From Classical to Quantum: Parallelizing AES, RSA Algorithm on CPU, GPU and Shor's Algorithm on Quantum Computers

MINI PROJECT REPORT

SUBMITTED BY:

A. NIHARIKA VARMA(100521729005)

B. YOGESHWAR (100521729011)

CH.MANASA GANGOTRI(10052129014)

In partial fulfillment for the award of

the degree of

BACHELOR OF ENGINEERING

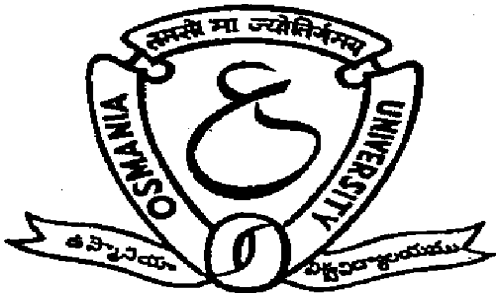
IN

Computer Science and Engineering (AIML)

University College of Engineering

OSMANIA UNIVERSITY: HYDERABAD-500007

JUNE-2024.



Department of Computer Science and Engineering,

University College of Engineering,Osmania University.

CERTIFICATE:

This is to certify that this project entitled is a bonafide work carried out by

A Niharika varma(100521729005),

B Yogeshwar (100521729011),

CH Manasa Gangotri(100521729014)

from the department of **Computer Science and Engineering(AIML)** in the partial fulfillment of academic requirements in VI-Semester(Sixth Semester) in **Artificial Intelligence & Machine Learning (CSE) from University College of Engineering(A), Osmania University**, Hyderabad

Project Guide	**Head of the Dept.**
Prof. K. Shyamala	Prof. P. V. SUDHA
Dept. of CSE, UCEOU	Dept. of CSE, UCEOU

STUDENT DECLARATION

We declare that the work reported in the project report entitled "**From Classical to Quantum: Parallelizing AES, RSA Algorithm on CPU, GPU and Shor's Algorithm on Quantum Computers**"submitted by A. Niharika Varma, B. Yogeshwar, CH. Manasa Gangotri is a record of the work done by us in the DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING, UNIVERSITY COLLEGE OF ENGINEERING,OSMANIA UNIVERSITY. No part of the report is copied from books/ journals/internet and wherever referred, the same has been duly acknowledged in the text. The reported data is based on the work done entirely by us and not copied from any other source or submitted to any other Institute or University for the award of a degree or diploma.

SIGNATURES:

A. Niharika Varma:

B. Yogeshwar:

CH. Manasa Gangotri:

ACKNOWLEDGEMENT

I would like to express my deep sense of gratitude and whole-hearted thanks to my project guide **Prof. K. Shyamala**, Associate Professor, **Department of Computer Science and Engineering, University College of Engineering Osmania University**, for giving me the privilege of working under her guidance, with tremendous support and cogent discussion, constructive criticism and encouragement throughout this dissertation work carrying out the project work.

I also **thank Dr. P. V. SUDHA, Head, Department of Computer Science and Engineering** for her support from the Department and allowing all the resources available to us students.

I also extend my thanks to the entire faculty of the Department of Computer Science and Engineering, University College of Engineering, Osmania University who encourages us throughout the course of our Bachelor degree and allows us to use the many resources present in the department. Our sincere thanks to our parents and friends for their valuable suggestions, morals, strength and support for the completion of our project.

Anumola Niharika Varma (100521729005)

Bhukya Yogeshwar(100521729005)

Chettaboina Manasa Gangotri(100521729014)

ABSTRACT

The project focuses on the implementation and analysis of the AES and RSA algorithms on both CPU and GPU, and executing Shor's algorithm on a quantum backend. The aim is to explore the efficiency and feasibility of quantum computing in solving cryptographic problems.

In this project, we explore the parallelization of critical sequential algorithms, focusing on AES (Advanced Encryption Standard), RSA (Rivest-Shamir-Adleman), and Shor's algorithm, to analyze their performance across different computing architectures—CPUs, GPUs, and QPUs. These algorithms are fundamental in the fields of cryptography and quantum computing. By parallelizing these algorithms, we aim to enhance computational efficiency and evaluate key performance metrics such as execution time on various hardware platforms. This project is significant because it seeks to improve the speed and security of cryptographic algorithms like AES and RSA and to demonstrate the potential of quantum processing units (QPUs) through Shor's algorithm. Through a comparative analysis, we intend to identify the most efficient platforms for specific computational tasks. Our methodology includes implementing both the original sequential and parallelized versions of these algorithms, executing them on CPUs, GPUs, and QPUs, and measuring their performance. The expected outcomes include detailed performance profiles, insights into the challenges and benefits of parallelization, and recommendations for optimal computational platforms for different types of algorithms.

TABLE OF CONTENTS

CHAPTER 1

1.1.INTRODUCTION

Quantum computing has the potential to revolutionize various fields, particularly cryptography. Traditional cryptographic algorithms, such as AES and RSA, rely on computational hardness assumptions which can be threatened by quantum algorithms like Shor's algorithm. This project aims to understand and implement these algorithms, comparing their performance on classical (CPU and GPU) and quantum backends.

Parallelization of Sequential Algorithms:

We will focus on parallelizing well-known sequential algorithms such as AES (Advanced Encryption Standard), RSA (Rivest-Shamir-Adleman), and Shor's algorithm. These algorithms are fundamental in the fields of cryptography and quantum computing.We will measure and analyze the time computations required for these algorithms on different hardware architectures, specifically CPUs, GPUs, and QPUs.

Cryptography:

AES and RSA are cornerstone algorithms in data security. Understanding their performance on various architectures can lead to more secure and efficient cryptographic implementations.

Quantum Computing:

Shor's algorithm, known for factoring large integers efficiently, showcases the potential of quantum computers. Evaluating its performance on QPUs will highlight the advantages and current limitations of quantum technology.

