AN ACCELERATED GPU LIBRARY FOR EFFICIENT HOMOMORPHIC ENCRYPTION OPERATIONS IN THE CKKS SCHEME

by
Enes Recep Türkoğlu

Submitted to
the Faculty of Engineering and Natural Sciences
in partial fulfillment of the requirements for the degree of
Master of Science

Sabancı Üniversitesi
İstanbul, Türkiye
October 2023

ABSTRACT

AN ACCELERATED GPU LIBRARY FOR EFFICIENT HOMOMORPHIC ENCRYPTION OPERATIONS IN THE CKKS SCHEME

ENES RECEP TÜRKOĞLU

ELECTRONICS ENGINEERING MSC. THESIS, OCTOBER 2023

Thesis Advisor: Prof.Erkay Savaş

Keywords: Homomorphic encryption, Secure computation, Lattice-based cryptography, Parallel processing, Accelerator

Homomorphic Encryption is an encryption method that enables secure computation on encrypted data. Among the many homomorphic encryption schemes developed today, one of the most popular is the CKKS scheme, which stands for the Cheon-Kim-Kim-Song scheme. This scheme supports both fully homomorphic encryption (FHE) and somewhat homomorphic encryption (SWHE). One of its most significant advantages is its capability to handle real numbers, making it highly suitable for applications requiring precise calculations and complex operations. This thesis elucidates the fundamental operations of the CKKS scheme, including homomorphic addition, multiplication, linearization, rescale, and rotation, as well as their practical applications.

One of the primary reasons for selecting the CKKS scheme can be attributed to its adeptness in managing real-number computations while mitigating noise accumulation, thus facilitating a wide array of operations. Moreover, it enables homomorphic evaluations of deep learning models, making it highly significant in modern cryptographic applications. The Microsoft Simple Encrypted Arithmetic Library (SEAL) is referenced to efficiently utilize the CKKS scheme. This thesis enhances the performance of the fundamental CKKS operations by leveraging the Graphics Processing Unit (GPU) to further optimize the CPU implementation in this library.

The GPU is preferred due to its computational density in homomorphic encryption operations and its parallelization ability. Since homomorphic encryption algorithms require

intense mathematical operations and extensive computations involving large numbers, the architecture of GPUs, which consists of thousands of cores, enables these algorithms to be efficiently executed in parallel.

The experimental evaluations on target GPUs were carried out on the RTX 3070 and RTX 4090. Once again, a powerful CPU, the AMD RYZEN 7 3800X, was employed for a fair comparison. According to the SEAL library, the results revealed acceleration by a factor of up to 105.04 in homomorphic addition, 246.65 in homomorphic multiplication, 161.07 in relinearization, 113 in rescale, and 121.1 in rotation. These values were obtained with a ring size of 2^{15} and a modulus bit size of 881. Furthermore, we designed different circuits with various multiplicative depths and implemented our GPU functions into these circuits. We achieved a 27.81-fold speedup compared to CPU implementation in a 2^{13} ring size and 218-bit modulus. To the best of our knowledge, this is the first work in the literature where a GPU library is used for different multiplicative depth circuits. Our findings underscore the significant practical impact of homomorphic encryption algorithms when leveraging GPU resources for real-world deployment.

ÖZET

CKKS ŞEMASININ HOMOMORFİK ŞİFRELEME İŞLEMLERİ İÇİN HIZLANDIRILMIŞ VERİMLİ GPU KÜTÜPHANESİ

ENES RECEP TÜRKOĞLU

ELEKTRONİK MÜHENDİSLİĞİ YÜKSEK LİSANS TEZİ, EKIM 2023

Tez Danışmanı: Dr. Öğr. Üyesi Erkay Savaş

Anahtar Kelimeler: Homomorfik şifreleme, Güvenli hesaplama, Kafes-tabanlı kriptografi, Eşzamanlı işleme, Hızlandırıcı

Homomorfik Şifreleme, şifrelenmiş veriler üzerinde güvenli bir şekilde hesaplama yapmayı sağlayan bir şifreleme metodudur. Günümüzde oluşturulmuş birçok homomorfik şifreleme şemaları arasından en popüler olanından biri de hiç şüphesiz Cheon-Kim-Kim-Song (CKKS) şemasıdır. Bu şema hem tamamen homomorfik şifreleme (FHE) hem de kısmen homomorfik şifreleme (SWHE) için destek sunmaktadır. Bu şemanın en önemli ayrıcalıklarından biri gerçel sayı işleyebilme kabiliyetidir. Bu sayede, hassas hesaplamalar ve karmaşık işlemler gerektiren uygulamaların kullanımı için oldukça uygundur. Bu tezde CKKS şemasının temel işlemleri olan homomorfik toplama, homomorfik çarpma, yeniden doğrusallaştırma, yeniden ölçeklendirme ve döndürme operasyonları nının GPU ile daha verimli ve hızlı bir şekilde gerçeklenebileceğini gösteriyoruz.

Gerçel sayı hesaplamalarını etkin bir şekilde yönetebilmesi ve minimal gürültü artışı ile çeşitli işlemleri desteklemesi CKKS şemasının tercih edilmesindeki başlıca sebeplerdir. Dahası, derin öğrenme modellerinin homomorfik değerlendirmelerini mümkün kılması, CKKS şemasını modern kriptografik uygulamalarda oldukça önemli hale getirmektedir. CKKS şemasını verimli bir şekilde kullanmak için Microsoft Simple Encrypted Arithmetic Library (SEAL) adlı kütüphane referans alınmıştır. Bu kütüphanedeki CPU uygulumasını daha da optimize etmek için Grafik İşlem Birimi'nden (GPU'dan) faydalanarak operasyonlar hızlandırılmıştır.

GPU'nun tercih edilme nedeni, homomorfik şifreleme işlemlerinin hesaplama yoğunluğu ve bu işlemleri paralelleştirebilme yeteneğidir. Homomorfik şifreleme algoritmaları, büyük sayılar ile yoğun matematiksel işlemler içermektedir ve GPU'ların, binlerce çekirdekten

oluşan mimarileri bu algoritmaların paralel olarak verimli bir sekilde yürütülmesini sağlar.

Bu tezdeki GPU uygulamaları RTX 3070 ve RTX 4090 ile yapılmıştır. Adil bir kıyaslama olması için yine güçlü bir CPU olan AMD RYZEN 7 3800X işlemci kullanılmıştır. Elde edilen sonuçlarda SEAL kütüphanesine göre homomorfik toplama işleminde 105.04 kata kadar, homomorfik çarpma işleminde 246.65 kata kadar, yeniden doğrusallaştırma işleminde 161.07 kata kadar, yeniden ölçeklendirme işleminde 113 kata kadar ve döndürme işleminde ise 121.1 kata kadar hızlandırma elde edildiği gözlemlenmiştir. Bu değerler halka boyutu 2^{15} ve modülüs bit boyutu 881 iken elde edilmiştir. Ayrıca, farklı çarpan derinliklerine sahip farklı devreler tasarladık ve GPU işlevlerimizi bu devrelere entegre ettik. 2^{13} halka boyutu ve 218 bit modülüs için CPU uygulaması ile karşılaştırıldığında 27.81 kata kadar bir hız artışı elde ettik. Bildiğimize kadarıyla, bu çalışma, GPU kütüphanesinin farklı çarpan derinliği devreleri için kullanıldığı literatürdeki ilk çalışmadır. Sonuçlarımız göstermektedir ki homomorfik şifreleme algoritmaları GPU kullanarak gerçek hayata konuşlandırılmasında büyük katkı sağlamaktadır.

ACKNOWLEDGMENT

I wish to extend my deepest appreciation to my esteemed advisor, Professor Erkay Savaş, for his unwavering support and invaluable guidance during my tenure as a graduate student at Sabanci University.

I am also profoundly grateful to Dr. Erdinç Öztürk and Professor Yaşar Gürbüz for their invaluable assistance and encouragement.

I would like to express my sincere appreciation to my colleagues at the Sabancı University Cryptography and Information Security Group (CISEC), namely Ali Şah Özcan, Dr. Ahmet Can Mert, Can Ayduman, and Kemal Derya. Their dedication to fostering a conducive and supportive working atmosphere greatly enriched my experience as a graduate student and contributed significantly to my growth as a researcher.

I extend my sincere gratitude to my friends, Oğuz Akat, Ege Akdeniz, Emre Tan, Atakan Dinç, Bartu Göncüler, Ege Keçelioğlu, Orhun Ayyıldız, Alperen Erbay, Yusuf Şener, Ahmet Talha Cengiz, Ataol Kalaycı, Ege Arar and my roommate, Ali Osman Berk Şapcı. Their patience in enduring my numerous discussions and unwavering encouragement in the pursuit of my aspirations are deeply appreciated.

Lastly, I wish to convey my deep appreciation to my parents, İlker and Hacer, and my beloved sister, Esra. Their unwavering love and support have been my constant motivation throughout this journey. Their sacrifices and encouragement have been instrumental in my achievements. I also extend my gratitude to my late grandfather, Fethi Türkoğlu, who instilled in me a love for knowledge by introducing me to numerous bookstores during my formative years. Furthermore, I am grateful to my uncle, Mustafa Başar, for his support during my undergraduate studies.

This thesis is partially supported by the European Union's Horizon Europe research and innovation program under grant agreement No: 101079319 and by TUBITAK under Grant Number 118E72.

Dedicated to
my parents IT & HT, my sister ET.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

List of Algorithms

LIST OF ABBREVIATIONS

ASIC Application-specific integrated circuit

DDR Double data rate

DRAM Dynamic random access memory

CKKS Cheon-Kim-Kim-Song

CPU Central process unit

CT Cooley-Tukey

CRT Chinese Remainder Theorem

CUDA Computing unified device architecture

DFT Discrete Fourier Transform

DIF decimation-in-frequency

DIT decimation-in-time

FPGA Field-programmable gate array

GDDR Graphics double data rate

GPU Graphics Processing Unit

GS Gentleman-Sande

HE Homomorphic Encryption

MIMD Multiple instruction, multiple data

NTT Number Theoretic Transform

RNS Residue Number System

SEAL Simple Encrypted Arithmetic Library

1. INTRODUCTION

Nowadays, the global population's internet access has surpassed a staggering 5.3 billion individuals, as reported by the International Communication Union (ITU, 2022). This extensive development of internet usage has led to an unprecedented accumulation of personal data. The significance of personal data in the modern world cannot be overstated. Both private enterprises and governmental agencies leverage this wealth of information to enhance their operations. Governments harness such data to streamline bureaucratic processes through e-government initiatives, while private companies predominantly employ it for marketing. This presence of digital information, often called big data, has evolved into a cornerstone of modern society, steering the course of technology and influencing various domains, from business to governance.

However, the development of these vast volumes of data demands a paramount concern for security and privacy. The digital processing and storage of data render it susceptible to various malicious attacks. In fact, reports show that over 80 percent of U.S. companies have fallen victim to cyberattacks with the intent of compromising the confidentiality, integrity, and availability of critical data (FUQUA, 2023). Therefore, to ensure both high-security levels and expedient computation within cloud services, it is essential for these applications to incorporate strong cryptographic algorithms.

Among the prominent cryptographic algorithms are Rivest–Shamir–Adleman (RSA) (Rivest et al., 1978) and elliptic curve cryptography (ECC), both of which belong to the realm of public key cryptography. They rely on hard mathematical problems, whose solutions remain beyond the reach of contemporary computing devices with constrained computational capabilities, defying solutions within polynomial time frames (Bernstein and Lange, 2017). Consequently, these cryptographic schemes offer

robust security levels for Internet applications. Nonetheless, quantum computers are anticipated to possess sufficient power to break these schemes, chiefly through Shor's algorithm (Shor, 1994). According to Bernstein and Lange, there exists a race against time to deploy post-quantum cryptography before the advent of quantum computers (Bernstein and Lange, 2017). One of the most promising approaches for resistance against quantum computer attacks rests in lattice-based cryptography, on which fully homomorphic encryption schemes are also based.

Homomorphic encryption enables us to perform computation over encrypted data. Therefore, cloud servers can process any privacy-sensitive data without decrypting it. In the work by Gentry (2009) the first functional fully homomorphic encryption scheme was introduced. This scheme, based on ideal lattices, allows the homomorphic evaluation of arbitrary circuits. Subsequently, Gentry and Halevi (Gentry and Halevi, 2011) achieved the first practical implementation of a fully homomorphic encryption (FHE) scheme. Later developments introduced more practical schemes based on ring learning with error (RLWE) problems (Lyubashevsky et al., 2013). In these schemes, plaintext and ciphertext are represented as polynomials, with ciphertext containing a "noise" that accumulates through homomorphic operations.

The main idea is to manage this noise through the adjustment of ring size and modulus, essentially creating a noise budget. When the noise surpasses this budget, further homomorphic operations are unfeasible, as the ciphertext message cannot be decrypted. These schemes fall under the category of somewhat homomorphic encryption (SHE). To enable a higher degree of homomorphic operations, Gentry's work Gentry (2009) presented a solution using a technique known as "bootstrapping", a process that homomorphically decrypts the ciphertext to refresh the noise budget. This advanced technique is known as fully homomorphic encryption (FHE).

In the last decade, numerous schemes for homomorphic encryption have emerged. These include Cheon-Kim-Kim-Song (CKKS) (Cheon et al., 2017), Ducas and Micciancio (DM) (Ducas and Micciancio, 2015), Torus FHE (TFHE) (Chillotti et al., 2018), Brakerski/Fan-Vercauteren (BFV) (Fan and Vercauteren, 2012), and Brakerski-Gentry-Vaikuntanathan (BGV) (Brakerski et al., 2012). Among these, the CKKS scheme stands out as one of the most promising approaches. Additionally, there are several software libraries available, such as SEAL (SEAL, 2020), PALISADE (PAL, 2021), and HELib (Halevi and Shoup, 2014).

Due to the computationally expensive nature of homomorphic encryption operations, it has been considered impractical for large-scale cloud applications. To enhance the

efficiency of these schemes, accelerators such as GPUs, FPGAs, and ASICs have been utilized (Wang et al., 2014; Mert et al., 2020; Doröz et al., 2015). In this study, we present algorithms and implementation techniques for accelerating the CKKS scheme within the SEAL library using NVIDIA GPUs. This implementation is developed within the framework of the Compute Unified Device Architecture (CUDA) (NVIDIA Corporation, 2010) programming model, aiming to accelerate CKKS scheme operations through various parallelization strategies that take into account GPU architectures. To the best of our knowledge, this work represents the first GPU library capable of executing the complete set of CKKS scheme operations, including homomorphic addition, multiplication, rescaling, relinearization, and rotation operations. Furthermore, our GPU implementation's timing results surpass those of other state-of-the-art implementations in the literature. We also designed various circuits to simulate real-world applications with varying levels of multiplicative depth, utilizing our GPU library functions. Our results demonstrate that, particularly in the case of large ring sizes, our implementation can effectively execute homomorphic operations in complex circuits with high multiplicative depths without requiring bootstrapping.

The thesis is organized as follows:

- In Chapter 2, we expound on the notation used throughout this thesis and present foundational mathematical background information, including Fixed-Point Arithmetic, Barrett Reduction, Number Theoretic Transform, Homomorphic Encryption, and the CKKS Scheme.
- GPU architecture is detailed in Chapter 3, which covers the history of GPUs, hardware fundamentals, programming models, GPU memory organization, and hierarchy.
- Chapter 4 elaborates on our implementation techniques for the reduction algorithm, NTT, and CKKS operations in detail.
- Chapter 5 presents timing results and comparisons with state-of-the-art works of our GPU implementations and our circuit designs.
- We present our conclusions and discuss future directions in Chapter 6.

2. BACKGROUND

In this section, we commence by introducing the mathematical notation that will be consistently employed throughout this thesis. Following that, we present the technical and arithmetic fundamentals that are crucial for comprehending the content within this work.

This thesis is centered on the efficient GPU implementation of homomorphic encryption operations using the CKKS scheme. To facilitate an understanding of the CKKS scheme, we provide a detailed explanation of fixed-point arithmetic immediately after the notation section. Subsequently, we delve into key components, including the residue number system, Barrett reduction, and the number theoretic transform. These algorithms serve as the essential building blocks for homomorphic encryption. Finally, we expound on the core concepts of homomorphic encryption, with a particular focus on the CKKS scheme.

2.1 Notation

In this thesis, we employed the CKKS scheme to perform SHE operations based on the Ring Learning with Errors (RLWE) problem. The difficulty of this problem forms the fundamental security assumption for various post-quantum cryptography and homomorphic encryption algorithms. The RLWE problem, more precisely, the Learning with Errors problem over rings, is a more efficient and practical variant of the Learning with Errors (LWE) problem. It specializes in working with polynomial rings over finite fields. The

details are as follows:

The CKKS scheme utilizes the polynomial ring $\mathbf{R}_q = \mathbb{Z}_q/f(x)$, where $\mathbb{Z}_q$ signifies the finite ring $\{0,1,...,q-1\}$, in which arithmetic is conducted modulo q. Here, $f(x)$ represents a monic irreducible polynomial of degree n. Specifically, $f(x)$ takes the form of a cyclotomic polynomial, $\phi_M(x)$, with $M = 2^d$, and $f(x) = x^n + 1$, with $n = \frac{M}{2} \in \mathbb{Z}^+$, which is referred to as the ring dimension. Thus, $\mathbf{R}$ denotes the set of polynomials with degrees less than n and integer coefficients. In some contexts, n is recognized as the dimension of $\mathbf{R}_q$, and we use the notation $\mathbf{R}_{q,n}$ to signify its dimension.

