High Throughput Acceleration of Scabbard Key Exchange and Key Encapsulation Mechanism Using Tensor Core on GPU for IoT Applications

Author's Version

Muhammad Asfand Hafeez*, *Graduate Student Member, IEEE,*, Wai-Kong Lee‡, *Member, IEEE,* Angshuman Karmakar†, *Member, IEEE,* and Seong Oun Hwang‡, *Senior Member, IEEE*

***Abstract*—High throughput key encapsulations and decapsulations are needed by IoT applications in order to simultaneously process a multitude of small data in secure communication. In this paper, we present two novel techniques for accelerating the implementation of polynomial convolution on a GPU, utilizing advanced Tensor cores, which benefit the performance of key encapsulations. First, a polynomial re-structuring technique is proposed to allow several polynomials with distinct public keys to be processed in a single communication cycle. This is an improvement compared to the previous work by Lee et al. Next, we observe that polynomial convolution in some key encapsulation mechanisms contains reduction patterns that are not friendly to parallel implementation. We propose separating the multiplication and reduction processes so they can be parallelized independently. To verify the effectiveness of our proposed techniques, we applied it to two key-encapsulation mechanisms from the Scabbard post-quantum key-encapsulation mechanism suite and evaluate their performance. Experimental results show that polynomial convolution using Tensor cores is 1.05× faster (for the Florete scheme) and 3.6× faster (for the Sable scheme) than using CUDA core-based multiplication with conventional cores on a GPU. The Tensor cores based encapsulations and decapsulations are faster than a reference implementation on a CPU supporting AVX2 by more than 5.6× and 6.4×, respectively, for the Florete scheme and 8.3× and 13.3× faster, respectively, for the Sable scheme. This shows that the proposed techniques can achieve significantly higher throughput for key exchange and encapsulation mechanisms, which are important for securing IoT applications.**

***Index Terms*—Cryptography, Polynomial Convolution, Tensor Cores, Post-quantum Cryptography, Lattice-based Cryptography, General Matrix Multiplication (GEMM).**

I. Introduction

With the advent of the quantum computer (QC), the security of many known classical cryptographic algorithms such as Rivest-Shamir-Adleman (RSA) and Elliptic-curve cryptography (ECC) is in danger. This is because the mathematically hard problems that form the foundation of classical cryptography can be solved by a QC at a much faster speed when employing Shor's algorithm [1] and as described by Proos and Zalka [2]. This is a severe threat to classical cryptography, which has been used for more than four decades. To mitigate such threats, post-quantum cryptography (PQC) has been actively researched since the early 2000s, which primarily aims to develop public-key cryptography that can resist the threats from the QC. In 2016, the National Institute of Standards and Technology (NIST) [3] initiated a standardization process to select PQC schemes such as the key encapsulation mechanism (KEM) and the signature schemes to be used in the post-quantum world. Since the introduction of this standardization process, various efficient implementations have been proposed on different platforms, such as the field programmable gate array (FPGA) [4], Apple processors [5], and the microcontroller [6]. Currently, NIST has shortlisted four candidates to be used as a standard: the CRYSTALS-KYBER KEM [7], CRYSTALS-Dilithium [8], FALCON [9], and SPHINCS$^+$ [10] signatures.

A. Development of PQC After the NIST Standardization

Although the selection of the Kyber algorithm as a standard is an important milestone in the development of PQC, that does not indicate that the field of PQC will stop innovating and evolving. Instead, the standardization process has served as a roadmap for future development and improvement of PQC schemes. For example, during this standardization process, we saw that aggressive and non-conventional choices of parameters (e.g., LAC and Round 5) [11] had been discouraged. These choices can be potentially exploited by attackers, leading to vulnerabilities in a PQC system. Similarly, the use of non-constant-time error correction codes which are very crucial in lattice-based PQC schemes. These codes can be sources of side-channel attacks, which can compromise the security of the system. Therefore, the use of error correction codes in PQC schemes must be carefully considered and evaluated to ensure they do not introduce vulnerabilities. Hence, a continuous effort to improve the existing PQC schemes while not compromising their security is still a highly productive research direction. For instance, Mera et al. [12] proposed Scabbard, a suite of KEM schemes that improve on Saber, the NIST PQC finalist [13]. Similarly, Liang et al. [14] proposed an improved version of the NTRU KEM [15] which was also a finalist in the NIST's standardization procedure. In this work, we focus only

The author* is with the Department of IT Convergence Engineering, Gachon University, Seongnam 13120, South Korea (e-mail: muhammadasfandh@gmail.com) and authors‡ are with the Department of Computer Engineering, Gachon University, Seongnam 13120, South Korea (e-mail: waikong.lee@gmail.com; sohwang@gachon.ac.kr). The author† is with the Department of Computer Science and Engineering, Indian Institute of Technology, Kanpur 208016, India (e-mail: angshuman@cse.iitk.ac.in) and with imec-COSIC, Department Electrotechniek, KU Leuven Belgium (e-mail:angshuman.karmakar@esat.kuleuven.be). (Corresponding author: Seong Oun Hwang.)

on Scabbard [12], which utilizes a modified version of the ring-learning-with-errors (RLWE) problem, that has been widely studied and analyzed. This allows Scabbard [12] to benefit from the extensive research and analysis conducted on the RLWE problem, giving it a strong foundation. Furthermore, Scabbard [12] was designed with trading security and real-world efficiency taken into consideration, making it a suitable candidate for practical implementations. Scabbard [12] offers improved security and efficiency compared to the original Saber scheme, making it a strong choice for an alternate PQC scheme in real-world implementations.

B. Deploying PQC Schemes on IoT Applications

Besides security aspects, current research interest is gradually increasing towards employing PQC schemes on unconventional hardware platforms such as massively parallel processors like the graphics processing unit (GPU) [16], instruction set architecture (ISA) [17] and the application-specific instruction set processor (ASIP) [18]. The growing trend of developing future applications around the Internet of Things (IoT) paradigm has spurred interest in enhancing security through using PQC schemes on non-traditional hardware platforms such as [19], [20], [21]. This academic pursuit seeks to explore and expand upon potential solutions for securing IoT sensor nodes and systems against emerging threats.

Many sensor nodes that are major constituents of the IoT have constrained computational resources, memory, and power and require an ASIP to support cryptographic operations. On the other hand, cloud servers in an IoT network need to handle massive amounts of communication from many sensor nodes, so it is natural to employ an accelerator like a GPU to offload cryptographic operations. This paper focuses on the latter case, wherein efficient techniques based on the latest GPU architecture are developed to accelerate PQC KEMs. Owing to its massively parallel cores, the GPU has been widely used to accelerate cryptographic schemes. Tensor cores are specialized components of the advanced GPU by NVIDIA manufactured after 2017, which is faster than the standard Compute Unified Device Architecture (CUDA) cores. Several applications for deep neural networks use NVIDIA's Tensor cores to increase the training and inference performance [22]. However, it was not widely used in the realm of cryptography, with very few published works available [23], [24]. The use of Tensor cores improves the speed of polynomial convolution, which is the key to achieving a high throughput KEM implementation on the GPU.

C. High Throughput Implementation of Key Exchange Schemes and Key Encapsulation Mechanisms on a GPU

In a key encapsulation mechanism (KEM), the public keys are non-ephemeral, i.e., they do not need to be changed after each communication session. But in key exchange (KEX) schemes, the public keys are ephemeral, i.e., they have to be changed after each session. Lattice-based KEM schemes are usually built on top of KEX schemes by applying Fujisaki-Okamoto transformation [25]. Owing to the limited resources in IoT sensor nodes, KEX schemes (using ephemeral keypairs) that are more computationally lightweight are more suitable than KEM using non-ephemeral keypairs.

Most lattice-based PQC schemes involve polynomial convolution, which is the most time-consuming component. Various methods have been proposed in order to speed up polynomial convolution on different platforms [26], [27], [28], [29]. Similarly, there are efforts to implement PQC candidates from the NIST standardization procedure on GPU platforms [30], [31], [32]. For instance, Supersingular Isogeny Key Encapsulation (SIKE) was accelerated by using a GPU [33]. Recently, Lee et al. [23] proposed the first implementation of NTRU (a PQC KEM) using the Tensor cores, which showed a significant speed-up compared to the standard CUDA cores. The proposed solution also applies to other lattice-based schemes, including FrodoKEM and LAC [23]. However, this high performance comes with a price: it is only applicable to the same keypair (non-ephemeral keypair), limiting the applicability of such implementations in IoT applications. DPCrypto [16] was later proposed wherein a dot-product instruction is used to compute polynomial convolutions. This allows the use of ephemeral keypairs, but the performance is slower than Tensor cores based implementations. But how to efficiently utilize the Tensor cores for computing a KEX with ephemeral keypairs remains an open research problem.

Tensor cores based polynomial convolution first pack the public key into matrix form before performing General Matrix Multiplication (GEMM). The previous approach proposed by Lee et al. [23] packed one public key into cyclic form, so GEMM using the Tensor cores can only perform polynomial convolution between one fixed public key and multiple random polynomials. This breaches the KEX requirement wherein the keypair must be refreshed after every instance.

In this paper, we present a new approach to resolving this issue. Our techniques allow polynomial convolution to be computed by Tensor cores while at the same time avoiding the issue with non-ephemeral keypairs. The proposed Tensor cores based solution was applied to two of the KEMs in Scabbard suite [12] (Florete and Sable) to evaluate its effectiveness in achieving high throughput KEX and KEM. The following are our key contributions.

1) The previous work that utilizes Tensor cores to compute polynomial convolution was proposed by Lee et al. [23]. It may suffer from security issues because one of the matrices is generated from a fixed polynomial, which violates the ephemeral keypair requirement in KEX. In this paper, a technique was proposed to restructure and pack multiple random polynomials (usually public keys) into a matrix before computing the GEMM. This allows our implementation to handle polynomial convolution between different public keys and multiple random polynomials. This also implies that the proposed Tensor cores based polynomial convolution can use ephemeral keypairs, which makes it a more secure way to implement KEX compared to the technique proposed by Lee et al. [23].
2) The proposed Tensor cores based polynomial convolution was evaluated on two lattice-based KEMs described in [12]: Florete and Sable. Sable employs nega-cyclic

polynomial convolution, which can be parallelized naturally [23] since there is no dependency between each polynomial coefficient. This makes it a good candidate for employing Tensor cores in polynomial convolution. On the other hand, Florete is constructed using the ring learning with rounding (RLWR) approach, which includes 768×768 polynomial multiplication and different reduction patterns for various parts. Therefore, reduction in Florete is dependent on multiple patterns, where multiple synchronizations are required to avoid data race issues. This makes it challenging to parallelize polynomial convolution using Tensor cores. To resolve this issue, we propose to first perform polynomial multiplication (which can be accelerated by Tensor cores) followed by various reduction steps that can be parallelized separately. This allows a high throughput implementation of Florete supported by the proposed Tensor cores based polynomial convolution.

3) For the implementation of Sable KEX on an RTX3060 Ti GPU, our Tensor cores based implementation can achieve $456,204$ encryptions per second and $557,2971$ decryptions per second, which is $1.7\times$ faster and $3.1\times$ faster, respectively, than using standard CUDA cores. The highest throughput achieved by a Sable KEM was $228,699$ encapsulations per second and $390,775$ decapsulations per second. Our Tensor cores based polynomial convolution achieved up to a $1.56\times$ speed-up, compared to the recent implementation of a Saber KEM [16] that had the same polynomial length as Sable. For the implementation of a Florete KEX, the throughput achieved by the Tensor cores based implementation was $230,202$ encryptions per second and $472,227$ decryptions per second, which is $1.1\times$ faster and $1.07\times$ faster, respectively, than standard CUDA cores. The highest throughput of the Florete KEM was $167,748$ encapsulations per second and $172,757$ decapsulations per second.

