# Tensor Crypto

Wai-Kong Lee[1], Hwa-Jeong Seo[2], Zhenfei Zhang[3] and Seongoun Hwang[1]

[1] Gachon University, Seongnam, South Korea,
waikong.lee@gmail.com,waikonglee@gachon.ac.kr,sohwang@gachon.ac.kr
[2] Hansung University, Seoul, South Korea, hwajeong84@gmail.com
[3] Ant Group, Hangzhou, China, zhenfei.zhang@hotmail.com

**Abstract.** Tensor core is a specially designed hardware included in new NVIDIA GPU chips, aimed at accelerating deep learning applications. With the introduction of tensor core, the matrix multiplication at low precision can be computed much faster than using conventional integer and floating point units in NVIDIA GPU. In the past, applications of tensor core were mainly restricted to machine learning and mixed precision scientific computing. In this paper, we show that for the first time, tensor core can be used to accelerate state-of-the-art lattice-based cryptosystems. In particular, we employed tensor core to accelerate **NTRU**, one of the finalists in NIST post-quantum standardization. Towards our aim, several parallel algorithms are proposed to allow the tensor core to handle flexible matrix sizes and ephemeral key pair. Experimental results show that the polynomial convolution using tensor core is 2.79× (**ntruhps2048509**) and 2.72× (**ntruhps2048677**) faster than the version implemented with conventional integer units of NVIDIA GPU. The proposed tensor core based polynomial convolution technique was applied to **NTRU** public key scheme (**TensorTRU**). It achieved 1.94×/1.95× (encryption) and 1.97×/2.02× (decryption) better performance for the two parameter sets, compared to the conventional integer based implementations in GPU. **TensorTRU** is also more than 20× faster than the reference implementation in CPU and 2× faster than the AVX2 implementation, for both encryption and decryption. To demonstrate the flexibility of the proposed technique, we have extended the implementation to other lattice-based cryptosystems that have a small modulus (**LAC** and two variant parameter sets in **FrodoKEM**). Experimental results show that the tensor core based polynomial convolution is flexible and useful in accelerating lattice-based cryptosystems that cannot utilize number theoretic transform in performing polynomial multiplication.

**Keywords:** Tensor Core · Graphics Processing Unit · Post-quantum Cryptography

## 1 Introduction

The security of traditional Public Key Cryptography (PKC), such as **RSA** and **ECC**, relies on one of the three hard mathematical problems: integer factorization, discrete logarithm, or the elliptic-curve discrete logarithm problem. These hard problems can be easily solved on a sufficiently powerful quantum computer with Shor's algorithm [Sho99, Ber09]. This creates the need of post-quantum PKC algorithms that can resist the threat from quantum computers in near future.

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 [AAAS+19]. Post-quantum candidates need to specify digital signature, public-key encryption, and key-establishment algorithms. The evaluation criteria not only focuses on the security aspects of an algorithm, but also looks into its implementation performance. From the security aspects, the algorithm should be secure against both classical and

quantum attacks. On the other hand, the performance aspects are also important, wherein 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 NIST post-quantum competition for consideration. Out of these candidates, seven finalists and eight alternate candidates were selected into third round, according to the announcement made by NIST on July 2020. These selected finalists will continue to be reviewed for consideration in standardization at the conclusion of the third round. NIST also noted that alternate candidates may still potentially be standardized after third round.

In the third round, five and two lattice-based cryptography algorithms were selected as finalists (i.e., **KYBER**, **NTRU**, **SABER**, **DILITHIUM**, and **FALCON**) and alternate candidates (i.e., **FrodoKEM** and **NTRU Prime**), respectively. Compared with other post-quantum cryptography candidates (e.g. multivariate, hash, code, and isogeny), lattice-based cryptography maintains a majority share in third round.

In order to evaluate the practicality of cryptographic algorithms, many works devoted to improve the implementation performance on various platforms, such as microcontrollers [KRSS19] and massively parallel processors (GPU) [BS10]. For the case of GPU, the first implementation of **NTRU** was presented in 2010 [HVP10]. The work showed that GPU can achieve very high encryption and decryption throughput by utilizing the product form polynomial and some bit-packing techniques. Many works have been proposed to accelerate the performance of **NTRU** on GPU [KY10, LKSP13, AT14, DSS+16, LGY+18]. However, previous works paid little attention on a power of new GPU tensor core, which would be a better choice than ordinary GPU instruction set (i.e. integer/floating point units). Tensor core is a specialized unit released by NVIDIA in its' latest GPU architectures (i.e. Volta, Turing and Ampere) [MDCL+18]. Many deep neural network applications take advantages of NVIDIA tensor core to improve the training and inference performance. However, it is unclear how cryptography implementations can exploit tensor core.

In this paper, we study how to exploit tensor core instructions to provide faster polynomial convolution for post-quantum cryptography implementations. Our main contributions are summarized below:

1. For the first time, a tensor core based polynomial convolution is presented. The proposed technique can handle polynomials with a degree in multiple of 16, which shows 3.41× faster performance compared to conventional implementation using 32-bit integer units in GPU, for polynomial degree $N = 1024$.

2. The first **NTRU** [CDH+20] implementation based on tensor core was proposed in this paper. Since polynomials 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, were proposed to achieve this, resulting a high performance **NTRU** implementation in GPU. The tensor core based **ntruhps2048509** can achieve encryption and decryption in $0.61\mu s$ and $1.16\mu s$ respectively, which are 3.39×/1.94× and 3.44×/1.97× faster than AVX2 in CPU/integer units in GPU implementation, respectively.

3. The proposed tensor core based polynomial convolution can handle various polynomial sizes. To validate this point, we have applied 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 a variant of **FrodoKEM** outperform integer units based implementations by 3.10× and 3.31×, respectively. Detailed steps to efficiently utilizing the proposed technique for polynomial/matrix multiplication in these two schemes are described in this paper.

4. The source code of tensor core based polynomial convolution is released in the public domain at https://github.com/benlwk/Tensorcrypto. This allows researchers

to easily re-produce our results on their development environments and utilize the tensor-core-aided lattice-based cryptography implementation for their own purposes.

We conclude the criteria of applying our technique over other lattice-based cryptography. At a high level, our 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** already use NTT-based multiplications, for which our technique cannot accelerate. Therefore, we restrict the scope of our paper to the schemes where NTT is slow or not applicable.

The remainder of this paper is organized as follows. In Section 2, we introduce related works. In Section 3, 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 4. Finally, we conclude the paper in Section 5.

# 2 Related Works

## 2.1 Overview of GPU Architecture and CUDA Programming Model

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

CUDA is the Software Development Kit (SDK) introduced by NVIDIA to facilitate the use of GPU in general purpose computing. It allows programmers to implement generic algorithms (other than graphics) in GPU with high level programming languages (e.g. C/C++, FORTRAN) as well as pseudo-assembly language (i.e. PTX). From the programming perspective, many parallel threads form a block and multiple blocks form a grid. This allows the flexible arrangement of software threads into the physical SM and cores across many different GPU architectures. The relationship between grid, blocks, and threads is illustrated in Figure 1.

NVIDIA GPU groups 32 threads into a *warp*, wherein all 32 threads execute the same instruction in parallel. Due to this reason, the number of threads per block is usually set as a multiple of 32 to avoid divergence in the instruction execution path. Besides that, the 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 in the warp level to maximize the efficiency of the GPU warp scheduler.

