Exploring the Advantages and Challenges of Fermat NTT in FHE Acceleration

Andrey Kim[1] †, Ahmet Can Mert[2] †, Anisha Mukherjee[2], Aikata Aikata[2], Maxim Deryabin[1], Sunmin Kwon[1], HyungChul Kang[1], and Sujoy Sinha Roy[2]

[1] Samsung Advanced Institute of Technology, Suwon, Republic of Korea
{andrey.kim,max.deriabin,sunmin7.kwon,hc1803.kang}@samsung.com
[2] Graz University of Technology, Graz, Austria
{ahmet.mert,anisha.mukherjee,aikata,sujoy.sinharoy}@iaik.tugraz.at

Abstract. Recognizing the importance of a fast and resource-efficient polynomial multiplication in homomorphic encryption, in this paper, we introduce a novel method that enables *integer multiplier-less* Number Theoretic Transform (NTT) for computing polynomial multiplication. First, we use a Fermat number as an auxiliary modulus of NTT. However, this approach of using Fermat number scales poorly with the degree of polynomial. Hence, we propose a transformation of a large-degree univariate polynomial into small-degree multi-variable polynomials. After that, we compute these NTTs on small-degree polynomials with Fermat number as modulus. We design an accelerator architecture customized for the novel multivariate NTT and use it for benchmarking practical homomorphic encryption applications. The accelerator can achieve 1,200× speed-up compared to software implementations. We further discuss the potential and limitations of the proposed polynomial multiplication method in the context of homomorphic encryption.

Keywords: Fermat number· NTT · Polynomial Multiplier · FHE · Hardware Accelerator

1 Introduction

Homomorphic encryption refers to a class of encryption algorithms that enable computations to be performed directly on encrypted data, eliminating the necessity for decryption. It has emerged as a prominent cryptographic algorithm for safeguarding sensitive data when data processing occurs within untrusted environments. One notable example of the application of homomorphic encryption is the privacy-preserving outsourcing of computation to a cloud, where the cloud evaluates a specific procedure (e.g., statistical analysis, machine learning, etc.) homomorphically on the encrypted data and sends the encrypted result to the user. Other applications where homomorphic encryption is used as a foundation include privacy-preserving Distributed Learning [21], Machine Learning as a Service (MLaaS) [25] and Private Set Intersection [10].

† Andrey Kim and Ahmet Can Mert declare equal contribution

The first Fully Homomorphic Encryption (FHE) scheme, which could perform arbitrary operations on encrypted data, was constructed by Gentry [20] using lattice-based cryptography. Although this construction was a breakthrough answer to a decades-old open problem [36], the first FHE scheme was extremely slow in processing ciphertexts. In the following years, several other FHE schemes [8, 9, 7, 12] were constructed relying on the hardness of solving the Learning with Errors (LWE) or its structured variant, the Ring Learning with Errors (RLWE) problems. In the present-day scenario, FHE schemes based on the RLWE problem are leading in computational efficiency.

RLWE-based homomorphic evaluation procedures perform computations in a polynomial ring $R_Q = \mathbb{Z}_Q/\langle f(x)\rangle$, where Q is a ciphertext modulus and $f(x)$ is an irreducible polynomial of the polynomial ring. The degree of f and the size of Q depend on the application's security level and multiplicative complexity that needs to be evaluated homomorphically. For example, the evaluation of logistic regression training [28, 38, 3] uses $f(x) = x^{2^{16}} + 1$ and $\log_2 Q = 1,728$. The arithmetic of large polynomials is at the center of homomorphic evaluation, making homomorphic evaluations four to five orders of magnitude computationally demanding compared to simple plaintext processing.

Efficient implementations of RLWE-based homomorphic encryption schemes in the literature generally apply two primary algorithmic optimizations. First, they use a composite $Q = \prod_{i=0}^{l-1} q_i$, a product of l (co)primes q_i, and apply the Residue Number System (RNS) to transform modulo Q arithmetic into modulo q_i arithmetic. Hence, polynomial arithmetic in R_Q transforms into arithmetic of residue polynomials in R_{q_i}. This optimization improves efficiency by enabling parallel processing of small-integer arithmetic operations. The second optimization is the application of number theoretical transform (NTT) based polynomial multiplication. The NTT method has the fastest asymptotic time complexity of $\mathcal{O}(n \cdot \log n)$ elementary integer modular operations, where n is the degree of the irreducible polynomial of R_Q. The NTT in R_{q_i} requires the existence of n-th primitive root of the unity modulo q_i and to satisfy this requirement, the usual choice is $q_i \equiv 1 \mod n$ with q_i being a prime. When an additional condition, $q_i \equiv 1 \mod 2n$, is satisfied, modular reduction of a polynomial multiplication by $f(x) = x^{2^k} + 1$ with $n = 2^k$ becomes implicit. Any NTT primarily computes the so-called "butterfly" operation, which internally performs additions, subtractions, and multiplications respective to RNS-based moduli q_i. A typical approach for designing high-performance FHE accelerators is to make the modular multiplication fast and/or resource efficient.

In this paper, we set out to answer the question, *Can we make these modular multiplications extremely inexpensive and even cost-free?* In particular, we investigate the impact of replacing the RNS moduli q_i's with Fermat numbers during an NTT operation. NTT using Fermat numbers (FNTT) were explored five decades ago [1] where the authors showed that FNTT is less computationally demanding compared to the Fast Fourier Transform (FFT) because the complex multiplications in case of FFT get mapped to simple shift operations using FNTT. Furthermore, a generalized NTT using Fermat-Mersenne numbers

was suggested in [16]. However, there is no work in the literature that implements FNTT in the context of homomorphic encryption. The special properties of Fermat numbers and the existing research gap motivate us to design an FHE hardware accelerator based on FNTT. We also discuss in detail why this approach is not so straightforward when it comes to homomorphic encryption and list its advantages as well as certain limitations.

1.1 Our contributions

In this paper, we adopt a curiosity-driven approach to explore the potential applications of Fermat numbers in optimizing NTT-based polynomial multiplications for hardware acceleration of fully homomorphic encryption. Our contributions are outlined below.

- For the first time in the context of FHE, we design an NTT-based polynomial multiplication method that uses a Fermat number [1, 16] of the form $F_K = 2^K + 1$ as the modulus. Specifically, by employing a suitably large F_K as an auxiliary modulus and defining corresponding ring mappings between R_{q_i} and R_{F_K}, all NTTs are performed modulo F_K instead of the individual moduli q_i. As the roots of unity modulo F_K are powers-of-2, the FNTT method eliminates the storage of twiddle factors (roots of unity). It transforms all modular multiplications during NTT into simple shift operations leading to a *multiplier-less* NTT design. Furthermore, using the same modulus for computing all NTTs enables the design of a 'somewhat' flexible or parameter-independent polynomial multiplier circuit on hardware platforms.

- One problem with FNTT with respect to F_K is that it can utilize the power-of-two twiddle factors only for polynomial degrees $N \leq K$ (i.e., it supports upto K-point NTT), so K has to be very large to accommodate an efficient FNTT algorithm corresponding to the large polynomial degree required in a homomorphic encryption scheme. This in turn will result in an unsuitably large machine word size proportional to K. We provide a simple algorithmic solution to overcome this problem by proposing to switch from a univariate polynomial structure to a multivariate one. This allows the utilization of smaller-dimensional FNTTs instead of one inefficient large-dimensional FNTT in the univariate case.

- In order to quantitatively assess the implementation cost of multivariate-FNTT over other established methods, we design a hardware accelerator tailored to the technique proposed above. The FNTT unit has no multipliers and has relatively low routing and, consequently, power overhead. In terms of area, it is 50% cheaper compared to state-of-the-art FNTT design for FHE [3] offering the same throughput. The FNTT unit is bi-directional and can hence support both FNTT as well as Inverse FNTT (IFNTT).

– Finally, we accomplish our goal of designing a hardware accelerator utilizing FNTT. To this end, we propose an architecture tailored to optimize the new FNTT-based polynomial multiplication and Key Switch routines for the RNS CKKS [12] scheme. We observe that although the FNTT unit is much cheaper compared to conventional NTT units in the literature, the overall architecture consumes more space due to increased on-chip storage requirements. Even with this added overhead, it could achieve a speed-up of 1,200× compared to CPU computation.

The paper is organized as follows. In Sec. 2, we provide the mathematical details required to present our algorithmic and architectural contributions. Sec. 3 presents our proposed polynomial multiplication approach using a Fermat number as an auxiliary modulus and a multivariate ring. In Sec. 4, we give a detailed design description that we adopted to implement FNTT on hardware. Finally, in Sec. 5, we provide an instance of a CKKS accelerator that integrates our proposed FNTT architecture. Our implementational results and comparisons with other works are given in Sec. 6. Finally, Sec. 7 discusses the potentials and limitations of our FNTT-based homomorphic encryption accelerator design approach.

2 Preliminaries

2.1 Notation

We denote a ring of univariate polynomials by $R[X]$ and in case of several indeterminates $X_1, \cdots, X_k$, the multivariate ring is denoted by $R[X_1, \cdots, X_k]$. Let $N \in \mathbb{N}$ be a power of two. For a number field $\mathbb{Q}[X]/(\phi_{2N}(X))$ we denote $\mathcal{R} = \mathbb{Z}[X]/(\phi_{2N}(X))$ as its ring of integers consisting of polynomials modulo the $2N$-th cyclotomic polynomial, $\phi_{2N}(X) = X^N + 1$. Also, let $\mathcal{R}_q = \mathcal{R}/q\mathcal{R}$ be the residue ring of $\mathcal{R}$ modulo an integer q. An element of $\mathcal{R}_q$ is a polynomial of the form, $a(X) = \sum_{i=0}^{N-1} a_i X^i$ with each of its coefficients a_i in $\mathbb{Z}_q$. We will use the same notation format for the multivariate case as well, replacing X by X_i's. For $q \in \mathbb{Z}$ and $q > 1$, we identify the ring $\mathbb{Z}_q$ with $[-q/2, q/2)$ as the representative interval, and for $x \in \mathbb{Z}$ we denote the centered remainder of x modulo q by $[x]_q \in \mathbb{Z}_q$. We extend these notations to elements of $\mathcal{R}_q$ by applying them coefficient-wise. All logarithms are base 2 unless otherwise indicated.

2.2 NTT for polynomial multiplication

As discussed previously, HE schemes employ polynomials of high degrees and large coefficients which makes multiplications between them one of the primary implementation bottlenecks. NTT is usually chosen as the preferred algorithm because it reduces the complexity of polynomial multiplication to $\mathcal{O}(n \cdot \log n)$ compared to the naive schoolbook ($\mathcal{O}(n^2)$) or Karatsuba algorithms ($\mathcal{O}(n^{\log 3})$). The NTT is an extension of the Fourier Transform over finite fields of the form $\mathbb{Z}_p$

that allows replacing polynomial multiplication with element-wise integer multiplication. An N-point NTT operation transforms a polynomial $a(X)$ of degree $N-1$ into a polynomial $\tilde{a}$ obtained by evaluating a over the powers of N-th root of unity, ω_N in $\mathbb{Z}_p$ (often called 'twiddle factors'), that is, $\tilde{a}_k = \sum_{i=0}^{N-1} a_i(\omega_N^{ik})$ for every $k = 0, \cdots, N-1$. The inverse of this transformation is called the Inverse NTT (INTT). In polynomial rings of the form $\mathcal{R}_q$, an additional condition of $q \equiv 1 \pmod{2N}$ ensures an efficient NTT-based polynomial multiplication because of cheap modular reduction. Fig. 2 describes the commutative diagram with respect to NTT with $\odot$ representing element-wise multiplication in the integer domain. We also provide a typical instance of an NTT algorithm implemented in hardware designs in Alg. 1.

