Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Βελτιστοποίηση κατανεμημένων αλγορίθμων πολλαπλασιασμού πινάκων για αποδοτική εκτέλεση σε υπερυπολογιστικά συστήματα επεξεργαστών γραφικών

Διπλωματική Εργασία

Βασίλης Βρεττός

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

Αθήνα, Οκτώβριος 2024

Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Βελτιστοποίηση κατανεμημένων αλγορίθμων πολλαπλασιασμού πινάκων για αποδοτική εκτέλεση σε υπερυπολογιστικά συστήματα επεξεργαστών γραφικών

Διπλωματική Εργασία

Βασίλης Βρεττός

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

Εγκρίθηκε από την τριμελή επιτροπή την 18η Οκτωβρίου 2024.

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

Αθήνα, Οκτώβριος 2024

Βασίλης Βρεττός
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Σκοπός της παρούσης διπλωματικής εργασίας είναι η μελέτη υπάρχοντων κατανεμημένων αλγορίθμων πολλαπλασιασμού πινάκων που εκτελούνται σε υπερυπολογιστές και γενικά συστήματα συστοιχιών και η υλοποίηση μοντέρνων εκδόσεων που εκτελούνται αποδοτικά σε επεξεργαστές γραφικών. Το μεγαλύτερο ποσοστό της προϋπάρχουσας έρευνας αγνοεί τα ιδιαίτερα χαρακτηριστικά των GPU, αφήνωντας αρκετά σημαντικό περιθώριο επίδοσης ανεκμετάλλευτο. Έχοντας μελετήσει την αρχιτεκτονική των GPU, τροποποιούμε τους ήδη υπάρχοντες αλγορίθμους δίνοντας βάση στην αποδοτική επικοινωνία μεταξύ συσκευών και την ελάχιστη χρήση μνήμης με τελικό στόχο την παράδοση μιας αποδοτικής βιβλιοθήκης πολλαπλασιασμού πινάκων η οποία εκτελείται σε μοντέρνα υπερυπολογιστικά συστήματα χρησιμοποιώντας επεξεργαστές γραφικών.

Λέξεις κλειδιά: Πολλαπλασιασμός πίνακα με πίνακα; Επεξεργαστές γραφικών; Υπερυπολογιστικά Συστήματα ; Συστοιχίες Υπολογιστών; Βελτιστοποίηση Επικοινωνίας/Υπολογισμού

Abstract

Multiplication of Dense Matrices is one of the most common and important mathematical kernels executed in both normal systems and clusters alike. Following the major increase of GPUs in the HPC world due to the recent ”AI boom”, the GEMM kernel is commonly found being executed by Graphics Processors instead of the conventional CPUs. The previous distributed algorithms used by the GEMM calls were heavily designed for use in CPU based systems and fall short when executed in modern GPU nodes. In this thesis, we study these distributed algorithms as well as PBLAS libraries that focus on GPU execution in an attempt to optimize them for use in modern cluster systems. In the process, we present our own method of execution, built upon existing algorithms, modified in such a way that many of the special features of contemporary Graphics Processors are utilized in order to increase performance.

Keywords: Distributed Dense Matrix Multiplication; Graphics Processors (GPUs); Cluster Systems ; HPC; Distributed Algorithm Optimization; Communication/Computation Overlap

Ευχαριστίες

Θα ήθελα να ευχαριστήσω θερμά τον επιβλέποντα της συγκεκριμένης διπλωματικής εργασίας, τον καθηγητή Γεώργιο Γκούμα, για την συνεχή εμπιστοσύνη, καθοδήγηση και έμπνευση που μου προσέφερε σε πολλά από τα ακαδημαϊκά μου βήματα καθώς και όλο το εργαστήριο υπολογιστικών συστημάτων της σχολής μας (CSLab).

Πιο συγκεκριμένα, θα ήθελα να ευχαριστήσω τον Δρ. Πέτρο Αναστασιάδη για την συνεχόμενη υποστήριξη, επικοινωνία και συμβουλές που προσέφερε καθ'όλη την διάρκεια μελέτης και υλοποίησης των έργων της διπλωματικής αλλά και εκτός αυτών σε προσωπικό επίπεδο.

Επίσης, θέλω να πω ένα μεγάλο ευχαριστώ στους φίλους μου για όλες τις όμορφες στιγμές που ζήσαμε αυτά τα χρόνια, τα προβλήματα που ξεπεράσαμε μαζί και για το μέλλον που - ελπίζω - να περάσουμε μαζί.

Το τελευταίο ευχαριστώ, όντας πάντα και το πιο σημαντικό, πηγαίνει στην οικογένειά μου και συγκεκριμένα στην μητέρα μου, Κατερίνα, τον αδερφό μου, Γιάννη και την γιαγιά μου, Βασιλική για την συνεχή αγάπη και φροντίδα που έλαβα όλα αυτά τα χρόνια - το έργο αυτό αφιερώνεται ιδιαιτέρως σε *εκείνους*.

Βασίλης Βρεττός,
Αθήνα, Οκτώβριος 2024

Περιεχόμενα

Chapter 1

Απαραίτητες Γνώσεις

1.1 Εισαγωγή

Η ταχύτατη ανάπτυξη της Τεχνητής Νοημοσύνης και εφαρμογών Μεγάλων Δεδομένων έχει οδηγήσει σε σημαντική αύξηση της ζήτησης για υπολογιστική ισχύ. Την ζήτηση αυτή κάλυψαν οι μοντέρνοι επεξεργαστές γραφικών (GPUs), οι οποίοι προσφέρουν εξαιρετικές δυνατότητες παράλληλης επεξεργασίας δεδομένων. Τα σύγχρονα υπερυπολογιστικά συστήματα, έχοντας πλέον υιοθετήσει σε σημαντικό βαθμό τις GPU ως βασικές ή υποστηρικτικές μονάδες επεξεργασίας, είναι σχεδιασμένα για να επεξεργάζονται τεράστια ποσά δεδομένων και να εκπαιδεύουν μεγάλα μοντέλα AI.

Ο πολλαπλασιασμός πίνακα με πίνακα, ένας υπολογιστικός πυρήνας με πολύ συχνή εμφάνιση στον τομέα του AI αλλά και σε πληθώρα επιλυτών μαθηματικών προβλημάτων και προσομοιώσεων, έχει υπάρξει το επίκεντρο της προσοχής διαφόρων ερευνητικών ομαδών και σχεδιαστών υλικού από την δεκαετία του 90. Παρά την εκτεταμένη χρήση αυτού του υπολογιστικού πυρήνα, ειδικά σε συστήματα συστοιχιών, και του αξιοσημείωτου όγκου έρευνας επάνω στο αντικείμενο, οι υπάρχοντες βιβλιοθήκες πολλαπλασιασμού πινάκων με εκτέλεση σε GPUs δεν αξιοποιούν πλήρως το εκάστοτε υλικό. Περαιτέρω, ορισμένες σχεδιαστικές αποφάσεις των βιβλιοθηκών αυτών περιορίζουν την ικανότητα κλιμακωσιμότητας σε πιο μοντέρνες αρχιτεκτονικές.

Στην παρούσα διπλωματική εργασία, προσπαθούμε να αναγνωρίσουμε τις αδυναμίες αυτές και να βασιστούμε σε προηγούμενες έρευνες ώστε να υλοποιήσουμε αποτελεσματικές μεθόδους εκτέλεσης πολλαπλασιασμού πίνακα με πίνακα σε GPUs, εντός συστημάτων συστοιχιών.

1.2 General Purpose Graphics Processing Units (GPGPUs)

1.2.1 Επεξεργαστές Γραφικών

Οι επεξεργαστές γραφικών ξεκίνησαν να αναπτύσσονται πολύ σύντομα μετά την δημιουργία του απλού μικροεπεξεργαστή, καθώς η υπολογιστική ισχύ των τελευταιών δεν επαρκούσε για την επεξεργασία ψηφιακής εικόνας. Χρησιμοποιούνται ως συνεπεξεργαστές σε κλασσικές μονάδες επξεργασίας. Αρχικά, στην δεκαετία του 70 και 80, σχεδιάστηκαν **μόνο** για γραφικά, για μονάδες

υπολογιστών ή καμπίνες παιχνιδιών Arcade, από τις εκείνης της εποχής καινοτόμους στον χώρο του σχεδιασμού υπολογιστών, Atari, Hitachi και IBM.

Κατά την διάρκεια της δεκατετίας του 90, η υιοθέτηση των 2D γραφικών περιβάλλοντων από τα τότε μοντέρνα λειτουργικά συστήματα παρότρυνε τους κατασκευαστές επεξεργαστών γραφικών να αναπτύξουν επιτάχυνση για εκείνα απευθίας στον ίδιο τον επεξεργαστή. Ταυτόχρονα, καθώς αναπτύσσονται τα 3D γραφικά πραγματικού χρόνου σε πλατφόρμες παιχνιδιών arcade αλλά και σε παιχνίδια που τρέχουν σε προσωπικούς υπολογιστές, η ανάγκη για επιτάχυνση αυτών των εφαρμογών αυξάνεται ραγδαία. Συγκεκριμένα για τους προσωπικούς υπολογιστές, εταιρείες όπως η ATi (πλέον παράρτημα της AMD) η 3dfx και η Nvidia προσπαθούν να προσφέρουν αυτή την υποστήριξη, αλλά αποτυγχάνουν στην αρχή. Η υποστήριξη για 3D γραφικά υλοποιούνταν ξεχωριστά από αυτήν για 2D, δημιουργώντας πολύπλοκα συστήματα με μεγάλο κόστος. Μελλοντικά, οι δύο αυτές λειτουργίες συγχωνεύτηκαν. Στα τέλη τις δεκαετίας αυτής, κυκλοφόρησε η πρώτη ”Μονάδα Επεξεργασίας Γραφικών” (GPU), η GeForce 256 της Nvidia, η οποία συμπεριλάμβανε υποστήριξη για 2D, 3D γραφικά και υπολογισμό γεωμετριών και φωτισμού.

Παράλληλα με την ανάπτυξη της υποστήριξης 2D και 3D γραφικών από τις κάρτες γραφικών, αναπτύχθηκαν 2 σημαντικές προγραμματιστικές διεπαφές, το OpenGL και το Direct3D (το οποίο αργότερα εξελίχθηκε στο DirectX). Οι διεπαφές αυτές επιτρέπουν στους προγραμματιστές να αξιοποιούν με ευκολία δυνατότητες που προσφέρουν οι επεξεργαστές γραφικών για την απεικόνιση γραφικών σε μια έξοδο εικόνας. Αξιοποιήθηκαν σε μεγάλο βαθμό για την ανάπτυξη γραφικά περιβάλλοντα χρήστη καθώς και για μηχανές γραφικών παιχνιδιών. Οι επεξεργαστές γραφικών αναφέρουν με ακρίβεια ποιες διεπαφές και ποιες εκδόσεις αυτών υποστηρίζουν με άμεση υποστήριξη υλικού.

Μετά το 2001, αφού αναπτύχθηκε η υποστήριξη υπολογισμού σκίασης (shaders) και πράξεων κινητής υποδιαστολής από επεξεργαστές γραφικών, η επιστημονική κοινότητα πειραματίστηκε με την εκτέλεση αριθμητικών αλγορίθμων σε συσκευές γραφικών. Συγκεκριμένα, δοκιμάστηκαν επιλυτές αραιών πινάκων σε εφαρμογές επίλυσης διαφορικών εξισώσεων και αποδίκτηκε ότι η επίδοσή τους είναι ανώτερη από αυτή μιας συμβατικής CPU[1] [2]. Ιστορικά, από το σημείο αυτό και αργότερα, γεννιέται η έννοια των επεξεργαστών γραφικών γενικού σκοπού (General Purpose Graphics Processing Unit, GPGPUs).

Το προγραμματιστικό μοντέλο CUDA παρουσιάστηκε το 2007 από την Nvidia ως η λύση για τον γενικό προγραμματισμό GPU, επιτρέποντας στους προγραμματιστές να αναπτύσσουν λογισμικά αγνοώντας τις σύνθετες αρχιτεκτονικές λεπτομέρειες του επιταχυντή. Λίγα χρόνια αργότερα, κυκλοφόρησε η πρώτη έκδοση του OpenCL (Open Computing Language) [3], ενός open source framework που επιτρέπει την δημιουργία λογισμικού που μπορεί να εκτελεστεί σε ετερογενή συστήματα, χρησιμοποιώντας directives της γλώσσας προγραμματισμού C ή C++. Τέλος, η AMD παρουσίασε το ROCm, μια open source ”στοίβα” λογισμικού για τον προγραματισμό ετερογενών συστημάτων. Συγκεκριμένα, το HIP (Heterogeneous Interface for Portability) είναι η αντίστοιχη έκδοση του CUDA, το οποίο όμως επιτρέπει τον προγραμματισμό οποιασδήποτε GPGPU και δεν περιορίζεται σε συσκευές συγκεκριμένης εταιρείας.

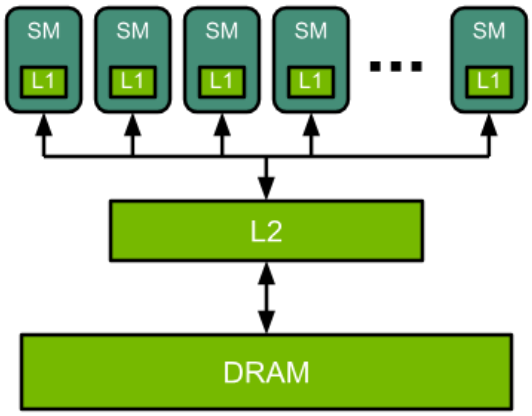
Από την ”εισαγωγή” αυτών των προγραμματιστικών διεπαφών, οι GPUs συνέχισαν να εξελίσσονται ως επιταχυντές πιο γενικού σκοπού και όχι μόνο σαν συσκευές υπολογισμού

γραφικών. Η αρχιτεκτονική τους έχει αποδειχθεί ιδιαίτερα ικανή για την εκτέλεση ντροπιαστικά παράλληλων εφαρμογών. Τις συναντάμε εκτεταμένα σε τομείς όπως στην μηχανική μάθηση, στην επίλυση προβλημάτων αριθμητικής ανάλυσης και υπολογιστικής επιστήμης και στον τομέα του Big Data.

1.2.2 Αρχιτεκτονική Επεξεργαστών Γραφικών

Η αρχιτεκτονική ενός επεξεργαστή γραφικών είναι υψηλά παράλληλη και αποτελείται από μονάδες επεξεργασίας και μια απλή ιεραρχία μνήμης. Για τις αρχιτεκτονικές της Nvidia, οι οποίες απασχολούν το συγκεκριμένο έργο, οι επεξεργαστές αποτελούνται από μονάδες επεξεργασίας που ονομάζονται Streaming Multiprocessors (SM). Κάθε μονάδα περιέχει ένα μεγάλο αρχείο καταχωρητών, στην τάξη των 10 χιλιάδων και άνω, έχει ξεχωριστό χρονοδρομολογητή εντολών και οργανώνει την συνεχή διοχέτευση εντολών ατομικά, προσφέροντας έτσι και παραλληλισμό σε επίπεδο εντολής.

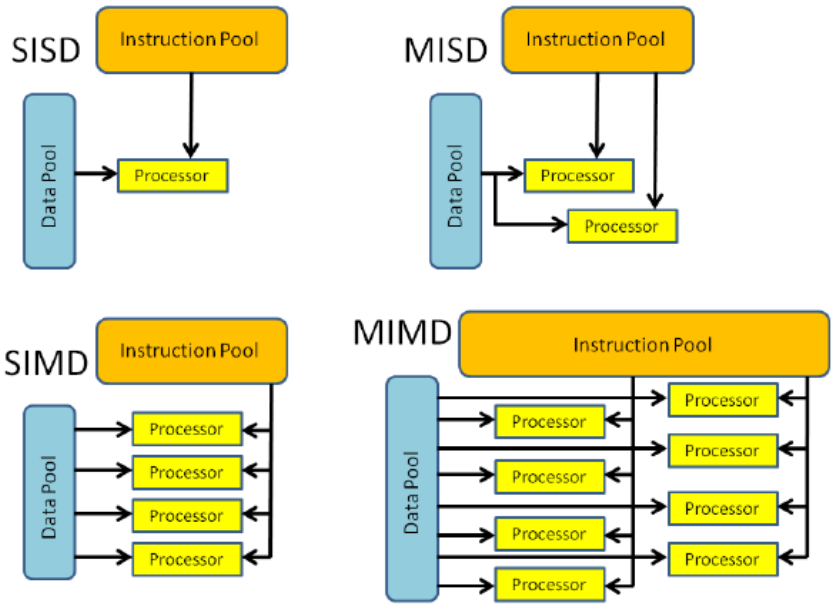
Η ιεραρχία μνήμης θυμίζει συμβατικού επεξεργαστή, με εξωτερική DRAM μνήμης μεγάλης χωρητικότητας, διαμοιραζόμενη L2 κρυφή μνήμη μεταξύ όλων των μονάδων επεξεργασίας και τοπική L1 κρυφή μνήμη ανά μονάδα επεξεργασίας. Εκτός από την L1 cache, κάθε streaming multiprocessor περιέχει και κρυφή μνήμη σταθερών τιμών, μόνο για ανάγνωση (read-only constant cache).



Σχήμα 1.1: Απλοποιημένη Αρχιτεκτονική Nvidia GPU [4]

Εκτός από τα κλασσικά streaming multiprocessors, οι πιο μοντέρνοι επεξεργαστές γραφικών της Nvidia προσφέρουν εξειδικευμένους πυρήνες οι οποίοι εκτελούν πράξεις Multiply and Accumulate σε πίνακες (Matrix Multiply and Accumulate, MMAC). Οι πυρήνες αυτοί ονομάζονται Tensor Cores και αποδεικνύονται πολύ χρήσιμοι σε εφαρμογές επεξεργασίας σημάτων και εικόνων, στην εκπαίδευση μοντέλων αλλά και σε εφαρμογές γραμμικής άλγεβρας.

Σύμφωνα με την ταξινομία του Flynn [5] , ένας επεξεργαστής γραφικών υλοποιεί μια Single Instruction, Multiple Data (SIMD) αρχιτεκτονική, στην οποία, μια συγκεκριμένη πράξη εκτελείται παράλληλα από όλους τους πυρήνες του επεξεργαστή αλλά σε διαφορετικά δεδομένα. Αυτό δεν σημαίνει πως **όλα** τα νήματα ενός SM είναι υποχρεωμένα να εκτελέσουν την ίδια εντολή. Η οργάνωση αυτή γίνεται από τους χρονοδρομολογητές εντολών οι οποίοι υλοποιούνται σε υλικό, και όχι σε επίπεδο λογισμικού.



Σχήμα 1.2: Ταξονομία του Flynn - η GPU αποτελεί μια αρχιτεκτονική SIMD [6]

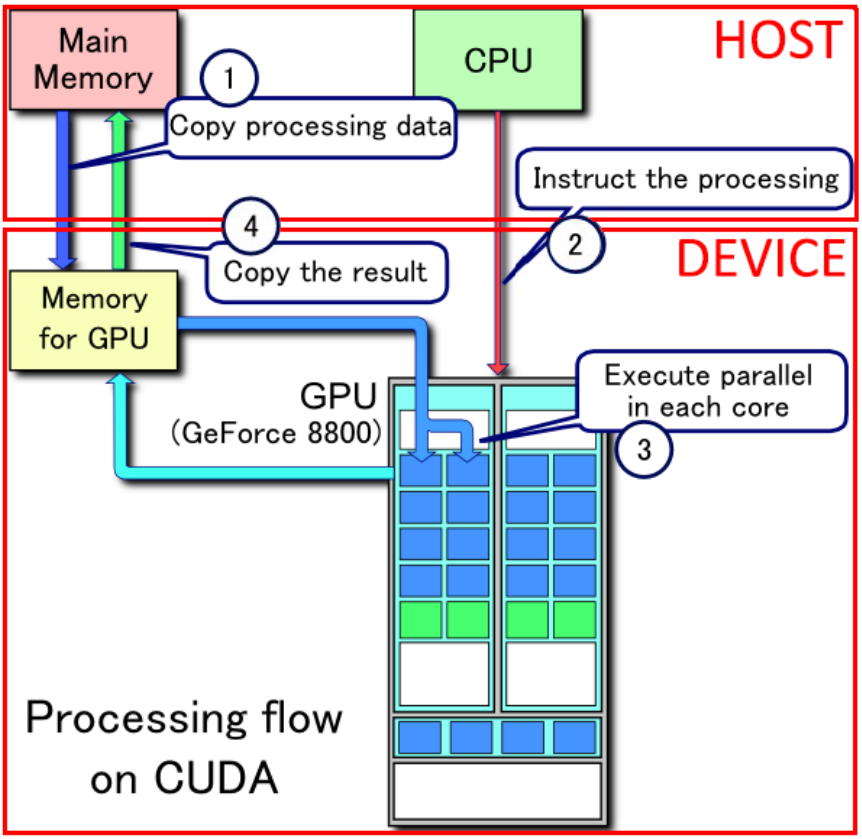
Ο διαμοιρασμός δεδομένων μεταξύ CPU και GPU γίνεται μέσω της ξεχωριστής μνήμης DRAM. Η σύνδεση των δύο γίνεται μέσω αμφίδρομου σειριακού δίαυλου. Το μοντέρνο standard είναι το PCI-e (Peripheral Component Interconnect Express) με την πιο πρόσφατη έκδοση μαζικής παραγωγής (version 6.0) να επιτρέπει ταχύτητες εώς και 64 GT/s (Giga Transfers per second) σε πλήρη συνδεσμολογία 16-λωρίδων. Η επικοινωνία μεταξύ 2 επεξεργαστών γραφικών σε κοινό σύστημα γίνεται είτε μέσω διάυλου PCI-e ή μέσω NVLink, ενός κλιμακώσιμου συστήματος επικοινωνίας ειδικά σχεδιασμένο για γρήγορες μεταφορές δεδομένων μεταξύ Nvidia GPU.

Κατά την εκτέλεση ενός προγράμματος, αναφερόμαστε στην CPU ως ”host” ενώ σε κάθε GPU ως ”device”. Αφού μεταφερθούν τα δεδομένα στις μνήμες των συσκευών από τον επεξεργαστή, ο ίδιος ξεκινάει την εκτέλεση κώδικα της GPU μέσω μιας διαδικασίας που αποκαλείται εκκίνηση πυρήνα (kernel launch). Αφού τελειώσει η εκτέλεση του κώδικα στον επιταχυντή, μεταφέροντα τα δεδομένα από αυτόν πίσω στην κύρια μνήμη της CPU. Η διαδικασία αυτή μπορεί να είναι σύγχρονη (blocking), όπου ο επεξεργαστής περιμένει μέχρι να τελειώσει η εκτέλεση κώδικα στην GPU ή ασύγχρονη (non-blocking), όπου ο πυρήνας του επεξεργαστή είναι ελεύθερος να συνεχίσει την ροή του προγράμματος που εκτελούσε.

Σύγκριση CPU με GPU

Αν και τα δύο αυτά είδη επεξεργαστών μπορούν να εκτελέσουν γενικές μαθηματικές και λογικές πράξεις, οι σημαντικές διαφορές στην αρχιτεκτονική τους καθιστά τον καθένα βέλτιστο αναλόγως την εφαρμογή που σκοπεύουμε να εκτελέσουμε. Τα καθοριστικά κριτήρια για την βέλτιστη επιλογή επεξεργαστή είναι ο βαθμός παραλληλίας και ο αριθμός υπολογισμού ανά δεδομένο (Operational Intensity) του αλγορίθμου που εκτελούμε.

Οι CPUs σχεδιάστηκαν για γενικούς υπολογισμούς και για ικανοποιητικές επιδόσεις οποιασδήποτε εφαρμογής. Εκτός από υπολογισμούς, αναλαμβάνουν τον φόρτο διαχείρισης του Λειτουργικό Συστήματος όπως μεταφορά δεδομένων μεταξύ μνήμης και δίσκου, διαχείριση



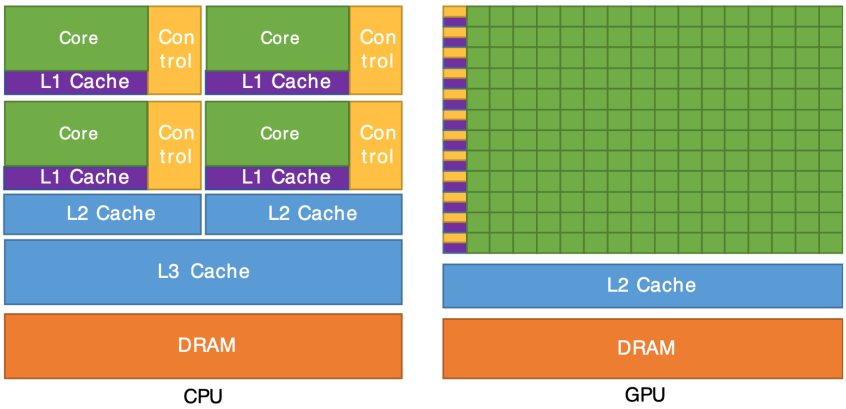
Σχήμα 1.3: Μοντέλο Εκτέλεσης GPU

διεργασιών και επικοινωνία με άλλα συστήματα μέσω δικτύου. Περιέχουν σύνθετα και πολύπλοκα κυκλώματα για συνεχή διοχέτευση εντολών, πρόβλεψη εντολών διακλαδώσεων και εκτέλεσης εντολών εκτός σειράς (Out of Order execution) με αποτέλεσμα να προσφέρουν αρκετά υψηλότερη επίδοση σε μονονηματικές ή σειριακές εκτελέσεις προγραμμάτων.

Αντιθέτως, οι GPUs ευνοούνται από εφαρμογές που εκτελούν παραλληλισμό σε επίπεδο δεδομένων αλλά δεν επιτρέπουν ταυτοχρονισμό. Σε αντίθεση με τους κλασσικούς επεξεργαστές που περιέχουν λίγους αλλά σύνθετους πυρήνες, οι επεξεργαστές γραφικών χρησιμοποιούν λιτά κυκλώματα ελέγχου ροής και caching και αξιοποιούν την μεγαλύτερη επιφάνεια του chip σε πολλαπλά και πιο αδύναμα Arithmetic Logic Units (ALUs). Άρα, επιτρέπουν την παράλληλη εκτέλεση πράξεων από χιλιάδες νήματα σε διαφορετικά δεδομένα, προσφέροντας υψηλές επιδόσεις σε πολύνηματικές εκτελέσεις προγραμμάτων, όπου αρκετές πράξεις εκτελούνται σε πολλαπλά δεδομένα. Αν και οι πυρήνες ενός επεξεργαστή γραφικών έχουν πολύ χαμηλότερες επιδόσεις από αυτές ενός κλασσικού επεξεργαστή, το μεγάλο πλήθος των πρώτων αντισταθμίζει τις υπολογιστικές ικανότητες των δύο ειδών επεξεργαστών.

1.2.3 Προγραμματιστικό Μοντέλο CUDA

Μέχρι στιγμής έχουμε περιγράψει την απτή αρχιτεκτονική ενός επεξεργαστή γραφικών Nvidia. Το CUDA είναι μια προγραμματιστική διεπαφή και πλατφόρμα η οποία αναπτύχθηκε από την Nvidia συγκεκριμένα για γενικό προγραμματισμό GPU. Στην πραγματικότητα, αποτελεί μια επέκταση των γλωσσών C/C++, Fortran και Python η οποία επιτρέπει στον προγραμματιστή να



Σχήμα 1.4: Απλοποιημένα διαγράμματα αρχιτεκτονικής CPU και GPU [7]

διαχειριστεί την εκτέλεση του κώδικα στην GPU, με κάποιους περιορισμούς, χωρίς να χρειαστεί να αναφερθεί στην υποκείμενη αρχιτεκτονική αυτής.

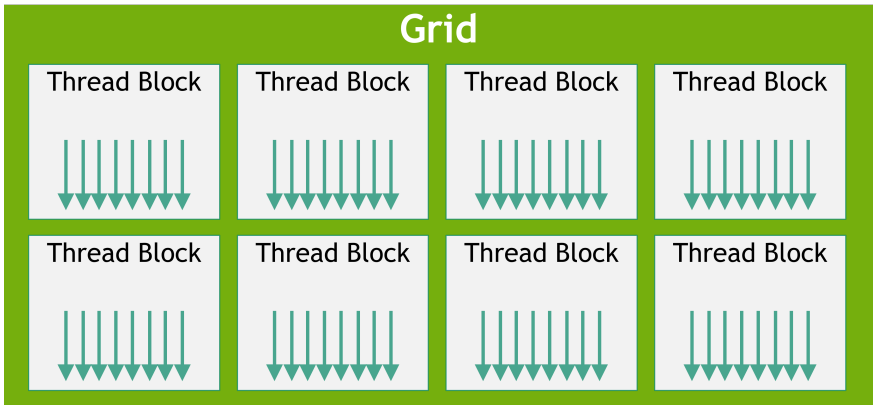
Η διαφοροποίηση μεταξύ κώδικα που τρέχει στην CPU ή στην GPU γίνεται μέσω λέξεων κλειδιών που εισάγει το CUDA. Συναρτήσεις που καλούνται και εκτελούνται μόνο από συσκευές πρέπει να ορίζονται με το αναγνωριστικό __device__. Αντίστοιχα, για συναρτήσεις της CPU, χρησιμοποιείται το αναγνωριστικό __host__. Αν χρησιμοποιηθούν και τα 2 αναγνωριστικά, τότε μπορεί να γίνει κλήση συνάρτησης και από τα 2 είδη επεξεργαστών, αλλά χρειάζεται διαφοροποίηση του κώδικα αναλόγως την συσκευή. Η προγραμματιστική βάση του framework είναι οι υπολογιστικοί πυρήνες (kernels), συναρτήσεις που φέρουν το αναγνωριστικό __global__, η κλήση των οποίων γίνεται από την CPU. Κατά την κλήση, ο προγραμματιστής επιλέγει 2 σημαντικές παραμέτρους, τον αριθμό των block και τον αριθμό των νημάτων ανά block.

Οργάνωση των πόρων

Η κυριότερη ευθύνη του προγραμματιστή είναι η κλήση των υπολογιστικών πυρήνων. Η χρονοδρομολόγηση των νημάτων γίνεται εξ ολοκλήρου από τον επεξεργαστή γραφικών σε επίπεδο υλικού. Τα νήματα, οργανωμένα σε blocks, έχουν ένα ξεχωριστό αναγνωριστικό, το threadId, βάσει του οποίου μπορεί το καθένα να ενεργήσει σε διαφορετικά δεδομένα. Περαιτέρω, κάθε block νημάτων έχει ένα blockId. Τα ίδια τα block μπορούν να κατανεμηθούν σε μία, δύο ή τρεις διαστάσεις. Αν και αυτή η κατανομή δεν είναι απαραίτητη, επιτρέπει ευκολότερο προγραμματισμό αλγορίθμων που χρησιμοποιούν πολυδιάστατες δομές, όπως διανύσματα, πίνακες ή τανυστές μεγαλύτερης τάξης. Ο μέγιστος αριθμός νημάτων ανά block, σε μοντέρνες GPU, είναι 1024. Το τελευταίο επίπεδο οργάνωσης είναι τα πλέγματα, στα οποία κατανέμονται τα blocks. Αντίστοιχα με τα τελευταία, μπορούν να οργανωθούν σε 1 εώς 3 διαστάσεις.

Οργάνωση της μνήμης

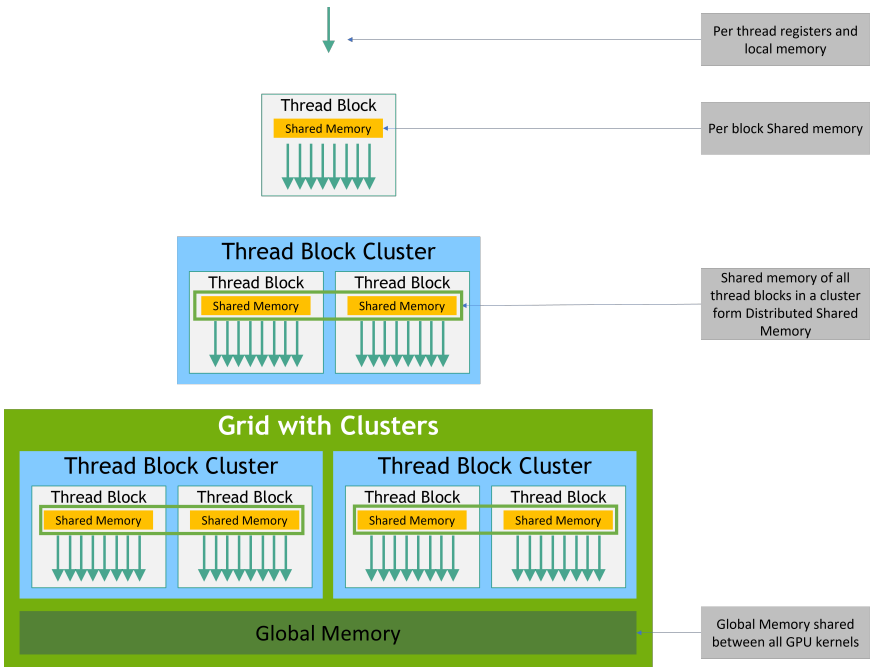
Αναλόγως την εφαρμογή, η διαχείριση της ιεραρχίας μνήμης της συσκευής παίζει καθοριστικό ρόλο στις επιδόσεις που μπορεί να πετύχει μια GPU. Κάθε νήμα έχει τοπική μνήμη και καταχωρητές η οποία δεν μοιράζεται με κανέναν άλλο πόρο εντός του block. Κάθε block περιέχει διαμοιραζόμενη μνήμη (__shared__ keyword κατά τον προγραμματισμό) την οποία τα νήματα μπορούν να προσπελαύνουν παράλληλα, εφόσον τα δεδομένα είναι συνεχόμενα και



Σχήμα 1.5: Οργάνωση νημάτων και block σε πλέγμα [7]

ευθυγραμμισμένα, προσφέροντας υψηλό bandwidth και ελάχιστο latency. Η μνήμη των σταθερών, με το αναγνωριστικό `__constant__` καθώς και η γενική μνήμη της GPU, με το αναγνωριστικό __device__, είναι ελεύθερες προς πρόσβαση από οποιοδήποτε νήμα του πλέγματος, έχουν μεγάλη χωρητικότητα αλλά χαμηλή ταχύτητα μεταφοράς δεδομένων, συγκριτικά με την διαμοιραζόμενη μνήμη ανά block.