## 2.2 Tensor Core

In 2017, NVIDIA released Volta GPU architecture, which introduced a specialized unit named as tensor core. This newly introduced tensor core is used to perform one matrix-multiply-and-accumulate (MMA) on a $4 \times 4$ matrices per clock cycle [MDCL+18]. Later on, NVIDIA released Turing architecture that supports MMA for $16 \times 16$ matrices with single and half precision floating point. Recently, Ampere architecture that supports

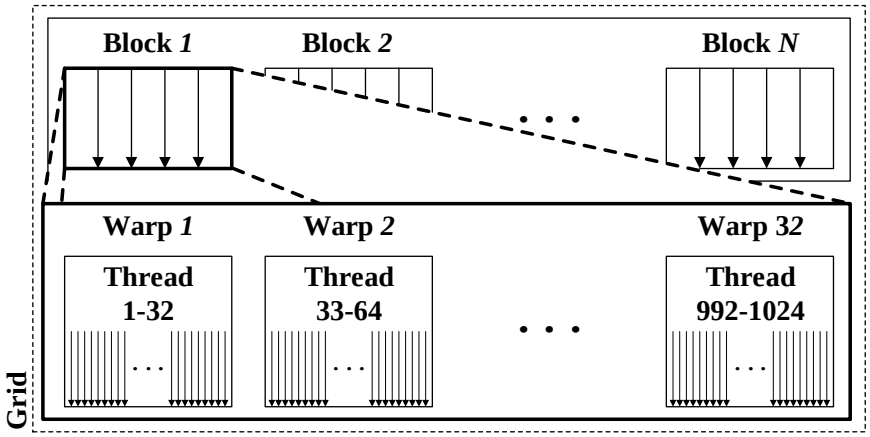



Figure 1: Relationship between grid, blocks and threads in CUDA.

double precision MMA was released, enabling the use of tensor core in generic scientific computing applications. The latest tensor core in Ampere architecture also support new formats (i.e. TensorFloat-32 (TF32) and Bfloat16 (BF16)) that reduces the floating point precision but maintains the same range.

Many deep neural network applications take advantage of the NVIDIA tensor cores. However, it is unclear how cryptography implementations can exploit this newly introduced tensor cores. In this work, we present implementation techniques to use the tensor cores for computing polynomial convolution in lattice-based cryptography.

## 2.3 Lattice-based Cryptography

Lattice-based cryptographic constructions are based on the hardness of Shortest Vector Problem (SVP) which is approximating the minimal Euclidian length of a lattice vector. Lattice-based cryptography is believed to be secure against both conventional and quantum computers. Furthermore, lattice-based cryptography shows fast execution timing for key encapsulation mechanism and signature generation. In the third round of NIST post-quantum cryptography standardization process, five lattice-based cryptography algorithms were selected as finalists (e.g. **CRYSTALS-KYBER**, **NTRU**, **SABER**, **CRYSTALS-DILITHIUM**, and **FALCON**) and another two are selected as alternate candidates (e.g. **FrodoKEM** and **NTRU Prime**), respectively (See Table 1). Due to its good features in security and efficiency, most of the selected candidates in third round are lattice-based cryptography. Many lattice-based schemes rely on polynomial multiplication, which has high computational complexity. In order to improve the performance of polynomial multiplication, we utilized the tensor core and show performance enhancements on lattice-based cryptography with small modulus, such as **NTRU**, **LAC**, and two variant parameter sets of **FrodoKEM**.

### 2.3.1 NTRU

**NTRU** encryption is a lattice-based one-way CPA-secure (OW-CPA) public-key encryption scheme which was invented around 1996 [HPS98]. The security of **NTRU** encryption has been reasonably-well understood and scrutinized for decades.

At 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 **NTRU** assumption says that given $h$, one cannot recover $g$ and $f$, or to distinguish $h$ from a random element over the ring.

Table 1: Comparison of lattice based cryptography in NIST PQC competition, PKE, KEMs, and DS represent Public Key Encryption, Key Encapsulation Mechanisms, and Digital Signature, respectively.

| Lattice-based candidates | Application | Category | Prime ($q$) | NIST PQC competition |
|---|---|---|---|---|
| **CRYSTALS-KYBER** [BDK+18] | PKE/KEMs | Module | 3329 | Round 3 finalists |
| **SABER** [DKRV18] | | | $2^{12}$ | |
| **NTRU** [CDH+20] | | Ideal | $2^{11}, 2^{12}$, and $2^{13}$ | |
| **CRYSTALS-DILITHIUM** [DLL+18] | DS | Module | $2^{23} - 2^{13} + 1$ | |
| **FALCON** [FHK+18] | | Ideal | 12289 | |
| **FrodoKEM** [ABD+20] | PKE/KEMs | Standard | $2^{15}$ and $2^{16}$ | Alternate candidates |
| **NTRU Prime** [BCLVV16] | | Ideal | 4591, 4621, ..., 7879 | |
| **LAC** [LLZ+18] | PKE/KEMs | Ideal | 251 | Round 2 candidate |

Table 2: Comparison of GPU computation modules used for implementing **NTRU**.

| [HVP10] | [KY10] | [LKSP13] | [AT14] | [DSS+16] | [LGY+18] | This work |
|---|---|---|---|---|---|---|
| Integer units | | | | | | Tensor core |

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 $cf = prg + mf \equiv mf \bmod p$ . Since $f$ is invertible modulo $p$, one can extract $m$ from $mf$ with $f^{-1} \bmod p$.

In the NIST PQC competition, there has been two flavors of the **NTRU**, differs in the choice of the ring. The original **NTRU** scheme, known as **NTRU-HPS** [HPS98, HPS+17], 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** [HRSS17], 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. For the purpose of this paper, we do not go in deep details of the scheme. We note, however, that the major computation bottleneck in both schemes is the polynomial multiplication over $\mathcal{R}_q := \mathbb{Z}_q[x]/(x^N - 1)$, which is essentially a polynomial convolution.

## 2.4 Previous NTRU Implementations on GPU

The first implementation of **NTRU** in GPU can be dated back to 2010. Hermans et al. [HVP10] showed that GPU can achieve very high encryption and decryption throughput by utilizing the product form polynomial and bit-packing techniques. Product form polynomial is no longer used in the NTRU submission to NIST. Following up this work, Lee et al. [LKSP13] proposed a sliding window technique to pre-compute some repeating patterns in **NTRU** polynomial and stored them into 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 side channel attack, as the look up table leaks timing information. The **NTRU** modular lattice signature (**NTRU-MLS**) scheme [HPS+14, DHP+20], which requires operations on large vectors, was optimized with parallel polynomial multiplication on GPU by Dai et al. [DSS+16]. Recently, Lee et al. [LGY+18] proposed to utilize Karatsuba algorithm to speed up the polynomial multiplication in **NTRU**. The flat form (schoolbook) and Karatsutba version are constant time and fast, but they still suffer from intensive access to the global memory [Kar63, Too63, CA69]. Unlike previous **NTRU** implementations on GPU, we introduce the first tensor core based **NTRU** implementations on GPU (See Table 2).

**Algorithm 1:** Schoolbook polynomial convolution.

**Input:** Polynomial $a$ with degree $N$, Polynomial $b$ with degree $N$.
**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: $\quad c[k] = 0$

3: $\quad$ **for** $i$ from 0 to $k+1$ **do**
4: $\quad\quad c[k] = c[k] + a[k-i] \times b[i]$

5: $\quad$ **for** $i$ from 1 to $N-k$ **do**
6: $\quad\quad c[k] = c[k] + a[k+i] \times b[N-i]$

7: **return** $c$

# 3 Optimized Implementation of NTRU

## 3.1 Polynomial Convolution Through Tensor Core

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 by schoolbook multiplication, wherein the operation exhibits 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 $k$ element following ordinary schoolbook multiplication. Next, it proceeds with the remaining polynomial 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 in GPU platform to achieve good performance. This technique was previously explored by Dai et al. [DSS+16] and it remains the most efficient way to compute polynomial convolution in GPU. Note that for **NTRU**, the polynomial convolution is performed with 32-bit integer units (INT32). It is also possible to compute **NTRU** polynomial convolution using single precision floating point units (FP32) by converting the polynomial elements to FP32. However, FP32 has the same throughput with INT32 across many generations of GPU architecture [NVI20]. The additional cost of type conversion back and forth introduces non-negligible overhead, which does not make FP32 an attractive choice compared to INT32.

