cuTraNTT: GPU-based Transposed Number Theoretic Transform with Low Latency Homomorphic Encryption for IoT Applications

Supriya Adhikary, *Student Member, IEEE,* Wai Kong Lee, *Member, IEEE,* Angshuman Karmakar, *Member, IEEE,* Yongwoo Lee, *Member, IEEE,* Seong Oun Hwang, *Senior Member, IEEE,* Ramachandra Achar, *Fellow, IEEE,*

***Abstract*—Large polynomial multiplication is one of the computational bottlenecks in fully homomorphic encryption implementations. Usually, these multiplications are implemented using the number-theoretic transformat to speed up the computation. State-of-the-art GPU-based implementation of fully homomorphic encryption computes the number theoretic transform in two different kernels, due to the necessary synchronization between GPU blocks to ensure correctness in computation. This can be a serious limitation in embedded systems that only have constrained computational resources to support the time-consuming homomorphic encryption. In this paper, we proposed a series of techniques to improve the performance of number theoretic transform targeting homomorphic encryption on a GPU device. Firstly, we proposed to arrange the polynomials in a transposed manner and skip the last two levels of radix-4 number theoretic transform, allowing us to completely avoid the block synchronization in NTT implementation. This technique improved the performance of homomorphic encryption by $1.37\times$ and $1.34\times$ on RTX 4060 and Jetson Orin Nano respectively, compared to the conventional approach that uses full NTT without skipping any levels. However, such an approach also introduces extra overhead in the subsequent point-wise multiplication, which slows down the homomorphic multiplication. To reduce this negative impact, a fast 16×16 point-wise multiplication implementation was proposed, which relies on the heavily optimized Toom-Cook 4-way algorithm. Experimental results show that our proposed homomorphic multiplication can achieve similar latency compared to Jung et al. and Yang et al., which are the best results to date. This shows that the proposed cuTraNTT is able to reduce the latency of homomorphic encryption without sacrificing the performance in homomorphic multiplication.**

***Index Terms*—Graphics processing units, fully homomorphic encryption, polynomial multiplication, and number theoretic transform.**

I. Introduction

The rapid growth of web technologies and digital communication in the past few decades has led to a huge increase in data and the development of data processing technologies. This has resulted in a data-driven era where use of machine learning, neural networks, and artificial intelligence based methods have been used to hyperdrive the innovations in the field of information processing. However, often these technologies rely on massive computations performed on this data. Delegated computing technologies such as cloud computing provide users with the flexibility to store their data and run data manipulation routines on their data. However, the storage of data in the cloud is not always reliable as there is a risk of data leakage and compromise of user privacy. A 2021 Statista survey [1] found that 64 percent of respondents were most concerned about data leakage when it came to cloud privacy concerns. Therefore, ethical handling of data is crucial to ensure user privacy is protected.

Laws like The Digital Personal Data Protection Bill of India, the European Union's GDPR, the Data Security Regulations of Israel, the Act on the Protection of Personal Information of Japan, etc., propose laws and guidelines for ethical handling of data, but these laws are bound by their specific jurisdictions and have limited effect outside some specific geographical region. While traditional cryptography provides security for data at rest and data in motion, but fails to provide security for data in use. Cryptographic techniques such as Computation Over Encrypted Data (COED) are the only solutions with provable security to achieve data privacy for data in use. These techniques allow running data manipulation routines on encrypted data. Thus protecting the privacy and integrity of the underlying data while allowing applications to process the data as needed. COED consists of Homomorphic Encryption (HE), Functional Encryption (FE), and Multi-party Computation (MPC) techniques. Our research mainly focuses on exploring HE for privacy-preserving solutions in the context of Fully Homomorphic Encryption (FHE) for delegated computation.

FHE has attracted a lot of attention in the past decade. Most FHE instantiations are based on the Ring-Learning with errors [2]. A major operation for such types of FHE is the multiplication between two large polynomials of degree 2^{16} where each coefficient can be as large as 1600 bits. Such large multiplications often constitute the bottleneck in practical deployment of FHE schemes. Therefore, various acceleration techniques have been proposed to speed up the execution of FHE on different hardware platforms, including ASIC [3], FPGA [4] and GPU [5]–[9]. However, GPUs present a strong basis for FHE implementations. GPUs provide a versatile platform for implementing and optimizing homomorphic encryption algorithms, making them a preferred choice for many applications in this domain. Furthermore, given that

Supriya Adhikary and Angshuman Karmakar are with IIT Kanpur, India.
Wai Kong Lee is with Universiti Tunku Abdul Rahman, Malaysia.
Yongwoo Lee is with Inha University, South Korea.
Seong Oun Hwang is with Gachon University, South Korea.
Ramachandra Achar is with Carleton University, Canada.
Manuscript received April 19, 2021; revised August 16, 2021.

many cloud servers already employ GPUs to expedite AI and machine learning tasks, the integration of homomorphic encryption into these systems would be notably seamless. The field of FHE is highly evolving and any change in security parameters can be done easily in GPU, unlike other hardware accelerators that require low-level hardware coding and configuration.

To deploy FHE on the IoT sensor nodes is a challenging task due to the heavy computations, especially the polynomial multiplication. There is only one prior work that focuses on this direction, which was performed by Natarajan and Dai [10], but they mainly discuss the memory optimization techniques in FHE for deployment in embedded devices. Our work differs from theirs as we focus on the implementation of number theoretic transform (NTT), which greatly affects the computational time in FHE. This can allow FHE to be deployed on IoT sensor nodes with a faster homomorphic encryption speed compared to the prior work.

In this paper, we present several techniques to improve the performance of CKKS [11] FHE targeting implementation on embedded devices equipped with GPU. However, it is important to note that the proposed method can also be applied to other RLWE-based FHE schemes such as BGV and BFV. The contributions are summarized below:

1) NTT is the most time-consuming operation in CKKS FHE scheme. To mitigate this bottleneck, we proposed an efficient implementation of NTT on GPU by employing radix-4 Cooley-Tukey algorithm. On top of that, our NTT implementation is solely based on 32-bit arithmetic, because GPU is a 32-bit architecture and naturally supports 32-bit arithmetic. This is in contrast to existing implementation that are mostly based on 64-bit arithmetic [7], [9]. Our radix-4 NTT implementation is 2% faster than [7] on V100 and 1.2% faster than [9] on A100.
2) Previous implementation of NTT on GPU platforms requires two separate kernels, which is not optimal due to the mandatory synchronization between two kernels. In this paper, we proposed cuTraNTT, a novel technique that skips two levels of NTT and arranges the polynomials in a transposed manner, thus allowing us to combine the two NTT kernels into one. This eliminates the block-wise synchronization found in previous works [7], [9] and greatly reduces the latency of NTT. Our cuTraNTT implementation is 6% and 2% faster than [7] and [9], respectively.
3) Skipping two levels of radix-4 NTT introduces extra overhead, whereby the original N point-wise multiplication becomes $N/16$ many 16×16 polynomial multiplication. To reduce the impact of this overhead, we proposed an optimized parallel implementation of Toom-Cook 4-way algorithm for the 16×16 polynomial multiplication. Experimental results show that our proposed parallel Toom-Cook 4 implementation is 29.25% faster than the original 16×16 polynomial multiplication.
4) By incorporating these optimization techniques, our implementation of homomorphic encryption (HENC) is $1.37\times$ faster than the baseline version. However, our approach does not provide significant gain for homomorphic multiplication (HMULT), due to the overhead introduced by skipping two levels of NTT. Our HMULT implementation is only 1.2% faster than [9] on an A100 GPU, but 2% slower than [7] on a V100 GPU. Note that although our work was developed mainly for GPU implementation, it also benefits an embedded device without a GPU. This is because the NTT proposed in this work is still more efficient than the conventional NTT, due to skipping the last two levels. Hence, this technique can also be used in microcontrollers [10].

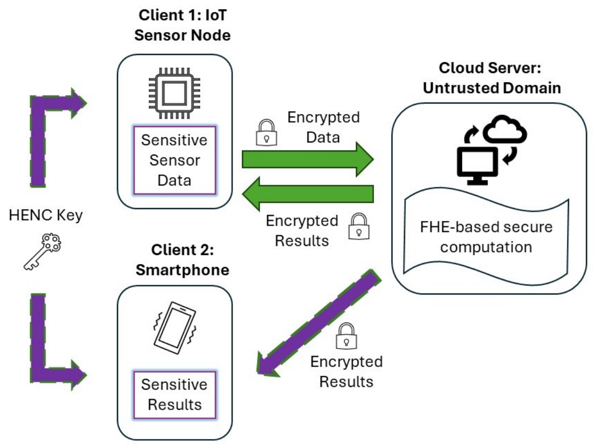


Fig. 1. Secure communication flow for IoT applications protected by FHE.

Fig. 1 shows the communication flow of IoT applications secured by FHE. Client 1 represents the sensor nodes in an IoT system; it is responsible in collecting sensor data, encrypting the sensitive information, and sharing it with the cloud server. Main computation (e.g., machine learning) is performed on the cloud server, which is considered as untrusted domain. The results of computation can be sent to the Client 1 and 2 for decryption, in which Client 1 must share a homomorphic encryption (HENC) key with Client 2. Due to the efficient NTT implementation, our work is especially useful for the client side, because the latency of HENC is greatly reduced. This is especially important for IoT applications, wherein the clients are implemented using embedded devices that do not have computational power compared to a desktop workstation. For instance, a Jetson Orin Nano equipped with an embedded GPU can utilize our solution to perform HENC more efficiently than the baseline version.

Note that in this work, We focused on the CKKS parameter with $N = 2^{16}$ because it allows bootstrapping to be performed. A parameter smaller than that (e.g., $N = 2^{15}$) cannot support bootstrapping, and a bigger parameter is usually not necessary as we are already able to evaluate deep circuit with bootstrapping using parameter $N = 2^{16}$.

