Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Υλοποίηση multi-GPU L3 BLAS βιβλιοθήκης με POSIX Threads και HIP.

Διπλωματική Εργασία

Σωκράτης Πούτας

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

Αθηνα,
Σεπτέμβριος 2023

Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Υλοποίηση multi-GPU L3 BLAS βιβλιοθήκης με POSIX Threads και HIP.

Διπλωματική Εργασία

Σωκράτης Πούτας

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

Εγκρίθηκε από την τριμελή επιτροπή στις 14 Σεπτεμβρίου 2023.

Νεκτάριος Κοζύρης	Γεώργιος Γκούμας	Διονύσιος Πνευματικάτος
Καθηγητής, Ε.Μ.Π.	Αν. Καθηγητής, Ε.Μ.Π.	Καθηγητής, Ε.Μ.Π.

Αθηνα,
Σεπτέμβριος 2023

Σωκράτης Πούτας
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π

Περίληψη

Σκοπός της παρούσας διπλωματικής εργασίας είναι η εξερεύνηση διαφορετικών υλοποιήσεων μιας βιβλιοθήκης δρομολόγησης υπο-προβλημάτων γραμμικής άλγεβρας σε συστήματα με πολλαπλούς επεξεργαστές γραφικών (multi-GPU BLAS), στοχεύοντας στην επιτάχυνση που προσφέρει η επικάλυψη του υπολογισμού σε GPU και της μεταφοράς δεδομένων μεταξύ CPU και GPU. Αυτό επιτυγχάνεται μέσω ουρών εργασιών και ενός συστήματος συγχρονισμού με βάση γεγονότα (events). Οι προηγούμενες εκδόσεις της βιβλιοθήκης χρησιμοποιούσαν CUDA, κάτι που την καθιστούσε λειτουργική μόνο σε συστήματα με Nvidia GPUs. Σε αυτή την εργασία υλοποιούμε δύο νέες εκδόσεις της βιβλιοθήκης: μία βασισμένη στα POSIX threads και μία που χρησιμοποιεί το HIP. Τέλος, συγκρίνουμε τις προγραμματιστικές δυνατότητες κάθε υλοποίησης και τις επιπτώσεις τους στην επίδοση της βιβλιοθήκης, συμπεραίνοντας ότι οι δικές μας υλοποιήσεις επεκτείνουν τις δυνατότητες εφαρμογής της βιβλιοθήκης και επιτυγχάνουν παρόμοιες ή καλύτερες επιδόσεις.

Λέξεις κλειδιά: Επεξεργαστές γραφικών, Παράλληλη εκτέλεση, GPGPU, Multi-GPU BLAS, CUDA, Pthreads, HIP, Ουρές εργασιών, Γεγονότα

Abstract

This thesis explores different library implementations for routing linear algebra sub-problems in multi-GPU systems, with the aim of achieving speedup through overlapping CPU-GPU communication with GPU computation. This is accomplished through task queues and an event-based synchronization system. The previous versions of the library utilized a CUDA back-end, limiting its functionality to Nvidia systems. In this thesis, we implement two new versions of the library: one based on POSIX threads and another that uses HIP. We then compare the programming capabilities of each implementation and their impact on library performance, concluding that our implementation extends the applicability of the library and achieves similar or superior performance.

Keywords: Graphics Processing Unit (GPU), Parallel execution, GPGPU, Multi-GPU BLAS, CUDA, Pthreads, HIP, Task queues, Events

Ευχαριστίες

Η συγκεκριμένη εργασία εκπονήθηκε στο Εργαστήριο Υπολογιστικών Συστημάτων (cslab) της σχολής Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Εθνικού Μετσοβίου Πολυτεχνείου. Θα ήθελα να ευχαριστήσω τον υπεύθυνό μου, τον Γεώργιο Γκούμα, για την ευκαιρία που μου έδωσε να ασχοληθώ με ένα τόσο ενδιαφέρον θέμα, αλλά και για όλη τη γνώση και έμπνευση που μου μετέφερε μέσα από τα μαθήματά του. Ακόμα, ευχαριστώ από καρδιάς τον υποψήφιο διδάκτορα Πέτρο Αναστασιάδη που με καθοδήγησε σε όλη τη διάρκεια εκπόνησης της διπλωματικής και προσέφερε άμεσα την καταλυτική του βοήθεια κάθε φορά που τη χρειαζόμουν.

Τέλος, οφείλω ένα μεγάλο ευχαριστώ στην οικογένειά μου, για τη στήριξή τους όλο αυτό το διάστημα, και στους φίλους μου που μοιραστήκαμε την πορεία μας. Χωρίς εκείνους δεν θα ήμουν αυτός που είμαι σήμερα.

Περιεχόμενα

Chapter 1

Απαραίτητες Γνώσεις

1.1 Επεξεργαστές Γραφικών - Graphics Processing Units (GPUs)

Η εξέλιξη των επεξεργαστών γραφικών

Τα πρώτα βήματα στην κατεύθυνση των επεξεργαστών γραφικών έλαβαν χώρα στη δεκαετία του 1970, όταν η υπολογιστική ισχύς των επεξεργαστών δεν μπορούσε να καλύψει τις ανάγκες για παραγωγή ψηφιακής εικόνας και βίντεο. Έτσι, αρκετοί κατασκευαστές, ανάμεσά τους η RCA η Atari και η Motorola, ξεκίνησαν να παράγουν υλικό, εξειδικευμένο στην επεξεργασία γραφικών, με τη μορφή προσαρμογέων οθόνης [Singer, 2013].

Αργότερα, στις αρχές της δεκαετίας του 1990 δημιουργήθηκε το OpenGL και το Direct3D (το οποίο θα εξελισσόταν στο DirectX), που αποτέλεσαν τις πρώτες προγραμματιστικές διεπαφές (Application Programming Interfaces, APIs) για 2D και 3D γραφικά. Οι προγραμματιστικές διεπαφές επιτρέπουν στις εφαρμογές να χρησιμοποιούν τμήματα του λογισμικού ή του υλικού. Για παράδειγμα, ένα πρόγραμμα μπορεί να χρησιμοποιήσει μια διεπαφή για να στείλει εντολές στη συσκευή που εμφανίζει σχήματα σε μια οθόνη [Hwu, 2012]. Τα τέλη αυτής της δεκαετίας σηματοδότησαν μια σημαντική καμπή, με την Nvidia να παρουσιάζει την GeForce 256: το πρώτο chip που ονομάστηκε “Μονάδα Επεξεργασίας Γραφικών” (GPU), και συμπεριλάμβανε μονάδα γεωμετρικών υπολογισμών και υπολογισμών φωτισμού. Τόσο η Nvidia όσο και η ATI (που σήμερα ανήκει στην AMD) συνέβαλαν καθοριστικά στην ανάπτυξη GPUs που μπορούσαν να προγραμματιστούν για να διεκπεραιώνουν πιο εξελιγμένες εργασίες απόδοσης εικόνας. Αυτή η καινοτομία επέτρεψε τη βελτίωση της ποιότητας και της επίδοσης των γραφικών.

Ο σχεδιασμός αυτών των επεξεραστών ειδικού σκοπού στοχεύει στην επίτευξη υψηλού βαθμού παράλληλης επεξεργασίας, καθώς η διαχείριση γραφικών είναι μια εγγενώς παράλληλη εργασία. Σύντομα, έγινε σαφές ότι περισσότεροι πυρήνες που λειτουργούν με χαμηλότερη συχνότητα είναι πιο αποδοτικοί από διατάξεις λιγότερων, αλλά πιο γρήγορων πυρήνων.

Μέχρι τότε οι GPUs αποτελούσαν υλικό με εφαρμογή στην επεξεργασία γραφικών. Ωστόσο, ήδη από το 2003 άρχισε η έρευνα για την χρήση αυτών των επεξεργαστών σε γενικά προβλήματα γραμμικής άλγεβρας, αφού οι πράξεις μεταξύ πινάκων και διανυσμάτων μπορούν να αξιοποιήσουν σε μεγάλο βαθμό την παραλληλία που παρέχει το υλικό [Krüger and Westermann, 2003]. Αυτό είναι και το σημείο εκκίνησης των επεξεργαστών γραφικών γενικού σκοπού (General Purpose GPUs, GPGPUs).

Αναγνωρίζοντας αυτή την ανάγκη, το 2007 η Nvidia κυκλοφορεί το CUDA, μια προγραμματιστική διεπαφή γενικότερου σκοπού για Nvidia GPUs. Με τον τρόπο αυτό οι χρήστες μπορούσαν, πλέον, να εκτελούν μαθηματικούς υπολογισμούς χωρίς να αναγκάζονται να σχεδιάζουν τα προγράμματά τους με βάση κάποια διεπαφή γραφικών, που χρησιμοποιεί shaders και textures [Abi-Chahla, 2008]. Ακολουθώντας, η Khronos Group εκδίδει το 2009 το OpenCL ως πρότυπο παράλληλου προγραμματισμού για κάθε επεξεργαστή γραφικών.

Έκτοτε, οι GPUs συνέχισαν να εξελίσσονται ως μονάδες πιο διευρυμένου σκοπού από την παραγωγή γραφικών. Σήμερα χρησιμοποιούνται εκτεταμένα σε τομείς όπου υπάρχει ανάγκη για πολλούς υπολογισμούς ανά δεδομένο, όπως οι επιστημονικοί αλγόριθμοι, η μηχανική μάθηση, τα Big Data, καθώς και σε αρκετά μοντέλα προβλέψεων.

Διαφορές με τις κεντρικές μονάδες επεξεργασίας

Οι GPUs, λοιπόν, μπορούν πλέον να εκτελούν υπολογισμούς γενικής φύσης, όπως και οι CPUs. Ωστόσο, υπάρχουν εργασίες που είναι κατάλληλες για εκτέλεση σε CPU, και άλλες που είναι αποδοτικότερο να χρησιμοποιηθεί GPU. Αυτό καθορίζεται κυρίως από τον βαθμό παραλληλίας και τον αριθμό υπολογισμών ανά δεδομένο της εκάστοτε εργασίας, καθώς οι επεξεργαστές γραφικών είναι καλοί στη διαχείριση τεράστιου όγκου επαναλαμβανόμενων πράξεων.

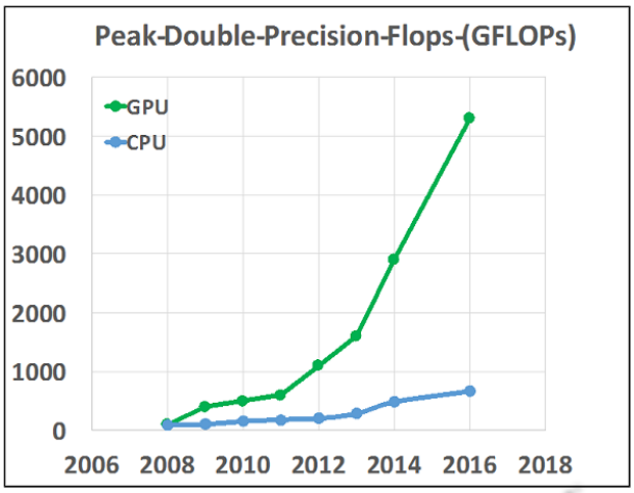
Πιο συγκεκριμένα, οι κεντρικές μονάδες επεξεργασίας σχεδιάζονται για να να χειρίζονται οποιουδήποτε είδους υπολογισμό. Αναλαμβάνουν διεργασίες του λειτουργικού συστήματος, τον χειρισμό εισόδου και εξόδου και την εκτέλεση γενικών προγραμμάτων. Διαθέτουν μια ποικιλία μηχανισμών που επιτρέπει σύνθετα σχήματα διακλαδώσεων, διοχέτευση με προβλέψεις εντολών και σεβασμό των εξαρτήσεων από προηγούμενους υπολογισμούς. Προσφέρουν υψηλή μονονηματική (single-threaded) επίδοση, πράγμα που σημαίνει ότι εκτελούν γρήγορα σειριακά προγράμματα.

Από την άλλη, οι GPUs αναδεικνύονται σε υπερβολικά παράλληλα προγράμματα. Περιέχουν εκατοντάδες ή χιλιάδες μικρούς πυρήνες στη θέση των λίγων (2-64), ισχυρών πυρήνων μιας CPU, και η εναλλαγή των νημάτων εκτέλεσης γίνεται στο υλικό. Με τον τρόπο αυτό, μπορούν να εκτελούνται ταυτόχρονα χιλιάδες νήματα, παρέχοντας τη δυνατότητα για τρισεκατομμύρια πράξεις κινητής υποδιαστολής ανά δευτερόλεπτο (GFLOPs). Συνεπώς, παρότι κάθε πυρήνας μιας GPU είναι πιο αδύναμος από έναν πυρήνα CPU, η συλλογική απόδοση σε παράλληλες εφαρμογές μεγάλης κλίμακας είναι πολύ καλύτερη.

Μια ακόμα σημαντική διαφορά είναι αυτή που σχετίζεται με τη μνήμη. Οι CPUs έχουν κρυφές μνήμες οργανωμένες σε 2 ή 3 επίπεδα (L1, L2 και L3 caches), που προσφέρουν ταχύτατη πρόσβαση σε συχνά χρησιμοποιούμενα δεδομένα και εντολές. Στη θέση τους, οι GPUs χρησιμοποιούν μνήμες με λιγότερα επίπεδα ιεραρχίας, οι οποίες εμφανίζουν μεγάλο εύρος ζώνης, ώστε να εξυπηρετούνται αποδοτικά οι πυρήνες ακόμα και σε περιπτώσεις εφαρμογών που χρειάζονται λιγότερες πράξεις κινητής υποδιαστολής ανά δεδομένο.

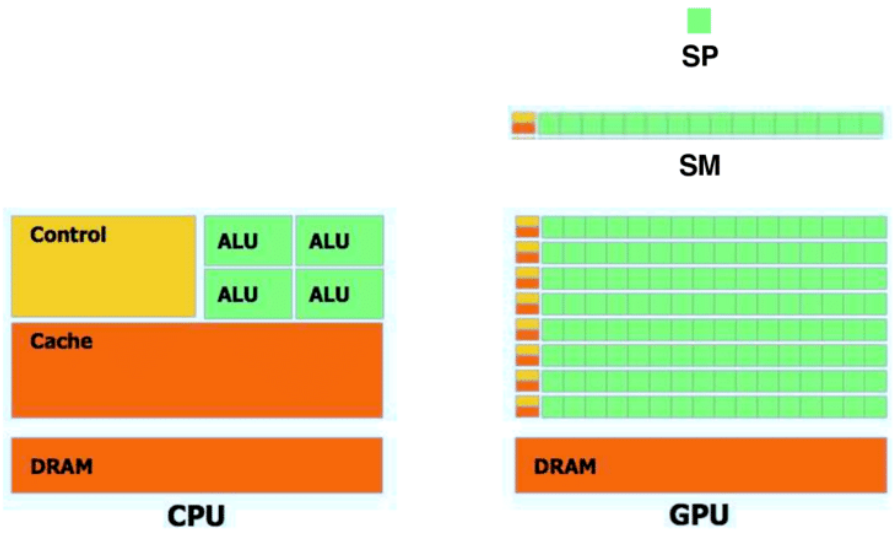
Η αρχιτεκτονική των επεξεργαστών γραφικών

Φυσικά, οι ιδιαιτερότητες που αναφέρθηκαν προκύπτουν από την αρχιτεκτονική των GPUs. Κατά συνέπεια, οι λεπτομέρειες της δομής τους, με τους περιορισμούς και τις δυνατότητες που προσφέρει, είναι απαραίτητες σε κάθε περίπτωση σχεδίασης προγραμμάτων για GPU.



Σχήμα 1.1: Επίδοση CPU και GPU. *source: HPC wire*

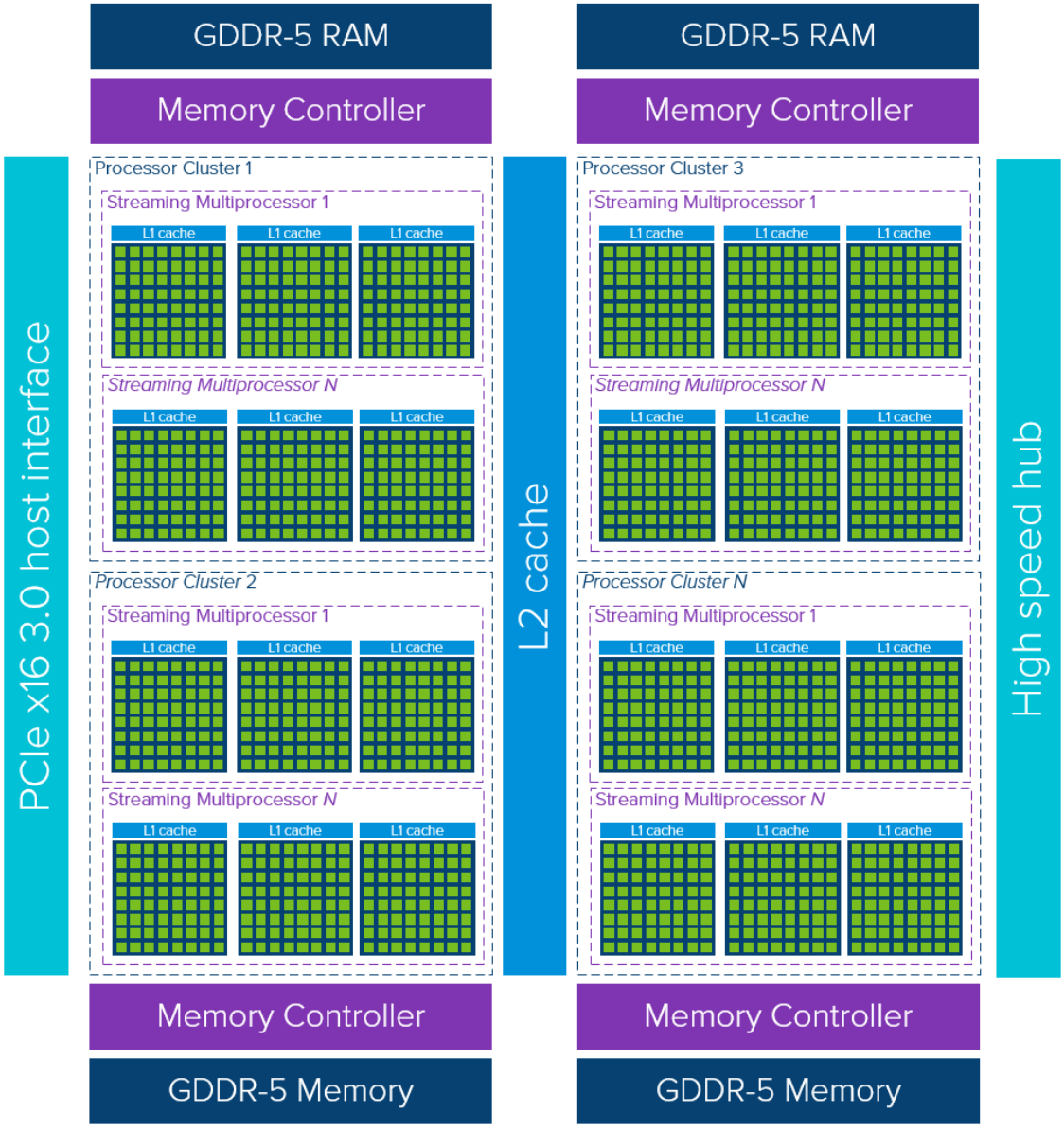
Στο υλικό οι σχεδιαστές εξοικονομούν πόρους χρησιμοποιώντας απλά κυκλώματα για τον έλεγχο ροής και το caching των δεδομένων, τους οποίους αξιοποιούν στο μέγιστο για να αυξήσουν μέγεθος του αρχείου καταχωρητών, το πλήθος των αριθμητικών και λογικών μονάδων (ALUs) (Σχήμα 1.2). Αυτό υποδεικνύει το γεγονός ότι η αρχιτεκτονική επικεντρώνεται στην παροχή καθαρής επεξεργαστικής δύναμης, με τον συμβιβασμό ότι τα προγράμματα για GPUs δεν μπορούν να περιέχουν σύνθετα σχήματα ελέγχου ροής, αφού κάτι τέτοιο θα υποαξιοποιήσει το υλικό.



Σχήμα 1.2: Το υλικό της GPU είναι προσανατολισμένο στην επεξεργασία δεδομένων. *source: Nvidia CUDA programming guide*

Όπως φαίνεται στο Σχήμα 1.3 κάθε επεξεργαστής γραφικών αποτελείται από συστοιχίες πολυεπεξεργαστών ροής (Streaming Multiprocessors, SMs), οι οποίοι με τη σειρά τους περιέχουν μια μοιραζόμενη μνήμη, επεξεργαστές ροής, ένα μεγάλο αρχείο καταχωρητών (8k-32k καταχωρητές των 32 bit), μνήμη σταθερών μόνο για ανάγνωση (read-only constant cache), κρυφή μνήμη υφών. Αυτοί οι πολυεπεξεργαστές μπορούν να εκτελούν εκατοντάδες, εώς και μερικές χιλιάδες, νήματα ταυτόχρονα σε SIMT (Single Instruction, Multiple Thread) αρχιτεκτονική. Αυτό

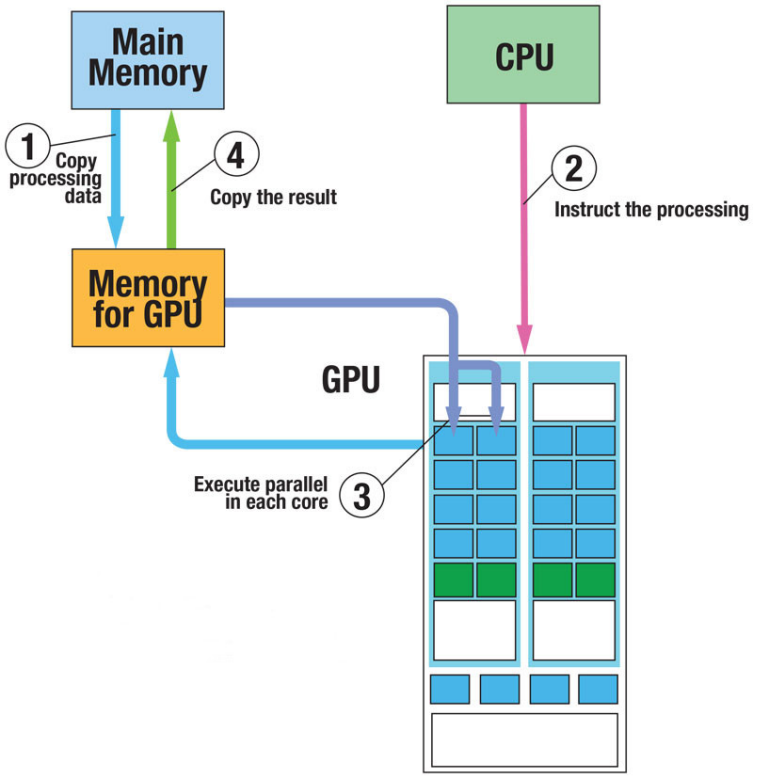
σημαίνει ότι τα νήματα (όχι όλα τα νήματα ενός SM) εκτελούν την ίδια εντολή, αλλά καθένα επεξεργάζεται διαφορετικά δεδομένα. Κάτι τέτοιο είναι ιδιαίτερα χρήσιμο σε υπολογισμούς γραμμικής άλγεβρας, όπου διαδοχικά νήματα εκτελούν την ίδια πράξη μεταξύ διαδοχικών στοιχείων διανυσμάτων ή πινάκων. Επιπλέον, εφαρμόζεται διοχέτευση (pipeline), ώστε να αξιοποιείται και η παραλληλία σε επίπεδο εντολής.



Σχήμα 1.3: Αρχιτεκτονική GPU. *source: https://core.vmware.com/resource/exploring-gpu-architecture*

Επιπλέον, οι επεξεργαστές γραφικών διαθέτουν ξεχωριστή μνήμη και είναι αναγκαίο να γίνεται μεταφορά των δεδομένων από και προς τη μνήμη της CPU. Η ροή ενός προγράμματος που χρησιμοποιεί GPU ξεκινάει με τον έλεγχο στον κεντρικό επεξεργαστή, ο οποίος αναφέρεται ως “host”, ενώ η GPU ονομάζεται “device”. Ο επεξεργαστής αναλαμβάνει την προετοιμασία των δεδομένων και την μεταφορά τους στη μνήμη της GPU. Έπειτα, πρέπει να εκκίνησει τον κώδικα που θα εκτελεστεί στην GPU, μια ενέργεια που αποκαλείται “εκκίνηση πυρήνα” (kernel launch),

και να επιστρέψει το αποτέλεσμα πίσω στην “host” μνήμη, όταν είναι έτοιμο. Αυτή η διαδικασία παρουσιάζεται στο Σχήμα 1.4



Σχήμα 1.4: Ροή προγράμματος που χρησιμοποιεί GPU. *source: cslab CUDA model presentation*

Συνεπώς, τα προγράμματα που αξιοποιούν σε μεγάλο βαθμό τις GPUs είναι εκείνα που μεταφέρουν τα δεδομένα που χρειάζονται για τον υπολογισμό και, στη συνέχεια, εκτελούν πάρα πολλές πράξεις με αυτά. Σε περιπτώσεις όπου μεταφέρονται πολλά δεδομένα και πραγματοποιούνται λίγες πράξεις ανά δεδομένο, μπορεί να υπάρξει υποαξιοποίηση του επεξεργαστή γραφικών, καθώς οι μεταφορές δεδομένων θα αποτελούν στενωπό για την GPU. Ο εν λόγω περιορισμός και οι επιλογές για την αποφυγή του θα μας απασχολήσουν αρκετά στη συνέχεια της παρούσας εργασίας.

1.1.1 Το προγραμματιστικό μοντέλο CUDA

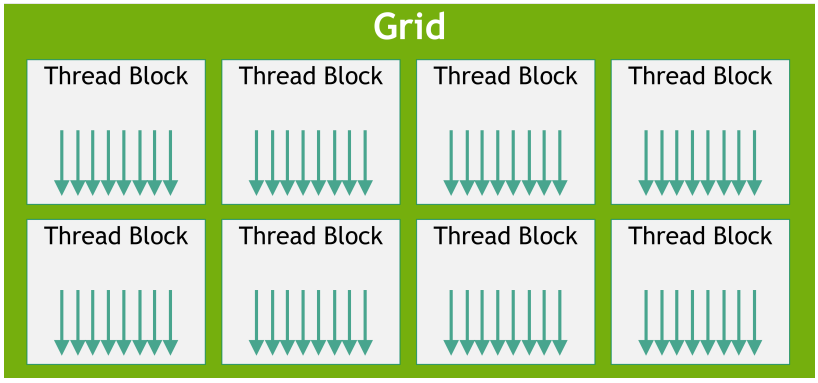
Το σύστημα που αναπτύχθηκε στο πλαίσιο αυτής της διπλωματικής εργασίας βασίζεται σε μεγάλο βαθμό στο προγραμματιστικό μοντέλο CUDA, που παρέχεται για προγραμματισμό Nvidia GPUs. Για αυτό κρίνεται σκόπιμο να παρουσιαστούν οι μηχανισμοί που παρέχει η εν λόγω διεπαφή.

Το CUDA κυκλοφορεί από την Nvidia ως επέκταση των γλωσσών προγραμματισμού C/C++ και Fortran, προσφέροντας στους προγραμματιστές τη δυνατότητα να χειρίζονται σε πιο χαμηλό επίπεδο την εκτέλεση του κώδικα, χωρίς να αποκαλύπτει σε βάθος την υποκείμενη αρχιτεκτονική. Θεμελιακό στοιχείο του είναι οι υπολογιστικοί πυρήνες (kernels): συναρτήσεις με το αναγνωριστικό `__global__` που εκτελούνται παράλληλα N φορές από N νήματα στην GPU. Κάθε νήμα φέρει ένα μοναδικό χαρακτηριστικό (*threadID*), που είναι προσβάσιμο κατά

την εκτέλεση του πυρήνα. Με βάση το *threadID* μπορεί να διαμοιράζεται η εργασία μεταξύ των νημάτων, με το καθένα να ενεργεί σε διαφορετικά δεδομένα, ανάλογα με το ID του.

Η οργάνωση των νημάτων

Στο CUDA τα νήματα οργανώνονται σε blocks, με τον αριθμό νημάτων ανά block να φτάνει τα 1024 στις σύγχρονες κάρτες γραφικών. Η κατανομή των νημάτων εντός του block αφήνεται στην επιλογή του προγραμματιστή και μπορεί να είναι σε μία, δύο ή τρεις διαστάσεις. Αυτό δεν θα ήταν απαραίτητο, όμως προσφέρει σημαντική προγραμματιστική ευκολία στο χειρισμό μονοδιάστατων (διανυσμάτων), δισδιάστατων (πινάκων), ή τρισδιάστατων (τανυστών μεγαλύτερης τάξης) δομών. Με τη σειρά τους τα thread blocks σχηματίζουν ένα πλέγμα (grid), όπως παρουσιάζεται στο Σχήμα 1.5, που μπορεί να είναι μίας, δύο ή τριών διαστάσεων, και πάλι ανάλογα με τις αναγκες της εφαρμογής. Ο διαχωρισμός των νημάτων σε blocks και η διάταξη στο πλέγμα ορίζεται από τον προγραμματιστή κατά την κλήση του πυρήνα.



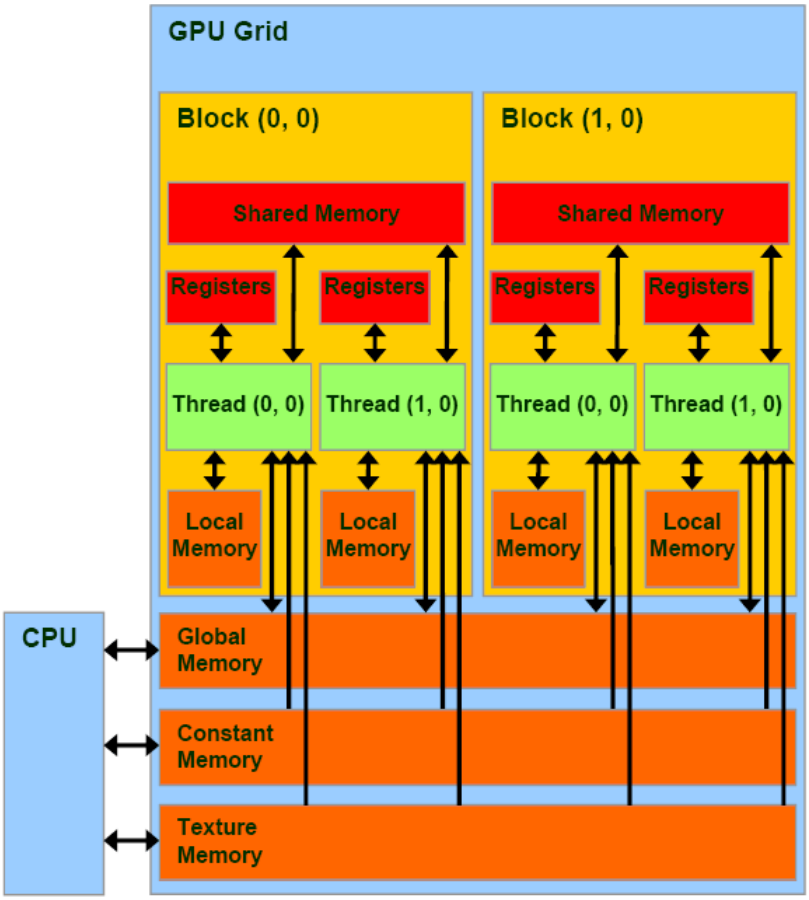
Σχήμα 1.5: Πλέγμα από thread blocks. *source: Nvidia CUDA Programming Guide*

Ένας πολυεπεξεργαστής ροής (Streaming Multiprocessor) αποτελείται από έναν μεγάλο αριθμό πυρήνων (CUDA cores). Για παράδειγμα, υπάρχουν 128 CUDA cores ανά SM στην Nvidia GeForce GTX 1060 (συνολικά 1280 πυρήνες)[Nvidia, 2016] και 64 CUDA cores ανά SM στην Nvidia Tesla V100 (συνολικά 5120 πυρήνες)[Nvidia, 2017], που χρησιμοποιήθηκαν σε αυτή την εργασία. Κάθε SM εκτελεί ένα ή περισσότερα thread blocks (εξαρτάται από το μέγεθος των blocks) σε οποιαδήποτε χρονική στιγμή, εκτελώντας παράλληλα όλα τα νήματα που έχει αναλάβει. Τα thread blocks πρέπει να σχεδιάζονται ως ανεξάρτητες εργασίες από τον προγραμματιστη καθώς δύνανται να δρομολογηθούν με οποιαδήποτε σειρά και σε οποιοδήποτε από τα διαθέσιμα SMs του επεξεργαστή γραφικών. Σε αυτό έρχεται να προστεθεί το γεγονός ότι το CUDA παρέχει μηχανισμούς συγχρονισμού μεταξύ των νημάτων που ανήκουν στο ίδιο block, όπως με χρήση της `__syncthreads()`, αλλά δεν μπορεί να οριστεί συγχρονισμός μεταξύ blocks του ίδιου πυρήνα, καθώς κάτι τέτοιο θα υπονόμευε τον μεγάλο βαθμό παράλληλης εκτέλεσης.

Τα νήματα κάθε block ομαδοποιούνται ανά 32 σε warps, τα οποία δρομολογούνται από ειδικούς warp schedulers. Όλα τα νήματα ενός warp εκτελούν κάθε στιγμή την ίδια εντολή. Αυτό σημαίνει ότι σε περίπτωση που υπάρχει διακλάδωση στον κώδικα ολόκληρο το warp θα εκτελέσει με τη σειρά τους πιθανούς κλάδους, απενεργοποιώντας κάθε φορά τα νήματα που δεν ανήκουν στον τρέχοντα κλάδο. Κάτι τέτοιο μπορεί να μειώσει σημαντικά την επίδοση, αφού μειώνεται ο αριθμός των χρήσιμων υπολογισμών.

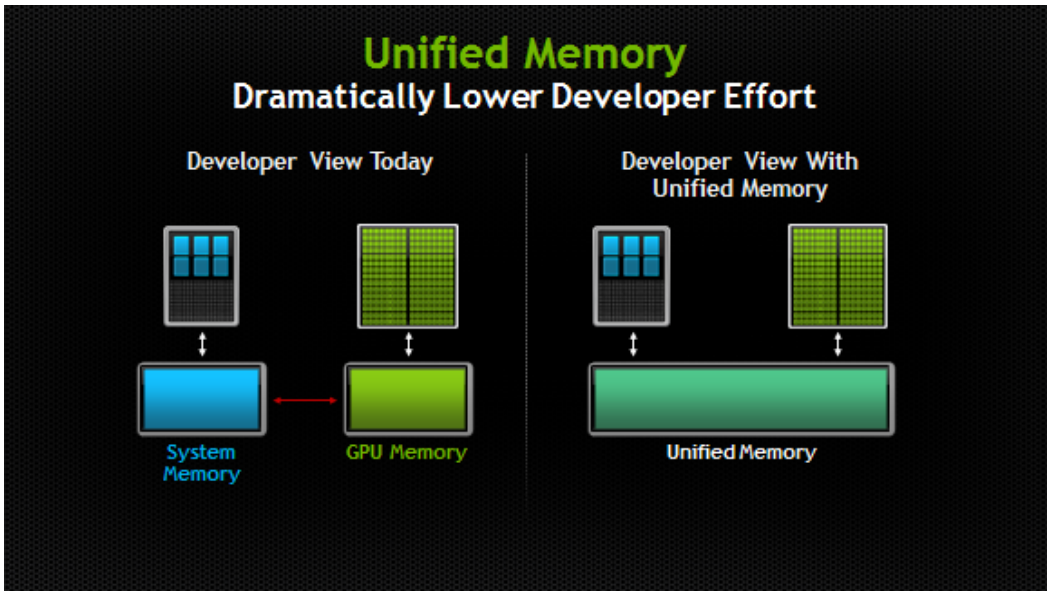
Η οργάνωση της μνήμης

Τέλος, η διάκριση των διαφόρων επιπέδων μνήμης στα οποία έχει πρόσβαση το εκάστοτε νήμα είναι σημαντική προκειμένου να επιτυγχάνεται υψηλή επίδοση. Αναλυτικότερα, κάθε νήμα έχει διαθέσιμη τοπική μνήμη και καταχωρητές, που του επιτρέπουν να διαφοροποιεί τους υπολογισμούς του από τα υπόλοιπα νήματα. Συγχρόνως, τα νήματα ενός block μπορούν να προσπελαύνουν μια ταχύτατη μοιραζόμενη μνήμη σε επίπεδο block (shared memory) (Σχήμα 1.6), η οποία δίνει τη δυνατότητα για εφαρμογή caching από τον προγραμματιστή. Για παράδειγμα, μπορεί το κάθε νήμα να φορτώνει ένα μέρος από τα δεδομένα στην μοιραζόμενη μνήμη και όλα τα νήματα του block να έχουν διαθέσιμο το σύνολο των δεδομένων με αμελητέο κόστος ανάγνωσης από την μοιραζόμενη μνήμη [Harris, 2013b]. Φυσικά, όλα τα νήματα έχουν πρόσβαση στην κεντρική μνήμη της GPU, τα περιεχόμενα της οποίας διατηρούνται ανάμεσα σε κλήσης πυρήνων από το ίδιο πρόγραμμα. Επίσης, όλα τα νήματα μπορούν να πραγματοποιούν αναγνώσεις από τη μνήμη σταθερών (constant memory) και από τη μνήμη υφών, που είναι βελτιστοποιημένη για προσβάσεις με τοπικότητα σε δύο διαστάσεις.



