IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.DOI

TensorCrypto: High Throughput Acceleration of Lattice-based Cryptography Using Tensor Core on GPU

WAI-KONG LEE[1], (Member, IEEE), HWAJEONG SEO[2], (Member, IEEE), ZHENFEI ZHANG [3], and SEONG OUN HWANG.[1], (Senior Member, IEEE)

[1]Department of Computer Engineering, Gachon University, South Korea. (e-mail: waikonglee, sohwang@gachon.ac.kr)
[2]College of IT Engineering at Hansung University, Seoul, South Korea.
[3]Ethereum Foundation.

Corresponding author: Seong Oun Hwang (e-mail: sohwang@gachon.ac.kr).

This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant (2021-0-00540, 30%), and National Research Foundation of Korea (NRF) grants (2019H1D3A1A01102607, 40% and 2020R1A2B5B01002145, 30%) funded by the Korean Government through Ministry of Science and ICT (MSIT).

ABSTRACT Tensor core is a newly introduced hardware unit in NVIDIA GPU chips that allows matrix multiplication to be computed much faster than in the integer and floating-point units. In this paper, we show that for the first time, tensor core can be used to accelerate state-of-the-art lattice-based cryptosystems. We employed tensor core to speed up polynomial convolution, which is the most time consuming operation in lattice-based cryptosystems. Towards that aim, several parallel algorithms are proposed to allow the tensor core to handle flexible matrix sizes and ephemeral key pairs. Experimental results show that the polynomial convolution computed using the tensor core is at least 2× faster than the version implemented with conventional integer units of the NVIDIA GPU. The proposed tensor-core-based polynomial convolution technique was applied to NTRU, one of the finalists in NIST post-quantum cryptography (PQC) standardization. It achieved 2.02×/1.98× (encapsulation) and 1.56×/1.90× (decapsulation) higher throughput on two parameter sets (ntruhps2048509 and ntruhps2048677), compared to the conventional integer-based implementations on a GPU. In particular, the proposed implementation techniques achieved throughput up to 793651 key encapsulations per second and 505051 decapsulations per second on a RTX2060 GPU. To demonstrate the flexibility of the proposed technique, we extend the implementation to other lattice-based cryptosystems that have a small modulus: LAC and two variant parameter sets in FrodoKEM. Considering that the IoT gateway devices and cloud servers need to handle massive connections from the sensor nodes, the proposed high throughput implementation on GPU is very useful in securing the IoT communication.

INDEX TERMS Cryptography, convolution, lattice-based cryptography, tensor core, graphics processing units, information security, polynomial convolution.

I. INTRODUCTION

THIS security of traditional Public-Key Cryptography (PKC), such as Rivest–Shamir–Adleman (RSA) and elliptic curve cryptography (ECC), relies on one of the three hard mathematical problems: integer factorization, discrete logarithm, or the elliptic-curve discrete logarithm. These hard problems can easily be solved on a sufficiently powerful quantum computer with Shor's algorithm [1], [2]. This creates the need for post-quantum PKC algorithms that can resist the threat from quantum computers in near future.

The National Institute of Standards and Technology (NIST) is in the process of selecting one or more post-quantum cryptography algorithms through a public competition-like process [3], where the candidates need to specify the digital signature and key encapsulation mechanism (KEM). The evaluation criteria not only focuses on security aspects of an algorithm, but also looks into performance from its implementation. Considering the perfor-

mance aspects, the algorithm should be evaluated on various classical platforms to show its efficiency in practical applications. In November 2017, 82 candidate algorithms were submitted to the NIST post-quantum competition for consideration. Of those candidates, seven finalists and eight alternate candidates were selected for the third round, according to the announcement made by NIST in July 2020.

In the third round, five lattice-based cryptography algorithms were selected as finalists (i.e., KYBER, NTRU, SABER, DILITHIUM, and FALCON) while another two are chosen as alternate candidates (i.e., FrodoKEM and NTRU Prime). Compared with other post-quantum cryptography candidates, lattice-based cryptography maintains the majority share in third round. In order to evaluate the practicality of cryptographic algorithms, a lot of work is devoted to improving the performance on various platforms, such as FPGA, microcontrollers and massively parallel processors (GPU). Recently, an FPGA implementation based on approximate computation was proposed [4] to accelerate lattice-based cryptography. The first GPU implementation of NTRU was presented in 2010 [5] and showed that the GPU can achieve very high encryption and decryption throughput by utilizing the product form polynomial and some bit-packing techniques. Other works have been proposed to accelerate the performance of NTRU on a GPU [6]–[10]. There are also several attempts to accelerate post-quantum cryptography (PQC) on various GPU platforms [11]–[14], targeting the parameters in NIST standardization process [15]. However, previous works paid little attention to the power of the new GPU tensor core, which would be a better choice than the ordinary GPU instruction set (i.e., integer/floating point units). The tensor core is a specialized unit released by NVIDIA in its' latest GPU architectures (Volta, Turing and Ampere). Many deep neural network applications take advantages of NVIDIA's tensor core to improve the training and inference performance. However, it is unclear how cryptography can exploit tensor core to improve the implementation performance.

In this paper, our aim is to exploit tensor core to speed up the lattice-based KEM implementation on GPU, in order to achieve high throughput KEM. Our main contributions are summarized below:

1) For the first time, a tensor-core-based polynomial convolution on GPU is presented. Experiments are carried out on two latest GPUs (RTX2060 and RTX3080) that supports tensor core. The proposed technique can handle polynomials with a degree in multiples of 16, which shows up to $3.41\times$ (RTX2060) and $5.77\times$ (RTX3080) faster performance compared to the conventional implementation using 32-bit integer units in the GPU, for polynomial degree $N = 1024$.
2) The first NTRU [16] implementation based on tensor core is proposed in this paper. Since the polynomial degree in NTRU is not a multiple of 16, some modifications are required in order to use the tensor-core-based polynomial convolution. A series of parallel algorithms, including zero padding, sign conversion and type casting, is proposed to achieve this, resulting a high-performance NTRU implementation in a GPU. For instance, the tensor-core-based **ntruhps2048509** can achieve a throughput of 793651 encapsulations per second and 505051 decapsulations per second on RTX2086 GPU. The results are $28.47\times$/$2.02\times$ and $66.50\times$/$1.56\times$ faster than implementation in Central Processing Unit (CPU)/integer units in GPU, for encapsulation and decapsulation, respectively. The same experiments were conducted on RTX3080, where similar speed-ups were obtained. Note that this is also the first NTRU implementation on GPU that follows the NIST PQC specifications [16].
3) The proposed tensor-core-based polynomial convolution can handle various polynomial sizes. To validate this point, we apply the proposed technique to another two lattice-based cryptosystems: LAC and two variant parameter sets of FrodoKEM. The tensor-core-based polynomial convolution in LAC and one variant of FrodoKEM outperformed integer-units-based implementations by $3.10\times$ and $3.31\times$, respectively (RTX2060). Detailed steps to efficiently utilize the proposed technique for polynomial/matrix multiplication in these two schemes are described Section IV.E and IV.F.
4) The source code for tensor-core-based polynomial convolution was placed in the public domain at https://github.com/benlwk/Tensorcrypto. This allows researchers to easily re-produce our results in their own development environments to utilize the tensor-core-aided lattice-based cryptography implementation for their own purposes.

Here we established the criteria to apply our technique over other lattice-based schemes. At a high level, our solution allows very high throughput key encapsulation/decapsulation using a same public-private key pair for each communication session. The proposed solution is applicable to all lattice-based cryptography with small modulus, where multiplication is expressed in the form of a vector and a matrix. This can be either an ideal lattice construction, as in NTRU and LAC, or a generic lattice construction, as in FrodoKEM. However, schemes such as KYBER [12] and NewHope [13] already use NTT-based multiplications, the polynomial convolution is already efficient, so it would be more advantageous to use NTT instead of schoolbook multiplication presented in this paper. Moreover, it is also difficult to use tensor cores to accelerate the NTT computations, since the size of modulus (q) is most likely to exceed 11-bit, which is larger than the supported size of half-precision tensor core. Therefore, we restrict the scope of our paper to the schemes where NTT is slow or not applicable.

Although this paper only use the proposed tensor-core-based polynomial convolution for cryptography, it can be extended to support other applications on GPU. For instance,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3152217, IEEE Access

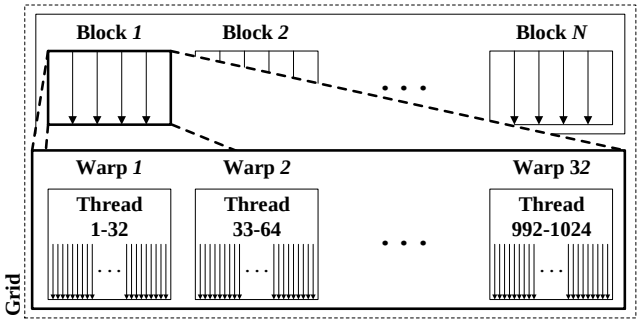


FIGURE 1: Relationship between grid, blocks and threads in CUDA.

polynomial convolution is used to perform image reconstruction [17], feature extraction [18], data preprocessing [19] and signal processing [20]. These applications may also benefits from the tensor-core-based polynomial convolution to achieve high throughput performance.

The remainder of this paper is organized as follows. In Section II, we introduce the background and related prior works. In Section III, we present a novel tensor-core-based polynomial convolution and the implementation of two parameter sets in NTRU. Thereafter, we summarize our experimental results for NTRU, LAC, and FrodoKEM in Section IV. Finally, we conclude the paper in Section V.

II. BACKGROUND AND RELATED WORKS

A. OVERVIEW OF GPU ARCHITECTURE AND CUDA PROGRAMMING MODEL

A GPU consists of thousands of cores, enabling massively parallel computation in many interesting applications. From the hardware perspective, the GPU groups many cores (e.g., 64, 128, or 192) into a Streaming Multiprocessor (SM). The memory in the GPU can be categorized into on-chip and off-chip. On-chip memory refers to register files and shared memory that resides near to the GPU cores. Registers are very fast, but come in small sizes (64–96K 32-bit words per SM). Shared memory is known as "user-managed cache", which is usually used to store frequently accessed values (e.g., look-up table or pre-cached values). Like registers, shared memory is fast but small in size (48–164K 32-bit words per SM). Off-chip memory refers to global memory, which is essentially the DRAM. It comes in a large size (2–16 GB), but the access latency can be up to $300\times$ slower than the registers.