Experimental results showed that the proposed Tensor cores based implementation can achieve high throughput KEX and KEM. For IoT applications that have sensor nodes with constrained resources, KEX (ephemeral keypairs) is the preferred choice; our solution can be used to implement a high throughput secure KEX in such circumstances. In addition, for IoT applications that prefer to adopt KEM instead of KEX, our solution can be adopted to achieve high throughput, secure KEM. Note that our proposed method is applicable to any lattice-based encryption with a modulus of 11 bits maximum and having both cyclic and nega-cyclic polynomial convolution.

The rest of this paper is structured as follows. Section II introduces the Scabbard [12] post-quantum KEM suite and related research. Our techniques are presented in Section III. Experiment results and the analysis are presented in Section IV. We conclude the paper in Section V.

II. Background and Related Work

In this section, we first provide some background information before introducing Scabbard [12] as an improvement to the Saber design. We conclude by discussing current work on high throughput PQC implementations on GPUs and other hardware.

A. Background

Secure communication is crucial, particularly in the IoT as its many sensor nodes collect and transmit sensitive data across networks. Therefore, IoT security has become a necessity, but it is challenging to implement due to the application-specific complexity of infrastructures, the infrequent maintenance of deployments, and the limited processing power and memory capacity in sensor nodes. Moreover, the strict energy constraints on battery-powered sensor nodes commonly used in the IoT, and the sometimes compelling need for rapid time-to-market, may pose significant security risks. As a result, the use of cryptography to provide security should be acknowledged as essential, because a powerful QC will be able to easily break widely used conventional cryptographic algorithms. PQC arose as a new area in response to QCs, and NIST recently concluded a multi-round standardization procedure for PQCs. These new methods are based on different mathematical problems compared to traditional ones, including problems regarding lattices, codes, multivariate polynomials, and hash functions. These mathematical problems are expected, or are known, to be resistant to QCs. The primary assessment criteria for PQC standards are security analysis and implementation performance. Besides software implementation offered by the NIST PQC candidates, some researchers have also developed purely hardware or hardware-software co-processors [34][35] to improve the performance of these candidates. This can be very useful in protecting the communication in IoT applications, which is briefly illustrated in Figure 1. Sensor nodes frequently send data to an IoT gateway, which relays the sensor data to the IoT cloud. The sensor data are usually encrypted before being sent to the IoT gateway, typically achieved through symmetric key encryption algorithms such as AES. The encryption keys for all sensor nodes need to be refreshed periodically to avoid potential attacks (e.g., side channel attacks [36]) that could reveal the encrypted sensor data. KEX/KEM can be used to achieve this goal in IoT applications, where the encryption keys can be exchanged securely between the IoT cloud and sensor nodes. However, some IoT applications that utilize sensor nodes with constrained resources may prefer KEX over KEM owing to the relatively low overhead in computation. In such a scenario, an ephemeral public keypair is required for security. Regardless of whether KEX or KEM is used, handling a large volume of secure communications from sensor nodes is a daunting task even for high-end servers. Hence, it is beneficial to offload some cryptographic computations to accelerators like GPUs to ensure timely communication in IoT applications [37].

B. Scabbard: An Improved Saber

Scabbard, a lattice-based KEM suite proposed by Mera et al. [12], contains three KEM schemes: Florete, Espada, and Sable. By examining the most recent development in lattice-based cryptography, Mera et al. provided concrete instantiations of the schemes in the suite to address the issues of

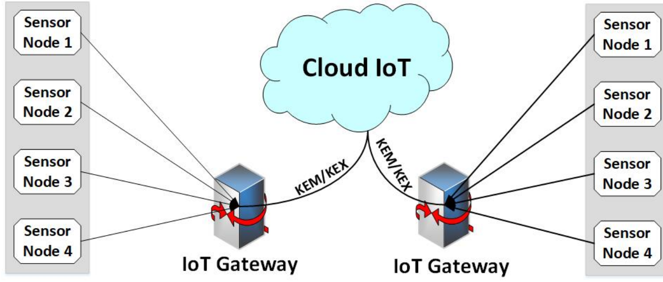


Fig. 1. A typical communication scenario in IoT application.

TABLE I
PARAMETERS OF FLORETE, SABLE AND ESPADA

Parameters	L	N	p	q	PQ Security	Moduli	Key Sizes
Florete	1	768	512	1024	2^{157}	ϵ_q: 10 ϵ_p: 9 ϵ_t: 3	PK: 896 SK: 1152 CT: 1248
Sable	3	256	512	2048	2^{169}	ϵ_q: 11 ϵ_p: 9 ϵ_t: 4	PK: 896 SK: 1152 CT: 1024
Espada	12	64	8192	32768	2^{128}	ϵ_q: 15 ϵ_p: 13 ϵ_t: 3	PK: 1280 SK: 1728 CT: 1304

constructing efficient cryptosystems from earlier ideas. These three KEMs are summarized below:

1) *Florete*: The main idea is to use ring lattices rather than module lattices. Also, Florete uses the highly optimized 256×256 polynomial multiplier that was already developed and improved for Saber on many platforms. This is achieved by combining 256×256 polynomial multiplication with Toom-Cook-3 multiplication. Therefore, to perform 768×768 polynomial multiplication in Florete, only seven 256×256 polynomial multiplications are required, compared to the nine 256×256 polynomial multiplications in Saber. Since the lattice size (768) is the same in both cases, this translates into a significant speed-up without compromising security. This makes Florete one of the fastest PQC KEMs available.
2) *Espada*: Similar to Saber, Espada uses the module learning with rounding (MLWR) technique but with degree-63 polynomials instead of degree-255 polynomials, as in Saber. Because the degree of the polynomial is proportional to the multiplier size, lower-degree polynomials help to reduce the footprint of hardware modules. Although Espada needs to perform 144 multiplications (compared to nine multiplications in Saber), the polynomial multiplications of Espada can be done in parallel. Thus, the Espada user has greater freedom to trade off speed and area, compared to other PQC schemes, making it a very interesting scheme for hardware implementations.
3) *Sable*: Based on the hard lattice problem known as learning with rounding (LWR). It can be shown that the LWR problem is at least as hard as the LWE problem [38]. In these schemes, the errors are generated implicitly using rounding instead of adding explicitly as in LWE. Mera et al. showed how errors can be properly estimated when they are generated implicitly by rounding [12]. Since errors are crucial for determining the security of lattice-based schemes, it is important to estimate them properly to eliminate the possibilities of overestimation or underestimation. This enabled Mera et al. to improve the parameters of Saber without compromising its security. This allowed a reduction of the key sizes, thereby reducing the bandwidth of Saber. This altered Saber is called Sable.

The security of the Scabbard [12] key-encapsulation mechanism is based on the hardness of the LWR problem in order to estimate the errors properly. Protocol 1 depicts the generalized LWR based KEX for Scabbard. It allows Alice and Bob to establish a shared secret key without revealing their private keys to each other. The proposed KEM is designed to be secure and efficient, with key encapsulation being performed using the recipient's public key, and the shared secret key being generated only by the intended parties. It involves three algorithms: key generation (Algo.1), encapsulation (Algo. 2), and decapsulation (Algo. 3).

Algorithm 1 generates a public key (PK) and a private key (SK) based on the security parameter *N*. Algorithm 2 accepts a PK as input and generates ciphertext (CT) and a shared secret key (K). Algorithm 3 performs decapsulation and accepts PK, CT and SK as input and returns the shared secret key as output. In Algorithms 1-3, H and $\mathcal{G}$ represents hash functions. h_1, h_2 and h_3 are constant polynomials having coefficients of $2^{(\epsilon_q-\epsilon_p-1)}$, ($2^{(\epsilon_q-\epsilon_p-1)}$ + $2^{(\epsilon_q-B-1)}$ - $2^{(\epsilon_q-\epsilon_t-1)}$) and $2^{(\epsilon_q-\epsilon_p-1)}$, respectively. The values of different parameters *i.e.* L, N, p, q, ϵ_p, ϵ_q, and ϵ_t are given in Table. I.

Similar to other lattice-based schemes, the performance bottleneck in Scabbard [12] is also a polynomial convolution. For more details about Scabbard, see the original Mera et al. study [12]. The design parameters for these algorithms are given in I. Although Saber is an ideal lattice-based scheme because of the higher security level of Sable due to the significant value of moduli q, the bandwidth consumption of Florete (2048 bytes) is even less than Saber (2080 bytes), assuming we take into account the bandwidth usage of each scheme, i.e., the combined sizes of both public key and ciphertext.

C. Previous Work

Recently, a lot of work has been done on efficient implementations of PQC on various platforms to increase its versatility. For instance, Zhu et al. [39] demonstrated that the Karatsuba method may be used to construct for the Saber KEM a high-speed hardware multiplier architecture suitable for IoT-based applications. Similarly, Seo et al. [40] provided a SIKE-optimized implementation for the ARM processor. An and Seo et al. [41] presented new optimized parallel techniques for FrodoKEM and NewHope GPU implementations for IoT and cloud computing technologies. Recent work by Lee and Hwang [31] and Gao et al. [30] described the implementation of Kyber and NewHope KEM on the GPU platform using the NTT approach, to provide high throughput.

Protocol 1 Scabbard generalized key-exchange scheme

Alice		Bob
$seed_A \leftarrow \mathcal{U}(\{0,1\}^{256})$		
$r \leftarrow \mathcal{U}(\{0,1\}^{256})$		
A $\leftarrow gen_N^{L\times L}$(XOF($seed_A$)) $\in (\mathcal{R}_q^N)^{L\times L}$		
s $\leftarrow \beta_\eta((\mathcal{R}_q^N)^L)$		
b = $\lfloor$ A.s$\rceil_p \in (\mathcal{R}_p^N)^L$		
	$\xrightarrow{(b', seed_A)}$	
		$r' \leftarrow \mathcal{U}(\{0,1\}^{256})$
		A $\leftarrow gen_N^{L\times L}$(XOF($seed_A$)) $\in (\mathcal{R}_q^N)^{L\times L}$
		s$' \leftarrow \beta_\eta((\mathcal{R}_q^N)^L)$
		b$' = \lfloor A^T.s' \rceil_p \in (\mathcal{R}_p^N)^L$
		$u' = b^T.(s' \bmod p) \in \mathcal{R}_p^N$
		$c' =$ HelpDecode$(u') \in \mathcal{R}_t^N$
	$\xleftarrow{(b', c')}$	
$u =$b$'.(s \bmod p) \in \mathcal{R}_p^N$		
$c =$Decode $(u, c') \in \mathcal{R}_{2^B t}^N$		
$k =$ Encode (c)		$k' =$ Encode (u')
$key_{Alice} =$Hash(k)		$key_{Bob} =$ Hash(k')

Algorithm 1 KEM KeyGenreation

Data: nil

Result PK = ($seed_A$, b), SK = (s, H(PK), r)

