TMVP-based Polynomial Convolution for Saber and Sable on GPU using CUDA-cores and Tensor-cores

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

Abstract*—Recently proposed lattice-based cryptography algorithms can be used to protect the IoT communication against the threat from quantum computers, but they are computationally heavy. In particular, polynomial multiplication is one of the most time-consuming operations in lattice-based cryptography. To achieve efficient implementation, the Number Theoretic Transform (NTT) algorithm is an ideal choice, but it has certain limitations on the parameters, which not all lattice-based schemes can employ directly. Hence, alternative techniques are proposed to accelerate polynomial multiplication on lattice-based schemes that cannot utilize the NTT directly. In this paper, we propose a parallel Toeplitz matrix-vector product (TMVP) version to accelerate the polynomial multiplication in PQC algorithms implemented it on a graphics processing unit (GPU). This is the first time a TMVP parallel version has been proposed and experimented on different GPU cores (i.e., CUDA-cores and Tensor-cores). The effectiveness of the proposed solution is validated on Saber (the NIST post-quantum standardization finalist) and Sable (an improved version of Saber) schemes. Experimental results show that TMVP-based polynomial convolution using CUDA-cores fails to exhibit a significant enhancement compared to the schoolbook CUDA-core method already proposed by Hafeez et al. 2023. However, when the TMVP technique is applied to Tensor-cores, it outperformed state-of-the-art implementations. The proposed Tensor-core approach outperformed the schoolbook Tensor-core method by up to 1.21×, and outperformed the dot-product-instructions method (Lee et al. 2022) by up to 3.63×. The proposed TMVP Tensor-cores is also faster than the TMVP CUDA-cores method by 13.76×.

***Index Terms*—Toeplitz Matrix-vector Product (TMVP), Cryptography, Tensor-cores, CUDA-cores, Post-quantum Cryptography, Lattice-based Cryptography, Matrix Multiplication.**

I. Introduction

SECURE communication is essential for protecting sensitive information and preserving privacy. Cryptography algorithms are the backbone of secure communication systems, ensuring data confidentiality, integrity, and authenticity. However, the emergence of quantum computers (QCs) poses a significant threat to the security provided by the classical cryptography schemes relying on the hardness of integer factorization and discrete logarithms. In response to this threat, the National Institute of Standards and Technology (NIST) [1] started a Post-Quantum Cryptography (PQC) standardization process in 2016. The goal was to identify cryptography algorithms that could resist attacks from classical and quantum computers in the long term. After a comprehensive evaluation process, lattice-based algorithms emerged as the most resilient option for PQC. The standardization process concluded in 2022 with four candidates: one key encapsulation mechanism (KEM), CRYSTALS-KYBER [2] and three signature schemes CRYSTALS-Dilithium [3], FALCON [4], and SPHINCS$^+$ [5].

Although using the Kyber algorithm as the primary standard for Post-Quantum Cryptography (PQC) is a significant step forward, it provides a framework for future advancements and improvements in PQC schemes while also reinforcing the importance of security. During the standardization process, non-traditional parameter choices, such as the LAC and Round 5 [6], were discouraged to mitigate the potential vulnerabilities that attackers could exploit. This cautious approach ensures that the security of PQC systems remains robust.

The use of non-constant-time error correction codes in lattice-based PQC schemes has raised concerns. Error correction codes play a crucial role in ensuring the accuracy and dependability of PQC schemes. However, if these codes are not implemented in a constant-time manner, they can become potential sources of side-channel attacks. These attacks can compromise the security of the system by exploiting information leaked through timing or power consumption. Therefore, it is essential to evaluate the use of error correction codes in PQC schemes with care. Researchers and developers must continually strive to enhance existing PQC schemes while maintaining their security. This ongoing effort includes exploring alternative parameter choices, optimizing error correction codes, and addressing potential side-channel vulnerabilities, etc. For example, Scabbard (a suite of KEM schemes proposed by Mera et al. [7]) improves on Saber [8], the NIST PQC finalist. SMAUG which is a candidate scheme submitted to the ongoing Korean PQC standardization [9] has been heavily influenced by the design elements of Scabbard. Similarly, Liang et al. [10] proposed an enhanced version of the NTRU KEM [11], which was also a finalist in the NIST standardization. Cho et al. [12] improved the key size and bit-security of the first-round pqsigRM signature scheme.

However, most of the lattice-based PQC schemes involve polynomial multiplication over polynomials with a high degree, making them computationally expensive. To achieve a better performance, some schemes like Kyber are designed to have a special ring structure that can utilize the Number Theoretic Transform (NTT) [13] for computing polynomial

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.)

multiplication. On the other hand, some other lattice-based schemes (e.g., Saber) do not have a ring structure that is NTT-friendly. As such, careful consideration should be given while implementing lattice-based schemes to ensure optimal performance. Substantial efforts have been directed toward enhancing the performance of polynomial multiplication for non-NTT-friendly schemes. For instance, classical techniques like Toom-Cook [14] and Karatsuba [15] are commonly used to achieve this. Recently the Toeplitz matrix-vector product (TMVP) emerged as an alternative method, and its effectiveness was demonstrated in recent works [16], [17]. These studies showed that the TMVP yields promising results in terms of performance and efficiency when compared to the Toom-Cook and Karatsuba methods. However, prior work was only focused on serial versions of the TMVP; it is still unclear if such an approach can perform equally well on a parallel architecture like the graphics processing unit (GPU). This motivated us to investigate the effectiveness of a parallel TMVP to speed up polynomial multiplication further.

Paksoy and Cenk [16], [17] proposed several techniques that target ARM Cortex-M4 microcontrollers to efficiently utilize the TMVP technique for Saber and NTRU. However, it is unclear if the TMVP techniques can be applied to a parallel architecture like the GPU or how to optimize performance on such architectures. Similarly, efforts have been made to improve the performance of lattice-based schemes on several alternative platforms such as the latest Intel AVX [18] instructions, hardware accelerators in a Field Programmable Gate Array (FPGA) [19], [20], reduced instruction set computer (RISC) [21] or an application-specific integrated circuit (ASIC) [22] platform. Besides that, massively parallel architectures like the GPU have attracted attention from the research community. For instance, Gupta et al. [23] presented early research on the feasibility of parallelizing PQC on the GPU, while Lee et al. [24], [25] demonstrated the effectiveness of using advanced GPU features like Tensor-cores and the dot-product to speed up polynomial multiplication.

In this paper, our primary aim is to investigate the feasibility of parallelizing TMVP to analyze its performance on the GPU platform. We also explore the possibility of utilizing Tensor-cores in conjunction with the TMVP to further improve the performance of polynomial multiplication.

1) For the first time, TMVP-based polynomial convolution on Tensor-cores in a GPU is presented. Parallel implementation of TMVP on a GPU presents certain challenges, including memory access patterns, shared memory limitations, and the choice of parallelization methods in order to optimally leverage the capability of GPU architecture. To meet these challenges, we prearrange the matrix following the reduction pattern of the selected schemes (Saber and Sable), and then apply the TMVP to break the matrix in a manner that maximizes parallelism. The experimental results on a RTX 3060Ti GPU demonstrate that our proposed TMVP-based polynomial convolution using Tensor-cores yields throughput that is 1.21× and 3.63× higher than the [26] and [25], respectively.
2) In addition to Tensor-cores, the proposed TMVP-based polynomial convolution was also implemented on CUDA-cores. The findings reveal that the TMVP using Tensor-cores outperformed its CUDA-cores counterpart by 6.2× in terms of throughput. This is because there is insufficient shared memory to hold multiple copies of vectors in the CUDA-cores TMVP implementation. In addition, many read/write operations are required in the CUDA-cores TMVP implementation, limiting its performance. This shows that the TMVP technique may not always yield good performance in a parallel architecture due to limitations in memory. In contrast, the Tensor-cores version does not use any shared memory because matrix multiplication is performed directly on the registers, thus eliminating most of the memory issues found in the CUDA-cores version.
3) The Saber [8] and Sable [7] KEMs were evaluated using the proposed techniques. Our Tensor-cores implementation achieved 424,437 encryptions per second and 6,259,781 decryptions per second implementing the Saber key exchange (KEX) on an RTX 3060Ti GPU, which is 2.58× and 6.83× faster, respectively, than using standard CUDA-cores. The highest throughput achieved by Saber KEM was 267,720 encapsulations per second and 294,020 decapsulations per second. For the Sable KEX, the throughput achieved by the TMVP-based Tensor-cores implementation was 457,155 encryptions per second and 5,621,925 decryptions per second, which is 2.67× and 6.22× faster, respectively, than on standard CUDA-cores. The highest throughput of the Sable KEM was 250,062 encapsulations per second and 295,061 decapsulations per second. The Tensor-core based TMVP implementation for Sable demonstrated satisfactory performance, wherein the encapsulation and decapsulation throughput were 4.7% and 4.97% faster than [26].
4) The source code for the proposed TMVP polynomial convolution is is publicly available https://github.com/Muhammad-Asfand/asfand-tmvp. We sincerely hope that this will enable researchers to easily replicate our findings. Also, we believe that it can encourage further studies and research on TMVP-based polynomial convolution on GPUs and other parallel accelerators.

This paper is organized as follows. Section II discusses background information for the proposed study and reviews related work in the literature. In Section III, we discuss in detail an implementation of the TMVP using CUDA-cores and Tensor-cores. In Section IV, we discuss our experiment results. Finally, Section V concludes the paper.

II. Preliminaries

In this section, an overview of TMVP and its variants are presented, followed by its applications to reduce the complexity of polynomial convolution for PQC. Two target PQC schemes that can utilize TMVP for improved performance are presented next. The first scheme (Saber) is one of the finalists in the NIST PQC standardization process, and the second scheme (Sable) is the improved version of Saber.