Algorithm 2 shows the parallel version of schoolbook polynomial convolution that can be implemented efficiently in GPU. This implementation utilizes $P$ blocks to perform $P$ polynomial convolution, where each block computes one polynomial convolution with $N$ threads. Polynomials are first loaded from the global memory and cached at the shared memory to reduce the read/write latency (lines 3-5). Next, each thread is responsible in accumulating one column independently (lines 7-10), with the intermediate results stored in a register (i.e. *sum*). Finally, results are copied to the array $c$ which resides in the 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 the global memory (two reads and one write operations), with majority of the operations resides in shared memory and registers.

**Algorithm 2:** Parallel schoolbook polynomial convolution in **NTRU**.

**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 $b$.

1: $tid$=thread ID
2: $bid$=block ID

`// Copy polynomials into shared memory in parallel`
3: shared_a[$tid$]= a[$bid \times N + tid$]
4: shared_b[$tid$]= b[$bid \times N + tid$]
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 $tid$+1 **do**
8: $sum = sum$+shared_a[$tid - i$] $\times$ shared_b[$i$]

9: **for** $i$ from 1 to $N - tid$ **do**
10: $sum = sum$+shared_a[$tid + i$] $\times$ shared_b[$N - i$]

11: **return** c[$bid \times N + tid$] = $sum\%q$

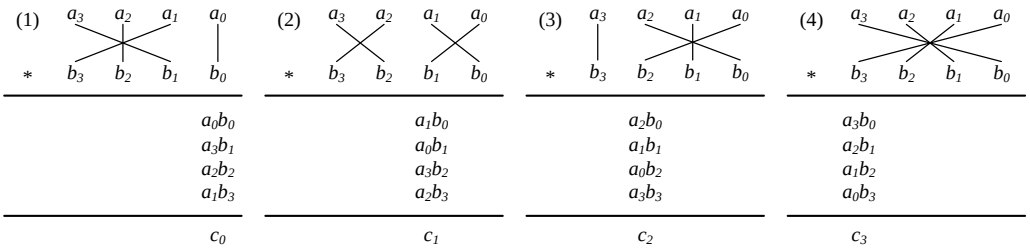



Figure 2: Parallel computation of polynomial convolution with integer units in GPU; operations from (1) to (4) are performed, independently.

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

Tensor core was introduced into GPU to accelerate MMA operations with much higher throughput. By taking a closer look into Algorithm 1, we found 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 $\boldsymbol{A}$ and $\boldsymbol{B}$ produces the same results as in 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 for the case where polynomial $a$ can be reused repeatedly. This is not a problem for encryption in **NTRU** that performs $r * h$, where $h$ is the public key and

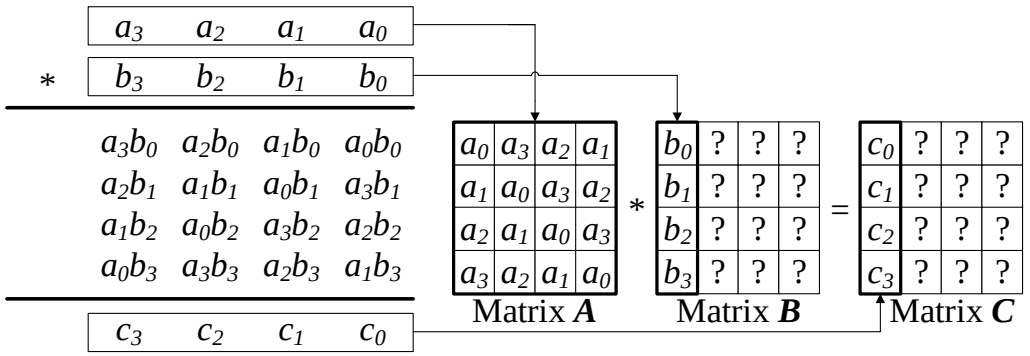



Figure 3: Computing polynomial convolution using tensor core in GPU: Matrix $A$, $B$, and $C$ represent constant polynomial (e.g. public key $h$), non-constant polynomial (e.g. random vectors, $r$), and result, respectively.

Table 3: Supported precision in tensor core.

| Configuration | Matrix $\boldsymbol{A}$ | Matrix $\boldsymbol{B}$ | Accumulator | Dimension |
|---|---|---|---|---|
| 1 | half (FP16) | half (FP16) | single (FP32) | $16 \times 16 \times 16$ |
| 2 | half (FP16) | half (FP16) | half (FP16) | $16 \times 16 \times 16$ |
| 3 | double (FP64) | double (FP64) | double (FP64) | $8 \times 8 \times 4$ |
| 4 | unsigned char (INT8) | unsigned char (INT8) | integer (INT32) | $16 \times 16 \times 16$ |
| 5 | signed char (INT8) | signed char (INT8) | integer (INT32) | $16 \times 16 \times 16$ |

$r$ 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 contrary, polynomial $b$ does not need to be reused, so we can pack many random vectors $r$ into matrix $\boldsymbol{B}$.

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

## 3.2 TensorTRU: NTRU Implementation Based on Tensor Core

### 3.2.1 Representing Polynomial in Floating Point

Referring to Table 1, **NTRU** requires the 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 3. Since 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 integer to floating point, and then utilize one of the three possible configurations (configurations 1-3). Since configurations 1 and 2 are having 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 the 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 of 11-bit, so each pair of multiplication between poly_a and poly_b produces a number with 22-bit maximum. However, one of the polynomial in **NTRU** is ternary (i.e. elements are only consists of -1, 0 and 1). Since we are using

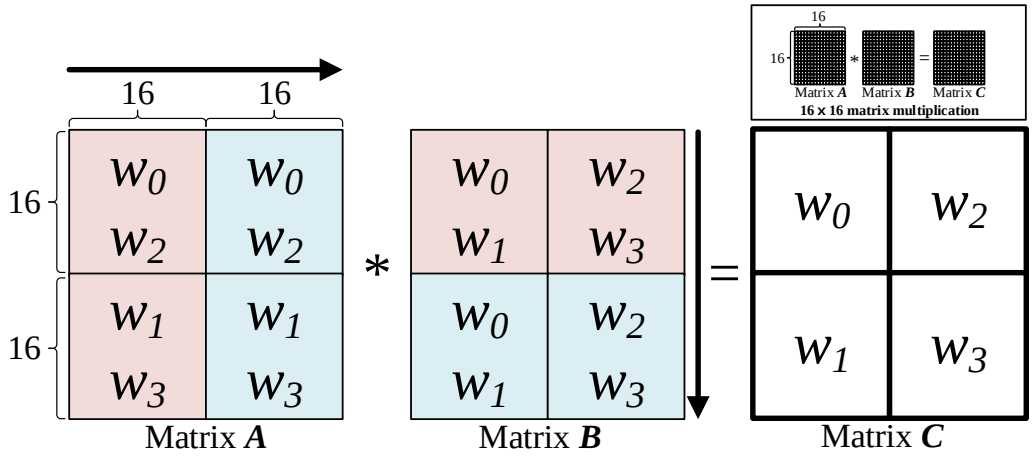



Figure 4: Matrix multiplication: $32 \times 32$ dimension, $w$: warps running in parallel, arrow indicates the computation order.

floating point to represent the polynomial elements, the multiplication produces only 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 ($log_2 509 \times log_2(2^{11}-1)$) and 21-bit ($log_2 677 \times log_2(2^{11}-1)$) data for **ntruhps2048509** and **ntruhps2048677**, respectively. Due to this restriction, we have utilized configuration 1 in accelerating **NTRU** polynomial convolution, because the single precision accumulator can hold an integer value of 24-bit maximum. Note that in practice, the accumulated values may well below 20-bit, because the accumulation can goes both directions (addition or subtraction) depending on the ternary polynomials.

