ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ
Πρόγραμμα Μεταπτυχιακών Σπουδών

Μηχανική Λογισμικού
για Διαδικτυακές & Φορητές Εφαρμογές

Κβαντικοί Υπολογισμοί με GPGPUs

ΜΕΤΑΠΤΥΧΙΑΚΗ ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Καραπάνου Μαρία (ΑΜ: Μ013123006)

Επιβλέπων: ***Ηλίας Σάββας, Καθηγητής***

ΛΑΡΙΣΑ 2025

UNIVERSITY OF THESSALY
Postgraduate Studies Program

Software Engineering
for Internet & Mobile Applications

Quantum Computing with GPGPUs

MASTER THESIS

Karapanou Maria (AM: M013123006)

Supervisor: ***Ilias Savvas, Professor***

LARISSA 2025

«Εγώ η Καραπάνου Μαρία δηλώνω υπεύθυνα ότι η παρούσα Πτυχιακή Εργασία με τίτλο «Κβαντικοί Υπολογισμοί με GPGPUs» είναι δική μου και βεβαιώνω ότι:

- *Σε όσες περιπτώσεις έχω συμβουλευτεί δημοσιευμένη εργασία τρίτων, αυτό επισημαίνεται με σχετική αναφορά στα επίμαχα σημεία.*
- *Σε όσες περιπτώσεις μεταφέρω λόγια τρίτων, αυτό επισημαίνεται με σχετική αναφορά στα επίμαχα σημεία. Με εξαίρεση τέτοιες περιπτώσεις, το υπόλοιπο κείμενο της πτυχιακής αποτελεί δική μου δουλειά.*
- *Αναφέρω ρητά όλες τις πηγές βοήθειας που χρησιμοποίησα.*
- *Σε περιπτώσεις που τμήματα της παρούσας πτυχιακής έγιναν από κοινού με τρίτους, αναφέρω ρητά ποια είναι η δική μου συνεισφορά και ποια των τρίτων.*
- *Γνωρίζω πως η λογοκλοπή αποτελεί σοβαρότατο παράπτωμα και είμαι ενήμερος(-η) για την επέλευση των νόμιμων συνεπειών»*

< υπογραφή >

Καραπάνου Μαρία

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή

Τόπος: .....

Ημερομηνία: .....

ΕΠΙΤΡΟΠΗ ΑΞΙΟΛΟΓΗΣΗΣ

1. .....
2. .....
3. .....

Περίληψη

Η ραγδαία ανάπτυξη του τομέα των κβαντικών υπολογιστών έχει δημιουργήσει την ανάγκη για αποτελεσματικά εργαλεία μελέτης. Παρότι η φυσική υλοποίηση κβαντικών συστημάτων βρίσκεται ακόμη σε πρώιμο στάδιο, η προσομοίωση κβαντικών κυκλωμάτων με τη βοήθεια κλασικών υπολογιστικών πόρων προσφέρει έναν εφικτό και πρακτικό τρόπο πειραματισμού και αξιολόγησης. Η εργασία αυτή εστιάζει στη δημιουργία συναρτήσεων που επιτρέπουν την υλοποίηση βασικών κβαντικών πυλών και κυκλωμάτων σε κάρτες γραφικών(GPU). Δημιουργήθηκαν συναρτήσεις που περιλαμβάνει βασικές λειτουργίες για την αναπαράσταση και εκτέλεση κβαντικών πυλών και κυκλωμάτων. Μέσα από παραδείγματα και δοκιμές, γίνεται αξιολόγηση της απόδοσής τους, ενώ παρέχεται και ένα απλό περιβάλλον γραμμής εντολών, ώστε να μπορούν να τη χρησιμοποιήσουν εύκολα και άτομα χωρίς προηγούμενη εμπειρία στον τομέα.

Abstract

The rapid development of the quantum computing field has created the need for effective study tools. Although the physical implementation of quantum systems is still at an early stage, the simulation of quantum circuits using classical computational resources offers a feasible and practical way for experimentation and evaluation. This work focuses on creating functions that enable the implementation of basic quantum gates and circuits on graphics cards (GPUs). Functions were developed that include fundamental operations for the representation and execution of quantum gates and circuits. Through examples and tests, their performance is evaluated, while a simple command-line interface is also provided so that it can be easily used by individuals without prior experience in the field.

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή της διπλωματικής μου εργασίας, κ. Σάββα Ηλία, για την καθοδήγηση και τις συμβουλές του σε όλα τα στάδια της εκπόνησης της εργασίας..

Ευχαριστώ επίσης όλους τους καθηγητές του μεταπτυχιακού προγράμματος για τις γνώσεις που μας μετέδωσαν και τη στήριξή τους καθ' όλη τη διάρκεια των σπουδών.

Τέλος, ευχαριστώ θερμά την οικογένεια και τους φίλους μου για τη στήριξη και την ενθάρρυνσή τους σε όλη τη διάρκεια των σπουδών μου.

Καραπάνου Μαρία

Ιούλιος 2025

Περιεχόμενα

1 Εισαγωγή

Η κβαντική μηχανική αναπτύχθηκε στις αρχές του 20ού αιώνα στο κλάδο της Φυσικής για να εξηγήσει τα ατομικά φαινόμενα. Η ιδέα να συνδυαστεί με την θεωρία της πληροφορίας υποδεικνύει ότι οι κβαντικοί υπολογισμοί μπορούν να ξεπεράσουν τα όρια της κλασικής λογικής [1].

Οι κβαντικοί υπολογιστές επιλύουν προβλήματα βασιζόμενοι στις αρχές της κβαντικής μηχανικής [2], [3]. Το βασικό στοιχείο τους είναι ότι αναπαριστούν τη πληροφορία με ένα διάνυσμα 2 διαστάσεων (qubit) και η εκτέλεση προγραμμάτων είναι οι πράξεις πινάκων. Με τη χρήση μεγάλου αριθμού qubits, οι πίνακες αυτοί είναι μεγάλοι σε μέγεθος, που έχει ως αποτέλεσμα τη μεγάλη απαίτηση σε υπολογιστικούς πόρους και χρόνο.

Πραγματικοί κβαντικοί υπολογιστές είναι ελάχιστοι και περιορισμένων δυνατοτήτων και δεν μπορούν να χρησιμοποιηθούν από την ευρύ επιστημονική κοινότητα. Συνεπώς, η χρήση εξομοιωτών είναι αναγκαία προκειμένου να μελετηθούν τα κβαντικά συστήματα και να αναπτυχθεί το κατάλληλο λογισμικό. Η προσομοίωση κβαντικών κυκλωμάτων και πυλών που μπορεί να υλοποιηθεί εύκολα με τη χρήση της τρέχουσας τεχνολογίας, καθώς περιλαμβάνει την εκτέλεση μαθηματικών πράξεων.

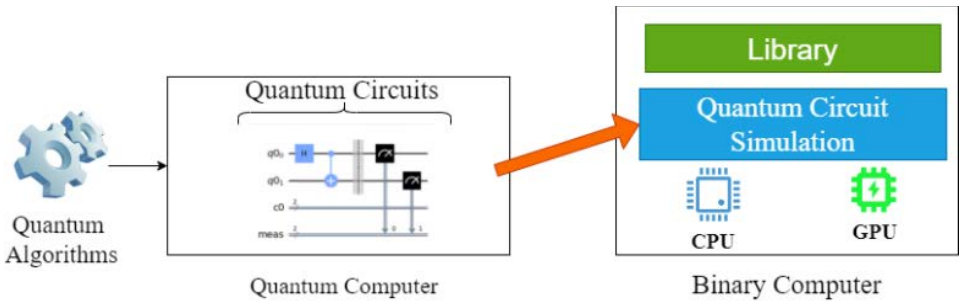
Υπάρχουν διαθέσιμα εργαλεία εξομοίωσης κυκλωμάτων κβαντικών υπολογιστών, τα οποία παρέχουν ένα προγραμματιστικό περιβάλλον για τη χρήση τους, όπως το Qiskit [4]. Τα εργαλεία αυτά δύναται να κάνουν χρήση GPGPU, αλλά ταυτόχρονα υπάρχουν και βιβλιοθήκες επιτάχυνσης κβαντικών υπολογισμών όπως η cuQuantum της NVIDIA [5].

Οι προσομοιωτές αυτοί είναι κλασσικές εφαρμογές λογισμικού που υλοποιούνται συνήθως ως σειριακά προγράμματα, τα οποία εκτελούνται σε επεξεργαστές γενικού σκοπού (CPUs). Με την ανάπτυξη πολυπύρηνων επεξεργαστών (Multi-core CPUs), επιτεύχθηκε η βελτίωση της απόδοσης μέσω παράλληλου προγραμματισμού, ο οποίος επιτρέπει την παράλληλη εκτέλεση πολλαπλών τμημάτων ενός προγράμματος, σε διαφορετικούς πυρήνες [6]. Οι παράλληλοι υπολογισμοί παρέχουν σημαντικό πλεονέκτημα απόδοσης για εξαιρετικά μεγάλες εφαρμογές σε πολλούς υπολογιστικούς τομείς [7]. Μια κατηγορία επεξεργαστών ειδικού σκοπού αποτελούν οι μονάδες επεξεργασίας γραφικών (Graphics

Processing Unit) οι οποίες αρχικά προορίζονταν αποκλειστικά για την απεικόνιση γραφικών, όμως πλέον χρησιμοποιούνται ως παράλληλοι επεξεργαστές για γενικούς υπολογιστικούς σκοπούς, καθώς πολλές εφαρμογές έχουν τροποποιηθεί για να αξιοποιούν τις GPU, επιτυγχάνοντας μεγάλες βελτιώσεις σε σχέση με τις αντίστοιχες υλοποιήσεις σε CPU [7] οδηγώντας έτσι στη δημιουργία του GPGPU (General-Purpose Computing on Graphics Processing Units).

Οι GPU σχεδιάστηκαν για να εκτελούν παράλληλους υπολογισμούς, όπου οι ίδιες εντολές εφαρμόζονται ταυτόχρονα σε πολλαπλά δεδομένα. Επίσης, επιτρέπει την κατάτμηση των προβλημάτων σε μικρότερα τμήματα, τα οποία εκτελούνται παράλληλα από τα νήματα της GPU, διευκολύνοντας την κλιμάκωση και την αποτελεσματική εκτέλεση των προγραμμάτων σε πολλούς επεξεργαστικούς πυρήνες [7]. Αυτή η τεχνολογία μπορεί να αξιοποιηθεί και στους εξομοιωτές κβαντικών υπολογιστών, προκειμένου να επιτευχθεί καλύτερη απόδοση.

Η εργασία αυτή επικεντρώνεται στην εξομοίωση κυκλωμάτων ενός κβαντικού υπολογιστή σε GPU, καθώς επίσης και στην αξιολόγησή τους από την πλευρά χρηστικότητά τους. Όπως φαίνεται και στο παρακάτω σχήμα στην Εικόνα 1.1, αναπτύχθηκε μια βιβλιοθήκη για να εκτελεί εξομοίωση λειτουργίας κβαντικών κυκλωμάτων GPU. Ο σκοπός θα είναι η ευκολότερη μελέτη και αξιολόγηση διαφορετικών αλγορίθμων, εξομοιωτών κτλ. Ταυτόχρονα, θα υλοποιηθεί και ένα απλό command line based interface, που καθιστά απλή τη χρήση της βιβλιοθήκης, και ως συνέπεια της εκτέλεσης εξομοιώσεων, από μη εξειδικευμένους χρήστες.



Εικόνα 1.1.: Τοποθέτηση προτεινόμενης λύσης ανάπτυξης βιβλιοθήκης.

2 Κβαντικοί Υπολογισμοί

Η κβαντική υπολογιστική είναι ένας νέος τομέας της επιστήμης και της υπολογιστικής μηχανικής που χρησιμοποιεί τις ιδιότητες της κβαντικής μηχανικής για την επίλυση προβλημάτων [8]. Εκμεταλλευόμενοι φαινόμενα της κβαντικής φυσικής, οι κβαντικοί υπολογιστές θα μπορούν να δίνουν λύσεις σε προβλήματα που οι κλασικοί υπολογιστές θα χρειάζονταν χρόνια για να υπολογίσουν [8], [2].Για παράδειγμα, ένας κβαντικός υπολογιστής θα μπορούσε να υλοποιήσει το σχεδιασμό και τη προσομοίωση νέων φαρμάκων σε πολύ μικρό χρονικό διάστημα σε σύγκριση με τα χρόνια που απαιτούνται μέχρι σήμερα για αυτό το σκοπό [8], [9]

Κορυφαίες εταιρίες ανά τον κόσμο δαπανούν μεγάλα χρηματικά ποσά για την έρευνα και ανάπτυξη κβαντικών υπολογιστών καθώς οι τομείς στους οποίου μπορούν να συμβάλουν σε πολλούς και διαφορικούς τομείς και όχι μόνο στον τομέα της πληροφορικής [8].

Στο κεφάλαιο αυτό παρουσιάζονται οι βασικές έννοιες των κβαντικών υπολογισμών και ο τρόπος με τον οποίο ένας κβαντικός υπολογιστής εκτελεί υπολογισμούς. Επίσης, παρατίθενται παραδείγματα που δείχνουν πώς προσομοιώνεται η λειτουργία ενός κβαντικού υπολογιστή.

2.1 Βασικές αρχές κβαντικής υπολογιστής

Οι κβαντικοί υπολογιστές επιλύουν προβλήματα με διαφορετικό τρόπο από ότι οι κλασικοί υπολογιστές. Η κβαντική υπολογιστική βασίζεται σε βασικές αρχές της κβαντικής μηχανικής [2], [3]:

- *Υπέρθεση (Superposition):* "ιδιότητα των σωματιδίων να βρίσκονται ταυτόχρονα σε πολλαπλές καταστάσεις"
- *Κβαντική διεμπλοκή (Quantum entanglement):* "μια κβαντική κατάσταση κατά τη οποία δύο σωματίδια ή ομάδες σωματιδίων αλληλοεπηρεάζονται ακαριαία"
- *Κβαντική τηλεμεταφορά*: "μετακίνηση κβαντικής πληροφορίας ακαριαία"
- *Θεώρημα μη κλωνοποίησης (No-cloning theorem):* "είναι αδύνατη η δημιουργία ενός αντίγραφου μίας άγνωστης κβαντικής κατάστασης"

- *Κβαντική Υπεροχή (Quantum supremacy):* "η επίλυση προβλημάτων από έναν κβαντικό υπολογιστή που δεν μπορούν να λυθούν από τους κλασσικούς υπολογιστές"
- Παρέμβαση (Interference): "επέμβαση στις κβαντικές πιθανότητες ώστε να οδηγήσουν στην σωστή απάντηση"
- Αποσυνοχή (Decoherence): "Η ιδιότητα να χανουν την πληροφορία τα qubits με την πάροδο του χρόνου".