Μία πρόσθετη έννοια μνήμης είναι η εννοποιημένη μνήμη (unified memory). Δημιουργώντας έναν εικονικό χώρο διευθύνσεων τον οποίο μοιράζονται η CPU και η GPU, επιτρέπεται η ελεύθερη χρήση μνήμης, αγνοώντας την πραγματική της θέση. Έτσι, προστίθεται ένα επίπεδο αφαίρεσης από τον προγραμματισμό, διευκολύνοντας την διαδικασία, μειώνοντας όμως την επίδοση.



Σχήμα 1.6: Ιεραρχία μνήμης στο μοντέλο CUDA [7]

Παράδειγμα αρχιτεκτονικής - Nvidia A100

Σαν πρακτικό παράδειγμα, θέτουμε την Nvidia A100, η οποία περιέχει 108 SM, 64 CUDA Cores ανά SM, με συνολικά 6912 πυρήνες, 4 πυρήνες Tensor Core ανά SM, με συνολικά 432 και 192KB

shared memory ανά SM [8]. Αναλόγως το μέγεθος του block, κάθε SM μπορεί να εκτελεί ένα ή περισσότερα thread block. Η εκτέλεση κώδικα ανά thread block είναι πλήρως ανεξάρτητη και θα πρέπει να σχεδιάζεται έτσι καθώς ο προγραμματιστής δεν γνωρίζει με ποιον τρόπο θα δρομολογηθούν προς εκτέλεση από την συσκευή.

Σχήμα 1.7: GA100 Floor Plan - η βάση της Nvidia A100 [8]

Έαν ο συγχρονισμός μεταξύ νημάτων κοινού block είναι απαραίτητος, παρέχεται στον ένας απλός μηχανισμός συγχρονισμού, τύπου barrier, με χρήση της συνάρτησης __syncthreads(). Ο συγχρονισμός μεταξύ διαφορετικών block γίνεται με άλλα μέσα.

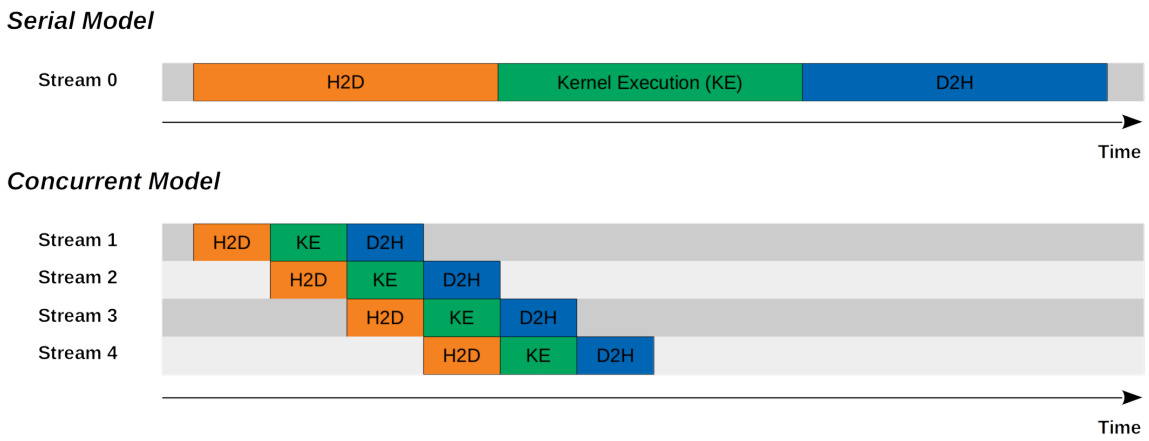
Συνολικά τα νήματα κάθε block οργανώνονται ανά 32 σε warps. Τα warps αποτελούν το κβάντο δρομολόγησης εντολών ανά SM. Τα νήματα ενός warp εκτελούν την ίδια εντολή, οποιαδήποτε στιγμή. Αναγκαίο μειονέκτημα αυτής της οργάνωσης είναι ότι σε πιθανή διακλάδωση όπου νήματα του ίδιου warp ακολουθήσουν διαφορετική ροή εκτέλεσης, οι εκτελέσεις των μονοπατιών σειριοποιούνται, πρόβλημα που καθιστά συγκεκριμένους αλγορίθμους, ειδικά αυτούς που χρησιμοποιούν γράφους ως δομή δεδομένων, λιγότερο αποδοτικούς για εκτέλεση σε GPU. Εάν ο συνολικός αριθμός των νημάτων ενός block δεν είναι πολλαπλάσιο του 32, οι περισσευόμενες θέσεις νημάτων στο τελευταίο wrap γεμίζουν με ”ανενεργά” νήματα.

Εύκολα παρατηρούμε ότι η σωστή οργάνωση των νημάτων σε blocks και πλέγματα είναι αρκετά σημαντική για την επίδοση του κώδικα που εκτελείτε σε μία GPU και αποτελεί καθαρά ευθύνη του χρήστη. Για να διευκολυνθεί ο προγραμματιστής στην οργάνωση, προσφέρονται εργαλεία υπολογισμού αξιοποίησης της GPU βάσει των νημάτων, block και μνήμης που επιθυμεί να χρησιμοποιήσει. Ο υπολογισμός αυτός γινόταν μέσω του Occupancy Calculator ο οποίος ήταν ένας χειροκίνητος και δύσχρηστος τρόπος να εξάγει συμπεράσματα για την θεωρητικά μέγιστη επίδοση της εκτέλεσης ενός kernel. Πλέον, προσφέρονται runtime εργαλεία τα οποία υπολογίζουν το occupancy, δηλαδή το ποσοστό των ενεργών warp ενός SM προς τον μέγιστο αριθμό ενεργών warp που μπορεί να εκτελέσει ένα SM.

Ταυτοχρονισμός σε GPU

Εκτός από τις δυνατότητες συγχρονισμού νημάτων εντός block και γενικών ατομικών λειτουργιών, το CUDA προσφέρει ένα εργαλείο για ταυτοχρονισμό εργασιών, στην μορφή των streams. Ένα CUDA Stream αποτελεί μια σειρά από εργασίες που εκτελούνται σειριακά στην GPU, δηλαδή, μια ουρά εργασιών. Εάν δεν έχει οριστεί από τον χρήστη, όλες οι λειτουργίες της GPU εκτελούνται μέσω του default stream, στο οποίο όλες οι εργασίες μεταφοράς δεδομένων, allocation μνήμης ή κλήσης kernel, γίνονται σύγχρονα με τον host. Εργασίες που βρίσκονται σε κοινό stream, εκτελούνται σειριακά, ενώ, με την προϋπόθεση ότι επαρκεί το υλικό, εργασίες διαφορετικού stream μπορούν να εκτελεστούν παράλληλα. Συγκεκριμένα για την μεταφορά δεδομένων, επιτρέπεται παράλληλη εκτέλεση μεταξύ διαφορετικών κατευθύνσεων, για παράδειγμα, μία μετάφορα από host προς device (H2D) και μια μεταφορά από device προς host (D2H), μπορούν να εκτελεστούν παράλληλα. Αντιθέτως, δύο H2D ή D2H μεταφορές, σειριοποιούνται, ακόμα και αν βρίσκονται σε διαφορετικά streams.

Για την πλήρη αξιοποίηση των streams, χρησιμοποιούνται οι ασύγχρονες εκδοχές των γενικών συναρτήσεων CUDA. Για την παράλληλη εκτέλεση 2 μεταφορών διαφορετικών κατευθύνσεων, χρησιμοποιούνται οι κλήσεις της συνάρτησης `cudaMemcpyAsync()`. Για την ασύγχρονη μεταφορά δεδομένων, η μνήμη του host, πρέπει να είναι "κλειδωμένη" ώστε να μην μετακινηθούν ή αντικαταστηθούν σελίδες μνήμης από το λειτουργικό σύστημα, καθώς η μεταφορά γίνεται μέσω DMA (Direct Memory Access), και η μετακίνηση σελιδών θα επηρέαζε το περιεχόμενο της μνήμης. Το "κλείδωμα" αυτό (page-locked ή pinned memory) γίνεται καλώντας την συνάρτηση `cudaMallocHost()` ή μέσω της βιβλιοθήκης `numactl` αντί για την κλασσική `malloc`.



Σχήμα 1.8: Παράδειγμα πολλαπλών CUDA Streams [9]

Περαιτέρω μηχανισμοί συγχρονισμού είναι ο γενικός συγχρονισμός της συσκευής, στον οποίο ο host περιμένει την συσκευή μέχρι να τελειώσει όλες τις εργασίες της, μέσω της `cudaDeviceSynchronize()`. Επίσης μπορεί να γίνει συγχρονισμός σε επίπεδο stream μέσω της `cudaStreamSynchronize()`, στην οποία ο host περιμένει μέχρι να ολοκληρωθούν όλες οι εργασίες ενός συγκεκριμένου stream. Σε επίπεδο εργασίας, συγχρονισμός επιτυγχάνεται με χρήση των γεγονότων (events), μιας δομής που προσφέρει το CUDA. Ένα CUDA event είναι μια δομή που θυμίζει timestamp και τοποθετείται σε ένα stream κατόπιν κλήσης συνάρτησης από τον χρήστη. Το event θεωρείται μη ολοκληρωμένο ως ότου ολοκληρωθούν όλες οι προηγούμενες

εργασίες που είχαν δρομολογηθεί στο stream πριν την τοποθέτησή του, κατόπιν θεωρείται ολοκληρωμένο. Για την εξέταση της κατάστασης, χρησιμοποιούμε την `cudaEventQuery()`. Τα events αποτελούν την πιο λεπτή μορφή ελέγχου για εργασίες εντός των stream, μεταξύ host και stream αλλά και μεταξύ διαφορετικών stream.

Compute Capability

Οι GPUs της Nvidia, όντας σύνθετοι και συνεχόμενα αναπτυσσόμενοι, περιγράφονται από ένα ζεύγος αριθμών που περιγράφει τις διαθέσιμες λειτουργίες τους. Το ζεύγος αυτό περιγράφει τις Δυνατότητες Συσκευής (Compute Capability) και περιέχει έναν μείζονα αριθμό που δηλώνει την γενική αρχιτεκτονική του επεξεργαστή και έναν ελάσσονα αριθμό που δηλώνει βελτιώσεις ή πρόσθετες λειτουργίες που υποστηρίζει. Για την Nvidia A100, το ζεύγος αυτό είναι 8.0 που σημαίνει ότι είναι σχεδιασμένο με την αρχιτεκτονική Nvidia Ampere.

1.3 Προγραμματισμός Συστημάτων Μεγάλης Κλίμακας

Η εργασία αυτή βασίζεται σε σχεδιασμό αλγορίθμων που εκτελούνται σε συστήματα μεγάλης κλίμακας, δηλαδή συστοιχίες υπολογιστών και υπερυπολογιστές. Οι συστοιχίες υπολογιστών είναι μια συλλογή από ανεξάρτητες υπολογιστικές μονάδες συνδεόμενες από γρήγορα δίκτυα διασύνδεσης και παρουσιάζονται στον χρήστη ως μια μοναδική οντότητα. Στις παρακάτω παραγράφους, αναφερόμαστε στις μεθόδους με τις οποίες δημιουργούμε εφαρμογές σε τέτοια συστήματα καθώς και τεχνικές που τις διαφοροποιούν από τον κλασσικό προγραμματισμό κοινής μνήμης.

1.3.1 Message Passing Interface (MPI)

Το κυρίαρχο εργαλείο προγραμματισμού συστημάτων μεγάλης κλίμακας είναι το μοντέλο περάσματος μηνυμάτων, γνωστό ως Message Passing Interface ή MPI. Ουσιαστικά, αποτελεί μια βιβλιοθήκη εργαλείων επικοινωνίας μεταξύ διεργασιών. Αν και αρχικά το μοντέλο δημιουργήθηκε για να υποστηρίζει μόνο επικοινωνία μεταξύ διεργασιών που ανήκουν σε διαφορετικά συστήματα, οι μοντέρνες υλοποιήσεις του MPI υποστηρίζουν και επικοινωνία μέσω διαμοιραζόμενης μνήμης.

Το MPI, μαζί με το προγραμματιστικό μοντέλο SPMD (Single Program Multiple Data) υιοθετήθηκε πλήρως από τον χώρο του HPC και γενικότερα στον προγραμματισμό συστημάτων κατανεμημένης μνήμης. Τυπικά, κάθε υπολογιστική μονάδα παρουσιάζεται ως μια διεργασία στο MPI. Η σύμβαση αυτή ξεκίνησε από τα πρώτα χρόνια ανάπτυξης των κατανεμημένων συστημάτων και των συστοιχίων υπολογιστών καθώς, συνήθως, ένας επεξεργαστής αποτελούταν από μόνο έναν πυρήνα. Πολυπύρηνες και πολυνηματικές αρχιτεκτονικές εμφανίστηκαν αρκετά αργότερα, αλλά πάλι ο αριθμός των πυρήνων ήταν περιορισμένος. Μοντέρνες αρχιτεκτονικές επεξεργαστών που περιέχουν πολύ μεγαλύτερο αριθμό πυρήνων καταργούν αυτή την σύμβαση και όπως θα δούμε αργότερα, το MPI επιτρέπει στον χρήστη να ορίζει εκείνος τους πόρους που θα δεσμεύει και θα ορίζει την κάθε διεργασία.

Μία εφαρμογή MPI υλοποιεί το μοντέλο SPMD, στο οποίο, το ίδιο εκτελέσιμο πρόγραμμα εκτελείται από πολλαπλές διεργασίες. Η κάθε διεργασία έχει ένα αναγνωριστικό, το rank της,

και αναλόγως αυτό το αναγνωριστικό, η ροή του εκτελέσιμου μπορεί να αλλάξει. Επικοινωνία μεταξύ διεργασιών μπορεί να γίνει είτε μεμονομένα από διεργασία προς διεργασία (point-to-point) ή σε συλλογική μορφή (collective).

Το MPI αρχικά ξεκίνησε από μια μικρή ομάδα ερευνητών που επιθυμούσαν μία τυποποιημένη και φορητή λύση για την επικοινωνία συστημάτων που ανήκουν σε συστοιχίες. Οι συζητήσεις αυτές προχώρησαν στην προτοτυποίηση της πρώτης έκδοσης, MPI 1.0 και την δημιουργία του MPI Forum, της ομάδας ανάπτυξης και συντήρησης αυτού του προτύπου η οποία είναι ανοιχτή σε όλα τα μέλη του HPC χώρου. Από τότε, αναπτύχθηκαν και αναπτύζονται νέες εκδόσεις με περισσότερες λειτουργίες. Η ίδια η υλοποίηση βιβλιωθηκών MPI γίνεται από διαφορετικούς δημιουργούς και μπορεί να είναι βιβλιοθήκες ανοιχτού κώδικα ή ιδιόκτητες εκδοχές κλειστού κώδικα. Γνωστές υλοποιήσεις ανοιχτού κώδικα είναι η OpenMPI και η MPICH, με την δεύτερη να έχει και ξεχωριστές εκδοχές κλειστού κώδικα για χρήση σε συστήματα ειδικών πωλητών όπως η Intel ή η Cray. Είναι σημαντικό να σημειωθεί πως η έκδοση της υλοποίησης του MPI μπορεί να μην ταυτίζεται με την έκδοση του MPI που υποστηρίζεται. Για παράδειγμα, η έκδοση OpenMPI 5.0 υποστηρίζει όλες τις δυνατότητες του MPI-3.1 και έχει μερική υποστήριξη του MPI-4.0.

Λόγω της φορητής φύσης του, η χρήση διαφορετικών βιβλιοθηκών MPI δεν επηρεάζει το αποτέλεσμα της εκτέλεσης, προσφέροντας καθολικότητα στην προγραμματιστική εμπειρία. Όμως, κάθε έκδοση μπορεί να αναλογεί σε διαφορετικές επιδόσεις, λόγω διάφορων βελτιστοποιήσεων που μπορεί να έχουν πάρει μέρος ώστε να τρέχουν βέλτιστα σε συγκεκριμένα συστήματα συστοιχιών. Παράδειγμα του παραπάνω φαινομένου αποτελεί η εταιρεία Cray, πλέον παράρτημα της Hewlett Packard, η οποία αποτελεί από τις πιο σημαντικές εταιρείες σχεδιασμού και κατασκευής υπερυπολογιστών. Η ίδια υλοποιεί ξεχωριστή έκδοση MPI, βασισμένη στο MPICH, η οποία προορίζεται για χρήση σε συστοιχίες Cray.

1.3.2 Τοπολογίες Διεργασιών

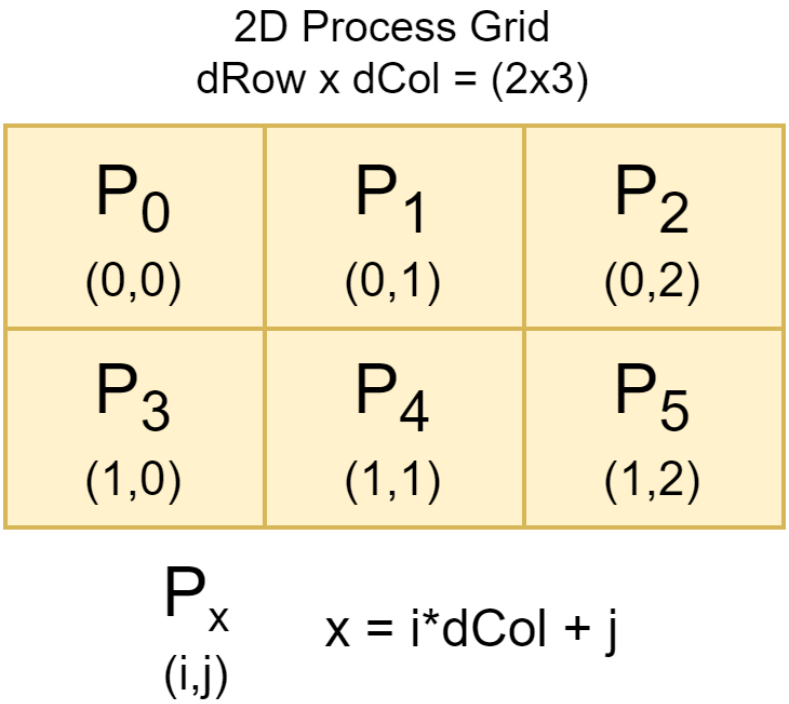
Κατά την εκκίνηση ενός MPI προγράμματος, όλες οι διεργασίες βρίσκονται σε έναν Communicator. Προϋπόθεση για την ανταλλαγή μηνυμάτων είναι οι διεργασίες να βρίσκονται σε κοινό communicator. Ο χρήστης μπορεί να δημιουργήσει καινούργιους communicators, που αποτελούν υποσύνολα του αρχικού ώστε να περιορίσει την επικοινωνία μεταξύ συγκεκριμένων διεργασιών. Η διαδικασία αυτή ευνοεί αρκετά την συλλογική επικοινωνία μεταξύ υποομάδας διεργασιών.

Υπάρχουν 2 τύποι communicator, οι intra-communicators που ορίζονται ως μια συλλογή διεργασιών που μπορούν να ανταλλάξουν μηνύματα μεταξύ τους ή να συμμετάσχουν σε συλλογικές μορφές επικοινωνίας, και οι inter-communicators που χρησιμοποιούνται για την αποστολή μηνυμάτων μεταξύ διεργασιών που ανοίκουν σε ασύνδετα σύνολα διεργασιών.

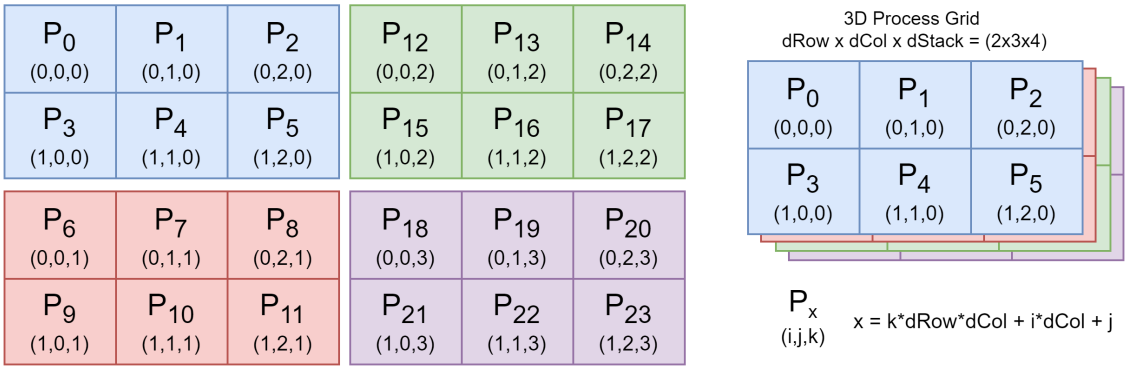
Σε πολλές εφαρμογές, η απλή περιγραφή του communicator, δηλαδή ενός συνόλου διεργασιών, δεν είναι αρκετή. Με σκοπό την προσθήκη περαιτέρω πληροφορίας σε ένα σύνολο διεργασιών, χρησιμοποιούμε εικονικές τοπολογίες (virtual topologies). Το MPI προσφέρει 2 διαφορετικά ήδη οργάνωσης, τις τοπολογίες γράφων και τις καρτεσιανές τοπολογίες. Οι δεύτερες χρησιμοποιούνται κυρίως σε εφαρμογές που χρησιμοποιούν κατανεμημένους πίνακες, όπως αυτές που θα αναφέρουμε αργότερα. Ιδανικά, η ταύτιση της εικονικής τοπολογίας με την πραγματική τοπολογία δικτύου της συστοιχίας επιτρέπει πλήρη αξιοποίηση όλων των δυνατοτήτων που μπορέι να προσφέρει η τελευταία, μεγιστοποιώντας έτσι την επίδοση της

επικοινωνίας.

Στις καρτεσιανές τοπολογίες, οι διεργασίες οργανώνονται σε καρτεσιανά πλέγματα, χωρίς περιορισμούς στην επιλογή των διαστάσεων. Οι διαστάσεις των πλεγμάτων εξαρτώνται από την φύση του αλγορίθμου και τις ανάγκες επικοινωνίας μεταξύ διεργασιών. Οι κατανεμημένοι αλγόριθμοι επίλυσης προβλημάτων γραμμικής άλγεβρας καθώς και υπολογισμοί stencil αξιοποιούν συνήθως δισδιάστατα ή τρισδιάστατα πλέγματα. Στα σχήματα 1.10 παρουσιάζονται τέτοια πλέγματα.



Σχήμα 1.9: 2D Πλέγμα Διεργασιών Διαστάσεων 2x3

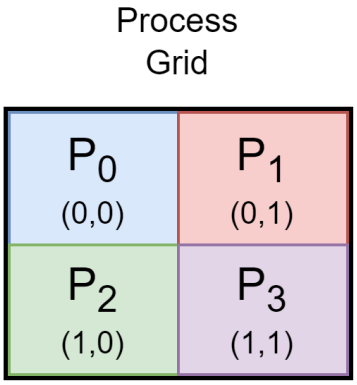


Σχήμα 1.10: 3D Πλέγμα Διεργασιών Διαστάσεων 2x3x4

Μία χρήσιμη δυνατότητα που προσφέρουν τα καρτεσιανά πλέγματα είναι οι λειτουργίες shift με τις οποίες μια διεργασία μπορεί να αναγνωρίσει τις διεργασίες-γείτονες του ως προς μια επιθυμητή κατεύθυνση. Η λειτουργία αυτή χρησιμοποιείται συχνά σε υπολογισμούς stencil στους οποίους η επικοινωνία και ανταλλαγή δεδομένων με τις διεργασίες γείτονες είναι απαραίτητη.

1.3.3 Κατανομή Κανονικών Δομών

Το προγραμματιστικό μοντέλο SPMD αποτελεί μια πιο γενικευμένη περίπτωση τεχνικής παραλληλισμού δεδομένων, στην οποία, μοιράζονται (ισότιμα) τα δεδομένα του προβλήματος σε όλες τις διεργασίες που συμμετέχουν στον αλγόριθμο. Συγκεκριμένα, θα αναφερθούμε στον διαμοιρασμό δισδιάστατων πινάκων, μιας δομής δεδομένων που συνατάται πολύ συχνά σε κατανεμημένους αλγορίθμους. Για ευκολότερη επεξήγηση των τεχνικών κατανομής δεδομένων, θεωρούμε το δισδιάστατο πλέγμα διεργασιών του σχήματος 1.11 ως τους συμμετάχοντες στην κατναομή.

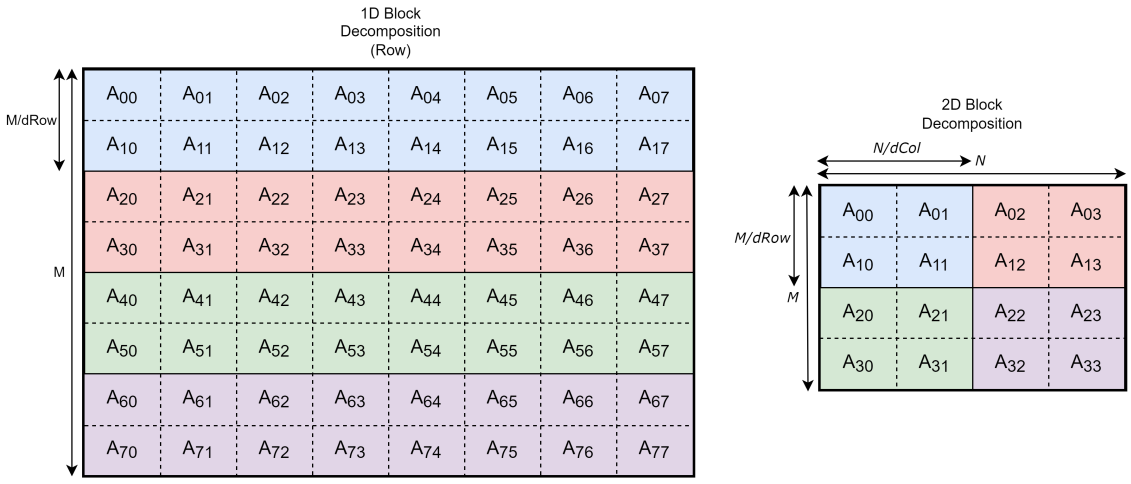


Σχήμα 1.11: Πλέγμα Διεργασιών για τις κατανομές

Block/Sequential

Στην sequential κατανομή, ένας πίνακας χωρίζεται βάσει των διαστάσεων του πλέγματος διεργασιών. Η κατανομή μπορεί να γίνει είτε μονοδιάστατα (γραμμές/στήλες) ή δισδιάστατα (block). Στην μονοδιάστατη κατανομή, κάθε διεργασία λαμβάνει αριθμό γραμμών/στηλών/διεργασίες γραμμών/στηλών. Στην δισδιάστατη κατανομή, ο αρχικός πίνακας χωρίζεται σε block οσα και ο αριθμός διεργασιών διαστάσεων αριθμός γραμμών/διεργασίες γραμμών:αριθμός στηλών/διεργασίες στηλών. Στο σχήμα 1.12 παρουσιάζονται τα 2 αυτά ήδη κατανομών. Τα χρώματα ταυτίζονται με τις διεργασίες οι οποίες θα λάβουν τα σκιασμένα δεδομένα.

Οι sequential κατανομές διανέμουν δεδομένα με συνεχή τρόπο, προσφέροντας τοπικότητα δεδομένων, αυξάνοντας σημαντικά την επίδοση σε αλγορίθμους που αξιοποιούν γειτονικά δεδομένα. Η απλοϊκή φύση αυτής της κατανομής επιφέρει αρνητικές συνέπειες στον μέγιστο παραλληλισμό που μπορεί να επιτευχθεί. Η απότομη αύξηση των διαστάσεων του προβλήματος μπορεί να μειώσει δραστικά την επίδοση, καθώς αγνοείται πλήρως η δυνατότητα των δεδομένων να χωράνε στις ταχύτερες κρυφές μνήμες των επεξεργαστών. Επίσης, ειδικά σε ετερογενή συστήματα , είναι αρκετά περιοριστική κατανομή για την εξισορρόπιση φόρτου, αφού ο διαμοιρασμός γίνεται βάσει του συνόλου διεργασιών που συμμετάσχουν στην εκτέλεση και δεν συμπεριλαμβάνονται υπόψιν οι περαιτέρω επιταχυντές που μπορεί να υπάρχουν στο σύστημα.



Σχήμα 1.12: Block Decomposition

Cyclic

Οι κυκλικές κατανομές διαμοιράζουν δεδομένα με round-robin τρόπο σε όλες τις διεργασίες. Αντίστοιχα με τις block, μπορούν να γίνουν είτε μονοδιάστατα (στοιχεία/γραμμές/στήλες) ή δισδιάστατα (block-cyclic). Τα σχήματα 1.13 και 1.14 επεξηγούν γραφικά την διαδικασία διανομής δεδομένων.

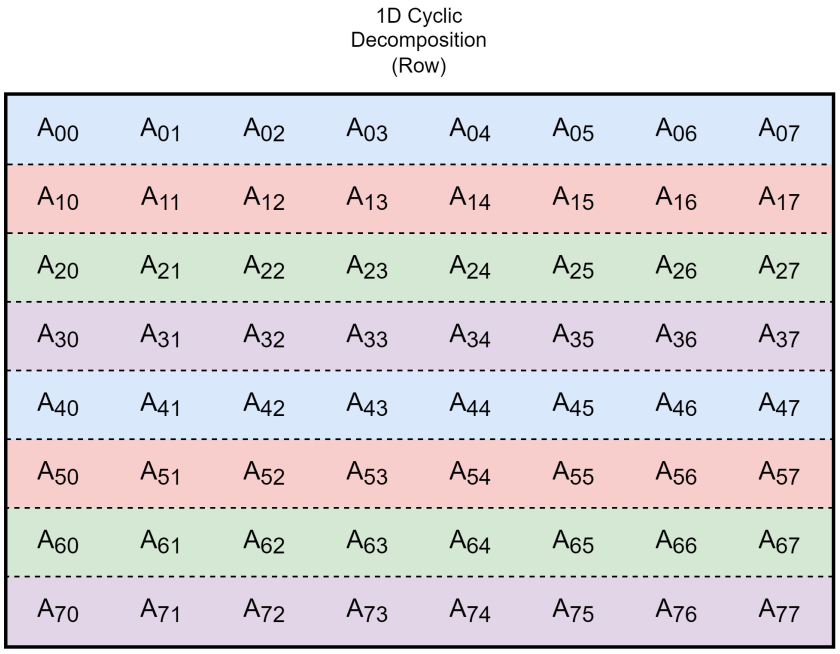
Οι μονοδιάστατες μορφές είναι αρκετά περιοριστικές. Αν και προσφέρουν αρκετά καλές δυνατότητες για μεγιστοποίηση παραλληλισμού, το μοτίβο πρόσβασης δεδομένων που επιβάλλουν μειώνει την επίδοση κατά την εκτέλεση καθώς τα γειτονικά δεδομένα βρίσκονται σε άλλη υπολογιστκή μονάδα και απαιτείται επικοινωνία μεταξύ διεργασιών για την ανταλλαγή τους.

Η δισδιάστατη μορφή κατέχει πρωταρχική θέση, καθώς χρησιμοποιείται από πληθώρα κατανεμημένων εφαρμογών, ειδικά στον χώρο του επιστημονικού προγραμματισμού. Συγκεκριμένα σε εφαρμογές γραμμικής άλγεβρας, χρησιμοποιούνται στην κατανομή πυκνών πινάκων, προσφέροντας εξισορρόπιση φορτίου υπολογισμών μεταξύ διεργασιών. Περαιτέρω, η δυνατότητα παραμετροποίησης των διαστάσεων του block M_b x N_b ευνοούν τον σχεδιασμό των αλγορίθμων γύρω από τις σύνθετες ιεραρχίες μνήμες των επεξεργαστών, δημιουργώντας cache-friendly συνθήκες.