From a programming perspective, many parallel threads form a block and multiple blocks form a grid. This allows flexible arrangement of software threads into the physical SM and cores across many different GPU architectures. The relationship between grids, blocks, and threads is illustrated in Figure 1. NVIDIA GPUs group 32 threads into a *warp*, wherein all 32 threads execute the same instruction in parallel. For this reason, the number of threads per block is usually set in multiples of 32 to avoid divergence in the instruction execution path. Besides that, shared memory also has 32 banks, allowing parallel access by all 32 threads within a warp. Additional features like warp shuffle instruction and tensor core are also designed to work at the warp level to maximize the efficiency of the GPU warp scheduler.

B. TENSOR CORE

In 2017, NVIDIA released the Volta GPU architecture, which introduced a specialized unit named as the tensor core. Tensor core is used to perform one half-precision matrix-multiply-and-accumulate (MMA) on a 4×4 (Volta) or 16×16 matrix (Turing, Ampere) per clock cycle. This greatly improves the throughput of MMA operations, compared to the conventional implementation using CUDA cores, which often requires multiple clock cycles to complete the same number of operations. The tensor core in Turing architecture supports MMA with half-precision inputs and single-precision accumulator. Recently, the Ampere architecture that supports double-precision MMA was released, enabling the use of tensor core in generic scientific computing applications. The latest tensor core in the Ampere architecture also supports TensorFloat-32 (TF32) and Bfloat16 (BF16) new formats that reduces the floating-point precision but maintains the same range. Note that for different precisions, the performance of tensor core varies. The detailed information about the performance of tensor core can be obtained from [28].

Many deep neural network applications can take advantage of the NVIDIA tensor core. For instance, the convolutional neural network (CNN) requires many dot-product computations, which can easily be expressed as MMA operations. However, it is unclear how cryptography implementations can exploit the newly introduced tensor core. In this work, we present a series of algorithms to map the polynomial convolution in lattice-based cryptography to matrix multiplication in order to exploit tensor core for faster performance.

C. LATTICE-BASED CRYPTOGRAPHY

Lattice-based cryptographic constructions are based on the hardness of Shortest Vector Problem (SVP) which approximates the minimal Euclidean length of a lattice vector. Lattice-based cryptography is believed to be secure against both conventional and quantum computers. In the third round of the NIST post-quantum cryptography standardization process, five lattice-based cryptography algorithms were selected as finalists (CRYSTALS-KYBER, NTRU, SABER, CRYSTALS-DILITHIUM, and FALCON) and another two are selected as alternate candidates (FrodoKEM and NTRU Prime). Table 1 shows the summary parameters for these candidates. Most of the lattice-based schemes rely on polynomial convolution, which has high computational complexity. In order to improve the performance of polynomial convolution, we utilize the tensor core and show performance enhancements on lattice-based cryptography with a small modulus, such as NTRU, LAC, and two variant parameter sets of FrodoKEM.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3152217, IEEE Access

TABLE 1: Comparison of lattice-based cryptography in the NIST PQC competition. PKE, KEMs, and DS stand for Public-Key Encryption, Key Encapsulation Mechanisms, and Digital Signature, respectively.

Lattice-based candidates	Application	Category	Modulus (q)	NIST PQC competition
CRYSTALS-KYBER [21]	PKE/KEMs	Module	3,329	Round 3 finalists
SABER [22]			2^{13}	
NTRU [16]		Ideal	2^{11}, 2^{12}, and 2^{13}	
CRYSTALS-DILITHIUM [23]	DS	Module	$2^{23} - 2^{13} + 1$	
FALCON [24]		Ideal	11289	
FrodoKEM [25]	PKE/KEMs	Standard	2^{15} and 2^{16}	Alternate candidates
NTRU Prime [26]		Ideal	4,591, 4,621, ..., 7,879	
LAC [27]	PKE/KEMs	Ideal	251	Round 2 candidate

NTRU encryption is a lattice-based one-way CPA-secure (OW-CPA) public-key encryption scheme that was invented in 1996 [29]. For the purpose of this paper, we do not go into the details of the scheme, interesting readers may refer to [16] for details. The major computation bottleneck in NTRU is the polynomial multiplication over the ring $\mathcal{R}_q := \mathbb{Z}_q[x]/(x^N - 1)$. Since the multiplication is carried out on a ring structure, it is essentially a polynomial convolution. In a nutshell, for a polynomial ring $\mathcal{R}_q := \mathbb{Z}_q[x]/F(x)$, and a small parameter p, an NTRU public key is the ratio of two small polynomials over $h = g/f$ for some small g and f, where f is also invertible modulo p. The small polynomials refer to polynomials with small coefficients, either binary or ternary. The NTRU assumption says that given h, one cannot recover g and f, or distinguish h from a random element over the ring. To encrypt a message polynomial m, one computes $c = prh + m$ for that is co-prime with q, and a randomly sampled small polynomial r. To decrypt, one then computes $cf = prg + mf \equiv mf \mod p$. Since f is invertible modulo p, one can extract m from mf with $f^{-1} \mod p$.

In the NIST PQC competition, there have been two flavors of NTRU, differing in the choice of. The original NTRU scheme, known as NTRU-HPS [29], [30], works over $\mathcal{R}_q := \mathbb{Z}_q[x]/(x^N - 1) = \phi_N(x)\phi_1(x)$. A newer design, referred to as NTRU-HRSS [31], works over $\mathbb{Z}_q[x]/\phi_N(x)$. Note that, although NTRU-HRSS works over $\mathbb{Z}_q[x]/\phi_N(x)$, computations are carried out over $\mathcal{R}_q$ for better efficiency. In addition, both schemes now use a variant of FO transformation to achieve CCA-2 security.

D. PREVIOUS PQC IMPLEMENTATIONS ON GPU

The first implementation of NTRU in GPU dates back to 2010. Hermans et al. [5] showed that GPU can achieve very high encryption and decryption throughput by utilizing product form polynomial and bit-packing techniques. The product form polynomial is no longer used in the NTRU submission to NIST. Following up this work, Lee et al. [7] proposed a sliding window technique to pre-compute some repeating patterns in NTRU polynomial and then stored them in a lookup table. With this technique, some of the multiplication operations can be skipped. Although this work is able to achieve high throughput, it may not be secure against a side channel attack, because the look up table leaks timing information. The NTRU modular lattice signature (NTRU-MLS) scheme [32], [33], which requires operations on large vectors, was optimized with parallel polynomial multiplication on a GPU by Dai et al. [9]. Recently, Lee et al. [10] proposed utilizing the Karatsuba algorithm to speed up polynomial multiplication in NTRU. These previous works were all implemented on the integer units available in GPU. Unlike previous NTRU implementations on GPU, we introduce the first tensor-core-based NTRU implementation.

Recently, there are also interests on implementing other PQC schemes on GPU. Lee et al. [34] show various ways to speed up NTT computation on GPU, targeting polynomials used in qTESLA signature scheme. A following up work from [35] demonstrated that Nussbaumer algorithm can be faster than NTT on the polynomial convolutions when executed on GPU. Sun et al. [14] show a parallel implementation of SPHINCS signature scheme on GPUs with significantly higher throughput compared to CPU implementation. Gupta et al. [11] analyzed various parallelism available in GPU (batch mode and single mode), and evaluate three PQC schemes (FrodoKEM, NewHope and Kyber). Later on, Lee et al. [12] demonstrated a low latency implementation of Kyber KEM, which can be beneficial to latency-sensitive applications. Another interesting work from Gao et al. [13] further improved the throughput achieved by NewHope on two different GPU platforms. Note that all these previous work also do not consider the possibility to use tensor core in their implementation. Besides high performance implementation of PQC, another interesting research direction is to develop efficient implementation of various sieving algorithms to solve hard lattice problems [36].

III. OPTIMIZED PARALLEL IMPLEMENTATION OF NTRU

A. PARALLEL POLYNOMIAL CONVOLUTION THROUGH SCHOOLBOOK CONVOLUTION

Polynomial convolution is known as “truncated polynomial multiplication”. This is the most time-consuming operation in NTRU PKC. A straightforward way to implement this is schoolbook multiplication, wherein the operation exhibits a high degree of parallelism. Referring to Algorithm 1, schoolbook polynomial convolution can be arranged in such a way that it processes one column at a time (the *k* loop, lines 2-6). The *i* loop first computes the multiplication and accumulation up to the *k* element by following ordinary schoolbook multiplication. Next, it proceeds with the remaining polynomial

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3152217, IEEE Access

Algorithm 1: Schoolbook polynomial convolution.

Input: Polynomial a with degree N, Polynomial b with degree N, modulus q.

Output: Polynomial c with degree N, which is the cyclic convolution of a and b.

```
// Accumulate each column serially
1: for k from 0 to N − 1 do
2:     c[k] = 0
3:     for i from 0 to k do
4:         c[k] = c[k] + a[k − i] × b[i]
5:     end for
6:     for i from 1 to N − k − 1 do
7:         c[k] = c[k] + a[k + i] × b[N − i]
8:     end for
9: end for
10: return c%q
```

Algorithm 2: Parallel schoolbook polynomial convolution in NTRU.

Input: P blocks of Polynomial a and Polynomial b with degree N, modulus q.

Output: Polynomial c with degree N, which is the cyclic convolution of a and b.

```
1: tid=thread ID
2: bid=block ID
// Copy polynomials into shared memory
// in parallel
3: shared_a[tid]= a[bid × N + tid]
4: shared_b[tid]= b[bid × N + tid]
5: __syncthreads() ▷ Synchronize all the
   threads
// Accumulate each column in parallel
// with N threads
6: sum=0 ▷ Use register to accumulate‘
7: for i from 0 to tid do
8:     sum = sum+shared_a[tid − i] × shared_b[i]
9: end for
10: for i from 1 to N − tid-1 do
11:     sum = sum+shared_a[tid + i] × shared_b[N − i]
12: end for
13: return c[bid × N + tid] = sum%q
```

convolution through cyclic computation.

Detailed illustrations are presented in Figure 2. One can observe that the operations within the k loop are independent of each other, which allows a highly parallel implementation on the GPU platform to achieve good performance. This technique was previously explored by Dai et al. [9] and it remains the most efficient way to compute polynomial convolutions in a GPU. Note that for NTRU, the polynomial convolution is performed with 32-bit integer unit (INT32).

Algorithm 2 shows the parallel version of schoolbook polynomial convolution that can be implemented efficiently in a GPU. This implementation utilizes P blocks in GPU to perform P polynomial convolutions, where each block computes one polynomial convolution with N threads. Polynomials are first loaded from global memory and cached in shared memory to reduce read/write latency (lines 3-5). Next, each thread is responsible for accumulating one column independently (lines 7-12), with the intermediate results stored in a register (i.e., *sum*). Finally, results are copied to array c which resides in global memory (line 11). One can also easily modify Algorithm 2 to perform nega-cyclic convolution. In particular, instead of performing addition in line 10, one can perform subtraction to achieve nega-cyclic convolution. Besides high parallelism, this implementation ensures minimal access to global memory (two reads operations and one write), with majority of the operations residing in shared memory and registers.