2.2 Qubit

Στους υπολογιστές που χρησιμοποιούν το δυαδικό σύστημα η πληροφορία αναπαρίσταται από τα Bits 0 ή 1 ενώ οι κβαντικοί υπολογιστές χρησιμοποιούν τα *Qubits (Quantum Bit)*[10]. Ένα απλός υπολογιστής μπορεί να βρίσκεται στην κατάσταση 0 ή 1 σε αντίθεση με τους κβαντικούς οι οποίοι δύναται να βρίσκονται στην κατάσταση 0 ή/και 1 ταυτόχρονα (υπέρθεση) [3], [10]. Ο αριθμός των πιθανών καταστάσεων ενός κβαντικού υπολογιστή είναι 2n, όπου n ο αριθμός των qubits [10]. Μία κβαντική κατάσταση αναπαρίσταται ως το άθροισμα των βασικών κβαντικών καταστάσεων $|0\rangle$ και $|1\rangle$, όπου a2, b2 οι πιθανότητες της κβαντικής κατάστασης 0 και 1 αντίστοιχα [3]:

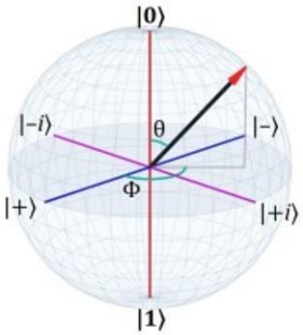
$$|0\rangle = \begin{pmatrix}1\\0\end{pmatrix} \qquad (2.1)$$

$$|1\rangle = \begin{pmatrix}0\\1\end{pmatrix} \qquad (2.2)$$

$$|x\rangle = \begin{pmatrix}a\\b\end{pmatrix} = a\begin{pmatrix}1\\0\end{pmatrix} + b\begin{pmatrix}0\\1\end{pmatrix} = a|0> + b|1> \ , \ a^2+b^2=1 \qquad (2.3)$$

Μετά από κάθε μέτρηση το qubit θα είναι αποκλειστικά σε μία από τις δύο θέσεις με πιθανότητες a και b και οι οποίες ονομάζονται πλάτη πιθανότητας [3]. Όταν τα πλάτη είναι μιγαδικοί αριθμοί δημιουργούν ένα χώρο Hilbert και αναπαρίστανται με τη σφαίρα Bloch (Εικόνα 2.1). Μια κβαντική κατάσταση είναι αποδεκτή όταν το άθροισμα των τετραγώνων των πλατών είναι ίσα με 1 [3], [11].

$$|q\rangle = cos\frac{\theta}{2}\,|0\rangle e^{i\varphi} sin\frac{\theta}{2}\,|1\rangle \ , \ \theta \text{ polar angle \& } \varphi \text{ azimuthal angle} \qquad (2.4)$$



Εικόνα 2.1. Σφαίρα Bloch [3]

Σύμφωνα με τον συμβολισμό Dirac, ένα qubit μπορεί να συμβολισθεί μαθηματικά με δυο τρόπους:

- **Bra:** πίνακας 1xn

  $$|0\rangle = \begin{bmatrix}1\\0\end{bmatrix}, \qquad |1\rangle = \begin{bmatrix}0\\1\end{bmatrix}, \qquad (2.5)$$

- **Ket**: πίνκας nx1

  $$\langle 0| = [1 \quad 0], \qquad \langle 1| = [0 \quad 1] \qquad (2.6)$$

2.3 Κβαντικός Καταχωρητής και Τανυστικό γινόμενο

Για την τοπική αποθήκευση της πληροφορίας και για την εκτέλεση των πράξεων σε έναν κλασσικό υπολογιστή χρησιμοποιείται ένας καταχωρητής ο οποίος είναι μια ακολουθία από bit 0 ή 1. Ένας κβαντικός υπολογιστής χρησιμοποιεί για το σκοπό αυτό κβαντικούς καταχωρητές οι οποίοι αποτελούνται από μια ακολουθία από qubits, επιτρέποντάς του έτσι να αναπαριστά περισσότερες από μια καταστάσεις ταυτόχρονα. Για παράδειγμα, ένας κλασσικός καταχωρητής 2 bits αποτελείται από τον συνδιασμό του 0 και του 1 (00 ή 01 ή 10 ή 11). Αντιθέτως, ένας κβαντικός καταχωρητής 2 qubits αποτελείται από το τανυστικό γινόμενο 2 qubits:

Για

$$|q1\rangle = \begin{bmatrix}a\\b\end{bmatrix} \;\&\; |q2\rangle = \begin{bmatrix}c\\d\end{bmatrix} \qquad (2.7)$$

$$|q1q2\rangle = |q1\rangle \otimes |q2 \geq \begin{bmatrix}a\\b\end{bmatrix} \otimes \begin{bmatrix}c\\d\end{bmatrix} = \begin{bmatrix}a*c\\a*d\\b*c\\b*d\end{bmatrix} \qquad (2.8)$$

Επομένως, ένας κβαντικός καταχωρητής 2 qubits αποτελείται από ένα διάνυσμα τεσσάρων διαστάσεων, το οποίο μπορεί να αναπαριστά τέσσερις διαφορετικές καταστάσεις ταυτόχρονα (βασική κβαντική αρχή της υπέρθεσης), σε αντίθεση με τον κλασσικό καταχωρητή που αναπαριστά μόνο μια κάθε φορά [3], [11], [12].

Εάν τα q1, q2 είναι τα βασικά qubits |0⟩,|1⟩ τότε οι βασικές καταστάσεις των 2 qubits θα είναι:

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

$$|01\rangle = |0\rangle \otimes |1\rangle = \begin{bmatrix}1\\0\end{bmatrix} \otimes \begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}1*0\\1*1\\0*0\\0*1\end{bmatrix} = \begin{bmatrix}0\\1\\0\\0\end{bmatrix} \qquad (2.9)$$

$$|10\rangle = |1\rangle \otimes |0\rangle = \begin{bmatrix}0\\1\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}0*1\\1*0\\1*1\\0*0\end{bmatrix} = \begin{bmatrix}0\\0\\1\\0\end{bmatrix}$$

$$|11\rangle = |1> \otimes |1\rangle = \begin{bmatrix}0\\1\end{bmatrix} \otimes \begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}0*0\\1*1\\0*0\\1*1\end{bmatrix} = \begin{bmatrix}0\\0\\0\\1\end{bmatrix}$$

Η εξίσωση 2.6 μπορεί να αναπαρασταθεί ως εξής:

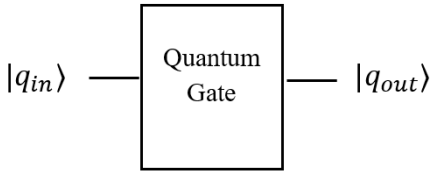
$$|q1q2\rangle = |q1\rangle \otimes |q2\rangle = \begin{bmatrix}a\\b\end{bmatrix} \otimes \begin{bmatrix}c\\d\end{bmatrix} = \begin{bmatrix}ac\\ad\\bc\\bd\end{bmatrix} = ac\begin{bmatrix}1\\0\\0\\0\end{bmatrix} + ad\begin{bmatrix}0\\1\\0\\0\end{bmatrix} + bc\begin{bmatrix}0\\0\\1\\0\end{bmatrix} + bd\begin{bmatrix}0\\0\\0\\1\end{bmatrix}$$

$$= ac|00\rangle + ad|01\rangle + bc|10\rangle + bd|10\rangle \qquad (2.10)$$

Για να είναι η παραπάνω κατάσταση αυτή αποδεκτή πρέπει $|ac|^2 + |ad|^2 + |bc|^2 + |bd|^2 = 1$. Δηλαδή, ο κβαντικός θα βρίσκεται στη κατάσταση |00⟩ με πιθανότητα $|ac|^2$, στην κατάσταση |01⟩ με πιθανότητα $|ad|^2$, στην κατάσταση |10⟩ με πιθανότητα $|bc|^2$ ή στην κατάσταση |10⟩ με πιθανότητα $|bd|^2$. Συνεπώς, ένας κβαντικός καταχωρητής οποίος αποτελείται από n qubits μπορεί να βρίσκεται σε 2^n διαφορετικές καταστάσεις ταυτόχρονα [4], [6].

2.4 Κβαντικές Πύλες

Οι λογικές πύλες σε έναν υπολογιστή δέχονται bits ως είσοδο και παράγει έξοδο βάσει συγκεκριμένων λογικών κανόνων. Αντίστοιχα, στους κβαντικούς υπολογιστές, οι κβαντικές πύλες δέχονται qubits, και μέσω μετασχηματισμών μεταβάλλουν την κβαντική κατάστασή τους, λαμβάνοντας υπόψη τις κβαντικές αρχές. Έτσι, αντί για καθορισμένες δυαδικές εξόδους, η έξοδος μπορεί να είναι μια υπέρθεση πολλαπλών καταστάσεων, καθιστώντας δυνατούς τους παράλληλους υπολογισμούς [13].



Εικόνα 2.2:Γενικός συμβολισμός κβαντικής πύλης

Η εφαρμογή μιας κβαντικής πύλης σε έναν κβαντικό καταχωρητή περιγράφεται μαθηματικά ως ο πολλαπλασιασμός του πίνακα της πύλης με το διάνυσμα κατάστασης του καταχωρητή.

Στην ενότητα αυτή, θα γίνει αναφορά σε ορισμένες βασικές κβαντικές πύλες, οι οποίες αποτελούν θεμελιώδη στοιχεία για την υλοποίηση κβαντικών υπολογισμών [3], [14].

2.4.1 Πύλη Αδράνειας (Identity)

Η πύλη αδρανείας (I) αντιστοιχεί σε έναν πίνακα 2x2 όπως φαίνεται στην εξ. 2.10 και δρα πάνω σε έναν καταχωρητή ενός qubit. Όταν η πυλή αυτή εφαρμόζεται, ο καταχωρητής διατηρεί την αρχική του κατάσταση [3], [11].

Εικόνα 2.3: Συμβολισμός πύλης I

$$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad (2.11)$$

$$Εάν\ |q\rangle = \begin{bmatrix} a \\ b \end{bmatrix}$$

$$I|q\rangle = \begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix} * \begin{bmatrix}a\\ b\end{bmatrix} = \begin{bmatrix}1*a+0*b\\ 0*a+1*b\end{bmatrix} = \begin{bmatrix}a\\ b\end{bmatrix} = |q\rangle \qquad (2.12)$$

2.4.2 Πύλη Υπέρθεσης (Hadamard)

Η πύλη υπέρθεσης (Η) αποτελείτε από έναν πίνακα 2x2 όπως φαίνεται στην εξ. 2.13 και εφαρμόζεται σε έναν καταχωρητή ενός qubit. Με την εφαρμογή της πύλης αυτής, ο καταχωρητής τίθεται σε κατάσταση υπέρθεσης.

Εικόνα 2.4: Συμβολισμός πύλης Η

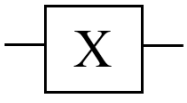
$$H = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix} \qquad (2.13)$$

$$Εάν\ |q\rangle = \begin{bmatrix}a\\ b\end{bmatrix}$$

$$H|q\rangle = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix} * \begin{bmatrix}a\\ b\end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix}(1*a)+(1*b)\\ (1*a)+(-1*b)\end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix}(a+b)\\ (a-b)\end{bmatrix} = \frac{1}{\sqrt{2}}(a+b)\begin{bmatrix}1\\ 0\end{bmatrix} + \frac{1}{\sqrt{2}}(a-b)\begin{bmatrix}0\\ 1\end{bmatrix} = \frac{1}{\sqrt{2}}(a+b)|0\rangle + \frac{1}{\sqrt{2}}(a-b)|1\rangle \qquad (2.14)$$

2.4.3 Πύλες Pauli (X, Y, Z)

Οι πύλες Pauli αποτελούνται από πίνακες 2x2 όπως φαίνεται στις εξισώσεις 2.15, 2.16, 2.17 και εκτελούνται πάνω σε έναν καταχωρητή ενός qubit. Η χρήση κάθε πύλης περιστρέφει το διάνυσμα κατάστασης του καταχωρητή πάνω στη σφαίρα Bloch (βλ. Εικόνα 2.1) γύρω από τους άξονες x, y, z κατά 180 μοίρες. Συγκεκριμένα, η πύλη X προκαλεί γύρω από τον άξονα των x, η πύλη Y γύρω από τον άξονα των y και η πύλη Z στον άξονα των z.



Εικόνα 2.5: Συμβολισμός Πύλης X

$$X = \begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix} \qquad (2.15)$$

Εικόνα 2.6: Συμβολισμός πύλης Y.

$$Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \qquad (2.16)$$

Εικόνα 2.7: Συμβολισμός Πύλης Z.

$$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \qquad (2.17)$$

$$Εάν\ |q\rangle = \begin{bmatrix} a \\ b \end{bmatrix}$$

$$X|q\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} * \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} (0*a)+(1*b) \\ (1*a)+(0*b) \end{bmatrix} = \begin{bmatrix} b \\ a \end{bmatrix} \qquad (2.18)$$

$$Y|q\rangle = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} * \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} (0*a)+(-i*b) \\ (i*a)+(0*b) \end{bmatrix} = \begin{bmatrix} -bi \\ ai \end{bmatrix} \qquad (2.19)$$

$$Z|q\rangle = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} * \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} (1*a)+(0*b) \\ (0*a)+(-1*b) \end{bmatrix} = \begin{bmatrix} a \\ -b \end{bmatrix} \qquad (2.20)$$

Ακολουθεί ένα παράδειγμα που δείχνει πώς δρουν οι πύλες Pauli όταν εφαρμόζονται στην κατάσταση $|0\rangle$.

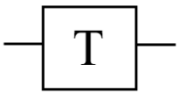
$$X|0\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} * \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} (0*1)+(1*0) \\ (1*1)+(0*0) \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} = |1>$$

$$Y|0\rangle = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} * \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} (0*1)+(-i*0) \\ (i*1)+(0*0) \end{bmatrix} = \begin{bmatrix} 0 \\ i \end{bmatrix} = |i>$$

$$Z|0\rangle = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} * \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} (1*0)+(0*0) \\ (0*1)+(-1*0) \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} = |1>$$

2.4.4 Πύλες Μετατόπισης Φάσης T & S

Οι πύλες μετατόπισης φάσης T και S εκφράζονται με πίνακες 2×2, εφαρμόζονται σε ένα qubit και προκαλούν περιστροφή γύρω από τον άξονα z της σφαίρας Bloch κατά γωνία π/4 και π/2, αντίστοιχα.



Εικόνα 2.8: Συμβολισμός πύλης T.

$$T = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix} \qquad (2.21)$$

Εικόνα 2.9: Συμβολισμός πύλης S.

$$S = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/2} \end{bmatrix} \qquad (2.22)$$

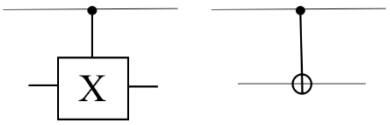
$$Εάν\ |q\rangle = \begin{bmatrix} a \\ b \end{bmatrix}$$

$$T|q\rangle = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/2} \end{bmatrix} * \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} (1*a)+(0*b) \\ (0*a)+\left(e^{i\pi/2}*b\right) \end{bmatrix} = \begin{bmatrix} a \\ e^{i\pi/2}b \end{bmatrix} \qquad (2.23)$$

$$S|q\rangle = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/2} \end{bmatrix} * \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} (1*a)+(0*b) \\ (0*a)+\left(e^{i\pi/2}*b\right) \end{bmatrix} = \begin{bmatrix} a \\ e^{i\pi/2}b \end{bmatrix} \qquad (2.24)$$

2.4.5 Πύλη Ελεγχόμενου NOT (cX)

Η πύλη αυτή εφαρμόζεται σε έναν καταχωρητή που αποτελείται από δύο qubits και περιγράφεται από έναν πίνακα 4×4 (Εξ. 2.25). Εάν το πρώτο qubit του καταχωρητή βρίσκεται στην κατάσταση |0⟩, τότε η συνολική κατάσταση παραμένει αμετάβλητη. Αντίθετα, όταν το πρώτο qubit είναι στην κατάσταση |1⟩, το δεύτερο qubit αναστρέφεται, όπως συμβαίνει με την πύλη X. Για τον λόγο αυτό, το πρώτο qubit ονομάζεται qubit ελέγχου (control qubit), καθώς ελέγχει τη μεταβολή της κατάστασης, ενώ το δεύτερο ονομάζεται qubit στόχου (target qubit), επειδή είναι αυτό που μεταβάλλεται.



Εικόνα 2.10: Συμβολισμός πύλης cX

$$cX = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \qquad (2.25)$$

$$Εάν\ |q_1\rangle = \begin{bmatrix} a \\ b \end{bmatrix} \quad \& \quad |q_2\rangle = \begin{bmatrix} c \\ d \end{bmatrix}$$

$$cX|q_1 q_2\rangle = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} * (q_1 \otimes q_2) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} * \left(\begin{bmatrix} a \\ b \end{bmatrix} \otimes \begin{bmatrix} c \\ d \end{bmatrix}\right) =$$

$$= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} * \begin{bmatrix} a*c \\ a*d \\ b*c \\ b*d \end{bmatrix} = \begin{bmatrix} (1*ac)+(0*ad)+(0*bc)+(0*bd) \\ (0*ac)+(1*ad)+(0*bc)+(0*bd) \\ (0*ac)+(0*ad)+(0*bc)+(1*bd) \\ (0*ac)+(0*ad)+(1*bc)+(0*bd) \end{bmatrix} = \begin{bmatrix} ac \\ ad \\ bd \\ bc \end{bmatrix} \qquad (2.26)$$

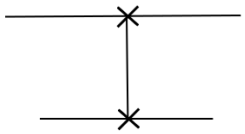
Παρακάτω παρουσιάζεται το αποτέλεσμα της εφαρμογής πύλης cX όταν το qubit ελέγχου βρίσκεται στη κατάσταση $|0\rangle$ ή $|1\rangle$.

$$cX|00\rangle = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} * \left(\begin{bmatrix} 1 \\ 0 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 0 \end{bmatrix}\right) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} * \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = |00\rangle$$

$$cX|10\rangle = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} * \left(\begin{bmatrix} 0 \\ 1 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 0 \end{bmatrix}\right) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} * \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = |11\rangle$$

2.4.6 Πύλη Αντιμετάθεσης – Swap

Η πύλη αντιμετάθεσης εκφράζεται με έναν πίνακα 4×4 και εφαρμόζεται σε έναν καταχωρητή δυο qubits. Η λειτουργία της συνίσταται στην αντιστροφή της σειράς των δύο qubits του καταχωρητή.



