TESLAC: Accelerating Lattice-Based Cryptography with AI Accelerator

Lipeng Wan[1,2,3], Fangyu Zheng[1,3](✉), and Jingqiang Lin[4]

[1] State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
zhengfangyu@iie.ac.cn

[2] School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China

[3] Data Assurance and Communication Security Research Center, Chinese Academy of Sciences, Beijing, China

[4] School of Cyber Security, University of Science and Technology of China, Hefei, China

Abstract. In this paper, we exploit AI accelerator to implement cryptographic algorithms. To the best of our knowledge, it is the first attempt to implement quantum-safe Lattice-Based Cryptography (LBC) with AI accelerator. However, AI accelerators are designed for machine learning workloads (e.g., convolution operation), and cannot directly deliver their strong power into the cryptographic computation. Noting that polynomial multiplication over rings is a kind of time-consuming computation in LBC, we utilize a straightforward approach to make the AI accelerator fit well for polynomial multiplication over rings. Additional non-trivial optimizations are also made to minimize the overhead of transformation, such as using low-latency shared memory, coalescing memory access. Moreover, based on NVIDIA AI accelerator, Tensor Core, we have implemented a prototype system named TESLAC and give a set of comprehensive experiments to evaluate its performance. The experimental results show TESLAC can reach tens of millions of operations per second, achieving a performance speedup of two orders of magnitude from the AVX2-accelerated reference implementation. Particularly, with some techniques, TESLAC can also be scaled to other LBC with larger modulo q.

Keywords: Lattice-based cryptosystems · Polynomial multiplication over rings · AI accelerator · Tensor Core · LAC

1 Introduction

Quantum computing has brought a huge security challenge to the widely-used conventional cryptosystems. If large-scale quantum computers are ever built,

This work was partially supported by National Key R&D Program of China under Award 2018YFB0804401 and National Natural Science Foundation of China under Award No. 61902392.

J. Garcia-Alfaro et al. (Eds.): SecureComm 2021, LNICST 398, pp. 249–269, 2021.
https://doi.org/10.1007/978-3-030-90019-9_13

they will be able to break many of the public-key cryptosystems currently in use, such as RSA and ECDSA [17], with Shor's algorithm [24]. That would seriously compromise the confidentiality and integrity of digital communications. In this situation, NIST has initiated a project [20] to solicit, evaluate, and standardize one or more quantum-safe public-key cryptographic algorithms (also called post-quantum cryptography, PQC [8]). The goal of PQC project is to develop cryptosystems that are secure against both quantum and classical computers and can be compatible with existing communication protocols and networks. Among the candidates, LBC, a kind of quantum-safe cryptographic algorithm that has been studied for several years, is considered to be the most promising public-key cryptographic standard scheme. The well-known LBC includes NTRU [12], NewHope [4], Kyber [5], Saber [11], etc.

On the other hand, high performance cryptographic computing has always been the pursuit of academia and industry. Since performance is also an important metric in the evaluation of NIST PQC project, researchers have tried to optimize the proposed schemes in both levels of algorithm design and hardware implementation, such as FPGA [6,19], ASIC [18], CPU supported by AVX2 or AVX-512 [1,23], and even GPU [2,3,10,14]. Compared with the basic implementation, these optimization solutions may have some improvement in performance. However, the performance is still difficult to meet the needs of practical applications.

Along with quantum computing and quantum-safe cryptography, AI (artificial intelligence) is another hot issue that attracts a lot of attention. At the same time, many processor vendors have designed their own dedicated AI processors or accelerators, including server products (e.g., NVIDIA Tensor Core[1], Google TPU), mobile terminals (e.g., Apple Neural Engine), and embedded devices (e.g., Intel neural network stick, and even Tesla self-driving car), to power AI applications. Because AI accelerators are designed for high-density machine learning workloads, they can deliver multiple times computing power than the general-purpose CPU or even GPU. Taking NVIDIA Tensor Core as an example, it can deliver up to 125 Tensor TFLOPS on Tesla V100 for training and inference applications [13], while typical CUDA cores can only provide up to 15 TFLOPS.

Such a huge performance advantage has inspired us to introduce AI accelerators to cryptographic implementation. However, little work has been done ever. In this paper, we have explored the feasibility of implementing LBC with AI accelerators. It is not easy to utilize AI accelerators to implement the cryptographic computation directly, since they are generally dedicated to specific operations and it is almost impossible for third-party developers to control AI accelerators in a more fine-grained way. As for Tensor Core, the 125 TFLOPS can only be achieved in its unique mixed-precision matrix-multiply-and-accumulate computing model which is designed for convolution operation. Specifically, the data precision is very small, the input multiplier of Tensor Core is half-precision floating-point format (abbreviated as FP16 or half) with only 11 bits of significance.

[1] NVIDIA has launched different generations of **Tensor Core**. If there is no additional explanation, **Tensor Core** in this paper refers to the one on the architecture of Volta.

Polynomial multiplication over rings is often the performance bottleneck of LBC. To demonstrate the application of AI accelerators in high performance cryptographic computing, we extend polynomials to matrices of a specific size to adapt it to the operating mode. Compared with other approaches, such as NTT, this method is straightforward and more suitable for Tensor Core. And we choose LAC [15], a kind of LBC, to be implemented with Tensor Core on Tesla V100, and name the prototype as TESLAC (**TE**n**S**or-core accelerated **LAC**).

Our Contributions. The highlight of TESLAC is to turn the machine learning workloads into cryptography workloads. To exploit the maximum potential of TESLAC, we have made the following contributions:

– Firstly, as far as we know, it is the first time to introduce AI accelerators into LBC acceleration, which can provide a new high-performance alternative platform for the development and application of cryptography.
– Secondly, we propose a framework to turn the machine learning workloads (more precisely, convolution operation) to polynomial multiplication over rings. For instance, we represent a method to expand the polynomial (or vector) into a matrix of a specific size to apply the calculation to the operating mode of Tensor Core.
– Thirdly, we have implemented the entire LAC prototype system TESLAC on NVIDIA Tesla V100. A series of optimizations have been made to bring the true power of Tensor Core into practice, such as making full use of low-latency shared memory to cache data, coalescing global memory accesses. Consequently, TESLAC can deliver tens of millions of LAC operations per second, which is two orders of magnitude faster than the LAC submission implementation running in CPU with AVX2, and outperforms other LBC schemes on CPU or GPU by a wide margin.