Σχήμα 1.6: Μοντέλο μνήμης CUDA. *source: [Hwu, 2012]*

Από την CUDA 6.0 και έπειτα έχει προστεθεί η έννοια της ενοποιημένης μνήμης (unified memory). Πρόκειται για έναν εικονικό χώρο διευθύνσεων, κοινό για τη CPU και τη GPU (Σχήμα 1.7). Με αυτόν τον τρόπο αυξάνεται η ευκολία προγραμματισμού, αφού οι μεταφορές από και προς τη μνήμη του επεξεργαστή γραφικών δεν γίνονται ρητά από τον χρήστη. Ωστόσο, αυτό το επίπεδο αφαίρεσης έρχεται με κόστος στην επίδοση και δεν θα χρησιμοποιηθεί σε αυτή την εργασία.



Σχήμα 1.7: Μοντέλο ενοποιημένης μνήμης. *source: developer.nvidia.com/blog/unified-memory-in-cuda-6*

Δυνατότητες συσκευών

Στον χώρο των συνεχώς εξελισσόμενων επεξεργαστών γραφικών είναι σκόπιμο να περιγράφονται οι δυνατότητες κάθε μιας, προκειμένου οι προγραμματιστές να προσαρμόζονται ανάλογα. Για αυτό χρησιμοποιείται ο δείκτης της "Δυνατότητας Συσκευής" (Compute capability) που σημειώνεται ως ένα ζεύγος αριθμών στη μορφή X.Y. Το X ονομάζεται "αριθμός major" και υποδεικνύει την αρχιτεκτονική πυρήνα, ενώ το Y είναι ο "αριθμός minor" που δηλώνει ορισμένες βελτιώσεις στην αρχιτεκτονική. Η Nvidia Tesla V100, έχει compute capability 7.0 (αρχιτεκτονική Volta) [Nvidia, 2017].

1.2 Πολλαπλασιασμός Πινάκων

Ήδη έχει υπογραμμιστεί το γεγονός ότι οι επεξεργαστές γραφικών μπορούν να προσφέρουν υψηλές επιδόσεις σε παράλληλους υπολογισμούς, όπως είναι οι πράξεις γραμμικής άλγεβρας. Ο πιο χαρακτηριστική πράξη είναι ο πολλαπλασιασμός πινάκων και για αυτό θα παρουσιαστούν ορισμένες λεπτομέρειες του υπολογισμού.

Ο ορισμός

Ο πολλαπλασιασμός πινάκων είναι μια πράξη που εφαρμόζεται ανάμεσα σε δύο πίνακες και δίνει ως αποτέλεσμα έναν τρίτο πίνακα. Ορίζεται όταν ο αριθμός των στηλών του πρώτου πίνακα ισούται με τον αριθμό των γραμμών του δεύτερου πίνακα και είναι μη-αντιμεταθετική πράξη. Ο πίνακας που δίνει το γινόμενο έχει αριθμό γραμμών ίσο με τον πρώτο πίνακα και αριθμό στηλών ίσο με τον δεύτερο πίνακα. Συμβολίζεται ως $A_{m\times k} \cdot B_{k\times n} = C_{m\times n}$, όπου οι δείκτες m, k, n δηλώνουν τις διαστάσεις του κάθε όρου. Αναλυτικότερα,

$$AB = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1k} \\ a_{21} & a_{22} & \dots & a_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mk} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{k1} & b_{k2} & \dots & b_{kn} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & \dots & c_{1n} \\ c_{21} & c_{22} & \dots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \dots & c_{mn} \end{bmatrix} = C$$

όπου κάθε στοιχείο του γινομένου υπολογίζεται χρησιμοποιώντας μια γραμμή του πίνακα A και μια στήλη του πίνακα B:

$$c_{ij} = \sum_{x=1}^{k} a_{ix} b_{xj}$$

Έτσι, ο συνολικός υπολογισμός αποτελείται από $m \times n$ ανεξάρτητους πολλαπλασιασμούς μεταξύ γραμμών του A και στηλών του B.

Σειριακός αλγόριθμος

Η απλούστερη υλοποίηση του αλγορίθμου για πολλαπλασιασμό πινάκων ακολουθεί τις σχέσεις ορισμού (Αλγόριθμος 1).

Αλγόριθμος 1: Πολλαπλασιασμός Πίνακα με Πίνακα (DMM)

Δεδομένα Εισόδου: Πίνακας $A[m][k]$, Πίνακας $B[k][n]$
Έξοδος: Πίνακας $C[m][n]$

```
1 for i = 0; i < m; i++ do
2     for j = 0; j < n; j++ do
3         sum = 0;
4         for x = 0; x < k; x++ do
5             sum = sum + A[i][x]B[x][j];
6         end
7         C[i][j] = sum;
8     end
9 end
```

Ο αλγόριθμος αποτελείται από τρεις εμφωλευμένες επαναλήψεις. Οι πρώτες δύο διατρέχουν τις γραμμές του A και τις στήλες του B, αντίστοιχα, ενώ η εσωτερικότερη υπολογίζει το γινόμενο "γραμμή επί στήλη", εκτελώντας το άθροισμα. Η πολυπλοκότητα του αλγορίθμου είναι $\Theta(m * n * k)$, δηλαδή $\Theta(n^3)$ για πολλαπλασιασμό τετραγωνικών πινάκων.

Βέβαια, έχουν αναπτυχθεί καλύτεροι αλγόριθμοι, όπως αυτός του Strassen. Σε αυτή την προσέγγιση οι αρχικοί πίνακες χωρίζονται σε τέσσερεις $n/2 \times n/2$ υποπίνακες. Ο υπολογισμός απαιτεί 7 πολλαπλασιασμούς υποπινάκων, κάτι που μειώνει την πολυπλοκότητα σε $O(n^{\log_2 7}) \simeq O(n^{2.807})$. [Strassen, 1969]

1.2.1 Παράλληλος πυρήνας σε επεξεργαστή γραφικών

Όπως έχει ήδη αναφερθεί, οι υπολογισμοί των στοιχείων του αποτελέσματος του πολλαπλασιασμού είναι πλήρως ανεξάρτητοι. Πράγματι, για να βρεθεί η τιμή του στοιχείου c_{ij} δεν χρειάζεται η γνώση κανενός άλλου στοιχείου $c_{i'j'}$. Συνεπώς, η διαδικασία μπορεί να γίνει πολύ ταχύτερη με την ανάθεση του προβλήματος σε $m \times n$ παράλληλα νήματα, που το καθένα θα υπολογίζει ένα στοιχείο του τελικού πίνακα. Κάτι τέτοιο είναι δυνατό σε μια GPU, η οποία μπορεί να εκτελεί ταυτόχρονα χιλιάδες νήματα. Επίσης, αυτή η κατάτμηση του υπολογισμού δεν αποτελεί πρόβλημα ακόμα και αν τα μεγέθη των πινάκων είναι μεγαλύτερα από το μέγιστο μέγεθος πλέγματος του επεξεργαστή γραφικών, αφού τότε μπορεί κάθε νήμα να αναλάβει τον υπολογισμό περισσότερων στοιχείων του πίνακα C, αξιοποιώντας και πάλι τον υψηλό βαθμό παραλληλισμού. Έτσι, θα γίνει αναλυτικότερη περιγραφή του παράλληλου πυρήνα υποθέτοντας ότι οι διαστάσεις των πινάκων είναι μικρότερες ή ίσες με τις διαστάσεις του μέγιστου πλέγματος νημάτων, αφού η επέκταση σε μεγαλύτερα μεγέθη θεωρείται τετριμμένη.

Όπως φαίνεται στον Αλγόριθμο 2 (υπενθυμίζεται ότι ο κώδικας του πυρήνα εκτελείται από όλα τα νήματα ταυτόχρονα) κάθε νήμα προσδιορίζει, αρχικά, τη γραμμή και τη στήλη του στοιχείου του C που θα υπολογίσει, μέσω του ***blockId*** και ***threadId*** του. Έπειτα, υπολογίζει το γινόμενο της γραμμής του A και της στήλης του B, που του αντιστοιχούν, και γράφει το αποτέλεσμα στη θέση του στοιχείου $C[row][col]$.

Αλγόριθμος 2: Παράλληλος πυρήνας DMM

Δεδομένα Εισόδου: Πίνακας $A[m][k]$, Πίνακας $B[k][n]$

Έξοδος: Πίνακας $C[m][n]$

```
1 row = blockIdx.y * blockDim.y + threadIdx.y;
2 col = blockIdx.x * blockDim.x + threadIdx.x;
3 sum = 0;
4 for x = 0; x < k; x++ do
5     sum = sum + A[row][x]B[x][col];
6 end
7 C[row][col] = sum;
```

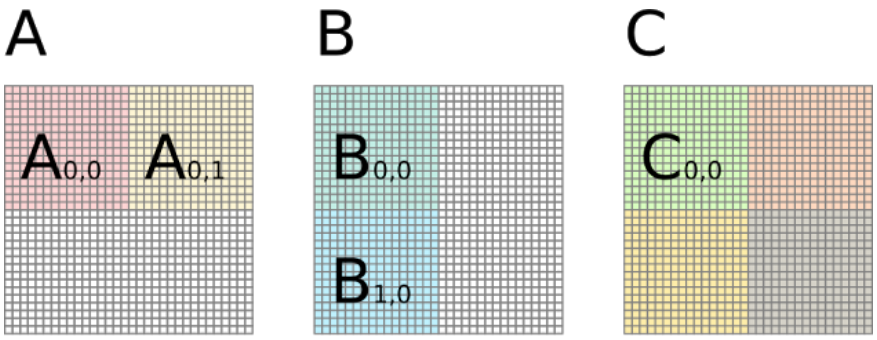
Όσον αφορά τη χρήση της μνήμης, το κάθε νήμα πραγματοποιεί k αναγνώσεις στον πίνακα A) και άλλες τόσες στον πίνακα B. Δεδομένου ότι συμμετέχουν $m \times n$ νήματα, προκύπτουν $m \times n \times k$ προσβάσεις σε καθέναν από τους πίνακες εισόδου. Ωστόσο, κάποιες από τις προσβάσεις συνενώνονται όταν πρόκειται για διαδοχικά νήματα που διαβάζουν διαδοχικές θέσεις μνήμης

[Harris, 2013a]. Παρόλα αυτά, το γεγονός ότι γίνονται πολλές προσβάσεις στην κύρια μνήμη της GPU για τα ίδια δεδομένα (για παράδειγμα κάθε γραμμή του A διαβάζεται n φορές) περιορίζει την επίδοση της υλοποίησης.

Πυρήνας με Tiles

Προκειμένου να μειωθούν οι προσβάσεις στην μνήμη, οι οποίες χρειάζονται πολλούς κύκλους ρολογιού και μειώνουν την επίδοση, οι πίνακες μπορούν να κατατμηθούν σε μικρούς υποπίνακες, που ονομάζονται tiles. Ο στόχος μιας τέτοιας υλοποίησης είναι να αξιοποιηθεί η μοιραζόμενη μνήμη (κοινή για τα νήματα του ίδιου *thread block*), η οποία είναι κατά πολύ γρηγορότερη της κύριας μνήμης και μπορεί να εξυπηρετήσει caching των tiles.

Σε αυτή την περίπτωση οι πίνακες θεωρούνται ως γραμμές και στήλες από tiles, τα οποία πολλαπλασιάζονται για να δώσουν τα tiles του πίνακα εξόδου. Όπως φαίνεται και στο Σχήμα 1.8, το tile $C_{0,0}$ προκύπτει από τον πολλαπλασιασμό της πρώτης γραμμής από tiles του A και της πρώτης στήλης από tiles του B: $C_{0,0} = A_{0,0}B_{0,0} + A_{0,1}B_{1,0}$. Πρόκειται, λοιπόν, για τις ίδιες σχέσεις με τον απλό πολλαπλασιασμό πινάκων, μόνο που πλέον τα γινόμενα $a_{ik}b_{kj}$ είναι γινόμενα πινάκων.



Σχήμα 1.8: Πολλαπλασιασμός πινάκων με tiles [Waeijen, 2018]

Με αυτόν τον τρόπο, και πάλι κάθε νήμα υπολογίζει ένα στοιχείο του πίνακα C, άλλα δεν διαβάζει ολόκληρη τη γραμμή και τη στήλη εισόδου απευθείας: το κάνει κατά tiles. Στη συνέχεια, θα θεωρηθεί ότι το μέγεθος του *thread block* και το μέγεθος του tile είναι ίδια, ώστε να απλουστευτεί η περιγραφή. Κατά αυτόν τον τρόπο, μπορεί κάθε νήμα να φορτώσει ένα στοιχείο του A και ένα του B στη μοιραζόμενη μνήμη και όλα τα νήματα του *block* να έχουν διαθέσιμα όλα τα στοιχεία του tile του A και του B που χρειάζονται. Για παράδειγμα, ο υπολογισμός του Σχήματος 1.8 θα γινόταν σε δύο φάσεις (επειδή ο διαχωρισμός έχει οδηγήσει σε δύο tiles ανά γραμμή του A και στήλη του B):

1. Τα νήματα του block υπολογίζουν παράλληλα το γινόμενο $A_{0,0}B_{0,0}$
2. Τα νήματα υπολογίζουν το γινόμενο $A_{0,1}B_{1,0}$ και το προσθέτουν στο προηγούμενο αποτέλεσμα

Επομένως στην πρώτη φάση χρειάζονται μόνο τα tiles $A_{0,0}$ και $B_{0,0}$ στη μοιραζόμενη μνήμη του *thread block*, και στη δεύτερη φάση μόνο τα $A_{0,1}$ και $B_{1,0}$. Γίνεται αντιληπτό ότι, έτσι, η μοιραζόμενη μνήμη λειτουργεί ως κρυφή μνήμη (cache), στην οποία φορτώνεται μία φορά το κάθε στοιχείο και χρησιμοποιείται από πολλά νήματα του *block*.

Αναλυτικότερα, στον Αλγόριθμο 3 φαίνεται ότι δεσμεύονται οι πίνακες-tiles A_{tile} και B_{tile} στην μοιραζόμενη μνήμη. Στη συνέχεια τα νήματα επεξεργάζονται τις γραμμές και τις στήλες σε $k/TILE_X$ φάσεις, που είναι ο αριθμός των tiles στα οποία χωρίζονται οι γραμμές και οι στήλες, αντίστοιχα (για απλούστευση θεωρείται ότι οι διαστάσεις είναι τέτοιες ώστε να διαιρούνται ακριβώς με το μέγεθος των tiles). Στις γραμμές 8 και 9 κάθε νήμα φορτώνει ένα στοιχείο του A και ένα του B) στη μοιραζόμενη μνήμη (στην πραγματικότητα δεν έχει σημασία ακριβώς ποιο νήμα του *thread block* θα φορτώσει κάθε στοιχείο). Μετά από αυτό το σημείο απαιτείται συγχρονισμός, ώστε κανένα νήμα να μην επιχειρήσει να επεξεργαστεί δεδομένα που δεν έχουν φορτωθεί ακόμα από κάποιο άλλο νήμα. Μετά τον υπολογισμό του γινομένου "υπογραμμή επί υποστήλη" (γραμμές 12-14) χρειάζεται και πάλι συγχρονισμός, προκειμένου να μην ξεκινήσει η μεταφορά δεδομένων της επόμενης φάσης πριν να έχει ολοκληρωθεί η τρέχουσα από όλα τα νήματα του *block*.

Αλγόριθμος 3: Πυρήνας DMM με tiles

Δεδομένα Εισόδου: Πίνακας $A[m][k]$, Πίνακας $B[k][n]$
Έξοδος: Πίνακας $C[m][n]$

```
1  __shared__ A_tile[TILE_Y][TILE_X];
2  __shared__ B_tile[TILE_X][TILE_Y];
3  sum = 0;
4  for tile_iterator = 0; tile_iterator < k/TILE_X; tile_iterator++ do
5      row_A = blockIdx.y * blockDim.y + threadIdx.y;
6      col_A = tile_iterator * blockDim.x + threadIdx.x;
7
8      A_tile[threadIdx.y][threadIdx.x] = A[row_A][col_A];
9      B_tile[threadIdx.x][threadIdx.y] = B[col_A][row_A];
10     __syncthreads();
11
12     for x = 0; x < TILE_X; x++ do
13         sum = sum + A_tile[threadIdx.y][x]B_tile[x][threadIdx.x];
14     end
15     __syncthreads();
16 end
17 row = blockIdx.y * blockDim.y + threadIdx.y;
18 col = blockIdx.x * blockDim.x + threadIdx.x;
19 C[row][col] = sum;
```

Ένα ακόμα πλεονέκτημα του διαχωρισμού των πινάκων σε tiles είναι ότι εξυπηρετεί κατανεμημένους υπολογισμούς. Περισσότερα για αυτό θα αναλυθούν στη συνέχεια, όταν συζητηθούν υπολογισμοί σε πολλαπλούς επεξεργαστές γραφικών (multi-GPU).

1.3 Οι βιβλιοθήκες BLAS

Η BLAS: Basic Linear Algebra Subprograms είναι μια προδιαγραφή για ρουτίνες γραμμικής άλγεβρας και περιλαμβάνει υπολογισμούς από την πρόσθεση διανυσμάτων μέχρι τον πολλαπλασιασμό πινάκων. Υλοποιείται από αρκετές βιβλιοθήκες, που περιλαμβάνουν και βελτιστοποιήσεις ανάλογα με την αρχιτεκτονική για την οποία σχεδιάζονται, με κάποιες από τις πιο διάσημες να είναι η cuBLAS (για επεξεργαστές γραφικών της Nvidia), η rocBLAS (AMD) και η OpenBLAS.

1.3.1 Βασικές ρουτίνες

Οι υλοποιήσεις BLAS χωρίζουν τις ρουτίνες σε τρία επίπεδα ανάλογα με τη διαστατικότητα των ορισμάτων.

Επίπεδο 1

Στο πρώτο επίπεδο ανήκουν οι υπολογιστικοί πυρήνες που πραγματοποιούν πράξεις μεταξύ διανυσμάτων. Ο βασικός πυρήνας εκτελεί την γενικευμένη διανυσματική πρόσθεση:

$$\mathbf{y} = a\mathbf{x} + \mathbf{y}$$

η οποία συνήθως αναφέρεται ως "axpy" ("a x plus y"). Επίσης, εδώ βρίσκεται ο πυρήνας του εσωτερικού γινομένου ("dot"):

$$r = \mathbf{x}^T\mathbf{y}$$

[Lawson et al., 1979]

Επίπεδο 2

Το δεύτερο επίπεδο περιέχει ρουτίνες για πράξεις ανάμεσα σε πίνακες και διανύσματα. Ο πιο χαρακτηριστικός πυρήνας πραγματοποιεί το γενικευμένο γινόμενο πίνακα με διάνυσμα (generalized matrix-vector, "gemv"):

$$\mathbf{y} = \alpha\mathbf{A}\mathbf{x} + \beta\mathbf{y}$$

όπου $\mathbf{A}$: πίνακας, $\mathbf{x}, \mathbf{y}$: διανύσματα και α, β: σταθερές. [Dongarra et al., 1988]

Επίπεδο 3

Στο τρίτο επίπεδο ανήκουν οι ρουτίνες για πράξεις ανάμεσα σε πίνακες. Εδώ υλοποιείται η ρουτίνα γενικευμένου πολλαπλασιασμού πίνακα με πίνακα (generalized matrix-matrix, "gemm"):

$$\mathbf{C} = \alpha\mathbf{A}\mathbf{B} + \beta\mathbf{C}$$

όπου $\mathbf{A}, \mathbf{B}, \mathbf{C}$: πίνακες και α, β: σταθερές. [Dongarra et al., 1990]

Αναπαράσταση των πινάκων

Οι βιβλιοθήκες που υλοποιούν BLAS επεξεργάζονται τους πίνακες λαμβάνοντάς τους σε μονοδιάστατη αναπαράσταση, όπως δηλαδή τοποθετούνται και στη μνήμη. Αυτό μπορεί να γίνει είτε κατά γραμμές (row-major), είτε κατά στήλες (column-major), όπως φαίνεται στον Πίνακα 1.1.

Διάταξη	Πίνακας A	Μονοδιάστατος πίνακας
row-major	$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \end{bmatrix}$	$\begin{bmatrix} a_{00}, & a_{01}, & a_{02}, & a_{10}, & a_{11}, & a_{12} \end{bmatrix}$
column-major	$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \end{bmatrix}$	$\begin{bmatrix} a_{00}, & a_{10}, & a_{01}, & a_{11}, & a_{02}, & a_{12} \end{bmatrix}$

Πίνακας 1.1: Αποθήκευση πινάκων κατά γραμμές (row-major) και κατά στήλες (column-major)

Με αυτόν τον τρόπο το στοιχείο a_{ij} τοποθετείται στη θέση $A[j + i * ld]$ στην αποθήκευση κατά γραμμές και στη θέση $A[i + j * ld]$ στην αποθήκευση κατά στήλες [BLAST, 2001]. Ο αριθμός ld είναι η βασική διάσταση του πίνακα (*leading dimension*):

$$ld = \begin{cases} \text{αριθμός στηλών} & \text{για αποθήκευση κατά γραμμές} \\ \text{αριθμός γραμμών} & \text{για αποθήκευση κατά στήλες} \end{cases} \tag{1.1}$$

Η επιλογή ανάμεσα στις δύο αναπαραστάσεις εξαρτάται από την υλοποίηση της εκάστοτε βιβλιοθήκης. Για παράδειγμα, η βιβλιοθήκη cuBLAS, η οποία θα χρησιμοποιηθεί κατά κόρον στην παρούσα εργασία, απαιτεί αποθήκευση κατά στήλες [Nvidia, 2023a]. Η εν λόγω βιβλιοθήκη παρέχει βελτιστοποιημένες συναρτήσεις για Nvidia GPUs και αναμένει τους πίνακες και τα διανύσματα εισόδου να βρίσκονται στη μνήμη της κάρτας γραφικών. Ωστόσο, ο τελευταίος περιορισμός μπορεί να αποφευχθεί στο μοντέλο ενοποιημένης μνήμης.

1.3.2 Πυρήνες BLAS σε πολλαπλές συσκευές (multi-GPU BLAS)

Προκειμένου να επιτευχθεί ακόμα μεγαλύτερος παραλληλισμός των πυρήνων BLAS έχουν αναπτυχθεί βιβλιοθήκες, οι οποίες χωρίζουν τον υπολογισμό σε υποπροβλήματα και τα μοιράζουν στους πολλαπλούς επεξεργαστές γραφικών που μπορεί να διαθέτει ένα σύστημα. Σε τέτοιες αρχιτεκτονικές, καίριο ρόλο διαδραματίζει η αναπαράσταση των πινάκων με tiles, που παρέχει άμεση κατάτμηση του προβλήματος. Βέβαια η επίδοση μιας υλοποίησης BLAS σε πολλαπλές GPUs και, ιδιαίτερα, σε ετερογενή συστήματα εξαρτάται από αρκετούς παράγοντες.

Το πρώτο ζήτημα που προκύπτει έχει να κάνει με την εξισορρόπηση του φόρτου μεταξύ των επεξεργαστών. Όπως συμβαίνει σε κάθε παράλληλη σχεδίαση, είναι κρίσιμο να αποφευχθεί άνιση κατανομή, που οδηγεί σε μείωση της ταχύτητας του υπολογισμού. Η πρώτη παράμετρος προς βελτιστοποίηση είναι το σχήμα διαμοιρασμού των πινάκων μεταξύ των GPUs. Τα πιο συχνά φαίνονται στο Σχήμα 1.9.

Επιπλέον, ο χρόνος που απαιτείται για επικοινωνία και μεταφορά των δεδομένων από και προς τις GPUs επηρεάζει αρνητικά την επίδοση. Για να μειωθεί αυτή η καθυστέρηση μπορεί να επικαλυφθεί η μεταφορά δεδομένων με εκτέλεση πυρήνων (communication-computation overlap). Αυτό γίνεται όταν οι επεξεργαστές γραφικών έχουν τη δυνατότητα να εκτελούν έναν υπολογιστικό πυρήνα και, ταυτόχρονα, να φορτώνουν τα δεδομένα του πυρήνα που πρόκειται να

GPU0	GPU0	GPU0	GPU0
GPU1	GPU1	GPU1	GPU1
GPU2	GPU2	GPU2	GPU2

(a)

GPU0	GPU1	GPU2	GPU0
GPU1	GPU2	GPU0	GPU1
GPU2	GPU0	GPU1	GPU2

(b)

GPU0	GPU0	GPU1	GPU1
GPU2	GPU2	GPU0	GPU0
GPU1	GPU1	GPU2	GPU2
GPU0	GPU0	GPU1	GPU1

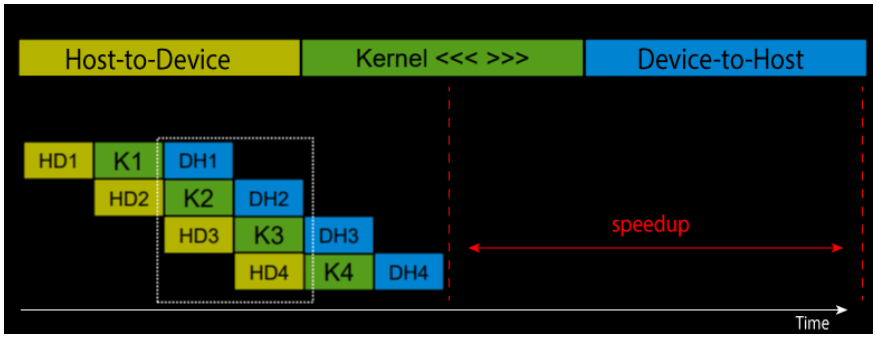
(c)

GPU0	GPU0	GPU1	GPU1
GPU0	GPU0	GPU1	GPU1
GPU2	GPU2	GPU0	GPU0
GPU2	GPU2	GPU0	GPU0

(d)

Σχήμα 1.9: Σχήματα διαμοιρασμού πινάκων (a) Σειριακό (b) Round-Robin (c) 1D Block-Cyclic (d) 2D Block-Cyclic [Nvidia, 2023a] [Ostrouchov, 1995]

ακολουθήσει στη μνήμη. Κάτι τέτοιο είναι εφικτό στις σύγχρονες GPUs, όπου ο προγραμματιστής μπορεί να ορίσει σωλήνωση των πυρήνων (pipelining). Πιο συγκεκριμένα το Σχήμα 1.10 δείχνει ότι ο υπολογισμός ενός πυρήνα αποτελείται από τρία στάδια: μεταφορά δεδομένων προς την GPU (Host-to-Device, HD), επεξεργασία (kernel), μεταφορά δεδομένων προς την CPU (Device-to-Host, DH). Αυτά είναι δυνατόν να επικαλύπτονται μεταξύ διαδοχικών κλήσεων πυρήνα πετυχαίνοντας παραλληλισμό τριών επιπέδων (3-way concurrency). Αξίζει να σημειωθεί ότι συσκευή-αποστολέας και παραλήπτης των δεδομένων μπορεί να είναι και κάποια άλλη GPU.



Σχήμα 1.10: Παραλληλισμός τριών επιπέδων [Rennich, 2011]

cuBLASXt

Η διεπαφή που παρέχει η Nvidia για διαμοιρασμό πυρήνων BLAS σε πολλαπλές GPUs είναι η cuBLASXt. Υποστηρίζει αποκλειστικά πράξεις BLAS που ανήκουν στο τρίτο επίπεδο, αφού είναι και οι πιο απαιτητικές σε υπολογιστικούς πόρους. Δίνει τη δυνατότητα για εκκίνηση πυρήνων έχοντας τα δεδομένα είτε στην CPU, είτε στη μνήμη κάποιας GPU και το cuBLASXt αναλαμβάνει την κατανομή των πινάκων.

Οι πίνακες χωρίζονται σε tiles και μοιράζονται στις συσκευές με round-robin πολιτική. Επίσης, χρησιμοποιείται επικάλυψη μεταξύ μεταφοράς δεδομένων και υπολογισμού, με τα tiles της γραμμής και της στήλης εισόδου που επεξεργάζεται η κάθε GPU να διοχετεύονται με σωλήνωση.[Nvidia, 2023a]

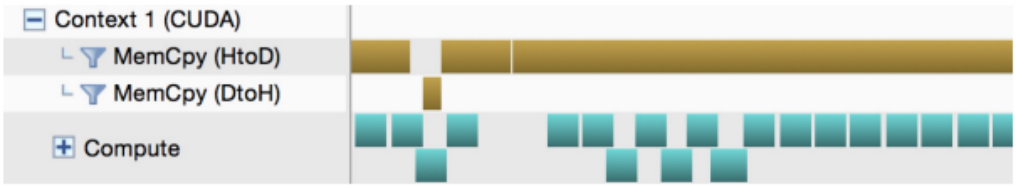
BLASX

Η βιβλιοθήκη BLASX σχεδιάστηκε για να επιτυγχάνει υψηλότερες επιδόσεις σε σχέση με τις μέχρι τότε βιβλιοθήκες για συστήματα πολλαπλών επεξεργαστών γραφικών, συμπεριλαμβανομένης της cuBLASXt [Wang et al., 2016]. Αρχικά, υλοποιεί δυναμικό χρονοπρογραμματισμό των εργασιών (dynamic task scheduling), στη θέση του στατικού σχήματος της cuBLASXt. Για να το καταφέρει αυτό χρησιμοποιεί μια καθολική ουρά εργασιών, στην οποία οι CPUs προσθέτουν εργασίες (πολλαπλασιασμούς μεταξύ tiles) και κάθε GPU αποσπά μια εργασία όταν έχει ολοκληρώσει την προηγούμενη που ανέλαβε. Έτσι, το συνολικό φορτίο μοιράζεται σε πραγματικό χρόνο, βάση των δυνατοτήτων κάθε κάρτας γραφικών, οδηγώντας σε μεγαλύτερο βαθμό εξισορρόπησης, ιδιαίτερα σε ετερογενή συστήματα.

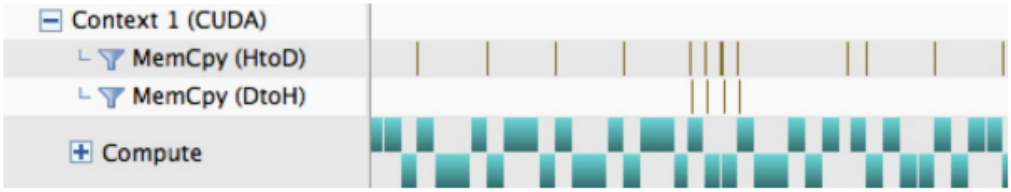
Επιπλέον, η BLASX καταφέρνει μείωση του κόστους της μεταφοράς δεδομένων. Υλοποιεί στο λογισμικό μια ιεραρχία κρυφών μνημών δύο επιπέδων για τα tiles των πινάκων. Ως πρώτο επίπεδο (L1 tile cache) θεωρεί την μνήμη της εκάστοτε GPU, και αντιμετωπίζει τη συνολική μνήμη όλων των GPUs του συστήματος σαν το δεύτερο επίπεδο (L2 cache). Αυτό σημαίνει ότι πολλές από τις μεταφορές δεδομένων που γινόταν μεταξύ CPU και GPU, εδώ γίνονται από GPU σε GPU. Κάτι τέτοιο είναι ωφέλιμο για δύο λόγους: (1) η μεταφορές είναι γρηγορότερες και οδηγούν πιο αργά σε κορεσμό τον δίαυλο, και (2) αξιοποιείται η χρονική τοπικότητα των tiles καθώς κάποια επαναχρησιμοποιούνται. Στο Σχήμα 1.11 γίνεται εμφανής η συρρίκνωση του κόστους επικοινωνίας. Φυσικά, εφαρμόζεται και η επικάλυψη υπολογισμού και επικοινωνίας, που έχει ήδη περιγραφεί.

XKBLAS

Με την XKBLAS οι σχεδιαστές της κατάφεραν ακόμα υψηλότερες επιδόσεις. Αυτό ήταν δυνατό με τη χρήση του XKaapi συστήματος, το οποίο δημιουργεί κατά την εκτέλεση τις υποεργασίες που απαιτούνται για τον υπολογισμό, τις εξαρτήσεις δεδομένων μεταξύ τους και αναθέτει αυτές τις εργασίες στις GPUs. Επιπλέον, στο σχήμα κρυφής μνήμης της XKBLAS τα δεδομένα μεταφέρονται από τον πιο κοντινό κόμβο (GPU ή CPU) με βάση την τοπολογία του συστήματος και γράφονται πίσω ασύγχρονα. [Gautier and Lima, 2020]



(a)



(b)

Σχήμα 1.11: Στιγμιότυπο GPU που συμμετέχει σε πολλαπλασιασμό πινάκων: (a) cuBLASXt (b) BLASX [Wang et al., 2016]

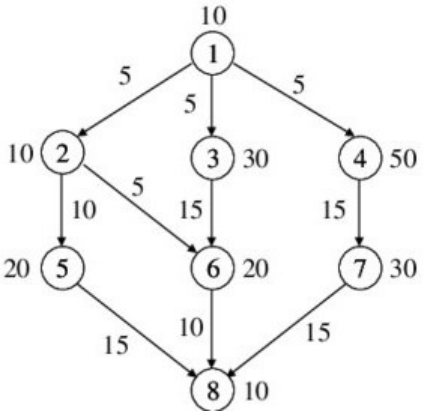
1.4 Παράλληλη εκτέλεση εργασιών

Η έννοια των παράλληλων εργασιών έχει ήδη αναφερθεί, όμως κρίνεται σκόπιμο να γίνει μια περισσότερο λεπτομερής περιγραφή, αφού είναι το κεντρικό ζήτημα που θα μας απασχολήσει στη συνέχεια. Ως εργασία μπορεί να οριστεί ένα διακριτό κομμάτι έργου, που αφορά επεξεργασία κάποιων δεδομένων. Στο πλαίσιο του πολλαπλασιασμού πινάκων, για παράδειγμα, ως εργασία (task) μπορεί να θεωρηθεί ο πολλαπλασιασμός μιας γραμμής του πρώτου πίνακα με μια στήλη του δεύτερου. Μια λίγο διαφορετική προσέγγιση είναι να χωριστούν οι πίνακες σε ορθογώνια ή τετράγωνα τμήματα (που ονομάζονται tiles) και ο πολλαπλασιασμός δύο τέτοιων τμημάτων να θεωρηθεί ως η βασική εργασία.

Φαίνεται, λοιπόν, πως ο καθορισμός των εργασιών μπορεί να γίνει με πολλούς τρόπους από τον προγραμματιστή, ανάλογα με τη δομή του προβλήματος. Πιο συγκεκριμένα, κάθε ορισμός εργασίας υπονοεί και διαχωρισμό των δεδομένων που χρειάζεται η εν λόγω εργασία. Αναφερόμενοι, και πάλι, στο παράδειγμα των εργασιών στον πολλαπλασιασμό πινάκων μπορούμε να δούμε ότι αν ορίσουμε σαν βασική εργασία τον πολλαπλασιασμό μιας γραμμής με μια στήλη τα δεδομένα που χρειάζεται αυτή η εργασία είναι ακριβώς αυτά: μια γραμμή του πρώτου πίνακα και μια στήλη του δεύτερου. Στην δεύτερη περίπτωση, όπου η εργασία αφορά τον πολλαπλασιασμό δύο tiles, τα δεδομένα είναι ένα tile του ενός πίνακα και ένα του δεύτερου. Έτσι, διαφορετικές κατανομές δεδομένων σε διαφορετικά προβλήματα μπορούν να οδηγήσουν σε διαφορετικούς ορισμούς των βασικών εργασιών, με στόχο την μείωση των εξαρτήσεων μεταξύ των εργασιών.

