cryptography

MDPI

Article

FPGA-Based Acceleration of K-Nearest Neighbor Algorithm on Fully Homomorphic Encrypted Data

Sagarika Behera [1,*,†] and Jhansi Rani Prathuri [2,†]

[1] Department of Computer Science and Engineering, CMR Institute of Technology, Visvesveraya Technological University, Belagavi 590018, India
[2] Department of Computer and Data Sciences, School of Engineering and Computational Sciences, Merrimack College, North Andover, MA 01845, USA; jhansirani.p@gmail.com
* Correspondence: sagarika.b@cmrit.ac.in
† These authors contributed equally to this work.

Abstract: The suggested solution in this work makes use of the parallel processing capability of FPGA to enhance the efficiency of the K-Nearest Neighbor (KNN) algorithm on encrypted data. The suggested technique was assessed utilizing the breast cancer datasets and the findings indicate that the FPGA-based acceleration method provides significant performance improvements over software implementation. The Cheon–Kim–Kim–Song (CKKS) homomorphic encryption scheme is used for the computation of ciphertext. After extensive simulation in Python and implementation in FPGA, it was found that the proposed architecture brings down the computational time of KNN on ciphertext to a realistic value in the order of the KNN classification algorithm over plaintext. For the FPGA implementation, we used the Intel Agilex7 FPGA (AGFB014R24B2E2V) development board and validated the speed of computation, latency, throughput, and logic utilization. It was observed that the KNN on encrypted data has a computational time of 41.72 ms which is 80 times slower than the KNN on plaintext whose computational time is of 0.518 ms. The main computation time for CKKS FHE schemes is 41.72 ms. With our architecture, we were able to reduce the calculation time of the CKKS-based KNN to 0.85 ms by using 32 parallel encryption hardware and reaching 300 MHz speed.

Keywords: field-programmable gate array (FPGA); K-Nearest Neighbor (KNN) algorithm; Cheon–Kim–Kim–Song (CKKS) encryption scheme; fully homomorphic encryption (FHE) scheme; cloud computing

Citation: Behera, S.; Prathuri, J.R. FPGA-Based Acceleration of K-Nearest Neighbor Algorithm on Fully Homomorphic Encrypted Data. *Cryptography* **2024**, *8*, 8. https://doi.org/10.3390/cryptography8010008

Academic Editor: Josef Pieprzyk

Received: 27 December 2023
Revised: 20 February 2024
Accepted: 22 February 2024
Published: 27 February 2024

1. Introduction

In the fully homomorphic encryption (FHE), process operations can be performed on the ciphertext without performing the decryption. Therefore, this process does not require to share the private key with others. In contrast to other cryptographic techniques, all operations require the sharing of the secret key needed for the decoding process, which may lead to the leakage of private information. Despite this amazing feat of theoretical cryptography, the high computational and memory costs of FHE continue to be a significant problem to its broad acceptance. To make it practically acceptable and usable widely, many researchers have been working on it since 2009 when it was first proposed by C. Gentry [1] in his Ph.D. thesis. Since then, a significant number of researchers [2–10] started working in this field to enhance the homomorphic encryption operation. In homomorphic encryption, after each operation on the encrypted data, the noise will be added. This noise grows with each operation. For this, the bootstrapping concept is used to keep the noise at a certain level. The bootstrapping concept was first proposed by C. Gentry et al. [11]. Since bootstrapping adds more cost to the encryption and decryption processes, it becomes a bottleneck for the adoption of FHE at large. More researchers are working in this field to reduce the computing cost of FHE for wide adoption by users to store their data securely at third-party servers and for various operations to be performed on these encrypted data.

The rapid accumulation of sensitive data including medical and financial records in cloud environments, and the corresponding analyses of such information, have understandably elevated concerns around maintaining privacy and ensuring security for such important consumer data. By enabling arbitrary operations to be executed on encrypted data without requiring decryption, FHE provides an effective means of addressing this issue through its capability for flexible handling of ciphertext. However, FHE's actual applicability in real-world applications has been limited due to its high computational cost and massive memory needs. FHE can be applied for secure analysis and classification of data in machine learning (ML).

The most widely used classification algorithm in machine learning is KNN. This algorithm is a fundamental tool in data classification, clustering, and anomaly detection. Healthcare, finance, and cyber security are just a few of the industries in which it finds use. However, running KNN on encrypted data using FHE is an exceptionally demanding task due to the intricate nature of distance calculations and the large datasets involved. This research project seeks to address this challenge by exploring the acceleration of the KNN algorithm on fully homomorphic encrypted data using field-programmable gate arrays (FPGAs).

CKKS is applicable to perform computations on ciphertext in the context of machine learning (ML) algorithms. One key application is in training ML models on private data. Using CKKS, data owners can encrypt their data and deliver them to a cloud-based ML platform, where the model can be trained on the ciphertext without the cloud provider having access to the underlying data. This allows for sensitive data to be kept private while still allowing for the benefits of cloud-based ML.

Motivation

The KNN classification of encrypted data is significant for several compelling reasons:

- **Preserving Data Privacy:** By performing KNN classification on encrypted data, organizations can ensure the confidentiality of their data while still deriving valuable insights from them. This is rather significant in fields like healthcare, finance, and legal, where data privacy regulations are stringent.
- **Machine Learning on Encrypted Data:** As machine learning continues to drive innovation across industries, the capacity to perform these tasks on encrypted data without compromising privacy is crucial. Developing efficient methods to run ML algorithms like KNN on encrypted data can open new possibilities for secure data analysis.
- **Real-World Applications:** The KNN algorithm finds applications in numerous real-world scenarios, such as personalized medicine, fraud detection, and secure collaborative data analysis. Accelerating KNN on encrypted data opens up opportunities for secure and privacy-preserving decision-making in these domains.
- **Resource Efficiency:** FPGAs are renowned for their flexibility and efficiency in computing. By leveraging FPGA-based acceleration, this research aims to reduce the computational burden of FHE, making it a more practical choice for privacy-conscious organizations and applications.

To overcome the limitations of FHE, this paper presents an FPGA-based acceleration method for the KNN algorithm applied to fully homomorphically encrypted data in a cloud environment. Faster processing times and less energy usage are possible with FPGAs than with standard software implementations because they offer a means of implementing specialized hardware that may be tuned for certain datasets and use cases. The proposed method is evaluated using different datasets, and the findings indicate that the FPGA-based acceleration method provides significant performance improvements over software implementation, making it more practical for large-scale applications.

2. Relevant Work

This section describes the recent research works in the area of FHE and machine learning algorithms. This study intends to enhance the acceleration of the KNN algorithm

using an FPGA on encrypted data. The CKKS technique, which makes it possible to perform operations on vectors of complex numbers as well as real values, is employed in this case for encryption and decryption. The design and implementation of KNN machine learning algorithms in FPGA, as well as several homomorphic encryption techniques, have all been studied in the literature to support this work.

Mohsin, Mokhles A. et al. [12] designed a novel hardware architecture for the FPGA implementation of the KNN algorithm on mobile devices. They have claimed that they obtained a speed-up of 127x over the software implementation.

Almomany, Abedalmuhdi et al. [13] proposed an improved KNN algorithm based on class contribution and feature weighting (DCT-KNN) implemented in FPGA using OpenCL, which is a high-level parallel programming tool. According to the authors of this research, their design and FPGA implementation are 44 times faster than CPU-based computing techniques.

Marquez-Viloria, David et al. [14] implemented two versions of the KNN algorithm using FPGA. The normal KNN algorithm was applied successfully for inter-channel interference mitigation in a 3 × 16 Gbaud 16-QAM Nyquist WDM system. In the modified KNN algorithm, the main focus is to reduce the comparison of symbols by using the rule of the eight connected clusters used for image processing to find the closest neighbor. The modified KNN algorithm gives a reduction of 47.25% in computational time as opposed to the original KNN algorithm.

A novel hardware architecture for the BV FHE scheme is proposed in [15]. The paper also covers the simulation of FPGA-based design using the Questasim simulator and implementation in Intel FPGA using the Quartus tool. Similarly, the FPGA-based design architecture for the LWE scheme is presented in [16]. The authors claim that the time taken for various operations such as key generation, encryption, and decryption operation is much lower compared to software implementation. They found that the throughput of 32 butter-fly is 15× compared to a single butter-fly.

The design of an FPGA-based accelerator with bootstrappable FHE is given by Agrawal, Rashmi et al. in [17] for the first time. This design is not a memory-bound design compared to the previous design which consumed more resources and was taking more execution time. Their design gives better performance of 213× compared to CPU and 1.5× compared to GPU. They have applied it to train a logistic regression model over encrypted data.

As the popularity of ML increases in the cloud computing environment, privacy and security of the data become a great concern for cloud users. To overcome these problems, the researchers have given different methods for the application of homomorphic encryption in machine learning in the paper [18–23].

The hardware implementation and acceleration of FHE on encrypted data are thoroughly explained by Nikola Samardzic [24]. The first FHE programmable accelerator [25] whose performance is 17,000× compared to software performance can be used for deep learning in the cloud as claimed by the authors. More researchers are concentrating on hardware implementation and simulation of FHE to make it practically implementable in real-time scenarios [26–31].

2.1. Research Gap

Considering the current state of research and technology in the fields of FPGA acceleration, the KNN algorithm, and Fully Homomorphic Encryption (FHE), there is still much space for improvement, and the following are some possible research gaps in this field.

One of the biggest obstacles is that FHE operations are very computationally expensive, making it difficult to implement KNN on FHE data in realtime. Another challenge is that FHE data are much larger than plaintext data, which can make it difficult to store and process on FPGAs.

The following are a few areas where research is lacking:

- Developing more efficient FHE algorithms and implementations that are specifically optimized for FPGAs.
- Designing FPGA-based architectures that can efficiently process large amounts of FHE data.
- Creating innovative techniques for reducing the size of FHE data without sacrificing accuracy.
- Evaluating the performance and energy efficiency of FPGA-based KNN accelerators for FHE data.
- The integration of FHE in machine learning tasks is driven by the need for secure computation of sensitive data. Research should explore the security and privacy implications of FPGA-accelerated KNN on FHE-encrypted data, including potential vulnerabilities and mitigation strategies.
- FPGA devices have limited resources, including memory and processing elements. Research should investigate how to effectively allocate and utilize these resources to optimize the acceleration of KNN while maintaining FHE security.
- Establishing standards and best practices for FPGA-based acceleration in the context of FHE-protected machine learning can be beneficial for researchers and practitioners.
- Developing user-friendly programming models and tools for designing and deploying FPGA-accelerated KNN on FHE-encrypted data can be a significant research gap. Simplifying the development process can encourage wider adoption.

2.2. Objectives

This research explores the fusion of hardware acceleration through FPGAs with the transformative capabilities of CKKS FHE while referring to the KNN technique. The main motive is to develop a novel system that accelerates the KNN algorithm while preserving data privacy through FHE, unlocking a new paradigm for privacy-preserving machine learning.

Our work focuses on several key aspects. First, we design and optimize FPGA-based hardware implementations of the CKKS scheme, enhancing computational efficiency and enabling real-time or large-scale applications. Simultaneously, we integrate FHE into the KNN workflow, allowing for computations to be performed on encrypted data without revealing sensitive information, thus addressing the ever-growing need for data privacy.

We showcase the performance and scalability of our FPGA-accelerated, FHE-protected KNN system through thorough benchmarking and evaluation. We evaluated the logic resources, latency, execution time, and throughput of computation while comparing it against existing non-encrypted and encrypted methods. Furthermore, we explore practical applications in healthcare where privacy-preserving machine learning is paramount due to regulatory and security constraints.

Overall, the objective of this research is to create a practical and efficient solution that leverages FPGA technology and FHE to enable privacy-preserving KNN algorithm processing on sensitive data, with the potential for real-world applications in privacy-conscious domains.

