Faster Homomorphic Encryption over GPGPUs via hierarchical DGT

Pedro Geraldo M. R. Alves[1] (✉), Jheyne N. Ortiz[1], and Diego F. Aranha[2]

[1] University of Campinas, Campinas, Brazil
{pedro.alves, jheyne.ortiz}@ic.unicamp.br
[2] Aarhus University, Aarhus, Denmark
dfaranha@cs.au.dk

Abstract. Privacy guarantees are still insufficient for outsourced data processing in the cloud. While employing encryption is feasible for data at rest or in transit, it is not for *computation* without remarkable performance slowdown. Thus, handling data in plaintext during processing is still required, which creates vulnerabilities that can be exploited by malicious entities. Homomorphic encryption schemes enable computation over ciphertexts without knowing the related plaintexts or the decryption key. This work focuses on the challenge of developing an efficient implementation of the BFV scheme on CUDA. This is done by combining and adapting different literature approaches, as the *double*-CRT representation and the Discrete Galois Transform. Moreover, we propose and implement an improved formulation of the DGT inspired by classical algorithms, which computes the transform up to 2.6 times faster than the state-of-the-art. By using these approaches, we obtain up to 3.6 times faster homomorphic multiplication.

Keywords: Fully Homomorphic Encryption · BFV · CUDA · Polynomial multiplication · Privacy-preserving computing

1 Introduction

With the growing data collection by governments and companies, protecting its secrecy becomes as important as processing and extracting useful information. However, how to efficiently collect and compute user data without undermining their privacy is an open problem. System breaches may happen even when data holders choose the most conservative practices and never share data intentionally.

The Breach Level Index provides distressful statistics about data leakage. It states that most breaches occur by accidental loss on leaving plaintext data exposed inadvertently. However, attacks from malicious parties, which explore vulnerabilities to subvert security mechanisms, are also far from negligible [27]. Data can be protected by encryption even in case of leakage. However, encryption-decryption cycles during its lifespan create a weak point in the system's security.

Hence, building the system founding attached to mathematical guarantees and dispensable decryption is the only way to achieve reliable security.

Homomorphic Encryption (HE) schemes enable data processing while protecting its confidentiality. They allow the evaluation of arithmetic circuits over ciphertexts by a third party without any knowledge of the corresponding plaintexts or the decryption key, preventing the computation's inputs and outcome to be learned. Hence, HE is a natural candidate for solving privacy issues caused by malicious third parties, careless administrators, or other security flaws during the processing, such as side-channel vulnerabilities.

Many of the HE schemes available in the literature rely on the hardness of the Ring-Learning with Errors (RLWE) problem. The RLWE assumption offers a strategy for protecting messages, encoded as polynomials in $R_q = \mathbb{Z}_q[x]/(f(x))$, by adding noise in a way that it can only be removed when given a trapdoor. There are several recent proposals following this approach in cryptosystems such as BFV [19], CKKS [11], and TFHE [12]. All depend on polynomial arithmetic as the main building block, so its efficient and reliable implementation is critical for adopting HE schemes in real-world scenarios.

CUDA is an important tool for the efficient implementation of polynomial arithmetic. It is a SIMD architecture developed and maintained by NVIDIA for employing the data parallelism potential of a GPU in tasks beyond graphical processing. However, the particularities of CUDA impose challenges for its cryptographic use. Its processing flow demands careful planning to align possible conditional branches with certain thread groups, and its memory paradigm considers several structures with different dimensions and latency characteristics, separated from the machine's main memory. Moreover, at this point, no general-purpose cryptographic library or polynomial arithmetic framework supports CUDA. Hence, these constraints motivate the development of a complete toolkit to work as an arithmetic engine aimed at RLWE-based cryptosystems.

Our contributions. This work presents mathematical tools and techniques for the efficient implementation of the BFV scheme in CUDA. We follow the literature by employing the Residue Number System (RNS) as the best approach for handling the multiprecision arithmetic required by the cryptosystem, and the Halevi, Polyakov, and Shoup modification of BFV to solve the division and rounding problem in the RNS domain [7,22]. The main contributions of this study are:

– A novel *hierarchical* formulation of the Discrete Galois Transform (DGT) that offers about two times lower latency on GPUs than the best version previously available in the literature. Moreover, we collect evidence that suggests it is faster than the commonly used Number Theoretic Transform (NTT) due to its lower memory bandwidth requirement. Such formulation is inspired by Bailey's version of the Fast Fourier Transform [5].
– Compatible choice of parameters between the DGT and the RNS representation. We show that the *double*-CRT representation proposed by Gentry et al. is a better implementation design than the usual approach of working with Mersenne or Solinas primes in different rings [8].

– A more efficient and polynomial-oriented state machine which reduces the need for moving data in and out of the DGT domain and between the main memory and the GPU global memory.

These contributions are not limited to the BFV cryptosystem and can be easily applied to other RLWE-based schemes, such as CKKS. Moreover, we provide latency benchmarks from a proof-of-concept implementation named SPOG, which was built based on the methods above. Two relevant works employing the DGT are considered for comparison with our results: Badawi, Polyakov, Aung, Veeravalli, and Rohloff [2]; and Badawi, Veeravalli, Mun, and Aung [4]. When considering homomorphic multiplication as the main performance-critical operation, SPOG offers higher performance against these works, surpassing a 3.6-factor performance improvement against the latter.

2 Mathematical background

The efficient implementation of an RLWE-based cryptosystem on CUDA requires carefully designed building blocks for adjusting the operations to the architecture's limitations. The BFV cryptosystem, as well as and other HE proposals, relies on large parameters for achieving proper security levels. This imposes a challenge in the light of GPGPUs' [3] constraints, for both the size of the coefficients, much larger than the native integer instruction set; and the polynomial arithmetic, that requires highly-optimized algorithms to reduce the computational complexity and improve the scalability of expensive operations, such as polynomial multiplication.

This Section describes the Fan and Vercauteren cryptosystem; presents the Residue Number System (RNS) representation, used to avoid the multiprecision arithmetic; and introduces the Discrete Galois Transform (DGT), a more suitable variant of the Fast Fourier transform (FFT) to GPU implementation.

2.1 The BFV cryptosystem

Fan and Vercauteren proposed a variant of Brakerski's homomorphic cryptosystem, nowadays referred to as BFV, that relies on the hardness of the Ring-Learning With Errors (RLWE) problem [19]. Classified as a leveled homomorphic encryption scheme (LHE), it is currently one of the most efficient cryptosystems of its class concerning speed and memory consumption and remains untouched by recent advances in cryptanalysis [1,14].

Let $p > 1$ be an integer and n a power-of-2. BFV's basic arithmetic is built upon polynomial rings of the form $R_p = \mathbb{Z}_p[X]/(X^n+1)$. The scheme defines the following parameter set: a security parameter λ; a decomposition base $\omega > 1$; the modulus $t \geq 2$ that determines the plaintext domain R_t; and the modulus $q \gg t$ that determines the ciphertext domain R_q. Moreover, it makes use of

[3] GPGPU, acronym for General-Purpose Graphics Processing Unit.

an error distribution χ_{err}, usually a zero-mean discrete Gaussian distribution parameterized by the standard deviation σ.

Let $l = \lfloor \log_\omega q \rfloor$. The main procedures of BFV are the following:

KeyGen(λ, ω): Let $\mathtt{sk} \leftarrow R_2$ be the secret key. Sample $a \leftarrow R_q$ uniformly at random and $e \leftarrow \chi_{err}$, and define the public key $\mathtt{pk} = (b, a) = ([-(a \cdot \mathtt{sk} + e)]_q, a)$. Generate the evaluation key $\mathtt{evk}$ as: Sample $\mathbf{a}_i \leftarrow R_q$ uniformly at random, $\mathbf{e}_i \leftarrow \chi_{err}$, and compute $\gamma_i = \big([-(\mathbf{a}_i \cdot \mathtt{sk} + \mathbf{e}_i) + \omega^i \cdot \mathtt{sk}^2]_q, \mathbf{a}_i\big)$. Define $\mathtt{evk} = \bigcup_{i=0}^{l} \gamma_i$. Output $(\mathtt{sk}, \mathtt{pk}, \mathtt{evk})$.