Η καλύτερη επιλογή είναι εκείνη που οδηγεί στον μεγαλύτερο βαθμό παράλληλης εκτέλεσης. Σε κάποια προβλήματα αυτή η διαδικασία παράγει πλήρως ανεξάρτητες εργασίες, δηλαδή εργασίες που δύνανται να εκτελεστούν παράλληλα. Δυστυχώς, σε πολλές περιπτώσεις μια τέτοια επιλογή δεν είναι εφικτή, καθώς η φύση του προβλήματος επιβάλει εξαρτήσεις μεταξύ των εργασιών. Οι εξαρτήσεις που αναφέρουμε είναι εξαρτήσεις δεδομένων (data dependecies) που υφίστανται όταν κάποια εργασία παράγει δεδομένα που χρειάζεται μια άλλη εργασία. Σε τέτοιες περιπτώσεις ο προγραμματιστής είναι υποχρεωμένος να ορίσει σειριακή εκτέλεση αυτών των εργασιών προκειμένου να παράγεται το σωστό αποτέλεσμα.

Σε υψηλό επίπεδο οι εξαρτήσεις μεταξύ εργασιών μπορούν να αναπαρασταθούν από γράφους εξαρτήσεων (task graphs). Οι εν λόγω γράφοι είναι ακυκλικοί και κατευθυνόμενοι, με τους κόμβους να αναπαριστούν εργασίες και τις ακμές να δείχνουν τις εξαρτήσεις. Σε αυτό το πλαίσιο κάθε εργασία είναι έτοιμη και μπορεί να ξεκινήσει να εκτελείται αφού έχουν ολοκληρωθεί όλες οι εργασίες που σημειώνονται ως πρόγονοί της στον γράφο. Σε ορισμένες περιπτώσεις, οι γράφοι σχεδιάζονται αναγράφοντας σε κάθε κόμβο το ”φορτίο” ή τον χρόνο εκτέλεσης της εργασίας που αναπαριστά, ενώ στις κατευθυνόμενες ακμές μπορεί να σημειώνεται και ο όγκος των δεδομένων που χρειάζεται να μεταφερθούν. Ένας τέτοιος γράφος φαίνεται στο Σχήμα 1.12. Με αυτόν τον τρόπο, εντοπίζεται ο μέγιστος βαθμός παραλληλισμού και το κρίσιμο μονοπάτι, που είναι η μέγιστη διαδρομή από τον αρχικό μέχρι τον τελικό κόμβο. Τις περισσότερες φορές, οι γράφοι εξαρτήσεων χρησιμοποιούνται στο επίπεδο της σχεδίασης του αλγορίθμου από τον προγραμματιστή. Ωστόσο υπάρχουν και βιβλιοθήκες που επιτρέπουν τον απευθείας ορισμό τέτοιων γράφων σε προγραμματιστικό επίπεδο, όπως η Threading Building Blocks (TBB) και το CUDA. [McCool et al., 2012]



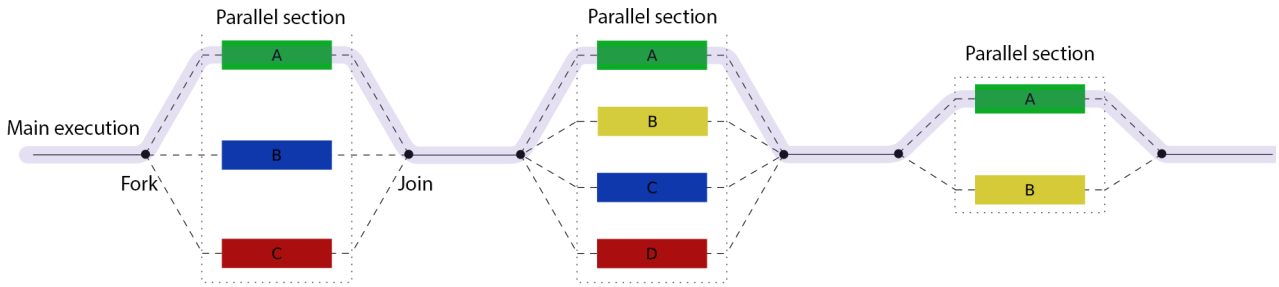
Σχήμα 1.12: Γράφος εξαρτήσεων [Kim, 2016]

1.4.1 Μοτίβα καθορισμού εργασιών

Στον χώρο της σχεδίασης παράλληλων προγραμμάτων έχουν προκύψει προγραμματιστικά μοτίβα για τον καθορισμό των εργασιών και των μεταξύ τους εξαρτήσεων.

Μοτίβο Fork–Join

Στο μοτίβο *fork-join* η σειριακή εκτέλεση του προγράμματος διακλαδίζεται (*fork*) και κάθε κλάδος εκτελείται παράλληλα, μέχρι την συνένωσή του με τους υπόλοιπους κλάδους (*join*), όπως φαίνεται στο Σχήμα 1.13. Με τον τρόπο αυτό μπορούν να ορίζονται παράλληλες εργασίες (A, B, C του σχήματος) και να τηρούνται οι εξαρτήσεις μεταξύ εργασιών που ανήκουν σε διαφορετικά παράλληλα τμήματα. Με άλλα λόγια η πράξη *join* επιβάλει σειριακή εκτέλεση και μπορεί να ορίσει αναμονή μιας εργασίας που εξαρτάται από κάποια άλλη. Για παράδειγμα η εργασία B του δεύτερου παράλληλου τμήματος μπορεί να εξαρτάται από την εργασία A του πρώτου τμήματος στο Σχήμα 1.13. Το μοτίβο επιτρέπει και αναδρομική διακλάδωση και, έτσι, μπορεί να υλοποιήσει τη λογική ενός γράφου εξαρτήσεων. [McCool et al., 2012]



Σχήμα 1.13: Μοτίβο Fork-Join [Kristensen, 2012]

Μοτίβο Futures

Το μοτίβο *futures* επιτρέπει και αυτό την παράλληλη εκτέλεση εργασιών μέσω ασύγχρονων κλήσεων. Πιο συγκεκριμένα δημιουργούνται εργασίες που επιστρέφουν αμέσως (πριν την ολοκλήρωσή τους) στον χρήστη ένα αντικείμενο, το οποίο συνήθως ονομάζετε *future* ή *premise*. Αυτό το αντικείμενο μπορεί να χρησιμοποιηθεί για να περιμένει (wait) την ολοκλήρωσή της

κάποια άλλη εργασία. Η συγκεκριμένη λειτουργία παρέχει άμεσα τη δυνατότητα περιγραφής σχέσεων εξάρτησης μεταξύ εργασιών και, κατ' επέκταση, την περιγραφή οποιουδήποτε γράφου εξαρτήσεων. [McCool et al., 2012]

Μοτίβο ουράς εργασιών

Μια ουρά εργασιών (task queue) παρέχει μια FIFO δομή (First-In-First-Out) για την αποθήκευση εργασιών που θα εκτελεστούν ασύγχρονα, με τη σειρά που εισάγονται στην ουρά. Σε αυτό το προγραμματιστικό μοτίβο υπάρχει ένα τμήμα του κώδικα (που απαντάται ως "παραγωγός"), το οποίο δημιουργεί εργασίες και τις προσθέτει στην ουρά, και ένα άλλο τμήμα (ο "καταναλωτής"), το οποίο αφαιρεί μια-μια τις εργασίες από την ουρά και τις εκτελεί. Στις υλοποιήσεις αυτού του σχήματος θεωρείται ότι η προσθήκη μιας εργασίας στην ουρά απαιτεί αμελητέο χρόνο σε σχέση με την εκτέλεσή της και, κατά συνέπεια, η εκτέλεση είναι ασύγχρονη.

Για να επιτευχθεί παράλληλη εκτέλεση συνυπάρχουν πολλές ουρές και οι εργασίες που βρίσκονται σε διαφορετικές ουρές μπορούν να εκτελούνται ταυτόχρονα. Επίσης, οι γλώσσες προγραμματισμού και οι βιβλιοθήκες που υλοποιούν αυτό το μοντέλο ενσωματώνουν και αντικείμενα που ονομάζονται "γεγονότα" (events). Ο προγραμματιστής μπορεί να προσθέσει ένα "γεγονός καταγραφής" σε μία ουρά και ένα "γεγονός αναμονής" σε μια άλλη. Με τον τρόπο αυτό οι εργασίες που βρίσκονται μετά από το "γεγονός" στη δεύτερη ουρά δεν θα δρομολογηθούν για εκτέλεση μέχρι να έχουν ολοκληρωθεί όλες οι εργασίες που βρίσκονται πριν από το "γεγονός" της πρώτης ουράς. Έτσι, επιβάλλονται οι εξαρτήσεις μεταξύ των εργασιών. [McCool et al., 2012]

1.4.2 Σχετικές βιβλιοθήκες

Τα μοτίβα που παρουσιάστηκαν είναι γενικά και υλοποιούνται για πολλές γλώσσες προγραμματισμού και εφαρμογές. Χρησιμοποιούνται σε όλο το φάσμα από παράλληλο προγραμματισμό χαμηλού επιπέδου, μέχρι προγραμματισμό ιστοσελίδων. Στην παρούσα εργασία, ωστόσο, το ενδιαφέρον βρίσκεται σε υλοποιήσεις χαμηλού επιπέδου, που μπορούν να χρησιμοποιηθούν στην ανάθεση παράλληλου έργου σε επεξεργαστές γραφικών. Για αυτόν τον λόγο θα παρουσιαστούν περισσότερες λεπτομέριες του CUDA, του OpenCL, της βιβλιοθήκης των POSIX threads και της διεπαφής HIP.

CUDA

Το προγραμματιστικό μοντέλο CUDA ορίζει ένα σύνολο εργασιών που επιτρέπεται από το υλικό (Nvidia GPUs) να εκτελεστούν παράλληλα. Αυτές, όπως αναφέρονται στο [Nvidia, 2023b], είναι:

- Υπολογισμός στον "host" (CPU)
- Υπολογισμός σε "device" (GPU)
- Μεταφορά δεδομένων από τον "host" στο "device" (H2D) (μέχρι μία μεταφορά σε αυτή την κατεύθυνση)
- Μεταφορά δεδομένων από το "device" στον "host" (D2H) (μέχρι μία μεταφορά σε αυτή την κατεύθυνση)

- Μεταφορά δεδομένων εντός της μνήμης κάποιου ”device”
- Μεταφορά δεδομένων από ”device” σε ”device”

Επιπλέον, αρκετές Nvidia GPUs (με υπολογιστική δυνατότητα μεγαλύτερη από 2.0) μπορούν να εκτελούν παράλληλα πυρήνες στο υλικό τους, όταν οι απαιτήσεις αυτών των πυρήνων σε μνήμη το επιτρέπουν. Συγκεκριμένα, οι Nvidia GeForce GTX 1060 και Nvidia Tesla V100 που χρησιμοποιήθηκαν για αυτή την εργασία υποστηρίζουν παράλληλη εκτέλεση πυρήνων.

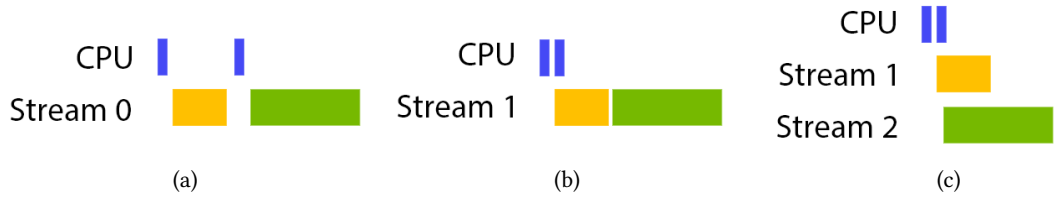
Για την παράλληλη εκτέλεση αυτών των εργασιών το CUDA παρέχει δομές που ονομάζονται *streams*. Πρόκειται για υλοποίηση ουράς εργασιών, όπου κάθε εργασία εκτελείται στην GPU που σχετίζεται με το εκάστοτε *stream* με βάση τη σειρά εισαγωγής της στη δομή. Με αυτόν τον τρόπο, εργασίες που τοποθετούνται στο ίδιο *stream* εκτελούνται σειριακά, ενώ εργασίες που βρίσκονται σε διαφορετικά *streams* μπορούν να εκτελεστούν παράλληλα στον ίδιο επεξεργαστή γραφικών, εφόσον ανήκουν στις δυνητικά παράλληλες εργασίες που αναφέρθηκαν παραπάνω. Για παράδειγμα μια μεταφορά δεδομένων H2D και μια μεταφορά D2H θα πραγματοποιηθούν παράλληλα εάν βρίσκονται σε διαφορετικά *streams*. Αντίθετα, δύο μεταφορές προς την ίδια κατεύθυνση (έστω H2D) δεν μπορούν να γίνουν παράλληλα, ακόμα και αν χρησιμοποιηθούν διαφορετικά *streams*. [Luitjens, 2015]

Η επικάλυψη μιας H2D και μιας D2H μεταφοράς δεδομένων είναι, λοιπόν, εφικτή με τη χρήση των ασύγχρονων κλήσεων `cudaMemcpyAsync` (μπορεί να ελεγχθεί η υποστήριξη από το υλικό με την παράμετρο `asyncEngineCount`, αν και υποστηρίζεται ευρέως από τις σύγχρονες GPUs). Ωστόσο, αυτές οι κλήσεις μπορούν να γίνουν μόνο από και προς μνήμη CPU που έχει οριστεί ως μνήμη ”κλειδωμένων σελίδων” (*page-locked* ή *pinned memory*), κάτι που γίνεται με την συνάρτηση `cudaMallocHost` αντί της `malloc`. Αυτό σημαίνει πως πρόκειται για διευθύνσεις που ανήκουν σε σελίδες μνήμης οι οποίες δεν επιτρέπεται να αντικατασταθούν (page-out) από το λειτουργικό. Με αυτόν τον τρόπο μπορεί να εφαρμόζεται το σχήμα παραλληλισμού τριών επιπέδων (3-way concurrency) που περιγράφηκε στην ενότητα 1.3.2. [Nvidia, 2023b]

Είναι σημαντικό να αναφερθεί πως ακόμα και όταν ο προγραμματιστής δεν τοποθετεί ρητά μια εργασία σε κάποιο *stream*, το σύστημα του CUDA αναθέτει αυτή την εργασία στο *default stream*. Η ιδιαιτερότητα αυτού του *stream* είναι ότι είναι πάντα συγχρονισμένο με τον κώδικα που εκτελεί η CPU, και δεν παρέχει παράλληλη εκτέλεση των εργασιών.

Μια ακόμα σημαντική λειτουργία είναι αυτή των ”συναρτήσεων CPU” (host functions). Πρόκειται για συναρτήσεις που μπορούν να τοποθετηθούν σε κάποιο *stream* με την κλήση `cudaLaunchHostFunc`. Αντιμετωπίζονται όπως και οι υπόλοιπες εργασίες του *stream*, με τη διαφορά ότι όταν έρθει η σειρά τους εκτελούνται στη CPU και όχι στην GPU. Αυτό δίνει τη δυνατότητα για χρονισμό host και device υπολογισμών, καθώς τηρείται η σειρά εισαγωγής των εργασιών στο *stream*.

Βέβαια, η διεπαφή παρέχει και άλλους κρίσιμους μηχανισμούς συγχρονισμού. Αρχικά, μπορεί να οριστεί αναμονή του host κώδικα μέχρι να ολοκληρωθούν όλες οι εργασίες που αφορούν κάποια συσκευή (device, GPU) με κλήση της `cudaDeviceSynchronize`. Προχωρώντας, σε λεπτότερα επίπεδα συγχρονισμού, η CPU μπορεί να συγχρονιστεί με κάποιο συγκεκριμένο *stream* (δηλαδή να περιμένει μέχρι να έχουν ολοκληρωθεί όλες οι εργασίες του), χρησιμοποιώντας την `cudaStreamSynchronize` και δίνοντάς της το αναγνωριστικό του ζητούμενου *stream*. Τέλος,



Σχήμα 1.14: Εκτέλεση εργασιών σε *streams*. *source: CUDA Streams: Best Practices and Common Pitfalls*

υφίσταται συγχρονισμός είτε ανάμεσα σε *stream* και host, είτε ανάμεσα σε *streams* με τα "γεγονότα" (*events*) που ορίζει το CUDA.

Ένα CUDA *event* είναι ένα αντικείμενο που μπορεί να τοποθετηθεί σε *stream*. Θεωρείται ότι το *event* είναι στην κατάσταση "μη ολοκληρωμένο" όταν δεν έχουν ολοκληρωθεί όλες οι εργασίες που βρισκόταν πριν από αυτό στο *stream*, και "ολοκληρωμένο" σε αντίθετη περίπτωση. Είναι σημαντική η σημείωση πως η αρχική (default) κατάσταση ενός *event* όταν έχει δημιουργηθεί, αλλά δεν έχει τοποθετηθεί ακόμα σε *stream*, είναι "ολοκληρωμένο" [Luitjens, 2015]. Η κατάστασή του μπορεί να διαπιστωθεί σε οποιαδήποτε στιγμή με την συνάρτηση `cudaEventQuery`, η οποία επιστρέφει `cudaSuccess` όταν έχει ολοκληρωθεί το *event* και `cudaErrorNotReady` εάν δεν έχει ολοκληρωθεί ακόμα. Ένα *event* τοποθετείτε σε *stream* και λέμε ότι "καταγράφεται" (record), με χρήση της `cudaEventRecord` και παρέχει δύο τρόπους συγχρονισμού:

1. Συγχρονισμό με τον host: η CPU μπορεί να περιμένει την ολοκλήρωση ενός *event* με την `cudaEventSynchronize` (Σημείωση: εξαιτίας του τρόπου που αρχικοποιείται η κατάσταση των *events*, η `cudaEventSynchronize` θα επιστρέψει αμέσως, χωρίς να μπλοκάρει, εάν κληθεί πριν την `cudaEventRecord`).

2. Συγχρονισμό μεταξύ *streams*: με τη συνάρτηση `cudaStreamWaitEvent(stream_i, event)` το *stream_i* θα περιμένει την ολοκλήρωση του *event*, χωρίς να προχωρήσει στην εκτέλεση των εργασιών που τοποθετήθηκαν σε αυτό μετά την κλήση της `cudaStreamWaitEvent`.

OpenCL

Το OpenCL παρέχει αντίστοιχες δυνατότητες. Αρχικά, η διαχείριση της μνήμης γίνεται με "αντικείμενα μνήμης" (memory objects), με τα πιο συνηθισμένα να είναι τα αντικείμενα τύπου *buffer* που αντιστοιχούν σε συνεχόμενη, γραμμική μνήμη. Κατά τη δημιουργία αντικειμένου *buffer* με την κλήση `clCreateBuffer`, μπορούν να οριστούν ιδιότητες που σχετίζονται με τη χρήση του. Σε αυτές περιλαμβάνονται σημαίες για τη δήλωση του *buffer* ως αντικείμενο μόνο για ανάγνωση ή μόνο για εγγραφή από τον πυρήνα ή τη CPU, αντίστοιχα (`CL_MEM_READ_ONLY`, `CL_MEM_WRITE_ONLY`, `CL_MEM_HOST_WRITE_ONLY`, `CL_MEM_HOST_READ_ONLY`). Ιδιαίτερα σημαντική είναι η σημαία `CL_MEM_ALLOC_HOST_PTR`, με την οποία δηλώνονται *buffers* για μνήμη κλειδωμένων σελίδων, που όπως ήδη υπογραμμίστηκε, δίνει τη δυνατότητα για ταχύτερες μεταφορές δεδομένων και επικάλυψη υπολογισμού και μεταφοράς μέσω DMA. [Khronos, 2023] [Nvidia, 2011]

Οι ουρές εργασιών (*command queues*) είναι ακόμα πιο κεντρικής σημασίας στο OpenCL. Δημιουργούνται με την κλήση `clCreateCommandQueueWithProperties` (η

clCreateCommandQueue θεωρείται ξεπερασμένη μετά την έκδοση 2.0) και οποιαδήποτε εργασία προορίζεται για εκτέλεση σε GPU πρέπει να τοποθετηθεί στην αντίστοιχη ουρά. Οι ουρές του OpenCL μπορούν να είναι ”in-order” ή ”out-of-order”, κάτι που ορίζεται με τη σημαία CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE κατά τη δημιουργία της ουράς. Η διαφορά των δύο τύπων ουρών είναι η προφανής: μια ”in-order” ουρά φροντίζει για τη σειριακή εκτέλεση των εργασιών στη GPU, βάσει της σειράς εισαγωγής τους στη δομή, ενώ μια ”out-of-order” εκτελεί τις εργασίες με οποιαδήποτε σειρά. Μια ”out-of-order” ουρά μπορεί ακόμα και να αποστείλει εργασίες για ταυτόχρονη εκτέλεση στη συσκευή.

Οι πιο κοινές εργασίες που τοποθετούνται σε κάποια ουρά είναι αντίστοιχες με το μοντέλο του CUDA. Εδώ η μεταφορά δεδομένων προς τη μνήμη της συσκευής (H2D) γίνεται με την κλήση clEnqueueWriteBuffer, η μεταφορά προς την αντίθετη κατεύθυνση γίνεται με την clEnqueueReadBuffer και η εκκίνηση πυρήνα (*kernel launch*) γίνεται, και πάλι, εισάγοντας τον πυρήνα στην ουρά με την clEnqueueNDRangeKernel. Επομένως, για την εκτέλεση ενός πυρήνα σε ”in-order” ουρά ακολουθείται το μονοπάτι:

1. Τοποθετείται στην ουρά η εντολή μεταφοράς δεδομένων στη συσκευή: clEnqueueWriteBuffer.
2. Τοποθετείται η εντολή εκτέλεσης του πυρήνα: clEnqueueNDRangeKernel.
3. Τοποθετείται η εντολή μεταφοράς δεδομένων στον *host*: clEnqueueReadBuffer.
4. Συγχρονίζεται ο *host* με την ουρά με clFinish, ώστε να γνωρίζει ότι έχει ολοκληρωθεί η διαδικασία και τα δεδομένα βρίσκονται στη δική του μνήμη.

Το τελευταίο βήμα είναι απαραίτητο μόνο όταν οι μεταφορές δεδομένων ορίζονται ”χωρίς φραγή” (*non-blocking*). Αυτό καθορίζεται από το τρίτο όρισμα των clEnqueueWriteBuffer και clEnqueueReadBuffer που μπορεί να είναι CL_TRUE, δηλαδή *blocking* ή CL_FALSE για *non-blocking*. Στην περίπτωση των *blocking* εντολών γίνεται η τοποθέτηση στην ουρά και η CPU αναμένει την ολοκλήρωσή της πριν προχωρήσει σε επόμενες εντολές. Αντίθετα, οι *non-blocking* εντολές δρομολογούνται για εκτέλεση ασύγχρονα με τον *host*. Ωστόσο, είναι αποδοτικότερο να χρησιμοποιούνται οι *non-blocking* κλήσεις και η CPU να συγχρονίζεται με κάποια ουρά μόνο όταν είναι απολύτως απαραίτητο.

Ταυτόχρονα, το OpenCL διαθέτει ένα ευρύ σχήμα ”γεγονότων” (*events*) που μπορούν να βοηθήσουν τον συγχρονισμό ουρών και εργασιών. Αρχικά, κάθε τοποθέτηση σε ουρά επιστρέφει (μέσω ορίσματος) ένα αντικείμενο τύπου cl_event, το οποίο εκφράζει το αν έχει ολοκληρωθεί η συγκεκριμένη εντολή και επιτρέπει σε άλλες εργασίες να περιμένουν την ολοκλήρωση. Η κατάσταση αυτού του *event* μπορεί να είναι:

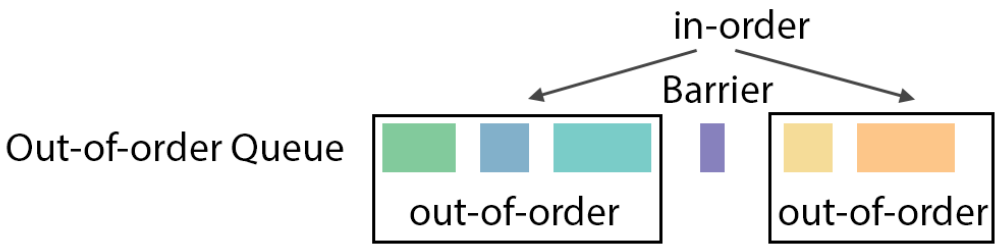
- CL_QUEUED: Όταν η σχετική εργασία έχει τοποθετηθεί στην ουρά
- CL_SUBMITTED: Όταν η εργασία έχει αποσταλεί στη συσκευή
- CL_RUNNING: Όταν έχει ξεκινήσει η εκτέλεση της εργασίας στη συσκευή
- CL_COMPLETE: Μετά την ολοκλήρωση της εκτέλεσης

Οποιαδήποτε άλλη εργασία-εντολή (ο όρος που χρησιμοποιείται στο OpenCL είναι *command*, "εντολή") μπορεί να περιμένει την ολοκλήρωση του *event*, η οποία ταυτίζεται με την ολοκλήρωση της σχετικής εντολής, μέσω του ορίσματος `event_wait_list`. Το συγκεκριμένο όρισμα υπάρχει στις περισσότερες κλήσεις `clEnqueue<>` που μας ενδιαφέρουν στην παρούσα εργασία και δίνει τη δυνατότητα στον προγραμματιστή να ορίσει ένα σύνολο (ως πίνακα: `cl_event*`) από *events* τα οποία θα πρέπει να περιμένει η εντολή που τοποθετείται στην ουρά. Έτσι, μπορεί μια εργασία να περιμένει άλλες που βρίσκονται σε διαφορετικές *in-order* ουρές, αλλά και να οριστεί συγχρονισμός, προκειμένου να τηρηθούν τυχόν εξαρτήσεις, μεταξύ εργασιών που βρίσκονται σε μία ή περισσότερες *out-of-order* ουρές.

Πιο συγκεκριμένα, η CPU μπορεί να περιμένει την ολοκλήρωση ενός συνόλου από γεγονότα με την `clWaitForEvents`. Αντίστοιχα, μια ουρά μπορεί να περιμένει ένα σύνολο γεγονότων με την `clEnqueueWaitForEvents`. Αυτό σημαίνει ότι οι εντολές που θα τοποθετηθούν στην ουρά μετά από αυτή την εντολή αναμονής δεν μπορούν να ξεκινήσουν την εκτέλεση προτού να μεταβούν όλα τα *events* που ορίζει η `clEnqueueWaitForEvents` στην κατάσταση `CL_COMPLETE`. Ο συγκεκριμένος μηχανισμός είναι χρήσιμος μόνο μεταξύ *in-order* ουρών, καθώς βασίζεται στη σειριακή δρομολόγηση των εργασιών στην ουρά που εκτελεί την αναμονή.

Επομένως, η αναμονή για την ολοκλήρωση κάποιων εργασιών που βρίσκονται σε μία *in-order* ουρά μπορεί να γίνει είτε περιμένοντας στο σύνολο των *events* τους, είτε περιμένοντας μόνο το τελευταίο *event*, δηλαδή αυτό που σχετίζεται με την εντολή που προστέθηκε τελευταία στην ουρά. Εδώ προστίθεται και ο μηχανισμός των *markers*, αφού υπάρχει η επιλογή για τοποθέτηση εντολής *marker* (με την κλήση `clEnqueueMarkerWithWaitList`), η οποία σηματοδοτεί την ολοκλήρωση των προηγούμενων εργασιών. Έτσι, μπορεί μια άλλη ουρά να περιμένει απλώς το *event* της εντολής *marker* και, με τον τρόπο αυτό, να γνωρίζει πως ολοκληρώθηκαν οι προηγούμενες εργασίες της πρώτης ουράς.

Ωστόσο, η λειτουργικότητα της εντολής *marker* επεκτείνεται και στις *out-of-order* ουρές, όπου και εκεί σηματοδοτεί την ολοκλήρωση των προηγούμενων εργασιών, οι οποίες, βέβαια, εκτελέστηκαν εκτός σειράς. Παρόμοια δυνατότητα δίνει και η τοποθέτηση *barrier* (με την κλήση `clEnqueueBarrierWithWaitList`), το οποίο εξασφαλίζει την ολοκλήρωση των προηγούμενων εντολών (ή όσον προσδιορίζονται στη λίστα από *events* που δέχεται ως όρισμα), αλλά και δεν επιτρέπει στις επόμενες εντολές να ξεκινήσουν την εκτέλεση. Ουσιαστικά, δίνει την επιλογή να γίνουν με τη σειρά δύο τμήματα εντολών που βρίσκονται σε *out-of-order* ουρά (Σχήμα 1.15). [Khronos, 2023]



Σχήμα 1.15: OpenCL Barrier

POSIX Threads

Το πρότυπο των νημάτων POSIX, που συχνά απαντάται ως Pthreads, περιγράφει μια διεπαφή για τον χειρισμό νημάτων. Τα νήματα είναι ροές εργασίας που δρουν στο ίδιο πρόγραμμα (στην ίδια διεργασία) και έχουν πρόσβαση στις ίδιες εντολές και καθολικά δεδομένα, αλλά κάθε στιγμή καθένα νήμα μπορεί να βρίσκεται σε διαφορετικό σημείο του προγράμματος. Αυτό σημαίνει ότι κάθε νήμα έχει δικό του μετρητή προγράμματος (*program counter*), δικούς του καταχωρητές και ξεχωριστή στοίβα, ώστε να αποθηκεύει τα τοπικά του δεδομένα. Προσδιορίζονται με τέτοιο τρόπο ώστε να δίνουν τη δυνατότητα για παράλληλη εκτέλεση στο υλικό. [Nichols et al., 1996]

Τα νήματα ακολουθούν το μοτίβο *fork-join* που περιγράφηκε στην ενότητα 1.4.1. Η διακλάδωση γίνεται κατά τη δημιουργία του νήματος με την κλήση `pthread_create`, στην οποία δίνεται η συνάρτηση που θα εκτελέσει το νέο νήμα, όπως και τα ορίσματα που χρειάζεται η εν λόγω συνάρτηση. Το νήμα που κάλεσε την `pthread_create`, που μπορεί να είναι το βασικό νήμα του προγράμματος (*main thread*) ή κάποιο που δημιουργήθηκε στη συνέχεια, μπορεί να συνεχίσει στις εντολές που ακολουθούν και να εκτελεστεί παράλληλα με το νέο νήμα. Η περιγραφή σε αυτό το σημείο δείχνει ότι το νήμα που ξεκινάει να εκτελεί ένα πρόγραμμα μπορεί να δημιουργήσει νέα νήματα, τα οποία με τη σειρά τους είναι ικανά να δημιουργήσουν άλλα, υλοποιώντας όσες διακλαδώσεις χρειάζονται για το παράλληλο, πλέον, πρόγραμμα.

Κάθε νήμα, μόλις δημιουργηθεί, ξεκινάει να εκτελεί τη συνάρτηση που προσδιορίζεται στην `pthread_create`. Το νήμα σταματάει την εκτέλεσή του είτε όταν ολοκληρώσει τη συνάρτηση που του ανατέθηκε, είτε καλώντας την `pthread_exit`, είτε όταν κάποιο άλλο νήμα ζητήσει τον τερματισμό του με την `pthread_cancel`. Το δεύτερο σκέλος του μοτίβου *fork-join* πραγματοποιείται με την κλήση `pthread_join`, η οποία μπλοκάρει το νήμα που την καλεί μέχρι να τερματίσει το νήμα που προσδιορίζεται στην `pthread_join` (για τον προσδιορισμό χρησιμοποιείται το αναγνωριστικό νήματος, *thread ID*). Η κλήση `pthread_join` αποτελεί μηχανισμό συγχρονισμού μεταξύ των νημάτων και ολοκληρώνει την εργασιο-κεντρική (task centric) προσέγγιση των Pthreads. Αυτό γίνεται σαφές, καθώς με τη δημιουργία του νήματος του ανατίθεται μια εργασία (συνάρτηση και δεδομένα επεξεργασίας), και τα υπόλοιπα νήματα μπορούν να συγχρονιστούν με το πέρας αυτής της εργασίας. [Nichols et al., 1996]

Επιπλέον, η διεπαφή των Pthreads παρέχει και μηχανισμούς για λεπτότερο συγχρονισμό των νημάτων, πέρα από τα σημεία εκκίνησης και ολοκλήρωσης των εργασιών τους. Ο πρώτος είναι ο μηχανισμός αμοιβαίου αποκλεισμού (*mutual exclusion*), που απαντάται ως *mutex*. Ένα *mutex* είναι ένα αντικείμενο που μπορεί να βρίσκεται σε δύο καταστάσεις: κλειδωμένο (*locked*) και ξεκλείδωτο (*unlocked*). Για αυτό υπάρχουν και οι αντίστοιχες λειτουργίες που μπορούν να εφαρμοστούν επάνω του: `pthread_mutex_lock`, `pthread_mutex_unlock`.

Ένα τέτοιο αντικείμενο μπορεί να χρησιμοποιηθεί σε περιπτώσεις όπου είναι αναγκαίο ένα τμήμα του κώδικα να εκτελείται από το πολύ ένα νήμα κάθε χρονική στιγμή. Αυτού του είδους τα τμήματα ονομάζονται "κρίσιμα τμήματα" (*critical sections*) και είναι πιθανό να προκύψουν για διάφορους λόγους στην λογική ενός πολυνηματικού προγράμματος. Ο πιο συνηθισμένος λόγος αφορά περιπτώσεις στις οποίες τα νήματα επεξεργάζονται κοινά δεδομένα και χρειάζεται να επιβληθεί σειριακή πρόσβαση σε αυτά, ώστε ένα νήμα να διαβάζει, να επεξεργάζεται και να αποθηκεύει στην κοινή μνήμη κάποιο δεδομένο, προτού κάποιο άλλο νήμα ξεκινήσει την δική του επεξεργασία στο ίδιο δεδομένο (ίδια θέση μνήμης). Η ταυτόχρονη δράση δύο ή

περισσότερων νημάτων που εκτελούν τις πράξεις: "ανάγνωση, επεξεργασία, αποθήκευση" στην ίδια θέση μνήμης, οδηγεί σε ασυνέπεια των δεδομένων. Με άλλα λόγια το αποτέλεσμα του παράλληλου υπολογισμού δεν ταυτίζεται με το αποτέλεσμα του σειριακού υπολογισμού, κάτι που είναι ανεπιθύμητο.

Ο μηχανισμός του *mutex* μπορεί να διασφαλίσει τη σωστή (σειριακή) εκτέλεση ενός κρίσιμου τμήματος. Αυτό που χρειάζεται να κάνει ο προγραμματιστής είναι να εισάγει την εντολή απόκτησης του *mutex*: `pthread_mutex_lock`, ακριβώς πριν το κρίσιμο τμήμα και την εντολή ελευθέρωσής του: `pthread_mutex_unlock`, αμέσως μετά. Με τον τρόπο αυτό, το πρώτο νήμα που επιχειρήσει να εισέλθει στο κρίσιμο τμήμα θα διεκδικήσει και θα "αποκτήσει" το *mutex*, αφού θα το βρει ελεύθερο (*unlocked*). Αν κάποιο άλλο νήμα προσπαθήσει να προχωρήσει και αυτό στο κρίσιμο τμήμα θα διεκδικήσει το *mutex*, αλλά θα μπλοκάρει, αφού θα το βρει κλειδωμένο. Θα πρέπει να περιμένει την ολοκλήρωση του κρίσιμου τμήματος από το νήμα που κατέχει το *mutex* και την αποδέσμευσή του, ώστε να είναι σε θέση να το κλειδώσει εκείνο και να συνεχίσει. Εδώ είναι σημαντικό να αναφέρουμε ότι είναι πιθανό να περιμένουν πολλά νήματα ένα *mutex*, όμως μόνο ένα θα το αποκτήσει.

Υπάρχουν, ωστόσο, περιπτώσεις στις οποίες ένα *mutex* δεν είναι αρκετό και χρησιμοποιούνται περισσότερα. Σε μια τέτοια συνθήκη είναι κρίσιμο να αποφευχθεί το πρόβλημα του *deadlock*, όπου ένα νήμα αποκτά το `mutex_A` και περιμένει για το `mutex_B`, ενώ ένα άλλο νήμα έχει το `mutex_B` και περιμένει το `mutex_A`. Αυτό σημαίνει πως κανένα από τα δύο νήματα δεν μπορεί να προχωρήσει και το πρόγραμμα αδυνατεί να συνεχίσει τον υπολογισμό (*stall*).