Note that we only need to perform the modulo operation ($sum\%q$) at the end of the convolution. This is because in a GPU implementation, sum is a 32-bit register that is large enough to accommodate the two selected NTRU parameter sets. It is also possible to use a 16-bit sum, because q is a power-of-2 for NTRU. Whenever sum experiences overflow, it carries out a free modulo operation over its word size. However, this is not beneficial to GPU because it does not support a native 16-bit register.

B. PROPOSED POLYNOMIAL CONVOLUTION THROUGH TENSOR CORE

The tensor core was introduced into the GPU to accelerate MMA operations with much higher throughput. By taking a closer look at Algorithm 1, we find that the polynomial convolution can be expressed in the form of matrix multiplication. To achieve this, polynomial a is first packed into a cyclic form to allow the convolution to take place, whereas polynomial b can be stored in a column major form. This operation is illustrated in Figure 3, where the multiplication between matrix A and B produces the same results as polynomial convolution. In other words, one can perform matrix-vector convolution between polynomial a (matrix) and polynomial b (vector), using tensor core. Note that this technique only works where polynomial a can be reused repeatedly. This is not a problem for encryption in NTRU that executes $r * h$, where h is the public-key and r is the random ternary polynomial. One can reuse the public-key h to encrypt multiple plaintexts, and renew the public-key from time to time. On the other hand, polynomial b does not need to be reused, so we can pack many random vectors r into matrix B.

With this proposed technique, NTRU polynomial convolution can be formulated as matrix multiplication and accelerated through the use of the tensor core, which is faster than the conventional INT32 operations.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3152217, IEEE Access

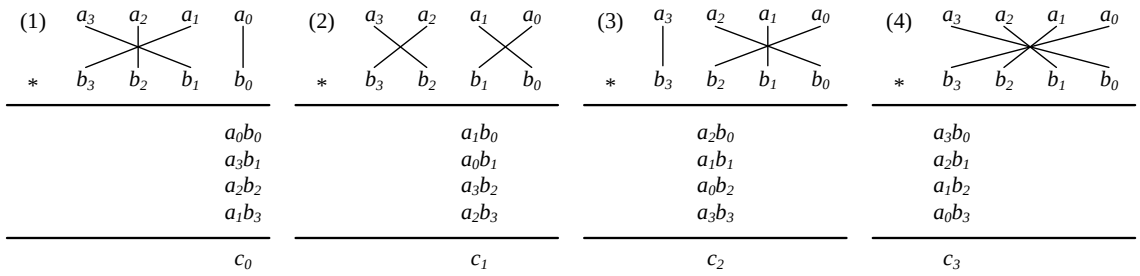


FIGURE 2: Parallel computation of polynomial convolution with integer units in GPU; operations from (1) to (4) are performed, independently.

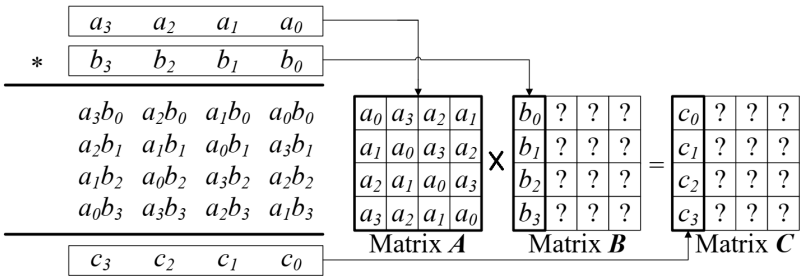


FIGURE 3: Computing polynomial convolution using tensor core in GPU: Matrix *A*, *B*, and *C* represent constant polynomial (e.g. public-key *h*), non-constant polynomial (e.g. random vectors, *r*), and result, respectively. Note that "?" refers to non-constant polynomials that are different from polynomial *b* and *c*.

C. TENSORTRU: NTRU IMPLEMENTATION BASED ON TENSOR CORE

1) Representing a Polynomial in Floating Point

TABLE 2: Supported precision in tensor core [28].

Conf.	Matrix *A*	Matrix *B*	Accumulator	Dimension
1	half (FP16)	half (FP16)	single (FP32)	$16 \times 16 \times 16$ $16 \times 16 \times 16$
2	half (FP16)	half (FP16)	half (FP16)	$16 \times 16 \times 16$ $16 \times 16 \times 16$
3	double (FP64)	double (FP64)	double (FP64)	$8 \times 8 \times 4$ $8 \times 8 \times 4$
4	unsigned (INT8)	unsigned (INT8)	signed (INT32)	$16 \times 16 \times 16$ $16 \times 16 \times 16$
5	signed (INT8)	signed (INT8)	integer (INT32)	$16 \times 16 \times 16$ $16 \times 16 \times 16$

Referring to Table 1, NTRU requires modulus q to be 2^{11}, 2^{12} or 2^{13} depending on the parameter sets chosen. To allow the use of tensor core in performing polynomial convolution, we need to ensure that the polynomial coefficients can be represented in the supported precision in tensor core, as depicted in Table 2. Because tensor core only support byte-level integers (configurations 4 and 5), we cannot represent the NTRU polynomial coefficients in integer due to insufficient precision. Another option would be to convert the polynomial coefficients from integers to floating point numbers, and then utilize one of the three possible configurations (configurations 1-3). Since configurations 1 and 2 have much higher performance compared to configuration 3, we explore these two options to implement NTRU.

The parameter sets **ntruhps2048509** and **ntruhps2048677** requires q= 2^{11}, which allows polynomial elements to be represented exactly in FP16. The accumulator needs to be sufficiently large to hold the results of matrix multiplication. For instance, by using q= 2^{11} the element size is 11-bit, so each pair of multiplications between poly_a and poly_b produces a number with a 22-bit maximum. However, one of the polynomials in NTRU is ternary (i.e., elements are only consists of -1, 0 and 1). Since we are using floating point numbers to represent the polynomial elements, multiplication produces only the maximum 11-bit results (i.e., $(2^{11}-1) \times 1 = 2^{11}-1$ and $(2^{11}-1) \times -1 = -2^{11}-1$). In the process of polynomial convolution, the accumulated value can grow up to a maximum of $N \times 2^{11} - 1$. Hence, for the two selected parameter sets, the accumulator must be able to hold at least 20-bit ($log_2(509 \times (2^{11}-1))$) and 21-bit ($log_2(677 \times (2^{11}-1))$) data for **ntruhps2048509** and **ntruhps2048677**, respectively. Due to this restriction, we utilized configuration 1 in accelerating NTRU polynomial convolution, because the single precision accumulator can hold 24-bit integer value at maximum. Note that in practice, the accumulated values may well below 20-bit, because the accumulation can go in both directions (addition or subtraction) depending on the ternary polynomials.

Another two NTRU parameters (**ntruhps4096821** and **ntruhrss701**) can be implemented in tensor core with double precision using configuration 3. However, the performance of FP64 tensor core is much slower compared to FP32, and they only support a smaller matrix size (8×8). A faster FP64

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3152217, IEEE Access

tensor core released in the future may open up opportunities to apply our technique into these two parameter sets.

2) Parallel Polynomial Convolution using Tensor Core

Tensor core were developed to handle a small matrix with dimension of 16×16 within a warp (32 threads), as depicted in the upper right part of Figure 4. To handle a larger matrix, one can utilize many warps computing different parts of the matrix, and then accumulate the results, iteratively. Referring to Figure 4, there are three steps to complete when we perform matrix multiplication for a 32×32 matrix. First, four warps (w_0 to w_3) are launched in parallel to compute matrix multiplication on 16×16 dimensions. For instance, w_0 and w_2 read the same piece of data (16×16) from Matrix *A*, but they read different data from Matrix *B* for multiplication and accumulation. Intermediate results from this step are stored in a temporary array. Next, the four warps proceed to compute another half of the matrix in parallel. In other words, two iterations are required to complete a 32×32 matrix multiplication. Lastly, results are stored into Matrix *C* in parallel in different memory locations. To compute a larger matrix, we need $(M/16)^2$ warps and $M/16$ iterations to compute $M \times M$ matrix multiplication in parallel. This tensor-core-based matrix multiplication is utilized to compute polynomial convolutions in NTRU.

Referring to Algorithm 3, the tensor-core-based polynomial convolution requires the input matrices to be in multiples of 16×16. Matrix *A* is the constant polynomial *a* organized in cyclic form (e.g. public-key *h* in NTRU), while Matrix *B* consists of *M* non-constant polynomials (e.g. random vector *r* in NTRU). Note that all matrices are stored as a one-dimensional memory array (i.e., global memory). The algorithm first initializes two fragments to hold the 16×16 sub-matrices and one fragment to hold the accumulated results (lines 1-3). Next, it loops through Matrix *A* (row major) and Matrix *B* (column major) to perform the matrix multiplication in parallel (lines 11-16). For each iteration, 16×16 sub-matrices are loaded from Matrix *A* and Matrix *B* (in global memory) to perform matrix multiplication in parallel. $(M/16)^2$ warps are executed in parallel, with each warp operating on different parts of Matrix *A* and Matrix *B* as depicted in Figure 4. Finally, the accumulated results are copied from the tensor core to Matrix *C* in global memory (line 17) in column major form.

Algorithm 3: TC-PC: parallel polynomial convolutions using tensor core.

Input: $M \times M$ matrix *A* (constant polynomial *a* in cyclic form), $M \times M$ matrix *B* (non-constant polynomials *b*), *M* must be multiple of 16.