Encrypt(m, pk): for a plaintext message $m \in R_t$ and a public key $\mathtt{pk} = (b, a)$, sample $u \leftarrow R_2$ uniformly at random and $e_1, e_2 \leftarrow \chi_{err}$, and compute the ciphertext $\mathbf{c} = \Big([\Delta m + b \cdot u + e_1]_q, [a \cdot u + e_2]_q\Big)$, where $\Delta = \lfloor q/t \rfloor$.

Decrypt(c, sk): for a ciphertext $\mathbf{c} = (c_0, c_1)$ and the secret key $\mathtt{sk} = s$, recover the plaintext $m = \left[\left\lfloor \frac{t}{q} [c_0 + c_1 \cdot s]_q \right\rceil\right]_t$.

Add($\mathbf{c}_0$, $\mathbf{c}_1$) : for ciphertexts $\mathbf{c}_0 = (c_{0,0}, c_{0,1})$ and $\mathbf{c}_1 = (c_{1,0}, c_{1,1})$, compute $\mathbf{c}_{\mathtt{add}} = ([c_{0,0} + c_{1,0}]_q, [c_{0,1} + c_{1,1}]_q)$.

Relin((c_0, c_1, c_2), evk) : for $c_0, c_1, c_2 \in R_q$, $\mathtt{evk} = (\mathbf{b}, \mathbf{a})$, and a decomposition of c_2 in base w such that $c_2 = \sum_{i=0}^{l} c_2^{(i)} w^i$, return

$$\left(\left[c_0 + \sum_{i=0}^{l} \mathbf{b}_i \cdot c_2^{(i)}\right]_q, \left[c_1 + \sum_{i=0}^{l} \mathbf{a}_i \cdot c_2^{(i)}\right]_q\right).$$

Mul($\mathbf{c}_0$, $\mathbf{c}_1$, evk) : for ciphertexts $\mathbf{c}_0 = (c_{0,0}, c_{0,1})$ and $\mathbf{c}_1 = (c_{1,0}, c_{1,1})$, compute

$$c = \left(\left[\left\lfloor \frac{t}{q} \cdot c_{0,0} \cdot c_{1,0} \right\rceil\right]_q, \left[\left\lfloor \frac{t}{q} \cdot (c_{0,0} \cdot c_{1,1} + c_{0,1} \cdot c_{1,0}) \right\rceil\right]_q, \left[\left\lfloor \frac{t}{q} \cdot c_{0,1} \cdot c_{1,1} \right\rceil\right]_q\right)$$

and return $\mathbf{c}_{\mathtt{mul}} = \mathtt{Relin}(c, \mathtt{evk})$.

2.2 Residue Number System

As can be observed in Section 2.1, BFV depends upon computationally expensive polynomial operations. Moreover, the literature reveals that big integer arithmetic is required to offer proper security levels [26]. A common strategy in implementations of BFV is to use the Chinese Remainder Theorem (CRT) on the Residue Number System (RNS) to map large integers to a set of smaller residues capable of being evaluated by processor native instructions [17,7].

Definition 1 (CRT). *Let x be a polynomial in R_q, and $\{p_0, \ldots, p_{\ell-1}\}$ a set of pairwise coprimes. The CRT decomposition results in a set X with ℓ residues such that $CRT(x) = \left\{[x]_{p_0}, \ldots, [x]_{p_{\ell-1}}\right\}$. The inverse $CRT(X)$ is defined as:* $\left[\sum_{i=0}^{\ell-1} \frac{M}{p_i} \cdot \left[\left(\frac{M}{p_i}\right)^{-1} X_i\right]_{p_i}\right]_M = x$, *where* $M = \prod_{i=0}^{\ell-1} p_i$.

Addition and multiplication in the RNS domain work by applying the operation residue-wise. However, division and modular reduction are more complicated and require a more advanced technique, as described next.

2.3 Division and rounding inside the RNS domain

Some parts of BFV are hardly compatible with RNS, such as coefficient-wise division and rounding used in decryption and homomorphic multiplication. Motivated by that, two variants of BFV can be found in the literature, BEHZ-BFV and HPS-BFV, which propose modifications to the cryptosystem to support them in the RNS domain [6,22].

Let $Q = \{q_0, q_1, \ldots, q_{\ell-1}\}$ be a RNS basis which we can use to represent any ciphertext, as described in Section 2.2. BEHZ-BFV and HPS-BFV claim that the division and rounding can be computed by extending base Q to a new basis $B = \{b_0, b_1, \ldots, b_{k-1}\}$ such that $\prod q_i < \prod b_j$. While BEHZ-BFV looks for an exact rounding, HPS-BFV shows how to build operations to minimize the error and merge it into the natural cryptosystem noise. This allows a much simpler procedure, with a lower computation cost, to be used. HPS-BFV's authors present a performance analysis that demonstrates that their procedures are simpler and have lower complexity and noise growth than those proposed by Bajard et al. .

The HPS-BFV methods are composed by a basis extension procedure, which computes a polynomial representation in a base B from its representation in base Q; and two scaling methods to scale down and round an integer in its RNS representation by t/q, one to be used on decryption, which is a more straightforward scenario that requires the output to be in base $\{t\}$, and one for homomorphic encryption, which is a bit more complicated since the outcome must lie in base B.

Both variants of BFV take the fact that q is not defined as a prime integer. Thus, they represent and work with R_q polynomials in an RNS base composed by a factorization of q, i.e. $q = \prod_{i=0}^{\ell-1} q_i$. One of the advantages of doing this is the automatic merge of the RNS bounds with the ciphertext coefficient domain.

2.4 Discrete Galois Transform

The Fast Fourier Transform (FFT) is a well-known method that offers linear computational cost for polynomial multiplication when the operands lie in its domain and quasi-linear when considering the computation of the transform itself. However, the FFT is defined on $\mathbb{C}$, which makes it harder for its direct applicability in the context of RLWE-based cryptosystems, defined on integer domains. Thus, variations offering the same functionality but built upon integer arithmetic were proposed in the literature, such as the Number Theoretic Transform (NTT) over $GF(p)$, and the Discrete Galois Transform (DGT) over $GF(p^2)$, for some convenient choice of a prime number p [24,15].

The main difference of DGT over NTT is caused by their domains, which results in memory bandwidth savings, as deeply discussed in Sections 3 and 4. Despite this, they are sufficiently similar so that they share computation data paths and their efficient implementation strategies. Furthermore, as $GF(p^2)$ can be represented in the set of Gaussian integers $\mathbb{Z}_p[i] = \{a + ib \mid a, b \in \mathbb{Z}_p\}$, it uses finite field arithmetic with $\mathbb{Z}_p$ elements as building blocks, which resonates with the representation used by RNS and BFV. In Definition 2 we introduce the base formulation, as done in [3].

Definition 2 (Discrete Galois Transform). *Let $p \geq 3$ be a prime number, $x = \{x_0, \ldots, x_{n-1}\}$ be a vector of length n such that $x_i \in GF(p^2)$ for $0 \leq k < n$, and g be an n-th primitive root of unity in $GF(p)$. Then, the DGT and its inverse are defined as: $X_k = \sum_{j=0}^{n-1} x_j g^{-jk} \in GF(p^2)$ and $x_k = n^{-1} \sum_{j=0}^{n-1} X_j g^{jk} \in GF(p^2)$, respectively.*

3 Efficient CUDA operation on cyclotomic rings

An efficient implementation of the arithmetic of cyclotomic polynomial rings requires a convenient approach for polynomial multiplication and a proper data representation, not only with low computational complexity but also that fits well in the processing hardware. This Section provides optimization strategies for implementing polynomial arithmetic on CUDA.

