A GPU LIBRARY FOR BFV HOMOMORPHIC ENCRYPTION SCHEME VIA THREE DIFFERENT NTT ALGORITHMS

by
Ali Şah Özcan

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

© 2023 by Ali Şah Özcan.
All Rights Reserved.

ABSTRACT

A GPU LIBRARY FOR BFV HOMOMORPHIC ENCRYPTION SCHEME VIA THREE DIFFERENT NTT ALGORITHMS

ALI ŞAH ÖZCAN

ELECTRONIC AND ENGINEERING MSC. THESIS, OCTOBER 2023

Thesis Advisor: Prof. Erkay Savaş

Keywords: Lattice Based Cryptography, Homomorphic Encryption, Number Theoretic Transform (NTT), GPU, Parallel Processing, Secure Computation

Homomorphic encryption (HE) is a cryptosystem that allows the secure processing of encrypted data. One of the most popular HE schemes is the Brakerski-Fan-Vercauteren (BFV), which supports somewhat (SWHE) and fully homomorphic encryption (FHE). Since overly involved arithmetic operations of HE schemes are amenable to concurrent computation, GPU devices can be instrumental in facilitating the practical use of HE in real world applications thanks to their superior parallel processing capacity.

We propose an optimized and highly parallelized GPU library to accelerate the BFV scheme. The BFV scheme is based on lattice-based cryptography and involves overly many polynomial multiplications of very high degrees and multiprecision coefficients. Since, schoolbook polynomial multiplication is inefficient for GPU at high degrees, firstly, we present three different Number Theoretic Transform (NTT) implementations based on Merge NTT and 4-Step NTT algorithms that are optimized for GPUs instead of using schoolbook multiplicaition. The best of these three implementations employs an approach i) to optimize the number of accesses to slow global memory for thread synchronization, and ii) to make better use of spatial locality in global memory accesses. It turns out that by controlling certain parameters in CUDA platform for general-purpose GPU computing (GPGPU) such as kernel count, block size and block shape, we can affect the performance of NTT. To the best of our knowledge, this work is unique for it suggests a recipe for selecting optimum CUDA parameters to obtain the best NTT performance for a given polynomial degree. Our implementation results on various GPU devices for all power-of-two polynomial degrees from 2^{12} to 2^{28} show that our algorithms compare favorably with the other state-of-the-art GPU implementations in the literature with the optimum selection of these three CUDA parameters. Furthermore, NTT can also be used in Zero-Knowledge systems apart from Homomorphic Encryption because it can work with high ring sizes.

The library also improves the performance of the homomorphic operations of the BFV scheme. Although the library can be independently used, it is also fully integrated with the Microsoft SEAL library, which is a well-known HE library that also implements the BFV scheme. For one ciphertext multiplication, for the ring dimension 2^{14} and the modulus bit size of 438, our GPU implementation offers 361.6 times speedup over the SEAL library running on a high-end CPU. The library compares favorably with other state-of-the-art GPU implementations of NTT and BFV operations.

Finally, we implement a privacy-preserving application that classifies encrypted genome data for tumor types and achieves speedups of 237.88 and 36.08 over CPU implementations using single and 16 threads, respectively.

Our results indicate that GPU implementations can facilitate the deployment of homomorphic cryptographic libraries in real-world privacy-preserving applications.

ÖZET

ÜÇ FARKLI HIZLANDIRILMIŞ NTT ALGORTIMASI KULLANARAK BFV HOMOMORFIK ŞIFRELEME ŞEMASI IÇIN BIR GPU KÜTÜPHANESI GELIŞTIRILMESI

ALI ŞAH ÖZCAN

ELEKTRONİK VE MÜHENDİSLİĞİ YÜKSEK LİSANS TEZİ, EKIM 2023

Tez Danışmanı: Prof. Dr. Erkay Savaş

Anahtar Kelimeler: A,B,C,D Keywords: Kafes-tabanlı Kriptogragi, Homomorfik Şifreleme, Sayılar Teorisi Dönüşümü (NTT), GPU, Paralel İşleme, Güvenli Hesaplama

Homomorfik şifreleme (İngilizcesi, homomorphic encryption, HE), şifrelenmiş verilerin güvenli bir şekilde işlenmesini sağlayan bir şifreleme sistemidir. En popüler HE şemalarından biri, sınırlı (İngilizcesi, somewhat homomorphic encryption, SWHE) ve tamamen homomorfik şifrelemeyi (İnglizcesi, fully homomorphic encryption, FHE) destekleyen Brakerski-Fan-Vercauteren (BFV) adı verilen şifreleme sistemidir. HE şemalarının karmaşık aritmetik işlemleri eşzamanlı hesaplamaya uygun olduğundan, grafik işlemci (GPU) cihazları, üstün paralel işleme kapasiteleri sayesinde HE algoritmalarının gerçek dünya uygulamalarında pratik kullanımını kolaylaştırmada etkili olabilir.

Bu tezde, BFV şemasını hızlandırmak için optimize edilmiş ve yüksek derecede paralelleştirilmiş bir GPU kütüphanesi öneriyoruz. BFV şeması, kafes tabanlı kriptografik (İngilizcesi, lattice-based encryption) sistemlere dayalıdır ve çok sayıda, katsayıları çok büyük tamsayılar olan yüksek dereceli polinom çarpmalarını içerir. GPU gerçeklemelerinde yüksek dereceli polinomları çarpmak için klasik polinom çarpma yöntemi kullanıldığında çok verimsiz olduğundan, klasik yöntem yerine GPU cihazları için eniyilenmiş ve bir çeşit hızlı Fourier dönüşümü olan ve tam sayılar üzerinde çalışan sayılar teorisi dönüşümünü (İngilizcesi, number theoretic transformation, NTT) kullanan yöntemler tercih edilir. NTT işlemini hızlı gerçekleyebilmek için temel olarak iki yöntem vardır: i) birleştirilmiş yinelemeli NTT ve ii) 4-adımlı NTT yöntemleri. Bu tezde, bu iki yöntemi GPU için uyarlayarak üç farklı algoritma sunuyoruz. Bu tezde de görüleceği üzere, bu üç algoritma içinde en iyisi, i) iş parçacığı senkronizasyonu için yavaş ana belleğe erişim sayısını eniyileyip ve ii) ana bellek erişimlerinde mekânsal yerellikten daha iyi yararlanan algoritmadır. Bahsi geçen NTT algoritması, GPU üzerinde genel amaçlı yazılımlar geliştirilmesine olanak sağlayan CUDA platformunda çekirdek sayısı, blok boyutu ve blok şekli gibi belirli parametreleri kontrol ederek NTT başarımını olumlu yönde etkileyebilmektedir. Bilgimiz dahilinde olduğu kadar, bu çalışma, verilen bir polinom derecesi için en iyi NTT performansını elde etmek için en elverişli CUDA parametrelerini seçmek için bir izlek öneren benzersiz bir çalışmadır. Literatürde bilinen en iyi GPU uygulamaları ile karşılaştırıldığında, 2^{12} ila 2^{28} arasındaki ikinin kuvveti olan tüm polinom dereceleri için çeşitli GPU cihazlarından elde ettiğimiz gerçekleme sonuçlarımız, algoritmamızın üstün bir başarım sağladığını göstermektedir. Üstelik geliştirdiğimiz NTT algoritması, çok yüksek halka boyutlarıyla çalışabildiğinden homomorfik şifreleme dışında sıfır bilgi ispatı adı verilen kriptografik protokollerde de kullanılabilir.

Geliştirdiğimiz GPU yazılım kütüphanesi, ayrıca BFV şemasının homomorfik işlemlerinin başarımının da arttırılmasını sağlamaktadır. Geliştirdiğimiz kütüphane bağımsız olarak kullanılabildiği gibi, BFV sisteminin yanısıra diğer HE sistemlerinin de yazılım gerçeklemelerini içeren ve çok kullanınılan Microsoft SEAL kütüphanesiyle tamamen entegre bir şekilde de kullanılabilir. 2^{14} halka boyutu ve 438 bit uzunluğunda bir modül boyutu kullanıldığında, GPU gerçeklememiz bir şifreli metin çarpımı için, üst düzey bir CPU üzerinde çalışan SEAL kütüphanesine göre $361,6$ kat hızlanma sağlar. Geliştirilen GPU Kütüphanesi, NTT ve BFV işlemleri karşılatırıldığında, literatürdeki benzer GPU gerçeklemelerine göre daha hızlıdır.

Son olarak, GPU gerçeklememizi tümör türlerinin tespiti için şifrelenmiş genom verilerini sınıflandıran bir uygulamada denedik. GPU uygulamamız, bir ve 16 iş parçacığı kullanan CPU uygulamalarına göre, sırasıyla, 237.88 ve 36.08'lık hız artışları sağlamıştır.

Sonuçlarımız, GPU gerçeklemelerimiz yardımıyla, homomorfik şifreleme sistemlerinin gerçek dünyadaki mahremiyet korumalı uygulamaların kullanımının yolunun açılmasına yardımcı olacağını göstermektedir.

ACKNOWLEDGMENT

Throughout the course of this thesis, I am indebted to numerous individuals for their invaluable assistance and support. Foremost among them, I would like to express my gratitude to my advisor, Prof Erkay Savaş, for his unwavering guidance, mentorship, support, and patience throughout the duration of my research. Completing this study without his expertise and recommendations would have been impossible. Furthermore, I wish to extend my appreciation to him for consistently offering guidance and serving as a source of inspiration for my research endeavors and prospective career opportunities.

I would like to convey my sincerest appreciation to Dr. Erdinç Öztürk as well. Although he no longer holds the formal role of my advisor, I wish to express my gratitude for the guidance and support he extended throughout the course of my research. Furthermore, I am thankful for his commitment to offering guidance and for his contributions to potential career prospects.

I wish to extend my heartfelt appreciation to the esteemed members of my thesis committee, Prof. Sıddıka Berna Örs Yalçın, and Prof. Kamer Kaya, for their dedicated commitment of time, labor, and invaluable critiques pertaining to my thesis.

I wish to extend my thanks to Enes, a long-standing friend, and many others for their invaluable friendships. Their support has facilitated this journey for me.

Despite my occasional neglect of my girlfriend Simge during this period, I earnestly dedicate this acknowledgment to her for the consistent support she offered during these trying times. I wish to extend my thanks for her unswerving backing, motivation, and patience.

I would like to express my gratitude to my mother, Şadiye, and my father, Mustafa. Without a doubt, their unconditional love and support were instrumental in bringing me to where I am today. I am thankful for all their efforts, and I will continue to strive to do my best throughout my life to not disappoint them.

Lastly, I would like to express my gratitude to Sabancı University for providing me with scholarship support throughout my studies, to the Scientific and Technological Research Council of Turkey (TÜBİTAK). I also would like to thank Ulvetanna Company and its valuable employees for providing me with a job opportunity during the one-year period of my master's program.

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 Şadiye & Mustafa, my brother Melikşah, and my beloved girlfriend Simge.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

1. INTRODUCTION

Fully Homomorphic Encryption (FHE) enables computation over encrypted data, which had been considered as the most sought-after cryptographic primitive for many years. In (Gentry, 2009), Gentry proposed the first functional FHE scheme, which is described over ideal lattices and permits the homomorphic evaluation of arbitrary circuits. Later, more practicable schemes based on learning with errors problem over rings (RLWE) (Lyubashevsky et al., 2013) were proposed, where plaintext and ciphertext messages are represented as polynomials and ciphertext contains “noise”, which, increases as homomorphic operations are applied. Thus, the scheme has a noise budget sufficient only for a certain number of homomorphic operations; and if noise reaches a certain limit, the homomorphic property will not hold and the ciphertext message does not decrypt correctly due to excessive noise. This scheme is, thus, aptly called somewhat homomorphic encryption (SHE). To continue with the homomorphic operations, a technique referred as bootstrapping was proposed originally by Gentry (Gentry, 2009), whereby the ciphertext is homomorphically decrypted to obtain a ciphertext with a replenished noise budget. This process can be applied repeatedly to obtain a fully homomorphic scheme, but bootstrapping is generally deemed to be a prohibitively expensive operation.

The first implementation of an FHE scheme was realized by Gentry and Halevi, as explained in (Gentry and Halevi, 2011). Then, several FHE realizations were introduced such as those in (Brakerski and Vaikuntanathan, 2014) and (Brakerski et al., 2014). One of the most promising approaches is the Brakerski-Fan-Vercauteren (BFV) scheme (Fan and Vercauteren, 2012a), and there are several practical implementations of this and other similar schemes such as those provided by well-known software libraries SEAL (Microsoft, 2020), PALISADE (PALISADE, 2021), and HELib (Halevi and Shoup, 2014).

However, due to their compute-intensive operations in involved mathematical structures, current FHE implementations are far from being easily deployable in practice such as in large-scale practical cloud applications. Besides algorithmic optimizations and theoretical advances, using hardware accelerators is also the most viable option for bridging the gap between FHE performance and the requirements of real-world applications. GPU, FPGA and ASIC architectures can be profitably utilized as accelerators (Wang et al., 2014) (Mert et al., 2020b) (Doröz et al., 2015), to push the boundaries of FHE performance. A recently announced software library (Badawi et al., 2022) provides support for hardware acceleration integration to software implementations of HE schemes using a standard Hardware Abstraction Layer (HAL).

New generation FPGA devices, with extra high bandwidth memory communication and relatively low-energy consumption, stand one of the best candidates for acceleration. GPU devices, on the other hand, can also be profitably utilized in acceleration of many cryptographic primitives due to their extraordinary computational resources, easy integration with software libraries, and superior general-purpose computing capabilities.

Current GPU devices consist of thousands of parallel running threads and high bandwidth memory hierarchy featuring on-chip and off-chip memory. Computations intended to run on GPU are performed by invoking GPU kernel functions, and applications running on a host CPU device can call many kernels to offload some of its computation to GPU. Invoking each kernel function incurs an overhead in execution time due to the fact that access to off-chip memory is required for thread synchronization. The threads are grouped into blocks, whose size and shape are configurable and the threads in the same group enjoy faster synchronization. Various factors such as the number of kernels, block size, and block shape may have major impact on the performance of the application. Therefore, developing efficient GPU applications necessitates novel algorithm design and systematic approach in addition to code optimization specific to GPU architectures.

In this thesis, we present algorithms and implementation techniques to accelerate the BFV scheme of the SEAL library via NVIDIA GPUs. Our implementation developed in computing Unified Device Architecture (CUDA) program model (NVIDIA Corporation, 2010) accelerates all homomorphic operations in the BFV scheme utilizing various parallelization strategies that can be applied on GPU architectures. To the best of our knowledge, ours is the first work, in which the entire SEAL BFV scheme (including addition, multiplication, relinearization, and rotation operations) can be offloaded onto GPU. We provide a GPU library for the BFV homomomorphic encryption scheme, which can be used as a standalone application or integrated with the SEAL library to accelerate chosen homomorphic operations. The fully GPU-operated version of the library is publicly available on Github[1]. Our implementations used in the library achieves a very high level of parallelization on GPU, targeting the compute-intensive nature of FHE operations.

The polynomial multiplication (e.g., the multiplication in polynomial rings $\mathbf{R}_q = \mathbb{Z}_q/\Phi(x)$, where $\Phi_n(x)$ is the cyclotomic polynomial of degree n), is (one of) the most time and resource consuming operation in both homomorphic encryption and modern zero-knowledge proof schemes such as zk-SNARK (Bitansky et al., 2012; Ben-Sasson et al., 2013). One widely adopted method for fast and memory efficient polynomial multiplication is based on number theoretic transform (NTT) (Agarwal and Burrus, 1974), where its fast implementations can be achieved via hardware acceleration on GPU (Özerk et al., 2022; Özcan et al., 2023; Shivdikar et al., 2022; Kim et al., 2020; Dai and Sunar, 2015; Zheng, 2020; Goey et al., 2021) and FPGA devices (Mert et al., 2022; Derya et al., 2021; Mert et al., 2020a; Hirner et al., 2023; Riazi et al., 2020; Zhao et al., 2023). Besides academia, industry is also keenly interested in the acceleration of the cryptographic primitives and organize the competitions to promote interest therein[2].

Fortunately, there is a good deal of room for algorithmic research to accelerate the polynomial multiplication by utilizing the inherent parallelism in the operation and the hardware infrastructures (FPGA, ASIC, GPU) to exploit them in different ways. GPU architectures support many concurrent threads, which can be employed to perform the multiplication of very high-degree polynomials. Therefore, the noise budget can be made sufficiently large to homomorphically evaluate relatively complex circuits without having to use the bootstrapping method.

Here, we present a full GPU implementation for homomorphic operations of the BFV scheme and show that it can be used to accelerate real-world applications significantly. Our work introduces 3 different NTT implementation based on Merge (Cooley and Tukey, 1965) and 4-Step NTT (Bailey, 1989) tehcniques for polynomial multiplication adapted to GPU architecture and one of them proves to be the fastest in comparison to those reported in the literature to the best of our knowledge. We can summarize our contributions in this thesis as follows:

- We propose three algorithms for efficient computation of NTT designed to fully exploit the outstanding parallel computing capabilities of GPU with carefully optimized memory access patterns. One of the algorithms is a form of well-known recursive algorithm while the other based on the Four-Step algorithm. We aim two degree ranges for the polynomials: i) $n \in [2^{12}, 2^{16}]$ intended for homomorphic

[1] https://github.com/Alisah-Ozcan/HE_GPU

[2] https://www.zprize.io/

encryption applications, and ii) $n \in [2^{20}, 2^{28}]$ for the zk-SNARK protocol. The algorithms are flexible and parametric as they can easily be adapted to work with any value of n provided that it is a power of two.

- We show that the performance of the algorithms is highly dependent on the selection of parameters such as the number of kernels, block size and block shape. We, then, propose a systematic approach to find out their optimum selection for the fastest implementation given a polynomial degree. The approach helps determine the selection of a specific parameter by considering the interplay of several parameters to improve computational aspects of an implementation such as fast access to global memory.
- We implement both algorithms on various GPU devices using all possible optimization techniques and present our implementation results for time efficiency. We provide both latency and throughput results, which suggest the performance of each algorithm varies depending on the ring dimension as well as the specific GPU device. Also, the timing results confirm that our algorithms compare favorably with other state-of-the-art algorithms for GPU in the literature.

The remaining of the thesis is organized as follows. Chapter 2 provides the mathematical background of Residue Number System (RNS), modular arithmetic, Number Theoretic Transform [NTT] and homomorphic encryption schemes. Chapter 3 reviews the GPU architecture and its working principles and points outs common practices to use GPU devices efficiently. Chapter 4 presents three different NTT algorithms customized for efficient GPU implementation. Chapter 5 presents the implementation results and compares them with those of the state-of-the-art implementations in the literature. The thesis is concluded in Chapter 6 with final remarks capturing the achievements and contribution. Additionally, the NTT-GPU[3] and GPU-HE Library[4] implementations are publicly available on Github.

[3] https://github.com/Alisah-Ozcan/GPU-NTT

[4] https://github.com/Alisah-Ozcan/HE_GPU

2. BACKGROUND

This section presents the notation used throughout the thesis and explains the Residue Number System, the Barrett reduction, the Montgomery reduction, the Goldilock reduction, the Plantard reduction, polynomial multiplication and reduction, Number Theoretic Transform, homomorphic cryptosystems.

2.1 Notation

The SHE scheme used in this work is BFV, one of the most efficient and widely used cryptographic schemes in the literature. The scheme is based on the ring learning with errors (RLWE) problem, whose difficulty serves as the security assumption for some post-quantum cryptography and homomorphic encryption algorithms. The RLWE problem, or more precisely, learning with errors problem over rings, is a more efficient and practicable version of the learning with errors (LWE) problem, which is specialized to work with polynomial rings over finite fields, whose details are given below.

The BFV scheme makes use of the polynomial ring $\mathbf{R}_q = \mathbb{Z}_q/\Phi(x)$, where $\mathbb{Z}_q$ represents the finite ring $\{0, 1, \dots, q-1\}$, in which the arithmetic is performed modulo q. Here, n is the degree of the cyclotomic polynomial $\Phi(x)$, and when its degree is selected as a power of two, we obtain $\Phi(x) = x^n + 1$. Then, the arithmetic in the ring $\mathbf{R}_q$ is optimized as the polynomial division is performed with $x^n + 1$. Abusing the terminology we sometimes refer n as the dimension of $\mathbf{R}_q$ and use the notation $\mathbf{R}_{q,n}$ to indicate its dimension.

Symbols and operations used in the subsequent parts of the thesis are as follows: $\lceil\cdot\rceil$, $\lfloor\cdot\rfloor$, $\lceil\cdot\rfloor$ represent round up, round down and round to nearest integer, respectively. The notation, $[a]_t$, indicates that the integer a lies in $[-t/2, t/2]$ while $|a|_t$ reduces a to the interval $[0, t-1]$. A polynomial $a(x) \in \mathbf{R}_q$ can be treated as a vector of n integers in $\mathbb{Z}_q$, which is composed of its coefficients. When the number theoretic transformation (NTT), which is a form of discrete Fourier transformation over rings $\mathbb{Z}_m$ (Section 2.4.2), is applied to the vector of $a(x)$, a vector of the same dimension is obtained, which is shown as $\bar{a}(x)$ (or just $\bar{a}$). While the symbols $+$,$-$ and $\times$ (or just $\cdot$) represent addition, subtraction and multiplication, respectively in either $\mathbb{Z}_q$ or $\mathbf{R}_q$ the symbol $\odot$ represents modular pointwise multiplication for vector representation of the elements of $\mathbf{R}_q$ in the NTT domain. Namely, an element in a vector is multiplied by the elements of another vector with the same index value, where multiplications are in $\mathbb{Z}_q$ (i.e., modulo q multiplication). λ is the security parameter denoted in unary notation. $a \leftarrow \mathbb{S}$ stands for the uniform sampling of a from the set $\mathbb{S}$. χ_{err}, a truncated zero-mean discrete Gaussian distribution, is used to sample the coefficients of error polynomials. The distribution is parameterized by the error bound β_{err} and standard deviation σ.

Now, we can give the most general and simplified definition of the RLWE problem. Suppose $a \leftarrow \mathbf{R}_q$ and the secret s and the error e are the elements of $\mathbf{R}$, whose coefficients are sampled from χ_{err}. Also suppose we have $b = as + e$. Then, the "search" RLWE problem can be defined as follows: Given a and b, it is hard to find s. In an HE scheme, s is the secret key whereas the $(b,\ a)$ are the public key.

2.2 Residue Number System

An integer $X < M$, can be represented using residues x_i, where $x_i = X \bmod m_i$ for $i = 1, \ldots, r$, if $M = \prod_{i=1}^{r} m_i$. Here, m_i form a set of pair-wise relatively prime integers that are known as moduli or "base" and a common notation is that $[X]_{m_i} = X \bmod m_i$. Due to the Chinese Remainder Theorem (CRT) we have

$$|X|_M = \left| \sum_{i=1}^{r} \left| x_i \cdot M_i^{-1} \right|_{m_i} \cdot M_i \right|_M ,$$

where $M_i = \frac{M}{m_i}$. Additionally, The RNS provides that the base can be extended to or converted to new bases using Algorithms 21 and 22 in Section 4.4.5. The RNS is preferred in cryptographic applications as it allows concurrent arithmetic with a set of small moduli in place of a big modulus; this is useful especially when the small moduli fit the word length of the underlying computing platform (Antao et al., 2012). It is also showed (Bajard et al., 2015), that RNS proves to be useful in accelerating the R-LWE based lattice-base somewhat homomorphic encryption schemes (Fan and Vercauteren, 2012b; Brakerski, 2012). Furthermore, RNS-variants of such schemes are proposed (Bajard et al., 2016) and their implementations achieve good speedups on platforms, where the concurrency of RNS is exploited (Al Badawi et al., 2019a).

2.3 Modular Addition/Substraction and Multiplication

In the fields of mathematics, computer science, and cryptography, modular arithmetic plays a pivotal role in constraining data within a finite numerical set during arithmetic operations. This practice is imperative as it facilitates the efficient execution of operations on data, preventing unwieldy numerical expansions after each computation. Noteworthy among the fundamental modular arithmetic operations are addition, subtraction, and multiplication. In the modular arithmetic, the division operation is replaced by multiplication, a concept that can be elucidated through the following illustrative example:

Example 2.1.

$$A, B \in [0, q-1]$$

$$B^{-1} \mod q = \equiv B^{q-2} \mod q$$

$$C = A \times B \mod q$$

In the context of Example 2.1, we encounter a scenario wherein the variables A and B are provided as elements of $\mathbb{Z}_q$, and the objective is to effectuate a division operation involving A divided by B. Modular arithmetic lacks a direct division operation. To surmount this limitation, the process entails the computation of the modular inverse of B with respect to the modulus q, resulting in the derivation of the value B^{-1}. After that, the division operation in $\mathbb{Z}_q$ is realized by multiplying A with calculated B^{-1}. Note that we seek the existence of $B^{-1} \bmod q$, which we have when $\gcd(B, q) = 1$.