Output: $M \times M$ matrix *C*, which contains the cyclic convolution of polynomial *a* and many polynomial *b*.

```
// Initialize fragment a and b with
// 16 × 16 dimension and FP16 precision
1: fragment<A, 16, 16, 16, half, row_major> a_frag
2: fragment<B, 16, 16, 16, half, col_major> b_frag
// Initialize fragment c with 16 × 16
// dimension and FP32 precision
3: fragment<accumulator, 16, 16, 16, float> c_frag

// Compute the warp ID and indices
4: tid=thread ID
5: bid=block ID
6: blockDim=block dimension
7: warpID = ⌊(bid × blockDim + tid)/32⌋ ▷ 32
   threads per warp
8: row_idx = (warpID%⌊M/16⌋) × 16
9: col_idx = (warpID/⌊M/16⌋) × 16
10: store_idx = row_idx + col_idx × M

11: for i from 0 to ⌊M/16⌋ do
12:     ldA = row_idx × M + i × 16
13:     ldB = col_idx × M + i × 16
//Load 16 × 16 sub-matrix from
//Mat. A and B
14:     load_matrix_sync(a_frag, A + ldA, M)
15:     load_matrix_sync(b_frag, B + ldA, M)
// Perform matrix multiplication and
// accumulate the results in c_frag
16:     mma_sync(c_frag, a_frag, b_frag, c_frag)
17: end for
// Store the results from c_grat
// into Matrix C
18: store_matrix_sync(C + store_idx, c_frag, M,
    col_major)
```

3) Handling a Matrix not in a multiple of 16×16

The polynomial degrees of two selected NTRU parameter sets (**ntruhps2048509** and **ntruhps2048677**) are $N = 509$ and $N = 677$ respectively. However, the tensor-core-based matrix multiplication can only work for matrices that are multiples of 16×16. This implies that we cannot use tensor core to accelerate these two NTRU parameter sets straightforwardly.

There are two methods to overcome this limitation. The first method is through a hybrid algorithm that combines the tensor core and integer-based polynomial convolution. Figure 5a shows a high-level illustration of such a hybrid algorithm. In this example (parameter set **ntruhps2048509**), one can utilize tensor core to compute the polynomial convolution of 496×496 (Region *A*), and then complete the remaining computations (region *B*, *C*, and *D*) in three steps. Note that this hybrid algorithm is less efficient, because some of the computations cannot be fully parallelized with tensor core. This limitation, however, allows us to utilize the fast tensor core to accelerate a polynomial convolution in NTRU and other similar lattice-based cryptographic schemes. On the other hand, one can utilize the second method by padding

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3152217, IEEE Access

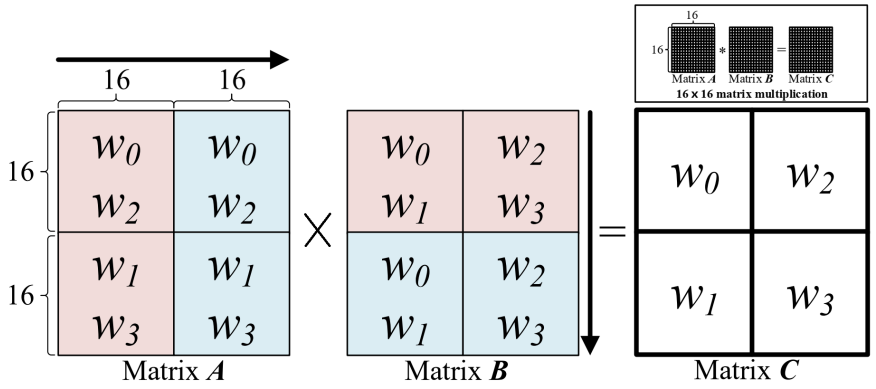
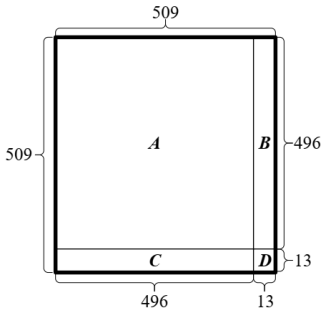
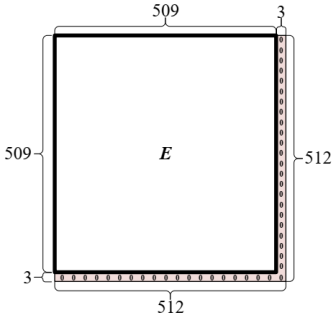


FIGURE 4: Matrix multiplication: dimensions 32×32; w: warps running in parallel; the arrow indicates computation order.

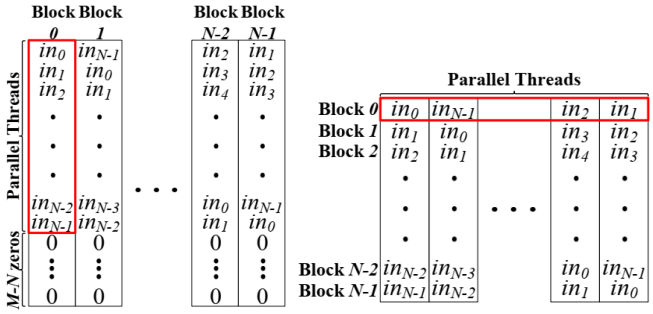


(a) Method 1: hybrid algorithm



(b) Method 2: padding with zeros

FIGURE 5: Handling a matrix that is not a multiple of 16×16 (parameter set: **ntruhps2048509**), $E = A + B + C + D$.



(a) Read from polynomial (b) Write to matrix

FIGURE 6: Arranging a polynomial in cyclic form and storing it in a matrix.

zeros into poly_a to form a matrix that is a multiple of 16×16. Referring to Figure 5b, zeros are padded to form a matrix of 512×512. This allows us to perform a polynomial convolution of $N \times N$ completely in tensor core, at the expense of some additional memory. The redundant storage required by this method can go up to a maximum of $(p - N) \times N + (p - N) \times p$, where p refers to the closest multiple of 16 that is larger than N. In this paper, we proposed utilizing the second method (i.e., zero-padded polynomial convolution), since it can be computed fully in tensor core, which is more efficient than first method (i.e., the hybrid approach).

To achieve high-performance NTRU implementation in GPU, we proposed a series of parallel algorithms to efficiently perform the following tasks:

1) Organize the polynomial in cyclic form and pad the remaining parts with zeros to construct the matrix in a multiple of 16×16 (Algorithm 4).
2) Convert unsigned 16-bit integer (U16) polynomial elements to 16-bit floating point (FP16) format (Algorithm 5).
3) Convert 32-bit floating point (FP32) elements to unsigned 16-bit integers (U16) and perform modulo operations (Algorithm 6).

Referring to Algorithm 4 and Figure 6, the input polynomial (*in*) is read by N threads in parallel, and then written to the output matrix (*out*). Note that each block reads different cyclic form in order to achieve high parallelism. Algorithm 5 shows the steps to convert U16 polynomial elements into FP16 format. Lines 5-8 are only necessary if we are dealing with ternary values; it converts -1 in integer format (i.e., 2047 when $q = 1048$) to FP16 format. Algorithm 6 first converts FP32 elements to INT32 format (line 4) to keep the original precision, then performs modulo q and store final results in U16 format.

With these three proposed algorithms, one can perform highly parallel polynomial convolution for NTRU using ten-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3152217, IEEE Access

Algorithm 4: ParCyc: parallel algorithm to arrange polynomial in cyclic form.

Input: Polynomial *in* with degree *N*.
Output: Matrix *out* with $M \times M$ dimension, which is the polynomial *in* organized in cyclic form and padded with zeros for unused elements.

1: tid=thread ID
2: bid=block ID
// Launch M blocks and M threads
// in parallel
3: **if** $tid <$N **then**
4: $out[bid + tid \times M] = in[(tid - bid)\%N]$
5: **else**
6: $out[bid + tid \times M] = 0$
7: **end if**

Algorithm 5: ParU16toFP16: parallel algorithm to convert polynomial elements from U16 to FP16.

Input: Matrix *in* with *N* different polynomials of degree *N* in U16 format.
Output: Matrix *out* with *N* different polynomials of degree *N* in FP16 format.

1: tid=thread ID
2: bid=block ID
3: $temp = 0$ ▷ Initialize a FP16 variable
// Launch N blocks and N threads
// in parallel
4: $temp = in[bid \times M] + tid$
5: **if** $temp = 2047$ **then**
6: $out[bid \times M + tid] = -1$ ▷ Converting -1 from U16 to FP16
7: **else**
8: $out[bid \times M + tid] = temp$
9: **end if**

Algorithm 6: ParFP32toU16: parallel algorithm to convert polynomial elements from FP32 to U16 and perform modulo q.

Input: $M \times M$ matrix *in* with elements in FP16 format.
Output: $M \times M$ matrix *in* with elements in U16 format and modulo q.

1: tid=thread ID
2: bid=block ID
3: $temp = 0$ ▷ Initialize a FP32 variable
// Launch N blocks and N threads
// in parallel
4: $temp = in[bid \times M] + tid$
5: $out[bid \times M + tid] = temp\%q$

Algorithm 7: Parallel implementation of NTRU polynomial convolution using tensor core in a GPU.

Input: polynomial a with degree *N* (constant polynomial), *N* polynomial b with degree *N* (non-constant polynomials), modulus q.
Output: $M \times M$ Matrix *C*, which contains the cyclic convolution of polynomial a and many different polynomial b.

// **CPU Phase:**
// Init. two matrices in FP16 to store
// the converted a and b
1: $fp16\_A$
2: $fp16\_B$
// Init. a matrix in FP32 to store
// the results from tensor core
3: $fp32\_C$

// **GPU Phase:**

// Calc. total number of warps required

4: $warp\_tot = (M/16)^2$
5: $tc\_threads = warp\_tot \times 32$
// Calc. number of blocks
6: $tc\_blocks = tc\_threads/max\_threads$
// Limit the number of threads
7: $tc\_threads = max\_threads$
8: ParCyc$< N, N >(fp16_A, a)$ ▷ Alg. 4
9: ParU16toFP16$< N, N >(fp16_B, b)$ ▷ Alg. 5
10: TC-PC$< tc_blocks, tc_threads >$ $(fp16\_A, fp16\_B, fp32\_C)$ ▷ Alg. 3
11: ParFP32toU16$< N, N >(C, fp32_C)$ ▷ Alg. 6

sor core, where the steps are given in Algorithm 7. Three floating point matrices are first initialized to zero in the CPU; this process is only performed once. Next, the two proposed algorithms are implemented in the GPU to perform pre-processing on Matrix *A* and Matrix *B* (lines 8-9). Subsequently, tensor core is used to perform the polynomial convolution, resulting in Matrix *fp32_C* in FP32 format (line 10). Lastly, this result is converted to Matrix *C* with U16 format and modulo with q to obtain the final output.

Another point to note is that when we use the proposed technique to implement NTRU, the polynomial convolution for decryption is slightly different from the encryption. During the encryption process, one computes $r * h$, where h is the public-key to be treated as a constant polynomial, while r is the non-constant and small ternary polynomial. On the other hand, the private key f used in decryption is a small ternary polynomial to be treated as a constant polynomial. In such a case, Algorithms 4 and 5 needs to be slightly revised. In particular, lines 5-6 in Algorithm 5 should be moved to

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3152217, IEEE Access

Algorithm 4 to cater for to the small ternary polynomial. In other words, one does not need to execute lines 5-6 in Algorithm 5 anymore, because the input polynomial does not contain any negative value.

