ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

Σχολή Τεχνολογίας

Τμήμα Ψηφιακών Συστημάτων

Προσομοίωση Κβαντικών Υπολογιστών με Πολύ-πύρινο Προγραμματισμό

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

ΦΙΛΟΚΩΣΤΑ ΙΩΑΝΝΑ (ΑΜ: 3120004)

Επιβλέπων: ***Σάββας Ηλίας,*** ***βαθμίδα***

ΛΑΡΙΣΑ, Μάρτιος 2025

UNIVERSITY OF THESSALY

School of Technology

Digital Systems Department

Simulation of Quantum Computers using multicore programming

BACHELOR THESIS

FILOKOSTA IOANNA (RN: 3120004)

Supervisor: ***Savvas Hlias, position***

LARISSA, Μάρτιος 2025

Υπεύθυνη Δήλωση περί Ακαδημαϊκής Δεοντολογίας και Πνευματικών Δικαιωμάτων

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

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

Ο/Η Δηλών/ούσα

(Υπογραφή)

Φιλοκώστα Ιωάννα

Ημερομηνία

Εγκρίνεται από την Επιτροπή Εξέτασης:

Επιβλέπων	**Ονοματεπώνυμο Επιβλέποντα** Βαθμίδα/ιδιότητα επιβλέποντα, Τμήμα Ψηφιακών Συστημάτων, Πανεπιστήμιο Θεσσαλίας
Μέλος	**Ονοματεπώνυμο Μέλους 1** Βαθμίδα/ιδιότητα μέλους 1, Τμήμα/Ίδρυμα μέλους 1
Μέλος	**Ονοματεπώνυμο Μέλους 2** Βαθμίδα/ιδιότητα μέλους 2, Τμήμα/Ίδρυμα μέλους 2

Ημερομηνία έγκρισης: dd-mm-yyyy

Περίληψη

Η παρούσα πτυχιακή εργασία εξετάζει την προσομοίωση κβαντικών συστημάτων μέσω της χρήσης της βιβλιοθήκης Numba. Βασικός στόχος ήταν δημιουργία κβαντικών προσομοιώσεων, εκμεταλλευόμενοι την παράλληλη επεξεργασία των σύγχρονων καρτών γραφικών.

Για την επίτευξη αυτού του στόχου, αρχικά παρουσιάστηκαν οι θεμελιώδεις αρχές του κβαντικού υπολογισμού, συμπεριλαμβανομένων των κβαντικών πυλών και της αναπαράστασης των qubits. Στη συνέχεια, υλοποιήθηκαν προσομοιώσεις βασικών κβαντικών αλγορίθμων, εφαρμόζοντας τη Numba για την επιτάχυνση των υπολογισμών σε GPU.

Τα αποτελέσματα έδειξαν ότι η χρήση GPU καθιστά δυνατή την παραλληλοποίηση των κβαντικών αλγορίθμων, ιδιαίτερα σε περιπτώσεις που απαιτούν εντατικούς υπολογισμούς. Ωστόσο, εντοπίστηκαν περιορισμοί, όπως η αυξημένη κατανάλωση μνήμης και η πολυπλοκότητα στη βελτιστοποίηση των CUDA kernels. Παρά τις προκλήσεις αυτές, η εργασία ανέδειξε τη δυναμική της GPU επιτάχυνσης στην κβαντική προσομοίωση και έθεσε τις βάσεις για περαιτέρω έρευνα στον τομέα.

Η εργασία απευθύνεται σε ερευνητές και προγραμματιστές που ενδιαφέρονται για τη βελτίωση των κβαντικών προσομοιώσεων μέσω σύγχρονων τεχνικών επιτάχυνσης, συμβάλλοντας στην κατανόηση και την εξέλιξη του κβαντικού υπολογισμού.

Abstract

This thesis explores the simulation of quantum systems using the Numba library. The primary objective was the creation of quantum pre-simulations, leveraging the parallel processing capabilities of modern graphics cards (GPUs).

To achieve this goal, the fundamental principles of quantum computing were first introduced, including quantum gates and qubit representation. Subsequently, simulations of basic quantum algorithms were implemented, utilizing Numba to accelerate computations on the GPU.

The results demonstrated that the use of GPUs enables the parallelization of quantum algorithms, particularly in cases requiring intensive computations. However, certain limitations were identified, such as increased memory consumption and the complexity of optimizing CUDA kernels. Despite these challenges, the study highlighted the potential of GPU acceleration in quantum simulation and established a foundation for further research in the field.

This work is intended for researchers and developers interested in enhancing quantum simulations through modern acceleration techniques, contributing to the understanding and advancement of quantum computing

Ευχαριστίες

Θα ήθελα να εκφράσω την ειλικρινή μου ευγνωμοσύνη στον επιβλέποντα καθηγητή μου, Σάββα Ηλία, για την εμπιστοσύνη και την υπομονή. Χωρίς αυτόν, δεν θα είχα την ευκαιρία να εξερευνήσω έναν τομέα με τεράστιες προοπτικές και δυνατότητες για το μέλλον, αυτόν της κβαντικής υπολογιστικής.

Επίσης, ευχαριστώ θερμά την οικογένειά μου, η οποία στάθηκε δίπλα μου με αμέριστη αγάπη και στήριξη. Τέλος, ένα μεγάλο ευχαριστώ στους φίλους μου για την ενθάρρυνση, την κατανόηση και τις στιγμές χαλάρωσης που μοιραστήκαμε. Κάποιοι από μακριά και κάποιοι από κοντά με στήριξαν και με παρότρυναν.

Σας ευχαριστώ!

Φιλοκώστα Ιωάννα

Λάρισα 2025

Περιεχόμενα

1 Εισαγωγή

Η πτυχιακή εργασία αυτή πραγματεύεται την προσομοίωση κβαντικών κυκλωμάτων χρησιμοποιώντας τις δυνατότητες των μονάδων επεξεργασίας γραφικών (GPU) μέσω της βιβλιοθήκης Numba. Ο κβαντικός υπολογισμός αναπτύσσεται ταχύτατα ως ένας από τους πιο σημαντικούς τομείς της υπολογιστικής επιστήμης, με δυνατότητες να φέρει επανάσταση σε πολλούς κλάδους, όπως η κρυπτογραφία, η χημεία, η βελτιστοποίηση και η τεχνητή νοημοσύνη. Παρόλο που οι κβαντικοί υπολογιστές έχουν τη δυνατότητα να επιλύσουν προβλήματα που είναι αδύνατον ή εξαιρετικά δύσκολα για τους κλασικούς υπολογιστές, η ανάπτυξή τους και η πρακτική τους χρήση είναι ακόμα σε πρώιμο στάδιο. Ωστόσο, η προσομοίωση κβαντικών αλγορίθμων σε κλασικούς υπολογιστές είναι ένας τομέας που επιτρέπει την αναβάθμιση και τη βελτίωση των κβαντικών υπολογισμών, καθώς παρέχει τη δυνατότητα να δοκιμάσουμε και να εξερευνήσουμε κβαντικά κυκλώματα και αλγόριθμους χωρίς την ανάγκη άμεσης επένδυσης σε ακριβό κβαντικό υλικό.

Η τεχνολογία GPGPU (General-Purpose Computing on Graphics Processing Units), η οποία εκμεταλλεύεται την παράλληλη επεξεργαστική ισχύ των μονάδων επεξεργασίας γραφικών (GPU), καθίσταται ιδιαίτερα σημαντική στην επιτάχυνση της προσομοίωσης κβαντικών αλγορίθμων. Οι GPUs, με την αρχιτεκτονική τους που είναι βελτιστοποιημένη για την εκτέλεση παράλληλων υπολογισμών, επιτρέπουν την ταχύτερη εκτέλεση υπολογιστικά απαιτητικών εφαρμογών, όπως οι κβαντικοί αλγόριθμοι, όπου οι υπολογισμοί αυξάνονται εκθετικά με τον αριθμό των qubits. Η Numba, μέσω του εργαλείου CUDA, επιτρέπει την αποτελεσματική χρήση αυτής της τεχνολογίας για την προσομοίωση κβαντικών αλγορίθμων σε κλασικούς υπολογιστές, ενισχύοντας την αποδοτικότητά τους.

Η πτυχιακή αυτή διερευνά το πώς οι συναρτήσεις της Numba μπορούν να χρησιμοποιηθούν για την προσομοίωση κβαντικών κυκλωμάτων και αλγορίθμων, με ιδιαίτερη έμφαση στην εφαρμογή κβαντικών πυλών, τη δημιουργία κβαντικών καταχωρητών και την προσομοίωση κβαντικών καταστάσεων σε διάφορους τύπους κβαντικών υπολογιστικών συστημάτων. Αν και το κβαντικό υλικό είναι ακόμα σε πρώιμο στάδιο, η εξομοίωση και η ανάπτυξη αλγορίθμων σε κλασικές πλατφόρμες καθίστανται κρίσιμες για τη μελέτη και τη βελτίωση των κβαντικών αλγορίθμων.

Η εργασία αυτή οργανώνεται σε τρία κύρια κεφάλαια:

Στο πρώτο κεφάλαιο θα αναλύσουμε βασικές έννοιες της κβαντικής υπολογιστικής. Μετά από μια σύντομη ιστορική αναδρομή θα επικεντρωθούμε στο τι είναι ένα qubit και σε δομικά στοιχεία όπως οι καταχωρητές και οι βασικές πύλες (Hadamard, Pauli, CNOT). Θα εξετάσουμε τα παραπάνω τόσο μαθηματικά όσο και οπτικά με την βοήθεια της σφαίρας Bloch. Τέλος, με την βοήθεια της γλώσσας Python και της βιβλιοθήκης qiskit θα υλοποιήσουμε κυκλώματα και θα παρατηρήσουμε την συμπεριφορά τους.

Στο δεύτερο κεφάλαιο θα αναφερθούμε αρχικά στην GPU. Τι είναι αυτό που την κάνει κατάλληλη για παράλληλους υπολογισμούς και γιατί ξεχωρίζει από την CPU; Στην συνέχεια θα αναφερθούμε στην CUDA και πως αυτή εκμεταλλεύεται την δύναμη της GPU για την επιτάχυνση διαδικασιών υπολογισμού κάνοντας χρήση πυρήνων (kernels). Τέλος, θα δώσουμε παραδείγματα κώδικα σε γλώσσα Python με χρήση Numba. Η Numba είναι ο συνδετικός κρίκος μεταξύ CUDA και Python χωρίς την ανάγκη εγγραφής πολύπλοκου κώδικα.

Το κεφάλαιο εξετάζει τις βασικές αρχές αυτών των τεχνολογιών και παρέχει παραδείγματα εφαρμογών, όπως η διαχείριση μεγάλου όγκου δεδομένων με τεχνικές όπως οι Grid Stride Loops, η επίλυση συνθηκών ανταγωνισμού μέσω ατομικών διεργασιών, και η αποτελεσματική χρήση της μνήμης. Παράλληλα, αναλύει τις βέλτιστες πρακτικές για την αποφυγή συγκρούσεων στην κοινή μνήμη της GPU και τη χρήση κοινής μνήμης (shared memory) για βελτιστοποίηση της απόδοσης.

Για το τρίτο κεφάλαιο της πτυχιακής η θεματολογία επικεντρώνεται στην αξιοποίηση της τεχνολογίας για την προσομοίωση κβαντικών υπολογισμών. Οι GPUs, με την ικανότητά τους να εκτελούν πολλές παράλληλες εργασίες, είναι ιδανικές για την προσομοίωση κβαντικών κυκλωμάτων, που απαιτούν υπολογιστική ισχύ λόγω της εκθετικής αύξησης της πολυπλοκότητας με τον αριθμό των qubits. Θα παρέχουμε συναρτήσεις που υλοποιούν τις πύλες και όλους τους υπολογισμούς που πραγματοποιούνται κατά την εφαρμογή του.

2 Κβαντικός Υπολογισμός

Η κβαντική υπολογιστική είναι από τις πιο σύγχρονες τεχνολογίες και χρησιμοποιεί αρχές της κβαντικής μηχανικής για την επεξεργασία πληροφοριών με τρόπους που δεν μπορεί ένας κλασικός υπολογιστής. Σε αντίθεση με τον κλασσικό υπολογιστή, ο οποίος βασίζεται σε τρανζίστορ, ο κβαντικός χειρίζεται ατομικά σωματίδια όπως ηλεκτρόνια και πρωτόνια για να εκτελέσει υπολογισμούς επιτρέποντάς του να λύνει πολύπλοκα προβλήματα πιο αποτελεσματικά .

Γενικά είναι ένας διεπιστημονικός τομέας που συνδυάζει τη φυσική, την επιστήμη των υπολογιστών, τα μαθηματικά και τη μηχανική. Η ανάπτυξή του θα μπορούσε να οδηγήσει σε ανακαλύψεις σε πολλούς τομείς, ωστόσο παραμένουν κάποια σημαντικά τεχνικά εμπόδια τα οποία πολλές εταιρίες προσπαθούν να υπερνικήσουν.

2.1 Ιστορική Αναδρομή

Η κβαντική υπολογιστική, έχει εξελιχθεί σημαντικά από τις θεωρητικές της απαρχές στις αρχές της δεκαετίας του 1980. Το ταξίδι ξεκίνησε με θεωρητικές εξερευνήσεις και έχει προχωρήσει σε πρακτικές εφαρμογές. Κινητήριος δύναμη είναι η υπόσχεση επίλυσης προβλημάτων που είναι ανέφικτα για τους κλασικούς υπολογιστές.

Θεωρητικά Θεμέλια (1980ς)

Οι ρίζες της κβαντικής υπολογιστικής μπορούν να ανιχνευθούν στον Paul Benioff το 1980, ο οποίος πρώτος πρότεινε την ιδέα ενός κβαντικού μηχανικού μοντέλου υπολογισμού βασισμένου στη Μηχανή Turing. Ο Benioff πρότεινε τη χρήση των σπιν των στοιχειωδών σωματιδίων για την αναπαράσταση δυαδικών ψηφίων, δείχνοντας ότι μπορούν να κατασκευαστούν κυκλώματα ατομικής κλίμακας χωρίς κατανάλωση ενέργειας. Ωστόσο, το μοντέλο του ήταν ουσιαστικά ισοδύναμο με τις κλασικές Μηχανές Turing και δεν προσέφερε υπολογιστικά πλεονεκτήματα.

Το 1982, ο Richard Feynman, βραβευμένος με Νόμπελ Φυσικής, εισήγαγε την ιδέα ενός κβαντικού υπολογιστή ικανού να προσομοιώνει κβαντικά μηχανικά φαινόμενα. Ο Feynman υποστήριξε ότι οι κλασικοί υπολογιστές ήταν ανεπαρκείς για την αποδοτική

προσομοίωση κβαντικών συστημάτων και οραματίστηκε έναν νέο τύπο υπολογιστή που λειτουργεί σύμφωνα με την κβαντική μηχανική. Η εργασία του έθεσε τα θεμέλια για την ιδέα ότι οι κβαντικοί υπολογιστές θα μπορούσαν να ξεπεράσουν τους κλασικούς σε συγκεκριμένες εργασίες.

Το 1985, ο David Deutsch τυποποίησε την έννοια του κβαντικού υπολογιστή προτείνοντας μια κβαντική Μηχανή Turing, η οποία γενίκευε την κλασική Μηχανή Turing για να συμπεριλάβει κβαντικές αρχές. Το μοντέλο του Deutsch, γνωστό ως ο καθολικός κβαντικός υπολογιστής, ήταν υπολογιστικά ισοδύναμο με μια κβαντική Μηχανή Turing και σημάδεψε το πρώτο συγκεκριμένο βήμα προς ένα θεωρητικό πλαίσιο για την κβαντική υπολογιστική. Εισήγαγε επίσης την έννοια των κβαντικών πυλών το 1988, οι οποίες είναι θεμελιώδεις για τα μοντέλα κβαντικών κυκλωμάτων.

Πρόοδος στην δεκαετία του 1990

Η δεκαετία του 1990 υπήρξε σημαντική πρόοδο τόσο στη θεωρία όσο και στην πράξη. Το 1993, οι Ethan Bernstein και Umesh Vazirani γενίκευσαν την κβαντική Μηχανή Turing του Deutsch και έθεσαν τα θεμέλια της θεωρίας κβαντικής πολυπλοκότητας, αποδεικνύοντας ότι οι κβαντικές Μηχανές Turing έχουν υπολογιστική ισχύ ισοδύναμη με τις κλασικές Μηχανές Turing αλλά με δυνητικές επιταχύνσεις για ορισμένα προβλήματα.

Μια σημαντική ανακάλυψη έγινε το 1994 όταν ο Peter Shor ανέπτυξε έναν κβαντικό αλγόριθμο πολυωνυμικού χρόνου για την παραγοντοποίηση πρώτων αριθμών και τους διακριτούς λογαρίθμους, γνωστός ως αλγόριθμος του Shor. Αυτός ο αλγόριθμος έδειξε ότι οι κβαντικοί υπολογιστές μπορούν να λύσουν προβλήματα όπως η παραγοντοποίηση ακεραίων εκθετικά γρηγορότερα από τους κλασικούς υπολογιστές, θέτοντας μια πιθανή απειλή για τα κλασικά κρυπτογραφικά συστήματα όπως το RSA.

Το 1996, ο Lov Grover εισήγαγε έναν κβαντικό αλγόριθμο αναζήτησης που μπορούσε να αναζητήσει σε μια μη δομημένη βάση δεδομένων σε χρόνο $\sqrt{n}$, προσφέροντας μια τετραγωνική επιτάχυνση σε σχέση με τους κλασικούς αλγορίθμους. Ο αλγόριθμος του Grover υπογράμμισε τη δυνατότητα της κβαντικής υπολογιστικής για προβλήματα βελτιστοποίησης.

Υλοποίηση και Προγραμματισμός(Τέλη 1990ς – Αρχές 2000ς)

Μέχρι τα τέλη της δεκαετίας του 1990, η θεμελιώδης έρευνα για την κβαντική υπολογιστική ήταν καλά εδραιωμένη και οι προσπάθειες στράφηκαν προς την υλοποίηση. Το 1998, ο Ömer ανέπτυξε τη QCL, μια γλώσσα προγραμματισμού για κβαντικούς υπολογιστές, επιτρέποντας την εξερεύνηση κβαντικών αλγορίθμων σε επίπεδο λογισμικού.

Την ίδια χρονιά, οι Isaac Chuang και Neil Gershenfeld στο MIT κατασκεύασαν τον πρώτο κβαντικό υπολογιστή 2 qubit χρησιμοποιώντας Πυρηνικό Μαγνητικό Συντονισμό (NMR). Αυτό σημάδεψε την πρώτη φυσική υλοποίηση ενός κβαντικού υπολογιστή. Μέχρι το 2001, η IBM επέκτεινε αυτή την εργασία, αναπτύσσοντας έναν κβαντικό υπολογιστή NMR 7 qubit και εφαρμόζοντας με επιτυχία τον αλγόριθμο του Shor, αποδεικνύοντας τη σκοπιμότητα του υλικού της κβαντικής υπολογιστικής.

Η ιστορία της κβαντικής υπολογιστικής αντανακλά μια αξιοσημείωτη αλληλεπίδραση μεταξύ θεωρητικών ενοράσεων και πρακτικών προόδων. Από τις πρώτες ιδέες των Benioff και Feynman μέχρι τους επαναστατικούς αλγορίθμους των Shor και Grover, και τις πρώτες φυσικές υλοποιήσεις των Chuang και Gershenfeld, η κβαντική υπολογιστική έχει εξελιχθεί σε ένα ζωντανό πεδίο με τη δυνατότητα να επαναπροσδιορίσει τον υπολογισμό. Ενώ παραμένουν προκλήσεις στην κλιμάκωση του κβαντικού υλικού και στην αντιμετώπιση σφαλμάτων, η πρόοδος που έχει γίνει μέχρι σήμερα υπογραμμίζει τη μετασχηματιστική δυνατότητα της κβαντικής υπολογιστικής σε επιστημονικούς τομείς, στην κρυπτογραφία και σε πολλά άλλα πεδία.

2.2 Qubit

Όπως όλοι γνωρίζουμε, ένας κλασικός υπολογιστής έχει τα μπιτ (bit) τα οποία είναι η μικρότερη μονάδα που μπορεί να επεξεργαστεί και να αποθηκεύσει ένας υπολογιστής. Τα bit μπορούν να πάρουν δύο τιμές. Μπορούν να είναι είτε 0 είτε 1. Ένας κβαντικός υπολογιστής χρησιμοποιεί τα qubits (quantum bits) τα οποία μπορούν να υπάρχουν σε πολλαπλές καταστάσεις ταυτόχρονα χάρη στους νόμους της κβαντομηχανικής.

Ένα qubit δεν περιορίζεται μόνο στις τιμές 0 και 1, αλλά μπορεί να βρίσκεται σε υπέρθεση αυτών των δύο καταστάσεων ταυτόχρονα. Η μαθηματική αναπαράσταση των πολλαπλών αυτών καταστάσεων γίνεται με το διάνυσμα κατάστασης. Το διάνυσμα κατάστασης περιέχει όλες τις πληροφορίες σχετικά με τις πιθανότητες που έχουμε να πάρουμε κάθε πιθανό αποτέλεσμα αν μετρήσουμε το σύστημα.

Διάνυσμα κατάστασης ενός qubit:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$$

- $|\psi\rangle$ είναι το διάνυσμα κατάστασης.
- α και β είναι μιγαδικοί αριθμοί που περιγράφουν την πιθανότητα να μετρήσουμε κάθε κατάσταση. Είναι αλλιώς γνωστοί ως πλάτη
- $|0\rangle$ Η κατάσταση το qubit να πάρει την τιμή 0 (έχει πιθανότητα α).
- $|1\rangle$ Η κατάσταση το qubit να πάρει την τιμή 1 (έχει πιθανότητα β).

Πρέπει να σημειώσουμε ότι η συνολική πιθανότητα πρέπει να είναι πάντα 1. Οπότε πρέπει

$$\alpha^2 + \beta^2 = 1$$

Συχνά αναπαριστούμε τον διάνυσμα κατάστασης σαν πίνακα.

$$|\psi\rangle = \begin{pmatrix} \alpha \\ \beta \end{pmatrix}$$

Το qubit έχει ορισμένες πολύ χρήσιμες ιδιότητες. Η δυνατότητα να βρίσκεται ταυτόχρονα σε πολλές καταστάσεις είναι μία από αυτές και ονομάζεται υπέρθεση. Το παραπάνω διάνυσμα κατάστασης περιγράφει αυτή την ιδιότητα. Λόγω της υπέρθεσης αλλά και άλλων κβαντικών φαινομένων, όπως η διεμπλοκή, οι κβαντικοί υπολογιστές μπορούν να εκτελούν πολλαπλούς υπολογισμούς ταυτόχρονα, δίνοντάς μας τεράστια υπολογιστική ισχύς σε σχέση με τους κλασικούς υπολογιστές.

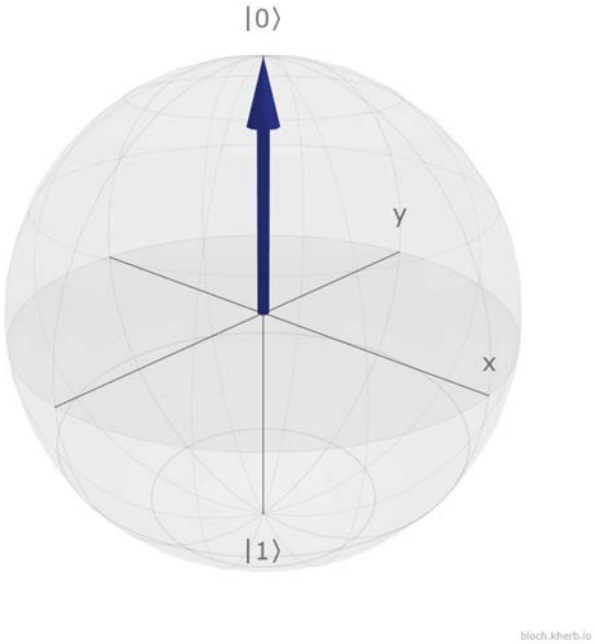
2.3 Η σφαίρα Bloch

Η σφαίρα Bloch είναι ένας τρόπος για να οπτικοποιήσουμε την κατάσταση ενός qubit στο χώρο. Δεδομένου ότι ένα qubit μπορεί να βρίσκεται σε υπέρθεση των καταστάσεων 0 και 1, δεν μπορούμε να το αναπαραστήσουμε απλά ως έναν διακόπτη ανοιχτό/κλειστό.

Γιατί όμως χρειαζόμαστε την σφαίρα Bloch; Τα πλάτη πιθανότητας, δηλαδή οι αριθμοί που περιγράφουν την κατάσταση στην οποία βρίσκεται το qubit είναι μιγαδικοί αριθμοί και επομένως χρειάζονται περισσότερες διαστάσεις για να αναπαρασταθούν. Η

σφαίρα Bloch μας επιτρέπει να δούμε την κατάσταση ενός qubit ως ένα διάνυσμα μέσα σε μία σφαίρα.

Πρακτικά η σφαίρα Bloch λειτουργεί ως εξής. Η βόρεια κορυφή της σφαίρας αντιστοιχεί στην κατάσταση 0, ενώ η νότια κορυφή στην κατάσταση 1. Όλες οι ενδιάμεσες θέσεις αντιστοιχούν σε υπερθέσεις των δύο καταστάσεων. Η γωνία θ καθορίζει πόσο το qubit πλησιάζει μία από τις δύο καταστάσεις 0 και 1. Ενώ η γωνία φ επηρεάζει τη σχετική φάση μεταξύ των δύο καταστάσεων, στοιχείο πολύ σημαντικό για τους κβαντικούς υπολογισμούς.



Εικόνα 1 Σφαίρα Bloch (https://bloch.kherb.io).

Η σφαίρα Bloch είναι ένα πολύ σημαντικό εργαλείο οπτικοποίησης της κατάστασης ενός qubit. Μας δείχνει τις άπειρες ενδιάμεσες τιμές που μπορεί να πάρει ένα qubit και γιατί η γωνία φάσης είναι σημαντική παρόλο που δεν αλλάζει την πιθανότητα μέτρησης. Τέλος είναι ένα θεμελιώδες εργαλείο στην κβαντική πληροφορική, γιατί μας επιτρέπει να κατανοήσουμε πώς λειτουργούν οι κβαντικές πύλες και οι υπολογισμοί.

2.4 Καταχωρητές

Ένας κβαντικός καταχωρητής είναι μια συλλογή από n αριθμό qubits. Χρησιμοποιείται για την αποθήκευση και επεξεργασία της κβαντικής πληροφορίας. Ξέρουμε πως η γενική μορφή ενός qubit είναι $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, δηλαδή μπορεί να βρίσκεται σε μία υπέρθεση των δύο καταστάσεων, με την προϋπόθεση πως $\alpha^2 + \beta^2 = 1$, όπου α και β είναι μιγαδικοί αριθμοί που αντιπροσωπεύουν τα πλάτη πιθανότητας. Αυτό σημαίνει πως αν μετρήσουμε το qubit με πιθανότητα $|\alpha|^2$ θα το βρούμε στην κατάσταση 0 και με πιθανότητα $|\beta|^2$ στην κατάσταση 1.

Στην περίπτωση που έχουμε πάνω από ένα qubit σε έναν καταχωρητή τότε η κατάσταση που θα δημιουργήσουν θα βρεθεί με την βοήθεια του τανυστικού γινομένου.

Ας δημιουργήσουμε έναν καταχωρητή με 2 qubits με αρχική κατάσταση και των δύο στο 0. Τότε ο καταχωρητής αυτός θα είναι $|00\rangle$:

$$|00\rangle = |0\rangle \otimes |0\rangle = \begin{bmatrix}1\\0\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}1 & \begin{bmatrix}1\\0\end{bmatrix}\\ 0 & \begin{bmatrix}1\\0\end{bmatrix}\end{bmatrix} = \begin{bmatrix}1\\0\\0\\0\end{bmatrix}$$

Οπότε στον καταχωρητή αυτό η πιθανότητα μέτρησης της κατάστασης $|00\rangle$ είναι 1 ενώ οι πιθανότητες για τις καταστάσεις $|01\rangle$, $|10\rangle$ και $|11\rangle$ είναι 0. Αυτό ακολουθεί και τον κανόνα πως το σύνολο των τετραγώνων των πιθανοτήτων πρέπει να ισούται με 1 αφού $1^2 + 0^2 + 0^2 + 0^2 = 1$.

Στον παραπάνω καταχωρητή αρχικοποιούμε τα qubits στο 0. Όμως υπάρχουν και άλλες καταστάσεις στις οποίες μπορούν να αρχικοποιηθούν. Αυτές είναι $|01\rangle$, $|10\rangle$ και $|11\rangle$. Σε αυτές τις περιπτώσεις τα διανύσματα θα είναι:

$$|01\rangle = \begin{bmatrix}0\\1\\0\\0\end{bmatrix} \qquad |10\rangle = \begin{bmatrix}0\\0\\1\\0\end{bmatrix} \qquad |11\rangle = \begin{bmatrix}0\\0\\0\\1\end{bmatrix}$$

Ένας καταχωρητής με 2 qubits είναι σχετικά απλός. Όσο αυξάνεται ο αριθμός των qubits το σύστημα γίνεται πιο περίπλοκο. Όταν τα qubits είναι τρία τότε οι πιθανές αρχικές καταστάσεις στις οποίες μπορούμε να τα αρχικοποιήσουμε είναι $|000\rangle$, $|001\rangle$,

|010⟩, |011⟩, |100⟩, |101⟩, |110⟩ και |111⟩. Για την κατάσταση του καταχωρητή θα ακολουθήσουμε την ίδια διαδικασία και θα εφαρμόσουμε τανυστικό γινόμενο. Έστω ότι η αρχική κατάσταση είναι |110⟩ τότε θα έχουμε:

$$|110\rangle = |1\rangle \otimes |1\rangle \otimes |0\rangle = \begin{bmatrix}0\\1\end{bmatrix} \otimes \begin{bmatrix}0\\1\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}0 & \begin{bmatrix}0\\1\end{bmatrix}\\ 1 & \begin{bmatrix}0\\1\end{bmatrix}\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}0 & \begin{bmatrix}1\\0\end{bmatrix}\\ 0 & \begin{bmatrix}1\\0\end{bmatrix}\\ 0 & \begin{bmatrix}1\\0\end{bmatrix}\\ 1 & \begin{bmatrix}1\\0\end{bmatrix}\end{bmatrix} = \begin{bmatrix}0\\0\\0\\0\\0\\0\\1\\0\end{bmatrix}$$

Η ίδια διαδικασία ισχύει σε οποιαδήποτε κατάσταση και αν βρίσκονται τα qubits. Έστω:

$$|q1\rangle = \frac{3}{5}|0\rangle + \frac{4}{5}|1\rangle \text{ και } |q2\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$$

Τότε η κατάσταση του καταχωρητή με την βοήθεια του τανυστικού γινομένου θα είναι:

$$|q1q2\rangle = |q1\rangle \otimes |q1\rangle = \begin{bmatrix}\frac{3}{5}\\ \frac{4}{5}\end{bmatrix} \otimes \begin{bmatrix}\frac{1}{\sqrt{2}}\\ \frac{1}{\sqrt{2}}\end{bmatrix} = \begin{bmatrix}\frac{3}{5} & \begin{bmatrix}\frac{1}{\sqrt{2}}\\ \frac{1}{\sqrt{2}}\end{bmatrix}\\ \frac{4}{5} & \begin{bmatrix}\frac{1}{\sqrt{2}}\\ \frac{1}{\sqrt{2}}\end{bmatrix}\end{bmatrix} = \begin{bmatrix}\frac{3}{5\sqrt{2}}\\ \frac{3}{5\sqrt{2}}\\ \frac{4}{5\sqrt{2}}\\ \frac{4}{5\sqrt{2}}\end{bmatrix}$$

$$= \frac{3}{5\sqrt{2}}\begin{bmatrix}1\\0\\0\\0\end{bmatrix} + \frac{3}{5\sqrt{2}}\begin{bmatrix}0\\1\\0\\0\end{bmatrix} + \frac{4}{5\sqrt{2}}\begin{bmatrix}0\\0\\1\\0\end{bmatrix} + \frac{4}{5\sqrt{2}}\begin{bmatrix}0\\0\\0\\1\end{bmatrix}$$

$$= \frac{3}{5\sqrt{2}}|00\rangle + \frac{3}{5\sqrt{2}}|01\rangle + \frac{4}{5\sqrt{2}}|10\rangle + \frac{4}{5\sqrt{2}}|11\rangle$$

Φυσικά πρέπει να ελέγξουμε αν η παραπάνω είναι αποδεκτή κβαντική κατάσταση.

$$\left(\frac{3}{5\sqrt{2}}\right)^2+\left(\frac{3}{5\sqrt{2}}\right)^2+\left(\frac{4}{5\sqrt{2}}\right)^2+\left(\frac{4}{5\sqrt{2}}\right)^2=1$$

Με την χρήση της Python και της βιβλιοθήκης qiskit ας δούμε πως αρχικοποιούμε qubits σε έναν καταχωρητή.