2.3. Our Contribution

This paper aims to design a novel hardware architecture for applying the KNN classification algorithm to encrypted data. This design is implemented and simulated using FPGA. In this work, the CKKS homomorphic encryption scheme is used for the computation of encrypted data. The CKKS-KNN block receives the data and the complex data are converted into a real polynomial using a systolic array based on QR decomposition for matrix inversion. In this research work, the bitonic sorting method is used for sorting the distance in the KNN module. After extensive simulation in Python and implementation in FPGA, it is evident that, with the proposed architecture with 32 parallel channels and enhancing the clock performance from 196 MHz to 300 MHz, i.e., 1.5 times, the computational time has been brought down to 1/40 of CKKS-KNN (0.85 ms) on encrypted data to a realistic value in the order of the KNN classification algorithm over plaintext (0.518 ms).

Outline of this paper: The organization of this paper is given here. Section 1 briefly introduces the CKKS FHE scheme, KNN algorithms, and the motivation behind this research. Section 2 presents a brief overview of related work carried out by various researchers in this field. This section also presents the research gap and the objective of our research in this field with our contribution. Section 3 explains the theoretical context of CKKS, Ring Learning with Error (RLWE), Number Theoretic Transform (NTT) for polynomial multiplication, the KNN algorithm, and the application of the CKKS scheme in machine learning. Section 4 illustrates the proposed method and design architecture. The different algorithms such as KNN classifier, encoding, encryption, decryption, decoding, and evaluation function on encrypted data are given in Section 5. Software implementation of the proposed method using Python and hardware implementation using FPGA are given in Section 6. The logic resource utilization results such as computational time, latency, and throughput of KNN on the plaintext, CKKS-KNN with a single channel and 32 parallel channels, are discussed in Section 7. Section 8, concludes this paper with the observed result. The future work that can be carried out in this emerging field is given in Section 9.

3. Theoretical Background

3.1. Cheon–Kim–Kim–Song (CKKS) Scheme

CKKS is a homomorphic encryption (HE) algorithm that allows for operations to be performed on ciphertext, without the need to first decrypt the data. When sensitive data need to be processed without being made public to the parties conducting the computation, this can be helpful. CKKS is a type of HE that is designed to work with real-valued numbers, as opposed to the integers that are typically used in other HE schemes. For machine learning and other numerical computations, this makes it effective.

3.2. Application of CKKS Scheme in Machine Learning

Here are some applications of the CKKS Scheme in machine learning:

- **Privacy-Preserving Predictive Analytics:** Assume a financial institution wishes to develop a predictive model to evaluate its clients' creditworthiness but is averse to disclosing client information to outside parties. CKKS can be used to encrypt customer financial data, train the predictive model on the encrypted data, and then apply the model to encrypted queries from customers to determine their creditworthiness without revealing their sensitive financial details.
- **Collaborative Machine Learning:** In a scenario where multiple organizations want to jointly train a machine learning model on their combined datasets without sharing the data, CKKS can be employed. Each organization can encrypt its data and share the encrypted inputs with a central entity. The central entity can then perform model training on the encrypted data and provide the model without ever seeing the raw data.
- **Privacy-Preserving Health Data Analysis:** Hospitals and medical research institutions may collaborate on medical research without sharing individual patient health records. CKKS can be used to encrypt patient data, allowing for secure joint analysis of the data for medical research, clinical trials, or disease pattern analysis while preserving patient privacy.
- **Privacy-Preserving Recommendations:** Online platforms that provide personalized recommendations (e.g., e-commerce, streaming services) can use CKKS to protect user data. User behavior data can be encrypted, and recommendation models can be trained and applied to the encrypted data, ensuring that users' preferences and histories remain confidential.
- **Encrypted Machine Learning on IoT Devices:** IoT devices may generate sensitive data that need to be analyzed for various purposes, such as anomaly detection in industrial settings. CKKS can be employed to perform machine learning on encrypted IoT data locally on the devices or in a secure gateway, ensuring data privacy.

- **Secure Natural Language Processing (NLP):** In a scenario where a cloud-based NLP service wants to analyze text data from users while keeping the text content private, CKKS can be applied. Users can encrypt their text, send it to the cloud service, and receive encrypted NLP analysis results. The cloud service provider can process the encrypted text and return encrypted insights without seeing the plaintext.

These examples illustrate how CKKS can enable various machine learning applications while maintaining data privacy and security. CKKS's homomorphic encryption properties allow for computations to be performed on encrypted data, making it a valuable tool for privacy-preserving machine learning in various domains.

3.3. Overview of CKKS Method

The overall view of the CKKS algorithm is shown in Figure 1. Sender A has the original message M, which is a vector of values. This message is encoded and converted into plaintext P, which is a polynomial. This plaintext is encrypted using the public key encryption method and the resultant ciphertext C is stored in the cloud server. Any computation function "f" is applied to this ciphertext and the result will be a ciphertext C'. This resultant ciphertext C' is decrypted and it will give the output as plaintext P'. Finally, this plaintext P' is decoded to obtain the resultant message M'.

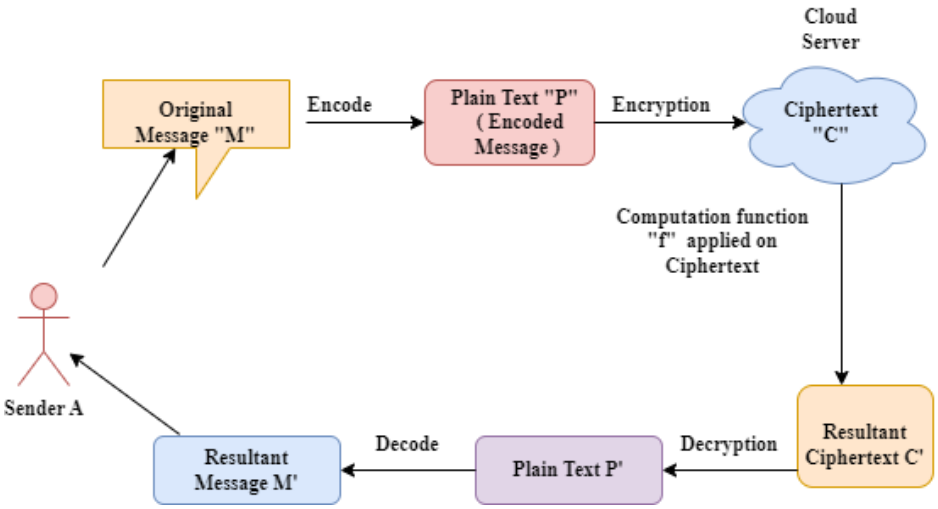

Figure 1. Block diagram of CKKS method.

The CKKS method given in paper [32] is used to perform homomorphic computation on real data. It is also known as HEAAN (Homomorphic Encryption for the Arithmetic of Approximate Numbers). A noise or error "e" is added with the message "M" to form the plaintext "P", i.e., (M + e), which we want to encrypt. Here, error "e" is added as part of the RLWE scheme for security purposes. The input to the encryption process is (M + e) and it will generate a ciphertext which is the approximate value of the message "M" in the encrypted form. Computers use fixed-point or floating-point representations to handle real data. Some real data cannot be represented using this system. Therefore, we will take their approximate value using some predefined functions such as truncation or rounding off. This approximation results in errors that can accumulate and expand during computation. But, in a realistic situation, these errors are not very significant, so the final result will be satisfactory. Therefore, we can think of the RLWE error in the encrypted domain as similar to a truncate or rounding error in the plain domain. The process of re-scaling is used to make plaintext smaller. The MODSWITCH concept is used for re-scaling. Y. Su et al. [33] provided the hardware architecture both for the keyswitch and modswitch modules to implement a leveled BGV scheme. Cheon, Jung Hee et al. [34] presented a variant of approximate homomorphic encryption that uses RNS decomposition of cyclotomic polynomials and the NTT conversion on each of the RNS components. Lee, Eunsang et al. [35] proposed

a method to find the degrees of component polynomials optimized for the RNS-CKKS scheme. The ciphertext in CKKS [32], BGV [3], BFV [2,6], and BV [5] contains a pair of polynomials. One polynomial is used for message information and the other is used for decryption. The coefficients of these polynomials are bound by integer q, which is known as the ciphertext coefficient modulus.

The procedure for plaintext encoding in the CKKS method is to multiply the number by a scaling factor Δ and round to the nearest integer. This process will shift the LSB (lowest significant bit) of message M to the left far away from "e". Here, the assumption is that the LSB of M, which is distorted by adding "e", is not very significant. The decoding procedure is to divide the encoded number by the same scaling factor Δ used for encoding. Here, we will explain it with an example.

Let $M = \Pi = 3.14159265358979323846$, $e = 20$, and $\Delta = 106$

The encoded message P is: $P = \Delta^* M + e = 3141592 + 20 = 3141612$

The decoded message M is: $M = 3141612/106 = 3.141612$

Plaintext and Ciphertext spaces: In the CKKS method, the plaintext space is the set of real numbers that can be represented with a fixed level of precision. This set of numbers is transformed into a fixed-point representation through the plaintext encoding process. The plaintext space is typically represented by a polynomial ring.

The ciphertext space, on the other hand, is the set of numbers that are obtained after encrypting the plaintext. In the CKKS method, these numbers are represented by polynomials with complex coefficients, which are obtained by applying an encryption algorithm to the plaintext. The encryption algorithm used in the CKKS method is generally based on the polynomial version of the Number Theoretic Transform (NTT).

The plaintext and the ciphertext spaces in CKKS are almost the same. The elements are from the polynomial ring given in (1).

$$R_q = Z_q[x]/f(x) \tag{1}$$

Here, q is an integer which is known as coefficient modulus and $f(x)$ is a polynomial which is known as polynomial modulus. All the elements of R_q are polynomials whose degree is bounded by the degree of $f(x)$ and coefficients are integers bounded by q.

Normally, $f(x) = x_n + 1$ and $n = 2^k$, where k is any positive integer and n is known as the ring dimension.

In the homomorphic encryption scheme, lattice-based cryptography schemes' popularity is increasing because they cannot be easily broken by quantum computers. Some of the works are based on Learning With Error (LWE) and some are based on Ring Learning With Errors (RLWE) given in [7,8,10,33]. To design a homomorphic encryption scheme, CKKS implements Ring Learning with Errors (RLWE). It enables the calculation of complex value vectors including real values. Until recently, only integers could be used for homomorphic computing. This moves us a step closer to practical uses for machine learning that protect privacy. It supports approximately adding and multiplying encrypted messages, as well as a novel re-scaling method for controlling the size of plaintext. Before going into a detailed explanation of the CKKS method, we will explain the RLWE method.

3.4. Ring Learning with Errors (RLWE)

RLWE, which stands for Ring Learning With Errors, is a mathematical problem that forms the basis for some cryptography schemes, particularly lattice-based encryption and homomorphic encryption. It is a variant of the Learning With Errors (LWE) problem, adapted for the ring structure of certain mathematical objects called polynomial rings. V. Lyubashevsky et al. [36] introduced an algebraic variant of LWE which is called RLWE. It is considered hard to solve, which makes it suitable for cryptography applications. Let us explain RLWE with some mathematical equations. RLWE is typically defined over a polynomial ring denoted as R_q, where q is a prime number. Elements of R_q are polynomials

with integer coefficients modulo q. For example, a polynomial $f(x)$ in R_q can be represented as given in (2).

$$f(x) = a_0 + a_1 * x + a_2 * x^2 + ... + a_n * x^n (\text{mod } q) \tag{2}$$

The RLWE problem [36] is defined over a polynomial ring R_q given in (3).

$$R_q = Z[x]/x^n + 1 \tag{3}$$

where n is a power of 2, and the goal is to find a secret polynomial s in R_q that is close to a given polynomial a in R_q, subject to the constraint that the coefficients of s are chosen from a small set, such as $\{0, 1, \ldots, q-1\}$ for some prime q. The security of the RLWE problem and its variants is based on the assumption that it is hard to approximate the coefficients of s to within a small error, even if one has access to many samples of the form $(a, as + e = b(\text{mod } q))$ for a randomly chosen a in R_q and a small error e in R_q. Here, (a, b) are elements of a ring R_q and are part of the public key.