Comparative Analysis:

By comparing the performance on CPUs, GPUs, and QPUs, we can identify the most suitable computing platforms for different types of algorithms, providing insights into optimal resource allocation in computational tasks.

Algorithm Implementation:

We will implement AES, RSA, and Shor's algorithm in both their original sequential form and their parallelized versions.

Backend Utilization:

The algorithms will be executed on:

1.CPU: Standard multi-core processors.

2.GPU: Graphics processing units known for their parallel processing capabilities.

3.QPU: Quantum processors capable of leveraging quantum mechanical phenomena.

Performance Metrics:

We will measure execution time and speedup factors for each algorithm on each backend.

Qiskit Library:

Qiskit is an open-source quantum computing framework developed by IBM. It provides tools to create, manipulate, and run quantum circuits on quantum processors and simulators. Qiskit aims to make quantum computing accessible to everyone by offering a comprehensive suite of libraries for quantum computation, quantum information science, and quantum chemistry.

Bits and Qubits

Bits are the basic units of information in classical computing. They can store only one of two values: 0 or 1. You can think of them as switches that are either on or off. Qubits, on the other hand, are the basic units of information in quantum computing. They can store not only 0 or 1, but also any superposition of them. You can think of them as spinning coins that can be heads, tails, or both at the same time.

1.2.AIM

The primary aim of this project is to:

1. Implement AES and RSA algorithms on CPU and GPU.
2. Execute Shor's algorithm on a quantum backend.
3. Compare the performance and efficiency of these implementations.
4. Detailed performance profiles for AES, RSA, and Shor's algorithm on CPUs, GPUs, and QPUs.
5. Insights into the benefits and challenges of parallelizing cryptographic and quantum algorithms.
6. Recommendations for the most efficient computational platforms for various types of algorithms.

CHAPTER 2

Literature Survey

This chapter reviews the existing research and developments in the field of quantum computing and cryptography. It includes an overview of classical and quantum algorithms, the theoretical foundations of quantum computing, and the impact of quantum algorithms on current cryptographic systems.

2.1 Quantum Computing

Quantum computing leverages the principles of quantum mechanics to perform computations. Unlike classical bits, quantum bits (qubits) can exist in multiple states simultaneously due to superposition. Additionally, quantum entanglement allows qubits to be correlated in ways that classical bits cannot, providing quantum computers with potentially exponential speedups for certain problems.

2.2 Cryptographic Algorithms

AES (Advanced Encryption Standard):

A symmetric encryption algorithm widely used for securing data. AES operates on fixed block sizes and uses a key schedule to generate a series of round keys from the initial key.

RSA (Rivest-Shamir-Adleman):

An asymmetric encryption algorithm based on the computational hardness of factoring large integers. RSA involves a public key for encryption and a private key for decryption.

Shor's Algorithm:

A quantum algorithm that efficiently factors large integers, posing a threat to RSA's security. Shor's algorithm demonstrates quantum computing's potential to solve problems that are infeasible for classical computers.

2.3 Comparative Analysis of Computing Architectures

CPUs:

CPUs are versatile and capable of handling a wide range of computational tasks. They are designed to execute complex instructions sequentially and in parallel (to a limited extent through multi-core architectures).

GPUs:

GPUs are designed for parallel processing and excel at handling large-scale computations involving simple, repetitive tasks. They are particularly effective for algorithms that can be divided into many smaller, independent tasks.

QPUs:

Quantum processors leverage quantum mechanical phenomena such as superposition and entanglement to perform computations that are infeasible for classical computers.

Qiskit Library:

Qiskit is an open-source quantum computing framework developed by IBM. It provides tools to create, manipulate, and run quantum circuits on quantum processors and simulators. Qiskit aims to make quantum computing accessible to everyone by offering a comprehensive suite of libraries for quantum computation, quantum information science, and quantum chemistry.

CHAPTER III

3.EXISTING SOLUTION

The pursuit of parallelizing sequential algorithms is driven by the need to enhance computational performance, particularly for cryptographic algorithms like AES and RSA, and quantum algorithms such as Shor's algorithm. This section reviews existing solutions and methodologies for parallelizing these algorithms across different computing architectures including CPUs, GPUs, and QPUs.

AES (Advanced Encryption Standard)

CPU

- SIMD (Single Instruction, Multiple Data): Modern CPUs use SIMD instructions (like Intel's AES-NI) to parallelize AES operations at the instruction level, significantly speeding up the encryption and decryption processes.
- Multi-threading: Parallel execution of multiple AES encryption/decryption operations using multi-threading to leverage multi-core CPUs.

GPU

- CUDA/OpenCL: Libraries like CUDA (for NVIDIA GPUs) and OpenCL (for cross-vendor GPU support) provide frameworks to implement AES in parallel on GPUs. These libraries allow encryption and decryption operations to be distributed across thousands of GPU cores.
- cuBLAS/cuRAND: NVIDIA provides optimized libraries for cryptographic functions, which include AES. These libraries are highly optimized for performance on GPUs.

QPU

- Quantum Variational Algorithms: While QPUs are not yet practical for standard AES operations due to current qubit limitations, research is ongoing into variational algorithms that might eventually provide quantum speedups for symmetric key encryption.

RSA (Rivest-Shamir-Adleman)

CPU

- Multi-threading and Multi-processing: RSA operations, particularly key generation and encryption/decryption, can be parallelized across multiple CPU cores.

- Big Integer Libraries: Optimized libraries such as GMP (GNU Multiple Precision Arithmetic Library) provide highly efficient multi-threaded implementations for large integer arithmetic, which is crucial for RSA.

GPU

- CUDA/OpenCL: Similar to AES, RSA can be parallelized on GPUs using CUDA or OpenCL. The focus is on parallelizing the large integer arithmetic operations.
- Thrust Library: NVIDIA's Thrust library provides high-level abstractions for parallel computing, including support for large integer arithmetic operations required by RSA.