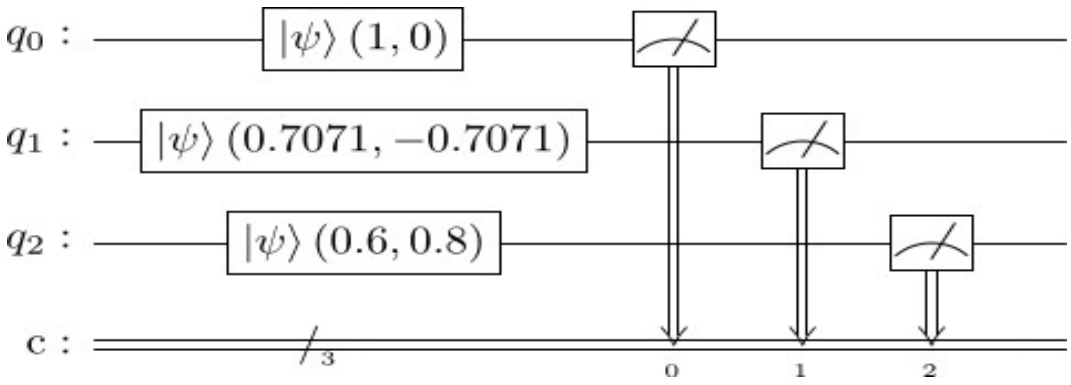
```
1. from qiskit import *
2. from qiskit.visualization import plot_histogram
3. from math import pi, sqrt
4. import matplotlib.pyplot as plt
5. %matplotlib inline
6. initial_state_0 = [1, 0]
7. initial_state_1 = [1/sqrt(2), -1/sqrt(2)]
8. initial_state_2 = [3/5, 4/5]
9.  n = 3
10. qr = QuantumRegister(n, 'q')
11. cr = ClassicalRegister(n, 'c')
12. qc = QuantumCircuit(qr, cr)
13. qc.initialize(initial_state_0, 0)
14. qc.initialize(initial_state_1, 1)
15. qc.initialize(initial_state_2, 2)
16. qc.measure(qr, cr)
17. qc.draw(output='latex')
```

Πρόγραμμα 1 Αρχικοποίηση qubits στον καταχωρητή.

Περιγραφή κώδικα :

- Γραμμές 1,2,3 και 4: Εισαγωγή βιβλιοθηκών που θα χρησιμοποιήσουμε.
- Γραμμή 5: Μια εντολή του Jupyter Notebook που διασφαλίζει ότι όλα τα διαγράμματα που δημιουργούνται από το matplotlib θα εμφανίζονται μέσα στο notebook.
- Γραμμές 6,7 και 8: Ορίζουμε τις αρχικές κβαντικές καταστάσεις.
- Γραμμή 9: Μεταβλητή που θα χρησιμοποιήσουμε για να ορίσουμε τον αριθμό των qubits στον καταχωρητή.
- Γραμμή 10: Δημιουργία κβαντικού καταχωρητή με 3 qubits.
- Γραμμή 11: Δημιουργία κλασσικού καταχωρητή με 3 qubits.
- Γραμμή 12: Δημιουργία κβαντικού κυκλώματος με βάση τους καταχωρητές.
- Γραμμή 13,14 και 15: Αρχικοποιούμε τα qubits στις αρχικές καταστάσεις που δηλώσαμε παραπάνω.

- Γραμμή 16: Μετράμε τα qubits και αποθηκεύουμε τις τιμές τους στα κλασσικά bits.
- Γραμμή 17: Παρουσιάζουμε το κύκλωμα με μια πιο καθαρή και μαθηματικά σωστή εμφάνιση, με την χρήση της βιβλιοθήκης latex.



Εικόνα 2 Κύκλωμα τριών qubits με αρχικές καταστάσεις.

Στην συνέχεια του κώδικα για την μέτρηση των πιθανοτήτων των κβαντικών καταστάσεων θα έχουμε:

```
1. backend = Aer.get_backend('qasm_simulator')
2. job = execute(qc, backend, shots=1024)
3. result = job.result()
4. counts = result.get_counts(qc)
5. plt.bar(sorted_keys, [normalized_counts[key] for key in sorted_keys])
6. plt.ylabel('Πιθανότητες')
7. plt.show()
```

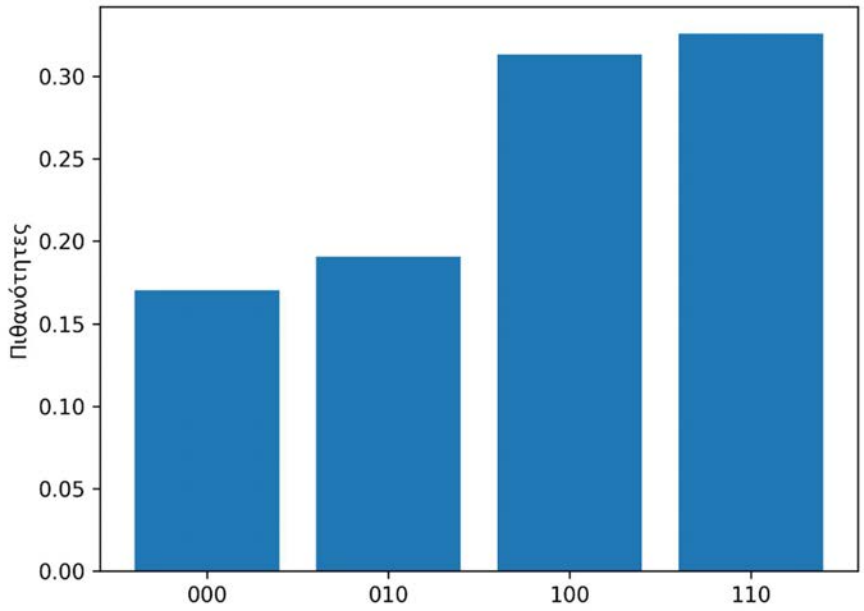
Πρόγραμμα 2 Μέτρηση και οπτικοποίηση αποτελεσμάτων.

Περιγραφή κώδικα :

- Γραμμή 1: Επιλέγουμε τον εξομοιωτή qasm_simulator από το Qiskit Aer.
- Γραμμή 2: Εκτελούμε το κβαντικό κύκλωμα qc στον επιλεγμένο εξομοιωτή. Το κύκλωμα θα εκτελεστεί 1024 φορές για να πάρουμε στατιστικά δεδομένα.
- Γραμμή 3 και 4: Λαμβάνουμε τα αποτελέσματα.

- Γραμμή 5,6 και 7: Οπτικοποιούμε τα αποτελέσματα σε ένα ιστόγραμμα με την βοήθεια της βιβλιοθήκης matplotlib.

Από το παραπάνω κώδικα θα προκύψει ένα ιστόγραμμα με τις εξής μετρήσεις:



Εικόνα 3 Ιστόγραμμα πιθανών αποτελεσμάτων καταχωρητή.

Ο κβαντικός καταχωρητής εκμεταλλεύεται την υπέρθεση και το τανυστικό γινόμενο για να αποθηκεύει και να επεξεργάζεται εκθετικά περισσότερη πληροφορία από έναν κλασικό καταχωρητή. Αυτή η ιδιότητα αποτελεί τη βάση της κβαντικής υπολογιστικής ισχύος, οδηγώντας σε ταχύτητες σημαντικά μικρότερες στην επίλυση προβλημάτων ό-πως η αναζήτηση σε βάσεις δεδομένων, με χρήση του αλγόριθμου του Grover ή η πα-ραγοντοποίηση αριθμών με χρήση του αλγορίθμου του Shor.

2.5 Πύλες

Οι κβαντικές πύλες επηρεάζουν τα qubits αλλάζοντας την κατάσταση τους με τρόπους που δεν μπορούν να επιτευχθούν. Είναι θεμελιώδη δομικά στοιχεία για την δημιουργία

κβαντικών κυκλωμάτων και συνεπώς στην εφαρμογή κβαντικών αλγορίθμων. Αναπαρίστανται με την χρήση πινάκων και βασίζονται σε αρχές της κβαντομηχανικής, όπως η υπέρθεση και η διεμπλοκή.

Κβαντικές πύλες υπάρχουν πολλές. Οι βασικές πύλες με τις οποίες θα ασχοληθούμε είναι οι πύλες μονής κβαντικής κατάστασης οι οποίες επηρεάζουν μόνο ένα qubit όπως η Hadamard και οι πύλες δύο κβαντικών καταστάσεων που επηρεάζουν δύο qubit ταυτόχρονα και δημιουργούν συσχετίσεις μεταξύ τους όπως η πύλη CNOT (Controlled-NOT).

2.5.1 Hadamard

Στην κλασική υπολογιστική ένα bit μπορεί να πάρει μόνο μία από τις δύο τιμές 0 και 1. Αντίθετα στην κβαντική υπολογιστική ένα qubit δεν είναι απλώς 0 ή 1, αλλά μπορεί να είναι και τα δύο ταυτόχρονα σε διαφορετικούς βαθμούς. Αυτή η ιδιότητα του qubit να βρίσκεται σε πολλαπλές καταστάσεις ταυτόχρονα ονομάζεται υπέρθεση.

Η Hadamard είναι μία από τις βασικότερες πύλες της κβαντικής υπολογιστικής. Η λειτουργία της είναι να θέτει τα qubit σε υπέρθεση ενώ επιδρά σε ένα μόνο qubit. Η δημιουργία υπερθέσεων επιτρέπει στους κβαντικούς υπολογιστές να κάνουν παράλληλους υπολογισμούς και έτσι να εκτελούν υπολογισμούς πολύ πιο γρήγορα από τους κλασικούς υπολογιστές.

Πως δρα η Hadamard στις βασικές καταστάσεις :

Αν εφαρμοστεί στο $|0\rangle$ $\quad H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$

Αν εφαρμοστεί στο $|1\rangle$ $\quad H|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$

Η εφαρμογή της πύλης είναι συμμετρική, δηλαδή αν εφαρμόσουμε δύο φορές την πύλη Hadamard σε ένα qubit τότε αυτό επιστρέφει στην αρχική του κατάσταση.

$$H \cdot H = I$$

$$H \cdot H|0\rangle = |0\rangle$$

$$H \cdot H|1\rangle = |1\rangle$$

Η μαθηματική αναπαράσταση της πύλης είναι ένας πίνακας 2x2.

$$H = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}$$

Για να δούμε πως εφαρμόζεται μαθηματικά η πύλη Hadamard σε ένα qubit. Έστω ότι έχουμε ένα qubit με διαφορετική αρχική κατάσταση των $|0\rangle$ και $|1\rangle$. Για παράδειγμα $|x\rangle = \begin{bmatrix}-0,8\\ 0,6\end{bmatrix}$, το οποίο είναι αποδεκτή τιμή αφού $(-0,8)^2 + (0,6)^2 = 1$. Αν λοιπόν η πύλη Hadamard επιδράσει στο qubit τότε το αποτέλεσμα που θα έχουμε θα είναι:

$$H|x\rangle = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}\begin{bmatrix}-0,8\\ 0,6\end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix}-0,2\\ -1,4\end{bmatrix} = \begin{bmatrix}-0,14\\ -0,99\end{bmatrix} = -0,14\begin{bmatrix}1\\ 0\end{bmatrix} - 0,99\begin{bmatrix}0\\ 1\end{bmatrix} = -0,14|0\rangle - 0,99|1\rangle$$

Πρακτικά αυτό σημαίνει πως η πιθανότητα να εμφανιστεί $|0\rangle$ είναι περίπου 98% ($0,99^2$) ενώ η πιθανότητα να εμφανιστεί $|1\rangle$ είναι περίπου 2% ($0,14^2$).

Η πύλη Hadamard αν εφαρμοστεί για δεύτερη φορά θα μας δώσει την αρχική τιμή του qubit.

$$H\begin{bmatrix}-0,14\\ -0,99\end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}\begin{bmatrix}-0,14\\ -0,99\end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix}-1.13\\ 0.85\end{bmatrix} = \begin{bmatrix}-0,80\\ 0.60\end{bmatrix}$$

Γράφοντας τον κώδικα στην qiskit θα έχουμε:

```
1. from qiskit import *
2. from qiskit.visualization import plot_histogram
3. import matplotlib.pyplot as plt
4. %matplotlib inline
5. initial_state_0 = [-4/5, 3/5]
6. qr = QuantumRegister(1, 'q')
7. cr = ClassicalRegister(1, 'c')
8. qc = QuantumCircuit(qr, cr)
9. qc.initialize(initial_state_0, 0)
10. qc.h(0)
11. qc.measure(qr, cr)
12. qc.draw(output='latex')
13. backend = Aer.get_backend('qasm_simulator')
14. job = execute(qc, backend, shots=1024)
15. result = job.result()
16. counts = result.get_counts(qc)
17. total_shots = sum(counts.values())
```

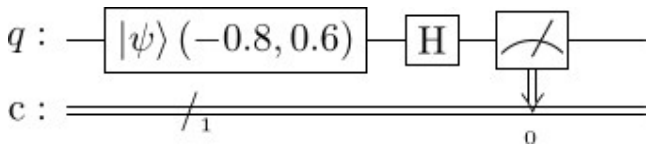
```
18. normalized_counts = {key: value / total_shots for key, value in counts.items()}
19. sorted_keys = sorted(normalized_counts.keys())
20. plt.bar(sorted_keys, [normalized_counts[key] for key in sorted_keys])
21. plt.ylabel('Πιθανότητες')
22. plt.show()
```

Πρόγραμμα 3 Εφαρμογή πύλης Hadamard σε ένα qubit.

Περιγραφή κώδικα:

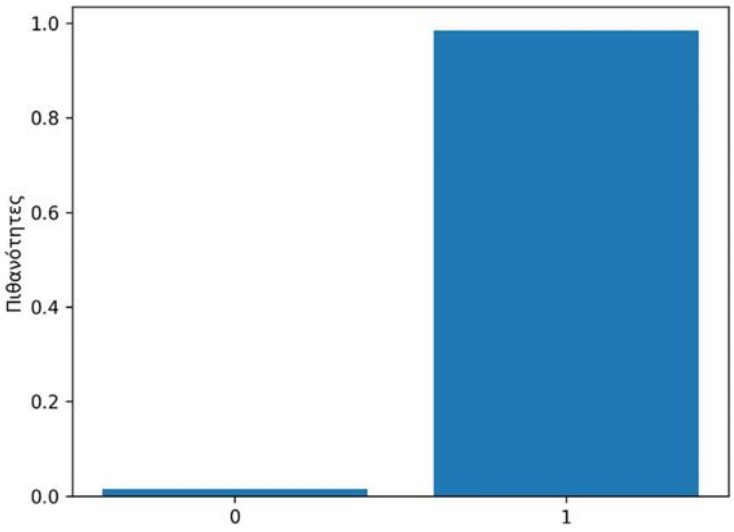
- Γραμμές 1 έως 9 όπως και στο προηγούμενο πρόγραμμα εισάγουμε τις βιβλιοθήκες που θα χρειαστούμε, δημιουργούμε τους καταχωρητές και το κύκλωμα και αρχικοποιούμε την κατάσταση του qubit.
- Γραμμή 10: Εφαρμογή της πύλης Hadamard στο qubit (0). Αν είχαμε παραπάνω από ένα qubit τότε μέσα στην παρένθεση θα πρέπει να δηλώσουμε σε ποιο θα θέλαμε να εφαρμόσουμε την πύλη.
- Γραμμές 11 έως 22 κάνουμε τα ίδια με το προηγούμενο πρόγραμμα. Μετράμε τα αποτελέσματα, εκτυπώνουμε το κύκλωμα, επιλέγουμε εξομοιωτή, εκτελούμε το κβαντικό κύκλωμα qc στον επιλεγμένο εξομοιωτή 1024 φορές, λαμβάνουμε τα αποτελέσματα και τα οπτικοποιούμε σε ένα ιστόγραμμα.

Το κύκλωμα θα είναι:



Εικόνα 4 Κύκλωμα ενός qubit με εφαρμογή της Hadamard

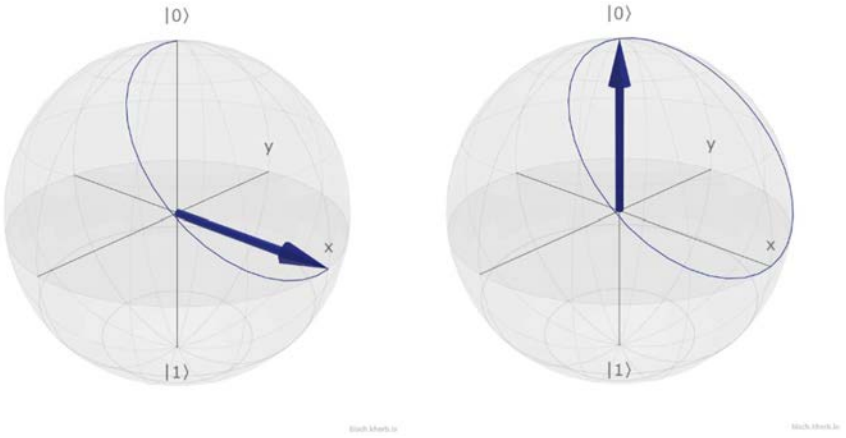
Και το ιστόγραμμα με τα αποτελέσματα θα είναι:



Εικόνα 5 Ιστόγραμμα αποτελεσμάτων.

Είναι εμφανές ότι τα αποτελέσματα είναι τα ίδια με αυτά που υπολογίσαμε με την βοήθεια των πινάκων.

Ας δούμε πως εμφανίζεται στην σφαίρα Bloch η εφαρμογή της πύλης Hadamard.



Εικόνα 6: Hadamard στην σφαίρα Bloch (https://bloch.kherb.io).

Στην Εικόνα 6 έχουμε αρχικά την εφαρμογή της Hadamard και στην συνέχεια την εφαρμόζουμε για δεύτερη φορά. Αυτό που παρατηρούμε είναι πως το qubit επιστρέφει στη αρχική του κατάσταση.

2.5.2 Πύλη Αδράνειας

Η πύλη αδράνειας (Identity Gate, I) είναι μια απλή αλλά σημαντική πύλη στην κβαντική υπολογιστική. Στην σχεδίαση και σύνθεση κβαντικών κυκλωμάτων η πύλη αυτή χρησιμοποιείται για να διατηρηθεί η κατάσταση των qubits. Εφαρμόζεται σε ένα qubit και δεν προκαλεί σε αυτό καμία αλλαγή αλλά αντίθετα διατηρεί την κατάστασή του αμετάβλητη. Η μαθηματική αναπαράσταση της πύλης είναι :

$$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

2.5.3 Πύλες Pauli

Οι πύλες Pauli είναι θεμελιώδης έννοια στον κβαντικό υπολογισμό. Είναι κβαντικές πύλες ενός qubit και πήραν το όνομά τους από τον Αυστριακό θεωρητικό φυσικό και πρωτοπόρο της κβαντικής φυσικής Wolfgang Pauli. Κάθε πύλη αναπαριστάτε με ένα γράμμα X, Y και Z και κάθε μία ισοδυναμεί αντίστοιχα με μία περιστροφή γύρω από τους άξονες x,y,z της σφαίρας Bloch κατά π μοίρες.

Πύλη X

Η πύλη X είναι ανάλογη της κλασσικής πύλης NOT και αυτό που κάνει είναι να αντιστρέφει το qubit. Στην σφαίρα Bloch αυτό μεταφράζεται σε περιστροφή 180^{O} στον x άξονα.

$$X = \sigma\chi = NOT = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Παράδειγμα

$$X\,|0\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} = |1\rangle$$

Κώδικας εφαρμογής πύλης X σε ένα qubit κατάστασης |0⟩ με qiskit.

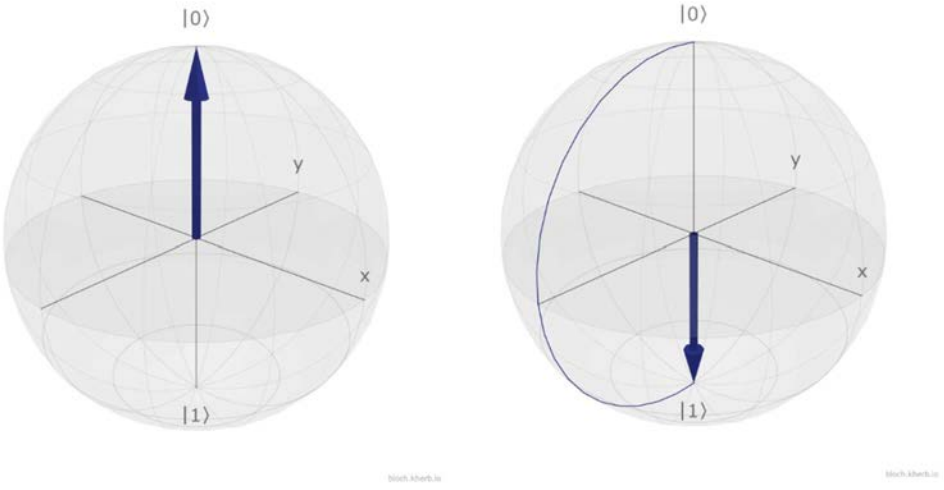
```
1. from qiskit import QuantumCircuit, Aer, execute
2. from qiskit.visualization import plot_bloch_multivector
3. from qiskit.quantum_info import Statevector
4. import matplotlib.pyplot as plt
5. n=1
6. qc = QuantumCircuit(n)
7. qc.x(0)
8. state = Statevector.from_instruction(qc)
9. plot_bloch_multivector(state)
```

Πρόγραμμα 4 Εφαρμογή πύλης X.

Περιγραφή κώδικα:

- Γραμμή 1 έως 4: Βιβλιοθήκες.
- Γραμμή 5: Σε μία μεταβλητή ορίζουμε τον αριθμό των qubits.
- Γραμμή 6: Καταχωρητής.
- Γραμμή 7: Εφαρμογή πύλης X.
- Γραμμή 8 και 9: Εμφάνιση της αλλαγής που θα επιφέρει η πύλη X με την βοήθεια της σφαίρας Bloch.

Όπως εφαρμόσαμε πριν την πύλη Hadamard με τον ίδιο τρόπο εφαρμόζουμε και την πύλη X με την εντολή qc.x(0) υποδεικνύοντας το κύκλωμα qc και το qubit στο οποίο θέλουμε να εφαρμόσουμε την πύλη.



Εικόνα 7: Εφαρμογή πύλης X (https://bloch.kherb.io).

Πύλη Y

Η πύλη αυτή περιστρέφει τις κβαντικές καταστάσεις κατά 180^o γύρω από τον άξονα Y. Είναι από τις βασικότερες πύλες που εφαρμόζονται σε ένα qubit και εκφράζεται ως ο ακόλουθος 2x2 πίνακας :

$$Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$$

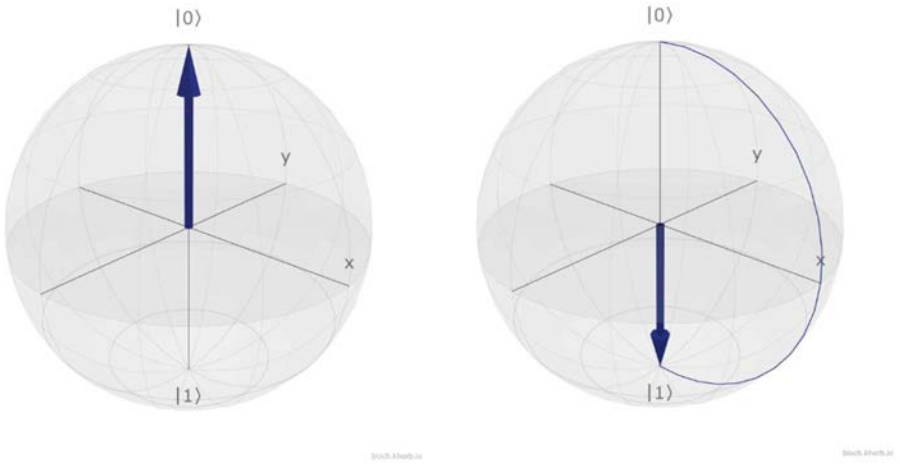
Όπου i είναι η φανταστική μονάδα ($i^2 = -1$)

Αυτή η πύλη είναι ένας ερμητιανός πίνακας. Ερμητιανός χαρακτηρίζεται ένας πίνακας με μιγαδικές τιμές που ισούται με τον ερμητιανό συζυγή του. Είναι επίσης ορθοκανονικός πίνακας, που σημαίνει ότι είναι ο ίδιος ο αντίστροφός του και άρα $Y^2 = I$.

Εφαρμογή Y πύλης σε qubit:

$$Y\,|0\rangle = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ i \end{bmatrix} = i\,|1\rangle$$

$$Y\,|1\rangle = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ i \end{bmatrix} = -i\,|1\rangle$$



Εικόνα 8: Εφαρμογή πύλης Y (https://bloch.kherb.io).

Πύλη Z

Η πύλη Z είναι άλλη μια από τις πύλες Pauli. Είναι μία θεμελιώδης πύλη ενός qubit με πολύ βασικό ρόλο στους κβαντικούς αλγορίθμους και τη διόρθωση κβαντικών σφαλμάτων. Η αναπαράσταση της πύλης γίνεται από τον πίνακα:

$$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

Αυτό που πρακτικά κάνει η πύλη Z είναι μια αναστροφή φάσης(αλλαγή στο πρόσημο του πλάτους) για την κατάσταση |1⟩ ενώ αφήνει την κατάσταση |0⟩ αμετάβλητη.

$$Z|0\rangle = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} = |0\rangle$$

$$Z|1\rangle = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ -1 \end{bmatrix} = -|1\rangle$$

Στην σφαίρα Bloch η Z πύλη εκτελεί μια περιστροφή 180 μοιρών γύρω από τον άξονα z. Παρόλο που αντιστρέφει τη φάση δεν αλλάζει τις πιθανότητες μέτρησης των καταστάσεων |0⟩ και |1⟩.

2.5.4 Πύλη CNOT (cX)

Η πύλη CNOT (Controlled-NOT gate), που συχνά δηλώνεται ως πύλη cX, είναι μία από τις πιο σημαντικές πύλες των δύο qubits στον κβαντικό υπολογισμό. Είναι ένα θεμελιώδες δομικό στοιχείο για την δημιουργία της διεμπλοκής και την εφαρμογή κβαντικών αλγορίθμων. Όπως είπαμε η πύλη δρα σε δύο qubits. Το ένα είναι το qubit ελέγχου και το άλλο το qubit στόχος.

Αυτό που κάνει η πύλη cX είναι να αναστρέφει την κατάσταση του qubit στόχου εάν και μόνο εάν το qubit ελέγχου βρίσκεται στην κατάσταση |1⟩. Το qubit ελέγχου δεν επηρεάζεται με κανένα τρόπο. Στα διαγράμματα κβαντικών κυκλωμάτων η CNOT αναπαρίσταται ως μία κατακόρυφη που συνδέει το qubit ελέγχου, που συμβολίζεται με μια κουκίδα, με το qubit στόχο, που συμβολίζεται με έναν κύκλο και το σύμβολο συν μέσα σε αυτόν.

Η μαθηματική αναπαράσταση της πύλης είναι ο πίνακας:

$$cX = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Πως επιδρά η πύλη στα qubits:

$|00\rangle \rightarrow |00\rangle$

$|01\rangle \rightarrow |01\rangle$

$|10\rangle \rightarrow |11\rangle$

$|11\rangle \rightarrow |10\rangle$

Ποιο αναλυτικά ας δούμε ένα παράδειγμα. Έστω πως έχουμε τον καταχωρητή $|10\rangle$. Τότε:

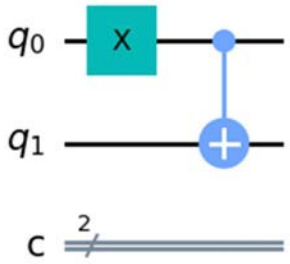
$$cX\,|10\rangle = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = |11\rangle$$

```
1. from qiskit import QuantumCircuit
2. %matplotlib inline
3. qc = QuantumCircuit(2, 2)
4. qc.x(0)
5. qc.cx(0, 1)
6. qc.draw(output='mpl')
```

Πρόγραμμα 5 Εφαρμογή CNOT πύλης.

Περιγραφή κώδικα:

- Γραμμή 4: Εφαρμόζουμε την πύλη X στο πρώτο qubit. Αυτό αλλάζει την κατάσταση του κυκλώματος από $|00\rangle$ σε $|10\rangle$.
- Γραμμή 5: Εφαρμόζουμε την πύλη cX με qubit ελέγχου το πρώτο και qubit στόχο το δεύτερο. Αυτό που περιμένουμε μετά και την εφαρμογή της cX είναι το αποτέλεσμα να αλλάξει σε $|11\rangle$.



Εικόνα 9: Κύκλωμα με εφαρμογή πύλης X και πύλης CNOT.

Στην περίπτωση που έχουμε παραπάνω από 2 qubits και το qubit ελέγχου με το qubit στόχος δεν είναι διαδοχικά τότε θα πρέπει να υπολογίσουμε τον πίνακα που αν πολλαπλασιαστεί με τον καταχωρητή θα μας δώσει το επιθυμητό αποτέλεσμα.

Έστω πως έχουμε έναν καταχωρητή με 3 qubits. Τότε η πύλη CNOT αναπαρίσταται ως ένας πίνακας 8x8. Αυτό συμβαίνει διότι τα τρία qubits έχουν 2^3 πιθανές βασικές καταστάσεις. Κάθε κατάσταση βάσης αντιστοιχεί σε έναν μοναδικό συνδυασμό των τριών qubits, οπότε η πύλη CNOT πρέπει να περιγράφει πως επιδρά σε κάθε μία από αυτές τις καταστάσεις. Στην περίπτωση των τριών qubits οι καταστάσεις βάσης είναι οι $|000\rangle$,$|001\rangle$,$|010\rangle$,$|011\rangle$,$|100\rangle$,$|101\rangle$,$|110\rangle$ και $|111\rangle$.

Για την κατασκευή της CNOT θα γράψουμε κάθε κατάσταση που θα προκύψει ως διάνυσμα στήλης και στην συνέχεια τοποθετώντας αυτά τα διανύσματα μαζί θα προκύψει ο 8x8 πίνακας που χρειαζόμαστε.

Οπότε έχουμε τις αρχικές καταστάσεις και τις καταστάσεις που θα προκύψουν αν εφαρμόσουμε την πύλη CNOT.

$$|000\rangle = \begin{bmatrix}1\\0\\0\\0\\0\\0\\0\\0\end{bmatrix} \quad \text{CNOT}\,|000\rangle = |000\rangle \qquad |001\rangle = \begin{bmatrix}0\\1\\0\\0\\0\\0\\0\\0\end{bmatrix} \quad \text{CNOT}\,|001\rangle = |001\rangle$$

$$|010\rangle = \begin{bmatrix}0\\0\\1\\0\\0\\0\\0\\0\end{bmatrix} \quad \text{CNOT}\,|010\rangle = |010\rangle \qquad |011\rangle = \begin{bmatrix}0\\0\\0\\1\\0\\0\\0\\0\end{bmatrix} \quad \text{CNOT}\,|011\rangle = |011\rangle$$

Οι τέσσερις πρώτες καταστάσεις δεν θα αλλάξουν με την εφαρμογή της CNOT αφού το qubit ελέγχου είναι σε όλες μηδέν.

$$|100\rangle = \begin{bmatrix}0\\0\\0\\0\\1\\0\\0\\0\end{bmatrix} \quad \text{CNOT}\,|100\rangle = |101\rangle \qquad |101\rangle = \begin{bmatrix}0\\0\\0\\0\\0\\1\\0\\0\end{bmatrix} \quad \text{CNOT}\,|101\rangle = |100\rangle$$

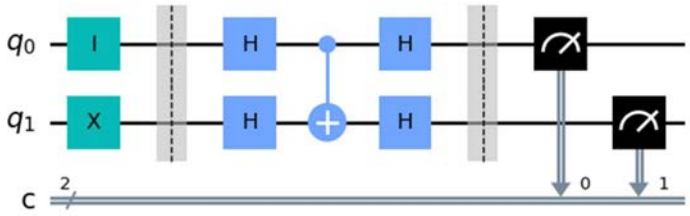
$$|110\rangle = \begin{bmatrix}0\\0\\0\\0\\0\\0\\1\\0\end{bmatrix} \quad \text{CNOT}\,|110\rangle = |111\rangle \qquad |111\rangle = \begin{bmatrix}0\\0\\0\\0\\0\\0\\0\\1\end{bmatrix} \quad \text{CNOT}\,|111\rangle = |110\rangle$$

Στην συνέχεια χρησιμοποιούμε όλα τα διανύσματα που προκύπτουν σε κάθε κατάσταση αφού εφαρμόσουμε την πύλη για να κατασκευάσουμε τον πίνακα. Κάθε διάνυσμα αντιστοιχεί σε μία κάθετη στήλη.

$$\begin{bmatrix}1&0&0&0&0&0&0&0\\0&1&0&0&0&0&0&0\\0&0&1&0&0&0&0&0\\0&0&0&1&0&0&0&0\\0&0&0&0&0&1&0&0\\0&0&0&0&1&0&0&0\\0&0&0&0&0&0&0&1\\0&0&0&0&0&0&1&0\end{bmatrix}$$

Γενικά σε ένα σύστημα με n qubits οι πιθανές καταστάσεις είναι 2^n, και μια κβαντική πύλη που επιδρά σε n qubits αναπαρίσταται ως ένας πίνακας $2^n \times 2^n$ διαστάσεων.

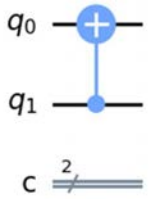
Μία ακόμη περίπτωση που έχει ενδιαφέρον είναι πως θα υλοποιήσουμε το κύκλωμα δύο qubit αν το πρώτο είναι το qubit στόχος και το δεύτερο το qubit ελέγχου. Σε αυτήν τη περίπτωση αρχικά θα εφαρμόσουμε και στα δύο qubit την πύλη Hadamard. Στην συνέχεια την πύλη CNOT κανονικά χωρίς να αλλάζουν τα qubit στόχου και ελέγχου. Τέλος θα εφαρμόσουμε ξανά την πύλη Hadamard.