3.1 Fast polynomial multiplication

The complexity to compute a polynomial multiplication using a textbook formula is $\Theta\left(n^2\right)$ for n-degree polynomials, which means that performance will be seriously affected with the increase of the degree.

In the context of cryptosystems based on RLWE, as observed by Lindner and Peikert, security is strongly related to the degree of the polynomial ring [23]. Specifically on BFV, Player concludes that a parameter set nowadays considered secure, with an estimated security upper bound close to $\lambda = 128$, requires n between 2^{11} and 2^{15} [26]. Hence, an efficient implementation of polynomial multiplication for operands with a large degree is vital for the cryptosystem's performance.

FFT-based transforms, such as the NTT, provide a domain in which the polynomial multiplication complexity is reduced to $\Theta(n)$, and among those, the DGT is a promising variant defined over $GF(p^2)$. As introduced in Section 2.4, this field can be represented as the set of Gaussian integers $\mathbb{Z}_p[i] = \{a + ib \mid a, b \in \mathbb{Z}_p\}$, which enables the polynomial folding of inputs and consequently halves their degree. This folding works such that, for a polynomial $P(x) = \sum_{j=0}^{n-1} a_j \cdot x^j$, we have $\mathit{fold}(P(x)) = \sum_{j=0}^{n/2-1} (a_j + i \cdot a_{j+n/2}) \cdot x^j$, for $i = \sqrt{-1}$.

Considering the use of Gaussian integer arithmetic, as defined in Appendix A, a first impression may be that the increased cost of the arithmetic nullifies the reduction of the polynomial degree due to the quadratic extension. However, it is important to notice that, by working with half the coefficients, only half the roots, like those in Definition 2, are required compared to the FFT NTT. In this way, in a memory-constrained scenario, this property implies a speedup caused by fewer memory accesses and enables a more coalesced pattern. In the case of CUDA, such operations may target the GPU's global memory, which is significant in size but has high latency, or even shared or constant memories, which are fast but very small. The resulting increased arithmetic density favors GPU implementations.

Badawi et al. propose Algorithm 1 for polynomial multiplication through the DGT. It first folds both input signals and then applies a twisting by powers of $n/2$-th primitive roots of i, which provides a negacyclic convolution. This equips the algorithm with a free polynomial reduction by a cyclotomic polynomial [15]. Finding these roots is a complex computational task usually performed by brute force when p is sufficiently small. Otherwise, numerical methods may be used. We offer in Appendix C a suggestion for their construction.

Algorithm 1: Polynomial multiplication in $\mathbb{Z}_p[x]/(x^n+1)$ via DGT

```
Input: Polynomials $a, b \in \mathbb{Z}_p[x]/(x^n+1)$, $p$ a prime number, $n$ a power-of-two
       integer, and $h$ a primitive $\frac{n}{2}$-th root of $i$ modulo $p$.
Output: $c = a \cdot b \in \mathbb{Z}_p[x]/(x^n+1)$.
1  for $j = 0; j < n/2; j = j + 1$ do
2      $a'_j = a_j + ia_{j+n/2}$                    // Folding the input polynomials
3      $b'_j = b_j + ib_{j+n/2}$
4  for $j = 0; j < n/2; j = j + 1$ do
5      $a'_j = h^j \cdot a'_j \pmod p$        // Applying the right-angle convolution
6      $b'_j = h^j \cdot b'_j \pmod p$
7  $a' = \mathrm{DGT}(a')$                          // Computing the DGT of both operands
8  $b' = \mathrm{DGT}(b')$
9  for $j = 0; j < n/2; j = j + 1$ do
10     $c'_j = a'_j \cdot b'_j \pmod p$              // Component-wise multiplying in $\mathbb{Z}_p[i]$
11 $c' = \mathrm{IDGT}(c')$       // Computing the IDGT of the multiplication result
12 for $j = 0; j < n/2; j = j + 1$ do
13     $u = h^{-j} \cdot c'_j \pmod p$                    // Removing the twisting factors
14     $c_j = u_{re}$                               // Unfolding the output polynomial
15     $c_{j+\frac{n}{2}} = u_{im}$
16 return $c$
```

There is no need for the bit-reversal procedure in the context of implementing a polynomial multiplication. Thus, an efficient implementation avoids it by selecting a decimation-in-frequency (DIF) algorithm for the forward transform and a decimation-in-time (DIT) algorithm for the inverse, as defined by Chu and George [13]. At this work, we follow the proposal of Badawi et al. and choose the Gentleman-Sande, a DIF, and the Cooley-Tukey, a DIT, data-paths for the forward and inverse versions of the DGT, respectively [3].

The canonical formulation of these contains a combination of three nested loops, which increases the complexity of its implementation, especially on the CUDA architecture. This structure creates dependencies between the loops and disturbs parallel execution. So, for better compatibility with the programming model, they have to be rewritten by wiping out one layer of nesting and leaving only two loops, an outer loop related to the stride and an inner loop that asserts the access patterns. For each outer loop iteration, the inner one can be completely parallelized. Our proposals for these have a much weaker dependency between iterations and can be seen in Algorithms 2 and 3.

Algorithm 2: Rewritten forward DGT via Gentleman-Sande

```
Input: A folded vector $x \in \mathbb{Z}[i]^k$, $p$ a prime number, $k$ a power-of-two integer,
       and $g$ a primitive $k$-th root of unity modulo $p$.
Output: $x \leftarrow \mathrm{DGT}(x)$ in bit-reversed ordering.
1 for $s = 0; s < \lfloor \log(k) \rfloor; s = s + 1$ do
2     $m = \frac{k}{2^{(s+1)}}$
3     for $l = 0; l < k/2; l = l + 1$ do
4         $j = \frac{2ml}{k}$
5         $i = j + \left(l \mod \frac{k}{2m}\right) \cdot 2m$
6         $a = g^{j \cdot \frac{k}{2^{(\log(k) - s)}}} \pmod p$
7         $(u, v) = (x[i], x[i + m])$
8         $(x[i], x[i + m]) = (u + v, a \cdot (u - v)) \pmod p$
9 return $x$
```

Algorithm 3: Rewritten inverse DGT via Cooley-Tukey

```
Input: A vector $x \in \mathbb{Z}[i]^k$ in bit-reversed order, $p$ a prime number, $k$ a
       power-of-two integer, and $g$ a primitive $k$-th root of unity modulo $p$.
Output: $x \leftarrow \mathrm{IDGT}(x)$ in standard ordering.
1 $m = 1$
2 for $s = 0; s < \lfloor \log(k) \rfloor; s = s + 1$ do
3     for $l = 0; l < k/2; l = l + 1$ do
4         $j = \frac{2ml}{k}$
5         $i = j + \left(l \mod \frac{k}{2m}\right) \cdot 2m$
6         $a = g^{-j \cdot \frac{k}{2^{s+1}}} \pmod p$
7         $(u, v) = (x[i], x[i + m])$
8         $(x[i], x[i + m]) = (u + a \cdot v, u - a \cdot v) \pmod p$
9     $m = 2 \cdot m$
10 return $x \cdot k^{-1} \pmod p$
```

3.2 An improved and hierarchical DGT

The procedures described in Algorithms 2 and 3 require synchronization at the end of each iteration of the outer loop. On CUDA, this enforces a limitation on the polynomial degree at the cost of latency, since the only data structure that provides such a level of synchronicity is Thread Blocks, and its dimension is limited to 1024 threads in modern hardware. An alternative implementation involves calling a different CUDA kernel for each iteration, imposing a CPU-sided synchronization. This incurs a considerable overhead caused by several kernel calls.

In this scenario, we propose a technique for splitting the DGT transform into smaller blocks that better fit the processing hardware and does not require synchronizing large sets of threads, called hierarchical DGT. It is an adaptation of the four-step FFT algorithm, initially proposed by David H. Bailey and later on revisited by Govindaraju et al. [5,21].