The notation $[a]_q$, for $q > 1$, signifies the set of integers to which a belongs, ranging within the interval $[-q/2, q/2]$, while $|a|_q$ constrains a to the range of $[0, q-1]$. A polynomial $a(x) \in \mathbf{R}_q$ can be regarded as a vector composed of n integers in $\mathbb{Z}_q$, represented by its coefficients. Upon subjecting the vector of $a(x)$ to the Number Theoretic Transform (NTT), a discrete Fourier transformation variant applied over rings $\mathbb{Z}_q$, it produces a vector of the same dimension denoted as $\bar{a}(x)$, often simply referred to as $\bar{a}$.

The symbols $+$, $-$, and $\times$ (or $\cdot$) signify addition, subtraction, and multiplication, either in $\mathbb{Z}_q$ or $\mathbf{R}_q$. Conversely, the symbol $\odot$ represents modular pointwise multiplication for vector representations of elements within $\mathbf{R}_q$ within the NTT domain. In this context, it denotes the multiplication of an element in a vector by the corresponding element in another vector, with these multiplications being executed within $\mathbb{Z}_q$, essentially a modulo q multiplication.

Here, λ is the security parameter, denoted in unary notation, while $a \leftarrow \mathbb{S}$ signifies the uniform sampling of a from the set $\mathbb{S}$. For the purpose of sampling coefficients for error polynomials, χ_{err} is employed as a truncated zero-mean discrete Gaussian distribution. The error characterizes the distribution's bounds, β_{err}, and standard deviation σ.

We can now provide a comprehensive and simplified definition of the RLWE problem. Suppose having a drawn from the ring $\mathbf{R}_q$, and the elements s and e, which form the secret and error, both sourced from $\mathbf{R}$ with coefficients sampled according to χ_{err}. Additionally, suppose we have $b = as + e$. The 'search' RLWE problem is, in essence, the challenge of making it difficult to determine s when given a and b. In an HE scheme, s functions as the secret key, while (b, a) serves as the public key.

2.2 Fixed-Point Arithmetic

The CKKS scheme relies on Fixed-Point Arithmetic; thus, a decent understanding of Fixed-Point Arithmetic is necessary. This arithmetic relies on the representation of numbers using fixed number of digits in their fractitonal part. Assuming an imaginary decimal point is fixed at a specified position, its location must not change throughout the entire computation to maintain numerical consistency and precision. This constraint ensures that the interpretation of numbers remains consistent, preventing unexpected variations in the scale or magnitude of the results. Stabilization via rescaling involves a process, where the imaginary decimal point's position is adjusted as needed during computation to maintain numerical stability and avoid excessive growth in the data. This adjustment, known as rescaling, is crucial for preventing overflow and maintaining the fixed-point representation's integrity. As an example of real numbers representation in fixed-point arithmetic: We denote the scaling factor as Δ where $\Delta = 2^{precision}$, this precision generally corresponds to RNS modulus size. In order to represent a real number (called r) in fixed-point arithmetic, $z = \text{RealToFP}(r) = \text{truncate}(\Delta \times r)$ for the reverse, $r = \text{FPToReal}(z) = \frac{z}{\Delta}$. In the subsequent paragraphs, we will use the aforementioned conversion primitives to explain CKKS addition, subtraction, and multiplication operations.

2.2.1 Addition and Subtraction

To perform a CKKS addition or subtraction both fixed-point numbers should be on the same scale. Here are examples of addition and subtraction for two fixed-point numbers, denoted as, FPADD and FPSUB respectively. Where $\text{FPADD}(z_1 = |\Delta \cdot r_1|, z_2 = |\Delta \cdot r_2|) = z_1 + z_2 = \Delta \cdot |r_1 + r_2|$ and, $\text{FPSUB}(z_1 = |\Delta \cdot r|, z_2 = |\Delta \cdot r_2|) = z_1 - z_2 = \Delta \cdot |(r_1 - r_2)|$.

2.2.2 Multiplication

The multiplication operation differs somewhat from addition and subtraction. Here is an example of multiplication for two fixed-point numbers, denoted as FPMUL:

$\text{FPMUL}(z_1 = |\Delta \cdot r_1|, z_2 = |\Delta \cdot r_2|) = \frac{1}{\Delta} \cdot (z_1 \cdot z_2) = \frac{1}{\Delta} \cdot (\Delta^2 |r_1 \cdot r_2|) = \Delta \cdot |r_1 \cdot r_2|$

The rescaling factor, represented by $\frac{1}{\Delta}$, plays a crucial role by ensuring that the multiplication operation results in low noise levels in the output, thus preserving the integrity of the computation.

2.3 Residue Number System

The Residue Number System (Garner, 1959) enables secure and efficient computation of encrypted data in Homomorphic Encryption (HE) methods. The RNS has a significant positive impact on HE systems with high coefficient moduli Q because it reduces the computational cost of multi-precision integer arithmetic into single-precision arithmetic. Arithmetic operations on ciphertexts can be carried out in parallel using the RNS, which reduces calculation time and resource usage.

Due to practicality and enhancing efficiency, the RNS has been leveraged in several existing HE schemes such as the CKKS scheme, within the context of privacy-preserving computation (Al Badawi et al., 2019), private information retrieval, and secure multiparty computation (Cortés-Mendoza et al., 2021).

In the context of modular arithmetic, given an integer X such that $X < M$, it can be represented using residues x_i, where $x_i = X \bmod m_i$ for $i = 1, ..., l$. Here, the m_i's constitute a group of pairwise relatively prime integers known as the moduli or base, often denoted as $[X]_{m_i} = X \bmod m_i$.

It is worth noting that in this context, we consider the product of these moduli, which is defined as $M = \prod_{i=1}^{l} m_i$. The technique we employ to combine these modular residues into a single representation, denoted as $[X]_M$, is the Chinese Remainder Theorem (CRT). The CRT is a fundamental theorem in number theory, and its application allows us to express $[X]_M$ as:

$$[X]_M = \left[\sum_{i=1}^{r} \left[x_i \cdot (\tfrac{M}{m_i})^{-1} \right]_{m_i} \cdot \tfrac{M}{m_i} \right]_M,$$

In essence, the Residue Number System (RNS) provides a powerful solution to alleviate the computationally intensive tasks involved in homomorphic computations. By leveraging the Chinese Remainder Theorem and employing the RNS representation, cryptographic algorithms can effectively process large integers through parallelism and reduction in computational complexity. This not only significantly enhances computational efficiency but also contributes to the security of homomorphic encryption and related protocols (Al Badawi et al., 2019).

2.4 Barrett Reduction

Modular multiplication operations are fundamental in the RNS (Residue Number System) form of homomorphic cryptographic methods, as exemplified by (Bajard et al., 2016), and often consume a significant portion of execution time in comparison to other operations.

The Barrett reduction (Barrett, 1986) and the Montgomery reduction (Montgomery, 1985) are widely adopted methods for modular reduction operations. Both techniques perform effectively. However, in this case, the Barrett reduction is preferred due to its simplicity, in contrast to the Montgomery reduction, which requires an additional step to convert integers to the Montgomery domain. This additional step involves a multiplication by R (where $R \in \mathbb{N}$, $R > q$, and $\gcd(R, n) = 1$) to transition into the Montgomery domain and another multiplication by R^{-1} to return to the polynomial domain.

As outlined in the inputs of Algorithm 1, there are some precomputed values, denoted as μ, which are used to calculate the reciprocal of the modulus q with the required level of accuracy. Although this precomputation process is time-consuming, it necessitates execution only once.

The method performs a right shift operation on the input integer C by $(k-2)$ bits, as shown by Step 1, $C_1 = \texttt{Right\_shift}(C, (k-2))$. This operation leaves the higher-order coefficients unaffected while discarding the bottom $(k-2)$ bits of C. Next, in Step 2, it multiplies the output of the right shift operation C_1 by the precomputed value of μ. This

stage is represented by the expression $C_2 = C_1 \cdot \mu$ and aims to calculate the quotient of C divided by q.

The result of the multiplication C_2, which has $2k+3$ bits, then undergoes a second right shift operation in Step 3, denoted as $C_3 = \texttt{Right\_shift}(C_2, (k+2))$. This step ensures the accuracy of the intermediate value C_3. The procedure computes the intermediate product $\hat{q} = q \cdot C_3$, representing the multiple of q believed to be the closest to C.

This intermediate product $\hat{q}$ is then subtracted from the original integer C to obtain $C_{out} = (C - \hat{q})$ in lines 4 and 5. This step calculates the difference integer C_{out}, which holds the residual value after dividing C by q.

Finally, the algorithm includes corrective steps (Steps 6-9), the *if-then-else* ensures the C_{out} falls within the valid range $[0, q)$.

Algorithm 1 Barrett Reduction

Input: $C = a \times b$, where $a, b < q$; $k = \lceil log_2(q) \rceil$; $\mu = \lfloor \frac{2^{2k+1}}{q} \rfloor$
Output: C_{out} $(C \mod q)$
1: $C_1 = \texttt{Right\_shift}(C, (k-2))$
2: $C_2 = C_1 \cdot \mu$
3: $C_3 = \texttt{Right\_shift}(C_2, (k+3))$
4: $\hat{q} = q \cdot C_3$
5: $C_{out} = (C - \hat{q})$
6: **if** $C_{out} >= q$ **then** $C_{out} = C_{out} - q$
7: **else** $C_{out} = C_{out}$
8: **end if**

2.5 Number Theoretic Transform

The Number Theoretic Transform (NTT) (Harvey, 2014) is a powerful mathematical technique that plays a crucial role in various computational algorithms, particularly in cryptography. NTT is defined as a Discrete Fourier Transform (DFT) defined on the ring $\mathbb{Z}_q/\phi_m(x)$, where $\phi_m(x)$ is the m-th cyclotomic polynomial, and $m = 2 \cdot n$. A one n-point NTT operation converts a polynomial in the (n − 1) degree polynomial domain $A(x) = \sum_{i=0}^{n-1} a_i x^i$; $(n-1)$, to a polynomial in the NTT domain of degree (n − 1) $\bar{A}(x) =$

$\sum_{i=0}^{n-1} \bar{a}_i x^i$, where

$$\bar{a}_i = \sum_{j=0}^{n-1} a_j \omega^{ij} \in \mathbb{Z}_q \text{ for } i = 0, 1, \ldots, n-1, \tag{2.1}$$

Similarly, the inverse NTT (INTT) operation can be defined as

$$a_i = n^{-1} \sum_{j=0}^{n-1} \bar{a}_j \omega^{-ij} \in \mathbb{Z}_q. \tag{2.2}$$

By comparing Equation 2.1 and 2.2 of NTT and INTT, two differences are notable. Firstly, in the NTT operation (equation 2.1), ω (twiddle factor) is used, whereas in INTT (Equation 2.2), $\omega^{-1} \pmod q$ is utilized. Moreover, at the end of the INTT operation, all coefficients are multiplied by $n^{-1} \pmod q$. The NTT operation uses the n-th root of unity constant $\omega \in \mathbb{Z}_q$, where $\omega^n \equiv 1 \pmod q$, and $\omega^i \neq 1 \pmod q \ \forall i < n$.

Algorithm 2 Merge Forward NTT

Input: $a(x) \in \mathbb{Z}_q[x]/(x^n+1)$ polynomial standard-order
Input: $\Psi_{br}[k] = \Psi^{br(k)}$ (Powers of Ψ stored in bit-reversed order)
Input: $n = 2^l$, q ($q \equiv 1 \mod 2n$)
Output: $\bar{a} \in \mathbb{Z}_q^n$ in bit-reversed order

1: $t = n; \quad m = 1$
2: **repeat**
3: $\quad t = t/2$
4: $\quad$ **for** i from 0 by 1 to m **do**
5: $\quad\quad j_1 = 2it$
6: $\quad\quad j_2 = j_1 + t - 1$
7: $\quad\quad$ **for** j from j_1 by 1 to $j_2 + 1$ **do**
8: $\quad\quad\quad U = a_j$
9: $\quad\quad\quad V = a_{j+t} \cdot \Psi_{br}[m+i] \pmod q$
10: $\quad\quad\quad a_j = U + V \pmod q$
11: $\quad\quad\quad a_{j+t} = U - V \pmod q$
12: $\quad\quad$ **end for**
13: $\quad$ **end for**
14: $\quad m = 2 \times m$
15: **until** $m < n$
16: **for** i from 0 by 1 to n **do**
17: $\quad \bar{a}_i = a_i \pmod q$
18: **end for**

Employing NTT/INTT in their original form (Equation 2.1, 2.2) in a practical setting

Algorithm 3 Merge Inverse NTT

Input: $\bar{a} \in \mathbb{Z}_q^n$ in bit-reversed order
Input: $\Psi_{rev}[k]$ (power of Ψ^{-1} stored in bit-reverse order ($\Psi_{rev}[k] = \Psi^{-br(k)} \pmod q$))
Input: $n = 2^l$, q ($q \equiv 1 \mod 2n$)
Output: $a(x) \in \mathbb{Z}_q[x]/(x^n + 1)$ standard-order

1: $t = 1$; $m = n$
2: **repeat**
3: $j_1 = 0$; $h = m/2$
4: **for** i from 0 by 1 to h **do**
5: $j_2 = j_1 + t - 1$
6: **for** j from j_1 by 1 to $j_2 + 1$ **do**
7: $U = \bar{a}_j$; $V = \bar{a}_{j+t}$
8: $\bar{a}_j = U + V \pmod q$
9: $\bar{a}_{j+t} = (U - V) \cdot \Psi_{rev}[h + i] \pmod q$
10: **end for**
11: $j_1 = j_1 + 2 \times t$
12: **end for**
13: $t = 2 \times t$
14: $m = m/2$
15: **until** $m < n$
16: **for** i from 0 by 1 to n **do**
17: $a_i = (\bar{a}_i \cdot n^{-1}) \pmod q$
18: **end for**

is computationally inefficient. To remediate this, in literature Mert et al. (2019); Scott (2017); Su et al. (2022); Pollard (1971); Dai and Sunar (2016); Chu and George (1999); Feng et al. (2019); Longa and Naehrig (2016), several approaches have been proposed to address this inefficiency. Algorithms 2 and 3 depict two efficient methods to efficiently compute NTT and INTT, respectively, which are used in this thesis as well. In the context of NTT algorithms, a critical component is the butterfly operations which are described in Steps 10-11 of Algorithm 2 and Steps 8-9 of Algorithm 3. Generally, they employ one of the two main butterfly structures called Cooley-Tukey (CT) (Cooley and Tukey, 1965) and Gentleman-Sande (GS) (Chu and George, 1999). Both butterfly structures are suitable for NTT operations. CT-based and GS-based NTT operations are referred to as decimation-in-time (DIT) (Hamood, 2016) and decimation-in-frequency (DIF) NTT operations, respectively. Additionally, NTT operations can be adaptable according to the input and output data order. For example, the input can be in standard order, and the output can be in bit-reversed order, or the input can be in bit-reverse order and the output can be in standard order, and so on. Here, standard order denotes the arrangement of data or elements in their natural sequential order, typically from 0 to $n-1$. Conversely, bit-reversed order entails the reordering of data or elements by reversing their binary

representations.

2.6 Homomorphic Encryption

Homomorphic encryption is a form of encryption that enables computation on encrypted data. (Li et al., 2010; Brakerski and Vaikuntanathan, 2011; Phong et al., 2018; Cheon and Kim, 2015) HE has practical applications in the cloud-computing setting in the scenarios of privacy-preserving computation. As shown in Figure 2.1(a), data is sent to the cloud in an encrypted form. However, the cloud needs to decrypt the data to perform computations, and then encrypt it again before sending it to the customer. In contrast, in homomorphic cloud computing, as illustrated in Figure 2.1(b), data does not need to be decrypted in the cloud. Computations can be performed over encrypted data. This approach enhances security and enables computation over the data while safeguarding personal information because the data is never decrypted, ensuring that the data contents remain confidential and undisclosed. To date, HE algorithms are deemed secure and resistant to quantum attacks due to their proven hardness. There are several techniques developed which provide this hardness. Several techniques have been developed to establish this level of security, with Ring Learning With Errors (RLWE) standing out as a notable example (Nejatollahi et al., 2019). However, a non-trivial trade-off persists between security and computation complexity. Consequently, HE is widely regarded as computationally expensive compared to its counterparts.

Homomorphic encryption (HE) can be applied in real-life scenarios to enhance e-voting security and transparency (Anggriane et al., 2016), secure data storage in the cloud (Potey et al., 2016), perform analysis on sensitive medical information (Munjal and Bhatia, 2022), provide machine learning as a service (Wu et al., 2021), and more.

HE is categorized into three broad categories: (*i*) Partially homomorphic encryption, (*ii*) somewhat homomorphic encryption (SWHE), and (*iii*) fully homomorphic encryption (FHE).