Algorithm 1 Decimation-in-time NTT with Cooley-Tukey Butterfly [30]

Input: $a \in \mathcal{R}_q$ (in normal order)
Input: ψ_{rev} (N powers of ψ in bit-reversed order)
Output: $a \leftarrow \text{NTT}(a) \in \mathcal{R}_q$ (in bit-reversed order)

1: $t \leftarrow n$
2: **for** $(m = 1; m < n; m = 2 \cdot m)$ **do**
3: $\quad t \leftarrow t/2$
4: $\quad$ **for** $(i = 0; i < m; i = i + 1)$ **do**
5: $\quad\quad j_1 \leftarrow 2 \cdot i \cdot t$, $j_2 \leftarrow j_1 + t - 1$
6: $\quad\quad w \leftarrow \psi_{rev}[m + i]$
7: $\quad\quad$ **for** $(j = j_1; j \leq j_2; j = j + 1)$ **do**
8: $\quad\quad\quad u \leftarrow a_j$
9: $\quad\quad\quad v \leftarrow a_{j+t} \cdot w \pmod{q}$
10: $\quad\quad\quad a_j \leftarrow (u + v) \pmod{q}$
11: $\quad\quad\quad a_{j+t} \leftarrow (u - v) \pmod{q}$
12: **return** a

2.3 Residue Number System (RNS)

The Residue Number System (RNS) makes use of the Chinese Remainder Theorem (CRT) to represent an integer as a vector of its residues modulo a basis of pairwise co-prime integers. The same can also be applied to polynomials in rings. If a is a polynomial in $\mathcal{R}_Q$ and $\mathcal{C} = \{q_0, \cdots, q_{k-1}\}$ is a basis such that $Q = \prod_{i=0}^{k-1} q_i$ then, there is a ring isomorphism from $a \in \mathcal{R}_Q$ to its representation $(a^{(0)}, a^{(1)}, \cdots, a^{(k-1)}) \in \prod_{i=0}^{k-1} \mathcal{R}_{q_i}$ being applied coefficient-wise. RNS is often used in conjunction with NTT for implementing HE schemes on hardware and software platforms as it enables the parallelization of computations. Fig. 1 gives a visual representation of the ring mappings involved.

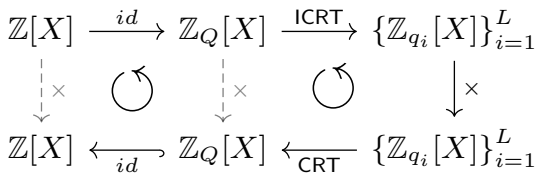

Fig. 1: RNS using CRT.

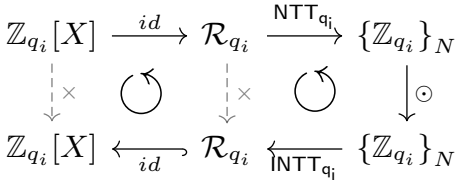

Fig. 2: NTT using RNS moduli q_i.

2.4 General structure of RLWE-based HE schemes

In a RLWE-based homomorphic encryption scheme, a client usually generates a secret-key $\mathtt{sk} = (1, s) \in \mathcal{R}_q^2$ and the corresponding public-key $\mathtt{pk} = (b = -a \cdot s + e, a) \in \mathcal{R}_q^2$ to encrypt a message m under the ciphertext $\mathtt{ct} \leftarrow (c_0 = v \cdot b + e_0 + m, c_1 = v \cdot a + e_1) \in \mathcal{R}_q^2$. The error polynomial e_i is sample from a Gaussian distribution and v is polynomial sampled from another distribution often specific to the scheme. The ciphertext $\mathtt{ct}$ is decryptable under $\mathtt{sk}$ because the operation $(c_0 \cdot 1 + c_1 \cdot s) \pmod q$ will return m (or its close approximation as in the case of CKKS). Assume that the client sends ciphertexts $\mathtt{ct}$ and $\mathtt{ct}'$ to the cloud to perform computations on them. Let $\mathtt{ct} = (c_0, c_1)$ and $\mathtt{ct}' = (c_0', c_1') \in \mathcal{R}_q^2$ encrypt m and m' respectively, then the cloud can compute a valid encryption of $m + m'$ by a simple component-wise addition of the ciphertexts, that is, $\mathtt{ct}_{\text{add}} \leftarrow (c_0 + c_0', c_1 + c_1') \in \mathcal{R}_q^2$. Note that addition is a linear operation and hence relatively simple. Homomorphic multiplication which consists of computing encryption of $m \cdot m'$, however, is more elaborate. A general intuitive idea is that multiplying two ciphertexts involves multiplying their respective components with each other, like, $\mathtt{ct}_{\text{mult}} = (c_0 \cdot c_0', c_0 \cdot c_1' + c_1 \cdot c_0', c_1 \cdot c_1') \in \mathcal{R}_q^3$. Notice that the ciphertext $\mathtt{ct}_{\text{mult}}$ has three polynomial components and is actually decryptable using $(1, s, s^2)$ but not using $\mathtt{sk} = (1, s)$, which deviates from our desired form of ciphertext. To again enable decryption using $\mathtt{sk}$, a 'Key-Switching' operation is used to transform the three-component ciphertext $\mathtt{ct}_{\text{mult}}$ back into the usual two-component ciphertext $\mathtt{ct}_{\text{relin}}$ decryptable under $(1, s)$. In this context, key-switching is called 'relinearization' as it produces a ciphertext that has a linear decryption equation with respect to $\mathtt{sk}$. It requires multiplying $\mathtt{ct}_{\text{mult}}$ with a relinearization key or 'evaluation' key that contains a public encryption of s^2. In order to perform arbitrary number of such homomorphic operations, a process known as 'bootstrapping' is applied to reduce the level of noise in the

ciphertext by homomorphically evaluating the decryption circuit. This is also often known as 'refreshing' the ciphertext and require some additional operations. While most ideal lattice-based homomorphic encryption schemes such as BGV [7], CKKS and RNS-CKKS [12] share a similar underlying protocol structure, the exact algorithms for multiplication and relinearization vary across these schemes. We describe the vital components of RNS-CKKS in the next paragraph.

RNS-CKKS: For a desired security parameter λ, RNS-CKKS sets a univariate polynomial degree N, a ciphertext modulus Q_L such that its corresponding RNS-basis is given by $\mathcal{C} = \{q_0, \cdots, q_L\}$. At a certain level $0 \leq l \leq L$, let the sub-basis be $\mathcal{C}_l = \{q_0, \cdots, q_l\}$ and Q_l be the product of $q_i's$ up to l. The secret polynomial $\texttt{sk}$ is sampled from a key distribution with pre-determined properties and then the public key $\texttt{pk}$ is generated as an RLWE sample as mentioned in the previous paragraph. The ciphertext in $\mathcal{R}^2_{Q_L}$, is also obtained similarly as before, but each ciphertext component c_0 and c_1 is now actually a tuple of L-elements (that is, each residue polynomial is in $\mathcal{R}_{q_i}$ where, $0 \leq i \leq L$). The relinearization key $\texttt{evk} = (evk_0, evk_1)$ is generated such that each of evk_0 and evk_1 is an L-tuple of polynomials in $\mathcal{R}_{pQ_L}$ (p is a special prime used only during key-switching/relinearization operations) and sent to the cloud to be used during relinearization. Further, each element in the L-tuple is again decomposed into $(L+1)$ residue polynomials corresponding to p and the RNS basis of Q_L. The important sub-routines are given below:

- $\texttt{CKKS.Add}(\texttt{ct}, \texttt{ct}')$: The addition operation between two ciphertexts $\texttt{ct} = (c_0, c_1)$ and $\texttt{ct}' = (c'_0, c'_1)$ in $\mathcal{R}^2_{Q_l}$ computes $\texttt{ct}_{\text{add}} = (d_0, d_1)$ where $d_0 = c_0 + c'_0 \in \mathcal{R}_{Q_l}$ and $d_1 = c_1 + c'_1 \in \mathcal{R}_{Q_l}$.
- $\texttt{CKKS.Mult}(\texttt{ct}, \texttt{ct}')$: Given two input ciphertexts $\texttt{ct} = (c_0, c_1) \in \mathcal{R}^2_{Q_l}$ and $\texttt{ct}' = (c'_0, c'_1) \in \mathcal{R}^2_{Q_l}$, it computes $d_0 = c_0 \cdot c'_0 \in \mathcal{R}_{Q_l}$, $d_1 = c_0 \cdot c'_1 + c_1 \cdot c'_0 \in \mathcal{R}_{Q_l}$, and $d_2 = c_1 \cdot c'_1 \in \mathcal{R}_{Q_l}$. The output is the non-linear ciphertext $\texttt{ct}_{\text{mult}} = (d_0, d_1, d_2) \in \mathcal{R}^3_{Q_l}$.
- $\texttt{CKKS.Relin}_{\texttt{evk}}(\texttt{ct}_{\text{mult}})$: With the help of the evaluation key, $\texttt{evk} = (evk_0, evk_1)$, compute $\texttt{ct}'' = (c''_0, c''_1)$ where $c''_0 = \sum_{i=0}^{l-1} d'_2[i] \cdot \texttt{evk}_0[i] \in \mathcal{R}_{pQ_l}$ and $c''_1 = \sum_{i=0}^{l-1} d'_2[i] \cdot \texttt{evk}_1[i] \in \mathcal{R}_{pQ_l}$, where the notation $[i]$ denotes the i-th element of the L-tuple. The final output is the relinearized ciphertext given by $\texttt{ct}_{relin} = (d_0, d_1) + (\texttt{CKKS.ModDown}(c''_0), \texttt{CKKS.ModDown}(c''_1)) \pmod{Q_l}$. The $\texttt{CKKS.ModDown}$ operation is used to reduce the coefficient modulus from pQ_l to Q_l.

An operation known as rescaling takes a ciphertext $\texttt{ct} = (c_0, c_1) \in \mathcal{R}^2_{Q_l}$ with level l and produces a ciphertext $\texttt{ct}' = (c'_0, c'_1)$ at level $l-1$. RNS-CKKS also includes automorphism operations which are special permutations (such as a rotation) of ciphertext coefficients. This can be considered as another type of key-switching operation that employs automorphism keys that encrypt the image of the secret key under the specified automorphism (for example, the rotated secret key), similar to key-switching or relinearization keys. Multiplication of the original ciphertext with the automorphism key ensures that the resultant ciphertext is an

encryption of the image of the message under the automorphism. Bootstrapping in CKKS relies on the aforementioned homomorphic subroutines [11]. We provide the pseudo-codes for `CKKS.Mult` and `CKKS.Relin` later in Sec. 5.

3 New multiplication using Fermat number and multivariate polynomial rings

We explain the new polynomial multiplication method in two steps, as elaborated in the following two subsections.

3.1 Multiplication using an auxiliary modulus mapping

With the application of RNS (Sec. 2.3), a polynomial multiplication in $\mathcal{R}_Q$ transforms into polynomial multiplications between the residue polynomials in the respective $\mathcal{R}_{q_i}$ rings. For computing these multiplications with respect to different moduli q_i, we use a common and sufficiently large auxiliary modulus P, following a similar approach presented in [13]. Fig. 3 illustrates the respective maps required to transition between the residue rings modulo q_i and the ring modulo P. We first switch from R_{q_i} to R_P, perform the multiplication in R_P, and switch back to R_{q_i} after the multiplication. For the correct computation of polynomial multiplications, it is essential to have a sufficiently large P such that no true modular reduction by P ever happens. This is ensured when $P > \frac{q_i^2}{4} \cdot N$ for all q_i. With the additional condition of $P \equiv 1 \pmod{2N}$, the multiplication in the ring $\mathcal{R}_P$ can be performed using the usual NTT approach for modulus P. Using a common auxiliary modulus P for polynomial multiplications in RLWE-based FHE schemes [7, 17, 12], where many q_i moduli are employed, is advantageous for achieving flexibility on hardware platforms. The disadvantage is the increased computation cost due to the use of a two times larger modulus.