D. EPHEMERAL KEY PAIR

The proposed tensor-core-based polynomial convolution can be more efficient than an integer-based implementation in a GPU. So far, we have only discussed situations that allow the same public/private key pair to be reused for a small number of encryption/decryption. For instance, one can perform K encryptions/decryptions with the same public/private key pair, and refresh the key pair before executing the next K encryption/decryption. In the previous discussion, we assume that $K = N$ to fully exploit the performance gain by using a tensor core that operates on a square matrix. For applications that need to refresh the key pair more frequently (i.e., $K < N$), we can scale the proposed technique accordingly by adjusting K, where $K = 1, 2, ..., N - 1$. By setting $K = 1$, we refresh the key pair for every single encryption/decryption. However, due to the tensor core limitation whereby it only handle a 16×16 matrix, the value of K must be a multiple of 16.

Algorithm 8: NTRU polynomial convolution using tensor core with scalable ephemeral key pair configurations.

Input: polynomial *a* with degree *N* (constant polynomial), *N* polynomial *b* with degree *K* (non-constant polynomials), modulus *q*.

Output: $K \times M$ Matrix *C*, which contains the cyclic convolution of polynomial *a* and many different polynomial *b*.

```
// CPU Phase:
```
1: (Same with Alg. 7)

```
// GPU Phase:

// Calc. total number of warps required
```
2: $warp\_tot = (M/16) \times (K/16)$
3: $tc\_threads = warp\_tot \times 32$
```
// Calc. number of blocks
```
4: $tc\_blocks = tc\_threads / max\_threads$
```
// Limit the number of threads
```
5: $tc\_threads = max\_threads$
6: ParCyc$< N, N >(fp16_A, a)$ ▷ Alg. 4
7: ParU16toFP16$< K, N >(fp16_B, b)$ ▷ Alg. 5
8: TC-PC$< tc_blocks, tc_threads >$ $(fp16\_A, fp16\_B, fp32\_C)$ ▷ Alg. 3
9: ParFP32toU16$< K, M >(C, fp32_C)$ ▷ Alg. 6

Referring to Algorithm 8, the number of warps required to perform tensor-core-base polynomial convolution (TC-PC) is reduced from $(M/16)^2$ in Algorithm 7 to $(M/16) \times (K/16)$ (line 1). Besides, the parallel blocks utilized to compute ParU16toFP16 and ParFP32toU16 are also reduced from N to K. This is because polynomial a is only used to convolute K polynomial b, where $K < N$. With these small changes, the proposed technique can be used for applications that need to refresh the key pair more frequently. Note that Algorithm 8 is less optimal than Algorithm 7 because the tensor core is only used to compute an $M \times K$ matrix instead of $M \times M$.

E. POLYNOMIAL ADDITION

NTRU Encrypt involves polynomial convolution followed by addition to another polynomial ($r * h + e$). Since the tensor core can perform MMA in one cycle, one can also utilize this feature to perform MMA for NTRU Encrypt. We utilized this feature in the NTRU implementation. However, polynomial addition itself is a lightweight operation; a simple parallel implementation using INT32 is already very efficient. Performing the accumulation in tensor core involves type conversion from U16 to FP16, which introduces a small overhead. Hence, the benefit of performing polynomial addition within the tensor core is not significant in this situation.

IV. EVALUATION

This section presents experimental results for the proposed tensor-core-based polynomial convolution and its application to three different lattice-based cryptographic schemes. Results are compared to the reference and AVX2 accelerated implementation found in the NIST PQC standardization submission package [15]. CPU implementations were evaluated on a machine with Intel Core i7-9700F clocked at 4.7 GHz with 16 GB RAM. The GPUs used in this paper are the NVIDIA RTX2060 with 8 GB RAM and RTX3080 with 10GB; both devices are clocked at 1.71 GHz. GPU implementations of NTRU follow closely the NIST submission package [15] and the results are verified against the test vectors provided.

A. PERFORMANCE EVALUATION OF TENSOR-CORE-BASED POLYNOMIAL CONVOLUTION

The first experiment was aimed at demonstrating the superiority of tensor-core-based polynomial convolution (TC-PC) against the conventional implementation using 32-bit integer units (INT32-PC). In INT32-PC implementation, *N* blocks are launched, where each block computes one polynomial convolution in parallel, with *N* threads. To optimize the performance of this implementation, we stored the two polynomials (poly_a and poly_b) in shared memory to reduce the overhead in accessing global memory. For the TC-PC implementation, $(N/16)^2$ warps were launched to complete the matrix multiplication. The performance of both INT32-PC and TC-PC implementations are presented in Table 3. Note that the results reported for GPU implementations are the average time of processing one polynomial convolution (i.e., (total time to process *N* polynomial convolutions)/N). Referring to the implementation results on RTX2060, when

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3152217, IEEE Access

TABLE 3: Performance of tensor-core-based polynomial convolution (Algorithm 7)

N			32	64	128	192	256	384	512	768	1024
Time (μs)*	**RTX2060**	INT32-PC**	0.22	0.15	0.15	0.22	0.41	0.53	0.92	2.06	3.79
		TC-PC	0.27	0.16	0.11	0.12	0.18	0.20	0.33	0.64	1.11
Ratio (INT32-PC / TC-PC)			0.81	0.94	1.36	1.83	2.28	2.65	2.79	3.22	3.41
Time (μs)*	**RTX3080**	INT32-PC**	0.15	0.09	0.06	0.07	0.10	0.21	0.36	0.76	1.27
		TC-PC	0.23	0.12	0.07	0.06	0.06	0.08	0.08	0.17	0.22
Ratio (INT32-PC / TC-PC)			0.65	0.75	0.86	1.17	1.67	2.5	4.5	4.47	5.77

* Average time of N polynomial convolutions.
** INT32: 32-bit integer units, TC: tensor core, PC: polynomial convolution.

the polynomial degree was small ($N \leq 64$), INT32-PC showed better performance than TC-PC. This is because TC-PC requires additional steps in reorganizing poly_a into cyclic form and converting the polynomial elements between integer and floating point formats. However, when N increases beyond 64, the benefit of using tensor core is obvious. The speed-up gained by TC-PC against INT32-PC increases steadily when $64 < N \leq 1024$, where it records the highest speed-up of $3.41\times$ when $N = 1024$. We do not report on the cases beyond 1024, because the speed-up gained does not increase anymore. Note that the implementation results on RTX3080, a similar behaviour can be observed, wherein the speed-up gained by TC-PC increases steadily when $128 < N \leq 1024$.

B. PERFORMANCE EVALUATION UNDER THE EPHEMERAL KEY PAIR SCENARIO

By changing the dimensions of matrix multiplication from $M \times M$ to $M \times K$, one can compute K polynomial convolutions using the proposed tensor core technique, with the same public/private key pair. Due to the current tensor core limitation in NVIDIA GPU that only handles 16×16 matrix, K has to be a multiple of 16. From Table 4, we observed that the proposed TC-PC is more efficient than the conventional integer-based implementation when $K \geq 32$ (RTX2060) or $K \geq 128$ (RTX3080). However, the performance is less efficient compared to the case of computing $M \times M$, and sometimes it is even slower than AVX2. For instance, considering the case where both M and N are 512, the GPU can complete one polynomial convolution in $0.33\mu s$ on average (see Table 3), which is faster than all the $M \times K$ combinations. This is because the same polynomial a is reused for N convolutions against polynomial b, so the overhead of pre- and post-processing are effectively amortized. On the other hand, AVX2 can complete one polynomial convolution in $1.51\mu s$, so it is only beneficial to employ the GPU to perform polynomial convolutions if $K \geq 64$ (RTX2060) or $K \geq 128$ (RTX3080). For cases where GPU does not provide good speed up, it is better to use AVX2 for accelerating the polynomial convolutions. The break-even point where GPU is more advantageous than AVX2 has to be determined through experiments, because the computational capability of each GPU platform differs.

C. PERFORMANCE EVALUATION OF NTRU

To demonstrate the benefit of tensor core in accelerating lattice-based cryptographic schemes, we implemented NTRU public-key encryption and KEM scheme with parameter sets **ntruhps2048509** and **ntruhps2048677** using TC-PC and INT32-PC. In the experiment, 512 blocks are launched, where each block computes one NTRU operation. Results of our GPU implementation are presented in Table 5, where they are compared against the reference and AVX2 implementation in CPU. Note that the AVX2 implementation is heavily optimized for performance. On the other hand, the reference implementation aims at providing a clear description to the underlying operations, so it is not optimized for performance.

Considering the results in RTX2060, for the **ntruhps2048509** parameter set, the TC-PC throughput of implementation was $2.02\times$ and $1.56\times$ higher than INT32-PC for encapsulation and decapsulation respectively. A similar speed-up ratio was also observed for **ntruhps2048677**, wherein the throughput of TC-PC implementation was $1.98\times$ and $1.90\times$ higher than INT32-PC. We observed that TC-PC achieved more than $20\times$ higher throughput compared to the reference implementation for encapsulation and decapsulation; it is also more than $2\times$ higher than the AVX2 implementation. The results in RTX3080 also shows similar speed up for TC-PC against INT32-PC, but it is significantly faster RTX2060 due to higher number of cores available. For instance, considering parameter set **ntruhps2048677**, throughput in RTX3080 is $2.52\times$ and $1.65\times$ higher than RTX2060 for encapsulation and decapsulation respectively.

Note that GPU is a throughput-oriented accelerator that is only useful when there are many operations to be computed. Conversely, the AVX2 implementation is advantageous in improving the latency of a single NTRU operation. In other words, the reported speed-up against AVX2 in Table 5 is the full throughput achievable when there is a sufficient workload (512 encapsulation/decapsulation or encryption/decryption). Under such circumstances, the GPU can be an effective accelerator to assist the computation in the CPU, especially in server environment where CPU cores are usually busy handling many other tasks. With insufficient workload, one can always fallback on the AVX2 implementation or employ the techniques for an ephemeral key pair (see Section 3.3).

Fig. 7 and 8 show the throughput achieved by CPU (using AVX2) and GPU (using TC-PC) at various batch sizes, for two different parameter sizes. In this experiment, each block

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3152217, IEEE Access

TABLE 4: Performance of tensor-core-based polynomial convolution for an $M \times K$ dimension ((Algorithm 8, where both M and N are 512)

K			16	32	64	128	256	384
Time, μs	**CPU**	AVX2	1.51					
	GPU*	INT32-PC**	2.53	2.05	1.30	1.05	0.93	0.91
	RTX2060	TC-PC	3.74	1.88	0.95	0.72	0.54	0.36
Ratio (INT32-PC / TC-PC)			0.68	1.09	1.37	1.46	1.72	2.53
Time, μs	**GPU***	INT32-PC**	1.57	0.75	0.46	0.39	0.37	0.33
	RTX3080	TC-PC	2.47	1.24	0.62	0.31	0.18	0.1
Ratio (INT32-PC / TC-PC)			0.64	0.6	0.74	1.26	2.06	3.3

* Average time of *K* polynomial convolution.
** INT32: 32-bit integer units, TC: tensor core, PC: polynomial convolution.