Organization. The rest of this paper is organized as follows. Section 2 presents background knowledge. Section 3 expounds the polynomial multiplication rule over rings, and explains the reason why we choose LAC. Section 4 demonstrates how to apply Tensor Core to the implementation of LAC and the details of TESLAC. Section 5 illustrates the environment configuration, shows the evaluation and analysis of experimental results, and compares with other implementations. Section 6 concludes our work.

2 Preliminaries

2.1 Lattice-Based Cryptography

A lattice $L \subset \mathbb{R}^n$ is the set of all integer linear combinations of basis vectors $\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_n \in \mathbb{R}^n$, i.e., $L := \{\sum a_i \boldsymbol{b}_i | a_i \in \mathbb{Z}\}$. LBC (Lattice-Based cryptography) is the generic term for constructions of cryptographic primitives that involve lattices, either in the construction itself or in the security proof. Lattice-Based constructions are currently important candidates for PQC. Compared with more widely used and known public-key schemes such as RSA, DH, and ECC, LBC is believed to be secure against both classical and quantum computers.

Vectors and Matrices. Vectors are denoted by bold lower-case characters, such as $\boldsymbol{a}$. And matrices are denoted by uppercase characters, such as $\boldsymbol{A}$.

An m-dimensional vector $\boldsymbol{a} = (a_0,\ ...,\ a_{m-1})$, where the a_i is the component of $\boldsymbol{a}$ for $0 \leq i < m$.

Algebraic Structures. Let $\mathbb{R}$ be real numbers, $\mathbb{Q}$ be rational numbers, and $\mathbb{Z}$ be integers. For an integer $q \geq 1$, let $\mathbb{Z}_q$ be the residue class ring modulo q and $\mathbb{Z}_q = 0, \ldots, q-1$. Define the ring of integer polynomials modulo $x^n + 1$ as $R = \mathbb{Z}[x]/(x^n + 1)$ for an integer $n \geq 1$, and the ring $R_q = \mathbb{Z}_q[x]/(x^n + 1)$ denotes the polynomial ring modulo $x^n + 1$ where the coefficients are from $\mathbb{Z}_q$. The addition and multiplication of the elements in R_q are performed according to those of polynomials.

2.2 LAC

Currently, most lattice-based cryptosystems are based on learning with errors (LWE) assumption [22] and its variants. In case of Ring-LWE, the noisy equation is $(\boldsymbol{a},\ \boldsymbol{b} = \boldsymbol{as} + \boldsymbol{e})$, where $\boldsymbol{a},\ \boldsymbol{s},\ \boldsymbol{e}$ are chosen from a ring. Usually, the integer polynomial ring $R_q = \mathbb{Z}_q[x]/(x^n+1)$ for suitable ring dimension n is used. LAC, a proposal to the NIST PQC standardization that has advanced to round 2, is a kind of LBC based on Poly-LWE (a simplification version of Ring-LWE). The basic primitive comprises three algorithms: KG (key generation), Enc (encryption), Dec (decryption). The core of the whole cryptographic scheme is based on the above three algorithms.

Notations. Define the message space $\mathcal{M} \in \{0,1\}^{l_m}$ for a positive integer l_m, and the space of random seeds $\mathcal{S}$ be $\{0,1\}^{l_s}$ for a positive integer l_s. We use n independently identical distribution of Ψ_σ, namely Ψ_σ^n.

Subroutines. In the subroutines dealing with the encoding and decoding of the error correction, ECCEnc, ECCDec, the conversion between $m \in \{0,1\}^{l_m}$ and its encoding $\widehat{m} \in \{0,1\}^{l_v}$ is provided, wherein l_v is a positive integer denoting the length of the encoding and depending on the specific choice of the parameter settings. $\mathsf{Key\ Generation}$ randomly generates a pair of public key and secret key (pk, sk). Details are described in **Algorithm** 1.

The algorithm $\mathsf{Encryption}$ on the input pk and a message m, encrypts m with the randomness seed. In case that seed is not given, the process is randomized. Otherwise, the encryption is deterministic for the same seed. Details are described in **Algorithm** 2. The subroutine ECCEnc converts the message into a codeword $\widehat{m}$.

The $\mathsf{Decryption}$ on input sk and a ciphertext $\boldsymbol{c}$, recovers the corresponding message $\boldsymbol{m}$. The subroutine ECCDec inputs an encoding $\widehat{\boldsymbol{m}}$, and decodes the codeword in it.

Algorithm 1. Key Generation

```
Ensure: A pair of public key and secret key $(pk, sk)$.
 1: $seed_a \xleftarrow{\$} \mathcal{S}$
 2: $\boldsymbol{a} \leftarrow \mathsf{Samp}(U(R_q; seed_a)) \in R_q$
 3: $\boldsymbol{s} \xleftarrow{\$} \Psi_\sigma^{n,h}$
 4: $\boldsymbol{e} \xleftarrow{\$} \Psi_\sigma^{n,h}$
 5: $\boldsymbol{b} \leftarrow \boldsymbol{as} + \boldsymbol{e}$
 6: return $(pk := (seed_a, b), sk := s)$
```

Algorithm 2. Encryption

```
Require: $(pk, seed_a, b), m \in \mathcal{M}; seed \in \mathcal{S}$
Ensure: A ciphertext $\boldsymbol{c}$.
 1: $seed_a \xleftarrow{\$} \mathcal{S}$.
 2: $\boldsymbol{a} \leftarrow \mathsf{Samp}(U(R_q; seed_a)) \in R_q$
 3: $\widehat{\boldsymbol{m}} \leftarrow \mathsf{ECCEnc(m)} \in \{0,1\}$
 4: $(\boldsymbol{r}, \boldsymbol{e}_1, \boldsymbol{e}_2) \leftarrow \mathsf{Samp}(\Psi_\sigma^{n,h}, \Psi_\sigma^{n,h}, \Psi_\sigma^{l_v})$
 5: $\boldsymbol{c}_1 \leftarrow \boldsymbol{ar} + \boldsymbol{e}_1 \in R_q$
 6: $\boldsymbol{c}_2 \leftarrow (\boldsymbol{br})_{l_v} + \boldsymbol{e}_2 + \lfloor \frac{q}{2} \rceil \cdot \widehat{\boldsymbol{m}} \in \mathbb{Z}_q^{l_v}$
 7: return $\boldsymbol{c} := (\boldsymbol{c}_1, \boldsymbol{c}_2) \in R_q \times \mathbb{Z}_q^{l_v}$
```