Using Fermat number as auxiliary modulus: Choosing $P = F_K = 2^K + 1$, where K is a power-of-2, offers several benefits and improves the algorithm's compatibility with hardware acceleration. For instance, reduction modulo F_K can be efficiently carried out due to its simple binary structure. Another significant advantage of using Fermat number F_K is that the $2K$-th root of unity

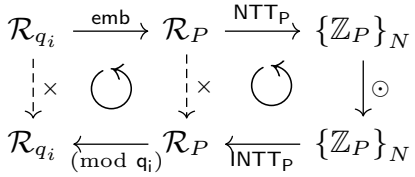

Fig. 3: Polynomial multiplication using auxiliary modulus

modulo F_K is 2, thus for $N \leq K$, twiddle factors used in NTT are powers-of-2. As a result, all the multiplications in the NTT algorithm can be replaced with bit shifting modulo F_K.

Challenge: The constraint of employing Fermat numbers F_K lies in the fact that efficient NTT is only feasible for ring dimensions $N \leq K$. For instance, with $F_{128} = 2^{128} + 1$, the maximum attainable NTT dimension is $N = 128$. However, in RLWE-based FHE schemes, the polynomial sizes are typically much larger, often reaching $N = 2^{16}$, necessitating the adoption of the substantial 65,537-bit Fermat number $F_{2^{16}}$ as an auxiliary modulus for NTT. If the moduli q_i are 54 bits, the above situation increases the coefficient size by 1214×.

3.2 Multivariate ring structure as a solution for the case, $N > K$

When $N > K$, it becomes essential to develop a new polynomial multiplication method that overcomes the constraint of FNTT, as we discussed in the previous subsection. To utilize the benefits of a reasonably small F_K, we introduce multivariate residue rings.

We transform the univariate polynomial structure, $\mathcal{R}_P = \mathbb{Z}_P[X]/(X^N + 1)$ into a multivariate polynomial structure, $\mathcal{S}_P = \mathbb{Z}_P[X_1, \dots X_k]/(X_1^{N_1}+1, \dots X_k^{N_k}+1)$, where $N_i|K$. We perform computations in the ring $\mathcal{S}_P$ and then revert back to $\mathcal{R}_P$. For instance, we set $X_1 = X$ and introduce the new variable $X_2 = X_1^{K/2}$ so that the univariate ring, $\mathcal{R}_P = \mathbb{Z}_P[X]/(X^N + 1)$ can be embedded into the bivariate ring, $\mathbb{Z}_P[X_1, X_2]/(X_2^{2N/K} + 1)$. We then perform polynomial multiplication in $\mathbb{Z}_P[X_1, X_2]/(X_2^{2N/K} + 1)$ and can then revert back to $\mathbb{Z}_P[X]/(X^N + 1)$ by substituting $X_2 = X_1^{K/2}$. Note that we set $X_2 = X_1^{K/2}$ instead of $X_2 = X_1^K$ because this constraint (overflow condition) guarantees that all powers of X_1 post-embedding are less than $K/2$, that is, a univariate polynomial $a(X) = a_0 + a_1X + \cdots + a_NX^N \in \mathcal{R}_P$ gets transformed into a bivariate polynomial $a'(X_1, X_2) = a_0 + a_1X_1 + \cdots + a_{K/2-1}X_1^{K/2-1} + a_{K/2}X_2 + a_{K/2+1}X_2^2 + \cdots + a_{2N/K}X_2^{2N/K}$. Consequently, all powers of X_1 post-multiplication are less than K and hence, the multiplication in the ring $\mathbb{Z}_P[X_1, X_2]/(X_2^{2N/K} + 1)$ mimics the polynomial multiplication in the ring $\mathbb{Z}_P[X_1, X_2]/(X_1^K + 1, X_2^{2N/K} + 1)$. The latter can be efficiently achieved using FNTTs for dimensions K and $2N/K$. Now, if $2N/K > K$, we introduce a third variable $X_3 = X_2^{K/2}$, and embed the ring $\mathcal{R}_P = \mathbb{Z}_P[X]/(X^N + 1)$ into $\mathbb{Z}_P[X_1, X_2, X_3]/(X_3^{4N/K^2} + 1)$, so that again by the same logic, multiplication in the later ring coincides with the multiplication in the former ring and the dimensions of the FNTTs would be K and $4N/K^2$. Thus by applying this technique iteratively, we substitute univariate polynomial multiplication in the ring $\mathcal{R} = \mathbb{Z}_P[X]/(X^N + 1)$ with multivariate polynomial multiplication in the ring $\mathcal{S} = \mathbb{Z}_P[X_1, \dots, X_k]/(X_1^{N_1} + 1, \dots, X_k^{N_k} + 1)$, such that $X_1 = X$, $X_{i+1} = X_i^{N_i/2}$, and $N_i|K$. This process is described in Fig. 4. Note that, from now on, we use term multivariate NTT (MV-NTT) to refer to the NTT operation in multivariate setting. Each smaller dimension NTT performed

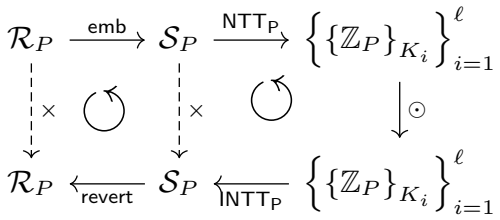

Fig. 4: Multivariate polynomial structure method

in MV-NTT is an NTT with Fermat number, i.e., an FNTT.

Advantages in hardware acceleration: Splitting the univariate polynomial ring into a multivariate ring reduces the dimension of FNTTs, which enables the use of Fermat numbers efficiently in NTT computations. These smaller dimensional FNTTs can be computed in parallel. Also, since each small FNTT with the same size has the same structure, it is possible to design a unified, fast and tailored circuit for all the small FNTTs. Further, smaller NTT dimension reduces the size difference between the auxiliary Fermat modulus F_K and the original RNS moduli q_i, hence reducing the overall area overhead introduced by the auxiliary modulus.

Challenges: This method introduces a trade-off: while decreasing the dimension of FNTTs, it increases the number of FNTTs to be performed with respect to each polynomial. For example, with every new variable introduced, the total dimension (i.e., total number of FNTTs) doubles, which increases the number of computations, memory and bandwidth requirements. The increased number of computations need to be addressed by leveraging the cost-efficient arithmetic units enabled by FNTT's shift-based modular multiplication.

3.3 The cost of the new technique in terms of bit operations

In this section, we first provide pseudocode for the implementation steps of the MV-NTT used in the new multiplication technique. Then, we evaluate and compare the cost of the new technique's MV-NTT and conventional NTT in terms of bit operations.

In Alg. 2, we present the MV-NTT operation for the new technique with $k = 3$, which leads to an MV-NTT operation consisting of three steps with sizes N_1, N_2 and N_3, respectively. As shown in Alg. 2, a transpose operation is required between each FNTT step. Note that M in Alg. 2 represents a three-dimensional structure with N_3 vertical planes. $M[i][j][:]$ represents the j-th row of i-th plane of M. Each small dimension NTT operation in Alg. 2 is performed using FNTT with Fermat number $F_K = 2^K + 1$ (FNTT_{N_1}, FNTT_{N_2}, FNTT_{N_3}).

Algorithm 2 Multivariate NTT with $k = 3$

Input: a (a polynomial of size N)
Output: $M = \text{MV-NTT}(a)$ (a three-dimensional matrix of dimension $N_2 \times N_1 \times N_3$)

Lift the centered coefficients of a in $\mathbb{Z}_{q_i}$ to $\mathbb{Z}_{F_K}$
for $(i = 0;\ i < N_3;\ i = i + 1)$ **do** ▷ Mapping to three-dimensional matrix
    for $(j = 0;\ j < N_2/2;\ j = j + 1)$ **do**
        for $(z = 0;\ z < N_1/2;\ z = z + 1)$ **do**
            $M[i][j][z] \leftarrow a[i \cdot (N_1/2) \cdot (N_2/2) + j \cdot (N_1/2) + z]$
for $(i = 0;\ i < N_3;\ i = i + 1)$ **do**
    for $(j = 0;\ j < N_2;\ j = j + 1)$ **do**
        $M[i][j][:] \leftarrow \text{FNTT}_{N_1}(M[i][j][:])$ ▷ Apply N_1-pt FNTT
Transpose the first and second dim. of M ▷$(N_3 \times N_2 \times N_1) \rightarrow (N_1 \times N_3 \times N_2)$
for $(i = 0;\ i < N_1;\ i = i + 1)$ **do**
    for $(j = 0;\ j < N_3;\ j = j + 1)$ **do**
        $M[i][j][:] \leftarrow \text{FNTT}_{N_2}(M[i][j][:])$ ▷ Apply N_2-pt FNTT
Transpose the second and third dim. of M ▷$(N_1 \times N_3 \times N_2) \rightarrow (N_2 \times N_1 \times N_3)$
for $(i = 0;\ i < N_2;\ i = i + 1)$ **do**
    for $(j = 0;\ j < N_1;\ j = j + 1)$ **do**
        $M[i][j][:] \leftarrow \text{FNTT}_{N_3}(M[i][j][:])$ ▷ Apply N_3-pt FNTT
return M

The inverse MV-NTT operation for the new technique uses a similar routine in the opposite order (i.e., it starts with N_3-pt IFNTTs and ends with lifting the centered coefficients in $\mathbb{Z}_{F_K}$ to $\mathbb{Z}_{q_i}$).

Bit operation cost: CPUs have integer multiplication units that can provide support for different applications with various word sizes. A software designer does not consider the number of bit operations as the size of integer multiplication units in CPUs is fixed. On the other hand, the number of bit operations in a hardware design can make a substantial difference as the operations could be parallelized or unrolled and the resource allocation for each arithmetic unit can be set manually. Thus, we believe that it is important to provide the cost of bit operations to evaluate the new multiplication technique and compare it to conventional NTT-based multiplication. To that end, we followed a similar approach as [6] and presented the implementation cost of the new multiplication technique in terms of the total number of bit operations, i.e., the total number of gate evaluations in arithmetic operation implementations.

In Table 1, we present the number of arithmetic operations, modular addition (ModAdd), modular subtraction (ModSub), integer multiplication (IntMul) and modular reduction (ModRed) for both a conventional N-pt NTT and our proposed MV-NTT technique with three variables (i.e., with three FNTT steps of sizes N_1, N_2 and N_3). Both ModAdd and ModSub operations require one adder, one subtractor and one 2-to-1 multiplexer unit. For ModRed, a generic modular reduction method such as Barrett reduction uses two multiplications, two subtractions and one 2-to-1 multiplexer [5]. In total, IntMul and ModRed

Table 1: The number of arithmetic operations

Operation	**ModAdd**	**ModSub**	**IntMul**	**ModRed**
Conv. NTT	$(N/2) \cdot \log_2(N)$	$(N/2) \cdot \log_2(N)$	$(N/2) \cdot \log_2(N)$	$(N/2) \cdot \log_2(N)$
MV-NTT ($k = 3$)	$(N_1/2) \cdot \log_2(N_1) \cdot N_2 \cdot N_3 +$ $(N_2/2) \cdot \log_2(N_2) \cdot N_1 \cdot N_3 +$ $(N_3/2) \cdot \log_2(N_3) \cdot N_1 \cdot N_2$	$(N_1/2) \cdot \log_2(N_1) \cdot N_2 \cdot N_3 +$ $(N_2/2) \cdot \log_2(N_2) \cdot N_1 \cdot N_3 +$ $(N_3/2) \cdot \log_2(N_3) \cdot N_1 \cdot N_2$	0 0 0	$(N_1/2) \cdot \log_2(N_1) \cdot N_2 \cdot N_3 +$ $(N_2/2) \cdot \log_2(N_2) \cdot N_1 \cdot N_3 +$ $(N_3/2) \cdot \log_2(N_3) \cdot N_1 \cdot N_2$

operations require three integer multiplications, two subtractions and one 2-to-1 multiplexer unit. On the other hand, IntMul for FNTT is a free-of-cost left shift operation and modular reduction is much simpler, which can be implemented using only one subtractor, one adder and one 2-to-1 multiplexer unit, thanks to its reduction-friendly form.