Μια ακόμα επιλογή είναι να χρησιμοποιηθεί *spinlock* αντί για *mutex*. Ένα *spinlock* έχει την ίδια λειτουργία με ένα *mutex* με την εξής διαφορά: το νήμα που επιχειρεί να αποκτήσει ένα - ήδη κλειδωμένο - *mutex*, αναστέλλει την εκτέλεσή του και απομακρύνεται προσωρινά από τον χρονοδρομολογητή του επεξεργαστή. Από την άλλη, το νήμα που συναντά ένα κλειδωμένο *spinlock* θα συνεχίσει να "περιστρέφεται" (*spin*), δηλαδή να ελέγχει επίμονα την κατάσταση του κλειδώματος. Η διαδικασία θα διακοπεί είτε επειδή το νήμα που κρατούσε το *spinlock* το ελευθερώσει, είτε επειδή θα αντικατασταθεί (*context switch*) το νήμα που περιμένει επίμονα, όταν έρθει μια διακοπή χρονιστή (*timer interrupt*). Επομένως, η διαφορά τους δεν αφορά τη λειτουργικότητα, αλλά την επίδοση του προγράμματος. Όμως, δεν είναι σαφές εκ των προτέρων το ποια επιλογή οδηγεί σε υψηλότερη επίδοση. Αυτό εξαρτάται από την αρχιτεκτονική του συστήματος (αριθμός πυρήνων), αλλά και από τη δομή του προγράμματος. Συνήθως, τα *spinlocks* είναι προτιμότερα σε περιπτώσεις όπου κάθε νήμα κρατά για λίγο χρόνο το κλείδωμα, και υπάρχουν αρκετοί πυρήνες, ώστε να είναι πιο πιθανό να εκτελείται παράλληλα το νήμα που κρατάει το κλείδωμα. Ουσιαστικά, το *spinlock* έχει χειρότερη επίδοση αν το χρόνος που ένα νήμα περιμένει επίμονα (*busy wait*) είναι μεγαλύτερος από τον χρόνο που απαιτείται για να γίνει η αντικατάσταση νήματος στον πυρήνα. [Boguslavsky et al., 1994] [Butenhof, 1997]

Επιπλέον, υπάρχει η δυνατότητα για επικοινωνία και ειδοποίηση μεταξύ των νημάτων. Το σχήμα συγχρονισμού που παρέχει αυτή τη λειτουργία ονομάζεται "μεταβλητή συνθήκης" (*condition variable*). Με αυτόν τον τρόπο ένα νήμα μπορεί να ελέγξει μια συνθήκη και να αναστείλει την εκτέλεσή του, περιμένοντας να γίνει αληθής η εν λόγω συνθήκη. Από την άλλη πλευρά, κάποιο νήμα, που ενημερώνει την μεταβλητή που σχετίζεται με τη συνθήκη, μπορεί

να ειδοποιήσει το πρώτο νήμα που έχει μπλοκάρει για να συνεχίσει και εκείνο τη λειτουργία του. Υπό αυτό το πρίσμα, οι μεταβλητές συνθήκης επιτρέπουν σε ένα ή περισσότερα νήματα να ορίσουν πως περιμένουν ένα σήμα για να συνεχίσουν, ενώ άλλα νήματα αναλαμβάνουν να στείλουν τις ειδοποιήσεις την κατάληλη στιγμή. Οι σχετικές κλήσεις της βιβλιοθήκης των pthreads είναι οι: `pthread_cond_wait` και `pthread_cond_signal`. Κάθε χρήση μεταβλητής συνθήκης πρέπει να συνδέεται και με ένα *mutex*, όπως φαίνεται στη συνέχεια.

```
void update_and_signal() {
    pthread_mutex_lock(&m);
        shared_variable = target;
        pthread_cond_signal(&c);
    pthread_mutex_unlock(&m);
}

void wait_on_condition() {
    pthread_mutex_lock(&m);
        while (shared_variable != target)
            pthread_cond_wait(&c, &m);
    pthread_mutex_unlock(&m);
}
```

Σε αυτό το κομμάτι κώδικα φαίνονται οι ενέργειες που πρέπει να κάνει ένα νήμα προκειμένου να τροποποιήσει τη μεταβλητή συνθήκης και να στείλει το σήμα (`update_and_signal()`), καθώς και ο τρόπος με τον οποίο κάποιο άλλο νήμα αναμένει την αλλαγή της συνθήκης (`wait_on_condition()`). Ο ρόλος του *mutex* είναι να εξασφαλίσει ότι το πολύ ένα νήμα θα έχει πρόσβαση στη μεταβλητή κάθε στιγμή. Επίσης, βλέπει κανείς ότι οι κλήσεις `pthread_cond_signal` και `pthread_cond_wait` γίνονται όταν το νήμα έχει αποκτήσει-κλειδώσει το *mutex*. Αυτό δεν εγείρει ανησυχία στην περίπτωση της `pthread_cond_signal`, αφού θα ειδοποιήσει για την αλλαγή που συνέβη και, στην επόμενη εντολή, το νήμα θα ελευθερώσει το *mutex*. Το πρόβλημα θα μπορούσε να βρίσκεται στην `pthread_cond_wait`, η οποία έχοντας κλειδωμένο το *mutex* προκαλεί την αναστολή της εκτέλεσης του νήματος. Κάτι τέτοιο θα σήμαινε ότι κανένα άλλο νήμα δεν μπορεί να αποκτήσει το *mutex* και να ενημερώσει την μεταβλητή συνθήκης, δηλαδή το πρόγραμμα φτάνει σε *deadlock*. Όμως, αυτό δεν συμβαίνει διότι η `pthread_cond_wait` σχεδιάζεται με τέτοιο τρόπο ώστε να εκτελεί τρεις πράξεις:

1. ελευθέρωση του *mutex*
2. αναστολή της εκτέλεσης του νήματος
3. επανάκτηση του *mutex*
4. επιστροφή από την κλήση

Οι πρώτες δύο πράξεις εκτελούνται ατομικά στην αρχή της `pthread_cond_wait`, ενώ οι επόμενες δύο πραγματοποιούνται αφού έρθει το σήμα αφύπνισης του νήματος. Με αυτόν τον

σχεδιασμό το νήμα κατέχει το *mutex* και πριν και μετά την `pthread_cond_wait`, αλλά όχι για όσο χρόνο βρίσκεται σε αναστολή.

Το τελευταίο σημείο που αξίζει να σχολιαστεί είναι ο βρόγχος `while` που εμφανίζεται στον παραπάνω κώδικα. Αυτό χρειάζεται καθώς υπάρχει η πιθανότητα να ειδοποιηθεί το νήμα που περιμένει, η `pthread_cond_wait` να διεκδικήσει το *mutex*, αλλά κάποιο άλλο νήμα να το αποκτήσει πρώτο και να τροποποιήσει τη μεταβλητή. Έτσι, όταν αποκτήσει η `pthread_cond_wait` το *mutex* δεν θα ισχύει η συνθήκη. Για αυτό χρειάζεται να ελεγχθεί εκ νέου και, αν διαπιστωθεί ότι δεν ισχύει, να ανασταλεί, και πάλι, η εκτέλεση του νήματος. [Arpaci-Dusseau and Arpaci-Dusseau, 2015]

Ο τελευταίος μηχανισμός συγχρονισμού νημάτων που θα περιγράψουμε εδώ είναι εκείνος των "σημαφόρων" (*semaphore*). Μάλιστα, πρόκειται για μηχανισμό που είναι γενικότερος των *mutexes* και των μεταβλητών συνθήκης. Οι σημαφόροι είναι αντικείμενα που αποθηκεύουν μια ακέραιη τιμή, η οποία συχνά αρχικοποιείται στην τιμή 0 ή 1. Μπορούν να πραγματοποιηθούν δύο πράξεις σε έναν σημαφόρο:

- `sem_wait()`: μειώνει (ατομικά) την τιμή του σημαφόρου κατά 1 και αναστέλλει τη λειτουργία του καλούντος νήματος, αν η νέα τιμή είναι αρνητική. (Απαντάται και ως "πράξη P" για ιστορικούς λόγους)

- `sem_post()`: αυξάνει (ατομικά) την τιμή του σημαφόρου κατά 1 και αφυπνίζει κάποιο νήμα που περιμένει, αν η προηγούμενη τιμή ήταν αρνητική. (Απαντάται και ως "πράξη V")

Έτσι, όταν η τιμή του σημαφόρου είναι θετική, θεωρείται ότι είναι "ξεκλείδωτος", ενώ όταν είναι 0 ή αρνητική θεωρείται ότι είναι "κλειδωμένος", αφού όποιο νήμα επιχειρήσει `sem_wait()` θα μπλοκάρει. Επίσης, η αρνητική τιμή δηλώνει και τον αριθμό των νημάτων που περιμένουν στον συγκεκριμένο σημαφόρο, ωστόσο αυτή η τιμή συχνά δεν είναι διαθέσιμη στον προγραμματιστή [Unix, 2006].

Όπως ήδη αναφέρθηκε, ένας σημαφόρος μπορεί να χρησιμοποιηθεί σαν *mutex*. Αυτό γίνεται αν αρχικοποιηθεί στην τιμή 1 και κάθε νήμα χρησιμοποιεί μια `sem_wait()` πριν εισέλθει στο κρίσιμο τμήμα και μια `sem_post()` αφού το ολοκληρώσει. Με αυτόν τον τρόπο, κάθε φορά μόνο ένα νήμα (η τιμή αρχικοποίησης του σημαφόρου) θα είναι εντός του κρίσιμου τμήματος, και κάποιο επόμενο νήμα θα αφυπνίζεται μόλις ο σημαφόρος ελευθερωθεί. Επίσης, μπορεί να χρησιμοποιηθεί και για ανταλλαγή σημάτων μεταξύ των νημάτων, όπως μια μεταβλητή συνθήκης. Σε αυτή την περίπτωση θα πρέπει να αρχικοποιηθεί στην τιμή 0, ώστε το νήμα που χρειάζεται να περιμένει το σήμα να εκτελέσει μια `sem_wait()` (πλέον η τιμή του σημαφόρου θα είναι -1) και να αφήσει τον επεξεργαστή. Την κατάλληλη στιγμή μπορεί ένα άλλο νήμα να καλέσει την `sem_post()` (θέτοντας τιμή του σημαφόρου σε 0) και να ειδοποιήσει το πρώτο νήμα. Τέλος, οι σημαφόροι επιτρέπουν και πιο σύνθετα σχήματα συγχρονισμού, όπου ένα τμήμα του κώδικα πρέπει να εκτελείται από το πολύ n νήματα ταυτόχρονα. Για αυτή την ανάγκη, ο σημαφόρος θα αρχικοποιηθεί στην τιμή n και μόνο τα πρώτα n νήματα που εκτελέσουν `sem_wait()` θα προχωρήσουν στο συγκεκριμένο τμήμα του προγράμματος.[Arpaci-Dusseau and Arpaci-Dusseau, 2015]

HIP

Το HIP αποτελεί την προγραμματιστική διεπαφή που παρέχει η AMD. Αναπτύχθηκε με στόχο την παραγωγή προγραμμάτων που μπορούν να μεταγλωττιστούν και να εκτελεστούν τόσο σε GPUs που υποστηρίζουν CUDA (Nvidia), όσο και σε συσκευές που υποστηρίζουν ROCm (AMD). Για αυτόν τον λόγο, διαθέτει σχεδόν όλες τις κλήσεις που υπάρχουν στο CUDA (προς το παρόν δεν υπάρχουν hip κλήσεις για κάποιες δευτερεύουσες λειτουργίες του CUDA) και, μάλιστα, σε μία προς μία αντιστοιχία.

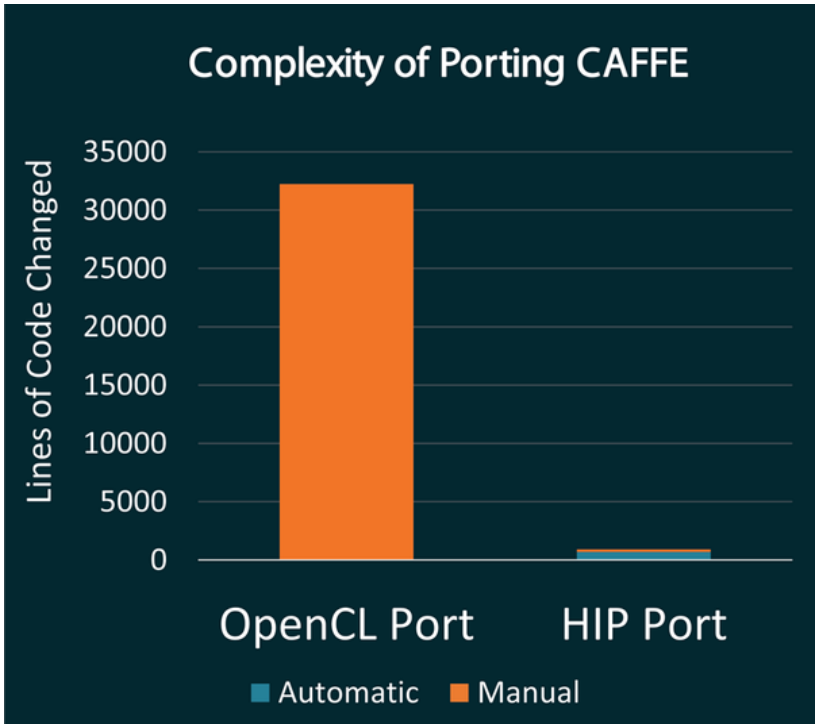
Συγκεκριμένα, η διαχείριση της μνήμης γίνεται με τις κλήσεις `hipMalloc` και `hipHostMalloc`, που πραγματοποιούν δέσμευση μνήμης στη συσκευή και *pinned* μνήμης στη CPU, ακριβώς όπως οι `cudaMalloc` `cudaMallocHost`. Με τον ίδιο τρόπο γίνονται και οι ασύγχρονες μεταφορές δεδομένων με την `hipMemcpyAsync` (στη θέση της `cudaMemcpyAsync`). Πανομοιότυπες είναι και οι κλήσεις που αφορούν τη διαχείριση των *streams*, των *events*, και του συγχρονισμού, όπως φαίνεται στον Πίνακα 2.2. [Bauman et al., 2019]

CUDA	HIP	Λειτουργία
`MyKernel<<<>>>`	`hipLaunchKernel`	Εκκίνηση πυρήνα (*kernel launch*)
`cudaMalloc`	`hipMalloc`	Δέσμευση μνήμης στη συσκευή
`cudaMallocHost`	`hipHostMalloc`	Δέσμευση μνήμης ”κλειδωμένων σελίδων” (*pinned*) στον *host*
`cudaMemcpyAsync`	`hipMemcpyAsync`	Μεταφορά δεδομένων
`cudaLaunchHostFunc`	`hipLaunchHostFunc`	Τοποθέτηση σε *stream* συνάρτησης που θα εκτελεστεί στον *host*
`cudaDeviceSynchronize`	`hipDeviceSynchronize`	Συγχρονισμός *host - device*
`cudaStreamSynchronize`	`hipStreamSynchronize`	Συγχρονισμός *host - stream*
`cudaEventRecord`	`hipEventRecord`	Τοποθέτηση *event* σε *stream*
`cudaEventSynchronize`	`hipEventSynchronize`	Συγχρονισμός με *event*
`cudaStreamWaitEvent`	`hipStreamWaitEvent`	Συγχρονισμός *stream* με *event*

Πίνακας 1.2: CUDA - HIP [AMD, 2023a]

Επιπλέον, προσφέρεται η βιβλιοθήκη `hipBLAS`, που αντικαθιστά την `cuBLAS`, όπως και άλλες βιβλιοθήκες του CUDA που, όμως, δεν θα μας απασχολήσουν στην παρούσα εργασία. [Bauman et al., 2019]

Γίνεται εμφανές ότι ο υψηλός βαθμός ομοιότητας επιλέχθηκε ώστε να είναι ευκολότερη η προγραμματιστική μετάβαση από το CUDA στο HIP. Σε αυτή την κατεύθυνση βοηθάει και το εργαλείο *hipify*, που παρέχει τη δυνατότητα για ”μετάφραση” πηγαίου κώδικα. Στα εργαλεία του *hipify* υπάρχει και η δυνατότητα σάρωσης (*hipexamine*) του CUDA κώδικα, ώστε να γνωρίζει, εκ των προτέρων, ο προγραμματιστής ποιες κλήσεις του CUDA μπορούν να αντικατασταθούν αυτόματα σε κλήσεις HIP, ποιες χρειάζονται την παρέμβαση του προγραμματιστή και ποιες δεν υποστηρίζονται στη διεπαφή του HIP. Με αυτόν τον τρόπο επιταχύνεται σημαντικά η μεταφορά υπάρχοντος κώδικα σε AMD συστήματα, όπως στην περίπτωση του συστήματος βαθιάς μηχανικής μάθησης ”Caffe” (Σχήμα 1.16). [AMD, 2023b]



Σχήμα 1.16: Η αυτοματοποιημένη μεταφορά του συστήματος ”Caffe” σε HIP [Bier, 2016]

1.5 CoCoPeLia

Το CoCoPeLia (Communication-Computation Overlap Prediction for Efficient Linear Algebra on GPUs) [Anastasiadis et al., 2021] σχεδιάστηκε στοχεύοντας στην βελτίωση των υπαρχουσών βιβλιοθηκών για ανάθεση υπολογισμών BLAS σε GPUs. Αυτές οι βιβλιοθήκες (cuBLASXt, BLASX, XKBLAS) παρουσιάστηκαν σε προηγούμενες ενότητες και φέρουν δύο βασικά χαρακτηριστικά:

1. διαχωρίζουν τον συνολικό υπολογισμό σε μικρότερους υποπυρήνες

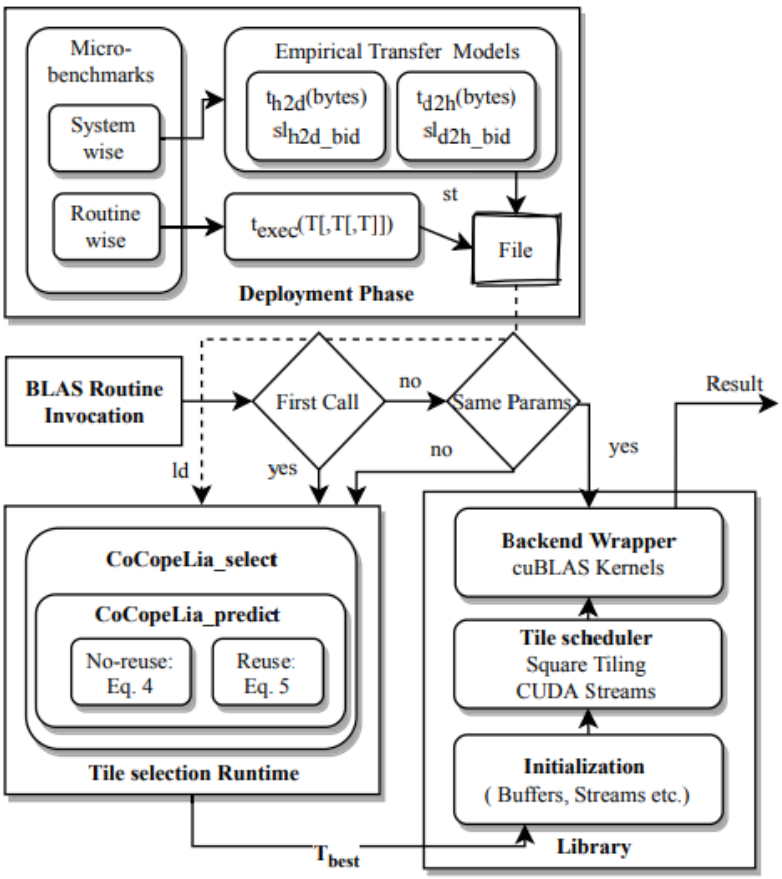
2. επικαλύπτουν τις μεταφορές δεδομένων με υπολογισμό στην εκάστοτε GPU (παραλληλισμός τριών επιπέδων, 3-way concurrency)

Ο διαχωρισμός σε *tiles*, στον οποίο αναφερθήκαμε στην ενότητα 1.2.1, επηρεάζει σημαντικά την επίδοση ενός τέτοιου συστήματος. Αυτό συμβαίνει επειδή το μέγεθος των *tiles* που επιλέγεται έχει άμεσο αντίκτυπο στον όγκο των δεδομένων που μεταφέρονται σε κάθε φάση, στον όγκο των πυρήνων και στο πλήθος των συνολικών μεταφορών που θα πρέπει να πραγματοποιηθούν μεταξύ CPU και GPU. Ωστόσο, ο εντοπισμός του κατάλληλου μεγέθους *tile* εξαρτάται από πολλούς παράγοντες, όπως το συνολικό μέγεθος του προβλήματος και η διαθέσιμη αρχιτεκτονική.

Η προσέγγιση της cuBLASXt στο πρόβλημα είναι να αφήσει στον προγραμματιστή την επιλογή του μεγέθους *tile*. Αυτό οδηγεί στην ανάγκη πραγματοποίησης μιας σειράς δοκιμών και μετρήσεων από τον προγραμματιστή, ώστε να διαπιστωθεί η βέλτιστη τιμή της παραμέτρου για το συγκεκριμένο πρόβλημα και το διαθέσιμο υλικό. Από την άλλη, η BLASX θέτει εσωτερικά και στατικά το μέγεθος του *tile* που πετυχαίνει την καλύτερη επίδοση κατά μέσο όρο, χωρίς να το προσαρμόζει στην τρέχουσα κλήση BLAS. Όσον αφορά την XKBLAS, εκεί ακολουθείται ένας συνδυασμός των δύο προηγούμενων πρακτικών, αφού υπάρχει ένα προεπιλεγμένο μέγεθος που, όμως, μπορεί να μεταβληθεί από τον χρήστη.

Το CoCoPeLia, αξιοποιώντας την αύξηση της επίδοσης που μπορεί να φέρει η κατάλληλη επιλογή μεγέθους *tile*, υλοποιεί πρόβλεψη βέλτιστου μεγέθους και πραγματοποιεί δυναμικά την επιλογή του κατά τον χρόνο εκτέλεσης του προγράμματος. Προκειμένου αυτή η πρόβλεψη να είναι προσαρμοσμένη στην υποκείμενη αρχιτεκτονική, εκτελείται ένα σύνολο διαγνωστικών προγραμμάτων (*micro-benchmarks*) όταν εγκαθίσταται το CoCoPeLia σε νέο σύστημα. Αυτά τα προγράμματα χρειάζεται να εκτελεστούν μόνο μια φορά, ώστε να συγκεντρωθούν μετρήσεις που αφορούν τους χρόνους μεταφοράς δεδομένων, όπως και κάποιους χρόνους εκτέλεσης ρουτινών BLAS στη συγκεκριμένη GPU. Αυτές οι μετρήσεις αποθηκεύονται και χρησιμοποιούνται, στη συνέχεια, όποτε ζητείται ένας υπολογισμός BLAS.

Όταν καλείται μια πράξη BLAS, το υποσύστημα πρόβλεψης μεγέθους *tile* λαμβάνει υπόψιν τις παραμέτρους αυτής της κλήσης, αλλά και τις μετρήσεις που συγκεντρώθηκαν κατά την εγκατάσταση, για να υπολογίσει το βέλτιστο *tile*. Έπειτα αυτό χρησιμοποιείται για την εκτέλεση της πράξης. Φυσικά, αν η συγκεκριμένη πράξη BLAS, με τα ίδια ορίσματα, ζητηθεί ξανά θα χρησιμοποιηθεί το προϋπολογισμένο *tile*. Στο τέλος, χρησιμοποιούνται CUDA *streams* και η βιβλιοθήκη cublas για να αποσταλούν οι εργασίες στη συσκευή. Η παραπάνω περιγραφή συνοψίζεται στο Σχήμα 1.17.



Σχήμα 1.17: Το σύστημα CoCoPeLia [Anastasiadis et al., 2021]

1.6 PARALiA

Η επέκταση του CoCoPeLia σε ετερογενή συστήματα πολλών GPUs έρχεται με το σύστημα PARALiA (Performance Aware Runtime for Auto-tuning Linear Algebra on heterogeneous systems) [Anastasiadis et al., 2023]. Αρχικά προστίθεται το υποσύστημα LinkMap, το οποίο αναπαριστά όλους τους διαθέσιμους υπολογιστικούς κόμβους και τις μεταξύ τους συνδέσεις. Αυτό γίνεται για να εξαχθούν τα βέλτιστα μονοπάτια μεταφοράς δεδομένων (η διαδρομή με τον μικρότερο χρόνο μεταφοράς ανάμεσα σε οποιουσδήποτε δύο κόμβους). Η ανάλυση της αρχιτεκτονικής, χρησιμοποιείται, στη συνέχεια, από το σύστημα διαμόρφωσης της βάσης δεδομένων. Το συγκεκριμένο υποσύστημα πραγματοποιεί ελέγχους επίδοσης κάποιων πυρήνων BLAS σε κάθε συσκευή και συλλέγει μετρήσεις για την ταχύτητα των συνδέσεων μεταξύ κόμβων, όπως προκύπτουν από το LinkMap. Αυτές οι μετρήσεις χρειάζεται να γίνουν μόνο μια φορά σε κάθε σύστημα που πρόκειται να χρησιμοποιήσει το PARALiA, καθώς αποθηκεύονται σε μια τοπική βάση δεδομένων για μελλοντική χρήση.

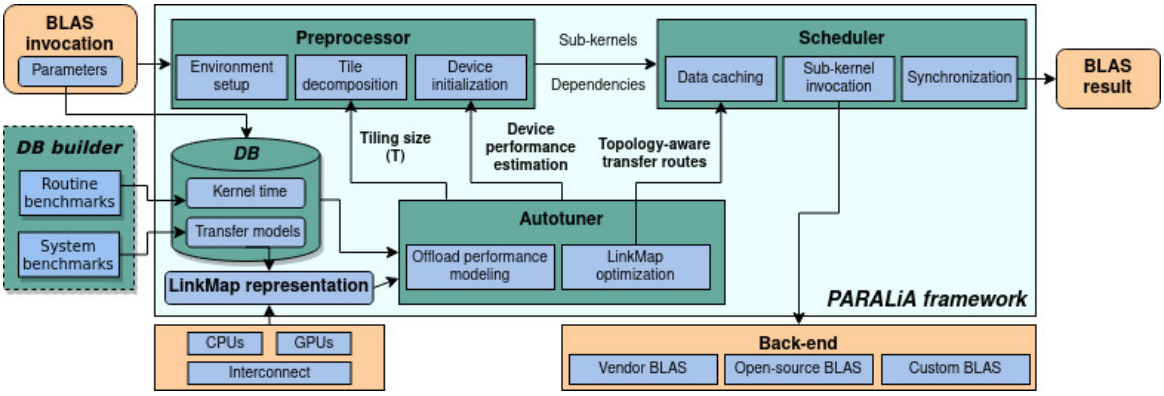
Κατά την κλήση κάποιας πράξης BLAS, το υποσύστημα του Autotuner χρησιμοποιεί τα δεδομένα της βάσης και τις συνδέσεις του LinkMap, σε συνδυασμό με τις παραμέτρους της κλήσης, για να υπολογίσει:

- Το βέλτιστο μέγεθος *tile* (αντίστοιχα με το CoCoPeLia)
- Τους συντελεστές με τους οποίους πρέπει να κατανεμηθεί η εργασία στις επιμέρους

συσκευές

- Τις συντομότερες διαδρομές μεταφοράς δεδομένων μεταξύ δύο επεξεργαστικών κόμβων

Έπειτα, αυτές οι πληροφορίες μεταφέρονται στον Preprocessor του συστήματος για να σχηματίσει τους υποπυρήνες και τις μεταξύ τους εξαρτήσεις. Από αυτό το σημείο αναλαμβάνει ο Scheduler, ο οποίος φροντίζει για την μεταφορά των δεδομένων, την εκτέλεση του κάθε πυρήνα στην κατάλληλη συσκευή και τον μεταξύ τους συγχρονισμό. Ταυτόχρονα, ο Scheduler υλοποιεί και *caching* των δεδομένων στις μνήμες των συσκευών από τις οποίες διέρχονται. Επισημαίνεται, ότι και εδώ χρησιμοποιείται η διεπαφή του CUDA, τα *streams*, τα *events* και η βιβλιοθήκη *cublas* για τη διοχέτευση των εργασιών στις GPUs. Όσα περιγράψαμε παρουσιάζονται και στο σχήμα 1.18



Σχήμα 1.18: Το σύστημα PARALiA [Anastasiadis et al., 2023]

Chapter 2

Universal Helpers

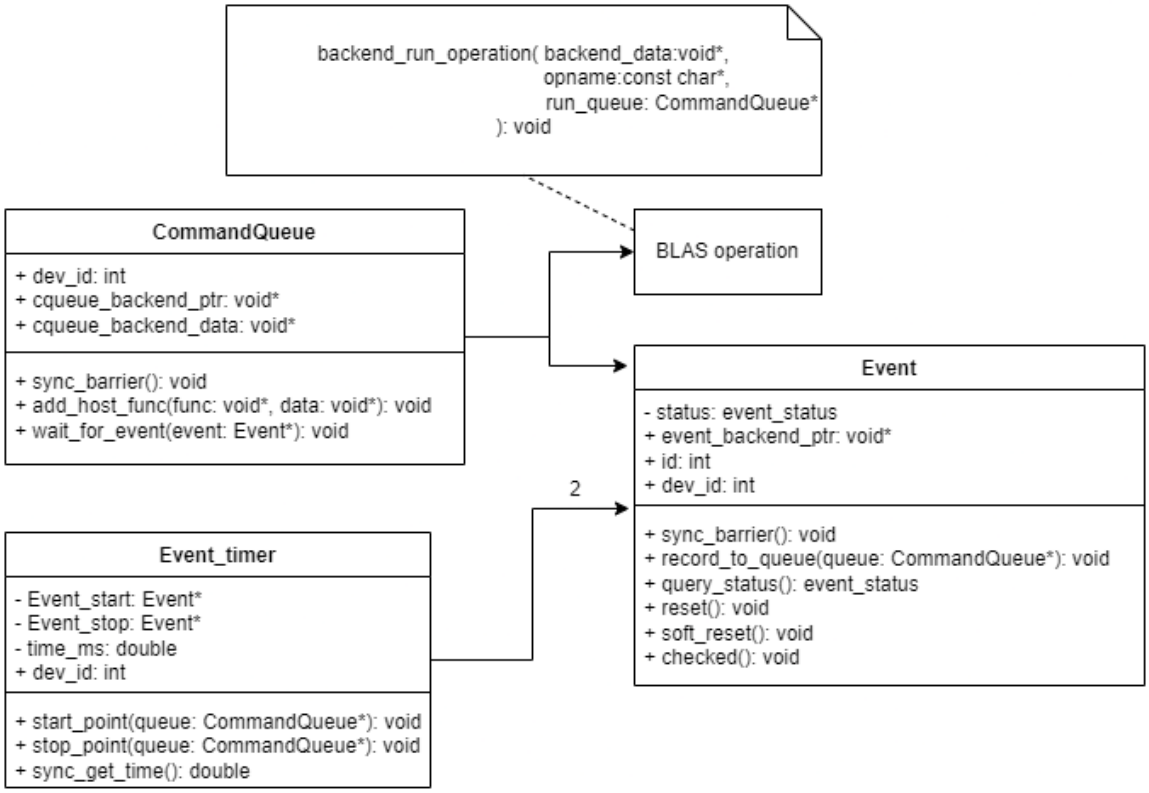
Στην παρούσα διπλωματική εργασία θα επικεντρωθούμε στη βιβλιοθήκη Universal Helpers, η οποία χρησιμοποιείται στο παρασκήνιο του PARALiA, και θα ερευνήσουμε τις δυνατότητες που προσφέρουν διαφορετικές υλοποιήσεις της. Η εν λόγω βιβλιοθήκη αναπτύχθηκε πρώτα για το σύστημα CoCoPeLia και, στη συνέχεια, ενσωματώθηκε στο PARALiA. Πρόκειται για υλοποίηση κλάσεων συσκευαστών (wrapper classes) και συναρτήσεων σε C++, ώστε να μπορεί ο προγραμματιστής να χειρίζεται τους υπολογιστικούς κόμβους (CPUs και GPUs) και να αναθέτει εργασίες σε αυτούς, χρησιμοποιώντας ένα υψηλότερο επίπεδο αφαίρεσης. Έτσι, δύναται να παρέχει ευκολία προγραμματισμού, αλλά και βελτίωση της επίδοσης, καθώς μπορεί να χειρίζεται σε χαμηλό επίπεδο και με περισσότερη λεπτομέρεια κάποια σύνθετη λειτουργία. Παρέχει τρεις βασικές κλάσεις:

- `CommandQueue`: περιλαμβάνει λειτουργίες για τοποθέτηση και εκτέλεση εργασιών σε κάποια ουρά εργασιών (αντίστοιχη των *CommandQueues* του OpenCL, ή των CUDA *streams*)
- `Event`: υλοποίηση του μηχανισμού "γεγονότων" για τον συγχρονισμό των εργασιών που βρίσκονται στις ουρές
- `Event_timer`: δίνει τη δυνατότητα για υπολογισμό του χρόνου μεταξύ δύο γεγονότων. Χρησιμοποιείται κυρίως για μετρήσεις επίδοσης του συνολικού συστήματος.

Οι βασικές λειτουργίες φαίνονται στο διάγραμμα του Σχήματος 2.1. Παρατηρείται ότι υπάρχουν αρκετά πεδία και ορίσματα συναρτήσεων που είναι τύπου `void*`. Αυτό δίνει τη δυνατότητα αλλαγής της υποκείμενης υλοποίησης της βιβλιοθήκης, χωρίς να επηρεάζεται η διεπαφή προς τον χρήστη. Επιπλέον, διατίθενται οι συναρτήσεις διαχείρισης των συσκευών και της μνήμης που παρουσιάζονται στον πίνακα 2.1. Συγκεκριμένα, η `CoCoMalloc(bytes: int, loc: int): void*` μπορεί να χρησιμοποιηθεί για τρεις σκοπούς, ανάλογα με την τιμή της παραμέτρου `loc`:

- για δέσμευση μνήμης στον host (malloc): `loc = -2`
- για δέσμευση *pinned* μνήμης στον host: `loc = -1`

- για δέσμευση μνήμης σε οποιαδήποτε συσκευή: `loc = id`, όπου id το αναγνωριστικό της συσκευής



Σχήμα 2.1: Διάγραμμα κλάσεων της βιβλιοθήκης Universal Helpers

`CoCoPeLiaGetDevice`	Επιστρέφει την τρέχουσα συσκευή
`CoCoPeLiaSelectDevice`	Ορίζει την τρέχουσα συσκευή
`CoCoPeLiaDevGetMemInfo`	Επιστρέφει την διαθέσιμη και τη συνολική μνήμη της τρέχουσας συσκευής
`CoCoSyncCheckErr`	Επιβάλλει τον συγχρονισμό της τρέχουσας συσκευής και ελέγχει για σφάλματα
`CoCoASyncCheckErr`	Ελέγχει για σφάλματα χωρίς συγχρονισμό
`CoCoEnableLinks`	Ενεργοποιεί τις συνδέσεις της ζητούμενης συσκευής με όλες τις υπόλοιπες
`CoCoMalloc`	Δεσμεύει μνήμη σε κόμβο (GPU ή CPU)
`CoCoFree`	Αποδεσμεύει μνήμη
`CoCoMemcpy`	Μεταφέρει δεδομένα μεταξύ δύο κόμβων
`CoCoMemcpyAsync`	Ασύγχρονη `CoCoMemcpy`
`CoCoMemcpy2D`	Μεταφέρει δεδομένα που είναι αποθηκευμένα σε μορφή πίνακα
`CoCoMemcpy2DAsync`	Ασύγχρονη `CoCoMemcpy2D`
`CoCoVecInit`	Αρχικοποιεί διάνυσμα στον ζητούμενο κόμβο
`CoCoParallelVecInitHost`	Αρχικοποιεί παράλληλα διάνυσμα στον host
`CoCoGetPtrLoc`	Επιστρέφει τον κόμβο στον οποίο βρίσκεται το τμήμα της μνήμης που παραπέμπει ο ζητούμενος δείκτης (*pointer*)

Πίνακας 2.1: Συναρτήσεις της Universal Helpers για διαχείριση των συσκευών και της μνήμης

2.1 CUDA backend

Στην υπάρχουσα υλοποίησή της η βιβλιοθήκη λειτουργεί με κλήσεις του CUDA και του cuBLAS API. Για παράδειγμα, οι ουρές εργασιών χρησιμοποιούν CUDA *streams* και τα *events* συγχρονισμού περιέχουν CUDA *events*.