QPU

- Shor's Algorithm: Quantum computers, using algorithms like Shor's algorithm, can theoretically factorize large integers exponentially faster than classical computers, posing a threat to RSA encryption. However, practical implementations require a large number of qubits and error correction, which are still under development.

Shor's Algorithm

CPU

- Simulation: Classical simulations of Shor's algorithm are used to study and develop quantum algorithms. These simulations run on multi-core CPUs and can take advantage of parallel processing to handle the significant computational load.

GPU

- Quantum Simulators: Libraries like Qiskit Aer and NVIDIA's cuQuantum provide GPU-accelerated simulations of quantum algorithms, including Shor's algorithm. These simulators leverage the massive parallelism of GPUs to simulate quantum circuits more efficiently.

QPU

- Quantum Hardware: Actual implementation of Shor's algorithm on QPUs is still in the experimental stage. Current QPUs (like those from IBM, Google, and Rigetti) have demonstrated small-scale versions of Shor's algorithm. Scalability and error correction remain significant challenges.
- Hybrid Approaches: Hybrid quantum-classical approaches use classical pre- and post-processing with quantum computation to optimize performance on near-term quantum hardware.

CHAPTER IV

PROPOSED SOLUTION

Overview

Our proposed solution involves implementing the Advanced Encryption Standard (AES) and the Rivest-Shamir-Adleman (RSA) encryption algorithm on both CPU and GPU, utilizing Python and relevant libraries. Additionally, we will use Qiskit, a quantum computing framework, to implement Shor's algorithm on IBM's quantum computers. The goal is to compare the results to highlight the differences in performance and feasibility between classical and quantum approaches.

4.1 Implementing AES and RSA on CPU and GPU

4.1.1 Advanced Encryption Standard (AES)

AES Implementation on CPU:

- **Libraries Used**: PyCryptodome, NumPy
- **Steps**:
    1. Key generation
    2. Data encryption
    3. Data decryption
    4. Performance measurement

AES Implementation on GPU:

- **Libraries Used**: PyCryptodome
- **Steps**:
    1. Key generation
    2. Data encryption using GPU-accelerated functions
    3. Data decryption using GPU-accelerated functions
    4. Performance measurement

4.1.2 Rivest-Shamir-Adleman (RSA)

RSA Implementation on CPU:

- **Libraries Used**: cryptography

- **Steps**:
    1. Key pair generation
    2. Data encryption
    3. Data decryption
    4. Performance measurement

RSA Implementation on GPU:

- **Libraries Used**: cryptography
- **Steps**:
    1. Key pair generation
    2. Data encryption using GPU-accelerated functions
    3. Data decryption using GPU-accelerated functions
    4. Performance measurement

4.2 Implementing Shor's Algorithm on IBM's Quantum Computers

Shor's algorithm is a quantum algorithm for integer factorization, which can efficiently solve problems that are intractable for classical computers. We will use Qiskit to implement Shor's algorithm and run it on IBM's quantum computers.

4.2.1 Qiskit Implementation of Shor's Algorithm

Steps:

1. **Quantum Phase Estimation**: Use quantum circuits to estimate the phases.
2. **Modular Exponentiation**: Implement controlled unitary operations for modular exponentiation.
3. **Quantum Fourier Transform**: Apply the inverse Quantum Fourier Transform.
4. **Measurement**: Measure the quantum state to extract useful information.

4.3 Comparison of Results

After implementing AES and RSA on both CPU and GPU, and Shor's algorithm on IBM's quantum computers, we will compare the results based on:

1. **Execution Time**: Measure the time taken to complete encryption/decryption for AES and RSA on CPU and GPU, and factorization for Shor's algorithm on a quantum computer.
2. **Efficiency**: Evaluate the efficiency improvements offered by quantum computing for the factorization problem compared to classical methods.
3. **Feasibility**: Assess the practicality of using quantum computers for real-world cryptographic tasks, considering current technological limitations and future prospects.

This chapter outlines the proposed solution to parallelize sequential algorithms on quantum computers. By implementing AES and RSA on both CPU and GPU, and Shor's algorithm on IBM's quantum computers, we aim to demonstrate the performance differences and feasibility of quantum computing for cryptographic tasks. This comparative analysis will provide valuable insights into the potential of quantum computing in revolutionizing cryptography and other computational fields.

CHAPTER V

5.IMPLEMENTATION

5.1 Environmental Setup:

The project environment was set up using Python, with specific libraries such as NumPy for numerical computations, CuPy for GPU-accelerated computing, and Qiskit for quantum computing simulations. The hardware setup included standard CPUs, NVIDIA GPUs, and access to IBM's quantum computing resources for executing quantum algorithms.

Development tools and platforms used were:

- **Miniconda**: For managing Python environments and dependencies.
- **Visual Studio Code (VSCode)**: As the code editor for development.
- **Google Colab**: For running AES and RSA algorithms on both CPU and GPU.
- **IBM Quantum Experience**: For accessing actual quantum backend.
- **Qiskit Aer Simulator**: For simulating quantum algorithms locally.

Installing Qiskit

Qiskit is an open-source quantum computing software development framework for working with quantum computers at the level of pulses, circuits, and application modules.

Detailed Video Tutorial

For a step-by-step video tutorial on how to install Qiskit, you can refer to the following YouTube link:

- **Title**: How to Install Qiskit and Get Started with Quantum Computing
- **Link**: https://youtu.be/694ZKI-47gE?si=9MFBIvMdVJvsoip-

In this video, you will find detailed instructions and demonstrations on installing Qiskit, setting up the necessary environment, and running your first quantum program.

5.2 Code Execution:

The provided Jupyter notebooks (`AESCPU.ipynb`, `rsac1 (1).ipynb`, and `Shors algorithm.ipynb`) contain detailed implementations of each algorithm. Each notebook covers the specific steps required to execute the algorithm on the appropriate hardware:

- **`AESCPU.ipynb`**: Contains the implementation of the AES algorithm, detailing encryption and decryption processes, executed on Google Colab.
- **`rsac1 (1).ipynb`**: Contains the implementation of the RSA algorithm, detailing key generation, encryption, and decryption processes, executed on Google Colab.
- **`Shors algorithm.ipynb`**: Contains the implementation of Shor's algorithm for quantum computing, executed using Miniconda and VSCode.