A. The Toeplitz matrix-vector product technique

The TMVP is a technique used in various cryptographic applications to perform multiplication. It was first introduced by Fan and Hasan [27] for multiplying binary extension fields. Since then, many proposals have been suggested by Hasan et al. [28], [29]. Similarly, in [30] and [31], the TMVP was used for speeding up the residue multiplication modulo in integer modular multiplication. It can also be used to calculate the product of two polynomials modulo a polynomial [32]. The following matrix T is an example of a 5×5 Toeplitz matrix where the elements along a line parallel to the principal diagonal possess a constant value.

$$T = \begin{pmatrix} t_0 & t'_1 & t'_2 & t'_3 & t'_4 \\ t_1 & t_0 & t'_1 & t'_2 & t'_3 \\ t_2 & t_1 & t_0 & t'_1 & t'_2 \\ t_3 & t_2 & t_1 & t_0 & t'_1 \\ t_4 & t_3 & t_2 & t_1 & t_0 \end{pmatrix} \quad (1)$$

To determine an $n \times n$ Toeplitz matrix, only $2n - 1$ elements are needed. This means that calculating the sum of two Toeplitz matrices can be done with just $2n - 1$ entry additions, resulting in another Toeplitz matrix. Additionally, all submatrices of a Toeplitz matrix are also Toeplitz matrices. These characteristics make it possible to efficiently compute Toeplitz matrix-vector multiplication using TMVP formulas rather than the conventional schoolbook method.

1) TMVP Formulas: Various split formulas are available to efficiently compute TMVPs (such as two-way, three-way, and four-way, given in [30], [16], [17]. We use X to denote an $n \times n$ Toeplitz matrix and Y to denote a vector of length n.

Two-way TMVP (TMVP-2): We can define $n \times n$ Toeplitz matrix T using three matrix vectors, (X_0, X_1, X_2) and an $n \times 1$ column vector $Y = (Y_0, Y_1)$. Toeplitz matrix T consists of three $(n/2) \times (n/2)$ Toeplitz matrices, namely P_0, P_1, and P_2. Equation 2 is the TMVP-2 using three $(n/2) \times (n/2)$ TMVPs [33].

$$T = X.Y = \begin{pmatrix} X_1 & X_0 \\ X_2 & X_1 \end{pmatrix} \begin{pmatrix} Y_0 \\ Y_1 \end{pmatrix} = \begin{pmatrix} P_0 + P_1 \\ P_0 - P_2 \end{pmatrix}, \quad (2)$$

where P_0, P_1 and P_2 represents three TMVPs:

$P_0 = X_1(Y_0 + Y_1)$,
$P_1 = (X_0 - X_1)Y_1$,
$P_2 = (X_1 - X_2)Y_0$.

Three-way TMVP (TMVP-3): Like TMVP-2, TMVP-3 allows us to calculate an n dimensional TMVP using six $n/3$-dimensional TMVPs. Consider the $n \times 1$ column vector $Y = (Y_0, Y_1, Y_2)$ and matrix-vector $X = (X_0, X_1, X_2, X_3, X_4)$, which is an $n \times n$ Toeplitz matrix. Here, Y_i (where $i = 0, 1, 2$) is an $(n/3) \times 1$ column vector, and X_i (where $i = 0, 1, 2, 3, 4$) is an $(n/3) \times (n/3)$ Toeplitz matrix [33]. By rewriting product $P = XY$, we get equation 3:

$$X.Y = \begin{pmatrix} X_2 & X_1 & X_0 \\ X_3 & X_2 & X_1 \\ X_4 & X_3 & X_2 \end{pmatrix} \begin{pmatrix} Y_0 \\ Y_1 \\ Y_2 \end{pmatrix} = \begin{pmatrix} P_0 + P_3 + P_4 \\ P_1 - P_3 + P_5 \\ P_2 - P_4 + P_5 \end{pmatrix}, \quad (3)$$

where P_0, P_1, P_2, P_3, P_4 and P_5 represents six TMVPs:

$P_0 = (X_0 + X_1 + X_2)Y_2$,
$P_1 = (X_1 + X_2 + X_3)Y_1$,
$P_2 = (X_2 + X_3 + X_4)Y_0$,
$P_3 = X_1(Y_1 - Y_2)$,
$P_4 = X_2(Y_0 - Y_2)$,
$P_5 = X_3(Y_0 - Y_1)$.

Four-way TMVP (TMVP-4): To compute an n-dimensional TMVP, a TMVP-4 formula was proposed in [17]. This utilizes a combination of seven $n/4$-dimensional TMVPs. Assuming that n is divisible by four, we take the $n \times 1$ column vector $Y = (Y_0, Y_1, Y_2, Y_3)$ and a matrix-vector $X = (X_0, X_1, X_2, X_3, X_4, X_5, X_6)$, which represents an $n \times n$ Toeplitz matrix. In this case, Y_i (where $i = 0, 1, 2, 4$) is an $n/4 \times 1$ column vector, and X_i (where $i = 0, 1, 2, 3, 4, 5, 6$) is an $n/4 \times n/4$ Toeplitz matrix. We divide the Toeplitz matrix and the vector, then compute the product as in equation 4:

$$X.Y = \begin{pmatrix} X_3 & X_2 & X_1 & X_0 \\ X_4 & X_3 & X_2 & X_1 \\ X_5 & X_4 & X_3 & X_2 \\ X_6 & X_5 & X_4 & X_3 \end{pmatrix} \begin{pmatrix} Y_0 \\ Y_1 \\ Y_2 \\ Y_3 \end{pmatrix} = \begin{pmatrix} P_1 - P_2 + 8P_3 - 8P_4 + 27P_5 + P_6 \\ P_1 + P_2 + 4P_3 + 4P_4 + 9P_5 \\ P_1 - P_2 + 2P_3 - 2P_4 + 3P_5 \\ P_0 + P_1 + P_2 + P_3 + P_4 + P_5 \end{pmatrix}, \quad (4)$$

where P_0, P_1, P_2, P_3, P_4, P_5 and P_6 represents six TMVPs:

$P_0 = \frac{1}{12}(12X_6 + 4X_5 - 15X_4 + 5X_3 + 3X_2 - X_1)Y_0$,
$P_1 = \frac{1}{12}(12X_5 + 8X_4 - 7X_3 - 2X_2 + X_1)(Y_0 + Y_1 + Y_2 + Y_3)$,
$P_2 = \frac{1}{24}(-12X_5 + 16X_4 - X_3 - 4X_2 + X_1)(Y_0 - Y_1 + Y_2 - Y_3)$,
$P_3 = \frac{1}{24}(-6X_5 - X_4 + 7X_3 + X_2 - X_1)(Y_0 - 2Y_1 + 4Y_2 - 8Y_3)$,
$P_4 = \frac{1}{120}(6X_5 - 5X_4 - 5X_3 + 5X_2 - X_1)(Y_0 - 2Y_1 + 4Y_2 - 8Y_3)$,
$P_5 = \frac{1}{120}(4X_5 - 5X_3 + X_1)(Y_0 + 3Y_1 + 9Y_2 + 27Y_3)$,
$P_6 = (-12X_5 + 4X_4 + 15X_3 - 5X_2 - 3X_1 + A_0)Y_3$,

Table I presents the arithmetic complexity of the TMVP-2, TMVP-3, and TMVP-4 formulas. The expressions in the table exhibit a recursive nature and reflect the number of operations required to compute the TMVP for a given size *n*. These expressions are defined in terms of the operations involved in smaller sizes. It is worth noting that, despite TMVP-4 breaking down the *n* into a smaller matrix compared to the others, TMVP-2 has the smallest recursive term coefficient (i.e., 3) among them, hence exhibiting the lowest arithmetic complexity of the three methods.

TABLE I
ARITHMETIC COMPLEXITY OF TMVP FORMULAS

TMVP's	Arithmetic Complexity
TMVP-2	$M_{TMVP-2}(n) = 3M(n/2) + 3n - 1$
TMVP-3	$M_{TMVP-3}(n) = 6M(n/3) + 5n - 1$
TMVP-4	$M_{TMVP-4}(n) = 7M(n/4) + 11n - 1$

2) TMVP vs Toom-Cook: TMVP and Toom-Cook-based multiplications are specialized techniques for optimizing polynomial multiplications and exhibit notable similarities. Nevertheless, the selection of an appropriate method depends heavily on the distinct computational context and hardware prerequisites. When considering the utilization of GPUs, it is recommended to opt for TMVP due to the following reasons.

Parallelism: TMVP allows for a high level of parallelism, which is a key optimization strategy on GPUs. Different parts of the vector can be multiplied in parallel with various sliding windows of the matrix, increasing function throughput.

Memory Bandwidth: TMVP can lead to reduced memory bandwidth requirements compared to Toom-Cook-based polynomial multiplications. Toom-Cook algorithms involve more complex operations and require more memory transfers, which cause a bottleneck on GPUs, especially for large polynomials.

Data Locality: Toeplitz matrices exhibit a discernible pattern where each diagonal that descends from left to right maintains a constant value. This structure efficiently manages memory coherence when storing and manipulating matrices, particularly in GPUs that support coalesced memory access.