Modular addition and modular subtraction are relatively straightforward operations when compared to modular multiplication. In both addition and subtraction, the reduction step entails a single arithmetic operation. This simplicity arises from the fact that the magnitude of the operands undergoes minimal change following the addition or subtraction. This phenomenon can be explained as follows: Consider two numbers, A and B, both subject to modular reduction modulo q. In this modular arithmetic field, the permissible numerical values for both A and B lie within the range of *0* to *q − 1*. When these two numbers are added, the resulting number's value becomes the interval from *0* to *2q − 2*. Given the objective of returning the result to the correct range (i.e., [0, $q-1$]), it becomes necessary to perform a subtraction operation if the result exceeds q. Since the highest value for the sum of A and B is *2q − 2*, a single subtraction suffices to realign it in the correct range. Even when the sum reaches its maximum value, subtracting q will yield a value of *2q − 2*, ensuring that it resides within the correct range. A similar rationale applies to modular subtraction.

In contrast, modular multiplication presents a more intricate challenge in terms of returning the resulting number to the range of $[0, q-1]$. While a straightforward approach involves repetitively subtracting q from the multiplication result until it falls within the desired range, this method is notably inefficient. The subsequent subsection will elaborate on four distinct reduction techniques optimized for modular multiplication scenarios.

2.3.1 Barrett Reduction

Algorithm 1 Barrett Reduction (Shivdikar et al., 2022)

```
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: $r \leftarrow C \gg (k-2)$
2: $r \leftarrow r \cdot \mu$
3: $r \leftarrow r \gg (k+3)$
4: $r \leftarrow q \cdot r$
5: $C_{out} \leftarrow (C - r)$
6: if $C_{out} >= q$ then   $C_{out} \leftarrow C_{out} - q$
7: else $C_{out} \leftarrow C_{out}$
```

The Barrett reduction algorithms, as described in Algorithm 1, represent one of the widely adopted approach for efficiently executing modular reduction operations. In this algorithmic formulation, the parameter μ is designated as the pre-computed value $\mu = \lfloor \frac{2^{2k}}{q} \rfloor$, where q denotes the modulus, and k corresponds to the bit length of the said modulus. In contrast to the Original Barrett, Algorithm 1, which involves a right-shift by

$k-2$ positions as its initial step, this variant adopts an alternate approach. Specifically, it initiates with a direct multiplication operation using the pre-computed value μ, followed by a subsequent right-shift of $2k$ positions. This right-shifted result is then multiplied by the modulus q. Due to the inherent computational cost associated with the multiplication by μ, this right-shifting operation is performed in two separate stages in Algorithm 1 unlike the Original Barrett Algorithm.

A salient feature of the Barrett reduction algorithm lies in its strategic utilization of multiplication, bitwise shifting, and subtraction operations, as opposed to the computationally costly division operation necessitated by conventional modular multiplication algorithms in the calculation of $C \mod q$.

2.3.2 Montgomery Reduction

Algorithm 2 Word Level Montgomery Reduction (Bos and Montgomery, 2017)

```
Input: $C = a \times b$, where $a, b < q$; $k = \lceil log_2(q) \rceil$;
Input: $\mu = -q^{-1} \mod 2^w, w <= k$;
Output: $C_{out}$ $(C \times R^{-1} \mod q)$ where $R = 2^{\lceil \frac{k}{w} \rceil \times w} \mod q$
1: $T \leftarrow C$
2: for $i$ from 0 by 1 to $\lceil \frac{k}{w} \rceil$ do
3:     $T1 \leftarrow T \mod 2^w$
4:     $T2 \leftarrow (T1 \times \mu) \mod 2^w$
5:     $T \leftarrow \lfloor \frac{T + (T2 \times q)}{2^w} \rfloor$
6: if $T > q$ then  $C_{out} \leftarrow T$
7: else  $C_{out} \leftarrow T - q$
```

The Montgomery reduction, as presented in Algorithm 2, stands as an efficient methodology widely used for modular reduction. Particularly, it exhibits notable efficiency when implemented at the word level (with w representing the word size of the computing device), especially when handling data consistingn of high bit sizes, typically in the range of 128 to 256 bits or more. Similarly the Barrett reduction algorithm, the Montgomery reduction method requires the pre-computation of certain parameters before executing reduction operations. However, it is sufficient to calculate these parameters once for each modulus, so there is no need to recalculate each time. The pivotal parameter in this context is denoted as μ and is determined as follows: $\mu = -q^{-1} \mod 2^w$. A certain choice of the modulus of the form $q = k2^\nu + 1$, where $\nu \geq w$, renders $\mu = -1$, thereby there is no need the multiplication operation in line 4 of Algorithm 2.

Unlike the Barrett reduction, the Montgomery reduction does not yield the anticipated result directly; rather, it generates the result within the so-called Montgomery domain, $C_{out} = ABR^{-1} \bmod q$, where $R = 2^{\lceil \frac{k}{w} \rceil \times w} \bmod q$. Post-processing should be applied to obtain the expected outcome. While this may prima facie appear to introduce inefficiency, two methods can alleviate the burden associated with this situation.

One approach involves pre-processing and post-processing. Numbers that be multiplied can be multiplied with R. Thus, in an application where arithmetic operations are iteratively performed, Montgomery reduction is executed multiple times without immediate post-processing. Any operand or value, A, in the Montgomery domain is represented as AR and the Montgomery arithmetic operations on them always results in the same form. To recover the correct result, the operand, AR, should be multiply 1 using the Montgomery multiplication algorithm, which will render the final result A.

Secondly, if one of the numbers to be multiplied is constant, it can be multiplied with R. In such cases, the result obtained following Montgomery reduction equal to the expected result and there is no need to post-processing steps.

2.3.3 Goldilock Reduction

Algorithm 3 Goldilock Reduction (Hamburg, 2015)

```
Input: $C = a \times b$, where $a, b < q \equiv 2^{64} - 2^{32} + 1$
Output: $C_{out}$ $(C \mod q)$
1: $X_3 \leftarrow C \gg 96$
2: $X_2 \leftarrow (C \gg 64) \mod 2^{32}$
3: $X_1 \leftarrow C \mod 2^{64}$
4: $C_{out} \leftarrow X_1 + (X_2 \times (2^{32} - 1)) - X_3$
5: if $C_{out} \geq q$ then $C_{out} \leftarrow C_{out} - q$
6: else $C_{out} \leftarrow C_{out}$
```

The Goldilock reduction, as shown in Algorithm 3, represents a specialized reduction technique renowned for its unparalleled speed when compared to other reduction methodologies. However, this method is exclusively applicable to specific modulus values, such as $\Theta^{2k} - \Theta^k + 1$, it cannot be applied with all modulus such as Barret and Montgomery. The Goldilock reduction algorithm is tailored for Goldilock primes, and for the variant presented in Algorithm 3, the parameter k is assigned a value of 32.

The most important reason why the Goldilock reduction algorithm is more efficient than other methods is that it does not use the costly multiplication operation during reduction. Instead, the reduction is executed through arithmetic operations encompassing addition, subtraction, and shift operations, all of which are notably less computationally expensive than multiplication operations. This reduction is chiefly attributable to the expressibility of the modulus $2^{64}-\Theta^{32}+1$ as $2^{64}=\Theta^{32}-1$, as elucidated within Algorithm 3. Furthermore, Goldilock Reduction gives the expected results directly, that is, it does not need any pre-processing or post-processing.

2.3.4 Plantard Reduction

Algorithm 4 Plantard Reduction (Plantard, 2021)

```
Input: $a, b$ where $a, b < q$; $k = \lceil log_2(q) \rceil$ and $w = 2^{k+2}$
Output: $C_{out}$ $(a \times b \mod q)$
1: $\sigma \leftarrow (-2^{2w}) \mod q$                                   # Pre-computed
2: $\tilde{b} \leftarrow b \times \sigma \mod q$                           # Pre-computed
3: $\tilde{b} \leftarrow \tilde{b} \times (q^{-1} \mod 2^{2w}) \mod 2^{2w}$   # Pre-computed
4: $T \leftarrow a \times \tilde{b} \mod 2^{2w}$
5: $T \leftarrow T \gg w$
6: $T \leftarrow (T \times (q+1)) \gg w$
7: if $T \equiv q$ then $C_{out} \leftarrow 0$
8: else $C_{out} \leftarrow T$
```

The Plantard reduction, as delineated in Algorithm 4, may prima facie appear inefficient due to its requisite extensive pre-processing procedures. However, it can be advantageous in scenarios, wherein one of the multiplicand is constant. The foremost rationale for this efficiency derives from the provisions elucidated within Algorithm 4, wherein it is stipulated that if one of the multiplicands assumes a constant value, the complete pre-processing phase can be executed beforehand and all pre-computed values subsequently stored in memory. In this case, a multiplication gain is achieved compared to the Barrett and Montgomery reduction algorithms explained above in the previous two sections.

In order for the Plantard reduction to work, the value $\tilde{b}$ must be computed as shown in Algorithm 4. NTT, one of the applications in which this method can be used efficiently, is explained in the following sections.

2.4 Polynomial Multiplication and Polynomial Reduction

Within this section, we introduce the critical concepts of polynomial multiplication and polynomial reduction, both of which will serve as foundational arithmetic components in the subsequent sections. Multiple ways exists to perform polynomial multiplication and polynomial reduction. Below, working principle of these methods and their advantages is elucidated.

2.4.1 Schoolbook Polynomial Multiplication

Algorithm 5 Schoolbook Polynomial Multiplication & Reduciton

```
Input: $a(x), b(x) \in \mathbb{Z}_q[x]/(x^n+1)$ or $\mathbb{Z}_q[x]/(x^n-1)$
Output: $c(x) = a(x) \times b(x) \in \mathbb{Z}_q[x]/(x^n+1)$
1: $t = n; \quad m = 1$
2: for $i$ from 0 by 1 to $n$ do                      # Polynomial Multiplication Part
3:     for $j$ from 0 by 1 to $n$ do
4:         $\bar{c}_{i+j} = a_i \times b_j \pmod{q}$
5: for $k$ from 0 by 1 to $n$ do                      # Polynomial Reduction Part
6:     if $\mathbb{Z}_q[x]/(x^n+1)$ then $c_k = \bar{c}_k - \bar{c}_{n+k} \pmod{q}$
7:     else $c_k = \bar{c}_k + \bar{c}_{n+k} \pmod{q}$          # $\mathbb{Z}_q[x]/(x^n-1)$ Condition
```

The schoolbook polynomial multiplication is fundamentally same to the polynomial multiplication methods taught at the high school level. Essentially, it involves the pairwise multiplication of coefficients belonging to both polynomials to be multiplied, followed by the summation of terms sharing the same degree. The polynomial multiplication in **R** requires also polynomial reduction by the cyclotomic polynomial $\Phi(x)$.

As an illustrative instance used in the lattice-based cryptography, let us consider the scenario, where $\Phi(x) = x^n + 1$ represents our designated reduction polynomial. In this context, the equality $x^n = -1$ is employed to simplify the polynomial reduction. Then, the polynomial reduction can be done by subtraction of the upper half of the polynomial from the lower half.

The Schoolbook multiplication and reduction are described in Algorithm 5. It is discernible that a total of n^2 multiplications are entailed in this process, thereby the complexity of Schoolbook polynomial multiplication is n^2. In the rest of Algorithm 5, it is explained how polynomial reduction is performed with two separate reduction polynomials (i.e., $x^n + 1$ and $x^n - 1$) that will be frequently used in lattice-based cryotography.

2.4.2 Number Theoretic Transform

The number theoretic transform (NTT) is a form of Discrete Fourier Transform (DFT) defined over the ring of integers $\mathbb{Z}_q$. In cryptography, it is commonly used for multiplication of high degree polynomials as it reduces quadratic complexity of the schoolbook multiplication to $O(n\log n)$. The coefficients of an $(n-1)$-degree polynomial can be thought as a vector of integers, $a = [a_0, a_1, \dots, a_{n-1}]$, which can be transformed to another vector $\bar{a} = [\bar{a}_0, \bar{a}_1, \dots, \bar{a}_{n-1}]$ using NTT. The definition of the m-point NTT and INTT can be given as in Eqns 2.1 and 2.2:

$$\bar{a}_i = \sum_{j=0}^{n-1} a_j \omega^{i\times j} \mod q \text{ for } i = 0, 1, \dots, m \tag{2.1}$$

$$a_i = \frac{1}{n}\sum_{j=0}^{n-1} \bar{a}_j \omega^{-i\times j} \mod q \text{ for } i = 0, 1, \dots, n-1. \tag{2.2}$$

where $m \geq n$. For NTT to be defined, we need the existence of a constant value $\omega \in \mathbb{Z}_q$., which can have two types

- $\omega \in \mathbb{Z}_q$: the primitive n-th root of unity in $\mathbb{Z}_q$, which satisfies the conditions $\omega^n \equiv 1 \pmod q$ and $\omega^i \neq 1 \pmod q$ $\forall i < n$, where $q \equiv 1 \pmod n$.
- ψ, where $\psi \in \mathbb{Z}_q$: the primitive $2n$-th root of unity, which satisfies the conditions $\psi^{2n} \equiv 1 \pmod q$ and $\psi^i \neq 1 \pmod q$ $\forall i < 2n$, where $q \equiv 1 \pmod{2n}$. Note that $\omega = \psi^2 \bmod q$ and $\psi^n \bmod q = -1$.

If two vectors in the NTT domain $\bar{a}$ and $\bar{b}$, where $\bar{a} = NTT(a(x))$ and $\bar{b} = NTT(b(x))$, are multiplied element-wise in $\mathbb{Z}_q$, the result is $\bar{c}$ in the NTT domain, where $c(x) = a(x)b(x)$. When the inverse NTT is applied on $\bar{c}$, $c(x)$ is obtained. The NTT-based polynomial multiplicaton can be captured by the following formulae:

$$\bar{c}(x) = NTT_n(a(x)) \odot NTT_n(b(x)) \tag{2.3}$$

$$c(x) = INTT_n(\bar{c}(x)) \mod q \tag{2.4}$$

Consequently, an NTT multiplication algorithm can be defined for an efficient multiplication in $\mathbf{R}_q$ as described in Eqns. 2.3 and 2.4. As the formulas in Eqns 2.1 and 2.2 result in quadratic complexity for computing NTT and inverse NTT operations, Section 2.4.2.1 and Section 2.4.2.2 elucidated more efficient computation of NTT and its INTT approaches.

2.4.2.1 Merge NTT

Algorithm 6 Merge Forward NTT (**Merge-NTT**)

```
Input: $a(x) \in \mathbb{Z}_q[x]/(x^n+1)$ polynomial standard-order
Input: $\Psi_{rev}[k] = \psi^{br(k)} \pmod q$ for $0 < k \leq n-1$   (Powers of $\psi$ stored in bit-reverse
    order)
Input: $n = 2^l$, $q$ ($q \equiv 1 \mod 2n$)
Output: $\bar{a} \leftarrow NTT(a)$ in bit-reversed order
 1: $t \leftarrow n$;   $m \leftarrow 1$
 2: while $m < n$ do
 3:     $t \leftarrow t/2$
 4:     for $i$ from 0 by 1 to $m$ do
 5:         $j_1 \leftarrow 2it$
 6:         $j_2 \leftarrow j_1 + t - 1$
 7:         for $j$ from $j_1$ by 1 to $j_2 + 1$ do
 8:             $a_j, a_{j+t} \leftarrow \mathbf{CT}(a_j, a_{j+t}, \Psi_{br}[m+i], q)$
 9:     $m \leftarrow 2 \times m$
10: return $a$
```

In this section, we present two algorithms, Algorithm 6 and Algorithm 8 based on recursive NTT technique for efficient computation of NTT and inverse NTT, respectively. Both algorithms are based on the factorization of the cyclotomic polynomial x^n+1 into n degree-1 polynomials as follows:

$$x^n + 1 \equiv \prod_{i=0}^{n-1} (x - \psi^{2i+1}) \bmod q \tag{2.5}$$

By reducing a given polynomial $a(x)$ by these degree-1 polynomials, we obtain n integers, which are, in fact, the elements of $\bar{a}$. This computation can be performed recursively. We first use the following factorization

$$
\begin{aligned}
(x^n+1) &\equiv (x^n-\psi^n) \\
&\equiv (x^{n/2}-\psi^{n/2})(x^{n/2}+\psi^{n/2}) \bmod q
\end{aligned} \tag{2.6}
$$

and reduce $a(x)$ by polynomials $(x^{n/2}-\psi^{n/2})$ and $(x^{n/2}+\psi^{n/2})$. Reducing $a(x)$ by the first and second factors can be realized by employing the equations $x^{n/2}=\psi^{n/2}$ and $x^{n/2}=-\psi^{n/2}$, respectively. This accounts for the Cooley-Tukey Butterfly operations in Step 8 of Algorithm 6. The description of the Cooley-Tukey Butterfly operation given in Algorithm 7.

Algorithm 7 Cooley-Tukey Butterfly (**CT**)

```
Input: $U$, $V$, $\psi$, $q$
Output: $\bar{U}$, $\bar{V}$
1: $\bar{U} \leftarrow U+(V\times\psi)$ (mod $q$)
2: $\bar{V} \leftarrow U-(V\times\psi)$ (mod $q$)
3: return $\bar{U},\bar{V}$
```

Factorization is further utilized as follows:

$$(x^{n/2}-\psi^{n/2}) \equiv (x^{n/4}+\psi^{n/4})(x^{n/4}+\psi^{n/4}) \bmod q$$

and

$$
\begin{aligned}
(x^{n/2}+\psi^{n/2}) \equiv (x^{n/2}-\psi^{n/2+n})& \\
(x^{n/4}-\psi^{n/4+n/2})(x^{n/4}+\psi^{n/4+n/2}) \bmod q&
\end{aligned}
$$

The factorization is repeated until degree-1 polynomials are obtained.

As can be observed from Algorithm 6, different powers of ψ are stored in the bit-reverse order in the table Ψ_{rev}, which simply means that the ith power of ψ is stored in the $(rev(i)-1)$th element of Ψ_{rev}. For instance, for $n=8$ the first element of Ψ_{rev} holds ψ^4 as the bit-reversed order of $4=100$ is 001.

The inverse NTT operation, whose steps are given in Algorithm 8, is performed following the recursive factorization of (x^n+1) in the reverse order of that applied during the NTT computation.

To illustrate the inverse NTT algorithm, its last iteration is demonstrated, which yields

Algorithm 8 Merge Inverse NTT (**INTT**)

```
Input: $\bar{a} \in \mathbb{Z}_q^n$ in bit-reversed order
Input: $\Psi_{rev}[k] = \psi^{-br(k)} \pmod q$ for $0 < k \leq n-1$   (power of $\psi^{-1}$ stored in bit-reverse order)
Input: $n = 2^l$, $q$ ($q \equiv 1 \mod 2n$)
Output: $a(x) \in \mathbb{Z}_q[x]/(x^n+1)$ standard-order
 1: $t \leftarrow 1$;   $m \leftarrow n$
 2: while $m < n$ do
 3:     $j_1 \leftarrow 0$;   $h \leftarrow m/2$
 4:     for $i$ from 0 by 1 to $h$ do
 5:         $j_2 \leftarrow j_1 + t - 1$
 6:         for $j$ from $j_1$ by 1 to $j_2 + 1$ do
 7:             $\bar{a}_j, \bar{a}_{j+t} \leftarrow \mathbf{GS}(\bar{a}_j, \bar{a}_{j+t}, \Psi_{rev}[h+i], q)$
 8:         $j_1 \leftarrow j_1 + 2 \times t$
 9:     $t \leftarrow 2 \times t$
10:     $m \leftarrow m/2$
11: for $i$ from 0 by 1 to $n$ do
12:     $a_i \leftarrow (\bar{a}_i \cdot n^{-1}) \pmod q$
13: return $a$
```

the final result. The vector $\bar{a}$ before the last iteration is as follows:

$$\begin{aligned} a &= (a_0 + a_{n/2}\psi^{n/2}), \dots, (a_{n/2-1} + a_{n-1}\psi^{n/2}) \\ &\quad + (a_0 - a_{n/2}\psi^{n/2}), \dots, (a_{n/2-1} - a_{n-1}\psi^{n/2}) \end{aligned} \tag{2.7}$$

If the first half is added to the second half, the first half of the resulting vector multiplied by 2 is obtained,

$$2(a_0, \dots, a_{n/2-1}) \tag{2.8}$$

Furthermore, if the second half of $\bar{a}$ is subtracted from its first half,

$$2\psi^{n/2}(a_{n/2}, \dots, a_{n-1}). \tag{2.9}$$

is obtained. Thus, the result in Eqn 2.9 needs to be multiplied by $\psi^{-n/2}$. This elaborates the Gentleman-Sande Butterfly operation (which is described in Algorithm 9) in Step 7 of Algorithm 8.

As there are $\log_2 n$ iterations in the outermost loop of Algorithm 8 and the vector elements are effectively multiplied by 2 in every iteration, the result needs to be divided by n in $\mathbb{Z}_q$.

Algorithm 9 Gentleman-Sande Butterfly (**GS**)

```
Input: $U$, $V$, $\psi$, $q$
Output: $\bar{U}$, $\bar{V}$
1: $\bar{U} \leftarrow U + V \pmod{q}$
2: $\bar{V} \leftarrow (U - V) \times \psi \pmod{q}$
3: return $\bar{U}, \bar{V}$
```

2.4.2.2 4-Step NTT

The four-step method (Bailey, 1989), described in Algorithm 10, is another way to compute NTT (and inverse NTT) operations. In the first stage, it arranges the input vector into a two-dimensional matrix of n_1-by-n_2, where $n = n_1 \times n_2$ (Steps 1-5 of Algorithm 10). In the second stage, it performs n_2 NTT operations of n_1-point each (Steps 7-9). In the third stage, the matrix elements are multiplied with certain powers of the primitive root of unity, ψ (Steps 11-15). And finally, the algorithm performs n_1 NTT operations of n_2-point each (Steps 16-18). The **4-Step NTT** algorithm computes much smaller (n_1-point and n_2-point against n-point NTT operations) and independent NTT computations, which exploits the parallel processing and locality of memory access much better than the **Merge NTT** algorithm. Nevertheless, the transpose operations (Steps 6, 10, and 19) can be challenging as they can cause a very fragmented memory access, which can be very disruptive for threads in a GPU device. The version of the **4-Step NTT** algorithm for computing inverse NTT operation is given in Algorithm 11.

2.5 Homomorphic Encryption

Homomorphic encryption (HE) is a cryptosystem that allows the secure processing of encrypted data without decrypting it. Homomorphic cryptosystems are widely used applications where privacy and security are essential. Secure data outsourcing, cloud computing, and privacy-preserving machine learning can be considered as a few of daily life applications, which use homomorphic encryption. Homomorphic encryption basic workflow can be shown as follows:

- **Encryption Part:** The data owner encrypts the data with respect to a particular homomorphic encryption scheme.