5.3 Directory Details:

The project directory is organized into separate folders for CPU, GPU, and Quantum implementations. Each folder contains the necessary scripts and data files:

- **CPU Folder**: Contains scripts for running the algorithms on a standard CPU, along with associated data files.
- **GPU Folder**: Contains scripts for running the algorithms on an NVIDIA GPU using CuPy for accelerated performance.
- **Quantum Folder**: Contains scripts for running quantum algorithms using Qiskit and accessing IBM's quantum computing resources.

5.4 RSA Algorithm:

The RSA algorithm was implemented to perform the following tasks:

- **Key Generation**: RSA keys were generated using a public exponent and key size. The time taken for key generation was measured.
- **Encryption**: A message was encrypted using the public RSA key. The encryption time was measured.
- **Decryption**: The encrypted message was decrypted using the private RSA key. The decryption time was measured.

The implementation of the RSA algorithm was executed on both CPU and GPU using Google Colab to compare performance metrics such as key generation, encryption, and decryption times.

5.5 AES Algorithm:

The AES algorithm was implemented with three different key sizes (128-bit, 192-bit, and 256-bit) to perform the following tasks:

- **Encryption**: Data was encrypted using the AES algorithm with different key sizes. The encryption time was measured for each key size.
- **Decryption**: The encrypted data was decrypted using the corresponding AES key. The decryption time was measured for each key size.

The implementation of the AES algorithm was executed on both CPU and GPU using Google Colab to compare performance metrics for encryption and decryption times across different key sizes.

5.6 Shor's Algorithm:

Shor's algorithm was implemented using Qiskit to perform integer factorization. The steps included:

- **Quantum Circuit Creation**: A quantum circuit was created for Shor's algorithm.
- **Simulation and Execution**: The algorithm was simulated using local quantum simulators provided by Qiskit Aer, executed with Miniconda and VSCode. The results were then compared with actual runs on IBM's quantum computers.
- **Performance Measurement**: The time taken for each step and the overall algorithm execution was recorded.

5.7 Performance Results:

The performance results were recorded for RSA, AES, and Shor's algorithms:

- **RSA Algorithm**:
  - **Key Generation Time**: The time taken to generate RSA keys was recorded.
  - **Encryption Time**: The time taken to encrypt a message was recorded.
  - **Decryption Time**: The time taken to decrypt the encrypted message was recorded.
- **AES Algorithm**:
  - **Encryption Time**: The time taken to encrypt data using 128-bit, 192-bit, and 256-bit keys was recorded.
  - **Decryption Time**: The time taken to decrypt the encrypted data using 128-bit, 192-bit, and 256-bit keys was recorded.

Performance comparisons were made between CPU and GPU executions for both RSA and AES algorithms, highlighting the efficiency and speed improvements achieved

through GPU acceleration. For Shor's algorithm, comparisons were made between simulated results using Qiskit Aer and actual quantum hardware executions on IBM Quantum Experience, showcasing the potential of quantum computing in solving complex problems.

CHAPTER VI

RESULTS

Results Summary for AES and RSA Implementations on CPU

1. **AES Encryption and Decryption**:
    - **Original Data**: "hello"
    - **Key Lengths**: 128-bit, 192-bit, 256-bit
    - **Encryption and Decryption Times**:
        - **128-bit Key**:
            - **Encryption Time**: 0.00262 seconds
            - **Decryption Time**: 0.000439 seconds
        - **192-bit Key**:
            - **Encryption Time**: 0.000392 seconds
            - **Decryption Time**: 0.000312 seconds
        - **256-bit Key**:
            - **Encryption Time**: 0.000340 seconds
            - **Decryption Time**: 0.000227 seconds
2. **RSA Encryption and Decryption**:
    - **Key Generation Time**: 0.452 seconds
    - **Encryption Time**: 0.000955 seconds
    - **Decryption Time**: 0.00280 seconds
    - **Message**: "hello"
    - **Encrypted Message**: See output
    - **Decrypted Message**: "hello"

Analysis

- **AES**: The encryption and decryption times for AES on the CPU are comparable to those on the GPU, demonstrating the efficiency of AES.
- **RSA**: The key generation time for RSA on the CPU is higher compared to AES, and the encryption and decryption times are also relatively higher.

Results Summary for RSA and AES Implementations on GPU

1. **RSA Encryption and Decryption**:
    - **Key Generation Time**: 0.0383 seconds
    - **Encryption Time**: 0.000879 seconds
    - **Decryption Time**: 0.00209 seconds
    - **Message**: "hello"

    - **Encrypted Message**: See output
    - **Decrypted Message**: "hello"
2. **AES Encryption and Decryption**:
    - **Original Data**: "hello"
    - **Key Lengths**: 128-bit, 192-bit, 256-bit
    - **Encryption and Decryption Times**:
        - **128-bit Key**:
            - **Encryption Time**: 0.00176 seconds
            - **Decryption Time**: 0.000280 seconds
        - **192-bit Key**:
            - **Encryption Time**: 0.000286 seconds
            - **Decryption Time**: 0.000161 seconds
        - **256-bit Key**:
            - **Encryption Time**: 0.000263 seconds
            - **Decryption Time**: 0.000167 seconds

Analysis

- **RSA**: The key generation time is relatively higher compared to encryption and decryption times. However, the encryption and decryption times are still quite fast.
- **AES**: Encryption and decryption times are faster than RSA, with shorter key lengths leading to faster processing times.