Εικόνα 10: Εφαρμογή cX με qubit ελέγχου το δεύτερο με χρήση Hadamard.

Όταν εφαρμόζουμε μία πύλη σε ένα qubit, αλλά θέλουμε το άλλο qubit να παραμείνει στην ίδια κατάσταση, τότε εφαρμόζουμε την πύλη I.

Στην qiskit μπορούμε να κάνουμε και άμεση υλοποίηση της cX χωρίς την χρήση των Hadamard πυλών. Αντί για την εντολή qc.cx(0, 1) θα γράψουμε qc.cx(1, 0).



Εικόνα 11: Άμεση εφαρμογή CNOT πύλης.

2.6 Κυκλώματα

Όλα όσα είδαμε μέχρι τώρα χρησιμοποιούνται για να υλοποιούμε κβαντικά κυκλώματα. Ένα κύκλωμα έχει τρία βασικά στοιχεία, τα qubits που είναι η βασικές μονάδες πληροφορίας, τις κβαντικές πύλες οι οποίες εφαρμόζονται στα qubits και οι μετρήσεις. Για την πραγματοποίηση των μετρήσεων απαιτούνται και τα ανάλογα κλασικά bits. Η μέτρηση αυτή ¨καταστρέφει¨ την υπέρθεση και επιστρέφει μία από τις πιθανές καταστάσεις με βάση τις πιθανότητες. Το σύστημα θα καταρρεύσει και η πληροφορία θα αποθηκευτεί στα κλασικά bits.

Με την χρήση κβαντικών κυκλωμάτων μπορούμε να αναπαραστήσουμε οποιονδήποτε κβαντικό υπολογισμό. Να σημειώσουμε ότι η πληροφορία δεν μεταφέρεται από πύλη σε πύλη, αλλά οι πύλες δρουν στους κβαντικούς καταχωρητές στους οποίους υπάρχει η πληροφορία.

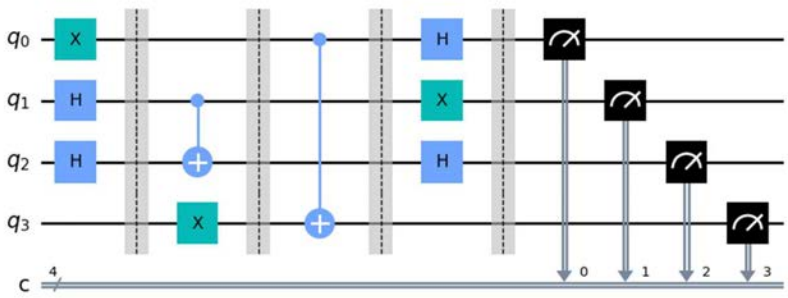
Βασικά στοιχεία των κβαντικών κυκλωμάτων είναι:

Ο αριθμός των qubits που χρησιμοποιούνται στο κύκλωμα. Περισσότερα qubits επιτρέπου την εκτέλεση πιο σύνθετων αλγορίθμων, αλλά αυξάνουν την πολυπλοκότητα.

Είναι το μέγιστο πλήθος πυλών που θα εκτελεσθούν διαδοχικά. Μεγαλύτερο βάθος σημαίνει περισσότερες πύλες και πιο πολύπλοκο κύκλωμα. Επίσης, μεγαλύτερος αριθμός πυλών σημαίνει περισσότερες πηγές σφάλματος.

Τα σφάλματα κατά την εκτέλεση πυλών ή μετρήσεων ορίζουν την αξιοπιστία των κβαντικών υπολογισμών. Για αξιόπιστους κβαντικούς υπολογισμούς χρειαζόμαστε χαμηλό ποσοστό σφάλματος.

Ο κβαντικός όγκος (Quantum Volume) είναι ένα μέτρο της συνολικής ισχύος ενός κβαντικού υπολογιστή, που συνδυάζει το πλήθος των qubits, την πιστότητα και τη συνδεσιμότητα του συστήματος. Χρησιμοποιείται για τη σύγκριση διαφορετικών κβαντικών συστημάτων.



Εικόνα 12: Παράδειγμα κβαντικού κυκλώματος.

Το συγκεκριμένο κύκλωμα αποτελείτε από 4 qubits και έχει βάθος 4. Η σειριακή διαδοχή των πυλών είναι x(0)H(1)H(2)→cX(1, 2)x(3)→cX(0, 3)→H(0)x(1)H(2).

Για τον συνδυασμό των πυλών κάνω χρήση τανυστικού γινομένου. Για παράδειγμα στο πρώτο βήμα που εφαρμόζω πύλη X στο πρώτο qubit και Hadamard στο δεύτερο και τρίτο θα υπολογίσω πρώτα το τανυστικό γινόμενο των πυλών και στην συνέχεια με πολλαπλασιασμό πινάκων θα τις εφαρμόσω στα qubits.

Δεν ξεχνάω ότι στα σημεία που δεν φαίνεται να εφαρμόζεται κάποια πύλη στην πραγματικότητα εφαρμόζεται η πύλη αδράνειας.

2.7 Προγραμματισμός με Python

Για την ανάπτυξη του κβαντικού προγραμματισμού η πιο διαδεδομένη γλώσσα προγραμματισμού είναι η Python. Σήμερα η Python διαθέτει αρκετές βιβλιοθήκες που καλύπτουν ένα ευρύ φάσμα αναγκών. Κάθε βιβλιοθήκη προφέρει διαφορετικά πλεονεκτήματα. Για παράδειγμα η Qiskit, η οποία αναπτύχθηκε από την IBM , περιέχει εργαλεία για τον σχεδιασμό, προσομοίωση και εκτέλεση κβαντικών αλγορίθμων τόσο σε προσομοιωμένα περιβάλλοντα όσο και σε πραγματικό κβαντικό υπολογιστή. Η PennyLane, δημιούργημα της Καναδικής εταιρίας Xanadu, βασίζεται επίσης στην Python και επιτρέπει την ενσωμάτωση με εργαλεία μηχανικής μάθησης. Άλλες βιβλιοθήκες που επίσης χρησιμοποιούνται ευρέως είναι η Cirq της Google και η PyQuil της εταιρίας Rigetti. Εμείς για την ανάπτυξη προγραμμάτων θα χρησιμοποιήσουμε αποκλειστικά την Qiskit και το περιβάλλον του jupyter notebook.

Η IBM εξέδωσε τον Μάρτιο του 2017 το πρώτο πακέτο της Qiskit v0.1 και επιτρέπει έτσι την χρήση ενός πραγματικού κβαντικού υπολογιστή μέσω διαδικτύου. Πριν από την διάθεση της Qiskit πρόσβαση σε κβαντικούς υπολογιστές διέθεταν μόνο ερευνητές αποκλειστικά σε χώρους εργαστηρίων. Από το 2017 μια ολόκληρη κοινότητα έχει δημιουργηθεί και υποστηρίξει την ανάπτυξη της. Πλέον η qiskit είναι η πιο δημοφιλής βιβλιοθήκη για προγραμματισμό κβαντικών υπολογιστών.

Για να χρησιμοποιήσουμε την Qiskit θα πρέπει να εγκαταστήσουμε κάποια εργαλεία.

1. Εγκατάσταση της Python: https://www.python.org/downloads/
2. Εγκατάσταση Pip: https://pip.pypa.io/en/stable/installation/
   Το pip είναι ένας διαχειριστής πακέτων της Python.Στις περισσότερες εκδόσεις Python το pip είναι προ εγκατεστημένο. Σε περίπτωση που δεν υπάρχει θα πρέπει να το κατεβάσετε ακολουθώντας τις οδηγίες.
3. Qiskit SDK: https://docs.quantum.ibm.com/guides/install-qiskit
   Με την χρήση της εντολής **pip install qiskit** μπορούμε να κατεβάσουμε το πακέτο της qiskit. Αν χρειαστεί να τρέξουμε προγράμματα στον κβαντικό υπολογιστή της IBM τότε πρέπει να εγκαταστήσουμε επίσης το Qiskit Runtime **pip install qiskit-ibm-runtime.**
4. Jupyter: https://jupyter.org/install
   Για την εγκατάσταση του jupyter θα χρησιμοποιήσουμε την εντολή **pip install jypyter.** Για περισσότερες λεπτομέρειες ακολουθήστε τις οδηγίες στο link.

Αφού εγκαταστήσουμε όλα τα παραπάνω μπορούμε να ξεκινήσουμε να γράφουμε το πρόγραμμά μας. Για αρχή, πρέπει να ανοίξουμε το jupyter notebook. Αυτό θα το κάνουμε ανοίγοντας της γραμμή εντολών και θα πληκτρολογήσουμε *jupyter notebook*. Στον φυλλομετρητή που θα μας ανοίξει επιλέγουμε new και στην συνέχεια Python. Στο κουτάκι με το νούμερο [1] μπορούμε τώρα να γράψουμε τον κώδικά μας. Για την εκτέλεση του μπορούμε να πατήσουμε Run από την γραμμή εντολών ή απλά Shift + Enter.

Για παράδειγμα αν πληκτρολογήσουμε :

```
1. import qiskit
2. qiskit.__version__
```

Πρόγραμμα 6 Έλεγχος qiskit.

Το αποτέλεσμα πρέπει να είναι η έκδοση qiskit που έχουμε στον υπολογιστή μας.

Τώρα είμαστε έτοιμοι να γράψουμε και να εκτελέσουμε το πρώτο μας κβαντικό πρόγραμμα.

```
1. import qiskit as q
2. from qiskit.tools.visualization import plot_histogram
3. nq = 2
4. qreg = q.QuantumRegister(nq, name = "qubit")
5. creg = q.ClassicalRegister(nq, name = "classic")
6. circuit = q.QuantumCircuit(qreg, creg)
7. circuit.h(qreg[0])
8. circuit.x(qreg[1])
9. circuit.cx(qreg[0],qreg[1])
10. circuit.measure(qreg,creg)
11. circuit.draw(output = 'mpl')
12. simulator = q.Aer.get_backend("qasm_simulator")
13. job = q.execute(circuit, simulator, shots = 1024)
14. result = job.result()
15. counts = result.get_counts(circuit)
16. plot_histogram(counts)
```

Πρόγραμμα 7 Κβαντικό πρόγραμμα

Περιγραφή κώδικα :

- Γραμμές 1 και 2: Εισαγωγή βιβλιοθηκών.
- Γραμμές 3, 4 και 5: Ορίζω τον Κβαντικό και Κλασικό καταχωρητή.
- Γραμμή 6: Δημιουργώ το κύκλωμα.
- Γραμμές 7,8 και 9: Εφαρμογή κβαντικών πυλών.
- Γραμμή 10: Μετράω τα qubits και αποθηκεύω τα αποτελέσματα.
- Γραμμή 11: Εμφανίζω το κύκλωμα.
- Γραμμές 12, 13, 14 και 15: Με την χρήση συγκεκριμένου προσομοιωτή τρέχω το κύκλωμα 1024 φορές.
- Γραμμή 16: Εμφανίζω ένα ιστόγραμμα με τα αποτελέσματα.

3 Πολύ-πύρινος Προγραμματισμός

Ο πολυπύρηνος προγραμματισμός (multi-core programming) είναι μια τεχνική προγραμματισμού που εκμεταλλεύεται την ύπαρξη πολλών πυρήνων (cores) σε έναν επεξεργαστή, ώστε να εκτελεί ταυτόχρονα πολλές εργασίες (parallel processing). Συστήματα με πυρήνες είναι η κεντρική μονάδα επεξεργασίας(CPU) και η μονάδα επεξεργασίας γραφικών(GPU), με την δεύτερη να έχει το προβάδισμα στο πλήθος των πυρήνων που διαθέτει. Γενικά οι πυρήνες αναλαμβάνουν την επεξεργασία των δεδομένων. Κάθε πυρήνας αναλαμβάνει διαφορετική εργασία γεγονός που βοηθάει στην παράλληλη επεξεργασία

Βασικές έννοιες

- Διεργασία (Process): Μια διεργασία είναι ένα ανεξάρτητο πρόγραμμα που εκτελείται στο λειτουργικό σύστημα. Κάθε διεργασία έχει τον δικό της χώρο μνήμης και πόρους.
- Νήμα (Thread): Ένα νήμα είναι μια ελαφριά διεργασία που μπορεί να εκτελείται μέσα σε μια διεργασία. Τα νήματα μοιράζονται τον ίδιο χώρο μνήμης και πόρους με την κύρια διεργασία και μεταξύ τους.
- Πυρήνες (Cores): Είναι μεμονωμένες μονάδες επεξεργασίας μέσα σ έναν επεξεργαστή. Είναι ένας μικρός επεξεργαστής μέσα σε έναν μεγαλύτερο επεξεργαστή.

Ποια είναι τα πλεονεκτήματα που μας προσφέρει ο πολυπύρηνος προγραμματισμός; Αρχικά επιτρέπει την ταυτόχρονη εκτέλεση πολλών εργασιών, γεγονός που μπορεί να μειώσει τον συνολικό χρόνο εκτέλεσης. Έτσι διεργασίες που θα χρειαζόταν αρκετό χρόνο αν εκτελούνταν σειριακά πλέον επιταχύνονται αισθητά. Βελτιώνει την απόδοση εφαρμογών και την αποδοτικότητά τους.

Βέβαια, υπάρχουν προκλήσεις και ζητήματα που χρειάζονται επίλυση. Η ταυτόχρονη πρόσβαση πολλαπλών νημάτων σε κοινά δεδομένα μπορεί να οδηγήσει σε προβλήματα συγχρονισμού και αδιέξοδα. Προσοχή χρειάζεται και η επιλογή αριθμού νημάτων αφού η δημιουργία πολλών νημάτων μπορεί να οδηγήσει σε υπερβολική κατανάλωση πόρων και σε δυσκολία διαχείρισης.

Ο πολυπύρηνος προγραμματισμός είναι μια ισχυρή τεχνική που μπορεί να βελτιώσει σημαντικά την απόδοση των εφαρμογών, αλλά απαιτεί προσεκτικό σχεδιασμό και διαχείριση για την αποφυγή προβλημάτων.

Τέλος να αναφέρουμε πως η GPU αποτελεί την καρδιά της παράλληλης υπολογιστικής δύναμης.

3.1 GPU

Η Graphic Processing Unit(GPU) αλλιώς μονάδα επεξεργασίας γραφικών, είναι ένα ηλεκτρονικό κύκλωμα που σχεδιάστηκε για την επιτάχυνση εκτέλεσης υπολογισμών που σχετίζονται με την αναπαράσταση γραφικών. Αρχικά, αναπτύχθηκε για την διαχείριση γραφικών σε ηλεκτρονικά παιχνίδια, επομένως συχνά αναφέρετε και ως Gaming Processing Unit. Ξεχωρίζει από την CPU λόγο της παράλληλης επεξεργασίας που την καθιστά κατάλληλη για την απόδοση σύνθετων γραφικών σε πραγματικό χρόνο.

Με το πέρασμα των χρόνων, το ενδιαφέρων επικεντρώνεται όχι μόνο στα ηλεκτρονικά παιχνίδια αλλά και στην εκτέλεση διεργασιών γενικού σκοπού γνωστή ως General-Purpose Computing (GPGPU). Πλέον χρησιμοποιείτε σε πεδία όπως η επεξεργασία μεγάλου όγκου δεδομένων, η μηχανική μάθηση και άλλα. Η ανάπτυξη της τεχνητής νοημοσύνης φέρνει την GPU στο προσκήνιο για άλλη μια φορά. Κάρτες έχουν σχεδιαστεί ειδικά για την υποστήριξη της μηχανικής μάθησης και τη εκπαίδευση νευρωνικών δικτύων.

Οι δημοφιλέστεροι κατασκευαστές είναι οι NVIDIA, η AMD και η Intel που εισήγαγε πρόσφατα τις δικές της κάρτες γραφικών. Στην εργασία μας όλα τα προγράμματα θα εκτελούνται σε κάρτα γραφικών της NVIDIA.

NVIDIA GeForce RTX 3060 Specs:

- GPU Architecture: Ampere
- CUDA Cores: 3584
- Base Clock Speed: 1.32 GHz
- Boost Clock Speed: 1.78 GHz
- Memory: 12 GB GDDR6
- Memory Speed: 15 Gbps
- Ray Tracing Cores: 28
- Tensor Cores: 112 (used for AI-based tasks like DLSS)

3.1.1 Ιστορική Αναδρομή

Η ανάπτυξη της μονάδας επεξεργασίας γραφικών (GPU) είναι ένα ταξίδι που ξεκίνησε με την ανάγκη για καλύτερα γραφικά και έχει φτάσει σήμερα να έχουν εξελιχθεί σε πολυλειτουργικούς επεξεργαστές ικανούς να εκτελούν παράλληλους υπολογισμούς γενικής χρήσης για πεδία όπως η επιστημονική έρευνα και η βαθιά μάθηση. Ας δούμε κάποια σημεία στον χρόνο που καθόρισαν την ανάπτυξη τους σε αυτό που είναι σήμερα.

Πρώιμοι επιταχυντές γραφικών πριν από την δεκαετία του 1990

Τον πρώτο καιρό, η επεξεργασία γραφικών γινόταν εξ ολοκλήρου από την Κεντρική Μονάδα Επεξεργασίας (CPU). Οι πρώτοι αλγόριθμοι γραφικών σχεδιάστηκαν για την CPU. Η CPU δεν είναι όμως τόσο αποτελεσματική στην απόδοση πολύπλοκων οπτικών γραφικών και έχει περιορισμένες δυνάμεις στην απόδοση δυναμικού περιεχομένου όπως εικόνες, τρισδιάστατα μοντέλα και κινούμενα σχέδια.

Για την επίλυση αυτών των περιορισμών δημιουργήθηκαν οι πρώιμοι επιταχυντές γραφικών. Αυτές ήταν συσκευές υλικού που είχαν σχεδιαστεί για να απαλλάξουν την CPU από συγκεκριμένες εργασίες όπως η απόδοση 2D γραφικών. Η εστίαση ήταν κυρίως στη βελτίωση της απόδοσης σε απλές εφαρμογές γραφικών 2D, όπως προγράμματα χρηστών και παιχνίδια. Μία από τις πρώτες κάρτες γραφικών ήταν η Hercules Graphics Card (1982), η οποία χρησιμοποιήθηκε για μονόχρωμες οθόνες. Αργότερα τα πρότυπα EGA (Enhanced Graphics Adapter) και VGA (Video Graphics Array) (1987) έφεραν βελτιωμένη ανάλυση και βάθος χρώματος.

Η άνοδος των τρισδιάστατων γραφικών στην δεκαετία του 90.

Η ζήτηση για τρισδιάστατη απόδοση σε βιντεοπαιχνίδια και εικονικά περιβάλλοντα αυξήθηκε την δεκαετία του 90. Έτσι οι επιταχυντές γραφικών εξελίχθηκαν σε ισχυρές κάρτες γραφικών με δυνατότητες ανάπτυξης 3D γραφικών. Τα 3D βιντεοπαιχνίδια και η αυξανόμενη πολυπλοκότητα των εφαρμογών γραφικών σε τομείς όπως η παραγωγή ταινιών και το CAD (Computer-Aided Design) ώθησαν την ανάπτυξη των καρτών αυτών στο μέγιστο.

Οι πρώτοι επιταχυντές γραφικών εισήχθησαν στις αρχές της δεκαετίας του 90. Κατασκευάστηκαν εξειδικευμένα τσιπ γραφικών 3D, τα οποία χειρίζονταν εργασίες όπως η απόδοση πολυγώνων, η εφαρμογή υφών και ο υπολογισμός των εφέ φωτισμού. Αυτά τα

τσιπ πρόσφεραν πολύ υψηλότερη απόδοση από τις CPU εκείνη την εποχή, καθιστώντας εφικτή την απόδοση 3D σε πραγματικό χρόνο.

Ένας από τους πρώτους 3D επιταχυντές, που χρησιμοποιήθηκε ευρέως, ήταν η κάρτα γραφικών Voodoo από την εταιρία 3dfx Interactive, σημείωσε ένα σημαντικό άλμα στα γραφικά. Υποστήριξε τρισδιάστατη απόδοση σε πραγματικό χρόνο και χαρτογράφηση υφής, φέρνοντας επανάσταση στη βιομηχανία βιντεοπαιχνιδιών. Ωστόσο, η κάρτα Voodoo εστίαζε μόνο σε εργασίες 3D και οι χρήστες εξακολουθούσαν να χρειάζονταν μια ξεχωριστή κάρτα 2D για τη βασική λειτουργία της οθόνης.

Η NVIDIA το 1997 εξέδωσε για δεύτερη φορά την κάρτα γραφικών RIVA 128, η οποία πλέον υποστήριζε τόσο γραφικά 2D όσο και 3D. Αυτή η κάρτα γραφικών έθεσε τα θεμέλια για την μετέπειτα κυριαρχία της NVIDIA στην αγορά των GPU.

Ανάπτυξη σύγχρονων γραφικών τέλη δεκαετίας του 1990 με αρχές του 2000.

Σημείο καμπής στην ανάπτυξη της GPU ήταν η εισαγωγή προγραμματιζόμενων Shaders στα τέλη της δεκαετίας του 1990 και στις αρχές του 2000. Πριν από αυτό, οι GPU χρησιμοποιούσαν αγωγούς σταθερής λειτουργίας, όπου συγκεκριμένες γραφικές εργασίες (όπως μετασχηματισμοί κορυφής (vertex transformations) ή σκίαση εικονοστοιχείων) ήταν ενσωματωμένες στο υλικό. Τα προγραμματιζόμενα shaders επέτρεψαν στους προγραμματιστές να γράφουν προσαρμοσμένο κώδικα για να ελέγχουν τον τρόπο εκτέλεσης αυτών των εργασιών, παρέχοντας πολύ μεγαλύτερη ευελιξία και ισχύ.

Η GeForce 3 της NVIDIA (2001) ήταν η πρώτη κάρτα που υποστήριξε το DirectX 8.0, το οποίο περιλάμβανε σκιαδιστές (shaders) εικονοστοιχείων και σκιαγραφητές κορυφής (vertex shaders). Αυτό σηματοδότησε την αρχή μιας νέας εποχής στα παιχνίδια και τα οπτικά εφέ, όπου τα γραφικά δεν περιορίζονταν πλέον σε προκαθορισμένα εφέ και οι προγραμματιστές μπορούσαν να δημιουργήσουν μοναδικά οπτικά στυλ.

Η ATI, που αργότερα εξαγοράστηκε από την AMD, παρουσίασε την Radeon 8500, η οποία διέθετε επίσης προγραμματιζόμενους shaders και ήταν άμεσος ανταγωνιστής της GeForce 3 της NVIDIA.

Καθώς οι GPU έγιναν πιο προγραμματιζόμενες, ήταν πλέον σε θέση να χειρίζονται περίπλοκες γραφικές εργασίες, όπως φωτισμό σε πραγματικό χρόνο, χαρτογράφηση σκιών και χαρτογράφηση προσκρούσεων. Αυτή η δυνατότητα βελτίωσε σημαντικά τον

ρεαλισμό των παιχνιδιών και των οπτικών εφέ, επιτρέποντας πιο περίπλοκα και καθηλωτικά γραφικά.

Η εποχή των καρτών υπολογισμού γενικού σκοπού (GPGPU) μέσα δεκαετίας 2000.

Η στροφή προς την ανάπτυξη GPU για υπολογισμούς γενικού σκοπού ήρθε στα μέσα της δεκαετίας του 2000, όπου οι ερευνητές άρχισαν να συνειδητοποιούν ότι η μαζικά παράλληλη αρχιτεκτονική των GPU, η οποία είχε βελτιστοποιηθεί για γραφικά, θα μπορούσε επίσης να χρησιμοποιηθεί για υπολογισμούς γενικού σκοπού. Αυτό οδήγησε στην άνοδο του GPGPU (General-Purpose computing on GPUs), όπου οι GPUs χρησιμοποιήθηκαν για την επιτάχυνση εργασιών πέρα από τα παραδοσιακά γραφικά, όπως επιστημονικές προσομοιώσεις, κρυπτογραφία και μηχανική μάθηση.

Το 2006 η NVIDIA εισήγαγε την CUDA (Compute Unified Device Architecture), μια πλατφόρμα παράλληλων υπολογιστών και ένα μοντέλο προγραμματισμού που επέτρεπε στους προγραμματιστές να χρησιμοποιούν τις GPU για εφαρμογές που δεν αφορούν γραφικά. Αυτό άλλαξε τα δεδομένα, ανοίγοντας την πόρτα στη χρήση GPU σε τομείς όπως η ανάλυση δεδομένων, η τεχνητή νοημοσύνη και η εξόρυξη κρυπτονομισμάτων.

Από την άλλη η AMD το 2008 εισήγαγε την τεχνολογία Stream, η οποία επέτρεψε στους προγραμματιστές να χρησιμοποιούν τις GPU για υπολογισμούς γενικού σκοπού χρησιμοποιώντας το πλαίσιο OpenCL (Open Computing Language). Αυτό διεύρυνε περαιτέρω το πεδίο εφαρμογής των εφαρμογών GPU.

Οι κάρτες γραφικών του σήμερα.

Στην δεκαετία του 2010 ξεκίνησε η άνοδος της τεχνητής νοημοσύνης και της μηχανικής μάθησης. Η χρήση των GPUs για βαθιά μάθηση και τεχνητή νοημοσύνη έγινε ένα από τα καθοριστικά χαρακτηριστικά των σύγχρονων GPUs. Η εκπαίδευση νευρωνικών δικτύων απαιτεί τεράστια υπολογιστική ισχύ και η παράλληλη αρχιτεκτονική των GPU τις καθιστά ιδανικές για αυτό το έργο.

Το 2007 η σειρά καρτών Tesla λανσαρίστηκε από την NVIDIA για επιστημονικούς υπολογισμούς και φόρτους εργασίας TN, καθιστώντας τις GPU ευρέως αποδεκτές σε κέντρα δεδομένων και περιβάλλοντα έρευνας. Αργότερα εταιρείες όπως η Google, το Facebook και η Open-AI υιοθέτησαν τις GPU για την επιτάχυνση της έρευνας TN, και

πλαίσια όπως το TensorFlow και το PyTorch βελτιστοποιήθηκαν για να τρέχουν σε GPU, καθιστώντας τα την προτιμώμενη επιλογή για την εκπαίδευση μοντέλων βαθιάς μάθησης.

Ray Tracing σε πραγματικό χρόνο είναι επίσης ένα σημαντικό άλμα στις δυνατότητες της GPU το οποίο έγινε με την εισαγωγή της ανίχνευσης ακτίνων σε πραγματικό χρόνο. Το ray tracing προσομοιώνει τη συμπεριφορά φωτός σε περιβάλλοντα 3D με απίστευτο ρεαλιστικό, σκιές και αντανακλάσεις. Αυτή η τεχνολογία κατέστη δυνατή με την ανάπτυξη εξειδικευμένων ray tracing πυρήνων στην GPU. Η σειρά RTX της NVIDIA, βασισμένη στην αρχιτεκτονική Turing, εισήγαγε την τεχνολογία ray tracing σε πραγματικό χρόνο με την δημιουργία πυρήνων υλικού για την ανίχνευσης ακτίνων και πυρήνες Tensor. Αυτό επέτρεψε ένα νέο επίπεδο ρεαλισμού σε βιντεοπαιχνίδια, εικονικές προσομοιώσεις και τρισδιάστατη απεικόνιση.

Σήμερα, οι GPU δεν χρησιμοποιούνται μόνο για παιχνίδια και γραφικά, αλλά και για κλάδους όπως τα αυτόνομα οχήματα, η γονιδιωματική έρευνα και η χρηματοοικονομική μοντελοποίηση. Η δυνατότητα παράλληλης επεξεργασίας τεράστιων όγκων δεδομένων έχει καταστήσει τις GPU απαραίτητες για την επιστημονική έρευνα αιχμής και τις εμπορικές εφαρμογές.

Στον τομέα των κβαντικών υπολογισμών, μας επιτρέπουν την προσομοίωση κβαντικών συστημάτων και αλγορίθμων. Για παράδειγμα, η προσομοίωση ενός συστήματος με n qubits απαιτεί την επεξεργασία 2^n καταστάσεων, κάτι που είναι υπολογιστικά πολύ απαιτητικό. Οι GPU μπορούν να επιταχύνουν σημαντικά αυτή τη διαδικασία. Ακόμη βοηθούν στην δοκιμή για την σωστή λειτουργία προγραμμάτων πριν αυτά εκτελεσθούν σε κάποιον κβαντικό υπολογιστή, όπως αυτός που παρέχει στο κοινό η IBM. Τέλος, οι κβαντικοί υπολογιστές αναλαμβάνουν τις κβαντικές εργασίες σε ένα πρόγραμμα, αλλά ένα κομμάτι ανατίθενται στους κλασσικούς υπολογιστές.

3.1.2 GPU vs CPU

Ενώ τόσο η GPU όσο και η CPU είναι μονάδες επεξεργασίας σε ένα σύστημα υπολογιστή, έχουν σχεδιαστεί για διαφορετικούς σκοπούς και οι αρχιτεκτονικές τους το κάνουν αυτό εμφανές.

Όσο αφορά τις εργασίες που εκτελούν η CPU είναι ένας επεξεργαστής γενικής χρήσης που χειρίζεται μια μεγάλη ποικιλία εργασιών, από την εκτέλεση λειτουργικών συ-

στημάτων έως την εκτέλεση μεμονωμένων εντολών σε εφαρμογές. Είναι βελτιστοποιημένη για διαδοχική επεξεργασία (η μία εργασία διαδέχεται την άλλη), γεγονός που την καθιστά ιδανική για εργασίες που απαιτούν σύνθετη λογική και λήψη αποφάσεων. Από την άλλη η GPU εξειδικεύεται στην παράλληλη επεξεργασία, δηλαδή έχει σχεδιαστεί για να εκτελεί πολλές λειτουργίες ταυτόχρονα. Οι GPU υπερέχουν σε επαναλαμβανόμενες εργασίες που περιλαμβάνουν μεγάλα σύνολα δεδομένων, όπως η απόδοση εικονοστοιχείων (rendering pixels) ή η εκτέλεση πράξεων με πίνακες.

Όταν εστιάζουμε στην αρχιτεκτονική των δύο αυτών μονάδων επεξεργασίας βλέπουμε πως η CPU έχει έναν μικρό αριθμό πυρήνων. Κάθε πυρήνας είναι ισχυρός, αλλά σχεδιασμένος να χειρίζεται μία ή λίγες εργασίες ταυτόχρονα. Μια GPU περιέχει χιλιάδες μικρότερους, απλούστερους πυρήνες (συχνά εκατοντάδες ή χιλιάδες πυρήνες) που είναι βελτιστοποιημένοι για την εκτέλεση της ίδιας λειτουργίας σε πολλά σημεία δεδομένων ταυτόχρονα. Αυτοί οι πυρήνες είναι πιο αποδοτικοί για εργασίες που μπορούν να διασπαστούν σε μικρότερες, παράλληλες υποεργασίες.

Οι CPUs έχουν μία μικρή σε χωρητικότητα αλλά με μεγάλες ταχύτητες προσωρινή μνήμη (cache memory) και μία μεγαλύτερη κύρια μνήμη (RAM), την οποία μοιράζονται όλοι οι πυρήνες. Η μνήμη της CPU είναι βελτιστοποιημένη για γρήγορη πρόσβαση σε μεγάλη ποικιλία δεδομένων. Μια GPU διαθέτει τη δική της αποκλειστική μνήμη βίντεο (VRAM), η οποία είναι βελτιστοποιημένη για την αποθήκευση και τη γρήγορη πρόσβαση σε μεγάλα μεγέθη οπτικών δεδομένων, όπως υφές και buffers καρέ. Επιπλέον, οι GPU περιλαμβάνουν πολλαπλά επίπεδα κρυφής μνήμης που συμβάλλουν στην επιτάχυνση της πρόσβασης σε συχνά χρησιμοποιούμενα δεδομένα.

Οι CPUs λειτουργούν σε υψηλότερες ταχύτητες ρολογιού (μετρούμενες σε GHz), επιτρέποντάς τους να εκτελούν πιο σύνθετους υπολογισμούς ανά δευτερόλεπτο. Αυτό είναι ζωτικής σημασίας για εργασίες που απαιτούν γρήγορη λήψη αποφάσεων, όπως η εκτέλεση ενός λειτουργικού συστήματος ή η μεταγλώττιση κώδικα. Αντίθετα, οι GPU λειτουργούν συνήθως σε χαμηλότερες ταχύτητες ρολογιού, αλλά αντισταθμίζουν το μειονέκτημά τους με πολύ υψηλότερο βαθμό παραλληλισμού. Ενώ οι μεμονωμένοι πυρήνες της GPU είναι πιο αργοί από τους πυρήνες της CPU, ο απόλυτος αριθμός των πυρήνων επιτρέπει στις GPU να ξεπερνούν τις CPU σε συγκεκριμένες παράλληλες εργασίες.

