Homomorphic Encryption on GPU

Ali Şah Özcan[1], Can Ayduman[1], Enes Recep Türkoğlu[1], Erkay Savaş
Faculty of Engineering and Natural Sciences, Sabanci University, Istanbul, Turkey
{alisah, canayduman, eturkoglu, erkays}@sabanciuniv.edu

***Abstract*—Homomorphic encryption (HE) is a cryptosystem that allows 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.**

This paper presents an optimized and highly parallelized GPU library to accelerate the BFV scheme. This library includes state-of-the-art implementations of Number Theoretic Transform (NTT) and inverse NTT that minimize the GPU kernel function calls. It makes an efficient use of the GPU memory hierarchy and computes 128 NTT operations for ring dimension of 2^{14} only in 176.1 μs on RTX 3060Ti GPU. To the best of our knowlede, this is the fastest implementation in the literature. 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 63.4 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 the BFV operations. Finally, we implement a privacy-preserving application that classifies encrpyted genome data for tumor types and achieve speedups of 42.98 and 5.7 over a 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.

***Index Terms*—Lattice Based Cryptography, Homomorphic Encryption, Number Theoretic Transform (NTT), GPU, Parallel Processing, Secure Computation.**

I. 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 [1], 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) [2] 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 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 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 [1], 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 a FHE scheme was realized by Gentry and Halevi, as explained in [3]. Then, several FHE realizations were introduced such as those in [4] and [5]. One of the most promising approaches is the Brakerski-Fan-Vercauteren scheme [6], and there are several practical implementations of this and other similar schemes such as those provided by well-known software libraries SEAL [7], PALISADE [8], and HELib [9].

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 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 [10] [11] [12], to push the boundaries of FHE performance. A recent announced software library [13] provides support for hardware acceleration integration to software implementations of HE schemes using a standard Hardware Abstraction Layer (HAL).

In this paper, we present algorithms and implementation techniques to accelerate the BFV scheme of the SEAL library via NVIDIA GPUs. Our implementation developed in compute Unified Device Architecture (CUDA) program model [14] accelerates all homomorphic operations in the BFV scheme utilizing various parallelizaton 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. Our implementation achieves a very high level of parallelization on GPU, targeting the compute-intensive nature of FHE operations.

It is established that multiplication in polynomial rings (which requires the multiplication of high degree polynomials and their division by high-degree cyclotomic polynomials) is the most time and resource critical operation in all FHE and SHE implementations.

[1]Co-first authors.

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 it in different ways. GPU architectures support many concurrent threads, which can be employed to perform 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 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 new NTT implementation improvements for polynomial multiplication adapted to GPU architecture and proves to be the fastest in comparison to those reported in the literature. As the main improvement is for the polynomial multiplication, our implementation can be used to accelerate existing implementations of other SHE and FHE schemes such as CKKS and BGV and other more involved homomorphic operations such as bootstrapping and scheme switching [15]. Although the latency of a single NTT operation of our implementation is also superior to those of other implementations in the literature, our NTT implementation is especially optimized for concurrent execution of many NTT operations, which is a typical use case scenario of all homomorphic operations and applications.

The rest of the paper is organized as follows. In Section II, we briefly explain the notation that we use and the mathematical background of NTT, Barrett reduction, Residue Number System (RNS), and homomorphic operations of the SEAL BFV scheme. In Section III, we present the essential working principle of the GPU architecture. In Section IV, we explain our GPU implementation and the algorithms that we used. In Section V, we discuss our implementation results and compare the results with state-of-the-art.

II. Preliminaries

This section presents the notation used throughout the paper and explains the Barrett Reduction, Residue Number System, Number Theoretic Transform, and FHE operations of the SEAL BFV scheme.

A. 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.

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 $\{0, 1, \ldots, q-1\}$, 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 paper 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 III), 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 with 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. $a \leftarrow \mathbb{Z}_q$ stands for the uniform sampling of a from $\mathbb{Z}_q$. 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.

B. Barrett Reduction

In the RNS variant of homomorphic cryptographic schemes such as [16], there is a multitude of modular multiplication operations that dominate the execution times of all homomorphic operations.

The Barrett reduction [17] and the Montgomery reduction [18] algorithms are two popular algorithms that perform the modular reduction operation efficiently. Since the Montgomery reduction needs the extra step for transformation of integers to the Montgomery domain, the Barrett reduction algorithm is selected here for its simplicity.

The Barret reduction is described in Algorithm 1. Here, μ is the precomputed value, $\lfloor \frac{2^{2k}}{q} \rfloor$, where q is the modulus and k is the bit length of the modulus. The Barrett reduction algorithm includes multiplication, shift, and subtraction operations instead of an expensive division operation, which is needed in the computation of $C \bmod q$ by conventional modular multiplication algorithms.

C. Residue Number System (RNS)

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_is forms 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

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

where $M_i = \frac{M}{m_i}$. The RNS is preferred in cryptographic applications as it allows concurrent arithmetic with a set

Algorithm 1 Barrett Reduction

```
Input: $C = a \times b$, where $a, b < q$; $k = \lceil log_2(q) \rceil$; $\mu = \lfloor \frac{2^{2k}}{q} \rfloor$
Output: $C_{out}$ ($C \mod q$)
1: $r \leftarrow C \gg (k-2)$
2: $r \leftarrow r \cdot \mu$
3: $r \leftarrow r \gg (k+2)$
4: $r \leftarrow q \cdot r$
5: $C_{out} \leftarrow (C - r)$
6: if $C_{out} >= 2q$ then   $C_{out} \leftarrow C_{out} - 2q$
7: else if $C_{out} >= q$ then   $C_{out} \leftarrow C_{out} - q$
8: else  $C_{out} \leftarrow C_{out}$
9: end if
```

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 [19]. It is also showed [20], that RNS proves to be useful in accelerating the R-LWE based lattice-base somewhat homomorphic encryption schemes [21], [22]. Furthermore, RNS-variants of such schemes are proposed [16] and their implementations achieved good speedups on platforms where the concurrency of RNS is exploited [23].

D. Number Theoretic Transform (NTT)

The number theoretic transform (NTT) is a version of Discrete Fourier Transform (DFT) over the ring $\mathbb{Z}_q$. Any vector $a = [a_0, a_1, \ldots, a_{n-1}]$ which has n elements in the polynomial domain can be transformed to another vector $\bar{a} = [\bar{a}_0, \bar{a}_1, \ldots, \bar{a}_{n-1}]$ which also has n elements in the NTT domain. The forward and inverse NTTs are defined as in Eqns 1 and 2:

$$\bar{a}_i = \sum_{j=0}^{n-1} a_j \omega^{i \times j} \mod q \text{ for } i = 0, 1, \ldots, n-1, \quad (1)$$

$$a_i = \frac{1}{n} \sum_{j=0}^{n-1} \bar{a}_j \omega^{-i \times j} \mod q \text{ for } i = 0, 1, \ldots, n-1. \quad (2)$$

The NTT (Eqn 1) and INTT (Eqn 2) calculations require the powers of a constant value $\omega \in \mathbb{Z}_q$ referred as the twiddle factor. Two types of twiddle factors are used:

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

As the formulas in Eqns 1 and 2 result in quadratic complexity, for efficient computation of NTT and its inverse, Algorithms 2 and 3 are utilized [24].

Both algorithms are based on the factorization of the cyclotomic polynomial $x^n + 1$ into n degree-1 polynomials as follows:

Algorithm 2 Merge In-place Forward NTT

```
Input: $a(x) \in \mathbb{Z}_q[x]/(x^n + 1)$ polynomial standard-order
Input: $\Psi_{rev}$ (Powers of $\Psi$ stored in bitverse order)
Input: $n = 2^l$, $q$ ($q \equiv 1 \mod n$)
Output: $\bar{a}(x) \in \mathbb{Z}_q[x]/(x^n + 1)$ in bit-reversed order
1: $t = n$;   $m = 1$
2: do
3:     $t = t/2$
4:     for $i$ from 0 by 1 to $m$ do
5:         $j_1 = 2it$
6:         $j_2 = j_1 + t - 1$
7:         for $j$ from $j_1$ by 1 to $j_2 + 1$ do
8:             $U = a_j$
9:             $V = a_{j+t} \cdot \Psi_{br}[m+i] \pmod q$
10:            $a_j = U + V \pmod q$
11:            $a_{j+t} = U - V \pmod q$
12:        end for
13:    end for
14:    $m = 2 \times m$
15: while $m < n$
```

Algorithm 3 Merge In-place Inverse NTT

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

$$x^n + 1 \equiv \prod_{i=0}^{n-1} (x - \psi^{2i+1}) \bmod q \quad (3)$$

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

$$(x^n + 1) \equiv (x^n - \psi^n)$$
$$\equiv (x^{n/2} - \psi^{n/2})(x^{n/2} + \psi^{n/2}) \bmod q \quad (4)$$

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 addition and subtraction operations in Steps 11 and 12 of Algorithm 2.