Another two **NTRU** parameters (**ntruhps4096821** and **ntruhrss701**) can also 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 it only supports a smaller matrix size ($8 \times 8$). A faster FP64 tensor core released in future may open up opportunities to apply our technique into these two parameter sets.

### 3.2.2 Parallel Polynomial Convolution using Tensor Core

Tensor core was developed to handle a small matrix with $16 \times 16$ dimension as depicted in right upper part of Figure 4, within a warp (32 threads). To handle a larger matrix, one can utilize many warps computing different parts of the matrix, and then accumulate the result, iteratively. Referring to Figure 4, there are three steps to complete when we perform matrix multiplication for a $32 \times 32$ matrix. Firstly, four warps are launched in parallel to compute matrix multiplication on $16 \times 16$ dimension. For instance, $w_0$ and $w_2$ read the same piece of data ($16 \times 16$) from Matrix $\boldsymbol{A}$, but they read different data from Matrix $\boldsymbol{B}$ 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 $32 \times 32$ matrix multiplication. Lastly, results are stored into Matrix $\boldsymbol{C}$ in parallel at different memory locations. In general, 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 are utilized to compute polynomial convolution in **NTRU**.

Referring to Algorithm 3, the tensor core based polynomial convolution requires the input matrices to be in multiple of $16 \times 16$. Matrix $\boldsymbol{A}$ is the constant polynomial $a$ organized

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

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

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

`// Initialize fragment a and b with` $16 \times 16$ `dimension and FP16 precision`

1: fragment<matrix_a, 16, 16, 16, half, row_major> a_frag
2: fragment<matrix_b, 16, 16, 16, half, col_major> b_frag

`// Initialize fragment c with` $16 \times 16$ `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: $warpID = \lfloor (bid \times blockDim + tid)/32 \rfloor$ ▷ `32 threads per warp`
8: $row\_idx = warpID \% \lfloor M/16 \rfloor \times 16$
9: $col\_idx = warpID \lfloor M/16 \rfloor \times 16$
10: $store\_idx = row\_idx + col\_idx \times M$

11: **for** i from 0 to $\lfloor M/16 \rfloor$ **do**
12: $ldA = row\_idx \times M + i \times 16$
13: $ldB = col\_idx \times M + i \times 16$

`// Load` $16 \times 16$ `sub-matrix from Matrix` ***A*** `and` ***B***

14: load_matrix_sync(a_frag, $\boldsymbol{A} + ldA$, $M$)
15: load_matrix_sync(b_frag, $\boldsymbol{B} + ldA$, $M$)

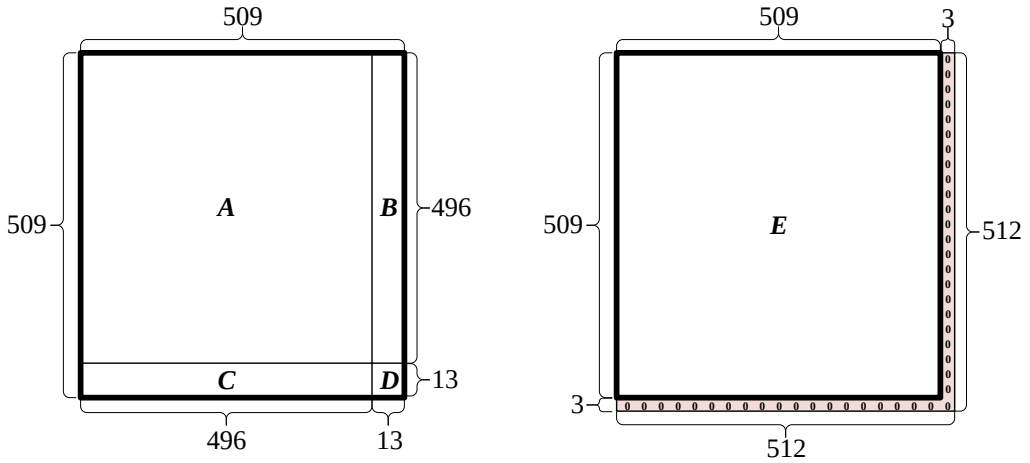
`// Perform matrix multiplication and accumulate the results in c_frag`

16: mma_sync(c_frag, a_frag, b_frag, c_frag)

`// Store the results from c_grat into Matrix` ***C***

17: store_matrix_sync($\boldsymbol{C} + store\_idx$, c_frag, $M$, col_major)

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 1-dimensional memory array (i.e. global memory). The algorithm first initializes two fragments to hold the $16 \times 16$ sub-matrices and one fragment to hold the accumulated results (lines 1-3). Next, it loops through Matrix $\boldsymbol{A}$ (row major) and Matrix $\boldsymbol{B}$ (column major) to perform the matrix multiplication in parallel (lines 11-16). For each iteration, $16 \times 16$ sub-matrices are loaded from Matrix $\boldsymbol{A}$ and Matrix $\boldsymbol{B}$ (in global memory) to perform matrix multiplication in parallel. $(M/16)^2$ warps are executing in parallel, with each warp operates on different parts of Matrix $\boldsymbol{A}$ and Matrix $\boldsymbol{B}$ as depicted in Figure 4. Finally, the accumulated results are copied from tensor core to Matrix $\boldsymbol{C}$ in global memory (line 17) in column major form.




(a) method 1: hybrid algorithm (b) method 2: padding with zeros

Figure 5: Handling matrix not in multiple of $16 \times 16$ (parameter set: **ntruhps2048509**), $E = A + B + C + D$.

### 3.2.3 Handling Matrix not in multiple of $16 \times 16$

The polynomial degree 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 a multiple of $16 \times 16$. This implies that we cannot use the 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 polynomial convolution of $496 \times 496$ (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. Albeit with this limitation, it allows us to utilize the fast tensor core to accelerate polynomial convolution in **NTRU** and other similar lattice-based cryptographic schemes. On the other hand, one can also utilize the second method by padding zeros to poly_a to form a matrix that is a multiple of $16 \times 16$. Referring to Figure 5b, zeros are padded to form a matrix of $512 \times 512$ in size. This allows us to perform polynomial convolution of $N \times N$ completely in tensor core, in 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 to utilize second method (i.e. zero-padding polynomial convolution), since the polynomial convolution can be computed fully in tensor core, which is more efficient than first method (i.e. hybrid approach).

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

1. Organize the polynomial in cyclic form and pad the remaining parts with zeros to construct the matrix in multiple of $16 \times 16$ (Algorithm 4).