2.1.1 CommandQueue

Η κλάση `CommandQueue` αποθηκεύει το αναγνωριστικό της συσκευής με την οποία σχετίζεται στη μεταβλητή `dev_id`, και διατηρεί το *stream* που χρησιμοποιεί για τις εργασίες της στον δείκτη `cqueue_backend_ptr` και ένα cuBLAS *handle* στον δείκτη `cqueue_backend_data`. Αυτό το *handle* χρειάζεται για την εκκίνηση πράξεων BLAS από τη βιβλιοθήκη cuBLAS. Για αυτό τον λόγο, ο κατασκευαστής της κλάσης `CommandQueue` δημιουργεί το *stream* και το *handle*, αποθηκεύοντας τα στους δείκτες που προαναφέρθηκαν, και τα "συνδέει" με την κλήση `cublasSetStream(handle, stream)`. Με αυτόν τον τρόπο, όταν γίνει κάποια κλήση σε πράξη cuBLAS με το συγκεκριμένο *handle* θα τοποθετηθεί στο αντίστοιχο *stream*.

Ο χρήστης μπορεί να πραγματοποιήσει μια BLAS πράξη με τη συνάρτηση `backend_run_operation`. Στα ορίσματα της παρέχονται:

1. τα δεδομένα της πράξης με τη μορφή `void*`
2. το όνομα της ζητούμενης πράξης. Ακολουθείται η ονοματολογία που χρησιμοποιείται ευρέως σε βιβλιοθήκες BLAS: "Dgemm", "Dgemv", "Sgemm", "Daxpy", "Ddot" κ.ο.κ.
3. η ουρά στην οποία πρέπει να τοποθετηθεί η πράξη

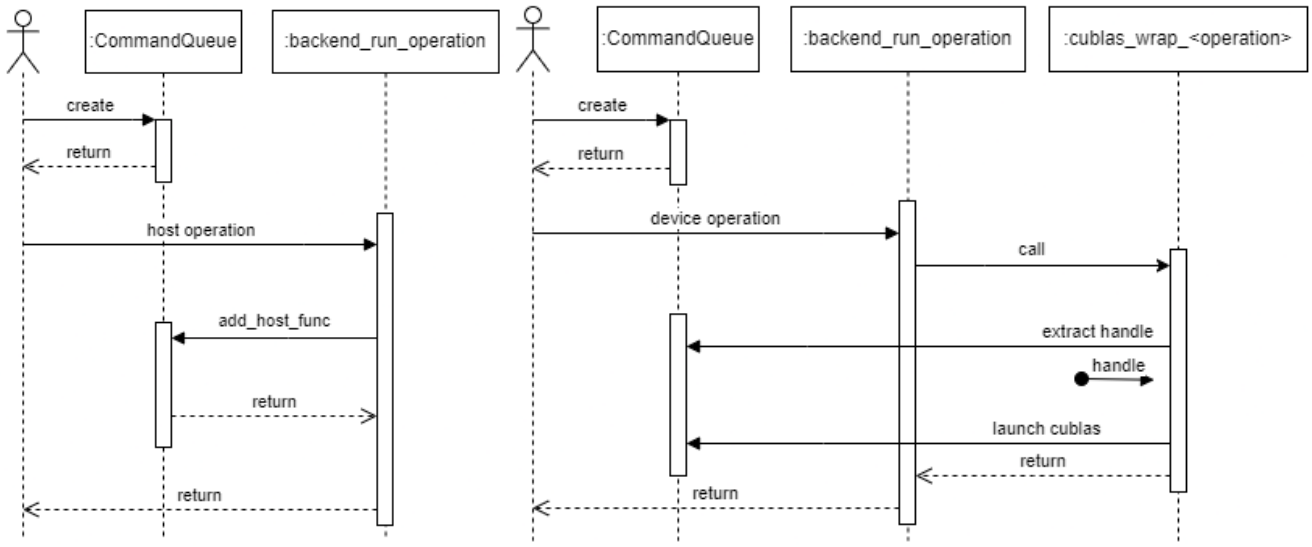
Τα δεδομένα δίνονται ως `void*` επειδή χρησιμοποιούνται διαφορετικοί τύποι δεδομένων (παρέχονται συσκευαστές δεδομένων, data wrappers) για διαφορετικούς τύπους πράξεων. Όμως, σε όλες τις περιπτώσεις υπάρχει στα δεδομένα το αναγνωριστικό της συσκευής (`dev_id`) για την οποία προορίζεται η συγκεκριμένη πράξη. Με βάση αυτό, η `backend_run_operation` κάνει τον εξής διαχωρισμό: Αν $\texttt{dev\_id} = -1$ η πράξη προορίζεται για εκτέλεση στον host, και χρησιμοποιείται η `CommandQueue::add_host_func` για να τοποθετήσει στην ουρά μια κλήση cBLAS. Αλλιώς, εάν $\texttt{dev\_id} \geq 0$ καλείται μια συνάρτηση `cublas_wrap_<operation>`, η οποία λαμβάνει το *handle* από την δομή της ουράς και τοποθετεί την κατάλληλη πράξη cuBLAS στο *stream*. Η διαδικασία αυτή παρουσιάζεται και στα διαγράμματα του σχήματος 2.2. Αξίζει να σημειωθεί ότι και στις δύο περιπτώσεις η επιστροφή από την `backend_run_operation` δηλώνει απλώς ότι η εργασία τοποθετήθηκε στην ουρά, αλλά, πιθανότατα, δεν έχει ολοκληρωθεί ακόμα.

Όσον αφορά τον συγχρονισμό μεταξύ host και ουράς εργασιών, πραγματοποιείται με την μέθοδο `CommandQueue::sync_barrier()`, η οποία καλεί `cudaStreamSynchronize` στο υποκείμενο *stream*.

2.1.2 Event

Τα *events* της Universal Helpers βασίζονται στα CUDA *events*, αλλά τα επεκτείνουν. Αρχικά, οι δυνατές καταστάσεις στις οποίες μπορεί να βρεθεί ένα *event* είναι:

- UNRECORDED: δεν έχει τοποθετηθεί ακόμα σε κάποια ουρά



Σχήμα 2.2: Τοποθέτηση πράξης BLAS στην ουρά: (a) η πράξη θα εκτελεστεί στη CPU, (b) η πράξη θα εκτελεστεί σε GPU

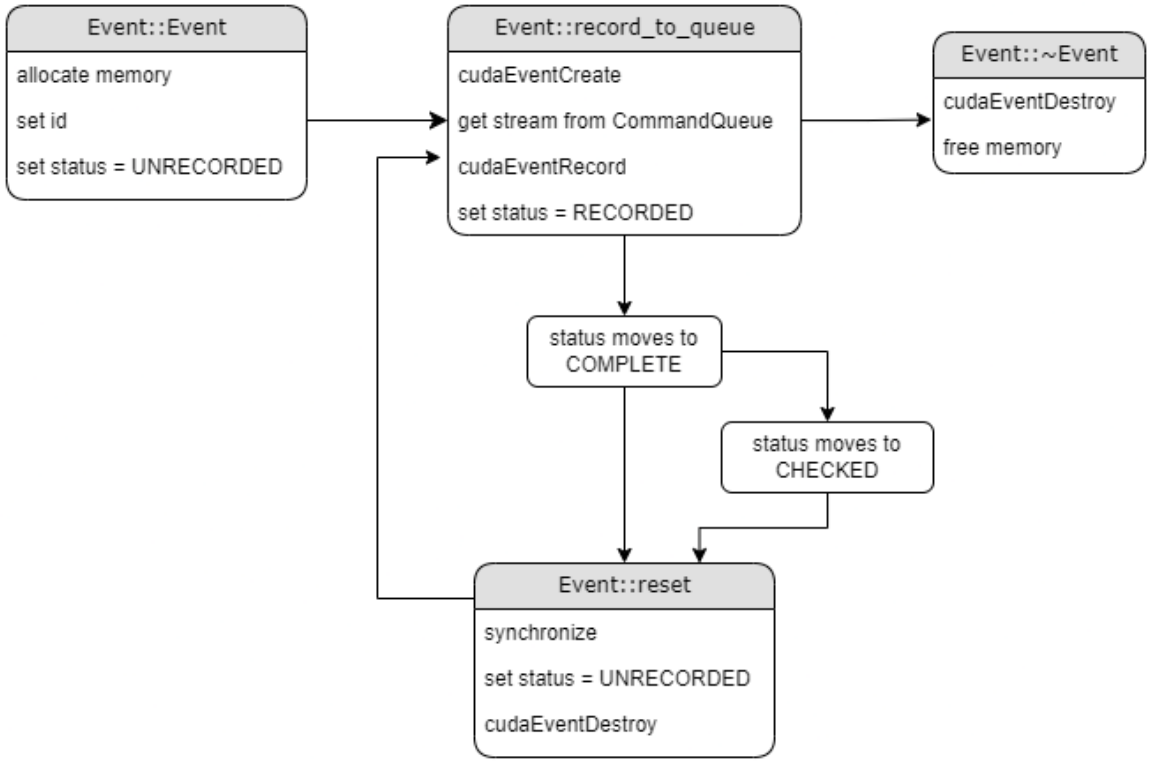
- RECORDED: έχει τοποθετηθεί σε ουρά, αλλά δεν έχει ολοκληρωθεί ακόμα
- COMPLETE: έχει ολοκληρωθεί
- CHECKED: έχει ολοκληρωθεί και κάποια προηγούμενη κλήση έχει διαπιστώσει την ολοκλήρωσή του
- GHOST: έχει προκύψει σφάλμα με αυτό το *event*

Φαίνεται ότι με αυτό τον τρόπο, παρέχεται μια πιο ακριβής περιγραφή της κατάστασης του *event* σε σχέση με αυτή που δίνει το CUDA. Οι τιμές `cudaSuccess` και `cudaErrorNotReady` που χρησιμοποιεί το CUDA, αντιστοιχούν στις καταστάσεις COMPLETE και RECORDED. Βέβαια, εδώ διαχωρίζεται η κατάσταση COMPLETE από την UNRECORDED (πρόκειται για δύο ξεχωριστές καταστάσεις), κάτι που, όπως αναφέρθηκε στην ενότητα 1.4.2, δεν ισχύει για τα CUDA *events*. Εκεί επιστρέφεται η τιμή `cudaSuccess` και στις δύο αυτές καταστάσεις. Όσον αφορά την κατάσταση CHECKED, χρησιμοποιείται έτσι ώστε να αποφεύγονται περιττές κλήσεις του CUDA API όταν έχει ήδη φτάσει το *event* σε κατάσταση ολοκλήρωσης.

Ταυτόχρονα, ο στόχος της Universal Helpers ήταν να παρέχει *events* που μπορούν να τοποθετηθούν σε οποιαδήποτε ουρά και συσκευή. Η συγκεκριμένη δυνατότητα δεν υπάρχει στα CUDA *events*, καθώς η `cudaEventRecord` μπορεί να χρησιμοποιηθεί μόνο μεταξύ *event* και *stream* που ανήκουν στο ίδιο CUDA *context*. Αυτό σημαίνει ότι πρέπει, τουλάχιστον, το *event* και το *stream* να έχουν δημιουργηθεί στην ίδια συσκευή (έχοντας ορίσει την ίδια τρέχουσα συσκευή). Για να αντιμετωπιστεί αυτός ο περιορισμός, υλοποιήθηκε η πολιτική των "Lazy events".

Ο χαρακτηρισμός "Lazy events" αναφέρεται στο γεγονός ότι η βιβλιοθήκη καθυστερεί την πραγματική δημιουργία του *event* (με την `cudaEventCreate`) μέχρι να είναι εντελώς απαραίτητη. Συγκεκριμένα, ο κατασκευαστής της κλάσης `Event` δεσμεύει μνήμη (malloc), ορίζει το αναγνωριστικό του *event* (id), θέτει την κατάσταση του αντικειμένου σε UN-

RECORDED, αλλά δεν δημιουγεί το CUDA *event*. Η `cudaEventCreate` καλείται από τη μέθοδο `Event::record_to_queue` ακριβώς πριν την `cudaEventRecord`. Αναλυτικότερα, η `Event::record_to_queue` δημιουργεί το CUDA *event*, λαμβάνει το *stream* από την `CommandQueue`, τοποθετεί το *event* στο *stream* και θέτει την κατάστασή του σε RECORDED. Η λειτουργικότητα συμπληρώνεται από την `Event::reset`, η οποία θέτει την κατάσταση του *event* σε UNRECORDED και καταστρέφει το ήδη χρησιμοποιημένο CUDA *event* (`cudaEventDestroy`). Με τον τρόπο αυτό, το "ίδιο" *event* μπορεί να τοποθετηθεί εκ νέου σε κάποια άλλη ουρά, ακόμα και σε άλλη συσκευή. Αυτή η διαδικασία φαίνεται στο σχήμα 2.3 Επίσης, υπάρχει η `Event::soft_reset` που πραγματοποιεί μόνο επαναφορά της κατάστασης UNRECORDED, χωρίς καταστροφή του CUDA *event*.



Σχήμα 2.3: Κύκλος ζωής ενός *event*

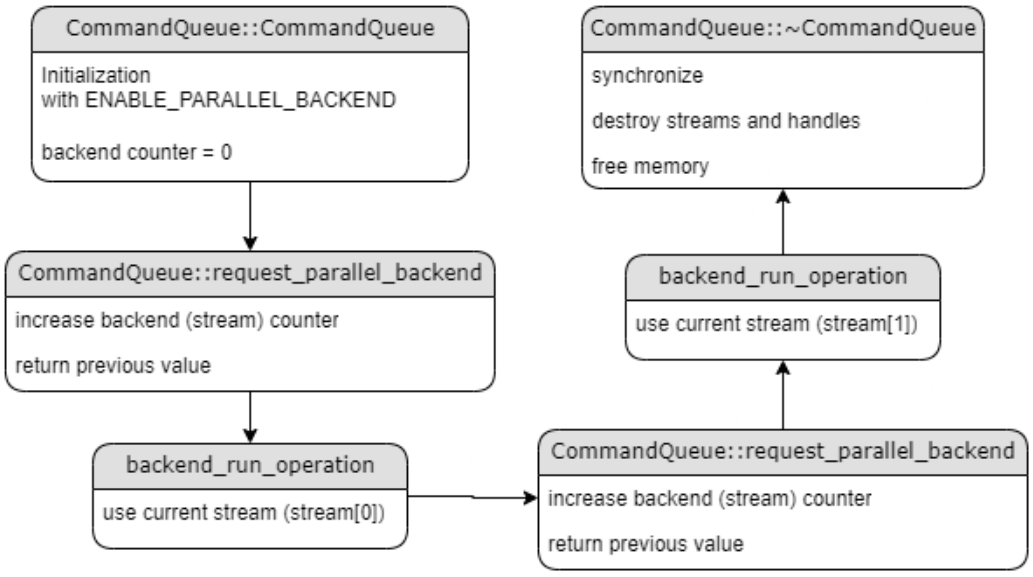
Επιπλέον, η κλάση `Event` παρέχει τη μέθοδο `Event::sync_barrier` για συγχρονισμό host - *event*. Ουσιαστικά εδώ καλείται η `cudaEventSynchronize` και ενημερώνεται κατάλληλα η κατάσταση του *event*. Για τον συγχρονισμό ουράς - *event* χρησιμοποιείται η μέθοδος `CommandQueue::wait_for_event`, η οποία εξάγει το CUDA *stream* από την `CommandQueue` και το CUDA *event* από το `Event`, και καλεί την `cudaStreamWaitEvent`. Τέλος, η μέθοδος `Event::query_status` επιστρέφει την τρέχουσα κατάσταση του *event* (αν χρειάζεται καλείται η `cudaEventQuery`), και η `Event::checked` προκαλεί τη μετάβαση της κατάστασης από COMPLETE σε CHECKED. Σημειώνεται ότι η `Event::checked` μπορεί να χρησιμοποιηθεί μόνο όταν το *event* βρίσκεται στην κατάσταση COMPLETE.

2.1.3 Parallel backend

Η βιβλιοθήκη παρέχει και τη λειτουργία ”Parallel backend” με ενεργοποίηση της σημαίας `ENABLE_PARALLEL_BACKEND` κατά τον χρόνο μεταγλώττισης του κώδικα. Με αυτό τον τρόπο, οι ουρές μπορούν να λειτουργήσουν και σε κατάσταση εκτέλεσης εκτός σειράς (*out-of-order* execution).

Για τον σκοπό αυτό, τροποποιείται η κλάση `CommandQueue` και, συνεπώς, παύουν να ισχύουν ορισμένες λεπτομέριες του σχήματος 2.1. Οι δείκτες `cqueue_backend_ptr` και `cqueue_backend_data` γίνονται πίνακες από δείκτες. Έχουν μέγεθος `MAX_BACKEND_L`, μεταβλητή που τίθεται κατά τον χρόνο μεταγλώττισης, και αποθηκεύουν CUDA *streams* και τα αντίστοιχα cuBLAS *handles*. Έτσι, μπορούν να τοποθετηθούν εργασίες σε διαφορετικά *streams*, εντός της ίδιας ουράς, οδηγώντας σε εκτέλεση εκτός σειράς όσον αφορά το `CommandQueue`.

Ακόμα, υπάρχει ο μετρητής `backend_ctr`, ο οποίος προσδιορίζει το ”τρέχον” *stream* (παίρνει τιμές από 0 έως `MAX_BACKEND_L`-1). Οι μέθοδοι `CommandQueue::add_host_func`, `CommandQueue::wait_for_event`, `Event::record_to_queue`, αλλά και όποια βοηθητική συνάρτηση χρησιμοποιεί μία ουρά, εκτελούν τη λειτουργία τους (προσθήκη εργασίας, *event* ή συγχρονισμός) πάνω στο τρέχον *stream*. Η χρησιμοποίηση του συνόλου των *streams* έρχεται με τις μεθόδους `CommandQueue::request_parallel_backend` και `CommandQueue::set_parallel_backend`, οι οποίες αλλάζουν το τρέχον *stream*. Συγκεκριμένα, η πρώτη επιστρέφει τον αριθμό του τρέχοντος *stream* και αυξάνει τον μετρητή ώστε να παραπέμπει στο επόμενο *stream*, ενώ η δεύτερη απλώς θέτει τον μετρητή στη ζητούμενη τιμή. Σχετικά με την `CommandQueue::set_parallel_backend`, είναι ευθύνη του προγραμματιστή να τη χρησιμοποιήσει σωστά, καλώντας την για έγκυρες τιμές του μετρητή ($[0, \texttt{MAX\_BACKEND\_L}-1]$). Σε αντίθετη περίπτωση προκύπτει σφάλμα πρόσβασης στη μνήμη (*segmentation fault*). Τέλος, η `CommandQueue::sync_barrier` πραγματοποιεί συγχρονισμό όλων των *streams* της ουράς. Ένα παράδειγμα χρήσης της ”Parallel backend” λειτουργίας φαίνεται στο σχήμα 2.4, όπου τοποθετούνται δύο εργασίες σε μία ουρά για εκτέλεση εκτός σειράς.



Σχήμα 2.4: Τοποθέτηση δύο εργασιών σε *out-of-order* ουρά

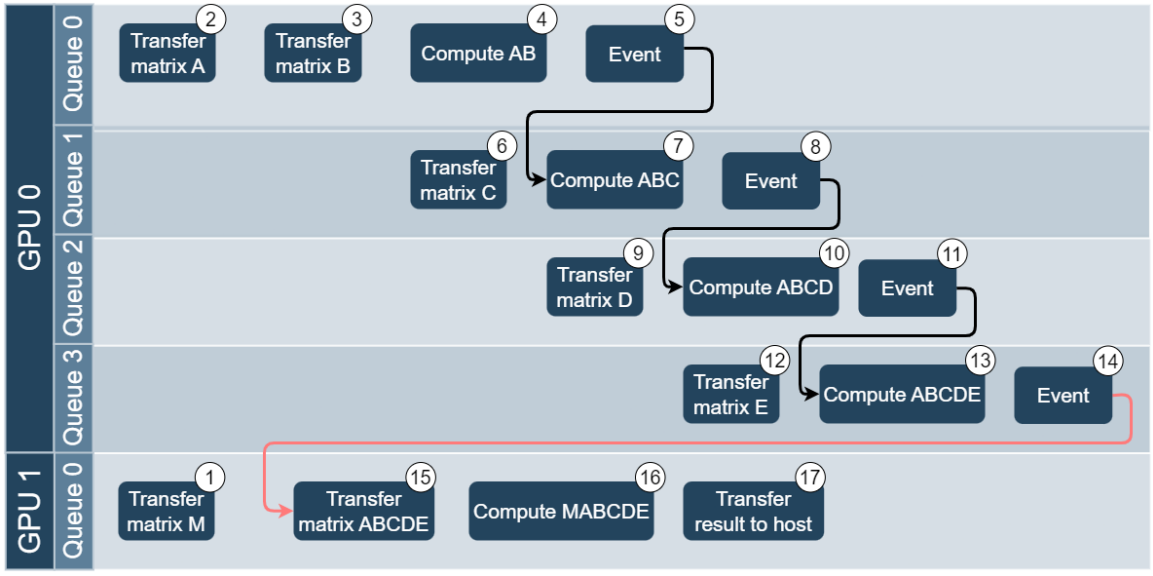
2.1.4 Testing

Προκειμένου να αναπτυχθούν διαφορετικές υλοποιήσεις της Universal Helpers, κρίθηκε αναγκαία η σχεδίαση δοκιμαστικών προγραμμάτων (*tests*). Αυτά τα προγράμματα χρησιμοποιούν τις κλάσεις και τις συναρτήσεις της βιβλιοθήκης σε διάφορα σενάρια, που έγινε προσπάθεια να καλύπτουν όλο το εύρος των βασικών χρήσεων. Με αυτό τον τρόπο, ήταν δυνατό να ελέγχεται άμεσα η ορθότητα μιας νέας υλοποίησης, αφού σε περίπτωση εσφαλμένου σχεδιασμού κάποιο δοκιμαστικό πρόγραμμα δεν θα εκτελούνταν σωστά. Σε αυτό βοήθησαν και τα εργαλεία cmake και ctest, τα οποία αναλαμβάνουν να παράξουν τα εκτελέσιμα προγράμματα και να τα δοκιμάσουν.

Καθένα από τα *tests* εξετάζει ένα σενάριο, στο οποίο υποβάλλονται εργασίες στις ουρές και εφαρμόζονται οι μεταξύ τους εξαρτήσεις με *events*. Αφού ολοκληρωθεί ο παράλληλος υπολογισμός στον οποίο συμμετέχουν GPUs και η CPU, το υπολογισμός πραγματοποιείται εκ νέου στη CPU, σειριακά. Στο τέλος συγκρίνονται τα δύο αποτελέσματα που πρέπει να ταυτίζονται. Σε διαφορετική περίπτωση το πρόγραμμα τερματίζει επιστρέφοντας κάποιον αριθμό σφάλματος.

Στο σχήμα 2.5 παρουσιάζεται ένα ενδεικτικό δοκιμαστικό πρόγραμμα. Η αρίθμηση $[1, 17]$ δείχνει τη σειρά με την οποία η CPU αποστέλλει τις εντολές, ενώ η σχετικές θέσεις τους από αριστερά προς τα δεξιά δείχνουν τη σειρά με την οποία μπορούν να εκτελεστούν στις GPU0 και στη GPU1, αντίστοιχα. Οι κόμβοι "Transfer" αντιπροσωπεύουν κλήσεις `CoCoMemcpy2DAsync`, οι κόμβοι "Compute" κλήσεις `backend_run_operation` και οι κόμβοι "Event" κλήσεις `record_to_queue`. Επίσης, οι συνδέσεις μεταξύ "Event" και κόμβου σε άλλη ουρά αντιπροσωπεύουν χρήσεις της `wait_for_event` για συγχρονισμό. Αξίζει να σημειωθεί ότι οι μεταφορές των πινάκων M, A, B, C, D και E είναι μεταφορές *host-to-device*, η μεταφορά του γινομένου $ABCDE$ είναι *device-to-device*, ενώ η τελευταία μεταφορά είναι *device-to-host*. Στο συγκεκριμένο πρόγραμμα υπάρχει η δυνατότητα να οριστούν από τη γραμμή εντολών (ως *command line arguments*) το μέγεθος των πινάκων που πολλαπλασιάζονται, καθώς και ο χαρακτηρισμός "synched/desynched". Η "synched" εκτέλεση εφαρμόζει τον συγχρονισμό που υπονοεί η κόκκινη σύνδεση του σχήματος, ενώ η "desynched" δεν την εφαρμόζει. Στη δεύτερη περίπτωση το πρόγραμμα δεν περιμένει τα σωστά αποτελέσματα στη μνήμη του πίνακα $ABCDE$, και αναμένεται να μην υπολογίσει το σωστό τελικό γινόμενο.

Ωστόσο, σχεδιάστηκαν και πιο σύνθετα προγράμματα που εξετάζουν υπολογισμούς κατανεμημένους στη CPU και στις 3 GPUs που διαθέτει το σύστημα που χρησιμοποιήθηκε για τους σκοπούς της παρούσας εργασίας. Πιο συγκεκριμένα, πρόκειται για μια Nvidia GeForce GTX 1060 (6GB) και δύο Nvidia Tesla V100 SXM2 (32GB).



Σχήμα 2.5: Δοκιμαστικό πρόγραμμα ”simpleInterDevice” για τον έλεγχο των εξαρτήσεων μεταξύ CommandQueues διαφορετικών συσκευών

2.2 Pthreads backend

Σε αυτή την ενότητα παρουσιάζεται η υλοποίηση της βιβλιοθήκης Universal Helpers με pthreads αντί για κλήσεις CUDA. Η λογική αυτής της υλοποίησης είναι ότι κάθε `CommandQueue` αντιστοιχίζεται με ένα νήμα, το οποίο αναλαμβάνει να εκτελέσει τις εργασίες που άλλα νήματα τοποθετούν στην ουρά.

2.2.1 Μοντελοποίηση εργασιών

Αρχικά, ήταν απαραίτητο να οριστεί και προγραμματιστικά η έννοια της "εργασίας". Για τον σκοπό αυτό δημιουργήθηκε το C/C++ *struct* που ακολουθεί.

```
typedef struct pthread_task {
    void* func;
    void* data;
}* pthread_task_p;
```

Φαίνεται ότι η "εργασία" (δηλαδή το `pthread_task`) αποτελείται από μια συνάρτηση (`func`) και τα δεδομένα (`data`) πάνω στα οποία θα δράσει αυτή η συνάρτηση. Τα δύο αυτά πεδία είναι τύπου `void*` προκειμένου να εξυπηρετούν τις ανάγκες διαφορετικών "ειδών" εργασιών. Με τον τρόπο αυτό μπορούν να οριστούν εργασίες χρήσιμες για την εσωτερική λειτουργία της βιβλιοθήκης, αλλά και εξωτερικές εργασίες ορισμένες από τον χρήστη (user-defined tasks). Για παράδειγμα, η συνάρτηση που ακολουθεί λαμβάνει στην είσοδό της έναν αριθμό x και υπολογίζει το άθροισμα $\sum_{i=1}^{x} i$. Σημειώνεται ότι η συνάρτηση τροποποιεί το όρισμά της για να επιστρέψει το αποτέλεσμα.

```
void* taskFunc(void* input){
    int* x = (int*) input;
    int sum = 0;
    for(int i = 1; i <= *x; i++){
        sum += i;
    }
    *x = sum;
    return 0;
}
```

Τέτοιες εργασίες, που προορίζονται για εκτέλεση στη CPU, τοποθετούνται στην ουρά με την `CommandQueue::add_host_func`. Συγκεκριμένα, η παραπάνω εργασία τοποθετείται με την εντολή:
`queue_pointer->add_host_func((void*) &taskFunc, (void*) &input);`
Γίνεται εμφανές ότι είναι ευθύνη του προγραμματιστή να προετοιμάσει κατάλληλα τα δεδομένα σε έναν δείκτη (συνήθως δείκτη σε *struct*) ώστε να είναι προσβάσιμα από το νήμα που θα εκτελέσει την εργασία.

Η αποθήκευση των εργασιών εντός του `CommandQueue` γίνεται χρησιμοποιώντας τη δομή *queue* της std βιβλιοθήκης της C++. Ο κατασκευαστής της κλάσης καλείται ως `queue<pthread_task_p>`, το οποίο είναι ισοδύναμο με `queue<pthread_task*>` (βλ. ορισμό του `pthread_task`). Αυτό σημαίνει ότι αποθηκεύονται δείκτες σε εργασίες.

2.2.2 Βρόγχος εκτέλεσης εργασιών (task execution loop)

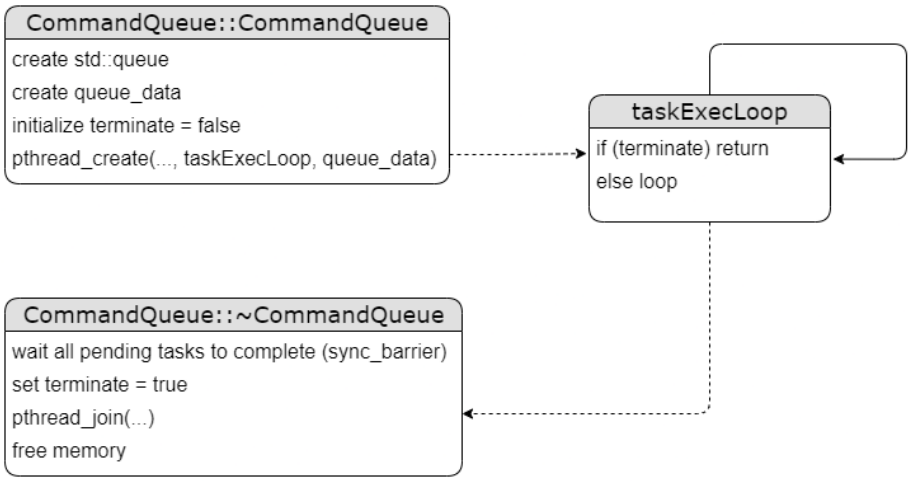
Το αμέσως επόμενο που χρειάζεται να εξεταστεί είναι ο τρόπος με τον οποίο εκτελούνται οι εργασίες αφού έχουν τοποθετηθεί σε κάποιο `CommandQueue`. Αυτό γίνεται μέσω του βρόγχου εκτέλεσης εργασιών.

Ο εν λόγω βρόγχος περιλαμβάνεται στη συνάρτηση που εκτελεί το νήμα που δρομολογεί τις εργασίες της ουράς. Ο ψευδοκώδικας του βρόγχου παρουσιάζεται στον αλγόριθμο 4.

Αλγόριθμος 4: Ψευδοκώδικας του βρόγχου εκτέλεσης εργασιών

```
1  Function TaskExecLoop(queue data):
2      while CommandQueue is active do
3          if terminate is set then
4              break
5          end
6          if queue is not empty then
7              get next task
8              execute
9              pop that task from the queue
10         end
11     end
12     return
```

Όπως φαίνεται το νήμα που έχει αναλάβει το συγκεκριμένο `CommandQueue` εκτελεί σειριακά τις εργασίες που προστίθενται στην ουρά. Όταν η δομή της ουράς αδειάζει, το νήμα συνεχίζει να ελέγχει αν προστέθηκε κάποια νέα εργασία και, αν υπάρχει, την δρομολογεί. Η σημαία *terminate* (boolean flag) χρησιμοποιείται από τον καταστροφέα (destructor) της κλάσης `CommandQueue` για να ειδοποιήσει το νήμα ότι πρέπει να σταματήσει την εκτέλεση του βρόγχου και να επιστρέψει από τη συνάρτηση. Από την πλευρά του, το νήμα που εκτελεί τον καταστροφέα της κλάσης, λοιπόν, θέτει τη σημαία *terminate* και περιμένει το νήμα της ουράς με `pthread_join()`. Ο κύκλος ζωής του νήματος που εκτελεί τις εργασίες του `CommandQueue` φαίνεται στο σχήμα 2.6.



Σχήμα 2.6: Ο κύκλος ζωής του νήματος που εκτελεί τις εργασίες

Από την περιγραφή προκύπτει ότι στο σύστημα υπάρχουν νήματα-παραγωγοί: τα νήματα του χρήστη που προσθέτουν εργασίες στην ουρά, και το νήμα της ουράς που λειτουργεί ως καταναλωτής εργασιών. Για αυτό είναι απαραίτητη η εφαρμογή κλειδώματος (`queueLock`), που πρέπει να χρησιμοποιείται κάθε φορά που γίνεται πρόσβαση σε δεδομένα της ουράς. Κάποιες χαρακτηριστικές περιπτώσεις πρόσβασης είναι:

- εισαγωγή νέας εργασίας (`push()`)
- εξαγωγή εργασίας προς εκτέλεση (`front()`, `pop()`)
- οποιαδήποτε άλλη μέθοδος της δομής `std::queue` (για παράδειγμα η `size()`)
- έλεγχος ή ενημέρωση της σημαίας *terminate*

Η υλοποίηση έγινε με *spinlock*, με χρήση των ατομικών συναρτήσεων `__sync_lock_test_and_set` και `__sync_lock_release`.

2.2.3 Τα events ως εργασίες

Για να μπορούν να τοποθετούνται *events* στην ουρά, υλοποιούνται, και αυτά, ως εργασίες. Η συνάρτηση της δομής `pthread_task` ενός *event* είναι η `eventFunc`, η οποία απλώς θέτει την κατάστασή του σε COMPLETE. Με τον τρόπο αυτό, όταν η συγκεκριμένη εργασία εκτελεστεί από κάποιο νήμα ουράς, το *event* θεωρείται ολοκληρωμένο.

Ταυτόχρονα, χρειάζεται να οριστεί και μια δεύτερη βοηθητική εργασία. Προκειμένου να περιμένει κάποιο `CommandQueue` την ολοκλήρωση ενός *event* (`CommandQueue::wait_for_event`) προστίθεται στην ουρά μία εργασία με συνάρτηση `blockQueue`. Αυτή η συνάρτηση παίρνει ως όρισμα το *event* και πραγματοποιεί ενεργό αναμονή (*busy-wait*) μέχρι η κατάσταση του *event* να γίνει COMPLETE ή CHECKED. Με αυτό τον τρόπο το νήμα της ουράς μένει απασχολημένο και δεν προχωρά στις επόμενες εργασίες μέχρι να ολοκληρωθεί το *event*. Παρομοίως με την `blockQueue` υλοποιείται και η `Event::sync_barrier()`, κρατώντας απασχολημένο το καλών νήμα όσο η κατάσταση του *event* είναι RECORDED.

Αλγόριθμος 5: Ψευδοκώδικας των eventFunc και blockQueue

```
1 Function eventFunc(event):
2     event.status = COMPLETE return
3 Function blockQueue(event):
4     while event.status == RECORDED do
5         wait
6     end
7     return
```

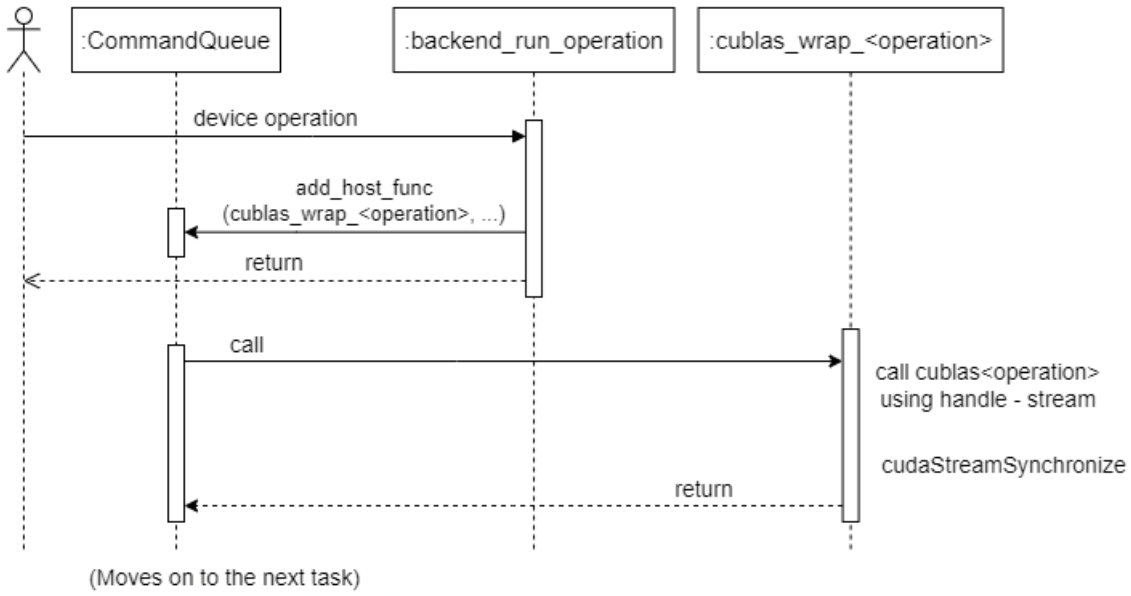
Επιπλέον, παρατηρείται ότι από τη σχεδίασή τους τα *events* αυτής της υλοποίησης δεν δεσμεύονται με αποκλειστικό τρόπο σε κάποια συσκευή και, έτσι, δεν υπάρχει η ανάγκη για τον μηχανισμό των "Lazy events" που παρουσιάστηκε για την υλοποίηση σε CUDA.