For proof-of-concept, we set $k = 3$, $K = 2^7$, $N_1 = 2^7$, $N_2 = 2^7$ and $N_3 = 2^4$ with Fermat number $F_{128} = 2^{128} + 1$ as FNTT modulus. These parameters support polynomial multiplication in univariate ring setting with $N = 2^{16}$ and up to 54-bit prime modulus q_i. We used a 28-nm ASIC library to synthesize and obtain gate counts for adder, subtractor, 2-to-1 multiplexer and integer multiplier units for 54-bit and 129-bit integers. The synthesized 54-bit adder, subtractor, 2-to-1 multiplexer and integer multiplier use 0.6K, 0.6K, 0.1K and 9.6K gate evaluations (based on NAND2 gate area), respectively, while the synthesized 129-bit adder, subtractor and 2-to-1 multiplexer use 0.8K, 0.8K and 0.2K gate evaluations, respectively. We then used these numbers and Table 1 to calculate the total gate evaluation cost of computations. While conventional NTT operation uses ≈39.8B gate evaluations, our method uses ≈36.2B gate evaluations, which is ≈10% less. Note that, for the selected parameters, although our MV-NTT uses 4.5× more arithmetic operations, eliminating twiddle factor multiplications during NTTs made the overall gate evaluation cost of MV-NTT less than the conventional NTT.

4 The MV-NTT architecture for the proposed technique

We split the hardware description into two parts: *i*) The MV-NTT design and *ii*) the overall architecture containing the MV-NTT for an FHE accelerator. In this section, we first briefly discuss different NTT architectures in the literature for univariate rings and our design rationale to implement MV-NTT efficiently. Then, we propose an architecture tailored for the proposed MV-NTT method.

4.1 Prior works and our design rationale

Designing an efficient NTT unit is crucial for the performance of an FHE accelerator as NTT operation consumes more than 50% of the total resources and run time in FHE accelerators [3]. NTT architectures in the literature are designed for univariate rings with NTT-friendly primes. Thus, adapting prior NTT architectures for the multivariate ring setting is not straightforward. Existing NTT

architectures in the literature follow either an iterative design approach such as [31], the hierarchical (i.e., 4-step NTT) approach such as [18, 38, 28, 27, 19, 3] or hybrid approaches [3].

In the iterative design approach [31], an N-pt NTT is performed in $\log_2(N)$ stages where each stage involves $N/2$ butterfly operations. The performance and cost of an NTT implementation with an iterative design approach are scalable. However, since the memory access pattern is different for each NTT stage, an iterative NTT design in hardware requires configurable connections between computation and memory units, which leads to complex multiplexer structures. The implementation complexity further increases significantly when a large number of butterfly units is employed.

The hierarchical approach breaks a large NTT into smaller steps. It converts an N-element polynomial into a matrix of dimension $N_1 \times N_2$, where $N = N_1 \cdot N_2$, and applies N_1-pt NTT to its columns. A Hadamard product with twiddle factors follows, and the resulting matrix undergoes transposition. The final step involves applying N_2-pt NTT to the columns of the transposed matrix. Despite its simplified control logic, this approach poses challenges in resource-constrained environments and incurs high costs for the transpose operation, especially with larger N values. Several ASIC-based NTT implementations [18, 38, 28, 27, 19] employ the 4-step NTT approach with $N = N_1 \cdot N_2$ and $N_1 = N_2$. The small N_1 or N_2-pt NTTs are implemented as radix-N_1 fully unrolled architectures for improving performance. This approach lacks scalability due to the necessity of $N_1 = N_2$. Additionally, the transpose operation is expensive to perform in hardware. A recent work [3] combines two orthogonal NTT design methods, removing the necessity for transpose operations in a hierarchical design, and offering the flexibility of $N_1 \neq N_2$ or $N_1 = N_2$.

4.2 The proposed MV-NTT architecture

The proposed FNTT in multivariate ring offers the following implementation advantages: *i*) multiplication by a twiddle factor becomes simple shift, *ii*) modular reduction becomes very cheap thanks to sparse structure of Fermat number, and *iii*) a big NTT is split into many smaller ones in the multivariate setting, allowing for fast and optimized circuits for each small NTT.

Despite its advantages, there are two main implementation challenges: the number of arithmetic operations increases compared to the conventional NTT (see Sec. 3.3), and the memory and bandwidth requirements increase because $F_K > \frac{q^2}{4} \cdot N$ and the total size of multivariate polynomial $N_3 \cdot N_2 \cdot N_1 > N$.

To mitigate the aforementioned challenges, we focus on a high-performance design strategy that capitalizes on the affordability of modular arithmetic units, enabled by the special properties of Fermat number as detailed in Sec. 3.3. Table 1 shows that our NTT requires many small NTTs. Nonetheless, the small NTTs can be implemented efficiently employing many low-cost processing units. We use multiple unrolled NTTs to realize parallel and pipeline processing.

Target parameter set: For proof-of-concept, we used the same parameter set described in Sec. 3.3, $K = 2^7$ with $N_1 = 2^7$, $N_2 = 2^7$ and $N_3 = 2^4$ in a three-variate ring setting. This parameter set supports polynomial multiplication in univariate ring for $N = 2^{16}$ and up to 54-bit q_i modulus. Note that similar parameter sets are used for bootstrapping in RLWE-based FHE schemes.

Data flow in MV-NTT: First, we illustrate the flow of the MV-NTT using a toy example with $N_1 = 2^3$, $N_2 = 2^3$ and $N_3 = 2$, which can compute univariate NTT for $N = 2^5$. The coefficients for the MV-NTT are arranged in a three-dimensional matrix M as shown in Fig. 5. The matrix has N_3 vertical planes where each plane has N_1 columns and N_2 rows. The transformation of a univariate polynomial into the three-dimensional M is shown in lines 2-5 of Alg. 2. As the number of elements in M is greater than N, the initial matrix contains empty cells with zeros.

In Fig. 5, we show how the elements of M are processed. In the first step, N_1-pt FNTTs are applied to the rows of the planes of M. Then, a transpose is performed to reshape M as $N_2 \times N_3 \times N_1$, followed by N_2-pt FNTTs on the rows of the planes again. Finally, another transpose operation is performed and N_3-pt FNTTs are applied to the rows. MV-INTT performs the same steps in opposite order. Further, it performs a final step, as shown in Fig. 5 (b), for reducing the number of elements in M to N. Next, we explain the proposed MV-NTT architecture tailored for multivariate ring setting and FNTT.

High-performance MV-NTT architecture: Our focus is on developing a high-performance hardware architecture while keeping implementation complexity to a minimum, with a particular emphasis on reducing the computational overhead associated with transpose operations. The architecture depicted in Fig. 6 illustrates our proposed design for MV-NTT tailored to a 3-variate ring ($k = 3$). This design necessitates three FNTT steps. Following the data flow diagram in the previous section, a transpose operation is required between two FNTT stages.

To eliminate transpose operation, we employ different types of FNTT architectures for each step. Specifically, we use two N_1-pt unrolled FNTT units for the first step, N_1 parallel N_2-pt FNTT units for the second step, and N_1 parallel N_3-pt pipelined FNTT units for the third step. In Fig. 5, the output coefficients (N_1 in total) generated by the first FNTT step serve as the input coefficients for N_1 parallel N_2-point FNTT units in the second step. By using a fully unrolled FNTT in the first step, we can directly feed all N_1 parallel pipelined FNTT units (each taking one coefficient per cycle as input) in the second step without requiring a transpose between these two steps. The proposed MV-NTT takes one row of a plane per cycle (N_1 coefficients), performs N_1-pt FNTT, and sends resulting coefficients to the N_1 parallel N_2-pt pipelined FNTT units.

After the second FNTT step, another transpose operation is required. As shown in step 2 of Fig. 5, all rows of a plane have to be processed by FNTT before starting the third FNTT step. Thus, we use N_1 parallel SRAM-based on-

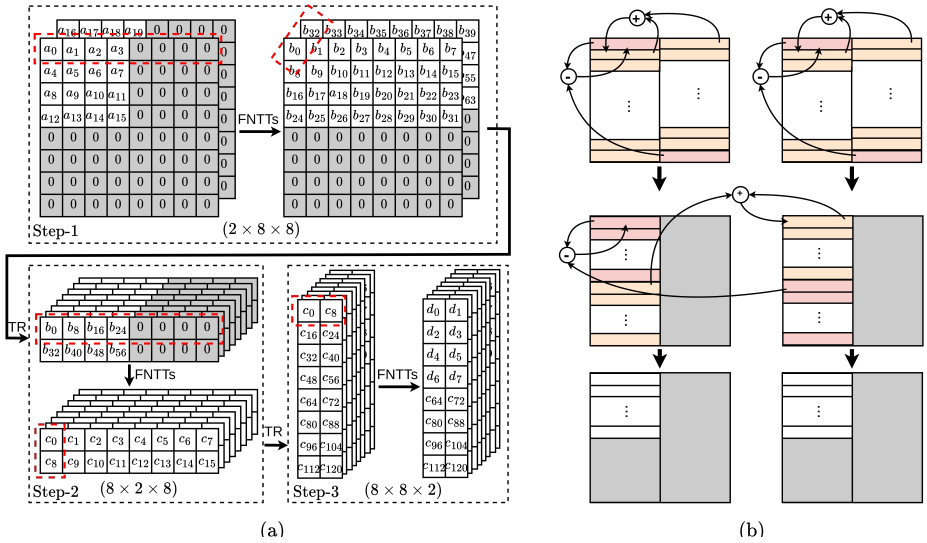

Fig. 5: (a) Data flow in MV-NTT in ring setting with $k = 3$, $N = 2^5$, $N_1 = 2^3$, $N_2 = 2^3$ and $N_3 = 2$. Gray boxes represent cells with zeros. (b) The final reduction operation after the MV-INTT. Coefficients in red boxes are subtracted while coefficients in yellow boxes are accumulated.

chip memory after the second FNTT step to store resulting planes. It should be noted that these SRAMs are only used to accumulate enough coefficients so the third FNTT step can start. They do not employ complex multiplexer or routing configurations as transpose units. After accumulating enough coefficients to start the third FNTT step, the coefficients are read from SRAMs accordingly to feed the N_1 parallel N_3-pt pipelined FNTT units.

An N-pt pipelined FNTT architecture uses $\log_2(N)$ cascaded butterfly units, where each butterfly unit performs one stage of FNTT iteratively. There are two pipelined FNTT architectures in the literature, SDF and MDC [22]. The SDF architecture takes one coefficient per cycle while MDC architecture takes two coefficients per cycle. In our design, we use MDC-based FNTT architecture. As shown in the second step of Fig. 5, half of the coefficients in each row of a plane are zeros. An N_2-pt MDC-based FNTT architecture takes two coefficients per cycle, where coefficient indices are separated by $N_2/2$ (i.e., a_i and $a_{i+N_2/2}$). Thus, using MDC-based pipelined FNTT architecture enables eliminating the first butterfly stage inside N_2-pt MDC-based FNTT architecture as the second butterfly input is zero. Employing an MDC-based configuration instead of an SDF-based one improves the performance of MV-NTT.