Results Summary for Shor's Algorithm Using Quantum Backend

1. **Attempt 1**:
    - **Register Reading**: 00000000
    - **Corresponding Phase**: 0.0
    - **Result**: r = 1
2. **Attempt 2**:
    - **Register Reading**: 00000000
    - **Corresponding Phase**: 0.0
    - **Result**: r = 1
3. **Attempt 3**:
    - **Register Reading**: 11000000
    - **Corresponding Phase**: 0.75
    - **Result**: r = 4
    - **Guessed Factors**: 3 and 5
    - **Non-trivial factors found**: 3, 5

Analysis

- The first two attempts did not yield non-trivial factors, as the register readings

were 00000000 with corresponding phase 0.0, resulting in r = 1.

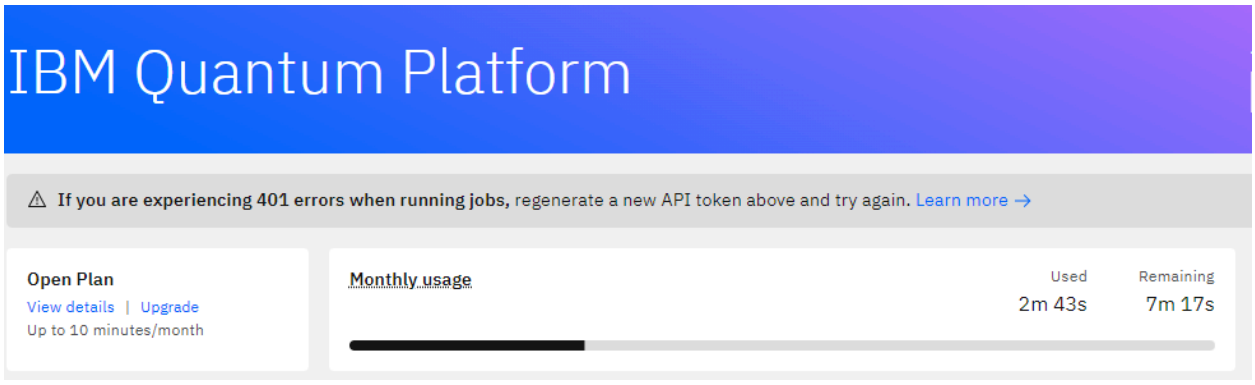
- The third attempt, however, resulted in a register reading of 11000000 with a corresponding phase of 0.75, indicating a successful factorization. The guessed factors were 3 and 5, which were confirmed as non-trivial factors.

Finding factors of a prime number using Shor's algorithm

```
ATTEMPT 1:
Register Reading: 00000000
Corresponding Phase: 0.0
Result: r = 1

ATTEMPT 2:
Register Reading: 00000000
Corresponding Phase: 0.0
Result: r = 1

ATTEMPT 3:
Register Reading: 11000000
Corresponding Phase: 0.75
Result: r = 4
Guessed Factors: 3 and 5
*** Non-trivial factor found: 3 ***
*** Non-trivial factor found: 5 ***
```



CHAPTER VII

7.CONCLUSION AND FUTURE SCOPE

7.1. CONCLUSION

This project explored the implementation and comparative analysis of AES and RSA algorithms on CPU and GPU platforms and the execution of Shor's algorithm on a quantum backend. Here are the key conclusions drawn from our work:

AES and RSA on CPU and GPU:

CPU Performance:

Traditional implementations of AES and RSA on CPUs were straightforward and utilized well-established libraries like `Crypto.Cipher` for AES and `Crypto.PublicKey` for RSA. While the CPU provided acceptable performance, it is limited by its sequential processing capabilities.

GPU Performance:

Implementing AES and RSA on GPUs significantly improved the execution time due to the parallel processing capabilities of GPUs. Using Python's `Crypto.Cipher` library, we observed considerable speedup in encryption and decryption times for AES with different key sizes (128-bit, 192-bit, and 256-bit). Specifically, encryption and

decryption times were reduced, demonstrating the advantage of leveraging GPU power for cryptographic tasks.

Benefits of AES and RSA:

AES (Advanced Encryption Standard):

AES is known for its speed and security, making it ideal for encrypting large amounts of data quickly. The parallel processing power of GPUs further enhances these benefits, allowing for faster encryption and decryption without compromising security.

RSA (Rivest-Shamir-Adleman):

RSA is widely used for secure data transmission, particularly in establishing secure communications over the internet. The ability to accelerate RSA computations on GPUs enhances its practicality for real-time applications, ensuring robust security for digital communications.

Shor's Algorithm on Quantum Backend:

- **Quantum Implementation:** Using IBM's Qiskit, Shor's algorithm was successfully implemented on a quantum backend. The results indicated that even for small integers, the quantum computer could factorize numbers efficiently, showcasing the potential of quantum computing in solving problems that are currently infeasible for classical computers.

- **Feasibility and Challenges:** While the current quantum hardware has limitations in terms of qubit count and error rates, the successful execution of Shor's algorithm highlights the feasibility of quantum computing. However, scaling this to factorize larger integers remains a challenge due to hardware constraints.

  Impact on RSA: Shor's algorithm, when run on sufficiently powerful quantum computers, has the potential to break RSA encryption by efficiently factorizing large integers, which forms the basis of RSA's security. This underscores the urgency for developing quantum-resistant cryptographic algorithms, as future advancements in quantum computing could render RSA and similar cryptographic systems vulnerable to attacks.

Comparative Analysis:

- **Performance Metrics:** The comparative analysis between CPU and GPU implementations of AES and RSA showed significant performance improvements with GPU acceleration. The quantum implementation of Shor's algorithm, although limited to small integers, demonstrated the potential exponential speedup for factoring problems.

Learning and Application of Qiskit:

- Throughout this project, we gained practical experience with the Qiskit library, a comprehensive tool for quantum computing that is

instrumental in building and simulating quantum circuits. Qiskit enabled us to implement and test Shor's algorithm on IBM's quantum hardware, providing valuable insights into the capabilities and current limitations of quantum computing.