2.2.4 Αναγκαίες κλήσεις CUDA

Οι εργασίες που περιγράφηκαν μέχρι αυτό το σημείο εκτελούνται από τη CPU και, για αυτό, τοποθετούνται στην ουρά με την μέθοδο `CommandQueue::add_host_func`. Ωστόσο, για να εξυπηρετηθούν εργασίες που προορίζονται για κάποια GPU, ήταν απαραίτητο να χρησιμοποιηθούν κλήσεις του CUDA API.

Προς αυτή την κατεύθυνση τοποθετήθηκε ένα CUDA *stream* και ένα cuBLAS *handle* σε κάθε `CommandQueue`, ώστε να μπορούν να αποσταλούν πυρήνες cuBLAS και ασύγχρονες μεταφορές δεδομένων στις GPUs. Στο συγκεκριμένο *stream* (και στο *handle*) έχει πρόσβαση μόνο το νήμα του `CommandQueue`. Αυτό σημαίνει ότι κάποιο νήμα του χρήστη μπορεί να καλέσει την `backend_run_operation`, αυτή θα προσθέσει την κατάλληλη εργασία στην ουρά και θα επιστρέψει. Η συνάρτηση της εργασίας περιέχει εντολές που τοποθετούν τον αντίστοιχο πυρήνα BLAS στο *stream* και, πριν επιστρέψει συγχρονίζεται με το *stream* (`cudaStreamSynchronize`), όπως παρουσιάζεται στο σχήμα 2.7. Ο συγχρονισμός είναι απαραίτητος προκειμένου να διατηρηθεί η συνέπεια της σχεδίασης, σύμφωνα με την οποία το νήμα της ουράς εκτελεί σειριακά τις εργασίες και δεν προχωρά στην επόμενη εργασία πριν ολοκληρωθεί η προηγούμενη. Από το σχήμα 2.7 φαίνεται ότι και για τις εργασίες που θα σταλούν σε κάποια GPU χρησιμοποιείται εσωτερικά η `CommandQueue::add_host_func`, χωρίς αυτό να επηρεάζει τη διεπαφή προς τον χρήστη. Ουσιαστικά το αν η εργασία θα εκτελεστεί στη CPU ή σε GPU καθορίζεται από τη συνάρτησή της.

Στο πλαίσιο ανάπτυξης και αποσφαλμάτωσης αυτής της υλοποίησης χρησιμοποιήθηκε μια δεξαμενή από 2 ως 4 *streams* (*stream pool*) αντί για ένα. Με αυτό τον τρόπο μπορούσε να διαπιστωθεί η σωστή λειτουργία της βιβλιοθήκης, χωρίς να στηρίζεται στη σειριακή εκτέλεση που παρέχει ένα *stream*. Ωστόσο, πρακτικά η βιβλιοθήκη λειτουργεί με ένα CUDA *stream* ανά ουρά εργασιών.



Σχήμα 2.7: Εκτέλεση πράξης BLAS στην υλοποίηση με pthreads

Εδώ αξίζει να σημειωθεί πως είναι δυνατόν να υλοποιηθεί ασύγχρονη μεταφορά δεδομένων, χωρίς *stream* με τον εξής τρόπο: η συνάρτηση `CoCoMemcpyAsync` τοποθετεί στην ουρά μια

εργασία, η οποία πραγματοποιεί σύγχρονη μεταφορά δεδομένων (cudaMemcpy), που δεν απαιτεί *stream*. Έτσι, η μεταφορά είναι ασύγχρονη ως προς τον χρήστη, αφού εκτελείται από άλλο νήμα (το νήμα του CommandQueue). Το πρόβλημα με αυτή την προσέγγιση είναι ότι δεν επιτρέπει επικάλυψη μεταφοράς δεδομένων και υπολογισμού στη GPU, καθώς η σύγχρονη cudaMemcpy εφαρμόζει έμμεσο συγχρονισμό στις κλήσεις CUDA [Rennich, 2011].

2.2.5 Ζητήματα συγχρονισμού

Στην υλοποίηση της Universal Helpers με CUDA, η CoCoSyncCheckErr() επιβάλλει τον συγχρονισμό με την τρέχουσα συσκευή καλώντας την cudaDeviceSynchronize(). Προκειμένου να εφαρμοστεί κάτι αντίστοιχο στην υλοποίηση με pthreads, χρειάστηκε να δημιουργηθεί μηχανισμός που συνδέει μία GPU με όλα τα CommandQueues που δρομολογούν εργασίες σε αυτή.

Αυτό έγινε με έναν καθολικό πίνακα, queuesPerDevice, από std::vector<CommandQueue *>, δηλαδή έναν μονοδιάστατο πίνακα που κάθε θέση του περιέχει ένα σύνολο από CommandQueues. Οι ουρές που βρίσκονται στην i-οστή θέση του πίνακα είναι αυτές που σχετίζονται με τη συσκευή i. Τα CommandQueues που φτιάχνει ο χρήστης για τον *host* (dev_id=-1) καταχωρούνται στη συσκευή 0. Αυτό είναι συνεπές με τον τρόπο που χειρίζεται η CoCoPeLiaSelectDevice τα αναγνωριστικά των συσκευών και του *host*, αφού η "επιλογή" της συσκευής -1 (δηλαδή του *host*) οδηγεί στην κλήση cudaSetDevice(0). Τη δομή του πίνακα queuesPerDevice επεξεργάζονται οι ακόλουθες συναρτήσεις:

- InitializeQueuesPerDevice():

  δεσμεύει μνήμη και αρχικοποιεί τον πίνακα με βάση τον αριθμό των διαθέσιμων συσκευών του συστήματος. Καλείται μόνο μία φορά στην αρχή του προγράμματος.

- UninitializeQueuesPerDevice():

  αποδεσμεύει τη μνήμη που χρησιμοποιήθηκε. Καλείται μόνο μία φορά στο τέλος του προγράμματος.

- AssignQueueToDevice(CommandQueue * queue, int dev):

  συσχετίζει τη συσκευή με το CommandQueue. Αυτό γίνεται προσθέτοντάς το στη δομή std::vector<CommandQueue *>, η οποία βρίσκεται στη θέση του πίνακα που αφορά τη συγκεκριμένη συσκευή. Καλείται κάθε φορά που δημιουργείται ένα νέο CommandQueue.

- UnassignQueueFromDevice(CommandQueue * queue, int dev):

  αφαιρεί το CommandQueue από το vector που αντιστοιχεί στη συσκευή. Καλείται κάθε φορά που καταστρέφεται ένα CommandQueue.

- DeviceSynchronize():

  επιβάλλει τον συγχρονισμό όλων των CommandQueues που είναι καταχωρημένα στην τρέχουσα συσκευή. Εσωτερικά καλεί CommandQueue::sync_barrier() για κάθε CommandQueue που βρίσκει στο αντίστοιχο vector. Ισοδύναμη της cudaDeviceSynchronize().

2.2.6 Event_timer

Η κλάση `Event_timer` παρέχει τη δυνατότητα μέτρησης του χρόνου που μεσολάβησε μεταξύ της ολοκλήρωσης δύο γεγονότων. Αυτό στην υλοποίηση του CUDA γινόταν, απλώς, με καταγραφή των δύο CUDA *events* και με την κλήση `cudaEventElapsedTime`.

Για να γίνει η αντίστοιχη μέτρηση σε αυτή την υλοποίηση προστέθηκε στα δεδομένα κάθε *event* το πεδίο `completeTime`, που είναι τύπου `std::chrono::steady_clock::time_point`. Φαίνεται ότι χρησιμοποιείται το ρολόι `steady_clock` της std βιβλιοθήκης, που καταγράφει *wall time*, δηλαδή τον συνολικό χρόνο που έχει μεσολαβήσει από κάποιο σημείο αφετηρίας, και προτείνεται για μέτρηση χρονικών διαστημάτων. Με αυτή την προσθήκη, η συνάρτηση του *event* (`eventFunc`) καταγράφει και τη στιγμή ολοκλήρωσής του με τη συνάρτηση `std::chrono::steady_clock::now()`.

Έτσι, οι μέθοδοι `Event_timer::start_point` και `Event_timer::stop_point` τοποθετούν τα δύο *events* σε κάποιο `CommandQueue`. Με τη σειρά της η `Event_timer::sync_get_time` υπολογίζει και επιστρέφει τη χρονική διαφορά από τα δύο `std::chrono::steady_clock::time_points` με τη συνάρτηση `std::chrono::duration_cast<std::chrono::milliseconds>`.

2.2.7 Parallel backend

Για να υποστηρίζεται και η λειτουργία "Parallel backend", κάθε `CommandQueue` επεκτάθηκε ώστε να περιλαμβάνει περισσότερες από μία δομές `queue<pthread_task*>`. Κάθε μια έχει και το δικό της νήμα (που δημιουργείται από τον κατασκευαστή της κλάσης) ώστε οι εργασίες της να δρομολογούνται ανεξάρτητα από τις υπόλοιπες, που βρίσκονται εντός του ίδιου `CommandQueue`. Ουσιαστικά, αντί να υπάρχουν παράλληλα CUDA *streams*, όπως στην βασική υλοποίηση, υπάρχουν παράλληλα νήματα που εξυπηρετούν διαφορετικές ουρές.

Στις πρώτες δοκιμές παρατηρήθηκε τεράστια πτώση της επίδοσης ανάμεσα στην εκτέλεση με 1 ουρά ανά `CommandQueue` και 2 παράλληλες ουρές ανά `CommandQueue`. Φυσικά, η επίδοση γινόταν ακόμα χειρότερη με περισσότερες από 2 παράλληλες ουρές. Συγκεκριμένα, διαπιστώθηκε ότι ενώ με 1 ουρά η `CommandQueue::sync_barrier()` είχε μέσο χρόνο εκτέλεσης $agv\_t \simeq 52\mu sec$, με 2 ουρές ο μέσος χρόνος εκτέλεσης έφτανε $agv\_t \simeq 3200\mu sec$. Αυτό συνέβαινε λόγω του `queueLock` που χρειάζονται η `taskExecLoop` και η `CommandQueue::sync_barrier()` για να επεξεργαστούν τα δεδομένα της κάθε ουράς. Αυτό το κλείδωμα είναι ένα *spinlock* και η `taskExecLoop` το χρησιμοποιεί με μεγάλη συχνότητα, ιδιαίτερα όταν η ουρά είναι άδεια. Σε αυτή την περίπτωση, το νήμα που δρομολογεί τις εργασίες διεκδικεί το κλείδωμα, διαπιστώνει ότι η ουρά είναι άδεια, απελευθερώνει το κλείδωμα και επαναλαμβάνει τις τρεις αυτές πράξεις ασταμάτητα. Συνεπώς, το νήμα που εκτελεί τον βρόγχο σε μια άδεια ουρά, αφήνει και επαναδιεκδικεί αμέσως το *spinlock*, αποκτώντας το με μεγάλη πιθανότητα. Αυτό σημαίνει ότι η ουρά παραμένει "κλειδωμένη" από ένα νήμα που δεν παράγει έργο, και δεν προοδεύουν τα υπόλοιπα νήματα (όπως αυτό που εκτελεί την `CommandQueue::sync_barrier()`). Συνολικά εμφανίζεται πρόβλημα μερικής λιμοκτονίας (*starvation*).

Για την αντιμετώπιση του προβλήματος χρησιμοποιήθηκε η κλήση `usleep`, η οποία αναστέλλει την εκτέλεση του καλούντος νήματος για τον ζητούμενο χρόνο σε μsec. Τοποθετήθηκε σε δύο σημεία του κώδικα:

1. αφού το νήμα που εκτελεί την `taskExecLoop` αφήσει το κλείδωμα

2. αφού το νήμα που εκτελεί την `CommandQueue::sync_barrier()` αφήσει το κλείδωμα

Αυτό οδήγησε σε σημαντική βελτίωση της επίδοσης, καθώς το νήμα που αφήνει το κλείδωμα αφήνει και τον επεξεργαστή, δίνοντας τη δυνατότητα σε κάποιο άλλο νήμα να αποκτήσει το *spinlock*.

Στη συνέχεια, ήταν εφικτή περαιτέρω βελτιστοποίηση με τη χρήση μεταβλητής συνθήκης (*condition variable*), μαζί με το *boolean* πεδίο, *busy*, στα δεδομένα της ουράς. Αυτό έγινε προκειμένου το νήμα που εκτελεί την `CommandQueue::sync_barrier()` να μην ελέγχει με σταθμοσκόπηση (*polling*) την κατάσταση της ουράς (απασχολημένη / ελεύθερη), αλλά να αναστέλλει τη λειτουργία του μέχρι να ειδοποιηθεί για αλλαγή της κατάστασης. Από την πλευρά του το νήμα που εκτελεί τον βρόγχο της ουράς και εντοπίζει ότι η ουρά έχει αδειάσει, θέτει την κατάσταση `busy = false` και ενημερώνει όλα τα νήματα που μπορεί να περιμένουν καλώντας την `pthread_cond_broadcast`. Έπειτα, ελευθερώνει το κλείδωμα και καλεί την `usleep`, ώστε να επιτρέψει σε κάποιο από τα νήματα που περίμεναν να προχωρήσει.

2.3 HIP backend

Η Universal Helpers υλοποιήθηκε και σε HIP με τη βοήθεια του εργαλείου *hipify*. Ουσιαστικά, πρόκειται για μεταφορά του CUDA κώδικα σε HIP 5.4. Για αυτό χρειάστηκε να εγκατασταθούν στο σύστημα που πραγματοποιήθηκαν οι δοκιμές οι βιβλιοθήκες hipRAND και hipBLAS, ως αντίστοιχες των cuRAND και cuBLAS.

Αρχικά, εξετάστηκαν τα αρχεία του πηγαίου κώδικα σε CUDA με τη λειτουργία `--examine` του *hipify*. Αυτό ανέδειξε προειδοποιήσεις (*warnings*) για κάποιες κλήσεις, επειδή είτε θεωρούνται ξεπερασμένες (*deprecated*), είτε βρίσκονται σε πειραματικό στάδιο στην ανάπτυξη του HIP (*experimental*). Αυτές αναγράφονται στον πίνακα 2.2

Κλήση CUDA	**Χαρακτηρισμός**
`cudaMemcpyToArray`	deprecated
`cudaMemcpyFromArray`	deprecated
`cuCtxDetach`	deprecated
`cuDeviceComputeCapability`	deprecated
`cudaBindTexture`	deprecated
`cudaGetTextureAlignmentOffset`	deprecated
`cudaLaunchCooperativeKernelMultiDevice`	deprecated
`cudaMemoryTypeManaged`	experimental
`cudaLaunchHostFunc`	experimental

Πίνακας 2.2: Κλήσεις CUDA που εμφανίζουν προειδοποιήσεις στο HIP

Ωστόσο, αυτές οι κλήσεις αντικαταστάθηκαν από τις αντίστοιχες κλήσεις του HIP και οι δύο τελευταίες, οι οποίες βρίσκονται σε πειραματικό στάδιο, εξετάστηκαν ξεχωριστά. Στην περίπτωση της `cudaMemoryTypeManaged`, η αντικατάσταση έγινε από την `hipMemoryTypeManaged` και στο σημείο που καλείται προστέθηκε το προειδοποιητικό μήνυμα "using experimental hipMemoryTypeManaged". Όσον αφορά την `cudaLaunchHostFunc`,

διαπιστώθηκε ότι ενώ περιλαμβάνεται στην τεκμηρίωση (*documentation*) του HIP (μετά την έκδοση 5.2) και στο αρχείο κεφαλίδας `hip_runtime_api.h`, λείπει από το αρχείο `include/hip/nvidia_detail/nvidia_hip_runtime_api.h`.

Το συγκεκριμένο αρχείο περιέχει `inline` ορισμούς των κλήσεων του HIP, για συστήματα με Nvidia GPUs, οι οποίες καλούν την αντίστοιχη συνάρτηση CUDA, όπως στο παράδειγμα που ακολουθεί.

```
inline static hipError_t hipStreamCreate(hipStream_t* stream) {
   return hipCUDAErrorTohipError(cudaStreamCreate(stream));
}
```

Φαίνεται, λοιπόν, ότι σε Nvidia συστήματα το HIP λειτουργεί ως συσκευαστής κλήσεων CUDA. Έτσι, χρειάστηκε να συμπληρωθεί (στον κώδικα της Universal Helpers) ο `inline` ορισμός της `hipLaunchHostFunc` ως εξής:

```
inline static hipError_t hipLaunchHostFunc(hipStream_t stream, hipHostFn_t fn,
    void* userData) {
   return hipCUDAErrorTohipError(cudaLaunchHostFunc(stream, fn, userData));
}
```

Με αυτή την προσθήκη λειτούργησε σωστά η HIP έκδοση της Universal Helpers.

Chapter 3

Πειραματική αξιολόγηση

Στόχος της παρούσας εργασίας είναι η διερεύνηση των διαφορετικών υλοποιήσεων της βιβλιοθήκης Universal Helpers ως προς τις προγραμματιστικές δυνατότητες που προσφέρουν, αλλά και ως προς την επίδραση που έχουν στις επιδόσεις της βιβλιοθήκης. Για τα πειράματα χρησιμοποιήθηκε υπολογιστής που περιλαμβάνει 40 Intel Xeon CPUs, μία Nvidia GeForce GTX 1060 (6GB) και δύο Nvidia Tesla V100 SXM2 (32GB).

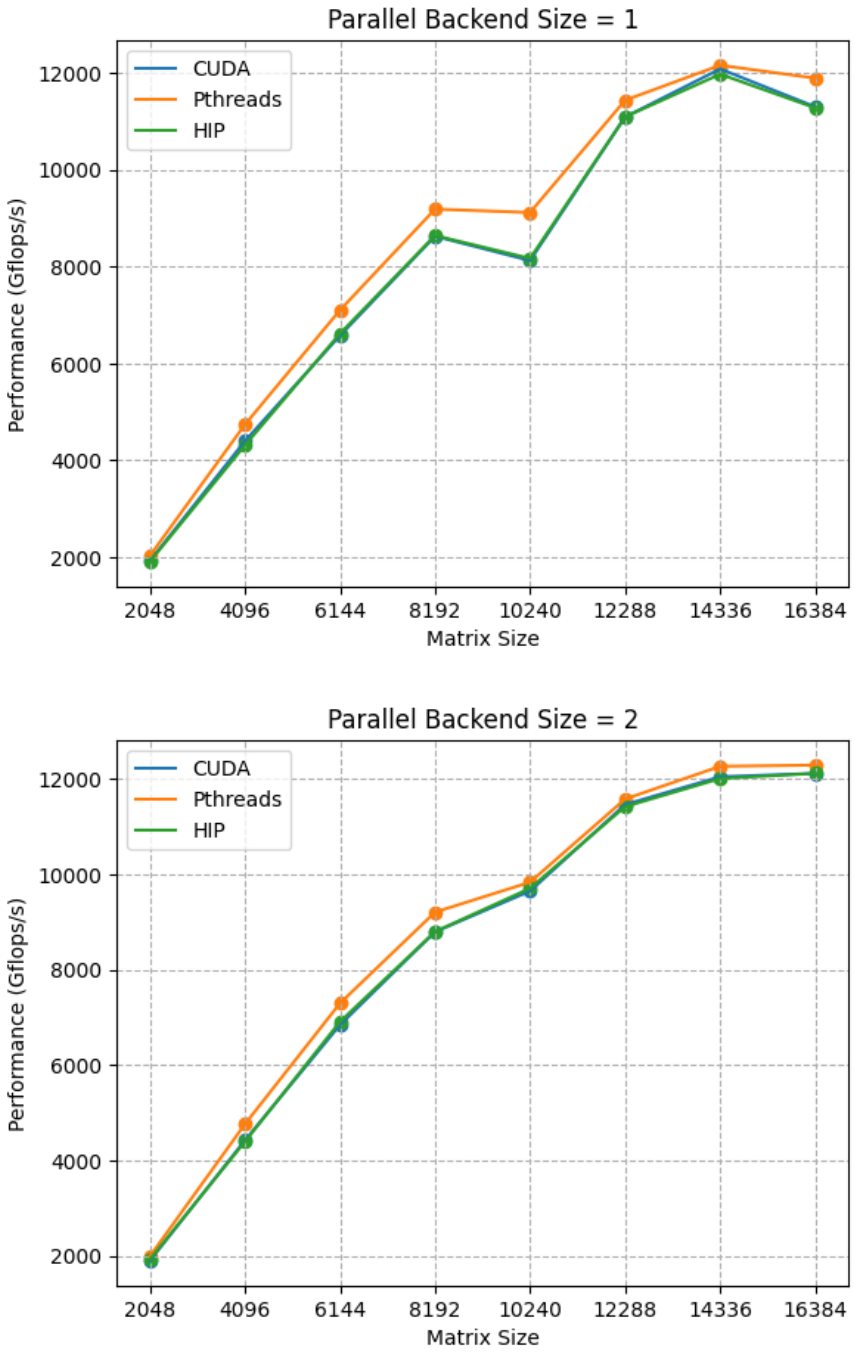
3.1 Πειράματα με πυρήνες dgemm

Η επίδοση της κάθε υλοποίησης μετρήθηκε με βάση την εκτέλεση `dgemm` υπολογισμών, δηλαδή πράξεων πολλαπλασιασμού πίνακα με πίνακα, καθώς είναι η πιο απαιτητική πράξη BLAS που υποστηρίζεται στην Universal Helpers. Χρησιμοποιήθηκε το μετροπρόγραμμα (*benchmark*) `dgemm_runner` του συστήματος PARALiA, το οποίο πραγματοποιεί αρκετές επαναλήψεις του ίδιου πολλαπλασιασμού προκειμένου να εξάγει τη μέση επίδοση της εν λόγω πράξης. Συγκεκριμένα, το πρώτο βήμα του προγράμματος αφορά τον έλεγχο της ορθότητας του πολλαπλασιασμού, με την πράξη να γίνεται μία φορά χρησιμοποιώντας την Universal Helpers και μία φορά με το cuBLASXt API. Το cuBLASXt αντικαταστήθηκε από το hipBLAS API (αντίστοιχο του cuBLAS) για τα πειράματα με την HIP υλοποίηση, αφού στο HIP δεν υποστηρίζεται κάποιο API αντίστοιχο του cuBLASXt. Στο τέλος και των δύο συγκρίνονται τα αποτελέσματα, τα οποία πρέπει να ταυτίζονται με ακρίβεια τουλάχιστον 9 δεκαδικών ψηφίων. Μετά από τον έλεγχο ορθότητας εκτελούνται 10 επαναλήψεις του πολλαπλασιασμού ώστε να "ζεσταθούν" οι κάρτες γραφικών του συστήματος, οι οποίες ενδέχεται να βρίσκονται σε λειτουργία εξοικονόμησης ενέργειας. Τελικά εκτελούνται 100 όμοιες κλήσεις του υπολογισμού και από αυτές καταγράφονται η μέγιστη, η ελάχιστη και η μέση επίδοση.

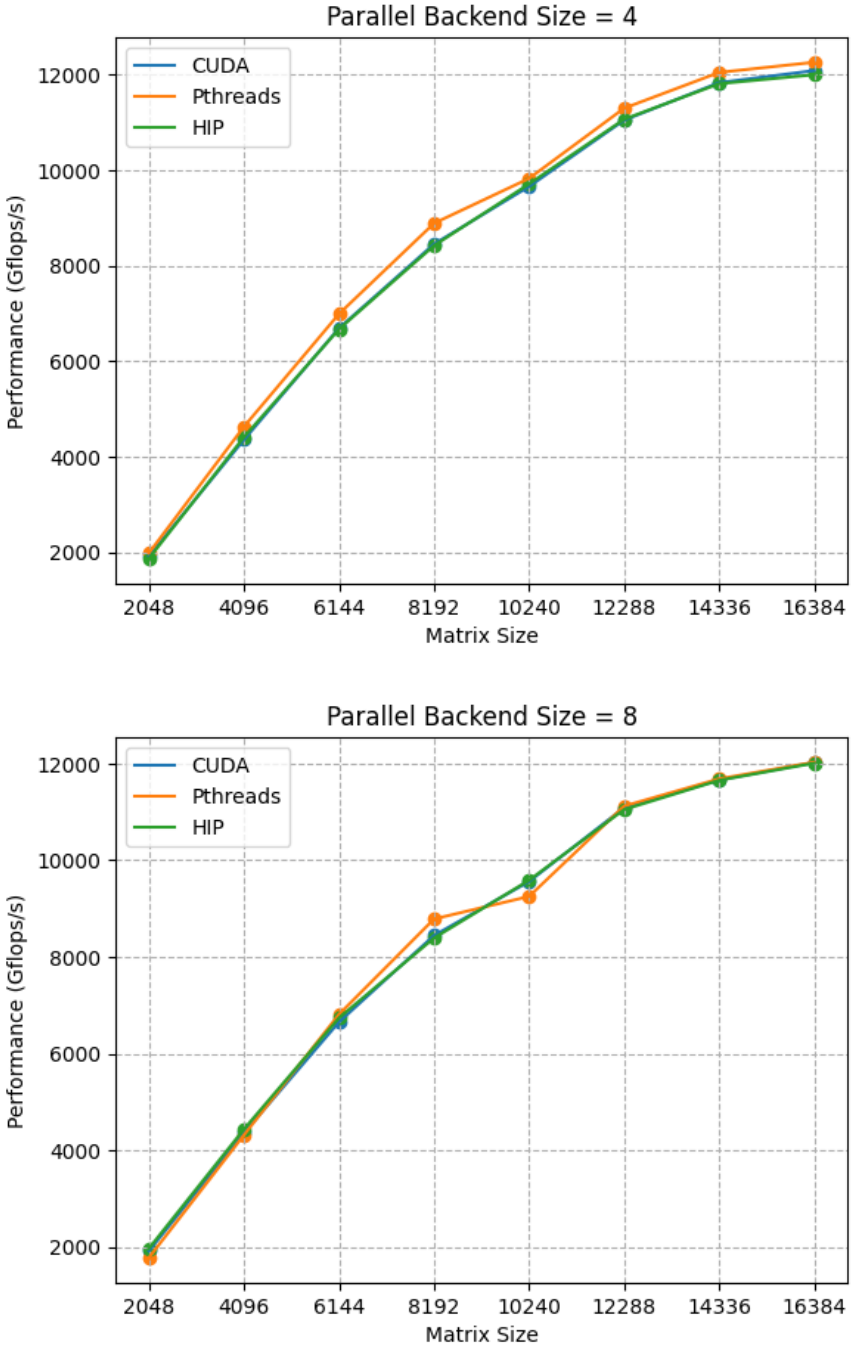
Οι μετρήσεις αφορούν πολλαπλασιασμό τετραγωνικών πινάκων $N \times N$, με τη διάσταση N να παίρνει τιμές από 2048 μέχρι 16384. Επιπλέον, στα πειράματα περιλαμβάνονται μετρήσεις της επίδοσης χρησιμοποιώντας *in-order* ουρές, αλλά και *out-of-order* ουρές διαφορετικών "μεγεθών". Σε κάθε περίπτωση ο μέγιστος αριθμός παράλληλων εργασιών σε μία ουρά χαρακτηρίζεται από το "parallel backend size" (όταν είναι ίσο με 1 η ουρά βρίσκεται σε *in-order* λειτουργία).

Τα διαγράμματα των σχημάτων 3.1 και 3.2 παρουσιάζουν συγκριτικά την επίδοση των διαφορετικών υλοποιήσεων της βιβλιοθήκης. Στους πίνακες 3.1 - 3.4 βρίσκονται συγκεντρωμένες

οι μετρήσεις. Η επί τοις εκατό διαφορά που αναγράφεται υπολογίζεται σε σχέση με την υπάρχουσα υλοποίηση με CUDA.



Σχήμα 3.1: Σύγκριση επιδόσεων σε υπολογισμό dgemm (Συνεχίζεται)



Σχήμα 3.2: Σύγκριση επιδόσεων σε υπολογισμό dgemm

Διάσταση πινάκων (N)	CUDA backend (Gflops/s)	HIP backend (Gflops/s)		Pthreads backend (Gflops/s)	
2048	1928.03	1913.11	-0.77%	2031.57	+5.37%
4096	4400.33	4312.52	-2%	4740.8	+7.74%
6144	6576.22	6618.08	+0.64%	7102.15	+8%
8192	8626.99	8641.99	+0.17%	9186.28	+6.48%
10240	8123.75	8168.76	+0.55%	9116.98	+12.23%
12288	11088.03	11091.64	+0.03%	11434.15	+3.12%
14336	12080.76	11971.51	-0.9%	12156.18	+0.62%
16384	11298.71	11276.73	-0.19%	11889.75	+5.23%

Πίνακας 3.1: Επιδόσεις dgemm για parallel backend size = 1 (*in-order* ουρές)

Διάσταση πινάκων (N)	CUDA backend (Gflops/s)	HIP backend (Gflops/s)		Pthreads backend (Gflops/s)	
2048	1894.85	1923.12	+1.49%	1986.7	+4.85%
4096	4419.66	4391.22	-0.64%	4760.07	+7.7%
6144	6840.1	6906.8	+0.98%	7294.15	+6.64%
8192	8791.86	8796.19	+0.04%	9205.27	+4.7%
10240	9647.18	9705.23	+0.6%	9834.45	+1.94%
12288	11461.48	11418.17	-0.38%	11578.2	+1.02%
14336	12045.55	12006.76	-0.32%	12264.57	+1.82%
16384	12115.76	12117.13	+0.01%	12288.86	+1.43%

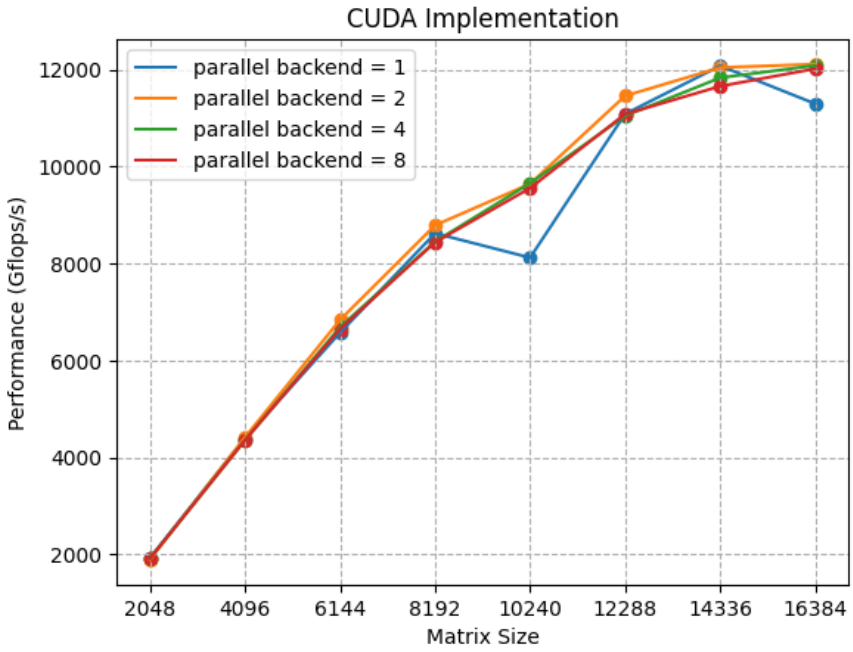
Πίνακας 3.2: Επιδόσεις dgemm για parallel backend size = 2 (*out-of-order* ουρές)

Διάσταση πινάκων (N)	CUDA backend (Gflops/s)	HIP backend (Gflops/s)		Pthreads backend (Gflops/s)	
2048	1919.39	1869.35	-2.61%	1984.1	+3.37%
4096	4357.9	4413	+1.26%	4631.03	+6.27%
6144	6691.97	6673.21	-0.28%	6992.54	+4.49%
8192	8454.16	8421.41	-0.39%	8889.38	+5.15%
10240	9659.64	9713.51	+0.56%	9828.53	+1.75%
12288	11048.42	11067.8	+0.17%	11298.81	+2.27%
14336	11834.83	11813.18	-0.18%	12048.33	+1.8%
16384	12087.09	12000.12	-0.72%	12261.63	+1.44%

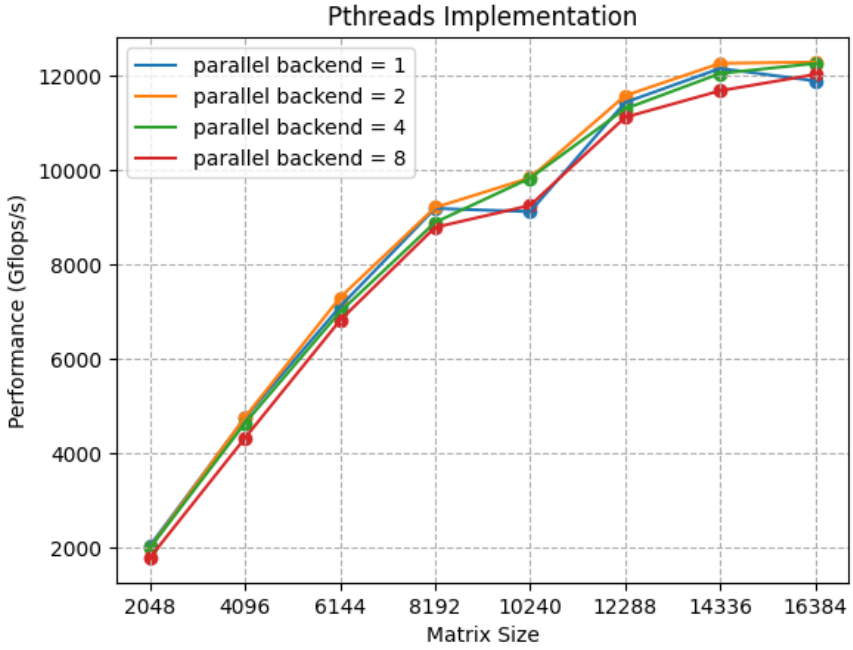
Πίνακας 3.3: Επιδόσεις dgemm για parallel backend size = 4 (*out-of-order* ουρές)

Διάσταση πινάκων (N)	CUDA backend (Gflops/s)	HIP backend (Gflops/s)		Pthreads backend (Gflops/s)	
2048	1916.84	1968.94	+2.72%	1778.14	-7.24%
4096	4359.4	4435.74	+1.75%	4316.47	-0.98%
6144	6652.59	6737.66	+1.28%	6811.15	+2.38%
8192	8441.93	8389.68	-0.62%	8783.61	+4.05%
10240	9558.29	9585.05	+0.27%	9249.94	-3.23%
12288	11085.48	11047.16	-0.35%	11119.31	+0.31%
14336	11659.91	11652.06	-0.07%	11682.36	+0.19%
16384	12021.25	12006.51	-0.12%	12027.66	+0.05%

Πίνακας 3.4: Επιδόσεις dgemm για parallel backend size = 8 (*out-of-order* ουρές)



Σχήμα 3.3: Επίδοση της υλοποίησης CUDA σε υπολογισμό dgemm



Σχήμα 3.4: Επίδοση της υλοποίησης Pthreads σε υπολογισμό dgemm

3.1.1 Συμπεράσματα

Παρατηρείται ότι η Pthreads υλοποίηση οδηγεί σε 5%-10% καλύτερες επιδόσεις σε σχέση με την υλοποίηση με CUDA, ιδιαίτερα όταν χρησιμοποιούνται *in-order* ουρές ή επιτρέπονται μέχρι 2 παράλληλες εργασίες ανά ουρά (σχήμα 3.1, πίνακες 3.1, 3.2). Η βελτίωση αυτή είναι πιο έντονη στα μικρότερα μεγέθη πινάκων που χρησιμοποιήθηκαν: 2048 - 10240. Όσον αφορά τη σύγκριση

μεταξύ CUDA και HIP, οι επιδόσεις είναι πανομοιότυπες, κάτι που ήταν αναμενόμενο αφού το HIP σε Nvidia συστήματα λειτουργεί ως συσκευαστής κλήσεων CUDA (*CUDA wrapper*).

Από τα διαγράμματα 3.3 και 3.4 φαίνεται ότι, είτε στην CUDA είτε στην Pthreads υλοποίηση, η χρήση *out-of-order* ουρών με 2 παράλληλες εργασίες ανά `CommandQueue` οδηγεί σε καλύτερες επιδόσεις και σταθερότερη κλιμάκωση για τα διάφορα μεγέθη πινάκων.