II. Background

A. Notations for Fully Homomorphic Encryption

This subsection presents the notations used to describe the FHE operations in this paper. Given a polynomial ring $\mathcal{R} = \mathbb{Z}[X]/\langle X^N + 1\rangle$, where N is a power-of-two integer, the residue ring modulo an integer q is denoted as $\mathcal{R}_q = \mathcal{R}/q\mathcal{R}$. χ_s and χ_e are the secret key distribution and error distribution over $\mathcal{R}$, respectively. Given a set S, $a \xleftarrow{\$} S$ refers to the process of sampling a uniformly or distribution S. For an integer n, we denote the set $\{i \mid 1 \le i \le n\}$ as $[n]$. $[\cdot]_q$ denotes the modular reduction by q. For any polynomial $m(X) = \sum_{i=0}^{N-1} m_i X^i \in \mathcal{R}$, we define $[m(X)]_q = \sum_{i=0}^{N-1}[m_i]_q X^i$. For a set of moduli $\mathcal{C} = \{q_0, q_1, \cdots q_k\}$, where $\{q_i\}_{0\le i<k}$ are all coprime to each other, we define $[s(X)]_{\mathcal{C}} = ([s(X)]_{q_0}, \cdots, [s(X)]_{q_k}) \in \prod_{i=0}^{k} \mathcal{R}_{q_i}$. We denote Φ_n to be a primitive $2n$-th root of unity.

B. Overview of RNS-CKKS

There are four major types of operations in CKKS:

- Element-wise RNS operations between polynomials
- NTT and INTT
- Standard basis conversion used in `ModUp` and `ModDown`
- Key-switching and HMULT

1) RNS operation: In CKKS, all polynomials are in the ring $\mathcal{R}_q$, where q is an interger. For RNS-CKKS, we take q as a product of multiple primes say $q = q_0 \cdot q_1 \cdots q_{\ell-1}$ and given a polynomial $a \in \mathcal{R}_q$, we consider the RNS-representation of the polynomial $(a^{(i)})_{i=0}^{\ell-1} \in \prod_{i=0}^{\ell-1} R_{q_i}$. For operations like addition, subtraction and multiplication, we apply the operations element-wise[1].

2) Number Theoretic transform: For the multiplication of two polynomials in $\mathcal{R}_q$, where q is prime, the Number Theoretic transform (NTT) and its inverse (iNTT) are used. For any prime q satisfying $q \equiv 1 \pmod{2N}$, NTT is a mapping

$$\text{NTT} : \mathcal{R}_q \to \prod_{i=0}^{N-1} \mathbb{Z}_q[X]/\langle X - \Phi_N^{2i+1}\rangle$$

defined as $\text{NTT}(f(X)) = (f(\Phi_N), f(\Phi_N^3), \cdots, f(\Phi_N^{2N-1}))$. The function iNTT takes $(f(\Phi_N), f(\Phi_N^3), \cdots, f(\Phi_N^{2N-1}))$ as input and returns $f(X)$. By converting a polynomial into its NTT form, the multiplication of two polynomials can be performed in the frequency domain using the Hadamard product, which is very efficient. After the Hadamard product, the results are converted back to ordinary form using INTT for further computation.

3) ModUp and ModDown: We describe RNS basis decomposition for the generalized key-switching and the basis conversion in CKKS. It is used in approximate modulus raising (ModUp) and approximate modulus reduction (ModDown), extending and shrinking the RNS basis of a polynomial, respectively.

Let $\mathcal{B} = \{p_0, \cdots, p_{k-1}\}$ and $\mathcal{C}_i = \{q_0, \cdots, q_{i-1}\}$ for $i \in [1, L]$, where $\{p_i\}_{i\in[0,k)}$, $\{q_j\}_{j\in[0,L)}$ are coprime to each other. Let $P = \prod_{p\in\mathcal{B}} p$ and $Q_i = \prod_{q\in\mathcal{C}_i} q$ for $i \in [1, L]$. Also consider $\hat{Q}_j''' = \prod_{q_i\in\mathcal{S}_{\mathcal{C}_L}} q_i \times \prod_{p_i\in\mathcal{S}_\mathcal{B}\wedge i\ne j} p_i$ and $\hat{Q}_j'' = \prod_{q_i\in\mathcal{S}_{\mathcal{C}_L}\wedge i\ne j} q_i \times \prod_{p_i\in\mathcal{S}_\mathcal{B}} p_i$. The Alg. 1 changes the basis of an input polynomial $[a(X)]_{\mathcal{S}_{\mathcal{C}_L}\cup\mathcal{S}_\mathcal{B}}$ to $\mathcal{S}'_{\mathcal{C}_L} \cup \mathcal{S}'_\mathcal{B}$ approximately.

Algorithm 1 $\text{Conv}_{\mathcal{S}_{\mathcal{C}_L}\cup\mathcal{S}_\mathcal{B}\to\mathcal{S}'_{\mathcal{C}_L}\cup\mathcal{S}'_\mathcal{B}}([a(X)]_{\mathcal{S}_{\mathcal{C}_L}\cup\mathcal{S}_\mathcal{B}})$ (Fast Basis Conversion)

```
Require: polynomial $a(X)$ under moduli basis $\mathcal{S}_{\mathcal{C}_L} \cup \mathcal{S}_\mathcal{B}$.
Ensure: polynomial $a(X)$ under moduli basis $\mathcal{S}'_{\mathcal{C}_L} \cup \mathcal{S}'_\mathcal{B}$.
1: for $q_i \in \mathcal{S}'_{\mathcal{C}_L} \cup \mathcal{S}'_\mathcal{B}$ do
2:     $[a_1(X)]_{q_i} \leftarrow [\sum_{q_j\in\mathcal{S}_{\mathcal{C}_L}} [[a(X)]_{q_j} \cdot \hat{Q}_j''^{-1}]_{q_j} \cdot [\hat{Q}_j'']_{q_i}]_{q_i}$
3:     $[a_2(X)]_{q_i} \leftarrow [\sum_{p_j\in\mathcal{S}_\mathcal{B}} [[a(X)]_{p_j} \cdot \hat{Q}_j'''^{-1}]_{p_j} \cdot [\hat{Q}_j''']_{q_i}]_{q_i}$
4:     $[\tilde{a}(X)]_{q_i} \leftarrow [a_1(X)]_{q_i} + [a_2(X)]_{q_i}$
5: end for
6: return $[\tilde{a}(X)]_{\mathcal{S}'_\mathcal{B}\cup\mathcal{S}'_{\mathcal{C}_L}}$
```

Given an integer $\mathtt{dnum}$, let $\alpha = \lceil L/\mathtt{dnum}\rceil$. The parameter $\mathtt{dnum}$ divides the basis $\mathcal{C}_L$ into the bases $\{\mathcal{C}'_i = \{q_j\}_{j\in[i\alpha, i\alpha+\alpha)}\}$. For a given level ℓ of a ciphertext, let $\beta = \lceil \ell/\alpha\rceil \le \mathtt{dnum}$. Before key-switching, CKKS first decomposes the RNS basis $\mathcal{C}_\ell$ into $\{\mathcal{C}'_i\}_{i\in[0,\beta)}$, each of which is extended to the basis $\mathcal{D}_\beta$, where $\mathcal{D}_i = \mathcal{B} \cup (\cup_{0\le j<i}\mathcal{C}'_j)$. After key-switching, the basis is reduced to $\mathcal{C}_\ell$.

We define $Q' = \prod_{i=\ell+1}^{\alpha\beta-1} q_i$, $\hat{Q} = PQ'$, $\{Q'_j = \prod_{i=j\alpha}^{(j+1)\alpha-1} q_j\}_{j\in[0,\mathtt{dnum})}$, and $\hat{Q}_j = \prod_{i=0\wedge i\ne j}^{\mathtt{dnum}-1} Q'_j$. The ModUp, ModDown and the decomposition algorithm (Dcomp) are shown in Alg. 2, Alg. 3 and Alg. 4 respectively.

Algorithm 2 $\text{ModUp}_{\mathcal{C}'_i\to\mathcal{D}_\beta}([a]_{\mathcal{C}'_i})$

```
Require: polynomial $a$ under moduli basis $\mathcal{C}'_i$ in NTT domain.
Ensure: polynomial $\tilde{a}$ under moduli basis $\mathcal{D}_\beta$ in NTT domain.
1: $([a(X)]_{\mathcal{C}'_i}) \leftarrow \text{iNTT}([a]_{\mathcal{C}'_i})$
2: $(\tilde{a}^{(j)})_{j\in A_i} \leftarrow (a^{(j)})_{j\in A_i}$          $\triangleright A_i = [i\alpha, (i+1)\alpha)$
3: $[\tilde{a}(X)]_{\mathcal{D}_\beta-\mathcal{C}'_i} \leftarrow \text{Conv}_{\mathcal{C}'_i\to\mathcal{D}_\beta-\mathcal{C}'_i}([a(X)]_{\mathcal{C}'_i})$
4: $(\tilde{a}^{(j)})_{j\in([0,k+\alpha\beta-1]-A_i)} \leftarrow \text{NTT}([\tilde{a}(X)]_{\mathcal{D}_\beta-\mathcal{C}'_i})$
5: return $[\tilde{a}]_{\mathcal{D}_\beta} = (\tilde{a}^{(0)}, \cdots, \tilde{a}^{(k+\alpha\beta-1)})$
```

Algorithm 3 $\text{ModDown}_{\mathcal{D}_\beta\to\mathcal{C}_\ell}(\tilde{b}^{(0)}, \cdots, \tilde{b}^{(k+\alpha\beta-1)})$

```
1: $[\tilde{b}]_{\mathcal{D}_\beta-\mathcal{C}_\ell} \leftarrow (\tilde{b}^{(0)}, \cdots, \tilde{b}^{(k-1)}, \tilde{b}^{(k+\ell)}, \cdots, \tilde{b}^{(k+\alpha\beta-1)})$
2: $[\tilde{b}(X)]_{\mathcal{D}_\beta-\mathcal{C}_\ell} \leftarrow \text{iNTT}([\tilde{b}]_{\mathcal{D}_\beta-\mathcal{C}_\ell})$
3: $[\tilde{a}(X)]_{\mathcal{C}_\ell} \leftarrow \text{Conv}_{\mathcal{D}_\beta-\mathcal{C}_\ell\to\mathcal{C}_\ell}([\tilde{b}(X)]_{\mathcal{D}_\beta-\mathcal{C}_\ell})$
4: $[\tilde{a}]_{\mathcal{C}_\ell} \leftarrow \text{NTT}([\tilde{a}(X)]_{\mathcal{C}_\ell})$
5: for $0 \le j < \ell$ do
6:     $b^{(j)} \leftarrow [\hat{Q}^{-1}]_{q_j} \cdot (\tilde{b}^{(k+j)} - \tilde{a}^{(j)}) \pmod{q_j}$
7: end for
8: return $(b^{(0)}, \cdots, b^{(\ell)})$
```

4) Key-switching and HMULT: An FHE-mult between two ciphertexts (HMULT) followed by rescaling, a function adjusting the scaling factor Δ of the message, reduces one level of the ciphertext.