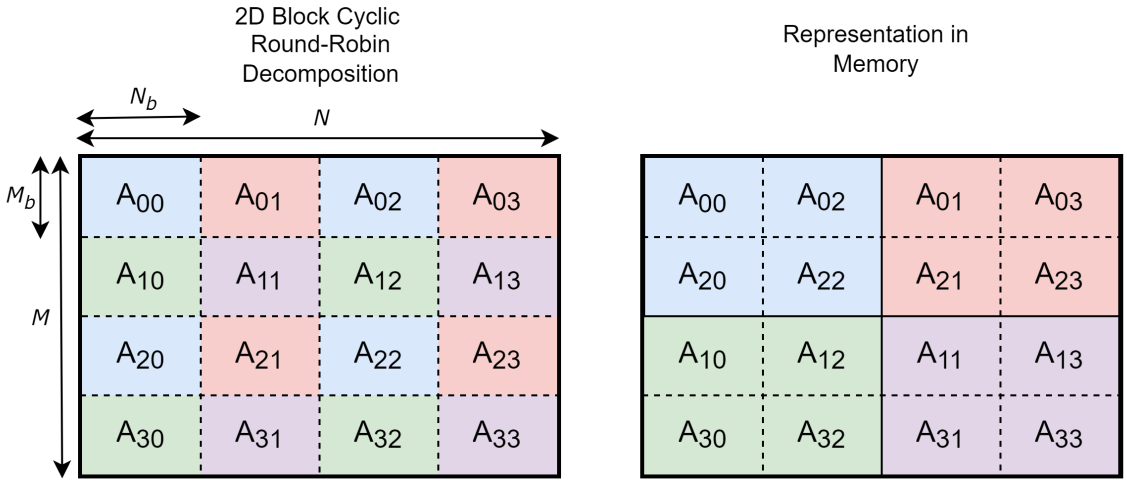
1.4 Οι GPUs στον χώρο του HPC

Η λίστα TOP500 [10] αξιολογεί και βαθμολογεί τους 500 πιο ισχυρούς μη-κατανεμημένους υπερυπολογιστές στον κόσμο. Η διαδικασία αυτή ξεκίνησε το 1993 και συνεχίζει ως σήμερα, ανανεώνοντας την λίστα 2 φορές τον χρόνο, κάθε Ιούνιο και Νοέμβριο. Η αξιολόγηση γίνεται χρησιμοποιώντας το HPL (High-Performance Linpack), ένα εργαλείο συνθετικών μετροπρογραμμάτων που εξετάζει την επίδοση των συστημάτων σε υπολογιστικά, επικοινωνιακά και μεικτά φορτία [11].

Από τις αρχές του 2010, παρουσιάζεται άνοδος στην χρήση επιταχυντών, και συγκεκριμένα, GPUs στα καινούργια υπερυπολογιστικά συστήματα. Το 2019, από τα 500 συνολικά



Σχήμα 1.13: 1D Cyclic Decomposition (Row)

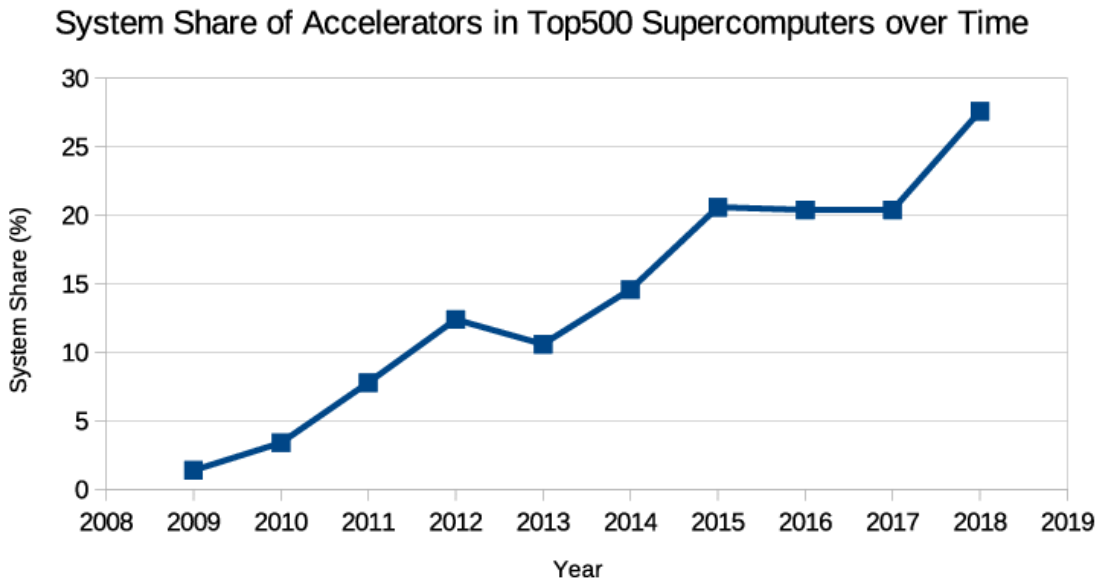


Σχήμα 1.14: 2D Block Cyclic Decomposition

μηχανήματα, τα 135 περιείχαν επιταχυντές. Το στατιστικό αυτό παραμένει αρκετά σταθερό τα επόμενα χρόνια, εώς και σήμερα. Συγκεκριμένα για τον Ιούνιο του 2024, από τα 144 συστήματα που περιείχαν επιταχυντές, τα 76 χρησιμοποιούσαν κάποια έκδοση της Nvidia A100. [12]

Meluxina

Το Meluxina [14], στο οποίο έγινε η πλειοψηφία της υλοποίησης και της αξιολόγησης των πρακτικών κομματιών αυτής της εργασίας, είναι ένας υπερυπολογιστής με βάση το Λουξεμβούργο που αποτελείται από ετερογενείς κόμβους. Περιέχει 4 διαφορετικά είδη κόμβων, με τα 2 από αυτά να υποστηρίζουν επιταχυντές τύπου GPU ή FPGA. Συνολικά, έχει 200 κόμβους GPU, με τον καθένα να περιέχει 4 Nvidia A100 GPUs. Το θεωρητικό μέγιστο υπολογιστικής επίδοσης του υπερυπολογιστή αγγίζει τα 20 PetaFLOP/s, ενώ μόνο οι GPU του συστήματος αναλογούν



Σχήμα 1.15: Ποσοστά συμμετοχής Accelerator στην λίστα Top 500 [13]

στο 78% της επίδοσης, καθώς αγγίζουν τα 15.6 PetaFLOP/s σε θεωρητικό μέγιστο. Πρακτικά, οι μετρήσεις HPL αξιολογούν το σύστημα σε peak 15.29 PetaFLOP/s ενώ το μέγιστο υπολογίζεται στα 10.52 PetaFLOP/s.

1.4.1 GPU-Aware MPI

Οι μοντέρνες βιβλιοθήκες MPI οι οποίες προσφέρουν άμεση υποστήριξη για GPUs ονομάζονται GPU-Aware. Συγκεκριμένα για συσκευές Nvidia, αποκαλούνται CUDA-Aware. Χρησιμοποιώντας GPU-Aware βιβλιοθήκες, επιτρέπεται κατά την κλήση συναρτήσεων μεταφοράς δεδομένων MPI, να περαστεί απευθείας ως όρισμα θέση μνήμης η οποία είναι δεσμευμένη στην συσκευή. Έτσι, επιτυγχάνεται άμεση επικοινωνία μεταξύ GPU διαφορετικών κόμβων χωρίς να χρειάζεται να περνάει από την μνήμη των CPU του κάθε κόμβου, αυξάνοντας τις επιδόσεις επικοινωνίας.

1.4.2 NVIDIA Collective Communications Library (NCCL)

Το NCCL (προφέρεται ”Nickel”), είναι ένα εργαλείο επικοινωνίας που προσφέρει η Nvidia για συστήματα πολλαπλών κόμβων και πολλαπλών GPU (multi-node, multi-GPU). Υλοποιεί σημαντικές ρουτίνες συλλογικής καθώς και point-to-point, όπως και το MPI και μπορεί να ενσωματωθεί απευθείας σε εφαρμογές MPI, αξιοποιώντας βέλτιστα τις υποδομές επικοινωνίας του συστήματος, ειδικά αυτές που είναι κατασκευασμένες από την Nvidia (NVLink, Mellanox κ.λ.π.). Προσφέρει σύγχρονες και ασύγχρονες εκδοχές επικοινωνίας, εκμεταλλευόμενοι τα CUDA Streams και τις δυνατότητες συγχρονισμού τους. Αν και το NCCL προσφέρει λειτουργίες ίδιες με το MPI, δεν μπορεί να θεωρηθεί υλοποίηση MPI. Σημαντική διαφορά επίσης είναι ότι δεν προσφέρει δυναμική point-to-point επικοινωνία, μια λειτουργία που είναι πολύ σημαντική για ορισμένους κατανεμημένους αλγορίθμους.

1.5 Βιβλιοθήκες BLAS

Το BLAS (Basic Linear Algebra Subprograms) [15] είναι ένα πρότυπο ρουτινών γραμμικής άλγεβρας για την εκτέλεση βασικών πράξεων μεταξύ διανυσματικών μεγεθών. Λόγω της φορητής φύσης αυτών των ρουτινών, η BLAS χρησιμοποιείται για την υλοποίηση σύνθετων λογισμικών γραμμικής άλγεβρας όπως το LAPACK.

Αναλόγως την αρχιτεκτονική της υπολογιστκής μονάδας, υπάρχουν αρκετές βελτιστοποιημένες βιβλιοθήκες BLAS. Για CPUs οι πιο δημοφιλείς είναι η OpenBLAS [16], η οποία περιέχει βελτιστοποιήσεις για πολλαπλές οικογένεις επεξεργαστών καθώς και για πληθώρα ISA (Instruction Set Architecture) και η Intel Math Kernel Library (iMKL) [17] η οποία υποστηρίζει και εξειδικευμένους επιταχυντές της Intel καθώς και ενσωματωμένους επιταχυντές γραφικών.

Για επεξεργαστές γραφικών, οι πιο διάσημες είναι η cuBLAS [18] και η rocBLAS [19], αντίστοιχα για συσκευές NVIDIA και AMD. Η καθεμιά είναι βελτιστοποιημένη ώστε να εκμεταλλεύεται όλες τις δυνατότητες των επεξεργαστών (NVIDIA Tensor Cores, AMD RDNA Compute Units) με σκοπό την αύξηση των επιδόσεων των βασικών πυρήνων.

1.5.1 Επίπεδα Ρουτινών

Ανάλογα με τις διαστάσεις των εισόδων μιας ρουτίνας, η BLAS χωρίζεται σε τρία επίπεδα.

Επίπεδο 1

Το πρώτο επίπεδο ρουτινών BLAS [20] εκτελεί βαθμωτές πράξεις και πράξεις μεταξύ διανυσμάτων. Ο βασικός πυρήνας εκτελεί μια γενικευμένη διανυσματική πρόσθεση.

$$y = \alpha x + y$$

η οποία ονομάζεται ”axpy”, λόγω της μαθηματικής σχέσης ”a x plus y”. Στο ίδιο επίπεδο, περιέχεται και το εσωτερικό γινόμενο (dot product).

Επίπεδο 2

Το δεύτερο επίπεδο [21] περιέχει ρουτίνες που εκτελούν πράξεις μεταξύ διανυσμάτων και πινάκων. Ο βασικός πυρήνας εκτελεί το γενικευμένο γινόμενο πίνακα με διάνυσμα (generalized matrix-vector multiplication, ”gemv”). Σε μαθηματική μορφή:

$$y = \alpha Ax + \beta y$$

όπου $\boldsymbol{\alpha}$,$\boldsymbol{\beta}$: βαθμωτά μεγέθη, $\mathbf{x}$,$\mathbf{y}$ διανύσματα, $\mathbf{A}$ πίνακας.

Επίπεδο 3

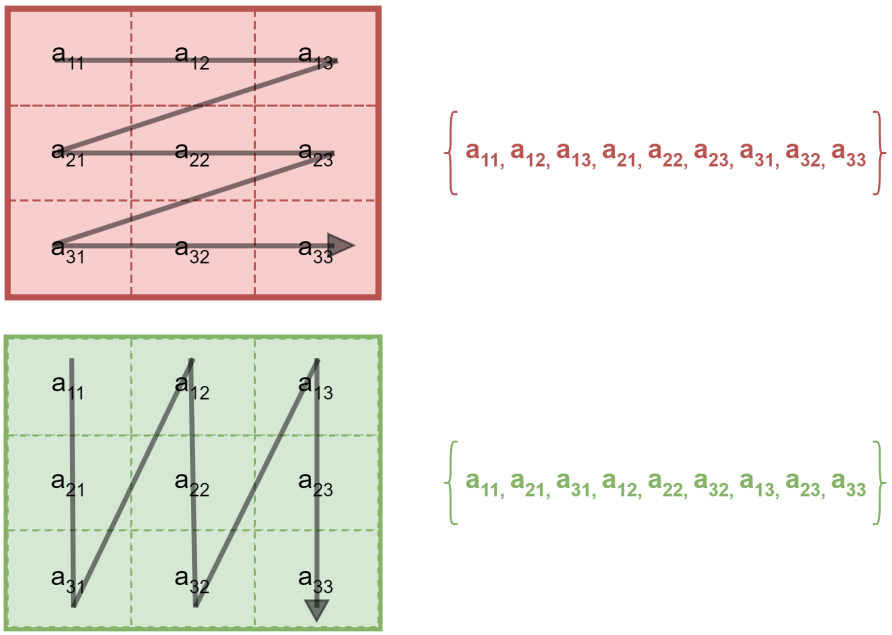
Το τρίτο και τελευταίο επίπεδο [22] περιέχει ρουτίνες που εκτελούν πράξεις μεταξύ πινάκων. Ο βασικός πυρήνας εκτελεί το γενικευμένο γινόμενο πίνακα με πίνακα (general matrix multiplication, ”gemm”). Σε μαθηματική μορφή:

$$C = \alpha AB + \beta C$$

Όπου **α**,**β**: βαθμωτά μεγέθη, **A**,**B**,**C**: πίνακες. Οι πίνακες **A**, **B** μπορούν να είναι, προεραιτικά, είτε ανάστροφοι ή συζυγείς (Hermitian adjoint), αν αναφερόμαστε σε μιγαδικούς πίνακες.

Αναπαράσταση Πινάκων Στην Μνήμη

Οι παραδοσιακές βιβλιοθήκες που υποστηρίζουν το πρωτόκολλο BLAS δέχονται διανύσματα και πίνακες ως ορίσματα σε μορφή μονοδιάστατων δεικτών που ”δείχνουν” στην πρώτη θέση του πίνακα. Για να είναι εφικτή η αποθήκευση πολυδιάστατων διανυσμάτων σε γραμμικές μνήμες, χρησιμοποιούνται 2 μέθοδοι αποθήκευσης, κατά γραμμές (row-major) ή κατά στήλες (column-major).



Σχήμα 1.16: Αναπαράσταση πινάκων στην μνήμη

Η θέση του στοιχείου (i,j) όπου i η γραμμή και j η στήλη σε έναν πίνακα A, δίνεται από την σχέση $(j + i * ld)$ στην αποθήκευση κατά γραμμές και $(i + j * ld)$ στην αποθήκευση κατά στήλες. Η τιμή ld ονομάζεται leading dimension και αποτελεί την βασική διάσταση του πίνακα. Ισχύει ότι:

$$ld = \begin{cases} \text{αριθμός στηλών}, & \text{στην αποθήκευση κατά γραμμές} \\ \text{αριθμός γραμμών}, & \text{στην αποθήκευση κατά στήλες} \end{cases}$$

Μερικές βιβλιοθήκες, όπως η OpenBLAS, αφήνουν στον χρήστη την επιλογή για το πως είναι αποθηκευμένοι οι πίνακες στην μνήμη. Συμβατικά, η εκτέλεση πυρήνων BLAS σε CPUs ευνοείται από αποθήκευση κατά γραμμές ενώ σε GPUs από αποθήκευση κατά στήλες. Καθώς το έργο αυτό παρουσιάζει εκτέλεση πολλαπλασιασμού πινάκων σε επεξεργαστές γραφικών, η μορφή αποθήκευσης που θα χρησιμοποιηθεί κατά κόρων είναι κατά στήλες, όπως υποστηρίζει η βιβλιοθήκη cuBLAS.

1.5.2 LAPACK

Η **LAPACK** [23] είναι βιβλιοθήκη-λογισμικό η οποία πρσοφέρει ρουτίνες για επίλυση προβλημάτων γραμμικής άλγεβρας σε συστήματα κοινής μνήμης. Χρησιμοποιείται για την επίλυση αλγεβρικών προβλημάτων όπως στην επίλυση γραμμικών εξισώσεων, εύρεση ιδιοτιμών πινάκων καθώς και για παραγοντοποίηση πινάκων χρησιμοποιώντας γνωστές αποσυνθέσεις όπως LU, QR, Cholesky ή Schur.

Ο αρχικός λόγος σχεδίασης του LAPACK ήταν για να καλύψει τους περιορισμούς που είχαν προηγούμενα λογισμικά όπως το EISPACK ή το LINPACK. Τα τελευταία, αρχικά σχεδιασμένα για συστήματα με επεξεργαστές χωρίς σύνθετη ιεραρχία μνήμης, είχαν μειωμένες επιδόσεις σε μοντέρνους επεξεργαστές καθώς τα μοτίβα πρόσβασης σε δεδομένα που χρησιμοποιούσαν δεν ήταν φιλικά ως προς τις κρυφές μνήμες των πιο μοντέρνων επεξεργαστών. Η LAPACK προσφέρει μια βελτιστοποιημένη λύση σε αυτό το πρόβλημα καθώς βασίζει την εκτέλεση των συναρτήσεων σε κλήσεις BLAS κατά το μέγιστο δυνατό. Η φορητή φύση των ρουτινών BLAS σημαίνει ότι η βελτιστοποίηση των ίδιων για μια συγκεκριμένη αρχιτεκτονική επαρκούσε για την αύξηση της επίδοσης σε σημαντικό βαθμό, χωρίς να χρειαστεί σχεδιασμός εκ νέου των επιλυτών.

1.5.3 Parallel BLAS

Η PBLAS (Parallel Basic Linear Algebra Subprograms)[24] είναι η έκδοση της BLAS για εκτέλεση σε συστήματα κατανεμημένης μνήμης. Αντίστοιχα με την BLAS, υπάρχουν 3 επίπεδα ρουτινών που συμπίπτουν πλήρως με την προυπάρχουσα.

Καθώς η συλλογή αυτή αναφέρεται σε πίνακες κατανεμημένης μορφής, κάθε πίνακας, αντί για μια απλή διεύθυνση μνήμης, συνοδεύεται από έναν περιγραφέα (matrix descriptor). Η PBLAS θεωρεί ότι οι πίνακες είναι ήδη κατανεμημένοι στις μνήμες των συστημάτων που θα εκτελέσει την ρουτίνα, σε μορφή 2D-Block-Cyclic με Round-Robin τρόπο.

1.5.4 ScaLAPACK

Η **ScaLAPACK** [25] είναι η κατανεμημένη μορφή της **LAPACK**. Χρησιμοποιεί το PBLAS σαν υπολογιστικό backend και το **BLACS** (Basic Linear Algebra Communication Subprograms) [26] σαν επικοινωνιακό backend. Το δεύτερο, αναπτύχθηκε την εποχή όπου το MPI δεν αποτελούσε μονόδρομο στον προγραμματισμό συστοιχιών καθώς υπήρχαν άλλα προγραμματιστικά μοντέλα διαθέσιμα, όπως το PVM [27] ή το MPL, τα οποία είναι πλέον απαρχαιωμένα. Επέτρεπε στον χρήστη να διαλέξει εκείνος το επικοινωνιακό εργαλείο που επιθυμούσε να χρησιμοποιήσει, προσφέροντας μια backend-agnostic λύση.

1.6 Γενικός Πολλαπλασιασμός Πίνακα-Πίνακα

Ο γενικός πολλαπλασιασμός πίνακα-πίνακα αποτελεί τον βασικό πυρήνα του 3ου επιπέδου BLAS και καθιστάται ως ο πιο σημαντικός πυρήνας γραμμικής άλγεβρας που μπορεί να προσφέρει μια βιβλιοθήκη BLAS. Σε τεχνική αναφορά του Berkeley [28], παρουσιάζεται ως ένας από τους “7 νάνους” της έρευνας στον τομέα του παράλληλου προγραμματισμού καθώς χρησιμοποιείται ευρέως σε πολλαπλές εφαρμογές γραμμικής άλγεβρας και αριθμητικής

ανάλυσης, από αποσύνθεση πινάκων (LU, QR, Cholesky decomposition) στην μέθοδο Gauss-Seidel για την επίλυση γραμμικών συστημάτων. Για συντομία, χρησιμοποιείται η λέξη "**gemm**" (Generalized Matrix Multiplication) και οι υπόλοιπες μελλοντικές αναφορές θα γίνονται ως εξής.

Η μαθηματική περιγραφή της πράξης ειναι η παρακάτω:

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

όπου κάθε στοιχείο του πίνακα C υπολογίζεται ως το εσωτερικό γινόμενο της ι-οστής γραμμής του πίνακα A και της j-οστής στήλης του πίνακα B:

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

1.6.1 Σειριακή Έκδοση

Η απλή μορφή του αλγορίθμου είναι η κλασσική εκτέλεση με τις τριπλά φωλιασμένες επαναλήψεις των διαστάσεων των πινάκων A,B και C. Στην μοφφή που περιγράφεται από το BLAS, η πράξη δέχεται ως είσοδο τρεις πίνακες A, B, C διαστάσεων (MxK), (KxN) και (MxN) καθώς και τις τιμές κλιμάκωσης α,β . Η έξοδος είναι ο πίνακας C.

Αλγόριθμος 1: Πολλαπλασιαμσός Πίνακας με Πίνακα(GEMM)

Δεδομένα Εισόδου: Πίνακας A[M][K], Πίνακας B[K][N], Πίνακας C[M][N], α,β

Έξοδος: Πίνακας C[M][N]

```
1 for i < 0; i < M; i++ do
2     for j < 0; j < N; j++ do
3         tempSum = 0;
4         for k < 0; k < K; k++ do
5             tempSum+ = α * A[i][k] * B[k][j];
6         end
7         C[i][j] = tempSum + β * C[i][j];
8     end
9 end
```

Η πολυλπλοκότητα του αλγορίθμου είναι $\Theta(M * N * K)$, όσο και τα όρια επανάληψης δηλαδή και για τετραγωνικούς πίνακες απλοποιείται σε $\Theta(N^3)$. Προσπάθειες για καλύτερη πολυπλοκότητα σε σειριακό αλγόριθμο κατέληξαν στον αλγόριθμο του Strassen [29], ο οποίος υποθέτει τετράγωνους πίνακες και τους χωρίζει σε τέσσερις $n/2xn/2$ υποπίνακες. Ο τελικός πίνακας προκύπτει με τον πολλαπλασιασμό 7 υποπινάκων, καταλήγοντας σε πολυπλοκότητα $O(n^{log_2 7})$. Περαιτέρω βελτιώσεις στην πολυπλοκότητα συνεχίζουν να πραγματοποιούνται, με την ελάχιστη δυνατή πολυπλοκότητα να βρίσκεται στο $O(n^{2.371552})$ [30]. Αυτοί οι αλγόριθμοι ονομάζονται galactic. Αν και καταφέρνουν ασυμπτοματικά να προσφέρουν καλύτερη επίδοση, πρακτικά δεν χρησιμοποιούνται λόγω προβλημάτων υλοποίησης ή γενικών προβλημάτων όπως έλλειψη δυνατότητας παραλληλοποίησης.

1.6.2 Υπολογισμός με χρήση GPU

Παρατηρούμε ότι ο σειριακός αλγόριθμος αποτελείται από πλήρως ανεξάρτητους υπολογισμούς. Για τον πίνακα C, ο υπολογισμός κάθε στοιχείου του C_{ij} δεν έχει εξάρτησει κανενός άλλου στοιχείου του πίνακα C. Ο αλγόριθμος αυτός ανήκει στην κατηγορία των "γελοία" παράλληλων και είναι ιδανικός για εκτέλεση σε επεξεργαστή γραφικών.

Ο γενικός αλγόριθμος εκτέλεσης σε GPU παρουσιάζεται στο 2. Κάθε νήμα χρησιμοποιώντας το αναγνωριστικό του και το αναγνωριστικό του block που ανήκει, αναγνωρίζει την γραμμή και την στήλη του στοιχείου C που πρέπει να υπολογίσει. Έπειτα, υπολογίζει το εσωτερικό γινόμενο της ίδιας γραμμής του A και της ίδιας στήλης του B, τοποθετώντας την τιμή αυτή στην θέση του C. Συνολικά, δρομολογούνται m x n νήματα.

Αλγόριθμος 2: Πολλαπλασιασμός Πίνακα με Πίνακα σε GPU

Δεδομένα Εισόδου: Πίνακας A[M][K], Πίνακας B[K][N], Πίνακας C[M][N], α, β
Έξοδος: Πίνακας C[M][N]

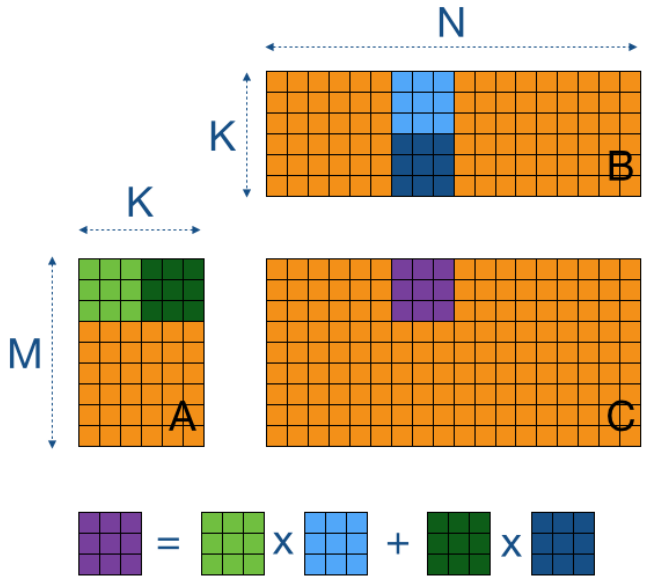
1 $row = threadIdx.y + blockIdx.y * BlockDim.y$;
2 $column = threadIdx.x + blockIdx.x * blockDim.x$;
3 $tempSum = 0$;
4 **for** *k = 0; k < K; k++* **do**
5 $\quad tempSum+ = \alpha * A[row][k] * B[k][column]$;
6 **end**
7 $C[row][column] = tempSum + \beta * C[row][column]$;

Οι προσβάσεις στην μνήμη που γίνονται από κάθε νήμα είναι $2 * K + 1$, αφού διαβάζεται ένα στοιχείο των πινάκων A και B για κάθε επανάληψη του βρόγχου συν η τελευταία ανάγνωση του πίνακα C πριν το γράψιμο του αποτελέσματος. Συνολικά, για τα M x N νήματα, πραγματοποιούνται $2 * M * N * K$ πράξεις και γίνονται συνολικά άλλες τόσες προσβάσεις στην μνήμη. Ο αλγόριθμος αυτός σίγουρα αποτελεί βελτίωση της σειριακής έκδοσης, αλλά γίνονται πολλαπλές προσβάσεις στην κύρια μνήμη της GPU, για τα ίδια δεδομένα, περιορίζοντας έτσι την επίδοση του.

1.6.3 Χρήση Tiling

Τα προβλήματα της προηγούμενης υλοποίησης στοχεύει να επιλύσει η χρήση tiles στους υπολογισμούς. Με σκοπό την μείωση των προσβάσεων στην μνήμη, κατανέμουμε τους αρχικούς πίνακες σε μικρότερους υποπίνακες που ονομάζονται tiles. Τελικός στόχος είναι να δημιουργήσουμε tiles μικρών διαστάσεων ώστε να αξιοποιηθεί πλήρως η διαμοιραζόμενη μνήμη των thread block.

Στον αλγόριθμο, αντί να μετράμε τις γραμμές και τις στήλες των πινάκων σε στοιχεία, τις μετράμε σε tiles. Όπως φάινεται και στο σχήμα 1.17, ο υπολογισμός του 3 x 3 μωβ tile του C, δηλαδή το tile $C_{0,2}$ προκύπτει από τον πολλαπλασιασμό της πρώτης γραμμής από tiles του A και της τρίτης στήλης από tiles του B. Υπενθυμίζουμε ότι χρησιμοποιούμε zero-based δεικτοδότηση. Άρα, $C_{0,2} = A_{0,0}B_{0,2} + A_{0,1}B_{1,2}$.



Σχήμα 1.17: Πολλαπλασιασμός Πίνακα με Πίνακα με την τεχνική του Tiling [31]

Αλγόριθμος 3: Πολλαπλασιασμός Πίνακα με Πίνακα σε GPU με χρήση Tiling [31]

Δεδομένα Εισόδου: Πίνακας A[M][K], Πίνακας B[K][N], Πίνακας C[M][N], α, β

Έξοδος: Πίνακας C[M][N]

```
1  __shared__ A_local[N_b][M_b];
2  __shared__ B_local[M_b][N_b];
3  tempSum = 0;
4  numberOfTiles = K/M_b;
5  for tile = 0; tile < numberOfTiles; tile++ do
6      row = blockIdx.y * blockDim.y + threadIdx.y;
7      col = tile * blockDim.x + threadIdx.x;
8      A_local[threadIdx.y][threadIdx.x] = A[row][col];
9      B_local[threadIdx.x][threadIdx.y] = B[col][row];
10     __syncthreads();
11     for x = 0; x < M_b; x++ do
12         tempSum+ = A_local[threadIdx.y][x] * B_local[x][threadIdx.x];
13     end
14     __syncthreads();
15 end
16 row = blockIdx.y * blockDim.y + threadIdx.y;
17 col = blockIdx.x * blockDim.x + threadIdx.x;
18 C[row][col] = tempSum;
```

1.6.4 Πολλαπλασιασμός Πίνακα με πολλαπλές GPUs

Η γενικότερη έρευνα εκτέλεσης πυκνών ρουτινών BLAS σε πολλαπλές GPUs χωρίστηκε σε 2 μέρη, όπου και στις δύο δόθηκε έμφαση στην εκτέλεση ρουτινών τρίτου επιπέδου. Το πρώτο μέρος έδωσε έμφαση στην εκτέλεση σε συστήματα ενός κόμβου με πολλαπλές GPUs ενώ το δεύτερο σε

συστήματα πολλαπλών κόμβων. Στην πρώτη κατηγορία, σκοπός είναι η επίτευξη της μέγιστης δυνατής επίδοσης χρησιμοποιώντας έναν μοναδικό κόμβο, ενώ στην δεύτερη η κλιμακωσιμότητα.

Οι υλοποιήσεις σε έναν κόμβο αποτελούν εφαρμογές παραλληλισμού σε κοινό χώρο διευθύνσεων και δίνουν έμφαση στην αποδοτική διαχείριση μνήμης, στην ευνοϊκή κατανομή δεδομένων και στον ταυτοχρονισμό διεργασιών μεταξύ επεξεργαστών ώστε να εξάγουν την πλήρη δυνατή επίδοση από όλες τις διαθέσιμες GPUs εντός του κόμβου.

cuBLASXt

Η πρώτη βιβλιοθήκη προέρχεται από την NVIDIA, στην μορφή του cuBLASXt [32], μιας επέκτασης της cuBLAS η οποία υποστηρίζει εκτέλεση σε πολλαπλές GPUs, με τα δεδομένα να βρίσκονται είτε στον host ή σε οποιαδήποτε από τις συσκευές. Εσωτερικά, η αποσύνθεση των πινάκων γίνεται με 2D Block Cyclic τρόπου και round-robin δρομολόγηση. Τέτοια αποσύνθεση συναντάμε σε κατανεμημένους αλγορίθμους, στους οποίους η επικοινωνία είναι (συνήθως) απαραίτητη. Στην συγκεκριμένη περίπτωση, εισάγεται περίσσια επικοινωνία η οποία περιορίζει τις επιδόσεις.

BLASX

Η BLASX [33] σχεδιάστηκε ώστε να ξεπεράσει τις επιδόσεις των ήδη υπάρχοντων βιβλιοθηκών, ειδικά της cuBLASXt. Σε αντίθεση με την στατική χρονοδρομολόγηση εργασιών που υιοθετεί το cuBLASXt, η BLASX χρησιμοποιεί δυναμική δρομολόγηση. Επίσης, υλοποιεί σε επίπεδο λογισμικού μια ιεραρχία κρυφών μνήμεων για τα tiles των πινάκων με σκοπό την αύξηση τοπικότητας δεδομένων και την μείωση κόστους κατά την μεταφορά δεδομένων.

XKBLAS

Η XKBLAS [34] επιτυγχάνει υψηλότερες επιδόσεις από τα προηγούμενα συστήματα, επιστρέφοντας σε στατική ανάλυση των εξαρτήσεων δεδομένων και δρομολόγησης εργασιών. Η ανάλυση αυτή πραγματοποιείται από το XKaapi, χρησιμοποιώντας ευρετικές μεθόδους λαμβάνοντας υπόψιν την τοπολογία των συσκευών. Σαν αποτέλεσμα, επιτυγχάνει υψηλές επιδόσεις, ειδικά συγκριτικά με την cuBLASXt και την BLASX, σε μικρά προβλήματα όπου οι μετακινήσεις των δεδομένων είναι σημαντικός περιοριστικός παράγοντας.

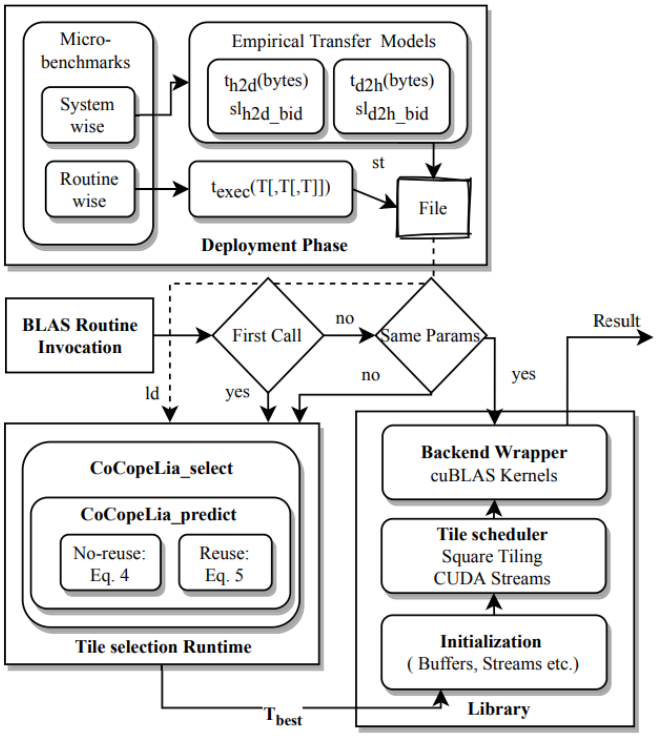
CoCoPeLia

Το CoCoPeLia (Communication-Computation Overlap Prediction for Efficient Linear Algebra on GPUs) [35] σχεδιάστηκε για βελτιστοποίηση εκτελέσεων πυρήνων BLAS σε μονές GPU, με τελικό σκοπό την βελτίωση των ήδη υπάρχοντων βιβλιοθηκών (cuBLASXt, BLASX, XKBLAS). Για την επίτευξη αποδεκτής επίδοσης, οι σημαντικές λειτουργίες που πρέπει να υποστηρίζουν οι βιβλιοθήκες είναι ο διαχωρισμός των αρχικών πινάκων σε *tiles* και την επικάλυψη μεταφορά δεδομένων με υπολογισμό.