Partially homomorphic encryption facilitates only a limited set of mathematical operations, typically restricted to either addition or multiplication. A significant milestone in the development of fully homomorphic encryption (FHE) was the introduction of the Paillier cryptosystem (Paillier, 1999), which enabled homomorphic addition on encrypted

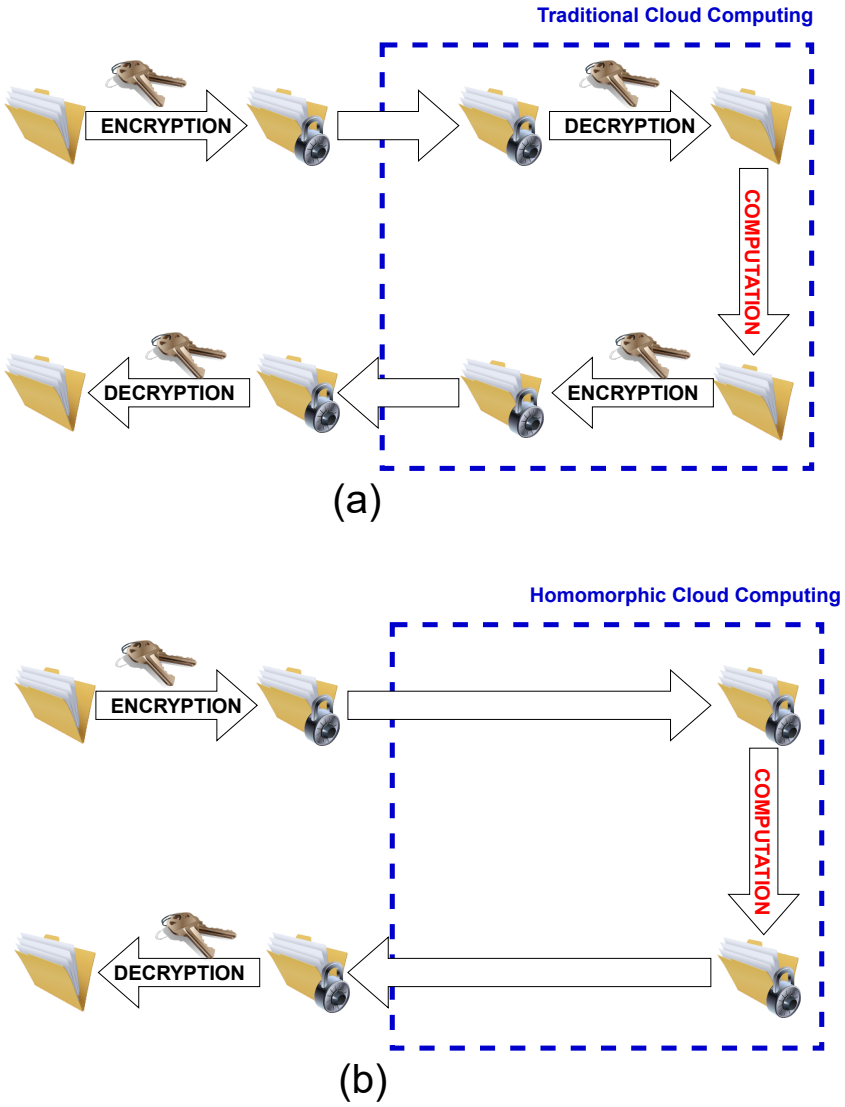


Figure 2.1 Traditional cloud computing (a) and homomorphic cloud computing (b) illustration.

data. Nonetheless, this scheme lacked support for multiplication operations, it can only multiply the ciphertext with a plaintext number, thus classifying it as a partially homomorphic encryption method. The method can perform two ciphertext additions as follows,

$$D_{private}(E_{public}(m_1) \cdot E_{public}(m_2) \; mod \; n^2) = (m_1 + m_2) \; mod \; n$$

SWHE allows for the utilization of homomorphic addition and homomorphic multiplication. This method can perform all computations within the constraints of the noise level. The available noise budget depends on the ring size and modulus. After multiplication,

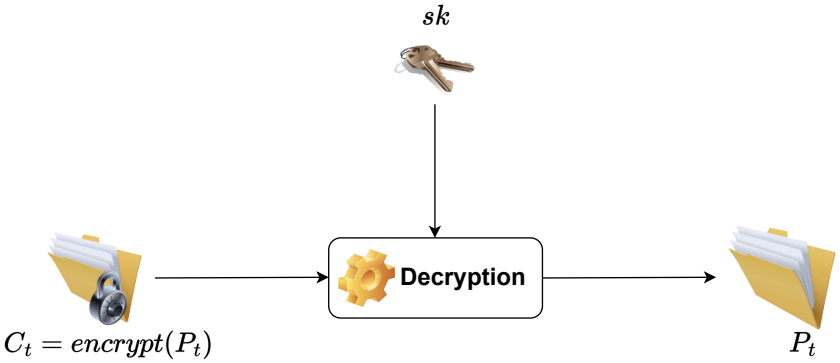
the noise increases. Furthermore, more noise in a ciphertext leads to higher computational overhead. Thus, the noise budget determines the maximum multiplication depth achievable in this scheme. If the application requires a multiplication depth beyond what the noise budget allows, the initial step is to increase the modulus size; however, this may reduce the security level. If there is no room for compromising security, the next option is to increase the ring size. The drawback of enlarging the ring size is that it increases computational complexity and can degrade performance. If the application cannot achieve the desired security level, multiplicative depth, and speed ratio, FHE provides a solution.

FHE can perform the same operations as SWHE and can solve the noise budget problem via the *bootstrapping* method. Gentry (2009) proposed one of the first FHE schemes. The main idea is briefly demonstrated in Figure 2.2. The traditional decryption method needs a secret key and ciphertext. During *bootstrapping*, when the ciphertext is at the edge of the noise budget, this ciphertext can be homomorphically decrypted using an encrypted secret key, and at the output, an equivalent ciphertext is obtained, which has lower noise and a larger computational budget (Badawi and Polyakov, 2023).

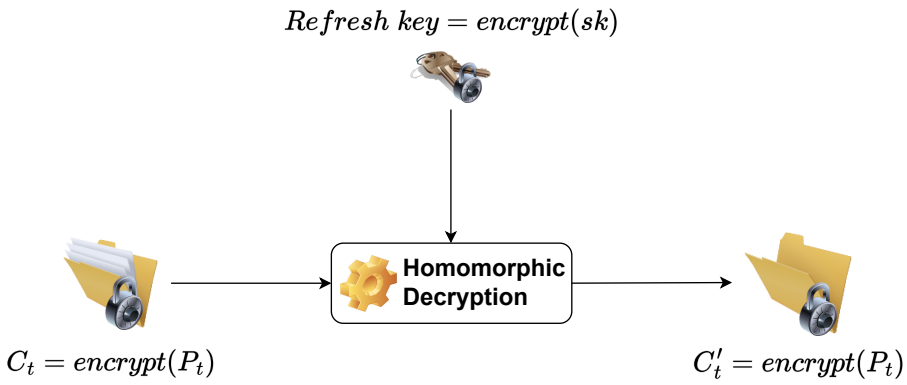
To date, there are a number of SWHE and FHE schemes available. The most promising approaches are CKKS (Cheon et al., 2017), DM (Ducas and Micciancio, 2015), CGGI (TFHE)(Chillotti et al., 2016)(Chillotti et al., 2018), BFV (Fan and Vercauteren, 2012), and BGV (Brakerski et al., 2012). These schemes can be *bootstrappable*, which means that if the *bootstrapping* method is implemented, the scheme can be called FHE; otherwise, SWHE. For example, Microsoft SEAL's CKKS scheme implementation (SEAL, 2020) is SWHE, but OpenFHE's CKKS scheme implementation (Badawi et al., 2022) is FHE. In this thesis, we focus on the SWHE implementation variant for the CKKS scheme.

2.6.1 CKKS Scheme

Several schemes have been proposed over a decade. One of the most promising and relatively recent schemes is CKKS, first proposed in 2017. It is important to note that FHE schemes can be broadly categorized into two families. One of them is binary FHE schemes, such as DM/FHEW and TFHE. These schemes perform operations on single bits or small groups of bits ($bit\_size < 8$). Therefore, they are ideal for binary circuits and logical bit operations. They can be used for Boolean operations, comparisons, lookup tables, programmable bootstrapping, and so on.



(a) Traditional decryption



(b) Bootstrapping

Figure 2.2 Traditional decryption and bootstrapping or homomorphic decryption.

The other category of schemes such as BGV, BFV, and CKKS; typically involves computations on vectors consisting of 16, 32, 64, or 128-bit words. Because of this, they are also capable of homomorphic evaluation of arithmetic circuits over multi-bit numbers. BGV and BFV are ideal for integers such as signed and unsigned integers, while CKKS is ideal for real or approximate numbers such as `double` or `float`.

2.6.1.1 Rescale

The CKKS and BGV schemes are examples of Homomorphic Encryption (HE) schemes that are implemented with a concept known as leveled encryption. In leveled implementations, the homomorphic operations are divided into levels, each with a certain depth or complexity. This approach allows for better control of the computation cost and security

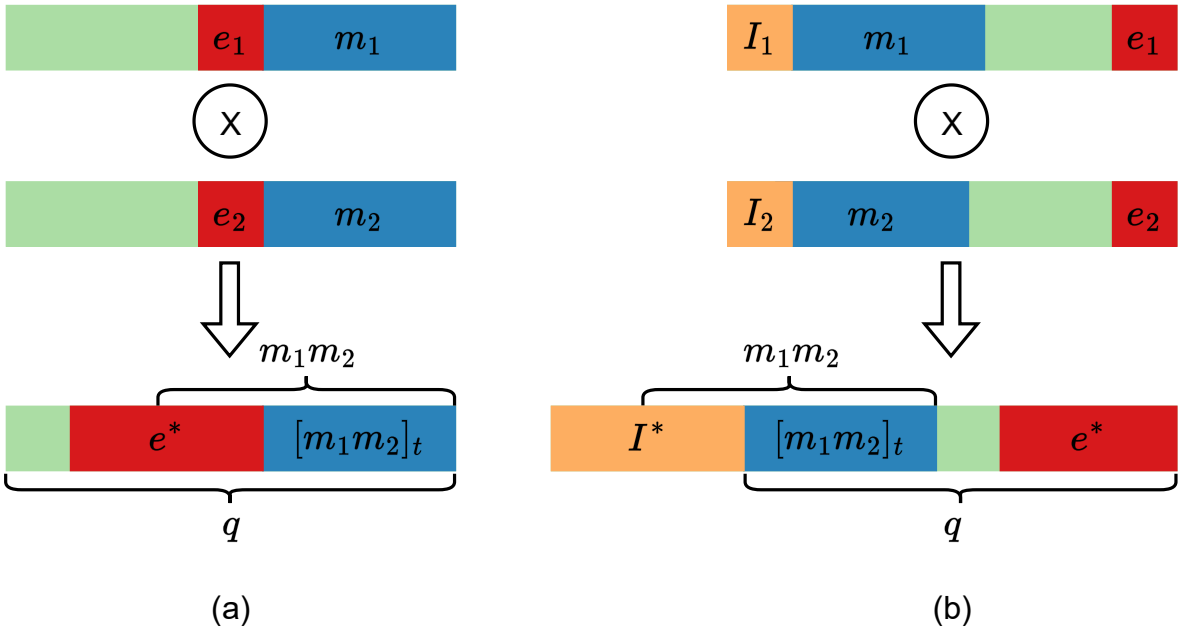


Figure 2.3 Homomorphic multiplication operations of BGV-type schemes (a) and FV-type schemes (b) (Cheon et al., 2017).

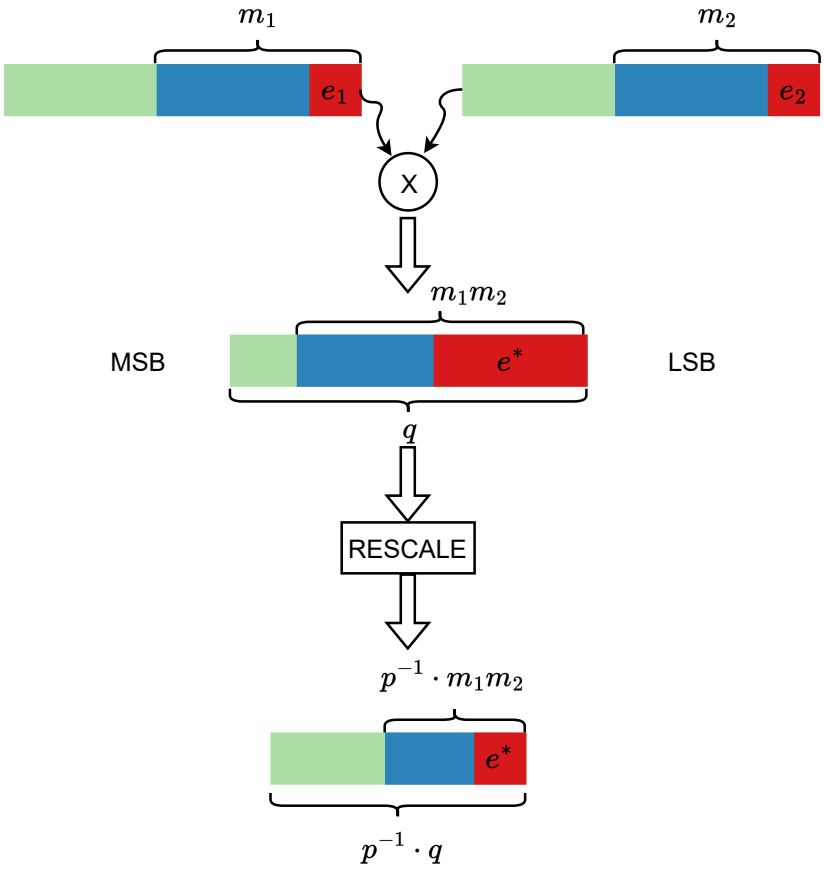


Figure 2.4 Homomorphic multiplication and rescale operation of CKKS scheme (Cheon et al., 2017).

parameters as the depth of operations increases. As previously discussed in Section 2.2, the CKKS scheme incorporates a rescaling operation, rendering it particularly well-suited for arithmetic operations in continuous spaces (Cheon et al., 2017). In contrast, the decryption structure of other HE scheme implementations is less amenable to operations in complex mathematical spaces (Brakerski et al., 2012; Doröz et al., 2015; López-Alt et al., 2012), such as continuous spaces. In these schemes, represented by equations such as $\langle c_i, sk \rangle = m_i + te_i \pmod{q}$, where t denotes the plaintext modulus and q represents the ciphertext modulus, performing operations such as $m_1 + m_2$ and $m_1 m_2$ results in the loss of Most Significant Bits (MSBs), primarily due to the e_i terms.This is illustrated in Figure 2.3 (a).

Similarly, the decryption structure of FV-type HE schemes, exemplified by equations such $\langle c_i, sk \rangle = qI_i + (q/t)m_i + e_i$, leads to a similar loss of MSBs during multiplication operations between two ciphertexts, expressed as $\langle c^*, sk \rangle = qI^* + (q/t)m_1 m_2 + e^*$, with $I^* = tI_1 I_2 + I_1 m_2 + I_2 m_1$ and $e^* \approx t(I_1 e_2 + I_2 e_1)$, as depicted in Figure 2.3 (b).

In contrast, the CKKS scheme offers an efficient solution for performing approximate computations by strategically introducing encryption noise. This noise plays a crucial role in optimizing the trade-off between computation accuracy and efficiency. The key concept involves introducing encryption noise. Consequently, the decryption structure of the CKKS scheme takes the form $\langle c, sk \rangle = m + e \pmod{q}$, where e represents an error of relatively small magnitude compared to the message. This design allows for approximate arithmetic by converting the message as $m' = m + e$. Due to the diminishing error magnitude, which aligns with the security requirements of the underlying hardness assumptions, the loss of most significant bits (MSBs) in the resulting message is unlikely, as illustrated in Figure 2.4. Furthermore, at the output of the rescaling operation, the ciphertext size diminishes, reducing both the size of $p^{-1} \cdot m_1 m_2$ and the ciphertext modulus size.