Given two ciphertexts $\mathbf{ct}_1$ and $\mathbf{ct}_2$, HMULT multiplies two ciphertexts. RESCALE performs the rescaling operation for $\mathbf{ct}$. KeySwitch decomposes the input polynomial $[d]_{\mathcal{C}_\ell}$ into $[d_j]_{\mathcal{C}'_j}$,

[1]For multiplication of two polynomials, the polynomials must be in NTT domain for element-wise multiplication

Algorithm 4 Dcomp$(d^{(0)}, \cdots, d^{\ell})$

```
1: $d^{(j)} \leftarrow 0 \quad \forall j \in [\ell+1, \alpha\beta - 1]$
2: for $0 \leq i < \alpha$ do
3:     $d_j^{(i)} \leftarrow d^{(j\alpha+1)} \cdot [Q']_{q_{j\alpha+i}} \cdot [\hat{Q}_j^{-1}]_{q_{j\alpha+i}} \quad \forall j \in [0, \beta)$
4: end for
5: $d_j \leftarrow (d_j^{(i)})_{i\in[0,\alpha)} \quad \forall j \in [0, \beta)$
6: return $\vec{d} = (d_j)_{j\in[0,\beta)}$
```

where $j \in [0, \beta)$, extends the moduli of the decomposed parts using ModUp, multiplies them by an evaluation key, and finally reduces the moduli to the original level using ModDown.

Algorithm 5 HMULT$(\mathbf{ct}_1, \mathbf{ct}_2, \mathbf{evk})$

```
1: $\mathbf{ct}_1 \rightarrow (a_1, b_1),\ \mathbf{ct}_2 \rightarrow (a_2, b_2)$
2: $d_1 \leftarrow (a_1 \odot a_2),\ d_3 \leftarrow (b_1 \odot b_2)$
3: $d_2 \leftarrow (a_1 \odot b_2 + a_2 \odot b_1)$
4: $(c_1, c_2) \leftarrow \text{KeySwitch}(d_3, \mathbf{evk})$
5: return $\mathbf{ct}_{\text{mult}} \leftarrow (c_1 + d_1, c_2 + d_2)$
```

Algorithm 6 KeySwitch$([d]_{\mathcal{C}_\ell}, \mathbf{evk})$

```
1: $\vec{d} \leftarrow \text{Dcomp}(d),\ (d_j)_{j\in[0,\beta-1]\leftarrow\vec{d}}$
2: $[\tilde{d}_j]_{\mathcal{D}_\beta} = (\tilde{d}_j^{(0)}, \tilde{d}_j^{(1)}, \cdots, \tilde{d}_j^{(k+\alpha\beta-1)}) \leftarrow \text{ModUp}([d_j]_{\mathcal{C}'_j})$
   for $j \in [0, \beta - 1]$
3: for $i = 0$ to $k + \alpha\beta - 1$ do
4:     $(c_0^{(i)}, c_1^{(i)}) \leftarrow \sum_{j=0}^{\beta-1} \tilde{d}_j^{(i)} \odot \mathbf{evk}_j^{(i)}$
5: end for
6: $([c_0]_{\mathcal{C}_\ell}, [c_1]_{\mathcal{C}_\ell}) \leftarrow (\text{ModDown}([c_0]_{\mathcal{D}_\beta}), \text{ModDown}([c_1]_{\mathcal{D}_\beta}))$
7: return $([c_0]_{\mathcal{C}_\ell}, [c_1]_{\mathcal{C}_\ell})$
```

C. CKKS Encryption

We consider the following distributions. For a real $\sigma > 0$, $\mathcal{DG}(\sigma^2)$ samples a vector in $\mathbb{Z}^N$ by drawing its coefficient independently from the discrete Gaussian distribution of variance σ^2. For a real $0 \leq \rho \leq 1$, the distribution $\mathcal{ZO}(\rho)$ draws each entry in the vector from $\{0, \pm1\}^N$, with probability $\rho/2$ for each of -1 and $+1$, and probability being zero $1 - \rho$. The encryption of CKKS is the RLWE encryption. Given a RLWE public key (a, b), the CKKS encryption takes a message m, samples $v \leftarrow \mathcal{ZO}(0.5)$, $e_0, e_1 \leftarrow \mathcal{DG}(\sigma^2)$ and returns the encryption as given in Alg. 7. Note that in lines 4 – 6, there are four NTT operations to convert the polynomials into frequency domain so that the multiplications in lines 7 – 8 are performed as point-wise multiplication.

This work mainly focused on improving the HMULT, KeySwitch and HENC operations in Algorithm 5, Algorithm 6 and Algorithm 7 respectively.

D. Related Works

1) Implementation of FHE on GPUs and Embedded Systems: Polynomial multiplication is one of the most time-consuming operations in FHE, which is usually implementation through NTT. Referring to Algorithm 5, HMULT in

Algorithm 7 HENC(pk, m)

```
Require: a public key $pk$ of the form $(a, b)$, a message $m$.
Ensure: Encryption of $m$ as $c = (c_0, c_1)$ in NTT domain.
1: $(a, b) \leftarrow pk$
2: Sample $v' \leftarrow \mathcal{ZO}(0.5)$
3: Sample $e'_0, e'_1 \leftarrow \mathcal{DG}(\sigma^2)$
4: $v \leftarrow \text{NTT}(v')$
5: $m' \leftarrow \text{NTT}(\Delta \cdot m)$        ▷ $\Delta$ is a scaling factor
6: $e_0 \leftarrow \text{NTT}(e'_0),\ e_1 \leftarrow \text{NTT}(e'_1)$
7: $c_0 \leftarrow v \odot a + m' + e_0$
8: $c_1 \leftarrow v \odot b + e_1$
9: return $(c_0, c_1)$
```

CKKS requires three NTT, four point-wise multiplication and three INTT to complete. The number of NTT and INTT operations in Keyswitch (Algorithm 6 and line 4 in Algorithm 5) varies according to the parameter `dnum`, but the amount of computations is generally intensive. Parallelizing polynomial multiplication on GPUs can provide a huge improvement in the computational performance of FHE. The first attempt to achieve this is demonstrated by Wang et al. [12], in which Strassen's FFT-based multiplication was implemented on a GPU following Bailey's four-step FFT [13]. A subsequent work, cuHE [14], improved the implementation of polynomial multiplication on GPU and applied it to support the DHS [15] somewhat homomorphic encryption (SHE). Following these development, Al Badawi et al. presented parallel implementation of FV [5] and RNS-BFV [6] SHE on GPU platform. The key idea behind these two works is the use of discrete Galois transform (DGT) that exploits Gaussian primes so that the NTT computation can be performed on *N/2* size. This saves *N/2* twiddle factors, which can be helpful in reducing the GPU memory bandwidth. A similar approach was adopted by but [16]. However, DGT requires an extra twisting step, which can offset the gain in reduced memory bandwidth.

Recently, RNS-CKKS [17] scheme is gaining popularity due to its ability to process real numbers, which is widely used in computing artificial intelligence (AI) algorithms. Kim et al. [18] showcased various techniques to optimize the NTT implementation on GPU, including different NTT radices and the choice of memory used (registers vs shared memory). Building on top of this work, Jung et al. [7] presented a highly optimized implementation of RNS-CKKS on V100 GPU, including the bootstrapping step, which was not found in previous works. They fused multiple kernels to avoid many read/write operations onto the global memory. However, they did not fuse the NTT kernels due to the data accessing pattern that requires global synchronization; NTT is still implemented in two separate kernels. Subsequently, Shen et al. [8] proposed some tweaks to the implementation in [7] and reported a slightly improved NTT performance. However, they did not report the performance of the entire homomorphic multiplication. Recently, Fan et al. [19] show that the tensor cores on a GPU can be utilized to compute NTT without using the asymptotically faster algorithm like Cooley-Tukey [20]. Although the authors claimed that this approach can achieve a faster performance than Jung et al. [7], they did not compare the results using the same parameter set. Hence, the practicality of such an approach is still not fully verified. Due to this reason, we believe that the work from Jung et al. [7] and [8] are still having the best performance to date. In a separate work, Özgün et al. [21] presented techniques to optimize the radix-2 NTT implementation on a GPU to handle different polynomial sizes in FHE. The latest implementation of FHE on GPUs come from Yang et al. [9], wherein they proposed a software framework to process BGV, BFV and CKKS schemes. Moreover, they also managed to eliminate some operations in key-switching to achieve a slight performance improvement compared to [7]. However, similar to [7], their NTT implementation also requires at least two kernels to complete due to the unavoidable global synchronization.

There is also an attempt to implement FHE on embedded systems [10], wherein the encryption is performed on embedded systems. The main target of this work [10] is memory optimization, which is critical due to the limited RAM available on embedded systems. Such techniques may not be useful for massively parallel architectures like GPU that aim at high performance rather than low memory consumption.

2) High-Performance Lattice-based Cryptography with GPU Acceleration: Due to the NIST standardization activity [22], many parallel implementations of lattice-based cryptographic schemes are proposed and evaluated on GPU platforms. Similar to FHE, NTT is one of the main bottlenecks in most of the lattice-based cryptographic schemes. Lee et al. [23] presented the first parallel implementation of the Kyber key encapsulation mechanism (KEM). They combine the first two levels in NTT in order to compute more butterfly operations using the registers instead of the slower shared memory. However, this method does not work well for the remaining NTT levels due to the warp divergence issue. Han et al. [24] proposed a new level-combination technique to combine all NTT levels, which results in improved performance compared to the technique in [23]. They have also evaluated this technique on an inner-product functional encryption (IPFE) scheme. Note that these two works [23], [24] focus on improving the implementation of NTT with small and medium length ($256 \leq N < 8192$), which is smaller than the NTT used in FHE. Due to this reason, these implementations do not need any global synchronization because all computations can be efficiently computed within individual blocks. In contrast, NTT used in FHE is computed across different blocks, thus requiring a global synchronization to avoid data race issues.

3) Incomplete NTT: Incomplete NTT refers to a technique to skip some levels of NTT computation in exchange for more work to be performed on point-wise multiplication. For instance, if we skip the last level of radix-2 NTT, we must perform 2×2 polynomial multiplication instead of the common 1×1 point-wise multiplication. Chung et al. [25] utilized incomplete NTT to speed up the computation of polynomial multiplication on some NIST candidates (e.g., Saber) that do not have an NTT-friendly ring. Becker et al. [26] illustrated that by combining incomplete NTT with some efficient modular reduction techniques, one can achieve a good speed-up on ARM Cortex-A processors with NEON Single Instruction Multiple Data (SIMD).