2. Convert unsigned 16-bit integer (U16) polynomial element to 16-bit floating point (FP16) format (Algorithm 5).

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

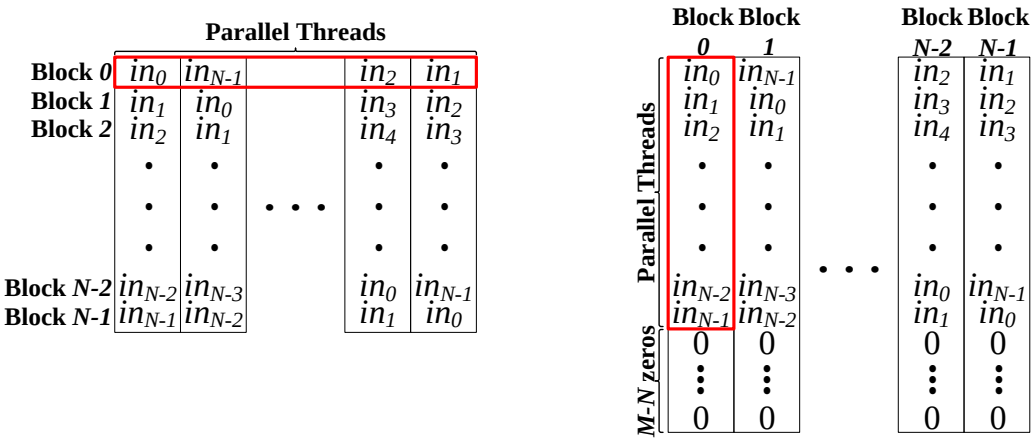
**Input:** Polynomial *in* with degree $N$.

**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: $\quad out[bid + tid \times M] = in[(tid - bid)\%N]$
5: **else**
6: $\quad out[bid + tid \times M] = 0$

3. Convert the 32-bit floating point (FP32) elements back to unsigned 16-bit integer (U16) and perform modulo operations (Algorithm 6).




(a) Read from polynomial *in* (b) Write to matrix *out*

Figure 6: Arranging polynomial in cyclic form and store them in a matrix.

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 in a different cyclic form in order to achieve a 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 $q = 2048$) to FP16 format. Lastly, Algorithm 6 first converts the elements in FP32 to INT32 format (line 4) to keep the original precision, and then performs modulo $q$ and store the final results in U16 format.

With these three proposed algorithms, one can perform highly parallel polynomial convolution for **NTRU** using tensor core, where the steps are given in Algorithm 7. Three floating point matrices are first initialized to zero in CPU; this process is only performed once. Next, the two proposed algorithms are implemented in GPU to perform pre-processing on Matrix $\boldsymbol{A}$ and Matrix $\boldsymbol{B}$ (lines 8-9). Subsequently, tensor core is used to perform the polynomial convolution, resulting a Matrix $\boldsymbol{fp32\_C}$ in FP32 format (line 10). Lastly, this result is converted to Matrix $\boldsymbol{C}$ with U16 format and modulo with $q$ to obtain the final output.

**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 FP16 variable`
4: $temp = in[bid \times M] + tid$ ▷ `Launch N blocks and N threads in parallel`
5: **if** $temp = 2047$ **then**
6: $out[bid \times M] + tid] = -1$ ▷ `Converting -1 from U16 to FP16`
7: **else**
8: $out[bid \times M] + tid] = temp$

**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 FP32 variable`

`// Launch N blocks and N threads in parallel`
4: $temp = in[bid \times M] + tid$
5: $out[bid \times M] + tid] = temp\%q$

Another point to take note is that when we use the proposed technique to implement NTRU, the polynomial convolution for decryption is slightly different from the one in encryption. During 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 constant polynomial. In such case, Algorithm 4 and 5 needs to be slightly revised. In particular, lines 5-8 in Algorithm 5 should be moved to Algorithm 4 to cater for the small ternary polynomial. In other words, one does not need to perform lines 5-8 in Algorithm 5 anymore as the input polynomial does not contain any negative value.

## 3.3 Ephemeral Key Pair

The proposed tensor core based polynomial convolution can be more efficient compared to integer based implementation in GPU. So far, we have only discussed the situations that allow the same public/private key pair to be reused for a small number of encryption/decryption. For instance, one can perform $K$ encryption/decryption 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 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 propose 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 limitation

**Algorithm 7:** Parallel implementation of **NTRU** polynomial convolution using tensor core in 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 $\boldsymbol{C}$, which contains the cyclic convolution of polynomial $a$ and many different polynomial $b$.

// CPU Phase:

1: $fp16\_A$ ▷ Initialize a matrix in FP16 to store converted $a$
2: $fp16\_B$ ▷ Initialize a matrix in FP16 to store converted $b$
3: $fp32\_C$ ▷ Initialize a matrix in FP32 to store results from tensor core

// GPU Phase:

4: $warp\_tot = (M/16)^2$ ▷ Calculate total number of warps required
5: $tc\_threads = warp\_tot \times 32$
6: $tc\_blocks = tc\_threads/max\_threads$ ▷ Calculate number of blocks
7: $tc\_threads = max\_threads$ ▷ Limit the number of threads to max_threads
8: ParCyc$< N, N > (fp16_A, a)$ ▷ Algorithm 4
9: ParU16toFP16$< N, N > (fp16_B, b)$ ▷ Algorithm 5
10: TC-PC$< tc_blocks, tc_threads > (fp16_A, fp16_B, fp32_C)$ ▷ Algorithm 3
11: ParFP32toU16$< N, N > (C, fp32_C)$ ▷ Algorithm 6

in tensor core that only handles $16 \times 16$ matrix, the value $K$ must be in 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 $\boldsymbol{C}$, which contains the cyclic convolution of polynomial $a$ and many different polynomial $b$.

// CPU Phase:

1: $fp16\_A$ ▷ Initialize a matrix in FP16 to store converted $a$
2: $fp16\_B$ ▷ Initialize a matrix in FP16 to store converted $b$
3: $fp32\_C$ ▷ Initialize a matrix in FP32 to store results from tensor core

// GPU Phase:

4: $warp\_tot = (M/16) \times (K/16)$ ▷ Calculate total number of warps required
5: $tc\_threads = warp\_tot \times 32$
6: $tc\_blocks = tc\_threads/max\_threads$ ▷ Calculate number of blocks
7: $tc\_threads = max\_threads$ ▷ Limit the number of threads to max_threads
8: ParCyc$< N, N > (fp16_A, a)$ ▷ Algorithm 4
9: ParU16toFP16$< K, N > (fp16_B, b)$ ▷ Algorithm 5
10: TC-PC$< tc_blocks, tc_threads > (fp16_A, fp16_B, fp32_C)$ ▷ Algorithm 3
11: ParFP32toU16$< K, M > (C, fp32_C)$ ▷ Algorithm 6

Referring to Algorithm 8, the number of warps required to perform TC-PC is reduced from $(M/16)^2$ in Algorithm 7 to $(M/16) \times (K/16)$ (line 1). Besides, the parallel blocks

Table 4: Performance of tensor core based polynomial convolution (multiple of $16 \times 16$), INT32, TC, and PC represent 32-bit integer units, tensor core, and polynomial convolution, respectively.

| $N$ | | 32 | 64 | 128 | 192 | 256 | 384 | 512 | 768 | 1024 |
|---|---|---|---|---|---|---|---|---|---|---|
| Time ($\mu s$) | 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 |

utilized to compute ParU16toFP16 and ParFP32toU16 are also reduced from $N$ to $K$. This is because polynomial $a$ is only used to convolute with $K$ 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 compared to Algorithm 7 as the tensor core is only used to compute $M \times K$ matrix instead of $M \times M$.

### 3.4 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 have utilized this feature in the **NTRU** implementation. However, polynomial addition itself is a lightweight operation, a simple parallel implementation using INT32 unit is already very efficient. Performing the accumulation part 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.