Algorithm 10 Four-Step NTT (**4Step-NTT**)

```
Input: $n_1, n_2 \le n$ and $n_1 \times n_2 = n$
Input: $a(x) \in \mathbb{Z}_q[x]/(x^n - 1)$ in polynomial standard-order
Input: $\Omega[k] = \Omega^{br(j)\times i} \pmod q$ for $0 < k \le n-1$, for $0 < j \le n_1 - 1$ for $0 < i \le n_2 - 1$
Input: $\Omega_{0_{br}}[k] = \omega_0^{br(k)} \pmod q$ where $\omega_0 = \Omega^{(n/n_1)} \pmod q$, for $0 < k \le n_1 - 1$
Input: $\Omega_{1_{br}}[k] = \omega_1^{br(k)} \pmod q$ where $\omega_1 = \Omega^{(n/n_2)} \pmod q$, for $0 < k \le n_2 - 1$
Output: $\bar{a} \leftarrow NTT(a)$ in bit-reversed order
1:  for $i$ from 0 by 1 to $n_1$ do                          # 1) Vector to matrix
2:      for $j$ from 0 by 1 to $n_2$ do
3:          $B_{i,j} \leftarrow a_{i\times n_2+j}$
4:  $B = B^T$                                                # Transpose operation
5:  for $j$ from 0 by 1 to $n_2$ do                          # 2) $n_2$, $n_1$-point NTTs
6:      $B_j \leftarrow \mathbf{NTT}(B_j, \Omega_{0_{br}}, n_1, q)$
7:  $B = B^T$                                                # Transpose operation
8:  for $i$ from 0 by 1 to $n_1$ do                          # 3) Correction step
9:      for $j$ from 0 by 1 to $n_2$ do
10:         $B_{i,j} \leftarrow B_{i,j} \times \Omega_{i\times n_2+j} \pmod q$
11: for $i$ from 0 by 1 to $n_1$ do
12:     $B_i \leftarrow \mathbf{NTT}(B_i, \Omega_{1_{br}}, n_2, q)$  # 4) $n_1$, $n_2$-point NTTs
13: $B = B^T$                                                # Transpose operation
14: for $j$ from 0 by 1 to $n_2$ do                          # Matrix to vector
15:     for $i$ from 0 by 1 to $n_1$ do
16:         $a_{j\times n_1+i} \leftarrow B_{j,i}$
17: return $A$
```

- **Homomorphic Operations Part:** The homomorphic operation scheme provides specific mathematical operations that can be performed on the encrypted data (ciphertext). Within that period, there is no need for any decryption operation. The homomorphic operations do not need the decryption key either. Furthermore, the results obtained after computation are also encrypted. In other words, only the data owner gets the decrypted results.

- **Decryption Part:** Since the data owner has a secret key, the ciphertext obtained after computation can be decrypted to obtain desired results.

Homomorphic encryption has three types: Partially Homomorphic Encryption, Somewhat Homomorphic Encryption, and Fully Homomorphic Encryption. Partially Homomorphic Encryption only allows single mathematical operations, either multiplication or addition. Additionally, since the security of actual constructions of partially homomorphic encryption schemes is based on number theoretic hard problems, they are not secure for attacks running on quantum computers. ElGamal and Paillier cryptosystems (Gamal,

Algorithm 11 Four-Step INTT

```
Input: $n_1, n_2 \leq n$ and $n_1 \times n_2 = n$
Input: $A(x) \in \mathbb{Z}_q[x]/(x^n - 1)$ in bit-reversed order
Input: $\Omega[k] = \Omega^{-br(j)\times i} \pmod q$ for $0 < k \leq n-1$, for $0 < j \leq n_1 - 1$ for $0 < i \leq n_2 - 1$
Input: $\omega_{0_{br}}[k] = \omega_0^{-br(k)} \pmod q$ where $\omega_0 = \Omega^{-(n/n_1)} \pmod q$, for $0 < k \leq n_1 - 1$
    (Powers of $\omega_0$ stored in bit-reverse order)
Input: $\omega_{1_{br}}[k] = \omega_1^{-br(k)} \pmod q$ where $\omega_1 = \Omega^{-(n/n_2)} \pmod q$, for $0 < k \leq n_2 - 1$
    (Powers of $\omega_1$ stored in bit-reverse order)
Output: $A \leftarrow INTT(A)$ in polynomial standard-order
 1: for $i$ from 0 by 1 to $n_1$ do                      # Vector to matrix
 2:     for $j$ from 0 by 1 to $n_2$ do
 3:         $B_{i,j} \leftarrow A_{i\times n_2+j}$
 4: $B = B^T$                                            # 1st transpose operation
 5: for $j$ from 0 by 1 to $n_2$ do                      # $n_2$, $n_1$-point INTTs
 6:     $B_j \leftarrow \mathbf{INTT}(B_j, \omega_0, n_1, q)$
 7: $B \leftarrow B^T$                                   # 2nd transpose operation
 8: for $i$ from 0 by 1 to $n_1$ do
 9:     for $j$ from 0 by 1 to $n_2$ do
10:         $B_{i,j} \leftarrow B_{i,j} \times \Omega[i \times n_2 + j] \pmod q$
11: for $i$ from 0 by 1 to $n_1$ do
12:     $B_i \leftarrow \mathbf{INTT}(B_i, \omega_1, n_2, q)$    # $n_1$, $n_2$-point INTTs
13: $B = B^T$                                            # 3rd transpose operation
14: for $j$ from 0 by 1 to $n_2$ do                      # Matrix to vector
15:     for $i$ from 0 by 1 to $n_1$ do
16:         $A_{j\times n_1+i} \leftarrow B_{j,i}$
17: for $k$ from 0 by 1 to $n$ do
18:     $A_k \leftarrow (A_k \times n^{-1}) \pmod q$
19: return $A$
```

1985; Paillier, 1999) are the most commonly used representatives of Partially Homomorphic Encryption. ElGamal only provides homomorphic multiplication, while Paillier only provides homomorphic addition.

Since Somewhat Homomorphic Encryption (SWHE) and Fully Homomorphic Encryption (FHE) work based on RLWE, they are safe against attacks that may come from quantum computers. Moreover, unlike Partial Homomorphic Encryption, the cryptosystems in SWHE and FHE can perform multiple homomorphic operation types. So, homomorphic multiplication and addition can be used in the same scheme. The most famous examples of SWHE and FHE are BFV, CKKS, BGV, and TFHE (Fan and Vercauteren, 2012c; Cheon et al., 2016; Brakerski et al., 2011; Chillotti et al., 2016). This thesis exclusively focuses on the BFV scheme. Since the ciphertext is encrypted with noise terms in those schemes, noise level in the ciphertext increases after each homomorphic operation, and after a certain number of operations, the ciphertext becomes undecryptable, meaning the noise budget is exhausted. In this case, the ciphertext is passed back to the owner for decryption before the noise budget is exhausted. This type of schemes is commonly referred as somewhat homomorphic encryption schemes (SWHE). Cryptographic schemes, which permits the so-called bootstrapping operation (Boura et al., 2020), is known as fully homoorphic encryption (FHE) schemes. Bootstrapping is the process of lowering the noise level in the ciphertext without decrypting it. As a result of bootstrapping, the noise budget of the ciphertext İs replenished, allowing homomorphic operations to continue. A scheme with an efficient bootstrapping algorithm is called FHE because theoretically it offers the ability to perform an infinite number of homomorphic operations on the ciphertext.

2.5.1 ElGamal

One of the public-key cryptosystems, the ElGamal encryption system(Gamal, 1985), was propseded by Taher ElGamal in the 1980s. The ElGamal cryptosystem is mainly used for secure communication and partial homomorphic encryption. The security of the ElGamal cryptosystem is based on the difficulty of the discrete logarithm problem. That is why it is not secure against attacks running on quantum computers. ElGamal cryptosystem basic workflow can be shown as follows:

- **Key Generation Part:**
  - Alice generates a key pair (public key, secret key)
  - Secret key consists of one parameter.
    - a**:** integer number chosen by Alice.
  - Public key consists of three parameters.(p, g, h)
    - p**:** Large prime number.
    - g**:** Generator of the multiplicative group modulo p.
    - h**:** $h = g^a \mod p$.
- **Encryption Part:**

  - Bob wants to send an encrypted message to Alice. Firstly Bob selects k, which is random integer. After that Bob can generates the ciphertext as follows:
    * Compute $c_1 = g^k \mod p$.
    * Compute $c_2 = h^k \times m \mod p$, where $m < p$ is the message.
    * Send the ciphertext (c_1, c_2) to Alice.

- **Decryption Part:**
  - Alice receives the ciphertext (c_1, c_2) from Bob.
  - Alice calculates shared secret $\mathbf{s} = c_1{}^{\mathbf{a}} \mod p$ using her secret key.
  - Alice computes message from Bob, using the formula $m = c_2 \times \mathbf{s}^{\mathbf{-1}} \bmod p$

As mentioned earlier, ElGamal also provides homomorphic multiplication feature in addition to secure communication. The basic homomorphic multiplication workflow can be shown as follows:

- **Homomorphic Multiplication with Elgamal Cryptosystem:**
  - Bob obtains ciphertext(c_1, c_2) which encryptes the message m_1 with Alice's public key.
  - Bob generates new ciphertext(b_1, b_2) which encrypt the message m_2 via Alice's public key.
  - Bob computes a new ciphertext$(\mathbf{d1}, \mathbf{d2})$ as follows:
    * $d_1 = b_1 \cdot c_1 \mod p$
    * $d_2 = b_2 \cdot c_2 \mod p$
  - Bob sends the ciphertext(d_1, d_2) to Alice and she decrypts it with using her secret key. Then she obtains the result $m_1 \cdot m_2 \bmod p$. Although, Alice knows $m_1 \cdot m_2 \bmod p$, she never knows the exact values of m_1 and m_2.

2.5.2 Paillier

The Paillier cryptosystem, developed by Pascal Paillier in 1999 (Paillier, 1999), is also a public-key cryptosystem as the ElGamal cryptosystem. The Paillier cryptosystem is widely used in privacy-preserving computations and secure multiparty computation applications. One of the most essential features of the Paillier cryptosystem is that it provides the capability to perform homomorphic addition operations on encrypted data. The basic Paillier cryptosystem and its homomorphic addition feature workflow can be shown as follows:

- **Key Generation Part:**
  - Key generation part consists of generation of the public key (N, g) and the secret key (λ, μ).
    - N**:** product of two large prime numbers (i.e., $N = p \times q$) and it defines the size of the encryption space.
    - g**:** Generator of the multiplicative group modulo N^2.
    - λ **and** μ**:** are derived from p and q ($\lambda = LCM(p-1, q-1)$, $\mu = (L(g^{\lambda} mod N^2))^{-1} \mod N$ where $L(x) = (x-1)/N$).
- **Encryption Part:**
  - A random value r is selected where $1 \leq r \leq N$. After that message m is encrypted as follows:
    - $c = (g^m \times r^N) \mod N^2$
- **Homomorphic Addition Part:**
  - Given two ciphertexts, c_1 and c_2, generated from messages m_1 and m_2, respectively, if one wants to calculate the sum of m_1 and m_2 homomorphically, the computation $C_{sum} = c_1 \times c_2 \mod N^2$ is performed.
- **Decryption Part:**
  - The recipient uses his secret key value, which is generated from his public key parameters, in order to decrypt an incoming ciphertext. In this case, the message can be obtained as:
    - $m = L(C_{sum}^{\lambda} \mod N^2) \times \mu \mod N$ where $L(x)$ is the L-function defined as $L(x) = (x-1)/N$.

2.5.3 Brakerski/Fan-Vercauteren

BFV is one of the homomorphic encryption schemes that allow fully homomorphic encryption, based on RLWE. Unlike schemes based on partially homomorphic encryption, BFV allows performing addition and multiplication on the same ciphertext. This means that arbitrary computations can be performed on encrypted data while it remains encrypted. In addition to all of these benefits, since the security level of BFV relies on the hardness of the Learning With Errors (LWE) problem, it is secure against attacks from quantum computers.

BFV has multiple variants since its development in 2011 when Junfeng Fan and Frederik Vercauteren published their first work. In this thesis, the textbook-FV and RNS variant of BFV by Microsoft SEAL (Microsoft, 2020) are used.

2.5.3.1 Key Generation

Since BFV based on RLWE (Regev, 2005), which is asymmetric key cryptosystems, it employs a pair of keys a public key and a secret key. The keys are generated as follows:

$$a \leftarrow \mathbf{R}_{q,n},\ s \leftarrow \mathbf{R}_{2,n} \text{ and } e \leftarrow \chi$$

$$sk = s,\ pk = (pk[0], pk[1]) = ([-(as+e)]_q, a),$$

where sk and pk are secret and public keys, respectively. Also, the degree of cyclotomic polynomial $\Phi(x)$, n and the size of the ciphertext modulus depends on λ, which is security parameter. Let $q = q(\lambda) \geq 2$ is an integer. For the secret key generation in Textbook-BFV, $s \leftarrow \mathbf{R}_{2,n}$ is used as shown above. For the public key generation in Textbook-BFV, first $a \leftarrow \mathbf{R}_{q,n}$ and $e \leftarrow \chi$ are sampled from the corresponding distribution at random. Recall that a is sampled uniformly randomly, while e is sampled from Gaussian distribution. Since the Textbook-BFV is working with single ciphertext modulus, these operations are applied only one times. However, the RNS variant of BFV uses multiple moduli. For this reason, certain operations are repeated for each modulus, q_i in the RNS base, as explained in Algorithm 12.

The evaluation key, which is another public key, is also of great importance because it plays a crucial role in reducing the increasing size of the ciphertext after homomorphic multiplication and in the operation of homomorphic rotation functions. Essentially, its

Algorithm 12 Secret & Public Key Generation (RNS)

```
1: $s \leftarrow \mathbf{R}_{(-1,0,1),n}$, $err \leftarrow \chi$
2: for $i$ from 0 by 1 to $r$ do                # $r$ = Modulus Count in RNS Domain
3:     $sk_i \leftarrow s \mod q_i$
4:     $a_i \leftarrow \mathbf{R}_{q_i,n}$
5:     $e_i \leftarrow err \mod q_i$
6:     $pk[0]_i \leftarrow -(sk_i \times a_i + e_i) \mod q_i$
7:     $pk[1]_i \leftarrow a_i$
```

size and calculation vary depending on the ciphertext size. Under normal conditions, the ciphertext consists of two parts, $ct[0]$ and $ct[1]$, but after homomorphic multiplication, this size increases to three. If the so-called relinearization operation is not applied to the non-linear part, $ct[2]$, the computational difficulty of the following multiplication increases. Therefore, the relinearization operation is generally used after homomorphic multiplications. For this reason, evaluation keys are usually derived from the square of the secret key as shown in Algorithm 13, but this can also be applied to other powers of the secret key.

Algorithm 13 Evaluation Key (Relinearization Key) Generation (RNS variant)

```
1: $e_i \leftarrow \chi$
2: for $i$ from 0 by 1 to $r-1$ do              # $r$ = Modulus Count in RNS Domain
3:     $a_i \leftarrow \mathbf{R}_{q_i,n}$
4:     $evk[0]_i \leftarrow (-(sk_i \times a_i + e_i)) + (s^2 \times q_{r-1}) \mod q_i$
5:     $evk[1]_i \leftarrow a_i$
```

2.5.3.2 Encryption

When a message is encrypted, an error term (a random polynomial sampled from the distribution χ) is added to the ciphertest, similar to the error terms used in the key generation operation. The definition of the encryption operation for textbook-FV can be shown as follows:

$$m \in \mathbf{R}_{t,n},\ u \leftarrow \mathbf{R}_{2,n} \text{ and } e_1, e_2 \leftarrow \chi,$$

$$ct = (ct[0], ct[1]) = ([m \cdot \Delta + pk[0]u + e_1]_q, [pk[1]u + e_2]_q).$$

Here, the message, m, is encoded as polynomial, whose coefficients are reduced modulo t, where t is referred as the plaintext modulus. The most significant difference that sets BFV apart from other schemes is that the message is multiplied by a scaling factor, Δ, during encryption, where $\Delta = \lfloor q/t \rfloor$. For some integer $t > 1$, where $t \ll q$, the ciphertext and plaintext spaces are taken as $\mathbf{R}_{q,n}$ and $\mathbf{R}_{t,n}$, respectively. Also, we note that neither q nor t has to be a prime integer. While the RNS variant of the BFV is fundamentally similar to textbook-BFV, it exhibits some differences due to the inclusion of multiple moduli used in the RNs artihmetic. These distinctions are detailed in Algorithm 14. As

Algorithm 14 BFV Encryption (RNS variant)

```
1: $u \leftarrow \mathbf{R}_{(-1,0,1),n}$, $err_0 \leftarrow \chi$ , $err_1 \leftarrow \chi$
2: for $i$ from 0 by 1 to $r$ do                                  # $r$ = Modulus Count in RNS Domain
3:     $ct[0]_i \leftarrow (pk[0]_i \times u) + err_0 \mod q_i$           # $pk$ = Public Key
4:     $ct[1]_i \leftarrow (pk[1]_i \times u) + err_1 \mod q_i$
5: $half \leftarrow q_{r-1}/2$
6: for $j$ from 0 by 1 to 2 do                                    # Divide and Round Last $q$
7:     for $i$ from 0 by 1 to $r-1$ do
8:         $ct[j]_i \leftarrow (ct[j]_i - ((ct[j]_{r-1} + half) \mod q_{r-1}) - half) \times q_{r-1}^{-1} \mod q_i$
9: $fix \leftarrow \lfloor (m + \lfloor t + 1/2 \rfloor)/t \rfloor$                # $m$ = Message
10: for $i$ from 0 by 1 to $r-1$ do
11:     $ct[0]_i \leftarrow ct[0]_i + (m \times \Delta) + fix \mod q_i$
```

shown in Algorithm 14, when encrypting in the BFV RNS variant, the last of the RNS bases is decomposed and added to the other bases. In this case, the number of RNS bases in the ciphertext is one less than the initial number of RNS bases. The primary reason for this decomposition is to achieve the most accurate results during decryption.

2.5.3.3 Decryption

As can be seen from the equation below, Textbook-BFV decryption is quite easy to understand.

$$ct = (ct[0], ct[1]) \in \mathbf{R}_{q,n} \text{ and } sk \in \mathbf{R}_{2,n}$$

$$m = [\lfloor \Delta^{-1}[ct[0] + ct[1]s]_q \rceil]_t, \text{ where } \Delta = \frac{q}{t}$$

The only criterion here is that the error disappears when ciphertext is divided by Δ during decryption. If the error is greater than Δ, that is, if the result is an integer when divided by delta, it will not give the correct result even if the ciphertext is decrypted. From this, it is evident that using a larger modulus will provide a larger noise budget. In this case, it enables more homomorphic operations but reduces the security level.

For the BFV RNS variant, decryption is considerably more complex compared to encryption and key generation. This complexity arises because it involves multiple base conversions. To perform these base conversions during decryption, certain parameters for t (the plaintext modulus) and γ (a threshold parameter for error handling) must be precomputed. Essentially, these parameters are based on the RNS conversion operation mentioned in Section 2.2, which is implemented using Algorithms 21 and 22. As described in Algorithm 15, the BFV RNS variant initiates with $ct[1]$ multiplied by the secret key, and then the ciphertext pairs are added together. However, the BFV RNS variant also includes additional operations to return to the domain of the plaintext modulus.

Algorithm 15 Decryption (RNS)

```
1: $P_\gamma \leftarrow 0$ , $P_t \leftarrow 0$
2: for $i$ from 0 by 1 to $r-1$ do          # $r$ = Modulus Count in RNS Domain
3:     $P_i \leftarrow (ct[1]_i \times sk_i) + ct[0]_i \mod q_i$          # $sk$ = Secret Key for each RNS Domain
4:     $P_i \leftarrow (P_i \times t \times \gamma) \times q^{-1} \mod q_i$          # Convert from base $q$ to base $(t, \gamma)$
5:     $P_\gamma \leftarrow P_\gamma + P_i \mod \gamma$
6:     $P_t \leftarrow P_t + P_i \mod t$
7: $P_\gamma \leftarrow P_\gamma \times (-q^{-1}) \mod \gamma$
8: $P_t \leftarrow P_t \times (-q^{-1}) \mod t$
9: for $i$ from 0 by 1 to $n$ do          # $n$ = Polynomial Ring Size
10:     if $P_\gamma[i] > (\gamma/2)$ then $m[i] \leftarrow P_t[i] + (\gamma - P_\gamma[i]) \mod t$          # $m$ = Message
11:     else $m[i] \leftarrow P_t[i] - P_\gamma[i] \mod t$
12: $m \leftarrow m \times (-\gamma^{-1}) \mod t$
```

2.5.3.4 Addition/Substract

In the BFV scheme, the most straightforward operations are addition and subtraction. It just consists of modular addition and subtraction of the coefficients of ciphertext polynomials that are in $\mathbf{R}_{q,n}$. As shown in Algorithm 16, two pairs of ciphertext polynomials in the same bases are added or subtracted coefficient-wise, where the moduli are q_i for $i = 0, \ldots r-1$. Here, ct_i stands for the ciphertext pair in the modulus q_i for $i = 0, \ldots r-1$; namely $ct_i = [ct]_{q_i}$ for ease of notation.

Algorithm 16 Addition

```
1: for $i$ from 0 by 1 to $r-1$ do                # $r$ = Modulus Count in RNS Domain
2:     $ct[0]_i \leftarrow ct[0]_i + \bar{ct}[0]_i \mod q_i$
3:     $ct[1]_i \leftarrow ct[1]_i + \bar{ct}[1]_i \mod q_i$
```

2.5.3.5 Multiplication

The Textbook-BFV multiplication takes two ciphertex (ct and $\bar{ct}$) and computes following equations:

$$c_0 = [ct[0] \times \bar{ct}[0]]_q$$

$$c_1 = [ct[0] \times \bar{ct}[1] + ct[1] \times \bar{ct}[0]]_q$$

$$c_2 = [ct[1] \times \bar{ct}[1]]_q$$

As can be seen in the equation, polynomials of two ciphertexts are multiplied with each other, and the product is a new ciphertext with a size of three. Although the multiplication of the textbook-BFV is easy, the multiplication of the BFV RNS variant, which visualised in Figure 2.1, is quite complicated. As pointed out earlier in the RNS variant of the BFV scheme, a set of smaller moduli q_i is used instead of one large coefficient modulus q for the ring arithmetic; a technique known as residue number system (hence, the abbreviation RNS). Using RNS arithmetic allows to perform operations in parallel and removes the need for arbitrary-precision arithmetic. The homomorphic multiplication operation takes two ciphertexts as inputs, each of which consists of two polynomials in $\mathbf{R}_{q,n}$ and performs a tensor product that produces three polynomials as output in each RNS base.

Due to complexity of using the RNS arithmetic in homomorphic multiplication (see (Bajard et al., 2016) for more details), the SEAL library uses the base extension technique and introduces additional auxiliary base ($\mathcal{B}$ and m_{sk}) in addition to the RNS base $\mathcal{Q} = \{q_0, q_1, \ldots, q_{r-1}$. The auxiliary base $\mathcal{B}$ consists of $\{B_0, B_1, \ldots, B_{\rho-1}\}$, which are pairwise co-prime while m_{sk} is a prime integer. Generally, the auxiliary base $\mathcal{B}$ and the prime m_{sk} are joined to form the base $\mathcal{B}_{sk}$ $(= \mathcal{B} \cup m_{sk})$. Thus, the homomorphic multiplication operation in BFV requires conversion between the $\mathcal{Q}$ base and the auxiliary base $\mathcal{B}_{sk}$. The conversion is implemented using a technique known as “fast base conversion”, which can introduce extra multiples of q in the computations that can lead to error in the ciphertext. To remedy this, a reduction operation through another modulus $\tilde{m}$ is required after the fast base conversion operation is applied.