B. Saber and Sable

Saber is a lattice-based KEM that relies on module lattices. Saber stands out for its unique feature of polynomial convolution without the use of NTT, which can be daunting for lattice-based cryptography. This approach of Saber has inspired other cryptography schemes like [7], [34] to adopt similar methods. It was named a finalist in the third round of the NIST PQC standardization competition, indicating its potential as a leading solution in cryptographic security. The strength of Saber's security relies on the conjectural hardness of the Module Learning with Rounding (MLWR) problem [35]. The security level of the target schemes can be configured by specifying dimension ℓ of the module with three distinct values: ℓ = 2 (LightSaber), ℓ = 3 (Saber), and ℓ = 4 (FireSaber), which correspond to security levels 1, level 3, and level 5, respectively. Note that in this paper, we focus on our implementation of Saber for ℓ=3, extending it to support different ℓ levels is straightforward. Saber's arithmetic operations are $R_q = R_{2^{13}} = \mathbb{Z}_{2^{13}}[x]/\langle x^{256}+1\rangle$ and $R_p = R_{2^{10}} = \mathbb{Z}_{2^{10}}[x]/\langle x^{256}+1\rangle$. As with many lattice-based cryptosystems defined on polynomial rings, the efficiency of this scheme is heavily impacted by multiplication in these rings. However, it is important to note that the rings R_q and R_p utilized by Saber are not directly compatible with the NTT, which is currently the most efficient polynomial multiplication algorithm known.

Mera et al. [7] introduced the Sable scheme in Scabbard as an improved version of Saber based on a hard lattice problem known as learning with rounding (LWR). In such schemes, errors are implicitly created through rounding instead of explicit addition, as seen in LWE. Since errors are crucial in determining the security of lattice-based schemes, proper estimation is essential to avoid overestimation or underestimation. By accurately estimating errors, Mera et al. [7] were able to enhance Saber's parameters without compromising its security. This resulted in reduced key sizes and bandwidth, and this improved version of Saber is known as Sable. The security level of Sable can be configured in the same way as Saber. For instance, ℓ = 2 (LightSable), ℓ = 3 (Sable), and ℓ = 4 (FireSable), correspond to security levels 1, 3, and 5, respectively. The Saber and Sable KEMs consist of three algorithms: key generation (Algorithm 1), encapsulation (Algorithm 2), and decapsulation (Algorithm 3). The values of different parameters used in the designing of both KEMs are given in Table II.

Algorithm 1 KEM Key Genreation

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: $(K', r') \leftarrow \mathcal{G}(m||H$(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

Algorithm 1 depicts the generation of a public key (PK) and a private key (SK) using security parameter N. Algorithm 2 takes the PK as input and produces ciphertext (CT) and a shared secret key (K). Algorithm 3 performs decapsulation, taking the PK, CT, and SK as input and returning the shared secret key as output. In Algorithms 1 to 3, H and $\mathcal{G}$ represent hash functions. The constant polynomials $h1, h2$, and $h3$ have 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.

C. Related work

In recent work, Lee et al. [25] introduced a novel approach to conduct polynomial convolution using dot-product instruc-

TABLE II
PARAMETERS OF SABER AND SABLE

Parameters	ℓ	N	p	q	Moduli	Key Sizes
Saber	3	256	2048	8192	ϵ_q:13 ϵ_p:10 ϵ_t:4	PK: 992 SK: 1440 CT: 1088
Sable	3	256	512	2048	ϵ_q : 11 ϵ_p:9 ϵ_t:4	PK: 1280 SK: 1728 CT: 1304

Algorithm 3 KEM Decapsulation

Data: PK = ($seed_A$, b), SK = (s,H(PK),r), CT = (c',b$'$)
Result key = K

```
u = b'.(s mod p) ∈ R_p^N
m'_1 = bits ((u + h_2 − 2^{ε_p−ε_t−B} m), ε_p, B) ∈ R_{2^B}^N    ▷ Decode
m_1 = original_msg(m'_1)
m_2 = arrange_msg(m_1)
(K'_1, r'_1) ← G(m_2||H(pk))
A ← gen_N^{L×L}(XOF(seed_A)) ∈ (R_q^N)^{L×L}
s'_1 ← β_η((R_q^N)^L)
b'_1 = bits (A^T.s'_1 + h_1, ε_q, ε_p)
// Rounding
u'_1 = b^T.(s'_1 mod p) ∈ R_p^N
c'_1 = bits((u'_1 + h_3 − 2^{ε_p−B} m), ε_p, (ε_t + B)) ∈ R_{2^B t}^N    ▷ HelpDecode
if c' = c'_1 then
    return K = H(K'_1, H(c'))
else
    return K = H(r, H(c'))
end if
```

tions. This method enables execution of *MULTIPLY*-and-*ADD* instructions in a single clock cycle, resulting in significantly improved throughput when compared to traditional 32-bit integer units. In other work, Lee et al. [24] utilized Tensor-cores in a GPU to compute polynomial convolution, which showed greater efficiency and speed compared to CUDA-cores. Following this, Hafeez et al. [26] introduced two techniques to address gaps in previous research. First, they extended the work of See et al. [36] on a GPU and proposed a polynomial restructuring technique that enables multiple polynomials with different public keys to be processed in a single communication cycle. Secondly, they introduce a new method to handle the reduction patterns that are not suitable for parallel implementation. Furthermore, Gao et al. [37], and Lee and Hwang [38] explored the use of the NTT on a GPU for implementing NewHope and Kyber, respectively. These studies revealed the potential for GPUs to effectively handle polynomial multiplication, which is crucial in many lattice-based cryptography schemes.

Other researchers have investigated GPU-based implementations of various cryptography schemes. For instance, Sun et al. [39] demonstrated an efficient parallel implementation of SPHINCS on a GPU, while Dai et al. [40] optimized the NTRU modular lattice signature scheme for parallel polynomial multiplication on a GPU. Their optimization is particularly important due to the scheme's reliance on large vectors, which can be efficiently processed in parallel on a GPU. Finally, Gupta et al. [23] analyzed the batch mode and single mode parallelism available in a GPU and evaluated implementation in different PQC schemes. The findings of these studies shed light on the potential from utilizing a GPU to provide efficient and scalable solutions for various cryptographic applications.

III. Proposed Parallel TMVP Technique

In this section, we describe how to parallelize the TMVP-2 formula and its implementation for Saber and Sable, using Tensor-cores and CUDA-cores.

A. Polynomial convolution using TMVP-2

Saber and Sable schemes both employ an efficient reduction pattern that resembles a nega-cyclic convolution. To facilitate matrix-vector multiplication, polynomial *A* is first transformed into the nega-cyclic matrix in equation 5 with dimensions of 256×256. Polynomial *B* is structured into the column-major matrix in equation 6.

$$A = \begin{pmatrix} a_0 & -a_{n-1} & -a_{n-2} & \dots & -a_3 & -a_2 & -a_1 \\ a_1 & a_0 & -a_{n-1} & \dots & -a_4 & -a_3 & -a_2 \\ a_2 & a_1 & a_0 & \dots & -a_5 & -a_4 & -a_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ a_{n-3} & a_{n-4} & a_{n-4} & \dots & a_0 & -a_{n-1} & -a_{n-2} \\ a_{n-2} & a_{n-3} & a_{n-4} & \dots & a_1 & a_0 & -a_{n-1} \\ a_{n-1} & a_{n-2} & a_{n-3} & \dots & a_2 & a_1 & a_0 \end{pmatrix} \quad (5)$$

$$B = \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_{n-3} \\ b_{n-2} \\ b_{n-1} \end{pmatrix} \quad (6)$$

Figures 1, 2, and 3 show polynomial convolution using TMVP-2, TMVP-3, and TMVP-4 respectively. We opted for TMVP-2 for polynomial convolution in Saber and Sable for the following reasons.

1) TMVP-2 break the 256×256 matrix into three non-identical 128×128 matrices as shown in Figure 1. Similarly, TMVP-3 and TMVP-4, respectively, produce five and seven non-identical matrices at 86×86 and 64×64 as depicted in Figure 2 and 3. The matrix size of TMVP-2 is bigger than the other two, so it provides more parallelism than TMVP-3 and TMVP-4.
2) Additionally, TMVP-2 performs only the three multiplication in equation 2 to compute the polynomial convolution (see Figure 1), while TMVP-3 and TMVP-4 required six (equation 3) and seven (equation 4) multiplications, respectively.
3) TMVP-3 is unsuitable for use in Saber and Sable because the polynomial convolution in these schemes has a length of 256, which is not divisible by 3. Hence, we have to create a 258×258 matrix (divisible by 3) and pad the unused rows and columns with zeroes. After padding, we can perform polynomial convolution using TMVP-3, but there will be some unused rows and columns that waste computational bandwidth.

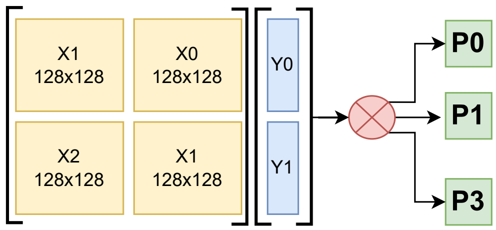


Fig. 1. Polynomial convolution using TMVP-2

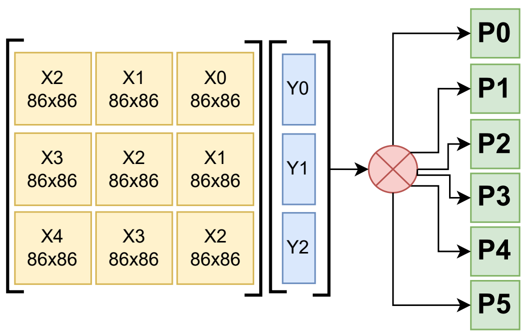


Fig. 2. Polynomial convolution using TMVP-3

B. TMVP-2 Implementation using CUDA-cores

For most of the lattice-based cryptography schemes, polynomial convolution is the most time-consuming task. This particular task entails the manipulation of two distinct polynomials: polynomial a, which typically represents a public or a private key and polynomial b consisting of random elements with small coefficients. In the Sable cryptography algorithms, polynomial b is ternary, i.e., composed of elements $b = \{-1, 0, 1\}$.