The complete algorithm for CKKS operations where the RLWE concept is used is given in Section 5.

3.5. Polynomial Multiplication and Number Theoretic Transform

Here is a high-level explanation of polynomial multiplication and the use of the Number Theoretic Transform (NTT) in the context of the CKKS FHE scheme.

Polynomial Multiplication: In the CKKS scheme, plaintexts are represented as polynomials with coefficients. For example, if m_1 and m_2 are plaintexts represented as polynomials given in (4) and (5), respectively.

$$m_1(x) = a_0 + a_1 * x + a_2 * x^2 + ... + a_{(n-1)} * x^{(n-1)} (\text{mod } q) \tag{4}$$

$$m_2(x) = b_0 + b_1 * x + b_2 * x^2 + ... + b_{(n-1)} * x^{(n-1)} (\text{mod } q) \tag{5}$$

The product of these polynomials, $m_1(x)$ and $m_2(x)$ is given in (6).

$$m_{prod}(x) = c_0 + c_1 * x + c_2 * x^2 + ... + c_{(2n-2)} * x^{(2n-2)} (\text{mod } q) \tag{6}$$

Polynomial multiplication can be performed using various methods, and the NTT is one such method that provides computational efficiency.

Number Theoretic Transform (NTT): The Number Theoretic Transform (NTT) is a mathematical technique that transforms a polynomial from its coefficient representation into a different representation called the frequency or point-value representation. It is a generalization of the Fast Fourier Transform (FFT) and is particularly useful in algebraic number fields and finite fields, which are relevant in the context of certain cryptographic algorithms, including fully homomorphic encryption schemes like CKKS.

The NTT involves evaluating the polynomial at the n^{th} roots of unity. In other words, let ω be a primitive n^{th} roots of unity. The values $\omega, \omega^2, ..., \omega^{(n-1)}$ are used in the transformation.

The forward NTT transforms the polynomial coefficients $a_0, a_1, ..., a_{n-1}$ into a set of values $A(\omega), A(\omega)^2, ..., A(\omega)^{(n-1)}$ using a specific formula given in (7).

$$A(k) = \sum_{j=0}^{n-1} a_j . \omega^{jk} (\text{mod } q) \tag{7}$$

This transformation converts the polynomial from its coefficient representation to its frequency representation.

The inverse NTT transforms the frequency representation back to the coefficient representation. It takes the values $A(\omega), A(\omega)^2, ..., A(\omega)^{(n-1)}$ and computes the coefficients $a_0, a_1, ..., a_{n-1}$ using another formula given in (8).

$$a_j = 1/n \sum_{k=0}^{n-1} A(k).\omega^{-jk} (\text{mod } q) \tag{8}$$

This transformation allows for the recovery of the original polynomial from its frequency representation. In the context of fully homomorphic encryption schemes like CKKS, the NTT is often used to perform efficient polynomial multiplications, which are crucial for various homomorphic operations. The efficiency gains from the NTT contribute to the practicality of homomorphic encryption in real-world applications.

3.6. KNN Algorithm

This approach for supervised machine learning can be applied to challenges related to regression and classification. A new data point should be categorized based on the majority class of the k-nearest data points, which may be found by using a distance metric to locate the k-closest data points. This is the basic idea behind KNN.

When classifying a sample, the method first determines its unknown class by comparing it using a distance metric to the k samples in the training set that are the closest to it. The majority class among these k-nearest neighbors is then assigned to the unknown sample. The basic pseudocode for the KNN algorithm for classification is given in Section 5.

Given a dataset with labeled examples (x_i, y_i), x_i represents the feature vector of the i^{th} data point and y_i is the corresponding class label and a new data point X_u for which the class label needs to be predicted.

Mathematically, the KNN algorithm for classification is given in (9).

$$\hat{y} = argmax_j \sum_{i=1}^{k} I(y_i = j) \tag{9}$$

where

- $\hat{y}$ is the predicted class label for the new data point X_u.
- k is the number of nearest neighbors to consider.
- y_i is the class label of the i^{th} nearest neighbor.
- I(.) is the indicator function, which equals 1 if the condition inside is true and 0 otherwise.
- $argmax_j$ finds the class label with the highest count among the k-nearest neighbors.

The KNN algorithm can also be used for regression tasks, wherein the goal is to predict a continuous target variable. In this case, the prediction for the new data point X_u is calculated as the average (or weighted average) of the target values of its k-nearest neighbors. The mathematical form for the regression equation is given in (10).

$$y_u = 1/k \sum_{i=1}^{k} y_i \tag{10}$$

where

- y_u is the predicted target value for the new data point X_u.
- y_i is the target value of the i^{th} neighbor.

Theoretically, the data points used in KNN can be encrypted using CKKS, protecting the privacy of the data while enabling the KNN algorithm to operate on it. However, the massive datasets and real-time computations needed for useful machine learning applications cannot be handled by the current implementation of CKKS due to its lack of

efficiency. More research and development are needed to make CKKS more efficient and practical for use in KNN and other machine learning algorithms.

4. Proposed Method

This section explains the overall design diagram of the proposed method that will be simulated using FPGA.

4.1. System Architecture

Different modules of the proposed architecture are given in Figure 2 and the functions of all the modules are as follows:

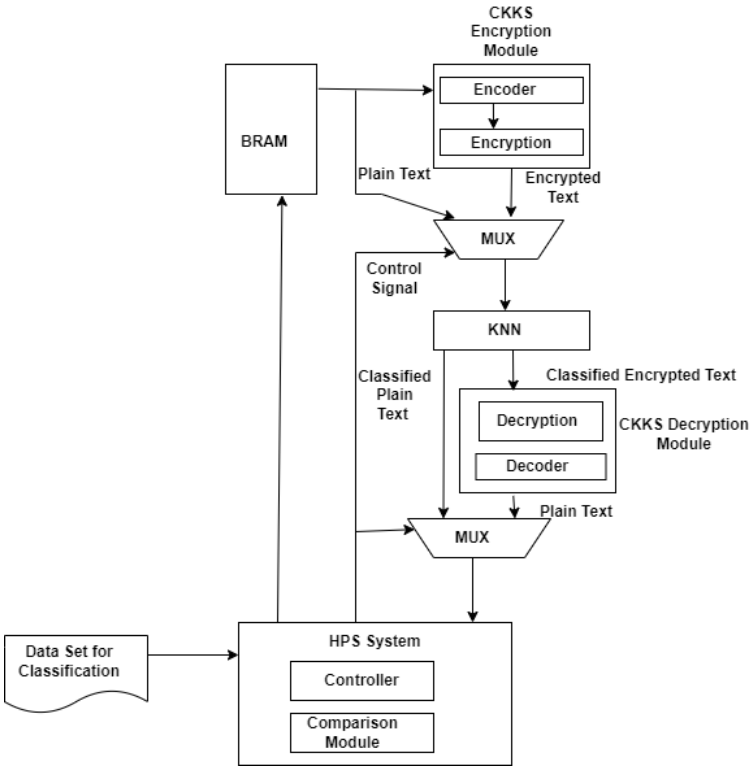

Figure 2. Proposed system architecture.

- The dataset for classification is transferred through the Ethernet interface of a Hard Processor System (HPS). Received data by HPS are stored in BRAM (Block Random Access Memory) inside FPGA for classification.
- The CKKS encryption module contains two sub-modules, an encoder, and an encryption module. It reads the input dataset from BRAM and performs the encoding operation. The encryption operation is applied to the encoded data and gives the output as the encrypted text.
- The KNN classification module performs the classification operation on the encrypted text and also on the plaintext. The input to the KNN module is controlled by MUX. The encrypted text is chosen if the control signal is 1; otherwise, the plaintext is chosen.
- The CKKS decryption module takes the input as classified encrypted text. Two sub-modules are included in it: one for decryption and the other for the decoder. The encrypted classified text is decrypted by the decryption module; then, it is decoded to obtain the plaintext.
- The hard processor system (HPS) of the FPGA contains the controller and the comparison module. The controller gives the control signal to the MUX, to select the input. The comparison module compares two classified results. The KNN classification is applied to the encrypted text in one instance, and to the plaintext in another. Finally, the comparison result is stored in the BRAM.

4.2. Design Architecture to Implement CKKS in Intel Agilex SoC FPGA

Figure 3 shows the design architecture to implement CKKS in FPGA. The dataset for classification is transferred through the Ethernet interface of HPS (ARM processor) and stored in a block RAM inside FPGA via AMBA-AXI Bus. BRAM1 and BRAM2 are used in a ping-pong fashion to have a continuous flow of data. The CKKS-KNN block receives the data and the complex data are converted into a real polynomial using a systolic array based on QR decomposition for matrix inversion. Polynomial coefficients are encrypted to apply the KNN algorithm. The distance calculation and sorting operation of KNN on the encrypted data are also implemented in FPGA. The encrypted sorted data are decrypted and decoded using matrix multiplication in FPGA. The final K clustering (voting) was carried out in the software residing in HPS.

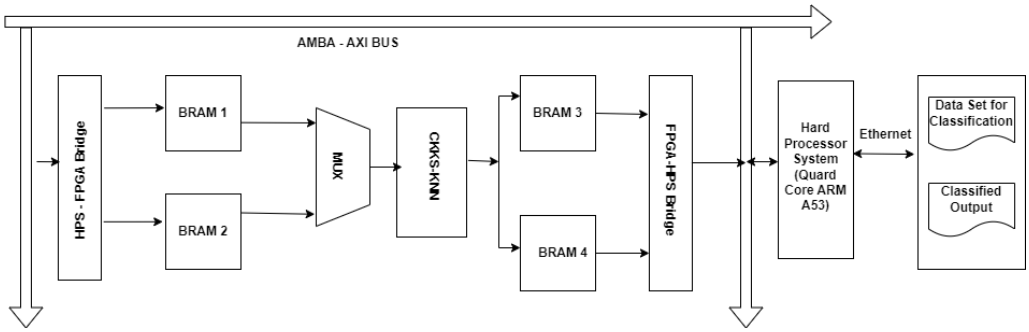

Figure 3. Design architecture to implement CKKS in Intel Agilex SoC FPGA.

In the subsequent sections, a detailed explanation of the CKKS module and KNN classification module is given with the design diagram.

Different parameters, which are used in the CKKS FHE scheme, are tabulated in Table 1.

Table 1. CKKS FHE parameters and their description.

Parameter	Description
M	Original Message
e	Noise (Very small in magnitude)
P	Plaintext $(M + e)$
Δ	Scaling factor
q	An integer coefficient modulus
f(x)	A polynomial modulus
n	Ring modulus
L	Level of freshly encrypted ciphertext
q_l	Coefficient modulus at level l, where $1 \leq l \leq L$
a	A random polynomial sampled uniformly from R_{q_L}
S_k	Secret key sampled from a polynomial of degree "n" with coefficients in $\{-1, 0, 1\}$
V	Vadermonde matrix
m	Polynomial modulus degree
d	Number of datasets
N	Size of the message after encoding
e_1, e_2	Small value compared to message and it is generated using Gaussian distribution. It is in the field of qL.
u	A sampled polynomial with coefficients in $\{-1, 0, 1\}$

4.3. FPGA-Based Architecture for the CKKS Method

The hardware architecture to speed up the CKKS FHE technique is presented in this section. CKKS operates on a polynomial ring, using a technique called polynomial

encoding. The plaintext data are represented as coefficients of a polynomial, and encryption involves transforming these coefficients into a polynomial with encrypted coefficients.