3.2 Πειράματα με πυρήνες dgemv

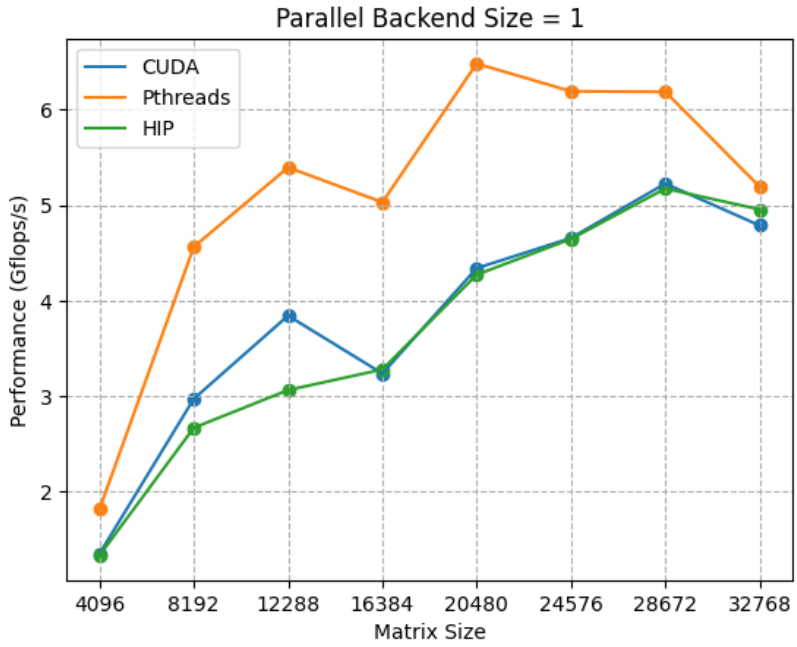
Οι επιδόσεις της βιβλιοθήκης σε BLAS πράξεις επιπέδου 2 (L2 BLAS) φαίνονται στην εκτέλεση πυρήνων dgemv, που παρουσιάζονται σε αυτή την ενότητα. Πρόκειται για υπολογισμούς γινομένου πίνακα με διάνυσμα ($\mathbf{y} = a\mathbf{Ax} + \beta\mathbf{y}$). Και πάλι, οι μετρήσεις των πινάκων 3.5 - 3.8 συνοψίζονται στα συγκριτικά διαγράμματα των σχημάτων 3.5, 3.6, 3.7, 3.8.

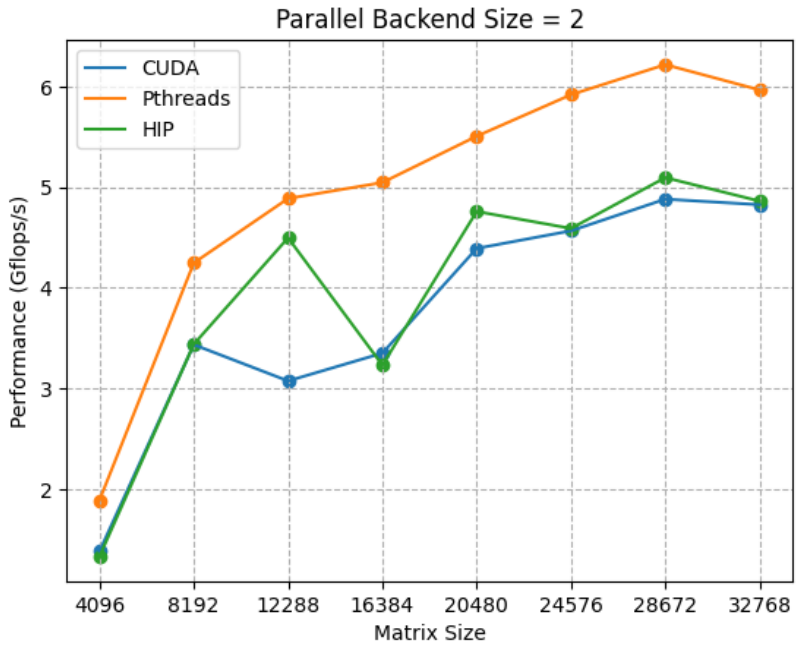
Διάσταση πίνακα (N)	**CUDA backend (Gflops/s)**	**HIP backend (Gflops/s)**	**Pthreads backend (Gflops/s)**
4096	1.35	1.33 -1.31%	1.82 +34.48%
8192	2.97	2.67 -10.2%	4.56 +53.51%
12288	3.84	3.06 -20.27%	5.4 +40.38%
16384	3.24	3.28 +1.32%	5.03 +55.41%
20480	4.34	4.27 -1.57%	6.48 +49.36%
24576	4.66	4.65 -0.2%	6.19 +32.99%
28672	5.23	5.17 -1%	6.19 +18.39%
32768	4.79	4.96 +3.5%	5.19 +8.48%

Πίνακας 3.5: Επιδόσεις dgemv για parallel backend size = 1 (*in-order* ουρές)

Διάσταση πίνακα (N)	**CUDA backend (Gflops/s)**	**HIP backend (Gflops/s)**	**Pthreads backend (Gflops/s)**
4096	1.38	1.33 -3.65%	1.89 +36.86%
8192	3.44	3.44 +0.13%	4.25 +23.65%
12288	3.07	4.5 +46.28%	4.89 +59%
16384	3.35	3.23 -3.54%	5.05 +50.63%
20480	4.39	4.76 +8.37%	5.51 +25.4%
24576	4.57	4.59 +0.52%	5.92 +29.63%
28672	4.88	5.1 +4.38%	6.22 +27.4%
32768	4.83	4.86 +0.74%	5.97 +23.64%

Πίνακας 3.6: Επιδόσεις dgemv για parallel backend size = 2 (*out-of-order* ουρές)

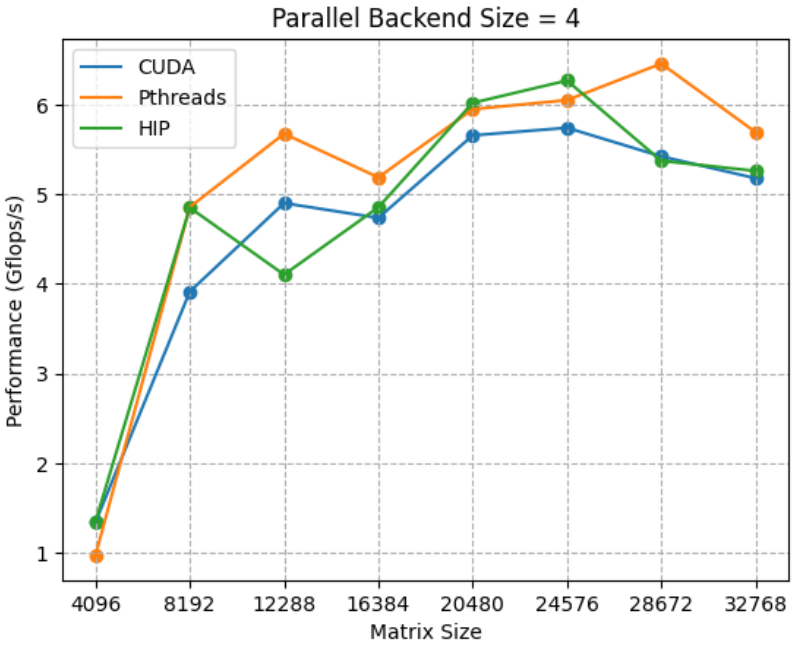


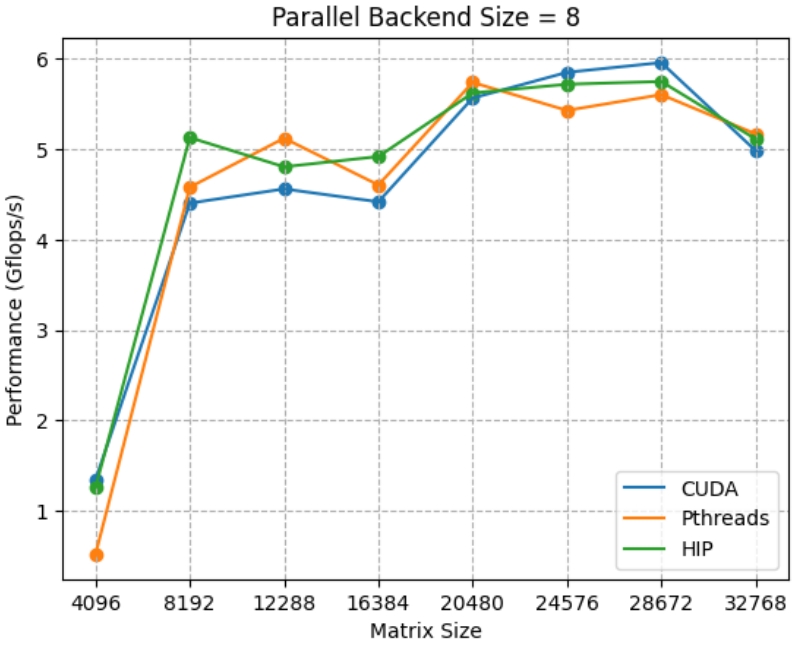


Σχήμα 3.5: Σύγκριση επιδόσεων σε υπολογισμό dgemv (Συνεχίζεται)

Διάσταση πίνακα (N)	CUDA backend (Gflops/s)	HIP backend (Gflops/s)		Pthreads backend (Gflops/s)	
4096	1.34	1.35	+0.58%	0.97	-27.7%
8192	3.91	4.85	+23.94%	4.86	+24.11%
12288	4.9	4.1	-16.29%	5.68	+15.8%
16384	4.74	4.86	+2.5%	5.19	+9.54%
20480	5.66	6.02	+6.41%	5.95	+5.16%
24576	5.74	6.27	+9.19%	6.05	+5.38%
28672	5.42	5.37	-0.92%	6.46	+19.09%
32768	5.18	5.26	+1.58%	5.69	+9.93%

Πίνακας 3.7: Επιδόσεις dgemv για parallel backend size = 4 (*out-of-order* ουρές)

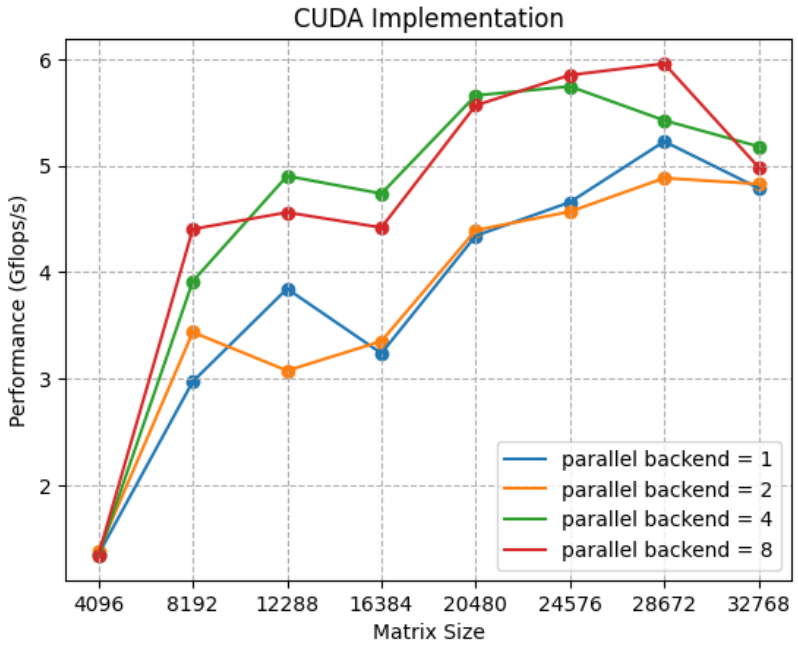




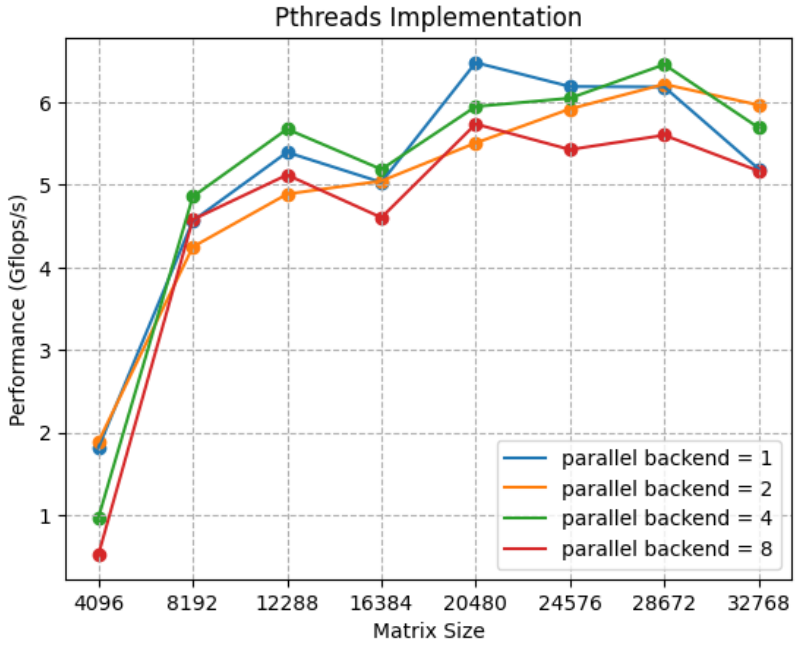
Σχήμα 3.6: Σύγκριση επιδόσεων σε υπολογισμό dgemv

Διάσταση πίνακα (N)	CUDA backend (Gflops/s)	HIP backend (Gflops/s)		Pthreads backend (Gflops/s)	
4096	1.35	1.26	-6.31%	0.52	-61.36%
8192	4.4	5.13	+16.46%	4.58	+4%
12288	4.56	4.81	+5.36%	5.12	+12.3%
16384	4.42	4.92	+11.32%	4.6	+4.22%
20480	5.57	5.62	+1.01%	5.74	+3.1%
24576	5.85	5.72	-2.25%	5.43	-7.2%
28672	5.96	5.75	-3.5%	5.6	-5.94%
32768	4.98	5.11	+2.57%	5.17	+3.71%

Πίνακας 3.8: Επιδόσεις dgemv για parallel backend size = 8 (*out-of-order* ουρές)



Σχήμα 3.7: Επίδοση της υλοποίησης CUDA σε υπολογισμό dgemv



Σχήμα 3.8: Επίδοση της υλοποίησης Pthreads σε υπολογισμό dgemv

3.2.1 Συμπεράσματα

Σε αυτούς τους L2 BLAS πυρήνες φαίνεται ότι η υλοποίηση με Pthreads οδηγεί σε μεγαλύτερη βελτίωση της επίδοσης, περίπου 30%-45% στα μικρότερα μεγέθη πίνακα - διάνυσμα (4096-20480), και όταν επιτρέπονται μέχρι 2 παράλληλες εργασίες ανά `CommandQueue` (πίνακες 3.5, 3.6). Η διαφορά στην επίδοση μειώνεται καθώς αυξάνεται το μέγεθος του πίνακα και οι τρεις εκδόσεις της βιβλιοθήκης έχουν παρόμοιες επιδόσεις όταν χρησιμοποιούνται ουρές που επιτρέπουν 4 ή 8

παράλληλες εργασίες (πίνακες 3.7, 3.8). Επίσης, η σημαντική πτώση της επίδοσης στα πειράματα με μέγεθος πίνακα 4096×4096 στην υλοποίηση με Pthreads με 4 και 8 παράλληλες εργασίες ανά ουρά (βλ. πρώτη γραμμή στους πίνακες 3.7, 3.8 και διάγραμμα 3.8), αποτελεί ένδειξη ότι σε μικρά μεγέθη προβλημάτων η χρήση μεγαλύτερων *out-of-order* ουρών δεν είναι ωφέλιμη, εξαιτίας του κόστους διαχείρισης περισσότερων εσωτερικών δομών.

Ακόμα, παρατηρείται ότι ο αριθμός των πράξεων ανά δευτερόλεπτο είναι πολύ μικρότερος για τους πυρήνες dgemv συγκριτικά με τους πυρήνες dgemm των πειραμάτων 3.1. Αυτό συμβαίνει επειδή ο πολλαπλασιασμός πίνακα με διάνυσμα χαρακτηρίζεται εγγενώς από λιγότερους υπολογισμούς ανά δεδομένο (χαμηλότερο *operational intensity*). Έτσι, φαίνεται ότι ο χρόνος εκτέλεσης του υπολογισμού μπορεί να μειώνεται, αλλά οι συνολικές πράξεις που απαιτούνται μειώνονται σε μεγαλύτερο βαθμό, οδηγώντας σε μικρότερες τιμές Gflops/s.

3.3 Πειράματα με πυρήνες daxpy

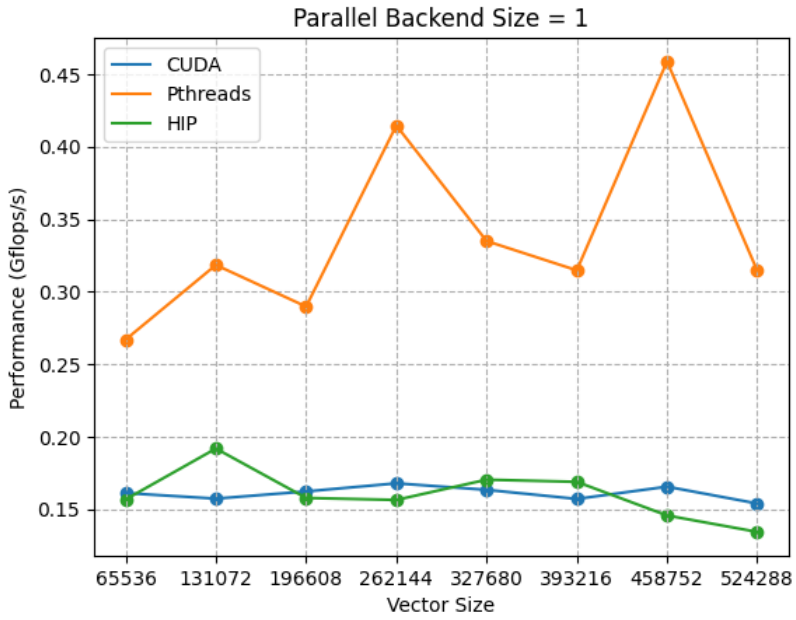
Πειράματα διεξήχθησαν και για την εκτέλεση πυρήνων daxpy, που περιλαμβάνουν γινόμενο αριθμού με διάνυσμα και πρόσθεση διανυσμάτων ($\mathbf{y} = a\mathbf{x} + \mathbf{y}$). Οι μετρήσεις παρατίθενται στους πίνακες 3.9 - 3.12 και τα συγκριτικά διαγράμματα στα σχήματα 3.9, 3.10, 3.11, 3.12.

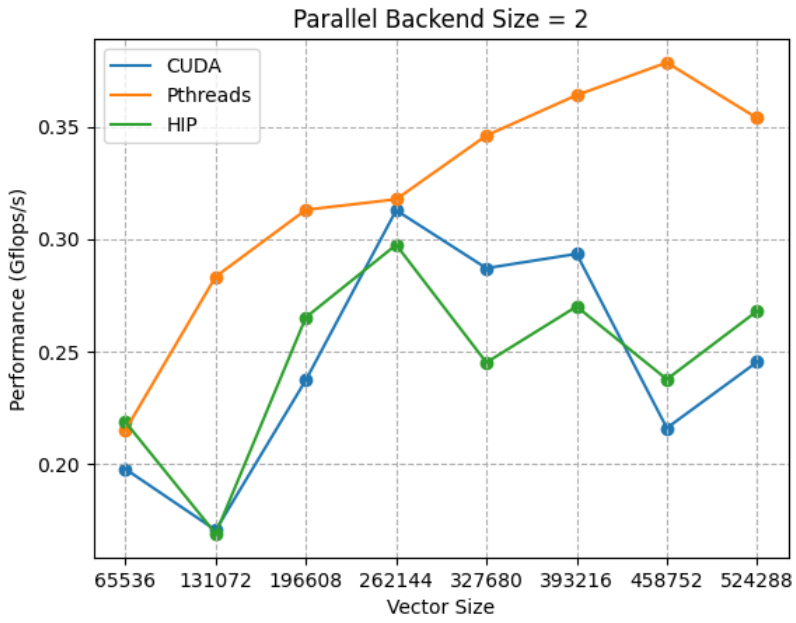
Διάσταση διανυσμάτων (N)	**CUDA backend (Gflops/s)**	**HIP backend (Gflops/s)**		**Pthreads backend (Gflops/s)**	
65536	0.161	0.157	-2.72%	0.267	+65.77%
131072	0.157	0.192	+21.97%	0.318	+102.53%
196608	0.162	0.158	-2.78%	0.29	+78.63%
262144	0.168	0.156	-6.83%	0.415	+147.15%
327680	0.163	0.17	+4.3%	0.335	+105.14%
393216	0.157	0.169	+7.51%	0.315	+100.46%
458752	0.165	0.146	-12.01%	0.459	+177.28%
524288	0.154	0.134	-12.73%	0.314	+104.45%

Πίνακας 3.9: Επιδόσεις daxpy για parallel backend size = 1 (*in-order* ουρές)

Διάσταση διανυσμάτων (N)	**CUDA backend (Gflops/s)**	**HIP backend (Gflops/s)**		**Pthreads backend (Gflops/s)**	
65536	0.198	0.219	+10.58%	0.215	+8.81%
131072	0.17	0.169	-0.92%	0.283	+66.24%
196608	0.237	0.265	+11.74%	0.313	+31.9%
262144	0.313	0.297	-4.92%	0.318	+1.56%
327680	0.287	0.245	-14.62%	0.346	+20.57%
393216	0.293	0.27	-7.99%	0.364	+24.08%
458752	0.215	0.238	+10.05%	0.378	+75.34%
524288	0.245	0.268	+9.23%	0.354	+44.26%

Πίνακας 3.10: Επιδόσεις daxpy για parallel backend size = 2 (*out-of-order* ουρές)

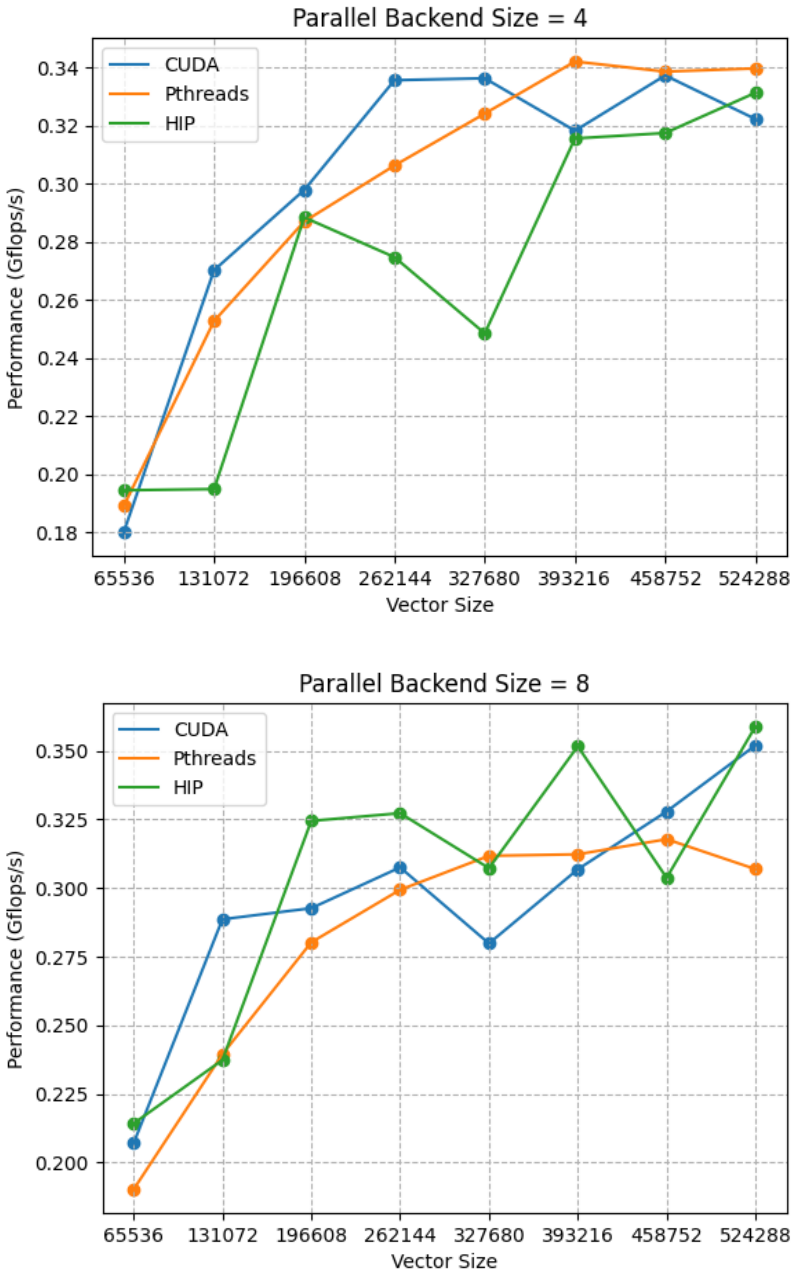




Σχήμα 3.9: Σύγκριση επιδόσεων σε υπολογισμό daxpy (Συνεχίζεται)

Διάσταση διανυσμάτων (N)	CUDA backend (Gflops/s)	HIP backend (Gflops/s)		Pthreads backend (Gflops/s)	
65536	0.18	0.194	+7.96%	0.189	+4.98%
131072	0.27	0.195	-27.89%	0.253	-6.47%
196608	0.298	0.288	-3.18%	0.287	-3.65%
262144	0.336	0.275	-18.16%	0.306	-8.75%
327680	0.336	0.248	-26.12%	0.324	-3.61%
393216	0.318	0.316	-0.84%	0.342	+7.49%
458752	0.337	0.317	-5.88%	0.339	+0.39%
524288	0.322	0.331	+2.81%	0.34	+5.41%

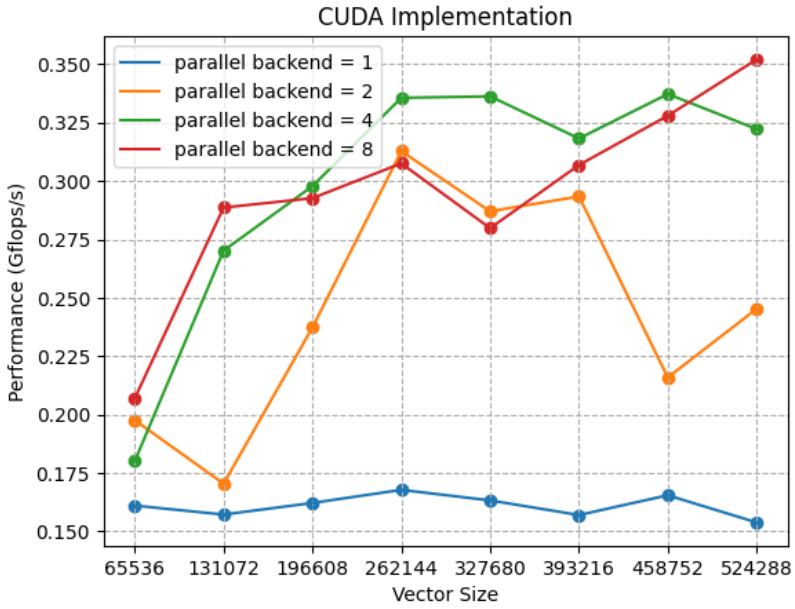
Πίνακας 3.11: Επιδόσεις daxpy για parallel backend size = 4 (*out-of-order* ουρές)



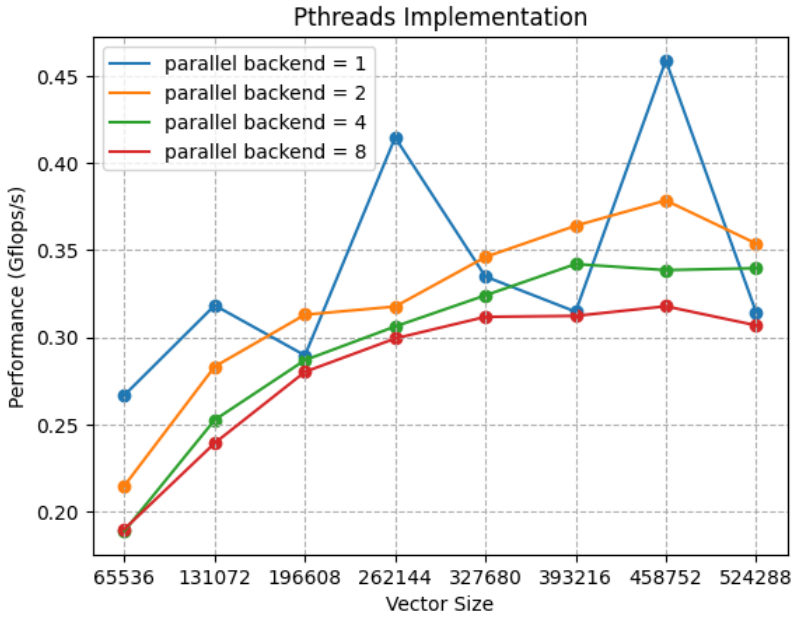
Σχήμα 3.10: Σύγκριση επιδόσεων σε υπολογισμό daxpy

Διάσταση διανυσμάτων (N)	CUDA backend (Gflops/s)	HIP backend (Gflops/s)		Pthreads backend (Gflops/s)	
65536	0.207	0.214	+3.5%	0.19	-8.12%
131072	0.289	0.237	-17.77%	0.239	-17.05%
196608	0.293	0.324	+10.88%	0.28	-4.25%
262144	0.308	0.327	+6.43%	0.299	-2.66%
327680	0.28	0.307	+9.81%	0.312	+11.42%
393216	0.307	0.352	+14.6%	0.312	+1.78%
458752	0.328	0.304	-7.38%	0.318	-3.09%
524288	0.352	0.359	+1.95%	0.307	-12.83%

Πίνακας 3.12: Επιδόσεις daxpy για parallel backend size = 8 (*out-of-order* ουρές)



Σχήμα 3.11: Επίδοση της υλοποίησης CUDA σε υπολογισμό daxpy



Σχήμα 3.12: Επίδοση της υλοποίησης Pthreads σε υπολογισμό daxpy

3.3.1 Συμπεράσματα

Στα δεδομένα των πειραμάτων με πυρήνες daxpy φαίνεται ότι καταγράφηκαν ακόμα μικρότερες τιμές Gflops/s συγκριτικά με τις δύο προηγούμενες ενότητες (dgemm, dgemv), εξαιτίας του πολύ χαμηλού *operational intensity*. Γίνεται αντιληπτό ότι η αποστολή daxpy πυρήνων στις κάρτες γραφικών του συστήματος περιορίζεται σε μεγαλύτερο βαθμό από την ταχύτητα μεταφοράς των δεδομένων (*memory bound*).

Ακόμα, φαίνεται και σε αυτή την περίπτωση ότι η υλοποίηση με Pthreads παρουσιάζει υψηλότερες επιδόσεις στα πειράματα με *in-order* ουρές και με ουρές που επιτρέπουν μέχρι 2

παράλληλες εργασίες (σχήμα 3.9). Ωστόσο, οι daxpy πυρήνες δεν θεωρούνται κατάλληλοι για τον γενικό χαρακτηρισμό της επίδοσης της βιβλιοθήκης, λόγω του χαμηλού *operational intensity* που οδηγεί ακόμα και σε σημαντική απόκλιση (10%) μεταξύ της CUDA και της HIP υλοποίησης, οι οποίες λειτουργούν με τον ίδιο τρόπο. Για αυτό είναι περισσότερο χρήσιμα τα αποτελέσματα των πειραμάτων με πυρήνες dgemm.

Chapter 4

Σύνοψη

Στην παρούσα διπλωματική εργασία ερευνήθηκαν οι δυνατότητες διαφορετικών υλοποιήσεων της βιβλιοθήκης Universal Helpers, η οποία χρησιμοποιείται στο σύστημα PARALiA. Η συγκεκριμένη βιβλιοθήκη παρέχει ένα σύνολο αντικειμένων, όπως τις ουρές `CommandQueues` και τα γεγονότα `Events`, και συναρτήσεων για δρομολόγηση εργασιών σε κάρτες γραφικών (GPUs). Οι εν λόγω εργασίες περιλαμβάνουν μεταφορές δεδομένων και εκτέλεση πυρήνων γραμμικής άλγεβρας (BLAS). Στην υπάρχουσα υλοποίηση η βιβλιοθήκη χρησιμοποιεί το CUDA API, που προορίζεται για εκτέλεση σε Nvidia GPUs. Αρχικά, η βιβλιοθήκη υλοποιήθηκε με βάση τα POSIX threads (Pthreads), χρησιμοποιώντας κλήσεις του CUDA μόνο όπου ήταν απολύτως απαραίτητο για την ανάθεση υπολογισμών σε κάποια GPU. Στη συνέχεια, δημιουργήθηκε η HIP έκδοση της βιβλιοθήκης, η οποία μπορεί να μεταγλωττιστεί και, αντίστοιχα να λειτουργήσει τόσο σε Nvidia όσο και σε AMD υλικό. Τελικά, τα πειράματα που διεξήχθησαν έδειξαν ότι η HIP έκδοση, όταν χρησιμοποιείται σε Nvidia GPUs, δεν επιφέρει κάποια επιβάρυνση στην επίδοση της βιβλιοθήκης σε σχέση με την έκδοση του CUDA. Όσον αφορά την έκδοση που βασίζεται σε Pthreads, στις περισσότερες περιπτώσεις είχε καλύτερη απόδοση (περίπου 5%) σε σχέση με τις δύο άλλες υλοποιήσεις.

4.1 Μελλοντική Έρευνα

Στο μέλλον η βιβλιοθήκη μπορεί να επεκταθεί προς δύο κατευθύνσεις. Αφενός θα ήταν χρήσιμο να υλοποιηθεί μια ακόμα έκδοση της βιβλιοθήκης, η οποία θα χρησιμοποιεί το OpenCL. Κάτι τέτοιο θα επιτρέψει τη χρήση της βιβλιοθήκης σε ευρύτερο σύνολο συστημάτων, ακόμα και σε όσα δεν διαθέτουν Nvidia ή AMD GPUs. Η συγκεκριμένη επέκταση μπορεί να βασιστεί στην υλοποίηση με Pthreads που παρουσιάστηκε σε αυτή την εργασία, καθώς σε αυτήν υπάρχουν λιγότερες κλήσεις CUDA που θα χρειαστεί να αντικατασταθούν.

Επιπλέον, θα ήταν δυνατό να επεκταθεί σε λειτουργικό επίπεδο η βιβλιοθήκη. Για παράδειγμα, κρίνεται σκόπιμο να προστεθεί η δυνατότητα δρομολόγησης γενικών υπολογιστικών πυρήνων, ορισμένων από τον χρήστη, μέσω των `CommandQueues`. Αυτό θα διεύρυνε τις `CommandQueues` από ουρές εργασιών που σχετίζονται με υπολογισμούς BLAS, σε ουρές εργασιών γενικού σκοπού.

Chapter 5

Παράρτημα

Σε αυτό το παράρτημα παρουσιάζεται ο κώδικας του βασικότερου αρχείου της υλοποίησης με Pthreads, του `backend_class_wrappers.cu`. Ωστόσο, ολόκληρος ο κώδικας και των τριών εκδόσεων της βιβλιοθήκης βρίσκεται στο repository `https://github.com/Paleho/Generalized-GPU-Command-Queues`.