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

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

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

As can be observed from Algorithm 2, different powers of ψ are stored in the bit-reverse order in the table Ψ_{br}, which simply means that the ith power of ψ is stored in the $(br(i)-1)$th element of Ψ_{br}. For instance, for $n = 8$ the first element of Ψ_{br} holds ψ^4 as the bit-reversed order of $4 = 100$ is 001.

The inverse NTT operation, whose steps are given in Algorithm 3, 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 the final result. The vector $\bar{a}$ before the last iteration is as follows:

$$a = (a_0 + a_{n/2}\psi^{n/2}), \ldots, (a_{n/2-1} + a_{n-1}\psi^{n/2})$$
$$+ (a_0 - a_{n/2}\psi^{n/2}), \ldots, (a_{n/2-1} - a_{n-1}\psi^{n/2}) \quad (5)$$

If the first half is added to the second half, the first half of the resulting vector multiplied by 2 is obtained, $2(a_0, \ldots, a_{n/2-1})$

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

$$2\psi^{n/2}(a_{n/2}, \ldots, a_{n-1}). \quad (6)$$

is obtained. Thus, the result in Eqn 6 needs to be multiplied by $\psi^{-n/2}$. This elaborates the core butterfly operation in Step 12 of Algorithm 3.

As there are $\log_2 n$ iterations in the outermost loop of Algorithm 3 and the vector elements are effectively multiplied by 2 in every iteration, the result needs to be divided by n in $\mathbb{Z}_q$.

The schoolbook multiplication of polynomials $c(x) = a(x) \times b(x)$, where $a(x), b(x) \in \mathbf{R}_q$, can be performed using the method given in Eqn 7.

$$c(x) = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} a_i \times b_j \times x^{i+j} \mod q \quad (7)$$

Due to the quadratic complexity of the schoolbook method, the multiplication in $\mathbf{R}_q$ is slow and inefficient. Moreover, the degree of the resulting polynomial is $2n-2$ as a result of the multiplication, and thus division with $\phi(x)$ must be applied in order to obtain the final result, which is in $\mathbf{R}_q$; i.e., a polynomial with degree at most $n-1$.

The NTT-based polynomial multiplication operation, on the other hand, has logarithmic complexity. Recall that the vector $\bar{a}$ is made up of scalar integers reduced by degree-1 polynomials that are factors of $(x^n + 1)$. This means that if two such vectors $\bar{a}$ 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 NTT, where $c(x) = a(x)b(x)$. When the inverse NTT is applied on $\bar{c}$, $c(x)$ is obtained.

Consequently, an NTT multiplication algorithm can be defined for an efficient multiplication in $\mathbf{R}_q$ as described in Eqns. 8 and 9.

$$\bar{c}(x) = NTT_n(a(x)) \odot NTT_n(b(x)) \quad (8)$$

$$c(x) = INTT_n(\bar{c}(x)) \mod q \quad (9)$$

Note that NTT operations are n-point and no extra polynomial reduction step by $(x^n + 1)$ is needed as $(x^n + 1)$ is factorized into degree-1 polynomials. This makes element-wise multiplications in Eqn. 8 isomorphic to the multiplication in $\mathbf{R}_q$.

E. SEAL BFV Scheme

In this section, we first briefly explain four main operations of the BFV homomorphic encryption scheme; namely key generation, evaluation key generation, encryption and decryption. Then, we give more detailed explanation for three fundamental homomorphic operations that are common in many homomorphic cryptographic applications: addition, multiplication and rotation over encrypted ciphertexts.

1) Key Generation, Encryption and Decryption

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 prime integer. The key generation, evaluation key generation, encryption and the decryption operations of the BFV scheme are shown below, where $\Delta = \lfloor q/t \rfloor$ and χ, ℓ, and w represent a discrete Gaussian distribution, the number of evaluation keys, and the decomposition base, respectively.

- **Key Generation:** $a \leftarrow \mathbf{R}_{q,n},\ s \leftarrow \mathbf{R}_{2,n}$ and $e \leftarrow \chi$,
  $$sk = s,\ pk = (p_0, p_1) = ([-(as + e)]_q, a)$$
- **Evaluation Key Generation:** $a_i \leftarrow \mathbf{R}_{n,q}$ and $e_i \leftarrow \chi$
  $$\text{for } j = 0, \ldots, r-1; i = 0, \ldots, r-2$$
  $$\text{where } f^i = q_{r-1} \mod q_i$$
  $$(evk_i^j[0], evk_i^j[1]) = ([-(a_j s_j + e_j) + f^i s_j^2]_{q_i}, a_j)$$
- **Encryption:** $m \in \mathbf{R}_{t,n},\ u \leftarrow \mathbf{R}_{2,n}$ and $e_1, e_2 \leftarrow \chi$,
  $$ct = (c[0], c[1]) = ([m \cdot \Delta + p_0 u + e_1]_q, [p_1 u + e_2]_q)$$
- **Decryption:** $ct = (c[0], c[1]) \in \mathbf{R}_{q,n}$ and $Sk \in \mathbf{R}_{2,n}$,
  $$m = [\lfloor \tfrac{t}{q}[c[0] + c[1]s]_q \rceil]_t$$

Note that the evaluation keys, evk, are needed to remove the "nonlinear" parts $c[2]$ of the ciphertext $(c[0], c[1], c[2])$ that occur after homomorphic multiplication operations; a process

Algorithm 4 BFV Addition

```
Input: $ct_i, \bar{ct}_i \in \mathbf{R}_{q_i}$ for $0 \leq i < r-1$
Output: $ct_i + \bar{ct}_i \in \mathbf{R}_{q_i}$ for $0 \leq i < r-1$
1: for $i$ from 0 by 1 to $(r-1)$ do
2:     for $k$ from 0 by 1 to 2 do
3:         $\tilde{ct}_i[k] = [ct_i[k] + \bar{ct}_i[k]]_{q_i}$
4:     end for
5: end for
6: return $\tilde{ct} = \tilde{ct}_0, \dots \tilde{ct}_{r-1}$
```

often referred as relinearization. The number of evaluation keys are $2r(r-1)$ in total. Note also that in the RNS variant of the BFV scheme, all operations have to be repeated for each prime base q_i.

2) Addition

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 4, two pairs of ciphertext polynomials in same bases are added or subtracted coefficient-wise, where the moduli are q_i for $i = 0, \dots r-1$. Here, ct_i stands for the ciphertext pair in the modulus q_i for $i = 0, \dots r-1$; namely $ct_i = [ct]_{q_i}$ for ease of notation.

3) Multiplication

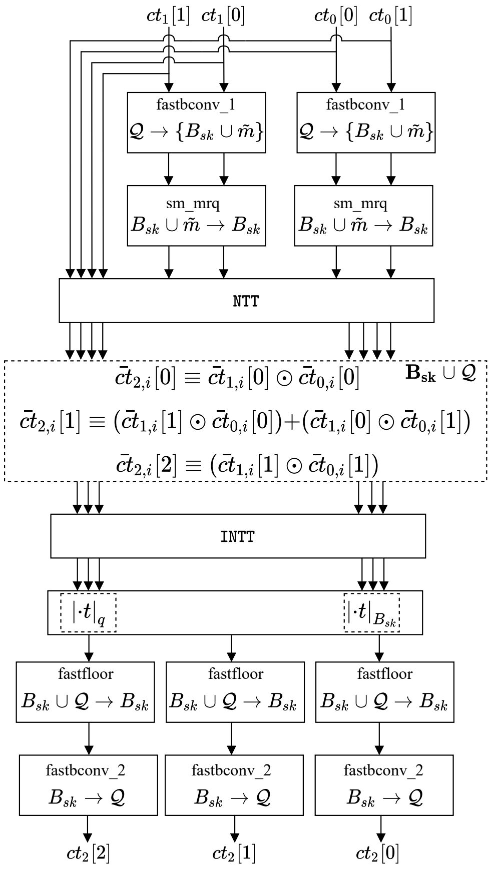
In this section, we explain the homomorphic multiplication operation as illustrated in Figure 1. As pointed out earlier in the RNS variant of the BFV scheme, a set of smaller moduli q_i is used instead of one large coefficient modulus q for the ring arithmetic; a technique known as residue number system (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 complications of using RNS arithmetic in homomorphic multiplication (see [16] 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 $\mathcal{Q}$ base $\{q_0, q_1, \dots, q_{r-1}\}$. The auxiliary base $\mathcal{B}$ consists of $\{B_0, B_1, \dots, B_{\rho-1}\}$, 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 1, the BFV multiplication operation starts by performing the fast base conversion operation `fastbconv_1`, which convert the inputs in $\mathcal{Q}$ to the base



$ct_1[1]$ $ct_1[0]$ $ct_0[0]$ $ct_0[1]$
fastbconv_1 $\mathcal{Q} \rightarrow \{B_{sk} \cup \tilde{m}\}$
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 $B_{sk} \cup \mathcal{Q} \rightarrow B_{sk}$
fastbconv_2 $B_{sk} \rightarrow \mathcal{Q}$
$ct_2[2]$ $ct_2[1]$ $ct_2[0]$

Fig. 1. Homomorphic Multiplication Operation in The BFV Scheme.