However, polynomial convolution in Saber and Sable is essentially the same, so we present the proposed TMVP implementation for both schemes in Algorithm 4. Note that this algorithm describes the basic implementation of the TMVP for polynomial convolution using CUDA-cores commonly found in a GPU. In the next subsection, we present the more advanced technique proposed in this work, which utilizes the Tensor-cores. Referring to Algorithm 4, line 1 rearranges polynomial A into a nega-cyclic pattern. Following this, line 2 pre-processes polynomials A and B for the three TMVP multiplications, as given in equation 2. Next, line 3 computes the matrix-vector product using CUDA-cores, as given in Algorithm 7. Finally, line 7 post-processes the products and calculates the final result.

Algorithm 5 is used to convert the polynomial *A* into a nega-cyclic pattern. The input is read by *N* threads and *N* blocks. Line 3 yields the difference between threads and blocks to arrange the elements into a nega-cyclic pattern. In line 5, if ($tid - bid$) is greater than the *(N-1)*, the arranged elements in the rows are converted to negative form. Otherwise, the elements are arranged without conversion.

In reference to Algorithm 6, it pre-processes the polynomial *A* and *B* into the required matrices and vectors to perform three TMVP multiplications in CUDA-cores. *N/2* threads and *N/2* blocks are launched in parallel. Lines 4 and 5 rearrange the elements for the first multiplication and move the elements into a_1 and b_1 in U16 format. Similarly, lines 6 and 7 rearrange the elements for the second multiplication, and then lines 8 and 9 rearrange the elements for the third and store the output in a_2, b_2, and in a_3, b_3 in U16 format, respectively.

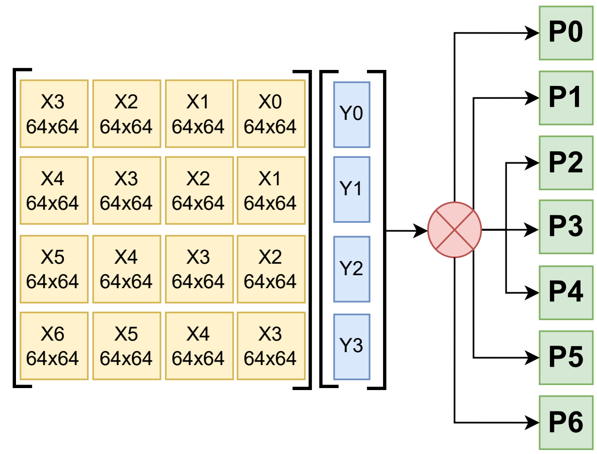


Fig. 3. Polynomial convolution using TMVP-4

Algorithm 4 CUDA-cores implementation of polynomial convolution in parallel on a GPU

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

1: ParNegCyc$< N, N >(fp16_A, A)$ ▷ Alg.5
2: PreArr$< N/2, N/2 >(fp16_B, B)$ ▷ Alg.6
3: CUDACores$< N, N >(fp16_A, fp16_B, fp32_C)$ ▷ Alg.7
4: PostProcess$< N/2, N/2 >(c, fp32_C)$ ▷ Alg.8

After pre-processing, Algorithm 7 describes the proposed method to execute the three TMVP multiplications. This is a crucial step in achieving accurate and efficient results in matrix-vector products. To compute the matrix-vector product, *N* threads are launched in parallel, ensuring more parallelism is exploited. It is worth noting that the three input matrices used in this process are denoted a_1, a_2, and a_3, while the vectors are denoted b_1, b_2, and b_3. In lines 3-6 of Algorithm 7, elements of a_1 and b_1 are loaded into the shared memory to compute the product. Loading elements into shared memory is crucial for efficient implementation. Line 8 initializes the register to accumulate the product, while lines 9-11 compute the matrix-vector product. Finally, at line 12, the result is moved from the register to p_1, indicating that one TMVP multiplication is completed. The second and third multiplications are done in lines 14–23 and 25–34, respectively, following a similar approach.

It is worth noting that we can compute the matrix-vector product by launching different numbers of threads (e.g., 128, 256, 512, and 1024) to increase the parallelism. However, there is very little increase in throughput because we cannot make multiple copies of vectors due to the limited amount of shared memory. The available shared memory in the RTX

Algorithm 5 ParNegCyc: arrange polynomial A into a 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.

$tid = thread$ ID
$bid = block$ ID
$idx = tid - bid$
// Launch N blocks and N threads in
// parallel
if $tid < N$ **then**
  if $idx > (N-1)$ **then**
    $out[bid + tid \times N] = in[(idx)\%N] \times (-1)$
  else
    $out[bid + tid \times N] = in[(idx)\%N]$
  end if
else
  $out[bid + tid \times N] = 0$
end if

Algorithm 6 PreArr: Pre-arrangements of elements for matrix-vector product.

Input: $N \times N$-length polynomial in_1 and N-length polynomial in_2
Output: Matrix a_1, a_2, a_3 and vector b_1, b_2, b_3 in U16 format

$tid = thread$ ID
$bid = block$ ID
// Launch $N/2$ blocks and $N/2$ threads in
// parallel
if $tid < N$ **then**
  $a_1[bid \times N/2 + tid] = in_1[bid \times N/2 + tid]$
  $b_1[tid] = in_2[tid] + in_2[N/2 + tid]$
  $a_2[bid \times N/2 + tid] = in_1[bid \times N/2 + (N \times N/4) + tid] - in_1[bid \times N/2 + tid]$
  $b_2[tid] = in_2[N/2 + tid]$
  $a_3[bid \times N/2 + tid] = in_1[bid \times N/2 + tid] + in_1[bid \times N/2 + (N \times N/2) + tid]$
  $b_3[tid] = in_2[tid]$
else
  $a_1, a_2, a_3[bid \times N/2 + tid] = 0$
  $b_1, b_2, b_3[tid] = 0$
end if

Algorithm 7 Polynomial multiplication of the *n/2*-dimensional TMVP using *256* threads

Input: $N/2 \times N/2-$length Matrix a_1, a_2, a_3 and $N-$length vector b_1, b_2 and b_3
Output: $N-$length vectors, $p1, p2$, and $p3$

1: $tid = thread$ ID
2: $bid = block$ ID
// Copy elements into shared memory for 1^{st} TMVP *p1* in parallel
3: **for** k from 0 to N/4 **do**
4:   $a\_shared[tid + k \times (N)] = a_1[tid + k \times (N)]$
5: **end for**
6: $b\_shared[tidx] = b_1[tidx]$
7: __syncthreads() ▷ Synchronize all the threads
// Accumulate each column in parallel with N threads
8: $sum1 = 0$ ▷ Use register to accumulate
9: **for** i from 0 to N/4 **do**
10:   $sum1 \mathrel{+}= a\_shared[tid \times (N/4) + i] \times b\_shared[(tid\%2) \times (N/4) + i]$
11: **end for**
12: $p_1[bid + tid] = sum1$
13: __syncthreads() ▷ Synchronize all the threads
// Copy elements into shared memory for 2^{nd} TMVP *p2* in parallel
14: **for** k from 0 to N/4 **do**
15:   $a\_shared[tid + k \times (N)] = a_2[tid + k \times (N)]$
16: **end for**
17: $b\_shared[tidx] = b_2[tidx]$
18: __syncthreads()
19: $sum2 = 0$
20: **for** i from 0 to N/4 **do**
21:   $sum2 \mathrel{+}= a\_shared[tid \times (N/4) + i] \times b\_shared[(tid\%2) \times (N/4) + i]$
22: **end for**
23: $p_2[bid + tid] = sum2$
24: __syncthreads() ▷ Synchronize all the threads
// Copy elements into shared memory for 3^{rd} TMVP *p3* in parallel
25: **for** k from 0 to N/4 **do**
26:   $a\_shared[tid + k \times (N)] = a_3[tid + k \times (N)]$
27: **end for**
28: $b\_shared[tidx] = b_3[tidx]$
29: __syncthreads()
30: $sum3 = 0$
31: **for** i from 0 to N/4 **do**
32:   $sum3 \mathrel{+}= a\_shared[tid \times (N/4) + i] \times b\_shared[(tid\%2) \times (N/4) + i]$
33: **end for**
34: $p_3[bid + tid] = sum3$

3060Ti GPU is 48KB, but each element in the matrix and vector is represented using 16 bits (two bytes). The number of elements for both matrix and vector is 128×128=16384. The total number of elements combined in both matrix and vector is 16,384×2=32,768. So, the total memory required by both matrix and vector is 32,768×2=65,536 bytes (64KB), which exceeds the available shared memory of 48KB (49,152 bytes) on the RTX 3060Ti GPU. Therefore, lines 10, 21, and 32 perform the modulus operation to find the exact element on the vector side. The modulus value for 128, 256, 512, and 1024 are 1,2,4 and 8, respectively. Nevertheless, it is imperative to note that this operation hinders the multiplication process and ultimately decreases throughput.