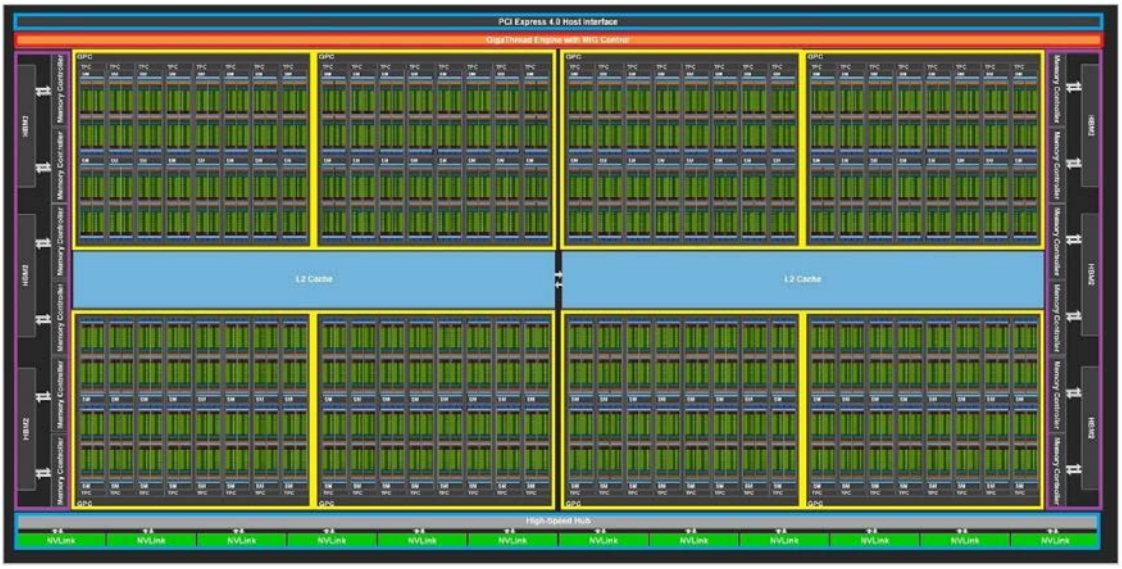
3.1.3 Βασική Αρχιτεκτονική

Η αρχιτεκτονική μιας Μονάδας Επεξεργασίας Γραφικών (GPU) είναι ιδιαίτερα εξειδικευμένη για να διαχειρίζεται παράλληλα φορτία εργασίας και ταυτόχρονα τεράστιες ποσότητες δεδομένων. Ας δούμε κάποια βασικά στοιχεία της αρχιτεκτονικής τους.

Πυρήνες (Cores)

Οι πυρήνες (cores) είναι η καρδιά της GPU. Είναι το πιο θεμελιώδες στοιχείο της αρχιτεκτονικής μια κάρτας γραφικών. Πρόκειται για μικρότερες μονάδες επεξεργασίας που διεκπεραιώνουν συγκεκριμένες εργασίες και συνεργάζονται παράλληλα για την επίτευξη υψηλής υπολογιστικής απόδοσης. Ο μεγάλος αριθμός πυρήνων επιτρέπει στις GPU να εκτελούν πολλές εργασίες ταυτόχρονα, καθιστώντας τις εξαιρετικά ισχυρές.

Οι GPU περιέχουν εκατοντάδες έως χιλιάδες πυρήνες, οργανωμένους σε ομάδες γνωστές ως Streaming Multiprocessors (SMs) ή Compute Units (CUs), ανάλογα με τον κατασκευαστή (η NVIDIA χρησιμοποιεί SMs, η AMD χρησιμοποιεί CUs). Κάθε πυρήνας είναι σχετικά απλός σε σύγκριση με έναν πυρήνα ΚΜΕ, αλλά σχεδιασμένος να εκτελεί παράλληλα πολλές εντολές. Μια μεμονωμένη GPU μπορεί να περιέχει αρκετές εκατοντάδες έως αρκετές χιλιάδες πυρήνες. Για παράδειγμα, η κάρτα γραφικών που χρησιμοποιούμε για τα προγράμματα της εργασίας είναι η GeForce RTX 3060 της NVIDIA με αρχιτεκτονική Ampere διαθέτει 3584 πυρήνες.



Εικόνα 13: Μία πλήρης GA100 GPU με 128 SMs (https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth)

Το παραπάνω διάγραμμα απεικονίζει την ampere αρχιτεκτονική μια GPU. Στο κέντρο τα κουτιά που περιβάλλονται με κίτρινο χρώμα είναι οι πολυεπεξεργαστές, αποτελούν την καρδιά της GPU καθώς εκεί γίνονται όλοι οι υπολογισμοί. Streaming multiprocessors SM

Στην αριστερή και δεξιά πλευρά με μοβ χρώμα βρίσκονται οι ελεγκτές μνήμης που βοηθούν να μετατοπίζουν και να αποθηκεύουν δεδομένα σε διάφορους τύπους μνήμης σε όλη την GPU. Ο Giga Thread Engine ψηλά στο διάγραμμα με κόκκινο είναι ο κύριος χρονοπρογραμματιστής των νημάτων.

Πάνω από τον giga thread engine με γαλάζιο και τέρμα κάτω στο σχεδιάγραμμα βρίσκονται οι διεπαφές. Ψηλά είναι ο γενικός δίαυλος IO, ο οποίος είναι κοινός μεταξύ της CPU και της GPU ώστε τα δεδομένα να μπορούν εύκολα να μεταφερθούν από την μία στην άλλη. Η κάτω διεπαφές είναι οι NVLink και high-speed link οι οποίοι είναι σύνδεσμοι υψηλής ταχύτητας και χρησιμοποιούνται για να κάνουν ταχύτερη μετάδοση δεδομένων μεταξύ των GPU.

Κάθε πυρήνας GPU λειτουργεί σε πολλαπλά κομμάτια δεδομένων ταυτόχρονα. Αυτό είναι γνωστό ως επεξεργασία SIMD (Single Instruction, Multiple Data), όπου η ίδια λειτουργία εφαρμόζεται σε διαφορετικά στοιχεία δεδομένων. Αυτό είναι ιδανικό για εργασίες όπως η απόδοση εικονοστοιχείων, όπου η ίδια λειτουργία (π.χ. υπολογισμός χρώματος ή φωτισμού) επαναλαμβάνεται για πολλά pixels.

Πιο πρόσφατα, οι σύγχρονες GPU χρησιμοποιούν την εκτέλεση SIMT (Single Instruction, Multiple Threads), η οποία αποτελεί επέκταση της SIMD. Αυτό επιτρέπει ακόμα μεγαλύτερο παραλληλισμό στις GPU, όπου πολλά νήματα μπορούν να εκτελέσουν την ίδια εντολή ταυτόχρονα σε πολλούς πυρήνες. Θα δούμε στην συνέχεια τι είναι τα νήματα.

Threads, blocks και warps

Τι είναι τα νήματα (threads), τα μπλοκ (blocks) και τα warps. Στις GPU της NVIDIA, οι πυρήνες ομαδοποιούνται σε πολυεπεξεργαστές ροής (Streaming Multiprocessors, SM) και τα νήματα εντός ενός SM οργανώνονται σε warps (συνήθως 32 νήματα). Κάθε warp εκτελεί παράλληλα την ίδια εντολή σε διαφορετικά σημεία δεδομένων, εξασφαλίζοντας τη μέγιστη παράλληλη απόδοση.

Μέσα σε κάθε πυρήνα, υπάρχουν μονάδες εκτέλεσης που χειρίζονται τους πραγματικούς υπολογισμούς, όπως πράξεις κινητής υποδιαστολής (που χρησιμοποιούνται στην απόδοση γραφικών και σε επιστημονικούς υπολογισμούς) και πράξεις ακεραίων (που

χρησιμοποιούνται στην τεχνητή νοημοσύνη και σε υπολογισμούς γενικού σκοπού). Οι GPU μπορεί επίσης να διαθέτουν εξειδικευμένους πυρήνες για εργασίες όπως η εξαγωγή συμπερασμάτων TN (Tensor Cores για την NVIDIA, Matrix Cores για την AMD).

Στην CUDA ένα warp είναι μια πολύ βασική έννοια που σχετίζεται με τον τρόπο εκτέλεσης των νημάτων. Ένα warp είναι μια ομάδα 32 νημάτων που εκτελούν την ίδια εντολή ταυτόχρονα σε έναν πυρήνα CUDA. Τα warps διαχειρίζονται από τον προγραμματιστή υλικού(scheduler) της GPU.

Όλα τα νήματα σε ένα warp εκτελούν την ίδια εντολή ταυτόχρονα. Αν τα νήματα σε ένα warp αποκλίνουν (π.χ., λόγω συνθηκών όπως if-else), η απόδοση μπορεί να υποβαθμιστεί λόγω warp divergence (απόκλιση warp). Όταν συμβαίνει απόκλιση, η GPU εκτελεί και τις δύο διαδρομές διαδοχικά, απενεργοποιώντας τα νήματα που δεν ακολουθούν την τρέχουσα διαδρομή. Αυτό μειώνει τον παραλληλισμό και την απόδοση.

Οι προσβάσεις μνήμης είναι πιο αποδοτικές όταν τα νήματα σε μια warp προελαύνουν διαδοχικές θέσεις μνήμης. Αυτό ονομάζεται coalesced memory access και θα το δούμε αναλυτικά στην συνέχεια. Μη συγχωνευμένες προσβάσεις μπορούν να μειώσουν σημαντικά την απόδοση.

Εικόνα 14: Ένας streaming multiprocessor (SM) (https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth)

Στο σχήμα αυτό βλέπουμε έναν streaming multiprocessor που χωρίζεται σε τέσσερα warps. Κάθε warp περιέχει μία κρυφή μνήμη L0, έναν χρονοδρομολογητή και μία συλλογή καταχωρητών που ονομάζεται αρχείο καταχωρητών και στη συνέχεια δύο μισά warps το ένα με τη δυνατότητα να κάνει πράξεις κινητής υποδιαστολής ή ακέραιες πράξεις 32-bit. Ένα αποκλειστικά αφιερωμένο σε πράξεις κινητής υποδιαστολής 32 bit. Υπάρχουν δυνατότητες φόρτωσης και αποθήκευσης. Υπάρχει ένας πυρήνας τανυστών που είναι ειδικός για το warp. Υπάρχουν και άλλες δυνατότητες κοινής χρήσης με τη χρήση του αποθηκευτικού χώρου δεδομένων L1 ως κοινή μνήμη. Στη συνέχεια, υπάρχουν τέσσερις πυρήνες υφής και ένας μοναδικός πυρήνας ανίχνευσης ακτίνων, οι οποίοι και οι δύο είναι αφιερωμένοι στην επεξεργασία γραφικών.

Μνήμη στην GPU

Η μνήμη είναι ένα άλλο κρίσιμο στοιχείο της αρχιτεκτονικής της GPU. Οι GPUs απαιτούν μνήμη υψηλής ταχύτητας για να τροφοδοτούν γρήγορα τους πυρήνες με δεδομένα και να αποθηκεύουν τις τεράστιες ποσότητες δεδομένων που απαιτούνται. Το σύστημα μνήμης της GPU διαφέρει από τη μνήμη της CPU, καθώς είναι βελτιστοποιημένο για διαφορετικές περιπτώσεις χρήσης.

Οι GPUs χρησιμοποιούν τη μνήμη Video RAM (VRAM) για την αποθήκευση των δεδομένων που πρέπει να υποβληθούν σε επεξεργασία. Η VRAM είναι βελτιστοποιημένη για πρόσβαση υψηλού εύρους ζώνης (high-bandwidth access), η οποία είναι πολύ σημαντικό κατά την επεξεργασία.

Η μνήμη GDDR (Graphics Double Data Rate) χρησιμοποιείται συνήθως στις σύγχρονες GPU, όπως GDDR5, GDDR6 ή GDDR6X. Είναι μνήμη υψηλής ταχύτητας σχεδιασμένη για φόρτο που παρέχει η επεξεργασία γραφικών. Επιτρέπει στην GPU να διαβάζει και να γράφει πολύ γρήγορα μεγάλο όγκο δεδομένων.

Οι GPU χρησιμοποιούν συνήθως μια ιεραρχία μνήμης για να μειώσουν την καθυστέρηση και να μεγιστοποιήσουν την απόδοση. Αυτό σημαίνει διαφορετικά επίπεδα μνήμης, όπως οι κρυφές μνήμες L1 και L2, οι οποίες αποθηκεύουν δεδομένα με συχνή πρόσβαση, επιτρέποντας στους πυρήνες να ανακτούν δεδομένα ταχύτερα χωρίς να χρειάζεται να έχουν πρόσβαση στην πιο αργή VRAM.

Σε ορισμένες αρχιτεκτονικές GPU, η κοινόχρηστη μνήμη είναι διαθέσιμη εντός μιας SM και χρησιμοποιείται για επικοινωνία μεταξύ των νημάτων και προσωρινή αποθήκευση. Αυτό είναι πολύ πιο γρήγορο από την πρόσβαση στην VRAM και συμβάλλει στην επιτάχυνση των παράλληλων υπολογισμών, ιδίως σε πολύπλοκους αλγορίθμους.

Οι GPU διαθέτουν ειδικές κρυφές μνήμες σε πολλαπλά επίπεδα:

L1 Cache: Γρήγορες, μικρές κρυφές μνήμες που βρίσκονται κοντά στους πυρήνες για γρήγορη πρόσβαση σε συχνά χρησιμοποιούμενα δεδομένα.

L2 Cache: Μια μεγαλύτερη, πιο αργή κρυφή μνήμη που αποθηκεύει δεδομένα που μοιράζονται σε πολλές SMs/CUs, μειώνοντας την ανάγκη για άμεση πρόσβαση στη μνήμη.

Αγωγοί (Pipelines)

Τα pipelines στις GPUs (Graphics Processing Units) είναι ένα θεμελιώδες στοιχείο της αρχιτεκτονικής τους και παίζουν κρίσιμο ρόλο στην απόδοση και την αποτελεσματικότητα της επεξεργασίας γραφικών και άλλων εργασιών. είναι μια σειρά από στάδια (stages) που χρησιμοποιούνται για την επεξεργασία δεδομένων σε μια GPU. Κάθε στάδιο εκτελεί μια συγκεκριμένη εργασία, και τα δεδομένα περνούν διαδοχικά από όλα τα στάδια για να παραχθεί το τελικό αποτέλεσμα. Αυτή η διαδικασία επιτρέπει την παράλληλη επεξεργασία, καθώς πολλά δεδομένα μπορούν να βρίσκονται σε διαφορετικά στάδια του pipeline ταυτόχρονα.

3.2 CUDA

Η CUDA (Compute Unified Device Architecture) είναι μια πλατφόρμα παράλληλων υπολογισμών και ένα μοντέλο προγραμματισμού που αναπτύχθηκε από την NVIDIA, το οποίο επιτρέπει στους προγραμματιστές να αξιοποιήσουν την ισχύ των μονάδων επεξεργασίας γραφικών (GPUs) για γενικούς υπολογισμούς (GPGPU).

Πριν την ανάπτυξη της CUDA, ο προγραμματισμός της GPU ήταν μια σχετικά δύσκολο διαδικασία. Οι προγραμματιστές έπρεπε να χρησιμοποιούν APIs ειδικά σχεδιασμένα για γραφικά, όπως το OpenGL ή το DirectCompute, για να εκτελέσουν γενικούς υπολογισμούς. Αυτά τα APIs ήταν σχεδιασμένα κυρίως για απόδοση γραφικών και όχι για ευέλικτη παράλληλη επεξεργασία.

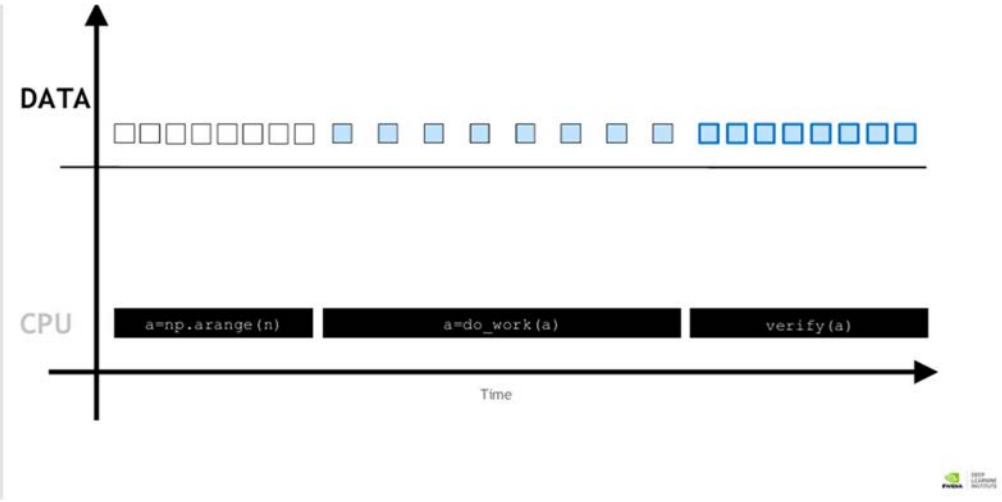
Η CUDA εισήχθη το 2006 από την NVIDIA με σκοπό να απλοποιήσει τον προγραμματισμό για GPU και να ξεκλειδώσει τις πλήρεις δυνατότητες επιτάχυνσης που προσφέρουν οι GPUs. Μας παρέχει μια διεπαφή προγραμματισμού τύπου C, την οποία χρησιμοποιούμε για δημιουργία προγραμμάτων που εκτελούνται παράλληλα, χρησιμοποιώντας γλώσσες όπως C, C++, Python και Fortran. Μας δίνει άμεση πρόσβαση στο υλικό και σε αντίθεση με προηγούμενες μεθόδους, η CUDA μας επιτρέπει την άμεση διαχείριση της μνήμης στην GPU, των νημάτων και της εκτέλεσης.

Η NVIDIA με την CUDA μετέτρεψε τις ήδη δυνατές κάρτες γραφικών σε ισχυρούς παράλληλους επεξεργαστές ανοίγοντας τον δρόμο στην ανάπτυξη τομέων όπως η τεχνητή νοημοσύνη.

Επισκόπηση της λειτουργίας προγραμμάτων για την GPU.

Θα πρέπει να πούμε πως στην ορολογία της CUDA η CPU είναι ο υποδοχέας (host) ενώ η GPU είναι η συσκευή (device). Για παράδειγμα, όταν λέμε ότι μεταφέρουμε τα δεδομένα από την συσκευή στον υποδοχέα εννοούμε πως μεταφέρουμε τα δεδομένα μας από την GPU στην CPU.

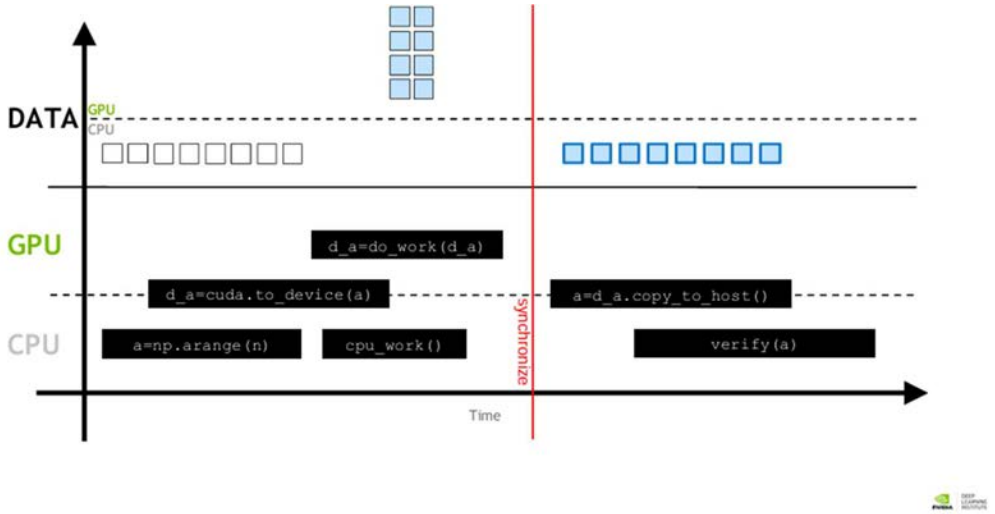
Ας δούμε, για αρχή τα βήματα που ακολουθούν οι εφαρμογές που εκτελούνται μόνο από την CPU.



Εικόνα 15: Εκτέλεση προγράμματος στην CPU (https://www.slideshare.net/JoshWyatt5).

Βλέπουμε στην εικόνα πως τα δεδομένα αποθηκεύονται στην CPU και όλοι οι υπολογισμοί εκτελούνται σειριακά.

Όμως στην περίπτωση εφαρμογών σχεδιασμένες να εκτελούνται παράλληλα στην GPU ο τρόπος εκτέλεσης τους αλλάζει.



Εικόνα 16: Εκτέλεση προγράμματος στην GPU (https://www.slideshare.net/JoshWyatt5).

Αν παρατηρήσουμε το σχήμα βλέπουμε πως για την εκτέλεση του προγράμματος χρειαζόμαστε τόσο τον υποδοχέα(host) όσο και την συσκευή(device) και κάθε ένα από αυτά έχει την δική του ξεχωριστή μνήμη.

Αρχικά, τα δεδομένα αποθηκεύονται στην CPU και στην συνέχεια μεταφέρονται στην GPU για να εργαστούμε παράλληλα.

Η GPU λειτουργεί ασύγχρονα από την κεντρική μονάδα επεξεργασίας. Αυτό πρακτικά σημαίνει πως η GPU εκτελεί εργασίες και υπολογισμούς ανεξάρτητα της CPU, η οποία δεν μένει ανενεργή μέχρι την ολοκλήρωση των εργασιών από την GPU αλλά εκτελεί κανονικά άλλες διεργασίες. Αυτός ο τρόπος λειτουργίας επιτρέπει την ταυτόχρονη χρήση και των δύο μονάδων επεξεργασίας βελτιώνοντας έτσι την αποδοτικότητα του συστήματος.

Βέβαια, αν κριθεί απαραίτητο μπορούμε να ορίσουμε σημεία συγχρονισμού μεταξύ τους. Στην Numba όπως θα δούμε παρακάτω καλούμε την συνάρτηση cuda.synchronize(). Αυτή η συνάρτηση εξασφαλίζει πως έχουν ολοκληρωθεί όλες οι εργασίας που ανατέθηκαν στην GPU πριν επιστρέψουμε στην CPU για την συνέχεια της εκτέλεσης.

Τέλος, τα αποτελέσματα τον υπολογισμών αντιγράφονται πίσω στην CPU.

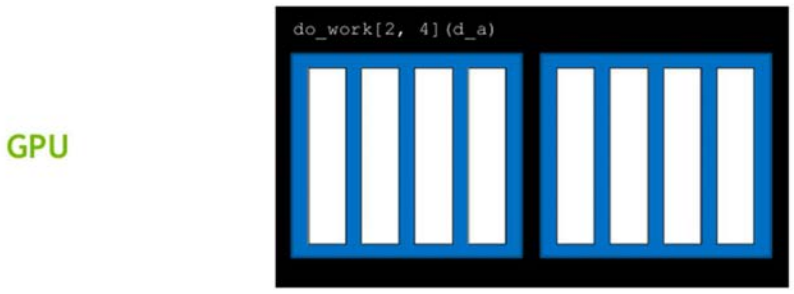
3.2.1 Μοντέλο προγραμματισμού με CUDA

Η CUDA ακολουθεί ένα μοντέλο εκτέλεσης SIMT (Single Instruction, Multiple Threads), όπου χιλιάδες νήματα εκτελούν το ίδιο πρόγραμμα αλλά λειτουργούν σε διαφορετικά τμήματα δεδομένων. Για την αποτελεσματική αξιοποίηση του παραλληλισμού της GPU, η CUDA παρέχει έναν δομημένο τρόπο οργάνωσης των υπολογισμών με τη χρήση πλεγμάτων, μπλοκ και νημάτων.

Να σημειώσουμε εδώ, αν και θα το δούμε και παρακάτω στην ανάπτυξη κώδικα για την GPU, πως η CUDA δεν κάνει ολόκληρα προγράμματα να εκτελούνται στην GPU αλλά συναρτήσεις. Οι συναρτήσεις αυτές ονομάζονται πυρήνες(kernels) και διαφέρουν από τους πυρήνες (cores) που έχει η GPU.

Ιεραρχία νημάτων: Πλέγμα, μπλοκ και νήματα.

Η CUDA οργανώνει τους υπολογισμούς με βάση μία ιεραρχία, στην οποία περιέχονται νήματα(threads), μπλοκ(blocks) και πλέγματα(grids). Ενώ οι GPU έχουν πολλά threads, blocks και grids, ας απλοποιήσουμε τα πράγματα με έναν μικρότερο αριθμό ώστε να καταλάβουμε καλύτερα.



Εικόνα 17: Ιεραρχία (πλέγμα, μπλοκ, νήματα) (https://www.slideshare.net/JoshWyatt5).

Η εκτέλεση εντολών στην GPU γίνεται από τα νήματα. Τα νήματα είναι η μικρότερη μονάδα επεξεργασίας. Κάθε νήμα εκτελεί τον ίδιο πυρήνα αλλά με διαφορετικά δεδομένα. Τα λευκά κουτιά στο κέντρο είναι τα threads και πολλά threads εκτελούνται παράλληλα. Η CUDA μπορεί να κάνει χρήση χιλιάδων threads ταυτόχρονα.

Μια ομάδα από threads αποτελούν ένα block. Στην εικόνα μας τα μπλοκ είναι τα μπλε κουτιά. Κάθε μπλοκ αποτελείτε από τέσσερα νήματα. Μπορούμε να έχουμε πολλά μπλοκ, αλλά στην περίπτωσή μας έχουμε δύο. Συνήθως, μια ομάδα από blocks είναι αυτή που εκτελεί έναν πυρήνα (kernel) και η ομάδα αυτών αποτελούν ένα πλέγμα (grid). Στην εικόνα είναι ολόκληρο το μαύρο κουτί.

Όπως είπαμε και παραπάνω οι συναρτήσεις που θα εκτελεσθούν ονομάζονται πυρήνες (kernels). Στην εικόνα ο πυρήνας αυτός είναι do_work. Οι πυρήνες για την εκτέλεσή τους χρειάζονται κάποιες παραμέτρους οι οποίες ορίζουν τον αριθμό των μπλοκ, που θέλουμε να εκτελέσουν τον πυρήνα, μέσα στο πλέγμα [**2**, 4]. Καθώς και τον αριθμό των νημάτων σε κάθε μπλοκ [2, **4**]. Κάθε μπλοκ μέσα στο πλέγμα θα περιέχει τον ίδιο αριθμό νημάτων.

Μεταβλητές ιεραρχίας νημάτων που παρέχονται από την CUDA

Όταν εκτελούμε έναν πυρήνα πρέπει να ορίσουμε τα νήματα, τα μπλοκ και τα πλέγματα που θα χρειαστούμε. Η CUDA μας παρέχει έτοιμες μεταβλητές για τον προσδιορισμό των θέσεων των νημάτων.

- gridDim.x: είναι ο αριθμός των μπλοκ μέσα σε ένα πλέγμα.

Στο παραπάνω παράδειγμα το gridDim.x είναι δύο. Αφού μέσα στο πλέγμα έχουμε δύο μπλοκ.

- blockIdx.x: είναι ένας δείκτης που ορίζει το τρέχον μπλοκ μέσα στο πλέγμα.

Εμείς έχουμε δύο μπλοκ. Το πρώτο θα έχει blockIdx.x μηδέν ενώ το δεύτερο τον δείκτη 1.

- blockDim.x: περιγράφει τον αριθμό των νημάτων μέσα σε ένα μπλοκ.

Στην περίπτωσή μας έχουμε τέσσερα.

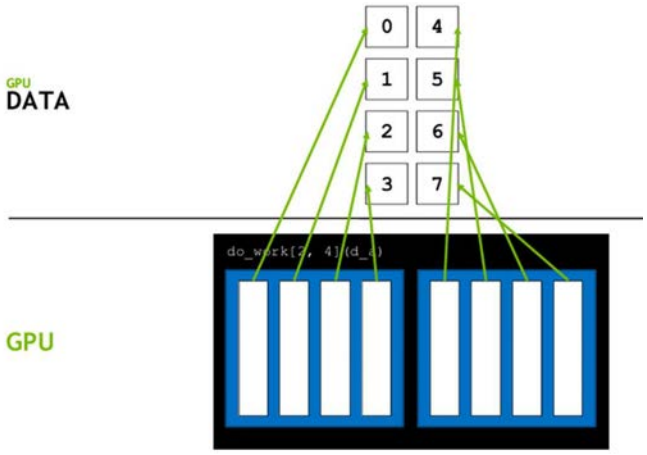
- threadIdx.x είναι ένας δείκτης που ορίζει ένα νήμα μέσα σε ένα μπλοκ.

Κάθε νήμα έχει ένα μοναδικό αναγνωριστικό νήματος. Για παράδειγμα, παραπάνω, το πρώτο μπλοκ με δείκτη 0, έχει 4 νήματα. Το πρώτο νήμα θα έχει δείκτη 0, το δεύτερο 1, το τρίτο 2 και το τέταρτο 3. Στο δεύτερο μπλοκ ο δείκτης των νημάτων

θα ξεκινήσει από την αρχή. Άρα στο μπλοκ με δείκτη 1, θα έχουμε τα νήματα με δείκτες 0,1,2,3 αντίστοιχα.

Συντονισμός παράλληλων νημάτων

Ας υποθέσουμε πως τα δεδομένα μας βρίσκονται αποθηκευμένα στην GPU σε μορφή διανύσματος με δείκτη 0. Για να εκτελεσθεί το πρόγραμμα σωστά πρέπει κάθε νήμα να εργάζεται σε διαφορετικό στοιχείο στα δεδομένα(Εικόνα 10).



Εικόνα 18: Αντιστοιχία νημάτων με τα δεδομένα (https://www.slideshare.net/JoshWyatt5).

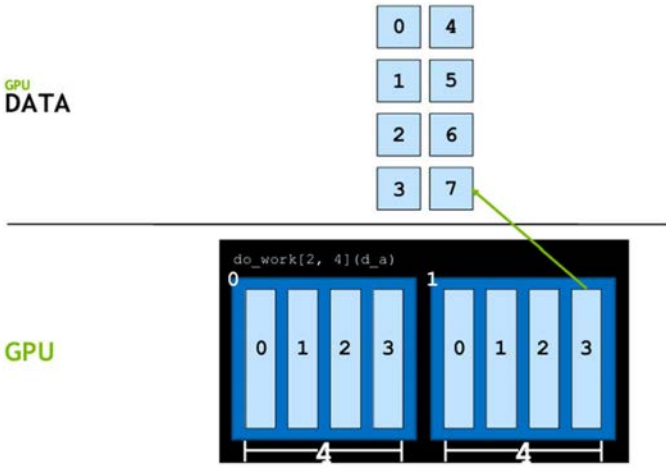
Αν μπορέσουμε να υπολογίσουμε τον δείκτη ενός νήματος, μέσα στο πλέγμα, τότε μπορούμε να αντιστοιχήσουμε το νήμα με τον δείκτη του στα δεδομένα με αντίστοιχο δείκτη. Με βάση τις μεταβλητές ιεραρχίας που μας παρέχει η CUDA μπορούμε να υπολογίσουμε τον δείκτη του νήματος. Ο τύπος για τον υπολογισμό αυτό είναι:

threadIdx.x + blockIdx.x * blockDim.x

Για παράδειγμα ας βρούμε τον δείκτη του τέταρτου νήματος από το δεύτερο μπλοκ(Εικόνα 11). Έχουμε:

threadIdx.x + blockIdx.x * blockDim.x

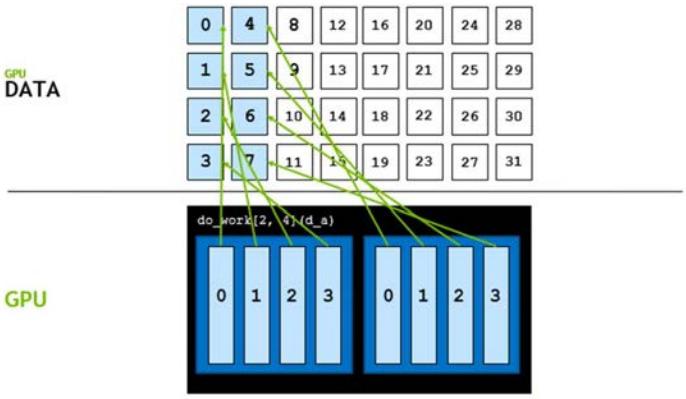
3 + 1 * 4 = 7



Εικόνα 19: Αντιστοίχιση νήματος με δείκτη δεδομένου (https://www.slideshare.net/JoshWyatt5).

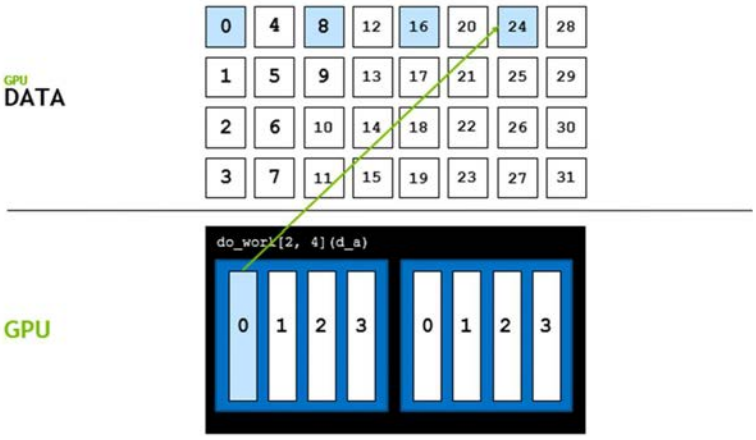
Grid Stride Loops

Είδαμε πως αντιστοιχούμε τα νήματα με τα δεδομένα. Όμως το πιο πιθανό είναι πως τα δεδομένα μας θα είναι πολύ περισσότερα από τον αριθμό των νημάτων μέσα στο πλέγμα(Εικόνα 12). Σε αυτή την περίπτωση κάθε νήμα πρέπει να εκτελεσθεί με παραπάνω από ένα στοιχεία δεδομένων. Αλλιώς η διεργασία δεν θα ολοκληρωθεί. Στην εικόνα 12 τα νήματα θα εκτελέσουν την διεργασία για τα δεδομένα με δείκτες 0 έως 7, όμως τα υπόλοιπα μέχρι το 31 θα μείνουν αχρησιμοποίητα.