TABLE 5: Comparing the throughput of TensorTRU against other GPU and CPU implementations.

N	Operation	**CPU**		**GPU RTX2060***			**GPU RTX3080***		
		Throughput, operations per second				Improvement ratio	Throughput, ops per second		Improvement ratio
		Reference	AVX2	INT32-PC	TC-PC	(TC-PC/INT32-PC)	INT32-PC	TC-PC	(TC-PC/INT32-PC)
509	Encrypt	47103	483092	847458	1639344	1.93	2362318	3606853	1.53
	Encapsulation	27878	271739	392157	793651	2.02	1113462	1666389	1.5
	Decrypt	11700	250627	436681	862069	1.97	866974	1515152	1.75
	Decapsulation	7595	240964	323625	505051	1.56	490461	904323	1.70
677	Encrypt	29551	334448	313223	606452	1.94	771642	1325030	1.60
	Encapsulation	19658	237530	248756	492611	1.98	463242	740083	1.60
	Decrypt	7959	152672	197239	398406	2.02	447978	715147	1.72
	Decapsulation	6158	123762	146843	278552	1.90	320051	529241	1.65

* INT32: 32-bit integer units, TC: tensor core, PC: polynomial convolution.

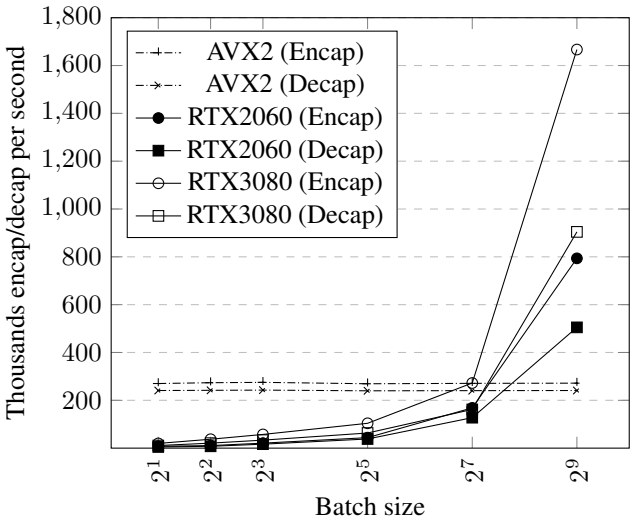


FIGURE 7: Comparing the throughput of AVX2 and GPU implementation of **ntruhps2048509** parameter set with various batch sizes

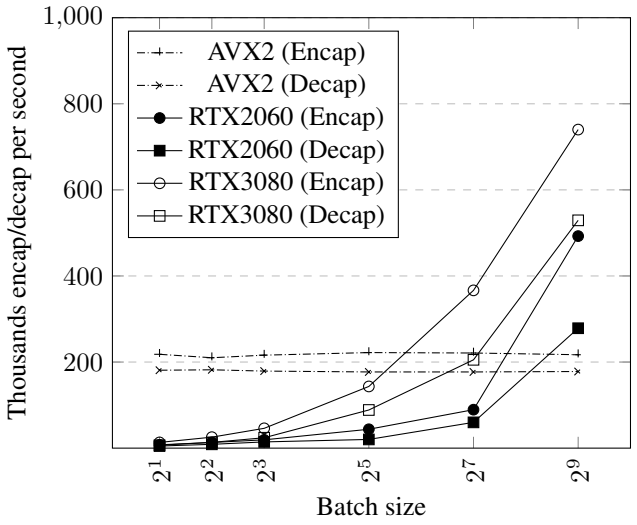


FIGURE 8: Comparing the throughput of AVX2 and GPU implementation of **ntruhps2048677** parameter set with various batch sizes

performs one encapsulation/decapsulation, so the batch size is essentially the number of blocks. When the batch size is small (2 to 32), AVX2 implementation is achieving higher or similar throughput compared to the GPU version. However, GPU can produce higher throughput when batch size increases beyond 128 (approximately). This shows that the proposed TensorTRU can be used in various applications to provide high throughput KEM.

D. COMPARISON WITH OTHER NTRU AND PQC IMPLEMENTATIONS ON A GPU

Existing NTRU implementations on GPU platforms are presented in Table 6. Note that the previous implementations do not follow the NIST NTRU specifications; they targeted different polynomial sizes and GPU devices, which are difficult to benchmark with our work directly. To allow a fair comparison, we scale the results from previous implementation to match our GPU, which is calculated as $\frac{Throughput}{1920/core}$, where

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3152217, IEEE Access

TABLE 6: Comparing the performance of TensorTRU (polynomial convolution) with other existing NTRU implementations on GPU. Throughput is measured as tasks per second.

Implementation	Year	N	GPU	Core	Throughput	Techniques
[7]	2013	251	GTX275	240	332557 / 2660456 *	Sliding window
[10]	2018	401	GTX1080	2560	508541 / 381405 *	Karatsuba
TensorTRU	2021	251 401	RTX2060	1920 + 240 tensor cores	3448275 1408451	Schoolbook

* Performance scaled to match the number of GPU cores in RTX2060 (1920 cores).

TABLE 7: Comparing the performance of TensorTRU KEM with other existing PQC implementations on GPU. Throughput is measured as key exchange per second.

Implementation	Year	GPU	Core	Throughput (KEx/s)	Techniques
NTRU-509 KEM (this work)	2021	RTX2060	1920 + 240 tensor cores	308641	Schoolbook
Kyber KEM [11]	2020	V100	5120	473000 / 177373*	NTT
Kyber KEM [12]	2021	RTX2060	1920 + 240 tensor cores	64200	NTT
NewHope KEM [13]	2021	GTX1650	1024	58337 / 109382*	NTT

* Performance scaled to match the number of GPU cores in RTX2060 (1920 cores).

the number of cores in our RTX2060 was 1920. We also configure the polynomial degree (*N*) in TensorTRU to match the one in previous implementations (i.e., $N = 251$ and $N = 401$.) However, we did not compare with Hermans et al. [5] because it targets the product-form polynomial, which is not used in the NTRU submission to NIST [16].

TensorTRU achieved $1.29\times$ higher throughput than the implementation by Lee et al. [7] for $N = 251$. Note that the sliding window approach requires pre-computation and storage of the polynomial in a look-up table, which can be vulnerable to side channel (timing) attacks. In contrast, the execution of TensorTRU does not depends on a look-up table or any secret information, since it was developed based on the schoolbook convolution. Referring to Algorithm 3, the polynomials are loaded (lines 14 and 15), computed (line 16) and stored (line 17) in a synchronized manner at the warp level. All parallel warps actually perform the same number of operations on different data (see Figure 4), resembling the matrix–matrix multiplication operation. This implies that TensorTRU is constant time and does not have the vulnerabilities found in the work by Lee et al. [7].

Compared to the most recent work by Lee et al. [10], TensorTRU also achieved $9.04\times$ higher throughput. Note that Lee et al. [10] exploited the Karatsuba algorithm to split the polynomials for more efficient computation. Due to the limitation in the GPU architecture (see Section II), a thread can only access the registers in other threads through a warp shuffle instruction. This instruction can only be used within a warp (32 threads), but NTRU polynomials are usually much larger than the warp size. Hence, Lee et al. [10] stored the polynomials in shared memory and accessed them across different parallel threads. This is considered the most efficient way to implement schoolbook and Karatsuba polynomial convolutions, but it still requires access to the shared memory, which is slower than registers. In contrast, TensorTRU stores the polynomials directly into registers without using shared memory. This is made possible by converting a polynomial convolution to matrix multiplication in the tensor core, through the series of algorithms proposed in this paper (Algorithm 3, 4, 5 and 6). Moreover, TensorTRU is executed in the tensor core, which are more optimized than ordinary GPU cores to process matrix multiplication. This justifies the high performance in TensorTRU compared to other NTRU GPU implementations.

Table 7 shows the comparison with existing PQC KEM implementations on GPU. Gupta et al. [11] shows that Kyber KEM can achieve a very high throughput when it is implemented in batch mode, which is essentially a serial implementation. The key exchange throughput achieved by TensorTRU is $1.74\times$ higher than Kyber [11]. Another recent Kyber implementation from Lee et al. [12] also shows high throughput; TensorTRU achieves $4.81\times$ higher throughput on the same GPU platform. Compared to the GPU implementation of another NIST Round-2 PQC candidate [13], NewHope, the proposed TensorTRU achieves $2.82\times$ higher throughput.

E. TENSORLAC: APPLICATION TO LAC

LAC is a cryptosystem based on the poly-LWE variant of the Learning with Errors problem, and was selected as Round 2 candidate in the NIST PQC competition. The modulus of LAC is restricted to $q = 251$, which allows each polynomial element to fit into a single byte [27]. The decoding correctness in LAC relies heavily on the ability of Bose-Chaudhuri-Hocquenghem (BCH) error correction code to recover errors. Even though LAC was not selected to advance to Round 3, it won first prize in the post-quantum cryptography competition hosted by the Chinese Association for Cryptologic Research (CACR). LAC remains an interesting candidate due to its superior implementation performance and simplicity in design.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3152217, IEEE Access

Algorithm 9: Parallel implementation of LAC polynomial convolution using tensor core in a GPU.

Input: polynomial a with degree N (constant polynomial), N polynomial b with degree N (non-constant polynomials), modulus q.

Output: $N \times N$ Matrix C, which contains the nega-cyclic convolution of polynomial a and many different polynomial b.

// **CPU Phase:**

1: $u8cyclic\_A$ ▷ Initialize one matrix in U8

2: $fp32\_C$ ▷ Initialize one matrix in FP32

// **GPU Phase:**

3: $warp\_tot = (N/16)^2$ ▷ Calc. total number of warps required

4: $tc\_threads = warp\_tot \times 32$

5: $tc\_blocks = tc\_threads/max\_threads$ ▷ Calc. number of blocks

6: $tc\_threads = max\_threads$ ▷ Limit the number of threads

7: LACParCyc$< N, N >$ $(u8cyclic\_A, a)$

8: TC-PC $< tc\_blocks, tc\_threads >$ $(u8cyclic\_A, B, fp32\_C)$ ▷ Algorithm 3

9: ParFP32toU8$< N, N >$ $(C, fp32\_C)$

In this paper, we have extended our idea of using tensor core to compute polynomial convolution in LAC. Since LAC uses modulus $q = 251$, one can use Configuration 5 (see Table 2) to implement polynomial convolution in tensor core. The polynomial degrees in LAC are $N = 512$ and $N = 1024$, which appear to be multiples of 16, so it can be computed by Algorithm 7 without padding zeros. However, the polynomial convolution in LAC is of nega-cyclic form, which implies that we cannot use Algorithm 4 to arrange the polynomial a (a constant) into cyclic form. However, this can be resolved easily by converting the relevant elements to a nega-cyclic form in line 4 (replace $in[(tid - bid)\%N]$ by $q - in[(tid - bid)\%N]$).