N_1-pt unrolled FNTT unit in the first step can take and generate N_1 coefficients per cycle. On the other hand, N_1 parallel MDC-based FNTT unit in the second step can take and generate $2 \cdot N_1$ coefficients per cycle. This leads to a difference between the bandwidths of the two steps, which can hinder the

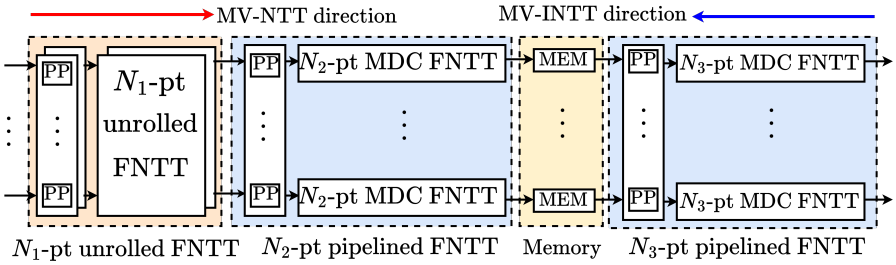

Fig. 6: Unified bi-directional MV-NTT/INTT architecture.

performance. To mitigate this problem and match the bandwidth of the first and second steps, we employ two unrolled FNTT units in the first step. It should be noted that the area overhead of the unrolled FNTT unit is very low thanks to the multiplication-free structure of FNTT. The proposed NTT architecture is scalable for different FNTT sizes and it can also be easily adapted for $k = 2$ setting (i.e., eliminating the second transpose and the third FNTT steps). The proposed architecture has an average latency of 1024 cycles for MV-NTT and MV-INTT. Details of low-level arithmetic units are presented in Appendix A.

Twiddle factor elimination: Besides removing integer multiplications, FNTT saves logic and on-chip memory resources needed to manage twiddle factors. For example, consider an FHE accelerator supporting 32 primes with NTT of size $N = 2^{16}$ and prime size 54-bit in a univariate ring setting. The conventional approach would necessitate $\approx$ 13.5MB of on-chip memory to store twiddle factors. By adopting an on-the-fly twiddle factor generation approach, the on-chip memory can be reduced up to 98.3% [3]. However, this efficiency comes at the cost of employing additional multiplier units for on-chip constant generation. Our proposed technique offers a much more efficient solution by eliminating the need for both on-chip memory and multipliers for twiddle factors.

Unifying MV-NTT and MV-INTT: Traditionally, INTT operation requires INTT twiddle factors (i.e., ω^{-1}, modular inverse of NTT twiddle factor ω) and uses DIF-based NTT butterfly (while NTT uses DIT-based NTT butterfly). This also applies to MV-NTT and MV-INTT. Thus, it is not straightforward to support both MV-NTT and MV-INTT using the same implementation efficiently. To support MV-INTT using the MV-NTT architecture, we used two techniques. First, we used negative-wrapped convolution with separate pre- and post-processing steps to implement FNTT/IFNTT operations inside MV-NTT/INTT. Normally, It is possible to merge an FNTT with pre-processing (using DIT-based butterfly) and IFNTT with post-processing (using DIF-based butterfly). However, this requires two different butterfly structures and complicates implementation. Instead, we used pre-processing followed by FNTT and IFNTT followed by post-processing [22]. As shown in Fig. 6, each FNTT unit is coupled

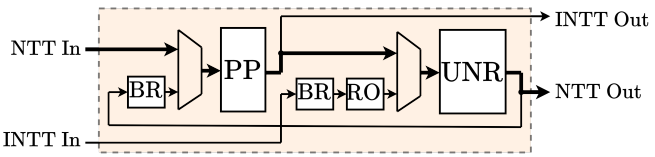

Fig. 7: Unified unrolled FNTT/IFNTT architecture. Here, the PP block represents a unified pre-/post-processing unit, the UNR block represents a DIT-based unrolled FNTT unit, the BR block represents a bit-reverse unit, and the RO block represents a re-ordering unit.

with pre-/post-processing (PP) units. This enables us to use one butterfly configuration (DIT-based butterfly) for both FNTT and IFNTT operations. Second, we use a technique proposed in [15] which enables performing IFNTT operation using the FNTT unit with FNTT twiddle factors, thus saving memory. This technique requires re-ordering (RO) IFNTT input $(a_0, a_1, \ldots, a_{N-2}, a_{N-1})$ as $(a_0, a_{N-1}, \ldots, a_2, a_1)$. Fig. 7 depicts the unified unrolled architecture for FNTT and IFNTT. Since we use either unrolled or pipelined architectures to implement FNTTs, bit-reverse (BR) and RO operations require only re-wiring the input/output coefficients and, hence are implemented free of cost.

5 The MV-NTT-based FHE accelerator architecture

As an application of MV-NTT, we introduce a hardware accelerator design tailored for the FHE scheme CKKS [12]. The detailed architecture is depicted in Fig. 8. The fundamental computational building blocks of any FHE accelerator [31, 3, 18, 28, 27, 38] consist of NTT/INTT, Multiply-and-Accumulate (MAC), and Automorphism units. In previous works on FHE acceleration, the NTT/INTT unit was solely required during the relinearization operation, leading to its specialized optimization via architecture design. However, this paradigm shifts when replacing the multi-modulus supporting NTT/INTT with the cost-effective MV-NTT/INTT. Therefore, before delving into the design decisions of the hardware accelerator, it is crucial to discuss the distinctions arising from the use of MV-NTT/INTT.

The aim of leveraging the MV-NTT/INTT is to analyze its cost-effectiveness and routing-friendly nature, as it eliminates the need for expensive modular multipliers. However, a challenge in realizing this objective is associated with the storage domain of polynomials. Typically, to minimize NTT to INTT conversions, ciphertexts and keys are consistently stored in NTT form. Consequently, MAC operations do not necessitate any additional NTT/INTT transformations. Given that the polynomial before and after the NTT/INTT operation is modulo the same prime, the size of the polynomial remains constant in this scenario.

The first naive option is to use the same technique and store all the keys and ciphertexts in the MV-NTT domain. This will imply that MV-NTT/INTT is

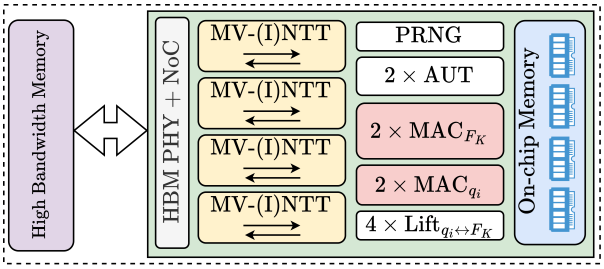

Fig. 8: The high-level architecture of MV-NTT based FHE accelerator.

again only required during the Relinearization operation. This naive approach is uncomplicated but costly in terms of storage and communication. This is primarily due to the expansion in polynomial size after MV-NTT, with the number of coefficients increasing by $4\times$ and the coefficient size growing by $\frac{129}{w}\times$ for the target parameter set (Sec. 4), where w is the size of RNS moduli. When applied to prior acceleration works, where w typically falls within the range of $28-54$, this results in an overall polynomial size increase of $9-18\times$. This polynomial expansion results in increased overhead on both communication and on-chip storage, particularly given that communication bottleneck is the major problem faced by the FHE accelerators in literature [38, 27, 28, 3, 2].

In order to avoid the challenges associated with off-chip to on-chip communication of larger polynomials and to ensure its short-term storage on-chip, we assert the continuous maintenance of data in coefficient form (non MV-NTT) off-chip. However, this necessitates transforming data to and from the MV-NTT domain for polynomial multiplication, as shown in Alg. 4. Relinearization, as showcased in the prior works, also requires NTT/INTT, although in a slightly different order due to ciphertexts being present in the coefficient form. However, the automorphism and additions/subtractions can be directly applied to data in coefficient form. Hence, the only operations that are affected by the use of MV-NTT/INTT are the ciphertext multiplication and Relinearization. Therefore, next, we will detail these two operations and see how the proposed accelerator (Fig. 8) is designed to optimize performance for both.

5.1 The ciphertext multiplication-oriented optimizations

An FHE ciphertext `ct` consists of two residue polynomials $(\tilde{c}_0, \tilde{c}_1)$. In order to multiply two such ciphertexts, each of the two sets is multiplied with each other residue-wise. To provide a more intuitive understanding of how ciphertext multiplication has changed, we present the previous and updated algorithms-Alg. 3 and Alg. 4, respectively. Only slot-wise multiplication was required in the former Alg. 3, where polynomials were already in NTT form. However, in our approach, aimed at minimizing communication and on-chip storage overhead, the polynomial must be transformed back and forth to the MV-NTT domain for slot-wise multiplications. Note that the MV-INTT step ensures that the polynomial,

post-multiplication, is in coefficient format and consequently smaller. As a result, the polynomials, after the MV-NTT transformation, have only a brief lifespan in on-chip memory consumption.

Algorithm 3 `CKKS.Mult` [12]

In: $\mathtt{ct} = (\tilde{c}_0, \tilde{c}_1)$, $\mathtt{ct}' = (\tilde{c}'_0, \tilde{c}'_1) \in R^2_{Q_l}$
Out: $\mathtt{d} = (\tilde{d}_0, \tilde{d}_1, \tilde{d}_2) \in R^3_{Q_l}$
1: **for** $j = 0$ to $l - 1$ **do**
2: $\quad \tilde{d}_0[j] \leftarrow \tilde{c}_0[j] \star \tilde{c}'_0[j]$
3: $\quad \tilde{d}_2[j] \leftarrow \tilde{c}_1[j] \star \tilde{c}'_1[j]$
4: $\quad \tilde{d}_1[j] \leftarrow \tilde{c}_0[j] \star \tilde{c}'_1[j] + \tilde{c}_1[j] \star \tilde{c}'_0[j]$

Algorithm 4 `Fermat.CKKS.Mult`

Input: $\mathtt{ct} = (c_0, c_1)$, $\mathtt{ct}' = (c'_0, c'_1) \in R^2_{Q_l}$
Out: $\mathtt{d} = (d_0, d_1, d_2) \in R^3_{Q_l}$
1: **for** $j = 0$ to $l - 1$ **do**
2: $\quad \tilde{c}_0[j], \tilde{c}_1[j] \leftarrow \mathtt{MV\text{-}NTT}(c_0[j], c_1[j])$
3: $\quad \tilde{c}'_0[j], \tilde{c}'_1[j] \leftarrow \mathtt{MV\text{-}NTT}(c'_0[j], c'_1[j])$
4: $\quad d_0[j] \leftarrow \mathtt{MV\text{-}INTT}(\tilde{c}_0[j] \star \tilde{c}'_0[j])$
5: $\quad d_2[j] \leftarrow \mathtt{MV\text{-}INTT}(\tilde{c}_1[j] \star \tilde{c}'_1[j])$
6: $\quad d_1[j] \leftarrow \mathtt{MV\text{-}INTT}(\tilde{c}_0[j] \star \tilde{c}'_1[j] + \tilde{c}_1[j] \star \tilde{c}'_0[j])$

For designing an accelerator, we observe that we now require seven MV-NTT/INTT operations per residue polynomial multiplication. A recent work in the literature [3] engineered all the building blocks to provide uniform throughput, enabling them to function both in a pipeline and in parallel. This approach assists in masking the overhead arising from extra NTT/INTT operations by placing all the NTTs and MACs in the pipeline. However, this choice comes with the cost of additional MV-NTTs in our design. Given that our MV-NTT is more cost-effective than NTT/INTT in [3], this additional cost is acceptable. Therefore, our architecture incorporates four MV-NTT units followed by two MAC units (refer to Fig. 8). These units exhibit the same throughput and write the output to memory. This configuration enables the simultaneous processing of two polynomial multiplications, requiring a total of two runs to process each residue polynomial set.