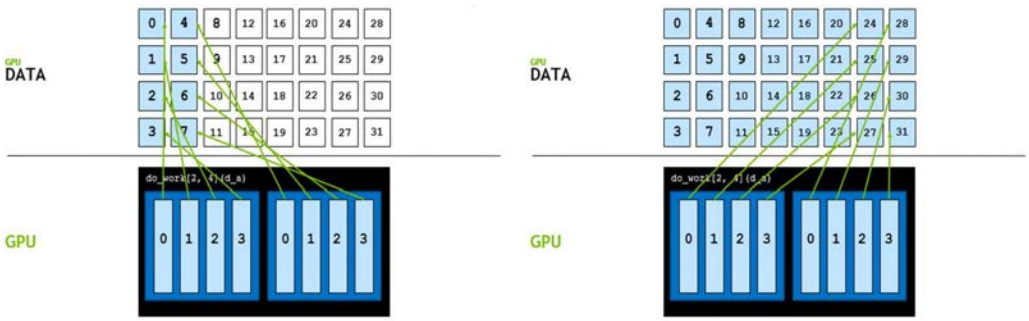
Εικόνα 20: Αντιστοίχιση νημάτων σε μεγάλο όγκο δεδομένων (https://www.slideshare.net/JoshWyatt5).

Ένας τρόπος να αντιμετωπισθεί αυτό προγραμματιστικά είναι με έναν βρόγχο. Μέσα σε αυτόν τον βρόγχο θα υπολογίσουμε το στοιχείο για κάθε νήμα όπως και πριν. Θα υπολογίσουμε αρχικά το πρώτο νήμα με τον τύπο που δώσαμε παραπάνω. Άρα το πρώτο νήμα στο πρώτο μπλοκ θα πάει στο στοιχείο 0 στα δεδομένα. Όμως δεν θα μείνει εκεί. Θα μεταπηδήσει στο επόμενο στοιχείο μετά τον αριθμό του συνόλου των νημάτων. Έτσι, λοιπόν θα πάει στο στοιχείο με δείκτη 8. Θα συνεχίσει με τον ίδιο τρόπο, στο στοιχείο με δείκτη 16 και μετά 24, μέχρι ο δείκτης του στοιχείου να είναι μεγαλύτερος από τον αριθμό των δεδομένων. Οπότε μετά το 24 θα υπολογίσει 32 το οποίο είναι εκτός του συνόλου των δεδομένων μας(Εικόνα 13).



Εικόνα 21: Αντιστοίχιση νημάτων μέχρι την χρήση όλων των δεδομένων (https://www.slideshare.net/JoshWyatt5).

Γνωρίζουνε πως όλα τα νήματα εργάζονται παράλληλα. Αν, λοιπόν εφαρμόσουμε τον ίδιο βρόγχο σε όλα τότε όλα τα δεδομένα θα χρησιμοποιηθούν(Εικόνα 14). Επιπλέον, η συσκευή συγκεντρώνει τις αναγνώσεις/εγγραφές μνήμης σε όσο το δυνατόν λιγότερες συναλλαγές για καλύτερες επιδόσεις. Με όλα τα νήματα να εργάζονται με αυτόν τον τρόπο, όλα τα στοιχεία καλύπτονται με το πλεονέκτημα της απόδοσης που προσφέρει η συνένωση μνήμης.



Εικόνα 22: Χρήση όλων των δεδομένων (https://www.slideshare.net/JoshWyatt5).

3.2.2 Μοντέλο μνήμης στην CUDA

Η αρχιτεκτονική που ακολουθεί η CUDA όσον αφορά τις μνήμες είναι ιεραρχική. Αυτό σημαίνει ότι διαφορετικοί τύποι μνήμης υπάρχουν σε διαφορετικά επίπεδα της GPU.

Καθολική μνήμη(Global Memory)

Η καθολική μνήμη είναι ο μεγαλύτερος σε μέγεθος τύπος μνήμης στον οποίο έχουμε πρόσβαση. Είναι η πιο ευέλικτη μνήμη αλλά η πιο αργή όσον αφορά την ταχύτητα πρόσβασης. Είναι η κύρια μνήμη στην GPU και μοιράζεται μεταξύ όλων των νημάτων σε όλα τα μπλοκ. Άρα όλα τα νήματα, ανεξάρτητα από το που βρίσκονται έχουν πρόσβαση σε αυτόν τον τύπο μνήμης. Βρίσκεται στην μνήμη DRAM (device memory) η οποία βρίσκεται σε ξεχωριστό σημείο από τους πυρήνες της GPU.

Δεδομένου ότι η καθολική μνήμη είναι αργή, η αναποτελεσματική πρόσβαση σε αυτήν μπορεί να εμποδίσει την απόδοση της εφαρμογής μας. Μία από τις στρατηγικές που μπορούμε να ακολουθήσουμε για την βελτίωση της απόδοσης χρήσης της καθολικής μνήμης είναι η συνένωση μνήμης (Memory Coalescing), κατά την οποία έχουμε πρόσβαση στα δεδομένα με βάση κάποιο μοτίβο.

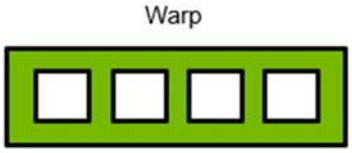
Memory Coalescing

Η συγχώνευση μνήμης (memory coalescing) είναι μια τεχνική που χρησιμοποιείται στο CUDA για τη βελτιστοποίηση της πρόσβασης στην καθολική μνήμη (global memory), διασφαλίζοντας ότι πολλαπλά νήματα προσπελαύνουν συνεχόμενες θέσεις

μνήμης με αποδοτικό τρόπο. Όταν οι προσβάσεις μνήμης είναι συγχωνευμένες, συνδυάζονται σε μια μοναδική συναλλαγή μνήμης, βελτιώνοντας σημαντικά την απόδοση.

Είδαμε, όταν αναφερθήκαμε στην GPU, πως η GPU προγραμματίζει warps σε ομάδες που ονομάζονται thread blocks (μπλοκ νημάτων). Κάθε thread block χωρίζεται σε warps, και ο scheduler αναθέτει warps στους streaming multiprocessors (SMs) για εκτέλεση. Το μέγεθος ενός warp είναι 32 νήματα σε όλες τις τρέχουσες GPUs της NVIDIA. Αυτό είναι ένα σταθερό χαρακτηριστικό του υλικού και δεν μπορεί να αλλάξει. Τα 32 νήματα που αποτελούν το warp εκτελούν παράλληλα την ίδια εντολή.

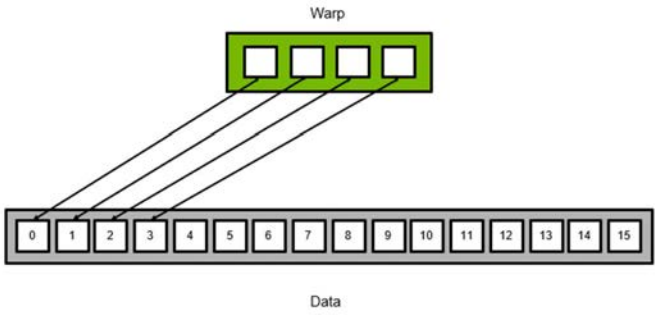
Για να καταλάβουμε καλύτερα πως τα νήματα μέσα στο warp έχουν πρόσβαση στην μνήμη ας θεωρήσουμε πως ένα warp αποτελείτε από 4 νήματα(Εικόνα 15).



Εικόνα 23: Παράδειγμα ενός warp (https://www.slideshare.net/JoshWyatt5).

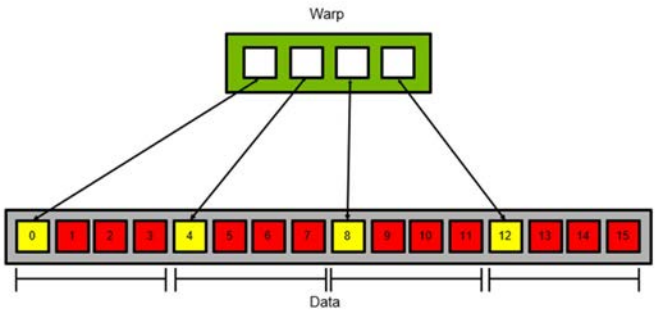
Τα δεδομένα μεταφέρονται από και προς την καθολική μνήμη σε κομμάτια των 32-byte (Στην L1 cache μνήμη μεταφέρονται σε τμήματα των 128-byte).

Σε μία γραμμή σταθερού μήκους συνεχόμενης μνήμης θα διαχειριζόμαστε κάθε φορά 4 στοιχεία. Το υποσύστημα μνήμης θα προσπαθήσει να ελαχιστοποιήσει τον αριθμό των συναλλαγών που απαιτούνται για την εκπλήρωση των απαιτήσεων ανάγνωσης/εγγραφής της warp. Αν οι διευθύνσεις των δεδομένων είναι συνεχόμενες τότε όλα τα δεδομένα στην σειρά θα χρησιμοποιηθούν και όλες οι αντιγραφές θα πραγματοποιηθούν με τον ελάχιστο αριθμό γραμμών. Όταν αυτό συμβαίνει τότε η πρόσβαση στην μνήμη είναι coalesced (συνεχής)(Εικόνα 16).



Εικόνα 24: Coalesced Memory (https://www.slideshare.net/JoshWyatt5).

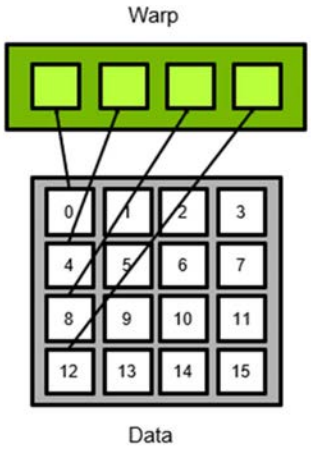
Όμως τα δεδομένα μας στην πραγματικότητα δεν είναι σχεδόν ποτέ στην σειρά και τα νήματα συχνά προσπελαύνουν μη συνεχόμενες ή τυχαίες θέσεις μνήμης. Καθώς η ζητούμενη μνήμη γίνεται λιγότερο συνεχής, τόσο θα χρειάζεται να μεταφέρουμε μεγαλύτερο μέρος των δεδομένων μας για να καλύψουμε τις ανάγκες του warp. Αλλά αυτό συνεπάγεται ότι μέρος των δεδομένων θα μείνει αχρησιμοποίητο(Εικόνα 17). Έτσι η απόδοση της μνήμης υποβαθμίζεται και ο χρόνος απόκρισης αυξάνεται.



Εικόνα 25: Δεδομένα που δεν θα χρησιμοποιηθούν (https://www.slideshare.net/JoshWyatt5).

Ένα εύκολα κατανοήσιμο παράδειγμα είναι όταν έχουμε να κάνουμε άθροισμα των στοιχείων ενός πίνακα κατά γραμμή σε αντίθεση με όταν έχουμε να κάνουμε άθροισμα στοιχείων κατά στήλη.

Στην περίπτωση που θέλουμε να αθροίσουμε τις σειρές τότε κάθε ένα από τα τέσσερα νήματα στο warp θα αναλάβει μία γραμμή του πίνακα(Εικόνα 18).

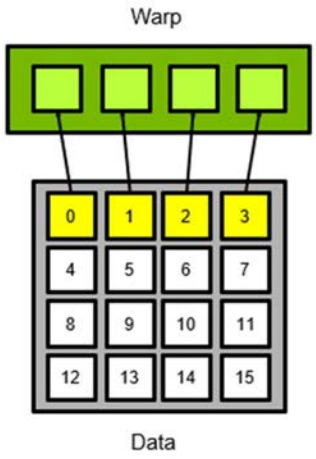


Εικόνα 26: Άθροισμα γραμμών πίνακα (https://www.slideshare.net/JoshWyatt5).

Κάθε νήμα θα χρησιμοποιήσει δεδομένα σε διαφορετικά σημεία της μνήμης και όχι συνεχή. Για παράδειγμα το πρώτο νήμα θα δεχτεί το 0 σαν δεδομένο. Στην μνήμη, το αμέσως επόμενο στοιχείο αποθηκευμένο είναι το 1 και στην συνέχεια το 2. Οπότε τα νήματα θα δεχτούν σαν δεδομένα στοιχεία που είναι αποθηκευμένα σε διάφορα σημεία της μνήμης.

Επιπλέον, για να πραγματοποιηθούν οι υπολογισμοί πρέπει όλες οι γραμμές να φορτωθούν όμως μόνο το 25% αυτόν θα χρησιμοποιείτε κάθε φορά. Καθώς τα νήματα θα διέρχονται στις γραμμές το ίδιο μη συγχωνευμένο πρότυπο θα επαναλαμβάνεται.

Στην περίπτωση που έχουμε άθροισμα στηλών ενός πίνακα βλέπουμε πως η πρόσβαση στην μνήμη από τα νήματα του warp είναι συγχωνευμένη(coalesced)(Εικόνα 19). Κάθε νήμα θα αναλάβει μία στήλη. Το πρώτο νήμα θα δεχτεί το στοιχείο 0, το δεύτερο το 1, το τρίτο το 2 και το τέταρτο το 3. Όλα αυτά τα στοιχεία είναι αποθηκευμένα στην μνήμη το ένα μετά το άλλο. Επομένως η πρόσβαση στην μνήμη είναι coalesced. Ακόμη, δεν χρειάζεται να φορτώσουμε ολόκληρο τον πίνακα ή τις στήλες του. Αρκεί να φορτωθεί η πρώτη σειρά, η οποία θα χρησιμοποιηθεί στους υπολογισμούς ολόκληρη χωρίς να μένουν στοιχεία αχρησιμοποίητα.



Εικόνα 27: Άθροισμα στηλών (https://www.slideshare.net/JoshWyatt5).

Μία εξίσου αποτελεσματική τεχνική είναι ελαχιστοποίηση της πρόσβασης στην καθολική μνήμη. Αφού η καθολική μνήμη είναι σχετικά αργή προσπαθούμε να μειώσουμε τις προσπελάσεις όσο γίνεται, επιτρέποντας μόνο τις πιο σημαντικές. Αυτό το πετυχαίνουμε με την χρήση της κοινής μνήμης(shared memory) που θα δούμε στην συνέχεια.

Κοινή μνήμη (Shared Memory)

Η κοινή μνήμη είναι ένας ειδικός τύπος μνήμης στην CUDA που βρίσκεται on-chip και αντίθετα από την καθολική μνήμη, στην οποία έχουν πρόσβαση όλα τα νήματα, η κοινή μνήμη είναι προσβάσιμη μόνο από τα νήματα (threads) μέσα σε ένα block νημάτων (thread block). Είναι πολύ πιο γρήγορη από την καθολική μνήμη (DRAM), καθώς βρίσκεται φυσικά πιο κοντά στους πολυεπεξεργαστές ροής (streaming multiprocessors, SMs). Ωστόσο, έχει περιορισμένο μέγεθος (συνήθως 48 KB ανά block, ανάλογα με την αρχιτεκτονική της GPU).

Η χρήση της κοινής αντί της καθολικής μνήμης βελτιώνει την απόδοση των αλγορίθμων μας, κυρίως όταν αυτοί απαιτούν συχνή πρόσβαση στη μνήμη.

Εφόσον η κοινή μνήμη είναι κοινή για όλα τα νήματα σε ένα block, ο σωστός συγχρονισμός είναι κρίσιμος. Χρησιμοποιήστε την __syncthreads() για να διασφαλίσετε ότι όλα τα νήματα έχουν ολοκληρώσει μια φάση (π.χ. φόρτωση δεδομένων) πριν προχωρήσουν στην επόμενη φάση (π.χ. υπολογισμός).

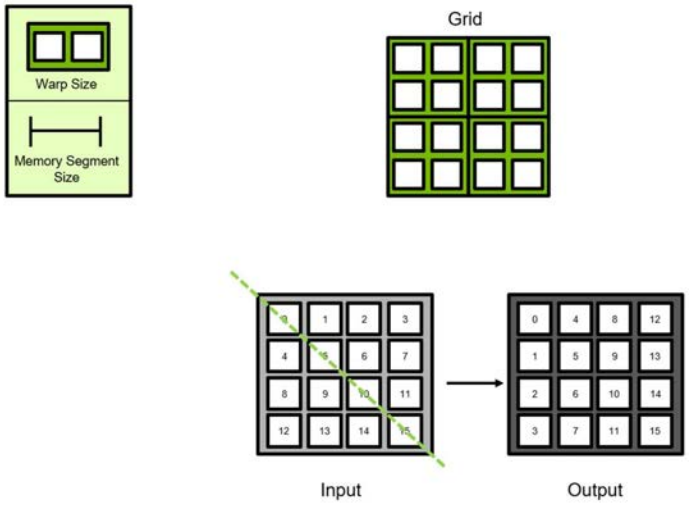
Η κοινή μνήμη χωρίζεται σε τράπεζες (banks), και η ταυτόχρονη πρόσβαση στην ίδια τράπεζα από πολλά νήματα μπορεί να προκαλέσει συγκρούσεις τραπεζών (bank conflicts), μειώνοντας την απόδοση. Για να αποφύγουμε τις συγκρούσεις:

- Βεβαιωνόμαστε ότι οι προσβάσεις στη μνήμη είναι συγχρονισμένες (coalesced).
- Χρησιμοποιούμε padding για να κατανείμουμε τις προσβάσεις στις τράπεζες.

Χρήση κοινής μνήμης για την υποστήριξη πρόσβασης σε συγχωνευμένα δεδομένα (coalesced memory access).

Για να καταλάβουμε καλύτερα πως μπορούμε να εκμεταλλευτούμε την κοινή μνήμη για την υποστήριξη συνεχόμενης μνήμης ας δούμε ένα παράδειγμα. Έστω ότι έχουμε έναν πίνακα (4, 4) σε μέγεθος και έναν ακόμη πίνακα εξόδου στον οποίο θα αποθηκεύσουμε το αποτέλεσμά μας. Στόχος μας είναι να μεταφέρουμε τον αρχικό πίνακα περιστρέφοντας όλα τα στοιχεία γύρω από την διαγώνιο.

Έστω, ότι έχουμε ένα πλέγμα (2, 2) και κάθε μπλοκ στο πλέγμα περιέχει (2, 2) νήματα. Για να απλουστεύσουμε τα πράγματα ας πούμε επίσης πως ένα warp έχει 2 νήματα και κάθε τμήμα μνήμης είναι μεγέθους 2 στοιχείων.



Εικόνα 28: Παράδειγμα χρήσης κοινής μνήμης (https://www.slideshare.net/JoshWyatt5).

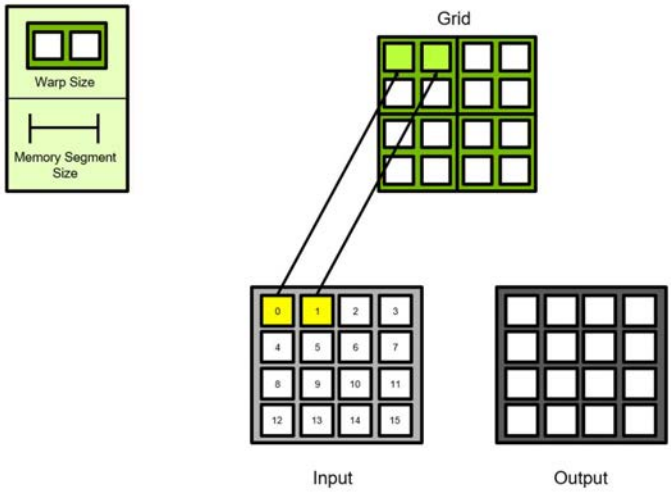
Ένας απλός τρόπος θα ήταν να μετρήσω τα στοιχεία του πίνακα και να χρησιμοποιήσω τόσα νήματα. Με αυτόν τον τρόπο κάθε νήμα θα διαβάσει ένα στοιχείο και θα το μεταφέρει στον πίνακα εξόδου.

```
x, y = cuda.grid(2)
```

```
out[x][y] = in[y][x]
```

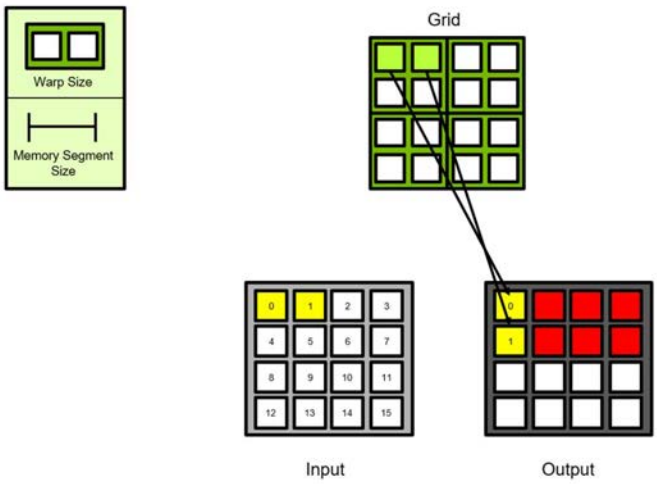
Ας δούμε για αρχή πως θα διαβάσει κάθε νήμα από τον αρχικό πίνακα. Συνεχόμενα νήματα μέσα στο ίδιο warp είναι γειτονικά κατά μήκος του άξονα x. Αυτά τα νήματα θα διαβάσουν τον πίνακα κατά σειρά, οπότε τα στοιχεία που θα διαβαστούν είναι επίσης συνεχόμενα. Επομένως τα δεδομένα στα οποία έχει πρόσβαση το warp είναι coalesced.

```
X = blockIdx.x * blockDim.x + threadIdx.x
Y = blockIdx.y * blockDim.y + threadIdx.y
Out[x][y] = in[y][x]
```



Εικόνα 29: Ανάγνωση από συνεχόμενη μνήμη (coalesced)
(https://www.slideshare.net/JoshWyatt5).

Για την ανάγνωση αφού αυτή είναι ήδη σε συνεχή μνήμη δεν χειμαζόμαστε την χρήση της κοινής μνήμης. Ας δούμε τι θα συμβεί κατά της εγγραφή των αποτελεσμάτων.



Εικόνα 30: Εγγραφή αποτελεσμάτων (https://www.slideshare.net/JoshWyatt5).

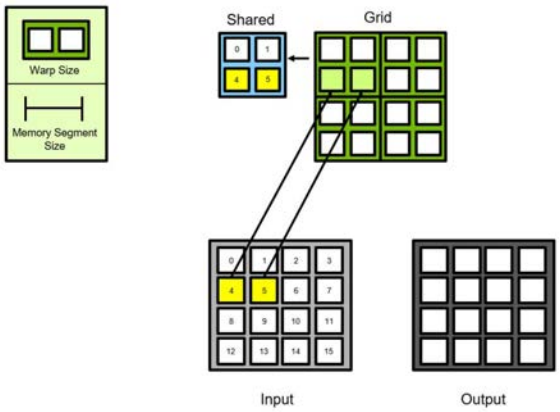
Βλέπουμε πως η εγγραφή στον πίνακα εξόδου γίνεται κατά μήκος των στηλών, οπότε η μνήμη στην οποία θα γράψουμε δεν είναι συνεχόμενη (not coalesced).

Εδώ είναι που θα μας βοηθήσει η χρήση της κοινής μνήμης. Για την χρήση της κοινής μνήμης θα χρειαστούμε κάθε μπλοκ να δεσμεύσει ένα πλακίδιο μνήμης (memory tile).

```
tile = cuda.shared.array(2, 2)
```

Αφού λοιπόν διαβάσουμε τα δεδομένα από τον αρχικό πίνακα, θα τα αποθηκεύσουμε στο shared memory tile. Επειδή σε κάθε tile έχουμε πρόσβαση μόνο από το αντίστοιχο μπλοκ, και όχι από ολόκληρο το πλέγμα, καταχωρούμε τα δεδομένα χρησιμοποιώντας τους δείκτες νημάτων και όχι τους δείκτες πλέγματος. Για αυτό και η αποθήκευση θα γίνει κατά σειρά στο tile.

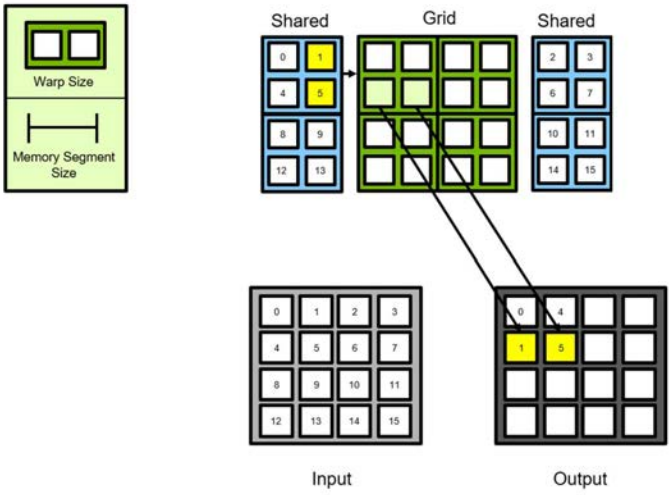
```
tile = cuda.shared_array (2, 2)
x, y = cuda.grid (2)
tile [tIdx.y][tIdx.x] = in [y][x]
cuda.syncthreads ()
```



Εικόνα 31: Εγγραφή δεδομένων σε memory tile (https://www.slideshare.net/JoshWyatt5).

Στην συνέχεια θέλουμε τα δεδομένα στον πίνακα εξόδου να είναι επίσης συνεχόμενα. Η εγγραφή όμως γίνεται κατά σειρά. Σε αυτή την περίπτωση αρκεί να διαβάσουμε τα δεδομένα από την κοινή μνήμη κατά στήλη και να τα αποθηκεύσουμε κατά σειρά.

```
tile = cuda.shared.array(2,2)
x, y = cuda.grid(2)
tile[tIdx.y][tIdx.x] = in[y][x]
cuda.syncthreads()
o_x = bId.y*bDim.y + tId.x
o_y = bId.x*bDim.x + tId.y
o[o_y][o_x] = tile[tIdx.x][tIdx.y]
```

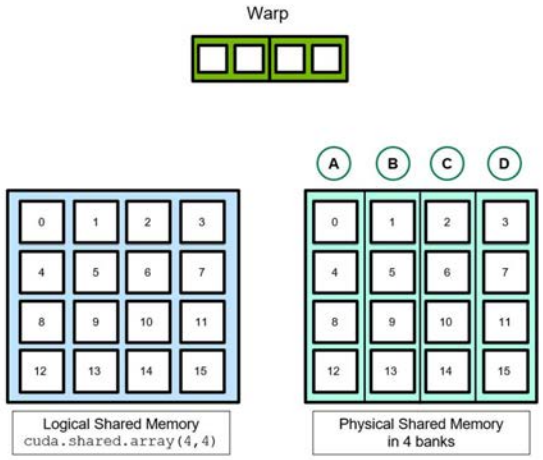


Εικόνα 32: Εγγραφή δεδομένων σε συνεχής μνήμη (coalesced) (https://www.slideshare.net/JoshWyatt5).

Συγκρούσεις τραπεζών (Bank Conflicts)

Η κοινή μνήμη είναι ένα ισχυρό εργαλείο για τη βελτιστοποίηση των πυρήνων CUDA, μειώνοντας την πρόσβαση στην καθολική μνήμη και επιτρέποντας αποδοτική επικοινωνία μεταξύ νημάτων. Ωστόσο, απαιτεί προσεκτική διαχείριση λόγω του περιορισμένου μεγέθους της και της πιθανότητας για συγκρούσεις τραπεζών. Με τη σωστή αξιοποίηση της κοινής μνήμης, μπορείτε να βελτιώσετε σημαντικά την απόδοση των εφαρμογών που περιορίζονται από τη μνήμη.

Στην φυσική της μορφή η κοινή μνήμη υπάρχει σε μορφή τραπεζών (banks). Η πραγματική κοινόχρηστη μνήμη στον υπολογιστή μας αποτελείτε από 32 banks μεγέθους 4 byte. Ας πούμε πως εμείς έχουμε κοινή μνήμη που αποτελείτε από 4 banks. Το warp μας αποτελείτε επίσης από 4 νήματα. Διαδοχικά στοιχεία των 4 byte αν μεταφερθούν στην κοινή μνήμη θα ανήκουν σε διαφορετικές τράπεζες. Το warp μας έχει τέσσερα νήματα οπότε θα έχει πρόσβαση σε 4 byte και σε 4 banks παράλληλα. Η πρόσβαση γίνεται ταυτόχρονα.

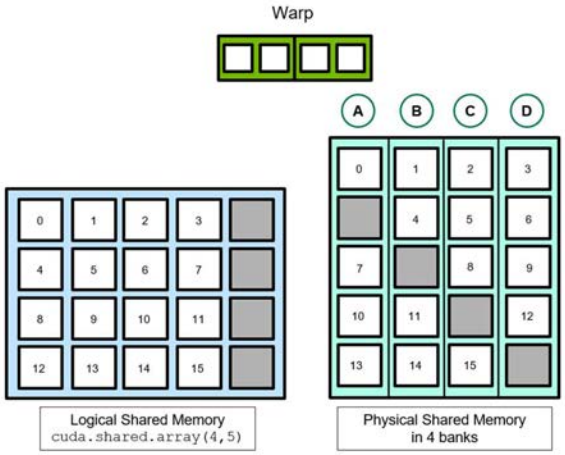


Εικόνα 33: Φυσική αναπαράσταση κοινής μνήμης (https://www.slideshare.net/JoshWyatt5).

Η ταυτόχρονη πρόσβαση των νημάτων στην ίδια τράπεζα έχεις ως αποτέλεσμα την δημιουργία συγκρούσεων (bank conflicts). Στο παράδειγμα μας (Εικόνα 32) όταν διαβάζουμε ανά στήλη στην κοινή μνήμη διαβάζουμε από την ίδια τράπεζα και αυτό θα δημιουργήσει προβλήματα. Μπορούμε να αποφύγουμε τέτοιου είδους προβλήματα με ορισμένες τεχνικές ανάλογα με το πρόβλημα που αντιμετωπίζουμε.

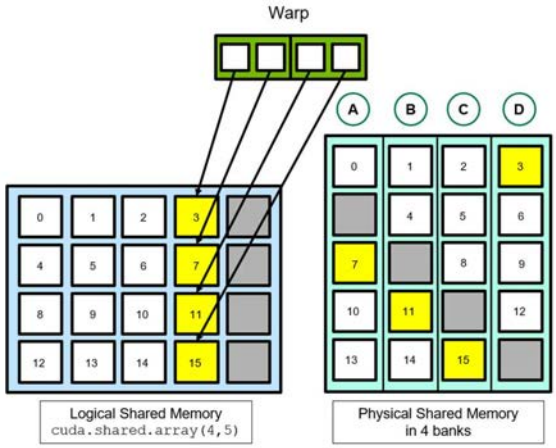
Στην περίπτωση που αντιμετωπίζουμε εμείς για να βεβαιωθούμε ότι κάθε νήμα διαβάζει από διαφορετική τράπεζα θα προσθέσουμε μια επιπλέον στήλη. Όταν όμως θα εργαζόμαστε στον πίνακα θα υποθέτουμε ότι έχουμε έναν 4 x 4 πίνακα και δεν θα δίνουμε

σημασία στην έξτρα στήλη. Η κοινόχρηστη μνήμη στο παράδειγμα μας αποτελείτε από τέσσερις τράπεζες. Η συμπλήρωση στήλης δεν θα επηρεάσει τον αριθμό των τραπεζών. Τα δεδομένα μας θα είναι στην πραγματικότητα αποθηκευμένα έτσι:



Εικόνα 34: Φυσική αποθήκευση δεδομένων στην κοινή μνήμη (https://www.slideshare.net/JoshWyatt5).

Όταν θέλουμε να διαβάσουμε μια στήλη του πίνακα κάθε στοιχείο θα υπάρχει σε διαφορετική τράπεζα αποφεύγοντας έτσι οποιαδήποτε σύγκρουση.



Εικόνα 35: Ανάγνωση δεδομένων ανά στήλη (https://www.slideshare.net/JoshWyatt5).

3.3 Προγραμματισμός CUDA σε Python με χρήση Numba

Ο ποιο συνηθισμένος τρόπος προγραμματισμού με CUDA είναι η χρήση της γλώσσας C/C++. Διαφορετικές επιλογές υπάρχουν και στην περίπτωση που θέλουμε να κάνουμε χρήση της γλώσσας Python. Εκεί έχουμε κυρίως την χρήση της pyCUDA και της Numba. Επιλέξαμε την Numba επιτρέπει μαζική επιτάχυνση, συχνά με ελάχιστες τροποποιήσεις στον κώδικα. Δίνει στους προγραμματιστές την ευκολία να γράφουν κώδικα απευθείας σε Python βελτιστοποιώντας τον κώδικα για τον CPU.