$\{B_{sk} \cup \tilde{m}\}$. The `fastbconv_1` operation is followed by the reduction operation, for which the additional base $\tilde{m}$ ise 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 multiplied with plaintext modulus t. Then, the floor operation is used instead of 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 [16] for more detail.

4) Relinearization

The SEAL BFV uses the switchkey technique (Figure 2), which consists of the mix of three different methods for relinearization operation [25], [26], [27]. 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 is the main building

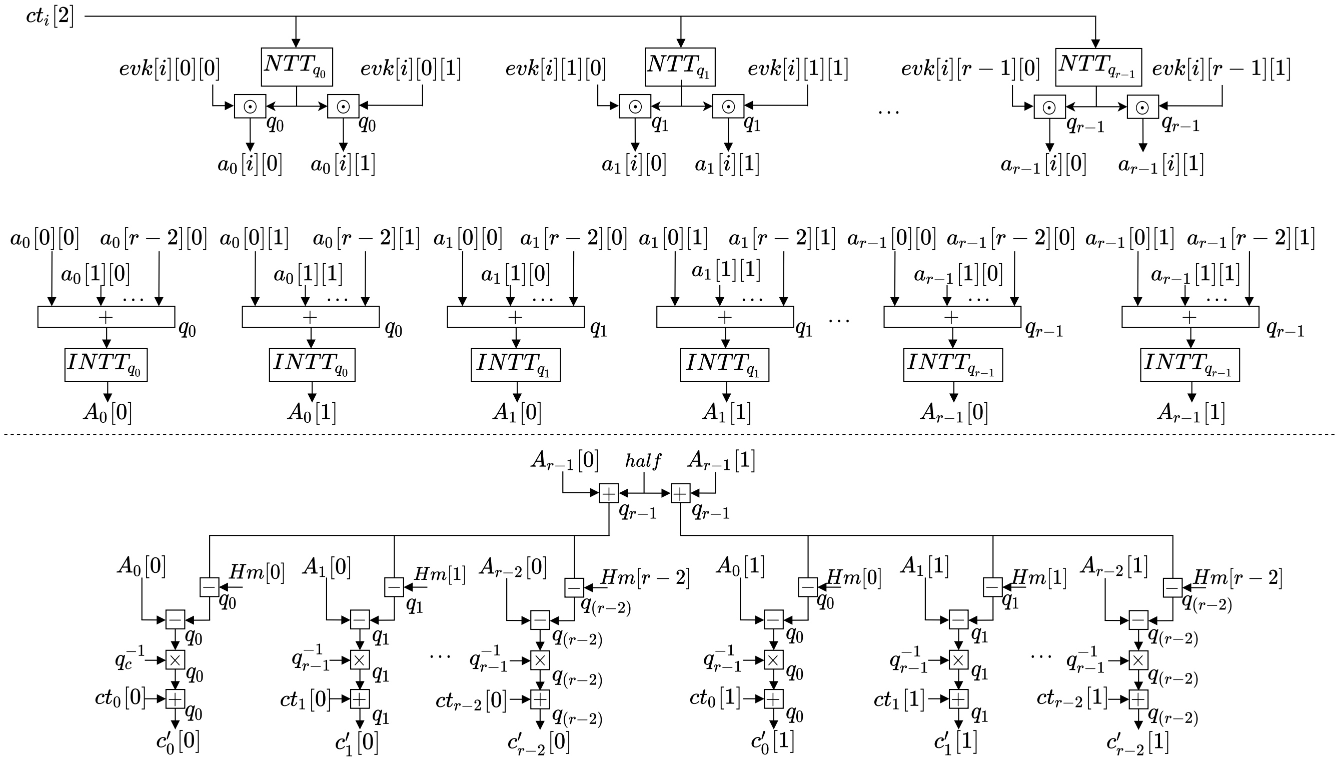


Fig. 2. Switch Key Operation in the BFV Scheme. The symbols $+, -$ and $\times$ represent addition, subtraction and multiplication, respectively in either $\mathbb{Z}_q$ or $\mathbf{R}_q$ while the symbol $\odot$ represents modular pointwise multiplication for vector representation of the elements of $\mathbf{R}_q$ in the NTT domain.

block of the relinearization and the rotation operations.

As shown in Figure 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 q_{r-1} after encryption; ct_i for $i = 0, \dots, r-1$. 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. Then, all results from the multiplication using the same modulus q_i in the RNS base are summed and the resulting vectors are transformed back to the polynomial domain using inverse NTT operation. Finally, as shown in Figure 2, necessary operations are applied to accommodate $ct[2]$ in $ct[0]$ and $ct[1]$. In the figure the half mode $Hm[i] = [\lfloor q_{r-1}/2 \rceil]_{q_i}$. See Algorithm 9 for the details.

5) Rotation

The rotation operation also uses the switch-key operation as in the case of relinearization. However, the operation is based on Galois automorphism [28] and therefore, Galois keys are used for the switch-key operation. For each power of 2 there is 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.

III. GPU Architecture

A graphics processing unit (GPU) is a computing platform, which consists of many cores that can operate on many tasks concurrently, which makes it more suitable for parallel computations. On the other hand, GPU cores are much simpler than CPU cores and run at lower clock frequencies (*cf.* AMD Ryzen7 3800X's cores working at up to 4.2 GHz and NVIDIA RTX 3060Ti's cores working only at 1.66 GHz). Thus, GPUs become much more favorable for performing many simple tasks simultaneously. We will show in Section IV that many time-critical BFV operations can be arranged as independent `for` loops, executed in GPU threads simultaneously.

One of the essential parts of GPUs is "streaming multiprocessors" (SMs); a unit of computing cores running the same GPU kernel. At the highest level of SMs, threads are combined as a 3-dimensional structure called blocks. Also, a grid is a group of blocks launched per GPU kernels. Using kernel launch parameters, one can determine the dimension of blocks and the number of threads per block as needed.

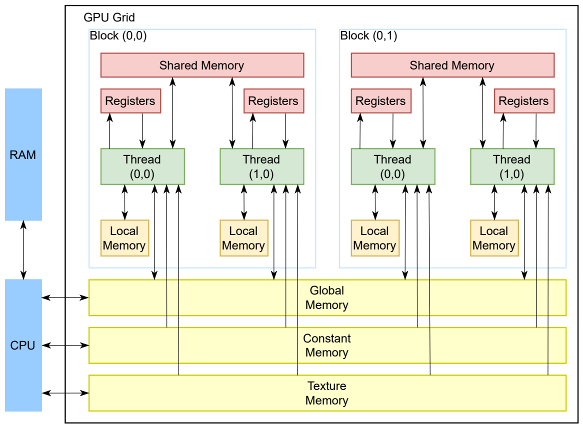


Fig. 3. CUDA Memory Model.

As shown in Figure 3, GPUs have different types of logical memory spaces; namely, shared memory (SMEM), registers, local memory (LMEM), constant memory (CMEM), texture memory (TMEM), and global memory (GMEM). They have different sizes and different usages. For instance, GMEM has large capacity; however, it has high access latency, especially in case of low locality of access. As shown in Table I, registers and SMEM are the fastest types of memory, and their read and write data speeds are similar to a typical L1 cache of a CPU. CMEM is a read-only data memory and since it is accessible from all threads, it performs well when multiple threads access the same data. Table I compares several aspects of GPU memory types [29].

TABLE I
VARIABLES AND ACCESS PENALTIES ON MODERN GPUS MEMORY ARCHITECTURE

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

IV. GPU Implementation

This section presents our implementation technics and methods for four different homomorphic operations of the SEAL BFV scheme: addition, multiplication, relinearization, and rotation. Moreover, we present the implementation of our NTT and INTT algorithms on GPUs. In all GPU implementations that are performed in this section, we minimize the number of kernels. Here, our concern is due to the fact that transferring data from one kernel to another is only possible by using the global memory, and accessing the global memory is prohibitively expensive as explained in Section III. Note that the global memory is accessed only at the beginning and at the end of the kernel. Also, we designed the operations within the kernel in such a way that all required data sharing or exchange among the threads are facilitated only through shared memory, which is much faster than the global memory. The optimal use of global and shared memory decreased the total number of clock cycles spent in memory accesses and boosted memory throughput.