Algorithm 3. Decryption

```
Require: $sk = \boldsymbol{s}, \boldsymbol{c} = (\boldsymbol{c}_1, \boldsymbol{c}_2)$
Ensure: A plaintext $\boldsymbol{m}$
 1: $\boldsymbol{u} \leftarrow \boldsymbol{c}_1 \boldsymbol{s} \in R_q$
 2: $\widetilde{\boldsymbol{m}} \leftarrow \boldsymbol{c}_2 - (\boldsymbol{u}_{l_v}) \in \mathbb{Z}_q^{l_v}$
 3: for $i = 0$ to $l_v - 1$ do
 4:     if $\frac{q}{4} \le \widetilde{m}_i < \frac{3q}{4}$ then
 5:         $\widehat{m}_i \leftarrow 1$
 6:     else
 7:         $\widehat{m}_i \leftarrow 0$
 8:     end if
 9: end for
10: $\boldsymbol{m} \leftarrow \mathsf{ECCDec}(\widehat{\boldsymbol{m}})$
11: return $\boldsymbol{m}$
```

2.3 Tensor Core

When we talk about NVIDIA GPU cores, it usually refers to CUDA Cores. From intelligent assistants to autonomous robots and beyond, the deep learning models are addressing challenges that are rapidly growing in complexity. But converging these models has become increasingly difficult and often leads to underperforming and inefficient training cycles. To mitigate these problems, NVIDIA adds

Tensor Core to their GPUs to accelerate AI training. Unlike CUDA Core, Tensor Core, available on Volta and subsequent architectures, is a kind of accelerator designed for computationally-intensive tasks such as fully-connected and convolutional layers in CNN. It consists of programmable matrix-multiply-and-accumulate units, and its associated data path is custom-crafted to dramatically increase floating-point computing throughput at only modest area and power costs.

The Convolution Operation. Taking Convolutional Neural Network (CNN) as an example, Fig. 1 shows the operating procedure of the convolutional layer when performing feature extraction on pictures. The input image (a 5×5 matrix) works with the convolution kernel (a 3×3 matrix).

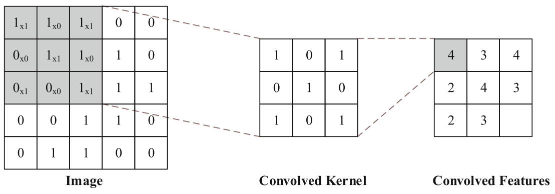


Fig. 1. Basic convolution operation.

The operating mode of Tensor Core is similar to this operation. Each Tensor Core of Tesla V100 provides a matrix processing array which performances the operation $\boldsymbol{D} = \boldsymbol{A} * \boldsymbol{B} + \boldsymbol{C}$, where $\boldsymbol{D}$, $\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{C}$ are 4×4 matrices.

Data Type and Precision. What Tensor Core really performs is Fused Multiply and Add (FMA) mixed-precision operation (FP16 as multipliers and FP32 as an accumulator, shown in Fig. 2).

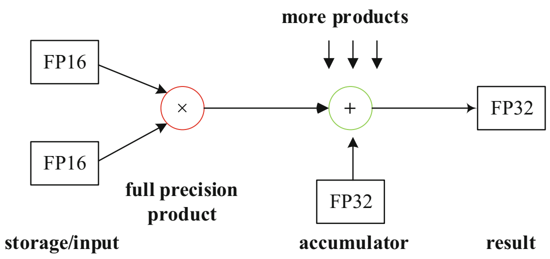


Fig. 2. Tesla V100 Tensor Core operation.

FP16, or half, is a binary floating-point computer number format that occupies 16 bits [9], shown in Fig. 3. In the IEEE 754–2008 standard, the 16-bit base-2 format is referred to as binary16. It is intended for the storage of floating-point values in the application where higher precision is not essential for performing arithmetic computations. The exponent is encoded using an offset-binary representation, with 11-bit (10 bits fraction and an implicit lead bit with value 1) significand precision. Therefore, the maximum value exactly represented is $2^{11} = 2048$.

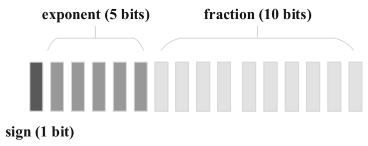


Fig. 3. IEEE 754 half-precision binary floating-point format.

2.4 CUDA Programming Model

CUDA is a general-purpose parallel computing platform launched by NVIDIA. Our proposed prototype is programmed based on this platform.

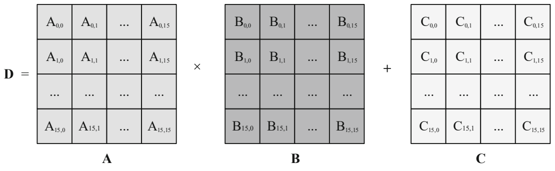
Thread Model. CUDA hardware can possess thousands of cores, which means thousands of threads can execute in parallel. In CUDA C++, it is allowed to define the function as kernel, executed by multiple threads simultaneously. CUDA threads are organized in three levels, grid, block, and thread.

A CUDA hardware contains several SMs (Streaming Multiprocessors), and a block is usually executed on an SM. However, SM schedules only one warp (usually 32 threads) of the block each time. These threads work in SIMT (Single-Instruction, Multiple-Thread) mode. And block is the basic unit of device resource (such as shared memory) allocation. Therefore, when setting the number of threads included in the block, it is better to set an integer multiple of the number of warp.

Memory Model. CUDA hardware has multiple available memory spaces, i.e., global, local, register, shared, constant, or texture memory, and they all have different scopes, lifetimes, and caches.

In general, registers are the fastest, but they are allocated and used by a single thread and are difficult to be used for data sharing and interaction between threads. Local memory is private to the thread and is automatically allocated by the compiler when all registers are used up. Shared memory is allocated according to a single block and visible to all threads in the block. Since shared memory is on-chip, it has much higher bandwidth and much lower latency than local and global memory.

Low-Level Programming of Tensor Cores. However, a single Tensor Core is the smallest execution unit, but not the smallest control unit. Multiple Tensor Cores are used concurrently by a full warp. A larger $16 \times 16 \times 16$ matrix operation (shown in Fig. 4), implicitly divided by multiple Tensor Cores, is conducted by the warp. In addition to using the cuBLAS and cuDNN libraries, developers can also program Tensor Core directly in CUDA C++ via a set of functions and types in the nvcuda::wmma namespace. These warp-level matrix operations are exposed in the CUDA warp matrix-multiply-and-accumulate (WMMA) API listed in the appendix.