Η cuBLASXt και η XKBLAS αφήνει την επιλογή των διαστάσεων των tiles καθαρά ως επιλογή του προγραμματιστή, οδηγώντας έτσι τον ίδιο να καταφεύγει σε δοκιμές και benchmarking ώστε να βρει το κατάλληλο μέγεθος tiles αναλόγως την εφαρμογή και το υλικό που χρησιμοποιεί. Η

BLASX θέτει εσωτερικά της βιβλιοθήκης ένα στατικό μέγεθος tiling το οποίο επιτυγχάνει καλή επίδοση κατά μέσο όρο σε διάφορα προβλήματα, χωρίς να προσαρμόζεται στα δεδομένα της κλήσης BLAS ή στις δυνατότητες της GPU.

Το CoCoPeLia, ακολουθεί μια μεθοδευμένη τεχνική βασισμένη στην μοντελοποίηση για την επιλογή μεγέθους tile. Στηριγμένο από δεδομένα προηγούμενων εκτελέσεων μικρό-μετροπρογραμμάτων (micro-benchmark), τα οποία εκτελούνται κατά την εγκατάσταση της βιβλιοθήκης, το σύστημα προβλέπει την βέλτιστη τιμή tile βασισμένο σε πραγματικά δεδομένα της εκάστοτε αρχιτεκτονικής που χρησιμοποιεί το σύστημα. Η πρόβλεψη αυτή γίνεται σε επίπεδο χρόνου εκτέλεσης, δηλαδή κατά την κλήση μιας ρουτίνας BLAS, λαμβάνοντας υπόψιν τις παραμέτρους της κλήσης. Το υπολογιστικό backend και ο συγχρονισμός των εργασιών της GPU γίνονται με εργαλεία που προσφέρει το CUDA, δηλαδή CUDA Streams για τον ταυτοχρονισμό και cuBLAS για τους υπολογισμούς.



Σχήμα 1.18: Διάγραμμα Ροής του συστήματος CoCoPeLia [35]

PARALiA

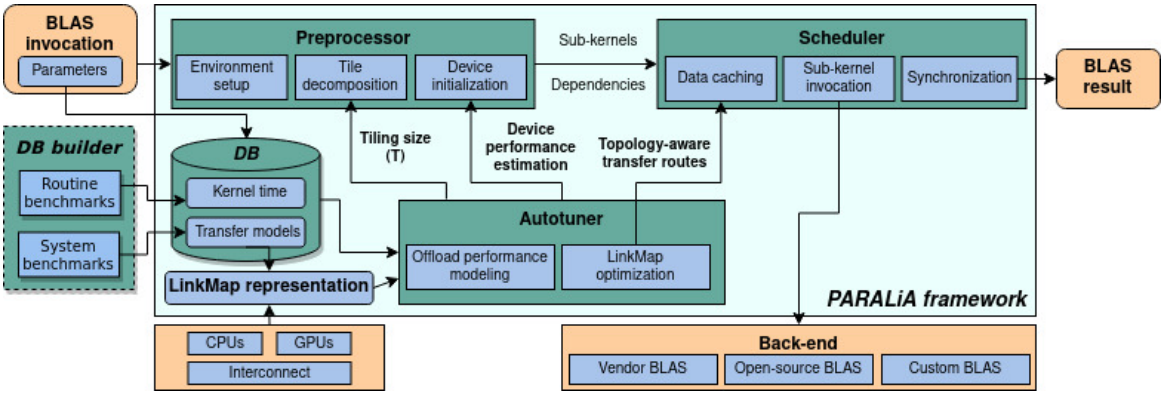
Η εξέλιξη του CoCoPeLia σε ετερογενή συστήματα πολλαπλών GPUs επιτυγχάνεται με το PARALiA (Performance Aware Runtime for Auto-tuning Linear Algebra on heterogeneous systems) [36]. Βασισμένο σε τεχνικές της προηγούμενης έκδοσης, προστίθεται το υποσύστημα LinkMap, ενός χάρτη που παρουσιάζει το bandwidth επικοινωνίας όλων των υπολογιστικών κόμβων του συστήματος. Μέσω του υποσυστήματος αυτού, εκτελώντας έναν απλό αλγόριθμο συντομότερων μονοπατιών, μπορούν να εξαχθούν τα βέλτιστα μονοπάτια μεταφοράς δεδομένων μεταξύ δύο κόμβων. Η μεταφορά αυτή μπορεί να γίνει είτε απευθείας μεταξύ των δύο κόμβων ή μέσω πεπερασμένων και ελεγχόμενων σε αριθμό βημάτων μέσα από άλλους κόμβους. Όπως

και στο CoCoPeLia, χρησιμοποιούνται μετρήσεις από micro-benchmark που εκτελούνται κατά την εγκατάσταση της βιβλιοθήκης για την μοντελοποίηση της επίδοσης και επικοινωνίας. Οι μετρήσεις αυτές γίνονται μόνο μία φορά και αποθηκεύονται τοπικά στο σύστημα.

Κατά την κλήση κάποιας ρουτίνας BLAS, το Autotuner χρησιμοποιεί τα προηγουμένως αναφερόμενα υποσυστήματα ώστε να υπολογίσει το βέλτιστο μέγεθος tile, τα ποσοστά του προβλήματος που πρέπει να αναλάβει κάθε συσκευή και τις συντομότερες διαδρομές μεταφοράς δεδομένων μεταξύ των υπολογιστικών κόμβων που χρησιμοποιούνται στο πρόβλημα. Σημειώνουμε πως αν εκτελείται κλήση BLAS σε ομογενές σύστημα, τότε ο φόρτος εργασίας κάθε συσκευής πρέπει να είναι ίσος.

Έχοντας υπολογίσει τις προηγούμενες παραμέτρους, το υποσύστημα του Preprocessor υπολογίζει τις εργασίες που πρέπει να αναλάβει κάθε συσκευή, στέλνοντας τις παραμέτρους των υπό-πυρήνων στον Scheduler. Ο Scheduler αναλαμβάνει την μεταφορά δεδομένων, την εκτέλεση των υπολογιστικών πυρήνων και τον συγχρονισμό των συσκευών. Επίσης αναλαμβάνει πιθανό caching που μπορεί να χρειαστεί, είτε για να εκτελεστεί πρόβλημα που δεν χωράει ολικά στην μνήμη των συσκευών ή για να συμβεί tile reusing, μειώνοντας τις περιττές μεταφορές δεδομένων. Το backend του Scheduler είναι βασισμένο σε λειτουργίες CUDA, όπως ακριβώς και το CoCoPeLia.

Πρακτικά, η αποσύνθεση του προβλήματος για συγκεκριμένες παραμέτρους και η χρονοδρομολόγηση των task, έχουν στατική φύση. Δεν χρησιμοποιούνται τεχνικές όπως work-stealing καθώς επηρεάζουν πολύ σημαντικά την τοπικότητα των δεδομένων που επεξεργάζεται η κάθε GPU. Παρατηρούμε ότι η φύση των προβλημάτων BLAS ευνοούνται από στατικές μεθόδους αποσύνθεσης προβλήματος.



Σχήμα 1.19: Διάγραμμα Ροής του συστήματος PARALiA [36]

PARALiA-Uncut

Η εξέλιξη του PARALiA αποστασιοποιείται σε μικρό βαθμό από την τεχνική της μοντελοποίησης και στηρίζει την βάση της σχεδίασής της σε ευρετικές τεχνικές. Η αποσύνθεση των πινάκων γίνεται με αρκετά παρόμοιο βαθμό όπως και στο αρχικό σύστημα, με μικρές διαφορές στην επιλογή μεγεθών tiling. Για την επιλογή αυτή, χρησιμοποιούνται τρεις ευρετικοί κανόνες (heuristics) ώστε να βρεθεί βέλτιστο tiling size. Επιθυμούμε το μέγεθος του tile να είναι:

1. Σωστό μέγεθος συγκριτικά με τις διαστάσεις M, N, K ώστε να μην χρειαστεί να εισάγουμε padding.

2. Αρκετά μικρό ώστε να δημιουργηθούν επαρκώς αρκετά tiles για να αυξήσουμε την δυνατότητα επικάλυψης επικοινωνίας/υπολογισμού

3. Αρκετά μεγάλο ώστε να ελαχιστοποιηθούν οι μεταφορές υπολογιστικών πυρήνων και latencies χρονοδρομολόγησης.

Για εύρεση αυτού του ιδανικού μεγέθους tiling T, χρησιμοποιείται μια συνάρτηση κόστους, η οποία μπορεί αναλυτικά να μελετηθεί στην αρχική δημοσίευση.

Το σύστημα με το οποίο πραγματοποιείται η επικάλυψη επικοινωνίας/υπολογισμού είναι βασισμένο στο κλασσικό μοντέλο ταυτοχρονισμού τριών σταδίων που έχουμε περιγράψει και σε προηγούμενες παραγράφους. Για την επίτρεψη αμφίδρομης επικοινωνίας μεταξύ όλων των συσκευών αλλά και του host, κάθε GPU δημιουργεί $(numberOfDevices + 1)$ streams. Οι εργασίες υπολογισμού εκτελούνται από πυρήνες cuBLAS σε διαμορφώσιμο αριθμό streams. Ακόμα και αν η παράλληλη εκτέλεση πυρηνών δεν είναι εφικτή, η δρομολόγηση των εργασιών σε πολλαπλά CUDA Streams επιτρέπουν την GPU να βρίσκεται σε συνεχή ετοιμότητα και να μειωθεί ο συνολικός χρόνος στον οποίο η GPU παραμένει idle. ΑΦήνοντας το υλικό να αναλάβει την δρομολόγηση, μειώνεται το latency εκκίνησης του task και συνολικά το overhead του συστήματος.

Το επόμενο υποσύστημα που δέχθηκε αλλαγές είναι το σύστημα απόφασης δρομολόγησης επικοινωνίας. Η προηγούμενη έκδοση του PARALiA αποφάσιζε την δρομολόγηση δεδομένων μεταξύ υπολογιστικών κόμβων βάσει του bandwidth των συνδέσμων, χρησιμοποιώντας το LinkMap και έναν απλό αλγόριθμο συντομότερων μονοπατιών. Αν και η μέθοδος αυτή αυξάνει το μέσο ολικό bandwidth που επιτυγχάνεται, αγνοεί πλήρως το φορτίο του συνδέσμου που χρησιμοποιείται. Εφόσον ο στόχος της δρομολόγησης είναι η ελαχιστοποίηση της εκτιμώμενης ώρας άφιξης του tile, η εύρεση κάποιου άλλου διαθέσιμου μονοπατιού, με μικρότερο bandwitdh μπορεί να αξιοποιηθεί ώστε να εξισορροπηθεί ο φόρτος των συνδέσεων μεταξύ κόμβων. Για να μειωθούν τα φαινόμενα όπου οι GPUs παραμένουν άεργες λόγω αναμονής επικοινωνίας, χρησιμοποιείται ένας δισδιάστατος πίνακας διαστάσεων $(numberOfDevices + 1)$ x $(numberOfDevices + 1)$. Κάθε τιμή (i, j) εκφράζει τον εκτιμώμενο χρόνο στον οποίο ο σύνδεσμος $src \rightarrow dst$ απελευθερώνεται και είναι έτοιμος για χρήση. Χρησιμοποιώντας πάλι μία συνάρτηση κόστους, αποφασίζεται αν είναι κερδοφόρα η αναμονή για την επικοινωνία με τον κόμβο που κατέχει την σύνδεση με το μέγιστο bandwidth ή αν κάποιος άλλος κόμβος, με πιο αργή διασύνδεση, μπορεί να μειώσει τον χρόνο που η GPU παραμένει idle.

Οι παραπάνω αλλαγές, μαζί με άλλες βελτιστοποιήσεις στον τομέα της επικοινωνίας, την διαχείριση μνήμης και της μείωσης του overhead, καθιστά το PARALiA μια πολύ ισχυρή βιβλιοθήκη εκτέλεσης πολλαπλασιασμού πίνακα με πίνακα σε συστήματα πολλαπλών GPU.

1.7 Κατανεμημένοι Αλγόριθμοι Πολλαπλασιασμού Πινάκων

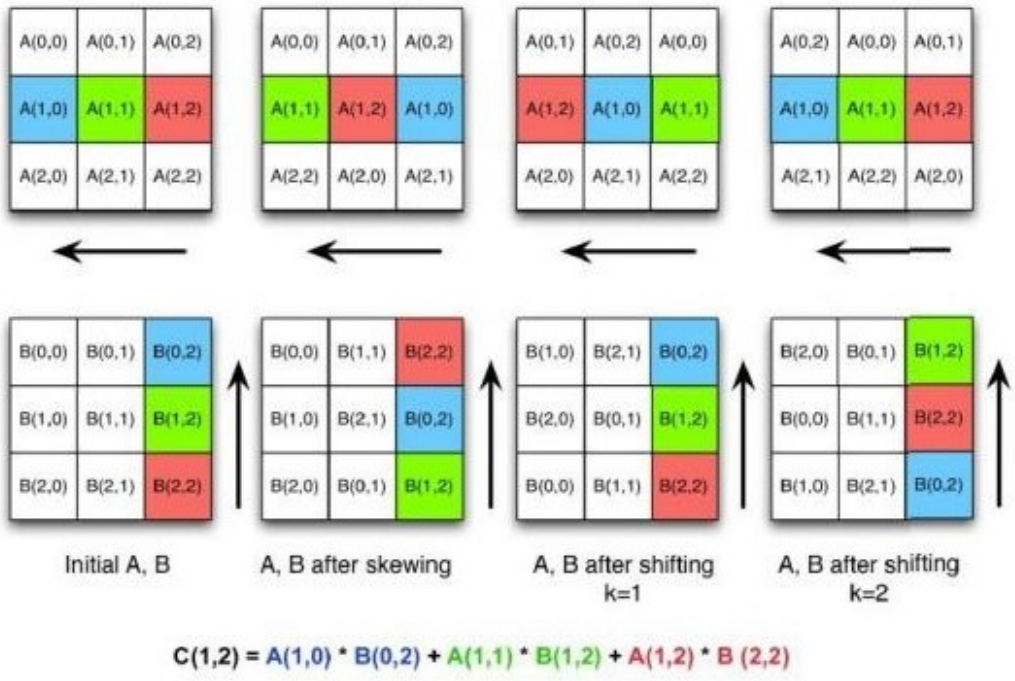
1.7.1 Cannon

Ο αλγόριθμος του Cannon [37] είναι από τους πρώτους κατανεμημένους αλγορίθμους πολλαπλασιασμών πινάκων που διαδόθηκαν. Είναι κατάλληλος για εφαρμογές που εκτελούνται σε δισδιάστατο καρτεσιανό πλέγμα διεργασιών διαστάσεων N x N και δέχεται σαν είσοδο

τεράγωνους πίνακες. Το κυριότερο πλεονέκτημα του είναι ότι οι απαιτήσεις ως προς μνήμη παραμένουν σταθερές και είναι ανεξάρτητες από τον αριθμό των επεξεργαστών που χρησιμοποιούνται.

2D Cannon

Ο αλγόριθμος είναι δομημένος ώστε n x n διεργασίες να πολλαπλασιάζουν πίνακες διαστάσεων n x n, μια περίπτωση ιδανική και μη ρεαλιστική. Σε πρακτικά συστήματα, δεν υπάρχουν όσες διεργασίες όσες και τιμές των πινάκων. Ο αλγόριθμος υλοποιείται διαιρώντας τους πίνακες με 2D Block Sequential τρόπο σε n x n blocks. Η περιγραφή αυτής της εκδοχής του αλγορίθμου γίνεται στο 4.



Σχήμα 1.20: Επικοινωνία του αλγορίθμου Cannon σε 3x3 πλέγμα διεργασιών[38]

Το κόστος μνήμης του αλγορίθμου είναι $M \approx \Theta(3n^2/p)$ όπου n η διάσταση των πινάκων και p το σύνολο των διεργασιών που συμμετάσχουν στην εκτέλεση. Το κόστος bandwidth είναι ίσο με $W = \Theta(n^2/\sqrt{p})$ ενώ το κόστος latency είναι το ελάχιστο δυνατό, με $S = \Theta(\sqrt{p})$ Η τελική πολυπλοκότητα του αλγορίθμου ανά διεργασία, ή αλλιώς το κόστος υπολογισμού είναι ίσο με $F = \Theta(n^3/p)$. Το μοντέλο αυτό είναι βασισμένο στο γεγονός ότι η επικοινωνία μεταξύ διεργασιών είναι σύγχρονη, άρα δεν επιτρέπεται η αποστολή πολλαπλών μηνυμάτων με το κόστος ενός μηνύματος.

Ο αλγόριθμος είναι αρκετά περιοριστικός αφού δέχεται μόνο τετράγωνους πίνακες σαν εισόδους και δεν μεταφράζεται εύκολα σε χρήση μη ομογενών πλεγμάτων διεργασιών, δηλαδή,

Αλγόριθμος 4: 2D Αλγόριθμος του Cannon με p διεργασίες [37]

Δεδομένα Εισόδου: Πίνακας A[n][n], Πίνακας B[n][n], Πίνακας C[n][n], n, p
Έξοδος: Πίνακας C[n][n]

/* A,B,C are distributed so that P_{ij} own A_{ij}, B_{ij}, C_{ij} */
1 $i = rank/\sqrt{p}$; // Row of process grid
2 $j = \text{mod}(rank, \sqrt{p})$; // Column of process grid
3 $s = \text{mod}(j - i, \sqrt{p})$;
4 $s' = \text{mod}(i - j, \sqrt{p})$;
5 P_{ij} sends A_{ij} to A_{local} on P_{is} ; // Skewing of A - Left shift by "i" tiles
6 P_{ij} sends B_{ij} to B_{local} on $P_{s'j}$; // Skewing of B - Up shift by "j" tiles
7 $C_{ij} = A_{local} * B_{local}$;
8 **for** *t = 1; t <* $\sqrt{p}$*; t++* **do**
9 $s = \text{mod}(j + 1, \sqrt{p})$; // Left shift of A by 1 tile
10 $s' = \text{mod}(i + 1, \sqrt{p})$; // Up shift of B by 1 tile
11 P_{ij} sends A_{ij} to A_{local} on P_{is} ;
12 P_{ij} sends B_{ij} to B_{local} on $P_{s'j}$;
13 $C_{ij} = C_{ij} + A_{local} * B_{local}$;
14 **end**

σε πλέγματα όπου οι διεργασίες γραμμών και στηλών είναι διαφορετικές σε μέγεθος. Η επέκταση του αλγορίθμου ώστε να επιτρέπει μη τετράγωνους πίνακες και ετερογενή πλέγματα είναι δυνατή, αλλά καθόλου πρακτική αφού χάνει την βελτιστότητά του στην επικοινωνία και τις απαιτήσεις μνήμης. Περαιτέρω, δεν λαμβάναει υπόψιν την ύπαρξη περίσσιας μνήμης, γεγονός που περιορίζει τις δυνατότητες επικάλυψης επικοινωνίας και υπολογισμού, μειώνοντας τις μέγιστες δυνατές επιδόσεις που μπορεί να κατορθώσει.

3D Cannon

Η εξέλιξη του αρχικού αλγορίθμου έγινε από τους [39] οι οποίοι παρουσίασαν τρισδιάστατο αλγόριθμο εκτέλεσης πολλαπλασιασμού πινάκων. Έστω πολλαπλασιασμός πινάκων A, B, C με διαστάσεις προβλήματος M, N, K. Μεταφράζουμε το πρόβλημα γεωμετρικά στον υπολογιστικό χώρο ως ένα ορθογώνιο παραλληλεπίπεδο ίδιων διαστάσεων. Επίσης θεωρούμε τρισδιάστατο πλέγμα διεργασιών διαστάσεων $P = (p_1, p_2, p_3)$. Για την διασφάλιση ισορροπίας υπολογιστικού φόρτου μεταξύ διεργασιών, κάθε εργάτης πρέπει να υπολογίσει το $1/P^{\text{th}}$ του υπολογιστικού ορθογώνιου παραλληλεπιπέδου. Άρα, ο υπολογιστκός όγκος που δέχεται κάθε διεργασία είναι ακριβώς ίσος με MNK/P. Θεωρώντας ότι κάθε διεργασία αναλαμβάνει έναν υποκύβο υπολογισμού, δηλαδή, οι διαστάσεις υποπροβλήματος m, n, k που επιλύει κάθε διεργασίας είναι ίσες μεταξύ τους, η μεταφορά δεδομένων για κάθε υποπρόβλημα είναι ίση με $3Pm^2$. Συνεχίζοντας την γεωμετρική αναπαράσταση του προβλήματος, μπορούμε να θεωρήσουμε ότι η μεταφορά m^2 δεδομένων είναι ανάλογη με το εμβαδόν μιας επιφάνειας τετραγώνου μήκους m. Άρα, το πρόβλημα ελαχιστοποίησης επικοινωνίας είναι ίδιο με ένα πρόβλημα ελαχιστοποίησης επιφάνειας για έναν δοσμένο όγκο. Αποδεικνύεται πως το κάτω όριο επικοινωνίας του συγκεκριμένου αλγορίθμου, συγκεκριμένα της τιμής $3Pm^2$, επιτυγχάνεται αν και μόνο αν: $M/p_1 = m = N/p_2 = n = K/p_3 = k$. Παρακάτω στο 5 παρουσιάζεται αναλυτικά ο αλγόριθμος.

Αλγόριθμος 5: 3D Αλγόριθμος του Cannon [39] [40]

Δεδομένα Εισόδου: Πίνακας A[n][n] κατανεμημένος έτσι ώστε η διεργασία P_{ij0} κατέχει το block A_{ij} διαστάσεων $\frac{n}{p^{1/3}}$ επί $\frac{n}{p^{1/3}}$

Δεδομένα Εισόδου: Πίνακας B[n][n] κατανεμημένος έτσι ώστε η διεργασία P_{0jk} κατέχει το block B_{jk} διαστάσεων $\frac{n}{p^{1/3}}$ επί $\frac{n}{p^{1/3}}$

Έξοδος: Πίνακας C[n][n] κατανεμημένος έτσι ώστε η διεργασία P_{i0k} κατέχει το block C_{ik} διαστάσεων $\frac{n}{p^{1/3}}$ επί $\frac{n}{p^{1/3}}$

1 $i = \text{mod}(rank/p^{1/3}, p^{1/3})$; `// Row of process grid`
2 $j = \text{mod}(\text{mod}(rank, p^{1/3}), p^{1/3})$; `// Column of process grid`
3 $k = rank/p^{2/3}$; `// Stack position of process grid`
4 **forall** $i, j, k \in \{0, 1, ..., p^{1/3} - 1\}$ **do**
5 P_{ij0} broadcasts A_{ij} to all P_{ijk} ;
6 P_{0jk} broadcasts B_{jk} to all P_{ijk} ;
7 $C_{ijk} = A_{ij} * B_{jk}$;
8 P_{ijk} contributesC_{ijk} to a sum-reduction to P_{i0k};
9 **end**

Ο αλγόριθμος 5 δέχεται ως εισόδους μόνο τετράγωνους πίνακες και συνολικό αριθμό διεργασιών τέλειους κύβους, αλλά, αντίστοιχα με τον δισδιάστατο, μπορεί να μετατραπεί καταλλήλως ώστε να δέχεται και άλλα μεγέθη. Μία πιο πρακτική μορφή του αλγορίθμου χρησιμοποιείται από το Elemental [41], μιας βιβλιοθήκης που υλοποιεί πληθώρα επιλυτών γραμμικής άλγεβρας, γενικών μεθόδων βελτιστοποίησης και βασικών πράξεων μεταξύ τανυστών. Παρουσιάζεται ως εναλλακτική λύση αντί του ScaLAPACK, με λειτουργίες υψηλότερου επιπέδου και ευκολότερου προγραμματισμού. Γενικά, οι τρισδιάστατες μορφές χρησιμοποιούνται σε βιβλιοθήκες με περιβάλλοντα χρόνου εκτέλεσης που αποφασίζουν αν η περαιτέρω πολυπλοκότητα και περίσσια μνήμη θα επιφέρουν καλύτερη επίδοση στην εκτέλεση του πολλαπλασιασμού.

2.5D Cannon

Μία ενδιάμεση λύση μεταξύ της δισδιάστατης και της τρισδιάστατης μορφής προσφέρουν οι [40] την οποία αποκαλούν 2.5D, αφού περιέχει τις προηγούμενες μορφές ως υποπεριπτώσεις. Για την εκτέλεση χρησιμοποιεί τρισδιάστατο πλέγμα διεργασιών, διαστάσεων $\sqrt{p/c}$ x $\sqrt{p/c}$ x c αλλά με δισδιάστατη αρχική κατανομή των πινάκων, στο δισδιάστατο υπό-πλέγμα διεργασιών διαστάσεων $\sqrt{p/c}$ x $\sqrt{p/c}$. Το μέγεθος c δηλώνει το μέγεθος της τρίτης διάστασης, για $c = 1$, ο αλγόριθμος εκτελεί την 2D μορφή ενώ για $c = p^{1/3}$ θυμίζει την 3D εκτέλεση, με κάποιες διαφορές στην αρχική επικοινωνία και την τελική θέση του πίνακα C. Ο περιορισμός πέρι ομογενών πλεγμάτων διεργασιών και τετράγωνων πινάκων συνεχίζει να ισχύει. Η αναλυτική περιγραφή του αλγορίθμου βρίσκεται στο 6.

Αντίστοιχα με την 3D εκδοχή, η συγκεκριμένη μορφή χρησιμοποιείται από αλγορίθμους που αποφασίζουν αν η διάθεση περίσσιας μνήμης και πολυπλοκότητας επικοινωνίας μπορεί να προσφέρει καλύτερη συνολική επίδοση. Το πλεονέκτημα της 2.5D εκδοχής είναι ότι μπορεί να επιλέξει οποιαδήποτε τιμή για $c \in \{1, ..., p^{1/3}\}$ που την καθιστά πιο ελκυστική και λιγότερη περιοριστική από την 3D εκδοχή.

Αλγόριθμος 6: 2.5D Αλγόριθμος του Cannon [40]

Δεδομένα Εισόδου: Πίνακας A[n][n], Πίνακας B[n][n], Πίνακας C[n][n], n, p, c
Έξοδος: Πίνακας C[n][n]

/* A,B,C are distributed so that P_{ij} own A_{ij}, B_{ij}, C_{ij} */
1 $i = \text{mod}(rank/\sqrt{p/c}, \sqrt{p/c})$; // Row of process grid
2 $j = \text{mod}(\text{mod}(rank, \sqrt{p/c}), \sqrt{p/c})$; // Column of process grid
3 $k = c * rank/p$; // Stack position of process grid
4 P_{ij0} broadcastsA_{ij} and$B_{ij}$to a all P_{ijk};
5 $s = \text{mod}(j - i + k\sqrt{p/c^3}, \sqrt{p/c})$;
6 $s' = \text{mod}(i - j + k\sqrt{p/c^3}, \sqrt{p/c})$;
7 P_{ijk} sends A_{ij} to A_{local} on P_{isk} ; // Initial circular shift on A
8 P_{ijk} sends B_{ij} to B_{local} on $P_{s'jk}$; // Initial circular shift on B
9 $C_{ijk} = A_{local} * B_{local}$;
10 **for** *t = 1; t <* $\sqrt{p}$*; t++* **do**
11 $s = \text{mod}(j + 1, \sqrt{p/c})$; // Left shift of A by 1 tile
12 $s' = \text{mod}(i + 1, \sqrt{p/c})$; // Up shift of B by 1 tile
13 P_{ijk} sends A_{local} to P_{isk} ;
14 P_{ijk} sends B_{local} to $P_{s'jk}$;
15 $C_{ijk} = C_{ijk} + A_{local} * B_{local}$;
16 **end**
17 P_{ijk} contributes C_{ijk} to a sum-reduction to P_{ij0};

Ο 2.5D αλγόριθμος χρησιμοποιείται από την βιβλιοθήκη Cyclops Tensor Framework [42] η οποία δημιουργήθηκε από τους αρχικούς author. Η βιβλιοθήκη εκτελεί συζευγμένες πράξεις τανυστών, μια πολύ ρεαλιστική και σύνηθης εφαρμογή για εκτέλεση σε συστήματα cluster.

1.7.2 SUMMA

Ο SUMMA (Scalable Universal Matrix Multiplication Algorithm) [43] αποτελεί τον πιο πρακτικό κατανεμημένο αλγόριθμο πολλαπλασιασμού πινάκων. Αρχικά σχεδιάστηκε για να συγκριθεί κυρίως με τον προϋπάρχοντα αλγόριθμο που χρησιμοποιούσε το ScaLAPACK, τον PUMMA [44], τον οποίο εν τέλει αντικατέστησε.

Για το κλασσικό πρόβλημα πολλαπλασιασμού πινάκων με διαστάσεις M, N, K, θεωρούμε δισδιάστατο πλέγμα διεργασιών $P = (d_{Row}$ x $d_{Col})$, στο οποίο **δεν** είναι απαραίτητο να ισχύει $d_{Row} = d_{Col}$. Για κάθε πίνακα $X \in \{A, B, C\}$, το block X_{ij} αναλογεί στην διεργασία P_{ij} με διαστάσεις m_i^X x n_i^X τέτοιες ώστε $\sum m_i^X = m$ και $\sum n_i^X = n$. Οι δείκτες i και j υποδηλώνουν την θέση της διερασίας με $rank = i * d_{Row} + j$ στο δισδιάστατο πλέγμα διεργασιών και ισχύει $i \in [0, d_{Row} - 1]$ και $j \in [0, d_{Col} - 1]$. Ο βασικός παράλληλος αλγόριθμος, ο οποίος βασίζεται σε πολλαπλά εσωτερικά γινόμενα στηλών του A και γραμμών του B παρουσιάζεται στο 9

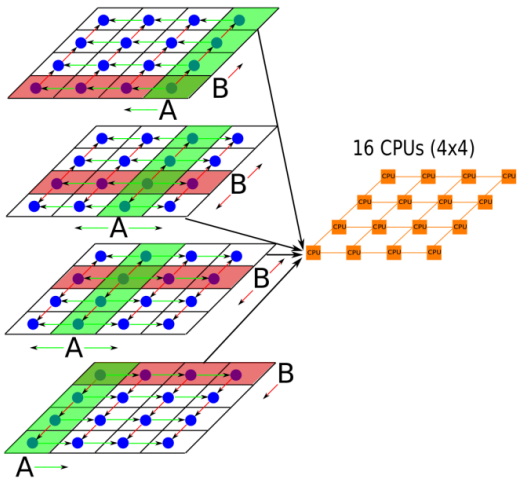
Άμεση βελτίωση του προηγούμενου αλγορίθμου επιτυγχάνεται αν αντί να βασίσουμε τον υπολογισμό των στοιχείων του C ως πολλαπλά εσωτερικά γινόμενα των στηλών/γραμμών A και B, εκτελέσουμε πολλαπλασιασμό μεταξύ υπο-πινάκων των προηγούμενων. Συσσωρεύοντας πολλαπλές στήλες του A_i και γραμμές του B_j πριν εκτελέσουμε την ανανέωση του τοπικού C, δημιουργούμε δισδιάστατους υποπίνακες ελεγχόμενου μεγέθους, μειώνοντας το latency και αυξάνοντας την υπολογιστική επίδοση των υπόπροβλημάτων. Η μορφή αυτή ονομάζεται block-

Αλγόριθμος 7: Ψευδοκώδικας SUMMA βασισμένο σε εσωτερικά γινόμενα

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

$C_{ij} = 0$;
for *l = 0; l < k; l++* **do**
    broadcast row a_i within all processes in row i ;
    broadcast column b_j within all processes in column j ;
    $temp = dot(k, a_i, b_j)$; `// Dot Product of` a_i, b_j
    $C_{ij} = C_{ij} + temp$;
end

ing. Για ευκολότερη παρουσίαση, θεωρούμε τετράγωνο δισδιάστατο πλέγμα τέτοιο ώστε $d_{Row} = d_{Col}$.



Σχήμα 1.21: Επικοινωνιακά βήματα εκτέλεσης του SUMMA σε δισδιάστατο πλέγμα [45]

Αλγόριθμος 8: Ψευδοκώδικας Blocking SUMMA βασισμένο σε gemm

Δεδομένα Εισόδου: Πίνακας A[m][k], Πίνακας B[k][n], Πίνακας C[m][n]
Έξοδος: Πίνακας C[m][n]
`/*` d_{Col} `or` $\sqrt{P}$ `instead of` d_{Row} `is the same here */`
for *l = 0; l <* d_{Row}*; l++* **do**
    broadcast A_{il} within all processes in row i ;
    broadcast B_{lj} within all processes in column j ;
    $C_{ij} = C_{ij} + A_{il} * B_{lj}$;
end

Η αρχική δημοσίευση προσφέρει αναλυτικό κώδικα υλοποίησης του παραπάνω αλγορίθμου χρησιμοποιώντας το MPI ως επικοινωνιακό backend.