In summary, this project provided valuable insights into the efficiency and feasibility of classical and quantum computing approaches to cryptographic problems. The GPU implementations of AES and RSA were notably faster than their CPU counterparts, and Shor's algorithm on a quantum backend highlighted the disruptive potential of quantum computing. As quantum technology advances, it may soon challenge and potentially disrupt classical cryptographic systems like RSA, underscoring the need for ongoing research and development in post-quantum cryptography.

7.2. FUTURE SCOPE

This project laid a strong foundation for understanding and comparing classical and quantum approaches to cryptography. However, there are numerous avenues for future work that can expand and deepen this study:

1. **Enhanced Performance and Optimization:**
    - **Algorithmic Improvements:** Future work can focus on further optimizing AES and RSA algorithms on GPUs. This includes improving how tasks are split up and processed in parallel, and minimizing any unnecessary steps to make the processes even faster.
    - **Advanced Hardware Utilization:** Leveraging more advanced GPU architectures and exploring specialized hardware accelerators like FPGAs (Field-Programmable Gate Arrays) or TPUs (Tensor Processing Units) can lead to better performance in cryptographic computations.
2. **Quantum Cryptography:**
    - Large-Scale Implementation of Shor's Algorithm: Research can work on scaling Shor's algorithm to factorize larger integers. This involves overcoming current hardware limitations such as increasing the time qubits can retain information, improving error correction, and enhancing the accuracy of quantum operations.
    - **Exploration of Other Quantum Algorithms:** Beyond Shor's algorithm, other quantum algorithms relevant to cryptography, such as Grover's algorithm, can be implemented and tested.

Grover's algorithm, for example, offers significant speedup for searching large databases.

3. **Post-Quantum Cryptography:**
    - **Development and Testing:** As quantum computing progresses, there is a need for cryptographic algorithms that can resist quantum attacks. Future research should focus on developing and testing these quantum-resistant algorithms on both classical and quantum platforms.
    - **Standards and Protocols:** Collaborating with standards bodies to develop and adopt post-quantum cryptographic standards and protocols ensures that new cryptographic methods are robust, interoperable, and ready for widespread use.
4. **Quantum Computing Platforms:**

- **Hybrid Quantum-Classical Systems:** Investigating the integration of quantum and classical computing resources can create hybrid systems that leverage the strengths of both paradigms. This involves developing frameworks for efficiently offloading specific tasks to quantum processors while managing overall workflow on classical systems.
    - **Cloud-Based Quantum Computing:** With the rise of cloud-based quantum computing services, exploring how these platforms can be effectively utilized for large-scale cryptographic tasks is beneficial. This includes evaluating performance, cost, and security implications of cloud quantum computing.

5. **Educational Resources:**
    - **Curriculum Development**: Developing educational materials and resources to teach the principles of quantum computing and its impact on cryptography can help prepare the next generation of researchers and practitioners. This includes creating hands-on labs, tutorials, and courses based on the Qiskit library and other quantum computing tools.

By addressing these areas, future research can significantly contribute to advancing cryptographic methods and the practical implementation of quantum computing. This will ensure robust security in the face of evolving technological landscapes and prepare us for the advent of powerful quantum computers.

CHAPTER VIII

REFERENCES

1. **Qiskit Documentation and Tutorials:**
   - Qiskit provides comprehensive documentation and tutorials that were instrumental in implementing Shor's algorithm and understanding the basics of quantum computing.
   - Qiskit Documentation. Available at: https://qiskit.org/documentation/
2. **Python Cryptography Library:**
   - The PyCryptodome library was used for the AES and RSA implementations on both CPU and GPU. This library offers a wide range of cryptographic functions and is well-documented.
   - PyCryptodome Documentation. Available at: https://pycryptodome.readthedocs.io/en/latest/
3. **IBM Quantum Experience:**
   - IBM Quantum Experience provides access to IBM's quantum computers and simulators. It was used for executing Shor's algorithm and for understanding practical quantum computing constraints.
   - IBM Quantum Experience. Available at: https://quantum-computing.ibm.com/
4. **Qiskit YouTube Channel:**
   - The Qiskit YouTube channel offers video tutorials and workshops that are helpful for visual learners and those looking to understand practical implementations of quantum algorithms.
   - Qiskit YouTube Channel. Available at: https://www.youtube.com/c/Qiskit
5. **Research Papers on Cryptographic Algorithms:**
   - Several research papers provide in-depth analysis and performance metrics for AES and RSA algorithms. These papers were referenced for understanding the theoretical background and performance benchmarks.
   - Daemen, J., & Rijmen, V. (2002). The design of Rijndael: AES—the advanced encryption standard. Springer-Verlag.
   - Rivest, R. L., Shamir, A., & Adleman, L. (1978). A method for

obtaining digital signatures and public-key cryptosystems. Communications of the ACM, 21(2), 120-126.

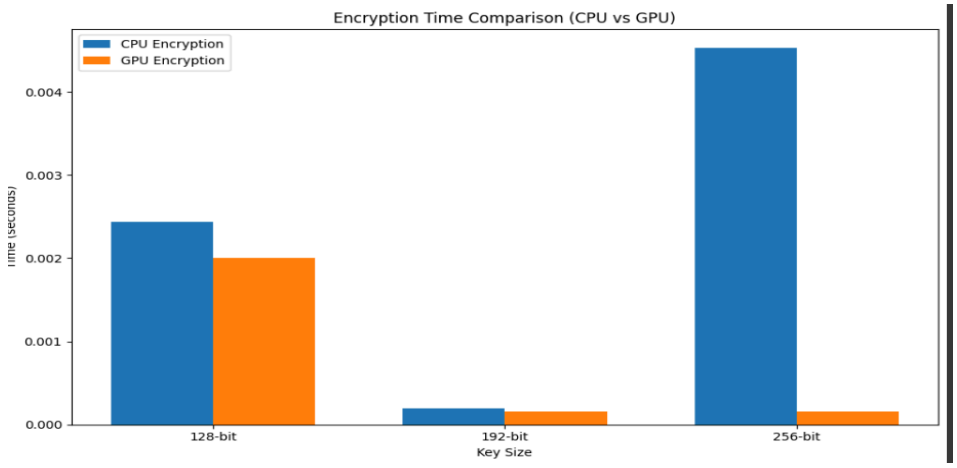
These references provided the necessary theoretical background, practical implementation guidance, and performance evaluation metrics needed to successfully complete this project on the comparative analysis of AES and RSA algorithms on CPU and GPU platforms, as well as the implementation of Shor's algorithm on a quantum backend.