D =
$A_{0,0}$ $A_{0,1}$ ... $A_{0,15}$
$A_{1,0}$ $A_{1,1}$ ... $A_{1,15}$
$A_{15,0}$ $A_{15,1}$ ... $A_{15,15}$
A
×
$B_{0,0}$ $B_{0,1}$ ... $B_{0,15}$
$B_{1,0}$ $B_{1,1}$ ... $B_{1,15}$
$B_{15,0}$ $B_{15,1}$ ... $B_{15,15}$
B
+
$C_{0,0}$ $C_{0,1}$ ... $C_{0,15}$
$C_{1,0}$ $C_{1,1}$ ... $C_{1,15}$
$C_{15,0}$ $C_{15,1}$ ... $C_{15,15}$
C

Fig. 4. A warp-level $16 \times 16 \times 16$ matrix operation

At first, the data need to be loaded or initialized to the specific format (fragment) required by Tensor Core, with load_matrix_sync or fill_fragment. In CUDA, fragment is an overloaded class containing a section of a matrix distributed across all threads in the warp. The mapping of matrix elements into fragment internal storage is unspecified. Only certain combinations of template arguments are allowed. The first template parameter specifies how the fragment will participate in the matrix operation. The namespace and class fragment are also shown in the appendix listing. Acceptable values for Use are: matrix_a (A), matrix_b (B), accumulator (C or D). The m, n, and k describe the shape of the warp-wide matrix tiles participating in the multiply-accumulate operation. The dimension of each tile depends on what value for Use. The data type, T, may be half, float on Tesla V100. After the MMA (matrix-multiply-and-accumulate) operation (mma_sync) is performed , the result needs to be stored into memory through the function store_matrix_sync.

Meanwhile, the parameter mptr in function load_matrix_sync and store_matrix_sync must be a 256-bit aligned pointer pointing to the first element of the matrix in memory. In addition, ldm describes the stride in elements between consecutive rows (for row-major layout) or columns (for column-major layout) and must be a multiple of 16 bytes.

Since CUDA does not provide more fine-grained APIs, these functions should be treated as atomic operations on the thread warp. And the focus of programming is how to divide matrix tiles, fill fragments, and achieve parallelism and synchronization.

3 Design

3.1 The Reason to Choose LAC

As mentioned previously, Tensor Cores have a special working mode that is based on dedicated matrix fragments, and the mapping of matrix elements into internal storage of fragment is unspecified. In addition to the warp matrix functions provided in CUDA, third-party developers cannot get more fine-grained programming interfaces.

On the other hand, polynomial multiplication over a ring in LBC is not the calculation of dot product, while the execution of Tensor Core is that of the multi-tuples dot product. It is impossible to utilize Tensor Core with LAC directly.

Compared with other NIST-PQC candidates, LAC is not well-known. However, LAC has unique design features: it uses a byte-wide modulus. Adopting error-correcting codes means that LAC can tolerate a higher decryption failure rate, which allows it to use a smaller modulus that leads to improved performance. The Table 1 lists some values for different LBC cryptosystems selected from NIST-PQC Round-2 [21]. In the table, *pk* stands for public key, *sk* for secret key, *ct* for cipher-text, n for dimension, and q for modulus. Since Kyber [5] is based on Module-LWE and Saber [11] is based on Module-LWR, k means the module rank.

Table 1. Comparison of Several LBC

Algorithm	*pk*(B)	*sk*(B)	*ct*(B)	n	q	k
KYBER512	800	1632	736	256	3329	2
LAC128	544	1056	712	512	251	–
NewHope512	928	869	1088	512	12289	–
NTRU443	611	701	611	443	2048	–
Saber(Light)	672	1568	736	256	2^{13}	2

Since the multiplier's data type of Tensor Core on Tesla V100 is FP16, only integers between 0 and 2048 can be exactly represented in a single data. In LAC, each coefficient is less than 251 (the modulus $q = 251$) and is very suitable to be directly represented in FP16. That is the main reason for choosing LAC. However, this is not to say that our scheme can only be utilized for LBC with a small modulus. For the modulus larger than 2048, taking additional processing and techs, such as multi-precision representation and KaraTsuba algorithm, this platform can also be applied to some other eligible algorithms. More details will be discussed later.

3.2 The Rule of Polynomial Multiplication Rule over Rings

Polynomial multiplication is the basic and most computationally intensive operation in Lattice-Based cryptosystems. There are some rules for polynomial multiplication over rings, which can be used for fast reduction.

Over ring $R_q = \mathbb{Z}_q[x]/(x^n + 1)$, since the modulus is $x^n + 1$, there is

$$x^n \equiv -1 \; mod \; (x^n + 1)$$

Assume $\boldsymbol{a}$, $\boldsymbol{b}$ are n-dimensional vectors on the ring, define $\boldsymbol{a}$, $\boldsymbol{b}$ as:

$$\boldsymbol{a} = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$$

$$\boldsymbol{b} = b_0 + b_1 x + b_2 x^2 + \cdots + b_{n-1} x^{n-1}$$

For $\boldsymbol{c} = \boldsymbol{ab}$, there is

$$\begin{aligned} \boldsymbol{c} &= a_0 b_0 + (a_0 b_1 + a_1 b_0)x + ... + (a_0 b_{n-1} + ... + a_{n-1} b_0) \\ &\quad x^{n-1} + (a_1 b_{n-1} + ... + a_{n-1} b_1)x^n + ... + a_{n-1} b_{n-1} x^{2n-2} \\ &= a_0 b_0 + (a_0 b_1 + a_1 b_0)x + ... + (a_0 b_{n-1} + ... + a_{n-1} b_0) \\ &\quad x^{n-1} - (a_1 b_{n-1} + ... + a_{n-1} b_1) - ... - a_{n-1} b_{n-1} x^{n-2} \end{aligned} \tag{1}$$

$\boldsymbol{c} = (c_0, \cdots, c_{n-1})$, then

$$\begin{aligned} c_0 &= a_0 b_0 - (a_1 b_{n-1} + a_2 b_{n-2} + \cdots + a_{n-1} b_1) \\ c_1 &= a_0 b_1 + a_1 b_0 - (a_2 b_{n-1} + \cdots + a_{n-1} b_2) \\ &\cdots \\ c_{n-1} &= a_0 b_{n-1} + a_1 b_{n-2} + \cdots + a_{n-1} b_0 \end{aligned}$$

$$\Rightarrow c_i = \sum_{j=0}^{i} a_j b_{i-j} - \sum_{j=i+1}^{n-1} a_j b_{n+i-j} \tag{2}$$

3.3 The Application of Tensor Core

We decide to apply Tensor Core to accelerate the polynomial multiplication and addition ($\boldsymbol{c} = \boldsymbol{ab} + \boldsymbol{e}$) over the ring $R_q = \mathbb{Z}_q/(x^n + 1)$. However, the parameter vectors are generally one-dimensional and cannot be used directly on Tensor Core. In a thread warp, Tensor Cores operate on a 16×16 tile, while the dimension $n = 512$ is greater than the tile capacity, then all coefficients cannot be completely loaded into the tile at one time. What's more, the tile operates in rows and columns. For each intermediate result, that is, 16 pairs of coefficients are multiplied and then accumulated. If a vector (part) is placed in the 16 rows of one tile, shown in Fig. 5, it will make the process more complicated. For instance, the intermediate values are at the diagonal position of the **Tile C**, and need to be rearranged and sorted.