As shown in Figure 2.1, the BFV multiplication operation starts by performing the fast

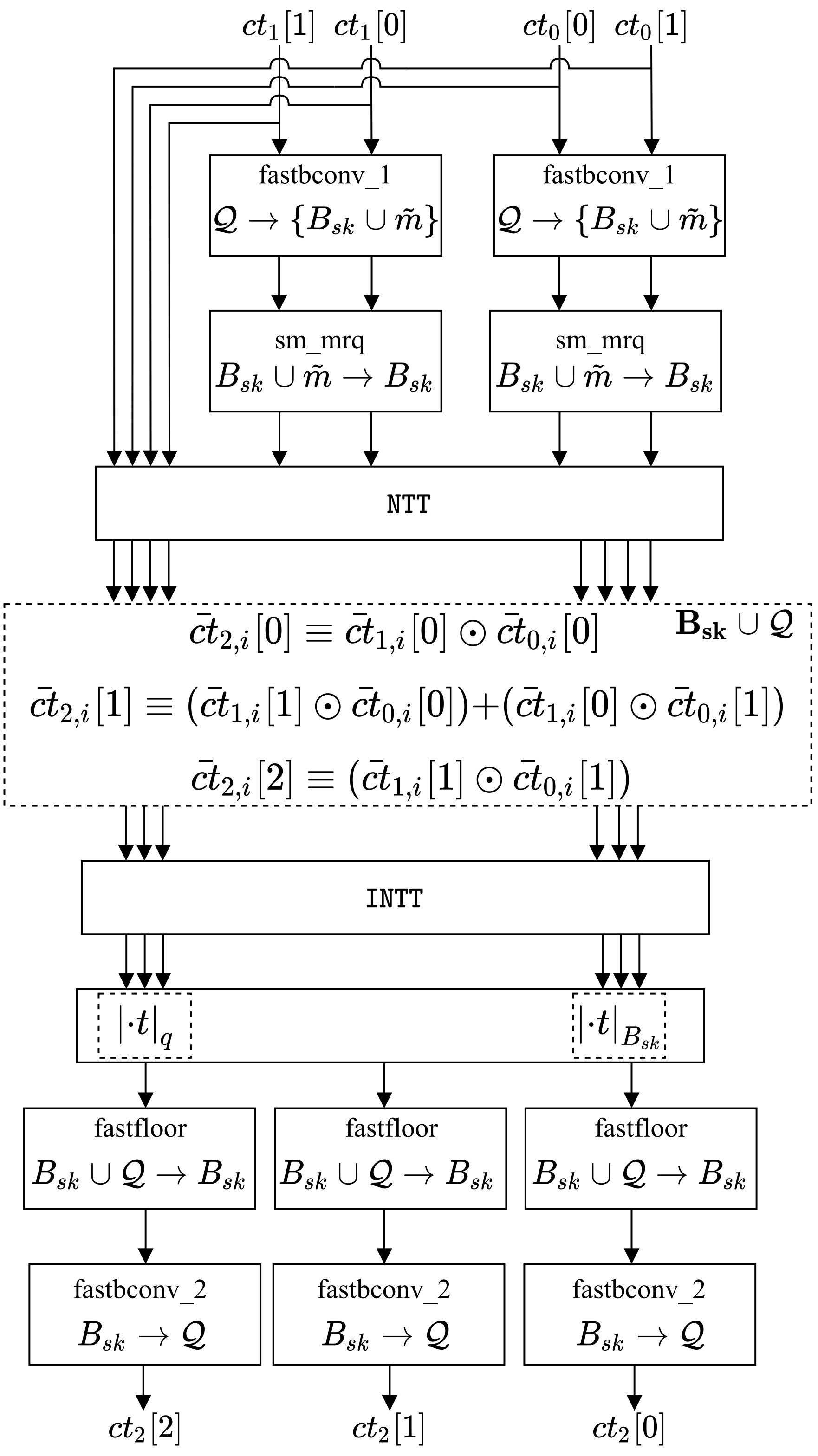
$ct_1[1]$ $ct_1[0]$ $ct_0[0]$ $ct_0[1]$
fastbconv_1
$\mathcal{Q} \rightarrow \{B_{sk} \cup \tilde{m}\}$
fastbconv_1
$\mathcal{Q} \rightarrow \{B_{sk} \cup \tilde{m}\}$
sm_mrq
$B_{sk} \cup \tilde{m} \rightarrow B_{sk}$
sm_mrq
$B_{sk} \cup \tilde{m} \rightarrow B_{sk}$
NTT
$\bar{ct}_{2,i}[0] \equiv \bar{ct}_{1,i}[0] \odot \bar{ct}_{0,i}[0]$
$\mathbf{B_{sk}} \cup \mathcal{Q}$
$\bar{ct}_{2,i}[1] \equiv (\bar{ct}_{1,i}[1] \odot \bar{ct}_{0,i}[0]) + (\bar{ct}_{1,i}[0] \odot \bar{ct}_{0,i}[1])$
$\bar{ct}_{2,i}[2] \equiv (\bar{ct}_{1,i}[1] \odot \bar{ct}_{0,i}[1])$
INTT
$|\cdot t|_q$
$|\cdot t|_{B_{sk}}$
fastfloor
$B_{sk} \cup \mathcal{Q} \rightarrow B_{sk}$
fastfloor
$B_{sk} \cup \mathcal{Q} \rightarrow B_{sk}$
fastfloor
$B_{sk} \cup \mathcal{Q} \rightarrow B_{sk}$
fastbconv_2
$B_{sk} \rightarrow \mathcal{Q}$
fastbconv_2
$B_{sk} \rightarrow \mathcal{Q}$
fastbconv_2
$B_{sk} \rightarrow \mathcal{Q}$
$ct_2[2]$ $ct_2[1]$ $ct_2[0]$

Figure 2.1 Homomorphic Multiplication Operation in The BFV Scheme.

base conversion operations `fastbconv_1`, which convert the inputs in $\mathcal{Q}$ to the base $\{B_{sk} \cup \tilde{m}\}$. The `fastbconv_1` operations are followed by the reduction operation, for which the additional base $\tilde{m}$ is used; this operation is known as small Montgomery reduction modulo q, `sm_mrq`. It limits the impact of the error and converts the inputs in the $\{\mathcal{B}_{sk} \cup \tilde{m}\}$ base to the $\mathcal{B}_{sk}$ base. After the `sm_mrq` operation, the NTT operation is applied to all ciphertext components (both in $\mathcal{B}_{sk}$ and $\mathcal{Q}$ bases) and ciphertext multiplication operation is performed coefficient-wise to all vectors in all bases. Then, the inverse NTT operation is performed to convert the result to the polynomial domain. After the inverse NTT operation, ciphertexts are multiplied with plaintext modulus t. Then, the floor operation is used instead of the rounding operation; via a method is called "`fastfloor` function", and convert the ciphertext in the base $\{q \cup B_{sk}\}$ bases to the base $\mathcal{B}_{sk}$ as it involves division by q. Finally, the `fastbconv_2` function is used to perform conversion from the B_{sk} base back to the original RNS base $\mathcal{Q}$. The reader is referred to (Bajard et al., 2016) for more detail.

2.5.3.6 Relineariazation

To remove the non-linear part of the ciphertext after homomorphic multiplication, the Relinearization operation is applied. It is essentially based on the so-called switch-key operation illustrated in Figure 2.2, which transforms the third ciphertext component $ct[2]$, which is decryptable by sk^2, into a form, which is decryptable by the original secret ky, sk. To perform the relinearization operation, the Relinkey, which is a evaluation key generated from the secret key in Algorithm 13, is needed. The following equation represents the textbook-BFV relinearization operation

$$\text{Decomposes } c_2 \text{ in } w \text{ as } c_2 = \textstyle\sum_{i=0}^{l} c_2^{(i)} \times w^i$$
$$ct_{new}[0] = [c_0 + \textstyle\sum_{i=0}^{l} evk[i][0] \times c_2^{(i)}]_q$$
$$ct_{new}[1] = [c_1 + \textstyle\sum_{i=0}^{l} evk[i][1] \times c_2^{(i)}]_q$$

The SEAL BFV uses the switch-key technique visualised in Figure 2.2, which consists of the mix of three different methods for relinearization operation (Bajard and Plantard, 2004), (Halevi et al., 2019), (Chen et al., 2019). The most current method of these techniques is the special modulus method, which improves relinearization in terms of noise performance. The switch-key method shown in Figure 2.2 is the main building block of the relinearization and the rotation operations. As shown in Figure 2.2, after the homomorphic multiplication, in addition to $ct[0]$ and $ct[1]$, the third ciphertext component $ct[2]$ is obtained. Recall that a ciphertext component is written in $r-1$ moduli excluding

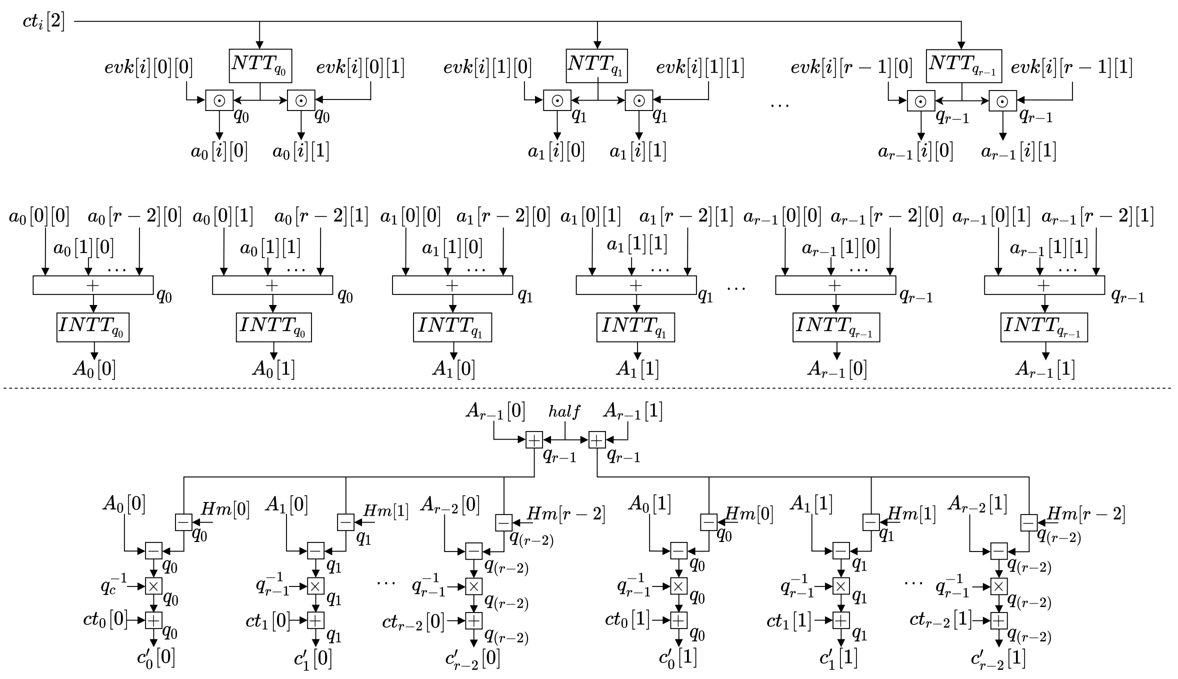

Figure 2.2 Switch Key Operation in the BFV Scheme. The symbols $+$,$-$ and $\times$ represent addition, subtraction and multiplication, respectively in either $\mathbb{Z}_q$ or $\mathbf{R}_q$ while the symbol $\odot$ represents modular pointwise multiplication for vector representation of the elements of $\mathbf{R}_q$ in the NTT domain.

q_{r-1} after encryption; ct_i for $i = 0, \ldots, r-1$. Firstly, all $ct_i[2]$ are transformed to the NTT domain using all moduli q_i in the RNS base to be multiplied with the evaluation keys that are already in the NTT domain. The number of NTT operations is, therefore, $r(r-1)$. After the NTT operations, the ciphertexts are multiplied with the evaluation keys in the NTT domain, where the multiplication is component-wise modulo multiplication. The modulus used in the multiplication is written next to the box that represents component-wise multiplication in Figure 2.2. Then, all results from the multiplication using the same modulus q_i in the RNS base are summed. Subsequently, the resulting vectors are transformed back to the polynomial domain using inverse NTT operation. Finally, as shown in Figure 2.2, necessary operations are applied to accommodate $ct[2]$ in $ct[0]$ and $ct[1]$. In the figure the half mode $Hm[i] = [\lfloor q_{r-1}/2 \rfloor]_{q_i}$. See Algorithm 23 for the details.

2.5.3.7 Rotation

The rotation operation also uses the switch-key operation as in the case of relinearization. However, the operation is based on Galois automorphism (Laine, 2017), and therefore, rotation mainly are used for the switch-key operation except initial part. At the beginning of the rotation operation, ciphertext are permuted with using Apply Galois operation as shown in Algoritm 24 and then the switch-key operation applied. For each power of 2, there is a different set of Galois keys and if the rotation amount is a power of 2, the switch-key operation is executed using the corresponding Galois key. On the other hand, if the rotation amount is not a power of two, the amount is written as the combination of powers of two, and the switch-key operation is applied multiple times with different Galois keys. For instance, if the rotation amount is 10, it can be implemented using two switch-key operations; the former uses the Galois keys for 8, the latter for 2.

3. GPU CODING AND ARCHITECTURE

GPU is a computing device that facilitates exceptional parallel processing capability due to that fact that it supports extremely high number of threads. Therefore, its instruction throughput far exceeds the one that can possibly be sustained by a conventional general-purpose CPU, which features comparably modest number of threads. As CPU and GPU are designed for different applications with different design principles, quantity, speed and computational power of CPU and GPU threads are also different. A CPU sustains fewer number of threads (in the order of tens) that can complete computationally more involved operations faster while recent GPU devices can support as high as 15K threads. But the GPU threads are computationally less capable running at slightly slower clock speeds. Therefore, the performance comparison of GPU and CPU can be involved and depends on benchmarks as pointed out in Lee et al. (2010) and advantage of GPU over CPU is overestimated at times.

Configurable hardware platforms such as FPGA, which can offer superior parallelization with better energy efficiency than GPU. However, being easier to program and integrate with applications running on CPU and containing more on-chip and off-chip memory, GPU can be a strong alternative to accelerate memory-bound applications such as advanced cryptographic operations, which heavily rely on arithmetic on extremely large mathematical objects; e.g. polynomial rings of very high dimensions.

CUDA-enabled GPU is a parallel computing device that can be used for general-purpose programming via CUDA®, which is a general purpose parallel computing platform and programming model. The CUDA software environment allows developers to use high-level programming languages such as C++ and Fortran.

In conclusion, GPUs are powerful devices, which proved to be accessible and easily programmable accelerators for a wide range of applications. However, it is essential to acquire deep insight into its micro-architectural details and to design algorithms to harness their computational resources.

3.1 High-Level Architecture of GPU

From a very high level, we can consider that a GPU platform consists of two main parts: i) GPU chip, and ii) off-chip memory. The GPU chip contains the main computation units known as streaming multiprocessors (SM) and on-chip memory that implements registers and shared memory. The local, global, constant, and texture memory types are all implemented in off-chip memory, for which the GDDR is used; a memory technology offering higher bandwidth and more power-efficient communication when compared with the DDR technology used in CPU memory systems. Nevertheless, the off-chip memory is still much slower than registers and on-chip shared memory.

A GPU contains an array of Streaming Multiprocessors (SM), which create, manage, schedule and execute threads on its functional units. An SM contains L1 cache (which is used to implement shared memory) and registers that are accessible to the threads scheduled to run on the SM. Execution happens in groups of 32 parallel threads called *warps*, in which threads start together at the same program address, but they can execute independently as they maintain separate states. An SM partitions threads into warps and its warps schedulers schedule them for execution on functional units of SM.

Another programming abstraction is thread block (or simply *block*), which can contain more than one warp. For instance, a block can contain as many as 32 warps or 1024 threads in current GPU devices. For easy indexing of threads, a block can be defined one, two or three dimensional. All threads in a block are scheduled to run on the same SM (accessing the same shared memory), which is capable of executing more than one block.

A *grid* is formed by combining multiple blocks, each of which contains the same number of threads. As the number of threads in a block is limited, grids are used to run larger number of threads in parallel than that can fit in a single block. Namely, different blocks in a grid may run in different SMs, and therefore, threads in different blocks do not use the same shared memory. Similar to blocks, a grid can be indexed one, two or three dimensional. We refer a particular configuration of block or grid as its *shape*.

Kernel is a function that is executed on a GPU device, which takes also the number of threads and blocks as arguments. As more than one block is used to execute a kernel, when threads from different blocks have to synchronize, this can be done by terminating the current kernel and start another if the computation is expected to continue. Starting and terminating kernels incur significant timing overhead, therefore an algorithm that limits the thread synchronization within a block is more efficient.

The *compute capability* of a GPU device, which determines hardware features and/or instructions available, is given by a version number and should be known to algorithm designers for developing better algorithms and their efficient implementations on GPU devices. The compute capability of a GPU device determines some pertinent information to our work such as the maximum number of blocks, warps, threads, number of 32-bit registers, maximum amount of shared memory per SM. In this thesis, we use GPU devices with the compute capability 8.6 and some of its relevant features are listed in Table 3.1.

Table 3.1 Configuration of compute capability 8.6

Maximum number of blocks per SM	16
Maximum number of warps per SM	48
Maximum number of threads per SM	1536
Number of 32-bit registers per SM	64 K
Maximum number of 32-bit registers per thread	255
Maximum amount of shared memory per SM	100 KB

3.2 GPU Memory Hierarchy

There are different types of memory in the GPU, and the access pattern to these memory types plays a very important role in memory latency performance. Each memory type has its advantages and disadvantages and can be used for different purposes accordingly. Table 3.2 lists these memory types and their specifications.

In the hierarchical structure of GPU memory system, the *global memory* is in the highest level implemented in GDDR (off-chip memory), and therefore, its size is larger than any other GPU memory type. Its access latency is slower than other memory types, because of the overhead of accessing the off-chip memory. Additionally, since the global memory

Table 3.2 The various principal traits of the memory types

Memory Types	Scope	Life Time	Access Latency
Register	1 Thread	Kernel	1×
Shared	All threads in block	Kernel	1×
Local	1 Thread	Kernel	≈ 100×
Global	All threads + host	Application	≈ 100×

allocations persist for the lifetime of a GPU application, data in global memory can be shared among kernels and all threads in a GPU can access global memory regardless of their block. The *local memory*, rather than a physical memory, is an abstraction of global memory, which is local to the thread and used to hold variables when register space is not sufficient.

The *shared memory* persists for the lifetime of a kernel, due to the fact that they are implemented in the L1 cache (on-chip memory) of an SM. As the shared memory is private to blocks, all threads of a block can access the same shared memory and synchronize. Furthermore, the shared memory, which is smaller in size, is fast and its access latency is comparable to that of registers. Consequently, memory-dependent operations can be profitably performed in the shared memory instead of the global memory. However, threads from different blocks can use only the global memory to share data.

The last on-chip memory type, the *register file* consists of 32-bit registers, which can be accessed in one clock cycle. The register file size is 64K 32-bit registers per SM and each thread has at most 255 32-bit registers.

3.3 Coalesced & Uncoalesced Access to Global Memory

As discussed in Section 3.1, each kernel needs to access global memory to load and store data. Therefore, accessing global memory plays a very important role in high performance GPU programming. The data in the off-chip global memory are delivered to the CUDA cores via caches. Each time the global memory is accessed, the entire memory block is fetched and placed in a cache line of the same size as the memory block. Therefore, the line sizes of L1 and L2 cache memories have particularly significant impact on the performance of CUDA programs. As the line sizes of both cache memories are

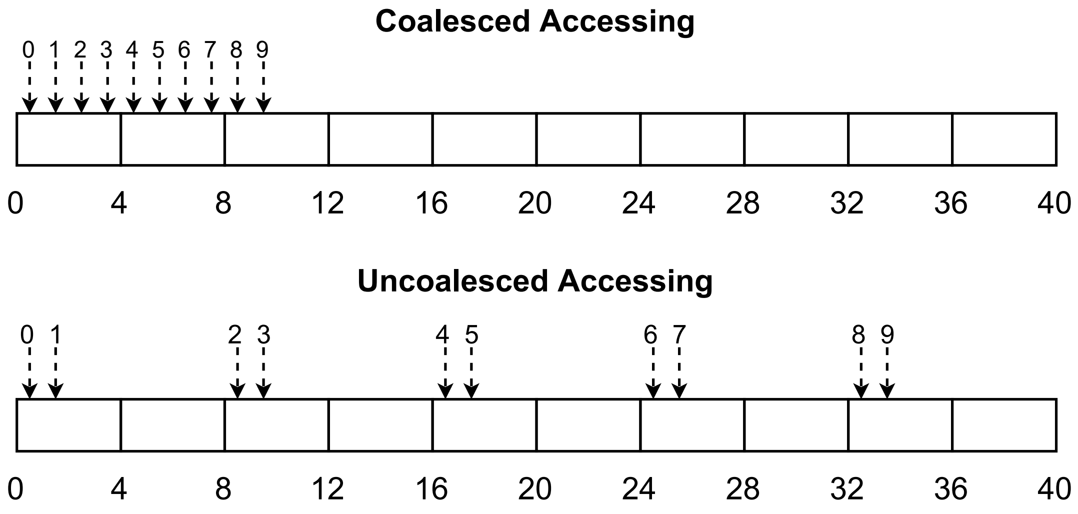

Figure 3.1 Coalesced and Uncoalesced Accessing Paterns

128 B, a group of threads accessing consecutive addresses that coincide with a memory block of 128 B contributes to achieving the global memory access latency values listed in Table 3.2. This is known as *coalesced* access in the GPU terminology which is illustrated in Figure 3.1. For instance, 16 threads accessing 8 B `unsigned long long` data types each, which happen to be consecutive in the same memory block, maximizes the performance as the entire 128 B block is brought to the cache with one global memory access operation. Otherwise, uncoalesced and strided accesses occur and the latency figures in Table 3.2 cannot be achieved as the same amount of data requires more than one memory block to be brought to the cache. One can affect the coalesced access following good programming practices.

3.4 Theoretical Occupancy

Occupancy is defined as the ratio between the number of actual active warps on an SM and the maximum *possible* number of active warps on the SM. As the occupancy plays an important role in efficient utilization of GPU resources (and ultimately the application performance), it is extremely essential to calculate the maximum theoretical occupancy of GPU before launching a kernel. When a kernel is created, the number of threads in a block, is determined as well as the shared memory size that will be available in the block. Since each block runs in one SM, the size of the register per thread is also fixed in the block. These three parameters (namely, block dimensions, the shared memory size, the number of registers per thread) directly affect the maximum theoretical occupancy. A block can include at most 1024 threads, which may not be necessarily optimum for achieving maximum theoretical occupancy. The reason is that increasing block size limits the resources per thread. For instance, for GPUs with compute capability 8.6, shared memory capacity per SM and is 100 KB, while maximum shared memory per thread block is 99 KB. Also, the L1 cache is configurable and its some parts can be used for shared memory and some parts can be used for data loaded or stored by the L2 cache. If the shared memory size is too high, a bottleneck occurs because there is little space left for data loaded or stored by the L2 cache. Therefore, in order to allocate more resources to threads one can consider deploying smaller block sizes such as 512 or even 256. This way, more warps can be active at a time. One can follow the NVIDIA documentation and guidelines to achieve higher occupancy rates. In Section 4.2 at Table 4.1, we report the achieved occupancy rates and how it affects the overall timings of our NTT implementation (see the discussion of the optimum size for a thread block).

4. GPU IMPLEMENTATONS

This section presents implementation techniques, methods, and algorithms for BFV homomorphic operations on GPUs: key generation, encryption, decryption, addition, multiplication, relinearization, and rotation. Moreover, since BFV operations rely on polynomial multiplication, two different NTT algorithms and three different implementations on GPU devices are presented in Sections 4.1, 4.2, and 4.3, respectively. We will detail their steps and explain the rationale behind the specific design choices, which are directly determined by the micro-architecture of GPU devices.

4.1 MERGE-NTT GPU IMPLEMENTATION APPROACH 1

In this section, we explain our first GPU implementation of NTT algorithm, which is based on the recursive algorithm in Section 2.4.2.1 (Algorithm 6) The Cooley-Tukey NTT and Gentleman-Sande INTT algorithms described in Section 2.4.2 is implemented on the GPU. NTT and INTT perform as the inverse of each other in terms of algorithm steps. Therefore, in this section, separate explanations of NTT and INTT are not needed. This section explains the challenges for fast and efficient implementation of NTT and presents our solutions to overcome them. Algorithm 17 shows the GPU pseudo-code for the NTT algorithm, which is essentially the same as the on given in Algorithm 6. One important adaptation to GPU is the synchronization operation in Step 12, whose effect on the correctness of the computations will be explained later in this section.

Algorithm 17 Merge in Place Forward NTT on GPU (with `syncthreads`)

```
Input: $A[n], PsiTable[n], q$
Output: $A[n]$
 1: $Idx = blockIdx.x \times blockDim.x + threadIdx.x$
 2: for $loop$ from 0 by 1 to $\log_2(n)$ do
 3:     $t = (n/2) \gg loop$
 4:     $m = 2 \ll loop$
 5:     $address = int(idx/t) * t + idx$
 6:     $U = A[address]$
 7:     $V = A[address + t]$
 8:     $Psi = PsiTable[int(idx/t) + m]$
 9:     $A[address] = (U + V)\%q$
10:     $V = (V \times Psi)\%q$
11:     $A[address + t] = (U - V)\%q$
12:     __syncthreads()
```

The NTT operation consists of $\log_2 n$ sequentially executed loops, each of which contains $n/2$ butterfly operations independent of each other, which can be performed simultaneously using $n/2$ threads on the GPU.