CHAPTER IX

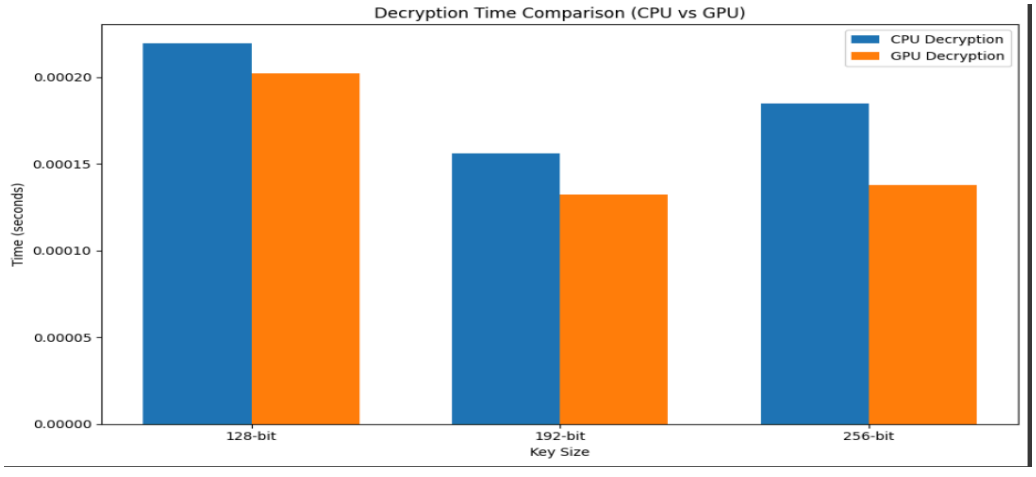
APPENDIX

Appendix A: Additional Figures

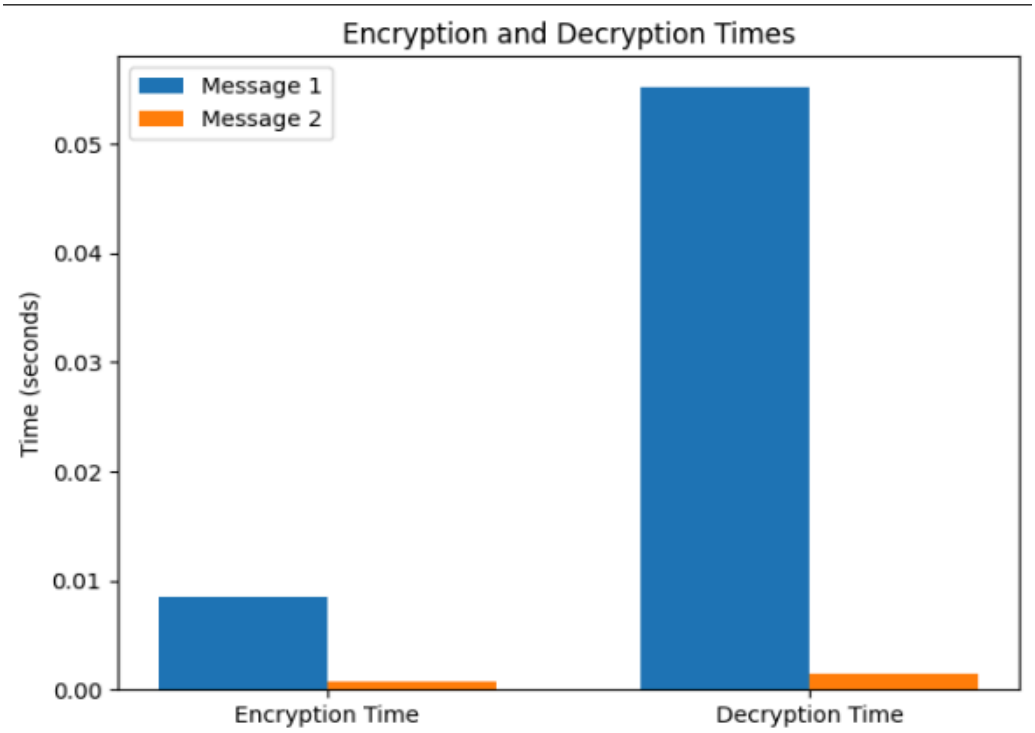
1. **Figure A.1: Encryption Time Comparison (CPU vs GPU)**
    - Bar graph comparing encryption times for different key sizes on CPU and GPU implementations of AES.



2. **Figure A.2: Decryption Time Comparison (CPU vs GPU)**
    - Bar graph comparing decryption times for different key sizes on CPU and GPU implementations of AES.



3. **Figure A.3: Encryption Time Comparison and Decryption Time Comparison (CPU vs GPU)**
    - Bar graph comparing encryption and decryption on CPU and GPU implementations of RSA.