To clarify the relationship between ciphertext and modulus sizes at varying multiplicative depths, refer to Figure 2.5. As illustrated in the example, there exist three ciphertexts denoted as ct_1, ct_2, and ct_3. At multiplicative depth 0, these ciphertexts have a size of $2 \times n \times decomp\_mod\_size$, where $decomp\_mod\_size$ is defined as $r-1$, indicating the count of RNS moduli minus one. ct_1 and ct_2 undergo multiplication, relinearization, and rescale operations, resulting in the ciphertext ct_1', whose size shrinks to $2 \times n \times (decomp\_mod\_size - 1)$ at multiplicative depth 1, referred to as ct_1'. Should one wish to execute multiplication operations involving ct_1' and ct_3, a *mod_drop* operation becomes necessary due to ct_3 not sharing the same modulus RNS modulus size with respect to ct_1'. The implementation of the *mod_drop* operation in the SEAL CKKS scheme is depicted in Algorithm 4. Subsequent to this, the multiplication operation can be performed, yielding

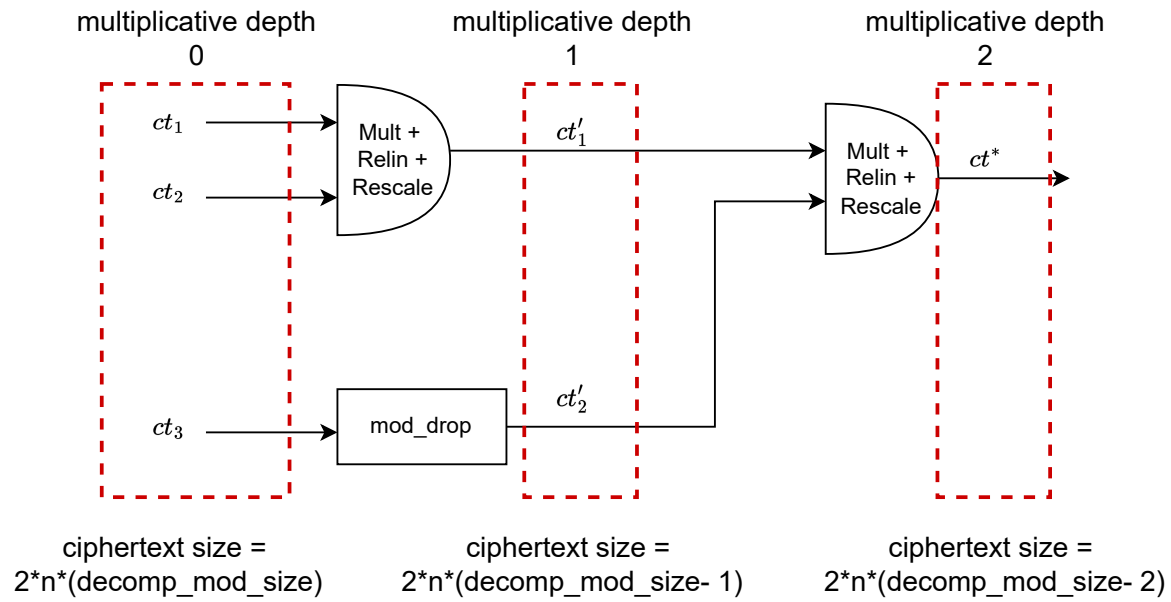


Figure 2.5 Multiplicative depth example via modDrop operation.

ct^*, which has a size $2 \times n \times (decomp\_mod\_size - 2)$ at multiplicative depth 2.

Algorithm 4 CKKS ModDrop Algorithm

Input: $c_i[0], c_i[1] \in \mathbf{R}_{q_i,n}$, $for\ 0 \leq i < r-1$, where r is the number of bases in RNS
Output: $ct_i[0], ct_i[1] \in \mathbf{R}_{q_i,n}$ $for\ 0 \leq i < r-2$

```
for i from 0 by 1 to (r−2) do
    for k from 0 by 1 to 2 do
        ct_i[k] = c_i[k]
    end for
end for
return ct_i[k]
```

The estimation of the ciphertext coefficient modulus denoted as $Q = p \cdot q^{r}$[1], plays a crucial role in the process. This estimation involves selecting a prime number p such that it significantly surpasses the noise parameter Δ, while ensuring $q \approx \Delta$. These parameters define a set of moduli, $Q_r = p \cdot q^r$, where r represents the number of levels or depth.

Figure 2.6 illustrates the progression. It begins with the encryption of a fresh ciphertext modulus. Following a multiplication operation, the ciphertext loses one RNS modulus, which signifies a reduction in its computational depth. When this process continues, the RNS modulus becomes depleted, requiring bootstrapping to maintain the computation's integrity. Without bootstrapping, decryption becomes necessary.

[1]Note that, we strive to maintain the original notation as introduced in Cheon et al. (2017); however, this thesis employs distinct RNS moduli.

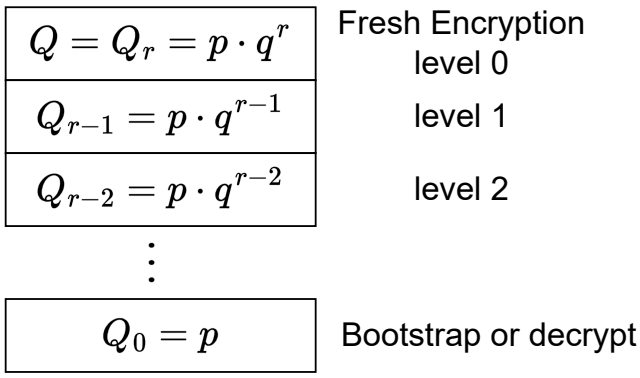


Figure 2.6 Utilization of RNS Modulus Across Various Multiplication Levels.

2.6.1.2 Encoding and Decoding

In the CKKS scheme, the plaintext space is denoted as $\mathbb{C}^{n/2}$, where the plaintext space itself is defined as $R = \mathbb{Z}[x]/(x^n+1)$. The ciphertext space is defined as $R^2_{Q_r} = R_{Q_r} \times R_{Q_r}$, where $R_{Q_r} = \mathbb{Z}_{Q_r}[x]/(x^n+1)$ and $R = \mathbb{Z}[x]/(x^n+1)$.

Encoding plays a pivotal role in transforming message vectors into polynomials. For instance, we can define the input and output of the encoding process as follows: $\mu = (m_0, m_1, ..., m_{n/2-1}) \in \mathbb{C}^{n/2}$ and $m(x) \in \mathbb{Z}/(x^n+1)$. Essentially, this process takes an input vector of complex numbers and yields a polynomial with integer coefficients as output. Furthermore, there exists a homomorphism between vectors and polynomials, defined as follows: $m_1(x) + m_2(x) = \mu_1 \bigoplus \mu_2$ and $m_1(x) \cdot m_2(x) = \mu_1 \bigodot \mu_2$.

We can divide the encoding process into three stages. To begin with, the input comprises a vector of complex numbers $\mu \in \mathbb{C}^{n/2}$. Firstly, to generate $\mathbf{z} \in \mathbb{C}^{n/2}$, such that $z_j = \overline{z_{-j}}$, the input μ is expanded. Subsequently, σ^{-1} is applied to $\mathbf{z}$ using a special inverse Discrete Fourier Transform (DFT) operation (Harris, 1978), employing ζ^j where ζ represents an M-th root of unity and $j \in \mathbb{Z}^*_m = 1, 3, 5, ..., m-1$. Finally, the result is multiplied by the scaling factor Δ and rounded to an integer. The output message is then represented as $m(x) \in R = \mathbb{Z}[x]/(x^n+1)$, such that $m(\zeta^j) = m_i$, with $i = \frac{j+1}{2} - 1$.

The decoding operation can be considered as an inverse encoding operation. First, we have an input vector of polynomials denoted as $m(x) \in R = \mathbb{Z}[x]/(x^n+1)$. The initial step involves multiplication by the scale factor Δ^{-1}. Subsequently, a special Discrete Fourier Transform (DFT) operation, denoted as σ, is applied to the coefficients of $m(x)$ using the M-th root of unity, ζ^{-j}, where ζ represents an m-th root of unity and $j \in \mathbb{Z}^*_m = 1, 3, 5, ..., m-1$. Finally, this process yields an output element denoted as $\tilde{m} \in \mathbb{C}^{n/2}$, such that $\tilde{m}_i = m(\zeta^i)$, with $j \in \mathbb{Z}^*_m$, and $i = \frac{j+1}{2} - 1$, effectively reducing the dimensionality of

$\mathbf{z} \in \mathbb{C}^n$.

2.6.1.3 Key Generation

For the key generation process, in summary, the secret key s is generated as a sample from the distribution χ_{key}, $s \leftarrow \chi_{key}$, where χ_{key} follows a typical probability distribution, often the uniform ternary distribution $(-1, 0, 1)$.

Regarding the public key generation, the following steps are followed: Firstly, a uniform sample $a \leftarrow \chi_{enc}$ is drawn from the encryption probability distribution, which is uniformly distributed over the ring R_Q. Next, an error sample $e \leftarrow \chi_{err}$ is taken, where χ_{err} is the probability distribution employed for error generation, typically following the discrete Gaussian distribution $\mathcal{D}_{\mathbb{Z},\sigma=3.19}$. Utilizing these two generated samples, the public key is derived as $pk = ([-a \cdot s + e]_Q, [a]_Q) \in R_Q^2$.

2.6.1.4 Encryption and Decryption

For the encryption process, it takes plaintext as input, which can be defined as $m(x) \in R$, and produces ciphertext as output, defined as $ct = (c_0, c_1) \in R_Q^2$. Specifically, c_0 and c_1 are computed as follows: $c_0 = [u \cdot pk_0 + m + e_0]_Q$ and $c_1 = [u \cdot pk_1 + e_1]_Q$, where u is sampled from the distribution χ_{enc} (uniform over R_Q), $e_0 \leftarrow \chi_{err}$ and $e_1 \leftarrow \chi_{err}$.

For the decryption process, it takes the ciphertext $ct = (c_0, c_1) \in R_{Q_r}^2$ as input and computes $[c_0 + c_1 \cdot s]_{Q_r}$, where we use the subscript l for leveled encryption. Considering that $c_0 = [u \cdot pk_0 + m + e_0]_Q$ and $c_1 = [u \cdot pk_1 + e_1]_Q$, these expressions can be substituted into the computation, resulting in $[u \cdot pk_0 + m + e_0 + s \cdot (u \cdot pk_1 + e_1)]_{Q_r}$. Additionally, knowing that $pk = ([-a \cdot s + e]_Q, [a]_Q) \in R_Q^2$, we can replace $[-a \cdot s + e]_Q$ with pk_0 and $[a]_Q$ with pk_1. This yields

$$[u \cdot (-a \cdot s + e) + m + e_0 + s \cdot (u \cdot a + e_1)]_{Q_r}$$

Subsequent simplification results in $[m + u \cdot e + e_0 + s \cdot e_1]_{Q_r}$, which can be expressed as $[m(x) + e']_{Q_r}$. It can be observed from the expression $m'(x) = m(x) + e' \in R_{Q_r}$ that smaller the e, results in smaller noise in enc and vice versa.

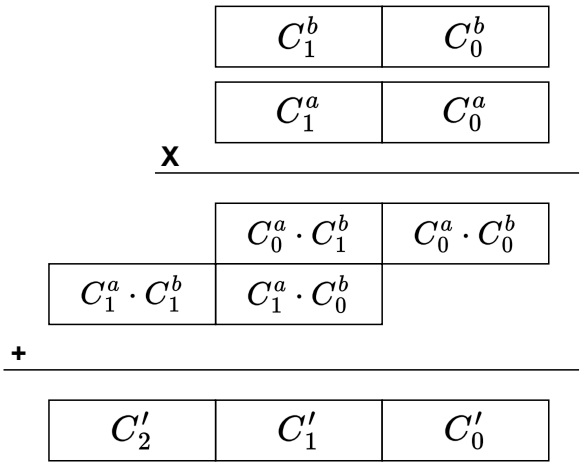


Figure 2.7 Multiplication of Ciphertexts Using Schoolbook Method.

2.6.1.5 Homomorphic Addition

Homomorphic addition represents one of the fundamental operations within the CKKS scheme. This operation involves two ciphertext inputs, denoted as $Enc(m^a) = ct^a = (c_0^a, c_1^a)$ and $Enc(m^b) = ct^b = (c_0^b, c_1^b)$, with each element belonging to $R_{Q_r}^2$. The operation yields the result $ct^+ = (c_0^a + c_0^b, c_1^a + c_1^b) \in R_{Q_r}^2$. As previously discussed in Section 2.6.1.4, upon simplification, we can express this as $[u^+ \cdot pk_0 + (m^a + m^b) + e_0^a, u^+ \cdot pk_1 + e_1^b]_{Q_r}$. Notably, this operation results in an additive increase in noise. Consequently, the output represents the encryption of the sum, $Encrypt(m^a + m^b)$.

2.6.1.6 Homomorphic Multiplication

Homomorphic multiplication takes two ciphertexts as input, denoted as, $Enc(m^a) = ct^a = (c_0^a, c_1^a)$ and $Enc(m^b) = ct^b = (c_0^b, c_1^b)$, both of which are elements of $R_{Q_r}^2$. Conceptually, this operation performs elementary schoolbook multiplication, as illustrated in Figure 2.7. Essentially, at the output of this multiplication, the result is expressed as tuple $(c_0^a \cdot c_0^b,\ c_0^a \cdot c_1^b + c_1^a \cdot c_0^b,\ c_1^a \cdot c_1^b) \in R_{Q_r}^3$. To reduce the dimensionality from $R_{Q_r}^3$ to $R_{Q_r}^2$, a relinearization or *Key Switching* operation is performed, as illustrated in Figure 2.8. Moreover, to mitigate the amplification of multiplicative noise, a rescaling operation is performed. Finally, at the output, it returns $[m^a \cdot m^b + e^{\times}]_{Q_r}$.

Figure 2.8 Relinearization or *Key switching* operation.

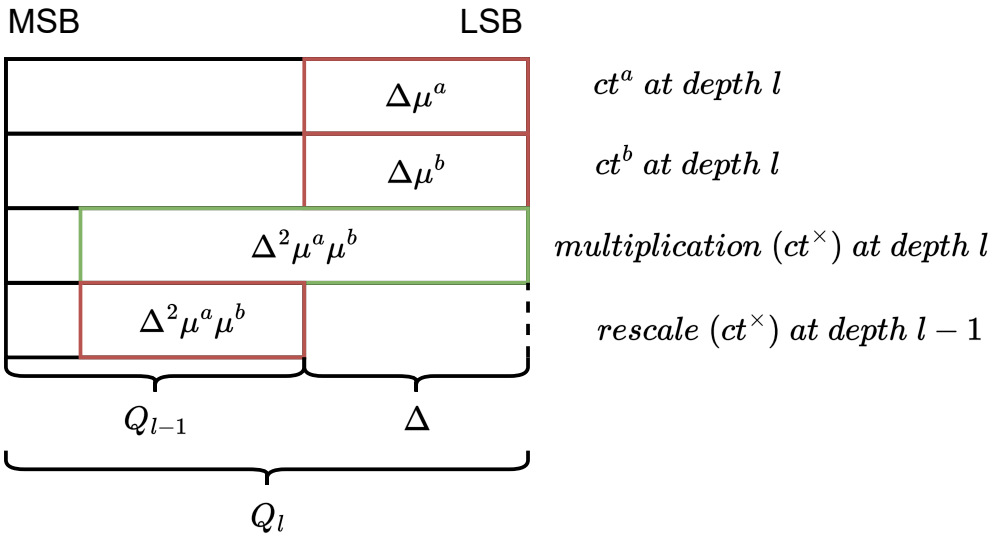


Figure 2.9 Homomorphic rescale operation.

2.6.1.7 Homomorphic Rescaling

After homomorphic multiplication, the result is expressed as $[m^a \cdot m^b + e^x]_{Q_r}$. As illustrated in Figure 2.9, both the message magnitude and noise experience an increase. This growth exhibits exponential behavior with respect to the depth l. Therefore, the rescale operation, often referred to as *Modulus Switching*, is employed to maintain the message magnitude at the same level as fresh encryption while reducing the accumulated noise.

In the third step of Figure 2.9, we have $ct^\times = Encryption(\Delta^2\mu^a\mu^b + e^\times) \in R_{Q_r}$. The rescale operation thus eliminates the least significant bits (LSBs) of the product as follows: $rescale(ct^\times) = Encryption(\lceil \frac{1}{\Delta} \cdot (\Delta^2 \cdot \mu^a \cdot \mu^b + e^\times) \rfloor) \in R_{Q_r}$, where $q \approx \Delta$, and then proceeds to the next level as follows: $rescale(ct^\times) = Encryption(\Delta \cdot \mu^a \cdot \mu^b + \frac{1}{q}e^\times + e_{rescale}) \in R_{Q_{l-1}}$.

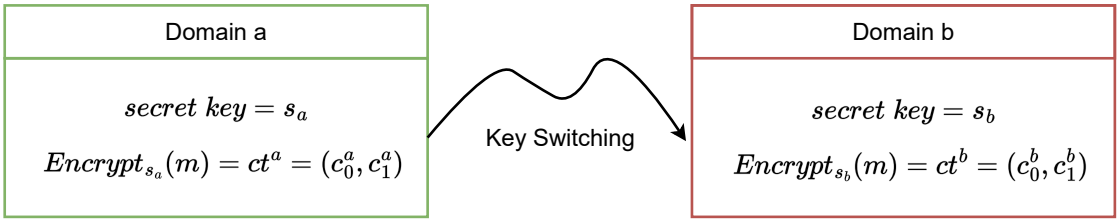


Figure 2.10 Illustration of the Key Switching operation.

2.6.1.8 Relinearization

As illustrated in Figure 2.10, the primary concept behind the relinearization operation involves transitioning a ciphertext encrypted under one key to an equivalent ciphertext. This equivalence signifies encrypting the same message but under a different key. For instance, given a ciphertext $ct^a = (ct_0^a, ct_1^a) \in R_{Q_r}^2$, encrypting a message m that can be decrypted using the secret key s_a, the objective is to transform ct^a into $ct^b = (ct_0^b, ct_1^b)$. This transformed ciphertext encrypts the same message m but can be decrypted using the secret key s_b.

