree=false);
```

```
    void set_owner(void** owner_adrs, bool lockfree=false);
    void reset(bool lockfree=false, bool forceReset=false);
    void init_writeback_info(CBlock_p WB_block, int* RW_master_p,
        int dim1, int dim2, int ldim, int ldim_wb, int dtype_sz,
        CQueue_p wb_queue, bool lockfree=false);
    void write_back(bool lockfree=false);
    state get_state();
    state set_state(state new_state, bool lockfree=false);
    int update_state(bool lockfree=false);
    void lock();
    void unlock();
    bool is_locked();
}* CBlock_p;

/// Device-wise software buffer class declaration
typedef class Buffer{
  private:
  public:
    int id;
    int dev_id;
    std::string Name;
    long long Size;
    int Lock;
    void* cont_buf_head;

    int SerialCtr;
    int BlockNum;
    long long BlockSize;
    CBlock_p* Blocks;

    Node_LL_p* Hash;
    LinkedList_p InvalidQueue;
    LinkedList_p Queue;

    Buffer(int dev_id, long long block_num, long long block_size);
    ~Buffer();

    // Functions
    void draw_buffer(bool print_blocks=true, bool print_queue=true,
        bool lockfree=false);
    void allocate(bool lockfree=false);
```

```
    void reset(bool lockfree=false, bool forceReset=false);
    CBlock_p assign_Cblock(state start_state=AVAILABLE,
        bool lockfree=false);
    void lock();
    void unlock();
    bool is_locked();
    double timer;
}* Buffer_p;

typedef struct CBlock_wrap{
  CBlock_p CBlock;
  bool lockfree;
}* CBlock_wrap_p;

void* CBlock_RR_wrap(void* CBlock_wraped);
void* CBlock_RW_wrap(void* CBlock_wraped);
void* CBlock_INV_wrap(void* CBlock_wraped);
void* CBlock_RR_INV_wrap(void* CBlock_wraped);
void* CBlock_RW_INV_wrap(void* CBlock_wraped);

// Node for linked list.
typedef struct Node_LL{
  Node_LL_p next;
  Node_LL_p previous;
  int idx;
  bool valid;
}* Node_LL_p;

typedef class LinkedList{
private:
  Node_LL_p iter;
  Buffer_p Parent;
public:
  std::string Name;
  Node_LL_p start;
  Node_LL_p end;
  int length;
  int lock_ll;

  LinkedList(Buffer_p buffer, std::string name="LinkedList");
  ~LinkedList();
```

```
  // Functions
  void draw_queue(bool lockfree=false);
  void invalidate(Node_LL_p node, bool lockfree=false);
  void push_back(int idx, bool lockfree=false);
  Node_LL_p start_iterration();
  Node_LL_p next_in_line();
  Node_LL_p remove(Node_LL_p node, bool lockfree=false);
  void put_first(Node_LL_p node, bool lockfree=false);
  void put_last(Node_LL_p node, bool lockfree=false);
  void lock();
  void unlock();
  bool is_locked();
}* LinkedList_p;

int BufferSelectBlockToRemove_naive(Buffer_p buffer,
        bool lockfree=false);
int BufferSelectExclusiveBlockToRemove_naive(Buffer_p buffer,
        bool lockfree=false);
Node_LL* BufferSelectBlockToRemove_fifo_mru_lru(Buffer_p buffer,
        bool lockfree=false);
Node_LL* BufferSelectExclusiveBlockToRemove_fifo_mru_lru(Buffer_p
       buffer, bool lockfree=false);

extern Buffer_p Global_Buffer[LOC_NUM];
#endif
```