The hardware design architecture for different blocks, as well as the NTT and INTT for polynomial operations, evaluation of encrypted data, encoding, decoding, encryption, and decryption, are addressed in the sections that follow.

The proposed FPGA-based architecture for the CKKS method, as shown in Figure 4, is divided into three sub-modules. Key generation, encryption, and decryption are all handled by separate blocks. The encoder block is included in the encryption module and the decoder block is included in the decryption module. This section explains the functioning of these modules.

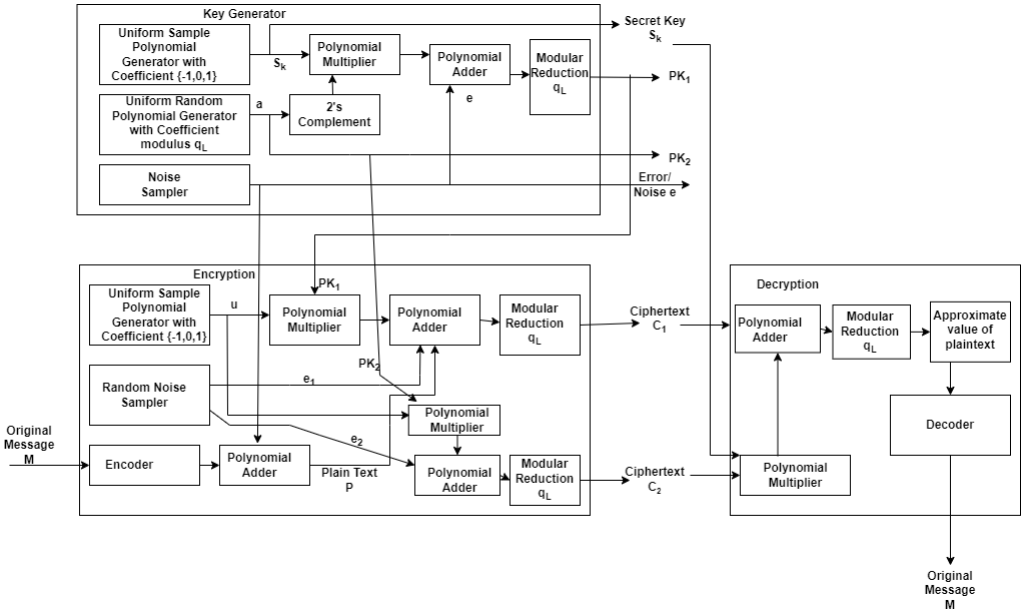

Figure 4. FPGA-based architecture for the CKKS method.

For encryption operation in CKKS, we need a secret key S_k and two public keys PK_1 and PK_2. The secret key S_k is generated by a sample polynomial generator with coefficients in $\{-1, 0, 1\}$. A uniform random polynomial generator with coefficient modulus q_L generates the "a", which is a random polynomial. The noise sampler generates the noise "e" sampled from discrete Gaussian distribution χ. Two public keys PK_1 and PK_2, which are two polynomials, are calculated as follows using a polynomial multiplier, adder, and modulus modules given in (11) and (12), respectively.

$$PK_1 = (-a * S_k + e) \mod q_L \tag{11}$$

$$PK_2 = a \tag{12}$$

To encrypt a plaintext message P, i.e., $(M + e)$, three random polynomials u, e_1, and e_2 are generated. Ciphertexts $C = (C_1, C_2)$ are generated as given in (13) and (14), respectively.

$$C_1 = (PK_1 * u + e_1 + M) \mod q_l \tag{13}$$

$$C_2 = (PK_2 * u + e_2) \mod q_l \tag{14}$$

The polynomial adder, multiplication, and modular reduction modules perform addition, multiplication, and modular operations, respectively.

Decryption is performed by evaluating the input ciphertext in level l on the secret key to generate an approximate value of the plaintext message as given in (15).

$$\bar{M} = (C_1 + C_2 * S_k) \mod q_l \tag{15}$$

This approximate value is decoded to obtain the original message.

4.4. Design Diagram for Encoder and Decoder

Encoding and decoding operation in CKKS involves matrix inversion. Here, FPGA-friendly QR decomposition is used for matrix inversion. Figure 5 shows the block diagram for QR decomposition and back substitution. The CKKS-KNN block receives the data and the complex data are converted into a real polynomial using a systolic array based on QR decomposition for matrix inversion. The systolic array to find the inverse of the 64 × 64 matrix is shown in Figure 6. In the context of a systolic array, each processing element (PE) is responsible for computing a part of the QR decomposition. Each processing element in the systolic array can be designed to handle polynomial computations, such as multiplication, addition, and other operations required during the matrix inversion process. The working concept of each processing element is shown in Figure 7 and explained below.

The QR decomposition algorithm [37] is first used to transform the Vadermonde matrix V into an upper triangular matrix R (N × N matrix) and the vector **M** into another vector **u**, as shown in (16) and (17). The coefficient vector **P** is computed using a procedure called back substitution (18) and (19). Table 2 gives the details of logic utilization for QR decomposition when it is implemented on Agilex7 FPGA (AGFB014R24B2E2V).

$$VP = M \tag{16}$$

$$RP = u \tag{17}$$

$$P_N = u_N / R_{NN} \tag{18}$$

$$P_i = 1/R_{ii}(u_i - \sum_{j=i+1}^{N} R_{ij}P_j); i = N-1, ..., 1 \tag{19}$$

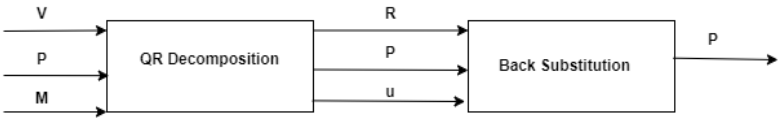

Figure 5. QR decomposition and back substitution block diagram.

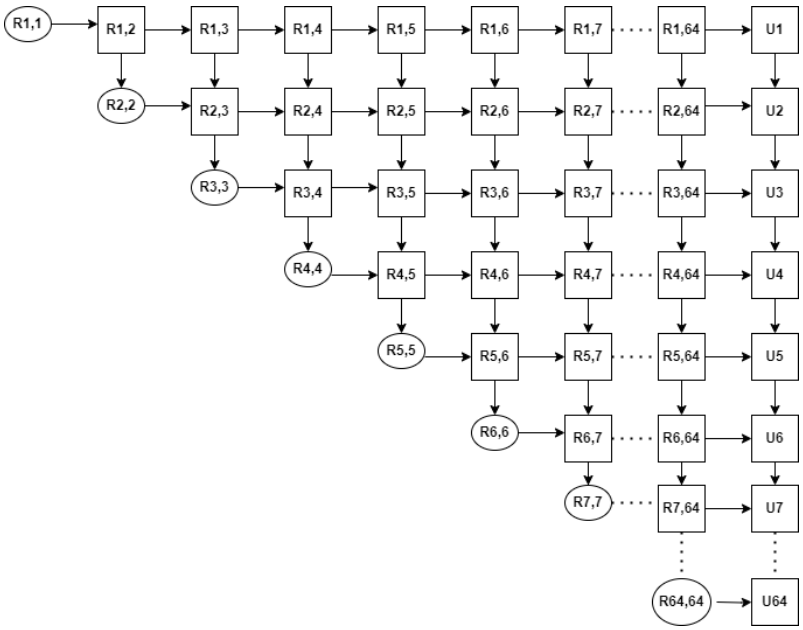

Figure 6. Systolic array to find the inverse of 64 × 64 matrix.

Table 2. FPGA resource utilization of CKKS encoder/decoder.

Family	Agilex7 FPGA
Device	AGFB014R24B2E2V
LUT (ALMs)	69,228
Block memory (in bits)	557,136
DSP Block	72

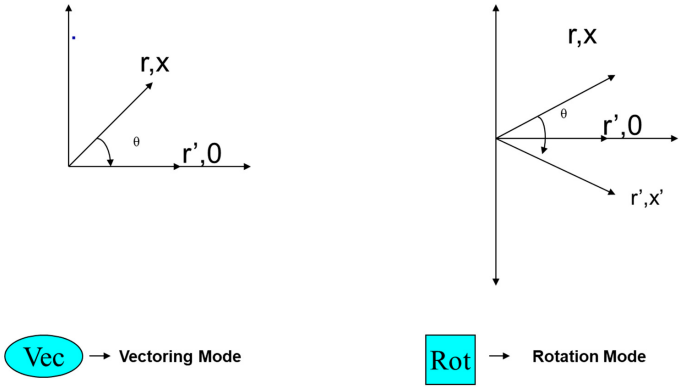

Figure 7. Vectoring mode and rotation mode of QR decomposition.

The vectoring operation converts rectangular coordinates into polar coordinates. The rotating block rotates any complex number by θ degrees. The oval-shaped block of the systolic array represents the vectoring block and the rectangular block represents the rotation block for QR decomposition.

4.5. Design Diagram for Number Theoretic Transform

The design diagram for NTT from our previous work [15] is shown in Figure 8. The entire data stream is split into three blocks, as seen in the figure. The Butterfly Unit, Bit Reverse Block, and Dual-Port Memory are these blocks. The entered data are simply saved in the dual-port memory. That keeps all n coefficients of the polynomial. It is easier to read and write simultaneously with this dual-port memory, which facilitates our proposed architecture's pipeline structure.

The two input and output data buses of this dual port memory are represented by the upper and lower data paths of the butterfly unit. It can handle two readings and two writes in a cycle. Both modular addition and modular multiplication are handled by the butterfly unit. The Butterfly computation is then carried out by providing the dual-port memory with the output of the Butterfly unit. The bit reverse block ultimately executes a bit reverse operation to reverse the bits of the NTT operation output since the input values are in natural order while the index of NTT values is in bit reverse order.

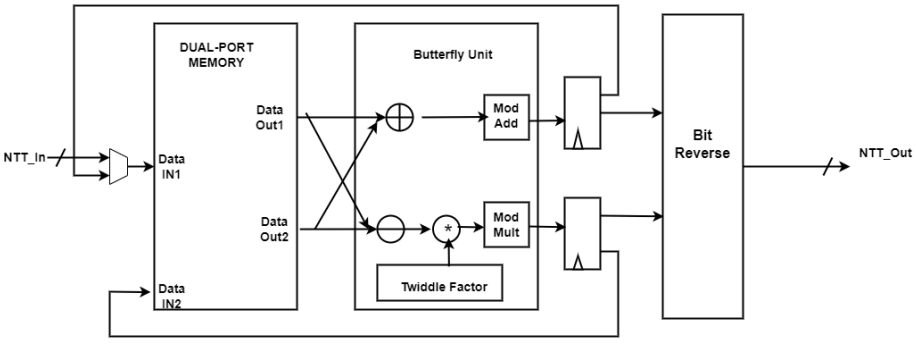

Figure 8. Design diagram for number theoretic transform.

4.6. Design Diagram for KNN Classifier Module

In this proposed design, the KNN classifier module is divided into four submodules. These submodules are normalization, calculating the distance, sorting the calculated distance, and assigning the classes. The detailed internal architecture is explained here with the diagram.

4.6.1. Normalization Process

Normalization is an important step in KNN classification before calculating the distance. Since the dataset contains values of different ranges and magnitudes, the normalization process is applied to make the features of data points a similar range. To perform this, we selected the MIN-MAX normalization process. This method will scale the features of data points within the range of 0–1 using the following formula given in (20):

$$X_{norm} = X - min(X)/(max(X) - min(X)) \tag{20}$$

where X is the original value of the feature, $min(X)$ is the minimum value of the feature from the dataset, $max(X)$ is the maximum value of the feature from the dataset, and X_{norm} is the new normalized value of X in the range 0–1.