Moreover, the shared memory restriction mandates that we can only load into shared memory the elements of one multiplication at a time, preventing us from performing three multiplications in parallel. This limits to performing one multiplication at a time. As a result, this factor adds another drawback to the implementation of TMVP in CUDA-cores,

Algorithm 8 PostProcess: Parallel algorithm to process the polynomial coefficients via three N-dimensional TMVPs and modulo p

Input: $N-$length vectors, p_1, p_2 and p_3
Output: Matrix out of N-length degree, with elements in U16 format and modulo p

1: $tid = thread$ ID
2: $bid = block$ ID
// `Launching` N `threads at maximum`
3: **if** $tid < N$ **then**
4: $out[bid+tid] += (p_1[bid+(tid\times 2)]+p_1[bid+(tid\times 2)+1]+p_2[bid+(tid\times 2)]+p_2[bid+(tid\times 2)+1])\%p$
5: $out[bid+tid] += (p_1[bid+(tid\times 2)]+p_1[bid+(tid\times 2)+1]-p_3[bid+(tid\times 2)]-p_3[bid+(tid\times 2)+1])\%p$
6: **else**
7: $out[bid+tid] = 0$
8: **end if**

Algorithm 9 Tensor-cores implementation of 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: $threads\_tot = 32 \times 2 \times (N/32)^2$
// `Calc. number of blocks`
2: $tc\_blocks = threads\_tot/max\_threads$
// `Number of thread`
3: $tc\_threads = max\_threads$
4: ParNegCyc$< N, N >$ $(fp16\_A, A)$ ▷ Alg.5
5: ParU16toFP16$< N/2, N/2 >$ $(fp16\_B, B)$ ▷ Alg.10
6: TensorCore$< tc_blocks, tc_threads >$ $(fp16\_A, fp16\_B, fp32\_C)$ ▷ Alg.11
7: FP32toU16$< N/2, N/2 >$ $(c, fp32\_C)$ ▷ Alg.12

Algorithm 10 ParU16toFP16: Pre-processing elements for the matrix-vector product converting from U16 to FP16

Input: $N \times N$-length polynomial in_1 and N-length polynomial in_2
Output: Matrix a_1, a_2, a_3 and vector b_1, b_2, b_3 in FP16 format

$tid = thread$ ID
$bid = block$ ID
// `Launch` *N/2* `blocks and` *N/2* `threads in`
// `parallel`
Algorithm 6 steps.

which significantly decreases throughput. However, if a GPU increases shared memory in the future, it could be possible to overcome these limitations. With more shared memory, we could potentially have access to more data, which helps to store multiple copies of vectors and would allow us to perform three multiplication in parallel. This, in turn, would lead to faster processing and improved overall performance.

After computing the matrix-vector product, we need to do some post-processing to get the final result. In Algorithm 8, input polynomials are read by *N/2* threads in parallel. Lines 4 and 5 show the post-processing steps given in equation 2 and we then perform the modulo *p* to get the final result.

C. TMVP-2 implementation using Tensor-cores

The implementation of TMVP on CUDA-cores can be improved by utilizing Tensor-cores, the technique for which is presented in Algorithm 9. Lines 1, 2, and 3 in Algorithm 9 calculate the required numbers of threads and blocks to perform multiplication in Tensor-cores. Line 4 rearranges polynomial A into the same nega-cyclic pattern discussed in Section III-A and described in Algorithm 5. After this, line 5 pre-processes polynomials A and B for the three TMVP multiplications in equation 2.

The arrangement of matrices and vectors is described in Algorithm 10. Similarly, line 6 executes Algorithm 11, which computes the matrix-vector product using Tensor-cores. Finally, line 7 post-processes the products and calculates the final result. Note that although the pre-processing steps for CUDA-cores (algorithms 5 and 6) and Tensor-cores (algorithms 5 and 10) are similar, the format of the output from Algorithms 6 and 10 differs. In CUDA-cores, the output is the same U16 format, whereas in Tensor-cores, the format changes to FP16.

Algorithm 11 shows the Tensor-cores polynomial convolution that computes all three multiplications in TMVP form. The matrix multiplication in Tensor-cores is performed as 16×16 having *32* threads in a warp. For larger matrices, multiple warps can be used to compute separate portions of the matrix. The results are then aggregated repeatedly to produce the final results. For example, to multiply a 32×32 matrix, four warps are launched in parallel to perform 16×16 matrix multiplication as shown in Figure 4. The other four warps compute the other half of the matrix in parallel. This means the process requires two iterations to perform 32×32 matrix multiplication. The final results are stored in Matrix *C* in parallel. However, for TMVP polynomial convolution in both Saber and Sable, $(128/16)^2$ warps and $128/16$ iterations are required to perform the operation.

In Algorithm 11, matrix a_1, a_2, and a_3 are comprised of public/private keys arranged and pre-processed in nega-cyclic form, and matrix b_1, b_2 and b_3 represent polynomial *B*. All matrices are stored in the global memory. Note that in this article, we use *fragment* to denote the temporary storage used to hold the matrices involved in Tensor-cores computations. First, Algorithm 11 initializes nine fragments: three for the 16×16 sub-matrices, three for sub-vectors, and three for collecting results of the multiplication of matrices and vector fragments (lines 1-9). The first multiplication, iterates through matrix a_1 (row-major) and matrix b_1 (column-major) to multiply in parallel (lines 18-22). In each iteration, 16×16 sub-matrices are loaded from matrix a_1 and matrix b_1 (in global memory) for concurrent matrix multiplication. $(N/32)$ Each warp operates on separate regions of matrix a_1 and matrix b_1. The collected results are transferred to matrix p_1 in global memory (line 24) in column-major form to ensure correctness. This is repeated for the other two multiplications (lines 26–40) and the outputs are stored in p_2 and p_3.

Algorithm 11 Tensor-cores: TMVP based parallel polynomial convolutions.

Input: $N/2 \times N/2-$length matrices a_1, a_2, a_3 and $N/2-$length vector b_1, b_2, b_3, where N is a multiple of 16.

Output: $N/2 \times N/2-$length matrix, p_1, p_2, and p_3 holds the nega-cyclic convolution of distinct polynomials $(a_1, b_1), (a_2, b_2)$, and (a_3, b_3).

// 16×16 with precision FP16 initialization of fragment $(a_1, b_1), (a_2, b_2)$, & (a_3, b_3)
$fragment < a_1, 16, 16, 16, half, row\_major > a_1\_frag$
$fragment < b_1, 16, 16, 16, half, col\_major > b_1\_frag$
$fragment < a_2, 16, 16, 16, half, row\_major > a_2\_frag$
$fragment < b_2, 16, 16, 16, half, col\_major > b_2\_frag$
$fragment < a_3, 16, 16, 16, half, row\_major > a_3\_frag$
$fragment < b_3, 16, 16, 16, half, col\_major > b_3\_frag$
// 16×16 with precision FP32 initialization of fragment C
$fragment < accumulator, 16, 16, 16, float > c_1\_frag$
$fragment < accumulator, 16, 16, 16, float > c_2\_frag$
$fragment < accumulator, 16, 16, 16, float > c_3\_frag$
// Compute the warp ID and indices
$tid = thread$ ID
$bid = block$ ID
$blockDim =$ block dimension
$id\_warp = (bid \times blockDim + tid)/32$
$row\_idx = (id\_warp\%(N/32)) \times 16$
$col\_idx = (id\_warp/(N/32)) \times 16$
$acc\_idx = row\_idx + col\_idx \times N/2$
for i from 0 to $(N/32)$ **do**
    $a_1\_id = row\_idx \times N/2 + i \times 16$
    $b_1\_id = col\_idx \times N/2 + i \times 16$
    $load\_matrix\_sync(a_1\_frag, a_1 + a_1\_id, N/2)$
    $load\_matrix\_sync(b_1\_frag, b_1 + b_1\_id, N/2)$
    $mma\_sync(c_1\_frag, a_1\_frag, b_1\_frag, c_1\_frag)$
end for
// Store c_1_frag output in p1
store_matrix_sync($p_1 + acc\_idx, c_1\_frag, N/2, col\_major$)
for i from 0 to $(N/32)$ **do**
    $a_2\_id = row\_idx \times N/2 + i \times 16$
    $b_2\_id = col\_idx \times N/2 + i \times 16$
    $load\_matrix\_sync(a_2\_frag, a_2 + a_2\_id, N/2)$
    $load\_matrix\_sync(b_2\_frag, b_2 + b_2\_id, N/2)$
    $mma\_sync(c_2\_frag, a_2\_frag, b_2\_frag, c_2\_frag)$
end for
// Store c_2_frag output in p3
store_matrix_sync($p_2 + acc\_idx, c_2\_frag, N/2, col\_major$)
for i from 0 to $(N/32)$ **do**
    $a_3\_id = row\_idx \times N/2 + i \times 16$
    $b_3\_id = col\_idx \times N/2 + i \times 16$
    $load\_matrix\_sync(a_3\_frag, a_3 + a_3\_id, N/2)$
    $load\_matrix\_sync(b_3\_frag, b_3 + b_3\_id, N/2)$
    $mma\_sync(c_3\_frag, a_3\_frag, b_3\_frag, c_3\_frag)$
end for
// Store c_3_frag output in p3
store_matrix_sync($p_3 + acc\_idx, c_3\_frag, N/2, col\_major$)

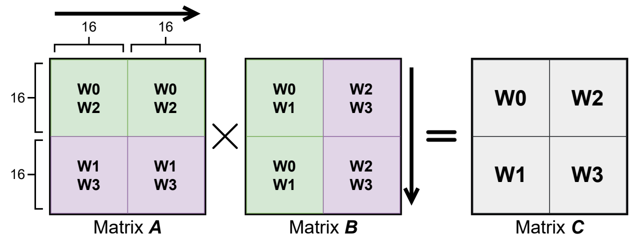


Fig. 4. Matrix multiplication in Tensor-cores of 32×32 having warps (w) running in parallel

Finally, referring to Algorithm 12, output matrix p_1, p_2, and p_3 combine to get the final result (lines 4 and 5), and the format is converted from FP16 to U16. This whole process is done by using *N/2* threads.