Εικόνα 2.11. Συμβολισμός πύλης Swap

$$\text{Swap} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad (2.27)$$

$$Εάν\ |q_1\rangle = \begin{bmatrix} a \\ b \end{bmatrix} \quad \& \quad |q_2\rangle = \begin{bmatrix} c \\ d \end{bmatrix}$$

$$cX|q_1 q_2\rangle = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * (q_1 \otimes q_2) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \left(\begin{bmatrix} a \\ b \end{bmatrix} \otimes \begin{bmatrix} c \\ d \end{bmatrix}\right) =$$

$$= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} a*c \\ a*d \\ b*c \\ b*d \end{bmatrix} = \begin{bmatrix} (1*ac)+(0*ad)+(0*bc)+(0*bd) \\ (0*ac)+(0*ad)+(1*bc)+(0*bd) \\ (0*ac)+(1*ad)+(0*bc)+(0*bd) \\ (0*ac)+(0*ad)+(0*bc)+(1*bd) \end{bmatrix} =$$

$$\begin{bmatrix} ac \\ bc \\ ad \\ bd \end{bmatrix} = \left(\begin{bmatrix} c \\ d \end{bmatrix} \otimes \begin{bmatrix} a \\ b \end{bmatrix}\right) = \ |q_2 q_1\rangle \qquad (2.28)$$

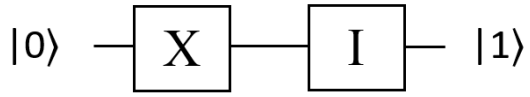
2.5 Κβαντικό Κύκλωμα & Κβαντικός Αλγόριθμος

2.5.1 Κβαντικό Κύκλωμα

Σύμφωνα με το όσα αναφέρθηκαν μέχρι στιγμής, η εκτέλεση υπολογισμών σε έναν κβαντικό υπολογιστή πραγματοποιείται μέσω των κβαντικών καταχωρητών και των κβαντικών πυλών. Σε έναν κλασικό υπολογιστή, ένα κύκλωμα αποτελείται από μια ακολουθία λογικών πυλών που επεξεργάζονται δεδομένα για την υλοποίηση υπολογισμών. Αντίστοιχα, ένα κβαντικό κύκλωμα εφαρμόζει κβαντικές πύλες σε διαδοχικό τρόπο προκειμένου να εκτελέσει κβαντικούς υπολογισμούς [11], [14].

Ένα απλό παράδειγμα ενός κβαντικού κυκλώματος είναι η διαδοχική εφαρμογή των πυλών X και I στην κατάσταση $|0\rangle$ (Εικόνα 2.12). Για την εκτέλεση του κυκλώματος αυτού,

πρώτα εφαρμόζεται η πύλη X στο |0⟩ και το αποτέλεσμα της πράξης μεταφέρεται στην επόμενη πύλη που είναι η I.



Εικόνα 2.12: Παράδειγμα κβαντικού κυκλώματος

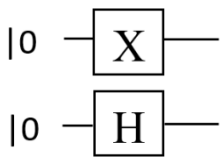
$$X \to |0\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} * \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

$$I \to \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} * \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} = |1\rangle$$

Όταν ένα κύκλωμα αποτελείται από n qubits, το συνολικό αποτέλεσμα υπολογίζεται μέσω του τανυστικού γινομένου των καταστάσεων των qubits (διάνυσμα διάστασης nx1) και του τανυστικού γινομένου των αντίστοιχων πυλών (πίνακας διάστασης nxn). Το τελικό αποτέλεσμα προκύπτει από τον πολλαπλασιασμό του πίνακα των πυλών με το διάνυσμα κατάστασης, με τις πύλες να προηγούνται. Αν μία πύλη δεν εφαρμόζεται σε όλα τα qubits, τότε εφαρμόζεται η πύλη αδρανείας (I) στα υπόλοιπα. Το τελικό αποτέλεσμα είναι ένα διάνυσμα διαστάσεων nx1, όπου n είναι ο αριθμός των qubits [3]. Εάν για παράδειγμα, υλοποιήσουμε 1 κύκλωμα με 2 qubits που στο πρώτο εφαρμοστεί η πύλη X και στο δεύτερο η H τότε:

$$\mathrm{XH}|00\rangle = (\mathrm{X} \otimes \mathrm{H}) * (|0\rangle \otimes |0\rangle) = \left(\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \otimes \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}\right) * \left(\begin{bmatrix} 1 \\ 0 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 0 \end{bmatrix}\right)$$

$$= \frac{1}{\sqrt{2}} \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & -1 \\ 1 & 0 & -1 & 0 \end{bmatrix} * \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \end{bmatrix} = \frac{1}{\sqrt{2}}|01\rangle + \frac{1}{\sqrt{2}}|11\rangle$$



Εικόνα 2.13: Παράδειγμα κβαντικού κυκλώματος με 2 qubits και 2 πύλες

2.5.2 Κβαντικός Αλγόριθμος

Ένας αλγόριθμος χαρακτηρίζεται ως κβαντικός όταν μπορεί να υλοποιηθεί σε έναν κβαντικό υπολογιστή κάνοντας χρήση της υπέρθεσης ή της διαπλοκής. Ο κβαντικός αλγόριθμος αναπαρίστανται μέσω κβαντικού κυκλώματος αποτελούμενο από κβαντικές πύλες που εφαρμόζονται σε ένα ή περισσότερα qubits για να αλλάξουν την κατάστασή τους [10].

Οι κβαντικοί αλγόριθμοι βρίσκουν εφαρμογή σε τομείς όπως η κρυπτογραφία, η προσομοίωση κβαντικών συστημάτων, η μηχανική μάθηση και άλλα. Εκμεταλλεύονται τις αρχές της υπέρθεσης και της διεμπλοκής για να επιτύχουν εκθετικά πλεονεκτήματα έναντι των κλασικών μεθόδων. Η υλοποίησή τους βασίζεται σε κβαντικά κυκλώματα με πύλες που τροποποιούν κβαντικές καταστάσεις [3], [14].

2.6 Κβαντικός Προσομοιωτής

Οι κβαντικοί υπολογιστές που υπάρχουν μέχρι σήμερα βρίσκονται ακόμα σε πρώιμο στάδιο και έχουν περιορισμένες δυνατότητες λόγω του μικρού αριθμού qubits που διαθέτουν, την ανοχή τους σε υψηλές θερμοκρασίες όπως επίσης και του αυξημένου κόστους τους [8]. Η δημιουργία και εκτέλεση των κβαντικών προγραμμάτων γίνεται μέσω προσομοιωτών που τρέχουν σε δυαδικούς υπολογιστές. Όπως γίνεται κατανοητό, το βασικό στοιχείο των εξομοιωτών, είναι οι πράξεις πινάκων. Με τη χρήση μεγάλου αριθμού qubits, οι πίνακες αυτοί είναι μεγάλοι σε μέγεθος, που έχει ως αποτέλεσμα τη μεγάλη απαίτηση σε υπολογιστικούς πόρους και χρόνου.

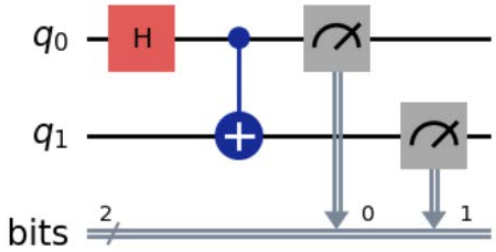
Η χρήση εξομοιωτών είναι αναγκαία προκειμένου να μελετηθούν τα κβαντικά συστήματα και να αναπτυχθεί το κατάλληλο λογισμικό. Η προσομοίωση κβαντικών κυκλωμάτων και πυλών που μπορεί να υλοποιηθεί εύκολα με τη χρήση της τρέχουσας τεχνολογίας, καθώς περιλαμβάνει την εκτέλεση μαθηματικών πράξεων. Υπάρχουν διαθέσιμα εργαλεία εξομοίωσης κυκλωμάτων κβαντικών υπολογιστών, τα οποία παρέχουν ένα προγραμματιστικό περιβάλλον για τη χρήση τους, όπως το Qiskit [4], [15].

Το Qiskit είναι μια βιβλιοθήκη της python που αναπτύχθηκε από την IBM και χρησιμοποιείται για την προσομοίωση κβαντικών κυκλωμάτων και αλγορίθμων με απλές εντολές. Περιλαμβάνει συναρτήσεις για την εφαρμογή πυλών σε qubits, την απεικόνιση κβαντικών καταστάσεων και άλλες χρήσιμες λειτουργίες [16], [17]. Ακολουθεί ένα απλό κύκλωμα απλά κυκλώματα με χρήση της βιβλιοθήκης Qiskit της python.

Στην εικόνα 2.14 παρατίθεται η υλοποίηση, σε Python, ενός κβαντικού κυκλώματος. Στο δεύτερο qubit δεν εφαρμόζεται κάποια πύλη, ενώ στο πρώτο εφαρμόζεται αρχικά η πύλη Hadamard (H) και στη συνέχεια η πύλη CNOT (CX), με το πρώτο qubit ως qubit ελέγχου και το δεύτερο ως στόχο. Για την κατασκευή του κυκλώματος απαιτείται η εισαγωγή των κατάλληλων βιβλιοθηκών της Qiskit. Μέσω αυτών δημιουργούνται οι κβαντικοί και κλασικοί καταχωρητές, το κβαντικό κύκλωμα, καθώς και η εφαρμογή των απαραίτητων πινάκων πυλών. Όλες οι μαθηματικές πράξεις εκτελούνται μέσω των ενσωματωμένων συναρτήσεων της Qiskit που χρησιμοποιούνται στο πρόγραμμα.

```
from qiskit import *

n = 2
qr = QuantumRegister(n, 'q')
cr = ClassicalRegister(n, 'bits')
qc = QuantumCircuit(qr, cr)

qc.h(0)
qc.cx(0,1)

qc.measure(qr, cr)
qc.draw('mpl')
```

Εικόνα 2.14: Bell state σε python qiskit



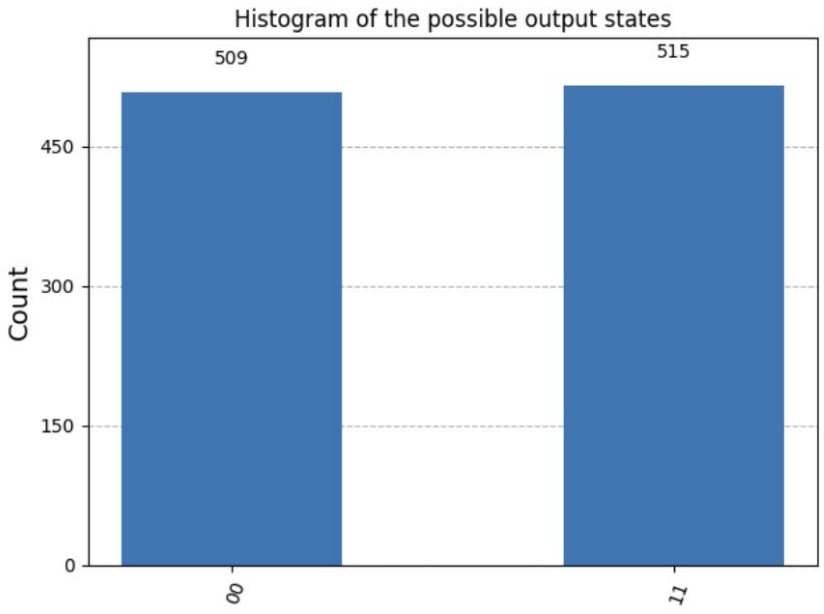
Εικόνα 2.15. Κβαντικό κύκλωμα που δημιουργήθηκε από την συνάρτηση draw

Στις εικόνες 2.16 και 2.17 εμφανίζονται η εκτέλεση και το ιστόγραμμα των πιθανών καταστάσεων εξόδου του κυκλώματος, όπως προέκυψαν από την εκτέλεση του προσομοιωτή AerSimulator.

```
from qiskit import *
from qiskit import QuantumCircuit, transpile
from qiskit_aer import AerSimulator
from qiskit.visualization import plot_histogram, plot_state_city
import qiskit.quantum_info as qi
```

```
n = 2
qr = QuantumRegister(n, 'q')
cr = ClassicalRegister(n, 'bits')
qc = QuantumCircuit(qr, cr)

qc.h(0)
qc.cx(0,1)

qc.measure(qr, cr)
qc.draw('mpl')

# Transpile for simulator
simulator = AerSimulator()
qc = transpile(qc, simulator)

# Run and get counts
result = simulator.run(qc).result()
counts = result.get_counts(qc)
plot_histogram(counts, title='Histogram of the possible output states')
```

Εικόνα 2.16: Υλοποίηση προσομοίωσης κβαντικού κυκλώματος με AerSimulator



Εικόνα 2.17:Ιστόγραμμα των πιθανών καταστάσεων εξόδου.

3 CUDA

Οι μονάδες επεξεργασίας γραφικών (Graphics Processing Units – GPU) σχεδιάστηκαν αρχικά για την προβολή γραφικών σε εφαρμογές πολυμέσων και βιντεοπαιχνιδιών και τρισδιάστατων απεικονίσεων [18]. Με την πάροδο των ετών, η αρχιτεκτονική τους εξελίχθηκε σημαντικά, καθιστώντας τις GPU ικανές να υποστηρίξουν αποδοτικά υπολογισμούς γενικού σκοπού (GPGPU – General Purpose Computing on Graphics Processing Units) [18], [19].

Η δυνατότητα αποδοτικής εκτέλεσης υπολογιστικών πράξεων οφείλεται στην ιδιαίτερα παραλληλισμένη αρχιτεκτονική των GPU, η οποία ενσωματώνει χιλιάδες επεξεργαστικούς πυρήνες μικρού μεγέθους. Οι πυρήνες αυτοί έχουν σχεδιαστεί για να εκτελούν επαναλαμβανόμενες συγκεκριμένες αριθμητικές πράξεις παράλληλα [18]. Ενδεικτικά, οι GPU χρησιμοποιούνται ευρέως σε πεδία όπως η προσομοίωση φυσικών φαινομένων, η μηχανική μάθηση, η χημεία, η ανακατασκευή ιατρικών εικόνων, καθώς και – όπως εστιάζει η παρούσα μελέτη – οι προσομοιώσεις κβαντικών υπολογιστικών συστημάτων [20].

Η χρήση των GPU για υπολογισμούς γενικού σκοπού έγινε εφικτή μέσω της ανάπτυξης εξειδικευμένων προγραμματιστικών μοντέλων από τις κατασκευάστριες εταιρείες. Η NVIDIA εισήγαγε το CUDA (Compute Unified Device Architecture), ένα ώριμο και διαρκώς εξελισσόμενο περιβάλλον ανάπτυξης, το οποίο επιτρέπει την αξιοποίηση των GPU πέρα από τις εφαρμογές γραφικών [19], [20]. Παράλληλα, η AMD αναπτύσσει το ROCm (Radeon Open Compute), μια ανοικτού κώδικα πλατφόρμα για την υποστήριξη των δικών της GPU [21]. Στην παρούσα εργασία, θα επικεντρωθούμε στο CUDA λόγω της πιο ευρείας διάδοσης και χρήσης τους.

3.1 Η Αρχιτεκτονική GPGPU της NVIDIA

Η GPU αποτελείται από πολλαπλούς Streaming Multiprocessors (SMs), εκ των οποίων ο καθένας περιλαμβάνει πολυάριθμους πυρήνες CUDA . Κάθε CUDA πυρήνας έχει τη δυνατότητα να εκτελεί βασικές αριθμητικές πράξεις ταυτόχρονα με άλλους πυρήνες του

ίδιου ή άλλου SM, προσφέροντας μαζική παραλληλία [6]. Οι SMs διαθέτουν επίσης κοινόχρηστους πόρους, όπως shared memory [20].

Η CUDA αρχιτεκτονική οργανώνεται σε ένα πολύ-επίπεδο μοντέλο [20], [22]:

- *Grids*: Συλλογές blocks που εκτελούνται σε όλο το GPU Grid.
- *Blocks*: Ομάδες από threads που εκτελούνται σε SMs. Μοιράζονται shared memory και μπορούν να συγχρονίζονται.
- *Threads*: Το βασικό στοιχείο που εκτελεί ανεξάρτητες εντολές και οργανώνονται σε warps (κομμάτι 32 threads) και ελέγχονται από warp schedulers.