1: $seed_A \leftarrow \mathcal{U}(\{0,1\}^{256})$
2: $r \leftarrow \mathcal{U}(0,1)^{256}$
3: A$\leftarrow gen_N^{L\times L}$(XOF($seed_A$)) $\in (\mathcal{R}_q^N)^{L\times L}$
4: s $\leftarrow \beta_n((\mathcal{R}_q^N)^L)$
5: b = bits(A.s + $h_1, \epsilon_q, \epsilon_p$) $\in (\mathcal{R}_q^N)^L$
   // Rounding
6: PK$\leftarrow$ ($seed_A$,b)$r \leftarrow_\$ \{0,1\}^{256}$
7: SK$\leftarrow$ (s, H(PK),r)
8: return
9: PK= ($seed_A$,b), SK = (s,H(PK), r)

Algorithm 2 KEM Encapsulation

Data: PK = ($seed_A$, b)

Result CT = (c', b'), key = K

1: $m' \leftarrow_\$ \{0,1\}^{256}$
2: $m =$ arrange_msg(m')
3: $(K', r') \leftarrow \mathcal{G}(m||H(\text{PK}))$
4: $r' \leftarrow \mathcal{U}(\{0,1\}^{256})$
5: A $\leftarrow gen_N^{L\times L}$(XOF($seed_A$)) $\in (\mathcal{R}_q^N)^{L\times L}$
6: s$' \leftarrow \beta_\eta((\mathcal{R}_q^N)^L)$
7: b$'$ =bits$(A^T.s' + h_1, \epsilon_q, \epsilon_p)$
   // Rounding
8: $u' = b^T.(s' \bmod p) \in \mathcal{R}_p^N$
9: c' =bits$((u' + h_3 - 2^{\epsilon_p - B} m), \epsilon_p, (\epsilon_t + B)) \in \mathcal{R}_{2^B t}^N$ ▷ HelpDecode
10: $K \leftarrow H(K', H(c'))$
11: return
   CT = (c', b'),key = K

Recently, Lee et al. [42] presented the Physical Unclonable Function (PUF) approach to implementing public key creation in SABER KEM for low-power devices, which is highly parallel and delivers high throughput. On the other hand, DPCrypto [16] utilized the dot-product instruction to speed up the matrix multiplication and polynomial convolution in post-quantum lattice-based cryptography methods. Following this work, TensorCrypto [23] explored Tensor cores to perform polynomial convolution. However, that work only supports non-ephemeral keypairs, which is not desirable for KEX owing to security concerns. Our work improves TensorCrypto [23] to support ephemeral keypairs and avoid such security limitations in KEX.

III. Proposed Technique

This section presents the proposed polynomial re-structuring technique for Sable and Florete. The technique allows parallel polynomial convolution to operate on different input polynomials, as opposed to the fixed one in TensorCrypto [23]. Additionally, parallel polynomial convolution is also implemented using CUDA cores in order to analyze the best implementation strategy for the targeted KEMs. Note that the proposed techniques are applicable to both KEX and KEM.

A. Proposed Polynomial Re-structuring Technique using Tensor Cores

Polynomial multiplication/convolution is one of the most time-consuming parts of most lattice-based cryptography schemes. Generally, polynomial convolution involves two polynomials: polynomial a, which usually represents the public/private key, and polynomial b with small coefficients, representing the random elements. For instance, polynomial b in Sable and Florete is ternary, i.e., $b = \{-1, 0, 1\}$.

The Tensor cores primarily designed to accelerate the throughput of artificial intelligence (AI) by performing matrix

Algorithm 3 KEM Decapsulation

Data: PK = ($seed_A$, b), SK = (s,H(PK),r), CT = (c',b')
Result key = K
1: $u =$b'.(s mod p) $\in \mathcal{R}_p^N$
2: $m_1' =$ bits $((u + h_2 - 2^{\epsilon_p - \epsilon_t - B} m), \epsilon_p, B) \in \mathcal{R}_{2^B}^N$ ▷ Decode
3: $m_1 =$ orignal_msg(m_1')
4: $m_2 =$ arrange_msg(m_1)
5: $(K_1', r_1') \leftarrow \mathcal{G}(m_2 || H(\text{pk}))$
6: A $\leftarrow gen_N^{L\times L}$(XOF($seed_A$)) $\in (\mathcal{R}_q^N)^{L\times L}$
7: $s_1' \leftarrow \beta_\eta((\mathcal{R}_q^N)^L)$
8: $b_1' =$ bits $(A^T.s_1' + h_1, \epsilon_q, \epsilon_p)$
// Rounding
9: $u_1' = b^T.(s_1' \bmod p) \in \mathcal{R}_p^N$
10: $c_1' =$bits$((u_1' + h_3 - 2^{\epsilon_p - B} m), \epsilon_p, (\epsilon_t + B)) \in R_{2^B t}^N$ ▷ HelpDecode
11: **if** $c' = c_1'$ **then**
12: return $K = H(K_1', H(c'))$
13: **else**
14: return $K = H(r, H(c'))$
15: **end if**

multiplication and accumulation (MMA) operations. TensorCrypto [23] demonstrated that cyclic polynomial convolution might also be conducted with the Tensor cores (Figure 2). Polynomial a is packed into cyclic form, while small polynomial b may be stored in column-major form. In this way, the GEMM performed by the Tensor cores eventually produces polynomial convolution between one fixed polynomial and multiple random polynomials. This approach, proposed by TensorCrypto [23], is effective when a large number of small polynomials (b) must be handled concurrently, which is typically found in the cloud environment. However, this technique repeatedly uses the same polynomial a for KEX/KEM, which breaches the requirement of the ephemeral keypair.

To allow the use of ephemeral keypairs in KEX, we need to ensure that multiple distinct polynomials (a) can be packed into matrix form. This requires a novel re-structuring method, which is proposed in this paper and illustrated in Figure 3. The original version in Figure 2 derives the output coefficients from the cyclic product of two polynomials. From Figure 2, we see that only polynomial a is arranged in a cyclic form while polynomial b is kept in the original form. See et al. [43] showed that if we vary the order of summations, the small coefficient polynomial b also shows a cyclic pattern throughout the computation (see Rearranged Form in Figure 3). Therefore, instead of only arranging polynomial a, as done in [23], we can arrange polynomial b in nega-cyclic order and rearrange polynomial a nega-cyclically by skipping odd-level arrangements as shown in Figure 3. By doing this, we can reduce the original matrix into a smaller one, which is helpful for implementing an embedded system that only needs to handle one polynomial convolution at a time [43]. However, this is not suitable for a GPU implementation, which needs to handle many KEXs at a time. Note that Matrix A and Matrix B in Figure 3 imply a nega-cyclic pattern, which is found in Sable.

In this paper, we extend the idea from See et al. [43] to support the computation of multiple polynomials. Referring to Figure 4, we propose to first restructure polynomials a and b into restructured form following the idea in [43], and then pack them into matrix form. For instance, polynomials a and x can be two unique public keys while b and y are two random polynomials, and the matrix multiplication produces the output polynomials c and z. By using this approach, the proposed method allows encapsulation/decapsulation with different polynomials a, which resolves the issue in TensorCrypto [23].

B. Parallel Polynomial Convolution using Tensor Core

Tensor cores in state-of-the-art NVIDIA GPU architectures (e.g., Turing and Ampere) are designed to handle 16×16 matrix multiplication in a warp (32 threads). To handle a bigger matrix, one can use multiple warps to compute separate portions of the matrix and then aggregate the results repeatedly to produce the final results. For instance, to multiply a 32-by-32 matrix, four warps are launched in parallel to perform 16×16 matrix multiplication. The other four warps calculate the other half of the matrix in parallel, as depicted in Figure 5. This means two iterations are needed to perform the operation. Finally, results are saved in Matrix *C* in parallel. $(N/16)^2$ warps and $N/16$ iterations are needed to perform a convolution of $N \times N$ matrices in parallel. However, in the case of multiplication, warps and iterations increase to $2 \times (N/16)2$ and $2 \times (N/16)$, respectively.

Algorithm 4 shows the Tensor cores based polynomial convolution that takes 16×16 input matrices. Matrix A is comprised of random public keys arranged in nega-cyclic form (in Sable) and in sequential sequence (in Florete), and Matrix *B* contains non-constant polynomials (plaintext). All matrices are kept in a one-dimensional array (i.e., global memory). First, the method initializes two fragments for the 16×16 sub-matrices and one fragment for collected results (lines 1-3). It cycles across Matrix A (row-major) and Matrix B (column-major) to multiply in parallel (lines 11-15). In each cycle, 16×16 sub-matrices are loaded from Matrix A and Matrix B (in global memory) for concurrent matrix multiplication. $(N/16)$ Each warp operates on separate regions of Matrix A and Matrix B. The collected results are transferred to Matrix *C* in global memory (line 17) in row-major form for Sable. However, in Florete, they are changed to column-major.

C. Implementation of Florete and Sable Based on Tensor Cores

The parameter sets in Florete, Sable, and Espada require a modulus of $q = (2^{10}, 2^{11}, \text{and } 2^{15})$, respectively. This allows polynomial coefficients to be expressed in a 16-bit floating point (FP16) representing a maximum of the 12-bit integer value. However, in the mentioned schemes, one of the polynomials is ternary (i.e., -1, 0, 1). Considering the worst case in which the element size is 11-bit, multiplication of the two numbers generates only an 11-bit result at maximum, i.e., $(2^{11} - 1) \times 1 = 2^{11} - 1$ and $(2^{11} - 1) \times (-1) =$

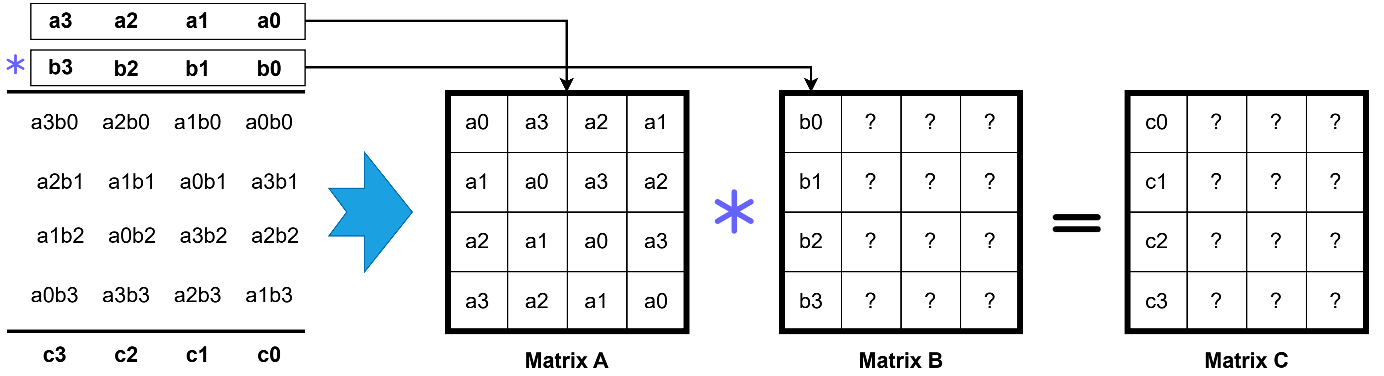


Fig. 2. The GEMM polynomial convolution technique proposed in [23]. In Matrix B, "?" indicates non-constant polynomials.