## 4 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. CPU implementations were evaluated on a machine with Intel(R) Core(TM) i7-9700F clocked at 4.7 GHz with 16 GB RAM. The GPU used in this paper is NVIDIA RTX2060 with 8 GB RAM, clocked at 1.71 GHz.

### 4.1 Performance Evaluation of Tensor Core Based Polynomial Convolution

The first experiment aimed at demonstrating the superiority of tensor core based polynomial convolution (TC-PC) against the conventional integer based implementation (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) into the shared memory to reduce the overhead in accessing global memory. For TC-PC implementation, $(N/16)^2$ warps are launched to complete the matrix multiplication. To allow a fair comparison with INT32-PC, we set the number of warps to be close to the one in INT32-PC for various sizes of $N$. The performance of both INT32-PC and TC-PC implementations are presented in Table 4. Note that the results reported for GPU implementations are the average time of one polynomial convolution (i.e. (total time to process $N$ blocks)/N).

When the polynomial degree is small ($N \leq 64$), INT32-PC shows better performance compared to 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 format. 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 when $N = 1024$. We do not report cases beyond 1024, as the speed-up gained does not increase anymore.

## 4.2 Performance Evaluation of NTRU

To demonstrate the benefit of tensor core in accelerating lattice-based cryptographic schemes, we have implemented **NTRU** public key encryption scheme with the parameter sets (**ntruhps2048509** and **ntruhps2048677**) using TC-PC and INT32-PC. Results of our GPU implementation are presented in Table 5, where they are compared against the reference and AVX2 implementation in CPU.

Table 5: Comparing **TensorTRU** against other GPU and CPU implementations; INT32, TC, and PC represent 32-bit integer units, tensor core, and polynomial convolution, respectively.

| $N$ | Operation | **CPU** | | **GPU** | | |
|---|---|---|---|---|---|---|
| | | Time ($\mu s$) | | | | Improvement ratio |
| | | Reference | AVX2 | INT32-PC | TC-PC | (INT32-PC / TC-PC) |
| 509 | Poly. Conv. | 13.26 | 1.51 | 0.92 | 0.33 | 2.79 |
| | Encrypt | 21.23 | 2.07 | 1.18 | 0.61 | 1.94 |
| | Decrypt | 85.47 | 3.99 | 2.29 | 1.16 | 1.97 |
| 677 | Poly. Conv. | 25.15 | 2.34 | 1.80 | 0.66 | 2.72 |
| | Encrypt | 33.84 | 2.99 | 2.42 | 1.24 | 1.95 |
| | Decrypt | 125.64 | 6.55 | 5.07 | 2.51 | 2.02 |

For **ntruhps2048509** parameter set, TC-PC implementation is 2.79×, 1.94× and 1.97× faster than INT32-PC for polynomial convolution, encryption and decryption respectively. A similar speed-up ratio is also observed in **ntruhps2048677**, wherein TC-PC implementation is 2.72×, 1.95× and 2.02× faster than INT32-PC. We observed that TC-PC is more than 20× faster than the reference implementation for encryption and decryption; it is also more than 2× faster than the AVX2 implementation. This shows that GPU can be an effective accelerator to assist the computation in CPU, especially in server environment where the CPU cores are usually busy in handling many other tasks.

## 4.3 Performance Evaluation Under Ephemeral Key Pair Scenario

By changing the dimension 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 limitation of current tensor core in NVIDIA GPU that only handles $16 \times 16$ matrix, $K$ has to be in multiple of 16. From Table 6, we

Table 6: Performance of tensor core based polynomial convolution for $M \times K$ dimension (where both $M$ and $N$ are 512), INT32, TC, and PC represent 32-bit integer units, tensor core, and polynomial convolution, respectively.

| $K$ | | | 16 | 32 | 64 | 128 | 256 | 384 |
|---|---|---|---|---|---|---|---|---|
| Time ($\mu s$) | **CPU** | AVX2 | 1.51 | | | | | |
| | **GPU** | INT32-PC | 2.53 | 2.05 | 1.30 | 1.05 | 0.93 | 0.91 |
| | | 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 |

Table 7: Comparing the performance of **TensorTRU** with other existing **NTRU** and classical PKC implementations in GPU. Timing is measured in $\mu s$.

| Implementation | Year | $N$ | GPU | Core | Time (original/scaled) | Techniques |
|---|---|---|---|---|---|---|
| Post-quantum cryptography (**NTRU**) | | | | | | |
| [HVP10] | 2010 | 1171 | GTX280 | 240 | (40.0/ 5.00*) | Schoolbook |
| [LKSP13] | 2013 | 251 | GTX275 | 240 | (3.00/0.38*) | Sliding window |
| [LGY+18] | 2018 | 401 | GTX1080 | 2560 | (1.97/2.63*) | Karatsuba |
| **TensorTRU** | 2020 | 251<br>401<br>1171 | RTX2060 | 1920 | 0.29<br>0.78<br>2.15 | Schoolbook |
| Pre-quantum cryptography (**RSA** and **ECC**) | | | | | | |
| **RSA-3072** [OJRZCCRH20] | 2020 | - | GTX1080 | 2560 | (3,400/4,533*) | Karatsuba |
| **ECC Curve25519** [GZE+20] | 2020 | - | TITAN V | 5120 | (0.01294/0.03451*) | Schoolbook |

* Performance scaled to match the number of GPU cores in RTX2060 (1,920 cores).

observed that the proposed TC-PC is more efficient than the conventional integer based implementation when $K \geq 32$. However, the performance is not as efficient as 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, GPU can complete one polynomial convolution in $0.33\mu s$ (See Table 4), which is faster than all the combinations of $M \times K$. 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 GPU to perform polynomial convolutions if we allow $K \geq 64$. For cases where $K < 64$, it is better to use AVX2 for speeding up the polynomial convolutions.

## 4.4 Comparison with Other NTRU Implementation in GPU

Existing **NTRU** implementations on GPU platforms are presented in Table 7. Note that the previous implementations targeted different polynomial sizes and GPU devices, which are difficult to benchmark the performance, directly. To overcome this, we have implemented the **NTRU** encryption to different polynomial sizes (i.e. $N = 251, 401, 1171$). We also scaled execution timing results of previous implementations to provide a fair comparison; it is calculated as $\frac{Time}{1920/core}$, where the number of cores in our GPU device (RTX2060) is 1920. For the case $N = 1171$, the performance achieved by **TensorTRU** is 2.32× faster than Hermans et al. [HVP10]. **TensorTRU** is also 1.31× faster than the implementation by Lee et al. [LKSP13] for $N = 251$. Note that the sliding window approach requires pre-computation and storage of the polynomial in look-up table, which can vulnerable to side channel (timing) attacks. The most recent work by Lee et al. [LGY+18] exploited Karatsuba algorithm to split the polynomials for more efficient computation; **TensorTRU** using tensor core based polynomial convolution is 3.37× faster than their implementation. Compared to **RSA** that relies on expensive modular exponentiation [OJRZCCRH20] and very large key size, **TensorTRU** ($N = 1171$) is 2108× faster. Scalar multiplication in **ECC** [GZE+20] is 62× faster than **TensorTRU** with $N = 1171$, but it is not consider safe in the post-quantum world.

## 4.5 TensorLAC: Application to LAC

**LAC** is a cryptosystem based on the poly-LWE variant of the Learning with Errors problem, which was selected as Round 2 candidate in NIST PQC competition. The modulus of **LAC** is restricted to $q = 251$, which allows each polynomial element to fit into a single byte [LLZ+18]. The decoding correctness in **LAC** relies heavily on the ability of error correction code (Bose-Chaudhuri-Hocquenghem (BCH)) to recover errors. Even though **LAC** is not selected to advance into Round 3, it won the first prize of the post-quantum cryptography competition hosted by Chinese Association for Cryptologic Research (CACR). **LAC** remains an interesting candidate due to its' superior implementation performance and simplicity in design.