III. Proposed Implementation of NTT on GPUs

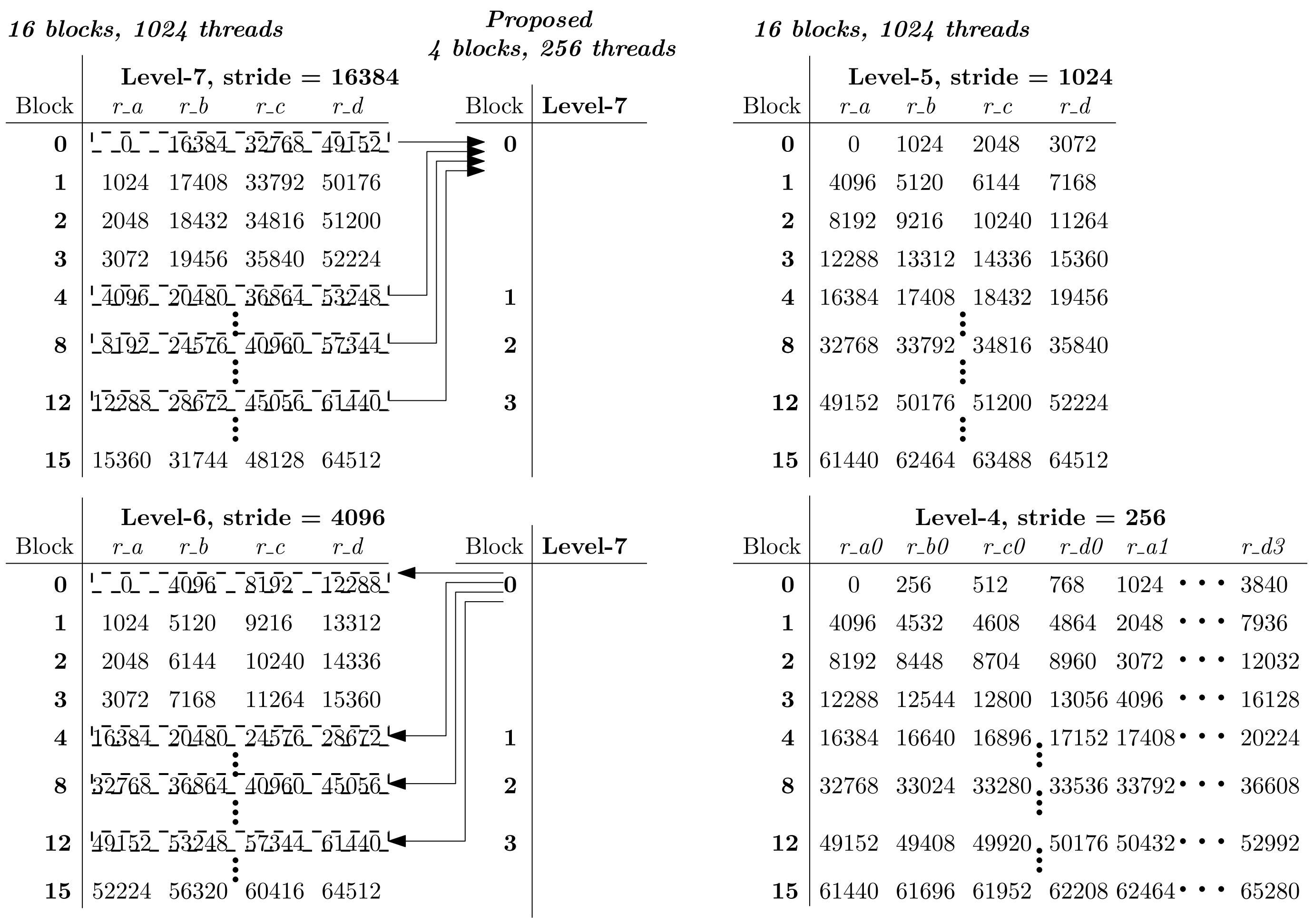
A. Reducing Synchronizations of radix-4 Implementation

Radix-2 Cooley-Tukey [20] is an algorithm commonly used in various NTT implementations on hardware and software. Based on the analysis by Özgün et al. [21], for a polynomial with length $N = 2^{16}$, five global synchronizations are required to complete the computation of radix-2 NTT. However, they did not provide any discussion for other radices (e.g., radix-4, -8 and -16). In this subsection, we provide a detailed analysis of the case of radix-4 and briefly discuss the case of radix-8 and -16.

Algorithm 8 Radix-4 NTT [27]

```
Require: A vector $a$ of length $N$, $\Phi_N$ be the $2N$-th primitive
    root of unity, $\omega_4$ be the 4-th primitive root of unity.
Ensure: $A \leftarrow NTT(a)$ in 2-bit reversed order.
1: Precompute the twiddle factors:
2: $r \leftarrow 0$
3: for $p = \log_4 N - 1$ to $0$ do
4:     $J \leftarrow 4^p$
5:     $\omega_m \leftarrow \Phi_N^J$
6:     for $k = 0$ to $N/(4J) - 1$ do
7:         $wa1[r] \leftarrow \omega_m^{\mathtt{brv}_2(k)+1}$
8:         $wa2[r] \leftarrow \omega_m^{2(\mathtt{brv}_2(k)+1)}$
9:         $wa3[r] \leftarrow \omega_m^{3(\mathtt{brv}_2(k)+1)}$
10:    end for
11: end for
12: Radix-4 NTT:
13: $r \leftarrow 0$
14: for $p = \log_4 N - 1$ to $0$ do
15:     $J \leftarrow 4^p$
16:     for $k = 0$ to $N/(4J) - 1$ do
17:         $w_1 \leftarrow wa1[r]$
18:         $w_2 \leftarrow wa2[r]$
19:         $w_3 \leftarrow wa3[r]$
20:         $r \leftarrow r + 1$
21:         for $j = 0$ to $J - 1$ do
22:             $t_0 \leftarrow a[4kJ + j] + a[(4k + 2)J + j] \cdot w_2$
23:             $t_1 \leftarrow a[4kJ + j] - a[(4k + 2)J + j] \cdot w_2$
24:             $t_2 \leftarrow a[(4k+1)J+j] \cdot w_1 + a[(4k+3)J+j] \cdot w_3$
25:             $t_3 \leftarrow a[(4k+1)J+j] \cdot w_1 - a[(4k+3)J+j] \cdot w_3$
26:             $A[4kJ + j] \leftarrow t_0 + t_2$
27:             $A[(4k + 1)J + j] \leftarrow (t_1 + t_3) \cdot \omega_4$
28:             $A[(4k + 2)J + j] \leftarrow t_0 - t_2$
29:             $A[(4k + 3)J + j] \leftarrow (t_1 - t_3) \cdot \omega_4$
30:         end for
31:     end for
32: end for
33: return $A$
```

Referring to Algorithm 8, the twiddle factors can be pre-computed before the NTT computation starts (lines 3 – 11). There are total $log_4 N - 1$ levels (line 14), in which N refers to the polynomial length. For each NTT level, the indexing



Block	Level-7, stride = 16384			
	r_a	*r_b*	*r_c*	*r_d*
0	0	16384	32768	49152
1	1024	17408	33792	50176
2	2048	18432	34816	51200
3	3072	19456	35840	52224
4	4096	20480	36864	53248
		⋮		
8	8192	24576	40960	57344
		⋮		
12	12288	28672	45056	61440
		⋮		
15	15360	31744	48128	64512

Block	Level-7
0	
1	
2	
3	

Block	Level-5, stride = 1024			
	r_a	*r_b*	*r_c*	*r_d*
0	0	1024	2048	3072
1	4096	5120	6144	7168
2	8192	9216	10240	11264
3	12288	13312	14336	15360
4	16384	17408	18432	19456
		⋮		
8	32768	33792	34816	35840
		⋮		
12	49152	50176	51200	52224
		⋮		
15	61440	62464	63488	64512

Block	Level-6, stride = 4096			
	r_a	*r_b*	*r_c*	*r_d*
0	0	4096	8192	12288
1	1024	5120	9216	13312
2	2048	6144	10240	14336
3	3072	7168	11264	15360
4	16384	20480	24576	28672
		⋮		
8	32768	36864	40960	45056
		⋮		
12	49152	53248	57344	61440
		⋮		
15	52224	56320	60416	64512

Block	Level-7
0	
1	
2	
3	

Block	Level-4, stride = 256						
	r_a0	*r_b0*	*r_c0*	*r_d0*	*r_a1*		*r_d3*
0	0	256	512	768	1024	• • •	3840
1	4096	4532	4608	4864	2048	• • •	7936
2	8192	8448	8704	8960	3072	• • •	12032
3	12288	12544	12800	13056	4096	• • •	16128
4	16384	16640	16896	17152	17408	• • •	20224
			⋮				
8	32768	33024	33280	33536	33792	• • •	36608
			⋮				
12	49152	49408	49920	50176	50432	• • •	52992
			⋮				
15	61440	61696	61952	62208	62464	• • •	65280