Η Numba είναι ένας μεταγλωττιστής κατά την στιγμή εκτέλεσης (just-in-time, JIT) για τη γλώσσα Python, που προσφέρει ένα εύχρηστο περιβάλλον για την επιτάχυνση αριθμητικών υπολογισμών. Είναι μια πολύ καλή επιλογή για προγραμματισμό με Python χωρίς την χρήση της γλώσσας C/C++. Η Numba είναι επίσης όταν εργαζόμαστε με υπολογιστικά απαιτητικές λειτουργίες της βιβλιοθήκης NumPy. Με το Numba, οι συναρτήσεις της Python μπορούν να βελτιστοποιηθούν τόσο για CPUs όσο και για GPUs της NVIDIA. Στην συνέχεια του κεφαλαίου θα εξερευνήσουμε πως να χρησιμοποιήσουμε την Numba για να επιταχύνουμε τα προγράμματα μας.

3.3.1 Εισαγωγή στην Numba

Η Numba είναι ένας μεταγλωττιστής συναρτήσεων που έχει σχεδιαστεί για να επιταχύνει κώδικα Python τόσο σε CPU όσο και σε GPU.

Αυτό που μεταγλωττίζει είναι μεμονωμένες συναρτήσεις Python και όχι ολόκληρα προγράμματα. Δεν αντικαθιστά τον διερμηνέα Python αλλά λειτουργεί ως δομικό στοιχείο (module) που βελτιστοποιεί τις συναρτήσεις για καλύτερη απόδοση.

Επίσης εξειδικεύει τον τύπο των δεδομένων (Type-specializing). Αυτό σημαίνει ότι βελτιώνει την ταχύτητα εκτέλεσης δημιουργώντας βελτιστοποιημένες εκδόσεις συναρτήσεων με βάση τους συγκεκριμένους τύπους δεδομένων. Η Python είναι αλήθεια πως χειρίζεται γενικούς τύπους δεδομένων με ευελιξία αλλά αυτό οδηγεί πολλές φορές σε επιβράδυνση κατά την εκτέλεση. Η Numba έρχεται να λύσει αυτή την επιβράδυνση δημιουργώντας εξειδικευμένες, υψηλής απόδοσης εφαρμογές που ειδικεύονται στους τύπους δεδομένων που χρησιμοποιούνται κάθε φορά.

Είναι ένας μεταγλωττιστής κατά την στιγμή εκτέλεσης (just-in-time). Δηλαδή μεταφράζει συναρτήσεις κατά τον χρόνο εκτέλεσης όταν αυτές καλούνται για πρώτη φορά. Αυτό επιτρέπει στον μεταγλωττιστή να βελτιστοποιεί τον κώδικα με βάση τους ακριβείς

τύπους ορισμάτων, καθιστώντας τον κατάλληλο για δια δραστικά περιβάλλοντα όπως τα σημειωματάρια Jupyter notebooks.

Επιπλέον, επικεντρώνεται στα αριθμητικά δεδομένα. Η ανάπτυξη της Numba εστιάζει στην βελτιωμένη χρήση τύπων δεδομένων όπως int, float και complex. Δεν υποστηρίζει τόσο καλά συμβολοσειρές καθώς οι περισσότερες λειτουργίες συμβολοσειρών δεν είναι κατάλληλες για επιτάχυνση προγραμμάτων στην GPU. Για την βέλτιστη απόδοση της βιβλιοθήκης η Numba χρησιμοποιείται μαζί με την NumPy. Ποιο συγκεκριμένα κάνει χρήση των πινάκων, τύπου δεδομένων που διαθέτει η NumPy.

Όπως είπαμε και παραπάνω η Numba δεν χρησιμοποιείται για βελτιστοποίηση προγραμμάτων μόνο στην GPU αλλά και για την CPU. Πριν δούμε για την GPU ας δούμε πως μια συνάρτηση με Numba εκτελείτε στην CPU. Ένα σημαντικό μέτρο σύγκρισης απόδοσης των προγραμμάτων είναι η διαφορά στους απαιτούμενους χρόνους όταν αυτά εκτελούνται στην CPU και όταν εκτελούνται στην GPU.

Ο μεταγλωττιστής Numba συνήθως ενεργοποιείται με την εφαρμογή ενός διακοσμητή συναρτήσεων (function decorator) σε μια συνάρτηση Python. Ας δούμε ένα απλό πρόγραμμα υπολογισμού της υποτείνουσας ενός τριγώνου με το Πυθαγόρειο θεώρημα.

```
1. from numba import jit
2. import math
3. @jit(nopython = True)
4. def hypotenuse_calc (x, y):
5.     x = abs(x)
6.     y = abs(y)
7.     t = min(x, y)
8.     x = max(x, y)
9.     t = t / x
10.    z = x * math.sqrt(1+t*t)
11.    return z
```

Πρόγραμμα 8 : Υπολογισμός υποτείνουσας στην CPU με Numba

Περιγραφή κώδικα :

- Γραμμές 1 και 2 : Ενσωμάτωση κατάλληλων βιβλιοθηκών
- Γραμμή 3 : Εφαρμογή του compiler jit. Το όρισμα εξασφαλίζει ότι η Numba θα μεταφράσει ολόκληρο τον κώδικα.
- Γραμμή 4 : Συνάρτηση υπολογισμού της υποτείνουσας
- Γραμμές 5 και 6 : Βρίσκουμε τις απόλυτες τιμές των πλευρών. Δεν γίνεται απόσταση να έχει αρνητική τιμή.

- Γραμμές 7 και 8 : Βρίσκουμε την μέγιστη και ελάχιστη τιμή από τις δύο.
- Γραμμή 9 : Κανονικοποιούμε την μικρότερη τιμή διαιρώντας την με την μεγαλύτερη για αποφυγή σφαλμάτων αριθμητικής ακρίβειας.
- Γραμμές 10 και 11 : Υπολογισμός της υποτείνουσας και αποστολή.

Εν συντομία για το nopython = True όρισμα να πούμε πως η Numba όταν έχει να κάνει με γενικούς τύπους δεδομένων της Python, όπως για παράδειγμα λίστες ή συμβολοσειρές, τότε επιστρέφει σε slow object λειτουργία αντί για nopython λειτουργία. Οπότε η εκτέλεση της συνάρτησης γίνεται με Python κώδικα αντί για κώδικα μηχανής που θα έπρεπε να τον μετατρέψει η Numba. Έτσι ο κώδικας δεν μπορεί να μεταγλωττιστεί αποτελεσματικά και αυτό επιβαρύνει τον χρόνο εκτέλεσης και την αποδοτικότητα. Έχει όμως ακόμη μία χρήση, να μας ειδοποιεί με εξαίρεση (exception) όταν ο τύπος δεδομένων δεν είναι συμβατός. Θα το δούμε στην συνέχεια.

Πρέπει να σημειώσουμε πως την πρώτη φορά που θα καλέσουμε την συνάρτηση hypotenuse_calc ο μεταγλωττιστής θα ενεργοποιηθεί και θα μεταγλωττίσει τον κώδικα σε κώδικα μηχανής. Η Numba επίσης θα αποθηκεύσει τον αρχικό Python κώδικα στο .py_func attribute ώστε να μπορούμε να καλούμε τον αρχικό κώδικα. Αυτό μας βοηθάει να δούμε ότι ο κώδικας δίνει το ίδιο αποτέλεσμα, καθώς και να συγκρίνουμε της απόδοση μεταξύ του μεταγλωττισμένου και μη κώδικα.

```
1. Hypotenus_culc.py_func(3.0, 4.0)          #Κώδικας χωρίς Numba
2. Hypotenus_calc(3.0, 4.0)                  #Κώδικας με Numba
```

Πρόγραμμα 9 : Εκτέλεση συναρτήσεων με και χωρίς Numba.

Αρκετά σημαντικό στην χρήση της Numba είναι να μπορούμε να μετρήσουμε την απόδοση του κώδικά μας. Ένας απλός τρόπος να το κάνουμε αυτό είναι με την χρήση της εντολής %timeit. Αυτό που κάνει η εντολή αυτή είναι να εκτελεί τον κώδικα που ζητάμε πολλές φορές και μας επιστρέφει τον μέσο όρο του χρόνου που χρειάστηκε για να εκτελεσθεί.

Αν λοιπόν τρέξουμε την εντολή:

```
1. %timeit hypotenuse_calc.py_func(3.0, 4.0)
```

Πρόγραμμα 10 : Μέτρηση χρόνου εκτέλεσης αρχικού κώδικα

Αποτέλεσμα:

401 ns ± 0.631 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)

Η συνάρτηση εκτελεί τον αρχικό κώδικα αρκετές φορές για να υπολογίσει με ακρίβεια τον χρόνο εκτέλεσης και επιστρέφει τον καλύτερο χρόνο.

Στην συνέχεια τρέχουμε την εντολή:

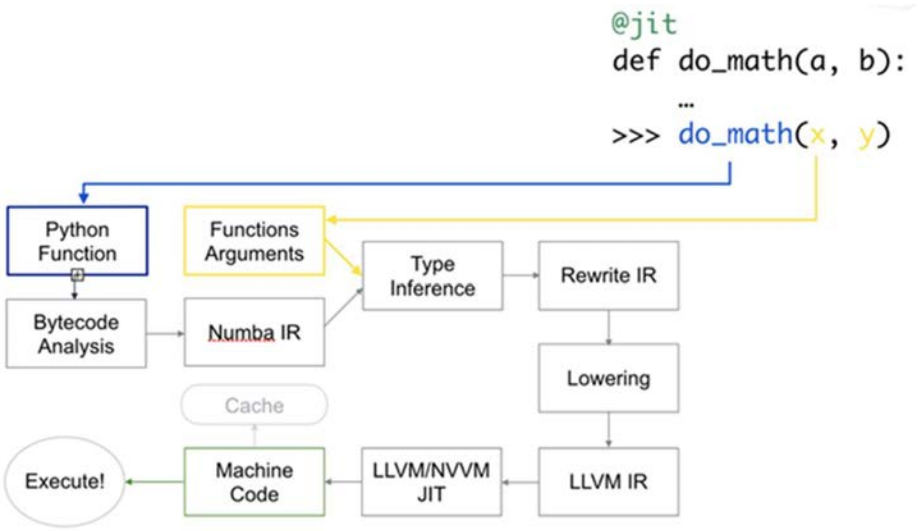
```
1. %timeit hypotenuse_calc (3.0, 4.0)
```

Πρόγραμμα 11 : Μέτρηση χρόνου εκτέλεσης κώδικα με Numba.

Αποτέλεσμα:

101 ns ± 0.159 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)

Η οποία εκτελεί τον κώδικα με την χρήση της Numba. Ο χρόνος εκτέλεσης βελτιώθηκε αρκετά. Την πρώτη φορά που θα εκτελέσουμε την συνάρτηση με την βοήθεια της Numba η εκτέλεση θα είναι σχετικά αργή, και αυτό γιατί θα έχουμε καθυστέρηση κατά την μεταγλώττιση του κώδικα. Μετά την πρώτη εκτέλεση η μεταγλωττισμένη έκδοση του κώδικα θα αποθηκευτεί στην προσωρινή μνήμη, οπότε κατά τις επόμενες εκτελέσεις δεν θα χρειαστεί να γίνει ξανά η μεταγλώττιση.



Εικόνα 36 : Πώς λειτουργεί η Numba (https://www.nvidia.com/en-us/glossary/numba).

Να σημειωθεί πως τα ονόματα των τύπων δεδομένων της Numba είναι ίδια με αυτά της NumPy. Έτσι ένας Python float τύπος δεδομένου είναι float64 στην Numba. Οι τύποι δεδομένων είναι πολύ σημαντικοί στην εκτέλεση κώδικα στην GPU που θα δούμε στην

συνέχεια. Η χρήση float32 και float64 μπορούν να αλλάξουν σημαντικά την απόδοση του κώδικα. Αυτό, πολλές φορές εξαρτάται και από τον τύπο της GPU.

Ο τύπος τον δεδομένων είναι σημαντικός γιατί η Numba δεν υποστηρίζει όλους τους τύπους δεδομένων της Python. Για παράδειγμα τα dictionaries δεν υποστηρίζονται ακόμη. Για να ελέγχουμε τον τύπο των δεδομένων η Numba μας παρέχει με την παράμετρο nopython η οποία θα μας επιστρέψει exception όταν δεν θα αναγνωρίζει τον τύπο.

Η παράμετρος αυτή θα κάνει την Numba να προσπαθήσει να μετατρέψει τον κώδικά μας σε κώδικα μηχανής. Αν για οποιονδήποτε λόγο αποτύχει, αυτό θα συμβεί αν έχουμε στοιχεία που δεν υποστηρίζονται, τότε θα επιστρέψει σφάλμα. Στην περίπτωσή μας δεν θα καταφέρει να μεταφράσει το dictionary.

```
@jit(nopython = True)
```

Εναλλακτικά, με τον ίδιο τρόπο λειτουργεί και η εντολή

```
@njit
```

3.3.2 Χρήση Numba για προγραμματισμό στην GPU

Όπως είναι γνωστό η GPU είναι σχεδιασμένη για παράλληλους υπολογισμούς και η χρήση των δυνατοτήτων τους βασίζεται στην εκτέλεση των ίδιων υπολογισμών ταυτόχρονα σε πολλά στοιχεία της.

Οι συναρτήσεις της NumPy (ufuncs) εκτελούν την ίδια λειτουργία σε κάθε στοιχείο ενός NumPy πίνακες και επομένως είναι από την φύση τους κατάλληλα για παράλληλους υπολογισμούς.

Python ufuncs

Τα ufuncs (συντομογραφία του "universal functions") είναι συναρτήσεις στη βιβλιοθήκη NumPy της Python που επιτρέπουν τη διεκπεραίωση στοιχειωδών πράξεων σε πίνακες (arrays) με τρόπο που είναι και αποδοτικός και εύκολος στη χρήση. Οι ufuncs λειτουργούν στοιχείο-προς-στοιχείο (element-wise), πράγμα που σημαίνει ότι εφαρμόζουν μια πράξη σε κάθε στοιχείο του πίνακα ξεχωριστά.

Τα ufuncs εκτελούνται γρήγορα, ιδιαίτερα σε μεγάλους πίνακες, είναι απλά στην σύνταξη και υποστηρίζουν πολλές πράξεις και τύπους δεδομένων. Είναι βασικό στοιχείο της NumPy που διευκολύνει την επεξεργασία και ανάλυση δεδομένων σε πίνακες.

```
1. import numpy as np
2. a = np.array([1, 2, 3, 4])
3. b = np.array([10, 11, 12, 13,])
4. np.add(a, b)
```

Πρόγραμμα 12: Παράδειγμα ufuncs

Αποτέλεσμα :

array([11, 13, 15, 17])

Περιγραφή κώδικα :

- Γραμμή 1: Καλώ την βιβλιοθήκη NumPy
- Γραμμές 2 και 3: Δημιουργώ δύο πίνακες με 4 στοιχεία μέσα.
- Γραμμή 4: Καλώ την συνάρτηση add της Python για την πρόσθεση των δύο.

Η Numba μας δίνει την δυνατότητα να μεταγλωττίσουμε αυτές τις συναρτήσεις με την χρήση του στοιχείου @vectorize, το οποίο θα μπει στην αρχή της συνάρτησης και η Numba από μόνη της θα βρει τον τρόπου που θα επικοινωνήσει με την GPU.

Το στοιχείο @vectorize μπορεί να πάρει και κάποιες παραμέτρους οι οποίες δηλώνουν τον τύπο των δεδομένων που θα χρησιμοποιηθούν για τους υπολογισμούς αλλά και τον τύπο δεδομένων που θα επιστραφούν σαν αποτέλεσμα.

```
1. from numba import vectorize
2. @vectorize(['float32(float32, float32)'], target='cuda')
3. def add_ufunc(a, b):
4.     return a + b
```

Πρόγραμμα 13 : Πρόσθεση δύο αριθμών στην GPU

Περιγραφή κώδικα :

- Γραμμή 1 : Εισαγωγή στοιχείου από την βιβλιοθήκη της numba
- Γραμμή 2 : @vectorize Επιτρέπει την επεξεργασία των στοιχείων σε πίνακες . Τα float32 δηλώνουν πως τα δύο νούμερα που δέχεται η συνάρτηση add_ufunc() είναι κινητής υποδιαστολής και το αποτέλεσμα που θα δώσει η συνάρτηση είναι επίσης τέτοιου τύπου. Η δήλωση target = ‘cuda’ μεταγλωττίζει την συνάρτηση για να εκτελεσθεί στην GPU.
- Γραμμές 3 και 4 : Η συνάρτηση πρόσθεσης δύο αριθμών.

Με αυτήν την απλή δήλωση, στην αρχή της συνάρτησης, η Numba κάνει ορισμένες πολύ σημαντικές ενέργειες. Αρχικά δημιουργεί έναν CUDA πυρήνα(kernel) για να εκτελέσει τις λειτουργίες της συνάρτησης παράλληλα με βάση τα στοιχεία εισόδου. Θα δεσμεύσει χώρο στην μνήμη της GPU για την αποθήκευση των δεδομένων εισόδου και για το τελικό αποτέλεσμα.. Στην συνέχεια θα αντιγράψει τα δεδομένα εισόδου στον χώρο που δέσμευσε στην GPU και θα εκτελέσει την συνάρτηση. Τέλος θα αποθηκεύσει το αποτέλεσμα στην GPU και θα το στείλει πίσω στην CPU. Το αποτέλεσμα θα είναι ένας Numpy πίνακας.

Αν προσπαθήσουμε να χρονομετρήσουμε την εκτέλεση αυτής της απλής συνάρτησης, περνώντας όπως και πιο πάνω 2 πίνακες με 4 στοιχεία, στην GPU και στην CPU τότε θα δούμε ότι η CPU εκτέλεσε τον κώδικα πιο γρήγορα. Αυτό συνέβη γιατί δεν χρησιμοποιήσαμε σωστά την GPU. Αρχικά τα δεδομένα που δίνουμε σαν είσοδο στην συνάρτηση είναι πολύ μικρά σε μέγεθος. Η GPU επιτυγχάνει μεγάλες αποδόσεις λειτουργώντας σε χιλιάδες τιμές ταυτόχρονα. Χρειαζόμαστε, λοιπόν πολύ μεγαλύτερες συστοιχίες για να κάνουμε χρήση της δύναμης της GPU.

Επίσης, ο υπολογισμός μας είναι πολύ απλός. Η εκτέλεση μια συνάρτησης στην GPU έχει αρκετά μεγαλύτερο κόστος σε σύγκριση με την εκτέλεσή της στην CPU. Οπότε αν ο υπολογισμός μας δεν περιλαμβάνει αρκετές μαθηματικές πράξεις τότε η GPU θα αφιερώσει τον περισσότερο χρόνο της περιμένοντας να μετακινηθούν τα δεδομένα.

Τα δεδομένα πρέπει να μετακινηθούν από και προς την GPU. Ενώ κάποιες φορές αυτή η επιβάρυνση αξίζει τον κόπο όταν δεν έχουμε απλές και λίγες πράξεις.

Ένα ακόμα στοιχείο που επιβαρύνει τον κώδικα συχνά είναι οι τύποι δεδομένων που χρησιμοποιούμε. Ίσως int32 θα ήταν μια καλύτερη επιλογή. Βέβαια στους ακεραίους η επιβάρυνση δεν είναι τόση όση των τύπων κινητής υποδιαστολής. Η NumPy όταν δημιουργεί πίνακες ορίζει από προεπιλογή τύπους 64 bit.

Τα ufuncs και η συνάρτηση @vectorize είναι πολύ χρήσιμα εργαλεία στην παραλληλοποίηση κώδικα για την GPU. Παρόλα αυτά πολλές φορές χρειαζόμαστε να φτιάξουμε τους δικούς μας πυρήνες. Σε αυτή την περίπτωση θα χρησιμοποιήσουμε numba.cuda.jit.

3.3.3 Μεταφορά Δεδομένων

Ένα πολύ σημαντικό κεφάλαιο στην CUDA είναι η μεταφορά δεδομένων από την GPU στην CPU και αντίστροφα. Η Numba για να μας διευκολύνει, με την χρήση του

@vectorize, μεταφέρει τα δεδομένα στην GPU για την επεξεργασία και στην συνέχεια αυτόματα μεταφέρει πίσω στην CPU τα αποτελέσματα. Βέβαια ο δικός μας στόχος είναι να μειώσουμε αυτές τις μετακινήσεις δεδομένων όσο τον δυνατόν περισσότερο, αφού επιβαρύνουν σημαντικά τον χρόνο εκτέλεσης των προγραμμάτων μας.

Χρειαζόμαστε έναν τρόπο τα δεδομένα να μην μεταφέρονται αυτόματα αλλά να ελέγχουμε εμείς τις μετακινήσεις τους. Ο τρόπος αυτός είναι η δημιουργία πινάκων (CUDA Device Arrays) τις οποίες θα περνάμε στις συναρτήσεις μας. Θα μεταφέρονται στην GPU, θα πραγματοποιούνται οι υπολογισμοί αλλά δεν θα μεταφέρονται πίσω στην CPU αυτόματα. Μπορούμε να χρησιμοποιήσουμε τα αποτελέσματα τους σε περεταίρω πράξεις και να επιστρέψουμε τα αποτελέσματα μόνο όταν αυτό είναι απαραίτητο.

Ένα απλό παράδειγμα. Έστω ότι έχουμε μια απλή συνάρτηση πρόσθεσης όπως και παραπάνω add_ufunc(). Η numba.cuda περιλαμβάνει μια συνάρτηση, η οποία θα μεταφέρει τα δεδομένα στην GPU, την cuda.to_device().

```
1. from numba import cuda
2. n = 100000
3. a = np.arange(n).astype(np.float32)
4. b = 2 * a
5. d_a = cuda.to _device(a)
6. d_b = cuda.to_device(b)
7. print(d_a)
8. print(d_a.shape)
9. print(d_a.dtype)
```

Πρόγραμμα 14 : Μεταφορά δεδομένων στην GPU

Περιγραφή κώδικα :

- Γραμμή 2: Η παράμετρος n με τιμή 100000 θα ορίσει το μέγεθος του πίνακα που θα δημιουργήσουμε.
- Γραμμή 3: Δημιουργία NumPy πίνακα με τιμές από 0 εώς n-1
- Γραμμές 5 και 6: Μεταφέρουμε τους πίνακες στην συσκευή.
- Γραμμές 7,8 και 9: Εκτυπώνουμε τον πίνακα d_a.

Το αποτέλεσμα της εκτύπωσης αυτής στην πραγματικότητα δεν θα είναι ο πίνακας. Το αποτέλεσμα θα είναι κάπως έτσι:

<numba.cuda.cudadrv.devicearray.DeviceNDArray object at 0x00000253E30BF640>

(100000,)

float32

Τα παραπάνω δεδομένα αφορούν στοιχεία του πίνακα, όπως μέγεθος και τύπος δεδομένων αλλά δεν είναι ο πίνακας. Αυτό συμβαίνει γιατί ο πίνακας δεν είναι αποθηκευμένος στην CPU αλλά στην GPU.

Μέχρι στιγμής μεταφέραμε τους πίνακες που θέλουμε να επεξεργαστούμε στην συσκευή μόνοι μας χωρίς να γίνει αυτόματα από την CUDA. Θα χρειαστούμε όμως έναν πίνακα για τα αποτελέσματα των πράξεων. Αντί να δημιουργήσουμε τον πίνακα αυτόν στην CPU και στην συνέχεια να τον μεταφέρουμε μπορούμε δημιουργήσουμε τον πίνακα με την συνάρτηση numba.cuda.device_array(), απευθείας στην GPU.

Ωστόσο, στην συνέχεια θα χρειαστεί να μεταφέρουμε το αποτέλεσμα πίσω στην CPU με την εντολή copy_to_host().

```
1. out_device = cuda.device_array(shape=(n,), dtype = np.float32
2. out_host = out_device.copy_to_host()
3. print(out_host[:10])
```

Πρόγραμμα 15 Μεταφορά δεδομένων στην CPU.

Περιγραφή κώδικα:

- Γραμμή 1: Δεσμεύει μνήμη, μεγέθους n με τύπο δεδομένων float32, στην GPU.
- Γραμμή 2: Μεταφέρουμε τα δεδομένα μας στην CPU.
- Γραμμή 3: Εκτυπώνουμε τα πρώτα 10 στοιχεία του πίνακα.

Με αυτόν τον τρόπο καταργήσαμε την αυτόματη μεταφορά δεδομένων.

3.3.4 Προσαρμοσμένοι πυρήνες CUDA με Numba

Οι προσαρμοσμένοι πυρήνες CUDA απαιτούν περισσότερη προσοχή για να εφαρμοστούν από ότι για παράδειγμα αν απλά διακοσμήσουμε μια ufunc με το @vectorize. Ενώ είναι πιο δύσκολη η σύνταξη προσαρμοσμένων πυρήνων παρέχει τεράστια ευελιξία. Οι συναρτήσεις, λοιπόν που σκοπό έχουν να εκτελεσθούν στην GPU ονομάζονται πυρήνες και εκκινούνται στου πολλούς πυρήνες που παρέχει η GPU σε παράλληλα νήματα(threads).

Προσαρμόζοντας την add_ufunc() συνάρτηση ας δούμε ένα ολοκληρωμένο πρόγραμμα που εκτελείτε στην GPU.

```
1. from numba import cuda
2. @cuda.jit
3. def add_kernel(x, y, out):
4.         idx = cuda.grid(1)
5.         out[idx] = x[idx] + y[idx]
```

```
1. import numpy as np
2. n = 4096
3. x = np.arange(n).astype(np.int32)
4. y = np.ones_like(x)
5. d_x = cuda.to_device(x)
6. d_y = cuda.to_device(y)
7. d_out = cuda.device_array_like(d_x)
8. threads_per_block = 128
9. blocks_per_grid = 32
```

```
1. add_kernel[blocks_per_grid, threads_per_block](d_x, d_y, d_out)
2. cuda.synchronize()
3. print(d_out.copy_to_host())
```

Πρόγραμμα 16: Πρόγραμμα εκτελέσιμο στην GPU.

Περιγραφή κώδικα :

Στο πρώτο κουτί κώδικα έχουμε την συνάρτηση αθροίσματος.

- Γραμμή 1: Καλούμε την βιβλιοθήκη Numba.
- Γραμμή 2: Δηλώνουμε ότι η συνάρτηση θα εκτελεσθεί στην GPU
- Γραμμή 3: Το όνομα της συνάρτησης και τα ορίσματά της. Παρατηρούμε ότι η συνάρτηση έχει την παράμετρο out στην οποία θα αποθηκευτεί το αποτέλεσμα. Αυτό συμβαίνει διότι οι πυρήνες με @cuda.jit δεν επιστρέφουν κάποια τιμή με την εντολή return.
- Γραμμή 4: Υπολογίζουμε έναν μοναδικό δείκτη νημάτων μέσα σε ένα πλέγμα (grid). Ένα μονοδιάστατο πλέγμα με νήματα επιστρέφει μία μόνο τιμή. Ο υπολογισμός αυτό μπορεί επίσης να γίνει ως εξής cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x
- Γραμμή 5: Η πράξει της πρόσθεσης με κάθε νήμα να αναλαμβάνει διαφορετικά στοιχεία.

Στο δεύτερο κουτί έχουμε τον κυρίως κώδικά μας.

- Γραμμή 1: Βιβλιοθήκη NumPy.

- Γραμμή 2,3 και 4: Αποθήκευση δεδομένων στην CPU. Μια μεταβλητή n με τιμή 4096. Έναν πίνακα με n μέγεθος και στοιχεία από το 0 εως το 4095 και έναν πίνακα σαν τον x αλλά με το στοιχείο 1 σε όλες τις θέσεις του πίνακα.
- Γραμμές 5 και 6: μεταφέρουμε τους πίνακες στην GPU.
- Γραμμή 7: Δεσμεύουμε χώρο στην μνήμη της GPU για τον πίνακα με τα αποτελέσματα.
- Γραμμές 8 και 9: Λόγω του τρόπου με τον οποίο γράψαμε τον πυρήνα παραπάνω, πρέπει να έχουμε αντιστοίχιση 1 νήματος προς ένα στοιχείο δεδομένων, επομένως ορίζουμε τον αριθμό των νημάτων στο πλέγμα (128*32) σε n (4096) όσα είναι και τα δεδομένα μας.

Στο τελευταίο κουτί κώδικα έχουμε την εκτέλεση της συνάρτησης και την μεταφορά των αποτελεσμάτων.

- Γραμμή 1: Εκτέλεση συνάρτησης με βάση τον αριθμό νημάτων που δηλώσαμε ότι θέλουμε.
- Γραμμή 2: Βεβαιωνόμαστε πως η GPU έχει ολοκληρώσει όλους τους υπολογισμούς.
- Γραμμή 3: Μεταφορά του αποτελέσματος στην CPU.
  Είναι πολύ ενδιαφέρον το τι θα συμβεί αν αλλάξουμε τον αριθμό των νημάτων ή των μπλοκ ή και τον δύο.

Ενδιαφέρον παρουσιάζει το αποτέλεσμα που θα έχουμε αν αλλάξουμε τις τιμές των threads_per_block και blocks_per_grid. Στο παραπάνω παράδειγμα, επειδή ο πυρήνας είναι γραμμένος έτσι ώστε κάθε νήμα να επεξεργάζεται ακριβώς ένα στοιχείο δεδομένων, είναι απαραίτητο ο αριθμός των νημάτων στο πλέγμα να ισούται με τον αριθμό των στοιχείων δεδομένων (128*32=4096).

- *Μειώνοντας τον αριθμό των νημάτων στο πλέγμα*, είτε μειώνοντας τον αριθμό των μπλοκ, είτε μειώνοντας τον αριθμό των νημάτων ανά μπλοκ, υπάρχουν στοιχεία στα οποία η εργασία δεν έχει ολοκληρωθεί και έτσι μπορούμε να δούμε στην έξοδο ότι τα στοιχεία προς το τέλος του πίνακα d_out δεν είχαν τιμές που προστέθηκαν σε αυτά. Εάν επεξεργαστείτε τη διαμόρφωση εκτέλεσης μειώνοντας τον αριθμό των νημάτων ανά μπλοκ, τότε στην πραγματικότητα υπάρχουν και άλλα στοιχεία στον πίνακα d_out που δεν επεξεργάστηκαν.

- *Αυξάνοντας το μέγεθος του πλέγματος* δημιουργούμε προβλήματα με πρόσβαση σε μνήμη εκτός ορίων. Αυτό το σφάλμα δεν θα εμφανιστεί στον κώδικά μας. Θα χρειαστούμε κάποιο εργαλείο για τον εντοπισμό σφαλμάτων (debugging).
- *Αφαιρώντας το σημείο συγχρονισμού*, ίσως να περιμένατε ότι θα εμφανιζόταν μια εκτύπωση που θα έδειχνε ότι δεν είχε γίνει καμία διεργασία, ή έστω ότι δεν ολοκληρώθηκε.. Όμως οι αντιγραφές μνήμης φέρουν σιωπηρό συγχρονισμό, καθιστώντας την κλήση στο cuda.synchronize περιττή.

Οι GPU της NVIDIA με δυνατότητα CUDA αποτελούνται από πολλούς Streaming Multiprocessors (SMs) σε ένα chip, με συνδεδεμένη DRAM. Κάθε SM περιέχει όλους τους απαραίτητους πόρους για την εκτέλεση του κώδικα πυρήνα, συμπεριλαμβανομένων πολλών CUDA cores. Όταν εκκινείτε ένας πυρήνας, κάθε block ανατίθεται σε έναν SM, με δυνατότητα πολλών blocks να ανατίθενται στον ίδιο SM. Οι SMs χωρίζουν τα blocks σε περαιτέρω υποδιαιρέσεις 32 νημάτων, που ονομάζονται warps, και είναι αυτά τα warps που λαμβάνουν παράλληλες εντολές για εκτέλεση.

Όταν μια εντολή χρειάζεται περισσότερο από έναν κύκλο ρολογιού για να ολοκληρωθεί (ή, στη γλώσσα CUDA, να "λήξει"), ο SM μπορεί να συνεχίσει να κάνει χρήσιμη εργασία εάν έχει πρόσθετα warps που είναι έτοιμα να εκτελέσουν νέες εντολές. Λόγω των πολύ μεγάλων αρχείων καταχωρητών (register files) στους SMs, δεν υπάρχει χρονικό κόστος για τον SM να αλλάξει περιβάλλον μεταξύ της εκτέλεσης εντολών σε ένα warp ή σε άλλο. Εν συντομία, η καθυστέρηση των πράξεων μπορεί να καλυφθεί από τους SMs με άλλη χρήσιμη εργασία, εφόσον υπάρχει άλλη εργασία προς εκτέλεση.

Επομένως, για να αξιοποιηθεί πλήρως η δυναμική του GPU και να γραφούν αποδοτικές εφαρμογές με επιτάχυνση, είναι απαραίτητο να δώσουμε στους SMs την δυνατότητα να καλύψουν την καθυστέρηση, παρέχοντάς τους έναν επαρκή αριθμό warps. Αυτό μπορεί να επιτευχθεί πιο απλά εκτελώντας πυρήνες με αρκετά μεγάλες διαστάσεις πλέγματος (grid) και block.