In this paper, we have extended our idea of using tensor core to compute polynomial convolution in **LAC**. Since **LAC** is using modulus $q = 251$, one can use Configuration 5 (See Table 3) to implement polynomial convolution in tensor core. The polynomial degrees in LAC are $N = 512$ and $N = 1024$, which appear to be multiple 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 $a$ (constant) into cyclic form. In this section, we present Algorithm 9 to resolve this issue. The idea is similar to Algorithm 4, except that some of the elements are converted to nega-cyclic form (lines 4-5).

**Algorithm 9:** LACParCyc: parallel algorithm to arrange polynomial in nega-cyclic form.

**Input:** Polynomial *in* with degree $N$.
**Output:** Matrix *out* with $N \times N$ dimension, which is the polynomial *in* organized in nega-cyclic form.

1: $tid$=thread ID
2: $bid$=block ID
3: $temp = in[(tid - bid)\%N]$ ▷ `Launch N blocks and N threads in parallel`

4: **if** $tid - bid < 0$ **then**
5: $\quad out[bid + tid \times M] = q - temp$
6: **else**
7: $\quad out[bid + tid \times M] = temp$

The implementation of polynomial convolution in **LAC** is similar to **NTRU**; it is presented in Algorithm 10. Since polynomial elements in **LAC** are already represented in 8-bit integer (U8), we can use configuration 5 in tensor core, 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 with FP32 are initialize in CPU. Next, Algorithm 9 is executed to 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 by $q$ (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 **LAC** in CPU (reference and AVX2) and GPU (integer units and tensor core), respectively. TC-NPC is showing 2.93× and 3.1× higher performance compared INT8-NPC, for $N = 512$ and $N = 1024$ respectively. These speed-up are slightly higher compared to the **Tensor-TRU**, because there is no need to convert the data from INT8 to FP16 as required in **NTRU**.

**Algorithm 10:** Parallel implementation of **LAC** polynomial convolution using tensor core in 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 $\boldsymbol{C}$, 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: $warp\_tot = (N/16)^2$ ▷ `Calculate total number of warps required`

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

5: $tc\_blocks = tc\_threads/max\_threads$ ▷ `Calculate number of blocks`

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

7: LACParCyc$< N, N >$ $(u8cyclic\_A, a)$ ▷ `Algorithm 9`

8: TC-PC$< tc_blocks, tc_threads >$ $(u8cyclic\_A, B, fp32\_C)$ ▷ `Algorithm 3`

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

Table 8: Comparing **TensorLAC** against other GPU and CPU Implementations; INT8, TC, and NPC represent 8-bit integer units, tensor core, and nega-cyclic polynomial convolution, respectively.

| Implementation ($N$) | Operation | CPU | | GPU | | |
|---|---|---|---|---|---|---|
| | | Time ($\mu s$) | | | | Improvement ratio |
| | | Reference | AVX2 | INT8-NPC | TC-NPC | (INT8-NPC/TC-NPC) |
| **LAC-128** (512) | Poly. Conv. | 55.93 | 4.35 | 1.23 | 0.42 | 2.93 |
| **LAC-192/256** (1024) | Poly. Conv. | 216.74 | 20.68 | 3.88 | 1.25 | 3.10 |

## 4.6 TensorFro: Application to FrodoKEM

**FrodoKEM** uses algebraically unstructured lattices, where its' security is based on the standard learning with errors problem [BCD+16]. Secrets in **FrodoKEM** are sampled from a discrete Gaussian distribution over the integers. **FrodoKEM** is selected as an alternate candidate in the third round of NIST PQC competition. The official **FrodoKEM** parameter sets require that the modulus $q = 2^{15}$ and $q = 2^{16}$, which is 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 to trade-off between security level, size of modulus and the probability of decryption failure. One of the interesting parameters is proposed by Bian et al. [BHS19], 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 **FrodoKEM** Round 3 submission [ABD+20] to obtain a parameter set with small modulus. In this paper, we have instantiated another parameter set for **FrodoKEM**, which is presented in Table 10. By restricting $q = 2048$ and $\sigma = 1.0$, we obtained a parameter set which 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.

The polynomial degree $N$ for **Frodo-II** and **TensorFro** are not in multiple of 16, so we need to use the proposed method to pad zeros to the polynomial (refer to Figure 5b). For **Frodo-II**, the server side can pack many polynomials into a matrix and perform

Table 9: Parameter instantiations of **FrodoKEM**.

| Implementation | $q$ | $\sigma$ | N | ñ | m̃ | Security level (bit) | Error distribution |
|---|---|---|---|---|---|---|---|
| **Frodo-Rec-1** [ABD+20] | $2^{15}$ | 2.8 | 640 | 8 | 8 | 141 | ±12 |
| **Frodo-II** [BHS19] | $2^{11}$ | 1.0 | 570 | 1 | 256 | 137 | ±4 |
| **TensorFro** | $2^{11}$ | 1.0 | 560 | 11 | 12 | 136 | ±4 |

Table 10: Performance of tensor core based matrix multiplication for **Frodo** variants, INT32, TC, and MM represent 32-bit integer units, tensor core, and matrix multiplication, respectively.

| Implementation (application, parameter) | Time ($\mu s$) | | Improvement ratio |
|---|---|---|---|
| | INT32-MM | TC-MM | (INT32-MM/TC-MM) |
| **Frodo-II** (server side, $570 \times 570$) [BHS19] | 1.35 | 0.44 | 3.07 |
| **Frodo-II** (client side, $570 \times 512$) [BHS19] | 1.41 | 0.43 | 3.28 |
| **TensorFro** (server side, $560 \times 550$) | 1.39 | 0.42 | 3.31 |
| **TensorFro** (client side, $560 \times 552$) | 1.39 | 0.42 | 3.31 |

many matrix-vector multiplication using Algorithm 3. The client side can pack two $N \times \tilde{m}$ ($570 \times 256$ and perform matrix multiplications with tensor core. Similar technique is also applicable to **TensorFro** on both client and 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 convolution.

The error vectors in **FrodoKEM** is spanning across a larger distribution compared to ternary values in **NTRU** and **LAC**. For instance, **Frodo-II** and **TensorFro** have error vector with values between {-4, -3, ...0, ..., +3, +4}. When the proposed tensor core based technique is used, the multiplication between a 11-bit ($q = 2048$) sample and error vector produces a maximum of 13-bit value in floating point (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 up to 23-bit (**Frodo-II**, $log_2 570 \times (2^{13} - 1)$; **TensorFro**, $log_2 560 \times (2^{13} - 1)$). This allows the matrix multiplication to be computed correctly within the single precision, so we can utilize the tensor core Configuration 1.

Table 10 shows the results of matrix multiplication for **Frodo** variant parameter sets. The achieved speed-up between INT32-MM and TC-MM is similar to the cases in **TensorTRU** and **TensorLAC**.

# 5 Conclusion

The introduction of tensor core into GPU had stimulated many efficient implementations of deep learning and mixed-precision scientific computing applications. In this paper, we present the first tensor core aided cryptography implementation on GPU. The proposed tensor core based polynomial convolution is faster than the conventional implementations that rely on integer units in GPU. Since the proposed tensor core based polynomial convolution is a generic algorithm, it can be applied to any sizes of matrix/polynomial. This is proven through experimental evaluations, where the proposed techniques are used to speed up various lattice-based cryptosystems. Although the current tensor core can only support limited floating point precisions and integer types, we believed that the situation may change in near future. In particular, the introduction of FP64 into tensor core recently opens up its' adoption into the mainstream scientific computing applications, fostering the use of GPU in a wider range of applications. As this development trend persists, we

believe that the performance of FP64 tensor core will increase and eventually support more parameters for lattice-base cryptography.