Algorithm 5 Merge in Place Forward NTT on GPU (with `syncthreads`)

Input: $A[n]$, $PsiTable[n], q$
Output: $A[n]$

```
1:  $Idx = blockIdx.x * blockDim.x + threadIdx.x$
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:      $A[addess] = (U + V)\%q$
10:     $V = (V * Psi)\%q$
11:     $A[addess + t] = (U - V)\%q$
12:     __syncthreads()
13: end for
```

A. NTT

The Cooley-Tukey NTT algorithm described in the preliminary section was implemented on the GPU. This section explains the challenges for fast and efficient implementation of NTT and presents our solutions to overcome them. Algorithm 5 shows the GPU pseudo-code for the NTT algorithm, which is essentially the same as the on given in Algorithm 2. One important adaptation to GPU is synchronization operation in Step 12, whose effect on the correctness of the computations will be explained later in this section.

The NTT operation consists of $\log_2 n$ back-to-back 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 GPU. Each SM consists of 4 warps of 32 threads for all GPU models, so the total number of physical threads equals $((\#SM) \times 4 \times 32)$.

When a GPU code is executed, the tasks are performed by warp groups. For example, if code uses the number of threads in the range of [96-128], a total of four warps will be 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 5 (suppose Step 12 of Algorithm 5 is not present). This can be explained

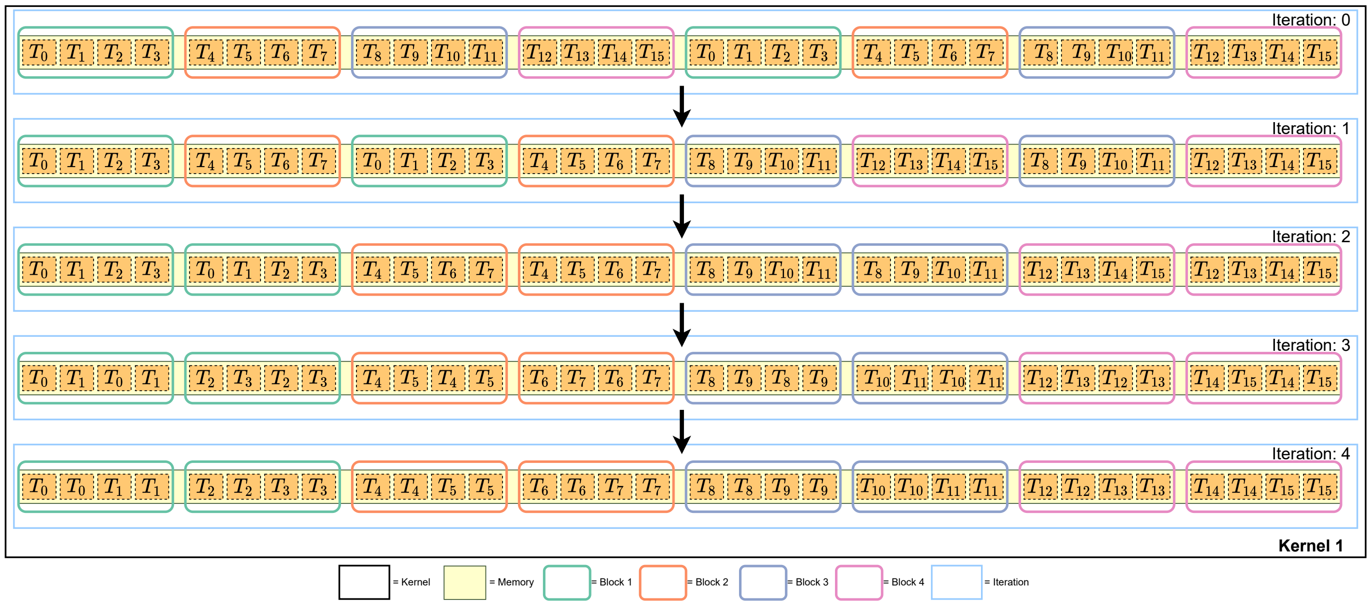


Fig. 4. Execution of Alg. 5 without synchronization in ideal circumstances where $n = 32$

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 5 without synchronization be executed for $n = 32$ on this GPU. Figure 4 portrays a visualization of the execution of Algorithm 5 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 5, perform 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 iteration of the algorithm and performs the same operation, only with a different value of `t` this time. Figure 4 represents the execution of the algorithm in the ideal circumstances as the threads are assumed to finish their tasks simultaneously.

However, Algorithm 5 does not always execute in ideal circumstances. Suppose that the number of threads is only 12 for the scenario in Figure 4. Even when the numbers of threads is less than the number of tasks, incidentally the execution can still be correct. One such scenario is depicted in Figure 5, 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 shows that the same 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 5.

Nevertheless, thread synchronization can be easily in error as visualized in Figure 6. For instance, suppose four threads in the dashed red line, namely T'_2, T'_3, T'_6, T'_7, which are assumed to be scheduled simultaneously. And, since they operate on the same memory locations in two consecutive iterations, there is data dependency between the first two and the last two threads. This will definitely leads 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 5, 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 maximum of 1024 threads for all GPU models, the barrier `__syncthreads()` in Step 12 of Algorithm 5 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 5 on the hypothetical GPU, depicted in Figure 7. The eight threads enclosed 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.

For n values 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 $n = 32$

[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.

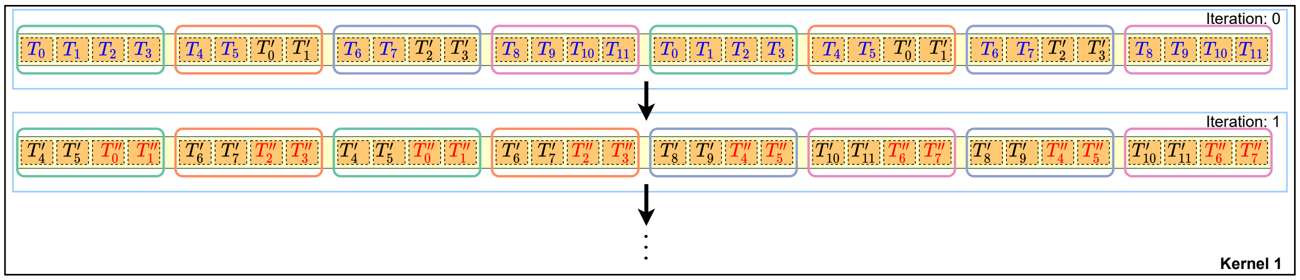


Fig. 5. One good scenario for Alg. 5 without synchronization, where $n = 32$ and $maximum\ block\ size = 4$

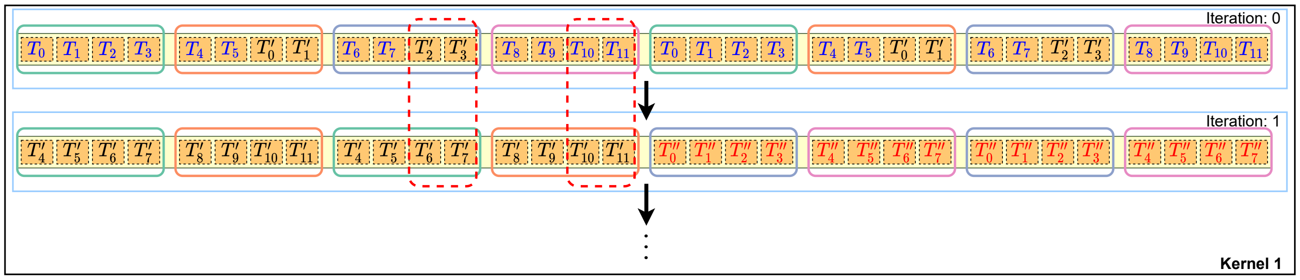


Fig. 6. One problematic scenario for Alg. 5 without synchronization, where $n = 32$