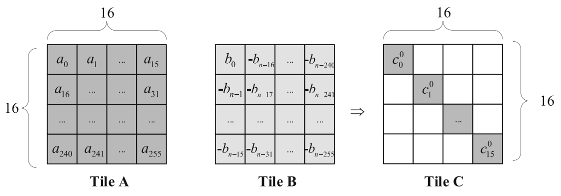

16
16
a_0 a_1 ... a_{15}
a_{16} ... ... a_{31}
a_{240} a_{241} ... a_{255}
b_0 $-b_{n-16}$ ... $-b_{n-240}$
$-b_{n-1}$ $-b_{n-17}$ ... $-b_{n-241}$
$-b_{n-15}$ $-b_{n-31}$ ... $-b_{n-255}$
c_0^0 c_1^0 ... c_{15}^0
⇒
Tile A
Tile B
Tile C

Fig. 5. Tensor Core operation on 1 vector

Some adjustments must be made. There are some idiosyncrasies of (2) and the matrix representation (3) can be inferred.

$$\boldsymbol{c} = \begin{bmatrix} a_0 \; a_1 \; \dots \; a_{n-1} \end{bmatrix} \begin{bmatrix} b_0 & b_1 & \cdots \; b_{n-1} \\ -b_{n-1} & b_0 & \cdots \; b_{n-2} \\ \dots & \dots & \ddots \quad \dots \\ -b_1 & -b_2 & \dots \quad b_0 \end{bmatrix} \tag{3}$$

Therefore, we consider placing a vector (part) in a single row instead of 16 rows of the **Tile A**. In a round of calculation, Tensor Cores manipulate 16 different vectors and get 16 different rows of valid intermediate results, shown in Fig. 6. Further, the vector corresponding to **Tile B** is transformed into an $n \times n$ matrix.

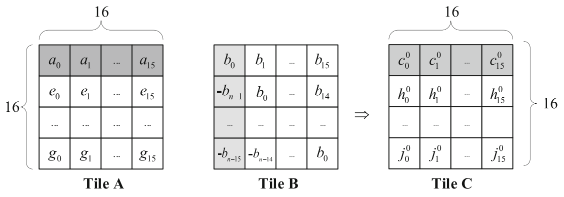

16
16
16
a_0 a_1 ... a_{15}
e_0 e_1 ... e_{15}
g_0 g_1 ... g_{15}
b_0 b_1 ... b_{15}
$-b_{n-1}$ b_0 ... b_{14}
$-b_{n-15}$ $-b_{n-14}$ ... b_0
c_0^0 c_1^0 ... c_{15}^0
h_0^0 h_1^0 ... h_{15}^0
j_0^0 j_1^0 ... j_{15}^0
⇒
Tile A
Tile B
Tile C

Fig. 6. Tensor Core operation on 16 vectors

In this way, the results of a vector are all in one row and can be directly stored in memory without filtering. What's more, this can also handle multiple (at least 16) vectors simultaneously, solving the applicability of Tensor Core. As for the final results, they can be obtained through the accumulation of these intermediate values and should look like (4), where $\boldsymbol{a}^i$, $\boldsymbol{c}^i$ represent different vectors and $\boldsymbol{b}^*$ is the expanded form of vector.

$$\begin{bmatrix} \boldsymbol{a}^0 \\ \boldsymbol{a}^1 \\ \boldsymbol{a}^2 \\ \dots \end{bmatrix} \boldsymbol{b}^* = \begin{bmatrix} \boldsymbol{c}^0 \\ \boldsymbol{c}^1 \\ \boldsymbol{c}^2 \\ \dots \end{bmatrix} \tag{4}$$

All multipliers on the right side of (2) (or (3)) are elements of $\boldsymbol{a}$ and $\boldsymbol{b}$. The difference lies in the index of the elements, and the sign may also change. For each c_i, supposing the sequence $\boldsymbol{a}$ ($\boldsymbol{b}$ is equivalent here) is fixed, and then the relative position of the coefficient sequence of $\boldsymbol{b}$ has not been changed, while it is similar to a cycle, shown in Fig. 7.

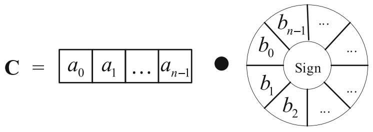


Fig. 7. $\boldsymbol{c} = \boldsymbol{ab}$

As for the matrix representation of $\boldsymbol{b}$, only $[b_{n-1}, b_{n-1}, \cdots, b_0, -b_{n-1}, -b_{n-2}, \cdots, -b_1]$ needs to be stored, and the number of elements is $2n-1$. On the other hand, the `mptr` in WMMA API must be a 256-bit aligned pointer pointing to the first element of the matrix in memory. 256-bit means 16 elements (`half`). For instance, when the pointer is pointing to b_0, if you need to execute a new line starting with b_1, you need to move to another row (stride of at least 16 FP16 elements or integral times of 16) instead of just adding 1 to the pointer. At the same time, the **Tile B** should contain 16 columns. Finally, to meet the alignment requirements of the Tensor Core, we expand a vector into a $2n \times 16$ matrix that is completely suitable, shown as Fig. 8. In addition, each column will be padded with 0.