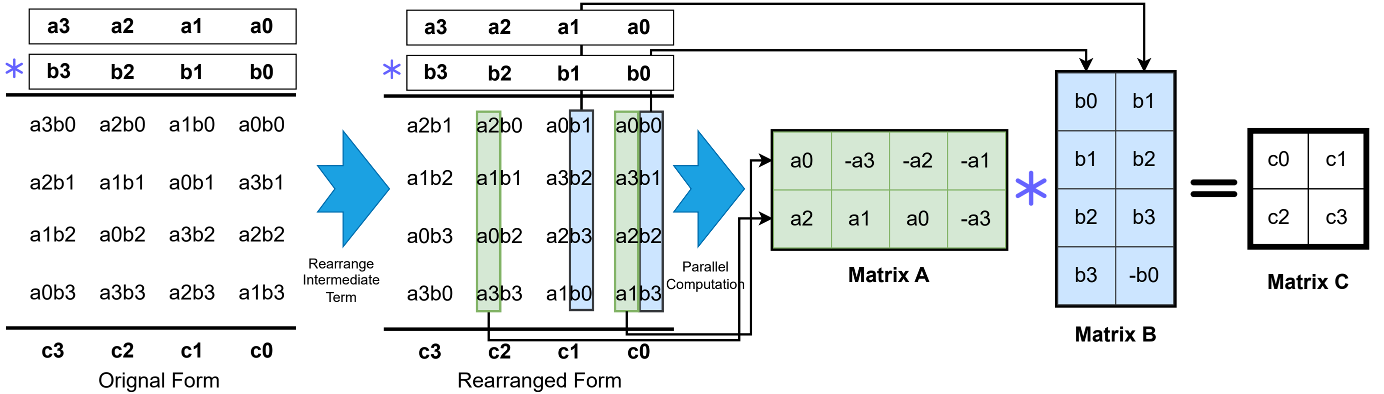


Fig. 3. Polynomial convolution using the proposed re-structuring technique [43].

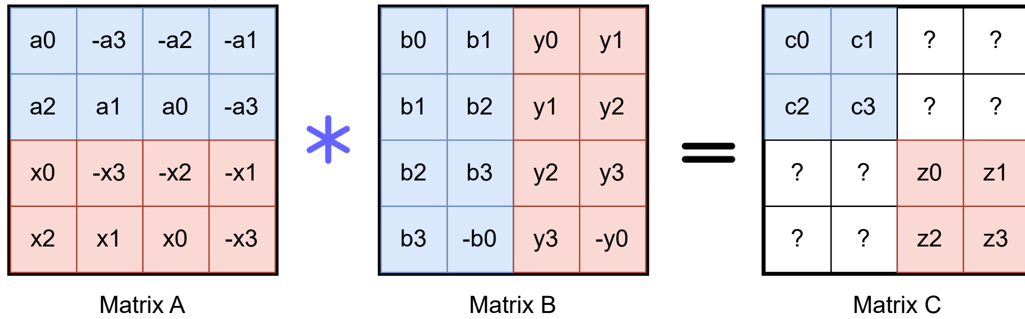


Fig. 4. Packing multiple polynomials a and b into matrix form.

$-2^{11}+1$. The accumulated value can grow to a maximum of $L \times N \times 2^{11}-1$ during the polynomial convolution process. Considering that N in Florete and Sable for the selected parameter sets are 768 and 256, respectively, and L is 1 and 3, respectively, the accumulator must hold at least 21-bit data for Florete, $\log_2(1 \times 768 \times (2^{11}-1))$, and 22-bit data for Sable, $\log_2(3 \times 256 \times (2^{11}-1))$. Nevertheless, the Tensor cores single precision accumulator may store 24-bit data at a maximum which is larger than the required threshold. Hence, the GEMM from Tensor cores can produce correct polynomial convolution in both schemes.

In Espada, the selected N and L parameters are 64 and 12, so the accumulator must hold at least 25-bit data, $\log_2(64 \times 12 \times (2^{15}-1))$, which is larger than the limit of the accumulator. Thus, Espada cannot be evaluated using Tensor cores.

1) Polynomial Convolution of Sable using Tensor Cores: Polynomial convolution is essentially the polynomial multiplication by two polynomials followed by a reduction process. It can be parallelized efficiently on a GPU using Tensor cores if the reduction process does not have any data dependencies [23]. This is the case for many existing lattice-based schemes like Saber [13] and Sable [12].

The proposed technique to perform parallel polynomial convolution is described in Algorithm 5. Note that this is different from TensorCrypto [23] because our approach restructures and packs the polynomials in a different way so it can process

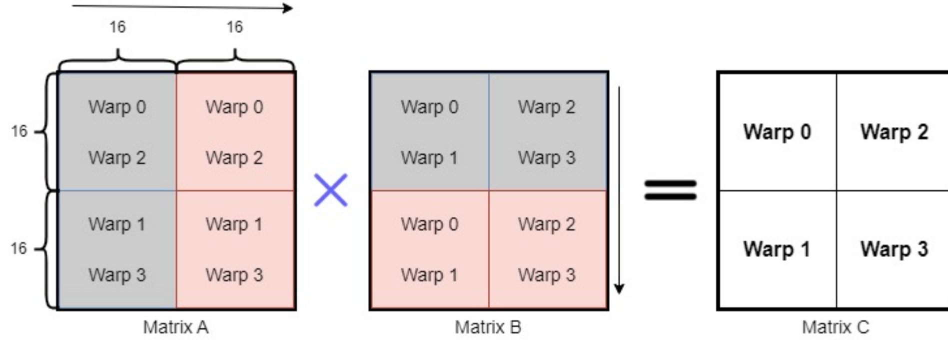


Fig. 5. Matrix multiplication in Tensor cores: warps are executed in parallel in a matrix of 32×32; the arrow denotes the computation sequence.

Algorithm 4 Tensor cores: parallel polynomial convolutions.

Input: Matrix A $(N \times N)$ and matrix B $(N \times N)$ with non-constant polynomials b, where N is a multiple of 16.
Output: $N \times N$ matrix C holds the nega-cyclic convolution of distinct polynomials a and b.

// 16×16 `with precision FP16 initialization of fragment A & B`
1: $fragment < A, 16, 16, 16, half, row_major > a_frag$
2: $fragment < B, 16, 16, 16, half, col_major > b_frag$
// 16×16 `with precision FP32 initialization of fragment C`
3: $tid = thread$ ID
4: $bid = block$ ID
5: $blockDim =$ block dimension
6: $ID_{warp} = (bid \times blockDim + tid)/32$
7: $row_{idx} = (ID_{warp}\%(N/16)) \times 16$
8: $col_{idx} = (ID_{warp}/(N/16)) \times 16$
9: $accu_{idx} = row_{idx} + col_{idx} \times N$
10: **for** i from 0 to $(N/16)$ **do**
11: $A_{id} = row_{idx} \times N + i \times 16$
12: $B_{id} = col_{idx} \times N + i \times 16$
13: $load_matrix_sync(a_frag, A + A_{id}, N)$
14: $load_matrix_sync(a_frag, A + B_{id}, N)$
15: $mma_sync(c_frag, a_frag, b_frag, c_frag)$
16: **end for**
// `Store c_frag output in C`
17: store_matrix_sync(C + $accu_{idx}$, c_frag, N, row_major)

multiple unique polynomials a. Referring to Algorithm 5, lines 1 and 2 calculate the number of blocks launched for the Tensor cores. Polynomials a and b are packed and rearranged into decomposed matrices (lines 4 and 5) as illustrated in Figure 4. After packing the polynomials into matrix form, the polynomial coefficients in unsigned 16-bit integer (U16) format are also converted to FP16 to allow computation in Tensor cores. In line 6, Tensor cores are launched to perform the polynomial convolution, where the nega-cyclic even terms are stored in matrix $fp32_c$ with 32-bit floating point (FP32) precision. Finally, the output from Tensor cores is transformed from FP32 to U16 in Line 7. The number of threads launched in lines 4, 5, and 7 is determined by the number of polynomials a we required to pack, which is denoted as $Poly_a$.

Algorithm 5 Tensor core implementation of Sable polynomial convolution in parallel on the GPU

Input: Polynomial a (constant), polynomial b (non-constant), modulus q
Output: M×M Matrix c holds the nega-cyclic convolution of polynomial a with polynomial b.

// `Calc. total number of threads required`
1: $threads_tot = 32 \times (N/16)^2$
// `Calc. number of blocks`
2: $tc_blocks = threads_tot/max_threads$
// `Number of thread`
3: $tc_threads = max_threads$
4: ParNegCycA$< N, N/Poly_a > (fp16_A, a)$ ▷ Alg.6
5: ParNegCycB$< N, N/Poly_a > (fp16_B, b)$ ▷ Alg.7
6: TensorCore$< tc_blocks, tc_threads >$ $(fp16_A, fp16_B, fp32_C)$ ▷ Alg.4
7: SableFP32toU16$< N, N/Poly_a > (c, fp32_C)$ ▷ Alg.8

To perform polynomial convolution in Sable, we first rearrange polynomials a into a nega-cyclic pattern by skipping the odd rows, as shown in Matrix A in Figure 3. After rearranging a, a small polynomial b is also rearranged in the nega-cyclic pattern to construct the matrix in a multiple of 16×16 to perform polynomial convolution in the Tensor cores. This process is described in Algorithm 6, where input polynomial *in* (representing a) is read in parallel by $N/Poly_a$ threads and N blocks. This is followed in Line 3 by modulo $2 \times tid - bid$ to obtain the rearranged terms according to the required nega-cyclic pattern to generate Matrix A seen in Figure 4, for polynomial convolution. Line 3 of Algorithm 6 is essential since it arranges polynomials in cyclic order and determines the intermediate values during rearrangement. Lines 5-9 describe two cases, first, when $idx < 0$, the terms are rearranged into nega-cyclic form (Line 6); second, when $idx \geq 0$ they are rearranged in a cyclic way (Line 8). The terms are added to the output matrix (out). Each block of the GPU arranges polynomials in a unique set of nega-cyclic patterns in which rows start with an even number of polynomials, which are arranged to achieve a high degree of parallelism.

Algorithm 7 is similar to Algorithm 6. First, input poly-

Algorithm 6 ParNegCycA: parallel algorithm to arrange polynomial A in rearranged nega-cyclic pattern

Input: N-length polynomial in
Output: Matrix out of $N \times N$ dimensions, with a polynomial arranged in a nega-cyclic form.

```
1:  tid = thread ID
2:  bid = block ID
3:  idx = 2 × tid − bid
    // Launch N blocks and N/Poly_a threads
    // in parallel
    // Poly_a represents the number of Poly-
    // nomials a we packed
4:  if tid < N then
5:      if idx < 0 then
6:          out[bid + tid × N] = in[(idx)%N] × (−1)
7:      else
8:          out[bid + tid × N] = in[(idx)%N]
9:      end if
10: else
11:     out[bid + tid × N] = 0
12: end if
```

Algorithm 7 ParNegCycB: parallel algorithm to arrange small polynomial b in nega-cyclic pattern

Input: N-length polynomial in
Output: Matrix out of $N \times N$ dimensions, with a polynomial arranged in a nega-cyclic pattern.

```
1:  tid = thread ID
2:  bid = block ID
3:  idx = tid + bid
    // Launch N/Poly_a blocks and N threads
    // in parallel
    // Poly_a represents the number of Poly-
    // nomials a we packed
4:  if tid < N then
5:      if idx > (N − 1) then
6:          out[bid + tid × N] = in[(idx)%N] × (−1)
7:      else
8:          out[bid + tid × N] = in[(idx)%N]
9:      end if
10: else
11:     out[bid + tid × N] = 0
12: end if
```

Algorithm 8 SableFP32toU16: parallel algorithm to convert polynomial coefficients from FP32 to U16 and perform modulo p

Input: $N \times N$ matrix in with elements in FP32 format.
Output: $N \times N$ matrix in with elements in U16 format and modulo p.

```
1: tid = thread ID
2: bid = block ID
   // Launch N blocks and N/Poly_a threads
   // in parallel
   // Poly_a represents the number of Poly-
   // nomials a we packed
3: while i = Poly_a do
4:     out[bid×N+(N/Poly_a) × i + tid] =
       (int32_t)in[bid × N + N × i + tid]%p
5: end while
```

nomial in is read in parallel by N threads and $N/Poly_a$ blocks, followed by modulo $tid - bid$ to yield the arranged terms in nega-cyclic form as depicted by Matrix *B* of Figure 4. In Line 5, if $threads - blocks$ is higher than $(N-1)$, it represents rotating rows in the nega-cyclic form to make it different from Algorithm 6, where nega-cyclic even terms are rearranged. The terms are then moved to the output matrix (out) in FP16 format.