The CKKS scheme requires the use of the *Key Switching* operation in three distinct scenarios: after the homomorphic multiplication of two ciphertexts, following the application of an automorphism to a ciphertext for slot rotation, and in proxy re-encryption, which transitions a ciphertext encrypted under one key to an equivalent ciphertext. In the case of proxy re-encryption, it is akin to encrypting the same message but under another key. In all of these scenarios, the *Key Switching* operation exhibits striking similarities with only minor variations.

As depicted in Figures 2.7 and 2.8, we are provided with ciphertexts $ct^a = (c_0^a, c_1^a)$ and $ct^b = (c_0^b, c_1^b)$, each belonging to R_{Q_r}. Following the multiplication operation, three intermediate results are obtained: $c_0' = c_0^a \cdot c_0^b$, $c_1' = c_0^a \cdot c_1^b + c_1^a \cdot c_0^b$, and $c_2' = c_1^a \cdot c_1^b$, all within R_{Q_r}. It is evident that the ciphertext size experiences exponential growth as the circuit depth increases linearly. The relinearization operation serves to compress the ciphertext, reducing it to a size of two elements. The core concept involves decrypting c' normally using $m(x) = [c_0' + c_1' \cdot s + c_2' \cdot s^2]_{Q_r}$. Here, via key switching, the decryption with s^2 can be turned into decryption only by s.

2.6.1.9 Homomorphic Permutation or Rotation

Since this operation is employed in the Key Switching algorithm, it bears a resemblance to the relinearization operation. Furthermore, this operation is utilized in both the *Galois Elt* Algorithm 10 and the *Apply Galois Algorithm* 9 in Chapter 4.

In terms of the mathematical foundation, we can define $Gal(\mathbb{Q}(\zeta_m)/\mathbb{Q})$ as the set of mappings k_j: $m(x) \rightarrow m(x^j) \pmod{\phi_m(x)}$, where $gcd(j,m) = 1$. When we apply k_j to the ciphertext polynomials, it results in a controlled permutation of the ciphertext slots. For instance, $Gal(\mathbb{Q}(\zeta_m)/\mathbb{Q}) \approx \mathbb{Z}^*_{m=2n} = \langle 5, -1 \rangle$.

- $m(x) = m(\zeta), m(\zeta^5), \ldots, m(\zeta^{2n-3})$
- $k_j(m(X^j)) = m(\zeta^j), m(\zeta^{5j}), \ldots, m(\zeta^{(2n-3)j})$

Furthermore, these operations can be performed homomorphically on ciphertext polynomials like $(c_0(x), c_1(x))$. However, it's important to note that when the Key Switching operation is applied without using the *Apply Galois Algorithm*, the resulting ciphertext cannot be decrypted using the original secret key s, but rather requires the use of $s' = k_j(s(x))$. Thanks to the *Apply Galois Algorithm* we can perform the decryption via original secret s.

3. GPU ARCHITECTURE

In this section, we present a brief overview of the GPU (graphics processing unit), its brief history, its general architecture, and its adaptation as a means to accelerate general-purpose computing. Furthermore, we briefly discuss the programming model of GPU and discuss the memory hierarchy associated with GPU.

3.1 History

Historically GPUs were employed to process and render computer graphics in real-time for visualization in application areas such as gaming, scientific computing, and digital image processing. Since these fields involve computationally intensive tasks that can be parallelized, using GPUs as accelerators is one of the most promising approaches.

By Moore's Law (Moore, 1965) transistor sizes on a die kept reducing year-by-year whereas, the operating clock frequencies kept increasing thus (Dennard et al., 1974), each subsequent generation of chips delivered higher performance until the year 2005, when this trend peaked. In order to move forward and meet ever-increasing demands for performance, newer architectures and alternate design organizations were adopted. Parallel architectures were especially explored and chosen as a way forward to keep up with performance demands. This parallelism could be achieved at both the data and instruction levels, marking the onset of the "parallel era" and the emergence of multi-core CPUs and GPUs. GPUs can employ many more threads executing concurrently than CPUs,

albeit with somewhat lower clock frequencies. All in all, CPUs are better suited for a diverse range of general-purpose computing tasks, whereas tasks that primarily revolve around numerical computation and large matrix operations, which can be parallelized at a granular level, are better accommodated by GPUs.

Additionally, various accelerating approaches exist aside from GPUs, such as field-programmable gate arrays (FPGAs) or application-specific integrated circuits (ASICs). GPUs offer greater flexibility in comparison to ASICs and FPGAs, making them a quicker solution for acceleration. However, these solutions are primarily application-specific, which is why they outperform GPUs in specific cases.

In conclusion, using an analogy, one might liken the CPU to a race car. Conversely, we can liken the GPU to a bus.

3.2 GPU Hardware Basics

In order to obtain more performance GPU and CPU should work together. For some cases opening a folder needs a single thread but for some cases like display screen needs GPU. Mostly there are 2 different implementations for CPU and GPU as shown in Figure 3.1. On the left side CPU and GPU work in separate chips. As an example of this NVIDIA RTX 3070. On the right side CPU and GPU are in the same chips for example AMD's Bristol Ridge APU. This thesis mostly focuses on separate integration.

To achieve higher performance, GPUs and CPUs should collaborate. In some cases, tasks such as opening a folder require a single thread, while others, like rendering the display screen, demand the GPU's processing power. Typically, there are two different implementations for CPUs and GPUs, as illustrated in Figure 3.1. On the left side, CPUs and GPUs operate on separate chips, as seen in examples like the NVIDIA RTX 3070. On the right side, CPUs and GPUs are integrated onto the same chip, as exemplified by AMD's Bristol Ridge APU. This thesis predominantly focuses on separate integration.

As shown in the left side of Figure 3.1, CPU and GPU have their own memory. Facilitating data transfer between the CPU and GPU in discrete systems necessitates the use of a BUS. In contrast, integrated systems feature a shared cache memory to facilitate low-latency data transfer.

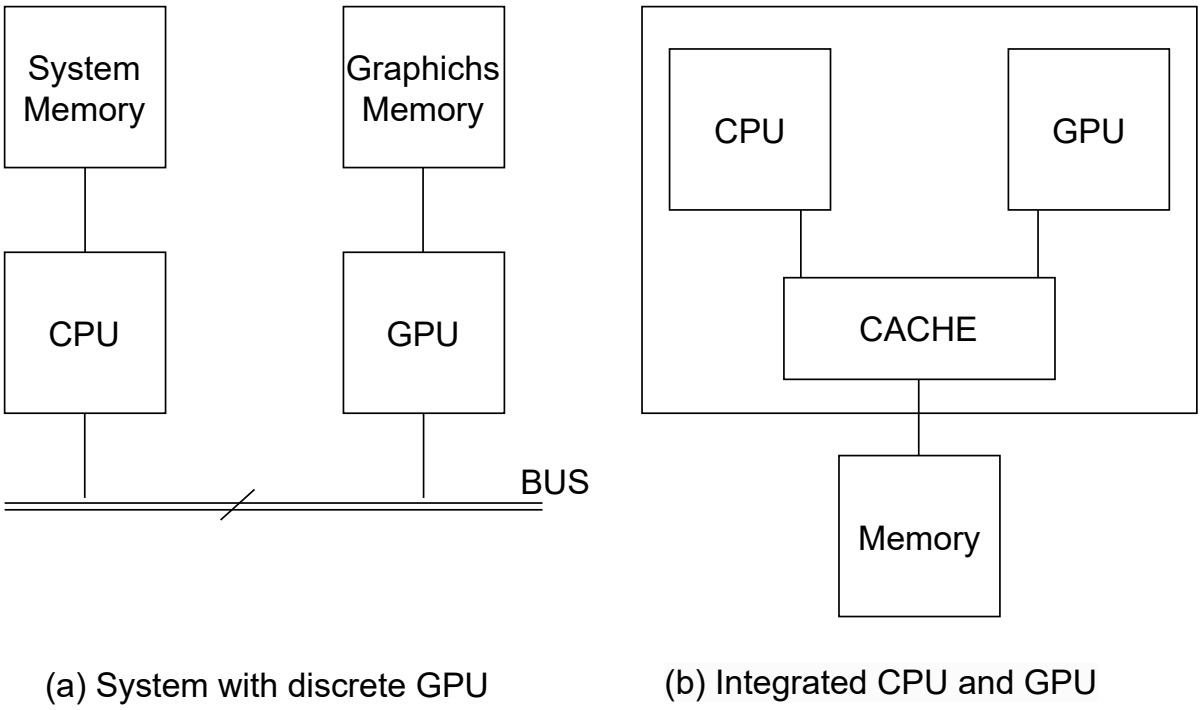


Figure 3.1 Discreate and integrated CPU and GPU examples.

3.3 Programming Model

Several computing APIs, such as OpenCL, ROCm (Radeon Open Compute), and CUDA, employ an MIMD-like programming model that facilitates the launching of a large array of scalar threads onto the GPU. Since we are working with the NVIDIA RTX 3070 and RTX 4090, this thesis utilizes the CUDA (Compute Unified Device Architecture) language to implement the CKKS scheme.

Firstly, the code begins execution on the CPU (host) and performs any necessary pre-computations. Afterward, it must allocate memory as needed using `cudaMalloc`. Following that, it copies the data onto that memory using `cudaMemcpy`, which takes memory direction as an input. If data is being copied from the CPU to the GPU, the last input should be `cudaMemcpyHostToDevice`; otherwise, it should be `cudaMemcpyDeviceToHost`. Then, a driver running on the CPU initiates computation on the GPU, with the GPU deciding which code will run using kernels.

Kernels take two inputs: the number of blocks and the number of threads for each block, as denoted here `<<<numBlocks, numThreads>>>`. As shown in Figure 3.2, kernels are executed as a grid of thread blocks, each of which contains threads specified during kernel

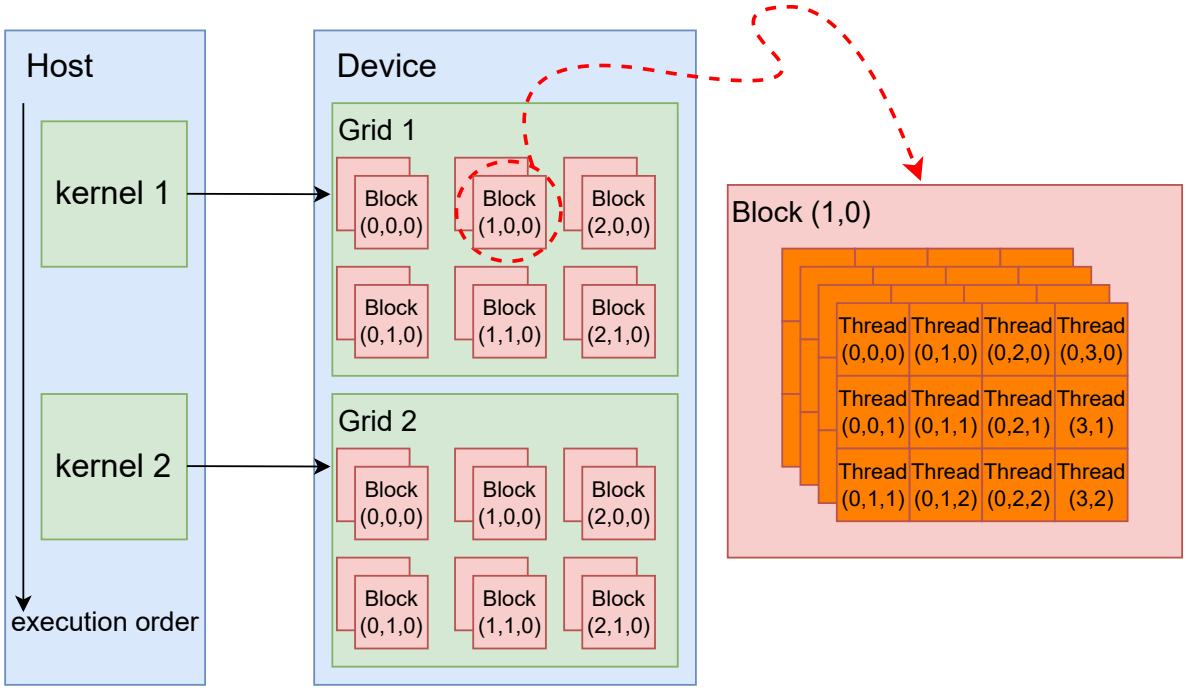


Figure 3.2 Illustration Example of Kernels, Grids, Blocks, and Threads.

generation, and kernels are executed sequentially. Additionally, blocks and threads can be defined as one, two-dimensional, or three-dimensional arrays. This simplifies memory addressing when processing multi-dimensional data.

In CUDA, GPU functions typically start with `__global__ void` or `__device__ void`. `__global__ void` indicates that the function is executed on the GPU but called from the CPU. Conversely, `__device__ void` means that the function is both executed and called on the GPU. To manage threads, a `Thread_idx` should be defined as follows: `blockIdx.x * blockDim.x + threadIdx.x`. While using `Thread_idx`, threads can work with desired patterns of indexes to access memory and execute the algorithm.

3.4 GPU Memory Organization and Hierarchy

One of the key components of NVIDIA GPUs is the Streaming Multiprocessor (SM), which is responsible for executing tasks in parallel. Each SM contains several CUDA cores. When specifying the number of blocks and threads in a kernel, these are organized into a grid and then assigned to the SMs for execution. Figure 3.3, adapted from (NVIDIA,

2021), provides a simple illustration of an SM in the RTX 3070 GPU.

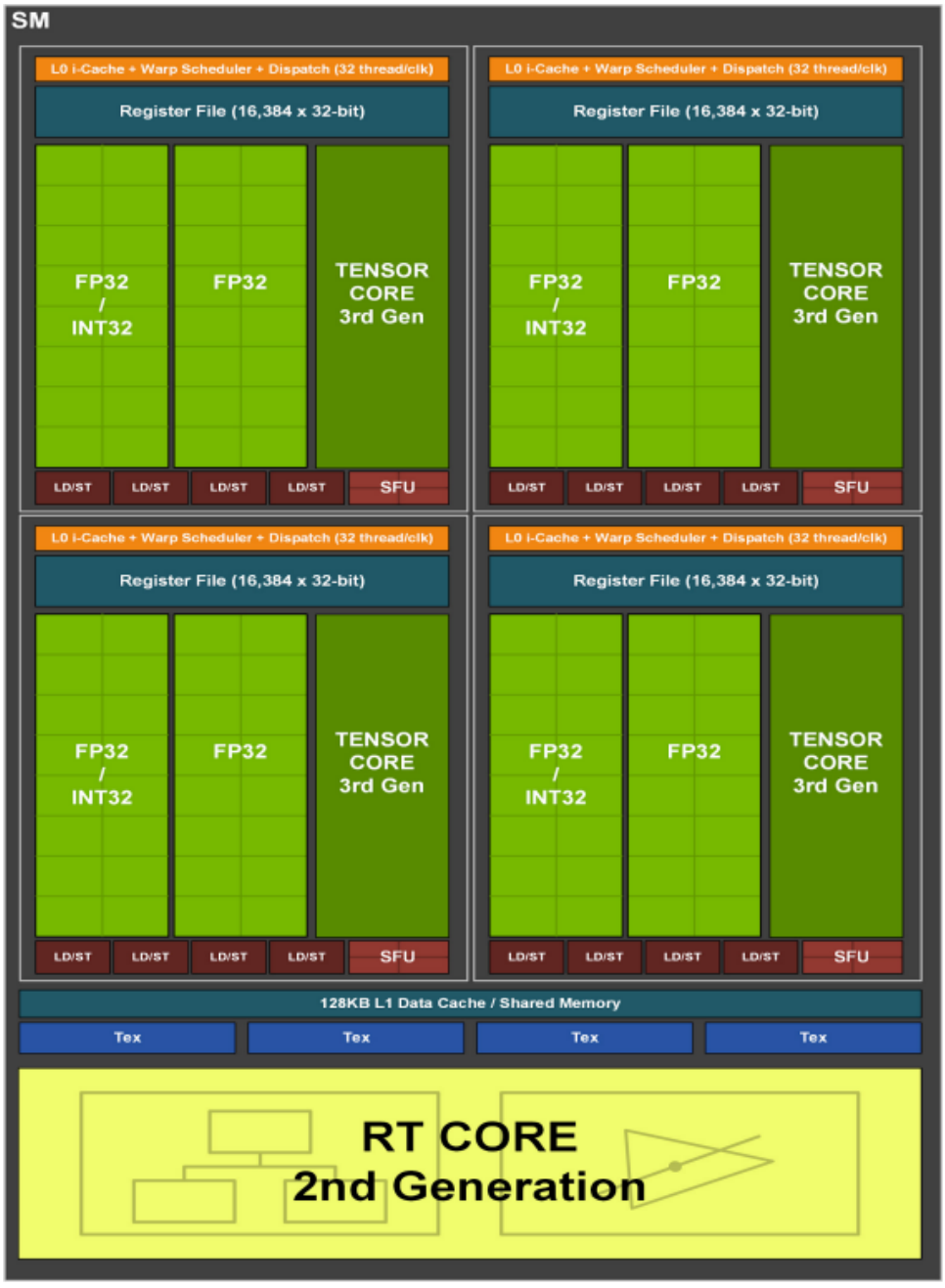


Figure 3.3 RTX 3070 Streaming Multiprocessor (NVIDIA, 2021).