The circuit diagram to find the max and min values from the array of vectors is shown in Figure 9.

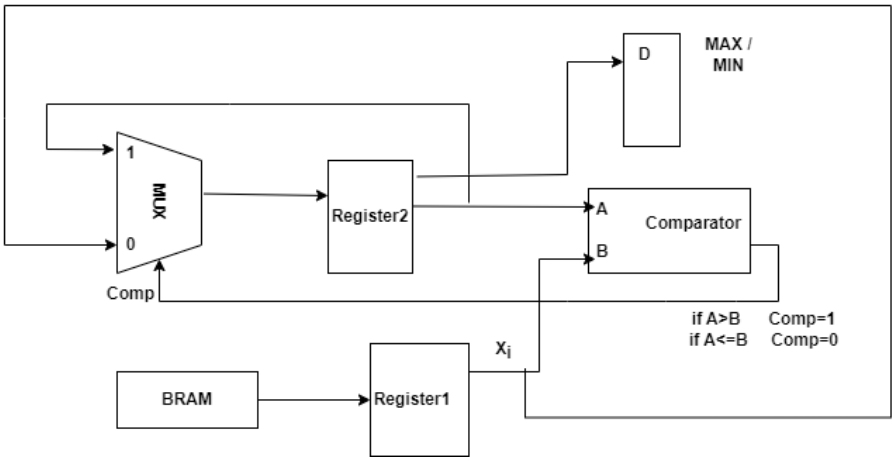

Figure 9. Finding maximum/minimum values from array of vectors.

The array of vectors is stored in the BRAM. Initially, register2 contains a minimum value of 0 as the first element for A. One element is read from BRAM and stored in register1. It is assigned to B. The comparator circuit will compare A and B; if $A > B$, then the output is 1; otherwise, it is 0. If it is 1, then register2 contents will not change. Otherwise, B's value will be stored in register2. This process will continue until all the elements are compared. Finally, the contents of register2 will be the maximum value. The final max value will be stored in the D flip-flop.

To find the minimum element from the array, register2 will be initialized with a large value. When $A < B$, the comparator will give the output as 1; otherwise, it will be 0. If it is 0, then B's value will be stored in register2. Repeat the process until all the elements are compared. In the end, the minimum element will be in register2 and will be transferred to D flip flop.

The comparison operations are given in (21) and (22).

$$GT(A, B) = \begin{cases} 1, & \text{if } A > B; \\ 0, & \text{otherwise} \end{cases} \tag{21}$$

$$LT(A,B) = \begin{cases} 1, & \text{if } A < B; \\ 0, & \text{otherwise} \end{cases} \tag{22}$$

4.6.2. Novel Hardware Architecture for KNN Classification

The KNN classification hardware architecture computes the distance and sorting of distances in FPGA. The voting and classification are performed in HPS software. The hardware architecture diagram is shown in Figure 10.

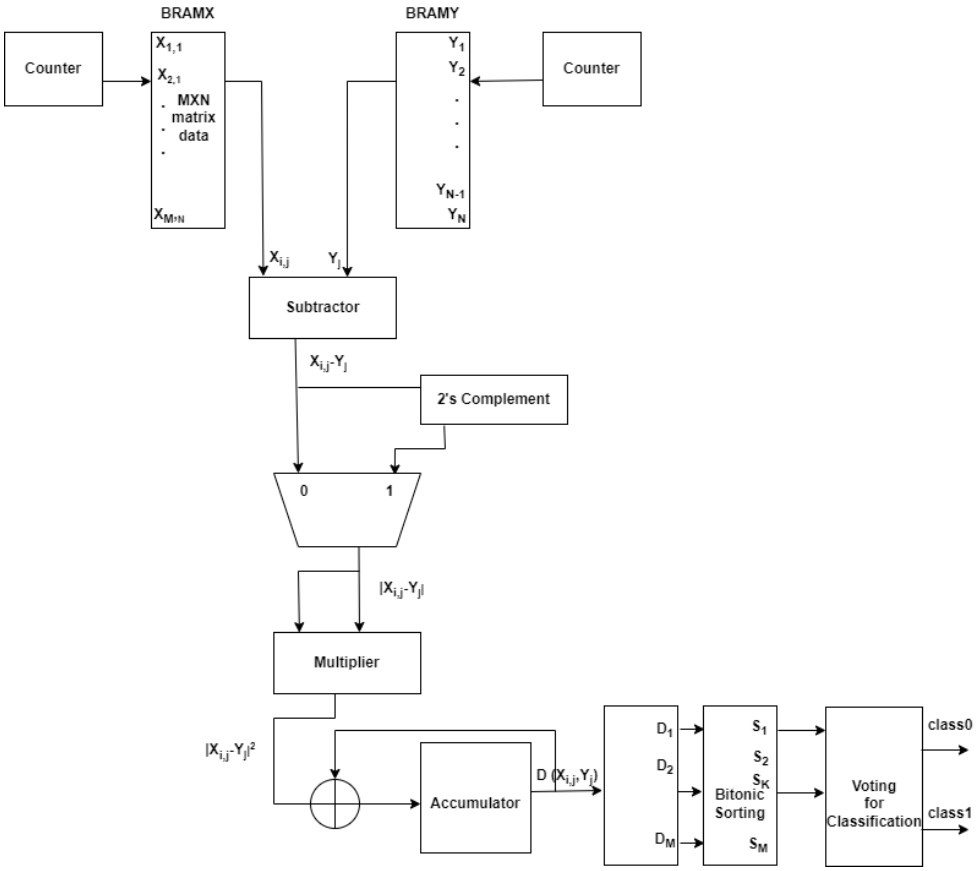

Figure 10. Hardware architecture for KNN classification.

The next step in KNN classification is calculating the distance between training data and test data. There are different distance calculation algorithms commonly used in KNN classification such as Euclidean distance, Manhattan distance, Minkowski distance, and Cosine similarity.

In this work, the Euclidean distance method is used to calculate the distance. The Euclidean distance formulae are shown in (23).

$$D(X,Y) = \sum_{i=1}^{N} |X_i - Y_i|^2 \tag{23}$$

BRAMX contains the training dataset, and BRAMY contains the testing dataset. The subtractor module finds the value of $X_i - Y_i$. The output of the MUX will be a mod of $X_i - Y_i$, i.e., $|X_i - Y_i|$. A multiplier is used to find the square of this value. We have used the squared distance instead of the square root for distance calculations. Since the computation of the square root on encrypted data is more complex, the calculated distances are sorted using a bitonic sorting algorithm.

Bitonic sorting [38] is a parallel sorting algorithm that was designed to efficiently sort sequences in a parallel processing environment. Bitonic sorting is particularly suitable for

parallel architectures like systolic arrays and other parallel computing systems. The algorithm works by recursively building a bitonic sequence and then repeatedly sorting the sequence in a bitonic manner until the entire sequence is sorted. Figure 11 shows the structure of a bitonic sorting network for a dataset of 512 values. The first step is repeatedly dividing the sequence into two halves and sorting each half in a bitonic manner. After constructing a bitonic sequence, the algorithm performs bitonic merges to sort the entire sequence. The bitonic merge step is repeated until the entire sequence becomes sorted. The number of stages required to sort 512 data values is $log_2 512 = 9$. The RTL view of bitonic sort in FPGA is shown in Figure 12. The FPGA resource utilization of the bitonic sort is presented in Table 3.

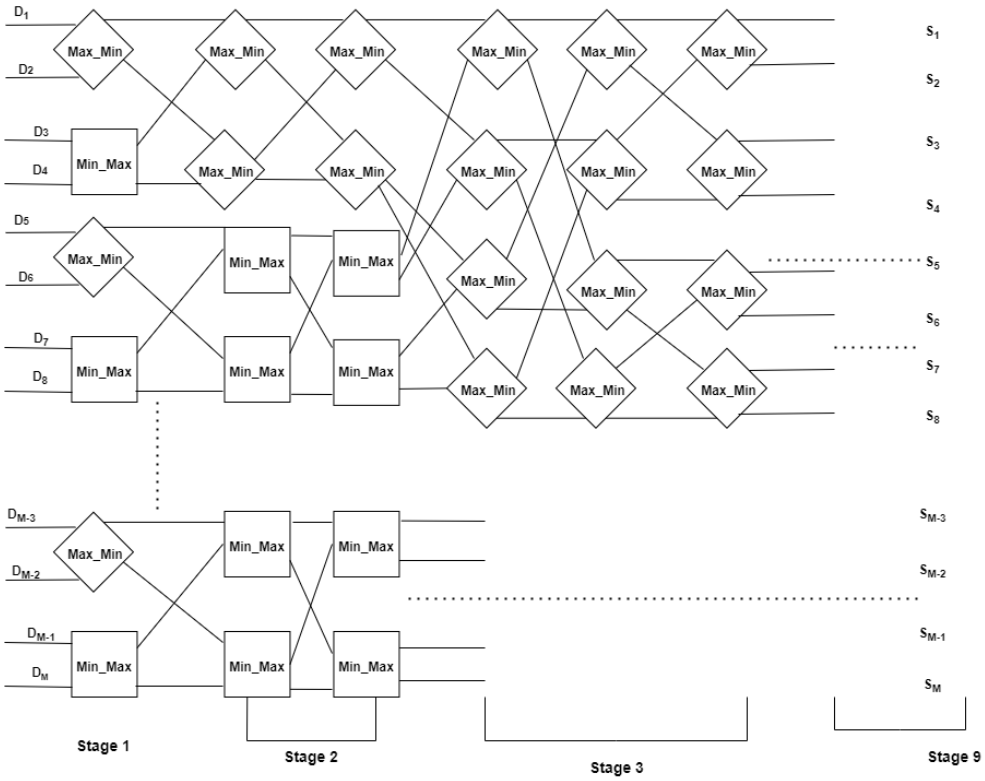

Figure 11. Bitonic sorting network for 512 data values.

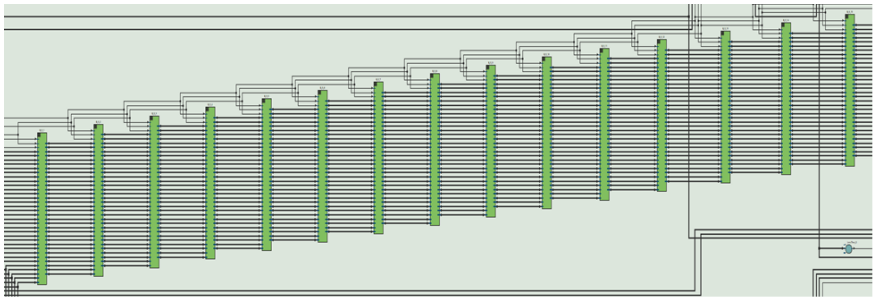

Figure 12. FPGA simulation of bitonic sorting for 512 data values.

Table 3. FPGA resource utilization of bitonic sorting.

Family	Agilex7 FPGA
Device	AGFB014R24B2E2V
LUT (ALMs)	268,560
Logic Register	410,400

5. Algorithm

This section gives the step-by-step process of performing all the operations in the form of the algorithms. The pseudo-code for the KNN classification algorithm is given in Algorithm 1. The breast cancer dataset is used as input to predict whether the cancer is benign or malignant. It gives the class label as benign or malignant for a new data point depending on the K neighbors.

Algorithm 1 KNN Algorithm for Classification

Input: Given a dataset with labeled data points: $D = (x_1, y_1), (x_2, y_2), ..., (x_n, y_n)$, where x_i represents the feature vector of the i^{th} data point and y_i is its corresponding class label.
Output: Prediction of the class label for the given data point X_u.