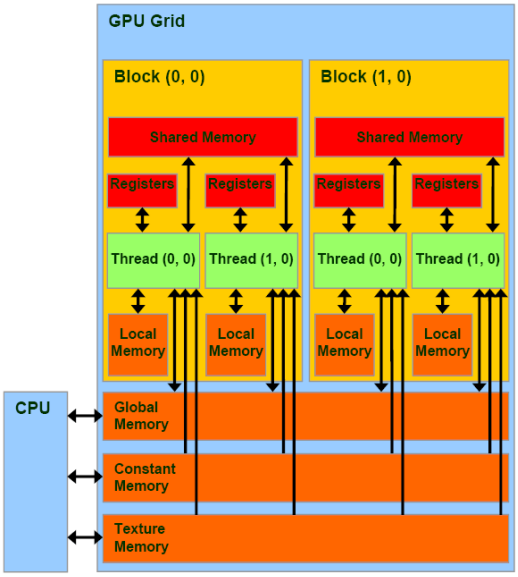
Η εκτέλεση βασίζεται στο μοντέλο SIMT (Single Instruction, Multiple Threads), σύμφωνα με το οποίο όλα τα threads ενός warp εκτελούν την ίδια εντολή ταυτόχρονα. Αυτό είναι ένα βασικό χαρακτηριστικό της αρχιτεκτονικής, καθώς διαφορετικούς είδους εντολές, δημιουργούν αποκλίσεις μέσα στο warp και μειώνουν την απόδοση. Για το λόγο αυτό είναι επιθυμητός ο ομοιόμορφος έλεγχος ροής στους υπολογισμούς, χωρίς να περιέχει ελέγχους συνθηκών [20], [23]. Κάθε thread εκτελεί εντολές μεμονωμένα, αλλά μπορεί να συνεργάζεται με άλλα threads εντός του ιδίου block μέσω κοινής μνήμης (shared memory).

Τα διαφορετικά στοιχεία μνήμης που περιέχονται είναι τα εξής:

- *Registers*: εξαιρετικά ταχεία, ιδιωτική μνήμη για κάθε thread.
- *Local memory*: προσωρινή μνήμη ανά thread (υλοποιείται στη global memory).
- *Shared memory*: κοινόχρηστη μνήμη ανά block.
- *Global memory*: κοινή για όλα τα threads και blocks.
- *Constant & Texture memory*: σταθερά ή ειδικά προσπελάσιμα δεδομένα.

Η υψηλή απόδοση μνήμης των GPU υποστηρίζεται από memory bandwidth που σε σύγχρονα μοντέλα όπως το NVIDIA H100 ξεπερνά το 1 TB/s [24].

Η παρακάτω εικόνα απεικονίζει το μοντέλο αυτό και τη ροή δεδομένων από τη CPU προς το GPU Grid:



Εικόνα 3.1: Αρχιτεκτονική GPGPU [25]

Το διάγραμμα αναπαριστά το μοντέλο μνήμης και υπολογιστικής ροής της CUDA. Φαίνεται η ροή δεδομένων από τη CPU προς την global memory της GPU και η κατανομή σε blocks, threads και SMs, με ανάδειξη της shared memory. Κάθε block (π.χ. Block (0, 0)) περιλαμβάνει threads (π.χ. Thread (0, 0), Thread (1, 0)) τα οποία έχουν πρόσβαση σε registers, local memory και κοινόχρηστη shared memory του block. Η global memory είναι προσβάσιμη από όλα τα blocks, ενώ η constant και texture memory παρέχουν εξειδικευμένες μορφές πρόσβασης. Οι μαύρες κατευθυνόμενες γραμμές αποτυπώνουν τις κατευθύνσεις μεταφοράς δεδομένων μεταξύ των επιπέδων μνήμης και επεξεργασίας.

3.2 Το Προγραμματιστικό Μοντέλο CUDA

Η *CUDA (Compute Unified Device Architecture)* αποτελεί ένα προγραμματιστικό μοντέλο που συνδυάζει το παραλληλισμό επιπέδου δεδομένων με άμεση πρόσβαση στην υπολογιστική ισχύ της GPU.

Παρακάτω, παρατίθεται ένα παράδειγμα με πολλαπλασιασμό δυο πινάκων NxN το οποίο είναι μια από τις πιο χαρακτηριστικές εφαρμογές για τη σύγκριση CPU και GPU.

```
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        C[i][j] = 0;
        for (int k = 0; k < N; k++) {
            C[i][j] += A[i][k] * B[k][j];
        }
```

```
    }
}
```

Εικόνα 3.2 Πολλαπλασιασμό δυο πινάκων NxN σε C (CPU)

Η CPU εκτελεί σειριακά τις επαναλήψεις. Οι σύγχρονες CPU μπορούν να εκτελούν πολλαπλές εντολές ταυτόχρονα χρησιμοποιώντας SIMD (Single Instruction, Multiple Data), μία τεχνική που επιτρέπει την εφαρμογή της ίδιας εντολής σε πολλαπλά δεδομένα ταυτόχρονα [26]. Η SIMD υλοποιείται μέσω επεκτάσεων εντολών όπως SSE (Streaming SIMD Extensions), AVX (Advanced Vector Extensions), και AVX-512 [27]. Οι AVX-512 εντολές επιτρέπουν την ταυτόχρονη εκτέλεση έως και 16 πράξεων κινητής υποδιαστολής (float) ή 8 πράξεων double ανά κύκλο ανά πυρήνα, βελτιώνοντας σημαντικά την απόδοση σε υπολογιστικά εντατικά φορτία, αν και σε μικρότερη κλίμακα σε σχέση με τις GPU.

Αντίθετα, η GPU αναθέτει κάθε υπολογισμό C[i][j] σε ένα thread. Χιλιάδες threads μπορούν να εκτελούνται ταυτόχρονα, οργανωμένα σε blocks και grids.

```
__global__ void matMul(float *A, float *B, float *C, int N) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0;
    for (int k = 0; k < N; ++k) {
        sum += A[row * N + k] * B[k * N + col];
    }
    C[row * N + col] = sum;
}
```

Εικόνα 3.3. Πολλαπλασιασμό δυο πινάκων NxN σε C++ (GPU)

Κάθε thread υπολογίζει μία θέση στον πίνακα C. Το kernel αυτό κατανέμεται σε blocks π.χ. 32×32 threads ανά block, και εκτελείται από το σύνολο των multiprocessors. Αν N=1024, απαιτούνται 32×32 blocks = 1024 threads σε κάθε διάσταση.

Οι επιδόσεις αυτής της μεθόδου είναι πολλαπλάσιες από την CPU για N > 512, λόγω της παράλληλης εκτέλεσης και του υψηλότερου memory bandwidth της GPU. Ειδικά αν γίνει χρήση shared memory και tile-based multiplication, οι χρόνοι εκτέλεσης μπορούν να βελτιωθούν ακόμα περισσότερο.

Η CPU εκτελεί εντολές με καθυστέρηση 3-6 κύκλων ανά εντολή. Χρησιμοποιεί out-of-order execution, speculative branching και multilevel cache (L1, L2, L3) για την επιτάχυνση της επεξεργασίας [28]. Παρά την υποστήριξη SIMD, ο αριθμός ταυτόχρονων πράξεων ανά κύκλο είναι περιορισμένος. Οι CPU είναι εξαιρετικά αποτελεσματικές για σειριακές και ελεγχόμενες ροές εκτέλεσης.

Από την άλλη, η GPU είναι σχεδιασμένη για throughput-oriented επεξεργασία. Η CUDA εκτελεί warps (32 threads) μέσω πολλαπλών ALUs σε κάθε SM, επιτυγχάνοντας χιλιάδες ταυτόχρονες πράξεις [20]. Η shared memory μειώνει τη συμφόρηση στην global memory, ενώ η υψηλή απόδοση επιτυγχάνεται με το overlap μεταξύ υπολογισμών και μεταφοράς δεδομένων.

Ο συνδυασμός threads/block, χρήση shared memory και δυνατότητα overlap μεταξύ υπολογισμών και μεταφοράς δεδομένων επιτρέπει στη GPU να επιτυγχάνει αποδόσεις που υπερβαίνουν κατά τάξεις μεγέθους εκείνες της CPU σε παράλληλες διεργασίες [29].

Εικόνα 3.4 Διάγραμμα Αρχιτεκτονικής CPU: Εικόνα: Σύγκριση CPU-GPU [30]

Η διαφορά στον σχεδιασμό καθιστά την GPU ιδανική για παραλληλία επιπέδου δεδομένων και αριθμητικές πράξεις μεγάλης κλίμακας, όπως σε κβαντικές προσομοιώσεις, big data analytics και deep learning, ενώ η CPU παραμένει κρίσιμη σε σειριακές διεργασίες, λογική και branching εντολές [6], [29].

3.3 Εφαρμογές των GPGPU

Οι GPU γενικού σκοπού έχουν αποκτήσει κομβική σημασία σε ένα ευρύ φάσμα επιστημονικών και τεχνολογικών πεδίων. Παρακάτω παρουσιάζονται χαρακτηριστικές κατηγορίες εφαρμογών με περισσότερες τεχνικές λεπτομέρειες [31]:

- *Μηχανική Μάθηση και Βαθιά Μάθηση*: Οι GPU επιτρέπουν την ταχεία εκπαίδευση βαθιών νευρωνικών δικτύων, επιτυγχάνοντας παράλληλη επεξεργασία εκατομμυρίων δεδομένων εισόδου. Ιδιαίτερα στα Convolutional Neural Networks (CNNs), η εκπαίδευση πραγματοποιείται πολλαπλάσια ταχύτερα σε GPU μέσω batch-based computation. Επιπλέον, στα μοντέλα τύπου Transformer, που χρησιμοποιούνται στο NLP, οι απαιτήσεις μνήμης και υπολογιστικής ισχύος καθιστούν τις GPU αναγκαίες για αποδοτική λειτουργία.

- *Επιστημονικές Προσομοιώσεις:* Τα εργαλεία GROMACS, AMBER και NAMD εκμεταλλεύονται την αρχιτεκτονική CUDA για προσομοιώσεις μοριακής δυναμικής, επιτυγχάνοντας σημαντική επιτάχυνση σε αναπαραστάσεις βιολογικών μορίων και συστημάτων. Οι GPU επιτρέπουν την ανάλυση χιλιάδων ατόμων ταυτόχρονα σε σύντομα χρονικά βήματα (femtoseconds), προσφέροντας ακρίβεια και ταχύτητα.
- *Υπολογιστική Βιολογία και Χημεία*: Εφαρμογές όπως το GPU-BLAST (βιολογική αλληλουχία) και CUDA-SW++ (ευθυγράμμιση DNA) μειώνουν δραματικά τον χρόνο ανάλυσης γενετικών δεδομένων. Οι GPU αξιοποιούνται επίσης σε προσομοιώσεις αλληλεπιδράσεων φαρμάκων, συμβάλλοντας στη γρήγορη ανακάλυψη νέων μορίων.
- *Χρηματοοικονομική Ανάλυση και High-Frequency Trading (HFT):* Οι GPU ενσωματώνονται σε πλατφόρμες συναλλαγών για την υλοποίηση σύνθετων στοχαστικών αλγορίθμων σε πραγματικό χρόνο. Παρέχουν latency σε επίπεδο μικροδευτερολέπτου για τη λήψη αποφάσεων βασισμένων σε χιλιάδες παραμέτρους αγορών. Οι υποδομές HFT συχνά περιλαμβάνουν clusters με πολλαπλές GPU για μέγιστη απόδοση.
- *Επεξεργασία Σήματος και Εικόνας*: Στην επεξεργασία εικόνας και video, οι GPU χρησιμοποιούνται για real-time αποσυμπίεση, φιλτράρισμα, ανακατασκευή εικόνας (e.g. medical imaging) και rendering. Το παράλληλο μοντέλο τους ευνοεί την ανάλυση pixel-by-pixel και frame-by-frame σε υψηλές αναλύσεις.
- *Τεχνητή Νοημοσύνη στην Αυτοκινητοβιομηχανία*: Τα αυτόνομα οχήματα βασίζονται σε GPU για τη γρήγορη ανάλυση εισόδων από αισθητήρες, την αναγνώριση αντικειμένων, και την κατανόηση της σκηνής (scene understanding). Αλγόριθμοι deep learning τρέχουν σε πραγματικό χρόνο, επιτρέποντας ασφαλείς αποφάσεις πλοήγησης.
- *Εικονική και Επαυξημένη Πραγματικότητα (VR/AR):* Οι εφαρμογές AR/VR απαιτούν υψηλής ακρίβειας 3D rendering, real-time tracking και προβολή, τα οποία οι GPU επιτυγχάνουν μέσω ray tracing και shader pipelines. Η χρήση GPU εξασφαλίζει ομαλή αλληλεπίδραση χωρίς καθυστερήσεις (latency < 20ms), που είναι κρίσιμη για την ανθρώπινη εμπειρία.

3.4 Σύγκριση CPU και GPU

Η NumPy είναι μια βιβλιοθήκη Python που χρησιμοποιείται για αριθμητικούς υπολογισμούς χρησιμοποιώντας τη CPU [32]. Η CuPy είναι μια παρόμοια βιβλιοθήκη της Python, η οποία όμως χρησιμοποιεί τη GPU[33]. Εφόσον και οι δύο βιβλιοθήκες διαθέτουν σχεδόν ίδιες συναρτήσεις, είναι εύκολο να συγκριθεί ο χρόνος εκτέλεσης των αντίστοιχων λειτουργιών τους.

```
import numpy as np
import time

N = 10000
A = np.random.rand(N, N)
B = np.random.rand(N, N)

start = time.time()
C = np.dot(A, B)
end = time.time()
print(f"NumPy CPU: {end - start:.4f} sec")
```

Εικόνα 3.5. Υπολογισμός εσωτερικού γινομένου με χρήση της βιβλιοθήκης NumPy

```
import cupy as cp
import time

N = 10000
A = cp.random.rand(N, N)
B = cp.random.rand(N, N)

cp.cuda.Device(0).synchronize()
start = time.time()
C = cp.dot(A, B)
cp.cuda.Device(0).synchronize()
end = time.time()
print(f"CuPy GPU: {end - start:.4f} sec")
```

Εικόνα 3.6. Υπολογισμός εσωτερικού γινομένου με χρήση της βιβλιοθήκης CuPy

Στον πίνακα 3.1 αναγράφονται τα αποτελέσματα των των παραπάνω υπολογισμών σε (CPU: i5-13500, GPU: RTX 4060 Ti 16GB). Η GPU εμφανίζει σαφές πλεονέκτημα όταν το πρόβλημα έχει αρκετά μεγάλο όγκο δεδομένων, ώστε να ξεπεραστεί το overhead της μεταφοράς από/προς τη μνήμη της GPU. Αντίθετα, για μικρούς πίνακες, η CPU (με SIMD) μπορεί να είναι ανταγωνιστική, λόγω μικρότερης καθυστέρησης μνήμης και ταχύτερης πρόσβασης στα cache lines.

Μέγεθος N	CPU Χρόνος (sec)	GPU Χρόνος (sec)
5000	1.5818	0.8973
10000	13.13	6.29
15000	47.28	20.88

Πίνακας 3.1 Αποτελέσματα εκτέλεσης εσωτερικού γινομένου σε NumPy και Cupy

4 Ανάπτυξη Κβαντικών συναρτήσεων σε GPU

Για τις απαιτήσεις της παρούσας διπλωματικής εργασίας, υλοποιήθηκαν συναρτήσεις για τις κβαντικές πύλες που αναφέρθηκαν στο κεφάλαιο 2 με τη γλώσσα προγραμματισμού Python και με χρήση τη βιβλιοθήκης CuPy. Πιο συγκεκριμένα, οι πράξεις τανυστικού και εσωτερικού γινομένου έγιναν μέσω των συναρτήσεων kron() και dot() αντίστοιχα της βιβλιοθήκης CuPy. Το περιβάλλον που χρησιμοποιήθηκε ήταν το Jupyter Notebook. Στο κεφάλαιο αυτό παρατεθείτε αναλυτικά ο κώδικας και η λογική υλοποίησης κάθε πύλης. Όλες οι συναρτήσει δημιουργήθηκαν με τη προϋπόθεση τα qubits να παίρνουν αποκλειστικά τις τιμές |0> ή |1>.

4.1 Δημιουργία αρχικής κατάστασης

Η συνάρτηση initial_state δημιουργεί την αρχική κβαντική κατάσταση ενός κβαντικού κυκλώματος. Τα ορίσματά της είναι:

- ql: λίστα ακεραίων διάστασης nx1, όπου κάθε στοιχείο αντιστοιχεί στη τιμή του αντίστοιχου qubit του κβαντικού κυκλώματος

  $\pi\chi$. Αν $ql = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$ $\tau\acute{o}\tau\varepsilon$ $q_0 = 0$ $\kappa\alpha\iota$ $q_1 = 0$

- n: ο αριθμός των qubit του κβαντικού κυκλώματος

Αρχικά, ορίζουμε τις 2 βασικές καταστάσεις |0⟩ και |1⟩. Για να δημιουργηθεί σωστά η αρχική κατάσταση πρέπει να εφαρμοστεί το τανυστικό γινόμενο $q_0 \otimes q_1 \otimes \ldots \otimes q_n$. Πρώτα, ελέγχουμε το πρώτο στοιχείο της λίστα ql για να καταλάβουμε με ποια κατάσταση θα ξεκινήσει το τανυστικό γινόμενο. Αν q0 είναι 0 το state γίνεται $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$,αν είναι 1 γίνεται $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$.