Our MV-NTT/INTT unit works in a pipelined manner and yields optimal results when processing data in a continuous flow. However, if this continuity is disrupted, the runtime increases due to the unit's latency being twice its throughput. Given that the same unit is utilized for MV-NTT/INTT for processing the first set of residue polynomials, we can only send two polynomials in a pipeline for MV-NTT, perform multiplication, and then must process them via MV-INTT before sending the data back. Following this, we proceed with the next residue polynomial set and perform MV-NTT.

In this scheduling, we reduce on-chip storage to just three polynomials. However, this comes with increased runtime as we do not fully utilize the throughput and incur latency costs. An alternative is to store all the MV-NTT results in on-chip and then, once all data is processed, send it for MV-INTT, storing the results in off-chip memory. This ensures that MV-NTT operates on a continuous data flow, allowing the runtime to be dominated by throughput rather than latency. However, for parameters with $l = 31$, this would necessitate 372MB of on-chip storage, which is both substantial and expensive.

Algorithm 5 `CKKS.Relin` [12]

In: $\mathtt{d} = (\tilde{d}_0, \tilde{d}_1, \tilde{d}_2) \in R^3_{Q_l}$, $\tilde{\mathtt{evk}}_0 \in R^l_{pQ_l}, \tilde{\mathtt{evk}}_1 \in R^l_{pQ_l}$
Out: $\mathtt{d}' = (\tilde{d}'_0, \tilde{d}'_1) \in R^2_{Q_l}$

for $j = 0$ to $l - 1$ **do**
  $d_2[j] \leftarrow \mathtt{INTT}(\tilde{d}_2[j])$ ▷ in $\mathbb{Z}_{q_j}$
for $j = 0$ to l **do** ▷ Here q_l is used to represent special prime p
  $(\tilde{c}''_0[j], \tilde{c}''_1[j]) \leftarrow 0$
  for $i = 0$ to $l - 1$ **do**
    $\tilde{r} \leftarrow \mathtt{NTT}([d_2[i]]_{q_j})$ ▷ in $\mathbb{Z}_{q_j}$
    $\tilde{c}''_0[j] \leftarrow [\tilde{c}''_0[j] + \tilde{\mathtt{evk}}_0[i][j] \star \tilde{r}]_{q_j}$, $\tilde{c}''_1[j] \leftarrow [\tilde{c}''_1[j] + \tilde{\mathtt{evk}}_1[i][j] \star \tilde{r}]_{q_j}$
$\tilde{d}'_0 \leftarrow \tilde{d}_0 + \mathtt{CKKS.ModDown}(\tilde{c}''_0)$, $\tilde{d}'_1 \leftarrow \tilde{d}_1 + \mathtt{CKKS.ModDown}(\tilde{c}''_1)$

Algorithm 6 `Fermat.CKKS.Relin`

In: $\mathtt{d} = (d_0, d_1, d_2) \in R^3_{Q_l}$, $\mathtt{evk}_0 \in R^l_{pQ_l}, \mathtt{evk}_1 \in R^l_{pQ_l}$
Out: $\mathtt{d}' = (d'_0, d'_1) \in R^2_{Q_l}$

for $j = 0$ to l **do** ▷ Here q_l is used to represent special prime p
  $(\tilde{c}''_0[j], \tilde{c}''_1[j]) \leftarrow 0$
  for $i = 0$ to $l - 1$ **do**
    $\tilde{r} \leftarrow \mathtt{MV\text{-}NTT}([d_2[i]]_{q_j})$ ▷ in $\mathbb{Z}_{fp}$
    $\tilde{\mathtt{evk}}_0[i][j], \tilde{\mathtt{evk}}_1[i][j] \leftarrow \mathtt{MV\text{-}NTT}(\mathtt{evk}_0[i][j], \mathtt{evk}_1[i][j])$ ▷ in $\mathbb{Z}_{fp}$
    $\tilde{c}''_0[j] \leftarrow [\tilde{c}''_0[j] + \tilde{\mathtt{evk}}_0[i][j] \star \tilde{r}]_{q_j}$, $\tilde{c}''_1[j] \leftarrow [\tilde{c}''_1[j] + \tilde{\mathtt{evk}}_1[i][j] \star \tilde{r}]_{q_j}$
for $j = 0$ to l **do**
  $c_0[j], c_1[j] \leftarrow \mathtt{MV\text{-}INTT}(\tilde{c}_0[j], \tilde{c}_1[j])$ ▷ in $\mathbb{Z}_{q_j}$
$d'_0 \leftarrow d_0 + \mathtt{CKKS.ModDown}(c''_0)$, $d'_1 \leftarrow d_1 + \mathtt{CKKS.ModDown}(c''_1)$

We adopt a middle-ground approach to strike a balance by dividing the computation in half. Instead of processing all l residue polynomial sets simultaneously, we first operate on $l/2$ sets of residue polynomials and then proceed to the second set. This allows us to halve the on-chip memory requirement to 186MB and incur the latency cost only twice instead of l times. It is worth noting that the division of computation can be adjusted based on the available on-chip memory, providing flexibility in managing the trade-off between storage requirements and runtime. We further note that the required on-chip memory can be reduced by using an extra MV-INTT unit that can work in parallel to send the result back to HBM. Given that the cost of MV-NTT/INTT is less than the on-chip storage required for multiple polynomials, this trade-off gives us a reduced chip area. Hence, our architecture features four MV-NTT/INTT units and much less on-chip storage.

5.2 The relinearization-oriented optimizations

The Relinearization operation is used to transform the ciphertext consisting of three residue polynomials after multiplication (Alg. 3 and Alg. 4) back to two

Table 2: Resource consumption breakdown of the FHE accelerator.

Building Block	**Area** (mm^2)
MV-NTT/INTT	4×21.5
MV-NTT/INTT (mem.)	4×17.4
MAC (for F_K)	2×1.3
MAC (for q_i)	2×1.5
$\text{Lift}_{q_i \leftrightarrow F_K}$	4×0.88
Automorphism	2×0.14
PRNG	0.14
On-chip memory	52
HBM3 PHY+NoC	33.8
Total	250.94

residue polynomial sets. Similar to the scenario of ciphertext multiplications, in Relinearization, we deviate slightly from the previous technique outlined in Alg. 5. Given that the data is not in NTT form, there is no need for an INTT operation for the mod-switch (Steps 1-2 Alg. 5). Instead, we can directly proceed with MV-NTT, as demonstrated in Steps 4 and 5 of Alg. 6. However, it is important to note that we have to transform not only the ciphertext but also the keys to MV-NTT for polynomial multiplications. Finally, after the computation, we must again transform the result to coefficient form using MV-INTT (Steps 7-8).

In the preceding subsection, we explored how the accelerator architecture in Fig. 8 is optimized for Polynomial Multiplication. Now, let us delve into this optimization in the context of Relinearization. Examining the second nested for loop (Steps 3-6), we observe that exactly three MV-NTT operations and two MAC operations are required. With the proposed architecture featuring four parallel MV-NTT units feeding two MAC units, these operations can be efficiently executed in a pipeline, and the results are then stored in memory. If there is a need to store all the results, a storage capacity of 256MB would be needed ($2 \times 32 \times 4MB$). Given we have two idle MV-INTTs during relinearization, we use them to process the result and send it back to the memory. Hence, requires very low on-chip storage ($4 \times 4MB$).

With this, we conclude our discussion on the optimized accelerator design. Next, we will discuss the area and performance results of the proposed MV-NTT-based FHE accelerator.

6 Results and comparisons

We synthesize building blocks of our proposed architecture with the TSMC 28nm library using Cadence Genus 2023. Our clock frequency is set to 1 GHz, and the cell libraries for low leakage power are utilized. Two HBM3 memories [35, 23, 34, 29] are utilized for off-chip storage, offering a bandwidth of $\approx$ 2TB/s. For loading data onto the HBM3, 32 lanes PCIe5 [39] offering a bandwidth of 128 GB/s

Table 3: Performance micro-benchmarks for FHE accelerator running at 1.2GHz.

Operation	**Level**	**Time** (ms)
ADD/MULT (pt)	31	0.025
MULT (ct)	31	0.025
MAC (ct)	31	0.05
Automorphism	31	0.025
Relinearization	31→30	1.05
Fully packed Bootstrapping	1→31→16	71

Table 4: Comparison with the existing works. All the results are normalized to 7nm technology [33]. w represents RNS moduli size.

Work$_\omega$	**Area** (mm^2)	**Avg. Power** (W)	$\mathbf{T}_{A.S.}$ (ns)	**Area**×**Time**/ω
$F1_{32}$ [18]	71.02	28.5	470	56.3
BTS_{64} [28]	373.06	163.2	45.3	14.3
ARK_{64} [27]	418.3	135	14.3	5.05
$CraterLake_{28}$ [38]	222.7	124	17.6	7.56
SH_{64} [26]	325.4	187	11.7	3.2
Our$_{54}$	**60.3**	**50.1**	**144.45**	**8.71**

can be utilized. Table 2 presents the area consumption of the proposed accelerator design as well as its respective building blocks. We have sufficient on-chip memory for storing 12 polynomials (in MV-NTT) other than the ones required for MV-NTT/INTT intermediate storage. This memory is realized solely using SRAMs instead of register files to reduce the area as well as power consumption.

Benchmarking: For performance modelling, we use cycle-accurate simulation technique as employed by the previous works in literature [27, 38]. We have the cycle-accurate results of our building blocks (MV-NTT/MACs/Automorphism), using which we estimate the runtime of operations such as Relinearization. This is then further utilized in estimating the runtime of higher-level operations such as bootstrapping. Note that the schedule for these operations is static and decoupled. Application benchmarking utilizes all these static routines; therefore, for estimating its runtime, we use the schedule followed by OpenFHE [4].

Table 3 displays the performance results of our accelerator (in ms) for various CKKS operations, and comparisons with previous FHE acceleration works can be found in Table 4. In terms of average power and area, our design outperforms other works. However, it comes with a higher computation time; this trade-off is expressed via the Area-Time product metric, placing our value between those of prior works. Despite the relatively low area of our MV-NTT/INTT unit, it does not offer the optimal trade-off due to the substantial on-chip memory requirement, which is more expensive than the compute logic within the unit. To address this, our design incorporates four MV-NTT/INTT units. This approach allows prompt processing of the results in the MV-NTT domain and transmitting them back to memory. Simultaneously, the MV-NTT/INTT unit can focus on handling

other polynomials without the need for prolonged on-chip storage. Nevertheless, the requirement of on-chip storage for the polynomials cannot be completely eliminated and MV-NTT-processed larger polynomials necessitate more on-chip memory compared to accelerators employing conventional NTT/INTT units.

Logistic regression training and inference on the iDASH model [25] takes 9.5ms and 1.15ms, respectively. On the CPU (using OpenFHE [4] on a 24-core, 2× Intel XEON CPU @3.47GHZ with 192GB DDR3 RAM), the same applications consume 11.18s and 1.27s. Hence, our design achieves $\approx 1,200\times$ the speedup compared to plain software implementation.

7 Discussions and conclusions

Starting with the initial curiosity of *designing a cheap polynomial multiplication circuit* for RLWE-based homomorphic encryption schemes, we explored the non-conventional method of performing NTTs with Fermat numbers (FNTT) as auxiliary modulus instead of different RNS-based moduli. Although FNTTs were discussed in prior works several decades ago, this technique was not readily usable in HE schemes because of very large polynomial degrees N. Thus, we focused on the second question, *can FNTT be used to design an FHE accelerator?* To tackle the problem of an increased coefficient-size proportional to the size of Fermat number (which in turn limits the dimension of the polynomial that the FNTT can process), we proposed to switch from a univariate ring structure $\mathcal{R}_q[X]$ with large N to an intermediate multivariate structure $\mathcal{R}'_q[X_1, \cdots, X_k]$. We proposed the multivariate NTT of MV-NTT, which uses polynomials of small-degrees N_i's with $N_i|N$, enabling efficient FNTT. Furthermore, we presented an MV-NTT architecture and assessed its effectiveness within an FHE accelerator. The designed FHE accelerator achieves a speed-up of 1,200× compared to CPU computations for applications requiring deep multiplicative depths. In the following part of this section, we highlight the potentials and limitations of our approach to serve as a guiding reference for any future works willing to explore alternative polynomial multiplication methods in the context of RLWE-based homomorphic encryption.