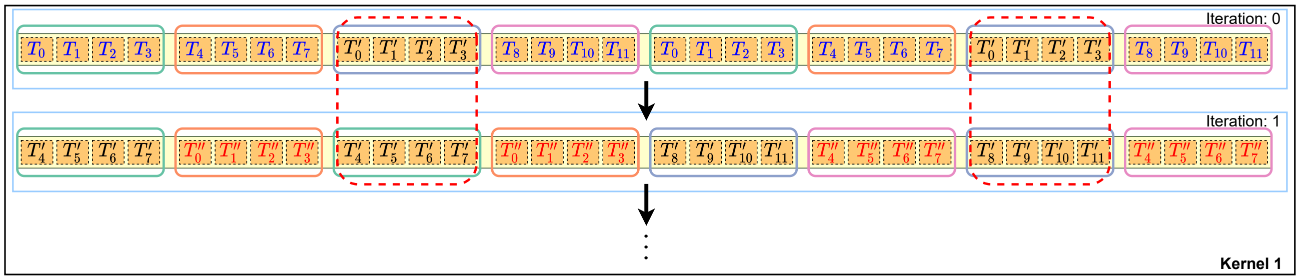


Fig. 7. Another problematic scenario for Alg. 5, where $n = 32$

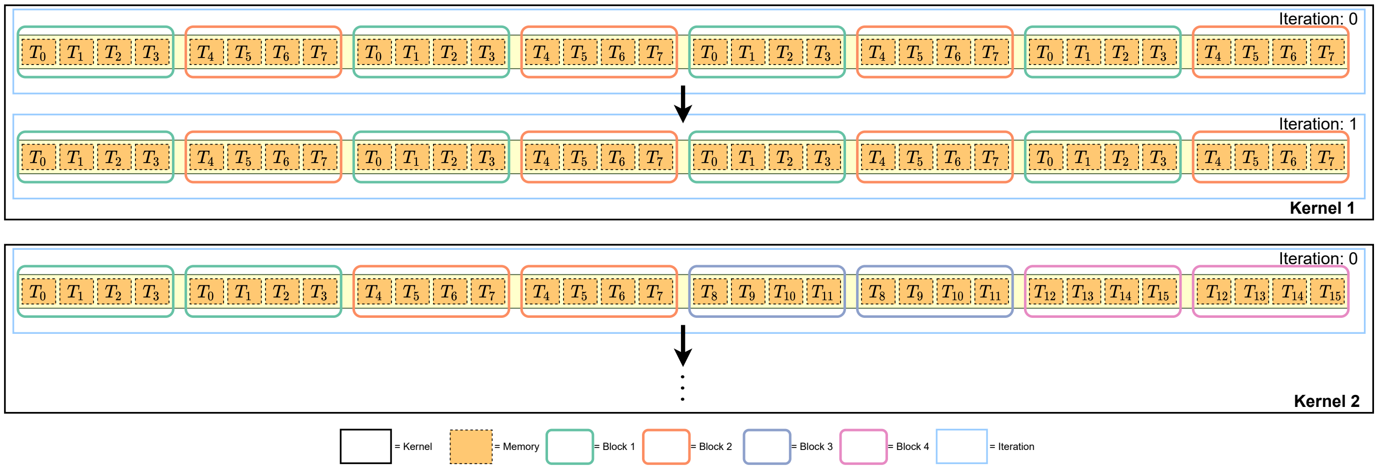


Fig. 8. An example of our NTT algorithm where, $n = 32$ and $maximum\ block\ size = 4$.

on our hypothetical GPU, to resolve the synchronization issue in Figure 7, 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 Section III). 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 [30], and as will show in the subsequent sections, our new approach in this paper 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 8, 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 8, 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 9. 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 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 global memory access. On the other hand, in the approach employed in [30], 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 6.

In the second kernel in Figure 8, 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×blocksize`. This poses no problem as the shared memory is the fastest type of GPU memory (in fact, as fast as the register 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. The new NTT implementation is fast and free of synchronization issues for all values of n and multiple concurrent NTT computations.

Algorithm 6 Kernel 1 in Figure 8

```
Input: $A[n]$, $PsiTable[n], q$
Output: $A[n]$
1: $Idx = blockIdx.x * blockDim.x + threadIdx.x$
2: $m = 1$
3: $k = n/(blockDim.x \times 4)$
4: for $i$ from 0 by 1 to $n/(blockDim.x \times 2)$ do
5:     $reg[i] = A[idx + (i * (blockDim.x \times 2))]$
6: end for
7: for $i$ from 0 by 1 to 2 do
8:     $address = int(idx/(i + 1))$
9:     for $j$ from 0 by 1 to $n/(blockDim.x \times 4)$ do
10:        $location = i \times j + j$
11:        $U = reg[location]$
12:        $V = reg[location + k]$
13:        $V = (V \times PsiTable[address + m]) \mod q$
14:        $reg[location] = (U + V) \mod q$
15:        $reg[location + k] = (U - V) \mod q$
16:    end for
17:    $m = m \times 2$
18: end for
19: for $i$ from 0 by 1 to $n/(blockDim.x \times 2)$ do
20:     $A[idx + (i * (blockDim.x \times 2))] = reg[i]$
21: end for
```

B. SEAL GPU implementation

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 operation are given as pseudo-codes as implemented in the Microsoft SEAL library. All of these algorithms are implemented so that they use our GPU implementation of the NTT algorithm as described in Section IV-A.

1) Homomorphic Addition/Subtraction

As explained in Section II-E2, 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 are 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.

2) Homomorphic 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 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.

The first two CUDA kernel functions are employed to implement base conversion operation from the RNS base $\mathcal{Q}$

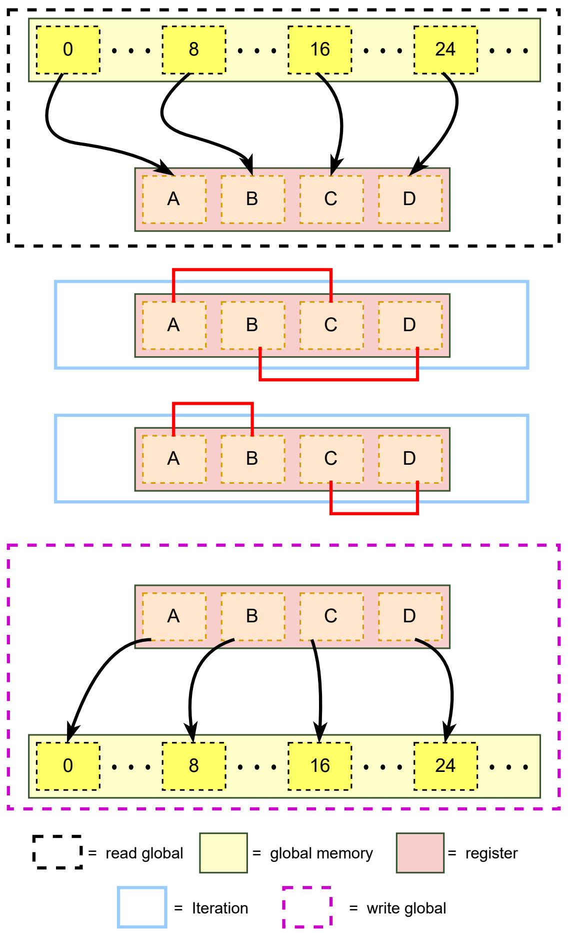


Fig. 9. Register Memory Usage in our NTT Algorithm for $n = 32$.

Algorithm 7 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: end for
4: for $i$ from 0 by 1 to $Bsk\_len - 1$ do
5:     $sum_i^n = 0$
6:     for $j$ from 0 by 1 to $r - 2$ do
7:         for $k$ from 0 by 1 to $n - 1$ do
8:             $sum_i^k = [sum_i^k + (mul_i^k \times base\_c_i^j)]_{Bsk_j}$
9:         end for
10:    end for
11:    $ct_i^n = sum_i^n$
12: end for
```

to $\mathcal{B}_{sk}$. The pseudo-code of the base conversion operation is given in Algorithm 7, as it is implemented in the Microsoft SEAL library. In particular, the first kernel implements the `for` loop in lines 1-3 of Algorithm 7. As the result of the `for` loop is needed in the subsequent operations in lines 4-12 of Algorithm 7 a second kernel is used.

Algorithm 8 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:         $c_i^k = [ct_i^k \times [q]_{Bsk_i}^{-1})]_{Bsk_i}$
6:     end for
7: end for
```

The third CUDA kernel function is implemented to perform the small Montgomery reduction operation (i.e., `sm_mrq` Figure 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 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 the first step of the `fast_floor` function (see Algorithm 8 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 functions 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.

3) *Relinearization*

The BFV relinearization operation uses `switchkey` operation as explained in Figure 2, a pseudo-code of which is given in Algorithm 9 as it is implemented in the Microsoft SEAL library. The relinerization operation usually follows a homomorphic multiplication of ciphertexts, which are given in polynomial domain in BFV. The third component of the ciphertext, $c[2]$, which are to be multiplied with evaluations keys are first converted to the NTT domain using our NTT implementation (see line 5 of Algorithm 9). Then, the multiplication with evaluation keys are performed in the lines 2-9 of Algorithm 9, 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 10-13 of Algorithm 9). The lines 14 and 16

Algorithm 9 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:         end for
 8:     end for
 9: end for
10: for $j$ from 0 by 1 to $r$ do
11:     $A_{j,0} = INTT_{n,q_j}(\bar{A}_{j,0})$
12:     $A_{j,1} = INTT_{n,q_j}(\bar{A}_{j,1})$
13: end for
14: $half = \lfloor \frac{q_{r-1}}{2} \rfloor$
15: for $i$ from 0 by 1 to $r-1$ do
16:     $halfmod = [half]_{q_i}$
17:     for $k$ from 0 by 1 to 1 do
18:         $tmp = [[A_{r-1,k} + half]_{q_{r-1}} - halfmod]_{q_i}$
19:         $tmp = [tmp \times q_r^{-1}]_{q_i}$
20:         $ct_i[k] = [c_i[k] + tmp]_{q_i}$
21:     end for
22: end for
```

Algorithm 10 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:             end if
11:             $c_i^j[k] = r\_val$
12:         end for
13:     end for
14: end for
```

are used to implement the arithmetic with the half modulus as previously described in Figure 2. Lastly, the operation between lines 14 to 22 in Algorithm 9 is implemented with a single kernel.