In reference to Algorithm 8, operations in Line 4 depend on the number of polynomials used (a). To maintain the prior degree of precision, Algorithm 8 first transforms FP32 elements to the INT32 format, then performs modulo q, and stores the final results in U16 format.

2) Polynomial Convolution of Florete using Tensor Core: Florete [12] uses three different reduction patterns to reduce the multiplication result between two polynomials. In other words, the reduction pattern in Florete is fragmented into three parts, as shown in Figure 6. For instance, polynomials A and B in Figure 6 are first multiplied and produce polynomials C as output, which are reduced to polynomials P. During reduction, polynomial $P1$ uses a nega-cyclic rounding pattern, while polynomial $P2$ uses a simple nega-cyclic pattern. The patterns for polynomials $P3$ and $P4$ are distinct from the preceding patterns. Therefore, parallelizing three different patterns for polynomial convolution in Florete is not possible, unlike Sable, which has only one reduction pattern.

Since we cannot perform parallel polynomial convolution in Florete due to the data dependency issue in the reduction process, we need to perform multiplication first, followed by the reduction process. Referring to Algorithm 9, the number of warps doubles from $(N/16)^2$ to $2 \times (N/16)^2$, thereby doubling the blocks required for matrix multiplication in Florete. Packing of polynomial a is done in two parts (lines 4 and 5). The Tensor cores are then used for polynomial matrix multiplication, resulting in Matrix $fp32_C$ being represented in the FP32 format (Line 7). In addition to accomplishing the reduction, the algorithm applied at Line 8 transforms the output of Matrix C into U16 format with modulo $p||q$.

Therefore, we propose performing polynomial multiplication first and then reduction, as depicted in Figure 7, to tackle the data dependency issue. For multiplication, the polynomial a is arranged in sequence as seen in Matrix A of Figure 7. In contrast, polynomial b is packed into column-major form. Matrix multiplication of Matrix A and Matrix B using Tensor cores produces Matrix C, which is reduced to Matrix D by performing a reduction on Matrix C.

The polynomial length in Florete is 768, so the resulting polynomial, after multiplying two polynomials, has a length of 1535. This indicates that polynomial a should be packed into 1535 rows in sequential order. However, one block in a GPU can only launch a maximum of 1024 threads, which cannot cover all the rows. Hence, we propose processing the

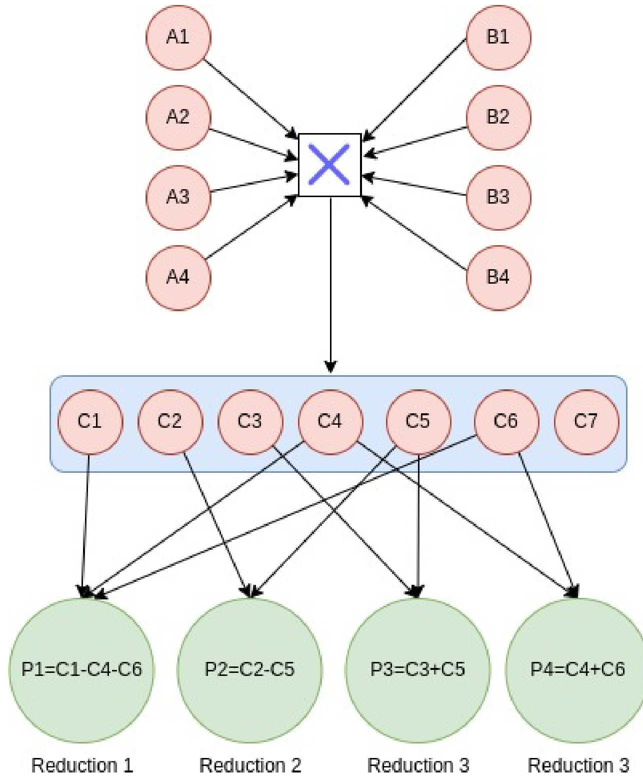


Fig. 6. Example of reduction patterns in Florete for four polynomials

Algorithm 9 Tensor core implementation of Florete polynomial convolution in parallel on the GPU

Input: Polynomial a, polynomial b, modulus $p||q$
Output: 2M×M Matrix c holds the nega-cyclic convolution of polynomial a with polynomial b.

```
    // Calculate total number of threads
    // required
 1: threads_tot = 32 × 2 × (N/16)^2
    // Calc. number of blocks
 2: tc_blocks = threads_tot/max_threads
    // Number of thread
 3: tc_threads = max_threads
 4: ParFirsthalf< N, N > (fp16_A, a)            ▷ Alg.10
 5: ParLasthalf< N, N > (fp16_A, a)             ▷ Alg.11
 6: ParU16toFP16< N, N > (fp16_B, b)            ▷ Alg.12
 7: TensorCore< tc_blocks, tc_threads >
    (fp16_A, fp16_B, fp32_C)                    ▷ Alg.4
 8: FloreteFP32toU16< N, N > (c, fp32_C)        ▷ Alg.13
```

rows in two parts, each part processing half of the rows in polynomial a. Algorithm 10 packs the first half of polynomial a in ascending consecutive sequence, whereas Algorithm 11 packs the second half in descending sequential order.

In Algorithm 10, N blocks and threads are launched in parallel. Lines 5-9 arrange the polynomial in ascending sequential order. Line 6 generates zero output when $(idx < 0)$, but, Line 8 packs the polynomials in the required sequence on output. Algorithm 11 works the same way as Algorithm 10 but in the opposite direction. In Line 6, when $idx < 0$, polynomials are packed on output in descending sequential order. Otherwise, the output is zero.

Algorithm 10 ParFirsthalf: Parallel algorithm to arrange polynomial A in ascending sequential order

Input: N-length polynomial in
Output: Matrix out with $N\times\ N$ dimension, with a polynomial arranged in consecutive sequence.

```
 1: tid = thread ID
 2: bid = block ID
 3: idx = tid − bid
    // Copy polynomials into shared memory
    // in parallel
 4: if tid < N then
 5:     if idx < 0 then
 6:         out[bid + tid × (2 × N)] = 0
 7:     else
 8:         out[bid + tid × (2 × N)] = in[(idx)%N]
 9:     end if
10: else
11:     out[bid + tid × N] = 0
12: end if
```

Algorithm 12 explains the conversion of U16 polynomial elements to FP16 format. Lines 5-8 are only required if we are working with ternary numbers; it converts -1 in integer format (i.e., 1023 or 511 depending on the moduli, we use q or p) to FP16 format.

Referring to Algorithm 13, lines 3-7 perform nega-cyclic reduction with rounding followed by a simple nega-cyclic reduction pattern in lines 8-12 for only one polynomial coefficient, while lines 13-17 rovide a cyclic rounding pattern. Hence, we can see that the reduction process is fragmented into three different parts, so it can only be partially parallelized. The multiplication results in FP32 format are converted to U16 by first transforming them to INT32 format and then reducing them through modulo p or q, depending on the algorithm. Even though we cannot fully parallelize polynomial convolution in Florete, we can still achieve relatively high throughput using the method we developed in this section.

D. Polynomial Convolution Using CUDA Cores

In addition to the Tensor cores based approaches, we analyzed multiplication using the conventional CUDA cores to achieve efficient polynomial convolution. Algorithm 14 shows that the schoolbook method for polynomial multiplication exhibits a high amount of parallelism. It can be parallelized by dividing the two polynomials into smaller sections and assigning each section to a separate thread (index j) to calculate the product.

Multiplying two polynomials of length N requires ($2N$-1) threads to compute the output. However, if a particular pattern is followed during reduction, the required threads reduce to N, and the entire polynomial convolution can be parallelized easily. This is found in Saber [13] and its previous GPU implementation [16]. In Florete, there are several reduction patterns, which makes it impossible to parallelize the entire

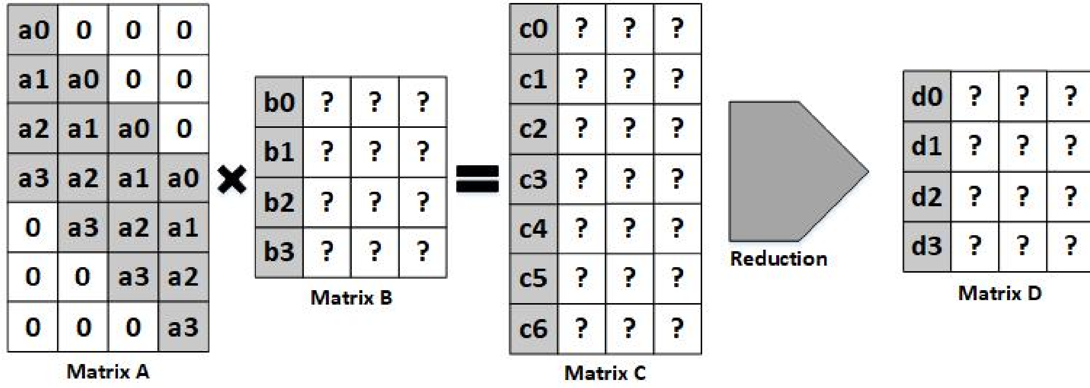


Fig. 7. Proposed solution for polynomial convolution in Florete

Algorithm 11 ParLastHalf: Parallel algorithm to arrange polynomial A in descending sequential order

Input: N-length polynomial in

Output: Matrix out with $N\times N$ dimension, with a polynomial arranged in consecutive sequence.

```
1: tid = thread ID
2: bid = block ID
3: idx = tid + bid
   // Copy polynomials into shared memory
   // in parallel
4: if tid < N then
5:     if idx < 0 then
6:         out[bid + tid × N] = in[(idx)%N]
7:     else
8:         out[bid + tid × N] = 0
9:     end if
10: else
11:     out[bid + tid × N] = 0
12: end if
```

Algorithm 12 ParU16toF16: Parallel algorithm to convert polynomial B from U16 to FP16.

Input: N-length polynomial in in U16 format.

Output: Matrix out with N with different polynomials of length N in F16 format.

```
1: tid = thread ID
2: bid = block ID
3: temp = 0
   // Launch N blocks and N threads
   // in parallel
4: temp = in[bid × N] + tid
5: if temp = p||q then
6:     out[bid × N + tid] = −1
7: else
8:     out[bid × N + tid] = temp
9: end if
```

polynomial convolution. In this paper, we propose performing the multiplication and reduction separately. The polynomial length of Florete is *768* which produces an output of *1535*. This means we need *1535* threads to compute the output, but in a GPU, we can launch a maximum of *1024* threads. To resolve this issue, we divide the polynomials into two halves, each with 768 coefficients. Referring to Algorithm 15, lines 5-7 compute the first part of the output (*768*), while lines 8-10 compute the remaining *767*. In the end, we return the output in Line 14 performing the modulus.