Στη συνέχεια, με χρήση της for, ξεκινάμε από το 2ο στοιχείο της λίστας και ελέγχουμε αν το qubit q1 είναι 0 ή 1. Αν είναι μηδέν υπολογίζουμε την πράξη initial_state ⊗|0⟩, αλλιώς initial_state ⊗|1⟩. Αυτό γίνεται διαδοχικά μέχρι να ελέγχουμε όλη τη gl λίστα.

Αν το κβαντικό κύκλωμα αποτελείτε από ένα μονο qubit η for δε θα εκτελεστεί και η συνάρτηση θα επιστρέψει το zero_state ή το one_state.

```
import cupy as cp
def initial_state(qubit_list, num_qubits):

    zero_state = cp.array([1, 0], dtype=cp.float32)
    one_state = cp.array([0, 1], dtype=cp.float32)

    if (qubit_list[0]==0):
        initial_state = zero_state
    else:
        initial_state = one_state

    for i in range(num_qubits-1):
        if (qubit_list[i+1]==0):
            initial_state = cp.kron(zero_state, initial_state)
        else:
            initial_state = cp.kron(one_state, initial_state)

    return initial_state
```

Εικόνα 4.1. Υλοποίηση initial_state με χρήση της βιβλιοθήκης CuPy

Ενδεικτικά παραδείγματα της εκτέλεσης της συνάρτησης αυτής είναι:

qubit_list	**num_qubits**	**return**
[0. 0.]	2	[1. 0. 0. 0]
[1. 1. 1.]	3	[0. 0. 0. 0. 0. 0. 0. 1]
[0. 1. 0. 0]	4	[0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0]

Πίνακας 4.1. Παραδείγματα εκτέλεσης της συνάρτησης initial_state

4.2 Εφαρμογή πυλών σε 1 qubit

Όλες οι συναρτήσεις αυτής της ενότητας υλοποιούνται με την ίδια λογική και εφαρμόζονται σε ένα qubit. Τα ορίσματά για κάθε συνάρτηση είναι:

- state: λίστα ακεραίων διάστασης nx1, όπου qi με i ∈ [0, n-1] αντιστοιχεί στη τρέχουσα κβαντική κατάσταση
- q: ο αριθμός του qubit που θα εφαρμοστεί η πύλη
- n: ο αριθμός των qubit του κβαντικού κυκλώματος

Αρχικά, ορίζουμε την πύλη I, H, T, Y, Z, S, T στην αντίστοιχη συνάρτηση, σύμφωνα με τη θεωρία της κβαντικής που αναφέρθηκε στο Κεφάλαιο 2. Σε ένα κβαντικό κύκλωμα ενός qubit η εφαρμογή της πύλης υλοποιείται με το εσωτερικό γινόμενο της πύλης με του διανύσματος κατάστασης. Όταν όμως το κύκλωμα αποτελείται από δυο ή περισσότερα qubits τότε ο υπολογισμός της πύλης στο επιθυμητό qubit είναι:

$$I \otimes \dots \otimes \dots \otimes GATE \otimes I \otimes \dots |q_0 \otimes \dots \otimes q_{gate} \otimes \dots \otimes q_{n-1}\rangle$$

Συνεπώς, πρέπει να δημιουργηθεί μια πύλη(gate) που είναι το τανυστικό γινόμενο της πύλης I και της τρέχουσας κατάστασης (state) στην οποία βρίσκεται το κβαντικό κύκλωμα. Εάν το κύκλωμα αποτελείται από ένα qubit τότε εκτελείτε εσωτερικό γινόμενο. Σε αντίθετη περίπτωση, εκτελείται μια for που διαπερνά διαδοχικά το state και εφαρμόζει τανυστικό γινόμενο μεταξύ των πυλών. Με τη ολοκλήρωση της for, εκτελείτε εσωτερικό γινόμενο με το state και η συνάρτηση επιστρέφει τη νέα τρέχουσα κατάσταση.

Στη συνέχεια της ενότητας αυτής, αναφέρονται οι υλοποιήσεις κάθε συνάρτησης και το αποτέλεσμα της.

4.2.1 Πύλη I

Η συνάρτηση id(state, q, n) εφαρμόζει στο qubit q την πύλη I στην τρέχουσα κβαντική κατάσταση state.

```
import cupy as cp

def id(state, q, n):
    #define I gate
    i_gate = cp.array([[1, 0], [0, 1]], dtype=cp.float32)

    if n==1:
        result = cp.dot(i_gate, state)
        return result

    #Create the gate I⊗...⊗I
    gate=i_gate
    for i in range(n-1):
        gate = cp.kron(gate, i_gate)

    #Apply the gate to the state vector
    state = cp.dot(gate, state)

    return state
```

Εικόνα 4.2. Υλοποίηση της πύλης I με χρήση της βιβλιοθήκης CuPy

Ενδεικτικά παραδείγματα της εκτέλεσης της συνάρτησης αυτής είναι:

state	q	n	return
[1. 0. 0. 0]	2	2	[1. 0. 0. 0]
[1. 0. 1]	1	3	[1. 0. 1]
[1. 1. 0. 0]	4	4	[1. 1. 0. 0]

Πίνακας 4.2. Παραδείγματα εκτέλεσης της συνάρτησης id

4.2.2 Πύλη H

Η συνάρτηση h(state, q, n) εφαρμόζει στο qubit q την πύλη Η στην τρέχουσα κβαντική κατάσταση state.

```
import cupy as cp

def h(state, q, n):
    #Define gates
    i_gate = cp.array([[1, 0], [0, 1]], dtype=cp.float32)
    h_gate = (1/cp.sqrt(2)) * cp.array([[1, 1], [1, -1]], dtype=cp.float32)

    if n==1:
        result = cp.dot(h_gate, state)
        return result

    #Create the gate I⊗.⊗H⊗..⊗I
    #If H is at the first qubit, start with H gate
    if (0==q):
        gate= h_gate
    else:
        gate = i_gate

    for i in range(n-1):
        if q == i+1:
            gate = cp.kron(gate, h_gate)
        else:
            gate = cp.kron(gate, i_gate)

    # Apply the gate to the state vector
    state = cp.dot(gate, state)

    return state
```

Εικόνα 4.3. Υλοποίηση της πύλης Η με χρήση της βιβλιοθήκης CuPy

Ενδεικτικά παραδείγματα της εκτέλεσης της συνάρτησης αυτής είναι:

state	q	n	return
[1. 0. 0. 0.]	0	2	[0.70710678 0. 0.70710678 0.]
[0. 1. 0. 0.]	1	2	[0.70710678 -0.70710678 0. 0.]
[1. 0. 0. 0. 0. 0. 0. 0.]	2	3	[0.70710678 0.70710678 0. 0. 0. 0. 0. 0.]

Πίνακας 4.3: Παραδείγματα εκτέλεσης της συνάρτησης h

4.2.3 Πύλη X

Η συνάρτηση x(state, q, n) εφαρμόζει στο qubit q την πύλη X στην τρέχουσα κβαντική κατάσταση state.

```
import cupy as cp

def x(state, q, n):
    #Define gates
    i_gate = cp.array([[1, 0], [0, 1]], dtype=cp.float32)
    x_gate = cp.array([[0, 1], [1, 0]], dtype=cp.float32)

    if n==1:
        result = cp.dot(x_gate, state)
        return result

    #Create the gate I⊗.⊗X⊗..⊗I
    #If X is at the first qubit, start with X gate
    if (0==q):
        gate= x_gate
    else:
        gate = i_gate

    for i in range(n-1):
        if q == i+1:
            gate = cp.kron(gate, x_gate)
        else:
            gate = cp.kron(gate, i_gate)

    #Apply the gate to the state vector
```

```
    state = cp.dot(gate, state)

    return state
```

Εικόνα 4.4. Υλοποίηση της πύλης X με χρήση της βιβλιοθήκης CuPy

Ενδεικτικά παραδείγματα της εκτέλεσης της συνάρτησης αυτής είναι:

state	q	n	return
[1. 0. 0. 0.]	0	2	[0. 0. 1. 0.]
[1. 0. 0. 0.]	1	2	[0. 1. 0. 0.]
[0. 0. 0. 1.]	0	2	[0. 1. 0. 0.]

Πίνακας 4.4: Παραδείγματα εκτέλεσης της συνάρτησης x

4.2.4 Πύλη Y

Η συνάρτηση y(state, q, n) εφαρμόζει στο qubit q την πύλη Y στην τρέχουσα κβαντική κατάσταση state.

```
import cupy as cp

def y(state, q,n):
    #Define gates
    i_gate = cp.array([[1, 0], [0, 1]], dtype=cp.complex64)
    y_gate=cp.array([[0, -1j], [1j, 0]], dtype=cp.complex64)

    if n==1:
        result = cp.dot(y_gate, state)
        return result

    #Create the gate I⊗.⊗Y⊗..⊗I
    #If Y is at the first qubit, start with Y gate
    if (0==q):
        gate= y_gate
    else:
        gate = i_gate

    for i in range(n-1):
        if q == i+1:
            gate = cp.kron(gate, y_gate)
        else:
            gate = cp.kron(gate, i_gate)

    #Apply the gate to the state vector
    state = cp.dot(gate, state)

    return state
```

Εικόνα 4.5. Υλοποίηση της πύλης Y με χρήση της βιβλιοθήκης CuPy

Ενδεικτικά παραδείγματα της εκτέλεσης της συνάρτησης αυτής είναι:

state	q	n	return
[0. 1. 0. 0.]	0	2	[0.+0.j 0.+0.j 0.+0.j 0.+1.j]
[0. 0. 0. 1.]	0	2	[0.+0.j 0.-1.j 0.+0.j 0.+0.j]
[1. 0. 0. 0. 0. 0. 0. 0.]	2	3	[0.+0.j 0.+1.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]

Πίνακας 4.5: Παραδείγματα εκτέλεσης της συνάρτησης y

4.2.5 Πύλη Z

Η συνάρτηση z(state, q, n) εφαρμόζει στο qubit q την πύλη Z στην τρέχουσα κβαντική κατάσταση state.

```
import cupy as cp

def z(state, q, n):
    #Define gates
    i_gate = cp.array([[1, 0], [0, 1]], dtype=cp.float32)
    z_gate = cp.array([[1, 0], [0, -1]], dtype=cp.float32)

    if n==1:
        result = cp.dot(z_gate, state)
        return result

    #Create the gate I⊗.⊗Z⊗..⊗I
    #If Z is at the first qubit, start with Z gate
    if (0==q):
        gate= z_gate
    else:
        gate = i_gate

    for i in range(n-1):
        if q == i+1:
            gate = cp.kron(gate, z_gate)
        else:
            gate = cp.kron(gate, i_gate)

    #Apply the gate to the state vector
```

```
    state = cp.dot(gate, state)

    return state
```

Εικόνα 4.6: Υλοποίηση της πύλης Z με χρήση της βιβλιοθήκης CuPy

Ενδεικτικά παραδείγματα της εκτέλεσης της συνάρτησης αυτής είναι:

state	**q**	**n**	**return**
[0. 1. 0. 0.]	0	2	[0. 1. 0. 0.]
[0. 0. 1. 0. 0. 0. 0. 0.]	0	3	[0. 0. 1. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 0. 0. 1.]	1	3	[0. 0. 0. 0. 0. 0. 0. -1.]

Πίνακας 4.6: Παραδείγματα εκτέλεσης της συνάρτησης z

4.2.6 Πύλη T

Η συνάρτηση t(state, q, n) εφαρμόζει στο qubit q την πύλη T στην τρέχουσα κβαντική κατάσταση state.

```
import cupy as cp

def t(state, q, n):
    #Define gates
    i_gate = cp.array([[1, 0], [0, 1]], dtype=cp.complex64)
    t_gate = cp.array([[1, 0], [0, 0.70710678 + 0.70710678j]], dtype=cp.complex64)

    if n==1:
        result = cp.dot(t_gate, state)
        return result

    #Create the gate I⊗...⊗T⊗...⊗I
    #If T is at the first qubit, start with T gate
    if (0==q):
        gate= t_gate
    else:
        gate = i_gate

    for i in range(n-1):
        if q == i+1:
            gate = cp.kron(gate, t_gate)
        else:
            gate = cp.kron(gate, i_gate)

    #Apply the gate to the state vector
```

```
    state = cp.dot(gate, state)

    return state
```

Εικόνα 4.7: Υλοποίηση της πύλης Z με χρήση της βιβλιοθήκης CuPy

Ενδεικτικά παραδείγματα της εκτέλεσης της συνάρτησης αυτής είναι:

state	**q**	**n**	**return**
[1. 0.]	0	1	[1. 0.]
[0. 0. 1. 0.]	1	2	[0.+0.j 0.+0.j 1.+0.j 0.+0.j]
[1. 0. 0. 0.]	0	2	[1.+0.j 0.+0.j 0.+0.j 0.+0.j]

Πίνακας 4.7: Παραδείγματα εκτέλεσης της συνάρτησης t

4.2.7 Πύλη S

Η συνάρτηση s(state, q, n) εφαρμόζει στο qubit q την πύλη S στην τρέχουσα κβαντική κατάσταση state.

```
import cupy as cp

def s(state, q, n):
    #Define gates
    i_gate = cp.array([[1, 0], [0, 1]], dtype=cp.complex64)
    s_gate = cp.array([[1, 0], [0, 1j]], dtype=cp.complex64)

    if n==1:
        result = cp.dot(s_gate, state)
        return result

    #Create the gate I⊗...⊗S⊗...⊗I
    #If S is at the first qubit, start with S gate
    if (0==q):
        gate= s_gate
    else:
        gate = i_gate

    for i in range(n-1):
        if q == i+1:
            gate = cp.kron(gate, s_gate)
        else:
            gate = cp.kron(gate, i_gate)

    #Apply the gate to the state vector
    state = cp.dot(gate, state)
```

```
    return state
```

Εικόνα 4.8: Υλοποίηση της πύλης Z με χρήση της βιβλιοθήκης CuPy

Ενδεικτικά παραδείγματα της εκτέλεσης της συνάρτησης αυτής είναι:

state	q	n	return
[0. 0. 1. 0.]	0	2	[0.+0.j 0.+0.j 0.+1.j 0.+0.j]
[0. 0. 0. 1.]	0	2	[0.+0.j 0.+0.j 0.+0.j 0.+1.j]
[0. 1. 0. 0. 0. 0. 0. 0.]	0	3	[0.+0.j 1.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]

Πίνακας 4.7: Παραδείγματα εκτέλεσης της συνάρτησης s

4.3 Εφαρμογή πυλών σε 2 qubits

Όλες οι συναρτήσεις αυτής της ενότητας είναι η υλοποίηση κβαντικών πυλών που εφαρμόζονται σε 2 qubits.

4.3.1 Πύλη cX

Η υλοποίηση της πύλης cx διαφέρει λίγο από αυτές που αναφέρθηκαν έως τώρα γιατί η πύλη πρέπει να εφαρμοστεί σε 2 qubits.

Η συνάρτηση έχει σαν ορίσματα:

- state: τρέχουσα κβαντική κατάσταση
- control: qubit ελέγχου
- target: qubit στόχου
- n: ο αριθμός των qubit του κβαντικού κυκλώματος

Η λογική είναι αντίστοιχη με τις προηγούμενες πύλες, πρέπει να εντοπιστεί σε ποιο σημείο πρέπει να εφαρμοστεί η πύλη cX για να χτιστεί το σωστό gate.

$$I \otimes \dots \otimes \dots \otimes cX \otimes I \otimes \dots |q_0 \otimes \dots \otimes q_{control} \otimes q_{target} \otimes \dots \otimes q_{n-1}\rangle$$

Για να υπολογιστεί σωστά η πύλη που θα εφαρμοστεί στο κύκλωμα θα πρέπει να διασφαλιστεί ότι το control και το target είναι διαδοχικά και το control προηγείται του target. Αν διατρέξουμε το state όπως και στις προηγούμενες πύλες μέχρι να εντοπιστεί το qubit ελέγχου, δεν σίγουρο ότι το qubit στόχου θα είναι ακριβώς μετά. Εάν δεν ισχύουν τα παραπάνω θα πρέπει να φέρουμε το state σε μια μορφή ώστε το control να προηγείται του target και να είναι γειτονικά μεταξύ τους.