Algorithm 12 FP32toU16: process polynomial coefficients from FP32 to U16 via three *n/2*-dimensional TMVPs and modulo p

Input: $N/2 \times N/2$ matrix p_1, p_2 and p_3 with elements in FP32 format

Output: Matrix out of N-length degree, with elements in U16 format and modulo p

1: $tid = thread$ ID
2: $bid = block$ ID
// Launching $N/2$ threads at maximum
3: **if** $(tid < N/2)$ **then**
4:     $out[bid + tid] += (int32\_t)\ (p_1[bid + tid] + p_2[bid + tid])\%p$
5:     $out[bid + N/2 + tid] += (int32\_t)\ (p_1[bid + tid] - p_3[bid + tid])\%p$
6: **else**
7:     $out[bid + tid] = 0$
8: **end if**

IV. Experiment Results and Discussion

This section presents a series of experiments to assess the efficacy of our proposed methodology. These experiments were conducted on a workstation equipped with a 2.10GHz Intel Core i7-12700F CPU with 16GB of RAM and an NVIDIA RTX3060 Ti GPU having a 1410 MHz frequency and 8 GB GDDR6 memory.

A. Performance of TMVP-2 polynomial convolution

In this section, we compare the performance of TMVP polynomial convolution using both CUDA-cores and Tensor-cores. We launched $(N/32)^2$ warps and N threads per block for polynomial convolution using Tensor-cores and CUDA-cores, respectively. Based on the results presented in Figure 5 and Table III, it is evident that the proposed TMVP polynomial convolution with Tensor-cores outperformed CUDA-cores. Although the difference is small at the initial batch sizes, the throughput on CUDA-cores starts to saturate when the batch size exceeds 64. The difference between Tensor-cores and CUDA-cores versions becomes significant as the batch

TABLE III
PERFORMANCE COMPARISON OF TMVP-BASED POLYNOMIAL CONVOLUTION USING TENSOR-CORES AND CUDA-CORES

Batch size (K)	CUDA-cores	Tensor-cores
	Throughput (1000 multiplications per second)	
1	9.91	13.8
8	91.326	110.558
32	310.89	443.89
64	461.56	893.348
128	577.2	1844.34
256	738.497	3654.79
512	811.230	6740.39
1024	851.13	10861.83

TABLE IV
READ/WRITE OPERATIONS ON SHARED MEMORY FOR THE TMVP IN CUDA-CORES

TMVP Multiplications	Tot. Read Elements	Tot. Write Elements	Tot. Read/Write Operations
P1	16640	16640	33280
P2	32896	16640	49536
P3	32896	16640	49536
Total Operations	82432	49920	132352

size increases beyond 64. For instance, at a batch size of 1, Tensor-cores is only 1.39× faster than CUDA-cores. However, this difference increases to 8.31× and 12.76× at batch sizes of 512 and 1024, respectively.

The low performance from CUDA-cores is due to the limited shared memory in the GPU and the large number of read/write operations required for polynomial multiplication. As mentioned in Section III-B, the limited shared memory is insufficient to hold multiple copies of vectors. Consequently, to locate the precise element in the vector for matrix multiplication, the modulo operation must be employed. Nonetheless, this operation acts as a conditional statement for each thread, resulting in a reduction in performance. Secondly, as seen in Algorithm 7, we are conducting three TMVP multiplications in a single kernel. Table IV depicts the number of read/write operations executed in one CUDA-cores kernel. Overall, 82,432 reads and 49,920 writes were performed in one kernel to accomplish three TMVP polynomial convolutions. According to Table II, when using Saber and Sable, the value of $\ell = 3$. This means that in order to complete one polynomial convolution, 82,432×3 read operations and 49,920×3 write operations are required. However, the TMVP in Tensor-cores does not have the same memory limitations and is capable of processing multiple copies of vectors simultaneously, resulting in high throughput compared to the TMVP in CUDA-cores.

B. Performance breakdown for TMVP Tensor-core and CUDA-core based implementations

Table V provides the performance breakdown of polynomial convolution in Saber and Sable by utilizing the proposed techniques on Tensor-cores and CUDA-cores. The analysis of execution times was conducted using a batch size of $K = 128$, with both CUDA-cores and Tensor-cores given a sufficient workload. In the tensor-cores version, organizing poly a into a nega-cyclic matrix takes up about 35% of the overall time, whereas pre-arrangement of poly A and poly B takes up 15% of the total time. Matrix-vector multiplication in Tensor-cores is the most time-consuming operation (about ≈ 37%), which is close to the nega-cyclic arrangement of poly a. Converting the format from FP32 to U16 and simultaneously performing reduction requires the least amount of time (about ≈ 12%). The performance breakdown in CUDA-cores shows that nega-cyclic rearrangement of poly a consumes only ≈ 11% of the total time, whereas pre-arrangement of both polynomials only takes 5% of the time. Matrix-vector multiplication in CUDA-cores consumes the most time (about ≈ 79%). Post-arrangements of elements and performing modulo consume the least amount of time (nearly 4%).

Based on the above discussion, it becomes apparent that polynomial convolution using Tensor-cores requires less shorter multiplication time compared to CUDA-cores. This can be attributed to the superior capabilities provided by Tensor-cores in executing matrix-vector multiplications more efficiently than CUDA-cores. Moreover, as elucidated in Section IV-A, the large number of read/write operations on shared memory required by CUDA-cores also needs more time for multiplication. This duration increases in proportion to larger batch sizes. In contrast, Tensor-cores do not use shared memory because most of the computations are performed directly in the registers.

C. Comparing KEX and KEM performance on a GPU

This section presents the KEX and KEM experiment results from Saber and Sable after implementing the TMVP techniques as proposed. The experiments take into account different batch sizes (K) and utilize two types of GPU: CUDA-cores and Tensor-cores. The KEX performance for both schemes is given in Table VI. Implementation of Saber encryption using TMVP on CUDA-cores and Tensor-cores yielded impressive results.

At a batch size of 16, Tensor-cores achieved 45,625 and 237,869 encryption and decryption operations per second, respectively, whereas CUDA-cores achieved only 35,358 and 179,921 operations per second. Notably, Tensor-cores was 1.2× faster for encryption and 1.3× faster for decryption than the CUDA-cores. We determined that increasing the batch size led to higher throughput for both schemes due to the increased workload in fully occupying a GPU. However, the difference in throughput between Tensor-cores and CUDA-cores also increased, with the highest throughput occurring at $K = 512$. In fact, Tensor-cores achieved 424,437 and 6,259,781 encryption and decryption operations per second, respectively, which were 2.6× and 6.8× faster than CUDA-cores. Similar results were observed with our implementation of Sable encryption and decryption. Initially, the difference between CUDA-cores and Tensor-cores encryption and decryption was small. At $K = 512$, Tensor-cores achieved 457,155 and 5,621,925 encryption and decryption operations per second, respectively, which were 2.7× and 6.2× faster than the CUDA-cores.

Table VII shows the throughput from Saber and Sable KEMs on a GPU using the TMVP on Tensor-cores and

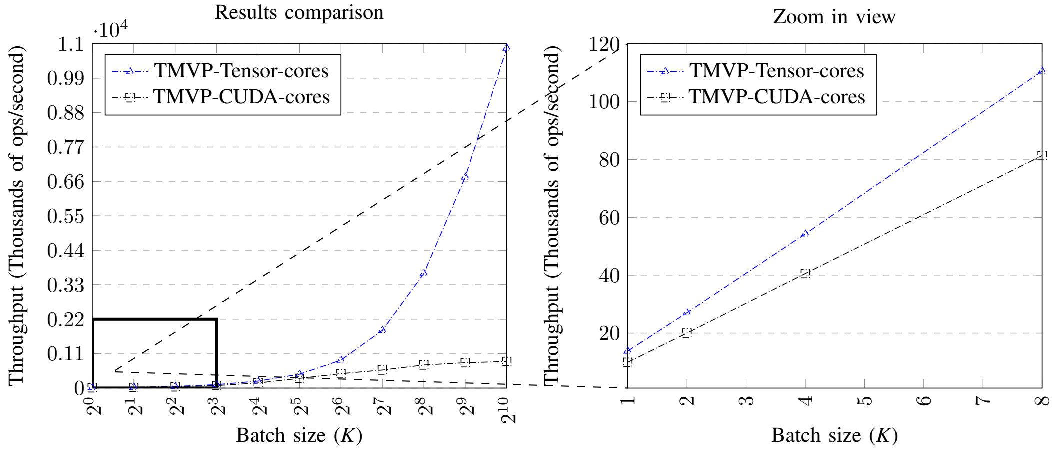


Fig. 5. Performance Comparison of TMVP-based polynomial convolution using Tensor-cores and CUDA-cores

TABLE V
PERFORMANCE BREAKDOWN OF THE TMVP POLYNOMIAL CONVOLUTION USING TENSOR-CORES AND CUDA-CORES AT K=128

Operation	Tensor-cores		CUDA-cores	
	Time (μs)	%	Time (μs)	%
ParNegCyc (Poly $A \rightarrow$ Algorithm 5)	25.74	35.08	25.83	11.64
ParU16toFP16 (Pre-arrangement of Poly A & $B \rightarrow$ Algorithm 10)	11.52	15.6	-	-
PreArr (Pre-arrangement of Poly A & $B \rightarrow$ Algorithm 6)	-	-	11.82	5.33
Tensor-cores (Matrix-vector multiplication $\rightarrow$ Algorithm 11)	27.18	37.04	-	-
CUDA-cores (Matrix-vector multiplication $\rightarrow$ Algorithm 7)	-	-	175.29	79.04
FP32toU16 (Post arrangement $\rightarrow$ Algorithm 12)	8.94	12.18	-	-
PostProcess (Post arrangement $\rightarrow$ Algorithm 8)	-	-	8.82	4.00
Total	73.38	100	221.76	100