The general idea of the hierarchical DGT and hierarchical inverse DGT, referred to respectively as HDGT and HIDGT, is to split the DGT computation over $\mathbb{Z}_p[x]/(x^n+1)$ into computations in smaller rings with optimal degree near $\sqrt{n}$. In practice, the vector of coefficients is treated as a matrix and the DGT is performed over the columns and rows of this matrix. The objective of this is to avoid the case in which one is unable to compute the DGT of an entire polynomial in a single CUDA kernel call. We move to a higher granularity approach in which we apply the transform multiple times over arbitrary small polynomials that can perfectly fit in our processing architecture.

The HDGT is described in Algorithm 4. Firstly, the polynomial $a(x)$ is represented by taking its coefficient embedding as $a = (a_0, a_1, \ldots, a_{n-1})$. To be represented in the DGT domain $GF(p^2)$, $a \in \mathbb{Z}_p^n$ is folded as a $(n/2)$-size vector of Gaussian integers $\tilde{a} \in \mathbb{Z}_p[i]^{n/2}$, as described in Section 3.1. In the Algorithm, the "right-angle" convolution is given by multiplying the j-th coefficient of $\tilde{a}$ by h^j, for $j \in \mathbb{Z}_{n/2}$, where h is the $(n/2)$-th primitive root of i in $\mathbb{Z}_p[i]$.

After the folding and twisting procedures, the $(n/2)$-length vector of Gaussian integers $\tilde{a}$ is treated as a matrix with dimensions (N_r, N_c). These dimensions shall be chosen so that each coefficient's subset fits in the processing hardware. In our case, the objective is to find a subset that fits in the GPU's shared memory so that the DGT can be performed in a single Thread Block.

Since the bit-reversal is not used in Algorithm 2, the called "step-2" of Bailey's method has to be rewritten. In line 8, the twiddle factors are the powers of g, the $(n/2)$-th root of unity modulo p. Since the output of the DGT is not corrected from the bit-reversed order, the twiddle factors become $g^{\texttt{bit-reversal}(j)\cdot k}$ instead of $g^{j\cdot k}$, which matches the position of the corresponding element in $\tilde{a}$ when it is seen as a matrix.

The inverse counterpart of the hierarchical DGT simply executes the inverse steps of the forward transform, and is described in Algorithm 5. It adopts the IDGT transform via Cooley-Tukey, described in Algorithm 3, without bit-reversing the input vector. The algorithm executes the inverse steps of the forward transform by first applying the IDGT over the rows of $\tilde{a}$. The twiddle factors are removed by multiplying $\hat{a}_{j,k}$ by $g^{-\texttt{bit-reversal}(j)\cdot k}$, since the column indexes of the output of the previous step still are in bit-reversed order. Considering that the powers of g can be precomputed, they can be multiplied by N_c^{-1}, avoiding the additional multiplication. Finally, the IDGT is applied over the columns of $\hat{a}$ and the matrix indexes are back to standard ordering. Following the same approach, the powers of h^{-1} can be precomputed already multiplied by the scalar N_r^{-1}. This avoids the multiplication by the scaling factor when applying the IDGT over the columns of $\hat{a}$.

As in FFT and NTT, the two operands are evaluated using the HDGT for further point-wise multiplication. The polynomial corresponding to $a \cdot b$ in $\mathbb{Z}_p[x]/(x^n+1)$ is obtained by computing the HIDGT.

Algorithm 4: Hierarchical forward DGT

```
Input: A polynomial $a \in \mathbb{Z}_p[x]/(x^n+1)$, $p$ a prime number, $n = 2 \cdot N_r \cdot N_c$ a
       power-of-two integer, $h$ a primitive $n/2$-th root of $i$ modulo $p$, and $g$ a
       primitive $n/2$-th root of unity modulo $p$.
Output: $\tilde{a} = \mathrm{HDGT}(a)$.
1  for $j = 0; j < n/2; j = j+1$ do
2      $\tilde{a}_j = a_j + i a_{j+n/2}$                      // Fold the input polynomial
3      $\tilde{a}_j = \tilde{a}_j \cdot h^j \pmod{p}$          // Twist the folded polynomial
4  for $k = 0; k < N_c; k = k+1$ do
5      $\tilde{a}_{\_,k} = \mathrm{DGT}(\tilde{a}_{\_,k})$     // Step 1: Apply the DGT through $N_c$ columns
6  for $j = 0; j < N_r; j = j+1$ do
7      for $k = 0; k < N_c; k = k+1$ do
8          $\tilde{a}_{j,k} = \tilde{a}_{j,k} \cdot g^{\texttt{bit-reversal}(j)\cdot k} \pmod{p}$   // Step 2: Multiplication by
            the twiddle factors in bit-reversal order
9  for $j = 0; j < N_r; j = j+1$ do
10     $\tilde{a}_{j,\_} = \mathrm{DGT}(\tilde{a}_{j,\_})$     // Step 3: Apply the DGT through the $N_r$ rows
11 return $\tilde{a}$
```

Algorithm 5: Hierarchical inverse DGT

```
Input: $\tilde{a} = \mathrm{HDGT}(a)$, $p$ a prime number, $n = 2 \cdot N_r \cdot N_c$ a power-of-two
       integer, $h$ a primitive $n/2$-th root of $i$ modulo $p$, and $g$ a primitive
       $n/2$-th root of unity modulo $p$.
Output: A polynomial $a \in \mathbb{Z}_p[x]/(x^n+1)$.
1  for $j = 0; j < N_r; j = j+1$ do
2      $\hat{a}_{j,\_} = \mathrm{IDGT}(\tilde{a}_{j,\_})$      // Step 3: Apply IDGT to each of $N_r$ rows
3  for $j = 0; j < N_r; j = j+1$ do
4      for $k = 0; k < N_c; k = k+1$ do
5          $\hat{a}_{j,k} = \hat{a}_{j,k} \cdot g^{-\texttt{bit-reversal}(j)\cdot k} \cdot N_c^{-1} \pmod{p}$   // Step 2: Remove
            twiddle factors
6  for $k = 0; k < N_c; k = k+1$ do
7      $\hat{a}_{\_,k} = \mathrm{IDGT}(\hat{a}_{\_,k})$        // Step 1: Apply IDGT to each of $N_c$ columns
8  for $j = 0; j < n/2; j = j+1$ do
9      $\hat{a}_j = \hat{a}_j \cdot h^{-j} \cdot N_r^{-1} \pmod{p}$   // Remove the twisting
10     $a_j = \hat{a}_{j_{re}}$                                 // Unfold the output polynomial
11     $a_{j+\frac{n}{2}} = \hat{a}_{j_{im}}$
12 return $a$
```

3.3 Polynomial representation and memory locality

The usability of an RLWE-based cryptosystem requires the careful selection of a parameter set that satisfies all the security constraints of the application. For instance, with BFV one must select q, t, n, and σ such that a security level λ is achieved. More than that, these parameters together determine the multiplicative depth supported by the scheme. Thus, as discussed by Fan and

Vercauteren, the selection of such parameters is too complex to be affected by the particularities of the implementation [19].

A constraint for choosing those is the hardware instruction set. By selecting a big q one may be confronted by the lack of hardware support for native processing of the coefficients. Through RNS, as described in Section 2.2, we handle this by splitting big integers in small residues following the limits of the underlying machine.

The link between the cryptosystem and RNS must be carefully designed so that data secrecy is provided with suitable performance. For that, Gentry et al. suggested the *double*-CRT representation, which encapsulates data into two layers [20]. The first layer is the RNS representation, as described in Definition 1. After that, a set of polynomial residues with full support for native hardware evaluation of addition and multiplication is obtained. However, we still need a second layer for the latter, since the multiplication of polynomials can achieve a quite high computational complexity without some well-designed algorithm, as mentioned in Section 3.1. Because of that, the second layer consists of moving each residue, individually, to a different domain with a convenient property for efficient polynomial multiplication. The original proposal of *double*-CRT is the use of the NTT as this transform, but a similar approach using the FFT would also be expected. This work, however, proposes that the second layer of the *double*-CRT should use the DGT instead of the NTT, since the former appears to suit much better the cyclotomic ring arithmetic and uses memory in a more efficient way [3].