Για το σκοπό αυτό έχει υλοποιηθεί η συνάρτηση *simple_swap*. Με δεδομένης μια κατάστασης και δυο διαδοχικών qubit εναλλάσσουμε τα στοιχεία μεταξύ τους.

Ορίσματα:

- state: τρέχουσα κβαντική κατάσταση
- q1: εναλλασσόμενο qubit
- q2: εναλλασσόμενο qubit
- n: αριθμός qubit του κυκλώματος

Η λογική υλοποίηση της είναι ίδια ακριβώς με της πύλες H. Σύμφωνα με τη θεωρία της κβαντικής υπολογιστής, αν σε 2 qubits εφαρμοστεί η πύλη swap, τα qubit εναλλάσσονται. Έτσι, αν εφαρμόστεί swap (state, q1, q1, n), θα εναλλαχθούν οι θέσεις τους και θα επιστραφεί η νέα κατάσταση. Αυτό όμως δουλεύει σωστά μόνο όταν τα qubits είναι διαδοχικά, σε διαφορετική περίπτωση το αποτέλεσμα θα είναι λανθασμένο.

```
import cupy as cp

def simple_swap(state, q1, q2, n):
    #Define Gates
    i_gate = cp.array([[1, 0], [0, 1]], dtype=cp.float32)
    swap_gate = cp.array([[1, 0, 0, 0], [0, 0, 1, 0], [0, 1, 0, 0], [0, 0, 0, 1]],
dtype=cp.float32)

    #Create the gate I⊗...I⊗SWAP⊗I⊗...⊗I
    #If SWAP is at the first two qubits, start with swap gate
    if (q1 == 0) or (q2==0):
        gate = swap_gate
        i=2
    else:
        gate = i_gate
        i=1

    while i < n:
        if i == min(q1, q2):
            gate = cp.kron(gate, swap_gate)
            i += 2
```

```
        else:
            gate = cp.kron(gate, i_gate)
            i += 1

    #Apply the gate to the state vector
    state = cp.dot(gate, state)

    return state
```

Εικόνα 4.9: Υλοποίηση της βοηθητικής συνάρτησης simple_swap με χρήση της βιβλιοθήκης CuPy

Στη συνέχεια της υλοποίησης cX, με βάση την συνάρτηση simple_swap, για να μετακινηθούν στη σωστή θέση τα qubits control και target, διατρέχουμε το state |control-target| φορές και εναλλάσσουμε το ένα από αυτά με το γειτονικό τους. Στη συνέχεια, εκτελούμε μια επανάληψη για να υπολογίσουμε το gate με τη σωστή σειρά πυλών και υπολογίζουμε την νέα κατάσταση. Πρέπει όμως να επαναφέρουμε στη κατάσταση στην αρχική της σειρά πριν τις εναλλαγές με τη βοήθεια της λίστα swaps[] μεγέθους nx2. Κάθε γραμμή της swaps κρατά τις εναλλαγές που έγιναν με κάθε κλήση της simple swaps, έχοντας με αυτό το τρόπο κρατημένες όλες τις διαδοχικές εναλλαγές που πραγματοποιήσαμε. Έτσι πριν επιστραφεί η νέα κατάσταση από την cx, διαπερνάμε το swaps από το τέλος προς την αρχή και καλούμε τη simple_swap.

```
import cupy as cp

def cx(state, control, target, n):
    # Define gates
    cx_gate = cp.array([[1, 0, 0, 0],[0, 1, 0, 0],[0, 0, 0, 1],[0, 0, 1, 0]],
dtype=cp.float32)
    i_gate = cp.array([[1, 0], [0, 1]], dtype=cp.float32)

    if control == target:
        print("The control qubit and target qubit must be different.")
        return state
    if n < 2:
        print("At least 2 qubits are required.")
        return state

    swaps = []
    c, t = control, target

    #Ensure control and target are adjacent
    while abs(c - t) > 1:
        if c < t:
            state = simple_swap(state, t - 1, t, n)
            swaps.append((t - 1, t))
            t -= 1
        else:
            state = simple_swap(state, t + 1, t, n)
```

```
            swaps.append((t + 1, t))
            t += 1

    #Ensure control<target
    if c>t:
        state = simple_swap(state, t, c, n)
        swaps.append((t, c))
        c, t = t, c


    #Create the gate I⊗...I⊗CX⊗I⊗...⊗I
    if c == 0:
        gate = cx_gate
        i = 2
    else:
        gate = i_gate
        i = 1

    while i < n:
        if i == min(c, t):
            gate = cp.kron(gate, cx_gate)
            i += 2
        else:
            gate = cp.kron(gate, i_gate)
            i += 1

    #Apply the gate to the state vector
    state = cp.dot(gate, state)

    # Reverse SWAPs to restore original qubit order
    for q1, q2 in reversed(swaps):
        state = simple_swap(state, q1, q2, n)

    return state
```

Εικόνα 4.10: Υλοποίηση της πύλης cX με χρήση της βιβλιοθήκης CuPy

Ενδεικτικά παραδείγματα της εκτέλεσης της συνάρτησης αυτής είναι:

state	control	target	n	return
[1. 0. 0. 0.]	0	1	2	[1. 0. 0. 0.]
[1. 0. 0. 0.]	1	0	2	[1. 0. 0. 0.]
[0. 0. 0. 1.]	0	1	2	[0. 0. 1. 0.]

Πίνακας 4.8: Παραδείγματα εκτέλεσης της συνάρτησης cx

4.3.2 Πύλη SWAP

Όπως η πύλη cX, έτσι και η swap εφαρμόζεται σε 2 qubits. Για να υπολογιστεί σωστά η πύλη που θα εφαρμοστεί στο κύκλωμα θα πρέπει να διασφαλιστεί ότι το control και το target είναι διαδοχικά και το control προηγείται του target.

$$I \otimes \dots \otimes \dots \otimes SWAP \otimes I \otimes \dots |q_0 \otimes \dots \otimes q_a \otimes q_b \otimes \dots \otimes q_{n-1}\rangle$$

Έτσι, η υλοποίηση είναι ακριβώς ίδια με τη cnot. Το μόνο που αλλάζει είναι το cx_gate που εδώ είναι swap_gate.

```
import cupy as cp

def swap(state, control, target, n):
    # Define gates
    swap_gate = cp.array([[1, 0, 0, 0],[0, 0, 1, 0],[0, 1, 0, 0],[0, 0, 0, 1]],
dtype=cp.float32)
    i_gate = cp.array([[1, 0], [0, 1]], dtype=cp.float32)

    if control == target:
        print("The control qubit and target qubit must be different.")
        return state
    if n < 2:
        print("At least 2 qubits are required.")
        return state

    swaps = []
    c, t = control, target

    #Ensure control and target are adjacent
    while abs(c - t) > 1:
        if c < t:
            state = simple_swap(state, t - 1, t, n)
            swaps.append((t - 1, t))
            t -= 1
        else:
            state = simple_swap(state, t + 1, t, n)
            swaps.append((t + 1, t))
            t += 1

    #Ensure control<target
    if c>t:
        state = simple_swap(state, t, c, n)
        swaps.append((t, c))
        c, t = t, c

    #Create the gate I⊗...I⊗SWAP⊗I⊗...⊗I
    if c == 0:
        gate = swap_gate
        i = 2
    else:
        gate = i_gate
```

```
        i = 1

    while i < n:
        if i == min(c, t):
            gate = cp.kron(gate, swap_gate)
            i += 2
        else:
            gate = cp.kron(gate, i_gate)
            i += 1

    #Apply the gate to the state vector
    state = cp.dot(gate, state)

    # Reverse SWAPs to restore original qubit order
    for q1, q2 in reversed(swaps):
        state = simple_swap(state, q1, q2, n)

    return state
```

Εικόνα 4.11: Υλοποίηση της πύλης SWAP με χρήση της βιβλιοθήκης CuPy

Ενδεικτικά παραδείγματα της εκτέλεσης της συνάρτησης αυτής είναι:

state	**control**	**target**	**n**	**return**
[0. 1. 0. 0.]	0	1	2	[0. 0. 1. 0.]
[0. 1. 0. 0.]	0	1	2	[0. 0. 1. 0.]
[0. 0. 0. 0. 1. 0. 0. 0.]	2	0	3	[0. 0. 0. 0. 1. 0. 0. 0.]

Πίνακας 4.9: Παραδείγματα εκτέλεσης της συνάρτησης swap

5 Υλοποίηση κβαντικών κυκλωμάτων σε GPU

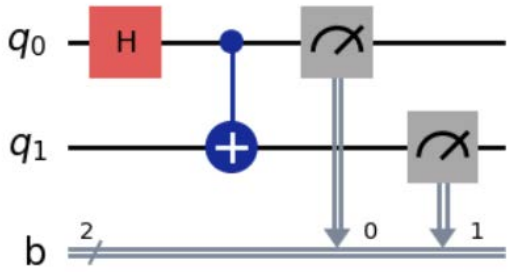
Οι συναρτήσεις που αναφέρθηκαν στο κεφάλαιο 4, χρησιμοποιήθηκαν για την εκτέλεση κβαντικών κυκλωμάτων σε GPU. Το αποτέλεσμα κάθε κυκλώματος συγκρίθηκε με το αποτέλεσμα του ίδιου κυκλώματος σε CPU με χρήση του Qiskit. Η εκτέλεση των κυκλωμάτων έγινε στον επεξεργαστή Intel i5-13500 και κάρτα γραφικών Nvidia RTX 4060 Ti 16GB. Στο κεφάλαιο αυτό αναφέρονται αναλυτικά τα κυκλώματα που υλοποιήθηκαν και τα αποτελέσματά τους.

5.1 Bell States

Υλοποιήθηκαν τα κβαντικά κυκλώματα Bell, με σκοπό την απόδειξη της κβαντικής διεμπλοκής. Τα κυκλώματα αυτά εφαρμόζουν τις πύλες Hadamard (H) και ελεγχόμενη-X (CX) σε δύο qubit, με όλες τις δυνατές αρχικές καταστάσεις εισόδου, ώστε να φανεί ότι το αποτέλεσμα της μέτρησης του ενός qubit εξαρτάται άμεσα από το αποτέλεσμα του άλλου [3], [11].

5.1.1 Bell State |Φ+

Το κβαντικό κύκλωμα που υλοποιήθηκε απεικονίζεται στην εικόνα 5.1.

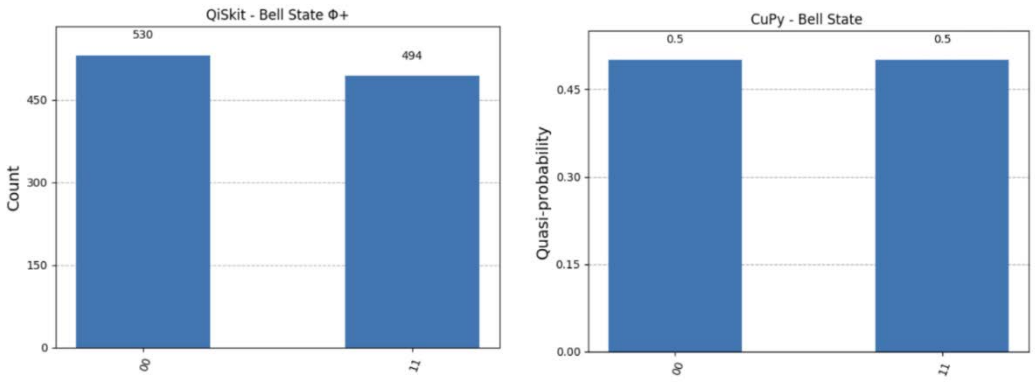


Εικόνα 5.1: Κύκλωμα Bell Φ+

```
import cupy as cp
import numpy as np

from qiskit import *
```

```
from qiskit import QuantumCircuit, transpile
from qiskit_aer import AerSimulator
from qiskit.visualization import plot_histogram, plot_state_city
import qiskit.quantum_info as qi
import matplotlib.pyplot as plt

def bell_state_phi_plus():
    # Qiskit circuit setup
    qr = QuantumRegister(2, 'q')
    cr = ClassicalRegister(2, 'b')
    qc = QuantumCircuit(qr, cr)

    # CuPy initial state |00>
    qubit_list = [0, 0]
    num_qubits = 2
    state = initial_state(qubit_list, num_qubits)

    # Apply H|0>
    state = h(state, 0, num_qubits)
    qc.h(0)

    # Apply CX|01>
    state = cx(state, 0, 1, num_qubits)
    qc.cx(0, 1)

    #Qiskit Measure
    qc.measure(qr, cr)
    qc.draw('mpl')

    # Qiskit simulation
    simulator = AerSimulator()
    compiled = transpile(qc, simulator)
    result = simulator.run(compiled).result()
    counts = result.get_counts()
    plot_histogram(counts, title="QiSkit - Bell State Φ+")

    # Plot CuPy result
    probabilities = cp.abs(state) ** 2
    probs_np = cp.asnumpy()

    format_str = '{0:0' + str(num_qubits) + 'b}'
    labels = [format_str.format(i)[::-1] for i in range(2 ** num_qubits)]
    filtered = {labels[i]: probs_np[i] for i in range(len(labels)) if probs_np[i] > 1e-
6}

    plot_histogram(filtered, title="CuPy - Bell State Φ+")
    plt.show()
```

Εικόνα 5.2: Υλοποίηση του κβαντικού κυκλώματος Bell Φ+

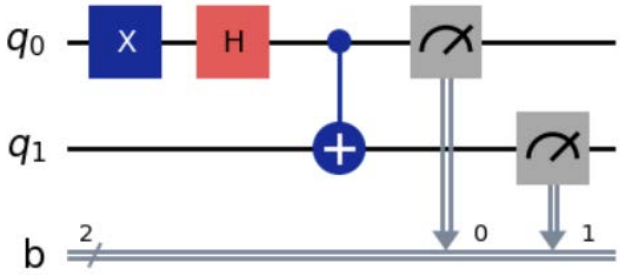


Εικόνα 5.4: Αποτέλεσμα των Qiskit και Cupy του Bell State Φ+

Παρατηρώντας την Εικόνα 5.4, διαπιστώνουμε ότι τα αποτελέσματα είναι ουσιαστικά τα ίδια. Η διαφορά οφείλεται στο γεγονός ότι το αποτέλεσμα του Qiskit δεν προκύπτει από την ιδανική κβαντική κατάσταση, αλλά από τον προσομοιωτή Aer Simulator, ο οποίος πραγματοποιεί επαναλαμβανόμενες μετρήσεις (shots). Αυτό εξηγεί και τις μικρές αποκλίσεις στα ύψη των στηλών, καθώς οι πιθανότητες υπολογίζονται στατιστικά.

5.1.2 Bell State |Φ-

Το κβαντικό κύκλωμα που υλοποιήθηκε απεικονίζεται στην εικόνα 5.5

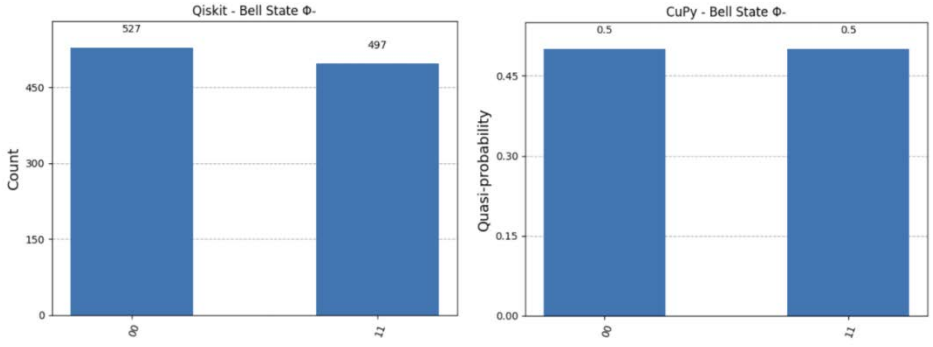


Είκόνα 5.5: Κύκλωμα Bell Φ-

```
import cupy as cp
import numpy as np

from qiskit import *
from qiskit import QuantumCircuit, transpile
from qiskit_aer import AerSimulator
from qiskit.visualization import plot_histogram, plot_state_city
import qiskit.quantum_info as qi
import matplotlib.pyplot as plt
```

```
def bell_state_phi_minus():
    # Qiskit circuit setup
    qr = QuantumRegister(2, 'q')
    cr = ClassicalRegister(2, 'b')
    qc = QuantumCircuit(qr, cr)

    # CuPy initial state |00>
    qubit_list = [0, 0]
    num_qubits = 2
    state = initial_state(qubit_list, num_qubits)

    # Apply X|0>
    state = x(state, 0, num_qubits)
    qc.x(0)

    # Apply H|0>
    state = h(state, 0, num_qubits)
    qc.h(0)

    # Apply CX|01>
    state = cx(state, 0, 1, num_qubits)
    qc.cx(0, 1)

    # Measure
    qc.measure(qr, cr)
    qc.draw('mpl')

    # Qiskit simulation
    simulator = AerSimulator()
    compiled = transpile(qc, simulator)
    result = simulator.run(compiled).result()
    counts = result.get_counts()
    plot_histogram(counts, title="Qiskit - Bell State Φ-")

    # Plot CuPy result
    probabilities = cp.abs(state) ** 2
    probs_np = cp.asnumpy(probabilities)
    format_str = '{0:0' + str(num_qubits) + 'b}'
    labels = [format_str.format(i)[::-1] for i in range(2 ** num_qubits)]
    filtered = {labels[i]: probs_np[i] for i in range(len(labels)) if probs_np[i] > 1e-
6}

    plot_histogram(filtered, title="CuPy - Bell State Φ-")
    plt.show()
```

Εικόνα 5.6: Υλοποίηση του κβαντικού κυκλώματος Bell Φ-

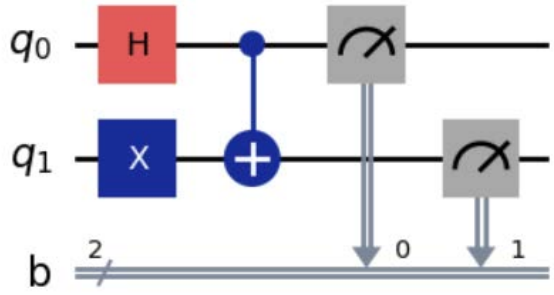


Εικόνα 5.7: Αποτέλεσμα των Qiskit και Cupy υλοποιήσεων για το Bell State Φ-

Και εδώ παρατηρείτε ότι τα αποτελέσματα είναι ίδια. Οι τις μικρές αποκλίσεις στα ύψη των στηλών, οφείλονται στον προσομοιωτή Aer Simulator της βιβλιοθήκης Qiskit.