Each GPU can run a certain number of streaming multiprocessors (SM), the number of which depends on the GPU model and computational capability (version) of the GPU. Each SM consists of 4 warps scheduler and each scheduler can perform 32 threads at same time for all GPU models. However, these warp schedulers can perform sequentially and generate blocks which have 1024 threads. The limitation here is each SM can perform 1536 virtual thread as shown Table 3.1. Therefore, if the executed blocks consist of 1024 threads, the SM can perform one block. But, if executed blocks consist of 512 threads, SM can perform as many as 3 blocks.

When a GPU code is executed, the tasks are performed by warp groups. For example, if the code uses a number of threads in the range of [96-128], a total of four warps is needed in both cases. Also, even if the warps perform the same task, they may not finish their share of tasks simultaneously. Therefore, shared data usage among threads can lead to synchronization problems.

For instance, as the ring dimension n or the number of simultaneous (I)NTT operations increases, synchronization problems can occur if proper synchronization operations are not employed during the execution of Algorithm 17 (suppose Step 12 of Algorithm 17 is not present). This can be explained with a simple example. Suppose that we have a hypothetical GPU with a total thread count of 16 and a maximum block size of 4 threads. Let one warp of this GPU consist of two threads and let Algorithm 17 without

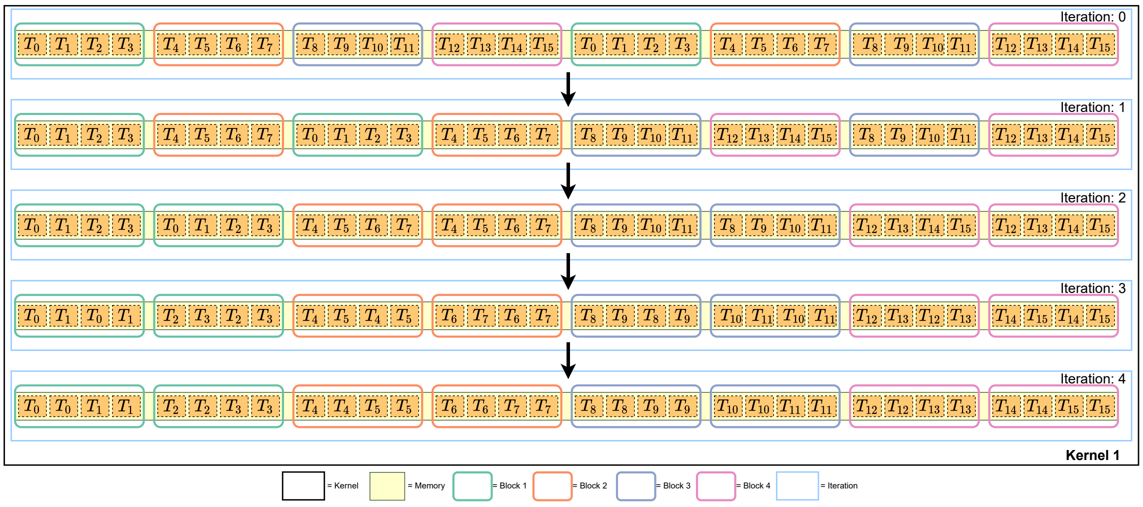

Figure 4.1 Execution of Alg. 17 without synchronization in ideal circumstances where $n = 32$.

synchronization be executed for $n = 32$ on this GPU. Figure 4.1 portrays a visualization of the execution of Algorithm 17 without synchronization on the hypothetical GPU.

The figure shows a total of $\log_2 32 = 5$ iterations. 16 threads are used, whose indexes are between 0 and 15 ($T_0, \ldots, T_{15}$). We use a different color for the oval rectangle that encircles a block of four threads. Each thread T_i accesses two different memory locations using the $address$ and $address + t$ values in Algorithm 17, performs the butterfly operation, writes two pieces of the results again in the same two memory locations. When a thread finishes its own task, it moves to the next iteration of the algorithm and performs the same operation, only with a different value of t this time. Figure 4.1 represents the execution of the algorithm in the ideal circumstances as the threads are assumed to finish their tasks simultaneously.

However, Algorithm 17 does not always execute in ideal circumstances. Suppose that the number of threads is only 12 for the scenario in Figure 4.1. Even when the number of threads is less than the number of tasks, incidentally the execution can still be correct. One such scenario is depicted in Figure 4.2, where we only show the first two iterations. As the scenario requires 16 threads, but the hypothetical GPU has 12 threads, the number of threads is not sufficient, and the code runs sequentially after a point. In the figure, we use primed letters to distinguish the multiple assignments of the same thread to different tasks. For instance, T_0 and T_0' show that the thread executes two different butterfly operations in the same iteration sequentially. Incidentally again, this does not necessarily lead to incorrect execution as shown in Figure 4.2.

Nevertheless, since thread synchronization is not implemented, an error in calculations

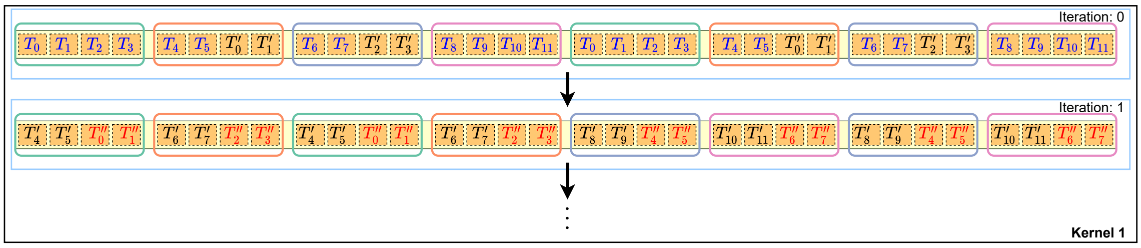

Figure 4.2 One good scenario for Alg. 17 without synchronization, where $n = 32$ and *maximum block size* $= 4$

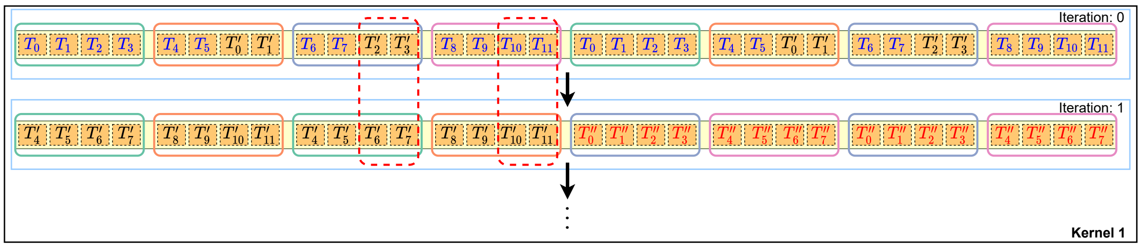

Figure 4.3 One problematic scenario for Alg. 17 without synchronization, where $n = 32$

can occur as visualized in Figure 4.3. For instance, suppose four threads in the dashed red line, namely T_2', T_3', T_6', T_7', are assumed to be scheduled simultaneously. And, since they operate on the same memory locations in two consecutive iterations, there is a data dependency between the first two and the last two threads. This will definitely lead to a race condition, resulting in incorrect results.

It is impossible to put a barrier between the warps to solve the aforementioned synchronization problem. Therefore, only block-level barriers can be used as shown in Step 12 of Algorithm 17, which resolves all synchronization problems as long as the ring dimension n is less than or equal to the block size. Since a block in GPU has a maximum of 1024 threads for all GPU models, the barrier `__syncthreads()` in Step 12 of Algorithm 17 cannot resolve the synchronization issue for higher values of n or when performing many NTT operations in batches[1].

The latter issue can be explained over another execution scenario of Algorithm 17 on the hypothetical GPU, depicted in Figure 4.4. The eight threads enclosed in the dashed red line belong to two different blocks as the block size of the hypothetical GPU is just four. Here, the thread block in the 2nd iteration run on data that has not yet been completed, leading to incorrect results.

[1]When multiple and independent NTT operations are executed, the threads are scheduled as if those independent NTT calculations are combined into a single big NTT operation.

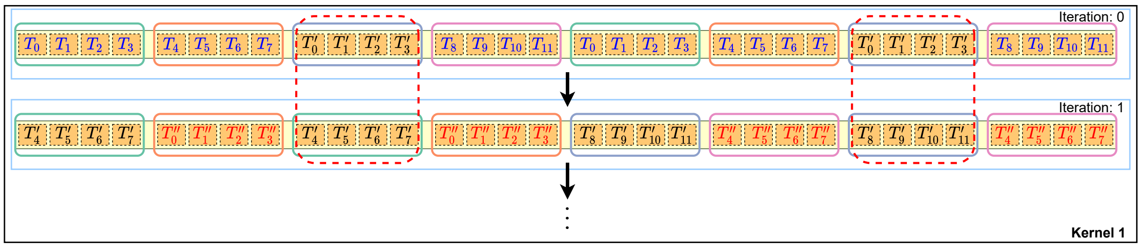

Figure 4.4 Another problematic scenario for Alg. 17, where $n = 32$

For values of n much higher than the block size and the high number of multiple NTT operations running simultaneously, an obvious solution to resolve all synchronization issues is simply using more than one kernel depending on the size of n or the number of NTT computations. For example, for $n = 32$ on our hypothetical GPU, to resolve the synchronization issue in Figure 4.4, we can use two consecutively executing kernels for the first two iterations of the NTT computation.

After the first two iterations are completed, the threads in one block will never need or use the data processed by another block and no synchronization problem occurs. Therefore, after the first two iterations, execution can continue within the third kernel using shared memory and block synchronization.

However, when more than one kernel is used, the only way to share data across kernels is to use global memory, which is the slowest of all GPU memory types (see Table 3.2). As the value of n increases, this method becomes prohibitively inefficient as the number of kernels required for NTT will also increase. This approach is used in (Özerk et al., 2022), and as demonstrated in the subsequent sections, our new approach in this thesis scales better as n increases.

The approaches described above have either synchronization issues or inefficient memory usage, both of which are efficiently addressed by our new NTT implementation. The new implementation consists of always two kernels for all values of n. An example with $n = 32$ is visualized in the hypothetical GPU in Figure 4.5, where the first two iterations are performed in the first kernel. Operations of these iterations in the first kernel are performed sequentially on purpose. In the example in Figure 4.5, two blocks and eight threads (recall 2 blocks $= 8$ threads on the hypothetical GPU) are scheduled twice in the first two iterations. Each thread in the two blocks writes the addresses of interest in the global memory to its registers, as illustrated in Figure 4.6. Then, each thread performs butterfly operations using its register memory. When a thread finishes its task in one iteration, it writes the data in its register memory to the corresponding global memory. Although the first kernel seems to be slower because it uses fewer number of threads

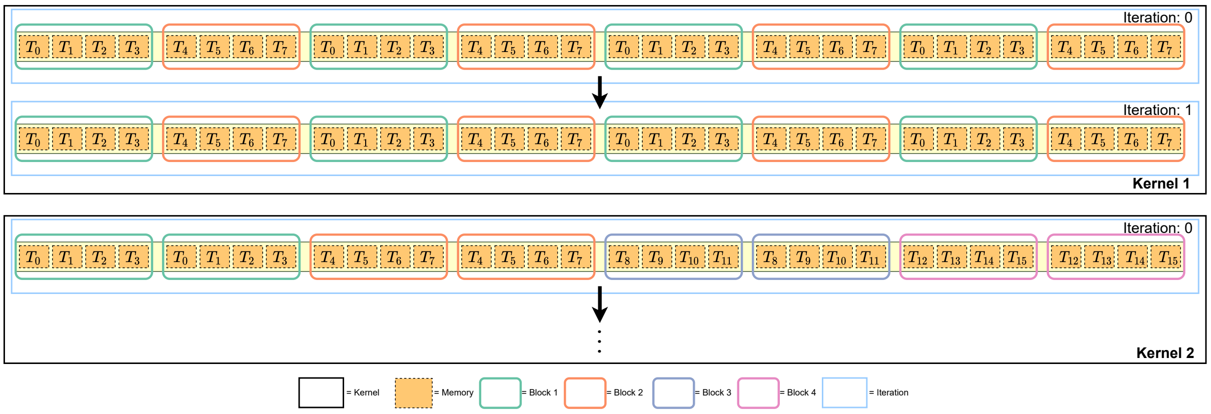

Figure 4.5 An example of our NTT algorithm where, $n = 32$ and $maximum\ block\ size = 4$.

than the above mentioned examples, the acceleration here comes not from the number of threads, but from the more efficient usage of memory as we minimize the number of global memory accesses. On the other hand, in the approach employed in (Özerk et al., 2022), the number of kernels along with global memory access increases as n becomes larger. The pseudo-code for the algorithm used to implement the operations in the first kernel is given in Algorithm 18.

In the second kernel in Figure 4.5, the number of threads in a block suffices to complete the remaining NTT operations. Since data sharing among threads within the block is required, each block has its shared memory consisting of $2 \times blocksize$. This poses no problem as the shared memory is the fastest type of GPU memory. Here, each thread accesses its own part of the memory using its respective indexes for each iteration, performs a butterfly operation, and writes the result to the shared memory of the block it is connected to until the last iteration. After all threads finishes their executions, the result, which is in the shared memory, are written to these global memory, and the NTT operation is terminated. This NTT implementation is fast and free of synchronization issues for power of two n values between 2^{12} and 2^{15}

This NTT implementation is fast and free of synchronization issues for power of two n values between 2^{12} and 2^{15} and multiple concurrent NTT computations. However, It is impossible to implement n size higher than 2^{15} using this implementation due to its register usage. However, the following section expresses a new method showing how NTT and INTT can be implemented efficiently for any n values.

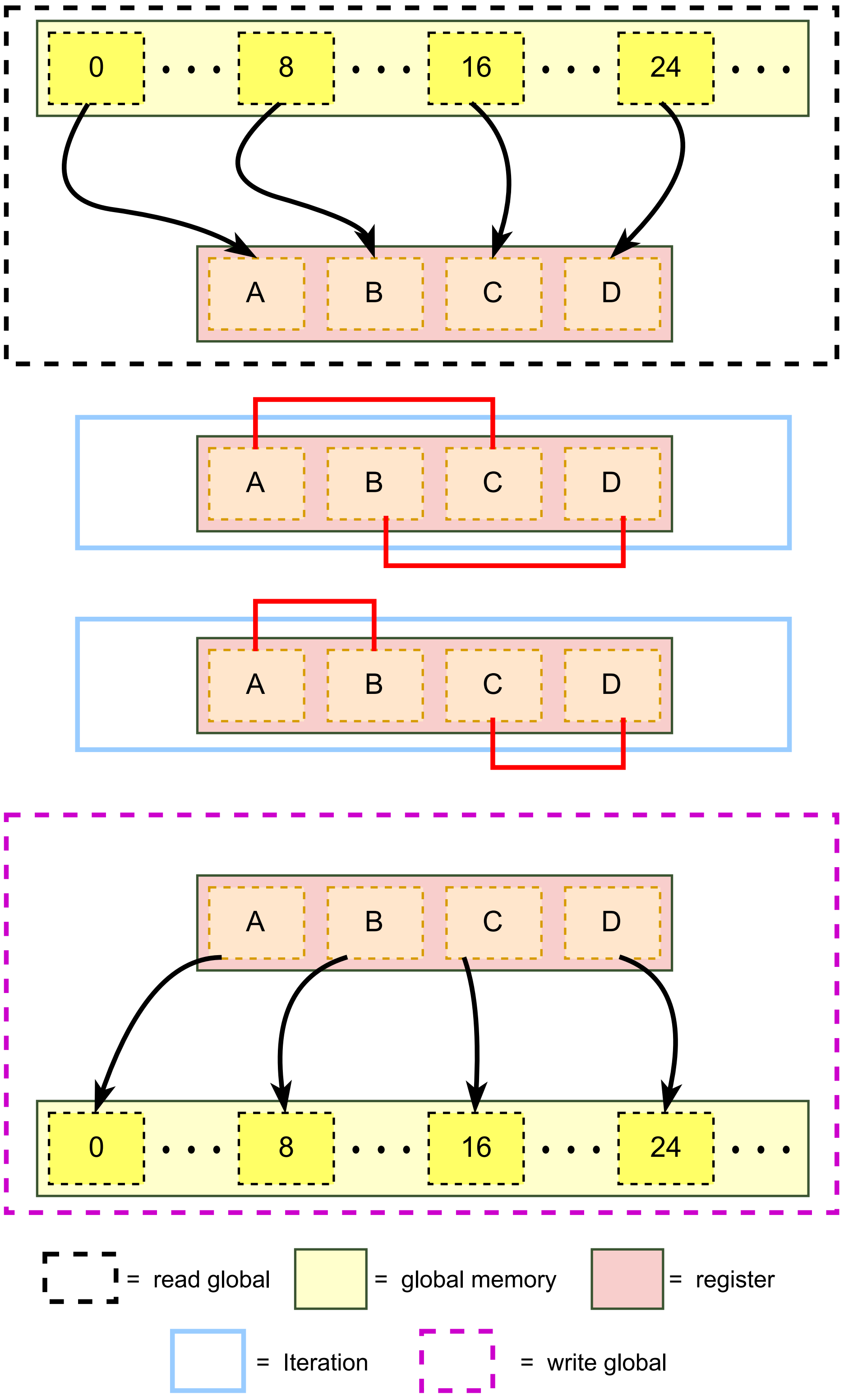

Figure 4.6 Register Memory Usage in our NTT Algorithm for $n = 32$.

4.2 MERGE-NTT GPU IMPLEMENTATION APPROACH 2

Algorithm 18 Kernel 1 in Figure 4.5

```
Input: $A[n], PsiTable[n], q$
Input: $bc$: no. of blocks ($bc = 2$)
Output: $A[n]$
 1: $idx = blockIdx.x \times blockDim.x + threadIdx.x$
 2: $m = 1$
 3: $k = n/(2 \times blockDim.x \times bc)$
 4: $t = n$
 5: for $i$ from 0 to $n/(2 \times blockDim.x\ )$ do
 6:     $reg[i] = A[idx + (i \times (2 \times blockDim.x))]$
 7: for $i$ from 0 to $\log_2(n/(2 \times blockDim.x \times bc)) + 1$ do
 8:     for $j$ from 0 to $n/(2 \times blockDim.x \times bc)$ do
 9:         $location = \lfloor \frac{j}{k} \rfloor \times k + j$
10:         $U = reg[location]$
11:         $V = reg[location + k]$
12:         $address = \lfloor \frac{idx}{t} \rfloor + m$
13:         $V = (V \times PsiTable[address]) \mod q$
14:         $reg[location] = (U + V) \mod q$
15:         $reg[location + k] = (U - V) \mod q$
16:     $m = m \times 2$
17:     $k = k/2$
18:     $t = t/2$
19: for $i$ from 0 by 1 to $n/(blockDim.x \times 2)$ do
20:     $A[idx + (i \times (blockDim.x \times 2))] = reg[i]$
```

In this section, we explain our second GPU algorithm of computing NTT, also based the recursive NTT algorithm, namely, Algorithm 6, which is free of limitations of Algorithm 18.

The **Merge-NTT** consists of two parts: i) **NTT host**, runing on the host device (i.e., general-purpose CPU), which determines the block size, block and grid shapes and the number of kernels etc., and ii) **NTT kernel** which performs the actual NTT computation on GPU. As observable in Algorithm 6, **Merge-NTT** consists of 3 main parts: i) the outer loop (the “**do-while**” loop starting in line 2), ii) the inner loop (the ”**for**” loop in line 4), and iii) butterfly operation (**CT** Coley-Tukey) in line 8). In the first outer loop iteration, there is one NTT operation operating on the entire elements of the input vector. The number of *independent NTT operations* doubles from one iteration to the next operating on the separate parts of the input vector. For instance, in the second iteration, we have two independent NTT operations operating the first and the second half of the input vector, respectively. We can even assign indices to the NTT operations in a outer loop iteration, increasing from left to right for easy reference to them.

While the outer loop needs to be executed sequentially as there is data dependency between an iteration of the outer loop and the next, the inner loop iterations are independent and, therefore, suitable for parallelization. The threads in a block can perform all iterations of the inner loop concurrently and use the `__syncthreads()` intrinsic function for synchronization provided that $bDim \geq n/2$ where n is the ring dimension. Otherwise, more than one block is needed for full parallelization of the inner loop and the synchronization becomes problematic as the only way for that is via global memory, which results in using multiple kernels. For example, when $n = 2^{11}$, where there are 1024 **CT** operations in each inner loop iteration, one block (and one kernel) suffices to implement NTT operation as the maximum number of threads in CUDA-capable GPUs is 1024.

When $n > 2^{11}$, however, multiple kernels will be needed and the approach adopted in Özerk et al. (2022) uses a new kernel for each outer iteration until the iteration number $\log_2 n - \log_2 2bDim$. Therefore, the number of kernels can be calculated using the formula $kc = \log_2 (n/2bDim)$. For example, when $n = 2^{15}$ there are 15 outer iterations, the number of kernels needed can be computed as 4 for $bDim = 1024$. When the ring dimension increases to 2^{20} and 2^{24}, 9 and 13 kernels are needed, respectively, which renders the approach in Özerk et al. (2022) prohibitively inefficient for high ring dimensions. The work in Özcan et al. (2023) (also our first method outlined in Section 4.1), adopting a completely different approach, performs all outer iterations up to $\log_2 n - \log_2 2bDim$ in a single kernel, whereby some iterations of the inner loop is serialized. The second kernel is used thereafter as inter-block dependency is no longer an issue. This way, access to global memory is reduced using only two kernels. But, unfortunately, it can only perform NTT operation up to $n = 2^{15}$ as the number of registers in the SM is insufficient to get all vector elements from the global memory at the start of the kernel.

We can keep track of the number of outer loop iterations performed in each kernel in an array, named koc (kernel outer iteration count). For example, for the method in Özerk et al. (2022), the number of kernels (kernel count) $kc = \log_2 n - \log_2 2bDim$ and the elements of the array can be written as $koc[0] = \log_2 2bDim$ and $koc[i] = 1$ for $i \geq 1$. For Özcan et al. (2023), we have $kc = 2$ and $koc[0] = log_2 2bDim$, $koc[1] = \log_2 n - \log_2 2bDim$ with $n \leq 2^{15}$. Nevertheless, the partitioning of the outer loop iterations into kernels can be done in different ways to obtain a better GPU implementation as demonstrated in the rest of this thesis.

Suppose the function `Partition` (see Algorithm 19) gets the ring dimension n, the maximum number of threads in a block (typically $mbd = 1024$ in CUDA-enabled GPU devices), and returns the optimal partition for n along with the block size $bDim$, without the particular shapes of a block and grid; namely their dimensions in different coordinates, which may vary depending on the particular kernel. Note that the returned block size is not necessarily the maximum block size and it turns out a smaller block size may be advantageous as will be shown in the subsequent sections. For example, when $bDim = 1024$ and $n = 2^{18}$ we can perform NTT in two kernels with seven outer loop iterations in the first kernel and 11 in the second kernel. Namely, we can write $kc = 2$, $koc[0] = 11$ and $koc[1] = 7$. In another example, when $bDim = 1024$ and $n = 2^{24}$, we have $kc = 3$, $kc[2] = 2, kc[1] = kc[0] = 11$. On the other hand, when we use a smaller block size such as $bDim = 256$, then we have $kc = 3$, $kc[2] = 6, kc[1] = kc[0] = 9$. Note that a kernel cannot perform more than $\log_2 2bDim$ outer loop iterations. One particular contribution of ours is that we propose such a novel access model to global memory that blocks in the kernels for $i \geq 1$ (i.e., those except the last kernel running the last $\log_2 2bDim$) perform the computations using shared memory in all outer loop iterations.