Το να καθοριστεί το βέλτιστο μέγεθος για το πλέγμα νημάτων CUDA είναι ένα πολύπλοκο πρόβλημα και εξαρτάται τόσο από τον αλγόριθμο όσο και από την υπολογιστική ικανότητα του συγκεκριμένου GPU. Ωστόσο, υπάρχουν κάποιες γενικές μέθοδοι που τείνουμε να ακολουθούμε και μπορούν να λειτουργήσουν καλά για αρχή:

1. Το μέγεθος ενός block πρέπει να είναι πολλαπλάσιο των 32 νημάτων (το μέγεθος ενός warp), με τυπικά μεγέθη block να κυμαίνονται μεταξύ 128 και 512 νημάτων ανά block.

2. Το μέγεθος του πλέγματος (grid) θα πρέπει να εξασφαλίζει ότι το GPU χρησιμοποιείται πλήρως, όπου είναι δυνατόν. Η εκκίνηση ενός πλέγματος όπου ο αριθμός των blocks είναι 2x έως 4x τον αριθμό των SMs στο GPU είναι ένα καλό σημείο εκκίνησης. Κάτι στην περιοχή των 20 έως 100 blocks είναι συνήθως ένα καλό σημείο εκκίνησης.
3. Το κόστος εκκίνησης ενός πυρήνα CUDA αυξάνεται με τον αριθμό των blocks, οπότε όταν το μέγεθος της εισόδου είναι πολύ μεγάλο, είναι καλύτερο να μην εκκινείτε ένα πλέγμα όπου ο αριθμός των νημάτων ισούται με τον αριθμό των στοιχείων εισόδου, καθώς αυτό θα οδηγούσε σε τεράστιο αριθμό blocks. Αντίθετα, χρησιμοποιούμε μία τεχνική για την αντιμετώπιση μεγάλων εισόδων. Η τεχνική ονομάζεται grid stride loop.

3.3.5 Διαχείριση μεγάλου όγκου δεδομένων με Grid Stride Loops

Είδαμε σε προηγούμενο σημείο του κεφαλαίου πως η CUDA εφαρμόζει πρακτικά αυτή την τεχνική (σελ. 46). Ας δούμε πως το κάνουμε πράξη στο πρόγραμμα που γράψαμε παραπάνω (Πρόγραμμα 16).

```
1. from numba import cuda
2. import numpy as np
3. @cuda.jit
4. def add_kernel(x, y, out):
5. start = cuda.grid(1)
6. stride = cuda.gridsize(1)
7. for i in range(start, x.shape[0], stride):
8.           out[i] = x[i] + y[i]
```

```
 1. n = 100000
 2. x = np.random.uniform(-12, 12, n).astype(np.float32)
 3. y = np.random.uniform(-12, 12, n).astype(np.float32)
 4. d_x = cuda.to_device(x)
 5. d_y = cuda.to_device(y)
 6. d_out = cuda.device_array_like(d_x)
 7. threads_per_block = 64
 8. blocks_per_grid = 128
 9. add_kernel[blocks, threads_per_block](d_x, d_y, d_out)
10. h_c = d_out.copy_to_host()
11. add_kernel[blocks_per_grid, threads_per_block](d_x, d_y, d_out)
12. print(d_out.copy_to_host())
```

Πρόγραμμα 17: Εφαρμογή Grid Stride Loop.

Αποτέλεσμα:

[6.6071815 -12.857121 -10.841019 ... -5.967269 -5.737959 13.019486]

Περιγραφή κώδικα:

Ο πυρήνας μας είναι η συνάρτηση add_kernel().

- Γραμμή 6: Στην προηγούμενη γραμμή ορίσαμε το grid(1) όπως και στο Πρόγραμμα 16 παραπάνω. Σε αυτή την γραμμή όμως ορίζουμε το βήμα με το οποίο θα προσπελάσει τα δεδομένα (Εικόνα 26).
- Γραμμές 7 και 8: Ορίζουμε έναν βρόγχο, ο οποίος θα ξεκινάει από τον δείκτη του κάθε νήματος (start), μέχρι να προσπελάσει όλα τα δεδομένα (x.shape[0] = συνολικός αριθμός στοιχείων στον πίνακα) με βήμα (stride) όπως το υπολογίσαμε πριν. Μέσα στον βρόγχο έχουμε την πράξη της πρόσθεσης.

Στο δεύτερο κουτί κώδικα ορίζουμε το μέγεθος του πίνακα n, τους πίνακες, στέλνουμε τα στοιχεία που θα χρειαστούμε για τους υπολογισμούς στην GPU, εκτελούμε τον πυρήνα, επιστρέφουμε το αποτέλεσμα στην CPU και εκτυπώνουμε το αποτέλεσμα.

3.3.6 Ατομικές διεργασίες και συνθήκες ανταγωνισμού.

Στην προσπάθειά μας να παραλληλίσουμε διεργασίες πολλές φορές οδηγούμαστε σε συνθήκες ανταγωνισμού (race conditions) στον κώδικά μας. Μια συνθήκη ανταγωνισμού στην CUDA προκύπτει όταν νήματα διαβάζουν ή γράφουν σε μια θέση μνήμης που μπορεί να τροποποιηθεί από ένα άλλο ανεξάρτητο νήμα. Γενικά, πρέπει να προσέχουμε δύο ειδών κινδύνους.

Πρώτον, τους κινδύνους ανάγνωσης μετά από εγγραφή (read after write hazards). Ένα νήμα διαβάζει μια θέση μνήμης την ίδια στιγμή που ένα άλλο νήμα μπορεί να γράφει σε αυτήν. Και δεύτερον κινδύνους εγγραφής μετά από εγγραφή (write after read hazards): Δύο νήματα γράφουν στην ίδια θέση μνήμης, και μόνο η μία εγγραφή θα είναι ορατή όταν ο πυρήνας ολοκληρωθεί.

Ένας τρόπος να αποφύγουμε τέτοιους κινδύνους είναι να οργανώσουμε έτσι το πρόγραμμά μας έτσι ώστε κάθε νήμα έχει πρόσβαση σε συγκεκριμένα στοιχεία ή να μην χρησιμοποιούν τον ίδιο πίνακα για είσοδο και για έξοδο σε μία μόνο κλήση πυρήνα.

Ωστόσο, υπάρχουν πολλές περιπτώσεις όπου διαφορετικά νήματα χρειάζεται να συνδυάσουν αποτελέσματα. Ένα τέτοιο απλό παράδειγμα είναι όταν έχουμε έναν καθολικό μετρητή counter όπου κάθε νήμα αυξάνει την τιμή του. Κατά την υλοποίηση ενός τέτοιου πυρήνα κάθε νήμα πρέπει να διαβάσει την τρέχουσα τιμή του counter, να υπολογίσει την τιμή counter + 1 και να γράψει αυτή την τιμή πίσω στην μνήμη. Ενώ το νήμα προσπαθεί να εκτελέσει αυτή την διαδικασία δεν υπάρχει καμία εγγύηση πως ένα άλλο νήμα δεν

έχει αλλάξει την τιμή του counter. Για την επίλυση τέτοιων προβλημάτων η CUDA παρέχει ατομικές πράξεις (atomic operations).

```
1. @cuda.jit
2. def thread_counter_race_condition(global_counter):
3.     global_counter[0] += 1
```

```
1. global_counter = cuda.to_device(np.array([0], dtype=np.int32))
2. thread_counter_race_condition[64, 64](global_counter)
3. print('Should be %d:' % (64*64), global_counter.copy_to_host())
```

Πρόγραμμα 18: Πρόγραμμα με κινδύνους ανταγωνισμού.

Αποτέλεσμα : Should be 4096: [1]

Περιγραφή κώδικα:

- Στο πρώτο κουτί όπου έχουμε τον πυρήνα, έχουμε μια συνάρτηση η οποία απλά διαχειρίζεται έναν μετρητή global_counter. Όταν θα εκκινήσουμε τον πυρήνα θέλουμε κάθε νήμα να προσθέσει μία μονάδα στον μετρητή.
- Στο δεύτερο κουτί έχουμε τον κώδικα που θα εκτελεσθεί στην CPU. Εδώ δημιουργούμε έναν πίνακα με ένα μόνο στοιχείο και τον στέλνουμε στην GPU. Στην συνέχεια εκτελούμε τον πυρήνα, με [64, 64] blocks per grid και threads per block, συνολικά 64 * 64 = 4096 threads.

Αυτό που θα περιμέναμε είναι κάθε ένα από τα νήματα να προσθέσει μια μονάδα στον μετρητή. Άρα στο τέλος ο μετρητής θα έπρεπε έχει την τιμή 4096. Αντί για αυτό η τιμή που μας επέστρεψε είναι 1. Αυτό συμβαίνει γιατί τα νήματα διαβάζουν όλα ταυτόχρονα την τιμή του μετρητή [0], προσθέτουν την τιμή [1] και προσπαθούν ταυτόχρονα να γράψουν ξανά. Οπότε η τιμή του μετρητή θα είναι πάντα 1.

Για να λύσουμε το πρόβλημα αυτό των συγκρούσεων θα γράψουμε τον πυρήνα μας ως εξής :

```
1. @cuda.jit
2. def thread_counter_safe(global_counter):
3.     cuda.atomic.add(global_counter, 0, 1)
```

```
1. global_counter = cuda.to_device(np.array([0], dtype=np.int32))
2. thread_counter_safe[64, 64](global_counter)
3. print('Should be %d:' % (64*64), global_counter.copy_to_host())
```

Πρόγραμμα 19: Πρόγραμμα χωρίς κινδύνους ανταγωνισμού.

Αποτέλεσμα: Should be 4096: [4096]

Περιγραφή κώδικα:

- Στο κουτί με τον πυρήνα μας τώρα την πράξη της πρόσθεσης με την χρήση ατομικών διεργασιών cuda.atomic.add(). Η συνάρτηση αυτή διασφαλίζει ότι τα νήματα δεν κάνουν ταυτόχρονα την πράξη της πρόσθεσης στον μετρητή.

3.3.7 Αποτελεσματική χρήση του συστήματος μνήμης

Είδαμε στο κεφάλαιο 3.2.2 πως ο ποιο αποτελεσματικός τρόπος χρήσης της μνήμης είναι όταν έχουμε πρόσβαση σε δεδομένα συνεχή (coalesced) στην μνήμη. Είδαμε επίσης πως αυτό γίνετε σε φυσικό επίπεδο στην μνήμη, στις περιπτώσεις που έχουμε άθροισμα γραμμών ενός πίνακα και άθροισμα στηλών. Ας δούμε ένα παράδειγμα που συγκρίνει την coalesced και μη χρήση μνήμης.

Άθροισμα πίνακα 2D

```
1. import numpy as np
2. from numba import cuda
3. n = 2048*2048
4. threads_per_block = (32, 32)
5. blocks = (64, 64)
6. a = np.arange(n).reshape(2048,2048).astype(np.float32)
7. b = a.copy().astype(np.float32)
8. out = np.zeros_like(a).astype(np.float32)
9. d_a = cuda.to_device(a)
10. d_b = cuda.to_device(b)
11. d_out = cuda.to_device(out)
```

```
1. @cuda.jit
2. def matrix_add(a, b, out, coalesced):
3.     x, y = cuda.grid(2)
4.     if coalesced == True:
5.         out[y][x] = a[y][x] + b[y][x]
6.     else:
7.         out[x][y] = a[x][y] + b[x][y]
```

```
1. %timeit matrix_add[blocks, threads_per_block](d_a, d_b, d_out, True); cuda.synchronize
```

203 µs ± 9.82 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)

```
1. result = d_out.copy_to_host()
2. truth = a+b
3. np.array_equal(result, truth)
```

True

```
1. %timeit matrix_add[blocks, threads_per_block](d_a, d_b, d_out, False); cuda.synchro-
nize
```

584 µs ± 2.16 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)

```
1. result = d_out.copy_to_host()
2. truth = a+b
3. np.array_equal(result, truth)
```

True

Πρόγραμμα 20: Άθροισμα 2D πίνακα με και χωρίς συνεχής(coalesced) μνήμη.

Περιγραφή κώδικα:

Στο πρώτο κουτί με το κυρίως πρόγραμμά μας

- Γραμμές 1-2: Εισάγουμε τις απαραίτητες βιβλιοθήκες.
- Γραμμή 3: Ορίζουμε το μέγεθος του πίνακα (n = 2048*2048).
- Γραμμή 4: Ορίζουμε τον αριθμό των νημάτων ανά block σε (32, 32). Αυτό σημαίνει ότι κάθε block θα έχει 32×32=1024 νήματα.
- Γραμμή 5: Ορίζουμε τον αριθμό των blocks σε (64, 64). Αυτό σημαίνει ότι το πλέγμα θα έχει 64×64=4096 blocks.
- Γραμμή 6: Δημιουργούμε έναν 2D πίνακα a μεγέθους 2048×2048×2048×2048 γεμάτο με αριθμούς από 0 έως n-1 και τον μετατρέπουμε σε np.float32.
- Γραμμή 7: Δημιουργούμε έναν αντίγραφο του πίνακα a και τον αποθηκεύουμε στον πίνακα b. Αυτός επίσης μετατρέπεται σε np.float32.
- Γραμμή 8: Δημιουργούμε έναν πίνακα out μεγέθους 2048×2048×2048×2048 γεμάτο με 0.0 και τον μετατρέπουμε σε np.float32. Αυτός θα αποθηκεύσει το αποτέλεσμα των πράξεων.
- Γραμμές 9-11: Μεταφέρουμε τους πίνακες a, b, και out στη μνήμη της GPU χρησιμοποιώντας την cuda.to_device.

Στον πυρήνα μας ο κώδικας είναι λίγο διαφορετικός αυτή την φορά, για να μας βοηθήσει να δούμε την διαφορά της coalesced και not coalesced χρήσης μνήμης.

- Γραμμή 3: Υπολογισμός δείκτη νήματος.
- Γραμμή 4: Ελέγχουμε αν η παράμετρος coalesced είναι True. Αν είναι True, η πρόσβαση στη μνήμη θα είναι συγχωνευμένη (coalesced), δηλαδή βελτιστοποιημένη για απόδοση.Αν είναι False, η πρόσβαση στη μνήμη θα είναι μη συγχωνευμένη (non-coalesced), δηλαδή λιγότερο αποδοτική.

Όταν θα μετρήσουμε τον χρόνο εκτέλεσης με την χρήση της συνάρτησης timeit θα δούμε ότι για την coalesced memory ο χρόνος θα είναι 203 μs, ενώ για την not coalesced memory ο χρόνος θα είναι 584 μs. Παρατηρούμε μια σημαντική βελτίωση στον χρόνο εκτέλεσης.

3.3.8 Κοινή Μνήμη (Shared Memory)

Μέχρι τώρα όλα τα προγράμματα που έχουμε γράψει, τα οποία εκτελούνται στην GPU κάνουν χρήση της καθολικής μνήμης. Σε προηγούμενο κεφάλαιο είδαμε πως μπορεί να μας βοηθήσει η χρήση μια άλλη περιοχή της μνήμης της συσκευής, η κοινόχρηστη μνήμη (σελ. 52). Εν συντομία, η μνήμη αυτή είναι ένας περιορισμένος πόρος, δεν μπορεί να προσπελαστεί από νήματα εκτός του block στο οποίο έχει εκχωρηθεί και δεν διατηρείται μετά την ολοκλήρωση της εκτέλεσης ενός πυρήνα. Ωστόσο, η κοινόχρηστη μνήμη έχει πολύ υψηλότερο εύρος ζώνης (bandwidth) σε σύγκριση με την καθολική μνήμη και μπορεί να χρησιμοποιηθεί με μεγάλη αποτελεσματικότητα σε πολλούς πυρήνες.

Η Numba παρέχει συναρτήσεις για την εκχώρηση κοινόχρηστης μνήμης (shared memory), καθώς και για τον συγχρονισμό μεταξύ των νημάτων σε ένα block, κάτι που συχνά είναι απαραίτητο αφού νήματα διαβάζουν ή γράφουν παράλληλα στην κοινόχρηστη μνήμη.

Όταν δηλώνουμε κοινόχρηστη μνήμη, παρέχουμε την μεταβλητή (shape) του κοινόχρηστου πίνακα, καθώς και τον τύπο (type) του, χρησιμοποιώντας έναν τύπο της Numba. Το σχήμα του πίνακα πρέπει να είναι μια σταθερή τιμή, και επομένως, δεν μπορούμε να χρησιμοποιήσουμε ορίσματα που περνάνε στη συνάρτηση ή μεταβλητές όπως η numba.cuda.blockDim.x, ή τις υπολογισμένες τιμές της cuda.gridDim.

Ας δούμε ένα παράδειγμα εναλλαγής στοιχείων ενός πίνακα με βάση την διαγώνιο, όπως το είδαμε να συμβαίνει πρακτικά σε προηγούμενο σημείο του κεφαλαίου (σελ.53).

```
1. import numpy as np
2. from numba import types, cuda
3. @cuda.jit
4. def swap_with_shared(vector, swapped):
5.     temp = cuda.shared.array(4, dtype=types.int32)
6.     idx = cuda.grid(1)
7.     temp[idx] = vector[idx]
8.     cuda.syncthreads()
9.     swapped[idx] = temp[3 - cuda.threadIdx.x]
```

```
1. vector = np.arange(4).astype(np.int32)
2. swapped = np.zeros_like(vector)
3. d_vector = cuda.to_device(vector)
4. d_swapped = cuda.to_device(swapped)
5. vector
```

```
array([0, 1, 2, 3], dtype=int32)
```

```
1. swap_with_shared[1, 4](d_vector, d_swapped)
```

```
2. result = d_swapped.copy_to_host()
3. result
```

```
array([3, 2, 1, 0], dtype=int32)
```

Πρόγραμμα 21: Εναλλαγή στοιχείων πίνακα με χρήση κοινής μνήμης.

Περιγραφή κώδικα:

Στον πυρήνα CUDA στο πρώτο κουτί ορίζουμε την συνάρτηση swap_with_shared().

Γραμμή 5: Δημιουργούμε έναν πίνακα κοινόχρηστης μνήμης (shared memory) με όνομα temp και μέγεθος 4, τύπου int32.

Γραμμή 6: Υπολογίζουμε τον καθολικό δείκτη idx του νήματος.

Γραμμή 7: Κάθε νήμα αντιγράφει το στοιχείο του διανύσματος vector[idx] στον πίνακα κοινόχρηστης μνήμης temp[idx].

Γραμμή 8: Καλούμε την cuda.syncthreads() για να διασφαλίσουμε ότι όλα τα νήματα έχουν ολοκληρώσει την αντιγραφή των δεδομένων στην κοινόχρηστη μνήμη πριν προχωρήσουμε. Αυτό είναι απαραίτητο για τη σωστή συγχρονισμό των νημάτων.

Γραμμή 9: Κάθε νήμα διαβάζει το στοιχείο από την κοινόχρηστη μνήμη σε αντεστραμμένη σειρά (temp[3 - cuda.threadIdx.x]) και το αποθηκεύει στο διάνυσμα εξόδου swapped[idx].

3.4 Εγκατάσταση CUDA Numba

Αρχικά το λειτουργικό σύστημα δεν είναι τόσο σημαντικό αφού λειτουργεί σε όλες τις εκδόσεις Windows από τα 7 και μετά. Σε macOS από τις εκδόσεις 10.9 και μετέπειτα και σχεδόν σε όλα τα Linux. Θα χρειαστούμε όμως έκδοση Python 3.4 τουλάχιστον καθώς και Numpy 1.10.

- Βεβαιωθείτε ότι η GPU σας υποστηρίζει CUDA. Δείτε στοιχεία για την κάρτα γραφικών σας στο site της NVIDIA (https://www.nvidia.com).
- Στην συνέχεια, κάντε εγκατάσταση του NVIDIA CUDA Toolkit (https://developer.nvidia.com/cuda-downloads). Κατεβάστε το toolkit από την ιστοσελίδα που δίνεται και ακολουθήστε τις οδηγίες εγκατάστασης για το λειτουργικό σας. Μετά την εγκατάσταση, επαληθεύστε ότι εγκαταστάθηκε επιτυχώς με την εντολή *nvcc –version*
- Η Numba είναι βιβλιοθήκη της Python και μπορούμε εύκολα να την εγκαταστήσουμε με την χρήση του pip. Επομένως, εκτελούμε την εντολή *pip install numba.*

4 Προσομοίωση Κβαντικού Υπολογισμού

Ο υπολογισμός γενικού σκοπού σε μονάδες επεξεργασίας γραφικών (GPGPU) είναι μια τεχνολογία μετασχηματισμού που αξιοποιεί την παράλληλη επεξεργαστική ισχύ των GPU για την εκτέλεση εργασιών που παραδοσιακά χειρίζονται οι CPU. Σε αντίθεση με τις CPU, οι οποίες είναι βελτιστοποιημένες για διαδοχική επεξεργασία, οι GPU αποτελούνται από χιλιάδες μικρότερους πυρήνες που έχουν σχεδιαστεί για να χειρίζονται πολλαπλές λειτουργίες ταυτόχρονα. Αυτή η αρχιτεκτονική καθιστά τις GPU εξαιρετικά ισχυρές για φόρτους εργασίας που μπορούν να αναλυθούν σε μικρότερες, παράλληλες εργασίες, όπως μηχανική μάθηση, επιστημονικές προσομοιώσεις και ανάλυση μεγάλων δεδομένων.

Αξιοποιώντας την ικανότητα της GPU να επεξεργάζεται τεράστιες ποσότητες δεδομένων παράλληλα, η GPGPU επιτρέπει δραματικές επιταχύνσεις στον υπολογισμό, μειώνοντας συχνά τους χρόνους επεξεργασίας από ώρες σε λεπτά ή και δευτερόλεπτα. Αυτή η ικανότητα έχει καταστεί απαραίτητη σε τομείς όπως η τεχνητή νοημοσύνη, όπου η εκπαίδευση μοντέλων βαθιάς μάθησης απαιτεί τεράστια υπολογιστική ισχύ για την επεξεργασία μεγάλων συνόλων δεδομένων και την εκτέλεση πολύπλοκων μαθηματικών πράξεων.

Η GPGPU (General-Purpose computing on Graphics Processing Units) μπορεί να διαδραματίσει σημαντικό ρόλο στην προώθηση του κβαντικού προγραμματισμού και της έρευνας στον τομέα των κβαντικών υπολογιστών, παρά τις θεμελιώδεις διαφορές τους με τους κβαντικούς επεξεργαστές. Ο κβαντικός προγραμματισμός περιλαμβάνει την προσομοίωση και την ανάπτυξη αλγορίθμων για κβαντικούς υπολογιστές, οι οποίοι λειτουργούν με βάση αρχές όπως η υπέρθεση και η κβαντική εμπλοκή. Ωστόσο, η προσομοίωση κβαντικών συστημάτων σε κλασικό υλικό είναι εξαιρετικά υπολογιστικά απαιτητική, καθώς η πολυπλοκότητα αυξάνεται εκθετικά με τον αριθμό των κβιτών.

Οι GPUs, με την μαζικά παράλληλη αρχιτεκτονική τους, είναι ιδανικά προσαρμοσμένοι για την αντιμετώπιση αυτών των απαιτητικών προσομοιώσεων, διασπώντας το πρόβλημα σε μικρότερες, παράλληλες εργασίες. Για παράδειγμα, η προσομοίωση της συμπεριφοράς ενός κβαντικού κυκλώματος ή η μοντελοποίηση κβαντικών καταστάσεων μπορεί να επιταχυνθεί χρησιμοποιώντας GPGPU, επιτρέποντας στους ερευνητές

να δοκιμάζουν και να διορθώνουν κβαντικούς αλγορίθμους πιο αποτελεσματικά πριν από την εκτέλεσή τους σε πραγματικό κβαντικό υλικό.

Επιπλέον, το GPGPU μπορεί να βοηθήσει στην υβριδική κβαντική-κλασική υπολογιστική, όπου κλασικοί υπολογιστικοί πόροι χρησιμοποιούνται για την υποστήριξη κβαντικών επεξεργαστών. Πολλοί κβαντικοί αλγόριθμοι, όπως αυτοί που χρησιμοποιούνται στην κβαντική χημεία ή σε προβλήματα βελτιστοποίησης, απαιτούν κλασική προ επεξεργασία, μετα-επεξεργασία ή επαναληπτική βελτίωση. Οι GPUs μπορούν να επιταχύνουν αυτά τα κλασικά στοιχεία, όπως πράξεις πινάκων ή υπολογισμούς κλίσεων, που συχνά αποτελούν εμπόδια σε υβριδικές ροές εργασίας. Επιπλέον, το GPGPU μπορεί να συμβάλει στην ανάπτυξη κωδίκων κβαντικής διόρθωσης σφαλμάτων και στην ανάλυση του κβαντικού θορύβου, που είναι κρίσιμα για την κατασκευή αξιόπιστων κβαντικών υπολογιστών. Με την αξιοποίηση του GPGPU, οι ερευνητές μπορούν να διευρύνουν τα όρια του κβαντικού προγραμματισμού, επιτρέποντας ταχύτερη δημιουργία πρωτοτύπων, βαθύτερη κατανόηση και πιο αποτελεσματική χρήση των περιορισμένων κβαντικών υπολογιστικών πόρων. Με αυτόν τον τρόπο, το GPGPU λειτουργεί ως γέφυρα μεταξύ της κλασικής και της κβαντικής υπολογιστικής, επιταχύνοντας την πρόοδο στον τομέα.

4.1 Python συναρτήσεις που προσομοιώνουν πύλες

Για τις πύλες των κβαντικών κυκλωμάτων μιλήσαμε αναλυτικά στο δεύτερο κεφάλαιο. Στο παρόν κεφάλαιο θα δούμε τις συναρτήσεις που αν καλεστούν μας επιστρέφουν τους αντίστοιχους πίνακες.

4.1.1 Hadamard

Η πύλη Hadamard είναι ο πίνακας:

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

```
1. # Hadamard Gate (H)
2. def hadamard_gate():
3.     return np.array([[1, 1],[1, -1]]) / np.sqrt(2)
```

```
1. import numpy as np
2. h = hadamard_gate()
3. print(h)
```

Πρόγραμμα 22: Η πύλη Hadamard.

Αποτέλεσμα:

[[0.70710678 0.70710678]

 [0.70710678 -0.70710678]]

Περιγραφή κώδικα:

Στο πρώτο κουτάκι έχουμε μια συνάρτηση η οποία δημιουργεί και επιστρέφει όταν αυτή καλεστεί ένα πίνακα μορφής numpy array, που αναπαριστά την πύλη Hadamard.

Έτσι όταν στο δεύτερο κουτάκι καλέσουμε την συνάρτηση και εκτυπώσουμε το αποτέλεσμα έχουμε τον παραπάνω πίνακα.

4.1.2 Πύλη Αδράνειας

Η πύλη αδράνειας δεν πραγματοποιεί καμία αλλαγή στην κατάσταση των qubits όμως είναι πολύ σημαντική κατά την δημιουργία κυκλωμάτων. Όταν εφαρμόζουμε μια πύλη σε ένα qubit ίσως πιστεύουμε πως εφαρμόσαμε μόνο αυτή, όμως στην πραγματικότητα εφαρμόσαμε την πύλη αδράνειας σε όλα τα εναπομείναντα qubit.

$$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

```
1. #Identity Gate (I)
2. def identity_gate():
3.     return np.eye(2, dtype=complex)
```

```
1. I = identity_gate()
2. print(I)
```

Πρόγραμμα 23: Πύλη Αδράνειας.

Αποτέλεσμα:

[[1.+0.j 0.+0.j]

 [0.+0.j 1.+0.j]]

4.1.3 Πύλη Pauli X

Η πύλη X όπως είδαμε στο δεύτερο κεφάλαιο όταν εφαρμοστεί σε ένα qubit αντιστρέφει την κατάσταση στην οποία βρίσκεται. Ο πίνακας που αντιπροσωπεύει την πύλη αυτή είναι:

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

```
# Pauli-X Gate
def pauli_x():
    return np.array([[0, 1], [1, 0]])
```

```
X = pauli_x()
print(X)
```

Πρόγραμμα 24: Πύλη Pauli X

Αποτέλεσμα:

[[0 1]

[1 0]]

4.1.4 Πύλη Pauli Y

Η πύλη Y επίσης αντιστρέφει την κατάσταση του qubit, όμως κατά τον άξονα y. Η μαθηματική αναπαράσταση της πύλης είναι:

$$Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$$

```
# Pauli-Y Gate
def pauli_y():
    return np.array([[0, -1j], [1j, 0]])
```

```
Y = pauli_y()
print(Y)
```

Πρόγραμμα 25: Πύλη Pauli Y

Αποτέλεσμα:

[[0.+0.j -0.-1.j]

[0.+1.j 0.+0.j]]

4.1.5 Πύλη Pauli Z

Μαθηματική αναπαράσταση πύλης Z.

$$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

```
# Pauli-Z Gate
def pauli_z():
    return np.array([[1, 0], [0, -1]])
```

```
1. Z = pauli_z()
2. print(Z)
```

Πρόγραμμα 26: Πύλη Pauli Z

Αποτέλεσμα:

[[1 0]

 [0 -1]]

4.1.6 Πύλη CNOT

Ο παρακάτω κώδικας υλοποιεί την κβαντική πύλη CNOT (Controlled – NOT) με την βοήθεια της CUDA ώστε να εκτελείτε παράλληλα στην GPU. Η CNOT πύλη εφαρμόζεται σε ένα κβαντικό καταχωρητή (quantum register) και ελέγχει την κατάσταση ενός qubit (control) για να εφαρμόσει την πύλη NOT σε ένα άλλο qubit (target). Ας δούμε τον κώδικα ποιο αναλυτικά:

```
1. def apply_CNOT(register, control, target, num_qubits):
2.     dim = 2 ** num_qubits
3.     result = np.zeros(dim, dtype=np.complex128)
4.     d_register = cuda.to_device(register)
5.     d_result = cuda.to_device(result)
6.     threads_per_block = 256
7.     blocks_per_grid = (dim + threads_per_block - 1)
8.     cnot_kernel[blocks_per_grid, threads_per_block](d_register, d_result, control, target,
num_qubits)
9.     return d_result.copy_to_host()
```

```
1. @cuda.jit
2. def cnot_kernel(register, result, control, target, num_qubits):
3.     idx = cuda.grid(1
4.     if idx >= len(register):
5.         raise ValueError(f"Out of bound.")
6.     control_mask = 1 << (num_qubits - control - 1)
7.     target_mask = 1 << (num_qubits - target - 1)
8.     if (idx & control_mask) != 0:
9.         flipped_idx = idx ^ target_mask
10.        temp = register[flipped_idx]
11.        result[idx] = temp
12.    else:
13.        result[idx] = register[idx]
```

Πρόγραμμα 27: Συνάρτηση εφαρμογής πύλης CNOT.

Περιγραφή κώδικα:

Συνάρτηση apply_CNOT()

- Η συνάρτηση apply_CNOT() εκτελείτε από την CPU. Δέχεται ως παραμέτρους:
  register: Το κβαντικό καταχωρητή (διάνυσμα καταστάσεων).
  control: Qubit ελέγχου (control qubit).
  target: Qubit στόχος (target qubit).
  num_qubits: Ο αριθμός των qubits στο σύστημα.
- Γραμμή 2: Υπολογίζεται η κατάσταση του διανύσματος καταστάσεων, που είναι 2^{num_qubits}, αφού κάθε qubit έχει 2 πιθανές καταστάσεις ($|0\rangle$ και $|1\rangle$).
- Γραμμή 3: Δημιουργείτε ένα κενό διάνυσμα result για να αποθηκευτεί το αποτέλεσμα μετά την εφαρμογή της πύλης.
- Γραμμή 4: Το διάνυσμα register μεταφέρεται από τη μνήμη της CPU στη μνήμη της GPU χρησιμοποιώντας τη συνάρτηση cuda.to_device.
- Γραμμή 5: Το διάνυσμα result μεταφέρεται επίσης στη μνήμη της GPU.
- Γραμμή 6: Ορίζεται ο αριθμός των threads ανά block (256 είναι μια συνηθισμένη τιμή).
- Γραμμή 7: Υπολογίζεται ο αριθμός των blocks που απαιτούνται για να καλύψουν όλα τα στοιχεία του διανύσματος. Ο τύπος εξασφαλίζει ότι όλα τα στοιχεία καλύπτονται, ακόμα και αν το dim δεν είναι πολλαπλάσιο του threads_per_block.
- Γραμμή 8: Καλείται ο πυρήνας (kernel) cnot_kernel στο GPU, με τα κατάλληλα blocks και threads. Τα ορίσματα του πυρήνα είναι τα διανύσματα d_register και d_result, καθώς και οι παράμετροι control, target και num_qubits.
- Γραμμή 9: Το αποτέλεσμα (d_result) μεταφέρεται από τη μνήμη της GPU πίσω στη μνήμη της CPU και επιστρέφεται από τη συνάρτηση.

Πυρήνας cnot_kernel()

- Γραμμή 1: Ορίζεται ο πυρήνας (kernel) cnot_kernel που θα εκτελεστεί στο GPU. Η σημείωση @cuda.jit δηλώνει ότι αυτή η συνάρτηση είναι ένα CUDA kernel.
- Γραμμή 2: Ορίζεται ο πυρήνας που δέχεται τα ίδια ορίσματα με τη συνάρτηση apply_CNOT.
- Γραμμή 3: Κάθε thread υπολογίζει το μοναδικό του δείκτη idx στον πίνακα, χρησιμοποιώντας τη συνάρτηση cuda.grid(1).
- Γραμμή 4: Ελέγχεται αν ο δείκτης idx είναι έγκυρος (δηλαδή, αν είναι μικρότερος από το μήκος του διανύσματος register).
- Γραμμή 5: Αν ο δείκτης είναι εκτός ορίων, τότε εγείρεται μια εξαίρεση.