5.1.3 Bell State |Ψ+

Το κβαντικό κύκλωμα που υλοποιήθηκε απεικονίζεται στην εικόνα 5.8.

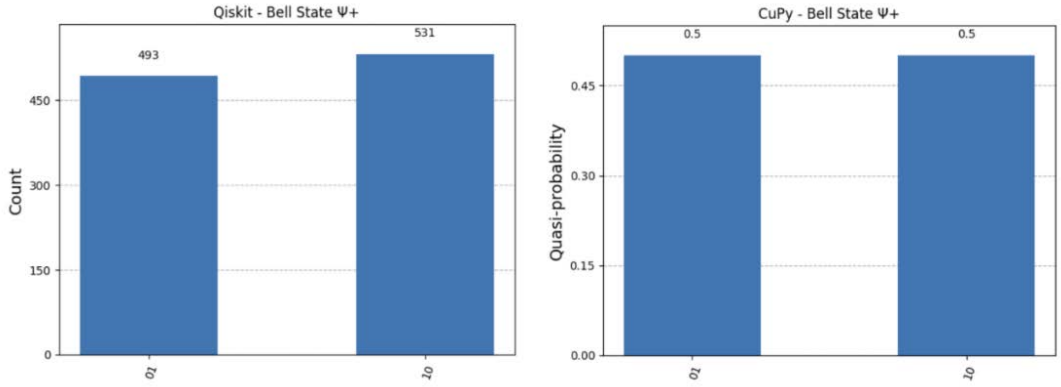


Εικόνα 5.8: Κύκλωμα Bell StateΨ+

```
import cupy as cp
import numpy as np

from qiskit import *
from qiskit import QuantumCircuit, transpile
from qiskit_aer import AerSimulator
from qiskit.visualization import plot_histogram, plot_state_city
import qiskit.quantum_info as qi
import matplotlib.pyplot as plt

def bell_state_psi_plus():
    # Qiskit circuit setup
    qr = QuantumRegister(2, 'q')
    cr = ClassicalRegister(2, 'b')
    qc = QuantumCircuit(qr, cr)

    # CuPy initial state |00⟩
    qubit_list = [0, 0]
    num_qubits = 2
    state = initial_state(qubit_list, num_qubits)

    # Apply H|0>
    state = h(state, 0, num_qubits)
    qc.h(0)

    # Apply X|1>
    state = x(state, 1, num_qubits)
    qc.x(1)

    # Apply CX|01>
    state = cx(state, 0, 1, num_qubits)
    qc.cx(0, 1)
```

```
    # Measure
    qc.measure(qr, cr)
    qc.draw('mpl')

    # Qiskit simulation
    simulator = AerSimulator()
    compiled = transpile(qc, simulator)
    result = simulator.run(compiled).result()
    counts = result.get_counts()
    plot_histogram(counts, title="Qiskit - Bell State Ψ+")

    # Plot CuPy result
    probabilities = cp.abs(state) ** 2
    probs_np = cp.asnumpy(probabilities)
    format_str = '{0:0' + str(num_qubits) + 'b}'
    labels = [format_str.format(i)[::-1] for i in range(2 ** num_qubits)]
    filtered = {labels[i]: probs_np[i] for i in range(len(labels)) if probs_np[i] > 1e-
6}

    plot_histogram(filtered, title="CuPy - Bell State Ψ+")
    plt.show()
```

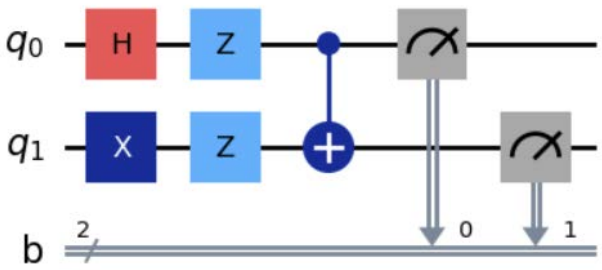
Εικόνα 5.9: Υλοποίηση του κβαντικού Bell State Ψ+



Εικόνα 5.10: Αποτέλεσμα των Qiskit και CuPy του Bell State Ψ+

5.1.4 Bell State |Ψ-

Το κβαντικό κύκλωμα που υλοποιήθηκα απεικονίζεται στην εικόνα 5.11.

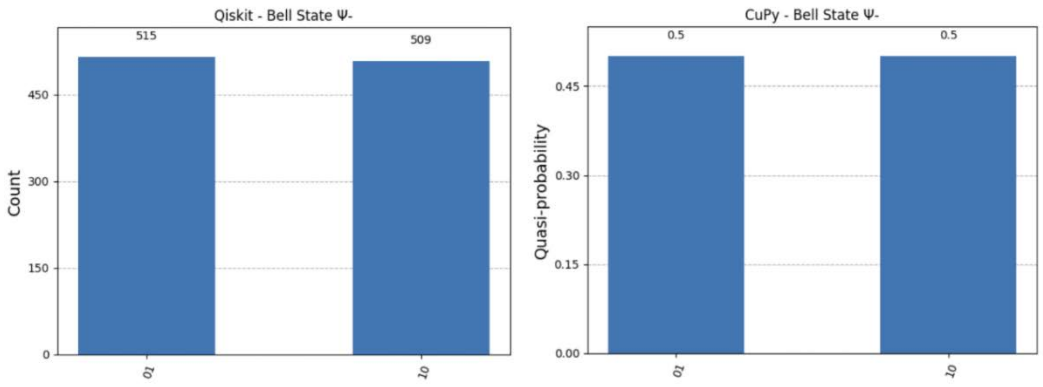


Εικόνα 5.11: Κύκλωμα Bell Ψ-

```
import cupy as cp
import numpy as np

from qiskit import *
from qiskit import QuantumCircuit, transpile
from qiskit_aer import AerSimulator
from qiskit.visualization import plot_histogram, plot_state_city
import qiskit.quantum_info as qi
import matplotlib.pyplot as plt

def bell_state_psi_minus():
    # Qiskit circuit setup
    qr = QuantumRegister(2, 'q')
    cr = ClassicalRegister(2, 'b')
    qc = QuantumCircuit(qr, cr)

    # CuPy initial state |00⟩
    qubit_list = [0, 0]
    num_qubits = 2
    state = initial_state(qubit_list, num_qubits)

    # Apply H|0>
    state = h(state, 0, num_qubits)
    qc.h(0)

    # Apply X|1>
    state = x(state, 1, num_qubits)
    qc.x(1)

     # Apply Z|0>
    state = z(state, 0, num_qubits)
    qc.z(0)

    # Apply Z|1>
    state = z(state, 1, num_qubits)
    qc.z(1)

    # Apply CX|01>
    state = cx(state, 0, 1, num_qubits)
    qc.cx(0, 1)

    # Measure
    qc.measure(qr, cr)
    qc.draw('mpl')

    # Qiskit simulation
    simulator = AerSimulator()
    compiled = transpile(qc, simulator)
    result = simulator.run(compiled).result()
    counts = result.get_counts()
    plot_histogram(counts, title="Qiskit - Bell State Ψ-")

    # Plot CuPy result
    probabilities = cp.abs(state) ** 2
    probs_np = cp.asnumpy(probabilities)

    format_str = '{0:0' + str(num_qubits) + 'b}'
    labels = [format_str.format(i)[::-1] for i in range(2 ** num_qubits)]
```

```
    filtered = {labels[i]: probs_np[i] for i in range(len(labels)) if probs_np[i] > 1e-
6}

    plot_histogram(filtered, title="CuPy - Bell State Ψ-")
    plt.show()
```

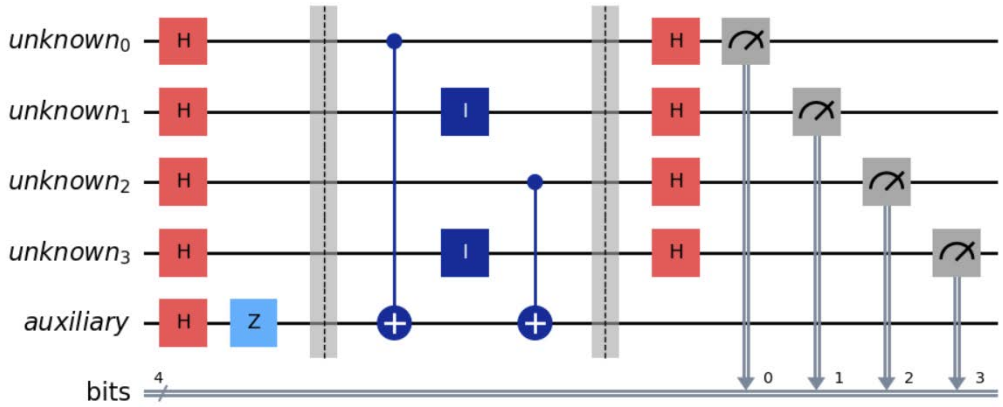
Εικόνα 5.12: Υλοποίηση του κβαντικού Bell State Ψ+



Εικόνα 5.10: Αποτέλεσμα των Qiskit και CuPy του Bell State Ψ-

5.2 Υλοποίηση αλγορίθμου Bernstein-Vazirani

Στα πλαίσια της εργασίας αυτής, υλοποιήθηκε ο αλγόριθμος Bernstein-Vazirani. Ο κβαντικός αυτός αλγόριθμος αυτός βρίσκει μια κρυφή λίστα που αποτελείται από 0 και 1. Σε ένα απλό υπολογιστή, ο αλγόριθμος θα έλεγχε ένα προς ένα τα στοιχεία της λίστας για να τα βρει. Αντίθετα, με τη χρήση της πύλης H, ο κβαντικός αλγόριθμος ελέγχει όλα τα σενάρια για κάθε στοιχεία ταυτόχρονα [3].



Εικόνα 5.11: Κύκλωμα αλγορίθμου Bernstein-Vazirani για [1, 0, 1, 0]

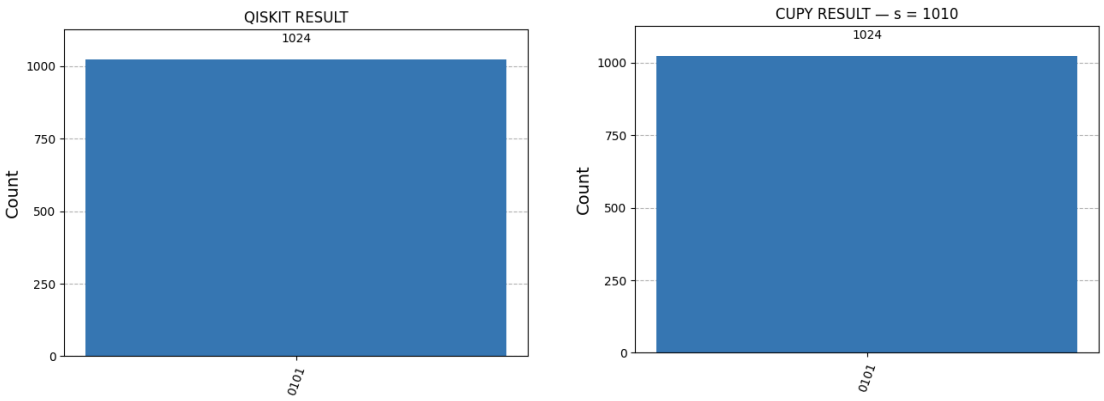
```
import cupy as cp
import numpy as np

from qiskit import *
from qiskit import QuantumCircuit, transpile
from qiskit_aer import AerSimulator
from qiskit.visualization import plot_histogram, plot_state_city
import qiskit.quantum_info as qi
import matplotlib.pyplot as plt

def bernstein_vazirani_qiskit(secret):
    n = len(secret)

    qr1 = QuantumRegister(n,'unknown')
    qr2 = QuantumRegister(1, 'auxiliary')
    cr = ClassicalRegister(n, 'bits')
    qc = QuantumCircuit(qr1, qr2, cr)

    qc.h(range(n+1))
    qc.z(n)

    qc.barrier()
    for i in range(n):
        if secret[i] == 1:
            qc.cx(i,n)
        else:
            qc.id(i)
    qc.barrier()
    qc.h(range(n))

    qc.measure(qr1, cr)
    qc.draw('mpl')

    # Qiskit simulation
    simulator = AerSimulator()
    compiled = transpile(qc, simulator)
    result = simulator.run(compiled).result()
    counts = result.get_counts()
    plot_histogram(counts, title="QISKIT RESULT")
```

Εικόνα 5.12: Υλοποίηση αλγορίθμου Bernstein-Vazirani με Qiskit

```
import cupy as cp
import numpy as np

from qiskit import *
from qiskit import QuantumCircuit, transpile
from qiskit_aer import AerSimulator
from qiskit.visualization import plot_histogram, plot_state_city
import qiskit.quantum_info as qi
import matplotlib.pyplot as plt

def bernstein_vazirani_cupy(secret):
    import cupy as cp
    import numpy as np
    import matplotlib.pyplot as plt
    from collections import Counter

    n = len(secret)
```

```
    qubit_list = [0] * num_qubits

    state = initial_state(qubit_list, num_qubits)

    #Hadamard on all qubits
    for i in range(num_qubits):
        state = h(state, i, num_qubits)

    #Z on last qubit
    state = z(state, n, num_qubits)

    #cX on
    for i in range(n):
        if secret[i] == 1:
            state = cx(state, i, n, num_qubits)
        else:
            state = id(state, i, num_qubits)

    #Hadamard on all qubits
    for i in range(n):
        state = h(state, i, num_qubits)

    #CuPy measurement simulation
    probabilities = cp.abs(state) ** 2
    probs_np = cp.asnumpy(probabilities)

    format_str = '{0:0' + str(num_qubits) + 'b}'
    labels = [format_str.format(i)[::-1] for i in range(2 ** num_qubits)]

    #Simulate shots and extract only input qubits (not auxiliary)
    shots = 1024
    samples = np.random.choice(labels, size=shots, p=probs_np)
    input_only = [''.join(s[1:]) for s in samples]  # remove aux qubit (which is qubit 0
rightmost)
    counts = dict(Counter(input_only))

    #Plot
    from qiskit.visualization import plot_histogram
    plot_histogram(counts, title=f"CUPY RESULT — s = {''.join(map(str, secret))}")
    plt.show()
```

Εικόνα 5.13: Υλοποίηση αλγορίθμου Bernstein-Vazirani με CuPy



Εικόνα 5.14: Σύγκριση αποτελεσμάτων αλγορίθμου Bernstein-Vazirani Qiksit με CuPy

Ο αλγόριθμος έτρεξε με secret=[1 0 1 0]. Παρατηρούμε ότι το αποτέλεσμα και στις δυο υλοποιήσεις να ταυτίζεται στο άγνωστο secret που δόθηκε σαν είσοδος.