Algorithm 19 NTT HOST

```
Input: $A[n], PsiTable[n], n, q, mbd$
Output: $A[n]$                                              # In-place calculation
 1: $\{bDim, kc, koc\} \leftarrow$ Partition$(n, mbd = 1024)$        # Optimal partition
 2: $bc \leftarrow n/(2 \times bDim)$                                 # # of blocks
 3: $olc \leftarrow log_2(n)$                             # # of outer loop iterations
 4: $oc \leftarrow -1$
 5: for $i$ from 0 by 1 to $kc-1$ do
 6:     $oc \leftarrow oc + koc[i]$
 7:     $ko[i] \leftarrow 2^{oc}$
 8:     $olc \leftarrow olc - koc[i]$
 9:     if $i = 0$ then
10:         $kgs[i] \leftarrow [1, bc]$
11:         $kbs[i] \leftarrow [bDim/ko[i], ko[i]]$
12:     else
13:         $kgs[i] \leftarrow [bc/(2^{olc}), 2^{olc}]$
14:         $kbs[i][1] \leftarrow (2 \times ko[i-1])/kgs[i][1]$
15:         $kbs[i][0] \leftarrow bDim/kbs[i][1]$
16: $m \leftarrow 1$
17: for $i$ from $kc-1$ by $-1$ to 0 do
18:     $dim3$ $\mathbf{B}(kgs[i][0], kgs[i][1])$
19:     $dim3$ $\mathbf{T}(kbs[i][0], kbs[i][1])$
20:     NTT $\lll \mathbf{B}, \mathbf{T} \ggg (A, PsiTable, m, ko[i], koc[i], q)$
21:     $m \leftarrow m \times (2^{koc[i]})$
```

The proposed approach for determining the block size, kernel count, grid and kernel shapes is detailed in Algorithm 19, which is intended to execute on the host device to invoke kernels. In line 1 of the algorithm, the function `Partition` takes the ring dimension n and the maximum block size on CUDA-enable GPU (by default $mbd = 1024$) and returns the block size ($bDim \leq 1024$), the number of kernels ($kc \geq 2$ for $n > 2bDim$) and the array koc, whose elements keep the number of outer iterations in the corresponding kernel in reverse index; e.g., $i = 0$ and $i = kc - 1$ represents the last and the first kernels, respectively. The function `Partition` relies on empirical investigation to a certain extent as shown in the subsequent sections.

In partitioning the outer loop iterations into kernels, it may seem intuitive to perform as many as possible iterations in last kernels and more likely fewer number of them in earlier kernels. Different partitioning schemes, however, can also benefit the memory access performance; but care must be taken on deciding the optimal partitioning.

In Algorithm 19, bc stands for the number of blocks in the grid (block count) while olc represents the number of outer loop iterations remaining to be performed. The kernel offset ko, keeps the difference between the indices of the vector elements in the butterfly operation at the start of a kernel. For instance, in the last kernel, in which each block processes $2bDim$ vector elements, $ko = bDim$ while $ko = n/2$ in the first kernel.

After the block dimension $bDim$, the number of kernels kc, and the number of iterations in each kernel are determined in Step 1 of Algorithm 19, shapes of grids and kernels are computed in lines between Steps 5 and 18. Block and grid shapes, which simply pertain to their dimensionality, are determined by taking into account the memory dependencies between the outer loop iterations of the NTT algorithm. Both kgs (kernel grid shape) and kbs (kernel block shape) are two-dimensional arrays and their elements keep track of dimensions of grids and blocks in each kernel.

With two-dimensional access structure, one can arrange the blocks of the grid into different *block groups*. The first and second dimensions of kgs designate the number of blocks in each group and the number of block groups, respectively. The blocks, executing the same NTT operation and thus accessing the same range of vector elements, are organized into the same block group.

Example 4.1. *Supposing $n = 2^{24}$ and $bDim = 1024$, the number of blocks is $bc = 8192$. Assume also $koc = [11, 11, 2]$. In the first iteration of the first kernel, all threads in the blocks access the entire input vector, then all blocks in the kernel belongs to the same group. Therefore, we have $kgs[2] = [8192, 1]$. As the first kernel iterates two times ($koc[2] = 2$), the second kernel will process four independent parts of the input vector in four independent NTT operations in its first iteration. Then, we can have four groups of blocks, i.e., $kgs[1] = [2048, 4]$. The final kernel has blocks, which are processing their own parts of the vector. Then, one can think there are as many block groups as the number of blocks. Thus, we have $kgs[0] = [1, bc]$.*

In a similar fashion, we can group the threads in a block, as well. While the first dimension of *kbs* designates the number of threads in each group, the second does the number of thread groups. The grouping strategy depends on the number of iterations in the kernel. The goal is simply to ensure that the threads in a block will access the same vector elements in all outer iterations performed in the kernel.

Example 4.2. *In Example 4.1, the first kernel performs the first two iterations of the outer loop as* $koc[2] = 2$*. Then, a block is grouped into two thread groups with* 512 *threads in each; namely,* $kbs = [512, 2]$*. This way, one group of threads will be accessing the vector elements in the second iteration, which are processed by the other thread group in the first iteration. As the first and second groups are in the same block, they use the same shared memory, which will eliminate accessing the global memory. In the second kernel, as there are 11 iterations, the block is organized into 1024 thread groups with a single thread in each group; namely* $kbs = [1, 1024]$*. Finally, in the last kernel, we can place all threads in the same group as a block is guaranteed to access the same* $2bDim$ *elements of the vector (i.e.,* $kbs = [1024, 1]$*).*

As observed in Example 4.2, the thread groups are excessively fragmented in the second kernel. Since threads in the same block process the vector elements that are located in distant locations in memory, this can adversely affect the memory access performance due to uncoalesced access pattern. Especially, if we can place the threads that access the same memory block (i.e., 128 B) in the same group, memory access will be optimized. Then, different partitioning of outer loop iterations into kernels should be considered.

Example 4.3. *Suppose* $koc = [11, 7, 6]$ *for* $n = 2^{24}$ *and* $bDim = 1024$*. Then we will have*

$$kgs = [[1, 8192], [128, 64], [8192, 1]] \text{ and}$$

$$kbs = [[1024, 1], [16, 64], [32, 32]].$$

Here, in the first two kernels, there are 32 and 16 threads in thread groups, respectively. For instance, in the first iterations of the second kernel, 16 threads access 16 consecutive vector elements from the global memory, which is likely to be kept in the same memory block. If each thread accesses 8 B data types, this will result in a perfect match with the size of the memory block of 128 B.

Working with the maximum block dimension of $bDim = 1024$ may not always result in optimum performance, as good *occupancy* rate cannot be achieved due to poor resource utilization as explained in Section 3.4. For instance, if we use 64-bit arithmetic (8 B) in the computation of NTT, then, each block uses 2048×8 B of the shared memory for the operands of the butterfly operation. This turns out to result in only a poor occupancy rate of 66% as an SM in a GPU device with compute capability 8.6 can run maximum of 1534 threads (see Table 3.1). If, however, $bDim = 256$, then the maximum theoretical occupancy will be achieved as an SM can run more blocks at the same time with each block using $512 \times 8 = 4$ KB. If, for example, the SM runs 6 blocks of 256, then computation uses 24 KB of the shared memory.

To see the effects of the occupancy rate on the performance we ran a set of experiments. We executed our NTT algorithm with two different block dimensions, $bDim = 1024$ and $bDim = 256$, on three GPU devices. As seen in Table 4.1, better occupancy rate can lead to more than 10% improvement in execution times.

Table 4.1 Effect of the block dimension on performance with $n = 2^{17}$.

$bDim$	koc	**GPU-A**	**GPU-B**	**GPU-C**
1024	[11,6]	30.1 μs	24.3 μs	12.2 μs
256	[9, 8]	25.5 μs	21.0 μs	12.2 μs

From the discussions, we can conclude that there are couple of factors that determine the overall performance: block dimension, the number of kernels, the number of outer iterations in the kernels, kernel and grid shapes. As all the internal architectural details of the GPU devices and the scheduling of threads in SMs are known to a certain extent, an exact formula for choosing the best value of a factor cannot be given. For example, maximum block dimension of $bDim = 1024$ for NTT computation of high ring dimensions is not optimum due to poor occupancy rate. However, it is not easy to determine whether $bDim = 256$ or $bDim = 128$ is better although both enjoy maximum occupancy. Depending on the ring dimension and other factors, $bDim = 128$ may not fully utilize coalesced access to the memory. All our experiments support that using $bDim = 256$ is the optimum choice. For the number of outer iterations in the kernels koc, we rely on experimental observations and manual adjustments to a certain extent.

Using the configuration obtained in Algorithm 19 for block dimension, kernel count, kernel and block shapes, the **NTT Kernel** function in Algorithm 20 is called in Step 20 of Algorithm 19. The index $GAddr$ in Algorithm 20 is used to access the global memory for the elements of the input vector A when the kernel is started. The threads access the global memory with $GAddr$ for array elements, which are placed in the shared memory. The size of the shared memory for each thread block depends on the block dimension $BDim$ and the size of input vector elements, w (e.g. $w = 4$ B or $w = 8$ B), and can be computed as $2 \times BDim \times w$. In order to exploit the coalesced access to the global memory, the threads are organized into groups, whose member threads access the global memory with consecutive indices of the input vector.

Algorithm 20 NTT KERNEL (NTT)

```
Input: $A[n], PsiTable[n], m, ko, koc, q$
Output: $A[n]$
 1: $t_1 \leftarrow bDim.x \times bDim.y$                              # Block dimension
 2: $\ell_1 \leftarrow tID.y \times (ko/2^{koc-1})$                    # Offset btw. thread groups
 3: $\ell_2 \leftarrow bDim.x \times bID.x$                            # offset within an NTT
 4: $\ell_3 \leftarrow 2 \times ko \times bID.y$                       # offset for an NTT
 5: $GAddr \leftarrow tID.x + \ell_1 + \ell_2 + \ell_3$                # For global mem.
 6: $SAddr \leftarrow tID.x + tID.y \times bDim.x$                     # For shared mem.
 7: $PsiAddr \leftarrow tID.x + \ell_1 + \ell_2 + \ell_3/2$            # For twiddle factors
 8: $offset_G \leftarrow ko$
 9: $offset_S \leftarrow bDim.x \times bDim.y$
10: SMem$[SAddr] \leftarrow A[GAddr]$
11: SMem$[SAddr + offset_S] \leftarrow A[GAddr + offset_G]$
12: for $i$ from 0 by 1 to $koc - 1$ do
13:     $u \leftarrow \lfloor SAddr/t_1 \rfloor \times t_1 + SAddr$
14:     $PsiIn \leftarrow m + \lfloor PsiAddr/ko \rfloor$
15:     CT(SMem$[u]$, SMem$[u + t_1]$, $PsiTable[PsiIn], q$)
16:     syncthreads()
17:     $m \leftarrow m \times 2$
18:     $t_1 \leftarrow t_1/2$
19:     $ko \leftarrow ko/2$
20: $A[GAddr] \leftarrow$ SMem$[SAddr]$
21: $A[GAddr + offset_G] \leftarrow$ SMem$[SAddr + offset_S]$
```

Example 4.4. *In a hypothetical GPU, assume* $n = 256$ *and* $bDim = 4$ *and the partition is that* $kc = 3$ *and* $koc = [3,3,2]$*. Then, we can compute the grid and block shapes as* $kgs = [[1,32],[8,4],[32,1]]$ *and* $kbs = [[4,1],[1,4],[2,2]]$*, respectively. In the first kernel, the blocks access the entire range of vector elements (as there is one NTT operation), therefore, there is one block group with 32 elements as* $bc = 32$ *and* $bID.x \in [0,31], bID.y = 0$*. The execution of the two outer loop iterations of the first block of the first kernel is depicted in Table 4.2.*

There are two groups of threads in each block and threads in the same group access the consecutive elements of the input vector. Thus, we have $tID.x \in [0,1]$ *and* $tID.y \in [0,1]$*. For example, the two threads in the first group access the four elements with indices [0,128] and [1,129], respectively. Note that, in the second iteration, there is no access to the global memory as all vector elements are already in the shared memory. The offset value of the indices between the first and second group of threads is calculated as* $ko/2^{koc-1} = 64$ *(See* ℓ_1 *in the second step of Algorithm 20).*

Table 4.2 The execution of the first block of the first kernel in Example 4.4

bID	*tID*	*GAddr*	*SAddr*	*Corr.GAddr*	iteration
[0,0]	[0,0]	**[0, 128]**	[0,4]	[0,128]	0
			[0,2]	[0,64]	1
	[1,0]	**[1, 129]**	[1,5]	[1,129]	0
			[1,3]	[1,65]	1
	[0,1]	**[64, 192]**	[2,6]	[64,192]	0
			[4,6]	[128,192]	1
	[1,1]	**[65, 193]**	[3,7]	[65,193]	0
			[5,7]	[128,192]	1

As mentioned previously, thread blocks are organized as two-dimensional arrays in grids and *bID.x* and *bID.y* are indices of a particular block. Here, *bID.y* is the index of the NTT sub-block while *bID.x* is the offset within the NTT sub-block.

Example 4.5. *In Example 4.4, as there is a single NTT operation in the first iteration of the first kernel, there is one block group in a grid; namely we have* $bID.y = 0$ *for all block groups. To calculate the index of A in the global memory, we need to compute the offset value* $\ell_2 = bDim.x \times bID.x$. *For example, when* $bID.x = 1$, *the offset value for the index of A in global memory will be 2 as* $bDim.x = 2$. *See Table 4.3 for the execution of the first three blocks of the first kernel for the first thread groups.*

As there are *bDim.x* threads in each thread group *bID.x* executing the same NTT operation, we need to add the offset value $\ell_2 = bDim.x \times bID.x$ (see Algorithm 20, Step 3) within an NTT operation to the index used to access to global memory at the start of a kernel. Finally, another offset value $\ell_3 = 2 \times ko \times bID.y$ (Algorithm 20, Step 4) is added to the global memory index. Here, *bID.y* is the index of the NTT operation in outer loop iterations, which needs to be multiplied by twice the kernel offset value of *ko*.

As mentioned earlier, *tID.y* is the index of a thread group in the same block, which refers to coalesced thread groups. For example, assume $bDim = 256$ and $kbs = [16, 16]$ (i.e., $tID.x \in [0, 15]$). As stated in Section 3.3, since L1 cache memory and L2 cache memory line sizes are 128 bytes and if we use vector elements of 8 B (64-bit), for the best value of minimum thread group size we should have $bDim.x \geq 16$. Otherwise, the number of clock cycles increases when accessing global memory due to the increase in the number of uncoalesced accesses, which leads to decrease in the bandwidth and increase in the latency of computation.

Except for the last kernel, *bDim.x* decreases as the number of outer iterations in kernels

Table 4.3 The execution of the first three blocks of the first kernel in Example 4.4

bID	*tID*	*GAddr*	*SAddr*	*Corr.GAddr*	iteration
[0,0]	[0,0]	**[0, 128]**	[0,4]	[0,128]	0
			[0,2]	[0,64]	1
	[1,0]	**[1, 129]**	[1,5]	[1,129]	0
			[1,3]	[1,65]	1
	...				
[1,0]	[0,0]	**[2, 130]**	[0,4]	[2,130]	0
			[0,2]	[2,66]	1
	[1,0]	**[3, 131]**	[1,5]	[3,131]	0
			[1,3]	[3,67]	1
	...				
[2,0]	[0,0]	**[4, 132]**	[0,4]	[4,132]	0
			[0,2]	[4,68]	1
	[1,0]	**[5, 133]**	[1,5]	[5,133]	0
			[1,3]	[5,69]	1

increases. Thus, using fewer outer loop iterations in those kernels must be considered. For example, in Algorithm 19, if the kernel performs maximum number of outer iterations (i.e., $log_2(2bDim)$), then we will cause the worst memory access pattern as $bDim.x = 1$.

Example 4.6. *Assume $n = 2^{24}$, $bDim = 1024$, $bc = 8192$. The partitioning the outer loop iterations as $koc = [11, 11, 2]$ will lead to $bDim.x = 512, bDim.x = 1$, in the first and the second kernels, respectively. However, if we use $koc = [11, 7, 6]$, we will have $bDim.x = 32, bDim.x = 16$ for the first two kernels, which will result in much better global memory access pattern.*

Reducing the number of outer iterations in a kernel, on the other hand, will increase the total number of kernels. Therefore, the number of accesses to the global memory will increase; as explained in Section 3.3, which is not good for the overall latency. However, the best NTT implementation can be achieved if a balance is found between the number of accesses to global memory and the number of clock cycles spent accessing global memory.

Example 4.7. *We examine the effect of the kernel count with a concrete example, with $n = 2^{18}$ and $bDim = 256$. We can use two different partitions: $kc_1 = 2$ and $koc_1 = [9, 9]$; and $kc_2 = 3$ and $koc_2 = [9, 5, 4]$. The results are given in Table 4.4. As can be observed in Table 4.4, when $kc = 2$, $bDim.x = 1$ for the first kernel, which will result in inferior latency. On the other hand, when three kernels are used, as the global memory access patterns are much better (due to $bDim.x \geq 16$), the latency values are improved. As can*

Table 4.4 Effect of the number of kernels on performance with $n = 2^{18}$ and $bDim = 256$.

koc	*bDim.x*	**GPU-A**	**GPU-B**	**GPU-C**
[9,9]	[256, 1]	53.0 μs	31.8 μs	21.2 μs
[9, 5, 4]	[256, 16, 32]	41.7 μs	28.4 μs	15.5 μs

be seen from the results, using extra kernels may be advantageous depending on input parameters, if global memory access patterns result in poor performance despite fewer number of kernels.

4.3 4STEP-NTT GPU IMPLEMENTATION

As can be observed in Algorithm 10 given for **4Step-NTT**, the algorithm consists of six main operation blocks: i) transpose of $n_1 \times n_2$ matrix B (Step 6), ii) n_2 n_1-point NTT of rows of B (Steps 7-9), iii) transpose of $n_2 \times n_1$ matrix B (Step 10), iv) multiplication with twiddle factors (Steps 11-15), v) n_1 n_2-point NTT of columns of B (Steps 16-18), and vi) transpose of $n_1 \times n_2$ matrix B (Step 19). Here, the NTT operation blocks can be performed in parallel as there are n_2 or n_1 independent NTT operations in each block.

Note that the vector-to-matrix and matrix-to-vector operations do not have to performed explicitly. Note also that the first and the last transpose operations are not necessary and can be skipped provided that both NTT and inverse NTT operations do not perform them. This naturally necessitates modification in data access patterns, which will not pose any significant performance penalty. The transpose operation, on the other hand, can be prohibitively expensive for large matrices, and eliminating them results in improvement in the latency. Finally, the multiplication with twiddle factors in fourth operations block can be incorporated into the second NTT operation block. With these optimizations, the **4Step-NTT** algorithm is simplified to have only two NTT operation blocks and a transpose operation in between.

In both NTT operation blocks, there are many independent NTT operations of much smaller sizes than those used in the **Merge-NTT** algorithm, which can be performed in parallel. The number and the sizes of NTT operations in the first and second NTT blocks can be important in the performance of its GPU implementation. Although in the original **4Step-NTT** algorithm, it is suggested that B be a square matrix (or as close to square matrix as possible), namely $n_1 \approx n_2$, the algorithm works with various selections of n_1 and n_2. Then, we need to determine specific values of of n_1 and n_2 for a given n to optimize the second transpose operation, which may be problematic as it can result in costly memory accesses.

Example 4.8. *Consider the ring dimension of $n = 2^{22}$, where the coefficients of input polynomial can be arranged into a $2^{11} \times 2^{11}$ matrix; i.e., $n_1 = n_2 = 2^{11}$. Then, the first NTT block consists of 2^{11}-point NTT operations. And, considering $bDim \leq 1024$, one block can only process at most $2bDim/n_1 = 4$ NTT operations. Consequently, only a small number of threads in a block will access the consecutive addresses in the global memory during the subsequent transpose operation. This will lead to sub-optimal access pattern to the global memory, which adversely affects the latency. Thus, after the first NTT operation block, it will be more efficient to terminate the kernel and launch another that performs transpose operation through shared memory. Although using an additional kernel for the transpose increases the number of global memory accesses, from latency perspective this turns out to be more efficient compared to storing the matrix elements in transposed format directly to global memory in the same kernel after the first NTT operation block. This is due to the fact that performing the transpose by the existing threads of the kernel blocks through global memory will lead to uncoalesced accesses by the threads, resulting in increased latency for global memory access. Instead, that storing the matrix elements to the global memory before the transpose and then loading them in a new kernel will enable to perform the transpose in the shared memory can be much more efficient. Therefore, using an additional kernel enables the transpose process to be performed with much lower latency.*

Alternatively, using a smaller value of n_1 can be advantageous to improve the memory access pattern during the transpose operation performed in the same kernel as the first block of NTT operations. For instance, we can use the dimensions $n_1 = 2^7$ and $n_2 = 2^{15}$ for $n = 2^{22}$. This way, each block's shared memory can be considered as two-dimensional array, each row of which corresponds to an independent NTT operation and by this means as many as $2bDim/n_1 = 16$ NTT operations can be performed for $n_1 = 128$. After all NTT operations completed, columns of the array are read by threads and stored to the global memory exploiting the advantages of coalesced accesses. In summary, using a suitably small values of n_1, one can eliminate the extra kernel for the transpose operation without the adverse effect of sub-optimal global memory accesses.

Table 4.5 shows the timing results of **4Step-NTT** *based on five different cases for $n = 2^{22}$ for the two matrix dimensions $(n_1, n_2) \in \{(2^{11}, 2^{11}), (2^7, 2^{15})\}$ on three different GPU devices, where different implementation techniques are applied.*

Table 4.5 Effect of the matrix dimension on the performance of the 4Step NTT Algorithm

n	**case**	$[n_1,n_2]$	**GPU-A**	**GPU-B**	**GPU-C**
	1	$[2^{11},2^{11}]$	1219.29 μs	439.99 μs	342.51 μs
	2	$[2^{11},2^{11}]$	1226.70 μs	413.65 μs	256.77 μs
2^{22}	3	$[2^{11},2^{11}]$	893.27 μs	302.35 μs	190.70 μs
	4	$[2^{7},2^{15}]$	780.11 μs	296.16 μs	175.92 μs
	5	$[2^{7},2^{15}]$	617.26 μs	252.71 μs	142.84 μs

The first three cases in Table 4.5 capture the effect of transpose operations in performance. For instance, in cases 1 and 2, the first and last transpose operations explained in Algorithm 10 are included in timings. In case 1, all three transpose operations described in Algorithm 10 are performed in the NTT kernels, which has an adverse effect on performance because of uncoalesced access to the global memory. In case 2, on the other hand, using an additional kernel for each transpose results in much better performance compared to case 1 for A 100 and RTX 4090.

As explained earlier, when used in an application, there is no need to execute the first and the last transpose operations. In case 3, only one transpose operation is performed in a separate kernel, which leads to significant acceleration.

In cases 4 and 5 are we use a rectangular matrix $(n_1,n_2) = (2^7,2^{15})$, where there is a significant speedup in comparison with the first three cases. The difference between cases 4 and 5 is that case 5 does not use an additional kernel for transpose operation. Since n_1 is small, many NTT operations can be performed in the same GPU block. After the NTT operation, the transpose operation can be performed in the same kernel via reading the columns of the shared memory.

The dimension of the second block of NTT operations also plays an important role and very large values of n_2 can lead to performance penalties. A balance between n_1 and n_2 should be reached to achieve the best performance. Table 4.6 contains the matrix dimensions for all ring sizes of interest, which is found to give the best performance in each case experimentally.

Table 4.6 Matrix Sizes for 4Step NTT Implementation

n	$n_1 \times n_2$	n	$n_1 \times n_2$
2^{12}	$2^5 \times 2^7$	2^{19}	$2^5 \times 2^{14}$
2^{13}	$2^5 \times 2^8$	2^{20}	$2^5 \times 2^{15}$
2^{14}	$2^5 \times 2^9$	2^{21}	$2^6 \times 2^{15}$
2^{15}	$2^6 \times 2^9$	2^{22}	$2^7 \times 2^{15}$
2^{16}	$2^7 \times 2^9$	2^{23}	$2^7 \times 2^{16}$
2^{17}	$2^5 \times 2^{12}$	2^{24}	$2^8 \times 2^{16}$
2^{18}	$2^5 \times 2^{13}$		

4.4 BFV GPU Library

This section explains our GPU implementations of homomorphic addition, multiplication, relinearization and rotation operations of the BFV homomorphic encryption scheme. Algorithms for all these homomorphic operations are given as pseudo-codes as implemented in the Microsoft SEAL library. All algorithms are implemented so that they use our GPU implementation of the NTT algorithm as described in Section 4.1 and Section 4.2.

4.4.1 Key Generation

As shown by Algorithms 12 and 13 in Section 2.5.3.1, key generation operations need uniform and Gaussian random number generators. The cuRAND[2] library, which provides facilities that focus on the simple and efficient generation of high-quality pseudorandom and quasirandom numbers, is used for random number generation on GPU. Since, all random numbers generated in kernel, the number of global memory access as well as latency are reduced. Rest of the key generation algoritm algorithms require polynomial multiplication, so that we used efficient GPU NTT algoritm, which explained before to convert polynomials to the NTT domain. Then, the polynomial multiplications are performed element-wise for all RNS bases. Since BFV ciphertexts need to be in the polynomial domain, after the polynomial multiplication performed in the NTT domain, the results are converted to polynomial domain.