4) Rotation

The BFV rotation operation uses the so-called `apply_galois` method, whose pseudo-code is given in

Algorithm 11 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:     end if
11:     $gen = 3$
12:     $galois\_elt = 1$
13:     for $i$ from 0 by 1 to $steps$ do
14:         $galois\_elt = galois\_elt \times gen$
15:         $galois\_elt = galois\_elt \; \& \; (m32 - 1)$
16:     end for
17:     return $galois\_elt$
18: end if
```

Algorithm 10 and the `switchkey` operation in Algorithm 9. Before the rotation operation, `galois_elt` algorithm for a given shift amount is executed in CPU using Algorithm 11 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 IV-B3.

V. Experimental Results

In this section, we present our GPU implementation results and their comparison with state-of-the-art works in the literature. Also we present the implementation of gradient boosting framework (XGBoost) [31] using our GPU library to show its performance in practical real-world applications.

For a fair comparison with GPU and CPU implementations of NTT and of the homomorphic operations of the BFV scheme, we used a powerful CPU and two GPU devices, whose configurations are listed in Table II.

TABLE II
Hardware features of the Testbed environment

Feature	CPU	GPU	
		RTX3060Ti (GPU 1)	GTX1080 (GPU 2)
Model	AMD Ryzen7 3800X	RTX3060Ti	GTX1080
Threads	16	4864	2560
Freq.	4.20 GHz	1665 MHz	1733 MHz
RAM	32 GB (3600 MHz)	8 GB	8 GB
Mem. Type	-	GDDR6	GDDR5X
Mem. Bus	-	256 bits	256 bits
Bandwidth	-	448 GB/s	320 GB/s

Mem.: Memory.

TABLE III
COMPARISON RESULTS OF SEAL BFV SCHEME OPERATIONS WITH OUR GPU IMPLEMENTATIONS OF BFV SCHEME OPERATIONS.

			GPU [30]		**GPU with new NTT**		**SEAL**	T
Operation	**n**	$\log_2 q$	RTX3060Ti	GTX1080	RTX3060Ti	GTX1080	CPU	T_s
Add.	2^{12}	109	4 μs	4.6 μs	4 μs	4.6 μs	14 μs	3.5×
	2^{13}	218	5.1μs	6.2 μs	5.1 μs	6.1 μs	58 μs	11.37×
	2^{14}	438	**12.3 μs**	19.4 μs	**12.3 μs**	19.4 μs	**233 μs**	**18.94**×
	2^{15}	881	44 μs	64.2 μs	44 μs	64.2 μs	778 μs	17.68×
Mult.	2^{12}	109	172 μs	259 μs	86 μs	155.8 μs	3212 μs	37.3×
	2^{13}	218	297 μs	532 μs	202 μs	423.4 μs	11883 μs	58.8×
	2^{14}	438	**1037 μs**	2294 μs	**768 μs**	1856.1 μs	**48757 μs**	**63.4**×
	2^{15}	881	5372 μs	10657 μs	3757 μs	-	205295 μs	54.6×
Relin.	2^{12}	109	46 μs	82.7 μs	39.51 μs	59.3 μs	625 μs	15.81×
	2^{13}	218	104 μs	145 μs	88.54 μs	143.4 μs	3100 μs	35.01×
	2^{14}	438	**462 μs**	1013 μs	**376.61 μs**	825.3 μs	**18295 μs**	**48.57**×
	2^{15}	881	3530 μs	6651 μs	3150 μs	-	111736 μs	35.47×
Rot.	2^{12}	109	51 μs	87 μs	42.1 μs	59.4 μs	642 μs	15.24×
	2^{13}	218	116 μs	172 μs	103.3 μs	162.7 μs	3157 μs	30.56×
	2^{14}	438	**544 μs**	1339 μs	**458.7 μs**	1067.2 μs	**18338 μs**	**39.97**×
	2^{15}	881	3879 μs	10504 μs	3464.5 μs	-	113437 μs	32.74×

Add.:BFV Addition. **Mult.**:BFV Multiplication. **Relin.**:BFV Relinearization. T_s: speed up compare to the SEAL with new NTT implemented on RTX 3060Ti *: Include relinearization operation.

A. GPU Implementation of NTT Results and Comparison with related works

Since the BFV scheme used here employs RNS, NTT must be concurrently calculated for each modulus in RNS. Therefore, it is essential to simultaneously perform multiple NTT operations in batches. Naturally, the throughput of NTT operation is as important as (if not more than) the latency of a single NTT operation on GPU. In our GPU implementation we aim to optimize both throughput and latency and we favor the former over the latter most of the time. In the literature, there are few works that report results for batch execution of NTT operations. Thus, in Table IV we included results from [30], which is the only work in the literature that reports batch computation results comparable to ours to the best of our knowledge. Also, the results in [30] represent the state-of-the-art in GPU implementation of NTT.

While most NTT GPU implementations in the literature use special form moduli to accelerate NTT operation, our implementation works with any NTT-friendly modulus and it is still faster. Furthermore, our implementation, which is optimized for performing NTT operations in batches, outperforms those that report only the timings for a single NTT operation in the literature. To compare our work with those that report only single NTT and inverse NTT timings, we include Table V, which shows that, our implementation also outperforms all works in the literature except for one case when a single NTT (iNTT) operation is executed.

In [30], the inverse NTT operation is faster than ours for ring sizes 2^{14} and 2^{15}. For the ring size 2^{14}, the total time of NTT and inverse NTT operations of our implementation is less than that in [30] (compare 42.4 μs and 50 μs). For 2^{15}, the implementation in [30], on the other hand, outperforms ours. Nevertheless, as Table IV shows that our batch implementation outperforms the one in [30] for every case. The performance of batch NTT is much more important as NTT (and inverse NTT) operations are always executed in batches in all homomorphic encryption applications.

We also note that the works [33], [34], and [35] use the special modulus, $Q = 2^{64} - 2^{32} + 1$ known as goldilock modulus, to perform NTT operations faster. However, Q serves as the *carrier* modulus for the actual moduli used in RNS arithmetic in homomorphic encryption applications. Thus, the actual moduli are much smaller due to the constraint $q_i^2 n < Q$ [36]. For example, for the ring size $n = 2^{14}$, each moduli in RNS arithmetic can be at most 25 bit. As our work can employ 64-bit RNS moduli, our actual performance is much better than the implementations in [33], [34], and [35]. For example, to match our size the implementations in those works should use at least twice as many RNS moduli.

The implementations in [33] and [34] take 83.3 μs and 57.8 μs, respectively, for 32768 ring size. And the implementation in [35] takes 66.8 μs for 16384 ring size. Our GPU implementation takes either 19.4 μs (32-bit implementation) or 35.9 μs (64-bit implementation) for the 32768 ring size. On the other hand, when the ring size is 16384, our GPU implementation takes either 13.8 μs (32-bit implementation) or 19.4 μs (64-bit implementation) to perform single NTT operations. As the works in [32], [33], [34] and [35] do not report timing results for batch execution, these works are not included in Table IV, which includes only comparison of batch NTT execution.

Table IV lists the GPU timings for NTT and inverse NTT operations, which are organized into two main columns. On the left are the GPU timings when the modular multiplication

TABLE IV
TIMINGS OF GPU IMPLEMENTATION OF NTT AND INVERSE NTT OPERATIONS AND THEIR COMPARISON WITH [30]

		32 Bit (Implemented On RTX 3060Ti)				64 Bit (Implemented On RTX 3060Ti)			
		Forward NTT		Inverse NTT		Forward NTT		Inverse NTT	
n	**NTT_count**	**[30]**	**T.W.**	**[30]**	**T.W.**	**[30]**	**T.W.**	**[30]**	**T.W.**
2^{12}	4	12.3 μs	**11 μs**	11.2 μs	**11.1 μs**	19.5 μs	**14.3 μs**	16 μs	**15.4 μs**
	16	13 μs	**11.2μs**	12.2 μs	12.2 μs	22.5 μs	**17.2 μs**	17.8 μs	19.4 μs
	32	13.9 μs	17.9 μs	18.4 μs	19.2 μs	23.5 μs	25.2 μs	29.3 μs	**26.6 μs**
	64	23.1 μs	28.6 μs	29.5 μs	**29.3 μs**	39.9 μs	43 μs	52.9 μs	**50.1 μs**
	128	38.9 μs	46.7 μs	48.1 μs	48.7 μs	75.7 μs	81.6 μs	96.5 μs	**91.1 μs**
2^{13}	4	17.1 μs	**12.2 μs**	14.1 μs	14.3 μs	24.4 μs	**16.6 μs**	21.1 μs	**20.5 μs**
	16	18.4 μs	18.4 μs	22.2 μs	**20.3 μs**	28.2 μs	**25.6 μs**	34.8 μs	**28.6 μs**
	32	28.6 μs	29.4 μs	34.9 μs	**30.3 μs**	47.9 μs	**44.4 μs**	59.4 μs	**49.9 μs**
	64	49.8 μs	**46.7 μs**	57.3 μs	**50.7 μs**	97.1 μs	**82.1 μs**	121.2 μs	**93.9 μs**
	128	88 μs	91.2 μs	124 μs	**96.1 μs**	170.7 μs	**156.4 μs**	224.1 μs	**173.6 μs**