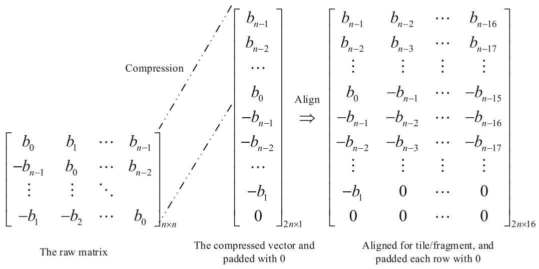

Compression
Align
⇒
$n \times n$
$2n \times 1$
$2n \times 16$
The raw matrix
The compressed vector and padded with 0
Aligned for tile/fragment, and padded each row with 0

Fig. 8. Compression and expand for alignment

4 Implementation

4.1 Overview

The overview of the prototype system is shown in Fig. 9. Data is transmitted between the host and CUDA Hardware through the PCIe bus. Because the two are heterogeneous, the overall task needs to be split. The random number

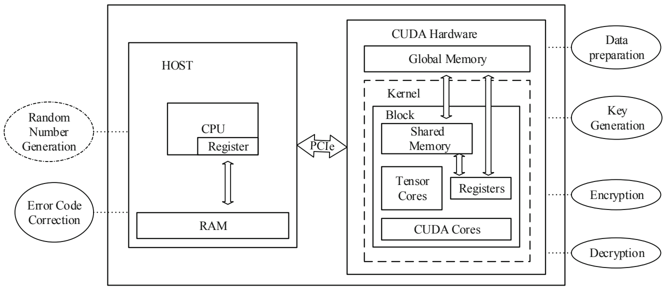


Fig. 9. The overview of the prototype system.

generation and error correction code are generally carried out by the host, while CUDA hardware, also called device, is mainly responsible for computing tasks involving polynomial multiplication. If the error correction code processing is assigned to the device, the control flow would become very complicated. As for random number generation, it can be considered as a portable module, and where it is generated doesn't need to be concerned if the security requirements are satisfied. In other words, the random numbers can be generated by the host, the device, or a third-party hardware module. For convenience, we arrange this task on the host.

The key generation, encryption, and decryption tasks are arranged for the device. However, there are specific requirements for the type and format of the data to be processed by Tensor Core, therefore, the data from the host need to go through a data preparation stage before they participate in the calculation.

4.2 Setting of LAC Parameters

Depending on different security strengths, the dimension n of the LAC can be 512 or 1024. Further, according to the length of plaintext, there are 3 categories of parameters, as Table 2.

Table 2. Parameter settings of LAC

Categories	n	q	Distribution	$\mathsf{BCH}[n_e, l_e, d_e, t_e]$	Plaintext length l_m
LAC128	512	251	Ψ_1	[511, 264, 59, 29]	256
LAC192	1024	251	$\Psi_{\frac{1}{2}}$	[511, 392, 27, 13]	384
LAC256	1024	251	Ψ_1	[1023, 520, 111, 55]	512

4.3 Data Type Conversion

On Tesla V100, the data processed by Tensor Core is required to be half, while the parameters generated by the host are byte or int. That requires the use of some CUDA built-in functions, such as short2half, to perform data type conversion on the original parameters.

4.4 Memory Coalescing

Grouping of threads into warps is not only relevant to computation but also global memory accesses. The device coalesces global memory loads and stores issued by threads of a warp into as few transactions as possible to minimize DRAM bandwidth. If each thread holds a running instance, the memory access stride will be very large. That might lead to an increase in the number of memory requests. Therefore, we coalesce all the instances' memory and use the block as a computing unit to ensure that the memory accessed by threads in a block is as continuous as possible.

4.5 Iteration

Intermediate iterations need to be set according to the size of block. As mentioned earlier, the WMMA APIs are warp-wide functions and manipulate 16 elements for a single vector parameter. An approach is to set up enough threads, and each warp (32 threads) performs only one WMMA operation. Due to the limitation of hardware resources, this method is not feasible. The other method is to iterate in a multiplexed manner, that is, each warp calculates multiple tiles. The iterations should satisfy:

$$\frac{Total\ Tensor\ Core\ operations}{Warps} = \frac{(n/16) \times (n/16)}{block\_size/warp\_size}$$

Meanwhile, n of LAC128 is 512, and a total of $512/16 \times 512/16$ Tensor Core operations are performed. In TESLAC, the block size should be equal or less than n. We set the default block size as 512 and the warp size as 32. That is to say, each warp iterates 64 times. Considering that Tensor Core can automatically sum up, the calculation results, which is obtained by iterating on the matrix $\boldsymbol{B}$ in 16 columns as a round, are parts of the final results without the need for subsequent processing. This also means that the workload of a warp is two columns of tiles of matrix $\boldsymbol{B}$. The partition of overall workload and iterations are shown in Fig. 10.

4.6 Exploiting Shared Memory and Optimization

For each load operation, data are read from global memory. This part of delay seriously affects the performance. When iterating, the matrix $\boldsymbol{B}$ takes the form of $n \times n$, and while storing, it is compressed into $2n \times 16$, which has been described

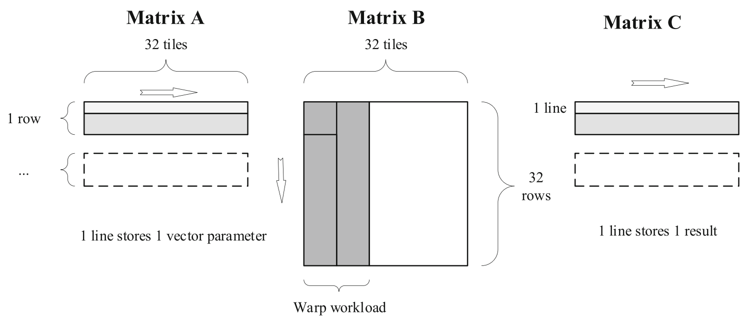


Fig. 10. Partition and iterations of $\mathbf{C} = \mathbf{AB}$

earlier. Even so, it is not a small piece of memory for the cache. Therefore, it is necessary to reduce the proportion of memory access time in the whole iteration. After comparing several memory spaces, we decide to use shared memory for caching, considering its low latency and relatively large capacity. On Tesla V100, each block can use up to 48 kB shared memory. However, as for matrix $\boldsymbol{B}$, its storage size is $2 \times 512 \times 16 \times sizeof(\mathsf{half}) = 32$ kB. After being copied to shared memory, the data is loaded from shared memory rather than global memory. Moreover, if the data in shared memory does not need to be used later, the intermediate results can also be cached in that storage.

Although, this will also encounter another trouble—n might be 1024 or larger, the capacity of shared memory is not enough for caching two or more expanded matrices. Simply, the solution is to divide the matrix into parts and copy data several times. Whatever, a little more cost is acceptable.

5 Results and Analysis

5.1 Setup

Our TESLAC is implemented on a machine with CPU (Intel(R) Xeon(R) CPU E5-2620 v2, 2.10 GHz) and CUDA hardware (NVIDIA Tesla V100). The operating system is Ubuntu 16.04.6 LTS, the CUDA runtime version is 10.1 and the driver version is 460.56. More technical specifications of Tesla V100 are shown in Table 3.

5.2 Performance

The main purpose of our experiments is to demonstrate the feasibility of combining AI accelerator with LBC and understand the performance gain. Firstly, we test polynomial multiplication ($\boldsymbol{c} = \boldsymbol{ab}$) to get the performance of running equal instances under different configurations. Then, we have evaluated and compared the calculation part of TESLAC with the original implementation. Furthermore, we have also compared TESLAC with other PQC implementations.