- Γραμμή 6: Δημιουργείται μια μάσκα (control_mask) για να ελεγχθεί η κατάσταση του control qubit. Η μάσκα αντιστοιχεί σε μια δυαδική τιμή με 1 στη θέση του control qubit.
- Γραμμή 7: Δημιουργείται μια μάσκα (target_mask) για να εφαρμοστεί η NOT πύλη στο target qubit. Η μάσκα αντιστοιχεί σε μια δυαδική τιμή με 1 στη θέση του target qubit.
- Γραμμή 8: Ελέγχεται αν το control qubit είναι στην κατάσταση |1⟩. Αυτό γίνεται με τη λογική πράξη AND (&) μεταξύ του δείκτη idx και της μάσκας control_mask.
- Γραμμή 9: Αν το control qubit είναι |1⟩, τότε υπολογίζεται ο νέος δείκτης flipped_idx με την εφαρμογή της πύλης NOT στο target qubit. Αυτό γίνεται με τη λογική πράξη XOR (^) μεταξύ του δείκτη idx και της μάσκας target_mask.
- Γραμμή 10: Η τιμή του διανύσματος register στον δείκτη flipped_idx αποθηκεύεται στη μεταβλητή temp.
- Γραμμή 11: Η τιμή temp αποθηκεύεται στο διάνυσμα result στον δείκτη idx.
- Γραμμή 12: Αν το control qubit είναι στην κατάσταση |0⟩, τότε δεν εφαρμόζεται η NOT πύλη.
- Γραμμή 13: Η τιμή του διανύσματος register στον δείκτη idx αντιγράφεται στο διάνυσμα result χωρίς αλλαγές.

Παράδειγμα εκτέλεσης κώδικα:

```
1. n = 2
2. qr = np.array([0, 0, 1, 0])
3. qr = apply_CNOT(qr, control=0, target=1, num_qubits=n)
4. Print(qr)
```

Πρόγραμμα 28: Παράδειγμα εφαρμογής CNOT σε 2 qubits.

Αποτέλεσμα:

[0.+0.j 0.+0.j 0.+0.j 1.+0.j]

Το qubit ήταν στην κατάσταση [0, 0, 1, 0] η οποία αντιστοιχεί στην κατάσταση |10⟩. Εφαρμόζοντας την πύλη CNOT με qubit ελέγχου το πρώτο που έχει την τιμή 1 περιμένουμε το αποτέλεσμα να είναι η αλλαγή του δεύτερου qubit από 0 σε 1. Όπως βλέπουμε το αποτέλεσμα είναι [0, 0, 0, 1] το οποίο όντος αντιστοιχεί στην κατάσταση |11⟩.

4.2 Συναρτήσεις για πράξεις σε πίνακες.

4.2.1 Πολλαπλασιασμός πινάκων

```
 1. def multiply(matrix, vector):
 2.     rows, cols = matrix.shape
 3.     result = np.zeros(rows, dtype=np.complex128)
 4.     d_matrix = cuda.to_device(matrix)
 5.     d_vector = cuda.to_device(vector)
 6.     d_result = cuda.to_device(result)
 7.     threads_per_block = 256
 8.     blocks_per_grid = (rows + threads_per_block - 1)
 9.     matrix_vector_multiply_gpu[blocks_per_grid, threads_per_block](d_matrix, d_vector,
d_result)
10.     return d_result.copy_to_host()
```

```
1. @cuda.jit
2. def matrix_vector_multiply_gpu(matrix, vector, result):
3.     row = cuda.grid(1)
4.     if row < matrix.shape[0]:
5.         sum_val = 0 + 0j
6.         for j in range(matrix.shape[1]):
7.             sum_val += matrix[row, j] * vector[j]
8.         result[row] = sum_val
```

```
1. matrix = np.array([[1, 0], [0, 1]], dtype=np.complex128)
2. vector = np.array([1, 0], dtype=np.complex128)
3. gpu_result = matrix_vector_multiply(matrix, vector)
4. print(gpu_result)
```

Πρόγραμμα 29: Πολλαπλασιασμός πινάκων για εφαρμογή πυλών.

Αποτέλεσμα:

[1.+0.j 0.+0.j]

Περιγραφή κώδικα:

Συνάρτηση multiply()

- Γραμμή 1: Συνάρτηση multipy() εκτελείτε στην CPU. Δέχεται ως είσοδο τον πίνακα που θα πολλαπλασιαστεί (matrix) και το διάνυσμα που θα πολλαπλασιαστεί με τον πίνακα (vector).
- Γραμμή 2: Εξάγονται οι διαστάσεις του πίνακα (rows για τις γραμμές και cols για τις στήλες).

- Γραμμή 3: Δημιουργείται ένα κενό διάνυσμα result για να αποθηκευτεί το αποτέλεσμα του πολλαπλασιασμού. Ο τύπος του διανύσματος είναι complex128 (μιγαδικός αριθμός).
- Γραμμή 4: Ο πίνακας matrix μεταφέρεται από τη μνήμη της CPU στη μνήμη της GPU χρησιμοποιώντας τη συνάρτηση cuda.to_device.
- Γραμμή 5: Το διάνυσμα vector μεταφέρεται επίσης στη μνήμη της GPU.
- Γραμμή 6: Το διάνυσμα result μεταφέρεται στη μνήμη της GPU.
- Γραμμή 7: Ορίζεται ο αριθμός των threads ανά block (256 είναι μια συνηθισμένη τιμή).
- Γραμμή 8: Υπολογίζεται ο αριθμός των blocks που απαιτούνται για να καλύψουν όλες τις γραμμές του πίνακα. Ο τύπος εξασφαλίζει ότι όλες οι γραμμές καλύπτονται, ακόμα και αν ο αριθμός των γραμμών δεν είναι πολλαπλάσιο του threads_per_block.
- Γραμμή 9: Καλείται ο πυρήνας matrix_vector_multiply_gpu στο GPU, με τα κατάλληλα blocks και threads. Τα ορίσματα του πυρήνα είναι οι μεταφερθείσες μεταβλητές d_matrix, d_vector και d_result.
- Γραμμή 10: Το αποτέλεσμα (d_result) μεταφέρεται από τη μνήμη της GPU πίσω στη μνήμη της CPU και επιστρέφεται από τη συνάρτηση.

Συνάρτηση matrix_vector_multiply_gpu()

- Γραμμή 1: Ορίζεται ο πυρήνας (kernel) matrix_vector_multiply_gpu που θα εκτελεστεί στο GPU. Η σημείωση @cuda.jit δηλώνει ότι αυτή η συνάρτηση είναι ένα CUDA kernel.
- Γραμμή 2: Ορίζεται ο πυρήνας που δέχεται ως είσοδο:

  matrix: Ο πίνακας που θα πολλαπλασιαστεί.

  vector: Το διάνυσμα που θα πολλαπλασιαστεί με τον πίνακα.

  result: Το διάνυσμα όπου θα αποθηκευτεί το αποτέλεσμα.
- Γραμμή 3: Κάθε thread υπολογίζει το μοναδικό του δείκτη row στον πίνακα, χρησιμοποιώντας τη συνάρτηση cuda.grid(1). Αυτός ο δείκτης αντιστοιχεί σε μια γραμμή του πίνακα.
- Γραμμή 4: Ελέγχεται αν ο δείκτης row είναι έγκυρος (δηλαδή, αν είναι μικρότερος από τον αριθμό των γραμμών του πίνακα).

- Γραμμή 5: Αρχικοποιείται μια μεταβλητή sum_val για να αποθηκευτεί το μερικό άθροισμα του πολλαπλασιασμού γραμμής-διανύσματος. Η μεταβλητή είναι τύπου complex (μιγαδικός αριθμός).
- Γραμμή 6: Ξεκινάει ένας βρόχος που διασχίζει όλες τις στήλες του πίνακα.
- Γραμμή 7: Κάθε στοιχείο της γραμμής row του πίνακα πολλαπλασιάζεται με το αντίστοιχο στοιχείο του διανύσματος, και το αποτέλεσμα προστίθεται στο sum_val.
- Γραμμή 8: Το τελικό άθροισμα αποθηκεύεται στο διάνυσμα result στη θέση row.

Στην συνέχεια στο κυρίως πρόγραμμα δημιουργούμε έναν πίνακα και ένα διάνυσμα, καλούμε την συνάρτησή μας και εκτυπώνουμε το αποτέλεσμα. Ο πίνακας αντιστοιχεί σε αυτόν της πύλης αδράνειας οπότε το διάνυσμα που αναπαριστά το qubit [1, 0] = |0⟩ παραμένει το ίδιο.

4.2.2 Τανυστικό γινόμενο

Δύο λόγια για το τανυστικό γινόμενο (tensor or Kronecker product), όπως και ο πολλαπλασιασμός πινάκων, είναι μία βασική πράξη στην γραμμική άλγεβρα για την κβαντική υπολογιστική. Συνδυάζει δύο διανύσματα ή πίνακες για να δημιουργήσει έναν νέο τανυστή (tensor) μεγαλύτερης διάστασης. Στην κβαντική υπολογιστική, το τανυστικό γινόμενο χρησιμοποιείται για να συνδυάσει κβαντικές καταστάσεις ή τελεστές.

```
1. def kronecker_product_gpu(A, B):
2.     m, n = A.shape
3.     p, q = B.shape
4.     result = np.zeros((m * p, n * q), dtype=np.complex128)
5.     d_A = cuda.to_device(A)
6.     d_B = cuda.to_device(B)
7.     d_result = cuda.to_device(result)
8.     threads_per_block = (32, 32)
9.     blocks_per_grid_x = (m * p + threads_per_block[0] - 1) // threads_per_block[0]
10.    blocks_per_grid_y = (n * q + threads_per_block[1] - 1) // threads_per_block[1]
11.    blocks_per_grid = (blocks_per_grid_x, blocks_per_grid_y)
12.    kronecker_kernel[blocks_per_grid, threads_per_block](d_A, d_B, d_result)
13.    return d_result.copy_to_host()
```

```
1. @cuda.jit
2. def kronecker_kernel(A, B, result):
3.     row, col = cuda.grid(2)
4.     p, q = B.shape
5.     if row < m * p and col < n * q:
6.         row_A = row // p
7.         col_A = col // q
```

```
8.        row_B = row % p
9.        col_B = col % q
10.       result[row, col] = A[row_A, col_A] * B[row_B, col_B]
```

```
1. A = np.array([[1, 0], [0, 1]], dtype=np.complex128)
2. B = np.array([[0, 1], [1, 0]], dtype=np.complex128)
3. gpu_result = kronecker_product_gpu(A, B)
4. print(gpu_result)
```

Πρόγραμμα 30: Τανυστικό γινόμενο.

Αποτέλεσμα:

[[0.+0.j 1.+0.j 0.+0.j 0.+0.j]

 [1.+0.j 0.+0.j 0.+0.j 0.+0.j]

 [0.+0.j 0.+0.j 0.+0.j 1.+0.j]

 [0.+0.j 0.+0.j 1.+0.j 0.+0.j]]

Περιγραφή κώδικα:

Συνάρτηση Kronecker_product_gpu()

- Γραμμή 1: Ορίζεται η συνάρτηση kronecker_product_gpu που δέχεται ως είσοδο δύο πίνακες A και B.
- Γραμμή 2: Εξάγονται οι διαστάσεις του πίνακα AA: m γραμμές και n στήλες.
- Γραμμή 3: Εξάγονται οι διαστάσεις του πίνακα B: p γραμμές και q στήλες.
- Γραμμή 4: Δημιουργείται ένας κενός πίνακας result για να αποθηκευτεί το αποτέλεσμα του Kronecker γινομένου. Ο πίνακας έχει διαστάσεις (m×p)×(n×q) και είναι τύπου complex128 (μιγαδικός αριθμός).
- Γραμμές 5 και 6: Οι πίνακες μεταφέρονται στην GPU
- Γραμμή 7: Ο πίνακας result μεταφέρεται στη μνήμη της GPU.
- Γραμμή 8: Ορίζεται ο αριθμός των threads ανά block ως 32×32.
- Γραμμή 9: Υπολογίζεται ο αριθμός των blocks κατά μήκος του άξονα x (γραμμές). Ο τύπος εξασφαλίζει ότι όλες οι γραμμές καλύπτονται, ακόμα και αν ο αριθμός των γραμμών δεν είναι πολλαπλάσιο του threads_per_block[0].
- Γραμμή 10: Υπολογίζεται ο αριθμός των blocks κατά μήκος του άξονα y (στήλες). Ο τύπος εξασφαλίζει ότι όλες οι στήλες καλύπτονται.
- Γραμμή 11: Ορίζεται το 2D grid των blocks.

- Γραμμή 12: Καλείται ο πυρήνας kronecker_kernel στο GPU, με τα κατάλληλα blocks και threads. Τα ορίσματα του πυρήνα είναι οι μεταφερθείσες μεταβλητές d_A, d_B και d_result.
- Γραμμή 13: Το αποτέλεσμα (d_result) μεταφέρεται από τη μνήμη της GPU πίσω στη μνήμη της CPU και επιστρέφεται από τη συνάρτηση.

Πυρήνας Kronecker_kernel()

- Γραμμή 3: Κάθε thread υπολογίζει το μοναδικό του δείκτη (row, col) στον πίνακα, χρησιμοποιώντας τη συνάρτηση cuda.grid(2).
- Γραμμή 4: Εξάγονται οι διαστάσεις του πίνακα B: pp γραμμές και qq στήλες.
- Γραμμή 5: Ελέγχεται αν οι δείκτες row και col είναι έγκυροι (δηλαδή, αν είναι μικρότεροι από τις διαστάσεις του πίνακα result).
- Γραμμή 6: Υπολογίζεται η γραμμή του πίνακα A που αντιστοιχεί στο τρέχον row του result.
- Γραμμή 7: Υπολογίζεται η στήλη του πίνακα A που αντιστοιχεί στο τρέχον col του result.
- Γραμμή 8: Υπολογίζεται η γραμμή του πίνακα B που αντιστοιχεί στο τρέχον row του result.
- Γραμμή 9: Υπολογίζεται η στήλη του πίνακα B που αντιστοιχεί στο τρέχον col του result.
- Γραμμή 10: Το στοιχείο του πίνακα result στη θέση (row, col) υπολογίζεται ως το γινόμενο των αντίστοιχων στοιχείων των πινάκων A και B.

Στο κυρίως πρόγραμμα δημιουργούμε δύο πίνακες, ο πρώτος είναι ο μοναδιαίος και ο δεύτερος αυτός της πύλης X. Και οι δύο πίνακες είναι μεγέθους 2x2 επομένως ο πίνακας που θα παραχθεί θα είναι 4x4.

4.2.3 Συνάρτηση εφαρμογής πυλών

```
1. def apply_gate_to_qubit(register, gate, qubit_index, num_qubits):
2.     I = np.eye(2, dtype=complex)  # Identity matrix
3.     full_gate = np.array([[1]])
4.     for i in range(num_qubits):
5.         if i == qubit_index:
6.             full_gate = kronecker_product_gpu(full_gate, gate)
7.         else:
8.             full_gate = kronecker_product_gpu(full_gate, I)
9.     return gpu_matrix_vector_multiply(full_gate, register.reshape(-1))
```

Πρόγραμμα 31: Εφαρμογή οποιαδήποτε πύλης στο qubit που επιλέγουμε.

Περιγραφή κώδικα:

- Γραμμή 1: Ορίζεται η συνάρτηση apply_gate_to_qubit που δέχεται ως είσοδο:
  register: Το διάνυσμα κατάστασης του κβαντικού καταχωρητή.
  gate: Η κβαντική πύλη (2x2 πίνακας) που θα εφαρμοστεί.
  qubit_index: Ο δείκτης του qubit στο οποίο θα εφαρμοστεί η πύλη.
  num_qubits: Ο συνολικός αριθμός των qubits στον καταχωρητή.
- Γραμμή 2: Καλούμε την συνάρτηση που μας επιστρέφει τον μοναδιαίο πίνακα.
- Γραμμή 3: Αρχικοποιείται ο τελεστής full_gate ως ένας 1x1 πίνακας με τιμή 1. Αυτός θα επεκταθεί σταδιακά μέσω τανυστικού γινομένου.
- Γραμμή 4: Ξεκινάει ένας βρόχος ο οποίος θα προσπελάσει όλα τα qubits.
- Γραμμή 5: Ελέγχεται αν ο τρέχων δείκτης i αντιστοιχεί στο qubit στο οποίο θέλουμε να εφαρμόσουμε την πύλη.
- Γραμμή 6: Αν i=qubit index, τότε η πύλη gate εφαρμόζεται στο qubit αυτό, και ο τελεστής full_gate επεκτείνεται με τανυστικό γινόμενο.
- Γραμμή 7: Αν i≠qubitindex τότε εφαρμόζεται η ταυτοτική πύλη I στο qubit
- Γραμμή 8: Ο τελεστής full_gate επεκτείνεται με τανυστικό γινόμενο με τον μοναδιαίο πίνακα I.
- Γραμμή 9: Επιστρέφεται το αποτέλεσμα του πολλαπλασιασμού του τελεστή full_gate με το διάνυσμα κατάστασης register. Το register αναδιαμορφώνεται σε ένα μονοδιάστατο διάνυσμα (flattened) για να είναι συμβατό με τη συνάρτηση gpu_matrix_vector_multiply.

4.3 Καταχωρητές και κυκλώματα

4.3.1 Δημιουργία καταχωρητή με n Qubits

Η συνάρτηση δημιουργίας ενώ καταχωρητή εκτελείτε στην CPU όμως καλεί την συνάρτηση kronecker_product_gpu() η οποία θα εκτελεστεί στην GPU και θα μας επιστρέψει τα αποτελέσματα.

```
1. def create_Quantum_register(n, initial_states=None):
2.     if n < 1:
3.         raise ValueError("Number of qubits must be at least 1.")
4.     if initial_states is None:
5.         initial_states = [np.array([[1], [0]], dtype=complex) for _ in range(n)]
```

```
    else:
        if len(initial_states) != n:
            raise ValueError(f"Length of initial_states must be equal to n ({n}).")
        for state in initial_states:
            if not isinstance(state, np.ndarray) or state.shape != (2, 1):
                raise ValueError("Each initial state must be a 2×1 column vector.")
            if not np.isclose(np.linalg.norm(state), 1.0):
                raise ValueError("All initial states must be normalized (norm = 1).")
    state_vector = initial_states[0]
    for i in range(1, n):
        state_vector = kronecker_product_gpu(state_vector, initial_states[i])
    return state_vector
```

```
n = 2
qr = create_Quantum_register(n)
print(qr)
```

Πρόγραμμα 32: Δημιουργία Κβαντικού καταχωρητή.

Αποτέλεσμα:

[[1.+0.j]

[0.+0.j]

[0.+0.j]

[0.+0.j]]

Περιγραφή κώδικα:

- Γραμμή 1: Ορίζεται η συνάρτηση create_Quantum_register που δέχεται ως είσοδο, n: Ο αριθμός των qubits και initial_states: Μια λίστα με τις αρχικές καταστάσεις των qubits (προαιρετικό).
- Γραμμή 2: Ελέγχεται αν ο αριθμός των qubits n είναι τουλάχιστον 1. Αν όχι, παρουσιάζεται μήνυμα.
- Γραμμή 3: Αν δεν παρέχεται η παράμετρος initial_states, τότε αρχικοποιούνται όλα τα qubits στην κατάσταση $|0\rangle$ (δηλαδή $\begin{bmatrix}1\\0\end{bmatrix}$).
- Γραμμή 4: Αν παρέχεται η παράμετρος initial_states, τότε ελέγχεται η εγκυρότητά της.
- Γραμμή 5: Ελέγχεται αν το μήκος της λίστας initial_states είναι ίσο με τον αριθμό των qubits n. Αν όχι επιστρέφει μήνυμα σφάλματος.
- Γραμμή 6: Για κάθε κατάσταση στην λίστα initial_states, ελέγχεται η εγκυρότητά της.

- Γραμμή 7: Ελέγχεται αν κάθε κατάσταση είναι ένας πίνακας NumPy με διαστάσεις 2×1 (δηλαδή, ένα διάνυσμα στήλης). Αν όχι επιστρέφει μήνυμα σφάλματος.
- Γραμμή 8: Ελέγχεται αν κάθε κατάσταση είναι κανονικοποιημένη (δηλαδή, αν η νόρμα της είναι 1). Αν όχι επιστρέφει μήνυμα σφάλματος.
- Γραμμή 9: Αρχικοποιείται το διάνυσμα κατάστασης state_vector με την κατάσταση του πρώτου qubit.
- Γραμμή 10: Ξεκινάει ένας βρόχος για να συνδυάσει τις καταστάσεις των υπόλοιπων qubits.
- Γραμμή 11: Το διάνυσμα κατάστασης state_vector ενημερώνεται με το τανυστικό γινόμενο (Kronecker product) του τρέχοντος state_vector και της κατάστασης του επόμενου qubit (initial_states[i]). Η συνάρτηση kronecker_product_gpu υπολογίζει αυτό το γινόμενο στο GPU.
- Γραμμή 12: Επιστρέφεται το τελικό διάνυσμα κατάστασης του κβαντικού καταχωρητή.

Στο κυρίως πρόγραμμα καλούμε την συνάρτηση για δύο qubits (n =2) χωρίς να δώσουμε παραμέτρους οπότε ορίζονται στην κατάσταση 0. Όταν εκτελεσθεί η συνάρτηση δημιουργίας του κβαντικού καταχωρητή περιμένουμε το αποτέλεσμα να είναι $|00\rangle = \begin{bmatrix}1\\0\\0\\0\end{bmatrix}$.

Βλέπουμε πως όντως αυτό είναι το αποτέλεσμα.

```
1. n = 2
2. initial_states = [
3.     np.array([[1], [0]], dtype=complex),
4.      np.array([[0], [1]], dtype=complex)]
5. qr = create_Quantum_register(n, initial_states)
6. print(qr)
```

Πρόγραμμα 33: Κβαντικός καταχωρητής στον οποίο ορίζω τις αρχικές καταστάσεις.

Αποτέλεσμα:

```
[[0.+0.j]
 [1.+0.j]
 [0.+0.j]
 [0.+0.j]]
```

Σε αυτή την περίπτωση ορίζω τις αρχικές καταστάσεις των δύο qubits. Το πρώτο είναι στην κατάσταση $|0\rangle$ και το δεύτερο στην κατάσταση $|1\rangle$. Το αποτέλεσμα που περιμένω είναι $|01\rangle = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$.

4.3.2 Δημιουργία κυκλώματος με εφαρμογή Hadamard

```
1. n = 2
2. qr = create_Quantum_register(n)
3. hadamard = hadamard_gate()
4. qr = apply_gate_to_qubit(qr, hadamard, 1, n)
5. print(qr)
```

Πρόγραμμα 34: Κύκλωμα με 2 qubits και εφαρμογή Hadamard.

Αποτέλεσμα:

[0.70710678+0.j 0.70710678+0.j 0.+0.j 0.+0.j]

Περιγραφή κώδικα:

Το πρόγραμμα αυτό δημιουργεί το κύκλωμα. Οι συναρτήσεις που καλούνται βρίσκονται παραπάνω.

- Γραμμή 1: Ορίζω τον αριθμό των qubits (n = 2).
- Γραμμή 2: Καλώ την συνάρτηση για την δημιουργία του κβαντικού καταχωρητή χωρίς να ορίσω αρχικές καταστάσεις για τα qubits.
- Γραμμή 3: Καλώ την συνάρτηση που μου επιστρέφει την πύλη Hadamard.
- Γραμμή 4: Καλώ την συνάρτηση που εφαρμόζει την πύλη στο κατάλληλο qubit.
- Γραμμή 5: Εκτυπώνω το αποτέλεσμα.

4.3.3 Δημιουργία κυκλώματος με εφαρμογή πυλών Pauli

```
1. n = 2
2. qr = create_Quantum_register(n)
3. print(qr)
```

[[1.+0.j]

[0.+0.j]

[0.+0.j]

[0.+0.j]]

```
1. qr = apply_gate_to_qubit(qr, x, 0, n)
2. qr = apply_gate_to_qubit(qr, x, 1, n)
3. print(qr)
```

[0.+0.j 0.+0.j 0.+0.j 1.+0.j]

```
1. qr = apply_gate_to_qubit(qr, z, 0, n)
2. print(qr)
```

[0.+0.j 0.+0.j 0.+0.j -1.+0.j]

```
1. qr = apply_gate_to_qubit(qr, y, 1, n)
2. print(qr)
```

[0.+0.j 0.+0.j 0.+1.j 0.+0.j]

Πρόγραμμα 35 Εφαρμογή πυλών Pauli X-Z-Y.

Περιγραφή κώδικα:

Στο κυρίως αυτό πρόγραμμα, ορίζουμε τον αριθμό των qubits σε 2 και κατασκευάζουμε έναν κβαντικό καταχωρητή με την κατάστασή τους στο 0. Άρα θα έχουμε $|00\rangle = \begin{bmatrix}1\\0\\0\\0\end{bmatrix}$.

Στο δεύτερο κουτί εφαρμόζουμε την πύλη X και στα δύο qubits. Οπότε η κατάσταση των qubits θα αλλάξει σε $|11\rangle$.

Στο επόμενο κουτί θα εφαρμόσουμε την πύλη Z μόνο στο πρώτο qubit. Δεν ξεχνώ πως όταν εφαρμόζω μια πύλη σε ένα qubit τότε σε όλα τα άλλα qubit θα εφαρμοστεί ο μοναδιαίος πίνακας. Μετά την εφαρμογή της Z πύλης θα έχω $-|11\rangle$.

Τέλος εφαρμόζω την πύλη Y και το αποτέλεσμα θα είναι $i|10\rangle$.

4.3.4 Δημιουργία κυκλώματος με εφαρμογή πύλης CNOT

Ένα φαινόμενο της κβαντομηχανικής είναι η κβαντική διεμπλοκή (quantum entanglement). Όταν αυτό εφαρμόζεται σε δύο qubits τότε έχουμε τις Bell states. Έχουμε τέσσερις τέτοιες καταστάσεις:

$|\varphi^+\rangle = \frac{1}{\sqrt{2}}(|00\rangle+|11\rangle)$

$|\varphi^-\rangle = \frac{1}{\sqrt{2}}(|00\rangle-|11\rangle)$

$|\psi^+\rangle = \frac{1}{\sqrt{2}}(|01\rangle+|10\rangle)$

$|\psi^-\rangle = \frac{1}{\sqrt{2}}(|01\rangle-|10\rangle)$

Ας δούμε την τελευταία κατάσταση με κώδικα:

Bell state $|\psi^-\rangle$

```
1. import numpy as np
2. from numba import cuda
3. n = 2
4. qr = create_Quantum_register(n)
5. hadamard = hadamard_gate()
6. qr = apply_gate_to_qubit(qr, hadamard, 0, n)
7. x = pauli_x()
8. qr = apply_gate_to_qubit(qr, x, 1, n)
9. z = pauli_z()
10. qr = apply_gate_to_qubit(qr, z, 0, n)
11. qr = apply_gate_to_qubit(qr, z, 1, n)
12. qr = apply_CNOT(qr, control=0,target=1, num_qubits=n )
13. print(qr)
```

Πρόγραμμα 36: Κατάσταση διεμπλοκής $|\psi^-\rangle$

Αποτέλεσμα:

[0.+0.j -0.70710678+0.j 0.70710678+0.j 0.+0.j]

Περιγραφή κώδικα:

Αρχικά έχουμε την εισαγωγή των βιβλιοθηκών. Ορίζουμε τον αριθμό των qubits και δημιουργούμε τον καταχωρητή.

Γραμμές 5 και 6: Καλούμε την συνάρτηση για την πύλη Hadamard και εφαρμόζουμε την πύλη στο πρώτο qubit.

Γραμμές 7 και 8: Καλούμε την πύλη X και την εφαρμόζουμε στο δεύτερο qubit.

Γραμμές 9,10 και 11: Καλούμε την πύλη Z και την εφαρμόζουμε και στα δύο qubits.

Γραμμή 12: Εφαρμόζουμε την πύλη CNOT

5 Συμπεράσματα

Στην παρούσα πτυχιακή εργασία, μελετήθηκε η προσομοίωση κβαντικών συστημάτων με τη χρήση της βιβλιοθήκης Numba. Παρουσιάστηκαν οι βασικές έννοιες του κβαντικού υπολογισμού, καθώς και οι θεμελιώδεις κβαντικές πύλες. Στη συνέχεια, αναλύθηκε η χρήση της Numba για τη βελτιστοποίηση της εκτέλεσης των κβαντικών προσομοιώσεων σε GPU, δείχνοντας πως η παράλληλη επεξεργασία μπορεί να προσφέρει σημαντικά οφέλη.

Κατά την εκπόνηση της εργασίας, προέκυψαν διάφορες προκλήσεις. Αρχικά, η πολυπλοκότητα των κβαντικών κυκλωμάτων και η αυξημένη απαιτούμενη μνήμη για την προσομοίωση πολλών qubits αποτέλεσαν βασικά τεχνικά εμπόδια. Επιπλέον, η διαχείριση και βελτιστοποίηση της Numba και των CUDA kernels δεν ήταν πάντα απλή, καθώς απαιτούσε βαθύτερη κατανόηση της αρχιτεκτονικής των GPU και του τρόπου με τον οποίο εκτελούνται οι παράλληλες διεργασίες.

Ένα σημαντικό συμπέρασμα που προκύπτει είναι ότι η προσομοίωση κβαντικών αλγορίθμων σε GPU αποτελεί μια πολλά υποσχόμενη προσέγγιση, αλλά υπάρχουν ακόμα πολλά περιθώρια βελτίωσης. Παρότι καταφέραμε να εφαρμόσουμε βασικές κβαντικές πύλες η υπολογιστική πολυπλοκότητα των πιο σύνθετων κβαντικών κυκλωμάτων παραμένει πρόκληση.

Για μελλοντική εργασία θα μπορούσαμε να ασχοληθούμε με περαιτέρω μελέτη πιο προχωρημένων κβαντικών αλγορίθμων, όπως ο αλγόριθμος Shor ή ο αλγόριθμος Grover, ώστε να διερευνηθεί πόσο αποδοτικά μπορούν να προσομοιωθούν σε GPU. Ένας άλλος τρόπος εξέλιξης την παρούσας εργασίας θα ήταν η σύγκριση της απόδοσης της Numba με άλλες GPU επιταχυνόμενες τεχνολογίες όπως η CuPy ή το TensorFlow Quantum, προκειμένου να αξιολογηθούν τα πλεονεκτήματα και τα μειονεκτήματα της κάθε προσέγγισης.

Συνοψίζοντας, η εργασία αυτή ανέδειξε την αξία της GPU στην προσομοίωση κβαντικών αλγορίθμων, αλλά και τις τεχνικές προκλήσεις που παραμένουν ανοιχτές για μελλοντική έρευνα. Η συνέχιση της μελέτης στον τομέα αυτό μπορεί να συμβάλει σημαντικά στη

βελτίωση των προσομοιώσεων, ενισχύοντας την κατανόηση και ανάπτυξη κβαντικών υ-πολογιστικών συστημάτων.

Βιβλιογραφία

Ηλίας Κ. Σάββας, & Μαρία Ε. Σαμπάνη. (2024). *ΚΒΑΝΤΙΚΗ ΥΠΟΛΟΓΙΣΤΙΚΗ από την θεωρία στην Πράξη*.

1. Introduction — CUDA C++ Programming Guide. (n.d.). Retrieved February 19, 2025, from https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#hardware-implementation

Ahmadzadeh, A., & Sarbazi-Azad, H. (2023). Fast and scalable quantum computing simulation on multi-core and many-core platforms. *Quantum Information Processing*, *22*(5). https://doi.org/10.1007/S11128-023-03955-W

Ahmadzadeh, A., & Sarbazi-Azad, H. (2024). Fast scalable and low-power quantum circuit simulation on the cluster of GPUs platforms. *Optical and Quantum Electronics*, *56*(10), 1646. https://doi.org/10.1007/s11082-024-07492-3

Akama, S. (2015). Elements of quantum computing: History, theories and engineering applications. In *Elements of Quantum Computing: History, Theories and Engineering Applications*. Springer International Publishing. https://doi.org/10.1007/978-3-319-08284-4

David B. Kirk, & Wen-mei W Hwu. (2010). *ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ ΜΑΖΙΚΑ ΠΑΡΑΛΛΗΛΩΝ ΕΠΕΞΕΡΓΑΣΤΩΝ* (ΓΚΙΖΟΠΟΥΛΟΣ ΔΗΜΗΤΡΗΣ, Ed.). Κλειδάριθμος.

Gutiérrez, E., Romero, S., Trenas, M. A., & Zapata, E. L. (2010). Quantum computer simulation using the CUDA programming model. *Computer Physics Communications*, *181*(2), 283–300. https://doi.org/10.1016/J.CPC.2009.09.021

Holm, H. H., Brodtkorb, A. R., & Saetra, M. L. (n.d.). *GPU Computing with Python: Performance, Energy Efficiency and Usability †*. https://doi.org/10.3390/computation8010004

Hughes, C., Isaacson, J., Perry, A., Ranbel, ·, Sun, F., & Turner, J. (n.d.). *Quantum Computing for the Quantum Curious*.

Numba documentation — Numba 0+untagged.1829.g8ec16ce.dirty documentation. (n.d.). Retrieved February 28, 2025, from https://numba.readthedocs.io/en/stable/

Peddie, J. (n.d.). *The History of the GPU - Steps to Invention*.