[2]https://docs.nvidia.com/cuda/curand/

4.4.2 Encryption

Similar to the key generation operation, the encryption operation also needs random number generation. As expressed in Algorithm 14 in Section 2.5.3.2, random numbers are multiplied and added with public key easily and parallel in a single kernel. Iterations given in lines 6 and 10 are merged into a single kernel.

4.4.3 Decryption

The decryption of the BFV scheme is implemented using Algorithm 15 in Section 2.5.3.3. Unlike key generation and encryption operations, decryption operation needs no random number generation and thanks to RNS, all for loops can be directly implemented in parallel. All operations except for NTT and INTT operations for the $ct[1] \times sk$ multiplication are merged as much as possible to maintain a minimum kernel count.

4.4.4 Addition/Substract

As explained in Section 2.5.3.4, addition/subtraction operations of the BFV scheme are simple and inexpensive and their implementation consists of only one kernel. In this kernel, each ring element is represented as a vector over Z_{q_i} for each modulus in the RNS base, and modulo addition/subtraction is performed over the elements of the vectors.

4.4.5 Multiplication

In addition to kernel functions to implement NTT and INTT operations, ten different CUDA kernel functions are implemented for the multiplication operation (see Figure 2.1 for these operations). Each of the kernel functions use a one-dimensional block and thread indexing. Before the GPU computation, all necessary parameters are generated on CPU of the host computer, then sent to GPU. In what follows, we briefly mention all of them, but provide pseudo-codes for some important ones in case they are more involved.

Algorithm 21 Fast Convert Array

```
Input: $r_i^n \in \mathbf{R}_{q_i,n}$
Input: $P_i = [\frac{q_i}{q}]_{q_i}$, $base\_c_i^j = [\frac{q}{q_i}]_{Bsk_j}$
Output: $ct_i^n \in \mathbf{R}_{Bsk,n}$
1: for $i$ from 0 by 1 to $r-2$ do
2:     $mult_i^n = [r_i^n \times P_i]_{q_i}$
3: for $i$ from 0 by 1 to $Bsk\_len - 1$ do
4:     $sum_i^n = 0$
5:     for $j$ from 0 by 1 to $r-2$ do
6:         for $k$ from 0 by 1 to $n-1$ do
7:             $sum_i^k = [sum_i^k + (mul_i^k \times base\_c_i^j)]_{Bsk_j}$
8:     $ct_i^n = sum_i^n$
```

Algorithm 22 Fast Floor

```
Input: $r_i^n \in \mathbf{R}_{q_i,n}$, $r_j^n \in \mathbf{R}_{Bsk_j,n}$, $P_i = [\frac{q_i}{q}]_{q_i}$
Input: $base\_c_i^j = [\frac{q}{q_i}]_{Bsk_j}$
Output: $c_i^n \in \mathbf{R}_{q_i,n}$
1: $ct_i^n \leftarrow fast\_conv\_array(r_i^n, P_i, base\_c_i^j)$
2: for $i$ from 0 by 1 to $Bsk\_len - 1$ do
3:     for $k$ from 0 by 1 to $n-1$ do
4:         $ct_i^k = [r_j^k - ct_i^k]_{Bsk_i}$
5:         $c_i^k = [ct_i^k \times [q]_{Bsk_i}^{-1})]_{Bsk_i}$
```

The first two CUDA kernel functions are employed to implement base conversion operation from the RNS base $\mathcal{Q}$ to $\mathcal{B}_{sk}$. The pseudo-code of the base conversion operation is given in Algorithm 21, as it is implemented in the Microsoft SEAL library. In particular, the first kernel implements the `for` loop in lines 1-2 of Algorithm 21. As the result of the `for` loop is needed in the subsequent operations in lines 3-8 of Algorithm 21 a second kernel is used.

The third CUDA kernel function is implemented to perform the small Montgomery reduction operation (i.e., `sm_mrq` Figure 2.1), which is employed to eliminate errors due to the base conversion operation in the previous step. After the NTT operations are applied to all vectors both in the RNS and extension bases, the fourth and fifth CUDA kernel functions are used to perform multiplication of the ciphertexts; the former in the RNS base $\mathcal{Q}$, `multip_q` and the latter in the extension base $\mathcal{B}_{sk}$, `multip_BSK` (see the middle block in Figure 2.1). Then, the INTT operation follows the multiplication operation to convert the ciphertexts back to the polynomial domain. The sixth CUDA kernel function is used to implement the multiplication of ciphertexts with the plaintext modulus t, `multip_t`. The seventh CUDA kernel function, named `first_fast_floor`, implements

Algorithm 23 Switch Key

```
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 $k \in \{0,1\}$, $0 \le j < r$, and $0 \le i < (r-1)$
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:     for $j$ from 0 by 1 to $r-1$ do
 4:         for $k$ from 0 by 1 to 1 do
 5:             $a_{i,j,k} = [NTT_{n,q_j}(c_i[2]) \odot evk_i^j[k]]_{q_j}$
 6:             $\bar{A}_{j,k} = [\bar{A}_{j,k} + a_{i,j,k}]_{q_j}$
 7: for $j$ from 0 by 1 to $r$ do
 8:     $A_{j,0} = INTT_{n,q_j}(\bar{A}_{j,0})$
 9:     $A_{j,1} = INTT_{n,q_j}(\bar{A}_{j,1})$
10: $half = \lfloor \frac{q_{r-1}}{2} \rfloor$
11: for $i$ from 0 by 1 to $r-1$ do
12:     $halfmod = [half]_{q_i}$
13:     for $k$ from 0 by 1 to 1 do
14:         $tmp = [[A_{r-1,k} + half]_{q_{r-1}} - halfmod]_{q_i}$
15:         $tmp = [tmp \times q_r^{-1}]_{q_i}$
16:         $ct_i[k] = [c_i[k] + tmp]_{q_i}$
```

the first step of the `fast_floor` function (see Algorithm 22 for the pseudo-code): the results of the `multip_t` kernel function in $\mathcal{Q}$ and $\mathcal{B}_{sk}$ bases are converted to the $\mathcal{B}_{sk}$ base. The eighth CUDA kernel function, named `second_fast_floor`, eliminates errors with the flooring method instead of the rounding method. After the `fast_floor` kernel function, the fast base conversion function is performed in the ninth and tenth CUDA kernel functions. The ninth kernel function performs the conversion from the extension base $\mathcal{B}_{sk}$ to the RNS base $\mathcal{Q}$. Finally, the tenth kernel function, `second_fastbdconv_sk` is used to eliminate the rounding errors.

4.4.6 Relineariazation

The BFV relinearization operation uses `switchkey` operation as explained in Figure 2.2, a pseudo-code of which is given in Algorithm 23 as it is implemented in the Microsoft SEAL library. The relinerization operation usually follows a homomorphic multiplication of ciphertexts, which are given in the polynomial domain in BFV. The third component of the ciphertext, $c[2]$, which are to be multiplied with evaluations keys, are first converted

Algorithm 24 Apply Galois

```
Input: $galois\_elt, c_i^j[k] \in \mathbf{R}_{q_i,n}$, where $0 \le i < (r-1)$ $0 \le j < n$ $k = 0,1$
Output: $c_i^j[k] \in \mathbf{R}_{q_i}$
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:             $r\_val = c_i^j[k]$
7:             if $(index\_raw \gg log_2(n)) \; \& \; 1$ then
8:                 $non\_zero = int(r\_val \neq 0)$
9:                 $r\_val = (q_i - r\_val)\&(-non\_zero)$
10:            $c_i^j[k] = r\_val$
```

to the NTT domain using our NTT implementation (see line 5 of Algorithm 23). Then, the multiplication with evaluation keys are performed in the lines 2-6 of Algorithm 23, which are implemented in a single kernel function.

Due to the fact that no polynomial multiplication is needed after line 9, the results are converted back to the polynomial domain (see lines 7-9 of Algorithm 23). The lines 10 and 12 are used to implement the arithmetic with the half modulus as previously described in Figure 2.2. Lastly, the operations between lines 10 to 16 in Algorithm 23 are implemented with a single kernel.

4.4.7 Rotation

The BFV rotation operation uses the so-called `apply_galois` method, whose pseudo-code is given in Algorithm 24 and the `switchkey` operation in Algorithm 23. Before the rotation operation, `galois_elt` algorithm for a given shift amount is executed in CPU using Algorithm 25 and the result `galois_elt` is sent to GPU. Then, a single kernel is used to implement `apply_galois` algorithm. Finally, another kernel function is used to implement `switchkey` operation as explained in part 2.5.3.7.

Algorithm 25 Galois Elt

```
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:     $gen = 3$
11:     $galois\_elt = 1$
12:     for $i$ from 0 by 1 to $steps$ do
13:         $galois\_elt = galois\_elt \times gen$
14:         $galois\_elt = galois\_elt \,\&\, (m32 - 1)$
15:     return $galois\_elt$
```

5. RESULT AND COMPARISON

In this section, we present the results of three different GPU implementations of NTT on three different GPU machines, whose architectural details are given in Table 5.1. Only NTT results and comparisons are included in this section since INTT is just an inverse of NTT, and its results are almost identical to NTT results. Addititionaly, we present the results of the BFV implementation on GPU and their comparison with those in the state-of-the-art works in the literature. Also, we present the implementation of gradient boosting framework (XGBoost) (ŞS Mağara et al., 2021) using our GPU library to show its performance in practical real-world applications.

In our NTT implementations, three different types of modular reduction, which are optimized in assembly, are applied. Two of them are Barret Shivdikar et al. (2022) and Plantard Plantard (2021) reductions, which can work with any NTT friendly prime, while the third is the Goldilocks reduction Hamburg (2015), if the 64-bit goldilocks prime (i.e., $q = 2^{64} - 2^{32} + 1$) is employed. The goldilocks reduction method, which consists of fewer

Table 5.1 Hardware features of the Testbed

	GPU		
Feature	GPU-A	GPU-B	GPU-C
Architecture	RTX 3060Ti	A100 80 GB	RTX 4090
Threads	4864	6912	16384
Boost Freq.	1665 MHz	1410 MHz	2520 MHz
Memory Size	8 GB	80 GB	24 GB
Memory Type	GDDR6	HBM2e	GDDR6X
Memory Bus	256 bit	5120 bit	384 bit
Bandwidth	448 GB/s	1935 GB/s	1008 GB/s

number of instructions, is used for more efficient modular arithmetic. Here, we also show the effects of a fast goldilocks modular reduction method on the overall performance of an NTT operation. Lastly, we include performance comparison of the proposed algorithms with those in the literature.

All the measurements in the subsequent tables are kernel timings only; namely, they do not include the transfer time from the CPU to the GPU. While taking the measurements, the **cudaEventRecord** function, which is one of CUDA's native functions, is used. It is a reliable function for measuring time as it measures all GPU activity from the time kernels are invoked to the terminations of the kernels. To ensure stability, all scenarios are repeated at least 50 times (as many as 1000 times for most cases) depending on the ring dimension, and the average values of the results are presented. In addition, the timings of all implementations are taken on devices with the same CUDA driver and the same operating system[1].

5.1 NTT Implementations Results and Comparison with Related Works

In this section, we report on our GPU implementations of NTT algorithms and compare their results with those running on CPU and other GPU implementations in the literature.

5.1.1 Comparison of Three NTT Implementations Between Each Other

In this section, we compare three GPU implementations of NTT, explained in Chapter 4, which are referred to as **MERGE-1**, **MERGE-2**, and **4-STEP**, respectively. As the names indicate the first two algorithms are two different implementations of the Merge algorithm while the third is based on the 4Step algorithm. For comparison we use latency and throughput as metrics. The latency results of a single NTT for all three NTT implementations are given in Table 5.2 with three different GPUs in Table IV, where the better timings are in bold. As can be observed from the table, the timings of the **MERGE-2** and **4-STEP** are generally close to each other and much better than

[1]Ubuntu 20.04 LTS and CUDA version 12.1

Table 5.2 Timings of NTT Algorithms for **Single** Forward NTT on different GPUs in μs (**Latency**)

	MERGE-1	**MERGE-2**	**4-STEP**
logn	GPU-A / GPU-B / GPU-C	GPU-A / GPU-B / GPU-C	GPU-A / GPU-B / GPU-C
12	12.71 / 18.15 / 10.89	8.32 / **12.57** / 7.64	**7.92** / 12.62 / **7.29**
13	15.02 / 20.64 / 12.26	8.43 / 12.94 / 7.74	**8.30** / **12.64** / **7.53**
14	19.18 / 27.11 / 16.02	9.15 / **12.97** / **7.80**	**8.95** / 13.70 / 8.08
15	33.81 / 42.79 / 24.69	10.49 / **13.84** / **8.03**	**10.36** / 14.41 / 8.49
16	- / - / -	14.79 / **15.93** / **8.66**	**14.33** / 16.34 / 9.04
17	- / - / -	24.66 / **21.94** / **11.81**	**23.97** / 22.57 / 11.86
18	- / - / -	41.70 / **28.38** / 15.46	**40.75** / 28.89 / **15.37**
19	- / - / -	**85.25** / 43.38 / 24.54	87.30 / **43.19** / **23.13**
20	- / - / -	**164.02** / **72.12** / 39.40	165.85 / 72.60 / **38.33**
21	- / - / -	321.88 / 142.29 / 70.68	**317.61** / **135.83** / **66.37**
22	- / - / -	647.32 / 272.99 / **135.19**	**617.26** / **252.71** / 142.84
23	- / - / -	1422.32 / 602.16 / **377.91**	**1221.85** / **499.01** / 422.91
24	- / - / -	3448.09 / 1152.67 / 1020.95	**2514.96** / **1016.50** / **969.92**

All time measurements are taken for the 64-bit Goldilocks prime.

MERGE-1. The main reason of this is that **MERGE-1** performs sequentially inside the kernel and its occupancy is relatively lower than the other two implementations. Additionally, **MERGE-1** is not designed to run for the ring sizes higher than 2^{15} due to its register-based design. For relatively small values of n, the **MERGE-2** algorithm tends to perform better than the **4-STEP** algorithm. For very large values of n, however, the **4-STEP** algorithm's performance is generally superior, due to the fact that the former algorithm becomes memory-bound with the increase of n and that better spatial locality of the latter algorithm becomes advantageous. Another observation from the table is that the architectural differences can affect the performance to a certain extent. For instance, while the **MERGE-2** is slower than the **4-STEP** on RTX 4090 for $n = 2^{20}$, the opposite is true for the other two GPU devices.

In HE implementations, as many independent NTT operations are executed concurrently, we also need to measure the throughput. GPU has many threads to execute many NTT operations in parallel. In fact, running a single NTT does not reveal the real potential of a GPU device as its resources are not fully utilized, especially for smaller values of n. Table 5.3 presents the timing results for different number of concurrently running NTT operations (i.e., 4, 16, 32,64, 128). It is clear from the table that GPU devices can compute many NTT operations in parallel without increasing the execution time significantly. However, when the GPU resources are fully utilized, the execution of NTT operations is serialized. For instance, on RTX 4090, when $n = 2^{14}$ while four NTT operations take 8.04 μs, 16 of them take only slightly more time, 11.48 μs when **MERGE-2** is used. The

Table 5.3 Timings of GPU implementations of Batch Forward NTT in μs

	NTT	**GPU-A**	**GPU-B**	**GPU-C**
n	**count**	• / † / ‡	• / † / ‡	• / † / ‡
	4	12.91 / 8.67 / 8.67	17.85 / **12.24** / 12.98	10.71 / **7.59** / 7.72
	16	15.56 / **12.27** / 12.63	18.39 / **13.91** / 14.20	10.99 / **7.86** / 8.14
2^{12}	32	23.08 / **17.09** / 17.41	18.72 / **15.65** / 16.98	11.63 / **8.54** / 9.25
	64	41.17 / **26.90** / 28.69	27.56 / **19.63** / 20.91	13.98 / **10.67** / 10.68
	128	80.74 / **53.04** / 54.71	37.09 / **27.85** / 28.97	22.21 / 15.62 / **15.25**
	4	15.28 / **9.45** / 9.82	20.78 / **13.25** / 13.66	12.31 / **7.65** / 8.04
	16	24.08 / **17.52** / 18.53	21.19 / **16.20** / 17.14	12.72 / **8.80** / 9.51
2^{13}	32	46.66 / **28.00** / 29.24	28.67 / **20.22** / 21.73	15.45 / **11.08** / 11.36
	64	92.75 / **54.41** / 57.89	40.65 / **29.10** / 30.48	25.08 / 16.05 / **15.92**
	128	169.42 / **101.53** / 104.55	60.93 / **46.07** / 48.11	44.39 / 24.89 / **24.72**
	4	22.52 / **13.24** / 13.70	27.53 / **14.94** / 15.10	16.22 / **8.04** / 8.55
	16	53.45 / **29.80** / 31.90	34.71 / **21.10** / 22.34	17.59 / **11.48** / 11.70
2^{14}	32	106.86 / **56.19** / 60.98	43.19 / **30.63** / 31.76	26.85 / 16.82 / **16.80**
	64	209.63 / **103.47** / 110.90	70.60 / **48.90** / 51.29	47.67 / 26.36 / 26.32
	128	400.73 / **197.25** / 209.85	121.22 / **88.52** / 90.58	86.84 / 47.11 / **45.16**
	4	41.88 / **19.41** / 20.46	43.48 / **17.26** / 18.59	25.57 / **9.19** / 9.86
	16	134.87 / **57.71** / 62.48	57.73 / **32.26** / 33.21	32.65 / 17.55 / **17.31**
2^{15}	32	266.47 / **106.48** / 112.25	74.95 / **51.85** / 53.47	53.20 / 27.81 / **27.41**
	64	539.45 / **200.83** / 211.89	143.50 / **93.86** / 94.58	97.02 / 49.83 / **47.72**
	128	1057.04 / **389.93** / 407.62	263.55 / 192.83 / **185.17**	186.45 / 90.66 / **85.48**
	4	- / **33.64** / 34.28	- / **23.26** / 24.30	- / 12.46 / 12.49
	16	- / 113.16 / **112.93**	- / **55.92** / 56.30	- / 30.76 / **28.56**
2^{16}	32	- / **219.09** / 211.73	- / 100.77 / **99.30**	- / 53.85 / **49.66**
	64	- / 422.00 / **406.88**	- / 196.43 / **188.83**	- / 97.16 / **91.00**
	128	- / 819.44 / **803.18**	- / 366.00 / **354.03**	- / **184.86** / 192.16

•: MERGE-1 NTT

†: MERGE-2 NTT

‡: 4STEP NTT

All time measurements are taken for the 64-bit Goldilocks prime.

increase in execution times is more salient for RTX 3060 Ti and A 100, which support fewer number of threads. Nevertheless, we observe the same effect for RTX 4090 also for higher ring dimensions, which require more resources.

When performances of the three GPUs are compared, all algorithms generally run faster on RTX 3060Ti than A 100 at low ring dimensions when the number of NTT operations is low. This is due to the fact that the clock frequency is more dominant than the bandwidth; in other words, the operations are compute-bounded for those input sizes. However, when the ring dimension and the number of NTT operations are high, the execution becomes memory-bound. Then, A 100 performs much better than RTX 3060 Ti with its superior memory bandwidth. RTX 4090 has a relatively high memory bandwidth, albeit only half as much as A100. Nonetheless, thanks to the new ADA architecture, RTX 4090 has many more threads and operates at a very high frequency. Thus, RTX 4090 is the best of the three GPU devices for all instances (see Table 5.2 and Table 5.3). However, as the operations become memory-bound for higher values of n, the difference in the timing results obtained on RTX 4090 and A 100 becomes mush less visible. For example, for $n = 2^{22}$ **MERGE-2** takes 272.99 μs and 135.19 μs on A100 and on RTX 4090, respectively, which translates into a speedup of 2.02× (see Table 5.2). However, the speedup values dramatically drop to 1.59× for $n = 2^{23}$ and 1.13× for $n = 2^{24}$.

In our all experiments, we report the timing results for the 64-bit goldilocks primes, for which the modular reduction is very fast. On the other hand, when residue number system (RNS) is used to work with much larger modulus, as in the case of Homomorphic Encryprion (HE) schemes, using random primes with a fast reduction algorithm can be advantageous as it decreases the number of base moduli in the RNS Dai et al. (2018). For HE applications, special goldilocks prime $Q = 2^{64} + 2^{32} + 1$ is used as *carrier* for smaller primes, q_i, employed in RNS arithmetic. When carrier prime is used for NTT, there is an upper bound for RNS prime bases, imposed by the inequality $q_i^2 n < Q$. For instance, each base can be at most 25-bit and 24-bit, respectively, for the ring sizes $n = 2^{14}$ and $n = 2^{15}$.

Thus, we also implemented the **MERGE-2** algorithm using 60-bit NTT-friendly random primes with Barret and Plantard reduction to see the performance penalty due to their usage. For the ring dimension range of $[2^{12}, 2^{24}]$, using 64-bit goldilocks primes offers maximum 25.5% speedup over the NTT implementation with random primes for A100 (the figures are very close for the other GPU devices). Consequently, we can conclude that using NTT-friendly random primes can be more advantageous for HE applications using RNS arithmetic.

5.1.2 Comparison of NTT Results with Related Works in the Literature

The literature contains several works that accelerate NTT (and related operations) using old as well as the contemporary GPU devices. They either use a randomly chosen NTT-friendly primes or specials primes (e.g., Goldilocks primes), which offer faster modular reduction. We compare our timing results of single NTT with those in the literature for the parameter sets suitable for the HE algorithms in Table 5.4. The table lists the

Table 5.4 Timings (μs) of GPU implementation of Our Single Forward NTT and their comparison with the works in literature

			$\log_2 n$				
Work	Device	$\log_2 q$	12	13	14	15	16
Kim et al. (2020)	Titan V	60	-	-	44.1	84.2	-
Zheng (2020)	RTX 2080 Ti	64*	-	-	-	83.3	-
Goey et al. (2021)	GTX 1070	64*	-	-	57.8	-	-
Dai and Sunar (2015)	GTX 1070	64*	-	-	66.8	-	-
Özerk et al. (2022)	V 100	55	-	-	29	39	-
Shivdikar et al. (2022)	A 100	62	-	-	13.3	-	16.5
Shivdikar et al. (2022)	V 100	62	-	-	11.5	-	16.4
T.W.	RTX 3060 Ti	64*	8.32	8.43	9.15	10.49	14.79
T.W.	A 100	64*	12.57	12.94	12.97	13.84	15.93
T.W.	RTX 4090	64*	7.64	7.74	7.78	8.03	8.66
T.W.	RTX 3060 Ti	60	8.45	8.67	9.38	11.45	15.00
T.W.	A 100	60	12.72	13.22	13.27	14.56	16.15
T.W.	RTX 4090	60	7.60	7.65	7.77	8.20	8.86

T.W.: This Work (**MERGE-2**)

*: uses constant prime $q = 2^{64} - 2^{32} + 1$

results for both 60-bit random primes implemented with the Plantard Reduction and the 64-bit Goldilocks prime implemented with the Goldilocks Reduction. Considering the architectural differences, our **MERGE-2** algorithm compares favorably with all works in the literature. More results including those taken with three different modular reduction methods, Goldilocks, Plantard, and Barret Reductions, respectively, are available Table 7.1 in Appendix.