Table 3. Some technical specifications of Tesla V100

Item	Spec	Item	Spec
CUDA Capability	7.0	Memory Bus Width	4096-bit
GPU Max Clock rate	1380 MHz	Memory Clock rate	877 Mhz
CUDA Cores	5120	SM	80
CUDA Cores/SM	64	Shared memory/block	49152 B
Total Global memory	16160 MB	Registers available/block	65536
Tensor Cores	640	Tensor Cores/SM	8
Max threads/SM	2048	Max threads/block	1024

Polynomial Multiplication. We randomly generate 81920 pairs $(\boldsymbol{a}, \boldsymbol{s})$ to test $\boldsymbol{b} = \boldsymbol{as}$. The grid size and block size is Configurable. And the test record is shown in Table 4.

Table 4. Test polynomial multiplication under different configurations

Grid size	80	80	160	160
Block size	128	256	128	256
Elapsed time (ms)	21.2664	25.4433	18.903	23.1496
Performance million pairs (a, s)/s	3.85208	3.21971	**4.33369**	3.53872

We have compared the performance of polynomial multiplication with the experimental results in [14], shown in Fig. 5. Lee et al. have performed polynomial multiplication using the Nussbaumer algorithm on RTX2060, 490,061 op/s, while our implementation result is 44,150,108 op/s. After scaled by GFLOPS (CUDA Cores×Boost Clock), our implementation is still about 4 times speedup, which mainly results from the utilization of Tensor Core.

Table 5. Comparison of polynomial multiplication implemented on different platform

	GPU Platform	Architecture	CUDA Cores	Boost Clock (MHz)	op/s	op/s (scaled)
[14]	RTX 2060	Turing	1920	1680	490,061	1,073,234
TESLAC	Tesla V100	Volta	5120	1380	4,333,690	4,333,690

Comparison with Original LAC Implementation. After modifying the CPU source code downloaded from the NIST PQC website [21], we compile and execute as the *readme* file, then evaluate the same calculation steps of LAC on Intel(R) Core(TM) i7-7700K CPU. Error correction coding and random number generation are not in the scope of our evaluation, because they are replaceable and not the core of the computation. The results are shown in Table 6. The original source code of LAC provides 3 implementation methods, including *Normal*, *Optimized*, and *AVX2*. We use the latter two to compare with TESLAC.

Table 6. Comparison with original work

	LAC128.KeyGen (op/s)	LAC128.Enc (op/s)	LAC128.Dec (op/s)
CPU Optimized [21]	55,488	28,615	61,118
AVX2 [21]	237,494	114,092	234,157
TESLAC	4,005,495	15,488,955	49,720,202

In terms of CPU implementations, the performance of the AVX2 implementation is about 4 times of the *Optimized*. In addition, the computing performance of LAC128.Enc is much lower than that of LAC128.KeyGen and LAC128.Dec. Because each round of encryption operation needs to perform polynomial multiplication and addition twice ($\boldsymbol{c}_1$ and $\boldsymbol{c}_2$). As for TESLAC, at least 16 sets of data were processed simultaneously with Tensor Core. So the computing performance has been greatly improved, which is about 16x, 135x, and 212x that of the *AVX2*, respectively.

However, these results mainly consider the calculation part of each algorithm. Developers will also encounter time-consuming operations such as data transmission. They can also use other techniques such as pipeline, multiple execution streams and preprocessing to circumvent these effects.

Comparison with Related Work. We have chosen the AVX2 implementation of Kyber512 [5], LightSaber [11] from NIST-PQC Round 2, and GPU implementation of NTRU443 from [2] to compare with TESLAC. The performance evaluation of Kyber512 and LightSaber in [5,11] is conducted by counting CPU cycles consumed. Here, we assume that the CPUs (Intel Core i7-4770K, Intel Core i7-6600U, respectively) are at the maximum frequency, and convert the results to operations per second. In addition, the implementation of NTRU443 is based on NVIDIA GTX1080. The details are shown as Table 7. TDP means the Thermal Design Power.

As far as we know, there is no solution to implement encryption algorithm on AI accelerator yet. Generally speaking, the results of GPU implementation are significantly better than CPU because of massively parallel computation. With the support of AI accelerator, TESLAC greatly exceeds pure GPU implementation. In particular, performance per watt of Tensor Core is far superior. It only needs to consume several times of power and gain hundreds of times of computing performance.

Table 7. Comparison with related work

Algorithm	Platform	Supported technology	Base frequency	Max frequency	TDP	KeyGen (op/s)	Enc (op/s)	Dec (op/s)
Kyber512 [5]	Intel i7-4770K	AVX2	3.50 GHz	3.90 GHz	84 W	116,669	79,294	96,144
LightSaber [11]	Intel i7-6600U	AVX2	2.60 Ghz	3.40 Ghz	15 W	54,973	46,773	48,155
NTRU443 [2]	GTX1080	CUDA Cores	–	–	180 W	–	508,541	–
TESLAC (LAC128)	Tesla V100	Tensor Core	–	–	250 W	4,005,495	15,488,955	49,720,202

Of course, these performance improvements might mainly come from the hardware revenue, while the characteristics of the algorithm itself can not be ignored, and how to make full use of the hardware resources to match the algorithm is the biggest challenge.

5.3 The Scalability for Larger Modulus

Certainly, the techniques used in TESLAC are not limited to LAC. In fact, it is very rare to use a small modulus like LAC, while the coefficients of most LBCs are greater than 2048. In this case, the modulus q exceeds the range represented by FP16. Then, multi-precision presentation can be used to deal with it. Now, suppose we need to calculate $Z = X \times Y$, where X, Y is greater than 2048. Then, we can represent X and Y in more than one FP16 data, for example, $X = X_h \cdot 2^{Base} + X_l$ and $Y = Y_h \cdot 2^{Base} + Y_l$, where X_h, X_l, Y_h, Y_l meets the requirements. The procedure in multi-precision presentation is shown in (5).

$$\begin{aligned} Z &= X \times Y \\ &= (X_h \cdot 2^{Base} + X_l) \times (Y_h \cdot 2^{Base} + Y_l) \\ &= X_h \times Y_h \cdot 2^{2Base} + (X_h \times Y_l + X_l \times Y_h) \cdot 2^{Base} + X_l \times Y_l \end{aligned} \tag{5}$$

Besides, other technologies such as Montgomery reduction [16] or Barrett reduction [7] can also be combined. With the upgrade of hardware products, the significand precision of Tensor Core may be larger, then there will be less restrictions on the scalable application.