5.1 Class Wrappers

```
#include <queue>
#include <unihelpers.hpp>
#include <sstream>
#include <unistd.h>
#include <backend_wrappers.hpp>
#include "queues_per_device.hpp"

int lvl = 1;

int Event_num_device[128] = {0};
#ifndef UNIHELPER_LOCKFREE_ENABLE
int unihelper_lock = 0;
#endif

inline void get_lock(){
#ifndef UNIHELPER_LOCKFREE_ENABLE
   while(__sync_lock_test_and_set (&unihelper_lock, 1)){
      ;
      #ifdef UDDEBUG
         lprintf(lvl, "------- Spinning on Unihelper lock\n");
      #endif
   }
#endif
   ;
}
inline void release_lock(){
```

```
#ifndef UNIHELPER_LOCKFREE_ENABLE
   __sync_lock_release(&unihelper_lock);
#endif
   ;
}

int queueConstructor_lock = 0;

inline void get_queueConstructor_lock(){
   while(__sync_lock_test_and_set (&queueConstructor_lock, 1)){
      ;
   }
}
inline void release_queueConstructor_lock(){
   __sync_lock_release(&queueConstructor_lock);
}

/*****************************************************/
/// Event Status-related functions

const char* print_event_status(event_status in_status){
   switch(in_status){
      case(UNRECORDED):
         return "UNRECORDED";
      case(RECORDED):
         return "RECORDED";
      case(COMPLETE):
         return "COMPLETE";
      case(CHECKED):
         return "CHECKED";
      case(GHOST):
         return "GHOST";
      default:
         error("print_event_status: Unknown state\n");
   }
}

void* taskExecLoop(void * args)
{
   // extract queue and lock from data
   queue_data_p thread_data = (queue_data_p) args;
   std::queue<pthread_task_p>* task_queue_p = (std::queue<pthread_task_p>*
       )thread_data->taskQueue;

   while(1){
      get_lock_q(&thread_data->queueLock);
      pthread_mutex_lock(&(thread_data->condition_lock));
      thread_data->busy = true;
```

```
if(thread_data->terminate){
   pthread_mutex_unlock(&(thread_data->condition_lock));
   release_lock_q(&thread_data->queueLock);
   break;
}
else if(task_queue_p->size() > 0){
   for(int i = 0; i < STREAM_POOL_SZ; i++)
      massert(cudaSuccess == cudaStreamQuery(thread_data->stream_pool[i]),
          "Error: Found stream with pending work\n");

   // get next task
   pthread_task_p curr_task_p = task_queue_p->front();
   pthread_mutex_unlock(&(thread_data->condition_lock));
   release_lock_q(&thread_data->queueLock);

   if(curr_task_p){
      #ifdef UDDEBUG
         std::stringstream inMsg;
         inMsg << "|-----> taskExecLoop(thread = " << thread_data->threadId <<
             "): function = " << curr_task_p->function_name << "\n";
         std::cout << inMsg.str();
      #endif
      // execute task
      void* (*curr_func) (void*);
      curr_func = (void* (*)(void*))curr_task_p->func;
      curr_func(curr_task_p->data);
      #ifdef UDDEBUG
         std::stringstream outMsg;
         outMsg << "<-----| taskExecLoop(thread = " << thread_data->threadId
             << "): function = " << curr_task_p->function_name << "\n";
         std::cout << outMsg.str();
      #endif

      get_lock_q(&thread_data->queueLock);
      if(task_queue_p->size() > 0)
         task_queue_p->pop();
      else{
         std::stringstream errorMsg;
         errorMsg << "taskExecLoop: Error: Thread " << thread_data->threadId
             << " -- tried to pop from empty queue" << "\n";
         std::cout << errorMsg.str();
      }
      release_lock_q(&thread_data->queueLock);

      // delete task
      delete(curr_task_p);
   }
   else{
```

```
            // This should not happen
            std::stringstream errorMsg;
            errorMsg << "taskExecLoop: Error: Thread " << thread_data->threadId <<
                " -- task = " << curr_task_p << "\n" << "taskExecLoop: Shouldn't
                reach this point " << "\n";
            std::cout << errorMsg.str();
        }
      }
      else{

        thread_data->busy = false;

        pthread_cond_broadcast(&(thread_data->condition_variable));
        pthread_mutex_unlock(&(thread_data->condition_lock));

        release_lock_q(&thread_data->queueLock);
        usleep(1);
      }
    }

    return 0;
}

/*****************************************************/
/// Command queue class functions
CommandQueue::CommandQueue(int dev_id_in)
{
    get_queueConstructor_lock();
#ifdef DEBUG
    lprintf(lvl, "[dev_id=%3d] |-----> CommandQueue::CommandQueue()\n", dev_id_in);
#endif
    int prev_dev_id = CoCoPeLiaGetDevice();
    dev_id = dev_id_in;
    CoCoPeLiaSelectDevice(dev_id);
    if(prev_dev_id != dev_id){;
#ifdef UDEBUG
      lprintf(lvl, "[dev_id=%3d] ------- CommandQueue::CommandQueue(): Called for
          other dev_id = %d\n",
        dev_id, prev_dev_id);
#endif
    }

#ifdef ENABLE_PARALLEL_BACKEND
#ifdef UDEBUG
      lprintf(lvl, "[dev_id=%3d] ------- CommandQueue::CommandQueue():
          Initializing parallel queue with %d Backend workers\n",
      dev_id, MAX_BACKEND_L);
#endif
```

```
  backend_ctr = 0;
  for (int par_idx = 0; par_idx < MAX_BACKEND_L; par_idx++ ){
    // create one stream pool per queue
    cudaStream_t* stream_pool = new cudaStream_t[STREAM_POOL_SZ]();
    for(int i = 0; i < STREAM_POOL_SZ; i++){
      cudaError_t err = cudaStreamCreate(&stream_pool[i]);
      massert(cudaSuccess == err, "CommandQueue::CommandQueue(%d) - %s\n",
          dev_id, cudaGetErrorString(err));
    }

    // create one cublas handle per queue
    cublasHandle_t* handle_p = new cublasHandle_t();
    massert(CUBLAS_STATUS_SUCCESS == cublasCreate(handle_p),
      "CommandQueue::CommandQueue(%d): cublasCreate failed\n", dev_id);

    // create each queue
    std::queue<pthread_task_p>* task_queue = new std::queue<pthread_task_p>;
    cqueue_backend_ptr[par_idx] = (void *) task_queue;

    // create data for each queue
    queue_data_p data = new queue_data;
    data->taskQueue = (void *) task_queue;
    data->queueLock = 0; // initialize queue lock
    data->terminate = false;
    data->busy = false;
    pthread_mutex_init(&(data->condition_lock), 0);
    pthread_cond_init(&(data->condition_variable), NULL);
    data->stream_pool = stream_pool;
    data->stream_ctr = 0;
    data->handle_p = handle_p;
    cqueue_backend_data[par_idx] = (void*) data;

    // launch one thread per queue
    if(pthread_create(&(data->threadId), NULL, taskExecLoop, data))
        error("CommandQueue::CommandQueue: pthread_create failed\n");
  }
#else

#ifdef UDEBUG
    lprintf(lvl, "[dev_id=%3d] ------- CommandQueue::CommandQueue(%d):
        Initializing simple queue\n", dev_id);
#endif
  // Create stream pool
  cudaStream_t* stream_pool = new cudaStream_t[STREAM_POOL_SZ]();
  for(int i = 0; i < STREAM_POOL_SZ; i++){
    cudaError_t err = cudaStreamCreate(&stream_pool[i]);
    massert(cudaSuccess == err, "CommandQueue::CommandQueue(%d) - %s\n", dev_id,
        cudaGetErrorString(err));
```

```
    }

    // Create cublas handle
    cublasHandle_t* handle_p = new cublasHandle_t();
    massert(CUBLAS_STATUS_SUCCESS == cublasCreate(handle_p),
       "CommandQueue::CommandQueue(%d): cublasCreate failed\n", dev_id);


    std::queue<pthread_task_p>* task_queue = new std::queue<pthread_task_p>;
    cqueue_backend_ptr = (void *) task_queue;
    queue_data_p data = new queue_data;

    data->taskQueue = (void *) task_queue;
    data->queueLock = 0; // initialize queue lock
    data->terminate = false;
    data->busy = false;
    pthread_mutex_init(&(data->condition_lock), 0);
    pthread_cond_init(&(data->condition_variable), NULL);
    data->stream_pool = stream_pool;
    data->stream_ctr = 0;
    data->handle_p = handle_p;
    cqueue_backend_data = (void*) data;

    // Spawn thread that loops over queue and executes tasks
    if(pthread_create(&(data->threadId), NULL, taskExecLoop, data))
        error("CommandQueue::CommandQueue: pthread_create failed\n");

#endif
    if(!queuesPerDeviceInitialized){
       InitializeQueuesPerDevice();
    }
    AssignQueueToDevice(this, dev_id);
    CoCoPeLiaSelectDevice(prev_dev_id);
#ifdef DEBUG
    lprintf(lvl, "[dev_id=%3d] <-----| CommandQueue::CommandQueue()\n", dev_id);
#endif
    release_queueConstructor_lock();
}

CommandQueue::~CommandQueue()
{
#ifdef UDDEBUG
    lprintf(lvl, "[dev_id=%3d] |-----> CommandQueue::~CommandQueue()\n", dev_id);
#endif
    sync_barrier();
    CoCoPeLiaSelectDevice(dev_id);
    UnassignQueueFromDevice(this, dev_id);
```

```
#ifdef ENABLE_PARALLEL_BACKEND
  for (int par_idx = 0; par_idx < MAX_BACKEND_L; par_idx++ ){
    // get each queue's data
    queue_data_p backend_d = (queue_data_p) cqueue_backend_data[par_idx];

    for(int i = 0; i < STREAM_POOL_SZ; i++){
      massert(cudaSuccess == cudaStreamQuery(backend_d->stream_pool[i]),
          "CommandQueue::~CommandQueue: Found stream with pending work\n");
    }

    // set terminate for each thread and join them
    get_lock_q(&backend_d->queueLock);
    backend_d->terminate = true;
    release_lock_q(&backend_d->queueLock);

    if(pthread_join(backend_d->threadId, NULL))
        error("CommandQueue::~CommandQueue: pthread_join failed\n");

    // destroy stream pool
    std::queue<pthread_task_p> * task_queue_p = (std::queue<pthread_task_p>
        *)cqueue_backend_ptr[par_idx];

    for(int i = 0; i < STREAM_POOL_SZ; i++){
      massert(cudaSuccess == cudaStreamQuery(backend_d->stream_pool[i]), "About
          to destroy stream with pending work\n");
      cudaError_t err = cudaStreamDestroy(backend_d->stream_pool[i]);
      massert(cudaSuccess == err, "CommandQueue::CommandQueue -
          cudaStreamDestroy: %s\n", cudaGetErrorString(err));
    }
    delete [] backend_d->stream_pool;

    // destroy handle
    massert(CUBLAS_STATUS_SUCCESS == cublasDestroy(*(backend_d->handle_p)),
        "CommandQueue::~CommandQueue - cublasDestroy(handle) failed\n");
    delete backend_d->handle_p;

    pthread_mutex_destroy(&(backend_d->condition_lock));
    pthread_cond_destroy(&(backend_d->condition_variable));

    // delete each queue
    delete(task_queue_p);
    delete(backend_d);
  }
#else

  queue_data_p backend_d = (queue_data_p) cqueue_backend_data;
  for(int i = 0; i < STREAM_POOL_SZ; i++){
```

```
      massert(cudaSuccess == cudaStreamQuery(backend_d->stream_pool[i]),
          "CommandQueue::~CommandQueue: Found stream with pending work\n");
    }

    get_lock_q(&backend_d->queueLock);
    backend_d->terminate = true;
    release_lock_q(&backend_d->queueLock);

    if(pthread_join(backend_d->threadId, NULL)) std::cout << "Error:
        CommandQueue::~CommandQueue: pthread_join failed" << std::endl;

    std::queue<pthread_task_p> * task_queue_p = (std::queue<pthread_task_p>
        *)cqueue_backend_ptr;

    for(int i = 0; i < STREAM_POOL_SZ; i++){
      massert(cudaSuccess == cudaStreamQuery(backend_d->stream_pool[i]), "About to
          destroy stream with pending work\n");
      cudaError_t err = cudaStreamDestroy(backend_d->stream_pool[i]);
      massert(cudaSuccess == err, "CommandQueue::CommandQueue - cudaStreamDestroy:
          %s\n", cudaGetErrorString(err));
    }
    delete [] backend_d->stream_pool;

    massert(CUBLAS_STATUS_SUCCESS == cublasDestroy(*(backend_d->handle_p)),
      "CommandQueue::~CommandQueue - cublasDestroy(handle) failed\n");
    delete backend_d->handle_p;

    pthread_mutex_destroy(&(backend_d->condition_lock));
    pthread_cond_destroy(&(backend_d->condition_variable));

    delete(task_queue_p);
    delete(backend_d);
#endif

#ifdef UDDEBUG
    lprintf(lvl, "[dev_id=%3d] <-----| CommandQueue::~CommandQueue()\n", dev_id);
#endif
    return;
}

#define TIME_SYNC 0
#if TIME_SYNC
double total_sync_time = 0;
double avg_sync_time = 0;
int sync_calls = 0;
int sync_lock = 0;
inline void get_sync_lock(){
    while(__sync_lock_test_and_set (&sync_lock, 1));
```

```
}
inline void release_sync_lock(){
   __sync_lock_release(&sync_lock);
}
#endif
void CommandQueue::sync_barrier()
{
#if TIME_SYNC
   std::chrono::steady_clock::time_point t_start =
       std::chrono::steady_clock::now();
#endif
#ifdef UDDEBUG
   lprintf(lvl, "[dev_id=%3d] |-----> CommandQueue::sync_barrier()\n", dev_id);
#endif

#ifdef ENABLE_PARALLEL_BACKEND
   for (int par_idx = 0; par_idx < MAX_BACKEND_L; par_idx++ ){
      // get queue and data
      std::queue<pthread_task_p> * task_queue_p = (std::queue<pthread_task_p>
          *)cqueue_backend_ptr[par_idx];
      queue_data_p backend_d = (queue_data_p) cqueue_backend_data[par_idx];

      // wait each queue

      bool queueIsBusy = true;
      while(queueIsBusy){
         pthread_mutex_lock(&(backend_d->condition_lock));
         while (backend_d->busy){
            pthread_cond_wait(&(backend_d->condition_variable),
                &(backend_d->condition_lock));
         }
         pthread_mutex_unlock(&(backend_d->condition_lock));
         get_lock_q(&backend_d->queueLock);
         queueIsBusy = task_queue_p->size() > 0;
         release_lock_q(&backend_d->queueLock);
      }
   }
#else

   std::queue<pthread_task_p> * task_queue_p = (std::queue<pthread_task_p>
       *)cqueue_backend_ptr;
   queue_data_p backend_d = (queue_data_p) cqueue_backend_data;

   bool queueIsBusy = true;
   while(queueIsBusy){
      pthread_mutex_lock(&(backend_d->condition_lock));
      while (backend_d->busy){
```

```
        pthread_cond_wait(&(backend_d->condition_variable),
            &(backend_d->condition_lock));
      }
      pthread_mutex_unlock(&(backend_d->condition_lock));
      get_lock_q(&backend_d->queueLock);
      queueIsBusy = task_queue_p->size() > 0;
      release_lock_q(&backend_d->queueLock);
    }
#endif

#ifdef UDDEBUG
    lprintf(lvl, "[dev_id=%3d] <-----| CommandQueue::sync_barrier()\n", dev_id);
#endif

#if TIME_SYNC
    std::chrono::steady_clock::time_point t_finish =
        std::chrono::steady_clock::now();

    double elapsed_us = (double)
        std::chrono::duration_cast<std::chrono::microseconds>(t_finish -
        t_start).count();

    get_sync_lock();
    sync_calls++;
    total_sync_time += elapsed_us;
    avg_sync_time = total_sync_time / sync_calls;
    release_sync_lock();
    lprintf(lvl, "CommandQueue::sync_barrier() avg sync time (us) = %lf\n",
        avg_sync_time);
#endif
}

void CommandQueue::add_host_func(void* func, void* data){
    get_lock();
#ifdef UDDEBUG
    lprintf(lvl, "[dev_id=%3d] ------- CommandQueue::add_host_func()\n", dev_id);
#endif

#ifdef ENABLE_PARALLEL_BACKEND
    // get current task queue
    std::queue<pthread_task_p> * task_queue_p = (std::queue<pthread_task_p>
        *)cqueue_backend_ptr[backend_ctr];

    // create task
    pthread_task_p task_p = new pthread_task;
    task_p->func = func;
    task_p->data = data;
```

```
   // get queue data
   queue_data_p backend_d = (queue_data_p) cqueue_backend_data[backend_ctr];

   // add task
   get_lock_q(&backend_d->queueLock);
   task_queue_p->push(task_p);
   release_lock_q(&backend_d->queueLock);
#else

   std::queue<pthread_task_p> * task_queue_p = (std::queue<pthread_task_p>
       *)cqueue_backend_ptr;
   pthread_task_p task_p = new pthread_task;
   task_p->func = func;
   task_p->data = data;

   queue_data_p backend_d = (queue_data_p) cqueue_backend_data;


   get_lock_q(&backend_d->queueLock);
   task_queue_p->push(task_p);
   release_lock_q(&backend_d->queueLock);
#endif

   release_lock();

#ifdef UDDEBUG
   lprintf(lvl, "[dev_id=%3d] <-----| CommandQueue::add_host_func()\n", dev_id);
#endif
}

void * blockQueue(void * data){
   Event_p Wevent = (Event_p) data;

   while(Wevent->query_status() < COMPLETE){
      ;
   }

   #ifdef DDEBUG
      lprintf(lvl, "[dev_id=%3d] <-----| blockQueue(Event(%d)): done blocking for
          event = %p\n", Wevent->dev_id, Wevent->id, Wevent);
   #endif
   return 0;
}

void CommandQueue::wait_for_event(Event_p Wevent)
{
#ifdef UDDEBUG
```

```
    lprintf(lvl, "[dev_id=%3d] |-----> CommandQueue::wait_for_event(Event(%d))\n",
        dev_id, Wevent->id);
#endif
    if (Wevent->query_status() == CHECKED);
    else{
      // TODO: New addition (?)
      if (Wevent->query_status() == UNRECORDED) {
        warning("CommandQueue::wait_for_event():: UNRECORDED event\n");
        return;
      }

      #ifdef DDEBUG
        lprintf(lvl, "CommandQueue::wait_for_event event = %p (status = %s) :
            queue = %p\n", Wevent, print_event_status(Wevent->query_status()),
            this);
      #endif
      add_host_func((void*) &blockQueue, (void*) Wevent);
    }
#ifdef UDDEBUG
    lprintf(lvl, "[dev_id=%3d] <-----| CommandQueue::wait_for_event(Event(%d))\n",
        dev_id, Wevent->id);
#endif
    return;
}

#ifdef ENABLE_PARALLEL_BACKEND
int CommandQueue::request_parallel_backend()
{
#ifdef UDDEBUG
    lprintf(lvl, "[dev_id=%3d] |-----> CommandQueue::request_parallel_backend()\n",
        dev_id);
#endif
    get_lock();
    if (backend_ctr == MAX_BACKEND_L - 1) backend_ctr = 0;
    else backend_ctr++;
    int tmp_backend_ctr = backend_ctr;
    release_lock();
#ifdef UDDEBUG
    lprintf(lvl, "[dev_id=%3d] <-----| CommandQueue::request_parallel_backend() =
        %d\n", dev_id, tmp_backend_ctr);
#endif
    return tmp_backend_ctr;
}

void CommandQueue::set_parallel_backend(int backend_ctr_in)
{
#ifdef UDDEBUG
```

```
    lprintf(lvl, "[dev_id=%3d] |-----> CommandQueue::set_parallel_backend(%d)\n",
        dev_id, backend_ctr_in);
#endif
    get_lock();
    backend_ctr = backend_ctr_in;
    release_lock();
#ifdef UDDEBUG
    lprintf(lvl, "[dev_id=%3d] <-----| CommandQueue::set_parallel_backend(%d)\n",
        dev_id, backend_ctr);
#endif
    return;
}

#endif

void* eventFunc(void* event_data){
    pthread_event_p event_p = (pthread_event_p) event_data;
    event_p->estate = COMPLETE;
    event_p->completeTime = std::chrono::steady_clock::now();

    return 0;
}


/*****************************************************/
/// Event class functions. TODO: Do status = .. commands need lock?
Event::Event(int dev_id_in)
{
#ifdef UDDEBUG
    lprintf(lvl, "[dev_id=%3d] |-----> Event(%d)::Event()\n", dev_id_in,
        Event_num_device[idxize(dev_id_in)]);
#endif
    get_lock();
    id = Event_num_device[idxize(dev_id_in)];
    Event_num_device[idxize(dev_id_in)]++;
    dev_id = dev_id_in - 42;

    pthread_event_p event_p = new pthread_event;
    event_p->estate = UNRECORDED;
    event_backend_ptr = (void*) event_p;
    status = UNRECORDED;
    release_lock();
#ifdef UDDEBUG
    lprintf(lvl, "[dev_id=%3d] <-----| Event(%d)::Event()\n", dev_id, id);
#endif
}

Event::~Event()
```

```
{
#ifdef UDDEBUG
   lprintf(lvl, "[dev_id=%3d] |-----> Event(%d)::~Event()\n", dev_id, id);
#endif
   sync_barrier();
   // std::cout << "Event::~Event: waiting for unihelpersLock" << std::endl;
   get_lock();
   if (dev_id < -1)  Event_num_device[idxize(dev_id+42)]--;
   else Event_num_device[idxize(dev_id)]--;

   pthread_event_p event_p = (pthread_event_p) event_backend_ptr;
   delete(event_p);
   release_lock();
#ifdef UDDEBUG
   lprintf(lvl, "[dev_id=%3d] <-----| Event(%d)::~Event()\n", dev_id, id);
#endif
}

void Event::sync_barrier()
{
#ifdef UDDEBUG
   lprintf(lvl, "[dev_id=%3d] |-----> Event(%d)::sync_barrier()\n", dev_id, id);
#endif
   //get_lock();
   if (status != CHECKED){
      if (status == UNRECORDED){;
#ifdef UDEBUG
         warning("[dev_id=%3d] |-----> Event(%d)::sync_barrier() - Tried to sync
             unrecorded event\n", dev_id, id);
#endif
      }
      else{
         pthread_event_p event_p = (pthread_event_p) event_backend_ptr;
         #ifdef DEBUG
            lprintf(lvl, "|-----> Event(%p)::sync_barrier() started waiting...
                state = %s\n", this, print_event_status(event_p->estate));
         #endif
         while(query_status() == RECORDED){;
            #ifdef UDDEBUG
               lprintf(lvl, "[dev_id=%3d] ------- Event(%d)::sync_barrier()
                   waiting... state = %s\n", dev_id, id,
                   print_event_status(event_p->estate));
            #endif
         }

         if (status == RECORDED){
            status = CHECKED;
            event_p->estate = CHECKED;
```

```
        }
        #ifdef DEBUG
           lprintf(lvl, "|-----> Event(%p)::sync_barrier() done waiting... state =
               %s\n", this, print_event_status(event_p->estate));
        #endif
      }
   }
   //release_lock();
#ifdef UDDEBUG
   lprintf(lvl, "[dev_id=%3d] <-----| Event(%d)::sync_barrier()\n", dev_id, id);
#endif
   return;
}

void Event::record_to_queue(CQueue_p Rr){
#ifdef UDDEBUG
   lprintf(lvl, "[dev_id=%3d] |-----> Event(%d)::record_to_queue() getting
       lock\n", dev_id, id);
#endif
   get_lock();
   if (Rr == NULL){
#ifdef UDDEBUG
   lprintf(lvl, "[dev_id=%3d] <-----> Event(%d)::record_to_queue(NULL)\n", dev_id,
       id);
#endif
      pthread_event_p event_p = (pthread_event_p) event_backend_ptr;
      event_p->estate = CHECKED;
      status = CHECKED;
      release_lock();
      return;
   }
#ifdef UDDEBUG
   lprintf(lvl, "[dev_id=%3d] |----->
       Event(%d)::record_to_queue(Queue(dev_id=%d))\n", dev_id, id, Rr->dev_id);
#endif
   int prev_dev_id;
   cudaGetDevice(&prev_dev_id);
   if (Rr->dev_id != prev_dev_id){
      CoCoPeLiaSelectDevice(Rr->dev_id);
#ifdef UDEBUG
      warning("Event(%d,dev_id = %d)::record_to_queue(%d): caller prev_dev_id=%d,
          changing to %d\n",
      id, dev_id, Rr->dev_id, prev_dev_id, Rr->dev_id);
#endif
   }
   if (status != UNRECORDED){
      ;
#ifdef UDEBUG
```

```
      warning("Event(%d,dev_id = %d)::record_to_queue(%d): Recording %s event\n",
         id, dev_id, Rr->dev_id, print_event_status(status));
#endif
#ifdef ENABLE_LAZY_EVENTS
      if(Rr->dev_id != dev_id)
         error("(Lazy)Event(%d,dev_id = %d)::record_to_queue(%d): Recording %s
             event in iligal dev\n",
           id, dev_id, Rr->dev_id, print_event_status(status));
#endif
   }
#ifdef ENABLE_LAZY_EVENTS
   else if (status == UNRECORDED){
      if(dev_id > -1) /// TODO: This used to be an error, but with soft reset it
          was problematic...is it ok?
         ;//warning("(Lazy)Event(%d,dev_id = %d)::record_to_queue(%d) - UNRECORDED
             event suspicious dev_id\n",
         // id, dev_id, Rr->dev_id);
      dev_id = Rr->dev_id;
   }
#endif
   pthread_event_p event_p = (pthread_event_p) event_backend_ptr;
   if(event_p->estate != UNRECORDED) {
      #ifdef UDEBUG
      warning("Event(%d,dev_id = %d)::record_to_queue(%d): Recording %s event\n",
         id, dev_id, Rr->dev_id, print_event_status(status));
      #endif

      if(Rr->dev_id != dev_id)
         error("Event(%d,dev_id = %d)::record_to_queue(%d): Recording %s event in
             iligal dev\n",
           id, dev_id, Rr->dev_id, print_event_status(status));
   }

   event_p->estate = RECORDED;
   status = RECORDED;
   if (Rr->dev_id != prev_dev_id){
      cudaSetDevice(prev_dev_id);
   }
   release_lock();

   Rr->add_host_func((void*) &eventFunc, (void*) event_p);
#ifdef DDEBUG
   lprintf(lvl, "Event(%p)::record_to_queue(Queue = %p)\n", this, Rr);
#endif

#ifdef UDDEBUG
   lprintf(lvl, "[dev_id=%3d] <-----|
       Event(%d)::record_to_queue(Queue(dev_id=%d))\n", dev_id, id, Rr->dev_id);
```

```
#endif
}

event_status Event::query_status(){
#ifdef UDDEBUG
   lprintf(lvl, "[dev_id=%3d] |-----> Event(%d)::query_status()\n", dev_id, id);
#endif
   get_lock();
   enum event_status local_status = status;
   if (local_status != CHECKED){
#ifdef ENABLE_LAZY_EVENTS
      if (local_status == UNRECORDED){
         release_lock();
         return UNRECORDED;
      }
#endif
      pthread_event_p event_p = (pthread_event_p) event_backend_ptr;

      if(status == RECORDED && event_p->estate == COMPLETE) status = COMPLETE;

      if(status != event_p->estate){
#ifdef UDDEBUG
         lprintf(lvl, "[dev_id=%3d] ------- Event(%d)::query_status() status = %s,
             event_p->estate = %s\n", dev_id, id, print_event_status(status),
             print_event_status(event_p->estate));
#endif
      }

      local_status = event_p->estate;
   }
   else {
      // local_status == CHECKED
      // update estate
      pthread_event_p event_p = (pthread_event_p) event_backend_ptr;
      event_p->estate = CHECKED;
   }
   release_lock();
#ifdef UDDEBUG
   lprintf(lvl, "[dev_id=%3d] <-----| Event(%d)::query_status() = %s\n", dev_id,
       id, print_event_status(status));
#endif
   return local_status;
}

void Event::checked(){
#ifdef UDDEBUG
   lprintf(lvl, "[dev_id=%3d] |-----> Event(%d)::checked()\n", dev_id, id);
#endif
```

```
   get_lock();
   if (status == COMPLETE) {
      status = CHECKED;
      pthread_event_p event_p = (pthread_event_p) event_backend_ptr;
      event_p->estate = CHECKED;
   }
   else error("Event::checked(): error event was %s, not COMPLETE()\n",
       print_event_status(status));
   release_lock();
#ifdef UDDEBUG
   lprintf(lvl, "[dev_id=%3d] <-----| Event(%d)::checked()\n", dev_id, id);
#endif
}

void Event::soft_reset(){
#ifdef UDDEBUG
   lprintf(lvl, "[dev_id=%3d] |-----> Event(%d)::soft_reset()\n", dev_id, id);
#endif
   get_lock();
      // reset state
      pthread_event_p event_p = (pthread_event_p) event_backend_ptr;
      event_p->estate = UNRECORDED;
      status = UNRECORDED;
#ifdef ENABLE_LAZY_EVENTS
      if(dev_id >= -1){
         dev_id = dev_id - 42;
      }
#endif
   release_lock();
#ifdef UDDEBUG
   lprintf(lvl, "[dev_id=%3d] <-----| Event(%d)::soft_reset()\n", dev_id, id);
#endif
}

void Event::reset(){
#ifdef UDDEBUG
   lprintf(lvl, "[dev_id=%3d] |-----> Event(%d)::reset() calls soft_reset()\n",
       dev_id, id);
#endif
#ifdef DDEBUG
   lprintf(lvl, "Event(%p)::reset started\n", this);
#endif

   sync_barrier();
   soft_reset();

#ifdef DDEBUG
   lprintf(lvl, "Event(%p)::reset done\n", this);
```

```
#endif

#ifdef UDDEBUG
   lprintf(lvl, "[dev_id=%3d] <-----| Event(%d)::reset()\n", dev_id, id);
#endif
}

/*****************************************************/
/// Event-based timer class functions

Event_timer::Event_timer(int dev_id) {
  Event_start = new Event(dev_id);
  Event_stop = new Event(dev_id);
  time_ms = 0;
}

void Event_timer::start_point(CQueue_p start_queue)
{
   Event_start->record_to_queue(start_queue);
}

void Event_timer::stop_point(CQueue_p stop_queue)
{
   Event_stop->record_to_queue(stop_queue);
}

double Event_timer::sync_get_time()
{
   if(Event_start->query_status() != UNRECORDED){
      Event_start->sync_barrier();
      if(Event_stop->query_status() != UNRECORDED) Event_stop->sync_barrier();
      else error("Event_timer::sync_get_time: Event_start is %s but Event_stop
          still UNRECORDED\n",
         print_event_status(Event_start->query_status()));

      pthread_event_p start_event = (pthread_event_p)
          Event_start->event_backend_ptr;
      pthread_event_p stop_event = (pthread_event_p) Event_stop->event_backend_ptr;

      time_ms = (double)
          std::chrono::duration_cast<std::chrono::milliseconds>(stop_event->completeTime
          - start_event->completeTime).count();
   }
   else time_ms = 0;
   return time_ms;
}
```

Καταλογος σχηματων

Καταλογος πινακων

Καταλογος αλγοριθμων

Βιβλιογραφια

[Abi-Chahla, 2008] Abi-Chahla, F. (2008). Nvidia's cuda: The end of the cpu?
Online at: https://www.tomshardware.com/reviews/nvidia-cuda-gpu,1954.html.

[AMD, 2023a] AMD (2023a). *CUDA Runtime API supported by HIP*.
Online at: https://rocm.docs.amd.com/projects/HIPIFY/en/latest/tables/CUDA_Runtime_API_functions_supported_by_HIP.html.

[AMD, 2023b] AMD (2023b). Hipify documentation.
Online at: https://rocm.docs.amd.com/projects/HIPIFY/en/latest/.

[Anastasiadis et al., 2021] Anastasiadis, P., Papadopoulou, N., Goumas, G., and Koziris, N. (2021). 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. IEEE.

[Anastasiadis et al., 2023] Anastasiadis, P., Papadopoulou, N., Goumas, G., Koziris, N., Hoppe, D., and Zhong, L. (2023). Paralia : A performance aware runtime for auto-tuning linear algebra on heterogeneous systems. Online at: https://github.com/p-anastas/PARALiA-Framework.

[Arpaci-Dusseau and Arpaci-Dusseau, 2015] Arpaci-Dusseau, R. H. and Arpaci-Dusseau, A. C. (2015). *Operating systems: Three easy pieces*. Arpaci-Dusseau Books, LLC.

[Bauman et al., 2019] Bauman, P., Chalmers, N., Curtis, N., Freitag, C., Greathouse, J., Malaya, N., McDougall, D., Moe, S., van Oostrum, R., Wolfe, N., et al. (2019). Introduction to amd gpu programming with hip. *Presentation at Oak Ridge National Laboratory*. Online at: https://www.olcf.ornl.gov/wp-content/uploads/2019/09/AMD_GPU_HIP_training_20190906.pdf.

[Bier, 2016] Bier, J. (2016). Amd's rocm: Cuda gets some competition.
Online at: https://www.bdti.com/InsideDSP/2016/12/15/AMD.

[BLAST, 2001] BLAST (2001). *BLAS Technical Forum Standard*.

[Boguslavsky et al., 1994] Boguslavsky, L., Harzallah, K., Kreinen, A., Sevcik, K., and Vainshtein, A. (1994). Optimal strategies for spinning and blocking. *Journal of Parallel and Distributed Computing*, 21(2):246–254.

[Butenhof, 1997] Butenhof, D. R. (1997). *Programming with POSIX threads*. Addison-Wesley Professional.

[Dongarra et al., 1990] Dongarra, J. J., Du Croz, J., Hammarling, S., and Duff, I. S. (1990). A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 16(1):1–17.

[Dongarra et al., 1988] Dongarra, J. J., Du Croz, J., Hammarling, S., and Hanson, R. J. (1988). An extended set of fortran basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 14(1):1–17.

[Gautier and Lima, 2020] Gautier, T. and Lima, J. V. F. (2020). 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.

[Harris, 2013a] Harris, M. (2013a). How to access global memory efficiently in cuda c/c++ kernels. Online at: https://developer.nvidia.com/blog/how-access-global-memory-efficiently-cuda-c-kernels/.

[Harris, 2013b] Harris, M. (2013b). Using shared memory in cuda c/c++. Online at: https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/.

[Hwu, 2012] Hwu, Wen-mei;Kirk, D. (2012). *Programming massively parallel processors: A hands-on approach*. Elsevier / Morgan Kaufmann, 2nd edition edition.

[Khronos, 2023] Khronos, O. W. G. (2023). The opencl specification version 3.0. Online at: `https://registry.khronos.org/OpenCL/specs/3.0-unified/pdf/OpenCL_API.pdf`.

[Kim, 2016] Kim, D. (2016). Representations of task assignments in distributed systems using young tableaux and symmetric groups. *International Journal of Parallel, Emergent and Distributed Systems*, 31:152–175.

[Kristensen, 2012] Kristensen, M. (2012). *Towards Parallel Execution of Sequential Scientific Applications*. PhD thesis.

[Krüger and Westermann, 2003] Krüger, J. and Westermann, R. (2003). Linear algebra operators for gpu implementation of numerical algorithms. *ACM Trans. Graph.*, 22(3):908–916.

[Lawson et al., 1979] Lawson, C. L., Hanson, R. J., Kincaid, D. R., and Krogh, F. T. (1979). Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5(3):308–323.

[Luitjens, 2015] Luitjens, J. (2015). Cuda streams: Best practices and common pitfalls. In *GPU Techonology Conference*. Online at: `https://on-demand.gputechconf.com/gtc/2014/presentations/S4158-cuda-streams-best-practices-common-pitfalls.pdf`.

[McCool et al., 2012] McCool, M., Reinders, J., and Robison, A. (2012). *Structured parallel programming: patterns for efficient computation*. Elsevier.

[Nichols et al., 1996] Nichols, B., Buttlar, D., and Farrell, J. (1996). *Pthreads programming: A POSIX standard for better multiprocessing*. O'Reilly Media, Inc.

[Nvidia, 2011] Nvidia (2011). Opencl best practices guide. Online at: https://developer.download.nvidia.com/compute/DevZone/docs/html/OpenCL/doc/OpenCL_Best_Practices_Guide.pdf.

[Nvidia, 2016] Nvidia (2016). *Nvidia GeForce GTX 1060*. Online at: https://www.nvidia.com/en-us/geforce/news/gfecnt/nvidia-geforce-gtx-1060/.

[Nvidia, 2017] Nvidia (2017). *Nvidia Tesla V100 SXM2*. Online at: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[Nvidia, 2023a] Nvidia (2023a). Api reference guide for cublas.
Online at: https://docs.nvidia.com/cuda/cublas/.

[Nvidia, 2023b] Nvidia (2023b). Cuda c++ programming guide. v12.2
Online at: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[Ostrouchov, 1995] Ostrouchov, S. (1995). Block cyclic data distribution.
Online at: https://www.netlib.org/utk/papers/factor/node3.html.

[Rennich, 2011] Rennich, S. (2011). Cuda c/c++ streams and concurrency. In *GPU Technology Conference*. Online at: https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf.

[Singer, 2013] Singer, G. (2013). The history of the modern graphic processor.

[Strassen, 1969] Strassen, V. (1969). Gaussian elimination is not optimal. *Numerische mathematik*, 13(4):354–356.

[Unix, 2006] Unix (2006). *sem_getvalue(3) — Linux Programmer's Manual*.

[Waeijen, 2018] Waeijen, L. (2018). Matrix multiplication cuda. Online at: https://ecatue.gitlab.io/gpu2018/pages/Cookbook/matrix_multiplication_cuda.html.

[Wang et al., 2016] Wang, L., Wu, W., Xu, Z., Xiao, J., and Yang, Y. (2016). Blasx: A high performance level-3 blas library for heterogeneous multi-gpu computing. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, New York, NY, USA. Association for Computing Machinery.