Another design decision, widespread to HE implementations, is the selection of a single special prime p for the transform and all RNS residues [16,18,3]. For instance, let x be a polynomial and $\{q_0, \ldots, q_{\ell-1}\}$ a set of ℓ pairwise coprimes, then $\left\{\mathrm{DGT}_p([x]_{q_0}), \ldots, \mathrm{DGT}_p([x]_{q_{\ell-1}})\right\}$ is the set of transformed residues. By using such a prime, one is capable of taking advantage of their intrinsic mathematical properties, as in the selection of a Mersenne or Solinas prime, which enables the use of a very efficient modular reduction. Nonetheless, this approach does not interplay well with the RNS layer and requires algorithmic efforts to correct these modular reductions and keep consistency for each residue. In this way, the *double*-CRT provides a simpler solution by computing the transform layer using the coprime related to each residue, at the cost of a more expensive modular reduction since, in most cases, there are not enough special primes for the required number of residues. Thus, in this representation, the set of residues becomes $\left\{\mathrm{DGT}_{q_0}([x]_{q_0}), \ldots, \mathrm{DGT}_{q_{\ell-1}}([x]_{q_{\ell-1}})\right\}$. Moreover, without the need for those corrections, we become capable of increasing RNS' residues to the biggest supported word size of the target architecture, reducing the number of residues needed. By choosing $q = \prod_{i=0}^{\ell-1} q_i$ we establish a bond between BFV, RNS, and the DGT.

Lastly, our state machine proposal targets the insistent maintenance of data in our version of the *double*-CRT representation in GPU's memory. Data copy between the main memory and the GPU's memory has high latency and must be avoided.

4 Experimental results

In this section we present SPOG[4], a proof-of-concept implementation that consolidates the aforementioned techniques by exploring parallel processing on GPGPUs through CUDA. Parts of the source code allowing reproducibility are in the process of being made available to the community.

Designed from scratch, SPOG is a modular implementation in which the arithmetic operations are separate from the cryptosystem. More precisely, the polynomial operations were implemented on a sister library named cuPoly, while BFV was implemented separated on SPOG. Both are implemented on top of CUDA and closely follow the sketch provided in Section 3, pursuing low-latency methods with a simple API and stretching the size of the residues to the highest supported by modern CUDA-supported GPUs, which is 63-bit residues with 1 bit for storing the sign. By doing this, we guarantee that BFV can be easily replaced by any other scheme based on the RLWE problem; thus, our work is not restricted to a single scheme. The entire arithmetic implementation can also be replaced without affecting the cryptosystem code. Hence, SPOG is flexible enough to encourage future work to develop and test different setups using the presented libraries.

cuRAND, a NVIDIA probabilistic library, was used for the sampling required by the BFV. This library offers sampling directly to the GPU memory, avoiding the cost of data copy. Sampling uniformly at random from R_q and R_2 is implemented through its uniform sampler, and the result is reduced by q or 2, respectively. On the other hand,the discrete Gaussian distribution is not supported by this library. Because of that, an alternative implementation works by truncating a normal distribution, natively supported by cuRAND. The statistical validity of this design still needs to be asserted at the cost of compromising the security. Moreover, to the best of our knowledge, cuRAND lacks sufficient scrutiny by the scientific community so that it can be seen as cryptographic secure. However, this is a common implementation decision in the literature and is also done by the related works cited in Section 4.1.

4.1 Related work

We consider Badawi, Polyakov, Aung, Veeravalli, and Rohloff, work, referred as BPAVR, the state-of-the-art implementation in GPUs for BFV [2]. It complements Halevi, Polyakov, and Shoup proposal and provides the first implementation of the HPS-BFV method on a high-end NVIDIA Tesla V100 GPU, demonstrated by the authors to be the fastest and most scalable variant of the scheme when compared to BEHZ-BFV [22,6].

BPAVR do not describe all details regarding their performance results, only presenting latency measurements for decryption and homomorphic multiplication. Because of that, and the fact of their source code is not publicly available, we also consider a similar work of Badawi, Veeravalli, Mun, and Aung,

[4] SPOG, acronym for “Secure Processing on GPGPUs”.

which offers timings for encryption, decryption, homomorphic addition, and homomorphic multiplication for a CUDA-based BFV implementation, denoted by BVMA[4]. The authors compare BVMA with Microsoft SEAL, a reference on the field with support for HPS-BFV [10]; and NFLlib-FV, an equally important work implementing the BEHZ-BFV variant; with impressive speedups on all scenarios [25]. Despite of their efforts for parallel computation, the other libraries presented in that work are CPU-based implementations and thus show a significant slowdown, up to 27 times, when compared to BVMA. Hence, we do not believe that the direct comparison with SPOG is relevant to this paper.

Lastly, both works apply the DGT as the underlying solution to handle polynomial multiplication. So, by comparing SPOG with them, we can collect evidence about the suitability of the HDGT over the DGT for such task.

4.2 Execution environment, methodology, and BFV parameters

The experimental results presented in the next Sections for BPAVR or BVMA are those reported by the authors in their corresponding publications. We do not re-execute the benchmarks provided in the related work. This decision is based on the fact that the implementations and benchmarking tools were not made available to the community. Because of that, we decided to collect our measurements in a similar processing hardware adopted in the related works using the same parameters.

We used Google Cloud's virtual machines (VMs) for emulating the computational environment described in those works. Two instances were considered: *gc.k80* and *gc.v100*, which provide a NVIDIA Tesla K80 GPU, used on BVMA measurements; and a NVIDIA Tesla V100 GPU, used on BPAVR. We precisely followed the execution environment described in each work, running GCC 7.2.1 and CUDA 8.0 at *gc.k80*; and GCC 7.3.1 and CUDA 9.0 at *gc.v100*. CUDA events were used to measure execution time, following the common methodology from the literature.

Our benchmark targets the most relevant primitives for HE. Regarding BFV, implemented in SPOG, we consider encryption, decryption, homomorphic addition, and homomorphic multiplication (including the relinearization cost). On the polynomial arithmetic side, implemented in cuPoly, we focus on the performance gains caused by the replacement of the canonical DGT by the HDGT.

In our measurements, we do not include initialization steps, which are performed only once and have negligible effect on long term runs. Because of that, the latency for generating cryptographic keys is not described in this work. Similarly, sampling is not explicitly considered in the benchmarks, despite of being included in the timings for encryption.

Two different setups are considered for compatibility with each work, both choosing $t = 256$ for the plaintext domain.

BPAVR parameters: Different polynomial ring settings are used identified by the pairs $(q, \log(n)) \in \{(60, 11), (60, 12), (120, 13), (360, 14), (600, 15)\}$ for the ciphertext coefficient domain and the ring degree, respectively. These offer a security level of at least 128 bits [2].

BVMA parameters: Different polynomial ring settings are used identified by the pairs $(q, \log(n)) \in \{(62, 11), (186, 12), (372, 13), (744, 14), (744, 15)\}$ for the ciphertext coefficient domain and the ring degree, respectively. These offer a security level of 80 bits [4].

4.3 Memory consumption

Let $\hat{q}$ and $\hat{b}$ be the main and auxiliary RNS bases used to represent elements of R_q and used by the HPS-BFV methods described in Section 2.3, respectively; and $\texttt{nres}_{qb}$ the quantity of elements in $\hat{q} \cup \hat{b}$. A BFV ciphertext on SPOG is composed by two N-degree polynomials represented as $\texttt{nres}_{qb}$ residues with 63-bits coefficients, thus requiring $s(N, \texttt{nres}_{qb}) := 63 \cdot (2 \cdot N \cdot \texttt{nres}_{qb})$ bits for storage.