Καθώς το PBLAS δέχεται ως εισόδους πίνακες με συγκεκριμένη κατανομή, την 2D Block Cyclic, η πιο συχνή εκδοχή του αλγορίθμου είναι η παρακάτω. Όπως πριν, θεωρούμε ομογενές δισδιάστατο πλέγμα διεργασιών $P = (d_{Row}$ x $d_{Col})$ με $d_{Row} = d_{Col}$.

Αλγόριθμος 9: Ψευδοκώδικας Blocking SUMMA με 2D Block Cyclic κατανομή

Δεδομένα Εισόδου: Πίνακας A[m][k], Πίνακας B[k][n], Πίνακας C[m][n]
Έξοδος: Πίνακας C[m][n]
/* Processor $P(r, c)$ owns each block A_{ij}, B_{ij}, C_{ij} with $i \equiv \mathrm{mod}(r, \sqrt{P})$ and $j \equiv \mathrm{mod}(c, \sqrt{P})$ */
1 **for** *l = 0; l < d_{Row}; l++* **do**
2 allgather A_{il} within all processes in row i ;
3 allgather B_{lj} within all processes in column j ;
4 $C_{ij} = C_{ij} + A_{il} * B_{lj}$;
5 **end**

Ο αλγόριθμος αυτός, τροποποιημένος ώστε να δέχεται μη ομογενή δισδιάστατα πλέγματα, χρησιμοποιείται από το ScaLAPACK, μέσω του PBLAS, καθώς και από πολλαπλές βιβλιοθήκες όπως το SLATE [46], το cuBLASMp [47], το [41] και άλλες. Το SLATE σχεδιάστηκε ως μια σύγχρονη εκδοχή του ScaLAPACK για χρήση σε μοντέρνα κατανεμημένα ετερογενή συστήματα υψηλών επιδόσεων. Το cuBLASMp είναι η λύση της NVIDIA για την εκτέλεση ρουτινών PBLAS σε GPUs, δίνοντας ιδιαίτερη έμφαση στον πυρήνα GEMM.

Κοινώς αποδεκτός ως ένας πολύ σημαντικός και πρακτικός αλγόριθμος πολλαπλασιασμού πινάκων, με την αρχική δημοσίευση να λαμβάνει αξιοσημείωτο αριθμό αναφορών από αργότερα έργα και έρευνες, συνεχίζει να χρησιμοποιείτει ευρέως, ειδικά από εφαρμογές που εκτελούνται σε συστήματα συστοιχιών όπως το NWChem [48] και το CP2K [49], λογισμικά που εκτελούν προσομοιώσεις σε δεδομένα μεγάλων διαστάσεων.

Αντίστοιχα με τον κλασσικό αλγόριθμο του Cannon, έχουν αναπτυχθεί εκδοχές που εκτελούνται σε μεγαλύτερες διαστάσεις διεργασιών, όπως 3D [50] ή 2.5D [51]. Ειδικά η τελευταία είναι αρκετά αξιοσημείωτη καθώς δέχεται δεδομένα αρχικά κατανεμημένα σε δισδιάστατο πλέγμα, αλλά εκτελείται σε τρισδιάστατο, δείχνοντας ότι οι επεκτάσεις σε τρίτη διάσταση μπορούν να υλοποιηθούν ως βελτιώσεις στον αρχικό αλγόριθμο χωρίς να αλλάζει η εφαρμογή του σε υπάρχοντα κώδικα.

1.7.3 Recursive (CARMA)

Μέχρι στιγμής, οι προηγούμενοι αλγόριθμοι χρησιμοποιούσαν κλασσικές τεχνικές παραλληλισμού δεδομένων, βασισμένες σε προγραμματισμό πλεγμάτων. Οι αλγόριθμοι αυτοί, αποδεικνύεται ότι είναι επικοινωνιακά βέλτιστοι για τετράγωνους πίνακες και επιτυγχάνουν υψηλές επιδόσεις, ειδικά όταν ταυτίζονται με το πλέγμα των πραγματικών τοπολογιών δικτύων που λαμβάνει το σχήμα ενός τόρου [52]. Όμως, οι επιδόσεις τους σε εκτελέσεις όπου οι πίνακες έχουν ορθογώνιο σχήμα ή σε πιο γενικές τοπολογίες δικτύων, είναι αρκετά χαμηλότερες, γεγονός που τους καθιστά αρκετά μη αποδοτικούς σε αρκετά ρεαλιστικά φορτία εργασίας (workloads). Για τους παραπάνω λόγους, οι κλασσικοί αλγόριθμοι πλεγμάτων χρειάζονται αρκετό fine-tuning επάνω στην εκάστοτε αρχιτεκτονική που εκτελούνται για να πετύχουν βέλτιστες επιδόσεις.

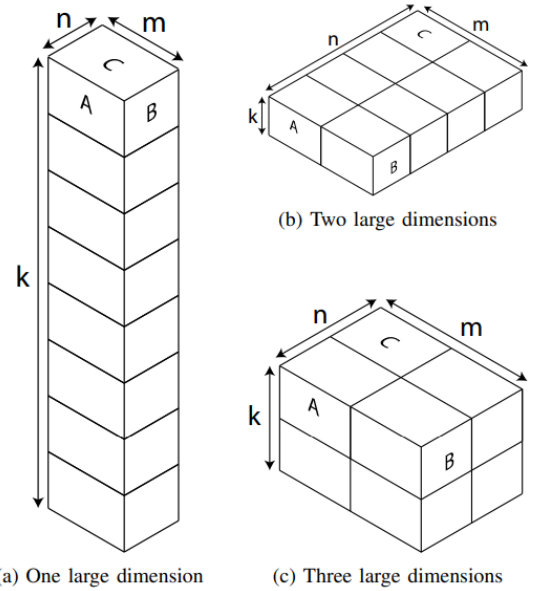
Το CARMA [53] προσφέρει μια αλγοριθμική λύση βασισμένη στην τεχνική διαίρει και βασίλευε, η οποία χρησιμποιείται συχνά σε αλγορίθμους που βασίζονται στην αναδρομή. Αποδεικνύεται ασυμπτοτικά βέλτιστη σε επικοινωνιακά κόστη για τις τρεις σημαντικές

κατηγορίες εκτέλεσης του πολλαπλασιασμού. Αυτές καθορίζονται από το μέγεθος των τριών διαστάσεων των πινάκων M, N, K.

1. Αν η διάσταση K είναι σημαντικά μεγαλύτερη από τις M, N, τότε θεωρούμε μία μεγάλη διάσταση. Τέτοιες διαστάσεις συναντούμε στην εκτέλεση του αλγορίθμου αποσύνθεσης πινάκων Cholesky QR.

2. Αν οι διαστάσεις M, N είναι σημαντικά μεγαλύτερες από την K, τότε θεωρούμε δύο μεγάλες διαστάσεις. Συχνή εμφάνιση είναι στην εκτέλεση της αποσύνθεσης πινάκων με την χρηση του LU αλγορίθμου.

3. Αν και οι τρεις διαστάσεις βρίσκονται σε κοινή τάξη μεγέθους, θεωρούμε τρεις μεγάλες διαστάσεις. Η πιο σύνηθης εκτέλεση πολλαπλασιασμού πινάκων, ειδικά για τετράγωνους πίνακες.



Σχήμα 1.22: Τα τρία πιθανά ενδεχομένα εκτέλεσης GEMM [53]

Ο αλγόριθμος που χρησιμοποιεί ανήκει στην κατηγορία των BFS/DFS, στους οποίους η διάταξη των επεξεργαστών παρουσιάζεται ως μια ιεραρχία αντί για το κλασσικό καρτεσιανό πλέγμα που έχουμε συνατήσει μέχρι στιγμής. Αναλυτικά, ο αλγόριθμος εκτελείται με δύο διαφορετικά ήδη βημάτων για την επίλυση των υπο-προβλημάτων, είτε με βήμα κατά πλάτος (Breadth-First step η BFS) ή με βήμα κατά βάθος (Depth-first step ή DFS). Τα BFS βήματα χρησιμοποιούνται όταν επαρκεί η μνήμη κάθε επεξεργαστή ώστε να υπολογίσει το υπό-πρόβλημα και αποτελεί αυτόνομο task στο οποίο δεν συμμετέχουν άλλοι επεξεργαστές. Τα DFS βήματα χρησιμοποιούνται σε προβλήματα με μεγαλύτερες απαιτήσεις μνήμης και συμμετέχουν όλοι οι επεξεργαστές στο ίδιο υπό-πρόβλημα σε μια δοσμένη χρονική στιγμή. Γενικά, τα BFS βήματα ελαχιστοποιούν την ανάγκη για επικοινωνία κατά την επίλυση αλλά χρειάζονται περισσότερη μνήμη συγκριτικά με τα DFS βήματα. Λόγω της αναδρομικής φύσης τους, οι BFS/DFS αλγόριθμοι αγνοούν πλήρως λεπτομέρειες εκτέλεσης όπως τις επιδόσεις του επεξεργαστή, το μέγεθος της κρυφής μνήμης και την τοπολογία του δικτύου συστήματος στο οποίο εκτελούνται,

άρα θεωρητικά δεν έχουν την ανάγκη για fine-tuning. Η εκτέλεση του αλγορίθμου γίνεται διχοτομώντας την μεγαλύτερη διάσταση του προβλήματος και μετά επιλύοντας τα 2 ξεχωριστά υπό-προβλήματα με BFS ή DFS τρόπο. Ο ψευδοκώδικας του αλγορίθμου παρουσιάζεται στο 10.

Αλγόριθμος 10: Σύντομος ψευδοκώδικας CARMA [53]

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

```
Split the largest of m, n, k in half, giving two subproblems;
if Enough Memory then
    Solve the two problems recursively with a BFS;
end
else
    Solve the two problems recursively with a DFS;
end
```

Οι αρχικοί συγγραφείς προσφέρουν υλοποίηση σε Cilk++ σε μορφή Github Repository [54], αλλά δεν προσφέρουν την υλοποίηση με MPI. Περαιτέρω, δεν υποστηρίζεται εκτέλεση σε GPU, καθιστώντας την εκτός ορίων ως σύγκριση για την συγκεκριμένη μελέτη. Σαν υποσημείωση, αναφέρεται ότι το CARMA δεν είναι πρακτικό ως αντικατάσταση του ScaLAPACK καθώς δεν υποστηρίζει την 2D Block Cyclic κατανομή δεδομένων και σε συνεχόμενες εκτελέσεις πολλαπλασιασμών, το τελικό layout μπορεί να είναι διαφορετικό λόγω εσωτερικών μετασχηματισμών κατά την εκτέλεση.

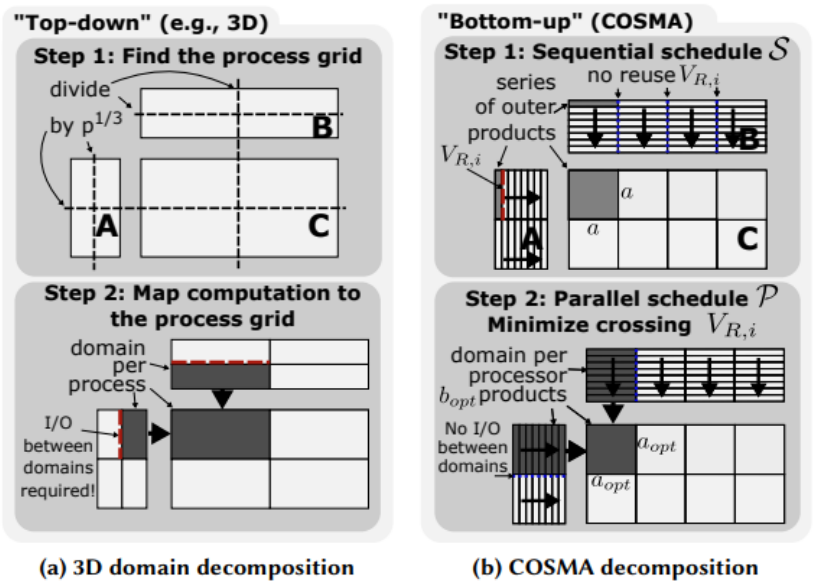
1.7.4 COSMA

Αν και το CARMA προσέφερε ασυμπτοτικά βέλτιστο κάτω όριο επικοινωνίας για όλα τα πιθανά μεγέθη πινάκων, η επιλογή της διχοτόμησης πάντα της μεγαλύτερης διάστασης μπορεί να προσθέσει περίσσια επικοινωνία της τάξεως του $\sqrt{3}$. Περαιτέρω, δεν λαμβάνει υπόψιν τον αριθμό των επεξεργαστών που καλούνται να εκτελέσουν τον υπολογισμό.

Δίνοντας έμφαση στο γεγονός ότι *η ασυμπτοτική πολυπλοκότητα είναι ανεπαρκής μετρική για την ακριβή πρόβλεψη της πρακτικής επίδοσης*, το COSMA (Communication Optimal Spartition-based Matrix multiplication Algorithm) [55] προσφέρει λύση για την αποδοτική εκτέλεση πολλαπλασιασμού πίνακα-πίνακα για οποιαδήποτε μεγέθη προβλημάτων **ΚΑΙ** αριθμό επεξεργαστών.

Σε αντίθεση με τους προηγούμενους αλγορίθμους, οι οποίει δρομολογούν την εκτέλεση του προβλήματος σε διεργασίες με top-down τρόπο, δηλαδή, μόνο βάσει των διαστάσεων του προβλήματος, το COSMA δρομολογεί με bottom-up τρόπο. Έστω S η γρήγορη μνήμη κάθε επεξεργαστή που συμμετάσχει στην εκτέλεση του αλγορίθμου. Αυτή η τιμή μπορεί να παριστάνει το μέγεθος κρυφής μνήμης ενός πυρήνα επεξεργαστή ή την μέγιστη αξιοποιήσιμη κοινόχρηστη μνήμη μιας GPU. Για τον πίνακα C, δημιουργούμε ένα tile με μέγεθος όσο και η γρήγορη μνήμη S και κοιτάμε τα data dependencies από τους πίνακες A, B. Θεωρούμε ότι το tile είναι τετράγωνο με διαστάσεις a x a. Αυτή η δρομολόγηση ονομάζεται σειριακή, και είναι βέλτιστη υπολογιστικά για έναν επεξεργαστή αφού χωράνε όλα τα δεδομένα στην γρήγορη μνήμη του Συνολικά, πρέπει να υπολογιστούν b εξωτερικά γινόμενα διαστάσεων a x a για τον τελικό υπολογισμό του tile. Η τιμή b υπολογίζεται από την διαδικασία εύρεσης της παράλληλης δρομολόγησης. Στην

διαδικασία αυτή, ο αλγόριθμος προσπαθεί να δημιουργήσει task υπολογισμού για κάθε διεργασία στα οποία δεν υπάρχει διαμοιρασμός δεδομένων μεταξύ διεργασιών, ελαχιστοποιώντας την ανάγκη για επικοινωνία. Έχοντας υπολογίσει τις τιμές a, b, κοινές για όλες τις διεργασίες ώστε να διατηρείται η συνθήκη ισομοιρασμού φόρτου εργασίας, δημιουργείται ένα πλέγμα διεργασιών το οποίο μοιράζει την κατανομή του προβλήματος σε ίσα μέρη κατά διεργασίες. Αν οι διαστάσεις M, N, K δεν διαιρούνται τέλεια από τα a, b, επιλέγονται νέες τιμές με πιθανή αφαίρεση κάποιας διεργασίας από το πρόβλημα. Με τον τρόπο αυτό, μειώνεται η συνολική υπολογιστική ικανότητα του συνόλου, αλλά η μείωση στην επικοινωνία μπορεί να είναι δραστική ώστε να καλύπτει το κενό αυτό και επιφέρει καλύτερες επιδόσεις.



Σχήμα 1.23: Αποσύνθεση προβλήματος GEMM στο COSMA [55]

Το COSMA αποδεικνύει ασυμπτοτικά το κάτω άκρο του επικοινωνιακού κόστους βάσει ενός παιγνίου, του red-blue pebble game [56]. Το παίγνιο αυτό μοντελοποιεί την επικοινωνιακή πολυπλοκότητα αλγορίθμων χρησιμοποιώντας έναν κατευθυνόμενο ακυκλικό γράφο. Ο γράφος αυτός παρουσιάζει τους κόμβους του ως υπολογισμούς και τις ακμές ως τις εξαρτήσεις δεδομένων. Το παιχνίδι κινείται σειριακά. Μια κίνηση μπορεί να είναι η πρόσθεση, η αφαίρεση ή ο χρωματισμός ενος κόμβου σε κόκκινο ή μπλε χρώμα, με το κόκκινο να χαρακτηρίζει δεδομένα που βρίσκονται στην γρήγορη μνήμη του επεξεργαστή (π.χ. Cache) και το μπλε δεδομένα στην αργή μνήμη (π.χ. Δίσκος ή εξωτερική μνήμη). Ένας κόμβος μπορεί να χρωματιστεί κόκκινος αν όλοι οι προηγούμενοι κόμβοι είναι επίσης κόκκινοι. Επίσης, μπορεί να αφαιρεθεί πλήρως το χρώμα από πάνω του ή να αλλάξει χρώμα από κόκκινο σε μπλε (εγγραφή δεδομένων στην αργή μνήμη ή store) ή από μπλε σε κόκκινο (ανάγνωση δεδομένων από αργή μνήμη ή load). Το παιχνίδι τελειώνει όταν οι κόμβοι που έχουμε θέσει ως έξοδοι του αλγορίθμου έχουν χρωματιστεί, ανεξαρτήτως χρώματος.

Η μοντελοποίηση συμβαίνει ως εξής: με σταθερό μέγιστο αριθμό κόκκινων κόμβων που μπορούν να υπάρξουν ταυτόχρονα στον γράφο, δηλαδή, με πεπερασμένη γρήγορη μνήμη, πρέπει να βρούμε μια εκτέλεση του παιχνιδιού με τους ελάχιστους πιθανούς αναχρωματισμούς κόμβων,

δηλαδή, με τις ελάχιστες λειτουργίες μνήμης.

Λόγω πολύπλοκων μαθηματικών και λογικών σχέσεων, οι οποίες δεν εμπίπτουν στο πλάισιο αυτής της εργασίας, η πλήρης απόδειξη του κάτω φράγματος αφήνεται στην ευχέρεια του αναγνώστη αν επιθυμεί να το διαβάσει από την αρχική δημοσίευση[55].

Το COSMA αναδεικνύει την επίδοσή του στην εκτέλεση με εισόδους μη-τετράγωνους πίνακες και ασυνήθιστα μεγέθη ή συνολικό αριθμό διεργασιών. Προσφέρεται ως βιβλιοθήκη που εκτελεί πράξεις σε CPU ή GPU χρησιμοποιώντας την δική τους μορφή αποσύνθεσης πινάκων ή απευθείας τις συναρτήσεις pxgemm που δέχονται ως είσοδο 2D Block Cyclic αποσυνθέσεις. Το υπολογιστικό backend για CPU είναι κλασσική OpenBLAS ενώ για GPU χρησιμοποιούν δικό τους σύστημα με χρήση τεχνικών tiling, βασισμένο στο cuBLAS, το Tiled-MM [57].

Chapter 2

Υλοποίηση

Σε αυτό το κεφάλαιο περιγράφουμε την αρχιτεκτονική των συστημάτων που υλοποιήθηκαν, την μεθοδολογία και τις αλλαγές ή προσθήκες που χρειάστηκαν για να επιτευχθεί η επιθυμητή επίδοση που αναζητούσαμε. Συνολικά σχεδιάστηκαν δύο συστήματα, το πρώτο πειραματικό χρησιμοποιώντας έτοιμες βιβλιοθήκες για βελτιστοποίηση πολλαπλασιασμού εντός κόμβου ενώ το δεύτερο πέρασε από αρκετά στάδια βελτίωσης και επανασχεδίασης.

2.1 Σειριακή Block αποσύνθεση με PARALiA

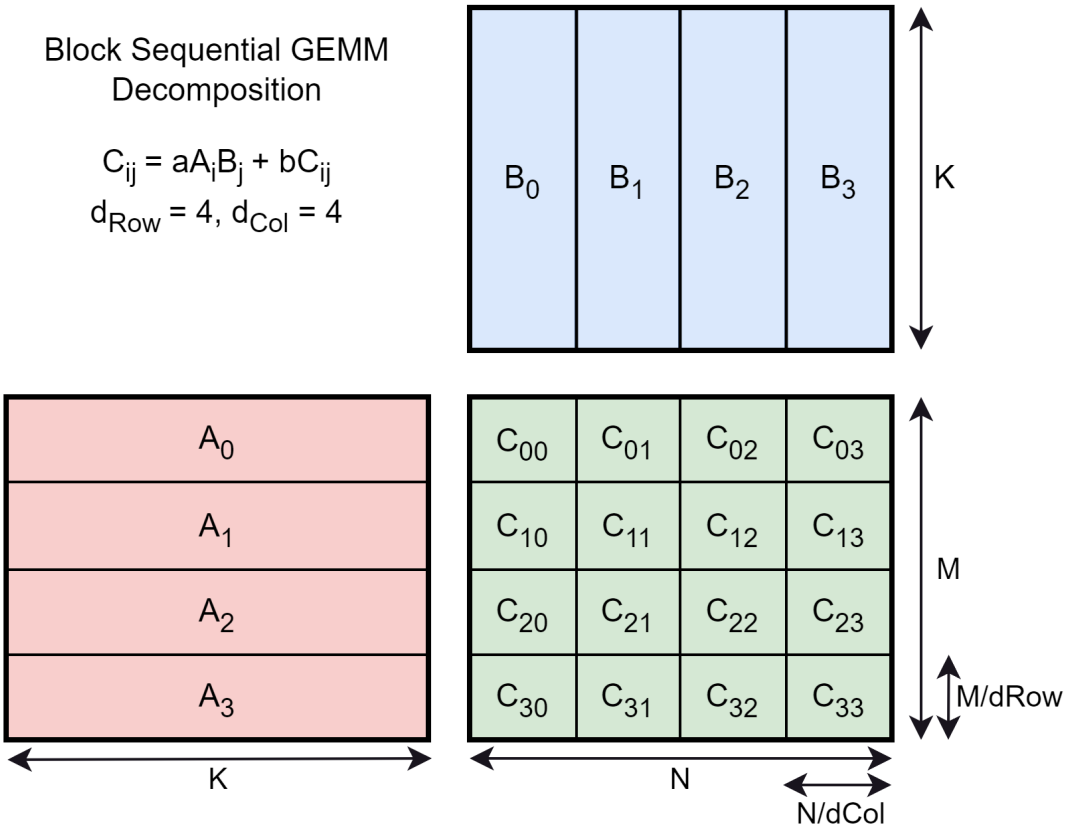
Η πρώτη προσπάθεια βασίστηκε στην χρήση του PARALiA ως υπολογιστικό backend. Έχοντας αποδείξει ότι επιτυγχάνει, οριακά, το θεωρητικό μέγιστο της επίδοσης του συστήματος, αποτελεί εύκολη λύση να αξιοποιήσουμε τις δυνατότητες του ανά κόμβο.

2.1.1 Αποσύνθεση Προβλήματος

Υπενθυμίζουμε ότι το πόσοι επεξεργαστές/πυρήνες/συσκευές αποτελούν μια διεργασία στο MPI, αφήνεται ως επιλογή στον προγραμματιστή ή στον τελικό χρήστη του λογισμικού. Οι βιβλιοθήκες εκτέλεσης GEMM σε GPU με τις οποίες συγκρινόμαστε, επιλέγουν κάθε GPU να αποτελεί και μια διεργασία. Στην συγκεκριμένη περίπτωση, καθώς το PARALiA εκτελείται σε συστήματα με πολλαπλές GPU, επιλέγουμε να δημιουργείται μια διεργασία ανά κόμβο.

Έστω ότι συμμετέχουν n κόμβοι στην εκτέλεση του κλασσικού προβλήματος πολλαπλασιασμού πινάκων $A \in R^{M\text{x}K}, B \in R^{K\text{x}N}, C \in R^{M\text{x}N}$. Αρχικά, αναπαριστούμε τις n διεργασίες σε ένα δισδιάστατο καρτεσιανλο πλέγμα, διαστάσεων $dRow$ x $dCol$, στο πιο τετράγωνο σχήμα που μπορούν να διαταχθούν. Βάσει των διαστάσεων αυτών, κατανέμουμε τον πίνακα C με 2D Block Sequential τρόπο, δηλαδή, κάθε κόμβος παραλαμβάνει block διαστάσεων $M/dRow$ x $N/dCol$, με κάθε διεργασία P_{ij} να κατέχει το block C_{ij}. Για ευκολία, ονομάζουμε τις διαστάσεις του block $local_M$ και $local_N$ Πλέον, μπορούμε να αναγνωρίσουμε τις εξαρτήσεις δεδομένων από τους πίνακες A, B για κάθε διεργασία. Ο πίνακας A χωρίζεται σε $dRow$ block διαστάσεων $local_M$ x K ενώ ο B σε $dCol$ block διαστάσεων K x $local_N$, με κάθε διεργασία P_{ij} να λαμβάνει τα A_i και B_j, ώστε να καλύπτονται πλήρως οι εξαρτήσεις δεδομένων. Πρακτικά, αυτή είναι 1D Block Sequential κατά γραμμές για τον πίνακα A και κατά στήλες για τον πίνακα B. Έχοντας κατανέμει τα δεδομένα με τον παραπάνω τρόπο, εκτελείται τοπικά από κάθε διεργασία

μια κλήση gemm, εκτελώντας το $C_{ij} = \alpha A_i * B_j + \beta C_{ij}$. Για την επιστροφή των δεδομένων στην διεργασία που κατείχε αρχικά τα δεδομένα, ακολουθείται η ακριβώς αντίστροφη διαδικασία από την παραπάνω.



Σχήμα 2.1: Αποσύνθεση προβλήματος - Σειριακή Block έκδοση με PARALiA

Αλγόριθμος 11: Ψευδοκώδικας 2D Block Sequential PARALiA

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

1 $i = rank/dRow$; /* Process Row */
2 $j = \text{mod}(rank, dRow)$; /* Process Column */
3 $localA = Decompose1D(A, ROW\_WISE, i, j)$;
4 $localB = Decompose1D(B, COLUMN\_WISE, i, j)$;
5 $localC = Decompose2D(C, BLOCK\_SEQUENTIAL, i, j)$;
6 $ParaliaDGEMM(localA, localB, localC, alpha, beta)$;
7 $Gather2D(localC, C, BLOCK\_SEQUENTIAL, i, j)$;

Κάθε κόμβος, λύνει ένα υποπρόβλημα πολλαπλασιασμού διαστάσεων $local_M, local_N, K$ με αποδοτικό τρόπο λόγω της συμμετοχής του PARALiA στην εκτέλεση. Καθώς το ίδιο περιλαμβάνει τεχνικές tiling και επικάλυψη επικοινωνίας/υπολογισμού, δεν χρειάζεται να ενσωματωθούν στο κατανεμημένο σύστημα εκτέλεσης. Περαιτέρω, η αποσύνθεση αυτή έχει την ιδαιτερότητα ότι δεν απαιτεί καθόλου επικοινωνία μεταξύ διεργασιών για την εκτέλεση του πολλαπλασιασμού, αφού

όλες οι εξαρτήσεις δεδομένων έχουν καλυφθεί στο κομμάτι της κατανομής και βρίσκονται ήδη στις μνήμες των συσκευών (ή του host).

2.1.2 Υλοποίηση με MPI

Η διαδικασία υλοποίησης του παραπάνω συστήματος με MPI είναι αρκετά άμεση. Αρχικά, γράφουμε αποδοτικές συναρτήσεις αποσύνθεσης πίνακα, για μονοδιάστατη και δισδιάστατη μορφή. Κατά κύριο λόγο χρησιμοποιούν point-to-point επικοινωνία μέσω των MPI_Send και MPI_Recv, αλλά, συγκεκριμένα για αποσύνθεση τετράγωνων πινάκων, μπορεί να χρησιμοποιηθεί συλλογική επικοινωνία (MPI_Scatterv) η οποία είναι αρκετά πιο αποδοτική. Ο λόγος ο οποίος δεν μπορεί να χρησιμοποιηθεί συλλογική επικοινωνία για μη τετράγωνους πίνακες είναι η διαφοροποίηση στις διαστάσεις των τοπικών πινάκων. Αν η διαίρεση $\frac{M}{d_{Row}}$ ή $\frac{N}{d_{Col}}$ αφήνει υπόλοιπο rem, οι διαστάσεις τοπικών πινάκων των τελευταίων διεργασιών είναι $local_M = \frac{M}{d_{Row}} + rem$ (αντίστοιχα για την διάσταση N). Οι κλήσεις όμως MPI_Scatterv και MPI_Gatherv χρησιμοποιούν το ίδιο MPI_Datatype ανά Scatter, δυσκολεύοντας την αποσύνθεση. Για τις περιπτώσεις αυτές, χρησιμοποιούμε εξ αρχής point-to-point επικοινωνία, προσθέτωντας βελτιώσεις όπου αυτό είναι εφικτό.

Η επιλογή του παραλήπτη των υπό-πινάκων γίνεται στην κλήση της συνάρτησης πολλαπλασιασμού. Εφόσον το PARALiA δέχεται είσοδο πίνακες είτε σε μνήμη host ή σε GPU, επιλέγουμε αντίστοιχα τον παραλήπτη αναλόγως με τις διαστάσεις του προβλήματος. Η παράδοση των δεδομένων απευθείας σε GPU είναι εφικτή αφού η έκδοση OpenMPI που χρησιμοποιούμε είναι CUDA-Aware.

2.1.3 Πρώιμα συμπεράσματα

Κατά την σχεδίαση και αξιολόγηση αυτού του συστήματος, βγήκαμε στο συμπέρασμα πως αν και αυτή λύση επιτυγχάνει άριστες επιδόσεις και αξιοποιεί πολύ μεγάλο ποσοστό του υλικού σε Compute Bound εφαρμογές, δεν είναι πολύ φιλική για χρήση σε κατανεμημένα συστήματα. Αρχικά, οι μονοδιάστατες αποσυνθέσεις των πινάκων A, B περιορίζουν σημαντικά τις μέγιστες διαστάσεις προβλήματος που μπορεί να επιλυθεί και δεν χρησιμοποιούνται από κανένα αξιοσημείωτο λογισμικό που χρησιμοποιεί ρουτίνες p_gemm, ώστε να μπορούμε να δικαιολογήσουμε την χρήση του συστήματος. Οι διεργασίες που βρίσκονται σε ίδια σειρά/στήλη στο πλέγμα μοιράζονται τα ίδια κομμάτια πίνακα των A, B, δημιουργώντας μια κατάσταση όπου πολλαπλές διεργασίες μοιράζονται μεγάλο όγκο ίδιων δεδομένων. Οι αυξημένες απαιτήσεις σε μνήμη είναι αναγακίες για την πλήρη εξάλειψη της επικοινωνίας μεταξύ διεργασιών. Περαιτέρω, κανένας από τους γνωστούς αλγορίθμους κατανεμημένου πολλαπλασιασμού πινάκων δεν υποστηρίζει αυτή την αποσύθνεση δεδομένων, δυσκολεύοντας την ”δίκαιη” σύγκριση με αυτούς στην διαδικασία της αξιολόγησης.

Παρ’όλα αυτά, βρίσκει χρησιμότητα σε εφαρμογές όπου ο πίνακας βρίσκεται εξ’ ολοκλήρου σε μία διεργασία και πρέπει να κατανεμηθεί για να εκτελεστεί ο πολλαπλασιασμός. Αυτό το workload, θα το συναντήσουμε στο επόμενο κεφάλαιο κατά την αξιολόγηση.

2.2 2.5D Cannon's με cuBLAS

Έχοντας πλέον κατανοήσει τα προβλήματα σχεδίασης συστήματος εκτέλεσης πολλαπλασιασμού πίνακα με πίνακα σε συστοιχίες, ακολουθούμε διαφορετική νοοτροπία σε αυτή την έκδοση. Το τελικό αποτέλεσμα θέλουμε να έχει τα εξής χαρακτηριστικά:

1. Να δέχεται σαν είσοδο δεδομένα κατανεμημένα σε μια μορφή που δέχονται εφαρμογές που χρησιμοποιούν τον πυρήνα p_gemm.
2. Να μπορεί να λύσει προβλήματα ρεαλιστικά μεγάλων διαστάσεων.
3. Να υποστηρίζει εκτέλεση με δεδομένα να βρίσκονται είτε στον host ή στις συσκευές, όπως το PARALiA.
4. Ιδανικά, να αξιοποιεί μεγάλο ποσοστό του εκάστοτε υλικού που χρησιμοποιείται.

Θέτοντας τους παραπάνω στόχους και βάσει της θεωρητικής μελέτης, επιλέγουμε να υλοποιήσουμε μια μορφή του 2.5D αλγορίθμου του Cannon, με εκτέλεση σε GPUs. Ο βασικός αλγόριθμος που χρησιμοποιείται βασίζεται στο έργο των [40] με μια μικρή επιρροή από το έργο των [51].