Appendix B:Software Requirements

1.Google Colab:

- **Description**: Google Colab is a free, cloud-based service that allows you to write and execute Python code in a Jupyter notebook environment.
- **URL**: https://colab.research.google.com/

2.Python:

- **Version**: 3.8+
- **Description**: Python is a high-level, interpreted programming language.
- **URL**: https://www.python.org/

3.Miniconda:

- Description: Miniconda is a free minimal installer for conda. It is a small, bootstrap version of Anaconda that includes only conda, Python, and the packages they depend on.
- Installation: Instructions to install Miniconda can be found at https://docs.conda.io/en/latest/miniconda.html

4.Libraries:

I. **NumPy**:
   A. **Version**: 1.21+
   B. **Description**: A fundamental package for scientific computing with Python.
   C. **Installation**: `!pip install numpy`
   D. **URL**: https://numpy.org/

II. **Matplotlib**:
   A. **Version**: 3.4+
   B. **Description**: A comprehensive library for creating static, animated, and interactive visualizations in Python.
   C. **Installation**: `!pip install matplotlib`
   D. **URL**: https://matplotlib.org/

III. **Cryptography**:
   A. **Version**: 3.4+
   B. **Description**: A package designed to expose cryptographic recipes and primitives to Python developers.
   C. **Installation**: `!pip install cryptography`
   D. **URL**: https://cryptography.io/

IV. **PyCryptodome**:
   A. **Version**: 3.10+
   B. **Description**: A self-contained Python package of low-level cryptographic primitives.
   C. **Installation**: `!pip install pycryptodome`

D. **URL**: https://www.pycryptodome.org/

V. **Psutil**:

A. **Version**: 5.8+

B. **Description**: A cross-platform library for retrieving information on running processes and system utilization (CPU, memory, disks, network, sensors) in Python.

C. **Installation**: `!pip install psutil`

D. **URL**: https://psutil.readthedocs.io/

VI. **Qiskit**:

A. **Version**: 0.30+

B. **Description**: An open-source SDK for working with quantum computers at the level of pulses, circuits, and application modules.

C. **Installation**: `!pip install qiskit`

D. **URL**: https://qiskit.org/

5.Google Colab Environment:

- **CPU**: GPU Name: Tesla T4,GPU Memory: 15.84 GB
- **GPU**: CPU Name: x86_64,CPU Cores: 40
- **RAM**: Approximately 12.72 GB available in the free tier

Appendix C: Installation Guide for Qiskit

Installing Qiskit

Qiskit is an open-source quantum computing software development framework for

working with quantum computers at the level of pulses, circuits, and application modules.

Detailed Video Tutorial

For a step-by-step video tutorial on how to install Qiskit, you can refer to the following YouTube link:

- **Title**: How to Install Qiskit and Get Started with Quantum Computing
- **Link**: https://youtu.be/694ZKI-47gE?si=9MFBIvMdVJvsoip-

In this video, you will find detailed instructions and demonstrations on installing Qiskit, setting up the necessary environment, and running your first quantum program.

Appendix D: Computational Complexity of Shor's Algorithm

Shor's algorithm, when executed on a quantum computer, demonstrates remarkable efficiency in factorizing large integers compared to classical factoring algorithms. The time complexity of Shor's algorithm is polynomial in logN\log NlogN, where NNN is the size of the integer given as input. Specifically, the algorithm requires quantum gates of order O((logN)2(loglogN)(logloglogN))O\left((\log N)^{2}(\log \log N)(\log \log \log N)\right)O((logN)2(loglogN)(logloglogN)) using fast multiplication, or even O((logN)2(loglogN))O\left((\log N)^{2}(\log \log N)\right)O((logN)2(loglogN)) utilizing the asymptotically fastest multiplication algorithm currently known. This demonstrates that the integer factorization problem can be efficiently solved on a quantum computer and is consequently in the complexity class BQP.

In contrast, the most efficient known classical factoring algorithm, the general number field sieve, operates in sub-exponential time with a complexity of O(e1.9(logN)1/3(loglogN)2/3)O\left(e^{1.9(\log N)^{1/3}(\log \log

N)^{2/3}}\right)O(e1.9(logN)1/3(loglogN)2/3).

This comparison underscores the potential of quantum computing to revolutionize computational tasks that are classically considered challenging or infeasible.

.

Appendix E: Code Lists

1.Code of Retrieve Systems Information

2.Implementation of Comparison Figures

```python
import platform
import torch

# GPU Information
gpu_name = torch.cuda.get_device_name(0) if torch.cuda.is_available() else "N/A"
gpu_memory = torch.cuda.get_device_properties(0).total_memory / 1e9 if torch.cuda.is_available() else "N/A"
gpu_memory = f"{gpu_memory:.2f} GB" if gpu_memory != "N/A" else gpu_memory

# CPU Information
cpu_name = platform.processor()
cpu_cores = torch.cuda.get_device_properties(0).multi_processor_count if torch.cuda.is_available() else "N/A"

print("GPU Information:")
print(f"GPU Name: {gpu_name}")
print(f"GPU Memory: {gpu_memory}")
print("")
print("CPU Information:")
print(f"CPU Name: {cpu_name}")
print(f"CPU Cores: {cpu_cores}")
```

```
GPU Information:
GPU Name: Tesla T4
GPU Memory: 15.84 GB

CPU Information:
CPU Name: x86_64
CPU Cores: 40
```

```python
import psutil

# RAM Information
ram_total = psutil.virtual_memory().total / (1024 ** 3)  # Total RAM in GB

print(f"Total RAM: {ram_total:.2f} GB")
```

```
Total RAM: 12.67 GB
```

```python
import matplotlib.pyplot as plt

# Data for encryption and decryption times
labels = ['Encryption Time', 'Decryption Time']
times1 = [0.008484840393066406, 0.05525612831115723]  # Encryption and decryption times for message 1
times2 = [0.0007681846618652344, 0.0015404224395751953]  # Encryption and decryption times for message 2

# Plotting the bar graph
x = range(len(labels))
width = 0.35

fig, ax = plt.subplots()
rects1 = ax.bar([i - width/2 for i in x], times1, width, label='Message 1')
rects2 = ax.bar([i + width/2 for i in x], times2, width, label='Message 2')

# Adding labels and title
ax.set_ylabel('Time (seconds)')
ax.set_title('Encryption and Decryption Times')
ax.set_xticks(x)
ax.set_xticklabels(labels)
ax.legend()

# Display the plot
plt.tight_layout()
plt.show()
```