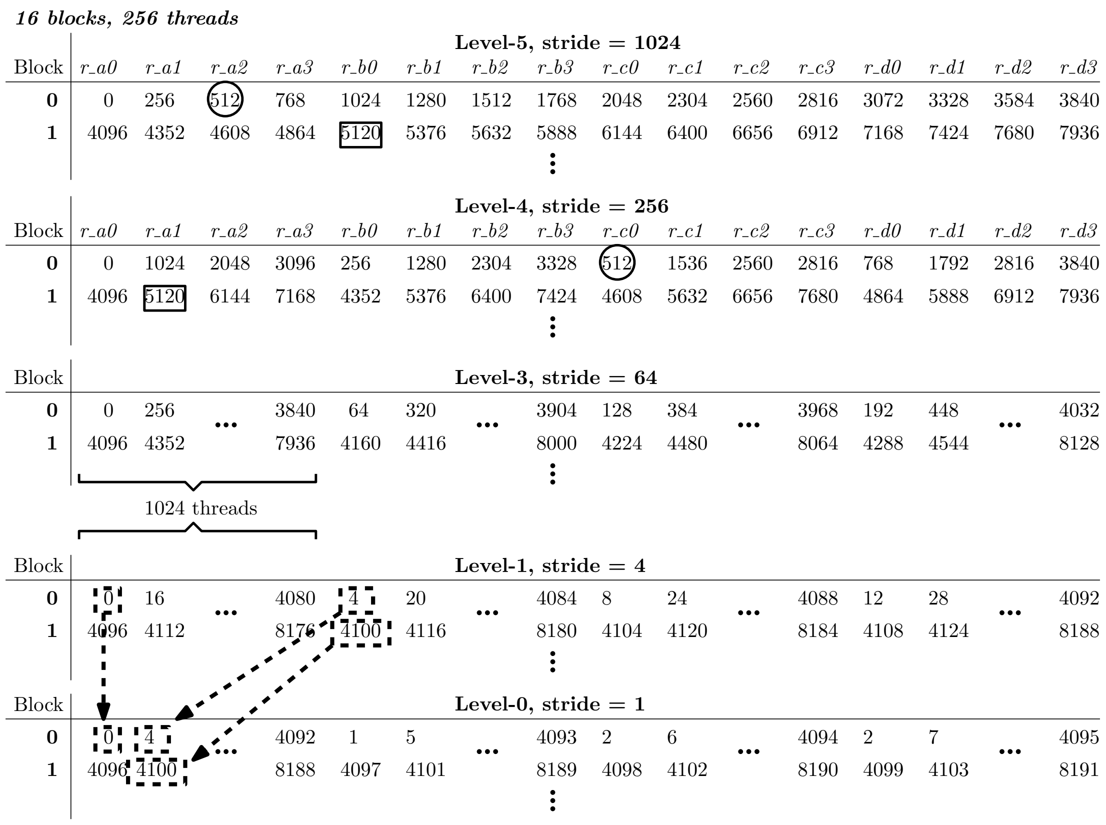
Fig. 2. Index patterns in Radix-4 NTT: 16 blocks and 1024 threads

patterns changes according to the values J (line 15), k (line 16) and j (line 21). This affects the way we implement the parallel NTT on a GPU across multiple blocks. Each step of radix-4 NTT computes four butterfly operations (lines 22 – 29).

Assuming that $N = 2^{16}$, there will be 65536 coefficients in a polynomial; we can distribute the computation of NTT to 16 different GPU blocks, wherein each block handles 4096 coefficients. Fig. 2 shows the patterns of indices in radix-4 NTT for the first four levels, wherein there are 16 GPU blocks and 1024 threads/block. Referring to level-7, block 0 processes coefficients 0–1023, 16384–17407, 32768–33791 and 49152–50175, which are consumed by block 0, 4, 8 and 12 in level-6. Due to this reason, we have to wait for all computations in level-7 to complete, before moving on to level-6. This implies that after computing level-7, synchronization across all blocks (block-wise synchronization) is required so that the race condition does not occur. In addition, since each block in level-6 is accessing data from other blocks, we need to store the intermediate results from level-7 in the global memory instead of the fast shared memory. In contrast, level-5 and level-4 do not show this restriction. Referring to level-5, block 0 processes coefficients 0–1023, 1024–2047, 2048–3071 and 3072–4095, which are consumed by block 0 in level-4. Hence, all the computations can be performed within the same block, and no block-wise synchronization is required. This pattern is also observed in the remaining levels of radix-4 NTT. Note that since we are using 1024 threads, the computations in level-4 are divided into four sections, and each section is processed by 256 threads (i.e., *r_a0*, *r_a1*, *r_a2*, ..., *r_d3*).

In summary, two synchronizations are required in the radix-4 implementation: one after level-7 and another one after level-6. This is lesser than radix-2 implementation demonstrated by Özgün et al. [21]. Following this method, we also found that one synchronization is required after the first level NTT in radix-8 and radix-16 implementation. Note that we can also allocate fewer threads per block (e.g., 512 or 256) and increase the total GPU blocks, but this does not affect the synchronization points required. Note that block-wise synchronization is a time-consuming process that should be avoided whenever possible.

A careful observation of Fig. 2 reveals that level-7 and level-6 can be combined into one kernel. Since we know that the coefficients processed by block 0 are consumed by blocks 0, 4, 8 and 12 in level-6, we can combine these four blocks into one. In other words, block 0 in level-7 processes indices that can be used by level-6 immediately; a similar observation applies to the other blocks. In this way, the proposed technique



Level-5, stride = 1024

Block	*r_a0*	*r_a1*	*r_a2*	*r_a3*	*r_b0*	*r_b1*	*r_b2*	*r_b3*	*r_c0*	*r_c1*	*r_c2*	*r_c3*	*r_d0*	*r_d1*	*r_d2*	*r_d3*
0	0	256	512	768	1024	1280	1512	1768	2048	2304	2560	2816	3072	3328	3584	3840
1	4096	4352	4608	4864	$\boxed{5120}$	5376	5632	5888	6144	6400	6656	6912	7168	7424	7680	7936
								⋮								

Level-4, stride = 256

Block	*r_a0*	*r_a1*	*r_a2*	*r_a3*	*r_b0*	*r_b1*	*r_b2*	*r_b3*	*r_c0*	*r_c1*	*r_c2*	*r_c3*	*r_d0*	*r_d1*	*r_d2*	*r_d3*
0	0	1024	2048	3096	256	1280	2304	3328	512	1536	2560	2816	768	1792	2816	3840
1	4096	$\boxed{5120}$	6144	7168	4352	5376	6400	7424	4608	5632	6656	7680	4864	5888	6912	7936
								⋮								

Level-3, stride = 64

Block																
0	0	256	...	3840	64	320	...	3904	128	384	...	3968	192	448	...	4032
1	4096	4352		7936	4160	4416		8000	4224	4480		8064	4288	4544		8128
								⋮								

Level-1, stride = 4

Block																
0	0	16	...	4080	4	20	...	4084	8	24	...	4088	12	28	...	4092
1	4096	4112		8176	4100	4116		8180	4104	4120		8184	4108	4124		8188
								⋮								

Level-0, stride = 1

Block																
0	0	4	...	4092	1	5	...	4093	2	6	...	4094	2	7	...	4095
1	4096	4100		8188	4097	4101		8189	4098	4102		8190	4099	4103		8191
								⋮								

Fig. 3. Top: Combine level-5 and -6 in radix-4 NTT with $N = 2^{16}$. Bottom: Optimize the read/write pattern in level-1 and -0.

managed to remove the block-wise synchronization between level-7 and -6. Moreover, the intermediate results from level-7 can be stored directly in registers and consumed by level-6, which reduces the expensive global memory accesses.

B. Combining Multiple Levels in Radix-4 NTT Implementation

Level-5 to level-0 radix-4 NTT can be computed within each block; this allows us to store the intermediate results for each level on the shared memory. Inspired by the technique proposed by Lee et al. [23], we found that level-5 and level-4 can also be combined. Referring to Fig. 3, we proposed to compute level-5 and level-4 with 256 threads/block. In this way, we can store the intermediate results for level-5 onto 16 different registers, which are consumed by level-4. For instance, level-5 writes the results of index 512 onto register *r_a2*, which is consumed by level-4 *r_c0*. Similarly, the result of block 1 level-5 *r_b0* is consumed by level-4 *r_a1*. This effectively reduces the accesses to shared memory between level-5 and -4. However, this technique also reduces the parallelism within a block by $4\times$ (from 1024 to 256). To combine with level-3, we need to reduce the parallel threads to 64 and use 64 registers to store the intermediate results. This can eventually harm the performance of NTT because the excessive use of registers and reduced parallel threads affect the computational occupancy on a GPU. Hence, we only apply this technique to level-5 and level-4.

C. Optimizing the Remaining levels in Radix-4 NTT

The remaining levels in radix-4 NTT are computed on the shared memory, wherein avoiding bank conflicts is important to achieve good performance. We propose to optimize the write patterns in these levels to reduce the number of bank conflicts. Referring to Fig. 3 level-1, after reading the data from shared memory (stride of 16) and performing the butterfly operations, the intermediate results are stored onto the same indices, causing bank conflicts. To avoid this issue, we proposed to store these intermediate results in the indices of the next level. For instance, level-1 can store the NTT results on indices of level-0, which effectively removes the write bank conflicts entirely. However, we can still not remove the read bank conflicts due to the specific memory patterns in NTT. Note that this technique can also be applied to level-2 to remove the write bank conflicts, but the performance gain is not significant for level-3 since it does not have any bank conflicts.

Besides that, level 3 – level 0 needs to read the twiddle factors in a stride of 64, 16, 4 and 1, respectively. These twiddle factors are stored on the global memory; they cannot be cached on the constant memory due to their huge size. Strided read on global memory reduces the performance seriously, which should be avoided whenever possible. To resolve this issue, we propose to compute the twiddle factors on the fly for these four levels.

IV. Proposed Transposed HENC and HMULT

For a prime q, NTT maps a polynomial $f(X) \in \mathbb{R}_q$ to $(f(\Phi_N), f(\Phi_N^3), \cdots, f(\Phi_N^{2N-1}))$, where Φ_N is the $2N$-

th primitive root of unity. The NTT over a polynomial ring $\mathbb{Z}_q[X]/\langle\phi(X)\rangle$ uses the fact that q is chosen such that $\phi(X) = \prod_{i=0}^{k-1} \phi_i(X)$ and we can define NTT of a polynomial $f(X) \in \mathbb{Z}_q[X]/\langle\phi(X)\rangle$ as $(f(X) \pmod{\phi_0(X)}, \cdots, f(X) \pmod{\phi_{k-1}(X)})$.