## References


[AAAS+19] Gorjan Alagic, Gorjan Alagic, Jacob Alperin-Sheriff, Daniel Apon, David Cooper, Quynh Dang, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene 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.

[ABD+20] Erdem Alkim, Joppe W Bos, Léo Ducas, Patrick Longa, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Chris Peikert, Ananth Raghunathan, and Douglas Stebila. FrodoKEM learning with errors key encapsulation, 2020.

[AT14] Sedat Akleylek and Zaliha Yüce Tok. Efficient interleaved Montgomery modular multiplication for lattice-based cryptography. *IEICE Electronics Express*, 11(22):20140960–20140960, 2014.

[BCD+16] Joppe Bos, Craig Costello, Léo Ducas, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Ananth Raghunathan, and Douglas Stebila. Frodo: Take off the ring! practical, quantum-secure key exchange from LWE. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1006–1018, 2016.

[BCLVV16] Daniel J Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, and Christine Van Vredendaal. NTRU Prime. *IACR Cryptol. ePrint Arch.*, 2016:461, 2016.

[BDK+18] Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Kyber: a CCA-secure module-lattice-based KEM. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 353–367. IEEE, 2018.

[Ber09] Daniel J Bernstein. Introduction to post-quantum cryptography. In *Post-quantum cryptography*, pages 1–14. Springer, 2009.

[BHS19] Song Bian, Masayuki Hiromoto, and Takashi Sato. Filianore: Better multiplier architectures for LWE-based post-quantum key exchange. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pages 1–6, 2019.

[BS10] Joppe W Bos and Deian Stefan. Performance analysis of the SHA-3 candidates on exotic multi-core architectures. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 279–293. Springer, 2010.

[CA69] Stephen A Cook and Stål O Aanderaa. On the minimum computation time of functions. *Transactions of the American Mathematical Society*, 142:291–314, 1969.

[CDH+20] Cong Chen, Oussama Danba, Jeffrey Hoffstein, Andreas Hülsing, Joost Rijneveld, John M Schanck, Peter Schwabe, William Whyte, and Zhenfei Zhang. NTRU algorithm specifications and supporting documentation. 2020.

[DHP+20] Dipayan Das, Jeffrey Hoffstein, Jill Pipher, William Whyte, and Zhenfei Zhang. Modular lattice signatures, revisited. *Des. Codes Cryptogr.*, 88(3):505–532, 2020.

[DKRV18] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. SABER: Module-LWR based key exchange, CPA-secure encryption and CCA-secure KEM. In *International Conference on Cryptology in Africa*, pages 282–305. Springer, 2018.

[DLL+18] Léo Ducas, Tancrede Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-DILITHIUM: Digital signatures from module lattices. 2018.

[DSS+16] Wei Dai, Berk Sunar, John Schanck, William Whyte, and Zhenfei Zhang. NTRU modular lattice signature scheme on CUDA GPUs. In *2016 International Conference on High Performance Computing & Simulation (HPCS)*, pages 501–508. IEEE, 2016.

[FHK+18] Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Prest, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. FALCON: Fast-fourier lattice-based compact signatures over NTRU. *Submission to the NIST's post-quantum cryptography standardization process*, 2018.

[GZE+20] Lili Gao, Fangyu Zheng, Niall Emmart, Jiankuo Dong, Jingqiang Lin, and Charles Weems. Dpf-ecc: Accelerating elliptic curve cryptography with floating-point computing power of gpus. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 494–504. IEEE, 2020.

[HPS98] Jeffrey Hoffstein, Jill Pipher, and Joseph H Silverman. NTRU: A ring-based public key cryptosystem. In *International Algorithmic Number Theory Symposium*, pages 267–288. Springer, 1998.

[HPS+14] Jeffrey Hoffstein, Jill Pipher, John M. Schanck, Joseph H. Silverman, and William Whyte. Transcript secure signatures based on modular lattices. In Michele Mosca, editor, *Post-Quantum Cryptography - 6th International Workshop, PQCrypto 2014, Waterloo, ON, Canada, October 1-3, 2014. Proceedings*, volume 8772 of *Lecture Notes in Computer Science*, pages 142–159. Springer, 2014.

[HPS+17] Jeffrey Hoffstein, Jill Pipher, John M. Schanck, Joseph H. Silverman, William Whyte, and Zhenfei Zhang. Choosing parameters for ntruencrypt. In Helena Handschuh, editor, *Topics in Cryptology - CT-RSA 2017 - The Cryptographers' Track at the RSA Conference 2017, San Francisco, CA, USA, February 14-17, 2017, Proceedings*, volume 10159 of *Lecture Notes in Computer Science*, pages 3–18. Springer, 2017.

[HRSS17] Andreas Hülsing, Joost Rijneveld, John M. Schanck, and Peter Schwabe. High-speed key encapsulation from NTRU. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, volume 10529 of *Lecture Notes in Computer Science*, pages 232–252. Springer, 2017.

[HVP10] Jens Hermans, Frederik Vercauteren, and Bart Preneel. Speed records for NTRU. In *Cryptographers' Track at the RSA Conference*, pages 73–88. Springer, 2010.

[Kar63] Anatolii Karatsuba. Multiplication of multidigit numbers on automata. In *Soviet physics doklady*, volume 7, pages 595–596, 1963.

[KRSS19] Matthias J Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. pqm4: Testing and benchmarking NIST PQC on ARM Cortex-M4. 2019.

[KY10] Abdel Alim Kamal and Amr M Youssef. Enhanced implementation of the NTRUEncrypt algorithm using graphics cards. In *2010 First International Conference On Parallel, Distributed and Grid Computing (PDGC 2010)*, pages 168–174. IEEE, 2010.

[LGY+18] Wai-Kong Lee, Bok-Min Goi, Wun-She Yap, Denis Chee-Keong Wong, and Sedat 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)*, pages 1923–1928. IEEE, 2018.

[LKSP13] Mun-Kyu Lee, Jung Woo Kim, Jeong Eun Song, and Kunsoo Park. Efficient implementation of NTRU cryptosystem using sliding window methods. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 96(1):206–214, 2013.

[LLZ+18] Xianhui Lu, Yamin Liu, Zhenfei Zhang, Dingding Jia, Haiyang Xue, Jingnan He, Bao Li, Kunpeng Wang, Zhe Liu, and Hao Yang. LAC: Practical Ring-LWE based public-key encryption with byte-level modulus. *IACR Cryptol. ePrint Arch.*, 2018:1009, 2018.

[MDCL+18] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S Vetter. NVIDIA Tensor Core programmability, performance & precision. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 522–531. IEEE, 2018.

[NVI20] CUDA NVIDIA. CUDA C programming guide, version 11.2. *NVIDIA Corp*, 2020.

[OJRZCCRH20] Eduardo Ochoa-Jiménez, Luis Rivera-Zamarripa, Nareli Cruz-Cortés, and Francisco Rodríguez-Henríquez. Implementation of rsa signatures on gpu and cpu architectures. *IEEE Access*, 8:9928–9941, 2020.

[Sho99] Peter W Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review*, 41(2):303–332, 1999.

[Too63] Andrei L Toom. The complexity of a scheme of functional elements realizing the multiplication of integers. In *Soviet Mathematics Doklady*, volume 3, pages 714–716, 1963.