TABLE VI
COMPARING THE THROUGHPUT OF SABER AND SABLE KEX WITH THE TMVP CUDA-CORES AND TENSOR-CORES IMPLEMENTATIONS AT DIFFERENT BATCH SIZES

Batch size (K)	Saber				Sable			
	Throughput (encryptions/decryptions per second)							
	CUDA-cores		Tensor-cores		CUDA-cores		Tensor-cores	
	Encrypt	Decrypt	Encrypt	Decrypt	Encrypt	Decrypt	Encrypt	Decrypt
16	35358	179921	45625	237869	37838	170706	45183	235404
32	66401	338410	86821	498256	70546	316055	87412	457456
64	98068	505945	155436	957854	102838	483675	156678	896861
128	127218	647564	257583	1912960	129941	629029	263695	1757469
256	153852	824997	359680	3546473	161075	811112	374619	3218020
512	164670	916223	424437	6259781	170956	902628	457155	5621925

CUDA-cores. It is important to note that KEM is an extension of KEX and involves additional hashing operations, which results in lower throughput compared to KEX. At batch size K = 512, the throughput of Saber on Tensor-cores technique was 2.0× faster (encapsulation) and 2.37× faster (decapsulation) than on CUDA-cores implementation. For Sable, the throughput for encapsulation and decapsulation was 1.9× and 2.35× higher than on CUDA-cores implementation, respectively.

Through a thorough analysis, it was found that the matrix-vector multiplication on CUDA-cores is the most time-consuming, particularly when handling large batch sizes. This is primarily attributed to the fact that when K exceeds 64, CUDA-cores are fully loaded, where adding more workload (i.e., increasing the batch size) does not increase the throughput. To achieve optimal performance from GPU implementation, it is critical to utilize fast shared memory, but transferring between global and shared memory also have significant overhead. In contrast, Tensor-cores offer faster processing times owing to their accelerated matrix operations and smaller matrices available for multiplication in the TMVP. Although we require three TMVPs with *128×128* matrices (see Algorithm 11) instead of one *256×256* multiplication, the total number of operations executed for three TMVPs is still less than that in one *256×256* multiplication, resulting a faster polynomial convolution. Detailed explanation of these points can be found in sections III-A and III-C.

TABLE VII
COMPARING THE THROUGHPUT AT DIFFERENT BATCH SIZES FOR THE SABER AND SABLE KEM TMVPS IN CUDA-CORES AND TENSOR-CORES

Batch size (K)	Saber				Sable			
	Throughput (encaps/decaps per second)							
	CUDA-cores		Tensor-cores		CUDA-cores		Tensor-cores	
	Encaps	Decaps	Encaps	Decaps	Encaps	Decaps	Encaps	Decaps
16	25264	24919	29860	30628	25604	25697	29472	30376
32	46803	46572	55878	58268	46790	47221	53395	57336
64	74206	71411	101569	105983	73118	71648	96950	102769
128	102249	94271	171248	179163	97370	93989	156109	173205
256	123648	115786	229200	245263	121304	115829	212816	241560
512	133919	124261	267720	294020	130675	125340	250062	295061

D. Comparison with State-of-the-Art implementations

The graphs in Figure 6 show the performance comparison of proposed TMVP CUDA-cores and Tensor-cores polynomial convolution with schoolbook polynomial convolution proposed by Hafeez et al. [26]. The results indicate that the schoolbook technique implemented on CUDA-cores (SB-CUDA) initially demonstrated impressive performance. However, as the batch size surpassed 64, performance began to saturate, and the Tensor-cores implementation surpassed CUDA-cores. Both Tensor-cores approaches exhibited similar performance until K=256. However, at $K \geq 512$, the proposed TMVP approach outperformed the schoolbook Tensor-cores (SB-TC) approach. It is worth noting that the performance of the TMVP on CUDA-cores was even slower than SB-CUDA [26]. This shows that the TMVP may not always provide performance superior to the schoolbook approach because memory movement plays a critical role in the achieved performance.

Table VIII presents a throughput comparison of our proposed technique, with Schoolbook Tensor-cores implementation (SB-TC) [26] and dot-product instructions (DPSaber) [25]. Hafeez et al. [26] proposed SB-TC for Sable, and Lee et al. [25] proposed DPSaber for Saber. Note that Sable KEM is an improvement over Saber becuase it employs polynomial convolution for efficient inner product and matrix-vector multiplication calculations. DPSaber [25] incorporates dot-product instructions found in GPUs to implement Saber, and SB-TC uses Tensor-cores for the schoolbook method for polynomial convolution in Sable. We conducted experiments on the same GPU used in the SB-TC [26] and directly adopted source code available in the public domain. Similarly, since the source code for DPSaber is open, we utilized that code and experimented on the same GPU for a fair comparison.

Table VIII provides insight into the performance of our proposed TMVP-TC version in comparison to SB-TC and DPSaber. Sp-up 1 denotes the ratio of TMVP-TC to SB-TC, while Sp-up 2 denotes the ratio of TMVP-TC to DPSaber. Referring to matrix-vector multiplication, our findings show that DPSaber performed better when $K \leq 64$. However, SB-TC achieved almost the same throughput as TMVP-TC. At $K \geq 128$, TMVP-TC outperformed DPSaber and achieved 4.24$\times$ higher throughput at $K = 1024$. Similarly, in comparison to S-TC, TMVP-TC achieved 1.12$\times$ higher throughput. Furthermore, our TMVP-TC achieved at least 3.63$\times$ higher throughput than DPSaber for the inner product at $K = 1024$, but with SB-TC, we achieved 1.21$\times$ higher throughput. These results demonstrate that our approach is more advantageous than SB-TC and DPSaber when the batch size is sufficiently large. This is due to the small matrices and fewer multiplication operations in the TMVP approach, as well as the higher instruction throughput on Tensor-cores in comparison to the dot-product instructions and the SB-TC approach.

E. IoT Applications

The efficient implementation of PQC algorithms plays a critical role in the rapidly expanding ecosystem of the Internet of Things (IoT). Edge devices, which are an essential component of this system, often operate under constrained environments that require low-power and resource-saving implementations for KEM and KEX operations. To address these requirements, the proposed TMVP technique is a promising solution, offering reduced storage requirements and algorithmic simplicity for polynomial convolution in lattice-based cryptography.

In addition, gateway servers typically handle the bulk of the data traffic and require high-throughput solutions. The proposed TMVP-based polynomial convolution using Tensor-cores provides significant enhancements over other methods in literature and is particularly well-suited for such scenarios, ensuring secure, fast, and efficient cryptographic operations. Furthermore, TMVP's dual adaptability makes it a versatile solution that can address the distinct needs of both edge devices and gateway servers in the varied landscape of IoT. By leveraging this approach, one can optimize the cryptographic operations and ensure the safety and security of their IoT ecosystem.

V. CONCLUSION

Our research demonstrated the effectiveness of parallel TMVP computations utilizing Tensor-cores and CUDA-cores in accelerating the execution of KEX and KEM algorithms. By applying this technique to the post-quantum KEMs (Saber and Sable), we achieved significant improvements in system performance where high throughput is required, especially for IoT applications. In the case of Sable, our proposed Tensor-cores implementation outperformed traditional CUDA-cores implementations in terms of encryption and decryption speeds.

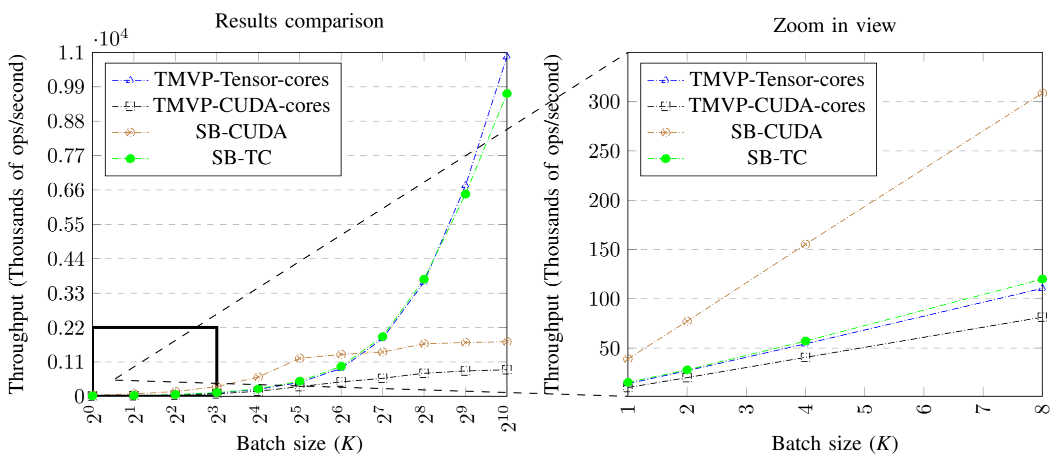


Fig. 6. Performance comparison of TMVP and Schoolbook based polynomial convolution using Tensor-cores and CUDA-cores

TABLE VIII
PERFORMANCE COMPARISON OF TMVP ON TENSOR-CORES FOR INNER-PRODUCT AND MATRIX-VECTOR MULTIPLICATION IN THE SABER AND SABLE KEM VERSUS SCHOOLBOOK TENSOR-CORES [26] AND DPSABER [25] APPROACHES

Batch size (K)	Inner Product (thousand of operations per second)					Matrix-vector (thousand of operations per second)				
	TMVP-TC	SB-TC [26]	DPSaber [25]	Sp-up 1[1]	Sp-up 2[2]	TMVP-TC	SB-TC [26]	DPSaber [25]	Sp-up 1[1]	Sp-up 2[2]
64	893	957	1161	0.93	0.77	366	353	445	1.03	0.82
128	1844	1910	1926	0.96	0.96	762	711	734	1.07	1.07
256	3655	3746	2598	0.97	1.41	1495	1362	1001	1.09	1.49
512	6740	6465	2832	1.04	2.38	2795	2553	1034	1.09	2.70
1024	10861	9681	2991	1.21	3.63	4643	4144	1096	1.12	4.24

[1] TMVP-TC / SB-TC; [2] TMVP-TC / DPSaber

Specifically, we achieved a minimum of 1.1× faster encryption and 1.07× faster decryption. Moreover, our approach demonstrated 1.7× higher throughput for encryption and an impressive 3.1× higher throughput for decryption in KEX operations.