Now consider $N = n_1 \times n_2$, then $\Phi_{n_1} = \Phi_N^{n_2}$ is the $2n_1$-th primitive root of unity. Also, observe that

$$X^N + 1 = \prod_{i=0}^{n_1-1} (X^{n_2} - \Phi_{n_1}^{2i+1}) \tag{1}$$

Therefore, we define $\mathrm{NTT}_{n_2}(f(X)) = (F_0, F_1, \cdots, F_{n_1-1})$ where $F_i = f(X) \pmod{X^{n_2} - \Phi_{n_1}^{2i+1}}$ for $0 \leq i \leq n_1 - 1$. Consider the polynomial $f(X)$ as

$$f(X) = \sum_{i=0}^{N-1} f_i X^i \tag{2}$$

For any $0 \leq i < N$, we can write $i = pn_2 + q$ where $0 \leq p < n_1$ and $0 \leq q < n_2$. Therefore,

$$\begin{aligned} f(X) &= \sum_{p=0}^{n_1} \sum_{q=0}^{n_2} f_{pn_2+q} X^{pn_2+q} \\ &= \sum_{q=0}^{n_2} \left(\sum_{p=0}^{n_1} f_{pn_2+q} \left(X^{n_2}\right)^p \right) X^q \end{aligned} \tag{3}$$

This implies for $0 \leq i < n_1$

$$\begin{aligned} F_i &= f(X) \pmod{X^{n_2} - \Phi_{n_1}^{2i+1}} \\ &= \sum_{q=0}^{n_2} \left(\sum_{p=0}^{n_1} f_{pn_2+q} \left(\Phi_{n_1}^{2i+1}\right)^p \right) X^q \end{aligned} \tag{4}$$

Consider the polynomial $f_{n_2,q}(Y) = \sum_{p=0}^{n_1} f_{pn_2+q} Y^p$. This is a polynomial in the ring $\mathbb{Z}_q[X]/\langle X^{n_1} + 1\rangle$ and the NTT transformation of this polynomial is $(f_{n_2,q}(\Phi_{n_1}^1), f_{n_2,q}(\Phi_{n_1}^3), \cdots, f_{n_2,q}(\Phi_{n_1}^{2n_1-1}))$. Now, for any $0 \leq i < n_1$

$$F_i = \sum_{q=0}^{n_2} f_{n_2,q}(\Phi_{n_1}^{2i+1}) X^q \tag{5}$$

Therefore to compute $\mathrm{NTT}_{n_2}(f(X))$ we first have to compute NTT transformation of the polynomials $f_{n_2,q}(Y)$ for each $0 \leq q < n_2$. Observe that if we can consider all the coefficients of $f(X)$ as an $n_1 \times n_2$ matrix as shown in Fig. 4, then the i-th column of the matrix corresponds to the polynomial $f_{n_2,i}(Y)$. To compute multiplication of two polynomials $f(X), g(X) \in \mathcal{R}_q$, first we compute $\mathrm{NTT}_{n_2}(f(X)) = (F_1, \cdots, F_{n_1-1})$ and $\mathrm{NTT}_{n_2}(g(X)) = (G_1, \cdots, G_{n_1-1})$ using the above-mentioned method. Let $h(X)$ be the multiplication of $f(X)$ and $g(X)$ in $\mathcal{R}_q$ then, we compute $H_i := F_i \cdot G_i \mod (X^{n_2} - \Phi_{n_1}^{2i+1})$ and compute $h(X)$ using the inverse of NTT_{n_2}.

A. cuTraNTT: Computing NTT with Only One GPU Kernel

The optimized radix-4 NTT presented in Algorithm 8 requires one block-wise synchronization (see Section III-A). Careful observation shows that we can remove this synchronization by representing a polynomial in a transposed form and skipping the remaining two NTT levels. Supposed that we want to perform NTT transformation on a polynomial $\mathbf{A} = <a_0, a_1, a_2, \ldots a_N>$ and we have taken B blocks and each block holds 4096 elements from the polynomial $\mathbf{A}$. Consider a polynomial with $N = 2^{16}$; we rearrange the polynomial coefficients following Fig. 4, in which $n_1 = 4096$ and $n_2 = 16$, which is also illustrated in Fig. 5. Each column is of the form $<a_{i+k\times\frac{4096}{N}} : k = 0, 1, 2, \ldots, 4095>$, where $N = 65536$ and i runs from 0 to 15. With 16 GPU blocks, each block taking one column, we can easily transform the full polynomial up to 6 levels without any synchronization. The remaining two levels (level 1 and 0) are skipped in cuTraNTT, at expense of a more costly Hadamard product, which will be explained in the next sub-section.

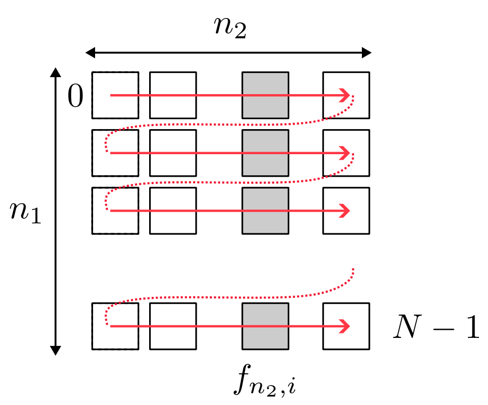


Fig. 4. Transposed form of a polynomial with length $N = 2^{16}$

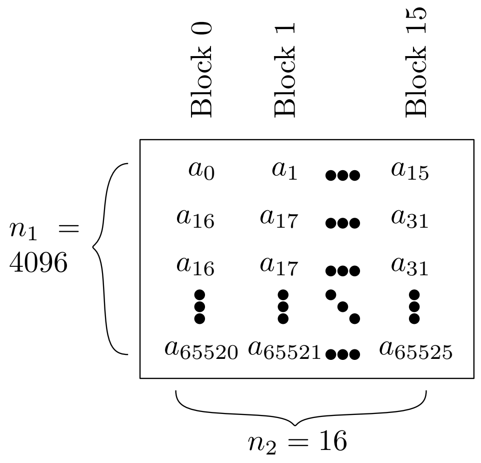

Block 0
Block 1
Block 15
$n_1 = 4096$
a_0 a_1 a_{15}
a_{16} a_{17} a_{31}
a_{16} a_{17} a_{31}
a_{65520} a_{65521} a_{65525}
$n_2 = 16$

Fig. 5. Transposed form of a polynomial with length N

B. Optimizing the Hadamard Product

The conventional way to apply NTT for accelerating polynomial multiplication requires us to first transform the inputs into NTT domain, then perform a Hadamard product, which is frequently referred to as point-wise multiplication because it only performs 1×1 Hadamard product. In this way, the computation time of polynomial multiplication can be greatly reduced. The proposed cuTraNTT skipped the last two levels (L1 and L0) of the radix-4 NTT and speed up the NTT computation, which is usually the bottleneck in FHE. However, we need to perform additional work on the Hadamard product so that the computation of polynomial multiplication can be correct. In particular, instead of performing a 1×1 Hadamard product, we need to perform 16×16 Hadamard product. In other words, we have shifted the bulk computation from NTT to the Hadamard product.

Algorithm 9 $k \times k$ Schoolbook-Multiplication(a, b, w)

```
1: for $i = 0 \to 2k-1$ do
2:     $c[i] \leftarrow 0$
3: end for
4: for $i = 0 \to k-1$ do
5:     for $j = 0 \to k-1$ do
6:         $c[i+j] \leftarrow c[i+j] + a[i] \cdot b[j]$
7:     end for
8: end for
9: for $i = 0 \to k-1$ do
10:    $c[i] \leftarrow c[i] + w \cdot c[k+i]$
11: end for
```

Referring to Algorithm 9, a simple way to perform $k \times k$ Hadamard product using schoolbook-multiplication can be easily parallelized. Since there are no dependency issues, we can launch many threads, each thread computing one $k \times k$ coefficient. For instance, if $N = 2^{16}$ and we skip 2 levels of radix-4 NTT, we need to perform 16×16 Hadamard product. We can implement this by using 16 blocks, each block having 256 threads, each thread computing 16 polynomial multiplications, each multiplication is 16×16 (lines 4 – 6). Although this method is easily parallelizable, the performance is very slow compared to the 1×1 Hadamard product.

To reduce the latency of Hadamard Product in our implementation, we proposed a parallel version of Toom-Cook-4 [28] 16×16 Hadamard Product. Referring to Algorithm 10, one 16×16 polynomial is broken into four 4×4 multiplications, followed by the evaluation step in the Took-Cook algorithm (lines 3 – 26). Line 27 – 29 perform the point-wise multiplication in the Took-Cook algorithm, which is essentially a 4×4 schoolbook multiplication (see Algorithm 9). Followed by this is the interpolation step in the Took-Cook algorithm (lines 30 – 53). Finally, lines 54 – 56 show the final reduction step, which returns the final results of 16×16 Hadamard Product.

C. Applying cuTraNTT and Toom-Cook-4 to HMULT and HENC

Referring to Algorithm 5, there are three NTT and three iNTT used in HMULT, which can benefit from our heavily optimized cuTranTT implementation. However, there are also four Hadamard products (lines 3 and 4) and a few smaller Hadamard products (Algorithm 6, line 4). This shows that HMULT performance is heavily affected by the Hadamard products, and the potential speed-up from using cuTraNTT is offset by the slow Hadamard products due to skipping two NTT levels. On the other hand, HENC (Algorithm 7) requires four NTT operations (lines 4 – 6), but only two Hadamard products (lines 7 and 8). In this case, we can leverage the proposed cuTranTT implementation to speed up