2^{14}	4	20.1 μs	**15.2 μs**	17.6 μs	**16.3 μs**	29.98 μs	**21.76 μs**	24.5 μs	**24.1 μs**
	16	33.7 μs	**30.5 μs**	40.9 μs	**30.1 μs**	54.4 μs	**46.54 μs**	65.9 μs	**53.2 μs**
	32	57.3 μs	**50.1 μs**	64.5 μs	**51.8 μs**	121.8 μs	**84.6 μs**	143.3 μs	**97.2 μs**
	64	112.6 μs	**96 μs**	147.4 μs	**99.3 μs**	218.5 μs	**160.5 μs**	266.8 μs	**180 μs**
	128	210.7 μs	**176.1 μs**	277.1 μs	**183 μs**	420.8 μs	**303.19 μs**	511.9 μs	**331.7 μs**
2^{15}	4	25.6 μs	26.4 μs	28.7 μs	**25.9 μs**	35.8 μs	41.2 μs	47.3 μs	50.3 μs
	16	64.1 μs	**52.2 μs**	74.7 μs	**53.2 μs**	147.1 μs	**100 μs**	170.1 μs	**95.6 μs**
	32	136.3 μs	**100.3 μs**	173 μs	**102.4 μs**	266.2 μs	**191.8 μs**	325.5 μs	**193.2 μs**
	64	254.9 μs	**192.1 μs**	322.2 μs	**193.6 μs**	514.2 μs	**372.2 μs**	633.7 μs	**377.7 μs**
	128	491.1 μs	**362.4 μs**	623.1 μs	**364.3 μs**	998.9 μs	**709.3 μs**	1202.9 μs	**725.2 μs**

T.W:This Work.

(see Barret reduction in Algorithm 1) is done using 32-bit modulus q, whereas, on the right, the same timings are listed for a 64-bit q. Note that as the BFV scheme works with integers, only integer arithmetic is employed for these modular multiplications.

Note that as the 64-bit implementation uses twice the size of the modulus than the 32-bit implementations, it can be advantageous for homomorphic operations. For instance, for an acceptable level of security, one must use a 218-bit size for q when $n = 2^{13}$. When we set the sizes of moduli q_i to 32 bit, we use seven RNS moduli q_i. On the other hand, if the size of each q_i is 64 bit, we use only five such moduli.

In the table, `NTT_count` represents the number of independent NTT operations performed simultaneously. We compared forward NTT and inverse NTT separately.

Our timing results show a significant acceleration in comparison with those of the state-of-the-art GPU implementation in the literature [30]. For the 32 bit case, the new forward NTT and inverse NTT are $1.36\times$ and $1.71\times$, faster than their counterparts in the work [30], respectively, where $n = 2^{15}$ and `NTT_count` $= 128$. The sum of NTT and INTT timings, which is $726.7\mu s$, is $1.53\times$ faster than that of [30]. For the 64 bit case with $n = 2^{15}$ and `NTT_count` $= 128$, the new forward NTT and inverse NTT are $1.4\times$ and $1.66\times$ faster than their counterparts in [30], respectively. These performance achievements obtained for NTT and INTT operations help accelerate the operations of the BFV-scheme and any application using homomorphic encryption as shown in the following section.

B. GPU Implementations of BFV HE operations Results and Comparison with related works

There are not many prior works in the literature that presents 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 [35] provides timing results on GPU for homomorphic applications of an old and completely different homomorphic encryption scheme, LTV [37], which is not in use today. We compare the results of our GPU implementation of the BFV-scheme operations with the work [38], which represents the state-of-the-art in the literature for GPU implementation of the BFV scheme. The work [38] provides only the timing results of homomorphic multiplication, which include those of the following relinearization operation.

We first provide our timing results separately in Table III, which includes all major homomorphic artihmetic operations, typically used in many homomorphic applications. Our GPU implementation shows significant improvements over the CPU implementation of the SEAL library running on a CPU. As shown in Table III the proposed GPU library provides up to $63.4\times$ faster BFV multiplication operation, $48.57\times$ faster BFV relinearization operation, $39.97\times$ faster BFV rotation operation, $18.94\times$ faster BFV addition operation, when $n = 2^{14}$

TABLE V
TIMINGS OF GPU IMPLEMENTATION OF SINGLE NTT AND SINGLE INVERSE NTT OPERATIONS AND THEIR COMPARISON WITH THE WORKS IN LITERATURE

Work	Device	n	$\log_2 q$	NTT	INTT
[32]	Titan V	2^{14}	60	44.1 μs	- μs
		2^{15}	60	84.2 μs	- μs
[33]	RTX 2080 Ti	2^{15}	64⋆	83.3 μs	96 μs
[34]	GTX 1070	2^{14}	64⋆	57.8 μs	- μs
[35]	GTX 1070	2^{14}	64⋆	66.8 μs	- μs
[30]	GTX 980	2^{14}	55	51 μs	41 μs
		2^{15}	55	73 μs	52 μs
	GTX 1080	2^{14}	55	33 μs	20 μs
		2^{15}	55	36 μs	24 μs
	Tesla V100	2^{14}	55	29 μs	21 μs
		2^{15}	55	39 μs	23 μs
This Work	RTX 3060 Ti	2^{12}	32	10.2 μs	10.2 μs
		2^{13}	32	10.9 μs	11.1 μs
		2^{14}	32	13.8 μs	14.2 μs
		2^{15}	32	19.4 μs	20.0 μs
		2^{12}	64	14.0 μs	15.0 μs
		2^{13}	64	14.9 μs	17.2 μs
		2^{14}	64	19.1 μs	23.1 μs
		2^{15}	64	35.9 μs	37.1 μs

⋆: Actual q_i is restricted by $q_i^2 n < 2^{64} - 2^{32} + 1$

and $q = 438$ with respect to the SEAL library, which is running on AMD Ryzen7 3800X. The total number of threads available in our GPU devices account for the optimum results obtained at $n = 2^{14}$. Since we parallelized all operations, RTX 3060Ti's threads become fully utilized and, the system cannot be parallelized more. Since the number threads on GTX 1080 is much fewer than RTX 3060 Ti, the best scenario for GTX 1080 is obtained at $n = 2^{13}$.

Then, we compare our results with the work in [38] only for homomorphic multiplication including the following relinearization operation as it is the only one reported. The GPU used in [38] 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 [38] for a fair comparison. As observable from Table VI, our GPU implementation outperforms that in [38] 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 [38], respectively.

C. Implementation Results of Privacy-Preserving Inference for Genome Data using XGBoost Trees

Mağara et al. [31] introduced 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 gradient-boosted tree 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.

TABLE VI
COMPARISON RESULTS OF OUR GPU IMPLEMENTATION WITH STATE-OF-THE-ART WORK

			[38]	**T.W**	
Operation	n	$\log_2 q$	Tesla V100	RTX 3060Ti	T
Mult. + Relin.	2^{12}	60	859 μs	136 μs	6.31 ×
	2^{13}	120	1012 μs	170 μs	5.95 ×
	2^{14}	360	2010 μs	661 μs	3.04 ×
	2^{15}	600	4826 μs	2875 μs	1.67 ×

Mult. + Relin:Sum of multiplication and relinearization operations
T.W.:This Work. T:The ratio of work [38] over this work.

As explained in [31], 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.

As shown in Table VII, our GPU library accelerates the classification operation at least 42.98 times with respect to the results obtained from AMD Ryzen7 3800X CPU with single thread. When all threads are used in the CPU, the speedup will be 5.7.

TABLE VII
IMPLEMENTATION OF GRADIENT BOOSTING FRAMEWORK(XGBOOST) RESULTS

		SEAL		**T.W**		
n	$\log_2 q$	S.T.	M.T.	RTX 3060Ti	T	S
2^{13}	218	25.62 s	3.4 s	0.596 s	42.98×	5.7×
2^{14}	438	127.028 s	19.27 s	2.41 s	52.7×	8×

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.

VI. CONCLUSION

In this paper, we presented a GPU library that features highly parallelized and optimized implementations of NTT and inverse NTT operations and homomorphic operations of the BFV scheme. Although the library can be independently used, it is also integrated with the Microsoft SEAL library and its functions can be called from any application code using SEAL.

Therefore, the library is truly an accelerator for homomorphic encryption applications.

By reducing the number of GPU kernel function calls and optimized use of fast memory on GPU, the library offers the best timing performance for NTT and inverse NTT operations in the literature. For instance, concurrent executions of 128 NTT and INTT operations for the ring degree of 2^{14} take 303.19 μs and 331.7 μs, respectively, on RTX3060Ti GPU, which are 1.39 and 1.54 times faster than those of the state-of-the-art GPU implementation reported in the literature.