The ciphertext expansion factor, however, depends also on its slot occupancy. Through batching, a single ciphertext can store up to N integer plaintexts [9]. Hence, the expansion factor is given by $\frac{s(N, \texttt{nres}_{qb})}{63 \cdot \texttt{batch\_size}}$.

4.4 SPOG operations

In Table 1 we compare SPOG with BVMA on *gc.k80*, and with BPAVR on *gc.v100*. As mentioned in Section 4.1, The authors of BPAVRoffer measurements for decryption and homomorphic multiplication only, what inhibits the comparison with SPOG for encryption and homomorphic addition.

One of the major motivations for using a FHE scheme is the applicability of its homomorphic primitives, and because of that, we focus on improving the performance of these. As can be seen, homomorphic multiplication, a critical and known expensive operation, reports speedup between 2.0 and 3.6 times when compared to the BVMA. When compared to the BPAVR these speedups lies between 2 and 2.4. The different characteristics between both setups, considering the processing hardware and the cryptosystem parameters, makes the direct comparison between both data sets impossible, however the performance gains are consistent.

Homomorphic addition, a much simpler operation, presented gains between 2 and 5.2 times when compared to the BVMA. The latter is probably not related to the HDGT, since this procedure is essentially a coefficient-wise addition, but to the better state machine our version of the *double*-CRT offers, as described at Section 3.3.

Despite our focus in this work does not being on encryption and decryption, the faster polynomial multiplication strategy and the improved state machine offered up to 4.6 times faster encryption and about 2 times faster decryption.

4.5 Efficiency of the HDGT

A major contribution of this work is the HDGT, a novel formulation of the DGT which better explores the parallel capability of GPUs and compensate its

Table 1. Comparison between SPOG and two state-of-the-art implementations, BVMA and BPAVR. Average running time of 100 independent executions, in milliseconds, for the most relevant BFV operations on *gc.k80* and *gc.v100* virtual machines for the setups described in Section 4.2.

		gc.k80				*gc.v100*				
	$\log n$	11	12	13	14	$\log n$	12	13	14	15
Encrypt	SPOG	0.303	0.309	0.575	1.630	-	-	-	-	-
	BVMA	0.541	1.440	2.645	6.657	-	-	-	-	-
	Ratio	**1.785**	**4.660**	**4.600**	**4.084**	-	-	-	-	-
Decrypt	SPOG	0.089	0.098	0.191	0.542	SPOG	0.029	0.031	0.049	0.099
	BVMA	0.151	0.194	0.252	0.610	BPAVR	0.054	0.059	0.087	0.111
	Ratio	**1.697**	**1.980**	**1.319**	**1.125**	Ratio	**1.862**	**1.903**	**1.776**	**1.121**
Hom. Add.	SPOG	0.009	0.010	0.021	0.066	-	-	-	-	-
	BVMA	0.037	0.052	0.068	0.127	-	-	-	-	-
	Ratio	**4.111**	**5.200**	**3.238**	**1.924**	-	-	-	-	-
Hom. Mul.	SPOG	0.926	1.214	3.061	13.914	SPOG	0.423	0.472	0.823	2.325
	BVMA	3.343	3.873	7.700	28.953	BPAVR	0.859	1.012	2.010	4.826
	Ratio	**3.610**	**3.190**	**2.516**	**2.081**	Ratio	**2.031**	**2.144**	**2.442**	**2.076**

memory limitations. However, a carefully evaluation of its quality must be done to understand the performance gains on realistic scenarios. Thus, at this Section, we provide a comparison between the HDGT and the best implementation designs for the canonical DGT.

As discussed before, the HDGT works by splitting a high-degree polynomial, which does not fit in the processing hardware, and applying the DGT in a divide-and-conquer approach through blocks of arbitrarily small size. To evaluate this design, we implemented the canonical DGT adopting two different strategies, namely DGT-I and DGT-II. The former uses a multi-kernel design which executes the loop synchronization employing a different CUDA kernel for each iteration. This way, the transformation requires $\log \frac{n}{2}$ kernels to process an n-degree polynomial. The latter uses a single-kernel design, which is only compatible with polynomial rings with degree smaller or equal than 4096 since these are the only that fit GPU's shared memory. These strategies are better described in Section 3.2. Lastly, we verified the impact of this change in two important procedures direct affected by the DGT, encryption and homomorphic multiplication.

Table 2 presents the latency measurements. The HDGT is about 2 times faster than the DGT-I, which results in speedups ranging from 1.4 to 2.2 times on BFV's primitives. The DGT-II, though, presents a slowdown in most cases. This relates to the need for serialization within HDGT's steps, which was implemented by splitting the algorithm into 4 sequential kernels. DGT-II is always executed by a single kernel, implying a much smaller overhead. This suggests that the single-kernel design better accommodates smaller instances. Such effect doesn't sustain on *gc.v100* that better handles the high-granularity of the HDGT. No other comparison is feasible with the DGT-II since this model is not scalable to bigger rings.

Table 2. Comparison between SPOG running the canonical DGT using a multi-kernel and a single-kernel strategy, called DGT-I and DGT-II, respectively; and the HDGT. The first row group compares the transform alone. Average running time of 100 independent executions, in milliseconds, on *gc.k80* and *gc.v100* virtual machines for the setups described in Section 4.2.

		gc.k80					*gc.v100*				
	$\log n$	11	12	13	14	15	11	12	13	14	15
DGT	HDGT	0.059	0.071	0.146	0.432	0.651	0.018	0.019	0.020	0.031	0.073
	DGT-I	0.114	0.131	0.281	0.711	1.637	0.035	0.034	0.040	0.078	0.188
	Ratio	**1.934**	**1.864**	**1.925**	**1.644**	**2.517**	**1.934**	**1.815**	**2.040**	**2.487**	**2.593**
	DGT-II	0.052	0.091	-	-	-	0.026	0.047	-	-	-
	Ratio	0.881	**1.292**	-	-	-	**1.423**	**2.492**	-	-	-
Encrypt	HDGT	0.303	0.309	0.575	1.630	3.127	0.103	0.098	0.099	0.153	0.315
	DGT-I	0.571	0.499	0.861	2.597	5.835	0.144	0.146	0.159	0.287	0.704
	Ratio	**1.882**	**1.614**	**1.499**	**1.593**	**1.866**	**1.395**	**1.498**	**1.615**	**1.883**	**2.238**
	DGT-II	0.276	0.377	-	-	-	0.120	0.188	-	-	-
	Ratio	0.910	**1.220**	-	-	-	**1.163**	**1.921**	-	-	-
Hom. Mult.	HDGT	0.926	1.214	3.061	13.914	28.990	0.436	0.423	0.472	0.823	2.325
	DGT-I	1.795	2.031	4.231	19.952	42.800	0.795	0.783	0.913	1.609	4.078
	Ratio	**1.938**	**1.673**	**1.382**	**1.434**	**1.476**	**1.825**	**1.850**	**1.934**	**1.956**	**1.754**
	DGT-II	0.642	0.983	-	-	-	0.362	0.466	-	-	-
	Ratio	0.693	0.810	-	-	-	0.830	**1.102**	-	-	-

5 Conclusion

This work investigates strategies to achieve an efficient implementation of the leveled homomorphic encryption scheme BFV on the CUDA architecture. To fulfill this objective, we explored different approaches for the utilization of the DGT in the reduction of the computational complexity of polynomial multiplications. The outcome is an optimized version of the hierarchical DGT, a high granularity implementation of DGT that better fits the GPU processing. Furthermore, the *double*-CRT concept is revisited and an efficient state machine is proposed so we can avoid the costs to alternate between DGT and RNS domains, and between the machine's main memory and GPU's memory.

Our implementation of BFV, named SPOG, is compared with two other works in the literature, BVMA and BPAVR, that represent the state-of-the-art implementations on CUDA. Homomorphic addition, in spite of being a simple and usually fast operation, presented speedups between 2 and 5.2 times over the BVMA. Furthermore, SPOG's homomorphic multiplication showed itself between 2.0 and 3.6 times faster over the BVMA.