As mentioned previously and illustrated in Figure 3.4, grids consist of several thread blocks. All threads within the same block can share their data via shared memory, but threads from other blocks cannot access this data. 32 threads form a warp or scheduling group. The RTX 3070 has up to 32 warps for each block, resulting in $32 \times 32 = 1024$ threads per block. Each block executes on a single SM, and the RTX 3070 has 46 SMs.

As shown in Figure 3.4 each thread can read and write its own registers and local memory.

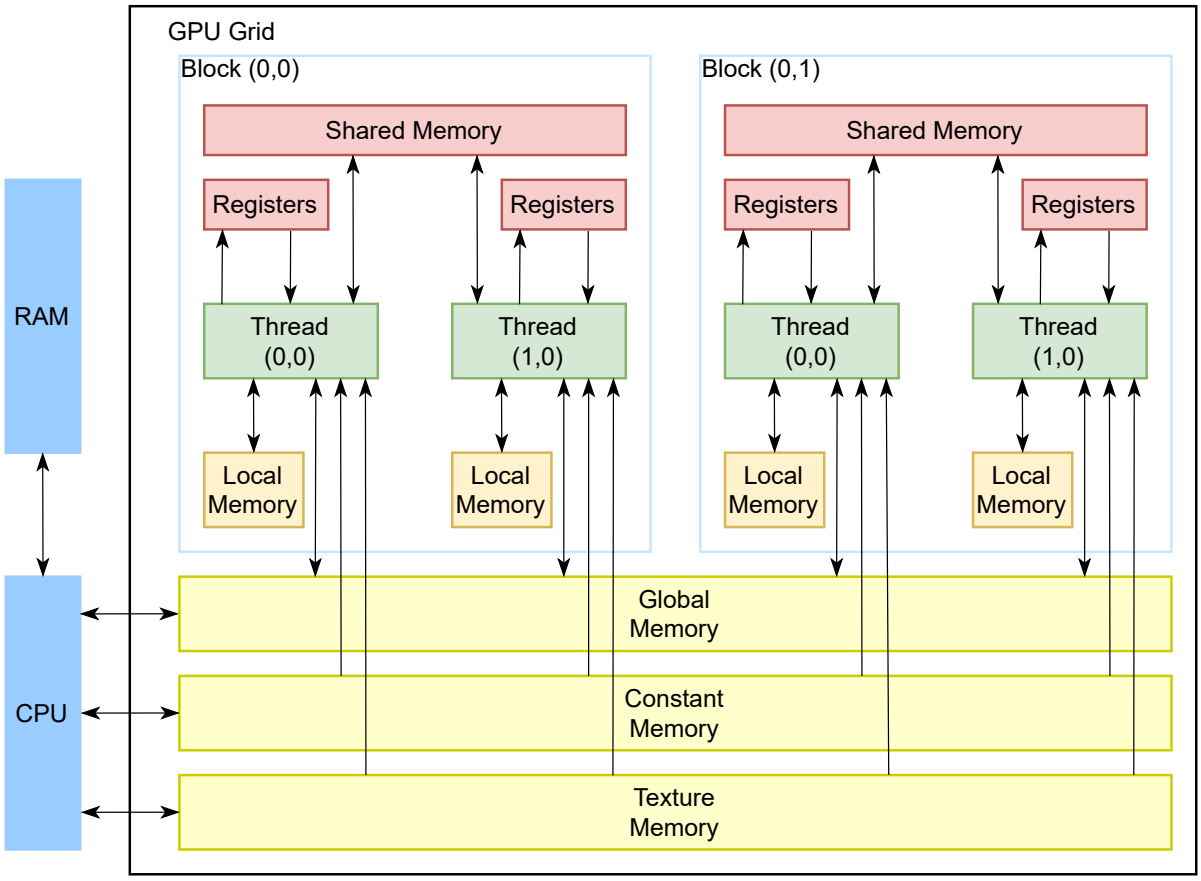


Figure 3.4 CUDA Memory Model.

Threads in each block can read from and write to the shared memory. Threads in each grid can read and write to global memory and can only read from constant and texture memory.

Registers and shared memory are dedicated hardware; therefore, they reside in on-chip memory. Threads can access these locations in a single clock cycle. Each thread has its private register memory to store frequently accessed variables. Shared memory facilitates inter-thread cooperation and allows the sharing of intermediate information among threads within the same block. Global memory is DRAM, off-chip memory. While all threads can access that memory, it takes significantly more clock cycles compared to register and shared memory access times. Moreover, global memory has a size in gigabytes, whereas shared memory is measured in kilobytes.

Table 3.1 displays different variable types and their associated access penalties. Life time indicates the duration of execution. If the Life time of a variable is within a kernel, it is declared within the kernel function's body; otherwise, if the Life time of a variable is within an application, the variable is defined outside of the function's body.

Table 3.1 Variables and access penalties on modern GPUs memory architecture

Variable Declaration	Memory	Life Time	Perform. Penalty
`int localVar;`	Register	Thread	1×
`int LocalArr[10];`	Local	Thread	100×
`__device__ int GVar;`	Global	Application	100×
`__constant__ int CVar;`	Constant	Application	1×
`__shared__ int SVar;`	Shared	Block	1×

4. Implementation of CKKS on GPU

In this chapter, we introduce our GPU implementations of integers multiplication and Barrett reduction, INTT, NTT implementations, and SEAL CKKS GPU implementation by presenting pseudo-codes and providing detailed explanations for the kernels. We use 64-bit precision for the arithmetic operations.

4.1 64-bit Multiplication and Barrett Reduction

As discussed in Section 2.4, Barrett reduction is implemented to achieve efficient reduction. Furthermore, for performing 64-bit multiplications, PTX code is used for GPU implementation.

Barrett reduction is implemented as described in Algorithm 1. To achieve the most efficient result for multiplication, PTX code is employed. The code takes two constants, `unsigned long` values, denoted as 'a' and 'b'. The algorithm performs multiplications as follows: $a.low \times b.low$, $a.high \times b.low$, $a.low \times b.high$, $a.high \times b.high$, and finally, carries out the addition of these results by aligning them properly. Then, the steps of the Barret reduction are executed as described in Algorithm 1.

4.2 INTT and NTT Implementation

In this thesis, we are utilizing the NTT implementation from the work of Ali Şah Özcan et al. (Şah Özcan and Savaş, 2023), to which one can refer for detailed information. Algorithms 2 and 3 present the pseudo-code for the NTT and INTT operations, respectively.

The Inverse and forward NTT implementation comprises two kernels for ring sizes 2^{12}, 2^{13},..., 2^{15}. This design prioritizes the full utilization of shared memory to minimize unnecessary kernel usage. Additionally, optimal efficiency is achieved by maximizing theoretical occupancy and ensuring coalesced access to global memory. Furthermore, the implementation supports not only individual NTT operations but also batch NTT operations.

4.3 SEAL CKKS GPU Implementation

In this section, we present the GPU implementation of the CKKS homomorphic operation algorithms, which include homomorphic addition, homomorphic multiplication, rescaling, relinearization, and rotation. To develop these algorithms, we utilize the Microsoft SEAL library. Initially, we created Python models and pseudocode to outline the implementation process on the GPU. Subsequently, we commenced the GPU implementation.

4.3.1 Homomorphic Addition

Homomorphic addition is one of the simplest and least expensive operations in the CKKS scheme. As depicted in Algorithm 5, it involves only element-wise addition operations. Therefore, the implementation consists of only one kernel with a maximum thread count allowed by the GPU, which is 1024 threads for each block. The kernel has $2 \times decomposition\_mod\_count \times (r/thread\_num)$ blocks, where $decomposition\_mod\_count = r - 1$ and $r =$ number of RNS bases.

Algorithm 5 CKKS Addition Algorithm

Input: $ct_i[k], \bar{ct}_i[k] \in \mathbf{R}_{q_i}$ for $0 \leq i < r-1$, for $0 \leq k < 2$
Output: $\tilde{ct}_i[k] \in \mathbf{R}_{q_i}$ for $0 \leq i < r-1$, for $0 \leq k < 2$
1: **for** i from 0 by 1 to $(r-1)$ **do**
2: **for** k from 0 by 1 to 2 **do**
3: $\tilde{ct}_i[k] = [ct_i[k] + \bar{ct}_i[k]]_{q_i}$
4: **end for**
5: **end for**
6: **return** $\tilde{ct}_i[k]$

4.3.2 Homomorphic Multiplication

Homomorphic multiplication is another simple and cost-effective operation in the CKKS scheme. As depicted in Algorithm 6, it involves the multiplication of two ciphertexts, each with two parts, denoted as 0 and 1. Therefore, the implementation consists of only one kernel. Initially, this operation was designed using the maximum number of threads per block. Since the output consists of ciphertexts with three parts, three 'if' statements were used to handle these operations. However, it was later determined that the algorithm could be optimized to eliminate the need for 'if' statements. An efficient memory access pattern was achieved using 256 threads per block.

Algorithm 6 CKKS Multiplication Algorithm

Input: $ct_i[k], \bar{ct}_i[k] \in \mathbf{R}_{q_i}$ for $0 \leq i < r-1$, for $0 \leq k < 2$
Output: $\tilde{ct}_i[k] \in \mathbf{R}_{q_i}$ for $0 \leq i < r-1$, for $0 \leq k < 3$
1: **for** i from 0 by 1 to $(r-1)$ **do**
2: $\tilde{ct}_i[0] = [ct_i[0] \times \bar{ct}_i[0]]_{q_i}$
3: $\tilde{ct}_i[1] = [(ct_i[0] \times \bar{ct}_i[1]) + (ct_i[1] \times \bar{ct}_i[0])]_{q_i}$
4: $\tilde{ct}_i[2] = [ct_i[1] \times \bar{ct}_i[1]]_{q_i}$
5: **end for**
6: **return** $\tilde{ct}_i[k]$

Furthermore, the blocks are two-dimensional, with the x-dimension having $r/256$ blocks and the y-dimension having *decomposition_mod_count* blocks $(r-1)$. To access $ct_i[0]$, $\tilde{ct}_i[0]$, and $\bar{ct}_i[0]$, the code uses the following pattern: $(r \times j) + thread\_idx$ (referred to as `address1`), where j is the block number in the y-dimension, and $thread\_idx$ is the thread ID, as discussed in Chapter 3.3. For accessing $ct_i[1]$, $\tilde{ct}_i[1]$, and $\bar{ct}_i[1]$, the code employs the pattern: $(r \times j) + ((r-1) \times r) + thread\_idx$. Finally, for writing $\tilde{ct}_i[2]$, the code utilizes the pattern: $2 \times ((r-1) \times r)$+`address1`. This new memory access pattern resulted in a 20% speedup of the algorithm.

4.3.3 Relinearization

As described in Chapter 2.6.1.8, after each multiplication operation, a relinearization operation should be performed to transform the three ciphertexts resulting from the multiplication into two ciphertexts. Algorithm 7 provides the pseudocode for the CKKS relinearization operation. This implementation comprises eleven kernels: eight of these kernels are NTT and INTT kernels (two kernels per NTT/INTT), and three of them are dedicated to the relinearization operation.

Algorithm 7 CKKS Relinearization Algorithm

Input: $c_i[0], c_i[1], c_i[2] \in \mathbf{R}_{q_i,n}$
Input: $evk_i^j[k] \in \mathbf{R}_{q_j,n}$, where $k \in \{0,1\}$, $0 \leq j < r$, and $0 \leq i < (r-1)$
Output: $ct_i[0], ct_i[1] \in \mathbf{R}_{q_i,n}$

1: $\bar{A}_{j,k} = 1$
2: **for** i from 0 by 1 to $r-2$ **do**
3: $\quad cpoly_i = INTT_{n,q_i}(c_i[2])$
4: $\quad$ **for** j from 0 by 1 to $r-1$ **do**
5: $\quad\quad$ **for** k from 0 by 1 to 1 **do**
6: $\quad\quad\quad a_{i,j,k} = [NTT_{n,q_j}(cpoly_i) \odot evk_i^j[k]]_{q_j}$
7: $\quad\quad\quad \bar{A}_{j,k} = [\bar{A}_{j,k} + a_{i,j,k}]_{q_i}$
8: $\quad\quad$ **end for**
9: $\quad$ **end for**
10: **end for**
11: **for** k from 0 by 1 to 1 **do**
12: $\quad A_{r-1,k} = INTT_{n,q_{r-1}}(\bar{A}_{r-1,k})$
13: **end for**
14: $half = \lfloor \frac{q_{r-1}}{2} \rfloor$
15: **for** i from 0 by 1 to $r-1$ **do**
16: $\quad halfmod = [half]_{q_i}$
17: $\quad$ **for** k from 0 by 1 to 1 **do**
18: $\quad\quad tmp = [[A_{r-1,k} + half]_{q_{r-2}} - halfmod]_{q_i}$
19: $\quad\quad tmp = NTT_{n,q_i}(tmp)$
20: $\quad\quad tmp = [\bar{A}_{i,k} - tmp]_{q_i}$
21: $\quad\quad tmp = [tmp \times q_r^{-1}]_{q_i}$
22: $\quad\quad ct_i[k] = [c_i[k] + tmp]_{q_i}$
23: $\quad$ **end for**
24: **end for**

Figure 4.1 illustrates the complete relinearization operation. As depicted in line 3, Algorithm 7 initially applies the Inverse Number Theoretic Transform (INTT) operation to $c_i[2]$, where $c_i[2]$ denotes the third ciphertext resulting from the multiplication operation. Subsequently, the next set of kernels commences to perform the Number Theoretic

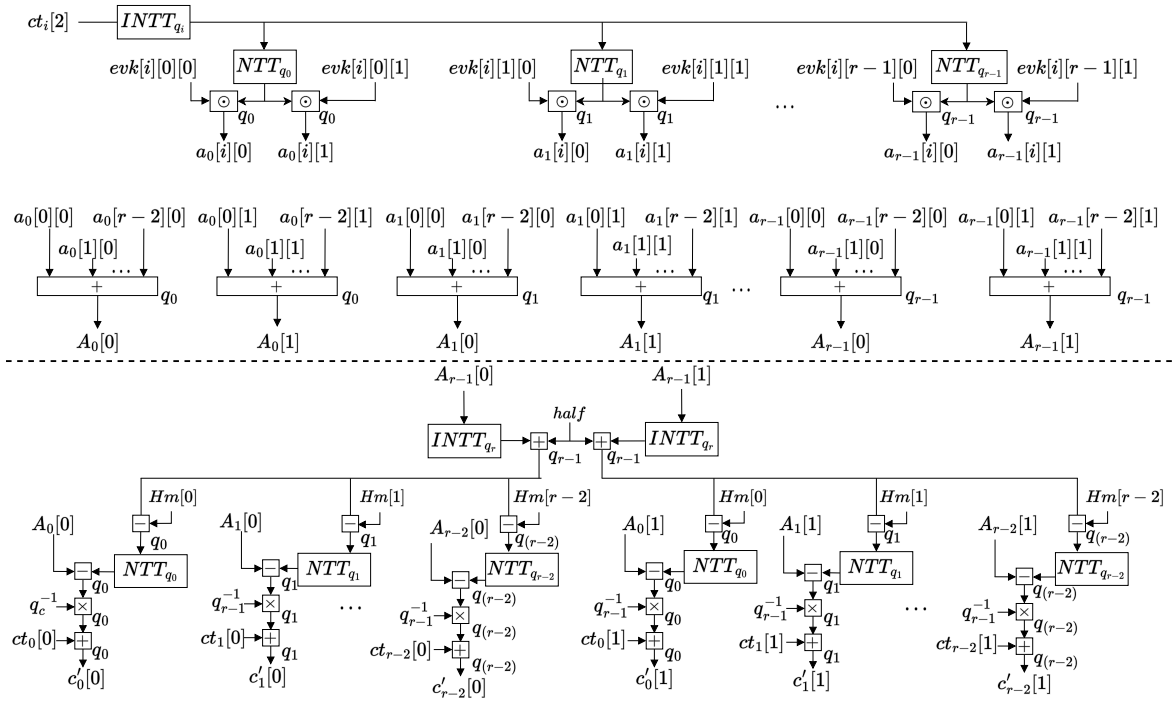


Figure 4.1 Overall, the implementation of the Relinearization operation can be summarized. The symbols $+$, $-$, and $\times$ represent addition, subtraction, and multiplication, respectively, in either $\mathbb{Z}_q$ or $\mathbf{R}_q$, while the symbol $\odot$ denotes modular pointwise multiplication for the vector representation of the elements in $\mathbf{R}_q$ within the NTT domain.

Transform (NTT) operation for each RNS modulus on the output of the first kernel, as described in line 6.

The next kernel begins with the evaluation key multiplication ($\odot$ in line 6 of the algorithm) and performs additions, as depicted in lines 6-7. Furthermore, this kernel utilizes a 2-dimensional grid size, where the x-dimension comprises $n/1024$ blocks, the y-dimension matches the number of bases in the RNS, and there are 1024 threads per block. This implies that the thread count is equal to the total number of coefficients in all RNS bases of the ciphertext $c_i[2]$. The kernel indexes the RNS moduli using the y-dimension of the grid and retrieves the input data using the following pattern: $thread\_idx + (blockDim.y \times n) + (i \times (r \times n))$, where i is the variable of the outer **for** loop and $thread\_idx$ ranges from 0 to n. Due to the dependency in the outer **for** loop, this kernel contains only one for loop, which iterates from 0 to $r-1$, and the innermost **for** loop (i.e., line 5) is unrolled.