In the case of Sable, the reduction pattern is known (i.e., nega-cyclic). Therefore, parallel polynomial convolution can be performed in a more straightforward way. In convolution, multiplying two polynomials of length N uses N threads rather than $2N-1$ threads, as in multiplication. Algorithm 16 illustrates the parallel form of polynomial convolution in Sable that can be done on a GPU. This approach exploits N GPU blocks to produce N polynomial convolutions, where each GPU block computes one polynomial convolution using N threads. To decrease read/write latency, polynomials are first loaded from global memory and cached in shared memory, as done in Algorithm 6 from [16].

IV. Experimental Results and Discussions

This section describes the findings of our proposed methodology when applied to other techniques. Experiments were conducted on a PC equipped with a 2.10GHz Intel Core i7-12700F CPU and 16GB of RAM. Performance was evaluated using an NVIDIA RTX3060 Ti GPU. The implementation results were validated against the test vectors obtained from the original Scabbard implementation [12].

A. performance Evaluation of Tensor Cores based Polynomial Multiplication and Convolution

This experiment evaluated the performance of proposed Tensor cores approach versus conventional techniques based on CUDA cores. $(N/16)^2$ and $2\times(N/16)^2$ warps were launched for polynomial convolution and multiplication, respectively, based on the proposed Tensor cores implementation. Referring to Table II, we see that the CUDA cores based method performed better than the proposed Tensor cores based technique when the batch sizes were small. This is because Tensor cores have additional steps for rearranging poly_a and

Algorithm 13 FloreteFP32toU16: Partial parallel algorithm to convert polynomial coefficients from FP32 to U16 while performing reduction and modulo p or q

Input: $2N \times N$ matrix in with elements in FP32 format
Output: Matrix out with $N \times N$ dimension, with elements in U16 format and modulo (p or q).

```
1: tid = thread ID
2: bid = block ID
   //  Launching N blocks and N/2 threads at
   // maximum
3: if tid < (M/2 − 1) then
4:     out[bid + tid] = (int32_t) (in[bid + tid] − in[bid + N + tid] − in[bid + N + N/2 + tid])%(p||q)
5: else
6:     out[bid + tid] = 0
7: end if
8: if tid < 1 then
9:     out[bid + (N/2 − 1) + tid] = (int32_t) (in[bid + (N/2 − 1) + tid] − in[bid + (N/2 − 1) + N + tid]%(p||q)
10: else
11:     out[bid + (N/2 − 1) + tid] = 0
12: end if
13: if tid < N/2 then
14:     out[bid + N/2 + tid] = (int32_t) (in[bid + (N/2) + tid] + in[bid + N + tid]%(p||q)
15: else
16:     out[bid + N/2 + tid] = 0
17: end if
```

Algorithm 14 Serial schoolbook polynomial multiplication

Input: N-length polynomial a, N-length polynomial b, and q-modulus.
Output: $(2N-1)$-length polynomial c, which is the multiplication of a and b.

```
   // Accumulating each polynomial serailly
1: for i from 0 to N do
2:     for j from 0 to N do
3:         c[i + j] = c[i + j] + (a[i] × b[j])
4:     end for
5: end for
6: return c%(p||q)
```

Algorithm 15 Parallel schoolbook polynomial multiplication in Florete using CUDA cores.

Input: N-length polynomial a, N-length polynomial b, and q-modulus.
Output: (N-length polynomial c polynomial, which is the cyclic convolution of a and b.

```
1: tid = thread ID
2: bid = block ID
3: if tid < N then
4:     c[2 × bid + tid] = 0
5:     for i from 0 to tid do
6:         c[bid + tid] += (a[i] × b[tid − i])
7:     end for
8:     for i from 1 to (N − tid) do
9:         c[bid + N + tid] += (a[N − i] × b[tid + i])
10:    end for
11: else
12:     c[2 × bid + tid] = 0
13: end if
14: return c[2 × bid + tid] = c[2 × bid + tid]%(p||q)
```

Algorithm 16 Parallel schoolbook polynomial convolution in Sable using CUDA cores.

Input: N-length polynomial a, N-length polynomial b, and q-modulus.
Output: (N-length polynomial c polynomial, which is the cyclic convolution of a and b.

```
1: tid = thread ID
2: bid = block ID
   // Parallel copying polynomials into
   // shared memory
3: a_shared[tid] = a[bid × N + tid]
4: b_shared[tid] = b[bid × N + tid]
5: sum=0                         ▷ Using register to accumulate
6: for i from 0 to tid do
7:     sum = sum + (a_shared[tid − i] × b_shared[i])
8: end for
9: for i from 1 to (N − tid − 1) do
10:    sum = sum − (a_shared[tid + i] × b_shared[N − I])
11: end for
12: return c[bid × N + tid] = sum%q
```

poly_b into nega-cyclic form and converting poly_b from integer to floating point.

However, the advantage with Tensor cores becomes clear when N exceeds 64 in Sable. This is illustrated in Figure 8; we can see that the proposed Tensor cores technique surpassed the CUDA cores method for batch sizes between $2^6 < K < 2^7$. At a batch size of 2^{10}, Tensor cores outperformed the CUDA cores and recorded the best performance. We can see that the proposed Tensor cores based technique experienced a steady increase in throughput. Note that we do not report performance beyond 2^{10} because throughput is saturated at this point and does not increase any further.

In the case of Florete, the scenario is different. The Florete Tensor cores implementation surpassed the CUDA cores method at batch sizes between $2^8 < K < 2^9$. This is because packing of poly_a in Florete is completed in two parts due to the large polynomial length, as explained in detail in Section III-C2. Therefore, as illustrated in Figure 9, Florete's Tensor cores implementation outperformed CUDA cores based multiplication at a batch size of 2^9 and recorded the best performance at a batch size of 2^{10}.

We observed that the performance of the CUDA cores based implementation in Florete was worse than Sable. This is because we performed multiplication and reduction in Sable simultaneously (see Algorithm 16), but these two operations are performed separately in Florete. Furthermore, due to the large polynomial length and the restriction on the maximum

TABLE II
PERFORMANCE EVALUATION OF POLYNOMIAL MULTIPLICATION/CONVOLUTION IN FLORETE AND SABLE

Batch size (K)	Florete		Sable	
	Throughput (1000 multiplications per second)			
	CUDA cores	Tensor cores	CUDA cores	Tensor cores
1	8.59	0.901	41.61	15.186
16	137.10	13.79	655.21	241.546
32	273.90	26.43	1283.08	483.749
64	315.01	52.46	1451.25	957.940
128	343.71	110.01	1509.99	1909.878
256	389.89	211.36	1722.63	3745.976
512	397.25	418.38	1791.75	6465.98

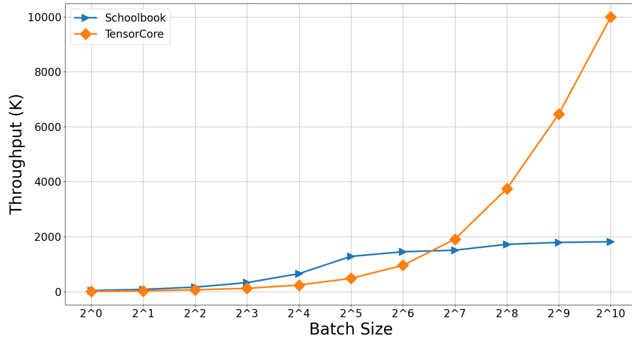


Fig. 8. Comparing throughput of GPU implementation of Sable with different techniques using various batch sizes

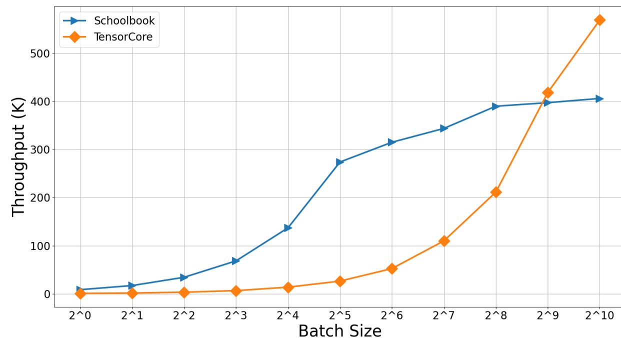


Fig. 9. Comparing the throughput of GPU implementation of Florete with different techniques using various batch sizes

number of threads per block (1024), we have to perform the polynomial convolution in Florete in two loops. This explains why CUDA cores are faster for Sable.

B. Performance Breakdown for Tensor Cores based Implementation

Table III provides the performance breakdown of polynomial convolution in Florete and Sable from utilizing the proposed Tensor cores technique. The outcomes were obtained by analyzing execution times at a batch size of $K = 128$. In Florete, organizing poly_a takes up about 8% of the overall time, whereas arranging poly_b takes up very little time. Polynomial convolution is still the most time-consuming step ($\approx$ 90%), although it is accelerated by the Tensor cores. Converting from FP32 format to U16 and simultaneously performing reduction requires time. The performance breakdown for Sable shows that rearranging poly_a in nega-cyclic intermediate form consumed the most time ($\approx$ 37%), whereas the arrangement of poly_b took only 13% of the total time. Multiplication in Tensor cores consumes almost the same time as packing poly_a. Performing modulo consumes nearly 13% of the entire time.

From the above discussion, we conclude that the computation of polynomial multiplication/convolution using Tensor cores took most of the time. For the case of Sable, elements are already arranged in order before performing the convolution. Arranging elements in a particular order for convolution created overhead. However, it reduced multiplications operations from $2N \times N$ to $N \times N$ in Tensor cores, which is the main reason Tensor cores consumed less time for multiplication in Sable compared to the Florete.

In Florete due to the data interdependence of the reduction pattern, it is challenging to fully parallelize it for convolution. As a result, we performed multiplication in Florete, and for multiplication, while packing poly_a, there were 0 redundant operations, as shown in Figure 7. These redundant operations slow the matrix multiplication process in the Tensor cores. This is the reason the Tensor cores required 90% of the overall time in Florete. This problem will need to be addressed in further research.

C. Performance breakdown of Florete and Sable KEX on GPUs and CPUs

Table IV summarizes the fundamental computations from encryption and decryption under Florete and Sable KEX schemes. The analysis considers the percentage of time spent for different operations on both the CPU and the GPU. In the GPU implementation of Florete, polynomial convolution consumed 94% and 96% of the overall time during encryption and decryption respectively. This shows that even after heavy optimization using the proposed Tensor cores approach, polynomial convolution remains the main bottleneck. This is due to the reduction patterns, where speeding them up is non-trivial. On the other hand, other operations that consume a significant amount of time in a CPU implementation, become insignificant compared to overall performance in the GPU

TABLE III
PERFORMANCE BREAKDOWN OF FLORETE AND SABLE FOR POLYNOMIAL MULTIPLICATION AND CONVOLUTION AT K=128

Operation	Florete		Sable	
	Time (μs)	%	Time (μs)	%
FirstHalf (Poly a $\rightarrow$ Algorithm 10)	49.22	4.09	-	-
LastHalf (Poly a $\rightarrow$ Algorithm 11)	48.63	4.04	-	-
Negcyc-InterArr (Poly a $\rightarrow$ Algorithm 6)	-	-	25.26	37.69
Negcyc-Arr (Poly b $\rightarrow$ Algorithm 7)	-	-	8.55	12.76
U16-FP16 (Poly b $\rightarrow$ Algorithm 12)	11.68	0.97	-	-
Tensor cores based polynomial convolution $\rightarrow$ Algorithm 4	1090	90.51	24.09	35.94
FP32-U16 $\rightarrow$ Algorithm 8 & 13	4.74	0.39	9.12	13.61
Total	1204.27	100	67.02	100

implementation. In the Sable GPU implementation, the computations were more evenly distributed for encryption between GenPoly/GenSecret (52%) and polynomial convolution (39%), which is similar to the CPU implementation. For decryption, the dominant computation comes from polynomial convolution, consuming up to 67% of the total execution time. In contrast to Florete, the proposed Tensor cores approach can speed up the polynomial convolution more effectively, due to the consistent reduction pattern. Note that the memory copy operation between the CPU and GPU had a minimal effect on the overall performance.