6 Conclusion

In this paper, we introduce AI accelerators to high performance cryptographic computing for the first time. Based on NVIDIA's Tensor Core, a kind of AI accelerator, we present a vector expanding method for polynomial multiplication over rings to adapt to the operating mode of Tensor Core. Considering the precision and simplicity of computation, we choose to implement LAC selected from NIST PQC with our techniques on Tesla V100. Consequently, the performance improvement is outstanding.

AI accelerator, which can be treated as an optional platform for high performance cryptographic computing, has great potentiality to be tapped. The performance gain mainly comes from the hardware revenue, but how to make full use of the hardware resources to match the algorithm is the biggest challenge. Meanwhile, these techniques can also be scalable to other LBCs of which the modulus is larger, with a method such as multi-precision representation.

A Appendix

```
#include <mma.h>
using namespace nvcuda;

template<typename Use,int m,int n,int k,typename T,typename
    Layout=void> class fragment;
```

Listing 1.1. The namespace and class fragment

```
void load_matrix_sync(fragment<...> &a, const T* mptr,
    unsigned ldm);
void load_matrix_sync(fragment<...> &a, const T* mptr,
    unsigned ldm, layout_t layout);
void store_matrix_sync(T* mptr, const fragment<...> &a,
    unsigned ldm, layout_t layout);
void fill_fragment(fragment<...> &a, const T &v);
void mma_sync(fragment<...> &d, const fragment<...> &a,
    const fragment<...> &b, const fragment<...> &c, bool
    satf=false);
```

Listing 1.2. The WMMA functions

References

1. Aguilar-Melchor, C., Barrier, J., Guelton, S., Guinet, A., Killijian, M.-O., Lepoint, T.: NFLlib: NTT-based fast lattice library. In: Sako, K. (ed.) CT-RSA 2016. LNCS, vol. 9610, pp. 341–356. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-29485-8_20
2. Akleylek, S., Goi, B., Yap, W., Wong, D.C., Lee, W.: 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), pp. 1923–1928 (2018)
3. Akleylek, S., Dağdelen, Ö., Yüce Tok, Z.: On the efficiency of polynomial multiplication for lattice-based cryptography on GPUs using CUDA. In: Pasalic, E., Knudsen, L.R. (eds.) BalkanCryptSec 2015. LNCS, vol. 9540, pp. 155–168. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-29172-7_10
4. Alkim, E., Ducas, L., Pöppelmann, T., Schwabe, P.: Post-quantum key exchange—a new hope. In: 25th USENIX Security Symposium (USENIX Security 16), pp. 327–343 (2016)

5. Avanzi, R., et al.: CRYSTALS-KYBER: algorithm specifications and supporting documentation. https://pq-crystals.org/kyber/. Accessed 15 Sep 2020
6. Aysu, A., Patterson, C., Schaumont, P.: Low-cost and area-efficient FPGA implementations of lattice-based cryptography. In: 2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST), pp. 81–86. IEEE (2013)
7. Barrett, P.: Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In: Odlyzko, A.M. (ed.) CRYPTO 1986. LNCS, vol. 263, pp. 311–323. Springer, Heidelberg (1987). https://doi.org/10.1007/3-540-47721-7_24
8. Bernstein, D.J.: Introduction to post-quantum cryptography. In: Post-Quantum Cryptography, pp. 1–14. Springer (2009). https://doi.org/10.1007/978-3-540-88702-7_1
9. Committee, I.S., et al.: 754–2008 IEEE standard for floating-point arithmetic. IEEE Comput. Soc. Std. **2008**, 517 (2008)
10. Dai, W., Sunar, B., Schanck, J., Whyte, W., Zhang, Z.: NTRU modular lattice signature scheme on CUDA GPUs. In: 2016 International Conference on High Performance Computing & Simulation (HPCS), pp. 501–508. IEEE (2016)
11. D'Anvers, J.P., Karmakar, A., Roy, S.S., Vercauteren, F.: Saber: Mlwr-based kem. https://www.esat.kuleuven.be/cosic/pqcrypto/saber/index.html. Accessed 15 Sep 2020
12. Hoffstein, J., Pipher, J., Silverman, J.H.: NTRU: a ring-based public key cryptosystem. In: Buhler, J.P. (ed.) ANTS 1998. LNCS, vol. 1423, pp. 267–288. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0054868
13. Jeremy Appleyard, S.Y.: Programming tensor cores in CUDA 9. https://devblogs.nvidia.com/programming-tensor-cores-cuda-9/. Accessed 5 Apr 2020
14. Lee, W.K., Akleylek, S., Wong, D.C.K., Yap, W.S., Goi, B.M., Hwang, S.O.: Parallel implementation of nussbaumer algorithm and number theoretic transform on a GPU platform: application to qTESLA. The Journal of Supercomputing, pp. 1–26 (2020)
15. Lu, X., et al.: LAC: practical Ring-LWE based public-key encryption with byte-level modulus. IACR Cryptology ePrint Archive 2018, 1009 (2018). https://eprint.iacr.org/2018/1009
16. Montgomery, P.L.: Modular multiplication without trial division. Math. Comput. **44**(170), 519–521 (1985)
17. Nist, F.: FIPS 186-4-Digital Signature Standard (DSS). National Institute of Standards and Technology (2013)
18. Oder, T., Güneysu, T., Valencia, F., Khalid, A., O'Neill, M., Regazzoni, F.: Lattice-based cryptography: From reconfigurable hardware to ASIC. In: 2016 International Symposium on Integrated Circuits (ISIC), pp. 1–4. IEEE (2016)
19. Pöppelmann, T., Güneysu, T.: Towards efficient arithmetic for lattice-based cryptography on reconfigurable hardware. In: Hevia, A., Neven, G. (eds.) LATINCRYPT 2012. LNCS, vol. 7533, pp. 139–158. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33481-8_8
20. Post-quantum cryptography project, N.: Post-quantum cryptography. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography. Accessed 23 Sep 2020
21. Post-quantum cryptography project, N.: Round 2 submissions. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-2-Submissions. Accessed 4 Apr 2020
22. Regev, O.: On lattices, learning with errors, random linear codes, and cryptography. J. ACM (JACM) **56**(6), 1–40 (2009)

23. Seiler, G.: Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography. IACR Cryptology ePrint Archive 2018, vol. 39 (2018)
24. Shor, P.W.: Algorithms for quantum computation: discrete logarithms and factoring. In: Proceedings 35th Annual Symposium on Foundations of Computer Science, pp. 124–134. IEEE (1994)