Algorithm 10 16×16 Toom-Cook-4(a, b, w) [28]

```
1: for $i = 0 \to 31$ do $res[i] \leftarrow 0$
2: end for
3: for $i = 0 \to 3$ do
4:     $(r_0, r_1, r_2, r_3) \leftarrow (a[4i], a[4i+1], a[4i+2], a[4i+3])$
5:     $r_4 \leftarrow r_0 + r_2$; $r_5 \leftarrow r_1 + r_3$
6:     $r_6 \leftarrow r_4 + r_5$; $r_7 \leftarrow r_4 - r_5$
7:     $aw_3[i] \leftarrow r_6$; $aw_4[i] \leftarrow r_7$
8:     $r_4 \leftarrow 8r_0 + r_2$
9:     $r_5 \leftarrow 4r_1 + r_3$
10:    $r_6 \leftarrow r_4 + r_5$
11:    $r_7 \leftarrow r_4 - r_5$
12:    $aw_5[i] \leftarrow r_6$; $aw_6[i] \leftarrow r_7$
13:    $r_4 \leftarrow 8r_3 + 4r_2 + 2r_1 + r_0$
14:    $aw_2[i] \leftarrow r_4$; $aw_7[i] \leftarrow r_0$; $aw_1[i] \leftarrow r_3$
15:    $(r_0, r_1, r_2, r_3) \leftarrow (b[4i], b[4i+1], b[4i+2], b[4i+3])$
16:    $r_4 \leftarrow r_0 + r_2$; $r_5 \leftarrow r_1 + r_3$
17:    $r_6 \leftarrow r_4 + r_5$; $r_7 \leftarrow r_4 - r_5$
18:    $bw_3[i] \leftarrow r_6$; $bw_4[i] \leftarrow r_7$
19:    $r_4 \leftarrow 8r_0 + r_2$
20:    $r_5 \leftarrow 4r_1 + r_3$
21:    $r_6 \leftarrow r_4 + r_5$
22:    $r_7 \leftarrow r_4 - r_5$
23:    $bw_5[i] \leftarrow r_6$; $bw_6[i] \leftarrow r_7$
24:    $r_4 \leftarrow 8r_3 + 4r_2 + 2r_1 + r_0$
25:    $bw_2[i] \leftarrow r_4$; $bw_7[i] \leftarrow r_0$; $bw_1[i] \leftarrow r_3$
26: end for
27: for $i = 1 \to 7$ do
28:    $w_i \leftarrow 4 \times 4$ Schoolbook-Multiplication$(aw_i, bw_i, w)$
29: end for
30: for $i = 0 \to 3$ do
31:    for $j = 0 \to 7$ do
32:        $r_j \leftarrow w_{j+1}[i]$
33:    end for
34:    $r_1 \leftarrow r_1 + r_4$; $r_5 \leftarrow r_5 - r_4$
35:    $r_3 \leftarrow 2^{-1}(r_3 - r_2)$
36:    $r_4 \leftarrow r_4 - r_0 - 64r_6$; $r_4 \leftarrow 2r_4 + r_5$
37:    $r_2 \leftarrow r_2 + r_3$; $r_1 \leftarrow r_1 - r_2 - 64r_2$
38:    $r_2 \leftarrow r_2 - r_0 - r_6$
39:    $r_1 \leftarrow r_1 + 45r_2$
40:    $r_4 \leftarrow 24^{-1}(r_4 - 8r_2)$
41:    $r_5 \leftarrow r_5 + r_1$
42:    $r_1 \leftarrow 18^{-1}(16r_3 + r_1)$
43:    $r_3 \leftarrow -r_3 - r_1$
44:    $r_5 \leftarrow 2^{-1}r_1 - 60^{-1}r_5$
45:    $r_2 \leftarrow r_2 - r_4$; $r_1 \leftarrow r_1 - r_5$
46:    $res[4i] \leftarrow res[4i] + r_6$
47:    $res[4i+1] \leftarrow res[4i+1] + r_5$
48:    $res[4i+2] \leftarrow res[4i+2] + r_4$
49:    $res[4i+3] \leftarrow res[4i+3] + r_3$
50:    $res[4i+4] \leftarrow res[4i+4] + r_2$
51:    $res[4i+5] \leftarrow res[4i+5] + r_1$
52:    $res[4i+6] \leftarrow res[4i+6] + r_0$
53: end for
54: for $i = 0 \to 15$ do
55:    $res[i] \leftarrow res[i] + w \cdot res[i+16]$
56: end for
57: return $(res[0], \cdots, res[15])$
```

HENC, wherein the effect of slow Hadamard products is less significant compared to HMULT.

V. Experimental Performance and Discussions

This section presents the experimental results of cuTraNTT and its application to the CKKS FHE scheme. The experiment was carried out on three different platforms, which is detailed in Table I.

TABLE I
EXPERIMENTAL PLATFORMS

	Platform-1	Platform-2	Platform-3	
	Desktop Workstation	Cloud System		Embedded System
GPU	RTX 4060	A100	V100	Jetson Orin Nano
CUDA Cores	3072	8192	5120	1024
Architecture	Lovelace	Ampere	Volta	Ampere
Comp. Cap.	8.9	8.0	7.0	8.7
Clock (GHz)	1.830	1.410	1.246	625
Memory BW (GB/s)	272	1935	900	69
No. of SM	24	80	64	8
CUDA SDK	CUDA 12.2			
CPU	Intel i7-14700K	Intel Xeon Gold 6150		ARM Cortex® -A78AE
Clock (GHz)	3.70	2.2		1.5

Platform-1 is a desktop workstation equipped with Intel Core i7-14700K CPU and an RTX 4060 GPU. Platform-2 is the Digital Research Alliance of Canada Cloud system [29] that allows flexible configuration to use a V100 or an A100 GPU. Note that RTX 4060 is a popular consumer-grade GPU, while V100 and A100 are server-grade GPU with much higher performance in terms of computation and memory bandwidth. On the other hand, Platform-3 is an embedded system consisting of an ARM Cortex-A78 processor and a small embedded GPU with 1024 CUDA cores. Platform-1 and -3 are used to simulate the client-side computation, wherein many HENC operations are performed. Platform-2 acts as a server-side unit that mainly computes HMULT operations.

A. Micro-benchmarking Radix-4 NTT Implementation on RTX 4060

Table II shows the results of the techniques proposed in this section. The baseline implementation, without any optimization techniques applied, can complete one NTT with length 2^{16} in $30.53\mu s$; it is implemented by using 16 blocks, and each block contains 256 threads. After combining the first four levels (L7 – L4), the performance is improved by $1.25\times$. By applying the optimized write patterns on shared memory, together with the computations of twiddle factors on the fly, we can complete the NTT computation in $22.41\mu s$, which speeds up the computation by $1.36\times$ compared to the baseline version. It is also $1.06\times$ and $1.02\times$ faster than the previous work [7] and Phantom [9], respectively. This shows that our radix-4 NTT implementation using 32-bit arithmetic can achieve a good performance compared to 64-bit arithmetic used by [7], [9]. By skipping the last 2 levels (L0 and L1), we achieve a $1.82\times$ speed-up compared to the baseline implementation.

TABLE II
BENCHMARKING THE PROPOSED IMPROVEMENTS ON RADIX-4 NTT IMPLEMENTATION.

Techniques[1]	N	Word Size	Time (μs)	Speed-up
Baseline	2^{16}	32-bit	30.53	1.00
Comb. L7 and L6			27.47	1.11
Comb. L5 and L4			24.39	1.25
Opt. L3 to L0			22.41	1.36
NTT-Skip-2 [3]			12.34	1.82
[9][2]	2^{16}	64-bit	22.95	1.06
[7][2]			23.85	1.02

[1] The experiments were carried out on an RTX 4060 GPU.
[2] The open-sourced software from the authors was executed on the same RTX 4060 GPU.
[3] Skipping L0 and L1, see Section IV-A.

TABLE III
PERFORMANCE OF DIFFERENT IMPLEMENTATIONS OF HADAMARD PRODUCT ON RTX 4060.

Hadamard Product [1]	Execution time (µs)	
	RTX 4060	Speed-up
1×1 point-wise	49.88	1
16×16 point-wise	99.46	0.50
Proposed Toom-Cook-4 16×16 point-wise	76.95	0.65

[1] The experiments were carried out with 66 polynomials, each polynomial having the size of $N = 2^{16}$.

B. Performance of the proposed Toom-Cook 4-way Hadamard Product

Table III shows the performance of different implementations of Hadamard products on an RTX 4060 GPU with 66 polynomials, each with 2^{16} coefficients. If we do not skip any NTT levels, the Hadamard product only performs 1×1 point-wise multiplication, which takes only $49.88\mu s$ to complete. Skipping two levels of NTT can achieve very fast timing performance, but the Hadamard product becomes $2\times$ more expensive (99.46 μs). To reduce this detrimental effect, we proposed the optimized Toom-Cook-4 Hadamard product (see Algorithm 10, effectively reducing the time to $76.95\mu s$, which is 29.25% faster. However, it is still slower than the 1×1 point-wise multiplication, and this is the trade-off that cannot be avoided if we skip the last two levels in NTT computation.

C. Comparison with state-of-the-art implementations

Table IV shows the results of the proposed cuTraNTT applied to CKKS scheme on various GPU platforms. Our HMULT implementation achieved similar performance compared to the state-of-the-art [7], [9]. In particular, our HMULT is 2.0% slower than [7] on a V100 GPU and 1.2% faster than [9] on an A100 GPU. This shows that although the proposed cuTraNTT can effectively reduce the latency of NTT computation, it suffers from the slow Hadamard product, which offsets the performance gain from NTT. On a Jetson Orin Nano, the HENC algorithm using full NTT takes 15.18ms to complete, while the version using cuTraNTT only takes 11.32ms. The HENC is sped up by $1.37\times$, which can be significant in a practical use case. We observed a similar level of speed up in HENC when the experiments were carried out on a desktop GPU RTX 4060, where the performance is

TABLE IV
EXPERIMENTAL RESULTS OF THE PROPOSED CUTRANTT APPLIED TO CKKS FHE SCHEME

Execution time (ms)							
	Phantom [9]		Over100x [7]	Ours			
log N	16		16	16			
log Q	1420	1220	1693	1693			
log QP	1720	1580	2364	2364			
L	34	29	32	32			
dnum	7	5	3	3			
GPU			V100	V100	A100	RTX 4060	Jetson Orin Nano
HMULT (cuTraNTT)	2188	1609	2960	3021	2162	5048	-
HENC (Full NTT)	-	-	-		-	1532	15177
HENC (cuTraNTT)	-	-	-		-	1120	11312

improved by $1.34\times$. From these experimental results, we can summarize that the proposed cuTraNTT is useful for reducing the latency in HENC, which is typically executed on edge devices with constrained resources, and it only marginally affects the performance of HMULT.

VI. CONCLUSIONS

cuTraNTT, a transposed version of NTT implementation, was proposed in this paper. By skipping the last two levels and rearranging NTT in a transposed form, the NTT computation can be performed with only one kernel. This removed the need for block-wise synchronization, thus allowing a very efficient NTT implementation on GPU platforms, which can be very useful to HENC. The extra overhead introduced by cuTraNTT on the HMULT was minimized by utilizing the proposed parallel Toom-Cook-4 implementation. Experimental results show that cuTraNTT applied to CKKS FHE achieved similar latency performance compared with state-of-the-art [7], [9] on HMULT but achieved significant improvement on HENC. The findings from this paper can be used by the IoT sensor nodes to efficiently encrypt sensitive data so that it can be performed by the untrusted third party (i.e., cloud server). In the future, we plan to extend this work to support machine learning applications in the encrypted domain. Various packing methods will be explored to leverage the proposed cuTraNTT for accelerated performance.

ACKNOWLEDGMENT

Wai-Kong Lee and Seong Oun Hwang were supported by the National Research Foundation of Korea (NRF) grant funded by the Korean government (Ministry of Science and ICT) (No. RS-2024-00340882). Wai-Kong Lee was also partially supported by the Ministry of Higher Education (MoHE), Malaysia, through the Fundamental Research Grant Scheme (FRGS/1/2021/ICT07/UTAR/01/1).

REFERENCES

[1] A. Borgeaud, "Top cloud security concerns worldwide 2021," Jun 2023. [Online]. Available: https://www.statista.com/statistics/1172265/biggest-cloud-security-concerns-in-2020/

[2] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," in *Advances in Cryptology – EUROCRYPT 2010*, H. Gilbert, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 1–23.