1: Compute the distance between the new data point X_u and every data point in the training dataset. The commonly used distance metric is Euclidean distance given in (23).
2: Sort the distances in ascending order and select the K data points (k neighbors) with the smallest distances to X_u.
3: Count the occurrences of each class label among the k neighbors.
4: Assign the class label that occurs most frequently as the predicted class label for the new data point X_u.
5: **return** Class Label

The secret key and the public key generation steps are shown in Algorithm 2. The input parameter n is the ring modulus, q is the coefficient modulus, and it is a power of 2.

Algorithm 2 Algorithm for CKKSKeyGeneration

Input: n, q, q_L, χ
Output: S_k, PK_1, PK_2
Initialization: Here, the parameters n, q, q_L selected based on the security parameter λ and q, q_L, n are the powers of 2. L is the level of recently encrypted ciphertext. Each level is associated with a coefficient modulus q_l.

1: Secret key S_k is a polynomial of degree n with coefficients in $\{-1, 0, 1\}$
$S_k \leftarrow \{-1, 0, 1\}^n$
2: Public key PK is a pair of polynomials (PK_1, PK_2)
$PK = (PK_1, PK_2)$
3: $e \leftarrow \chi$
4: $PK_2 = a \leftarrow R_{q_L}$
5: $PK_1 \leftarrow (< -a * S_k > + e) \bmod q_L$
6: **return** Secret key S_k, Public Key $PK = (PK_1, PK_2)$

Algorithm 3 shows the steps for encoding the original message into real values. The input will be a $n/2$ vector of complex numbers and it will be expanded by adding its complex conjugate. Then, it will be multiplied by the scaling factor Δ. The output will be a single plaintext P.

Algorithm 3 Algorithm for CKKSEncoder

Input: $n/2$ vector of complex numbers $Z \in C^{n/2}$
Output: A single plaintext $P \in R$

1: Expands a vector of complex numbers $C^{n/2}$ by expanding it with its complex conjugate. It is denoted as $\pi^{-1}Z$.
2: Multiply it by Δ for precision.
3: **return** P = ENCODE$(Z, \Delta) = \lfloor \Delta * \pi^{-1}Z \rceil$

Algorithm 4 shows the steps for decoding to obtain the original message back. In this step, it will be divided by the scaling factor Δ.

The encryption algorithm is given in Algorithm 5. The output of the Algorithm 3 is the input for this algorithm as plaintext and the public key generated by Algorithm 2. The output will be a pair of ciphertexts.

Algorithm 4 Algorithm for CKKSDecoder

Input: A single plaintext $P \in R$
Output: $n/2$ vector of complex numbers $Z \in C^{n/2}$
1: $Z = \text{DECODE}(P, \Delta) = \pi(1/\Delta * P)$
2: **return** Z

Algorithm 5 Algorithm for CKKSEncryption

Input: Plaintext P which is the encoded form of the original message M, public key PK, three sample random polynomials u, e_1, e_2.
Output: Ciphertext $C \leftarrow (C_1, C_2)$
Initialization: The plaintext $P \in R_{q_l}$ will be encrypted using the public key (PK_1, PK_2), a sample vector $u \leftarrow \{-1, 0, 1\}^n$, and $e_1, e_2 \leftarrow \chi$.
1: $C_1 \leftarrow \{(PK_1.u + e_1 + P)\} \bmod q_l$.
2: $C_2 \leftarrow \{(PK_2.u + e_2)\} \bmod q_l$.
3: **return** ciphertext $C \leftarrow (C_1, C_2)$

The decryption algorithm is given in Algorithm 6. The evaluation algorithm for performing different operations on the ciphertext is given in Algorithm 7.

Algorithm 6 Algorithm for CKKSDecryption

Input: C, S_k
Output: Plaintext P
1: $P \leftarrow \{(C_1 + C_2.S_k)\} \bmod q_l$.
2: **return** Plaintext P

Algorithm 7 Algorithm for Evaluation

Input: The cipher-texts $CT1, CT2$
Output: Evaluated output
Initialization: Operations on the cipher-texts
1: Homomorphic Addition:
2: EvalAdd ($CT1$,$CT2$) = $((CT1^1 + CT2^1) \bmod q_l, (CT1^2 + CT2^2) \bmod q_l) = (CT3^1, CT3^2)$ = $CT3$
3: **return** $CT3$

6. Implementation

This section gives the implementation details of the proposed method. Initially, the CKKS scheme and KNN algorithm are implemented in Python. The computational time is calculated. To program the Agilex7 FPGA, VHDL language is used. Both the CKKS scheme and KNN algorithm are simulated using Agilex7 FPGA. The KNN classification algorithm is applied both on plaintext and ciphertext. The details of the datasets and simulation results are given in Section 7.

The length of the original message M is 32. For encoding to real coefficients, the message has to be replicated twice with its complex conjugate values N = 64. This process is called canonical embedding. Therefore, the size of the data matrix P after encoding is 64×1. The size of the Vadermonde Matrix V used for encoding is 64×64. To encrypt the encoded message P, a secret key S_k, and a public key $PK(PK_1, PK_2)$ are generated, each of size 64.

After encryption, the ciphertext $C(C_1, C_2)$ is generated whose size is (64,64). The size of the polynomial modulus degree m is 64.

Implementation on FPGA Platform

To validate the above claim, the Agilex7 FPGA board from Intel is used.
The board has the following FPGA (Intel Agilex 7 FPGA, AGFB027R24C2E2V) components:

- 2,692,760 Logic Elements (LE).
- 912,800 Adaptive Logic Elements (ALM).
- 13,272 M20K Blocks.
- 45,640 MLABs.
- 8528 DSP Blocks.
- 17,056 18 × 19 Multipliers.
- 744 GPIOs.
- 372 LVDS.

All the CKKS operations such as encoding, encryption, decoding, and decryption of the dataset, and the distance calculation of KNN, are implemented in FPGA as per the diagrams given in Figures 4 and 10. The libraries, called IP blocks in platform designer of Quartus tools of Intel, such as multipliers, DSP block, memory(RAM, ROM), dual port RAM, and FIFO, are used. We designed a partitioned dataset to 32 groups and paralleled the CKKS, but the distance calculation of KNN is performed on all of the data. The details of hardware realizations with utilization of multipliers, LUTs (Lookup Table), ALM, and speed performance with a max clock frequency of operation were tabulated. These results are given in Section 7.

7. Result and Discussion

To test the proposed design model, a breast cancer dataset is used as input to predict whether the cancer is benign or malignant. This dataset is taken from the breast cancer Wisconsin (diagnostic) dataset and can also be found in the UCI machine learning repository. It contains 30 features computed from a digitized image of a breast mass. The dataset contains 569 entries. Each sample contains the following attributes: (1) Patient id, (2) Radius_mean, (3) Texture_mean, (4) Perimeter_mean, (5) Area_mean, (6) Smoothness_mean, (7) Compactness_mean, (8) Concavity_mean, (9) Concave points_mean, (10) Symmetry_mean, (11) Fractal_dimension_mean, etc. All the attribute values are real numbers except the patient ID. We have labeled malignant (M) as "0" and benign (B) as "1". This dataset contains 37.26% patient diagnoses labeled as malignant and 62.74% patient diagnoses labeled as benign.

The KNN algorithm is applied to classify this dataset for different K values and the percentage of accuracy is calculated. The dataset is encrypted using the CKKS method and KNN classification is applied to the encrypted data. The percentage of accuracy is calculated. Both the results are tabulated and shown in Table 4.

Table 4. Percentage of accuracy for different K values.

K Value	Accuracy in %Age (Without Encryption)	Accuracy in %Age (With Encryption)
3	91.23	64
5	93.86	65
7	94.74	68
8	94.74	63
10	94.74	66
11	96.49	68
13	96.49	74
14	96.49	75
21	96.49	75
40	95.61	75
120	90.35	77
242	82.46	71

7.1. Computational Complexity

The computational complexity is calculated for the proposed model for CKKS operation and KNN is applied on encrypted data for classification. A detailed description is given in the following subsections.

7.1.1. CKKS Operation

CKKS encryption involves additional computation of encoding complex numbers to real coefficient polynomials, which involves matrix inversion. The polynomial multiplications consume maximum resources, mainly the number of multipliers. The computational complexity of the CKKS method is calculated for different operations such as Encoding, Key generation, Encryption, Decryption, and Decoding. This computational complexity is calculated concerning the number of multiplications, additions, polynomial multiplications, and memory storage for different operations of CKKS. This computational complexity is shown in Table 5.

Here, NTT(Sk) can be pre-computed and stored to reduce the number of multiplications. Therefore, the number of polynomial multiplications is reduced to $2n \times log_2 n$ in place of $3n \times log_2 n$.

Table 5. Computational complexity of CKKS.

	Encoding	**Key Generation**	**Encryption**	**Decryption**	**Decoding**
Data Dimension	64×1	64, (64,64)	(64,64)	64×1	64×1
Operation	$\mathbf{P = V^{-1}M}$	**Sk**,**Pk** $(\mathbf{Pk_1}, \mathbf{Pk_2})$	$\mathbf{C} = (\mathbf{C1}, \mathbf{C2})$	$\mathbf{P = C1 + C2 * Sk}$	$\mathbf{M = VP}$
Multiplication	$d \times (m \times m)$	$d \times m$	$d \times m, d \times m$	$d \times m$	$d \times (m \times m)$
Addition	$d \times (m \times m)$	$d \times m$	$d \times 2m, d \times m$	$d \times m$	$d \times (m \times m)$
Polynomial Multiplication	0	$d \times 2m \times log_2 m^*$	$d \times 3m \times log_2 m$, $d \times 3m \times log_2 m$	$d \times 2m \times log_2 m$	0
Memory Storage	$570 \times 64 \times 16b$	$570 \times 64 \times 16b$, $570 \times 64 \times 16b$	$570 \times 64 \times 52b$, $570 \times 64 \times 52b$	$570 \times 64 \times 52b$	$570 \times 64 \times 16b$

7.1.2. KNN on Ciphertext

Table 6 shows the computational complexity for calculating the distance on ciphertext concerning the number of multiplication, addition, and memory storage.

Table 6. Computational complexity of applying KNN on encrypted data.

Distance Calculation	$\sum_{i=1,j=1}^{m,d} (C_{ij} - CT_{ij})^2$
Multiplication	$d \times m$
Addition	$d \times 2m$
Memory Storage	$570 \times 52b$

7.2. FPGA Resource Utilization

In FPGA, mathematical computations can be conducted sequentially or in parallel. Sequential operation takes less resources but it takes more time. On the contrary, parallel schemes utilize more resources with faster computation. Figure 13 shows the single-channel diagram for implementing the proposed CKKS-KNN model in FPGA. The input to the model is the dataset which contains 570 patient data elements with 32 attributes (570×32). In this work, the single-channel CKKS-KNN is implemented in FPGA. The resource utilization and computational time of a single channel were carried out. However, we proposed a 32-channel CKKS-KNN which can be implemented in a larger FPGA available from Intel.

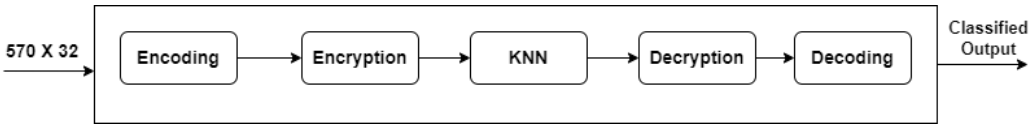

Figure 13. Single-channel CKKS-KNN.

For a typical sequential implantation scheme, the utilization of Logical resources such as LUTs, the number of dedicated registers, block memory, RAM blocks, DSP blocks, and frequency required for all the operations are tabulated and given in Table 7.

Table 7. FPGA resource utilization of single-channel CKKS-KNN in Intel Agilex FPGA.