The implementation of polynomial convolution in LAC is similar to NTRU, and is presented in Algorithm 9. Since polynomial elements in LAC are already represented in 8-bit integer (U8) form, we can use Configuration 5 in tensor core, and no type conversion is required. This reduces one step compared to Algorithm 7. Firstly, one $N \times N$ matrix with U8 and another one $N \times N$ matrix in FP32 are initialize in the CPU. Next, we arrange the polynomial in nega-cyclic form (line 7), followed by matrix multiplication in tensor core (line 8). Finally, the results from tensor core (FP32) are converted to U8 and modulo q (line 9). Note that the last step is similar to Algorithm 6, except that we are converting the results to U8 instead of U16.

Table 8 shows the implementation results of nega-cyclic polynomial convolution of LAC in a CPU (reference and AVX2) and a GPU (integer units and tensor core), respectively. TC-NPC is showed $2.93\times$ and $3.1\times$ higher performance than INT8-NPC, for $N = 512$ and $N = 1024$ respectively. These speed-ups are slightly higher than with TensorTRU, because there is no need to convert the data from INT8 to FP16 as required in NTRU.

F. TENSORFRO: APPLICATION TO FRODOKEM

FrodoKEM was selected as an alternate candidate in the third round of the NIST PQC competition. The official FrodoKEM parameter sets require that modulus $q = 2^{15}$ and $q = 2^{16}$, which are too large to be represented in FP16, so we cannot utilize tensor core to perform the matrix multiplication. However, Frodo allows flexible configuration on its parameters as a trade-off between security level, size of modulus and the probability of decryption failure. One of the interesting parameters was proposed by Bian et al. [37], wherein the modulus can be as small as $q = 2^{11}$. This parameter set allows the server side to perform only matrix-vector multiplication ($N\times$ ñ), but it requires the client side to do much more work ($N\times$ m̃). On the other hand, one can also utilize the parameter searching script provided by the FrodoKEM Round 3 submission [25] to obtain a parameter set with small modulus. In this paper, we instantiated another parameter set for FrodoKEM, which is presented in Table 10. With the restrictions $q = 2048$ and $\sigma = 1.0$, we obtained a parameter set that has a balanced workload between server and client, since m̃ and ñ is close to each other. We show that the proposed tensor core technique can be utilized to accelerate the matrix multiplication in these two variant parameter sets.

Polynomial degree *N* for Frodo-II and TensorFro is not a multiple of 16, so we need to use the proposed method to pad zeros into the polynomial (see Figure 5b). For Frodo-II, the server side can pack many polynomials into a matrix and perform many matrix–vector multiplications using Algorithm 3. The client side can pack two $N\times$m̃ (570×256) matrices and can perform matrix multiplications with tensor core. A similar technique is applicable to TensorFro on both the client and the server side by packing multiple smaller matrices to form a larger one. Note that we do not need to arrange the polynomial in cyclic form, since FrodoKEM does not perform convolutions.

The error vectors in FrodoKEM span a larger distribution compared to ternary values in NTRU and LAC. For instance, Frodo-II and TensorFro have error vector with values in the range {-4, -3, ...0, ..., +3, +4}. When the proposed tensor-core-based technique is used, multiplication between a 11-bit ($q = 2048$) sample and an error vector produces a maximum of 13-bit value in floating point format i.e., $(2^{11}-1)\times 4 \approx 2^{13}$ and $(2^{11}-1\times -4 \approx -2^{13})$. In the process of polynomial convolution, the accumulated value can grow up to a maximum of $N\times(2^{13}-1)$. Hence, for the two variant parameter sets, the values stored in the accumulator can grow to 23-bit (Frodo-II, $log_2(570 \times (2^{13} - 1))$; TensorFro,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3152217, IEEE Access

TABLE 8: Comparing TensorLAC (polynomial convolution) against other GPU and CPU implementations.

Implementation	Operation	CPU		GPU (RTX2060)*			GPU (RTX3080)*		
		Time (μs)				Impr. Ratio	Time (μs)		Impr. Ratio
		Reference	AVX2	INT8-NPC	TC-NPC	INT8-NPC /TC-NPC	INT8-NPC	TC-NPC	INT8-NPC /TC-NPC
LAC-128 ($N = 512$)	Poly. Conv.	55.93	4.35	1.23	0.42	2.93	1.108	0.1913	5.79
LAC-256 ($N = 1024$)	Poly. Conv.	216.74	20.68	3.88	1.25	3.10	0.38	0.06	6.33

* INT8: 8-bit integer units, TC: tensor core, NPC: nega-cyclic polynomial convolution.

TABLE 9: Performance of tensor-core-based matrix multiplication for Frodo variants, implemented on RTX2060 and RTX3080.

Implementation	GPU (RTX2060)*			GPU (RTX3080)*		
	Time (μs)		Improvement ratio	Time (μs)		Improvement ratio
	INT32-MM	TC-MM	INT32-MM/ TC-MM	INT32-MM	TC-MM	INT32-MM/ TC-MM
Frodo-II (server side, 570×570) [37]	1.35	0.44	3.07	0.42	0.17	2.54
Frodo-II (client side, 570×512) [37]	1.41	0.43	3.28	0.42	0.14	3.07
TensorFro (server side, 560×550)	1.39	0.42	3.31	0.43	0.17	2.62
TensorFro (client side, 560×552)	1.39	0.42	3.31	0.46	0.15	3.15

INT32: 32-bit integer units, TC: tensor core, MM: matrix multiplication.

TABLE 10: Parameter instantiations of FrodoKEM.

Implementation	q	σ	N	ñ	m̃	Security level	Error distr.
Frodo-Rec-1 [25]	2^{15}	2.8	640	8	8	141-bit	± 12
Frodo-II [37]	2^{11}	1.0	570	1	256	137-bit	± 4
TensorFro	2^{11}	1.0	560	11	12	136-bit	± 4

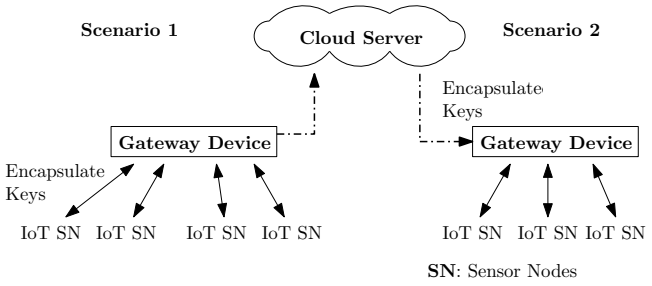


FIGURE 9: Key refreshment in IoT communication systems

$log_2(560 \times (2^{13} - 1)))$. This allows matrix multiplication to be computed correctly within the single precision, so we can utilize the tensor core Configuration 1.

Table 9 shows the results of matrix multiplication for Frodo variant parameter sets implemented on RTX2060. The achieved speed-up between INT32-MM and TC-MM is similar to cases in TensorTRU and TensorLAC.

G. USE CASE: SECURE COMMUNICATION IN IOT APPLICATIONS

IoT applications typically employ symmetric encryption schemes (e.g., AES) to encrypt the IoT data. To reduce the risks of compromising symmetric encryption keys, we need to refresh these encryption keys frequently. Referring to Figure 9, this key refreshment task in a typical IoT application can be carried out in two ways:

1) Scenario 1: The IoT sensor nodes generate new symmetric encryption keys locally and forward them to the gateway device and cloud server via KEM. This process takes place in every communication session, in which pseudorandom number generator (PRNG) can be employed to produce the new symmetric encryption key.
2) Scenario 2: The cloud server generates new symmetric encryption keys and forward them to the gateway device and each sensor node via KEM. Similarly, PRNG can be utilized to generate new keys for every session.

In scenario 1, the gateway device and cloud server need to handle many key decapsulations in real time. On the other hand, in scenario 2, the cloud server needs to generate and encapsulate many keys, so that these keys can be sent to each sensor node in a timely manner. Under the IoT application scenario, a gateway device typically needs to communicate with tens to hundreds of sensor nodes. On the IoT cloud servers, this connection can go up to tens of thousands. With such massive connections in IoT communication, it is clear that both scenarios needs high throughput key encapsulation/decapsulation, which is difficult even for a high-end workstation. To mitigate this challenge, one can offload the KEM to GPU, which is already found in many gateway device (e.g., Jetson AGX Xavier [38]) and cloud server platforms. The proposed tensor-core-based technique can be very useful in handling this kind of massive key encapsulations/decapsulations.

Due to constrained energy storage, IoT sensor nodes usually transmit the collected sensor data in a coordinated session [39], intermittently. Instead of using a separate public-private key pair for each sensor node, the same key pair is used for key encapsulations/decapsulations for a particular session in all sensor node. This implies that performing hundreds to thousands of KEMs using the same public-private key pair for one communication session is common

in IoT applications [12]. In the subsequent sessions, the user can choose to use the same key pair or generate a new one (ephemeral key pair).

V. CONCLUSION

In this paper, we present the first tensor-core-aided cryptography implementation on a GPU. The proposed tensor-core-based polynomial convolution is faster than conventional implementations that rely on integer units in the GPU. Since the proposed tensor-core-based polynomial convolution is a generic algorithm, it can be applied to various sizes of matrix/polynomial. Although the current tensor core can only support limited floating point precision and integer types, we believed the situation may change in near future. In particular, the introduction of FP64 into tensor core recently opened up its adoption into the mainstream scientific computing applications, fostering the use of the GPU in a wider range of applications. As this trend persists, we believe the performance of FP64 tensor core will increase and eventually support more parameters for lattice-base cryptography. On top of that, a recently released embedded GPU (the Jetson AGX Xavier) also offers tensor core to accelerate deep learning inference. This embedded GPU can be used to implement gateway device in IoT applications (e.g., road side units in a smart city), in which our solution can be applied to enable high throughput key encapsulations/decapsulations.