As future work, we intend to verify the gains of applying our methods on other relevant RLWE-based cryptosystems such as the CKKS [11], and SPOG as a tool for the acceleration of privacy-focused deep learning algorithms.

Acknowledgements

This work was supported in part by CNPq, grants number 164489/2018-5, 203175/2019-0, and 44265/2019-2; CAPES grant number 1591123. We specially thank LG for financial support within project "*Privacy-preserving analytics*", project number 5296; and Google for GCP Research Credits Program under number 106101194491.

References

1. Albrecht, M., Bai, S., Ducas, L.: A Subfield Lattice Attack on Overstretched NTRU Assumptions. In: Robshaw, M., Katz, J. (eds.) Advances in Cryptology – CRYPTO 2016. pp. 153–178. Springer Berlin Heidelberg, Berlin, Heidelberg (2016)
2. Badawi, A.A., Polyakov, Y., Aung, K.M.M., Veeravalli, B., Rohloff, K.: Implementation and performance evaluation of RNS variants of the BFV homomorphic encryption scheme. IACR Cryptol. ePrint Arch. **2018**, 589 (2018)
3. Badawi, A.Q.A., Veeravalli, B., Aung, K.M.M.: Efficient Polynomial Multiplication via Modified Discrete Galois Transform and Negacyclic Convolution. In: AISC. vol. 886, pp. 666–682. Springer, Cham (2019)
4. Badawi, A.Q.A., Veeravalli, B., Mun, C.F., Aung, K.M.M.: High-Performance FV Somewhat Homomorphic Encryption on GPUs: An Implementation using GPUs. TCHES **1**(2), 70–95 (2018)
5. Bailey, D.H.: FFTs in external or hierarchical memory. J. Supercomput. **4**(1), 23–35 (1990)
6. Bajard, J., Eynard, J., Hasan, M.A., Zucca, V.: A full RNS variant of FV like somewhat homomorphic encryption schemes. In: SAC. Lecture Notes in Computer Science, vol. 10532, pp. 423–442. Springer (2016)
7. Bajard, J.C.J., Meloni, N., Plantard, T.: Efficient RNS bases for Cryptography. IMACS World Congress: Scientific Computation, Applied Mathematics and Simulation (2005)
8. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (Leveled) Fully Homomorphic Encryption without Bootstrapping. ACM Trans. Comput. Theory **6**(3), 13:1–13:36 (2014)
9. Chen, H., Gilad-Bachrach, R., Han, K., Huang, Z., Jalali, A., Laine, K., Lauter, K.E.: Logistic regression over encrypted data from fully homomorphic encryption. IACR Cryptol. ePrint Arch. **2018**, 462 (2018)
10. Chen, H., Laine, K., Player, R.: Simple encrypted arithmetic library - SEAL v2.1. IACR Cryptol. ePrint Arch. **2017**, 224 (2017)
11. Cheon, J.H., Kim, A., Kim, M., Song, Y.S.: Homomorphic encryption for arithmetic of approximate numbers. In: ASIACRYPT (1). Lecture Notes in Computer Science, vol. 10624, pp. 409–437. Springer (2017)
12. Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: TFHE: fast fully homomorphic encryption over the torus. J. Cryptol. **33**(1), 34–91 (2020)
13. Chu, E., George, A.: Inside the FFT black box: serial and parallel fast Fourier transform algorithms. CRC press (1999)
14. Costache, A., Smart, N.P.: Which ring based somewhat homomorphic encryption scheme is best? In: CT-RSA. Lecture Notes in Computer Science, vol. 9610, pp. 325–340. Springer (2016)

15. Crandall, R.E.: Integer convolution via split-radix fast Galois transform. Center for Advanced Computation Reed College (1999)
16. Dai, W., Sunar, B.: cuHE: A Homomorphic Encryption Accelerator Library. In: BalkanCryptSec. Lecture Notes in Computer Science, vol. 9540, pp. 169–186. Springer (2015)
17. Ding, C., Pei, D., Salomaa, A.: Chinese remainder theorem: applications in computing, coding, cryptography. World Scientific (1996)
18. Emmart, N., Weems, C.C.: High precision integer multiplication with a GPU using strassen's algorithm with multiple FFT sizes. Parallel Process. Lett. **21**(3), 359–375 (2011)
19. Fan, J., Vercauteren, F.: Somewhat practical fully homomorphic encryption. IACR Cryptol. ePrint Arch. **2012**, 144 (2012)
20. Gentry, C., Halevi, S., Smart, N.P.: Homomorphic evaluation of the AES circuit. In: CRYPTO. Lecture Notes in Computer Science, vol. 7417, pp. 850–867. Springer (2012)
21. Govindaraju, N.K., Lloyd, B., Dotsenko, Y., Smith, B., Manferdelli, J.: High performance discrete fourier transforms on graphics processors. In: SC. p. 2. IEEE/ACM (2008)
22. Halevi, S., Polyakov, Y., Shoup, V.: An Improved RNS Variant of the BFV Homomorphic Encryption Scheme. In: CT-RSA. Lecture Notes in Computer Science, vol. 11405, pp. 83–105. Springer (2019)
23. Lindner, R., Peikert, C.: Better key sizes (and attacks) for lwe-based encryption. In: CT-RSA. Lecture Notes in Computer Science, vol. 6558, pp. 319–339. Springer (2011)
24. Longa, P., Naehrig, M.: Speeding up the number theoretic transform for faster ideal lattice-based cryptography. In: CANS. Lecture Notes in Computer Science, vol. 10052, pp. 124–139 (2016)
25. Melchor, C.A., Barrier, J., Guelton, S., Guinet, A., Killijian, M., Lepoint, T.: NFLlib: NTT-Based Fast Lattice Library. In: CT-RSA. Lecture Notes in Computer Science, vol. 9610, pp. 341–356. Springer (2016)
26. Player, R.: Parameter selection in lattice-based cryptography. Ph.D. thesis, PhD thesis, Royal Holloway, University of London (2018)
27. Thales: 2019 Thales Data Threat Report. https://go.thalesesecurity.com/rs/480-LWA-970/images/2019-DTR-Global-USL-Web.pdf, USA (2019)
28. Wuthrich, C.: Further Number Theory. `https://www.maths.nottingham.ac.uk/plp/pmzcw/download/fnt_chap5.pdf` (2011), last accessed: 2020/06/18

A Gaussian integers

The set of Gaussian integers can be used to represent elements of $GF(p^2)$, that is $\mathbb{Z}_p[i] = \{a + ib \mid a, b \in \mathbb{Z}_p\}$, for $i = \sqrt{-1}$. Arithmetic in $\mathbb{Z}_p[i]$ is similar to complex number arithmetic with a reduction modulo p for the real and imaginary parts.

Arithmetic Let $a, b \in GF(p^2)$. The main operations can be defined as follows:

$$\begin{aligned}
add(a,b) &= (a_{re} + b_{re}) + i(a_{im} + b_{im}) \mod p\\
sub(a,b) &= (a_{re} - b_{re}) + i(a_{im} - b_{im}) \mod p\\
mul(a,b) &= (a_{re}b_{re} - a_{im}b_{im}) + i(a_{re}b_{im} + a_{im}b_{re}) \mod p\\
div(a,b) &= \left(a \cdot \overline{b}\right) \cdot \left(b_{re}^2 + b_{im}^2\right)^{-1} \mod p\\
rem(a,b) &= a - (a/b) \cdot b \mod p
\end{aligned}$$

B Properties of Gaussian integers

This Appendix presents important properties of Gaussian integers and useful results that can be applied on their implementation. In the following, we recall some important properties stated by Wuthrich that are useful to this work [28].