ACKNOWLEDGMENT

The work of Wai-Kong Lee was supported by the Brain Pool Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science and Information Communication Technology (ICT) under Grant 2019H1D3A1A01102607. The work of Seong Oun Hwang was supported by the NRF and funded by the Ministry of Science and ICT under Grant 2020R1A2B5B01002145.

REFERENCES

[1] 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

[2] J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé, "CRYSTALS-Kyber: a CCA-secure module-lattice-based KEM," in *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2018, pp. 353–367.

[3] V. Lyubashevsky, L. Ducas, E. Kiltz, T. Lepoint, P. Schwabe, G. Seiler, D. Stehlé, and S. Bai, "Crystals-Dilithium," *Algorithm Specifications and Supporting Documentation*, 2020.

[4] P.-A. Fouque, J. Hoffstein, P. Kirchner, V. Lyubashevsky, T. Pornin, T. Prest, T. Ricosset, G. Seiler, W. Whyte, Z. Zhang *et al.*, "Falcon: Fast-Fourier lattice-based compact signatures over NTRU," *Submission to the NIST's post-quantum cryptography standardization process*, vol. 36, no. 5, pp. 1–75, 2018.

[5] J.-P. Aumasson, D. J. Bernstein, W. Beullens, C. Dobraunig, M. Eichlseder, S. Fluhrer, S.-L. Gazdag, A. Hülsing, P. Kampanakis, S. Kölbl *et al.*, "SPHINCS," 2019.

[6] P.-Q. Cryptography, "Round 2 submissions," *Electronic resource. Access mode: https://csrc. nist. gov/Projects/post-quantum-cryptography/round-2-submissions*, 2021.

[7] J. M. B. 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*, pp. 474–509, 2021.

[8] J.-P. D'Anvers, A. Karmakar, S. Sinha Roy, and F. Vercauteren, "Saber: Module-LWR based key exchange, CPA-secure encryption and CCA-secure KEM," in *Progress in Cryptology–AFRICACRYPT 2018: 10th International Conference on Cryptology in Africa, Marrakesh, Morocco, May 7–9, 2018, Proceedings 10*. Springer, 2018, pp. 282–305.

[9] KpqC, "Korean pqc competition," 2022, [Online; accessed 30-June-2023]. [Online]. Available: https://www.kpqc.or.kr/competition.html

[10] Z. Liang, B. Fang, J. Zheng, and Y. Zhao, "Compact and Efficient KEMs over NTRU Lattices," *Cryptology ePrint Archive*, 2022.

[11] C. Chen, O. Danba, J. Hoffstein, A. Hülsing, J. Rijneveld, J. M. Schanck, P. Schwabe, W. Whyte, and Z. Zhang, "Algorithm specifications and supporting documentation," *Brown University and Onboard security company, Wilmington USA*, 2019.

[12] J. Cho, Y. Lee, Z. Koo, J.-S. No, and Y.-S. Kim, "Improving Key Size and Bit-Security of Modified pqsigRM," in *2022 13th International Conference on Information and Communication Technology Convergence (ICTC)*. IEEE, 2022, pp. 1463–1467.

[13] 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.

[14] C.-Y. Lee, P. K. Meher, and W.-Y. Lee, "Subquadratic space complexity digit-serial multiplier over binary extension fields using Toom-Cook algorithm," in *2014 International Symposium on Integrated Circuits (ISIC)*. IEEE, 2014, pp. 176–179.

[15] Z.-Y. Wong, D. C.-K. Wong, W.-K. Lee, K.-M. Mok, W.-S. Yap, and A. Khalid, "KaratSaber: New Speed Records for Saber Polynomial

Multiplication using Efficient Karatsuba FPGA Architecture," *IEEE Transactions on Computers*, 2023.

[16] I. K. Paksoy and M. Cenk, "TMVP-based multiplication for polynomial quotient rings and application to saber on arm C ortex-M4," *Cryptology ePrint Archive*, 2020.

[17] ——, "Faster NTRU on ARM Cortex-M4 with TMVP-based multiplication," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 69, no. 10, pp. 4083–4092, 2022.

[18] R. Avanzi, J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé, "Crystals-kyber," *NIST, Tech. Rep*, 2017.

[19] V. B. Dang, K. Mohajerani, and K. Gaj, "High-speed hardware architectures and FPGA benchmarking of crystals-KYBER, NTRU, and Saber," *IEEE Transactions on Computers*, 2022.

[20] Z. Chen, Y. Ma, T. Chen, J. Lin, and J. Jing, "Towards efficient Kyber on FPGAs: A processor for vector of polynomials," in *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2020, pp. 247–252.

[21] G. Xin, J. Han, T. Yin, Y. Zhou, J. Yang, X. Cheng, and X. Zeng, "VPQC: A domain-specific vector processor for post-quantum cryptography based on RISC-V architecture," *IEEE transactions on circuits and systems I: regular papers*, vol. 67, no. 8, pp. 2672–2684, 2020.

[22] F. Farahmand, V. B. Dang, M. Andrzejczak, and K. Gaj, "Implementing and benchmarking seven round 2 lattice-based key encapsulation mechanisms using a software/hardware codesign approach," in *Second PQC Standardization Conference*, 2019.

[23] N. Gupta, A. Jati, A. K. Chauhan, and A. Chattopadhyay, "PQC acceleration using GPUs: Frodokem, NewHope, and KYBER," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 3, pp. 575–586, 2020.

[24] 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.

[25] 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*, vol. 69, no. 9, pp. 3591–3604, 2022.

[26] M. A. Hafeez, W.-K. Lee, A. Karmakar, and S. O. Hwang, "High Throughput Acceleration of Scabbard Key Exchange and Key Encapsulation Mechanism Using Tensor Core on GPU for IoT Applications," *IEEE Internet of Things Journal*, 2023.

[27] H. Fan and M. A. Hasan, "A new approach to subquadratic space complexity parallel multipliers for extended binary fields," *IEEE Transactions on Computers*, vol. 56, no. 2, pp. 224–233, 2007.

[28] M. A. Hasan, N. Meloni, A. H. Namin, and C. Negre, "Block recombination approach for subquadratic space complexity binary field multiplication based on Toeplitz matrix-vector product," *IEEE Transactions on Computers*, vol. 61, no. 2, pp. 151–163, 2010.

[29] M. A. Hasan and C. Negre, "Multiway splitting method for Toeplitz matrix-vector product," *IEEE Transactions on Computers*, vol. 62, no. 7, pp. 1467–1471, 2012.

[30] S. Ali and M. Cenk, "Faster residue multiplication modulo 521-bit Mersenne prime and an application to ECC," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 8, pp. 2477–2490, 2018.

[31] H. K. Taşkin and M. Cenk, "Speeding up curve25519 using Toeplitz matrix-vector multiplication," in *Proceedings of the Fifth Workshop on Cryptography and Security in Computing Systems*, 2018, pp. 1–6.

[32] S. Winograd, *Arithmetic complexity of computations*. Siam, 1980, vol. 33.

[33] J.-S. Pan, C.-Y. Lee, A. Sghaier, M. Zeghid, and J. Xie, "Novel systolization of subquadratic space complexity multipliers based on Toeplitz matrix-vector product approach," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 7, pp. 1614–1622, 2019.

[34] J. H. Cheon, H. Choe, D. Hong, and M. Yi, "SMAUG: Pushing Lattice-based Key Encapsulation Mechanisms to the Limits," *Cryptology ePrint Archive*, 2023.

[35] A. Banerjee, C. Peikert, and A. Rosen, "Pseudorandom functions and lattices," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2012, pp. 719–737.

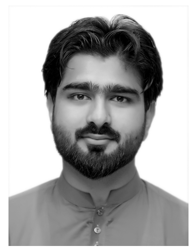
[36] J.-C. See, H.-F. Ng, H.-K. Tan, J.-J. Chang, K.-M. Mok, W.-K. Lee, and C.-Y. Lin, "Cryptensor: A resource-shared co-processor to accelerate convolutional neural network and polynomial convolution," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2023.

[37] 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, 2021.

[38] 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.

[39] S. Sun, R. Zhang, and H. Ma, "Efficient parallelism of post-quantum signature scheme SPHINCS," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 11, pp. 2542–2555, 2020.

[40] W. Dai, B. Sunar, J. Schanck, W. Whyte, and Z. Zhang, "NTRU modular lattice signature scheme on CUDA GPUs," in *2016 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 2016, pp. 501–508.



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. Prior to joining academia, he worked in several multi-national companies including Agilent Technologies (Malaysia) as an R&D engineer. His research interests include 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