Advanced Matrix Extensions (AMX) is a new x86 instruction set architecture (ISA) released by Intel to support matrix multiplication, which is similar to the tensor core on GPUs. Adapting the proposed tensor-core-based polynomial convolution on such advanced ISA would be an interesting future work that we wish to pursue. On the other hand, a recent work utilized the Strassen's algorithm to speed-up the matrix-multiplication [40]. This can also be an interesting future direction, as the performance of such approach on a GPU is still unknown.

The proposed tensor-core-based polynomial convolution can be implemented on consumer (RTX2060 and RTX3080) and server grade (T4 and A100) GPU platforms with Turing and Ampere architecture, with similar performance gain. However, the tensor core in Volta architecture GPU (e.g., V100) is not as powerful as the one in Turing and Ampere architectures. Hence, we expect that the performance of our solution implemented on Volta architecture GPU may not be as impressive compared these two architectures.

REFERENCES

[1] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM review*, vol. 41, no. 2, pp. 303–332, 1999.

[2] D. J. Bernstein, "Introduction to post-quantum cryptography," in *Post-quantum cryptography*. Springer, 2009, pp. 1–14.

[3] G. Alagic, G. Alagic, J. Alperin-Sheriff, D. Apon, D. Cooper, Q. Dang, Y.-K. Liu, C. Miller, D. Moody, R. Peralta *et al.*, *Status report on the first round of the NIST post-quantum cryptography standardization process*. US Department of Commerce, National Institute of Standards and Technology, 2019.

[4] A. Khalid, S. Bian, C. Wang, M. O'Neill, W. Liu *et al.*, "Axrlwe: A multi-level approximate ring-lwe co-processor for lightweight iot applications," *IEEE Internet of Things Journal*, 2021.

[5] J. Hermans, F. Vercauteren, and B. Preneel, "Speed records for NTRU," in *Cryptographers' Track at the RSA Conference*. Springer, 2010, pp. 73–88.

[6] A. A. Kamal and A. M. Youssef, "Enhanced implementation of the NTRUEncrypt algorithm using graphics cards," in *2010 First International Conference On Parallel, Distributed and Grid Computing (PDGC 2010)*. IEEE, 2010, pp. 168–174.

[7] M.-K. Lee, J. W. Kim, J. E. Song, and K. Park, "Efficient implementation of NTRU cryptosystem using sliding window methods," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. 96, no. 1, pp. 206–214, 2013.

[8] S. Akleylek and Z. Y. Tok, "Efficient interleaved Montgomery modular multiplication for lattice-based cryptography," *IEICE Electronics Express*, vol. 11, no. 22, pp. 20 140 960–20 140 960, 2014.

[9] 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.

[10] W.-K. Lee, B.-M. Goi, W.-S. Yap, D. C.-K. Wong, and S. Akleylek, "Fast NTRU encryption in GPU for secure IoP communication in post-quantum era," in *2018 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*. IEEE, 2018, pp. 1923–1928.

[11] 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.

[12] 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*, 2021.

[13] 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.

[14] 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.

[15] C. S. Division, "Round 3 submissions - post-quantum cryptography: Csrc." [Online]. Available: https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions

[16] 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," 2020.

[17] C.-c. Lin, M.-h. Sheu, C. Liaw, and H.-k. Chiang, "Fast first-order polynomials convolution interpolation for real-time digital image reconstruction," *IEEE transactions on circuits and systems for video technology*, vol. 20, no. 9, pp. 1260–1264, 2010.

[18] Y. Zeng, L. Zhang, J. Zhao, J. Lan, and B. Li, "Jrl-yolo: A novel jump-join repetitious learning structure for real-time dangerous object detection," *Computational Intelligence and Neuroscience*, vol. 2021, 2021.

[19] Q. Ding, A. Rehman Sheikh, W. Pan, X. Gu, N. Sun, X. Su, L. Luo, H. Ma, R. He, and T. Zhang, "In situ monitoring of grape seed protein hydrolysis by raman spectroscopy," *Journal of Food Biochemistry*, vol. 45, no. 4, p. e13646, 2021.

[20] X. Zhang, J. Saniie, and A. Heifetz, "Spatial temporal denoised thermal source separation in images of compact pulsed thermography system for qualification of additively manufactured metals," in *2021 IEEE International Conference on Electro Information Technology (EIT)*. IEEE, 2021, pp. 209–214.

[21] 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.

[22] J.-P. D'Anvers, A. Karmakar, S. S. Roy, and F. Vercauteren, "SABER: Module-LWR based key exchange, CPA-secure encryption and CCA-secure KEM," in *International Conference on Cryptology in Africa*. Springer, 2018, pp. 282–305.

[23] L. Ducas, T. Lepoint, V. Lyubashevsky, P. Schwabe, G. Seiler, and D. Stehlé, "CRYSTALS-DILITHIUM: Digital signatures from module lattices," 2018.

[24] P.-A. Fouque, J. Hoffstein, P. Kirchner, V. Lyubashevsky, T. Pornin, T. Prest, T. Ricosset, G. Seiler, W. Whyte, and Z. Zhang, "FALCON: Fast-fourier lattice-based compact signatures over NTRU," *Submission to the NIST's post-quantum cryptography standardization process*, 2018.

[25] E. Alkim, J. W. Bos, L. Ducas, P. Longa, I. Mironov, M. Naehrig, V. Nikolaenko, C. Peikert, A. Raghunathan, and D. Stebila, "FrodoKEM learning with errors key encapsulation," 2020.

[26] D. J. Bernstein, C. Chuengsatiansup, T. Lange, and C. Van Vredendaal, "NTRU Prime." *IACR Cryptol. ePrint Arch.*, vol. 2016, p. 461, 2016.

[27] X. Lu, Y. Liu, Z. Zhang, D. Jia, H. Xue, J. He, B. Li, K. Wang, Z. Liu, and H. Yang, "LAC: Practical Ring-LWE based public-key encryption with byte-level modulus." *IACR Cryptol. ePrint Arch.*, vol. 2018, p. 1009, 2018.

[28] C. NVIDIA, "CUDA C programming guide, version 11.2," *NVIDIA Corp*, 2020.

[29] J. Hoffstein, J. Pipher, and J. H. Silverman, "NTRU: A ring-based public key cryptosystem," in *International Algorithmic Number Theory Symposium*. Springer, 1998, pp. 267–288.

[30] J. Hoffstein, J. Pipher, J. M. Schanck, J. H. Silverman, W. Whyte, and Z. Zhang, "Choosing parameters for ntruencrypt," in *Topics in Cryptology - CT-RSA 2017 - The Cryptographers' Track at the RSA Conference 2017, San Francisco, CA, USA, February 14-17, 2017, Proceedings*, ser. Lecture Notes in Computer Science, H. Handschuh, Ed., vol. 10159. Springer, 2017, pp. 3–18. [Online]. Available: https://doi.org/10.1007/978-3-319-52153-4_1

[31] A. Hülsing, J. Rijneveld, J. M. Schanck, and P. Schwabe, "High-speed key encapsulation from NTRU," in *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, ser. Lecture Notes in Computer Science, W. Fischer and N. Homma, Eds., vol. 10529. Springer, 2017, pp. 232–252. [Online]. Available: https://doi.org/10.1007/978-3-319-66787-4_12

[32] J. Hoffstein, J. Pipher, J. M. Schanck, J. H. Silverman, and W. Whyte, "Transcript secure signatures based on modular lattices," in *Post-Quantum Cryptography - 6th International Workshop, PQCrypto 2014, Waterloo, ON, Canada, October 1-3, 2014. Proceedings*, ser. Lecture Notes in Computer Science, M. Mosca, Ed., vol. 8772. Springer, 2014, pp. 142–159. [Online]. Available: https://doi.org/10.1007/978-3-319-11659-4_9

[33] D. Das, J. Hoffstein, J. Pipher, W. Whyte, and Z. Zhang, "Modular lattice signatures, revisited," *Des. Codes Cryptogr.*, vol. 88, no. 3, pp. 505–532, 2020. [Online]. Available: https://doi.org/10.1007/s10623-019-00694-x

[34] W.-K. Lee, S. Akleylek, W.-S. Yap, and B.-M. Goi, "Accelerating number theoretic transform in gpu platform for qtesla scheme," in *International Conference on Information Security Practice and Experience*. Springer, 2019, pp. 41–55.

[35] W.-K. Lee, S. Akleylek, D. C.-K. Wong, W.-S. Yap, B.-M. Goi, and S.-O. Hwang, "Parallel implementation of nussbaumer algorithm and number theoretic transform on a gpu platform: application to qtesla," *The Journal of Supercomputing*, vol. 77, no. 4, pp. 3289–3314, 2021.

[36] H. Satılmış, S. Akleylek, and C.-C. Lee, "Efficient implementations of sieving and enumeration algorithms for lattice-based cryptography," *Mathematics*, vol. 9, no. 14, p. 1618, 2021.

[37] S. Bian, M. Hiromoto, and T. Sato, "Filianore: Better multiplier architectures for LWE-based post-quantum key exchange," in *Proceedings of the 56th Annual Design Automation Conference 2019*, 2019, pp. 1–6.

[38] CUDA, "Nvidia agx xavier module." [Online]. Available: https://developer.nvidia.com/embedded/jetson-agx-xavier

[39] S.-Y. Liew, C.-K. Tan, M.-L. Gan, and H. G. Goh, "A fast, adaptive, and energy-efficient data collection protocol in multi-channel-multi-path wireless sensor networks," *IEEE Computational Intelligence Magazine*, vol. 13, no. 1, pp. 30–40, 2018.

[40] J. W. Bos, M. Ofner, J. Renes, T. Schneider, and C. v. Vredendaal, "The matrix reloaded: Multiplication strategies in frodokem," in *International Conference on Cryptology and Network Security*. Springer, 2021, pp. 72–91.

WAI-KONG LEE received the B.Eng. degree in electronics and the M.Sc. degree from Multimedia University in 2006 and 2009, respectively, and the Ph.D. degree 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 R&D engineer. His research interests are in the areas of cryptography, numerical algorithms, GPU computing, Internet of Things, and energy harvesting. He is currently a post-doctoral researcher in Gachon University, South Korea

HWAJEONG SEO received the B.S., M.S. and Ph.D degrees in Computer Engineering at Pusan National University. He is currently an assistant professor in Hansung university. His research interests include cryptographic engineering.

ZHENFEI ZHANG received the PhD. degree from the University of Wollongong, Australia, in 2014. He was the director of cryptographic research with OnBoard Security, a company that developed NTRU and the related technologies. He was also the CTO and co-founder of Manta Network. Currently, he serves as a cryptographer in Ethereum Foundation. His main research interests include quantum-safe cryptography, specifically, and lattice-based cryptography.

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