TABLE IV
PERFORMANCE BREAKDOWN OF FLORETE AND SABLE KEX SCHEMES

	Florete	RTX 3060 TI	CPU
Encryption	Mem. copy b/t CPU and GPU	1%	-
	GenPoly/GenSecret	2%	10%
	Polynomial convolution	94%	50%
	Packing/unpacking	2%	23%
	Others	1%	17%
Decryption	Mem. copy b/t CPU and GPU	1%	-
	Polynomial convolution	96%	79%
	Packing/unpacking	2%	13%
	Others	1%	8%
	Sable		
Encryption	Mem. copy b/t CPU and GPU	2%	-
	GenPoly/GenSecret	52%	21%
	Polynomial convolution	39%	65%
	Packing/unpacking	5%	10%
	Others	2%	4%
Decryption	Mem. copy b/t CPU and GPU	14%	-
	Polynomial convolution	67%	72%
	Packing/unpacking	13%	19%
	Others	6%	9%

D. Comparing KEX and KEM Performance on GPU and CPU

This section presents the experimental results from both KEX and KEM under Florete and Sable with different batch sizes (K) and using two types of GPU cores: CUDA and Tensor. Table V provides the performance of KEX for both schemes. At a batch size of 16, the Florete implementation on CUDA cores achieved $62,003$ encryption/s and $142,490$ decryption/s, while the Tensor cores achieved $7,533$ encryption/s and 15303 decryption/s. Our approach was $8.2\times$ and $9.3\times$ slower, respectively, than on CUDA cores based implementation. As the batch size increased, the throughput of both implementations also increased, achieving the highest throughput when $K = 512$. The Tensor cores approach records $240,202$ encryption/s and $482,227$ decryption/s, which are $1.1\times$ and $1.07\times$ times faster, respectively, than the CUDA cores implementation. Similar performance was also observed for our implementation with Sable. When batch size $K = 16$, the CUDA cores implementation provided throughput of $52,609$ encryption/s $400,641$ decryption/s, while the Tensor cores only achieved $45,553$ encryption/s and $229,990$ decryption/s. However, the performance from Tensor cores surpassed the CUDA cores when $K \geq 128$. The Tensor cores approach recorded $456,204$ encryption/s and $5,572,971$ decryption/s, which are $1.7\times$ and $3.1\times$ times faster, respectively, than the CUDA cores implementation.

From these results, we can see that the throughput with the Tensor cores approach was generally lower than the CUDA cores for small batch sizes. However, at larger batch sizes, Tensor cores approach gained the upper hand, so it is beneficial to use our method when the KEX/KEM workload is high.

Table VI shows the throughput from Florete and Sable KEMs on a GPU and a CPU supporting AVX2. Note that KEM is built on top of the KEX scheme; it has additional operations (e.g., hashing) in addition to the KEX. Hence, the throughput achieved by KEM is always lower than KEX. At batch size $K = 512$, the throughput of Florete with the proposed Tensor cores technique is 5.6$\times$ faster (encapsulation) and 6.4$\times$ faster (decapsulation) than AVX2 implementation. For Sable, the throughput for encapsulation and decapsulation, respectively, was 8.3$\times$ higher and 13.3$\times$ higher than the AVX2 implementation.

Similarly, the throughput of CUDA cores based multiplication at a batch size of 512 for encapsulation and decapsulation in Florete outperformed the AVX2 implementation by 5.4$\times$ and 6.1$\times$, respectively. Sable outperformed the AVX2 implementation by factors of 7.1$\times$ and 8.3$\times$ respectively. However, at smaller batch sizes, AVX2 performed better than the GPU implementation because a GPU is better suited for batch processing large KEX/KEM workloads. At the same time, AVX2 can improve the latency of a single operation

TABLE V
PERFORMANCE BREAKDOWN OF FLORETE AND SABLE KEX WITH DIFFERENT BATCH SIZES

Batch Size (K)	Florete				Sable			
	Throughput (encryptions/decryptions per second)							
	CUDA cores		Tensor cores		CUDA cores		Tensor cores	
	Encrypt	Decrypt	Encrypt	Decrypt	Encrypt	Decrypt	Encrypt	Decrypt
16	62003	142490	7533	15303	52609	400641	45553	229990
32	12114	276166	14999	30616	99522	771604	86805	439174
64	155086	340483	30120	61282	157035	1050972	155086	898876
128	176056	369925	59780	122264	212224	1319261	263157	1747488
256	211327	438813	118821	242204	254323	1628333	374812	3116713
512	218066	452194	240202	482227	267379	1752656	456204	5572971

by utilizing specialized SIMD instructions. In short, the GPU can be regarded as a throughput-oriented accelerator, while a CPU supporting AVX2 is helpful in improving the latency in individual operations.

E. Comparison with State-of-the-Art: DPSaber [16]

The Sable KEM is an improvement over Saber. Similar to Saber, polynomial convolution in Sable is used to compute inner product and matrix-vector multiplication. In 2022, Lee et al. presented DPSaber [16] which implemented Saber using dot-product instructions available in a GPU. Table VII comapres throughput from our work and from DPSaber [16]. Since the source code of DPSaber is open, we utilized their codes and re-performed the experiment on our GPU for a fair comparison.

Referring to the case of matrix-vector multiplication, when $K \leq 128$, DPSaber showed a higher throughput than our approach. When $K > 128$, the Tensor cores approach outperformed DPSaber, achieving $1.36\times$ higher throughput. Similarly, when $K = 256$, the Tensor cores approach can achieve at least $1.44\times$ higher throughput for the inner product. This shows that our approach is more advantageous than the dor-product method when the batch size is large enough because of the Tensor cores' higher instruction throughput, compared to dot-product instructions [44].

F. Discussion of the proposed technique to IoT applications

IoT systems are becoming more widespread and increasingly used for critical applications, such as industrial control systems, healthcare devices, and smart homes. IoT sensor nodes often collect and transmit sensitive information, making it crucial to secure communication and data against potential threats, including those posed by quantum computers. PQC offers a promising solution for securing communication and data against potential threats, but it is computationally intensive and requires significant resources to process cryptographic operations efficiently. Hence, accelerating the execution of these PQC algorithms ensures the security of IoT communication in the post-quantum era, at the same time, allowing IoT systems to handle a large number of connections to meet the need for growing communication in the future. In particular, the encryption keys used for encrypting the IoT sensor data are refreshed from time to time to ensure confidentiality. KEX or KEM can be used to exchange these encryption keys between sensor nodes, gateway and cloud server. The proposed acceleration methodologies presented in this article is able to produce high throughput implementations of PQC KEX and KEM, which fit the IoT systems very well. This can resolve the security issue in IoT communication while minimizing the impact on communication performance.

V. CONCLUSION

With a growing number of IoT sensor nodes, there is a need to accelerate the execution of KEM/KEX to ensure system performance and security. In this article, we proposed several techniques to showcase the ability of GPU and Tensor cores in achieving high throughput KEX and KEM. The proposed polynomial re-structuring technique allows distinct public keys to be handled by the Tensor cores based polynomial convolution. This approach can be used to implement high throughput KEX, which is useful in securing IoT applications. We also proposed a technique to enable parallel implementation of polynomial convolution with different reduction patterns. The proposed techniques are applied to two post-quantum KEMs, Sable and Florete. Compared to a traditional implementation using CUDA cores, the proposed Tensor cores based implementation achieved at least $1.1\times$ and $1.07\times$ faster encryption and decryption, respectively, in Florete. It also achieved $1.7\times$ and $3.1\times$ higher KEX throughput for encryption and decryption, in Sable. The proposed Tensor cores based polynomial re-structuring is a general approach that can be used for polynomial convolution and multiplication of different lengths.

In future work, we intend to explore the possibility of applying Tensor cores based techniques to other lattice-based cryptographic schemes that use number theoretical transform (NTT). One particularly interesting direction is fully homomorphic encryption [45] that needs to perform many polynomial convolutions with a large N. Such applications may benefit from the high computational performance of Tensor cores based polynomial convolution which has higher throughput than the CUDA cores.

ACKNOWLEDGMENT

The work of Muhammad Asfand Hafeez and Seong Oun Hwang was supported by the Gachon University research fund under Grant GCU-202110270001. The work of Wai-Kong Lee was supported by the Brain Pool Program through

TABLE VI
PERFORMANCE BREAKDOWN OF FLORETE AND SABLE KEM WITH DIFFERENT BATCH SIZES

Batch Size (K)	Florete						Sable					
	Throughput (encapsulation/decapsulation per second)											
	AVX2		CUDA cores		Tensor cores		AVX2		CUDA cores		Tensor cores	
	Encap	Decap	Encap	Decap	Encap	Decap	Encap	Decap	Encap	Decap	Encap	Decap
16			36179	37466	6899	7003			33179	37500	27847	31830
32			50520	51458	13623	11945			59463	70722	50286	60430
64	29885	26804	100275	105566	27144	27360	27368	29213	99432	121038	87885	116959
128			127262	132782	52717	46738			140260	172362	137831	211416
256			152491	159509	97900	99192			169970	217458	183070	331166
512			162367	163566	167748	172757			194615	242902	228699	390775

TABLE VII
COMPARISON OF THE PROPOSED TENSOR CORES IMPLEMENTATION FOR INNER PRODUCT AND MATRIX-VECTOR MULTIPLICATION IN THE SABLE KEM VERSUS DPSABER [16]

	Inner Product (thousand of operations per second)			Matrix-Vector (thousand of operations per second)		
K	Tensor core	DPSaber [16]	Sp-up	Tensor core	DPSaber [16]	Sp-up
64	958	1161	0.82	353	445	0.79
128	1910	1926	0.99	711	734	0.97
256	3746	2598	1.44	1362	1001	1.36

the National Research Foundation of Korea (NRF) funded by the Ministry of Science and Information Communication Technology (ICT) under Grant 2019H1D3A1A01102607. The work of Angshuman Karmakar was supported by Research Foundation-Flanders (FWO) as a Junior Post-Doctoral Fellow under Grant 203056/1241722N LV and C3I Center-Cyber Security Center, IIT Kanpur, India.

REFERENCES

[1] P. Shor, “Algorithms for quantum computation: discrete logarithms and factoring,” in *Proceedings 35th Annual Symposium on Foundations of Computer Science*, 1994, pp. 124–134.

[2] J. Proos and C. Zalka, “Shor’s discrete logarithm quantum algorithm for elliptic curves,” *Quantum Info. Comput.*, vol. 3, no. 4, p. 317–344, jul 2003.