5.3 Προσαρμοσμένο Κβαντικό Κύκλωμα

Για την εκτέλεση οποιαδήποτε κβαντικής πύλης ή κυκλώματος υλοποιήθηκε η συνάρτηση custom_quantum_circuit. Εισάγοντας από τη κονσόλας όλα τα απαραίτητα δεδομένα που χρειάζονται ένα κβαντικό κύκλωμα, όπως τον αριθμό των qubits που θα έχει το κύκλωμα, ποια θα είναι η αρχική κατάσταση ή ποια πύλη θα εφαρμοστεί και σε ποιο qubit. Με αυτόν το τρόπο, προσομοιώνουμε ένα κβαντικό κύκλωμα δυναμικά χωρίς την ανάγκη δημιουργίας νέας συνάρτησης για κάθε διαφορετικό κύκλωμα.

Πρώτα εισάγουμε τον αριθμό των qubit που θα έχει το κύκλωμα. Στην συνέχεια επιλέγουμε την αρχική κατάσταση για κάθε qubit. Η συνάρτηση έχει υλοποιηθεί μόνο για qubit 0 και 1. Εφόσον έχει οριστεί το κβαντικό κύκλωμα επιλέγουμε την πύλη που θέλουμε να εφαρμόσουμε και σε ποιο qubit. Μόλις εφαρμοστεί η πύλη η διαδικασία επαναλαμβάνεται. Αν αντί για πύλη εισάγουμε end, εμφανίζεται το κύκλωμα που φτιάξαμε και το αποτέλεσμα του κυκλώματος με δυο ιστογράμματα:

- Ιστόγραμμα με το αποτέλεσμα της εκτέλεσης του κυκλώματος σε CPU με χρήση της βιβλιοθήκης Qiskit
- Ιστόγραμμα με το αποτέλεσμα της εκτέλεσης του κυκλώματος σε GPU με χρήση της βιβλιοθήκης Qiskit

```
import cupy as cp
import numpy as np

from qiskit import *
from qiskit import QuantumCircuit, transpile
from qiskit_aer import AerSimulator
from qiskit.visualization import plot_histogram, plot_state_city
import qiskit.quantum_info as qi
import matplotlib.pyplot as plt

def custom_quantum_circuit():
    qubit_list = []

    num_qubits = int(input("How many qubits will the circuit have?"))

    qr = QuantumRegister(num_qubits, 'q')
    cr = ClassicalRegister(num_qubits, 'bits')
    qc = QuantumCircuit(qr, cr)
```

```
    print("Set initial state for each qubit (0 or 1):")
    for i in range(num_qubits):
        while True:
            val = int(input(f"q{i} = |0> ή |1>: "))
            if val==0 or val==1:
                qubit_list.append(int(val))

                #Set the default Qiskit initial state according to input
                bit = int(val)
                if bit == 1:
                    qc.x(i)

                break
            else:
                print("Not valid input. Set 0 or 1.")

    # Initial cupy sate
    state = initial_state(qubit_list, num_qubits)


    print("\nStart Quantum Circuit\n")

    while True:
        if num_qubits >= 2:
            print("Available Gates: id, h, x, y, z, s, t, cx, swap")
        else:
            print("Available Gates: id, h, x, y, z, s, t")

        gate_input = input("Choose gate name or choose 'end' to see the result:
").strip().lower()

        if gate_input == 'end':
            break

        if gate_input in ['id','h', 'x', 'y', 'z', 's', 't']:
            if num_qubits>1:
                qubit = int(input("Select qubit for this gate: "))
            else:
                qubit = qubit_list[0]

            if (qubit>=0) and (qubit<=num_qubits):
                q = int(qubit)
                if 0 <= q < num_qubits:
                    if gate_input == 'id':
                        state = id(state, q, num_qubits)
                        qc.id(q)
                    elif gate_input == 'h':
                        state = h(state, q, num_qubits)
                        qc.h(q)
                    elif gate_input == 'x':
                        state = x(state, q, num_qubits)
                        qc.x(q)
                    elif gate_input == 'y':
                        state = y(state, q, num_qubits)
                        qc.y(q)
                    elif gate_input == 'z':
                        state = z(state, q, num_qubits)
                        qc.z(q)
                    elif gate_input == 's':
                        state = s(state, q, num_qubits)
```

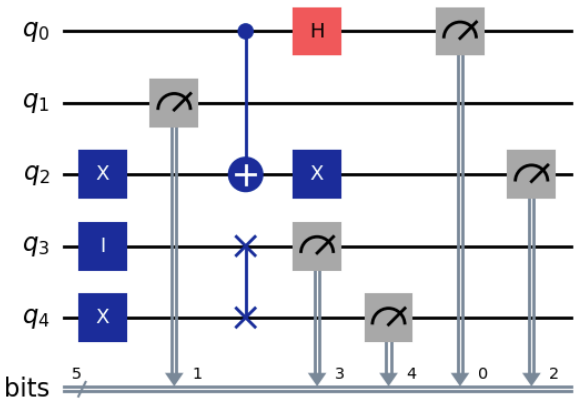
```
                    qc.s(q)
                elif gate_input == 't':
                    state = t(state, q, num_qubits)
                    qc.t(q)
                print(f"Applied {gate_input.upper()} at q{q}>")
            else:
                print("Invalid qubit number.")
        else:
            print("Must set number")

    elif gate_input in ['cx', 'swap']:
        control = int(input("Qubit control: "))
        target = int(input("Qubit target: "))
        if control != target and 0 <= control < num_qubits and 0 <= target <
num_qubits:
            if gate_input == 'cx':
                state = cx(state, control, target, num_qubits)
                qc.cx(control, target)
            elif gate_input == 'swap':
                state = swap(state, control, target, num_qubits)
                qc.swap(control, target)
            print(f"Applied {gate_input.upper()} at {control} -> {target}")
        else:
            print("Invalid qubit pair (they must be different and within valid
range)")

    else:
        print("Invalid Gate!")

#Measure and draw the circuit in Qiskit
print("\nFinal quantum circuit:")
qc.measure(qr, cr)
qc.draw('mpl')


#Plot Qiskit result
simulator = AerSimulator()
qc = transpile(qc, simulator)
result = simulator.run(qc).result()
counts = result.get_counts(qc)
plot_histogram(counts, title='QISKIT RESULT')


# Plot CuPy result
probabilities = cp.abs(state) ** 2
probs_np = cp.asnumpy(probabilities)

format_str = '{0:0' + str(num_qubits) + 'b}'
labels = [format_str.format(i)[::-1] for i in range(2 ** num_qubits)]

filtered = {labels[i]: probs_np[i] for i in range(len(labels)) if probs_np[i] > 1e-
6}

plot_histogram(filtered, title="CUPY RESULT")
plt.show()
```

Εικόνα 5.15: Υλοποίηση συνάρτησης custom_quantum_circuit

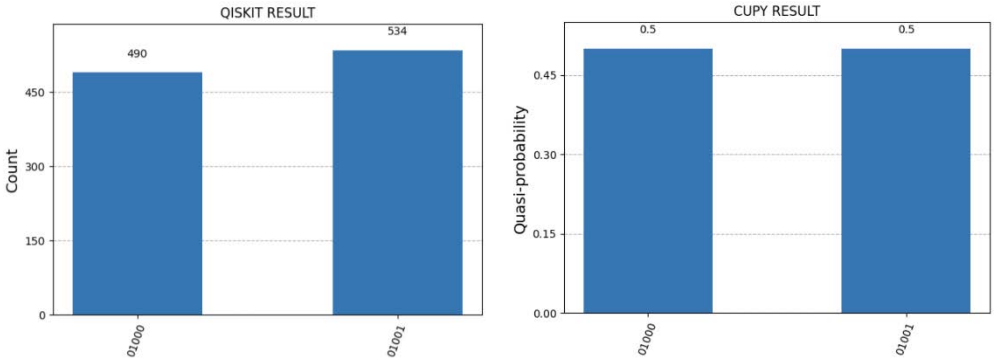
Το παρακάτω κβαντικό κύκλωμα δημιουργήθηκε τυχαίο με χρήση της custom_quantum_circuit.

```
How many qubits will the circuit have? 5
Set initial state for each qubit (0 or 1):
q0 = |0> ή |1>:  0
q1 = |0> ή |1>:  0
q2 = |0> ή |1>:  1
q3 = |0> ή |1>:  0
q4 = |0> ή |1>:  1

Start Quantum Circuit

Available Gates: id, h, x, y, z, s, t, cx, swap
Choose gate name or choose 'end' to see the result:  cx
Qubit control:  0
Qubit target:  2
Applied CX at 0 -> 2
Available Gates: id, h, x, y, z, s, t, cx, swap
Choose gate name or choose 'end' to see the result:  h
Select qubit for this gate:  0
Applied H at q0>
Available Gates: id, h, x, y, z, s, t, cx, swap
Choose gate name or choose 'end' to see the result:  id
Select qubit for this gate:  3
Applied ID at q3>
Available Gates: id, h, x, y, z, s, t, cx, swap
Choose gate name or choose 'end' to see the result:  x
Select qubit for this gate:  2
Applied X at q2>
Available Gates: id, h, x, y, z, s, t, cx, swap
Choose gate name or choose 'end' to see the result:  swap
Qubit control:  3
Qubit target:  4
Applied SWAP at 3 -> 4
Available Gates: id, h, x, y, z, s, t, cx, swap
Choose gate name or choose 'end' to see the result:  end

Final quantum circuit:
```

Εικόνα 5.16: Επιλογές μέσω της κονσόλας για τη δημιουργία κβαντικού κυκλώματος



Εικόνα 5.17: Κβαντικό κύκλωμα με χρήση της custom_quantum_circuit



Εικόνα 5.18: Αποτέλεσμα του κυκλώματος της εικόνας 5.17

6 Συμπεράσματα

Στην παρούσα εργασία υλοποιήθηκαν βασικές κβαντικές πύλες μέσω κατάλληλων συναρτήσεων με χρήση του τανυστικού γινομένου της βιβλιοθήκης CuPy. Η προσέγγιση αυτή επιτρέπει την αναπαράσταση και επεξεργασία κβαντικών καταστάσεων απευθείας στην κάρτα γραφικών, αξιοποιώντας τη δυνατότητα παραλληλισμού των σύγχρονων GPUs.

Μέσα από την ανάπτυξη βασικών κβαντικών πυλών (όπως H, X, Z, CNOT, SWAP) και την κατασκευή απλών κβαντικών κυκλωμάτων (όπως τα Bell states και ο αλγόριθμος Bernstein–Vazirani), διαπιστώθηκε ότι η GPU μπορεί να χρησιμοποιηθεί αποτελεσματικά για την προσομοίωση κβαντικών υπολογισμών. Για την επιβεβαίωση της ακρίβειας των υλοποιήσεων πραγματοποιήθηκε μέσω σύγκρισης των αποτελεσμάτων με τα αντίστοιχα του Qiskit. Οι συναρτήσεις που υλοποιήθηκαν απέδωσαν τα αναμενόμενα αποτελέσματα, επιβεβαιώνοντας θεωρητικά σχήματα όπως η δημιουργία υπέρθεση.

Συνοψίζοντας, η εργασία επιβεβαιώνει ότι η χρήση GPU αποτελεί μια ρεαλιστική και αποδοτική προσέγγιση για την προσομοίωση κβαντικών λειτουργιών και μπορεί να αποτελέσει βάση για μελλοντική έρευνα πάνω σε πιο σύνθετους αλγόριθμους και μεγαλύτερης κλίμακας συστήματα.

Βιβλιογραφία

[1] S. Gamble, ‘Quantum Computing: What It Is, Why We Want It, and How We’re Trying to Get It’, στο *Frontiers of Engineering: Reports on Leading-Edge Engineering from the 2018 Symposium*, National Academies Press (US), 2019. https://www.ncbi.nlm.nih.gov/books/NBK538701/

[2] S. M.-L. Pfaendler, K. Konson, και F. Greinert, ‘Advancements in Quantum Computing—Viewpoint: Building Adoption and Competency in Industry’, *Datenbank-Spektrum*, τ. 24, τχ. 1, σελ. 5–20, Μαρτίου 2024, doi: 10.1007/s13222-024-00467-4.

[3] ‘Πανεπιστημίο Θεσσαλίας | Πολυπύρηνος και Κβαντικός Προγραμματισμός’. https://eclass.uth.gr/courses/DS_P_101/

[4] ‘Qiskit | IBM Quantum Computing’. https://www.ibm.com/quantum/qiskit

[5] The cuQuantum development team, *NVIDIA cuQuantum SDK*. (3 Νοέμβριος 2023). Zenodo. doi: 10.5281/ZENODO.6385574.

[6] D. Kirk και W. W. Hwu, *Programming massively parallel processors: a hands-on approach*, 2. ed. Amsterdam: Elsevier, Morgan Kaufmann, 2013.

[7] B. Oancea, T. Andrei, και R. M. Dragoescu, ‘GPGPU Computing’, 29 Αύγουστος 2014, *arXiv*: arXiv:1408.6923. http://arxiv.org/abs/1408.6923

[8] ‘What Is Quantum Computing? | IBM’. https://www.ibm.com/topics/quantum-computing

[9] ‘Quantum Computing Explained’, *NIST*, Μαρτίου 2025, https://www.nist.gov/quantum-information-science/quantum-computing-explained

[10] D. Voorhoede, ‘What is a qubit?’, Quantum Inspire. https://www.quantum-inspire.com/kbase/what-is-a-qubit/

[11] I. Καραφυλλίδης, *Κβαντική Υπολογιστική*. 2015.

[12] N. a V. for Y. G. A. C. is a managed vector database built on M. perfect for building G. applications T. Free, ‘What is a quantum register, and how does it store quantum information?’ https://milvus.io/ai-quick-reference/what-is-a-quantum-register-and-how-does-it-store-quantum-information

[13] Adjunct Professor, Golden Gate University, Ageno School of Business, Data Analytic, San Francisco, California, USA, B. Zohuri, F. M. Rahmani, και Professor of Finance and Director of MBA School of Business and Management, National University, San Diego, California, USA, ‘What is Quantum Computing and How it Works’, *J. Mater. Sci. Manuf. Res.*, σελ. 1–5, Δεκεμβρίου 2020, doi: 10.47363/JMSMR/2020(1)104.

[14] M. A. Nielsen και I. L. Chuang, ‘Quantum Computation and Quantum Information: 10th Anniversary Edition’, Higher Education from Cambridge University Press.

[15] *Qiskit/qiskit-qasm3-import*. (19 Ιούνιος 2024). Python. Qiskit. https://github.com/Qiskit/qiskit-qasm3-import

[16] ‘IBM Quantum Documentation’, IBM Quantum Documentation. https://docs.quantum.ibm.com/docs.quantum.ibm.com

[17] ‘Simulators - Qiskit Aer 0.17.1’. https://qiskit.github.io/qiskit-aer/tutorials/1_aer-simulator.html

[18] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, και J. C. Phillips, 'GPU Computing', *Proc. IEEE*, τ. 96, τχ. 5, σελ. 879–899, Φεβρουαρίου 2008, doi: 10.1109/JPROC.2008.917757.
[19] 'Scalable Parallel Programming with CUDA | Request PDF', *ResearchGate*, doi: 10.1145/1401132.1401152.
[20] NVIDIA, 'CUDA C Programming Guide'. 2024. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
[21] 'AMD ROCm documentation — ROCm Documentation'. https://rocm.docs.amd.com/en/latest/
[22] V. Volkov, 'Better Performance at Lower Occupancy'.
[23] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, και W. W. Hwu, 'Optimization principles and application performance evaluation of a multithreaded GPU using CUDA', στο *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, Salt Lake City UT USA: ACM, Φεβρουαρίου 2008, σελ. 73–82. doi: 10.1145/1345206.1345220.
[24] 'NVIDIA H100 Tensor Core GPU', NVIDIA. https://www.nvidia.com/en-us/data-center/h100/
[25] 'Figure 3: Memory Model (from [5])', ResearchGate. https://www.researchgate.net/figure/Memory-Model-from-5_fig3_221396861
[26] 'CPU/SIMD optimizations — NumPy v2.3 Manual'. https://numpy.org/doc/stable/reference/simd/
[27] 'Manuals for Intel® 64 and IA-32 Architectures', Intel. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html
[28] 'Gamer Resources, Articles, News, and How To's - Intel'. https://www.intel.com/content/www/us/en/gaming/resources/overview.html
[29] 'CUDA C Programming Guide'.
[30] '1-Schéma de principe des architectures CPU et GPU (documentation NVIDIA) | Download Scientific Diagram'. https://www.researchgate.net/figure/Schema-de-principe-des-architectures-CPU-et-GPU-documentation-NVIDIA_fig33_299748956
[31] 'NVIDIA Developer', NVIDIA Developer. https://developer.nvidia.com/
[32] 'NumPy'. https://numpy.org/
[33] 'CuPy', CuPy. https://cupy.dev/