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 runnning on RTX3060Ti achieves speedups of 18.94, 63.4, 48.57, and 39.97 for homomorphic addition, 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 out ours is up to 6.31 times faster than the latter.

We also showed that the proposed GPU library is profitably used in homomorphic processing of real data such as the classification of encrypted genome data for tumor types and reported at least a speedup of 5 in comparison with a powerful CPU running 16 threads.

In conclusion, 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 a future work, we envision to integrate our GPU accelerator to other HE libraries, and use 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 [13].

Acknowledgment

This research is supported in part by The Scientific and Technological Research Council of Turkey under Grant Number 118E725.

References

[1] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*, ser. STOC '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 169–178. [Online]. Available: https://doi.org/10.1145/1536414.1536440

[2] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," *J. ACM*, vol. 60, no. 6, Nov. 2013. [Online]. Available: https://doi.org/10.1145/2535925

[3] C. Gentry and S. Halevi, "Implementing gentry's fully-homomorphic encryption scheme," in *Advances in Cryptology – EUROCRYPT 2011*, K. G. Paterson, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 129–148.

[4] Z. Brakerski and V. Vaikuntanathan, "Efficient fully homomorphic encryption from (standard) LWE," *SIAM Journal on Computing*, vol. 43, no. 2, pp. 831–871, 2014. [Online]. Available: https://doi.org/10.1137/120868669

[5] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," *ACM Trans. Comput. Theory*, vol. 6, no. 3, Jul. 2014. [Online]. Available: https://doi.org/10.1145/2633600

[6] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption." *IACR Cryptol. ePrint Arch.*, vol. 2012, p. 144, 2012.

[7] "Microsoft SEAL (release 3.6)," https://github.com/Microsoft/SEAL, Nov. 2020, microsoft Research, Redmond, WA.

[8] "PALISADE Lattice Cryptography Library (release 1.11.5)," https://palisade-crypto.org/, 2021.

[9] S. Halevi and V. Shoup, "Algorithms in helib," in *Advances in Cryptology – CRYPTO 2014*, J. A. Garay and R. Gennaro, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 554–571.

[10] W. Wang, Z. Chen, and X. Huang, "Accelerating leveled fully homomorphic encryption using gpu," in *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2014, pp. 2800–2803.

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

[12] Y. Doröz, E. Öztürk, E. Savaş, and B. Sunar, "Accelerating ltv based homomorphic encryption in reconfigurable hardware," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2015, pp. 185–204.

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

[14] NVIDIA Corporation, "NVIDIA CUDA C programming guide," 2010, version 3.2.

[15] C. Boura, N. Gama, M. Georgieva, and D. Jetchev, "CHIMERA: combining ring-lwe-based fully homomorphic encryption schemes," *J. Math. Cryptol.*, vol. 14, no. 1, pp. 316–338, 2020. [Online]. Available: https://doi.org/10.1515/jmc-2019-0026

[16] J. Bajard, J. Eynard, M. A. Hasan, and V. Zucca, "A full RNS variant of FV like somewhat homomorphic encryption schemes," in *Selected Areas in Cryptography - SAC 2016 - 23rd International Conference, St. John's, NL, Canada, August 10-12, 2016, Revised Selected Papers*, ser. Lecture Notes in Computer Science, R. Avanzi and H. M. Heys, Eds., vol. 10532. Springer, 2016, pp. 423–442. [Online]. Available: https://doi.org/10.1007/978-3-319-69453-5_23

[17] P. Barrett, "Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor," in *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings*, ser. Lecture Notes in Computer Science, vol. 263. Springer, 1986, pp. 311–323.

[18] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, pp. 519–521, 1985.

[19] S. Antao, J. Bajard, and L. Sousa, "RNS-based elliptic curve point multiplication for massive parallel architectures," *Comput. J.*, vol. 55, no. 5, pp. 629–647, 2012. [Online]. Available: https://doi.org/10.1093/comjnl/bxr119

[20] J. Bajard, J. Eynard, N. Merkiche, and T. Plantard, "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*. IEEE, 2015, pp. 113–120. [Online]. Available: https://doi.org/10.1109/ARITH.2015.30

[21] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," *IACR Cryptol. ePrint Arch.*, p. 144, 2012. [Online]. Available: http://eprint.iacr.org/2012/144

[22] Z. Brakerski, "Fully homomorphic encryption without modulus switching from classical gapsvp," in *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, ser. Lecture Notes in Computer Science, R. Safavi-Naini and R. Canetti, Eds., vol. 7417. Springer, 2012, pp. 868–886. [Online]. Available: https://doi.org/10.1007/978-3-642-32009-5_50

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

[24] J. W. Cooley and J. W. Tukey, “An algorithm for the machine calculation of complex fourier series,” *Mathematics of Computation*, vol. 19, pp. 297–301, 1965.

[25] J.-C. Bajard and T. Plantard, “Rns bases and conversions,” in *SPIE Optics + Photonics*, 2004.

[26] S. Halevi, Y. Polyakov, and V. Shoup, “An improved rns variant of the bfv homomorphic encryption scheme,” in *Cryptographers’ Track at the RSA Conference*. Springer, 2019, pp. 83–105.

[27] H. Chen, W. Dai, M. Kim, and Y. Song, “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*, ser. CCS ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 395–412. [Online]. Available: https://doi.org/10.1145/3319535.3363207

[28] K. Laine, “Simple encrypted arithmetic library 2.3.1,” *Microsoft Research, WA, USA*, 2017.

[29] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, “Dissecting the NVIDIA volta GPU architecture via microbenchmarking,” *CoRR*, vol. abs/1804.06826, 2018. [Online]. Available: http://arxiv.org/abs/1804.06826

[30] Ö. Özerk, C. Elgezen, A. C. Mert, E. Öztürk, and E. Savas, “Efficient number theoretic transform implementation on GPU for homomorphic encryption,” *J. Supercomput.*, vol. 78, no. 2, pp. 2840–2872, 2022. [Online]. Available: https://doi.org/10.1007/s11227-021-03980-5

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

[32] S. Kim, W. Jung, J. Park, and J. H. Ahn, “Accelerating number theoretic transformations for bootstrappable homomorphic encryption on gpus,” in *2020 IEEE International Symposium on Workload Characterization (IISWC)*, 2020, pp. 264–275.

[33] Z. Zheng, “Encrypted cloud using GPUs,” Master’s thesis, KU Leuven, 2020.

[34] J.-Z. Goey, W.-K. Lee, B.-M. Goi, and W.-S. Yap, “Accelerating number theoretic transform in gpu platform for fully homomorphic encryption,” *The Journal of Supercomputing*, vol. 477, pp. 1455–1474, 2021. [Online]. Available: https://doi.org/10.1007/s11227-020-03156-7

[35] W. Dai and B. Sunar, “cuhe: A homomorphic encryption accelerator library,” in *International Conference on Cryptography and Information Security in the Balkans*. Springer, 2015, pp. 169–186.

[36] W. Dai, Y. Doröz, and B. Sunar, “Accelerating NTRU based homomorphic encryption using gpus,” in *IEEE High Performance Extreme Computing Conference, HPEC 2014, Waltham, MA, USA, September 9-11, 2014*. IEEE, 2014, pp. 1–6. [Online]. Available: https://doi.org/10.1109/HPEC.2014.7041001

[37] A. López-Alt, E. Tromer, and V. Vaikuntanathan, “Multikey fully homomorphic encryption and applications,” *SIAM J. Comput.*, vol. 46, no. 6, pp. 1827–1892, 2017. [Online]. Available: https://doi.org/10.1137/14100124X

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