2.2.1 Αποσύνθεση Προβλήματος

Έστω ότι συμμετέχουν p διεργασίες, κάθεμια αποτελώντας μια συσκευή GPU. Σχηματίζουμε ένα τρισδιάστατο πλέγμα διεργασιών, διαστάσεων d_{Row} x d_{Col} x c με $d_{Row} = d_{Col} = \sqrt{p/c}$, με κάθε διεργασία να χαρακτηρίζεται από τρεις δείκτες, P_{ijk}. Κάθε $k \in [0, c-1]$ αποτελεί και ένα διαφορετικό δισδιάστατο πλέγμα που εκτελεί το $1/c$ του υπολογιστικού μέρους του αλγορίθμου, ενώ τα $i, j \in [0, d_{Row}]$ δείχνουν την θέση της διεργασίας στο εκάστοτε δισδιάστατο πλέγμα. Θεωρούμε πίνακες αρχικά κατανεμημένους με 2D Block Sequential τρόπο στις διεργασίες με P_{ijk} με $k = 0$, δηλαδή, οι διεργασίες P_{ij0} κατέχουν τα block A_{ij}, B_{ij}, C_{ij}. Στην αρχή της εκτέλεσης, οι ίδιες διεργασίες στέλουν μέσω broadcast τους τοπικούς τους πίνακες σε κάθε διεργασία που έχουν κοινό (i, j), δηλαδή, που βρίσκονται στην ίδια θέση στα αντίστοιχα δισδιάστατα επίπεδα με διαφορετικά k. Το αρχικό shift, δηλαδή η ανταλλαγή των τοπικών πινάκων A_{ij}, B_{ij} μεταξύ 2 διεργασιών, γίνεται ως... Σημαντική λεπτομέρεια είναι ότι η κάθε διεργασία γνωρίζει το αναγνωριστικό του παραλήπτη αλλά **ΔΕΝ** γνωρίζει το αναγνωριστικό του αποστολέα των τοπικών πινάκων που θα παραλάβει. Μετά από κάθε shift δεδομένων, κάθε διεργασία υπολογίζει το τοπικό υπο-πίνακα C_{ij}.

Η υπόλοιπη εκτέλεση του αλγορίθμου εξαρτάται από την τιμή c. Ο 2D αλγόριθμος λύνεται σε $\sqrt{p}$ βήματα, ενώ ο 2.5D λύνεται σε $\sqrt{p/c^3}$ βήματα. Τα περισσευόμενα βήματα γίνονται όπως ακριβώς και ο 2D αλγόριθμος, δηλαδή, γίνονται διαδοχικά shift προς τα αριστερά του πίνακα A και shift προς τα πάνω του πίνακα B ως προς το κάθε δισδιάστατο επίπεδο διεργασιών. Όπως και πριν, μετά από κάθε shift εκτελείται μια κλήση στα τοπικά δεδομένα κάθε διεργασίας. Τέλος, οι διεργασίες που έχουν κοινό (i, j) εκτελούν κάνουν reduce με άθροισμα τον πίνακα C_{ij}. Αναλυτικά, ο αλγόριθμος είναι αυτός που περιγράφθηκε με ψευδοκώδικα στο 6.

Ο 2.5D αλγόριθμος προσφέρει πολλά πρακτικά πλεονεκτήματα από τους υπόλοιπους αλγορίθμους που εξετάσαμε στο προηγούμενο κεφάλαιο. Αρχικά, ρυθμίζοντας την τιμή c

μπορούμε εύκολα να εναλλάξουμε μεταξύ της 2D και της 3D εκδοχής, χωρίς κάποια περαιτέρω αλγοριθμική αλλαγή. Πρακτικά, χρησιμοποιώντας c αντιγραφές των δεδομένων εισόδου, μειώνεται το κόστος bandwidth κατά $\sqrt{c}$, δηλαδή από $W_p = O(n^2/\sqrt{p})$ μεταβαίνει σε $W_p = O(n^2/\sqrt{cp})$.

2.2.2 Υλοποίηση με MPI

Η πρώτη υλοποίηση έγινε με CUDA-Aware MPI ώστε να επιτυγχάνεται άμεσα η επικοινωνία μεταξύ συσκευών και να μην χρειάζεται η συμμετοχή των host μεταξύ κόμβων. Η επίλυση του τοπικού προβλήματος γίνεται με χρήση της cuBLAS. Λόγω της στατικής φύσης του αλγορίθμου, η μνήμη που χρειάζεται διεργασία (της τάξεως των $O(M^2/d_{Row})$) δεσμεύεται από την αρχή, πριν την εκτέλεση, είτε σε host ή σε GPU, ανάλογα με τις διαστάσεις του προβλήματος. Επίσης, στην τωρινή μορφή του, η επικάλυψη επικοινωνίας και υπολογισμού δεν είναι εφικτή, άρα δεν υπάρχει ανάγκη για δημιουργία streams. Οι μετακινήσεις δεδομένων και η εκτέλεση των κλήσεων cuBLAS γίνονται όλες στο default stream της κάθε GPU. Οι περισσότερες κλήσεις MPI που χρησιμοποιούνται είναι ασύγχρονες και γίνεται πρόσθεση κατάλληλου συγχρονισμού όπου χρειαστεί.

Communicators

Για ευκολότερη σύνθεση των συναρτήσεων, χρησιμοποιούμε MPI Communicators για την οργάνωση του κώδικα όπου αυτό είναι εφικτό. Συγκεκριμένα, χρησιμοποιούμε δύο ειδών communicator:

- Οι διεργασίες P_{ijk} με κοινό (i, j) βρίσκονται στις ίδιες θέσεις στα εκάστοτε δισδιάστατα επίπεδα για κάποιο δοσμένο k και τοποθετούνται σε κοινό communicator (Common Stack Communicator).
- Οι διεργασίες P_{ijk} με κοινό k βρίσκονται όλες στο ίδιο δισδιάστατο επίπεδο και τοποθετούνται σε κοινό communicator (Common Grid Communicator). Ειδικά για $k = 0$ βρίσκεται το communicator στο οποίο έχει γίνει η αποσύνθεση των αρχικών πινάκων.

Οι Common Stack Communicator χρησιμοποιούνται στις ενέργειες broadcast και reduce αντίστοιχα στην αρχή και στο τέλος του αλγορίθμου. Οι Common Grid Communicator χρησιμοποιούνται για την 2D εκτέλεση του αλγορίθμου Cannon ανά επίπεδο (ανά k δηλαδή) και για την αρχική αποσύνθεση των δεδομένων.

Λειτουργίες Shift

Το μεγαλύτερο κόστος επικοινωνίας συμβαίνει στην διαδικασία ανταλλαγής πινάκων, δηλαδή στα shifts δεδομένων. Η ανταλλαγή γίνεται με την συνάρτηση MPI_Isendrecv_replace, η οποία αναλαμβάνει ταυτόχρονα την αποστολή και παραλαβή δεδομένων στην ίδια θέση μνήμης, σειριοποιώντας τις μεταφορές ανάλογα με το ποια διαδικασία ολοκληρώθηκε πρώτα. Η συνάρτηση καλείται 2 φορές για κάθε βήμα του αλγορίθμου, μια για κάθε πίνακα (A και B). Υπενθυμίζουμε πως οι διεργασίες δεν γνωρίζουν το αναγνωριστικό της διεργασίας αποστολέα από τον οποίο λαμβάνουν τους υποπίνακες. Στην θέση της τιμής *source*, τοποθετούμε την ειδική

επιλογή MPI_ANY_SOURCE. Εφόσον η επικοινωνία είναι διεργασία-προς-διεργασία, δεν γίνεται κάποιος να λάβει λάθος υπο-πίνακα. Πιθανή είναι όμως η παραλαβή δεδομένων του υπό-πίνακα B στην θέση του A ή το αντίθετο. Για να αποφεχθεί αυτό, πολύ απλά χρησιμοποιούμε διαφορετικά *tags* μηνύματος για κάθε υπο-πίνακα.

2.2.3 Υλοποίηση με NCCL

Κατά την αξιολόγηση του αλγορίθμου, παρατηρήσαμε χαμηλές συνολικές επιδόσεις. Κάνοντας πρόχειρη ανάλυση επιδόσεων, σημειώσαμε πολύ αυξημένο χρόνο επικοινωνίας, ειδικά στις κλήσεις ανταλλαγής δεδομένων (MPI_Isendrecv_replace). Μελετώντας παραπάνω την υποστήριξη κλήσεων MPI με περασμένα ορίσματα δείκτες μνήμης GPU, επιβεβαιώσαμε ότι υποστηρίζεται πλήρως η κλήση Sendrecv_replace, άρα αποκλείσαμε οποιοδήποτε ζήτημα συμβατότητας με συσκευές NVIDIA. Ο πραγματικός λόγος που καθυστερεί πολύ η επικοινωνία μεταξύ διεργασιών είναι το εσωτερικό buffering που εκτελεί το MPI για να υποστηρίξει την σειριοποίηση των εντολών Send και Recv σε κοινό buffer. Το στάδιο αυτό αναγκαστικά περνάει από μνήμη του host, μειώνοντας σημαντικά το throughput επικοινωνίας. Περαιτέρω, εφόσον η εκτέλεση του αλγορίθμου γίνεται με NVIDIA GPUs, θεωρήσαμε πιο ορθή την χρήση του NCCL στο τμήμα επικοινωνίας του πολλαπλασιασμού.

Το NCCL μπορεί να χρησιμοποιηθεί υβριδικά, μαζί με το MPI σε μια εφαρμογή. Ειδικά στην συγκεκριμένη περίπτωση όπου κλήσεις επικοινωνίας MPI και NCCL δεν θα υπάρξουν ποτέ ταυτόχρονα, δεν τίθεται ζήτημα thread-safety ή πιθανών race condition.

Διαφορές με την αρχική υλοποίηση

Ακριβώς όπως και στην προηγούμενη έκδοση, δημιουργούμε τους αντίστοιχους NCCL Communicators. Μια σημαντική διαφορά με το MPI και το NCCL, που σημειώθηκε και στο προηγούμενο κεφάλαιο, είναι ότι δεν υποστηρίζεται η δυναμική point-to-point επικοινωνία. Πρακτικά, αυτό σημαίνει ότι δεν υπάρχει αντιστοιχία του MPI_ANY_SOURCE στις κλήσεις ncclSend και ncclRecv. Για να λύσουμε αυτό το πρόβλημα, πριν την ανταλλαγή των υπο-πινάκων, κάθε διεργασία λαμβάνει το αναγνωριστικό της διεργασίας από την οποία θα λάβει δεδομένα μέσω κλήσεων MPI_Sendrecv και δυναμικής επικοινωνίας. Η διαδικασία αυτή προσθέτει μηδαμινό κόστος και δεν επηρεάζει την επίδοση του τελικού συστήματος.

Επίσης, το NCCL δεν υποστηρίζει in-place point-to-point επικοινωνία, δηλαδή αποστολή και παραλαβή ταυτόχρονα από τον ίδιο buffer. Αναγκαία προσθήκη είναι ένας δεύτερος buffer για τους πίνακες A και B ώστε να ανταλάζονται αποδοτικά οι υπο-πίνακες, γεγονός που πρακτικά διπλασιάζει τις απαιτήσεις για μνήμη ανά συσκευή για την εκτέλεση ενός προβλήματος. Στο NCCL, μια εντολή Send+Recv γίνεται μέσω ενός Group Call. Τα Group Functions χρησιμοποιούνται για την συγχώνευση πολλαπλών κλήσεων σε μία, δημιουργώντας πολύπλοκες ρουτίνες επικοινωνίας και πιθανότατα αυξάνοντας την επίδοση. Ένα σύνολο κλήσεων μετατρέπεται σε Group Call επικαλύπτοντάς τες με τις συναρτήσεις *ncclGroupStart()*/*ncclGroupEnd()* αντίστοιχα στην αρχή και στο τέλος.

```
ncclGroupStart();
ncclSend(localA[0], localM*localK, ncclDouble,
```

```
        receiverRank, ncclCommunicator, stream);
    ncclRecv(localA[1], localM*localK, ncclDouble,
        senderRank, ncclCommunicator, stream);
    ncclGroupEnd();
```

Listing 2.1: Παράδειγμα Group Call - Αντίστοιχη κλήση με MPI_Sendrecv

Οι broadcast και gather κλήσεις είναι ίδιες με τις αντίστοιχες MPI και η αντικατάστασή τους είναι απλή και άμεση.

Διαχείριση CUDA Streams

Οι κλήσεις επικοινωνίας NCCL δέχονται, προεραιτικά, όρισμα CUDA Stream, επιτρέποντας ταυτοχρονισμό στις μεταφορές, όπου αυτός είναι εφικτός. Οι κλήσεις NCCL είναι blocking, με την έννοια ότι μια συνάρτηση επιστρέφει τον έλεγχο στο νήμα εκτέλεσης όταν η ρουτίνα επικοινωνίας έχει ανατεθεί στο stream και όχι όταν αυτή ολοκληρωθεί. Προσθέτουμε κατάλληλα CUDA Streams στην επικοινωνία και στις κλήσεις cuBLAS. Συνολικά, δημιουργούμε $\sqrt{p/c^3}$ streams και events, δηλαδή, όσα και τα βήματα του αλγορίθμου. Ο πίνακας C μοιράζεται από όλα τα βήματα του αλγορίθμου, αφού ανανεώνεται κάθε φορά με το γινόμενο $A_{ij} * B_{ij}$. Για τον λόγο αυτό, οι κλήσεις cuBLAS μοιράζονται ένα stream (όχι το default), σειριοποιώντας τα gemm tasks, εφόσον δεν είναι δυνατή η ταυτόχρονη εκτέλεση. Κάθε stream επικοινωνίας αναλαμβάνει την ανταλλαγή του τοπικού πίνακα A και B για εκείνο το βήμα, εκτελώντας event record στο τέλος του Group Call. Αφού τα stream επικοινωνίας και υπολογισμού είναι ξεχωριστά, γίνονται κλήσεις *cudaStreamWaitEvent()* για να συγχρονιστούν οι μεταφορές δεδομένων με την έναρξη του υπολογισμού.

Η παραπάνω διαδικασία δεν μπορεί να θεωρηθεί επικάλυψη επικοινωνίας/υπολογισμού. Η φύση του αλγορίθμου αποτρέπει οποιαδήποτε μορφή επικάλυψης αφού αναγκαστικά πρέπει να ολοκληρωθεί η μετακίνηση των δεδομένων για να ξεκινήσει ο υπολογισμός εκείνου του βήματος και για να ξεκινήσει το επόμενο βήμα πρέπει να ολοκληρωθεί ο υπολογισμός του προηγούμενου. Η προσθήκη όμως των Streams και Events δίνουν μία έτοιμη ροή εκτέλεσης στην συσκευή, αφήνοντας πλήρως την δρομολόγηση της στο υλικό και, ιδανικά, μειώνοντας το latency εκτέλεσης. Επίσης, αποτελεί πρόδρομος για την τελική τεχνική βελτίωσης επίδοσης του αλγορίθμου.

Οι επιδόσεις που επιτυγχάνει η έκδοση με NCCL είναι σημαντικά καλύτερες από αυτή της σκέτης MPI εκδοχής. Αναλυτική σύγκριση μεταξύ των δύο καθώς και των υπολογιστικών και επικοινωνιακών σταδιών παρουσιάζονται στο επόμενο κεφάλαιο.

2.2.4 Προσθήκη Tiling

Μέχρι στιγμής έχουμε περιγράψει τον κλασσικό αλγόριθμο του Cannon, χωρίς κάποια ουσιώδη μετατροπή για εκτέλεση σε GPUs. Παρατηρούμε όμως μια βελτιστοποίηση η οποία είναι πιθανή χωρίς κάποια αύξηση στις συνολικές απαιτήσεις μνήμης. Εμπνεόμενοι από τις βιβλιοθήκες εκτέλεσης σε συστήματα κοινού χώρου διευθύνσεων, μπορούμε να εφαρμόσουσε τεχνική tiling για να εισάγουμε περιθώριο επικάλυψης επικοινωνίας/υπολογισμού.

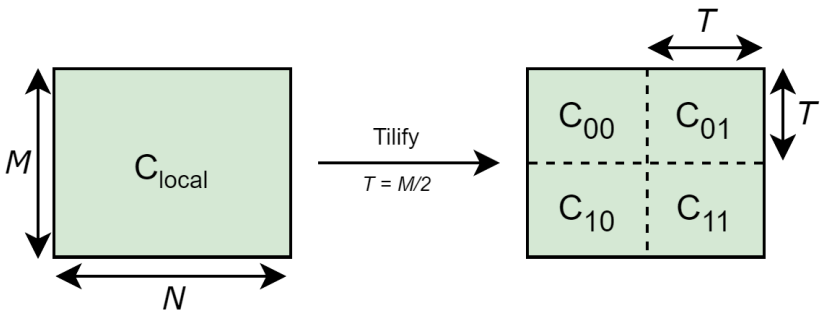
Ο συνολικός χρόνος εκτέλεσης ενός αλγορίθμου είναι ίσος με $T_{total} = T_{communication} + T_{computation} - T_{overlap}$. Οι προηγούμενες εκδόσεις είχαν $T_{overlap} = 0$. Διαιρώντας τους τοπικούς υπο-πίνακες κάθε διεργασίας σε tiles, μπορούμε να παραλληλοποιήσουμε τα νέα task, πρακτικά αυξάνοντας τον όρο του χρονικού overlap.

Δόμηση επικάλυψης - Ταυτοχρονισμός Task

Για να παραλληλοποιηθεί η διαδικασία, η εφαρμογή του tiling γίνεται σε τοπικό επίπεδο ανά διεργασία αντί να πραγματοποιηθεί στον πίνακα εισόδου. Σε τοπικό πίνακα διαστάσεων M x N (δεν σχετίζονται με τις διαστάσεις M, N, K του προβλήματος), επιλέγουμε διάσταση T ώστε να διαιρέσουμε τον αρχικό πίνακα. Οι κανόνες επιλογής αυτού του μεγέθους είναι οι ακόλουθοι:

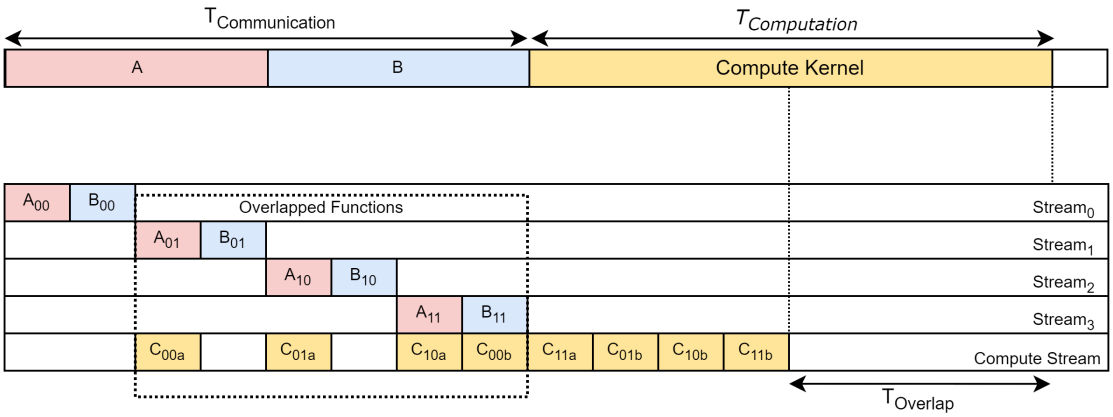
- Ιδανικά, να βρεθεί μια τιμή ώστε το υπόλοιπο των διαιρέσεων $\frac{M}{T}$ και $\frac{N}{T}$ είναι ίσο με μηδέν, ώστε να μην χρειαστεί padding.
- Να δημιουργηθούν tiles κατάλληλα σε μέγεθος ώστε να επιτυγχάνει μέγιστες υπολογιστικές επιδόσεις η GPU.
- Να δημιουργηθούν tiles αρκετά σε αριθμό ώστε να επιτυγχάνεται αρκετή επικάλυψη.

Ειδικά για την συγκεκριμένη εφαρμογή, η διαίρεση του τοπικού πίνακα σε tiles είναι αρκετά απλή και δεν χρειάζεται μοντελοποίηση για την επιλογή κατάλληλων διαστάσεων. Οι τοπικοί πίνακες κάθε διεργασίας είναι τετράγωνοι με διάσταση $local_M$ x $local_M$ με $local_M = \frac{M}{d_{Row}}$. Επιλέγοντας στατικά το μέγεθος του tile ώς $T = \frac{local_M}{2^n}, n \in N$, δημιουργούμε 2^{2n} κομμάτια πίνακα, ένα αρκετά ελεγχόμενο μέγεθος για τις ανάγκες του αλγορίθμου.



Σχήμα 2.2: Μετατροπή τοπικού πίνακα σε Tiles

Για την μεταφορά, δημιουργούμε ένα stream για κάθε αριθμό tile που χωρίστηκε ένας πίνακας και για τον υπολογισμό μόλις ένα tile ανά διεργασία. Έτσι, οι μεταφορές μεταξύ βημάτων στο ίδιο tile συγχρονίζονται εντός των stream. Ο λόγος που χρησιμοποιούμε ένα stream υπολογισμού είναι ότι η πιθανότητα να τρέξουν παράλληλα δύο cuBLAS kernels τέτοιων διαστάσεων είναι απίθανη, άρα η τοποθέτησή τους σε διαφορετικά streams δεν προσφέρει κάποιο ουσιώδες πλεονέκτημα. Η μεταφορά των tiles γίνεται σειριακά, δηλαδή πρώτα ανταλλάσονται τα tiles A_0, B_0, έπειτα τα A_1, B_1, κ.λ.π. Μετά την κλήση των εντολών *ncclRecv()*, δηλαδή την παραλαβή των tiles, τοποθετούνται events, ώστε να συγχρονιστούν οι αργότερες εκτελέσεις των cuBLAS kernels με την παραλαβή των δεδομένων εξαρτήσεων.



Σχήμα 2.3: Απεικόνιση Streams στην λειτουργία επικάλυψης επικοινωνίας/υπολογισμού

Παρατηρούμε και από το σχήμα 2.3 ότι οι μεταφορές δεδομένων δεν είναι εφικτό να γίνουν παράλληλα. Αυτό δεν ισχύει για όλες τις μεταφορές δεδομένων, αλλά συγκεκριμένα για μεταφορές που γίνονται με ίδιες κατευθύνσεις μεταξύ 2 συσκευών. Στον αλγόριθμο του Cannon, όπου η επικοινωνία λαμβάνει την μορφή βηματικών shift, αναγκαστικά χρησιμοποιούνται μόνο 2 συσκευές με ίδιες κατευθύνσεις πάντα, μειώνοντας την μέγιστη επικάλυψη που μπορεί να επιτευχθεί. Παρ'όλα αυτά, χρησιμοποιούνται πολλαπλά streams στο κομμάτι της επικοινωνίας για διευκόλυνση στον προγραμματισμό.

2.2.5 Προγραμματιστικό Πακέτο

Εμπνεόμενοι από την προγραμματιστική εμπειρία παρόμοιων βιβλιοθηκών, το σύστημα προσφέρεται ως μια κλάση C++. Οι συναρτήσεις που έχει πρόσβαση ο χρήστης είναι αυτές της αποσύνθεσης των πινάκων (decomposition) που δίνει σαν είσοδο, του πολλαπλασιασμού (multiply) και της επιστροφής των δεδομένων στην διεργασία που κατείχε αρχικά τα δεδομένα (gather ή reorder), αντίστοιχα όπως βιβλιοθήκες σαν το SLATE. Αυτή την στιγμή, υποστηρίζονται μόνο κλήσεις Double Precision (dgemm), αλλά η επέκταση για υπόλοιπες ακρίβειες είναι πολύ απλή με χρήση δυνατοτήτων template της C++.

Η πιο συνηθισμένη χρήση σε βιβλιοθήκες που εκτελούνται σε clusters είναι η εκτέλεση με δεδομένα ήδη διαμοιρασμένα στις διεργασίες. Για αυτό τον λόγο, υπάρχει και ειδικός constructor που λαμβάνει την θέση μνήμης των δεδομένων και αναλύει στατικά το πρόβλημα, χωρίς να εκτελεί αποσύνθεση.

Chapter 3

Αξιολόγηση

Στο κεφάλαιο αυτό παρουσιάζουμε τις μεθόδους αξιολόγησης που χρησιμοποιήθηκαν για να δημιουργηθούν καθαρές γραφικές παραστάσεις και να εξαχθούν συμπεράσματα. Ολόκληρο το στάδιο της αξιολόγησης έγινε στο Meluxina.

3.1 Μεθοδολογία Αξιολόγησης

Για δίκαιη και μεθοδευμένη σύγκριση, ακολουθούμε την ίδια διαδικασία αξιολόγησης που χρησιμοποιούν αξιοσημειώτα ερευνητικά έργα όπως το SLATE [46] και το COSMA [55], με τα οποία συγκρινόμαστε κίολας.

3.1.1 Σύστημα Εκτέλεσης

Αναλυτικά αναφέρονται τα σημαντικά χαρακτηριστικά υλικού και λογισμικού των κόμβων που χρησιμοποιήσαμε.

CPU	2x AMD EPYC 7452
GPU	4x NVIDIA A100 40GB PCIe
RAM	512GB
GCC	12.3.0
CUDA	12.2.0
MPI	OpenMPI 5.0.5 - CUDA Aware w/ GPUDirect - Compiled with GCC-12.3.0

3.1.2 Μέτρηση Επίδοσης

Σημειώνουμε μεταξύ των τμημάτων κώδικα όπου συμβαίνει επικοινωνία ή υπολογισμός τον χρόνο που χρειάστηκε για να ολοκληρωθεί η εντολή. Σε κλήσεις MPI χρησιμοποιούμε την εντολή *MPI_Wtime()* στην αρχή και το τέλος της εντολής για να εξάγουμε τον χρόνο εκκίνησης και ολοκλήρωσης της εντολής. Εκτελούμε *MPI_Reduce()* με πράξη *MPI_MAX* στην διαφορά αυτών των δύο χρόνων για να βρούμε τον μέγιστο χρόνο εκτέλεσης από όλες τις διεργασίες. Πριν κληθεί η *MPI_Wtime()* στο τέλος της εντολής που εξετάζουμε, βεβαιώνουμε να συγχρονίσουμε όλες τις διεργασίες μέσω της *MPI_Barrier()*.

Αντίστοιχα για συναρτήσεις CUDA, χρησιμοποιούμε την ίδια εντολή MPI για τον χρόνο και για το reduction στις σύγχρονες κλήσεις. Σε ασύγχρονες συναρτήσεις, χρησιμοποιούμε CUDA Events στην αρχή και στο τέλος της εντολής που εξετάζουμε και βρίσκουμε τον χρόνο μέσω της *cudaEventElapsedTime()*, η οποία υπολογίζει τον χρόνο μεταξύ δύο event σε millisecond.

Η τελική μετρική επίδοσης που μας ενδιαφέρει περισσότερο μετριέται σε FLOPs (FLoating point OPerations per second) και δείχνει πόσες εντολές κινητής υποδιαστολής κατάφερε να εκτελέσει η συσκευή κατά την διάρκεια του πολλαπλασιασμού. Ιδανικά, σκοπός είναι να επιτυγχάνεται τιμή όσο πιο κοντά στο θεωρητικό μέγιστο της εκάστοτε συσκευής σε compute-bound εφαρμογές. Στην δική μας περίπτωση, compute-bound εφαρμογές αποτελούν οι τετράγωνοι πίνακες ενώ όσο πιο μεγάλη είναι η απόκλιση της τιμής K από τις M, N τόσο πιο memory-bound είναι και το πρόβλημα. Για μια κλίση GEMM, η τιμή TFLOPs (σε Terabyte) μετριέται ως: $TFLOPs_{GEMM} = 10^{-12} \times \frac{2MNK}{executionTime}$.

3.1.3 Σενάριο εκτέλεσης

Στην αξιολόγηση αλγορίθμων p_gemm χρησιμοποιείται κοινό σενάριο εκτέλεσης με το οποίο συγκρίνονται οι επιδόσεις των συστημάτων. Το σενάριο αυτό θεωρεί δεδομένα ήδη κατανεμημένα στις διεργασίες βάσει την αποσύνθεση δεδομένων κάθε αλγορίθμου. Ουσιαστικά θέλουμε η εκτέλεση να θυμίζει πραγματική εκτέλεση p_gemm σε κάποια επιστημονική εφαρμογή. Σε τέτοιες περιπτώσεις, η διαμόρφωση του πλέγματος διεργασιών και η αποσύνθεση των πινάκων έχει γίνει από προηγούμενο στάδιο για λόγους επιδόσεων.

Αναλυτικά, παρουσιάζουμε την ροή εκτέλεσης των πειραμάτων για κάθε σύστημα.

- Διαμοιράζουμε τους πίνακες A, B, C στις διεργασίες είτε στην μνήμη του host ή στην συσκευή (αναλόγως το σύστημα).

- Εκτελούμε συναρτήσεις Warmup ώστε οι συσκευές να μην βρίσκονται σε idle κατάσταση πριν την εκτέλεση (10 εκτελέσεις).

- Εκτελούμε κλήση gemm. Μετράμε αυτόν τον χρόνο ως χρόνο εκτέλεσης (50 εκτελέσεις).

- **Προεραιτικό**: Αν ο αλγόριθμος χρειάζεται στάδιο reduction ώστε ο πίνακας C να βρίσκεται ανανεωμένος με το τελικό αποτέλεσμα στις ίδιες διεργασίες που βρισκόταν και αρχικά, εκτελείται και αυτή η διαδικασία. Ο χρόνος αυτός συμπεριλαμβάνεται στον χρόνο εκτέλεσης.

- Καλούμε συναρτήσεις συγχρονισμού, πρώτα *cudaDeviceSynchronize()* και μετά *MPI_Barrier()* ώστε να βεβαιωθούμε ότι όλες οι συσκευές και διεργασίες είναι διαθέσιμες και δεν εκτελούν task. Ο χρόνος εκτέλεσης τελειώνει εδώ.

Έχοντας σημειώσει τον χρόνο εκτέλεσης για τις 50 εκτελέσεις (μετά από συγχρονισμό για καθεμιά), παίρνουμε την μέση τιμή και υπολογίζουμε τα TFLOPs που κατάφερε το εκάστοτε σύστημα. Η διαδικασία αυτή επαναλαμβάνεται για όλα τα μεγέθη πινάκων που αποφασίζουμε να δοκιμάσουμε.

3.1.4 Συστήματα Σύγκρισης

Εξηγούμε συνοπτικά τις λειτουργίες και δυνατότητες που προσφέρουν τα συστήματα με τα οποία συγκρινόμαστε καθώς και τις προγραμματιστικές διεπαφές που χρησιμοποιούν για εκτέλεση πολλαπλασιασμού.

cuBLASMp

Το cuBLASMp είναι η λύση της NVIDIA για εκτέλεση p_gemm. Υποστηρίζει δεδομένα κατανεμημένα με 2D Block Cyclic τρόπο και δέχεται ορίσματα σε μορφή παρόμοια με αυτά που δέχεται και το ScaLAPACK, απλά με διαφορετικούς τύπους. Η απόφαση αυτή αν και επιτρέπει για εύκολη μετασκευή (retrofitting) έτοιμων εφαμοργών ώστε να εκτελούνται σε GPUs, δυσκολεύει αρκετά την δημιουργία νέων εφαρμογών καθώς η μορφή δεδομένων που υποστηρίζει το ScaLAPACK είναι αρκετά απαρχαιωμένη.

Όπως και το cuBLAS, αναθέτουμε ένα stream στο οποίο εκτελούνται τα task της βιβλιοθήκης. Για κάθε πίνακα, δημιουργούμε έναν Matrix Descriptor, μεταφέρουμε τα δεδομένα στις μνήμες των GPU και έπειτα μπορούμε να ξεκινήσουμε την εκτέλεση της κλήσης p_gemm. Εφόσον χρησιμοποιεί αποσύνθεση δεδομένων 2D Block Cyclic, πρέπει να επιλέξουμε και κατάλληλο blocking μέγεθος για την κατανομή των δεδομένων. Αυτό θα καθορίζει και την επίδοση στην εκτέλεση των υπό-ρουτίνων gemm και την συνολική επικοινωνία που θα χρειαστεί να γίνει. Πειραματικά έγιναν εκτελέσεις σε πολλαπλά μεγέθη εισόδου και με διαφορετικά blocking size, αποφασίστηκε στατικά η χρήση $M_b = N_b = \{2048, 4096\}$, αναλόγως το μέγεθος των πινάκων και τις διαστάσεις του πλέγματος διεργασιών.

Η υλοποίηση είναι κλειστού κώδικα - η NVIDIA δεν αναφέρει λεπτομέρειες σημαντικές για την ανάλυση της εκτέλεσης. Για αυτό τον λόγο, δεν γνωρίζουμε αν χρησιμοποιεί τεχνικές communication/computation overlap ή lookahead και δεν μπορούμε να προσφέρουμε σχόλια ή προβλέψεις για την επίδοσή του.

SLATE

Το SLATE ακολουθεί μια πιο μοντέρνα λογική στην σχεδίασή του. Η χρήση των μαθηματικών εργαλείων που προσφέρει βασίζεται σε μια ισχυρή δομή δεδομένων κατανεμημένου πίνακα. Σε αυτή την δομή πρακτικά συμπεριλαμβάνονται όλες οι αναγκαίες πληροφορίες που χρειάζονται για την εκτέλεση των κατανεμημένων ρουτινών PBLAS και πραγματοποιούνται όλες οι λειτουργίες διαχείρισης μνήμης από μεριάς Host και GPU. Το SLATE προσφέρει μια πολύ χρήσιμη ιδιότητα, την υβριδική τοποθέτηση tiles σε host και GPU memory. Για ομοιογενή εκτέλεση σε GPUs, τέτοιου είδους τοποθέτηση σε αντίθεση με την τοποθέτηση μόνο σε μνήμη GPU μειώνει την επίδοση που μπορεί να επιτευχθεί, αφού απαιτούνται περαιτέρω μετακινήσεις μεταξύ συσκευών πριν την εκτέλεση task, όμως επιτρέπει και την εκτέλεση μεγαλύτερων προβλημάτων δεδομένου ότι χρησιμοποιείται και κατάλληλη τεχνική caching.