[3] I. T. L. Computer Security Division, “Post-quantum cryptography standardization - post-quantum cryptography: Csrc.” [Online]. Available: https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization

[4] P. He, T. Bao, J. Xie, and M. Amin, “Fpga implementation of compact hardware accelerators for ring-binary-lwe based post-quantum cryptography,” *ACM Trans. Reconfigurable Technol. Syst.*, oct 2022, just Accepted. [Online]. Available: https://doi.org/10.1145/3569457

[5] 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.

[6] J. Lablanche, L. Mortajine, O. Benchaalal, P.-L. Cayrel, and N. E. Mrabet, “Optimized implementation of the nist pqc submission rollo on microcontroller,” Cryptology ePrint Archive, Paper 2019/787, 2019, https://eprint.iacr.org/2019/787. [Online]. Available: https://eprint.iacr.org/2019/787

[7] “Crystals-kyber,” 2017. [Online]. Available: https://pq-crystals.org/kyber/

[8] “Crystals,” 2017. [Online]. Available: https://pq-crystals.org/dilithium/

[9] P.-A. Fouque, J. Hoffstein, and P. Kirchner, 2017. [Online]. Available: https://falcon-sign.info/

[10] J.-P. Aumasson, D. J. Bernstein, and C. Dobraunig, 2017. [Online]. Available: https://sphincs.org/

[11] I. T. L. Computer Security Division, “Round 2 submissions - post-quantum cryptography: Csrc.” [Online]. Available: https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-2-submissions

[12] J. M. Bermudo Mera, A. Karmakar, S. Kundu, and I. Verbauwhede, “Scabbard: a suite of efficient learning with rounding key-encapsulation mechanisms,” *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2021, no. 4, p. 474–509, Aug. 2021.

[13] J.-P. D’Anvers, A. Karmakar, S. S. Roy, and F. Vercauteren, *SABER: Mod-LWR based KEM*. National Institute of Standards and Technology, 2022. [Online]. Available: https://www.esat.kuleuven.be/cosic/publications/article-2953.pdf

[14] Z. Liang, B. Fang, J. Zheng, and Y. Zhao, “Compact and efficient kems over ntru lattices,” 2022. [Online]. Available: https://arxiv.org/abs/2205.05413

[15] C. Chen, O. Danba, J. Hoffstein, A. Hülsing, J. Rijneveld, J. M. Schanck, P. Schwabe, W. Whyte, and Z. Zhang, “NTRU algorithm specifications and supporting documentation,” Online, 2020. [Online]. Available: https://ntru.org/release/NIST-PQ-Submission-NTRU-20201016.tar.gz

[16] W.-K. Lee, H. Seo, S. O. Hwang, R. Achar, A. Karmakar, and J. M. B. Mera, “Dpcrypto: Acceleration of post-quantum cryptography using dot-product instructions on gpus,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2022.

[17] Y. Kim, J. Song, and S. C. Seo, “Accelerating falcon on armv8,” *IEEE Access*, vol. 10, pp. 44 446–44 460, 2022.

[18] Y. Zhao, R. Xie, G. Xin, and J. Han, “A high-performance domain-specific processor with matrix extension of risc-v for module-lwe applications,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2022.

[19] M. Schöffel, F. Lauer, C. C. Rheinländer, and N. Wehn, “On the energy costs of post-quantum kems in tls-based low-power secure iot,” in *Proceedings of the International Conference on Internet-of-Things Design and Implementation*, 2021, pp. 158–168.

[20] P. Sajimon, K. Jain, and P. Krishnan, “Analysis of post-quantum cryptography for internet of things,” in *2022 6th International Conference on Intelligent Computing and Control Systems (ICICCS)*. IEEE, 2022, pp. 387–394.

[21] J. Xie, K. Basu, K. Gaj, and U. Guin, “Special session: The recent advance in hardware implementation of post-quantum cryptography,” in *2020 IEEE 38th VLSI Test Symposium (VTS)*. IEEE, 2020, pp. 1–10.

[22] S. Markidis, S. W. D. Chien, E. Laure, I. B. Peng, and J. S. Vetter, “NVIDIA tensor core programmability, performance & precision,” *CoRR*, vol. abs/1803.04014, 2018. [Online]. Available: http://arxiv.org/abs/1803.04014

[23] W.-K. Lee, H. Seo, Z. Zhang, and S. O. Hwang, “Tensorcrypto: High throughput acceleration of lattice-based cryptography using tensor core on gpu,” *IEEE Access*, vol. 10, pp. 20 616–20 632, 2022.

[24] L. Wan, F. Zheng, G. Fan, R. Wei, L. Gao, Y. Wang, J. Lin, and J. Dong, “A novel high-performance implementation of crystals-kyber with ai

accelerator," in *European Symposium on Research in Computer Security*. Springer, 2022, pp. 514–534.

[25] E. Fujisaki and T. Okamoto, "Secure integration of asymmetric and symmetric encryption schemes," *J. Cryptol.*, vol. 26, no. 1, pp. 80–101, 2013. [Online]. Available: https://doi.org/10.1007/s00145-011-9114-1

[26] J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehle, "Crystals - kyber: A cca-secure module-lattice-based kem," in *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2018, pp. 353–367.

[27] N. Zhang, B. Yang, C. Chen, S. Yin, S. Wei, and L. Liu, "Highly efficient architecture of newhope-nist on fpga using low-complexity ntt/intt," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2020, no. 2, p. 49–72, Mar. 2020. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8544

[28] A. Jati, N. Gupta, A. Chattopadhyay, and S. K. Sanadhya, "Spqcop: Side-channel protected post-quantum cryptoprocessor," Cryptology ePrint Archive, Paper 2019/765, 2019, https://eprint.iacr.org/2019/765. [Online]. Available: https://eprint.iacr.org/2019/765

[29] J. Lablanche, L. Mortajine, O. Benchaalal, P.-L. Cayrel, and N. E. Mrabet, "Optimized implementation of the nist pqc submission rollo on microcontroller," Cryptology ePrint Archive, Paper 2019/787, 2019, https://eprint.iacr.org/2019/787. [Online]. Available: https://eprint.iacr.org/2019/787

[30] Y. Gao, J. Xu, and H. Wang, "cunh: Efficient gpu implementations of post-quantum kem newhope," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 3, pp. 551–568, 2022.

[31] 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*, pp. 1–1, 2021.

[32] S. C. Seo and S. An, "Parallel implementation of crystals-dilithium for effective signing and verification in autonomous driving environment," *ICT Express*, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2405959522001126

[33] Y. O. A. Karatsuba, "Sike on gpu: Accelerating supersingular isogeny-based key encapsulation mechanism on graphic processing units," *IEEE Access*, vol. 9, pp. 116 731–116 744, 2021.

[34] F. Farahmand, D. T. Nguyen, V. B. Dang, A. Ferozpuri, and K. Gaj, "Software/hardware codesign of the post quantum cryptography algorithm ntruencrypt using high-level synthesis and register-transfer level design methodologies," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2019, pp. 225–231.

[35] V. Kostalabros, J. Ribes-González, O. Farràs, M. Moretó, and C. Hernandez, "Hls-based hw/sw co-design of the post-quantum classic mceliece cryptosystem," in *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2021, pp. 52–59.

[36] S. Patranabis, N. Datta, D. Jap, J. Breier, S. Bhasin, and D. Mukhopadhyay, "Scadfa: Combined sca+ dfa attacks on block ciphers with practical validations," *IEEE Transactions on Computers*, vol. 68, no. 10, pp. 1498–1510, 2019.

[37] 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.

[38] A. Banerjee, C. Peikert, and A. Rosen, "Pseudorandom functions and lattices," in *Advances in Cryptology–EUROCRYPT 2012: 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings 31*. Springer, 2012, pp. 719–737.

[39] Y. Zhu, M. Zhu, B. Yang, W. Zhu, C. Deng, C. Chen, S. Wei, and L. Liu, "Lwrpro: An energy-efficient configurable crypto-processor for module-lwr," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 68, no. 3, pp. 1146–1159, 2021.

[40] H. Seo, P. Sanal, A. Jalali, and R. Azarderakhsh, "Optimized implementation of sike round 2 on 64-bit arm cortex-a processors," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, pp. 2659–2671, 2020.

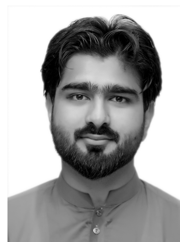
[41] S. An and S. C. Seo, "Efficient parallel implementations of lwe-based post-quantum cryptosystems on graphics processing units," *Mathematics*, vol. 8, no. 10, 2020. [Online]. Available: https://www.mdpi.com/2227-7390/8/10/1781

[42] K. Lee, M. Gowanlock, and B. Cambou, "Saber-gpu: A response-based cryptography algorithm for saber on the gpu," in *2021 IEEE 26th Pacific Rim International Symposium on Dependable Computing (PRDC)*, 2021, pp. 123–132.

[43] J.-C. See, H.-F. Ng, H.-K. Tan, J.-J. Chang, K.-M. Mok, W.-K. Lee, and C.-Y. Lin, "Cryptensor: A resource-shared co-processor to accelerate convolutional neural network and polynomial convolution," 2022.

[44] C. NVIDIA, "CUDA C programming guide, version 11.6," *NVIDIA Corp*, 2022.

[45] 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, 2021.



Muhammad Asfand Hafeez received a B.S. degree in electrical engineering from the University of Management and Technology in 2021. He is currently pursuing his master's degree in IT convergence engineering at Gachon University, South Korea. His research pursuits center on cryptography, GPU computing, deep learning, and hardware implementations.

Wai-Kong Lee received a B.Eng. in electronics and an M.Eng.Sc. from Multimedia University, Malaysia in 2006 and 2009, respectively. He received a Ph.D. in engineering from Universiti Tunku Abdul Rahman, Malaysia in 2018. He was a Visiting Scholar with Carleton University, Canada, in 2017, Feng Chia University, Taiwan, in 2016 and 2018, and OTH Regensburg, Germany, in 2015, 2018, and 2019. Prior to joining academia, he worked in several multi-national companies including Agilent Technologies (Malaysia) as an R&D engineer. His research interests are in the areas of cryptography, numerical algorithms, GPU computing, the Internet of Things, and energy harvesting. He is currently a post-doctoral researcher at Gachon University, South Korea.

Angshuman Karmakar received the B.E. degree in computer science and engineering from Jadavpur University, Kolkata, India, the M.Tech. degree in computer science and engineering from the Indian Institute of Technology, Kharagpur, India, and the Ph.D. degree from Katholieke Universiteit Leuven (KU 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 KEM scheme which is one of the finalists in the NIST's post-quantum standardization procedure. He is currently working as an assistant professor at the Indian Institute of Technology, Kanpur, in India. Earlier he was an FWO Post-Doctoral Fellow with the COSIC Research Group, KU Leuven. His research interest spans different aspects of lattice-based post-quantum cryptography and computation on encrypted data.

Seoung Oun Hwang received a 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 a Ph.D. degree in computer science from the Korea Advanced Institute of Science and Technology, South Korea. He worked as a Software Engineer with LG-CNS Systems, Inc., from 1994 to 1996. He also worked as a Senior Researcher with the Electronics and Telecommunications Research Institute (ETRI), from 1998 to 2007. He worked as a Professor at the Department of Software and Communications Engineering, Hongik University, from 2008 to 2019. He is currently a Professor at the Department of Computer Engineering, at Gachon University. He is also an Editor of the ETRI Journal. His research interests include cryptography, cybersecurity, and artificial intelligence.