[3] M. V. Beirendonck, J. D'Anvers, F. Turan, and I. Verbauwhede, "FPT: A fixed-point accelerator for torus fully homomorphic encryption," in *CCS*. ACM, 2023, pp. 741–755.

[4] J. Bertels, M. V. Beirendonck, F. Turan, and I. Verbauwhede, "Hardware acceleration of FHEW," in *DDECS*. IEEE, 2023, pp. 57–60.

[5] A. Al Badawi, B. Veeravalli, C. F. Mun, and K. M. M. Aung, "High-performance fv somewhat homomorphic encryption on gpus: An implementation using cuda," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 70–95, 2018.

[6] A. Al Badawi, Y. Polyakov, K. M. M. Aung, B. Veeravalli, and K. Rohloff, "Implementation and performance evaluation of rns variants of the bfv homomorphic encryption scheme," *IEEE Transactions on Emerging Topics in Computing*, vol. 9, no. 2, pp. 941–956, 2019.

[7] W. Jung, S. Kim, J. H. Ahn, J. H. Cheon, and Y. Lee, "Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with gpus," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 114–148, 2021.

[8] S. Shen, H. Yang, Y. Liu, Z. Liu, and Y. Zhao, "Carm: Cuda-accelerated rns multiplication in word-wise homomorphic encryption schemes for internet of things," *IEEE Transactions on Computers*, 2022.

[9] H. Yang, S. Shen, W. Dai, L. Zhou, Z. Liu, and Y. Zhao, "Phantom: a cuda-accelerated word-wise homomorphic encryption library," *IEEE Transactions on Dependable and Secure Computing*, 2024.

[10] D. Natarajan and W. Dai, "Seal-embedded: A homomorphic encryption library for the internet of things," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 756–779, 2021.

[11] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I 23*. Springer, 2017, pp. 409–437.

[12] W. Wang, Y. Hu, L. Chen, X. Huang, and B. Sunar, "Exploring the feasibility of fully homomorphic encryption," *IEEE Transactions on Computers*, vol. 64, no. 3, pp. 698–706, 2013.

[13] D. H. Bailey, "Ffts in external of hierarchical memory," in *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, 1989, pp. 234–242.

[14] W. Dai and B. Sunar, "cuhe: A homomorphic encryption accelerator library," in *Cryptography and Information Security in the Balkans: Second International Conference, BalkanCryptSec 2015, Koper, Slovenia, September 3-4, 2015, Revised Selected Papers 2*. Springer, 2016, pp. 169–186.

[15] Y. Doröz, A. Shahverdi, T. Eisenbarth, and B. Sunar, "Toward practical homomorphic evaluation of block ciphers using prince," in *Financial Cryptography and Data Security: FC 2014 Workshops, BITCOIN and WAHC 2014, Christ Church, Barbados, March 7, 2014, Revised Selected Papers 18*. Springer, 2014, pp. 208–220.

[16] P. G. M. Alves, J. N. Ortiz, and D. F. Aranha, "Faster homomorphic encryption over gpgpus via hierarchical dgt," in *Financial Cryptography and Data Security: 25th International Conference, FC 2021, Virtual Event, March 1–5, 2021, Revised Selected Papers, Part II 25*. Springer, 2021, pp. 520–540.

[17] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "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*. Springer, 2019, pp. 347–368.

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

[19] S. Fan, Z. Wang, W. Xu, R. Hou, D. Meng, and M. Zhang, "Tensorfhe: Achieving practical computation on encrypted data using gpgpu," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 922–934.

[20] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," *Mathematics of computation*, vol. 19, no. 90, pp. 297–301, 1965.

[21] Ö. Özerk, C. Elgezen, A. C. Mert, E. Öztürk, and E. Savaş, "Efficient number theoretic transform implementation on gpu for homomorphic encryption," *The Journal of Supercomputing*, vol. 78, no. 2, pp. 2840–2872, 2022.

[22] "Post-quantum cryptography: Round 1 submissions." [Online]. Available: https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-1-submissions

[23] W.-K. Lee and S. O. Hwang, "High throughput implementation of post-quantum key encapsulation and decapsulation on gpu for internet of things applications," *IEEE Transactions on Services Computing*, vol. 15, no. 6, pp. 3275–3288, 2021.

[24] K. H. Han, W.-K. Lee, A. Karmakar, J. M. B. Mera, and S. O. Hwang, "cufe: High performance privacy preserving support vector machine with inner-product functional encryption," *IEEE Transactions on Emerging Topics in Computing*, 2023.

[25] C.-M. M. Chung, V. Hwang, M. J. Kannwischer, G. Seiler, C.-J. Shih, and B.-Y. Yang, "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*, pp. 159–188, 2021.

[26] H. Becker, V. Hwang, M. J. Kannwischer, B.-Y. Yang, and S.-Y. Yang, "Neon ntt: Faster dilithium, kyber, and saber on cortex-a72 and apple m1," *Cryptology ePrint Archive*, 2021.

[27] X. Chen, B. Yang, S. Yin, S. Wei, and L. Liu, "Cfntt: Scalable radix-2/4 ntt multiplication architecture with an efficient conflict-free memory mapping scheme," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 94–126, 2022.

[28] J. M. Bermudo Mera, A. Karmakar, and I. Verbauwhede, "Time-memory trade-off in toom-cook multiplication: an application to module-lattice based cryptography," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2020, no. 2, pp. 222–244, 2020.

[29] "Digital research alliance of canada," https://alliancecan.ca/en/, 2021, accessed: 2024-08-19.

Supriya Adhikary (Student Member, IEEE) received his B.Sc. degree in Mathematics and M.Sc. degree in Pure Mathematics from University of Calcutta. He received his M.Tech degree in Cryptology and Security from Indian Statistical Institute, Kolkata, and currently, he is a Ph.D. student at the Indian Institute of Technology, Kanpur, India.

Wai-Kong Lee (Member, IEEE) received the B.Eng. in Electronics and M.Eng.Sc. degree from Multimedia University in 2006 and 2009 respectively. In between 2009 – 2012, he served as R&D engineer in several multinational companies including Agilent Technologies (now known as Keysight) in Malaysia. He obtained Ph.D. degree in Engineering from University Tunku Abdul Rahman, Malaysia (UTAR) in 2018, where he served as an assistant professor and deputy dean (R&D) for Faculty of Information and Communication Technology. From 2020 - 2023, he was a post-doctoral researcher in Gachon University, South Korea. Currently, he serves as an associate professor in UTAR. His research interests include cryptographic engineering, GPU computing, numerical algorithms, lightweight machine learning, Internet of Things (IoT) and energy harvesting.

Angshuman Karmakar (Member, IEEE) received the BE degree in computer science and engineering from Jadavpur University, Kolkata, and the MTech degree in computer science and engineering from the Indian Institute of Technology, Kharagpur. He received his doctorate from Katholieke Universiteit Leuven, Belgium for his dissertation titled "Design and implementation aspects of post-quantum cryptography". He is one of the primary designers of the post-quantum Saber key-encapsulation mechanism scheme which is one of the finalists in the National Institute of Standards and Technology's post-quantum standardization procedure. He is currently an FWO post-doctoral fellow in the COSIC research group of KU Leuven. Earlier, he worked as an engineer in Citrix R&D India Ltd, Bangalore, and as a research intern at Microsoft Research, Redmond, USA. His research interest spans different aspects of lattice-based post-quantum cryptography and computation on encrypted data.

Seong Oun Hwang (Senior Member, IEEE) received the B.S. degree in mathematics from Seoul National University, in 1993, the M.S. degree in information and communications engineering from the Pohang University of Science and Technology, in 1998, and the Ph.D. degree in computer science from the Korea Advanced Institute of Science and Technology, in 2004, South Korea. He worked as a Software Engineer with LG-CNS Systems, Inc., from 1994 to 1996. He worked as a Senior Researcher with the Electronics and Telecommunications Research Institute (ETRI), from 1998 to 2007. He worked as a Professor with the Department of Software and Communications Engineering, Hongik University, from 2008 to 2019. He is currently a Professor with the Department of Computer Engineering, Gachon University. His research interests include cryptography, cybersecurity, and artificial intelligence. He is an Editor of *ETRI Journal*.

Ramachandra Achar (Fellow, IEEE) received the B. Eng. degree in electronics engineering from Bangalore University, India in 1990, M. Eng. degree in micro-electronics from Birla Institute of Technology and Science, Pilani, India in 1992 and the Ph.D. degree in Electrical Engineering from Carleton University in 1998. Dr. Achar currently is a professor in the department of electronics engineering at Carleton University. Prior to joining Carleton university faculty (2000), he served in various capacities in leading research labs, including T. J. Watson Research Center, IBM, New York (1995), Larsen and Toubro Engineers Ltd., Mysore (1992), Central Electronics Engineering Research Institute, Pilani, India (1992) and Indian Institute of Science, Bangalore, India (1990). His research interests include signal/power integrity analysis, circuit simulation, parallel and numerical algorithms, EMC/EMI analysis and mixed-domain analysis. Dr. Achar is a practicing Professional Engineer of Ontario and is a Fellow of IEEE and a Fellow of Engineers Institute of Canada.

Dr. Achar has published over 200 peer-reviewed articles in international transactions/conferences, six multimedia books on signal integrity and five chapters in different books. Dr. Achar received several prestigious awards, including Carleton university research achievement awards (2010 & 2004), NSERC (Natural Science and Engineering Research Council) doctoral medal (2000), University Medal for the outstanding doctoral work (1998), Strategic Microelectronics Corporation (SMC) Award (1997) and Canadian Microelectronics Corporation (CMC) Award (1996). He was also a co-recipient of the IEEE best transactions paper award for T-AdvP (2007) and T-CPMT (2013). His students have won numerous best student paper awards in international forums. He is a founding faculty member of the Canada-India Center of Excellence, chair of the joint chapters of CAS/EDS/SSC societies of the IEEE Ottawa Section, and is a consultant for several leading industries focused on high-frequency circuits, systems and tools.

Yongwoo Lee (Member, IEEE) received his B.S. degree in Electrical Engineering and Computer Science from the Gwangju Institute of Science and Technology, Korea, in 2015. He obtained his M.S. and Ph.D. degrees in Electrical and Computer Engineering from Seoul National University, Korea, in 2017 and 2021, respectively. Prior to his current role as an Associate Professor at Inha University, he worked as a Staff Researcher at the Samsung Advanced Institute of Technology (SAIT). His primary research interests focus on privacy-enhancing technologies.