Definition 3 (Norm). *The norm of a Gaussian integer is defined as its product with its conjugate*[5]*. That is,* $N(a+ib) = (a+ib)\cdot(a-ib) = a^2+b^2$*, so* $N(\alpha) = \alpha\cdot\overline{\alpha}$*.*

Proposition 1 (Wuthrich's Proposition 5.7). *For each prime number* $p \equiv 1 \mod 4$ *there are exactly two Gaussian primes* π *and* $\overline{\pi}$ *of norm* p*.*

Lemma 1 (Wuthrich's Lemma 5.4). *If* $\pi \in \mathbb{Z}[i]$ *is such that* $N(\pi)$ *is a prime number, then* π *is a Gaussian prime.*

Lemma 2 (Wuthrich's Lemma 5.6). *Let* p *be a prime number with* $p \equiv 1 \mod 4$*. Then there exists a Gaussian prime* π *such that* $p = \pi.\overline{\pi}$*.*

Lemma 3 (Wuthrich's Lemma 5.10). *Any prime* $p \equiv 1 \mod 4$ *can be written as a sum of two squares. This is a manifestation of Fermat's theorem on sums of two squares.*

From Lemma 2 and Proposition 1, if p is prime such that $p \equiv 1 \mod 4$, then we know that it can be factored as a product of exactly two Gaussian primes that are the conjugate of each other. Lemma 3 is a direct consequence since we know that a prime $p \equiv 1 \mod 4$ can be factored as $p = \pi \cdot \overline{\pi}$ and, assuming that $\pi = a + bi$, we obtain that $\pi \cdot \overline{\pi} = a^2 + b^2$.

C Generating k-th primitive roots of i modulo p

The use of the DGT for polynomial multiplication in a polynomial ring modulo $x^n + 1$ requires the computation of a k-th root of i modulo a prime p, discussed in Section 3.1. This element is used for achieving a cyclotomic polynomial reduction for free when n is a power of two. When p is a Mersenne prime, the literature presents efficient analytic methods; for other choices of p, the best option still is a trial-and-error approach.

[5] Let $x = a + ib$ be a Gaussian integer. If y is x's conjugate then $y = a - ib$.

Badawi et al. state that a naive implementation of such approach takes 156 hours to find a 2^{14}-th primitive root of i for $p = 2^{64} - 2^{32} + 1$ in a highly optimized Mathematica script [3]. Because of that, they propose a more efficient strategy, when $p \equiv 1 \mod 4$, by factoring p in two Gaussian primes, namely f_0 and f_1. This decomposition of p is quite simple and relies on Lemma 2 and Proposition 1.

Algorithm 6: `decompose_in_gaussian_primes`: Returns elements f_0 and f_1 such that $f_0 \cdot f_1 = p$.

```
Input: A prime $p$
Output: Gaussian integers $f_0$ and $f_1$ such that $f_0 \cdot f_1 = p$
1 do
2     $n = \texttt{sample}(\mathbb{Z}_p)$
3 while $n^{(p-1)/2} \not\equiv -1 \mod p$
4 $k = n^{(p-1)/4} \mod p$
5 $u = \texttt{gcd}(p, k+i)$
6 return $(f_0, f_1) = (u, \overline{u})$
```

Algorithm 6 starts from the Fermat's Little Theorem, which states that if p is a prime then $n^{p-1} \equiv 1 \mod p$ for all $n \in \mathbb{Z}_p$. Hence, the square root of that must be equivalent to either 1 or -1. In the latter case, we can find a number k^2 such that $k \equiv n^{(p-1)/4} \equiv i \mod p$. In other words, if $k^2 \equiv -1 \mod p$ then $k^2 + 1 \equiv 0 \mod p$ and p divides $k^2 + 1$. Since $k^2 + 1$ factors in $(k+i) \cdot (k-i)$, we found a factorization of p.

At this point, there is no guarantee that $k + i$ is a Gaussian prime. By Lemma 4, we find that the greatest common divisor of p and $k + i$ is either $k + i$ or that there exists some u such that $u \mid p$ and $u \mid k + i$. Thus, since $u = \texttt{gcd}(p, k+i)$ results in a Gaussian prime, we take it as the first factor of p. From Lemma 2, $\overline{u}$ is the second factor.

Lemma 4. *Let p be an odd prime such that $p \equiv 1 \mod 4$ and $k \in \mathbb{Z}_p$. The greatest common divisor of p and $k+i$ is $k+i$ or a Gaussian prime u such that $u \mid p$ and $u \mid k+i$.*

Proof. By the Fermat's theorem on sums of two squares, we have that an odd prime p can be expressed as $p = x^2 + y^2$, with $x, y \in \mathbb{Z}$, if, and only if, $p \equiv 1 \mod 4$. Since $x^2 + y^2 = (x+iy)(x-iy)$ and $N(x+iy) = N(x-iy) = p$, then $x+iy$ and $x-iy$ are Gaussian primes and $p = (x+iy)(x-iy)$ is the unique factorization of p in $\mathbb{Z}[i]$, not considering the order of the factors[6].

On the other hand, we have that $(k+i)(k-i) \equiv p \mod p$, by construction. Combining the two facts, we obtain that $p = (x+iy)(x-iy) \equiv (k+i)(k-i)$, which is equivalent to $(k+i)(k-i) = \ell(x+iy)(x-iy)$, for some $\ell \in \mathbb{Z}$.

[6] Wuthrich proves in Theorem 5.8 that every $0 \neq \alpha \in \mathbb{Z}[i]$ has a unique factorization [28].

When $\ell = 1$, we have an equality and we find that $(k+i)$ and $(k-i)$ are indeed the factors of p. When $\ell \neq 1$, $(k+i)$ is not a Gaussian prime and still can be factored in $\mathbb{Z}[i]$; otherwise, it would be a factor of p. We know that p divides $(k+i)(k-i)$ but not $k+i$, or its conjugate, since $k < p$ and $(k+i)/p$ is not a Gaussian integer. Then, $k+i$ and p must share a common factor u that can be found as the greatest common divisor. Since the two factors of p are $x+iy$ and $x+iy$, u must be one of them.

Finally, the factors of p can be found by computing the greatest common divisor of p and $k+i$ and then computing its conjugate. Since $p = x^2 + y^2$ and $N(x+iy) = N(x-iy) = x^2+y^2$, by Lemma 1, the factors are Gaussian primes.

Given a method for factoring a prime number $p \equiv 1 \mod 4$ in $\mathbb{Z}[i]$, Badawi et al. propose Algorithm 7, which makes much faster the step of precomputing a k-th root of i for a prime $p \equiv 1 \mod 4$ [3]. The method starts by finding the factorization $p = f_0 \cdot f_1 \in \mathbb{Z}_p[i]$ using the Algorithm 6.

At this point, we have that each Gaussian prime f_j, with $j = \{0,1\}$, defines a cyclic group corresponding to the set of Gaussian integers modulo f_j. Then, a k-th root of i modulo p, denoted as h, is constructed via CRT using that $h_j = \zeta_j^{\frac{(p-1)}{4n}} \mod f_j$, with $j = \{0,1\}$, where ζ_j is a generator for the cyclic group j.

Algorithm 7: Compute the k-th primitive root of $i \mod p$, for a prime number $p \equiv 1 \mod 4$.

```
   Input: An integer $k$ and a prime $p \equiv 1 \mod 4$.
   Output: The $k$-th primitive root of $i \mod p$.
1  $f_0, f_1 = \texttt{decompose\_in\_gaussian\_primes}(p)$
2  do
3      for $j = 0; j < 2; j = j + 1$ do
4          $\zeta_j = \texttt{sample\_generator}(f_j)$
5          $h_j = \zeta_j^{\lfloor (p-1)/(4k) \rfloor} \mod f_j$
6      $h = f_1 \cdot \left(f_1^{-1} \cdot h_0 \mod f_0\right) + f_0 \cdot \left(f_0^{-1} \cdot h_1 \mod f_1\right) \mod p$
7      if $h^k \equiv i \mod p$ then
8          return $h$
9  while True
```