7.1 Limitations

The main limitations of the proposed technique are caused by the increase in polynomial size when MV-NTT operation is performed due to multivariate ring setting (e.g., $N_3 \cdot N_2 \cdot N_1 \approx 4N$ for three variables) and the use of a comparatively larger Fermat number as an auxiliary modulus instead of smaller RNS modulus ($F_K > q_i^2$). In Sec. 3.3, we show that the proposed MV-NTT reduces the total gate evaluations compared to conventional NTT, even though it uses a larger modulus compared to conventional NTT. However, this analysis only focuses on the absolute computation in MV-NTT and excludes on-chip memory requirements, which becomes prominent when the proposed MV-NTT is utilized in a higher-level application, such as hardware acceleration of FHE. Thus, the

huge on-chip memory and bandwidth requirements of the proposed MV-NTT hinder its effective utilization in accelerator systems. In our proposed FHE accelerator, we use extra MV-NTT/INTT units to overcome the bandwidth bottleneck. An intriguing avenue for future research would be to explore more efficient univariate-to-multivariate transformations that maintain $\prod N_i \approx N$.

7.2 Potentials of MV-NTT in emerging PIM architectures

Processing-in-Memory (PIM) is a new way of computing that uses advanced memory technologies like Resistive-RAM (ReRAM) or Phase-Change Memory (PCM) to handle data processing right within the memory. PIM platforms have shown potential to make energy use more efficient, especially in applications involving a lot of data, like Machine Learning. However, PIM platforms have very limited computational capabilities [40], and implementing modular multiplication becomes expensive as it must be computed using bit-level operations.

The proposed approach, utilizing FNTT-based MV-NTT, presents a potential avenue for implementing FHE within PIM environments. Firstly, the considerable data bandwidth requirements associated with MV-NTT are resolved within PIM, where data movements occur internally within the memory. Notably, to align with the off-chip bandwidth requirement, the ASIC-based FHE accelerator in Sec. 5 maintained the ciphertexts in the polynomial domain rather than transitioning them to the MV-NTT domain. This introduced a performance bottleneck by increasing the number of MV-NTTs and MV-INTTs. However, in a PIM context, where bandwidth constraints are less pronounced, storing ciphertexts in the NTT domain while employing MV-NTT could alleviate the bottleneck, reducing the required number of NTT units from four to one. Secondly, MV-NTT does not require expensive multipliers as it replaces them with simple shifts. This property of MV-NTT aligns very well with PIM, where expensive multiplier circuits are unavailable.

8 Acknowledgements

This work was supported in part by Samsung Electronics co. ltd., Samsung Advanced Institute of Technology and the State Government of Styria, Austria – Department Zukunftsfonds Steiermark.

References

1. Agarwal, R.C., Burrus, C.S.: Number theoretic transforms to implement fast digital convolution. Proceedings of the IEEE **63**(4), 550–560 (1975)
2. Agrawal, R., de Castro, L., Yang, G., Juvekar, C., Yazicigil, R.T., Chandrakasan, A.P., Vaikuntanathan, V., Joshi, A.: FAB: an fpga-based accelerator for bootstrappable fully homomorphic encryption. In: IEEE International Symposium on High-Performance Computer Architecture, HPCA 2023, Montreal, QC, Canada, February 25 - March 1, 2023. pp. 882–895. IEEE (2023).

https://doi.org/10.1109/HPCA56546.2023.10070953, https://doi.org/10.1109/HPCA56546.2023.10070953
3. Aikata, A., Mert, A.C., Kwon, S., Deryabin, M., Roy, S.S.: Reed: Chiplet-based scalable hardware accelerator for fully homomorphic encryption. arXiv preprint arXiv:2308.02885 (2023)
4. Al Badawi, A., Bates, J., Bergamaschi, F., Cousins, D.B., Erabelli, S., Genise, N., Halevi, S., Hunt, H., Kim, A., Lee, Y., Liu, Z., Micciancio, D., Quah, I., Polyakov, Y., R.V., S., Rohloff, K., Saylor, J., Suponitsky, D., Triplett, M., Vaikuntanathan, V., Zucca, V.: OpenFHE: Open-Source Fully Homomorphic Encryption Library. In: Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography. pp. 53–63. WAHC'22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3560827.3563379, https://doi.org/10.1145/3560827.3563379
5. Barrett, P.: Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In: Advances in Cryptology—CRYPTO'86: Proceedings. pp. 311–323. Springer (2000)
6. Bernstein, D.J., Chou, T.: Faster binary-field multiplication and faster binary-field macs. In: International Conference on Selected Areas in Cryptography. pp. 92–111. Springer (2014)
7. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: Fully homomorphic encryption without bootstrapping. Electron. Colloquium Comput. Complex. p. 111 (2011), https://eccc.weizmann.ac.il/report/2011/111
8. Brakerski, Z., Vaikuntanathan, V.: Efficient fully homomorphic encryption from (standard) lwe. In: 2011 IEEE 52nd Annual Symposium on Foundations of Computer Science. pp. 97–106 (2011). https://doi.org/10.1109/FOCS.2011.12
9. Brakerski, Z., Vaikuntanathan, V.: Fully homomorphic encryption from ring-lwe and security for key dependent messages. In: Rogaway, P. (ed.) Advances in Cryptology – CRYPTO 2011. pp. 505–524. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
10. Chen, H., Laine, K., Rindal, P.: Fast Private Set Intersection from Homomorphic Encryption. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. p. 1243–1255. CCS '17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3133956.3134061, https://doi.org/10.1145/3133956.3134061
11. Cheon, J.H., Han, K., Kim, A., Kim, M., Song, Y.: Bootstrapping for approximate homomorphic encryption. In: Nielsen, J.B., Rijmen, V. (eds.) Advances in Cryptology – EUROCRYPT 2018. pp. 360–384. Springer International Publishing, Cham (2018)
12. Cheon, J.H., Han, K., Kim, A., Kim, M., Song, Y.: A full rns variant of approximate homomorphic encryption. In: Selected Areas in Cryptography–SAC 2018: 25th International Conference, Calgary, AB, Canada, August 15–17, 2018, Revised Selected Papers 25. pp. 347–368. Springer (2019)
13. Chung, C.M.M., Hwang, V., Kannwischer, M.J., Seiler, G., Shih, C.J., Yang, B.Y.: NTT Multiplication for NTT-unfriendly Rings: New Speed Records for Saber and NTRU on Cortex-M4 and AVX2. IACR Transactions on Cryptographic Hardware and Embedded Systems **2021**(2), 159–188 (Feb 2021). https://doi.org/10.46586/tches.v2021.i2.159-188, https://tches.iacr.org/index.php/TCHES/article/view/8791
14. Cook, S.A., Aanderaa, S.O.: On the minimum computation time of functions. Transactions of the American Mathematical Society **142**, 291–314 (1969)

15. Dai, W., Sunar, B.: cuhe: A homomorphic encryption accelerator library. In: International Conference on Cryptography and Information Security in the Balkans. pp. 169–186. Springer (2015)
16. Dimitrov, V., Cooklev, T., Donevsky, B.: Generalized fermat-mersenne number theoretic transform. Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on **41**, 133 – 139 (03 1994). https://doi.org/10.1109/82.281844
17. Fan, J., Vercauteren, F.: Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Paper 2012/144 (2012), `https://eprint.iacr.org/2012/144`, `https://eprint.iacr.org/2012/144`
18. Feldmann, A., Samardzic, N., Krastev, A., Devadas, S., Dreslinski, R., Eldefrawy, K., Genise, N., Peikert, C., Sanchez, D.: F1: A fast and programmable accelerator for fully homomorphic encryption (extended version) (2021)
19. Geelen, R., Van Beirendonck, M., Pereira, H.V.L., Huffman, B., McAuley, T., Selfridge, B., Wagner, D., Dimou, G., Verbauwhede, I., Vercauteren, F., Archer, D.W.: BASALISC: Flexible Asynchronous Hardware Accelerator for Fully Homomorphic Encryption (2022). https://doi.org/10.48550/ARXIV.2205.14017, `https://arxiv.org/abs/2205.14017`
20. Gentry, C.: A Fully Homomorphic Encryption Scheme. Ph.D. thesis, Stanford, CA, USA (2009)
21. Gil, Y., Jiang, X., Kim, M., Lee, J.: Secure and differentially private bayesian learning on distributed data. CoRR **abs/2005.11007** (2020), `https://arxiv.org/abs/2005.11007`
22. Hirner, F., Mert, A.C., Roy, S.S.: Proteus: A tool to generate pipelined number theoretic transform architectures for fhe and zkp applications. Cryptology ePrint Archive (2023)
23. JEDEC: High Bandwidth Memory DRAM (HBM3). Tech. Rep. JESD238 (2022)
24. Karatsuba, A.A., Ofman, Y.P.: Multiplication of many-digital numbers by automatic computers. In: Doklady Akademii Nauk. vol. 145, pp. 293–294. Russian Academy of Sciences (1962)
25. Kim, A., Song, Y., Kim, M., Lee, K., Cheon, J.: Logistic regression model training based on the approximate homomorphic encryption. BMC Medical Genomics **11** (10 2018). https://doi.org/10.1186/s12920-018-0401-7
26. Kim, J., Kim, S., Choi, J., Park, J., Kim, D., Ahn, J.H.: SHARP: A short-word hierarchical accelerator for robust and practical fully homomorphic encryption. In: Solihin, Y., Heinrich, M.A. (eds.) Proceedings of the 50th Annual International Symposium on Computer Architecture, ISCA 2023, Orlando, FL, USA, June 17-21, 2023. pp. 18:1–18:15. ACM (2023). https://doi.org/10.1145/3579371.3589053, `https://doi.org/10.1145/3579371.3589053`
27. Kim, J., Lee, G., Kim, S., Sohn, G., Kim, J., Rhu, M., Ahn, J.H.: ARK: Fully Homomorphic Encryption Accelerator with Runtime Data Generation and Inter-Operation Key Reuse (2022). https://doi.org/10.48550/ARXIV.2205.00922, `https://arxiv.org/abs/2205.00922`
28. Kim, S., Kim, J., Kim, M.J., Jung, W., Kim, J., Rhu, M., Ahn, J.H.: BTS: An Accelerator for Bootstrappable Fully Homomorphic Encryption. In: Proceedings of the 49th Annual International Symposium on Computer Architecture. p. 711–725. ISCA '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3470496.3527415, `https://doi.org/10.1145/3470496.3527415`
29. Lee, D., Lee, K.S., Lee, Y., Kim, K.W., Kang, J., Lee, J., Chun, J.H.: Design considerations of HBM stacked DRAM and the memory architecture extension. In: 2015 IEEE Custom Integrated Circuits Conference,

CICC 2015, San Jose, CA, USA, September 28-30, 2015. pp. 1–8. IEEE (2015). https://doi.org/10.1109/CICC.2015.7338357, https://doi.org/10.1109/CICC.2015.7338357