Operation	LUT (ALMs)	No. of Dedicated Registers	Block Memory (in Bits)	RAM Blocks	DSP Blocks	F_{max}
Encoding	69,228	0	557,136	0	72	196 MHZ
Encryption	87,912	110,028	479,232	288	960	196 MHZ
KNN	268,560	410,400	29,640	48	160	196 MHZ
Decryption	29,304	36,676	159,744	96	320	196 MHZ
Decoding	69,228	0	557,136	0	72	196 MHZ

The design partition of CKKS-KNN in 32 parallel channels each for 18 data elements to speed up the computation is shown in Figure 14.

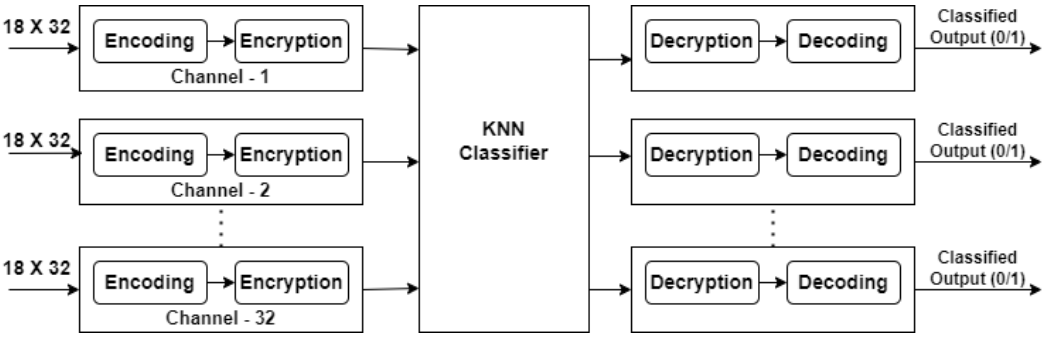

Figure 14. The 32-channel CKKS-KNN.

Table 8 details the resource utilization summary of 32-channel implementation in FPGA. Each channel caters to the encryption and decryption of 18 data elements, i.e., a total of $18 \times 32 = 570$. After the 32 parallel encryption, the classification is carried out over 570 data elements. Then, 32 parallel decryption is carried out. The utilization of LE, ALM, and DSP blocks is increased by 32 times but the memory utilization remains the same.

Table 8. FPGA resource utilization of 32-channel CKKS-KNN in intel Agilex FPGA.

Operation	LUT (ALMs)	No. of Dedicated Registers	Block Memory (in Bits)	RAM Blocks	DSP Blocks	F_{max}
Encoding	2,215,296	0	17,828,352	0	2304	300 MHZ
Encryption	2,813,184	3,520,896	479,232	288	30,720	300 MHZ
KNN	268,560	410,400	29,640	48	160	300 MHZ
Decryption	937,728	1,173,632	159,744	96	10,240	300 MHZ
Decoding	2,215,296	0	17,828,352	0	2304	300 MHZ

The resource utilization of FPGA for plaintext KNN is tabulated in Table 9.

Table 9. FPGA resource utilization of plaintext KNN in Intel Agilex FPGA.

Operation	LUT (ALMs)	No. of Dedicated Registers	Block Memory (in Bits)	RAM Blocks	DSP Blocks	F_{max}
Max Calculation	0	12,348	0	0	0	350 MHZ
Min Calculation	0	12,345	0	0	0	350 MHZ
Distance	0	26,392	18,240	4	8	350 MHZ
Decision	0	1231	3400	0	0	350 MHZ

7.3. Computational Time, Latency, and Throughput

The computational time is calculated in milliseconds (41.726 ms) and the latency in milliseconds for all the operations such as encoding, key generation, encryption, KNN on ciphertext, decryption, and decoding. Table 10 summarizes the computation time and memory usage for the calculation of Latency, for the single-channel CKKS-KNN.

Table 10. Calculation of latency and computational speed for the single-channel CKKS-KNN.

	Encoding	Key Generation	Encryption	KNN	Decryption	Decoding
Computational Time (ms)	18.67	0.819	2.311	0.437	0.819	18.67
Latency (ms)	0.146	0.291	0.583	0.146	0.146	0.146

Table 11 summarizes the computational time and latency for plaintext KNN in Intel Agilex FPGA.

Table 11. Calculation of latency and computational time for plaintext KNN.

	Max Calculation	Min Calculation	Distance	Decision
Computational Time (µs)	54.88	54.88	174	235
Latency (µs)	1.624	1.624	2.57	5.6

The computation time of the 32-channel parallel scheme is given in (24).

$$CT_p = \frac{CT_s}{N_{ch}} * \frac{f_s}{f_p} \tag{24}$$

where CT_p is the computation time of parallel architecture, CT_s is the computation time of a single channel, N_{ch} is the number of channels, f_s is the clock speed of the single channel, and f_p is the clock speed of the parallel channel. The parallel computational time of CKKS-KNN is 0.85 ms.

The summary of computational time for single-channel CKKS-KNN, 32-channel CKKS-KNN, and plaintext KNN in FPGA implementation is tabulated in Table 12.

Table 12. Summary of computational time.

Single-Channel CKKS-KNN	32-Channel CKKS-KNN	Plain Text-KNN
41.72 ms	0.85 ms	0.518 ms

7.4. Comparison with State-of-the-Art Research

CKKS-KNN is implemented in Agilex7 FPGA. In the literature [39,40], CKKS homomorphic encryption is implemented in Xilinx Virtex Ultrascale FPGA. The comparison is not easy due to the following differences:

- FPGAs are from different make (Intel and Xilinx).
- CKKS applied on machine learning algorithm only vis-a-vis CKKS encryption and decryption.

To conduct the comparison, the resource utilized by KNN is excluded. It is assumed that LUT, FF of Xilinx Vertex-Ultrascale FPGA is similar to ALM and dedicated registers of Intel Agilex FPGA. The hardware resource utilization of the proposed work is compared with the state-of-the-art research [39,40] and the summary is shown in Table 13. The comparison in Table 13 is conducted using the following parameters: N (length of the polynomial), LUT, No. of Registers, BRAM, DSP blocks, Frequency, and Latency.

It is observed that the main resource-consuming block of encryption and decryption is the polynomial multiplier and the length of the polynomial. The polynomial length in [39] is 2^{14} and in [40] it is 2^{16}. In this work, the polynomial length is 2^6 due to the features of the dataset. It is found that the resource utilization of the CKKS (single-channel) of the proposed work is lesser compared to the CKKS of [39,40]. However, for 32-channel CKKS, the resource utilization is higher.

Since the encoder/decoder (matrix inversion) and bitonic sort for KNN are implemented in FPGA, the suggested architecture's logic utilization to implement the whole CKKS-KNN over 570 × 32 datasets is greater. Moreover, 32-channel CKKS-KNN consumes more logic resources to accelerate the execution comparable to KNN calculation on plaintext. The frequency achieved for 32-channel CKKS-KNN is 300 MHZ, which is faster compared to the 196 MHZ of single-channel and 250 MHZ of [39,40].

Table 13. Comparison of FPGA resource utilization of CKKS-KNN in Intel Agilex FPGA for encryption and decryption operation.

Operation	Related Work	Device	N	LUT	No. of Registers	BRAM	DSP Blocks	Frequency (MHZ)	Latency (ms)
Encryption	[39] 2023	Xilinx Virtex Ultrascale XCU250 FPGA	2^{14}	883 K	897 K	1563	6042	250	0.1021
	[40] 2023	Xilinx Virtex Ultrascale XCU250 FPGA	2^{16}	1179 K	1036 K	828.5	12,288	250	16.869
	Proposed Work (Single Channel)	Agilex7 FPGA	2^6	87.9 K	110.028 K	288	960	196	0.583
	Proposed Work (32 Channel)	Agilex7 FPGA	2^6	2813.184 K	3520.896 K	288	30,720	300	0.018
Decryption	[40] 2023	Xilinx Virtex Ultrascale XCU250 FPGA	2^{16}	10.7*K*	6.9*K*	3	133	250	3.937
	Proposed Work (Single Channel)	Agilex7 FPGA	2^6	29.3 K	36.6 K	96	320	196	0.146
	Proposed Work (32 Channel)	Agilex7 FPGA	2^6	937.728 K	1173.632 K	96	10,240	300	0.0045

NTT-based polynomial multiplications for FPGA implementation have been published in [15,41,42] to speed up the encryption and decryption process. Polynomial multiplication is a computationally demanding technique that is frequently employed in homomorphic encryption and decryption operations.

The FPGA used in [41] is Zynq Ultrascale+ and Virtrex 7 is used in [42]. We have used Agilex 7 FPGA in our previous work [15] and in this current work. However, an attempt has been made to compare logic utilization and latency. The state of the art for polynomial multiplication using NTT is given in Table 14 for the polynomial length N = 1024.

It is observed that the logic utilization and latency of similar-length polynomials are of the same order. The logic utilization of the proposed NTT utilizes fewer LUT and DSP resources compared to [41,42]. The proposed NTT-based polynomial uses a single butter-fly taking the time of 19.76 µs compared to 27.71 µs of [41] and 2.6 µs of [42]. The LUT used

for NTT in [41] is 4.4 times more and the LUT used for NTT in [42] is 14.2 times more compared to this current work.

Table 14. Comparison of NTT performance.

Related Work	Device	N	LUT	No. of Dedicated Registers	BRAM	DSP Blocks	Frequency (MHZ)	Latency (µs)
[41] 2023	Zynq Ultrascale+	1024	3168	1440	19.5	42	185	27.71
[42] 2022	Virtex 7	1024	10,272	6704	87	80	250	2.60
Proposed Work	Agilex 7	1024	720	1159	96	3	259	19.76

Since the butter-fly consumes less LUT in the proposed work, it requires more memory to store. Therefore, the BRAM used here is 96 compared to 87 in [42] and 19.5 in [41]. The number of DSP blocks used in [41] is 14 times higher and in [42] it is 26.6 times higher compared to the proposed work. The frequency achieved here is 259 MHZ, which is faster compared to 250 MHZ in [42] and 185 MHZ in [41].

Throughput for Single and Parallel Channel

The throughput is calculated using the formulae given in (25).

$$TP_s = \frac{1}{CT_s} * DS \tag{25}$$

where TP_s is the throughput for a single channel, CT_s is the computation time for a single channel, and DS is the data size to be sent during the computation time. The throughput is 6.99 Gbits/s per single and 280 Gbits/s for the parallel channel.

8. Conclusions

In this work, FPGA-based programmable logic architecture is designed to implement the KNN algorithm on an encrypted training dataset using the CKKS fully homomorphic encryption scheme. For this experiment, a dataset consisting of 570 numbers of patients with 32 attributes is used to compare the KNN on encrypted data for the execution in the cloud and the plaintext KNN. We implemented a flexible design architecture where one can choose the fully serial, fully parallel, and mix-serial and parallel architecture to meet the real-time execution of KNN on encrypted data in a cloud computing environment. Specifically, the logic resources, latency, execution time, and throughput of computation are calculated. The throughput of the system is 6.99 Gbits/s. The time execution of KNN on plaintext (570×32) was taken as a reference and, using the new programmable architecture, one can achieve the speed of execution in a similar order of KNN in FPGA. For the FPGA implementation, we used the Intel Agilex7 FPGA (AGFB014R24B2E2V) development board and validated the speed of computation, latency, throughput, and logic utilization. The KNN on encrypted data (computational time of 41.72 ms) is 80 times slower than the KNN on plaintext (computational time of 0.518 ms). The main computation time for CKKS FHE schemes is 41.72 ms. In this work, the single-channel CKKS-KNN is implemented in FPGA. The resource utilization and computational time of a single channel were carried out. However, we proposed a 32-channel CKKS-KNN which can be implemented in a larger FPGA available from Intel. With the proposed architecture of 32-channel parallel encryption hardware and achieving 300 MHz speed, we could make the computational time of CKKS-based KNN 0.85 ms. Resource utilization and speed are easily achieved in Intel Agilex FPGA of higher density.