Στον τομέα της επικάλυψης επικοινωνίας/υπολογισμού, χρησιμοποιεί μια τεχνική που αποκαλούν lookahead. Βάσει ενός αριθμού l που αποφασίζεται πριν την κλήση των συναρτήσεων εκτέλεσης, κάθε διεργασία μπορεί να προετοιμάζει και να εκτελεί το στάδιο της επικοινωνίας ταυτόχρονα για τα l επόμενα βήματα. Πρακτικά, για την κλήση gemm, ενώ γίνεται fetched ένα

tile, ταυτόχρονα υπολογίζονται και προετοιμάζονται για αποστολή τα επόμενα l tiles τα οποία πρέπει να μεταφερθούν στην μνήμη της διεργασίας ώστε να γίνει ο υπολογισμός στα αντίστοιχα βήματα.

Όπως και το cuBLASMp, χρησιμοποιεί 2D Block Cyclic κατανομή και παρέχει απευθείας υποστήριξη με κλήσης ScaLAPACK. Η υποστήριξη αυτή επιτυγχάνεται μέσω ειδικών headers και object files που δημιουργούνται στην διαδικασία της εγκατάστασης της βιβλιοθήκης. Ο χρήστης μπορεί να κάνει αντικαταστήσει το ScaLAPACK με το SLATE απευθείας, κάνοντας link το δεύτερο αντί του πρώτου στην διαδικασία εγκατάστασης της εφαρμογής που τον ενδιαφέρει. Με αυτόν τον τρόπου, οι ήδη υπάρχοντες εφαρμογές που χρησιμοποιούν το ScaLAPACK ως μαθηματικό backend μπορούν να χρησιμοποιήσουν το SLATE άμεσα χωρίς την ανάγκη του σταδίου μετασκευής κώδικα.

Σημαντικοί παράμετροι κατά την εκτέλεση είναι πάλι το μέγεθος blocking της κατανομής και η τιμή lookahead. Όπως και με το cuBLASMp, δοκιμάσαμε διαφορετικά μεγέθη blocking σε πληθώρα διαστάσεων M, N, K και πλέγματα διεργασιών. Σε αντίθεση με το προηγούμενο όμως, το SLATE χρησιμοποιεί batched gemm κλήσεις, ιδανικά μικρών διαστάσεων tiles, οι οποίες εκτελούνται παράλληλα. Η χρήση μεγάλων block size θα δυσκόλευαν αυτή την παράλληλη εκτέλεσης, με τους ίδιους τους δημιουργούς να προτείνουν μικρά μεγέθη block, ώστε να επιτυγχάνουν καλή επίδοση και σε CPU αλλά και σε GPU [46]. Για τον λόγο αυτό, παρατηρούμε ότι τα καλύτερα μεγέθη blocking έιναι στην τάξη των $M_b = N_b = \{512, 1024\}$.

Για το lookahead τα πειράματα που εκτελέσαμε ήταν τα ίδια, με την μόνη παρατήρηση που έχουμε να κάνουμε είναι πως η αύξηση του lookahead επιφέρει και αύξηση στην επίδοση. Για αυτό τον λόγο, επιλέγουμε το μέγιστο δυνατό lookahead βάσει της διαθέσιμης μνήμης της GPU που μπορεί να χρησιμοποιηθεί από το εκάστοτε πρόβλημα. Συνοπτικά, $lookahead = min(\frac{gpuMemory}{numTilesNeededForComp \times memoryRequirementsPerTile}, numTilesNeededForComp)$

COSMA

Το COSMA, όντας κυρίως ερευνητικό έργο και όχι βιβλιοθήκη εκτέλεσης GEMM καθ'αυτου, προσφέρει μια πιο λιτή αλλά ισχυρή προγραμματιστική διεπαφή. Αρχικά, προσφέρει πλήρη υποστήριξη με κλήσεις p_gemm της ScaLAPACK με τον ίδιο ακριβώς τρόπο όπως το SLATE, δηλαδή με link κατά την εγκατάσταση της εφαρμογής. Το COSMA είναι η μόνη βιβλιοθήκη που επιτρέπει την εκτέλεση του ίδιου αλγορίθμου αλλά με οποιαδήποτε κατανομή δεδομένων θελήσει ο χρήστης. Χρησιμοποιώντας το COSTA (Communication Optimal Shuffle & Transpose Algorithm) [58] για βέλτιστες αναδιατάξεις κατανομών, αφήνει τον χρήστη να επιλέξει ακριβώς την κατανομή που βολεύει την δική του εφαρμογή. Στην δική μας περίπτωση, χρησιμοποιούμε την default κατανομή που χρησιμοποιεί το COSMA, η οποία ειναι και επικοινωνιακά βέλτιστη.

Οι μόνοι παράμετροι που αφήνει το COSMA στον χρήστη είναι η επιλογή για επικάλυψη επικοινωνίας/υπολογισμου, την οποία αφήνουμε πάντα ανοιχτή, και την διαθέσιμη μνήμη του μηχανήματος, την οποία αφήνουμε πάντα στο μέγιστο δυνατό.

Η υποστήριξη για εκτέλεση σε GPUs είναι κάτι πρόσθετο και όχι η ερευνητική βάση αυτού του αλγορίθμου. Για τον λόγο αυτό, τα δεδομένα των κατανεμημένων πινάκων βρίσκονται αρχικά στην μνήμη του host και όχι της συσκευής. Όμως, υποστηρίζει άμεσα το NCCL για επικοινωνία εντός και εκτός κόμβου, μια λειτουργία την οποία χρησιμοποιήσαμε κατά την εγκατάστασή

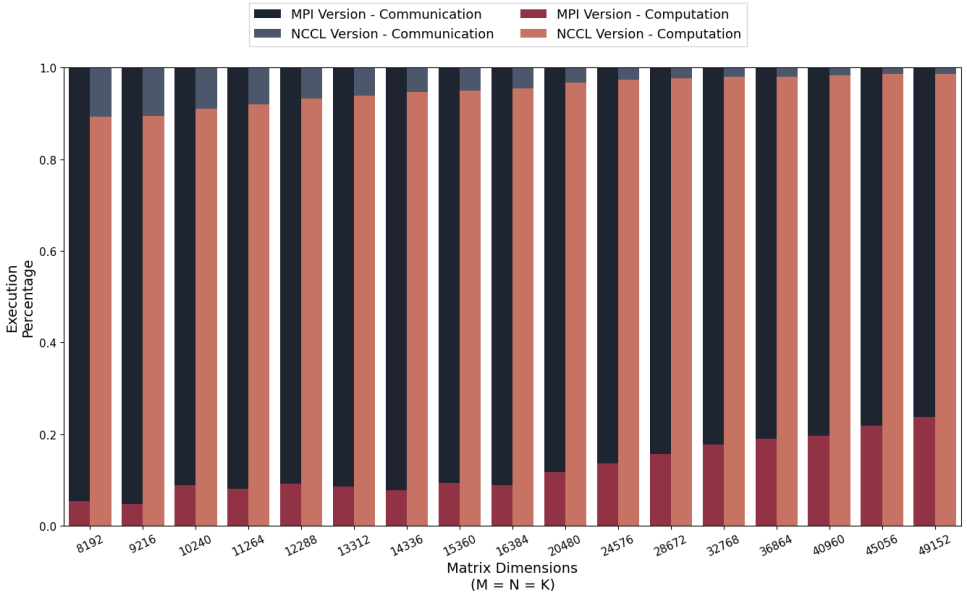
του, όπως προτείνουν και οι δημιουργοί για χρήση σε multi-node multi-gpu περιβάλλοντα. Η λεπτομέρεια αυτή είναι σημαντική για την αργότερη αξιολόγηση των αποτελεσμάτων του συστήματος.

3.2 Ανάλυση των σταδίων εκτέλεσης της 2.5D

Προβαίνουμε σε ανάλυση των σταδίων εκτέλεσης των υλοποιήσεων μας. Η ανάλυση αυτή είναι δύσκολη να συμβεί για τα υπόλοιπα συστήματα και δεν εμπίπτει στο πλάισιο αυτής της εργασίας. Επίσης, δεν υπάρχει λόγος ανάλυσης των σταδίων της Sequential PARALiA έκδοσης αφού δεν απαιτείται επικοινωνία μεταξύ κόμβων για την εκτέλεση του πολλαπλασιασμού.

3.2.1 Σύγκριση MPI και NCCL

Αρχικά, συγκρίνουμε τις δύο εκδόσεις με τα διαφορετικά επικοινωνιακά συστήματα. Η έκδοση με tiling δεν συμπεριλαμβάνεται αφού έχει ίδια στάδια με την απλή NCCL έκδοση, απλά υπάρχει επικάλυψη μεταξύ λειτουργιών εκτέλεσης και επικοινωνίας. Ο χρόνος αυτός, $T_{overlap}$, μπορεί να υπολογιστεί χονδρικά ως την διαφορά του χρόνου εκτέλεσης της απλής NCCL έκδοσης με την tiled έκδοση.

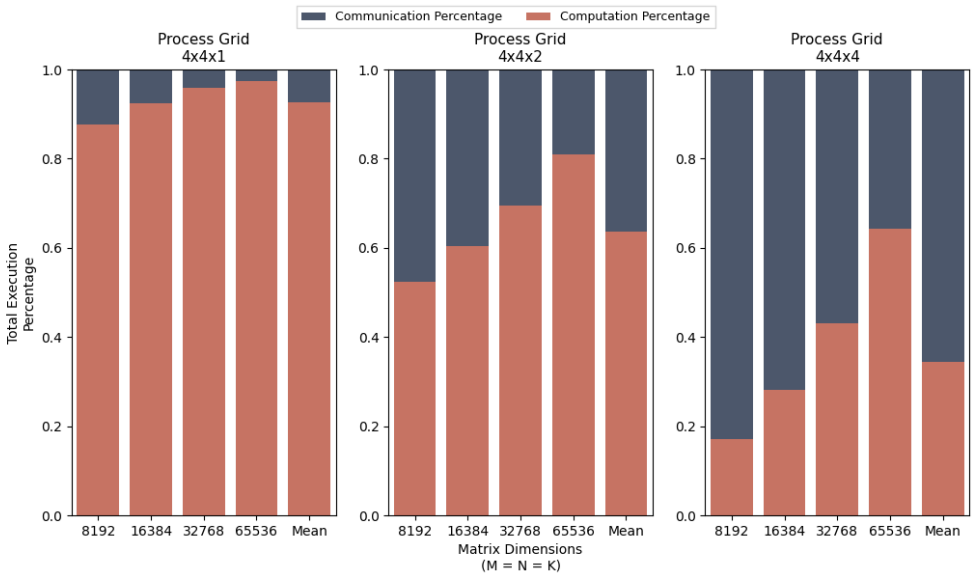


Σχήμα 3.1: Στάδια εκτέλεσης ανά διεργασία | MPI - NCCL

Αρχικά, εύκολα παρατηρούμε ότι στην MPI έκδοση το κυρίαρχο στάδιο είναι αυτό της επικοινωνίας ενώ στην NCCL αυτό του υπολογισμού. Γνωρίζοντας όμως ότι ο χρόνος εκτέλεσης του υπολογισμού είναι ακριβώς ίδιος μεταξύ των δύο, αυτό που βλέπουμε στην πραγματικότητα είναι το ποσοστό βελτίωσης της επικοινωνίας από το NCCL συγκριτικά με το MPI. Η σημαντική διαφορά στην επίδοση δικαιολογείται από το γεγονός ότι το MPI χρησιμοποιεί buffering στην μεριά του host για την σειριοποίηση των λειτουργιών *Send* και *Receive* στην κλήση *MPI_Sendrecv_Replace()*.

3.2.2 Σύγκριση για διαφορετικά μεγέθη c

Σε ίδιο δισδιάστατο πλέγμα, εξετάζουμε τα ποσοστά επικοινωνίας και υπολογισμού για διαφορετικές τιμές του c. Στις μετρήσεις μας, χρησιμοποιούμε πλέγμα $(4\ x\ 4\ x\ c)$, όπου $c \in \{1, 2, 4\}$. Έγιναν εκτελέσεις για $M = N = K$ με $M \in \{4096, 5120, 6144, ..., 65536\}$, δηλαδή $M = 4096 + (n-1) * 1024, n \in [1, 61]$.



Σχήμα 3.2: Στάδια εκτέλεσης για διαφορετικές τιμές c ανά διεργασία, σε ποσοστό

Το διάγραμμα παρουσιάζει τα ποσοστά επικοινωνίας και υπολογισμού ανά διεργασία κατά την εκτέλεση τετράγωνων προβλημάτων. Με αύξηση του όρου c, πάμε από εκτέλεση 2D προς εκτέλεση σε 3D. Όσο πιο κοντά φτάνουμε στο όριο της 3D εκτέλεσης, τόσο μεγαλώνει και ο αριθμός διεργασιών που συμμετέχουν στην συλλογική επικοινωνία, αυξάνοντας τον χρόνο εκτέλεσής τους. Επίσης, όσο μεγαλώνει η τιμή c, τόσο μειώνονται και τα βήματα του αλγορίθμου και εν τέλει τα βήματα υπολογισμού που εκτελεί κάθε διεργασία.

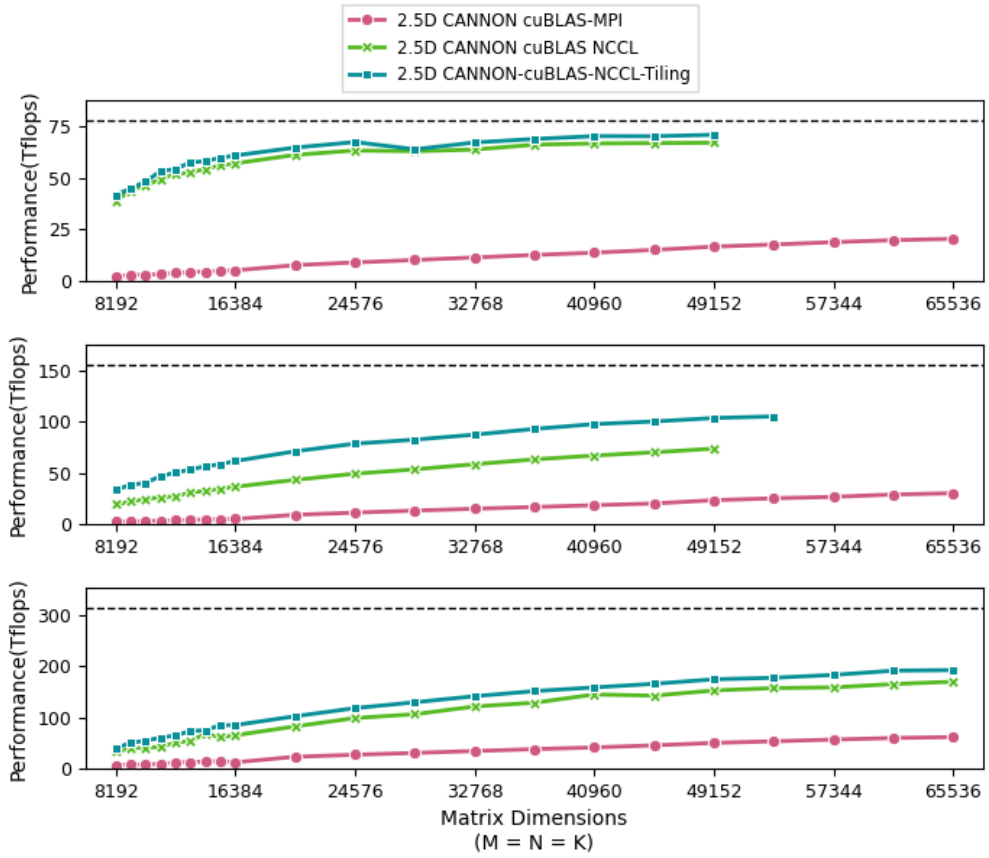
Εύκολα βγαίνουμε στο συμπέρασμα ότι αυξομειώνοντας τον παράγοντα αντιγραφής (replication factor) c μετατρέπουμε την εκτέλεση από compute dominant σε communication dominant.

3.2.3 Σύγκριση επιδόσεων σε εκτελέσεις GEMM

Έχοντας μία πιο ολοκληρωμένη εικόνα για τα στάδια εκτέλεσης του αλγορίθμου, προχωράμε σε εκτελέσεις GEMM και με τις 3 εκδοχές ώστε να δούμε τις μέγιστες επιδόσεις που επιτυγχάνουν. Οι εκτελέσεις γίνονται για αριθμό κόμβων $n = 1, 2, 4$ και ίδιες διαστάσεις πινάκων όπως και στην προηγούμενη σύγκριση.

Σε εκτελέσεις ενός κόμβου, οι επιδόσεις των εκδοχών με NCCL φτάνουν αρκετά κοντά στο θεωρητικό μέγιστο των συσκευών. Η Tiling έκδοση πάντα επιτυγχάνει καλύτερη επίδοση από την απλή, αριθμητικά κοντά σε 5-10% βελτίωση, ειδικά σε εκτελέσεις μεγαλύτερων προβλημάτων. Η MPI έκδοση είναι αρκετά πιο αδύναμη και καταφέρνει λιγότερο από 25% αξιοποίηση υλικού σε όλες τις διαστάσεις που δοκιμάσαμε.

Στις εκτελέσεις με αριθμό κόμβων 1 ή 2, παρατηρούμε ότι οι μέγιστες διαστάσεις που μπορεί να εκτελέσει τα συστήματα με NCCL είναι κοντά στα 50000 ενώ το MPI δεν δυσκολεύεται να



Σχήμα 3.3: Εκτελέσεις GEMM για 2.5D Cannon - Αριθμός κόμβων: 1, 2, 4 (από πάνω προς τα κάτω)

εκτελέσει μεγαλύτερα προβλήματα. Ο περιορισμός αυτός εξηγείται από την πρακτικά διπλάσια ανάγκη από μνήμη που έχουν τα συστήματα με NCCL αφού δεν υποστηρίζουν ανταλλαγή in-place σε λειτουργίες Send & Receive.

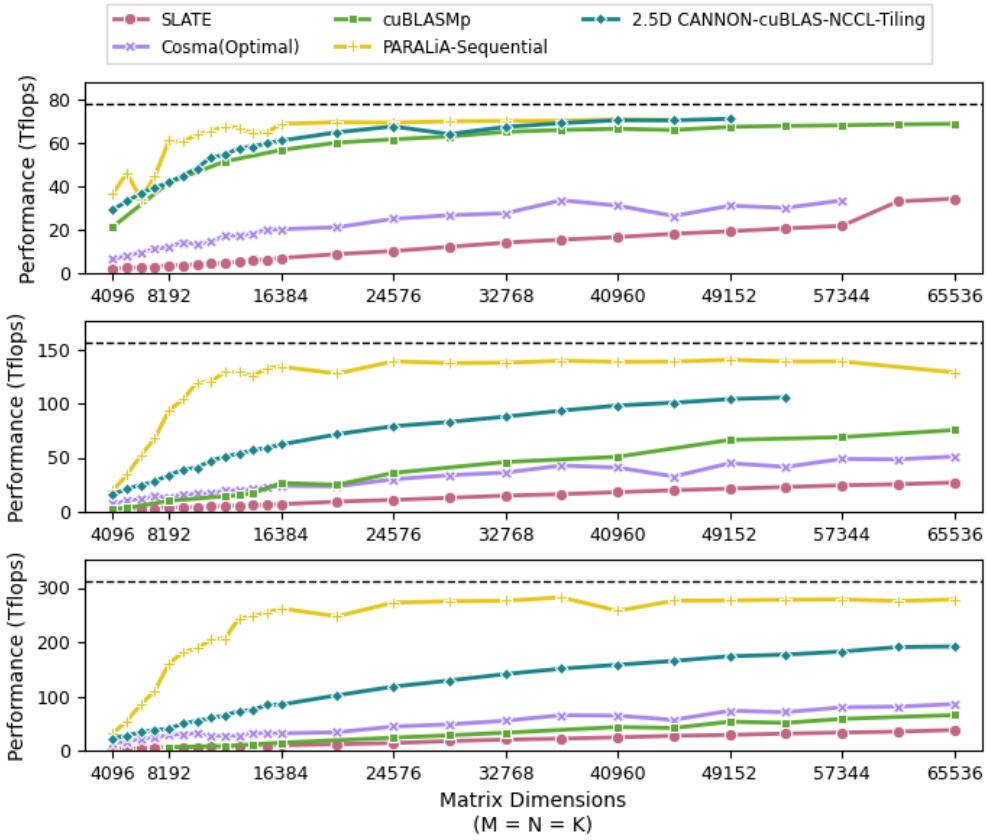
3.3 Εκτελέσεις GEMM - Συστήματα Σύγκρισης

Στα επόμενα πειράματα συμμετάσχουν τα συστήματα που υλοποιήσαμε καθώς και τα συστήματα σύγκρισης που περιγράψαμε προηγουμένως. Από τα δικά μας συστήματα, χρησιμοποιούμε μόνο την έκδοση με το Tiling για τον 2.5D Cannon καθώς πέτυχε τις καλύτερες επιδόσεις από τα 3.

3.3.1 Square Problems

Ξεκινάμε αρχικά με τετράγωνα προβλήματα. Όπως περιγράψαμε προηγουμένως, τα τετράγωνα προβλήματα συναντιούνται αρκετά συχνά σε εκτελέσεις κατανεμημένου πολλαπλασιασμού σε συστοιχίες και είναι compute-bound εφαρμογή. Ιδανικά, θέλουμε τα συστήματα να επιτυγχάνουν επίδοση όσο πιο δυνατά κοντινή στο θεωρητικό μέγιστο. Οι παράμετροι εκτέλεσης (αριθμός κόμβων και διαστάσεις πινάκων) είναι ίδιοι με το προηγούμενο υποκεφάλαιο.

Σε εκτελέσεις ενός κόμβου το cuBLASMp, το PARALiA και ο 2.5D αλγόριθμος καταφέρνουν αξιοσημείωτες επιδόσεις, κοντά στο θεωρητικό μέγιστο. Επίσης, συγκλίνουν αρκετά γρήγορα στην επίδοση αυτή, κάνοντας τα αποδεκτά για χρήση και σε προβλήματα μικρών διαστάσεων



Σχήμα 3.4: Εκτελέσεις GEMM σε τετράγωνα προβλήματα - Αριθμός κόμβων: 1, 2, 4 (από πάνω προς τα κάτω)

που κανονικά δεν εκτελούνται βέλτιστα σε κατανεμημένα περιβάλλοντα. Το COSMA και το SLATE πετυχαίνουν περίπου το 25% του θεωρητικού μέγιστου, ακόμα και σε μεγαλύτερα προβλήματα που είναι πιο ευνοϊκά στην εκτέλεση. Τα COSMA, PARALiA και 2.5D Cannon δεν καταφέρνουν να εκτελέσουν κάποια προβλήματα μεγαλύτερων διαστάσεων, ενώ κανονικά χωράνε στις μνήμες των συσκευών.

Αυξάνοντας τους κόμβους, τα 3 συστήματα που συγκρινόμαστε δυσκολεύονται να αξιοποιήσουν το επιπλέον υλικό που χρησιμοποιείται. Σημαντική εντύπωση μας αφήνει το COSMA, το οποίο εφαρμόζει communication avoidant αλγόριθμο - η αύξηση κόμβων θα έπρεπε να επιφέρει αρκετά καλύτερες επιδόσεις, ειδικά αφού χρησιμοποιεί το NCCL για γρήγορη επικοινωνία μεταξύ ranks.

Το SLATE και το cuBLASMp, που χρησιμοποιούν την ίδια κατανομή δεδομένων και εκτελούν μια μορφή 2D SUMMA, δυσκολεύονται να πετύχουν καλές επιδόσεις σε μικρά προς μέτρια προβλήματα, αλλά παρουσιάζουν μια μικρή άνοδο σε μεγαλύτερα προβλήματα, ειδικά όταν ο αριθμός των κόμβων είναι μικρός άρα η επικοινωνία είναι λιγότερη. Συγκεκριμένα για το SLATE το οποίο χρησιμοποιεί τεχνικές επικάλυψης, η απόφαση να εκτελεί batched gemm με χρήση μικρότερων tiles, το τιμωρεί στην εκτέλεση προβλημάτων μικρών διαστάσεων αφού το overhead επικοινωνίας και υπολογισμού επικαλύπτει αξιοποιήσιμα περιθώρια επίδοσης. Επίσης η κλιμακωσιμότητα του cuBLASMp το οποίο χρησιμοποιεί (μάλλον) μια απλοϊκή μορφή επικοινωνίας χωρίς κάποια μορφή παραλληλισμού επικοινωνίας και υπολογισμού, φαίνεται να

"σπάει" γρήγορα αφού η προσθήκη κόμβων δεν επιφέρει σημαντικές βελτιώσεις στην επίδοση.

Ο 2.5D αλγόριθμος και το PARALiA καταφέρνουν από αξιοσημειώτες μέχρι και καταπληκτικές επιδόσεις. Το μεγαλύτερο μειονέκτημα του κατανεμημένου PARALiA, η κατανομή δεδομένων σε διεργασίες, πρακτικά αγνοείται σε αυτό το σενάριο εκτέλεσης αφού δεν συμπεριλαμβάνεται ο χρόνος κατανομής και συλλογής δεδομένων στις μετρήσεις. Αν η απόσυνθεση αυτή ήταν λίγο πιο ρεαλιστική, θα αποτελούσε μια πολύ καλή λύση για εκτέλεση σε κατανεμημένα περιβάλλοντα.

Ο 2.5D αλγόριθμος του Cannon για GPUs έχει πάντα μέσες με καλές επιδόσεις, ειδικά σε μεγαλύτερες διαστάσεις πινάκων.

3.3.2 Overall Problems

Προχωράμε σε σύγκριση επίδοσεις για γενικές εκτελέσεις - όχι μόνο τετράγωνων προβλημάτων. Οι κατηγορίες εκτέλεσεις των gemm είναι οι παρακάτω, με τις πρώτες δύο να έιναι κυρίως I/O-bound εφαρμογές η τρίτη compute-bound.

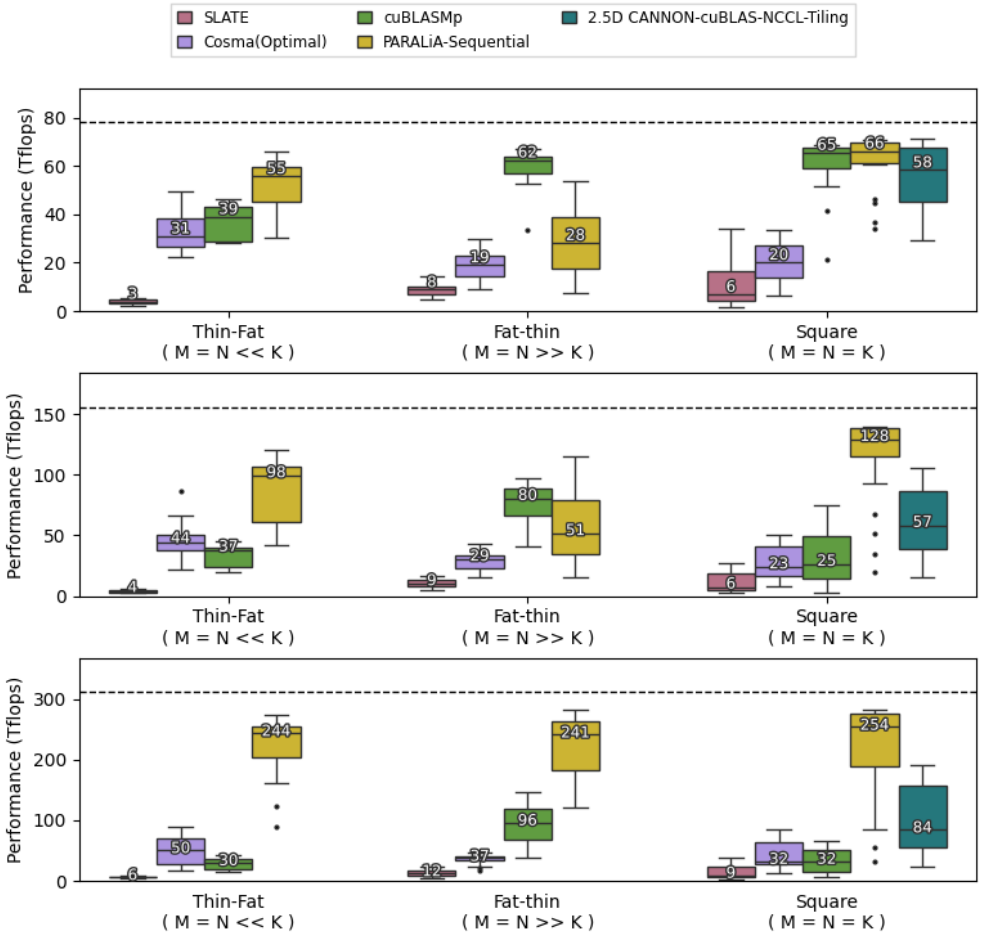
- **Thin-by-Fat**: $M = N << K$ με $M =$ "$thin$", $K =$ "fat". Δηλαδή, μία μεγάλη διάσταση.
- **Fat-by-Thin**: $M = N >> K$ με $M =$ "fat", $K =$ "$thin$". Δηλαδή, δύο μεγάλες διαστάσεις.
- **Square**: $M = N = K$. Δηλαδή, τρεις μεγάλες διαστάσεις.

Οι τετράγωνες εκτελέσεις είναι ίδιων διαστάσεων με τα προηγούμενα πειράματα. Για τα άλλα προβλήματα, το καθένα εκτελεί 21 σενάρια με τα fat-by-thin να ισχύει: $M_{fat} = N_{fat} = (16 \xrightarrow{\text{step=4}} 40) \cdot 2^{10}, K_{thin} = \frac{M_{fat}}{r}, r \in [4, 8, 16]$ και για τα thin-by-fat να ισχύει: $M_{thin} = N_{thin} = (5 \xrightarrow{\text{step=1}} 11) \cdot 2^{10}, K_{fat} = M_{thin} \cdot r, r \in [4, 8, 16]$.

Τα boxplot δείχνουν την μέση επίδοση των συστημάτων στα σενάρια που περιγράψαμε παραπάνω. Η αρχική υλοποίηση της 2.5D δεν υποστήριζε εκτέλεση σε μη-τετράγωνους πίνακες, και για αυτόν τον λόγο απουσιάζει από τις αντίστοιχες γραφικές παραστάσεις. Από τότε, η έλλειψη αυτή καλύφθηκε αλλά δεν υπήρχαν διαθέσιμοι πόροι στο Meluxina για την εκτέλεση benchmark.

Τα συμπεράσματα για τους τετράγωνους πίνακες είναι ίδια με αυτά της προηγούμενης ανάλυσης.

Τα **thin-by-fat** προβλήματα αναπαριστούν καταστάσεις όπου ο πίνακας C είναι ένας μικρός και τετράγωνος πίνακας και οι A, B είναι μακριοί πίνακες με πολλές στήλες και λίγες γραμμές. Τα προβλήματα αυτά είναι περισσότερο I/O-bound προβλήματα από τα υπόλοιπα και οι αποσυνθέσεις που βασίζονται με αρχική διάσπαση του C πίνακα, όπως αυτή των αλγορίθμων SUMMA, Cannon δεν καταφέρνουν σημαντικές επιδόσεις. Πρακτικά, παρατηρούμε ότι η απλοϊκή αποσύνθεση που χρησιμοποιεί το κατανεμημένο PARALiA ευνοεί την εκτέλεση τέτοιων προβλημάτων καθώς οι ανάγκες για read-only δεδομένα (δηλαδή οι πίνακες A, B) είναι ήδη καλυμμένες από την διαδικασία της αποσύνθεσης. Επίσης, εφόσον δεν "κόβουμε" την τρίτη διάσταση, μειώνεται σημαντικά το overhead πολλαπλών εκτελέσεων υπό-πινάκων και το τελικό reduction που θα χρειαστεί για ανανέωση του πίνακα C. Το cuBLASMp επιτυγχάνει αξιοσημείωτη επίδοση για τον τύπο προβλήματος που επιλύει, αλλά πάλι παρατηρούμε πρόβλημα στα θέματα κλιμάκωσης αφού η αύξηση κόμβων επιφέρει μείωση επίδοσης, κατά μέσο όρο. Το COSMA



Σχήμα 3.5: Εκτελέσεις GEMM - Αριθμός κόμβων: 1, 2, 4 (από πάνω προς τα κάτω)

είναι σχεδιασμένο συγκεκριμένα για τέτοιου είδους προβλήματα και δείχνει καλύτερες επιδόσεις συγκριτικά με τετράγωνα προβλήματα και μέτριες προοπτικές κλιμάκωσης. Το SLATE είναι το χειρότερο από όλα τα συστήματα, σε κάθε διαμμόρφωση διαστάσεων και πλεγμάτων.

Τα **fat-by-thin** προβλήματα είναι μια μέση κατάσταση μεταξύ I/O και compute-bound - θεωρούνται πιο εύκολα σε εκτέλεση από τα thin-by-fat. Αναπαριστούν καταστάσεις όπου ο πίνακας C είναι μεγάλος τετράγωνος πίνακες ενώ οι A, B είναι μακριοί πίνακες με πολλές γραμμές και λίγες στήλες. Το κατανεμημένο PARALiA δεν αποδίδει βέλτιστα στις πρώτες δύο διαμορφώσεις πλέγματος αφού ο ένας από τους 2 read-only πίνακες δεν σπάει καθόλου. Υπενθυμίζουμε ότι το πλέγμα που χρησιμοποιεί αυτή η υλοποίηση θεωρεί μια διεργασιά ανά κόμβο, άρα για εκτέλεση σε 2 κόμβους, αναγκαστικά θα "κοπεί" μόνο μία από τις δύο διαστάσεις, είτε η M είναι η N. Έτσι όμως, δημιουργούνται δύο υπό-προβλήματα τα οποία είναι επίσης fat-by-thin σε φύση και δυσκολεύουν την εκτέλεση. Στην τελευταία διαρρύθμιση όπου η αποσύνθεση κόβει και τους δύο πίνακες, οι επιδόσεις είναι πολύ καλύτερες, φτάνοντας πάλι κοντά στο θεωρητικό μέγιστο. Το cuBLASMp, παραδόξως, καταφέρνει αξιοσημειώτη επίδοση σε όλες τις εκτελέσεις και επιτυγχάνει για πρώτη φορά, έστω και μικρή, κλιμάκωση.