The next two kernels process the output of the third kernel and perform the INTT operation on the last RNS modulus base, as depicted in line 12.

The following kernel performs addition operations involving `half` and `halfmod`, corre-

Algorithm 8 CKKS ModSwitch Algorithm

Input: $c_i[0], c_i[1] \in \mathbf{R}_{q_i,n},\ for\ 0 \leq i < r-1$
Output: $ct_i[0], ct_i[1] \in \mathbf{R}_{q_i,n}\ for\ 0 \leq i < r-2$
1: **for** k from 0 by 1 to 1 **do**
2: $\quad A_{r-1,k} = INTT_{n,q_{r-1}}(c_{r-1}[k])$
3: **end for**
4: $half = \lfloor \frac{q_{r-1}}{2} \rfloor$
5: **for** i from 0 by 1 to $r-2$ **do**
6: $\quad halfmod = [half]_{q_i}$
7: $\quad$ **for** k from 0 by 1 to 1 **do**
8: $\quad\quad tmp = [[A_{r-1,k} + half]_{q_{r-1}} - halfmod]_{q_i}$
9: $\quad\quad tmp = NTT_{n,q_j}(tmp)$
10: $\quad\quad tmp = [c_i[k] - tmp]_{q_i}$
11: $\quad\quad ct_i[k] = [tmp \times q_r^{-1}]_{q_i}$
12: $\quad$ **end for**
13: **end for**

sponding to lines 16 and 18 in Algorithm 7. The GPU code retrieves `half` and `halfmod` as precomputed values from the CPU. This kernel also utilizes a 2-dimensional grid size, where the x-dimension comprises $n/1024$ blocks, and the y-dimension consists of 2 blocks, each with 1024 threads. To access the input data, the kernel employs the following pattern: $thread\_idx + (blockDim.y \times n)$. Here, $thread\_idx$ ranges from 0 to n. Additionally, the for loop in line 15 introduces a dependency. Therefore, this kernel also contains a single for loop that iterates $r-1$ times. Subsequently, this kernel is followed by the NTT kernels, as indicated in line 19.

Finally, the last kernel utilizes a 3-dimensional grid size, with the x-dimension having $n/1024$ blocks, the y-dimension having $r-1$ blocks, and the z-dimension consisting of 2 blocks, each with 1024 threads. This kernel implements the operations described in lines 20 to 22 of Algorithm 7. The result is two ciphertexts at the output.

4.3.4 Rescale

As mentioned in Chapter 2.6.1.1, the rescale operation is one of the most essential operations in the CKKS scheme. The algorithm for the rescale operation is referred to as `ModSwitch` as depicted in Algorithm 8. This algorithm closely resembles the relinearization operations described in lines 12 to 21 of Algorithm 7. This operation consists of 6 kernels: 4 of them are for the NTT/INTT operations, and 2 of them are for the rescale

operation.

The first two kernels take the 2 ciphertexts from the relinearization operation and convert the NTT domain to the polynomial domain via the INTT operation, as described in line 2 of Algorithm 8.

The next kernel executes lines 4 to 8. This kernel also utilizes the precomputed values `half` and `halfmod` from the CPU, performing addition with `half` and subtraction with `halfmod`. The kernel is structured with $(r-2) \times 2 \times (n/1024)$ blocks and 1024 threads per block. Within this kernel, the first and second components of the ciphertext, $c_i[0], c_i[1]$, are processed using an `if` branch, instead of the **for** loop in the algorithm. The `if` section involves threads from 0 to $(r-2) \times n$ for accessing the first ciphertext, while the `else` section involves threads from $(r-2) \times n$ to the last thread for accessing the second ciphertext. Subsequently, the kernel initiates computation at line 8.

The following set of 2 kernels takes the inputs from the previous kernel's output and transfers them to the NTT domain, as shown in line 9.

The last kernel finally performs subtraction and multiplication with the modular inverse of the last modulus. The modular inverse of the last modulus is calculated on the CPU, and the kernel takes it as a precomputed value. This kernel also consists of $(r-2) \times 2 \times (n/1024)$ blocks and 1024 threads per block, similar to the third kernel. Since the input sizes are the same, the memory access pattern also remains the same as the third kernel. After accessing the data via a single `if` branch (the branch is used to process the $k = 0$ and $k = 1$ values of the **for** loop in linen 7), the kernel initiates computation on lines 10 and 11. Subsequently, the decomposition modulus size is reduced by one at the output.

4.3.5 Rotation

This operation consists of 3 algorithms: `Apply Galois`, `Galois Elt`, and `Switch Key`. It is implemented using 12 kernels: 8 kernels for the NTT/INTT operations, 3 kernels for the `switchkey` operation, and 1 kernel for `Apply Galois` (Algorithm 9) and `Galois Elt` (Algorithm 10).

Since the `Apply Galois` algorithm operates on the polynomial domain, the first set of 2 kernels applies the inverse NTT operation.

The next kernel, which executes Algorithms 10 and 9, has a block size of $(r-1) \times 2 \times$

Algorithm 9 Apply Galois Algorithm

Input: $galois\_elt, {c_i}^j[k] \in \mathbf{R}_{q_i,n}$, where $0 \leq i < (r-1)$ $0 \leq j < n$ $k = 0,1$

Output: $\bar{c_i}^j[k] \in \mathbf{R}_{q_i}$

1: **for** i from 0 by 1 to $r-2$ **do**
2: **for** j from 0 by 1 to $n-1$ **do**
3: $index\_raw = j \times galois\_elt$
4: $index = index\_raw \,\&\, (n-1)$
5: **for** k from 0 by 1 to 1 **do**
6: $r\_value = {c_i}^j[k]$
7: **if** $(index\_raw \gg log_2(n)) \,\&\, 1$ **then**
8: $non\_zero = int(r\_value \neq 0)$
9: $r\_value = (q_i - r\_value) \,\&\, (-non\_zero)$
10: **end if**
11: $\bar{c_i}^j[k] = r\_value$
12: **end for**
13: **end for**
14: **end for**

$(n/1024)$, with each block containing 1024 threads. This makes the overall thread count equal to the total number of coefficients in all RNS bases of the ciphertext $c_i[0]$ and $c_i[1]$. Algorithm 10 takes an input `steps` that defines the number of rotations for the ciphertext and returns the `galois_elt` variable, which serves as input for Algorithm 9. The kernel allocates half the threads for the first component of the ciphertext, $c[0]$ and the other half for $c[1]$. Originally, the output returns the ciphertext coefficients ${c_i}^j[k]$ with a size of $(r-1) \times 2 \times n$, but for simplicity, this kernel returns 2 ciphertexts with sizes of $(r-1) \times n$.

Subsequently, the following operation is the `switchkey` algorithm. The following 5 sets of kernels are the same as those used in the relinearization's where the lines 6 to 19 in Algorithm 7. The last kernel is identical to the last kernel used in rescaling.

Algorithm 10 Galois Elt Algorithm

Input: $steps, n$
Output: $galois\_elt$

1: $m32 = n \times 2$
2: **if** $steps == 0$ **then**
3: **return** $m32 - 1$
4: **else**
5: $pop\_steps = abs(steps)$
6: **if** $steps < 0$ **then**
7: $steps = (n \gg 1) - pop\_steps$
8: **else**
9: $steps = pop\_steps$
10: **end if**
11: $gen = 3$
12: $galois\_elt = 1$
13: **for** i from 0 by 1 to $steps$ **do**
14: $galois\_elt = galois\_elt \times gen$
15: $galois\_elt = galois\_elt \,\&\, (m32 - 1)$
16: **end for**
17: **return** $galois\_elt$
18: **end if**

5. RESULTS

In this section, we present the results of our GPU implementation by conducting a comprehensive comparison with state-of-the-art works. Furthermore, we designed circuits to demonstrate the performance of our implementations. These circuits are designed with varying multiplicative depths to capture the computational requirements of applying homomorphic computation in real-life applications.

Table 5.1 provides an overview of our testbed environment. The GPU results are obtained using the NVIDIA RTX 3070 and 4090 GPUs. To ensure a fair comparison, we employed a powerful CPU, namely the AMD Ryzen 7 3800X.

Table 5.1 Hardware features of the Testbed environment

Feature	**CPU**	**GPU**	
		RTX3070	RTX 4090
Model	Ryzen7 3800X	RTX3070	RTX4090
Threads	16	5888	16384
Frequency	4.20 GHz	1920 MHz	2520 MHz
RAM	32 GB (3600 MHz)	12 GB	24 GB
Memory Type	-	GDDR6X	GDDR6X
Memory Bus	-	256 bits	384 bits
Bandwidth	-	504.2 GB/s	1,008 GB/s

CUDA version: 11.6.2

5.1 GPU Implementations of CKKS HE Operations Results and Comparison With State of the Art Works

There are limited prior works in the literature presenting GPU implementations of homomorphic operations within the CKKS scheme, and the existing ones often lack comprehensive performance results for all homomorphic operations, let alone results for complete homomorphic applications. Furthermore, our work focuses on achieving high levels of security, necessitating the execution of operations using large modulus sizes.

In Table 5.2, we provide GPU-based CKKS implementation results and compare them with state-of-the-art implementation in Microsoft SEAL library implementation of the CKKS scheme running on a CPU whose specifications are given in Table 5.1. One significant observation is that as the ring dimension increases so do the speedup values. For homomorphic multiplication, we break down timing results into separate categories: multiplication, relinearization, and rescale. We achieve impressive speedups of up to 264.65×, 161.07×, and 113× respectively.F or the homomorphic rotation operation, we attain a speedup of up to 121.1× when compared to CPU results.

We also compared our GPU implementations against those in the literature (Yang et al., 2023; Jung et al., 2021; Badawi et al., 2020; Shen et al., 2022), and summarized the results in Table 5.3. Note that the timing results in the literature for homomorphic multiplication account for both multiplication and relinearization operations while we report them separately. While Work Yang et al. (2023) exhibits faster results beyond a ring size of 2^{13}, it employs significantly smaller modulus sizes than our work. Consequently, we contend that our implementation achieves superior performance while maintaining much higher levels of noise budget. Similarly, in comparison to Work Badawi et al. (2020), our implementation remains faster despite utilizing larger moduli for ring dimensions 2^{13} and 2^{14}; namely, 218 and 438 bits modulus, respectively, compared to their 200 and 360 bits modulus.

In summary, our implementation delivers substantial speed improvements compared to CPU-based results. Moreover, it either outperforms or comparable to other existing works in the literature, particularly when aiming for high levels of nonise budget.

Table 5.2 Comparison of homomorphic operations' timing results of the SEAL CKKS scheme implementation on CPU and our GPU implementation. (times in microseconds)

			GPU Imp.		**SEAL CPU Imp.**	T
Operation	**n**	$\log_2 q$	RTX3070	RTX4090	CPU	T_s
Addition	2^{12}	109	4	3.42	11	3.22×
	2^{13}	218	5.1	3.66	42	11.48×
	2^{14}	438	12.3	4.32	184	42.59×
	2^{15}	881	44	7.14	750	105.04×
Multiplication	2^{12}	109	5.1	4.6	93	20.21×
	2^{13}	218	7.1	5.1	416	81.56×
	2^{14}	438	16.8	9.4	1583	168.4×
	2^{15}	881	67.5	23	6087	264.65×
Relinearization	2^{12}	109	67.5	50.1	565	11.28×
	2^{13}	218	98.3	60.12	2905	48.32×
	2^{14}	438	398.3	133.02	18150	136.45×
	2^{15}	881	2867.2	697.95	112415	161.07×
Rescaling	2^{12}	109	47.1	22.85	150	6.56×
	2^{13}	218	57.3	46.24	681	14.72×
	2^{14}	438	95.1	56.09	3191	56.89×
	2^{15}	881	328.7	112.48	12710	113×
Rotation	2^{12}	109	81.9	60.9	586	9.62×
	2^{13}	218	121.8	86.78	2921	33.66×
	2^{14}	438	481.2	194.24	18236	93.88×
	2^{15}	881	3444.7	939.2	113732	121.1×

T_s: CPU vs RTX 4090 speed up.

5.2 Circuit Design using the CKKS GPU Library with Varying

Multiplicative Depth

In order to assess the performance of our CKKS scheme implementation on GPU, we also designed some benchmark circuits to capture the computational complexity of real-world applications whose complexity increases with the depth of (Boolean) circuits. Here, the depth of a circuit is determined by the number of serially executed successive multiplication operations. Figure 5.1 shows a circuit design, that has a multiplicative depth of three. As shown in the figure, after each depth, ciphertext size decreased by 1 RNS modulus.

Table 5.3 Timing results for the CKKS operations GPU implementation compared to prior works in the literature (Times in microseconds)

Work	Device	n	$\log_2 q$	Add.	Mult.	Relin.	Resc.	Rot.
(Yang et al., 2023)	Tesla V100	2^{12}	70	5	139*	–	69	140
		2^{13}	105	5	125*	–	70	124
		2^{13}	142	5	149*	–	72	148
		2^{14}	259	6	178*	–	79	176
		2^{14}	260	6	203*	–	79	201
		2^{15}	500	24	382*	–	141	348
		2^{15}	550	31	484*	–	159	447
		2^{16}	820	75	1119*	–	374	1026
		2^{16}	980	89	1445*	–	437	1329
(Jung et al., 2021)	Tesla V100	2^{16}	1,693	162	2,960*	–	490	2,550
(Badawi et al., 2020)	Tesla V100	2^{13}	200	30$'$	400*	–	120	3740
		2^{14}	360	40$'$	740*	–	140	6090
		2^{15}	600	50$'$	2340*	–	270	17820
		2^{16}	1770	180$'$	33580*	–	1280	324900
		2^{16}	2300	300$'$	55880*	–	1630	444790
(Shen et al., 2022)	Tesla V100	2^{12}	–	–	240	–	–	–
		2^{13}	–	–	230	–	–	–
		2^{14}	–	–	310	–	–	–
		2^{15}	–	–	590	–	–	–
		2^{16}	–	–	1570	–	–	–
T.W.	RTX 3070	2^{12}	109	4	5.1	67.5	47.1	81.9
		2^{13}	218	5.1	7.1	98.3	57.3	121.8
		2^{14}	438	12.3	16.8	398.3	95.1	481.2
		2^{15}	881	44	67.5	2867.2	328.7	3444.7
	RTX 4090	2^{12}	109	3.42	4.6	50.1	22.85	60.9
		2^{13}	218	3.66	5.1	60.12	46.24	86.78
		2^{14}	438	4.32	9.4	133.02	56.09	194.24
		2^{15}	881	7.14	23	697.95	112.48	939.2

$\star$: Homomorphic multiplication + relinearization operation result.
$'$: Homomorphic addition with ciphertext and plaintext.
T.W.: This work.

Table 5.4 displays the timing results for various multiplicative depths achieved by our GPU CKKS library implementation in this thesis in comparison with the GPU BFV implementation in the work by Şah Özcan et al. (2022), and SEAL's CPU-based CKKS and BFV implementations. The provided timings represent the homomorphic evaluations of the entire circuit and have been obtained using the RTX 3070 GPU. As there is no prior work involving GPU implementation and circuit design, our primary point of comparison

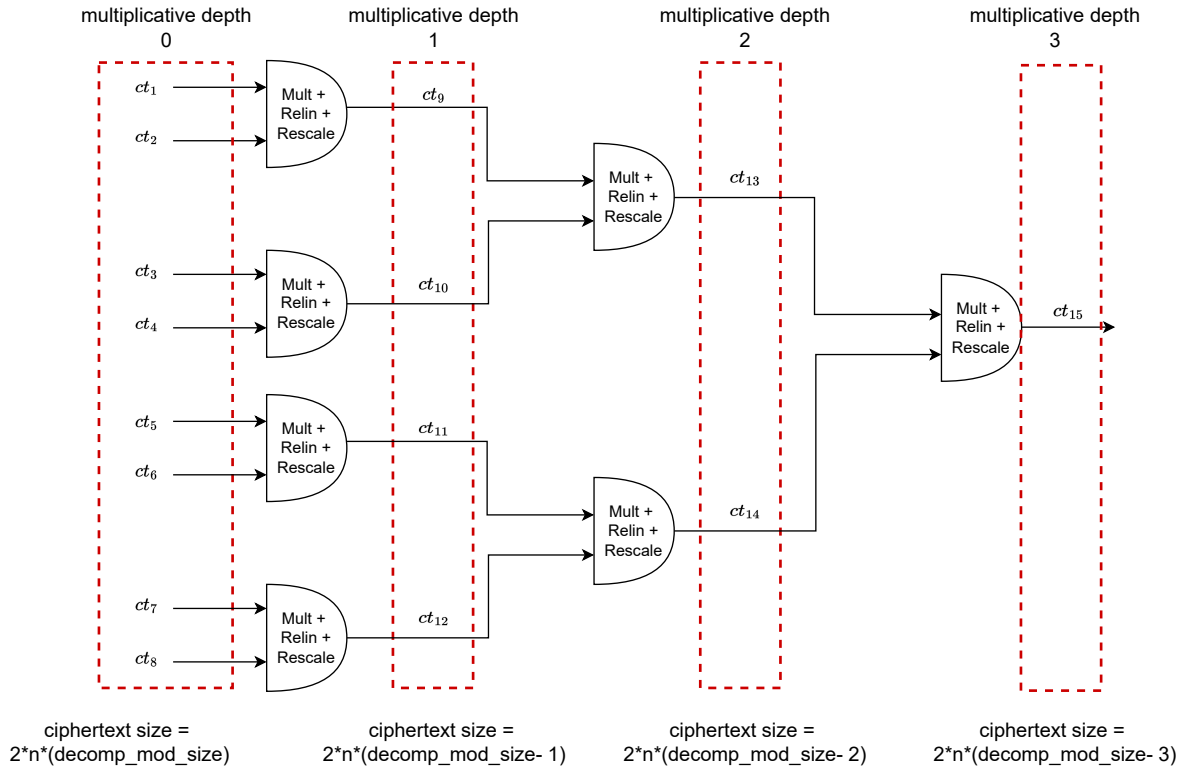


Figure 5.1 Circuit design and ciphertext size comparison for 3 multiplicative depth.

is the BFV GPU implementation and CKKS CPU implementation.