After extensive simulation in Python and implementation in FPGA, it is evident that, with the proposed architecture with 32 parallel channels and enhancing the clock performance from 196 MHz to 300 MHz, i.e., by 1.5 times, the computational time has been

brought down to 1/40 of CKKS-KNN (0.85 ms) on encrypted data to a realistic value in the order of the KNN classification algorithm over plaintext (0.518 ms).

9. Future Work

In future work, it is planned to develop new architectures of FHE encryption to implement in Intel FPGA for different machine and deep learning algorithms. Furthermore, a demonstration of the FHE framework for cloud and edge computation on Intel Agilex Dev. Board is planned. As the FHE is a recent field, it has tremendous potential for theoretical research to develop new algorithms. Our future work will be dedicated to developing new hardware architectures for new FHE schemes and to improve the computation speed and implementation in re-configurable hardware such as FPGA.

Author Contributions: Conceptualization, methodology, software, validation, formal analysis, investigation, resources, data curation, writing, original draft preparation, S.B.; writing, review, and editing, J.R.P. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: In this research work, a breast cancer dataset is used. Here is the publicly available link: https://www.kaggle.com/datasets/uciml/breast-cancer-wisconsin-data?resource=download (accessed on 20 February 2024).

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Gentry, C. A Fully Homomorphic Encryption Scheme. Ph.D. Thesis, Stanford University. Available online: https://crypto.stanford.edu/craig/ (accessed on 20 February 2024).
2. Fan, J.; Vercauteren, F. Somewhat Practical Fully Homomorphic Encryption. *Cryptology ePrint Archive, Paper 2012/144*, 2012. Available online: https://eprint.iacr.org/2012/144 (accessed on 20 February 2024).
3. Zvika, B.; Craig, G.; Vinod, V. (Leveled) fully homomorphic encryption without bootstrapping. *ACM Trans. Comput. Theory (TOCT)* **2014**, *6*, 1–36.
4. Ilaria, C.; Nicolas, G.; Mariya, G.; Malika, I. TFHE: Fast fully homomorphic encryption over the torus. *J. Cryptol.* **2020**, *33*, 34–91.
5. Zvika, B.; Vinod, V. Efficient fully homomorphic encryption from (standard) lwe. *SIAM J. Comput.* **2014**, *43*, 831–871.
6. Zvika, B. Fully homomorphic encryption without modulus switching from classical GapSVP. In Proceedings of the Advances in Cryptology-CRYPTO 2012, 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, 19–23 August 2012; pp. 868–886.
7. Zvika, B.; Vinod, V. Lattice-based FHE as secure as PKE. In Proceedings of the 5th Conference on Innovations in Theoretical Computer Science, Princeton, NJ, USA, 12–14 January 2014; pp. 1–12.
8. Zvika, B.; Vinod, V. Fully homomorphic encryption from ring-LWE and security for key dependent messages. In Proceedings of the Annual Cryptology Conference, Santa Barbara, CA, USA, 14–18 August 2011; pp. 505–524.
9. Majedah, A.; Liu, H.; Washington, C. Homomorphic encryption algorithms and schemes for secure computations in the cloud. In Proceedings of the 2016 International Conference on Secure Computing and Technology, Washington, DC, USA, 4–5 November 2016.
10. Craig, G. Fully homomorphic encryption using ideal lattices. In Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing, Bethesda, MD, USA, 31 May–2 June 2009; pp. 169–178.
11. Craig, G.; Halevi, S.; Smart, N.P. Better bootstrapping in fully homomorphic encryption. In Proceedings of the International Workshop on Public Key Cryptography, Berlin/Heidelberg, Germany, 21 May 2012; pp. 1–16.
12. Mohsin, M.A.; Darshika, G.P. An FPGA-based hardware accelerator for K-nearest neighbor classification for machine learning on mobile devices. In Proceedings of the 9th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies, Toronto, ON, Canada, 20–22 June 2018; pp. 1–7.
13. Abedalmuhdi, A.; Ayyad, W.R.; Jarrah, A. Optimized implementation of an improved KNN classification algorithm using Intel FPGA platform: Covid-19 case study. *J. King Saud Univ.-Comput. Inf. Sci.* **2022**, *34*, 3815–3827.
14. David, M.; Luis, C.; Neil, G. A modified KNN algorithm for high-performance computing on FPGA of real-time m-qam demodulators. *Electronics* **2021**, *10*, 627.
15. Sagarika, B.; Rani, P.J. Design of Novel Hardware Architecture for Fully Homomorphic Encryption Algorithms in FPGA for Real-Time Data in Cloud Computing. *IEEE Access* **2022**, *10*, 131406–131418.
16. Behera, S.; Prathuri, J.R. FPGA-Based Design Architecture for Fast LWE Fully Homomorphic Encryption. In Proceedings of the Cyber Security and Digital Forensics: Proceedings of ICCSDF 2021, Springer, The NorthCap University, Gurugram, Haryana, India, 3–4 April 2021; pp. 575–584.

17. Agrawal, R.; de Castro, L.; Yang, G.; Juvekar, C.; Yazicigil, R.; Chandrakasan, A.; Vaikuntanathan, V.; Joshi, A. FAB: An FPGA-based accelerator for bootstrappable fully homomorphic encryption. In Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA), Montreal, QC, Canada, 25 February–1 March 2023; pp. 882–895.
18. Lee, J.W.; Kang, H.; Lee, Y.; Choi, W.; Eom, J.; Deryabin, M.; Lee, E.; Lee, J.; Yoo, D.; Kim, Y.S.; et al. Privacy-preserving machine learning with fully homomorphic encryption for deep neural network. *IEEE Access* **2022**, *10*, 30039–30054.
19. Yogachandran, R. Privacy-preserving similarity calculation of speaker features using fully homomorphic encryption. *arXiv* **2022**, arXiv:2202.07994.
20. Al Badawi, A.; Louie, H.; Fook, M.C.; Kim, L.; Mi, A.K.M. Privft: Private and fast text classification with homomorphic encryption. *IEEE Access* **2020**, *8*, 226544–226556.
21. Behera, S.; Prathuri, J.R. Application of homomorphic encryption in machine learning. In Proceedings of the 2020 2nd Ph.D. Colloquium on Ethically Driven Innovation and Technology for Society (Ph.D. EDITS), IEEE, Bangalore, India, 8 November 2020; pp. 1–2.
22. Haokun, F.; Quan, Q. Privacy-preserving machine learning with homomorphic encryption and federated learning. *Future Int.* **2021**, *13*, 94.
23. Behera, S.; Rekha, B.; Pandey, P.; Vidya, B.; Prathuri, J.R. Preserving the Privacy of Medical Data using Homomorphic Encryption and Prediction of Heart Disease using K-Nearest Neighbor. In Proceedings of the 2022 IEEE International Conference on Data Science and Information System (ICDSIS), IEEE, Malnad College of Engineering, Hassan, India, 29–30 July 2022; pp. 1–6.
24. Nikola, S. Making Computation on Encrypted Data Practical through Hardware Acceleration of Fully Homomorphic Encryption. Ph.D. Thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2022.
25. Nikola, S.; Axel, F.; Aleksandar, K.; Srinivas, D.; Ronald, D.; Christopher, P.; Daniel, S. F1: A fast and programmable accelerator for fully homomorphic encryption. In Proceedings of the MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture, Athens, Greece, 18–22 October 2021; pp. 238–252.
26. Sadegh, R.M.; Kim, L.; Blake, P.; Wei, D. HEAX: An architecture for computing on encrypted data. In Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 16–20 March 2020; pp. 1295–1309.
27. Lei, J.; Qian, L.; Nrushad, J. Matcha: A fast and energy-efficient accelerator for fully homomorphic encryption over the torus. In Proceedings of the 59th ACM, IEEE Design Automation Conference, San Francisco, CA, USA, 10–14 July 2022; pp. 235–240.
28. Can, M.A.; Sunmin, K.; Youngsam, S.; Donghoon, Y.; Yongwoo, L.; Sinha, R.S. Medha: Microcoded Hardware Accelerator for Computing on Encrypted Data. *Cryptology ePrint Archive, Paper 2022/480*. Available online: https://eprint.iacr.org/2022/480 (accessed on 20 February 2024).
29. Tian, Y.; Kuppannagari, S.R.; Kannan, R.; Prasanna, V.K. Performance modeling and FPGA acceleration of homomorphic encrypted convolution. In Proceedings of the 2021 31st International Conference on Field-Programmable Logic and Applications (FPL), IEEE, Dresden, Germany, 30 August–3 September 2021; pp. 115–121.
30. Cao, X.; Moore, C.; O'Neill, M.; O'Sullivan, E.; Hanley, N. Accelerating fully homomorphic encryption over the integers with super-size hardware multiplier and modular reduction. *Cryptol. Eprint Arch.* **2013**.
31. Sinha, R.S.; Turan, F.; Jarvinen, K.; Vercauteren, F.; Verbauwhede, I. FPGA-based high-performance parallel architecture for homomorphic computing on encrypted data. In Proceedings of the 2019 IEEE International Symposium on High-Performance Computer Architecture (HPCA), Washington, DC, USA, 16–20 February 2019; pp. 387–398.
32. Cheon, J.H.; Kim, A.; Kim, M.; Song, Y. Homomorphic encryption for the arithmetic of approximate numbers. In Proceedings of the Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, 3–7 December 2017; pp. 409–437.
33. Su, Y.; Yang, B.; Yang, C.; Tian, L. Fpga-based hardware accelerator for leveled ring-lwe fully homomorphic encryption. *IEEE Access* **2020**, *8*, 168008–168025.
34. Cheon, J.H.; Han, K.; Kim, A.; Kim, M.; Song, Y. A full RNS variant of approximate homomorphic encryption. In Proceedings of the Selected Areas in Cryptography–SAC 2018: 25th International Conference, Calgary, AB, Canada, 15–17 August 2018; Springer: Cham, Switzerland, 2019; pp. 347–368.
35. Lee, E.; Lee, J.W.; Kim, Y.S.; No, J.S. Optimization of homomorphic comparison algorithm on rns-ckks scheme. *IEEE Access* **2022**, *10*, 26163–26176.
36. Vadim, L.; Chris, P.; Oded, R. On ideal lattices and learning with errors over rings. *J. ACM (JACM)* **2013**, *60*, 1–35.
37. Colin, R.G. 13 Computation using the QR decomposition. *Handb. Stat.* **1993**, *9*, 467–508.
38. Peters, H.; Schulz-Hildebrandt, O.; Luttenberger, N. A novel sorting algorithm for many-core architectures based on adaptive bitonic sort. In Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium, IEEE, Shanghai, China, 21–25 May 2012; pp. 227–237.
39. Nguyen, T.T.; Kim, J.; Lee, H. CKKS-Based Homomorphic Encryption Architecture using Parallel NTT Multiplier. In Proceedings of the 2023 IEEE International Symposium on Circuits and Systems (ISCAS), IEEE, Monterey, CA, USA, 21–25 May 2023; pp. 1–4.
40. Lee, J.; Duong, P.N.; Lee, H. Configurable Encryption and Decryption Architectures for CKKS-Based Homomorphic Encryption. *Sensors* **2023**, *23*, 7389. [CrossRef]

41. Stefano, D.M.; Lo, G.M.; Sergio, S. VLSI Design and FPGA Implementation of an NTT Hardware Accelerator for Homomorphic SEAL-Embedded Library. *IEEE Access* **2023**, *11*, 72498–72508.
42. Su, Y.; Yang, B.L.; Yang, C.; Yang, Z.P.; Liu, Y.W. A highly unified reconfigurable multicore architecture to speed up NTT/INTT for homomorphic polynomial multiplication. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2022**, *30*, 993–1006.