We also compare our results with the GPU implementation (known as **SPPARK**[2]), which is intended for ZK-SNARK protocols working with much higher ring dimensions and a larger 253-bit random prime. For a fair comparison, we adopt the SPARKK implementation of the Montgomery multiplication algorithm in our CUDA code. Only the kernels of SPPARK for NTT are included in the timings excluding the kernels for LDE or bit reverse order operations. In addition, the execution times of same number of NTT operations are measured for both SPPARK and our implementations and their averages are calculated. The average timings of a single NTT operation for ring dimensions be-

[2]https://github.com/supranational/sppark/tree/main

Table 5.5 Timings (μs) of GPU implementation of Our Single Forward NTT for higher values of n with $\log q = 253$ (q is BLS12-377 prime)

	GPU-B	**GPU-C**
logn	**T.W. / SPPARK**	**T.W. / SPPARK**
19	**230.68** / 246.66	**111.60** / 132.96
20	**416.47** / 515.72	**206.88** / 259.88
21	**854.36** / 937.38	**424.25** / 504.40
22	**1690.40** / 1912.60	**959.88** / 1075.67
23	**3511.66** / 4284.51	**2027.27** / 2362.28
24	**7294.05** / 8060.20	**4178.86** / 4484.67
25	**15251.5** / 16952.4	**8620.7** / 9131.73
26	**31388.1** / 38815.0	**17728.4** / 20213.7
27	**65097.4** / 74886.3	**36780.5** / 38177.9
28	**137242.0** / 179563.0	**78886.7** / 82987.3

T.W.: This Work (**MERGE-2**)

tween 2^{19} and 2^{28} are reported in Table 5.5[3]. The results in Table 5.5 show that our implementation is 6.9/30.8% and 3.8/25.6% min/max faster than SPPARK on A 100 and RTX 4090, respectively. In addition, our implementation also supports higher ring dimensions than 2^{28} as the SPPARK implementation.

5.2 BFV Library Operation Results and Comparison with Related Works

There are not many prior works in the literature that present GPU implementations of homomorphic operations of the BFV scheme, and the existing ones do not give performance results for all homomorphic operations let alone the homomorphic application results. Therefore, the comparison of our work with other works in the literature cannot be comprehensive. The work in (Dai and Sunar, 2015) provides timing results on GPU for homomorphic applications of an old and completely different homomorphic encryption scheme, LTV (López-Alt et al., 2017), which is not in use today. We compare the results of our GPU implementation of the BFV-scheme operations with the work (Al Badawi

[3]The extended version of the execution times comparison for ring dimensions between 2^{12} and 2^{28} is available in Table 7.2 in Appendix.

et al., 2019b), which represents the state-of-the-art in the literature for GPU implementation of the BFV scheme. The work (Al Badawi et al., 2019b) provides only the timing results of homomorphic multiplication, which includes those of the following relinearization operation.

We first provide our timing results separately in Table 5.6, which includes all major homomorphic arithmetic operations, typically used in many homomorphic applications.

Table 5.6 Comparison results of SEAL BFV Scheme operations and literature with our GPU implementations of BFV scheme operations(μs).

			GPU-A		**GPU-C**		**Tesla V100**	**CPU**	T
Operation	**n**	$\log_2 q$	MERGE-1	MERGE-2	MERGE-1	MERGE-2	Al Badawi et al. (2019b)	SEAL	T_s
Enc.	2^{12}	109	37.15	26.20	29.46	21.35	-	938	43.9×
	2^{13}	218	48.32	36.56	33.93	22.71	-	2299	101.2×
	2^{14}	438	143.33	105.22	49.48	34.40	-	8656	251.6×
	2^{15}	881	640.8	383.87	119.07	**80.49**	-	**35212**	**437.5**×
Dec.	2^{12}	109	34.65	22.62	27.42	19.41	-	289	14.9×
	2^{13}	218	52.09	27.41	31.91	20.45	-	859	42.0×
	2^{14}	438	87.51	58.62	44.86	27.31	-	3992	146.2×
	2^{15}	881	332.7	206.9	87.86	**53.63**	-	**16893**	**315.0**×
Add.	2^{12}	109	4	4	3.42	3.42	-	14	4.1×
	2^{13}	218	5.1	5.1	3.66	3.66	-	58	15.8×
	2^{14}	438	12.3	12.3	4.32	4.32	-	233	53.9×
	2^{15}	881	44	44	**7.14**	**7.14**	-	**778**	**109.0**×
Mult.	2^{12}	109	64.59	50.84	36.97	29.37	-	3212	109.4×
	2^{13}	218	157.24	123.45	51.53	44.16	-	11883	269.1×
	2^{14}	438	737.86	531.49	167.32	**134.83**	-	**48757**	**361.6**×
	2^{15}	881	3982.3	2838.6	791.56	637.27	-	205295	322.1×
Relin.	2^{12}	109	36.03	26.63	28.98	20.89	-	625	29.9×
	2^{13}	218	59.89	49.18	33.56	23.42	-	3100	132.36×
	2^{14}	438	376.3	255.83	83.12	**56.96**	-	**18295**	**321.2**×
	2^{15}	881	2960.0	1573.7	656.27	460.25	-	111736	242.8×
Rot.	2^{12}	109	36.53	27.82	29.15	20.91	-	642	30.7×
	2^{13}	218	63.0	48.78	34.22	24.15	-	3157	130.7×
	2^{14}	438	386.17	268.03	93.57	**67.64**	-	**18338**	**271.1**×
	2^{15}	881	3245.2	1846.16	705.58	519.90	-	113437	218.2×
Mult. + Relin.	2^{12}	60	78.6	59.4	59.1	**42.3**	**859**	-	**20.3**×
	2^{13}	120	157.9	127.5	75.1	56.0	1012	-	18.0×
	2^{14}	360	789.5	556.1	189.5	141.6	2010	-	14.2×
	2^{15}	600	3671.5	2289.9	772.9	563.0	4826	-	8.5×

Enc.:BFV Encryption. **Dec.**:BFV Decryption. **Add.**:BFV Addition.
Mult.:BFV Multiplication. **Relin.**:BFV Relinearization. **Rot.**:BFV Rotation.
T_s: speed up

Our GPU implementation shows significant improvements over the CPU implementation of the SEAL library running on a powerful CPU. The Microsoft SEAL library, which is open-source and written in C++ programming language, provides highly optimized NTT implementation and therefore, represents state-of-the-art implementations of both the NTT and the BFV algorithms. As shown in Table 5.6 the proposed GPU library provides up to 361.6× faster BFV multiplication operation, 321.2× faster BFV relinearization operation, 271.1× faster BFV rotation operation, 53.9× faster BFV addition operation, when $n = 2^{14}$ and $\log_2 q = 438$ with respect to the SEAL library, which is running on AMD Ryzen7 3800X. The total number of threads available in our GPU devices accounts for the optimum results obtained at $n = 2^{14}$.

Care must be taken when evaluating CPU and GPU timing results as a fair comparison of GPU and CPU implementations can be very difficult due to differences in their architectures and applicable optimization techniques. As pointed out in Lee et al. (2010) acceleration figures can overestimate the performance of a GPU device when compared to a CPU. The Microsoft SEAL library is probably the fastest CPU implementation of the BFV scheme in the open source and we are unable to identify any further optimization or parallelization technique that outperforms its NTT or homomorphic operation implementation. On the other hand, we find that CPU parallelization when deployed in the right way can be extremely useful in acceleration at application level as shown in Section 5.3. Indeed, the speedup of GPU implementation turns out to be more moderate when all computing power of CPU is utilized for XGBoost classification application in Section 5.3; a result which is in line with those in Lee et al. (2010).

Then, we compare our results with the work in Al Badawi et al. (2019b) only for homomorphic multiplication including the following relinearization operation as it is the only one reported. The GPU used in Al Badawi et al. (2019b) has 5120 cores and 16 GB of memory operating at the clock frequency of 1.380 GHz, which is comparable to RTX 3060Ti used in our measurements. The execution times are also measured for the same ciphertext modulus sizes and the ring dimensions used in the work Al Badawi et al. (2019b) for a fair comparison. As observable from Table 5.6, our GPU implementation outperforms that in Al Badawi et al. (2019b) for all cases. For instance, our multiplication including relinearization implementation results are 6.31× faster for $n = 2^{12}$, 5.95× faster for ring size $n = 2^{13}$, 3.04× faster for ring size $n = 2^{14}$, and 1.67× faster for ring size $n = 2^{15}$ than the work Al Badawi et al. (2019b), respectively.

5.3 Results of Privacy-Preserving Inference for Genome Data using BFV

GPU Library

Mağara et al. (ŞS Mağara et al., 2021) introduce a privacy-preserving gradient boosting inference framework (XGBoost) algorithm using homomorphic encryption for the classification of the encrypted genome data of different tumor types. We implemented their framework using our GPU library of the BFV scheme. XGBoost is a learning algorithm, which uses Extreme Gradient Boosting ensembles. The model consists of classification trees that are constructed by training data. Trees of the ensemble evaluate the test data that are classified into one of the leaves. Lastly, a final prediction score is formed by summing up the numerical scores obtained from each tree. To decrease the complexity of the model and the depth of the corresponding circuit to be homomorphically evaluated, shallow trees are selected.

As explained in (ŞS Mağara et al., 2021), test data is encrypted, and the XGBoost trees are homomorphically evaluated for a total of 258 test data points. The total number of homomorphic multiplications, rotations, subtractions, plain multiplications, addition, and relinearization operations are 1290, 1806, 1806, 1290, 3354, and 2322, respectively.

We run the inference framework both on GPU and CPU, and all known possible optimization and parallelization techniques to us are employed for the CPU implementation. As shown in Table 5.7, our GPU library accelerates the classification operation at least 237.88 times with respect to the results obtained from AMD Ryzen7 3800X CPU with a single thread while the speedup is 36.08 when multi-threaded version of the CPU implementation is used.

Table 5.7 Implementation of gradient boosting framework(XGBoost) results

		SEAL		**T.W**		
n	$\log_2 q$	S.T.	M.T.	GPU-C	T	S
2^{13}	218	25.62 s	3.4 s	0.142 s	180.42×	23.94×
2^{14}	438	127.028 s	19.27 s	0.534 s	237.88×	36.08×

S.T.:Single Thread **M.T.**:Multi Thread.(16 threads) **T.W.**:This Work. T:The ratio of single-thread results over this work. S: The ratio of multi-thread results over this work.

6. CONCLUSION AND FUTURE WORK

In this thesis, we presented three different and highly parallelized and optimized GPU algorithms and their implementations of NTT based on Merge and 4-Step algorithms and homomorphic operations of the BFV scheme. Although our implementations can be independently used, they are also integrated with the Microsoft SEAL library and their functions can be called from any application code using SEAL. Therefore, our implementations are true accelerators for homomorphic encryption applications. The main objective of the proposed algorithms is to optimize the memory access latency by optimizing the number of CUDA kernel invocations, and by determining the optimum sizes and shapes of the thread blocks to take advantage of coalesced access to the global memory. We provided two types of timing results for both algorithms on three powerful GPUs: i) execution time of a single NTT operation (*latency*) and ii) the execution times of many concurrently running NTT operations (throughput). Throughput is a more important metric for the performance of NTT in homomorphic encryption applications as they execute a multitude of them in parallel. Nevertheless, all other works on the subject report only latency results for a very limited input parameters such as ring dimension and coefficient modulus size, which may be misleading to asses the performance of an NTT algorithm. Therefore, ours is a unique work in the literature by providing a very extensive timing results on a wide range of input parameter set and three different GPU devices. Our latency results suggest that the two of the three algorithms (the iterative algorithm incorporating the latest optimizations; i.e., MERGE-2, and the one based on the 4-Step algorithm) perform comparably with very close timing results. The throughput results, on the other hand, indicate that the the iterative algorithm performs markedly better than the 4-step algorithm. While the later algorithm is usually preferred in the literature for its better spatial locality, our results suggest that the former can be an alternative for homomorphic encryption applications. Also, when compared with the best known implementation for very high degree polynomials in the range of $[2^{12}, 2^{28}]$, our implementation is superior.

Then, all homomorphic operations of the BFV scheme are also implemented on GPU and compared against the SEAL library running on a CPU. When compared with CPU implementation for the ring size of 2^{14} and the modulus bit size of 438, the GPU library running on RTX4090 achieves speedups of 361.6, 321.2, and 271.1 for homomorphic multiplication, relinearization, and homomorphic rotation, respectively. We also compared our homomorphic multiplication followed by a relinearization operation with that of the state-of-the-art GPU implementation in the literature, and found that ours is up to 6.31 times faster than the latter.

We also showed that the proposed GPU library is profitably used in the homomorphic processing of real data such as the classification of encrypted genome data for tumor types and reported at least a speedup of 23.94 in comparison with a powerful CPU running 16 threads.

The reported performance gains establish that GPU implementations of homomorphic encryption prove to be useful to help privacy-preserving data processing applications become more practicable.

As future work, we are planning to develop a method for the twiddle factor generation in the iterative NTT algorithm as it suffers from the increased number of them processed in a large portion of iterations. Other optimization techniques such as higher radix butterfly circuits can be incorporated to further accelerate both algorithms. We also envision integrating our GPU accelerator into other HE libraries and using it to accelerate other more challenging operations such as bootstrapping and scheme switching. We can achieve these goals by joining recent open-source efforts in the development of HE software libraries such as OpenFHE (Badawi et al., 2022).

BIBLIOGRAPHY

Agarwal, R. C. and Burrus, C. S. (1974). Fast convolution using fermat number transforms with applications to digital filtering. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 22:87–97.

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

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

Antao, S., Bajard, J., and Sousa, L. (2012). RNS-based elliptic curve point multiplication for massive parallel architectures. *Comput. J.*, 55(5):629–647.

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.

Bailey, D. H. (1989). Ffts in external or hierarchical memory. pages 234–242.

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.

Bajard, J., Eynard, J., Merkiche, N., and Plantard, T. (2015). RNS arithmetic approach in lattice-based cryptography: Accelerating the "rounding-off" core procedure. In *22nd IEEE Symposium on Computer Arithmetic, ARITH 2015, Lyon, France, June 22-24, 2015*, pages 113–120. IEEE.

Bajard, J.-C. and Plantard, T. (2004). Rns bases and conversions. In *SPIE Optics + Photonics.*

Ben-Sasson, E., Chiesa, A., Genkin, D., Tromer, E., and Virza, M. (2013). Snarks for C: verifying program executions succinctly and in zero knowledge. *IACR Cryptol. ePrint Arch.*, page 507.

Bitansky, N., Canetti, R., Chiesa, A., and Tromer, E. (2012). From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. page 326–349.

Bos, J. W. and Montgomery, P. L. (2017). Montgomery arithmetic from a software perspective. Cryptology ePrint Archive, Paper 2017/1057. https://eprint.iacr.org/2017/1057.

Boura, C., Gama, N., Georgieva, M., and Jetchev, D. (2020). CHIMERA: combining ring-lwe-based fully homomorphic encryption schemes. *J. Math. Cryptol.*, 14(1):316–338.

Brakerski, Z. (2012). Fully homomorphic encryption without modulus switching from classical gapsvp. In Safavi-Naini, R. and Canetti, R., editors, *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, volume 7417 of *Lecture Notes in Computer Science*, pages 868–886. Springer.

Brakerski, Z., Gentry, C., and Vaikuntanathan, V. (2011). Fully homomorphic encryption without bootstrapping. Cryptology ePrint Archive, Paper 2011/277. https://eprint.iacr.org/2011/277.

Brakerski, Z., Gentry, C., and Vaikuntanathan, V. (2014). (leveled) fully homomorphic encryption without bootstrapping. *ACM Trans. Comput. Theory*, 6(3).

Brakerski, Z. and Vaikuntanathan, V. (2014). Efficient fully homomorphic encryption from (standard) LWE. *SIAM Journal on Computing*, 43(2):831–871.

Chen, H., Dai, W., Kim, M., and Song, Y. (2019). Efficient multi-key homomorphic encryption with packed ciphertexts with application to oblivious neural network inference. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 395–412, New York, NY, USA. Association for Computing Machinery.

Cheon, J. H., Kim, A., Kim, M., and Song, Y. (2016). Homomorphic encryption for arithmetic of approximate numbers. Cryptology ePrint Archive, Paper 2016/421. https://eprint.iacr.org/2016/421.

Chillotti, I., Gama, N., Georgieva, M., and Izabachène, M. (August 2016). TFHE: Fast fully homomorphic encryption library. https://tfhe.github.io/tfhe/.

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

ŞS Mağara, Yıldırım, C., Yaman, F., Dilekoğlu, B., TutaŞ, F., Öztürk, E., Kaya, K., TaŞtan, O., and SavaŞ, E. (2021). Ml with he: Privacy preserving machine learning inferences for genome studies. ACM CCS 2021 Privacy Preserving Machine Learning Workshop. In Press.

Dai, W., Doröz, Y., Polyakov, Y., Rohloff, K., Sajjadpour, H., Savas, E., and Sunar, B. (2018). Implementation and evaluation of a lattice-based key-policy ABE scheme. *IEEE Trans. Inf. Forensics Secur.*, 13(5):1169–1184.

Dai, W. and Sunar, B. (2015). cuhe: A homomorphic encryption accelerator library. pages 169–186.

Derya, K., Mert, A. C., Öztürk, E., and Savas, E. (2021). Coha-ntt: A configurable hardware accelerator for ntt-based polynomial multiplication. *IACR Cryptol. ePrint Arch.*, page 1527.

Doröz, Y., Öztürk, E., Savaş, E., and Sunar, B. (2015). Accelerating ltv based homomorphic encryption in reconfigurable hardware. pages 185–204.

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

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

Fan, J. and Vercauteren, F. (2012c). Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Paper 2012/144. https://eprint.iacr.org/2012/144.

Gamal, T. E. (1985). A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472.

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.

Goey, J.-Z., Lee, W.-K., Goi, B.-M., and Yap, W.-S. (2021). Accelerating number theoretic transform in gpu platform for fully homomorphic encryption. *The Journal of Supercomputing*, 477:1455–1474.

Halevi, S., Polyakov, Y., and Shoup, V. (2019). An improved rns variant of the bfv homomorphic encryption scheme. In *Cryptographers' Track at the RSA Conference*, pages 83–105. Springer.

Halevi, S. and Shoup, V. (2014). Algorithms in helib. pages 554–571.

Hamburg, M. (2015). Ed448-goldilocks, a new elliptic curve. *IACR Cryptol. ePrint Arch.*, page 625.

Hirner, F., Mert, A. C., and Roy, S. S. (2023). Proteus: A tool to generate pipelined number theoretic transform architectures for fhe and zkp applications. Cryptology ePrint Archive, Paper 2023/267. https://eprint.iacr.org/2023/267.

Kim, S., Jung, W., Park, J., and Ahn, J. H. (2020). Accelerating number theoretic transformations for bootstrappable homomorphic encryption on gpus. pages 264–275.

Laine, K. (2017). Simple encrypted arithmetic library 2.3.1. *Microsoft Research, WA, USA.*

Lee, V. W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A. D., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., Singhal, R., and Dubey, P. (2010). Debunking the 100x GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU. *SIGARCH Comput. Archit. News*, 38(3):451–460.

López-Alt, A., Tromer, E., and Vaikuntanathan, V. (2017). Multikey fully homomorphic encryption and applications. *SIAM J. Comput.*, 46(6):1827–1892.

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

Mert, A. C., Karabulut, E., Öztürk, E., Savas, E., and Aysu, A. (2022). An extensive study of flexible design methods for the number theoretic transform. *IEEE Trans. Computers*, 71(11):2829–2843.

Mert, A. C., Öztürk, E., and Savas, E. (2020a). FPGA implementation of a run-time configurable ntt-based polynomial multiplication hardware. *Microprocess. Microsystems*, 78:103219.

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

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

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

Özerk, Ö., Elgezen, C., Mert, A. C., Öztürk, E., and Savas, E. (2022). Efficient number theoretic transform implementation on GPU for homomorphic encryption. *J. Supercomput.*, 78(2):2840–2872.

Paillier, P. (1999). Public-key cryptosystems based on composite degree residuosity classes. In Stern, J., editor, *EUROCRYPT*, volume 1592 of *Lecture Notes in Computer Science*, pages 223–238. Springer.

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

Plantard, T. (2021). Efficient word size modular arithmetic. *IEEE Transactions on Emerging Topics in Computing*, 9(3):1506–1518.

Regev, O. (2005). On lattices, learning with errors, random linear codes, and cryptography. In *Proceedings of the Thirty-seventh Annual ACM Symposium on Theory of Computing*, STOC '05, pages 84–93, New York, NY, USA. ACM.

Riazi, M. S., Laine, K., Pelton, B., and Dai, W. (2020). Heax: An architecture for computing on encrypted data. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1295–1309, New York, NY, USA. Association for Computing Machinery.

Shivdikar, K., Jonatan, G., Mora, E., Livesay, N., Agrawal, R., Joshi, A., Abellan, J. L., Kim, J., and Kaeli, D. (2022). Accelerating polynomial multiplication for homomorphic encryption on GPUs. In *2022 IEEE International Symposium on Secure and Private Execution Environment Design (SEED)*, pages 61–72, Los Alamitos, CA, USA. IEEE Computer Society.

Wang, W., Chen, Z., and Huang, X. (2014). Accelerating leveled fully homomorphic encryption using gpu. pages 2800–2803.

Zhao, H., Ding, D., Wang, F., Hua, P., Wang, N., Wu, Q., and Chai, Z. (2023). Hardware acceleration of number theoretic transform for zk-snark. *Engineering Reports*, page e12639.

Zheng, Z. (2020). Encrypted cloud using GPUs.

Özcan, A. ., Ayduman, C., Türkoğlu, E. R., and Savaş, E. (2023). Homomorphic encryption on GPU. *IEEE Access*, pages 1–1.

7. APPENDIX

Table 7.1 Timings of **MERGE-2** NTT for single Forward NTT on different GPUs with three different modular reduction algorithms in μs (Latency)

	GPU-A	**GPU-B**	**GPU-C**
logn	† / ‡ / ♠	† / ‡ / ♠	† / ‡ / ♠
12	8.32 / 8.45 / 8.93	12.57 / 12.72 / 12.95	7.64 / 7.60 / 7.76
13	8.43 / 8.67 / 9.31	12.94 / 13.22 / 13.42	7.74 / 7.65 / 7.80
14	9.15 / 9.38 / 9.71	12.97 / 13.27 / 13.46	7.80 / 7.77 / 8.06
15	10.49 / 11.45 / 11.88	13.84 / 14.56 / 14.69	8.03 / 8.20 / 8.23
16	14.79 / 15.00 / 17.67	15.93 / 16.15 / 17.37	8.66 / 8.86 / 8.96
17	24.66 / 28.24 / 28.45	21.94 / 22.45 / 24.09	11.81 / 12.94 / 12.52
18	41.70 / 45.26 / 49.90	28.38 / 28.46 / 32.36	15.46 / 16.33 / 17.44
19	85.25 / 94.46 / 97.88	43.38 / 41.67 / 51.74	24.54 / 24.18 / 27.97
20	164.02 / 179.82 / 188.30	72.12 / 68.87 / 89.69	39.40 / 38.36 / 47.73
21	321.88 / 351.73 / 373.72	142.29 / 135.10 / 182.17	70.68 / 66.64 / 87.19
22	647.32 / 735.41 / 759.05	272.99 / 262.17 / 342.66	135.19 / 129.97 / 168.43
23	1422.32 / 1596.05 / 1607.2	602.16 / 580.07 / 727.43	377.91 / 465.20 / 409.42
24	3448.09 / 3882.28 / 3596.6	1152.67 / 1122.70 / 1445.5	1020.95 / 1173.72 / 1028.55

†: Goldilock Reduction used. (64bit)
‡: Plantard Reduction used. (60bit)
♠: Barret Reduction used. (60bit)

Table 7.2 Timings (μs) of Our Single Forward NTT(**MERGE-2**) And Comparison with Sppark's Implementation with $\log q = 253$ (q is BLS12-377 prime)

	GPU-B	**GPU-C**
logn	**T.W. / SPPARK**	**T.W. / SPPARK**
12	26.79 / **23.33**	15.38 / **11.71**
13	28.88 / **28.08**	16.25 / **15.84**
14	29.97 / **28.48**	**17.04** / 18.73
15	**31.72** / 37.73	**18.50** / 20.35
16	48.27 / **47.14**	**21.64** / 27.58
17	**77.10** / 77.61	**39.44** / 42.04
18	120.06 / **116.45**	**64.04** / 65.81
19	**230.68** / 246.66	**111.60** / 132.96
20	**416.47** / 515.72	**206.88** / 259.88
21	**854.36** / 937.38	**424.25** / 504.40
22	**1690.40** / 1912.60	**959.88** / 1075.67
23	**3511.66** / 4284.51	**2027.27** / 2362.28
24	**7294.05** / 8060.20	**4178.86** / 4484.67
25	**15251.5** / 16952.4	**8620.7** / 9131.73
26	**31388.1** / 38815.0	**17728.4** / 20213.7
27	**65097.4** / 74886.3	**36780.5** / 38177.9
28	**137242.0** / 179563.0	**78886.7** / 82987.3

T.W.: This Work (**MERGE NTT**)