For the circuit with the multiplicative depth of four, we achieved timing results that closely resembled those of the GPU BFV circuit design, with a ring size of 2^{13} and a 218-bit modulus. In this case, we utilized a set of RNS moduli $\{40, 33, 33, 33, 33, 40\}$ bits and a 33-bit scaling factor. Beyond the fifth and sixth depth, the BFV scheme is unable to perform additional multiplications with this ring dimension. However, it is worth noting that by decreasing the bit sizes of each RNS moduli and increasing their quantity, we can extend the depth of the circuit to 5 and 6 with the CKKS scheme. As demonstrated in Table 5.4, we can achieve computation for circuits with the multiplicative depth of five using a set of RNS moduli consisting of $\{37, 30, 30, 30, 30, 30, 30\}$ bits, resulting in a remarkable 27.81 times speedup compared to the CPU single thread implementation. Furthermore, for circuits with the multiplicative depth of six, $\{35, 26, 26, 26, 26, 26, 26, 26\}$ bits set of RNS moduli enabled a 24.07 times speedup.

Our results demonstrate the adaptability of our GPU library for the design of complex circuits, making it a valuable tool for accelerating applications that employ the CKKS scheme.

Table 5.4 Comparison between the SEAL CKKS and BFV scheme implementations, our CKKS GPU implementation, and the BFV implementation from the work by Şah Özcan et al. (2022) (Times are in milliseconds taken using RTX 3070).

					CPU				
	Depth	**n**	$\log_2 q$	$\mathcal{P}$	S.T	M.T.	GPU	$\mathcal{S}_{S.T.\rightarrow GPU}$	$\mathcal{S}_{M.T.\rightarrow GPU}$
CKKS	4.	2^{13}	218	33	76	23.6	2.95	25.76	8
	5.	2^{13}	218	30	183	43.8	6.58	27.81	6.65
	6.	2^{13}	218	26	467.74	87	23.42	24.07	3.71
BFV	4.	2^{13}	218	–	198.3	82.7	2.82	70.32	29.32
	5.	2^{14}	218	–	1958.7	516	29.62	66.13	17.42
	6.	2^{14}	218	–	4021.1	873.3	59.88	67.15	14.58

S.T.: Single thread
M.T.: Multi-thread
$\mathcal{S}_{S.T.\rightarrow GPU}$: Speed up between single thread CPU and GPU.
$\mathcal{S}_{M.T.\rightarrow GPU}$: Speed up between multi-thread CPU and GPU.
$\mathcal{P}$: Precision.

6. CONCLUSION

In this chapter, we present a concise summary and conclusions of our work, along with potential avenues for future research.

Within this thesis, we introduce a GPU library featuring highly parallelized and optimized implementations of homomorphic operations designed for the CKKS scheme. Furthermore, our library seamlessly integrates with the Microsoft SEAL library. These GPU-accelerated functions are accessible from any application code through SEAL. Consequently, this library serves as an accelerator for homomorphic encryption applications relying on the CKKS scheme.

To achieve the most competitive timing results available in current literature, we streamlined the kernel function calls and optimized GPU memory usage. Our implementation and timing results affirm that using a GPU as an accelerator represents a sound approach for the efficient execution of homomorphic encryption operations. For instance, our GPU implementation outperforms the CPU implementation by factors of 105.04, 264.65, 161.07, 113, and 121.1 for homomorphic addition, homomorphic multiplication, relinearization, homomorphic rescale, and homomorphic rotation operations, respectively, using a ring dimension of 2^{15} and an 881-bit modulus via an RTX 4090 GPU.

We also examined the behavior of CKKS scheme operations for homomorphic evaluation of circuits with various multiplicative depths. Furthermore, we devised several benchmark circuits to represent different real-life applications of homomorphic encryption and presented outcomes at different multiplicative depths, which we subsequently compared with CPU implementations and the BFV GPU implementation detailed in the work by Şah Özcan et al. (2022). Our findings indicate that we can achieve speedups of up to 27.81 when compared to a CPU implementation. Additionally, our benchmark circuits

showed the relation between precision, speed up, and security level.

The reported timing results from our circuit designs underscore the library's efficacy as an accelerator for applications ranging from simple to highly complex.

Given that this work focuses on somewhat homomorphic encryption (SWHE), we consider the bootstrapping operation as a potential avenue for future research, particularly to implement fully homomorphic encryption (FHE).

BIBLIOGRAPHY

(2021). PALISADE Lattice Cryptography Library (release 1.11.5). https://palisade-crypto.org/.

Al Badawi, A., Polyakov, Y., Aung, K., Veeravalli, B., and Rohloff, K. (2019). Implementation and performance evaluation of rns variants of the bfv homomorphic encryption scheme. *IEEE Transactions on Emerging Topics in Computing*, PP:1–1.

Anggriane, S. M., Nasution, S. M., and Azmi, F. (2016). Advanced e-voting system using paillier homomorphic encryption algorithm. In *2016 International Conference on Informatics and Computing (ICIC)*, pages 338–342.

Badawi, A. A., Bates, J., Bergamaschi, F., Cousins, D. B., Erabelli, S., Genise, N., Halevi, S., Hunt, H., Kim, A., Lee, Y., Liu, Z., Micciancio, D., Quah, I., Polyakov, Y., R.V., S., Rohloff, K., Saylor, J., Suponitsky, D., Triplett, M., Vaikuntanathan, V., and Zucca, V. (2022). Openfhe: Open-source fully homomorphic encryption library. Cryptology ePrint Archive, Paper 2022/915. https://eprint.iacr.org/2022/915.

Badawi, A. A., Hoang, L., Mun, C. F., Laine, K., and Aung, K. M. M. (2020). PrivFT: Private and fast text classification with homomorphic encryption. *IEEE Access*, 8:226544–226556.

Badawi, A. A. and Polyakov, Y. (2023). Demystifying bootstrapping in fully homomorphic encryption. Cryptology ePrint Archive, Paper 2023/149. https://eprint.iacr.org/2023/149.

Bajard, J., Eynard, J., Hasan, M. A., and Zucca, V. (2016). A full RNS variant of FV like somewhat homomorphic encryption schemes. In Avanzi, R. and Heys, H. M., editors, *Selected Areas in Cryptography - SAC 2016 - 23rd International Conference, St. John's, NL, Canada, August 10-12, 2016, Revised Selected Papers*, volume 10532 of *Lecture Notes in Computer Science*, pages 423–442. Springer.

Barrett, P. (1986). Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings*, volume 263 of *Lecture Notes in Computer Science*, pages 311–323. Springer.

Bernstein, D. and Lange, T. (2017). Post-quantum cryptography. *Nature*, 549(7671):188–194.

Brakerski, Z., Gentry, C., and Vaikuntanathan, V. (2012). (leveled) fully homomorphic encryption without bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, page 309–325, New York, NY, USA. Association for Computing Machinery.

Brakerski, Z. and Vaikuntanathan, V. (2011). Efficient fully homomorphic encryption

from (standard) lwe. In *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*, pages 97–106.

Cheon, J., Kim, A., Kim, M., and Song, Y. (2017). Homomorphic encryption for arithmetic of approximate numbers. pages 409–437.

Cheon, J. H. and Kim, J. (2015). A hybrid scheme of public-key encryption and somewhat homomorphic encryption. *IEEE Transactions on Information Forensics and Security*, 10(5):1052–1063.

Chillotti, I., Gama, N., Georgieva, M., and Izabachène, M. (2016). Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In Cheon, J. H. and Takagi, T., editors, *Advances in Cryptology – ASIACRYPT 2016*, pages 3–33, Berlin, Heidelberg. Springer Berlin Heidelberg.

Chillotti, I., Gama, N., Georgieva, M., and Izabachène, M. (2018). Tfhe: Fast fully homomorphic encryption over the torus. *IACR Cryptology ePrint Archive*, 2018:421.

Chu, E. and George, A. (1999). *Inside the FFT black box: serial and parallel fast Fourier transform algorithms*. CRC press.

Cooley, J. W. and Tukey, J. W. (1965). An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301.

Cortés-Mendoza, J. M., Radchenko, G., Tchernykh, A., Pulido-Gaytan, B., Babenko, M., Avetisyan, A., Bouvry, P., and Zomaya, A. (2021). Lr-gd-rns: Enhanced privacy-preserving logistic regression algorithms for secure deployment in untrusted environments. In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 770–775.

Dai, W. and Sunar, B. (2016). cuHE: A Homomorphic Encryption Accelerator Library. In Pasalic, E. and Knudsen, L. R., editors, *Cryptography and Information Security in the Balkans*, pages 169–186, Cham. Springer International Publishing.

Dennard, R., Gaensslen, F., Yu, H.-N., Rideout, V., Bassous, E., and LeBlanc, A. (1974). Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268.

Doröz, Y., Öztürk, E., Savaş, E., and Sunar, B. (2015). Accelerating ltv based homomorphic encryption in reconfigurable hardware. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 185–204. Springer.

Doröz, Y., Hu, Y., and Sunar, B. (2015). Homomorphic aes evaluation using the modified ltv scheme. *Designs, Codes and Cryptography*, 80.

Ducas, L. and Micciancio, D. (2015). Fhew: Bootstrapping homomorphic encryption in less than a second. pages 617–640.

Fan, J. and Vercauteren, F. (2012). Somewhat practical fully homomorphic encryption. *IACR Cryptol. ePrint Arch.*, 2012:144.

Feng, X., Li, S., and Xu, S. (2019). RLWE-Oriented High-Speed Polynomial Multiplier

Utilizing Multi-lane Stockham NTT Algorithm. *IEEE Transactions on Circuits and Systems II: Express Briefs*, pages 1–1.

FUQUA (2023). More than 80 percent of firms say they have been hacked. *https://cfosurvey.fuqua.duke.edu/press-release/more-than-80-percent-of-firms-say-they-have-been-hacked*.

Garner, H. L. (1959). The residue number system. *IRE Transactions on Electronic Computers*, EC-8(2):140–147.

Gentry, C. (2009). Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*, STOC '09, page 169–178, New York, NY, USA. Association for Computing Machinery.

Gentry, C. and Halevi, S. (2011). Implementing gentry's fully-homomorphic encryption scheme. In Paterson, K. G., editor, *Advances in Cryptology – EUROCRYPT 2011*, pages 129–148, Berlin, Heidelberg. Springer Berlin Heidelberg.

Halevi, S. and Shoup, V. (2014). Algorithms in helib. In Garay, J. A. and Gennaro, R., editors, *Advances in Cryptology – CRYPTO 2014*, pages 554–571, Berlin, Heidelberg. Springer Berlin Heidelberg.

Hamood, M. (2016). New decimation-in-time fast hartley transform algorithm. *International Journal of Electrical and Computer Engineering (IJECE)*, 6:1654.

Harris, F. (1978). On the use of windows for harmonic analysis with the discrete fourier transform. *Proceedings of the IEEE*, 66(1):51–83.

Harvey, D. (2014). Faster arithmetic for number-theoretic transforms. *Journal of Symbolic Computation*, 60:113–119.

ITU (2022). Measuring digital development: Facts and figures 2022. *https://www.itu.int/en/ITU-D/Statistics/Pages/facts/default.aspx*.

Jung, W., Kim, S., Ahn, J. H., Cheon, J. H., and Lee, Y. (2021). Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with gpus. Cryptology ePrint Archive, Paper 2021/508. https://eprint.iacr.org/2021/508.

Li, F., Luo, B., and Liu, P. (2010). Secure information aggregation for smart grids using homomorphic encryption. In *2010 First IEEE International Conference on Smart Grid Communications*, pages 327–332.

Longa, P. and Naehrig, M. (2016). Speeding up the Number Theoretic Transform for Faster Ideal Lattice-Based Cryptography. In Foresti, S. and Persiano, G., editors, *Cryptology and Network Security*, pages 124–139, Cham. Springer International Publishing.

López-Alt, A., Tromer, E., and Vaikuntanathan, V. (2012). On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In *Proceedings of the Forty-Fourth Annual ACM Symposium on Theory of Computing*, STOC '12, page 1219–1234, New York, NY, USA. Association for Computing Machinery.

Lyubashevsky, V., Peikert, C., and Regev, O. (2013). On ideal lattices and learning with errors over rings. *J. ACM*, 60(6).

Mert, A. C., Öztürk, E., and Savaş, E. (2020). Design and implementation of encryption/decryption architectures for bfv homomorphic encryption scheme. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 28:353–362.

Mert, A. C., Öztürk, E., and Savaş, E. (2019). Design and implementation of a fast and scalable ntt-based polynomial multiplier architecture. In *2019 22nd Euromicro Conference on Digital System Design (DSD)*, pages 253–260.

Montgomery, P. L. (1985). Modular multiplication without trial division. *Mathematics of Computation*, 44:519–521.

Moore, G. E. (1965). Cramming more components onto integrated circuits. *Electronics*, 38(8).

Munjal, K. and Bhatia, R. (2022). A systematic review of homomorphic encryption and its contributions in healthcare industry. *Complex & Intelligent Systems*, 9:1–28.

Nejatollahi, H., Cammarota, R., and Dutt, N. (2019). Flexible ntt accelerators for rlwe lattice-based cryptography. In *2019 IEEE 37th International Conference on Computer Design (ICCD)*, pages 329–332.

NVIDIA (2021). Nvidia ampere ga102 gpu architecture.

NVIDIA Corporation (2010). NVIDIA CUDA C programming guide. Version 3.2.

Paillier, P. (1999). Public-key cryptosystems based on composite degree residuosity classes. In Stern, J., editor, *Advances in Cryptology — EUROCRYPT '99*, pages 223–238, Berlin, Heidelberg. Springer Berlin Heidelberg.

Phong, L. T., Aono, Y., Hayashi, T., Wang, L., and Moriai, S. (2018). Privacy-preserving deep learning via additively homomorphic encryption. *IEEE Transactions on Information Forensics and Security*, 13(5):1333–1345.

Pollard, J. M. (1971). The fast Fourier transform in a finite field. *Mathematics of computation*, 25(114):365–374.

Potey, M. M., Dhote, C., and Sharma, D. H. (2016). Homomorphic encryption for security of cloud data. *Procedia Computer Science*, 79:175–181. Proceedings of International Conference on Communication, Computing and Virtualization (ICCCV) 2016.

Rivest, R. L., Shamir, A., and Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126.

Scott, M. (2017). A note on the implementation of the number theoretic transform. Cryptology ePrint Archive, Paper 2017/727. https://eprint.iacr.org/2017/727.

SEAL (2020). Microsoft SEAL (release 3.6). https://github.com/Microsoft/SEAL. Microsoft Research, Redmond, WA.

Shen, S., Yang, H., Liu, Y., Liu, Z., and Zhao, Y. (2022). Cuda-accelerated rns multiplication in word-wise homomorphic encryption schemes. Cryptology ePrint Archive, Paper 2022/633. https://eprint.iacr.org/2022/633.

Shor, P. (1994). Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134.

Su, Y., Yang, B.-L., Yang, C., Yang, Z.-P., and Liu, Y.-W. (2022). A highly unified reconfigurable multicore architecture to speed up ntt/intt for homomorphic polynomial multiplication. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 30(8):993–1006.

Wang, W., Chen, Z., and Huang, X. (2014). Accelerating leveled fully homomorphic encryption using gpu. In *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 2800–2803.

Wu, W., Liu, J., Wang, H., Hao, J., and Xian, M. (2021). Secure and efficient outsourced k-means clustering using fully homomorphic encryption with ciphertext packing technique. *IEEE Transactions on Knowledge and Data Engineering*, 33(10):3424–3437.

Yang, H., Shen, S., Dai, W., Zhou, L., Liu, Z., and Zhao, Y. (2023). Implementing and benchmarking word-wise homomorphic encryption schemes on gpu. Cryptology ePrint Archive, Paper 2023/049. https://eprint.iacr.org/2023/049.

Şah Özcan, A., Ayduman, C., Türkoğlu, E. R., and Savaş, E. (2022). Homomorphic encryption on gpu. Cryptology ePrint Archive, Paper 2022/1222. https://eprint.iacr.org/2022/1222.

Şah Özcan, A. and Savaş, E. (2023). Two algorithms for fast gpu implementation of ntt. Cryptology ePrint Archive, Paper 2023/1410. https://eprint.iacr.org/2023/1410.