30. Longa, P., Naehrig, M.: Speeding up the number theoretic transform for faster ideal lattice-based cryptography. In: Cryptology and Network Security: 15th International Conference, CANS 2016, Milan, Italy, November 14-16, 2016, Proceedings 15. pp. 124–139. Springer (2016)
31. Mert, A.C., Kwon, S., Shin, Y., Yoo, D., Lee, Y., Roy, S.S., et al.: Medha: Microcoded hardware accelerator for computing on encrypted data. arXiv preprint arXiv:2210.05476 (2022)
32. Mert, A.C., Ozturk, E., Savas, E.: Design and implementation of encryption/decryption architectures for bfv homomorphic encryption scheme. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **28**(2), 353–362 (2020). https://doi.org/10.1109/TVLSI.2019.2943127
33. Narasimha, S., Jagannathan, B., Ogino, A., Jaeger, D., Greene, B., Sheraw, C., Zhao, K., Haran, B., Kwon, U., Mahalingam, A.K.M., Kannan, B., Morganfeld, B., Dechene, J., Radens, C., Tessier, A., Hassan, A., Narisetty, H., Ahsan, I., Aminpur, M., An, C., Aquilino, M., Arya, A., Augur, R., Baliga, N., Bhelkar, R., Biery, G., Blauberg, A., Borjemscaia, N., Bryant, A., Cao, L., Chauhan, V., Chen, M., Cheng, L., Choo, J., Christiansen, C., Chu, T., Cohen, B., Coleman, R., Conklin, D., Crown, S., da Silva, A., Dechene, D., Derderian, G., Deshpande, S., Dilliway, G., Donegan, K., Eller, M., Fan, Y., Fang, Q., Gassaria, A., Gauthier, R., Ghosh, S., Gifford, G., Gordon, T., Gribelyuk, M., Han, G., Han, J., Han, K., Hasan, M., Higman, J., Holt, J., Hu, L., Huang, L., Huang, C., Hung, T., Jin, Y., Johnson, J., Johnson, S., Joshi, V., Joshi, M., Justison, P., Kalaga, S., Kim, T., Kim, W., Krishnan, R., Krishnan, B., Anil, K., Kumar, M., Lee, J., Lee, R., Lemon, J., Liew, S., Lindo, P., Lingalugari, M., Lipinski, M., Liu, P., Liu, J., Lucarini, S., Ma, W., Maciejewski, E., Madisetti, S., Malinowski, A., Mehta, J., Meng, C., Mitra, S., Montgomery, C., Nayfeh, H., Nigam, T., Northrop, G., Onishi, K., Ordonio, C., Ozbek, M., Pal, R., Parihar, S., Patterson, O., Ramanathan, E., Ramirez, I., Ranjan, R., Sarad, J., Sardesai, V., Saudari, S., Schiller, C., Senapati, B., Serrau, C., Shah, N., Shen, T., Sheng, H., Shepard, J., Shi, Y., Silvestre, M., Singh, D., Song, Z., Sporre, J., Srinivasan, P., Sun, Z., Sutton, A., Sweeney, R., Tabakman, K., Tan, M., Wang, X., Woodard, E., Xu, G., Xu, D., Xuan, T., Yan, Y., Yang, J., Yeap, K., Yu, M., Zainuddin, A., Zeng, J., Zhang, K., Zhao, M., Zhong, Y., Carter, R., Lin, C.H., Grunow, S., Child, C., Lagus, M., Fox, R., Kaste, E., Gomba, G., Samavedam, S., Agnello, P., Sohn, D.K.: A 7nm cmos technology platform for mobile and high performance compute application. In: 2017 IEEE International Electron Devices Meeting (IEDM). pp. 29.5.1–29.5.4 (2017). https://doi.org/10.1109/IEDM.2017.8268476
34. Park, M., Lee, J., Cho, K., Park, J.H., Moon, J., Lee, S., Kim, T., Oh, S., Choi, S., Choi, Y., Cho, H.S., Yun, T., Koo, Y.J., Lee, J., Yoon, B.K., Park, Y.J., Oh, S., Lee, C.K., Lee, S., Kim, H., Ju, Y., Lim, S., Lee, K.Y., Lee, S., We, W.S., Kim, S., Yang, S.M., Lee, K., Kim, I., Jeon, Y., Park, J., Yun, J.C., Kim, S., Lee, D., Oh, S., Shin, J., Lee, Y., Jang, J., Cho, J.: A 192-Gb 12-High 896-GB/s HBM3 DRAM With a TSV Auto-Calibration Scheme and Machine-Learning-Based Layout Optimization. IEEE J. Solid State Circuits **58**(1), 256–269 (2023). https://doi.org/10.1109/JSSC.2022.3193354, https://doi.org/10.1109/JSSC.2022.3193354

35. Rambus: HBM3 Memory: Break Through to Greater Bandwidth, https://go.rambus.com/hbm3-memory-break-through-to-greater-bandwidth
36. Rivest, R.L., Adleman, L., Dertouzos, M.L.: On data banks and privacy homomorphisms. Foundations of Secure Computation, Academia Press pp. 169–179 (1978)
37. Roy, D.B., Mukhopadhyay, D., Izumi, M., Takahashi, J.: Tile before multiplication: An efficient strategy to optimize dsp multiplier for accelerating prime field ecc for nist curves. In: 2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC). pp. 1–6 (2014). https://doi.org/10.1145/2593069.2593234
38. Samardzic, N., Feldmann, A., Krastev, A., Manohar, N., Genise, N., Devadas, S., Eldefrawy, K., Peikert, C., Sanchez, D.: CraterLake: A hardware accelerator for efficient unbounded computation on encrypted data. In: Proceedings of the 49th Annual International Symposium on Computer Architecture. p. 173–187. ISCA'22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3470496.3527393, https://doi.org/10.1145/3470496.3527393
39. Sun, Y., Yuan, Y., Yu, Z., Kuper, R., Jeong, I., Wang, R., Kim, N.S.: Demystifying CXL memory with genuine cxl-ready systems and devices. CoRR **abs/2303.15375** (2023). https://doi.org/10.48550/arXiv.2303.15375, https://doi.org/10.48550/arXiv.2303.15375
40. Wan, W., Kubendran, R., Schaefer, C.J.S., Eryilmaz, S.B., Zhang, W., Wu, D., Deiss, S.R., Raina, P., Qian, H., Gao, B., Joshi, S., Wu, H., Wong, H.P., Cauwenberghs, G.: A compute-in-memory chip based on resistive random-access memory. Nat. **608**(7923), 504–512 (2022). https://doi.org/10.1038/s41586-022-04992-8, https://doi.org/10.1038/s41586-022-04992-8

A Field Arithmetic Units

Integer multiplier: Although our technique eliminates integer multiplications during MV-NTT/INTT operation by transforming them into a cost-effective left-shift operation, coefficient-wise multiplication or MAC operations after NTT (i.e., line 6 of Algorithm 6) necessitates the use of large $(K+1)$-bit integer multiplier. Our target parameter set mandates a 129-bit integer multiplier which can be implemented using various methods including tiling [37], Karatsuba [24], and Toom-Cook [14]. Although Toom-Cook offers better time complexity, it is unsuitable for hardware realization due to its reliance on division operations. Consequently, we opted for a two-level Karatsuba method that implements a large 129-bit multiplication using multiple smaller multiplier units. Specifically, the proposed multiplier implementation uses five 32-bit multipliers, three 33-bit multipliers and one 34-bit multiplier.

Modular reduction unit for modulo Fermat number: After integer multiplication, a modular reduction operation is necessary to bring the multiplication result back to $\mathbb{Z}_{F_K}$. The modular reduction algorithm for $F_K = 2^K + 1$ is shown in Algorithm 7. The proposed reduction algorithm uses relation $2^K \equiv -1 \pmod{F_K}$ to perform the reduction operation. For two integers $x, y \in [0, F_K - 1]$, the multiplication $x \cdot y$ could be at most $x_{max} \cdot y_{max} = 2^K \cdot 2^K = 2^{2 \cdot K}$. Similarly, a K-pt FNTT/IFNTT operation will require an integer $x \in [0, F_K - 1]$ to be shifted

Algorithm 7 Modular Reduction Algorithm for $F_K = 2^K + 1$

In: $a \leq 2^{2 \cdot K}$, a $(2 \cdot K + 1)$-bit integer
Out: $b = a \pmod{2^K + 1} \in [0, 2^K]$, a $(K + 1)$ bit integer

1: $a_l = a \pmod{2^K}$
2: $a_h = a \gg K$
3: $t = a_l - a_h$ ▷ a_h is at most 2^K, so t_{min} is -2^K
4: $s = t + F_K$
5: $b = (t_{sign} == 1)\ ?\ s\ :\ t$ ▷ t_{sign} is the sign bit of t
6: **return** b

Algorithm 8 Tailored Word-level Montgomery Modular Reduction

Input: $d \in \mathbb{Z}_F$, $q = M \cdot 2^{23} + 1$
Out: $c = d \cdot 2^{-23 \cdot 4} \pmod{q}$

1: $T \leftarrow d$
2: **for** $(i = 0; i < 3; i = i + 1)$ **do**
3: $T_H, T_L \leftarrow T \gg 23, T \pmod{2^{23}}$
4: $T2 \leftarrow -T_L \pmod{2^{23}}$, $cin \leftarrow T2[23 - 1] \vee T_L[23]$
5: $T \leftarrow (M \cdot T2) + T_H + cin$
6: **return** $c \leftarrow (T \geq q)\ ?\ T - q\ :\ T$

to the left by at most K, which can yield an integer at most $x_{max} \ll K = 2^{2 \cdot K}$. Thus, the proposed modular reduction algorithm assumes that the input integer will be in the range $[0, 2^{2 \cdot K}]$. This also shows that only one addition with F_K is required after the reduction operation to guarantee a positive integer result (see line 4 of Algorithm 7). The proposed modular reduction unit for F_K is implemented using one subtractor, one adder and one 2-to-1 multiplexer.

Lifting unit: As shown in line 1 of Algorithm 2, MV-NTT operation requires centered input coefficients in $\mathbb{Z}_{q_i}$ to be lifted to $\mathbb{Z}_{F_K}$. Similarly, MV-INTT operation requires output coefficients to be lifted from $\mathbb{Z}_{F_K}$ to $\mathbb{Z}_{q_i}$. The former is very easy to implement as $F_K > q_i$ while the latter will require reducing a $(K + 1)$-bit integer (i.e., in $\mathbb{Z}_{F_K}$) to $\mathbb{Z}_{q_i}$. The size of integers in $\mathbb{Z}_{F_K}$ is more than 2 times larger compared to the integers in $\mathbb{Z}_{q_i}$, which complicates the reduction operation. For our target parameter set, the size of F_K is 129-bit and the size of RNS modulus q_i is 54-bit. We adopted the word-level Montgomery reduction algorithm [32] and tailored it for primes in form $M \cdot 2^2 3 + 1$, where M is at most a 31-bit integer. We set the number of reduction step as $L = 4$ so the reduction circuit can support any prime size, $\log_2(q_i)$, between 37-bit and 59-bit (i.e., $\lceil (129 - 37)/23 \rfloor = 4$ and $\lceil (129 - 59)/23 \rfloor = 4$). This method simplifies the modular reduction unit as it avoids large multiplier units. The tailored word-level Montgomery algorithm is shown in Algorithm 8. We implemented a unified lifting unit that can perform lifting operations for both MV-NTT and MV-INTT as shown in Fig. 9.

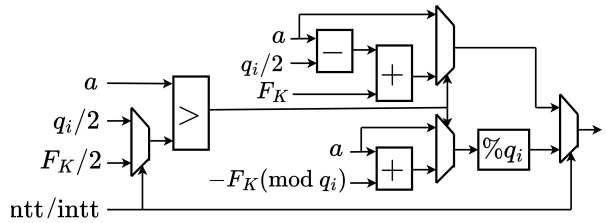

Fig. 9: Unified lifting unit. The input a is lifted from centered $\mathbb{Z}_{q_i}$ to $\mathbb{Z}_{F_K}$ before FNTT while it is lifted from centered $\mathbb{Z}_{F_K}$ to $\mathbb{Z}_{q_i}$ before IFNTT.