Το COSMA φαίνεται ότι αποδίδει μέτρια σε εφαρμογές που τείνουν προς compute-bound - φαινόμενο που δείχνει κάποια ατέλεια στο υπολογιστικό backend που χρησιμοποιεί.

Υπενθυμίζουμε ότι αρχικά σχεδιάστηκε για εκτέλεση σε CPUs, η υποστήριξη των GPUs προστέθηκε αργότερα. Για αυτό τον λόγο, δεν γίνεται εκτέλεση του συστήματος με τα δεδομένα τοποθετημένα εξ αρχής στις μνήμες των GPUs. Η μεταφορά από host προς device σίγουρα περιορίζει τις μέγιστες επιδόσεις του συστήματος. Όμως, η σχεδίασή του είχε σκοπό την εκτέλεση με πολύ μεγάλο αριθμό διεργασιών και μη-τετράγωνους πίνακες σε CPUs.

Τέλος, το SLATE, αφήνει μεγάλο περιθώριο επιδόσεων ανεκμετάλλευτο. Η σχεδιαστική απόφαση για αποδοτική εκτέλεση ταυτόχρονα σε CPU και GPU εμποδίζει τις δεύτερες να αξιοποιήσουν ένα μεγάλο ποσοστό των δυνατοτήτων τους. Η σχεδίαση γύρω από χρήση μικρών μεγεθών blocking ώστε να εκτελούνται batched gemm calls μεγαλώνει σημαντικά το συνολικό overhead υπολογισμού και επικοινωνίας.

3.3.3 Scalability

Τελευταία εξετάζουμε την κλιμακωσιμότητα των συστημάτων. Παρουσιάζουμε τα δύο σημαντικά είδη scaling, την ισχυρή και την ασθενή.

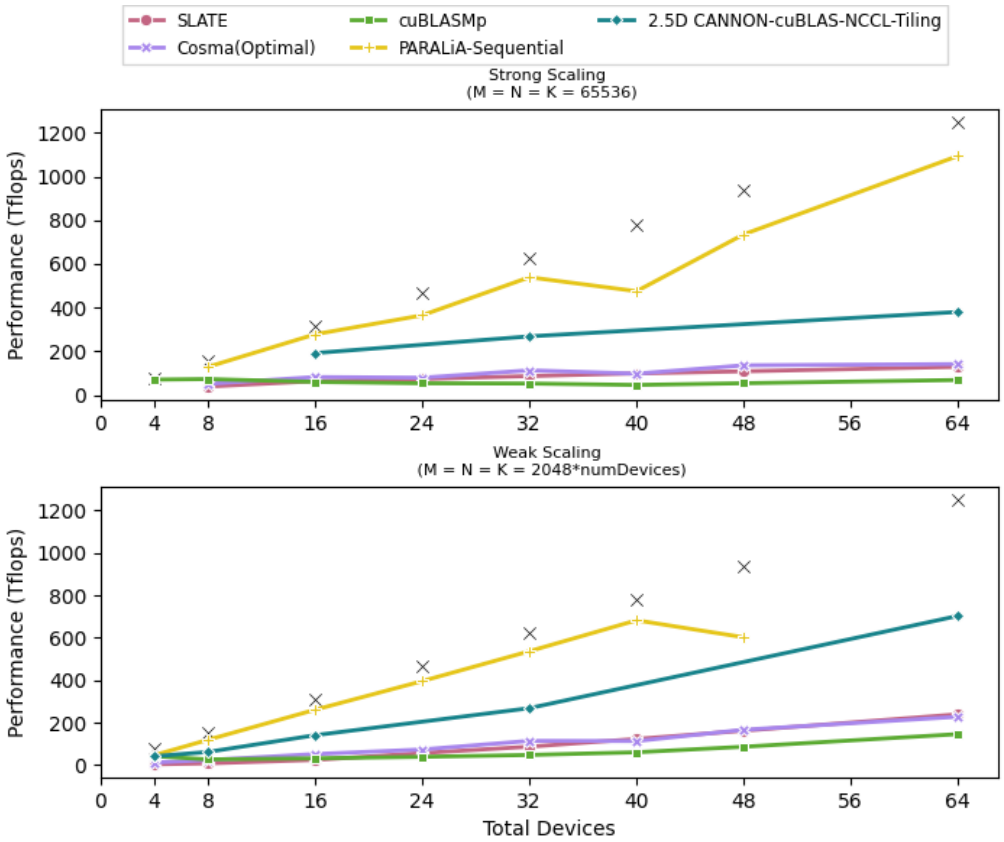
- **Strong Scaling**: Περιγράφει πως επηρεάζεται ο χρόνος εκτέλεσης συναρτήσει των αριθμό συσκευών που χρησιμοποιούνται σε εκτέλεση προβλήματος *σταθερού* μεγέθους
- **Weak Scaling**: Περιγράφει πως επηρεάζεται ο χρόνος εκτέλεσης συναρτήσει των αριθμό συσκευών που χρησιμοποιούνται σε εκτέλεση προβλήματος *σταθερού μεγέθους ανά συσκευή*

Το Strong Scaling ουσιαστικά αποτελεί την μαθηματική εξήγηση του νόμου του Amdahl και εξετάζει πόσο μεγάλο είναι το παραλληλοποιήσιμο τμήμα ενός αλγορίθμου. Στην συγκεκριμένη εφαρμογή, ο αλγόριθμος είναι ντροπιαστικά παράλληλος στα υπολογιστικά μέρη αλλά φράζεται εύκολα από τα επικοινωνιακά τμήματα. Υπενθυμίζουμε ότι σύμφωνα με τον νόμο του Amdahl, το speedup υπολογίζεται ως $S(s) = \frac{1}{(1-p)+\frac{p}{s}}$.

Το Weak Scaling είναι η εξήγηση του νόμου του Gustafson και εξετάζει πόσο καλά αποδίδει ένας αλγόριθμος όταν αυξάνονται οι συνολικές απαιτήσεις σε μνήμη, αλλά οι απαιτήσεις ανά διεργασία παραμένουν σταθερές. Και οι δύο είναι σημαντικές μετρικές για την εξαγωγή συμπερασμάτων αν και συνήθως εξετάζουμε compute-bound εφαρμογές χρησιμοποιώντας ως κύρια μετρική το strong scaling ενώ I/O-bound εφαρμογές με το weak scaling.

Τρέχουμε εκτελέσεις για αριθμό κόμβων $n = \{1, 2, 4, 6, 8, 10, 12, 16\}$. Για το Strong Scaling, επιλέγουμε τετράγωνο πίνακα διαστάσεων $M = N = K = 65536$ ενώ για το Strong Scaling επιλέγουμε τετράγωνο πίνακα μεταβλητής διάστασης συναρτήσει των αριθμό συσκευών που χρησιμοποιούνται. Κάθε κόμβος έχει 4 GPUs, επιλέγουμε διαστάσεις προβλήματος $M = N = K = 2048 * numDevices$.

Και στα δύο είδη Scaling, το κατανεμημένο PARALiA αποδίδει πολύ κοντά στο θεωρητικό μέγιστο σχεδόν σε όλες τις διαρρυθμίσεις κόμβων. Το *ελάχιστο* ποσοστό που επιτυγχάνει είναι 50% του συνολικού συστήματος, καθιστώντας το ένα πολύ αποδοτικό σύστημα. Η επιλογή να βασιστεί σε ένα ήδη υπερ-αποδοτικό σύστημα εκτέλεσης πολλαπλασιασμού πινάκων σε multi-gpu κόμβους απέδωσε στην εκτέλεση προβλημάτων μικρών και μεσαίων διαστάσεων. Δυστυχώς, το γράφημα του weak scaling δείχνει και το σημαντικότερο μειονέκτημα του. Η γραφική παράσταση σταματάει μόλις στους 12 κόμβους, χρησιμοποιώντας έναν αρκετά δίκαιο



Σχήμα 3.6: Strong & Weak Scaling - Το 'X' αναπαριστά την θεωρητική μέγιστη επίδοση

μέγεθος διαστάσεων ανά συσκευή, δείχνοντας ότι η υλοποίηση αυτή δεν είναι ικανή για εκτέλεση μεγάλων προβλημάτων, χαρακτηριστικό απαραίτητο για βιβλιοθήκες gemm που εκτελούνται σε συστοιχίες.

Τα cuBLASMp, COSMA και SLATE βρίσκονται σε πολύ παρόμοια κατάσταση μεταξύ τους. Το συμπέρασμα από τα προηγούμενα πειράματα για το cuBLASMp πέρι κακής κλιμάκωσης επιβεβαιώνεται με αυτές τις γραφικές παραστάσεις. Μπορούμε να εξάγουμε παρόμοια συμπεράσματα για το SLATE και το COSMA, δικαιολογώντας και τα δύο βάσει των συμπερασμάτων των προηγούμενων αναλύσεων.

Τέλος, η έκδοση μας που υλοποιεί 2.5D αλγόριθμο Cannon για εκτέλεση σε GPU δείχνει αρκετά ικανοποιητική ροή στο διάγραμμα weak scaling ενώ είναι τουλάχιστον 2 φορές καλύτερο από το COSMA, την τρίτη καλύτερη υλοποίηση, στο διάγραμμα strong scaling. Δεν μπορούμε να εξάγουμε περαιτέρω συμπεράσματα καθώς χρειάζεται να γίνουν εκτελέσεις με σημαντικά μεγαλύτερο αριθμό διεργασιών και σε πολύ μεγαλύτερες διαστάσεις πινάκων.

Chapter 4

Σύνοψη

4.1 Συμπεράσματα

Η σχεδίαση αλγορίθμων που εκτελούνται σε συστήματα μεγάλης κλίμακας είναι μια δύσκολη και χρονοβόρα διαδικασία. Έχοντας μελετήσει τα σημαντικότερα και καινοτόμα ερευνητικά έργα που έχουν πραγματοποιηθεί στον τομέα του πολλαπλασιασμού πινάκων σε συστοιχίες υπολογιστών, βασιστήκαμε σε έναν πρόσφατο αλγόριθμο και τον τροποποιήσαμε κατάλληλα ώστε να εκτελείται πιο αποδοτικά σε GPUs.

Ένα από τα κυριότερα συμπεράσματα που μπορούμε να εξάγουμε είναι ότι η σχεδίαση συστημάτων για εκτέλεση σε επεξεργαστές γραφικών είναι αρκετά διαφορετική διαδικασία από αυτή για σχεδίαση κλασσικών αλγορίθμων CPU. Οι αλγόριθμοι που μελετήσαμε βασίζουν την εκτέλεσή τους σε πολύ μεγάλο αριθμό διεργασιών, όπου η επικοινωνία μεταξύ κόμβων είναι ο σημαντικότερος περιοριστικός παράγοντας. Όμως, εφόσον τα συστήματα αυτά προορίζονται για εκτέλεση σε CPU, αγνοούν κάποια σημαντικά χαρακτηριστικά όπως τις ταχύτητες των διαύλων επικοινωνίας μεταξύ κόμβων/συσκευών.

Εν τέλει, σχεδιάσαμε και υλοποιήσαμε δύο διαφορετικές λύσεις για το ίδιο πρόβλημα. Η πρώτη αποδείκτηκε πολύ αποδοτική, αφού βασίστηκε σε ένα σύγχρονο και εκλεπτυσμένο έργο όμως είχε πολύ σημαντικούς περιορισμούς που την καθιστά ανεπαρκή για χρήση σε συστήματα συστοιχιών. Η δεύτερη ξεκίνησε από έναν αποδοτικό αλγόριθμο, δέχτηκε αρκετές βελτιώσεις και τροποποιήσεις ώστε να εκτελείται αποδεκτά σε επεξεργαστές γραφικών και κατάφερε αποδεκτές επιδόσεις με αποτέλεσμα να συγκρίνεται άμεσα με σύνθετες βιβλιοθήκες και αξιοσημείωτα ερευνητικά έργα.

Τα πορίσματα που μπορούμε να εξάγουμε είναι αρκετά λακωνικά. Η ανάλυση και δημιουργία βιβλιοθηκών που εκτελούνται σε συστήματα μεγάλης κλίμακας απαιτεί και τις κατάλληλες υποδομές, οι οποίες στην δική μας περίπτωση ήταν χρονικά περιορισμένες. Ελπίζουμε ότι με περαιτέρω διαθεσιμότητα τέτοιων υποδομών, μπορούμε να βελτιώσουμε το ήδη υπάρχον έργο σε σημαντικό βαθμό.

4.2 Μελλονιτική Έρευνα

4.2.1 Εκτέλεση πειραμάτων μεγαλύτερης κλίμακας

Αρχικά, είναι πολύ σημαντική η εκτέλεση πειραμάτων για διαστάσεις και αριθμό διεργασιών πολύ μεγαλύτερο από αυτόν που δοκιμάστηκαν στην συγκεκριμένη εργασία, όχι μόνο για τα συστήματα που υλοποιήσαμε εμείς αλλά και για αυτά με τα οποία συγκρινόμαστε. Η εφαρμογή των βιβλιοθηκών PBLAS γίνεται σε συστήματα με πολύ περισσότερους πόρους και δεν αρκεί η εξαγωγή συμπερασμάτων από πειράματα που μέγιστος αριθμός κόμβων που χρησιμοποιήθηκαν ήταν μόλις 16.

Η περιορισμένη φύση των πειραμάτων ευθύνεται, σε έναν βαθμό, στα χρονικά όρια χρήσης του υπερυπολογιστή Meluxina.

4.2.2 Υποστήριξη 2D Block Cyclic κατανομής

Το ScaLAPACK, αν και αρχαίο για τα μοντέρνα δεδομένα έργο, έχει υπάρξει σημαντικό ορόσημο στον τομέα του HPC. Για τον λόγο αυτό, κύρια προτεραιότητα όλων των βιβλιοθηκών που εξετάσαμε ήταν η υποστήριξη δεδομένων που έχουν κατανεμηθεί με 2D Block Cyclic τρόπο και η υποστήριξη εκτέλεσης συναρτήσεων PBLAS από GPUs.

Η σημαντικότερη αιτία για την οποία γίνεται αυτό είναι η πληθώρα λογισμικών και εφαρμογών που χρησιμοποιούν το ScaLAPACK ως βιβλιοθήκη εκτέλεσης μαθηματικών πράξεων. Η ανάγκη για συνεχή υποστήριξη αυτών των εφαρμογών καθιστά την εκτέλεση του gemm βάσει του προτύπου που ορίζει το PBLAS αναγκαία συνθήκη.

4.2.3 Μοντελοποίηση Tiling Size

Παρατηρήσαμε συχνά την έννοια του tiling σε αυτή την εργασία και τον άμεσο ρόλο που παίζει στους χρόνους υπολογισμού, επικοινωνίας και στην επικάλυψη αυτών. Ειδικά για την 2D Block Cyclic κατανομή όπου η εφαρμογή του tiling γίνεται σε πολλά επίπεδα, η επιλογή των σωστών μεγεθών χρησιμοποιώντας μοντελοποίηση θα ήταν μια πολύ βοηθητική προσθήκη στο σύστημα εκτέλεσης.

Δείξαμε ότι συγκεκριμένα για το SLATE και το cuBLASMp ότι η λάθος επιλογή μεγέθους μπορεί να μειώσει δραστικά την επίδοση υπολογισμού και να αυξήσει το overhead επικοινωνίας. Το PARALiA, επιλέγοντας μέγεθος tiling βάσει μοντέλου με κριτήρια την επικοινωνία, τον παραλληλισμό του προβλήματος και το μέγεθος των εισόδων, πετυχαίνει αξιοσημείωτες επιδόσεις σε όλα τα προβλήματα που δοκιμάσαμε.

Περαιτέρω, η ενσωμάτωση παρόμοιου συστήματος, λαμβάνοντας υπόψιν και τις διασυνδέσεις μεταξύ κόμβων και όχι μόνο τις εσωτερικές μεταξύ συσκευών, θα αποτελούσε μια καλή ευκαιρία για μελέτη επικοινωνιακής μοντελοποίησης σε συστήματα μεγάλης κλίμακας.

4.2.4 Ενσωμάτωση τεχνολογίας Multi-Instance GPU - Αύξηση Διεργασιών

Σε μερικά πειράματα παρατηρήσαμε ότι η κατανομή των διεργασιών σε πλέγματα επίσης επηρεάζει σημαντικά την επίδοση των εφαρμογών. Οι αλγόριθμοι πλεγμάτων, όπως αυτός του Cannon, ο PUMMA και ο SUMMA, δείχνουν ότι τα βέλτιστα πλέγματα είναι αυτά που παίρνουν

το σχήμα τετραγώνου - όσο πιο ορθογώνιο το σχήμα τόσο και δυσκολότερη η αποσύνθεση του προβλήματος. Η ρύθμιση των αριθμών διεργασιών σε κόμβους με CPU είναι αρκετά εύκολη υπόθεση αφού ο προγραμματιστής μπορεί να επιλέξει κάθε διεργασία να ορίζεται από ένα νήμα, πυρήνα, επεξεργαστή ή κόμβο. Έτσι, η δημιουργία τετράγωνων πλεγμάτων είναι πιο βατή διαδικασία.

Αντιθέτως, για τις GPU, τέτοιος έλεγχος είναι αρκετά δυσκολότερος. Μία άξια προσπάθεια θα ήταν η χρήση του συστήματος GPU partitioning της NVIDIA για να διαχωρίσουμε μια GPU σε περισσότερες ”λογικές” συσκευές. Συγκεκριμένα, το σύστημα αυτό ονομάζεται Multi-Instace GPU [59] και συναντίεται συχνά σε εφαρμογές IoT όπου εκτελούνται πολλαπλά real-time μοντέλα AI σε κοινή συσκευή και χρειάζονται διαφορετική διαχείριση ή σε συστήματα παροχής υπηρεσιών όπως το AWS της Amazon και το Azure της Microsoft, όπου σκοπός είναι ο διαχωρισμός μιας συσκευής σε πολλαπλές εικόνες και ο περαιτέρω διαμοιρασμός αυτών σε χρήστες. Η GPU μπορεί να διαχωριστεί λογικά σε επίπεδο SM, νημάτων, μνήμης κ.λ.π, έτσι ώστε να ταιριάζει στις απαιτήσεις του χρήστη και να επιτυγχάνεται μεγαλύτερο ποσοστό αξιοποίησης από την συσκευή.

Μια παρόμοια τεχνική μπορεί να χρησιμοποιηθεί ώστε να διαμοιραστεί ισότιμα μια GPU και κάθε τμήμα της να θεωρείται και μια διεργασία, δίνοντας έτσι παραπάνω έλεγχο στο σχήμα που μπορεί να δομηθεί το πλέγμα διεργασιών.

Καταλογος γραφικων παραστασεων

List of Algorithms

Βιβλιογραφια

[1] J. Krüger and R. Westermann, "Linear algebra operators for gpu implementation of numerical algorithms," *ACM Trans. Graph.*, vol. 22, p. 908–916, jul 2003.

[2] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder, "Sparse matrix solvers on the gpu: conjugate gradients and multigrid," *ACM Trans. Graph.*, vol. 22, p. 917–924, jul 2003.

[3] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in Science and Engineering*, vol. 12, no. 3, pp. 66–73, 2010.

[4] NVIDIA, "GPU Performance Background NVIDIA Docs." `https://docs.nvidia.com/deeplearning/performance/pdf/GPU-Performance-Background-User-Guide.pdf`, 2020. Accessed: 27-08-2024.

[5] M. Flynn, "Very high-speed computing systems," *Proceedings of the IEEE*, vol. 54, no. 12, pp. 1901–1909, 1966.

[6] M. Ilg, J. Rogers, and M. Costello, "Projectile monte-carlo trajectory analysis using a graphics processing unit," 08 2011.

[7] NVIDIA, "CUDA C++ Programming Guide." `https://docs.nvidia.com/cuda/archive/11.2.0/pdf/CUDA_C_Programming_Guide.pdf`, 2021. Accessed: 27-08-2024.

[8] NVIDIA, "NVIDIA A100 Tensor Core GPU Architecture." `https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf`, 2020. Accessed: 30-08-2024.

[9] L. Mao, "Lei Mao CUDA Stream Github Post." `https://leimao.github.io/blog/CUDA-Stream/`, 2020. Accessed: 30-08-2024.

[10] H. Meuer, "The top500 project. looking back over 15 years of supercomputing experience," *Praxis der Informationsverarbeitung und Kommunikation*, vol. 31, pp. 122–132, 10 2008.

[11] J. Dongarra, P. Luszczek, and A. Petitet, "The linpack benchmark: past, present and future," *Concurrency and Computation: Practice and Experience*, vol. 15, pp. 803–820, 08 2003.

[12] Top500, "Top500 June 2024 List." https://top500.org/lists/top500/list/2024/06/, 06 2024. Accessed: 01-09-2024.

[13] H. Khaleghzadeh, R. R. Manumachu, and A. Lastovetsky, "A novel data-partitioning algorithm for performance optimization of data-parallel applications on heterogeneous hpc platforms," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 10, pp. 2176–2190, 2018.

[14] LuxProvide, "Meluxina System Overview Page." https://docs.lxp.lu/system/overview/, 2021. Accessed: 01-09-2024.

[15] J. Dongarra, "An updated set of basic linear algebra subprograms (blas)," *ACM Trans. Math. Softw.*, vol. 28, p. 135–151, jun 2002.

[16] Q. open source. Initial contributors : Wang, X. Zhang, Y. Zhang, and Q. Yi, "OpenBLAS - An optimized BLAS library, howpublished = www.openblas.net, year = 2009, note = Accessed: 20-07-2024."

[17] Intel, "Intel Math Kernel Library (Intel MKL)." https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html, 2003. Accessed: 2024-08-11.

[18] NVIDIA, "cuBLAS Developer Guide Page." developer.nvidia.com/cublas, 2007. Accessed: 24-07-2024.

[19] AMD, "rocBLAS Introduction Page." https://rocm.docs.amd.com/projects/rocBLAS/en/docs-5.0.1/intro.html„ 2017. Accessed: 20-07-2024.

[20] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic linear algebra subprograms for fortran usage," *ACM Trans. Math. Softw.*, vol. 5, p. 308–323, sep 1979.

[21] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson, "An extended set of fortran basic linear algebra subprograms," *ACM Trans. Math. Softw.*, vol. 14, p. 1–17, mar 1988.

[22] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff, "A set of level 3 basic linear algebra subprograms," *ACM Trans. Math. Softw.*, vol. 16, p. 1–17, mar 1990.

[23] E. Angerson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammarling, J. Demmel, C. Bischof, and D. Sorensen, "Lapack: A portable linear algebra library for high-performance computers," in *Supercomputing '90:Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, pp. 2–11, 1990.

[24] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, and R. C. Whaley, "A proposal for a set of parallel basic linear algebra subprograms," in *Applied Parallel Computing Computations in Physics, Chemistry and Engineering Science* (J. Dongarra, K. Madsen, and J. Waśniewski, eds.), (Berlin, Heidelberg), pp. 107–114, Springer Berlin Heidelberg, 1996.

[25] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, "Scalapack: A portable linear algebra library for distributed memory computers — design issues and performance," in *Applied Parallel Computing Computations in Physics, Chemistry and Engineering Science* (J. Dongarra, K. Madsen, and J. Waśniewski, eds.), (Berlin, Heidelberg), pp. 95–106, Springer Berlin Heidelberg, 1996.

[26] J. Dongarra and R. Whaley, "Lapack working note 94 a user's guide to the blacs v1.1," tech. rep., University of Tennessee, 1997.

[27] V. S. Sunderam, "Pvm: A framework for parallel distributed computing," *Concurrency: Practice and Experience*, vol. 2, no. 4, pp. 315–339, 1990.

[28] K. Asanović, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from berkeley," Tech. Rep. UCB/EECS-2006-183, Dec 2006.

[29] V. Strassen, "Gaussian elimination is not optimal.," *Numerische Mathematik*, vol. 13, pp. 354–356, 1969.

[30] V. V. Williams, Y. Xu, Z. Xu, and R. Zhou, *New Bounds for Matrix Multiplication: from Alpha to Omega*, pp. 3792–3835. 2024.

[31] C. Nugteren, "Tutorial: OpenCL SGEMM tuning for Kepler." `https://cnugteren.github.io/tutorial/pages/page4.html`, 2014. Accessed: 03-09-2024.

[32] NVIDIA, "cuBLASXt Developer Guide Page." `developer.nvidia.com/cublasxt`, 2013. Accessed: 02-09-2024.

[33] L. Wang, W. Wu, J. Xiao, and Y. Yang, "BLASX: A high performance level-3 BLAS library for heterogeneous multi-gpu computing," *CoRR*, vol. abs/1510.05041, 2015.

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

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

[36] P. Anastasiadis, N. Papadopoulou, G. Goumas, N. Koziris, D. Hoppe, and L. Zhong, "Paralia: A performance aware runtime for auto-tuning linear algebra on heterogeneous systems," *ACM Trans. Archit. Code Optim.*, vol. 20, dec 2023.

[37] L. E. Cannon, *A cellular computer to implement the kalman filter algorithm.* PhD thesis, -, USA, 1969. AAI7010025.

[38] K. Srikanth, "Cannon's algorithm for distributed matrix multiplication." `https://iq.opengenus.org/cannon-algorithm-distributed-matrix-multiplication/`, 2018. Accessed: 05-09-2024.

[39] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar, "A three-dimensional approach to parallel matrix multiplication," *IBM Journal of Research and Development*, vol. 39, no. 5, pp. 575–582, 1995.

[40] E. Solomonik and J. Demmel, "Communication-optimal parallel 2.5d matrix multiplication and lu factorization algorithms," in *Euro-Par 2011 Parallel Processing* (E. Jeannot, R. Namyst, and J. Roman, eds.), (Berlin, Heidelberg), pp. 90–109, Springer Berlin Heidelberg, 2011.

[41] J. Poulson, B. Marker, R. A. van de Geijn, J. R. Hammond, and N. A. Romero, "Elemental: A new framework for distributed memory dense matrix computations," *ACM Trans. Math. Softw.*, vol. 39, feb 2013.

[42] E. Solomonik, D. Matthews, J. R. Hammond, J. F. Stanton, and J. Demmel, "A massively parallel tensor contraction framework for coupled-cluster computations," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3176–3190, 2014. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[43] R. A. Van De Geijn and J. Watts, "Summa: scalable universal matrix multiplication algorithm," *Concurrency: Practice and Experience*, vol. 9, no. 4, pp. 255–274, 1997.

[44] J. Choi, D. W. Walker, and J. J. Dongarra, "Pumma: Parallel universal matrix multiplication algorithms on distributed memory concurrent computers," *Concurrency: Practice and Experience*, vol. 6, no. 7, pp. 543–570, 1994.

[45] E. Solomonik, "Cs 598: Communication cost analysis of algorithms lecture 3: communication avoiding algorithms for matrix multiplication." 2016.

[46] M. Gates, J. Kurzak, A. Charara, A. YarKhan, and J. Dongarra, "Slate: design of a modern distributed and accelerated linear algebra library," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, (New York, NY, USA), Association for Computing Machinery, 2019.

[47] NVIDIA, "cuBLASMp Developer Guide Page." `developer.nvidia.com/cublasmp-downloads`, 2023. Accessed: 24-07-2024.

[48] E. Aprà, E. J. Bylaska, W. A. de Jong, N. Govind, K. Kowalski, T. P. Straatsma, M. Valiev, H. J. J. van Dam, Y. Alexeev, J. Anchell, V. Anisimov, F. W. Aquino, R. Atta-Fynn, J. Autschbach, N. P. Bauman, J. C. Becca, D. E. Bernholdt, K. Bhaskaran-Nair, S. Bogatko, P. Borowski, J. Boschen, J. Brabec, A. Bruner, E. Cauët, Y. Chen, G. N. Chuev, C. J. Cramer, J. Daily, M. J. O. Deegan, T. H. Dunning, M. Dupuis, K. G. Dyall, G. I. Fann, S. A. Fischer, A. Fonari, H. Früchtl, L. Gagliardi, J. Garza, N. Gawande, S. Ghosh, K. Glaesemann, A. W. Götz, J. Hammond, V. Helms, E. D. Hermes, K. Hirao, S. Hirata, M. Jacquelin, L. Jensen, B. G. Johnson, H. Jónsson, R. A. Kendall, M. Klemm, R. Kobayashi, V. Konkov, S. Krishnamoorthy, M. Krishnan, Z. Lin, R. D. Lins, R. J. Littlefield, A. J. Logsdail, K. Lopata, W. Ma, A. V. Marenich, J. Martin del Campo, D. Mejia-Rodriguez, J. E. Moore, J. M. Mullin, T. Nakajima, D. R. Nascimento, J. A. Nichols, P. J. Nichols, J. Nieplocha, A. Otero-de-la Roza, B. Palmer, A. Panyala, T. Pirojsirikul, B. Peng, R. Peverati, J. Pittner, L. Pollack, R. M. Richard, P. Sadayappan, G. C. Schatz, W. A. Shelton, D. W. Silverstein, D. M. A. Smith, T. A. Soares, D. Song, M. Swart, H. L. Taylor, G. S. Thomas, V. Tipparaju, D. G. Truhlar, K. Tsemekhman, T. Van Voorhis, . Vázquez-Mayagoitia, P. Verma, O. Villa, A. Vishnu, K. D. Vogiatzis, D. Wang, J. H. Weare, M. J. Williamson, T. L. Windus,

K. Woliński, A. T. Wong, Q. Wu, C. Yang, Q. Yu, M. Zacharias, Z. Zhang, Y. Zhao, and R. J. Harrison, "Nwchem: Past, present, and future," *The Journal of Chemical Physics*, vol. 152, no. 18, p. 184102, 2020.

[49] T. D. Kühne, M. Iannuzzi, M. Del Ben, V. V. Rybkin, P. Seewald, F. Stein, T. Laino, R. Z. Khaliullin, O. Schütt, F. Schiffmann, D. Golze, J. Wilhelm, S. Chulkov, M. H. Bani-Hashemian, V. Weber, U. Borštnik, M. Taillefumier, A. S. Jakobovits, A. Lazzaro, H. Pabst, T. Müller, R. Schade, M. Guidon, S. Andermatt, N. Holmberg, G. K. Schenter, A. Hehn, A. Bussy, F. Belleflamme, G. Tabacchi, A. Glöß, M. Lass, I. Bethune, C. J. Mundy, C. Plessl, M. Watkins, J. VandeVondele, M. Krack, and J. Hutter, "CP2K: An electronic structure and molecular dynamics software package - Quickstep: Efficient and accurate electronic structure calculations," *The Journal of Chemical Physics*, vol. 152, p. 194103, 05 2020.

[50] H. Jeong, Y. Yang, V. Gupta, C. Engelmann, T. M. Low, V. Cadambe, K. Ramchandran, and P. Grover, "3d coded summa: Communication-efficient and robust parallel matrix multiplication," in *Euro-Par 2020: Parallel Processing* (M. Malawski and K. Rzadca, eds.), (Cham), pp. 392–407, Springer International Publishing, 2020.

[51] D. Mukunoki and T. Imamura, *Implementation and Performance Analysis of 2.5D-PDGEMM on the K Computer*, pp. 348–358. 03 2018.

[52] E. Solomonik and J. Demmel, "Matrix multiplication on multidimensional torus networks," in *High Performance Computing for Computational Science - VECPAR 2012* (M. Daydé, O. Marques, and K. Nakajima, eds.), (Berlin, Heidelberg), pp. 201–215, Springer Berlin Heidelberg, 2013.

[53] J. Demmel, D. Eliahu, A. Fox, S. Kamil, B. Lipshitz, O. Schwartz, and O. Spillinger, "Communication-optimal parallel recursive rectangular matrix multiplication," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pp. 261–272, 2013.

[54] D. Eliahu, "CARMA GitHub Repository." `https://github.com/dose78/CARMA`, 2012. Accessed: 05-09-2024.

[55] G. Kwasniewski, M. Kabić, M. Besta, J. VandeVondele, R. Solcà, and T. Hoefler, "Red-blue pebbling revisited: Near optimal parallel matrix-matrix multiplication," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–22, 2019.

[56] H. Jia-Wei and H. T. Kung, "I/o complexity: The red-blue pebble game," in *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing*, STOC '81, (New York, NY, USA), p. 326–333, Association for Computing Machinery, 1981.

[57] M. Kabic, "Tiled-MM GitHub Page." `https://github.com/eth-cscs/Tiled-MM`, 2019. Accessed: 03-09-2024.

[58] M. Kabić, S. Pintarelli, A. Kozhevnikov, and J. VandeVondele, "Costa: Communication-optimal shuffle and transpose algorithm with process relabeling," in *High Performance Computing* (B. L. Chamberlain, A.-L. Varbanescu, H. Ltaief, and P. Luszczek, eds.), (Cham), pp. 217–236, Springer International Publishing, 2021.

[59] NVIDIA, "NVIDIA Multi Instance GPU Information Page." https://www.nvidia.com/en-eu/technologies/multi-instance-gpu/, 2020. Accessed: 10-10-2024.