A Novel High-performance Implementation of CRYSTALS-Kyber with AI Accelerator

Lipeng Wan[1,2,3], Fangyu Zheng[1,3,⋆](✉), Guang Fan[1,2,3], Rong Wei[1,2,3], Lili Gao[1,2,3], Jiankuo Dong[5], Jingqiang Lin[4], and Yuewu Wang[1,2,3]

[1] State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China

[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

[5] School of Computer Science, Nanjing University of Posts and Telecommunications, Nanjing, China

Abstract. Public-key cryptography (PKC), including conventional cryptosystems (e.g., RSA, ECC) and post-quantum cryptography, involves computation-intensive workloads. With noticing the extraordinary computing power of AI accelerators, in this paper, we further explore the feasibility to introduce AI accelerators into high-performance cryptographic computing. Since AI accelerators are dedicated to machine learning or neural networks, the biggest challenge is how to transform cryptographic workloads into their operations, while ensuring the correctness of the results and bringing convincing performance gains.

After investigating and analysing the workload of the commercial off-the-shelf AI accelerators, we utilize NVIDIA's AI accelerator, Tensor Core, to accelerate the polynomial multiplication, usually the most time-consuming part in lattice-based cryptography. A series of measures are taken, such as accommodating the matrix-multiply-and-add mode of Tensor Core and making a trade-off between precision and performance, to leverage Tensor Core as a high-performance NTT box performing NTT/INTT through CUDA C++ WMMA API. Meanwhile, we take CRYSTALS-Kyber, one of the NIST PQC 3rd round candidates, as a case study on RTX 3080 with the Ampere Tensor Core. The empirical results show that the defined NTT of polynomial vector $(n = 256, k = 4)$ with our NTT box obtains a speedup around 6.47x that of the state-of-the-art implementation on the same platform. Compared with the AVX2 implementation submitted to NIST, our Kyber-1024 can achieve a speedup of 26x, 36x, and 35x for each phase.

Keywords: Lattice-Based Cryptography · Polynomial Multiplication Over Rings · NTT · AI accelerator · Tensor Core · Kyber.

⋆ Fangyu Zheng is the corresponding author (E-mail: *zhengfangyu@iie.ac.cn*)

1 Introduction

Quantum computing and Shor's algorithm [30] have raised concern about the security of conventional public-key schemes, such as widely used RSA and ECDSA. In this situation, a new class of cryptosystem with anti-quantum properties, which is known as post-quantum cryptography (PQC, sometimes referred to as quantum-proof, quantum-safe, or quantum-resistant), is in urgent need. To this end, NIST has initiated a process to solicit, evaluate, and standardize one or more quantum-resistant public-key cryptographic algorithms in 2017 [23].

Up to now, many quantum-resistant schemes have been proposed. The security is based on different mathematical hard problems, such as the code-based and the hash-based, among which the lattice-based hardness is the most prevailing one. Performance is an important metric in the evaluation of cryptographic algorithms, and thus many research efforts are made to improve the performance of lattice-based cryptography (LBC). Generally speaking, for the cryptographic schemes based on lattice related problem, such as Ring-LWE [17], Module-LWE [6,15], and Module-LWR [3], polynomial multiplication (over the ring R_q) and hash functions are the time-consuming parts. The hash functions mainly involve bit operations, which can be accelerated by the current commercial off-the-shelf products with processor-aided accelerations (e.g., SHA extension in ARM and Intel CPU). In this way, the principal efforts in LBC acceleration focus on the polynomial multiplication part.

Apart from adopting the Karatsuba multiplication [14] and the Toom-Cook algorithm [31] to improve polynomial multiplication, the more prevailing practice is to exploit Number Theoretic Transform (NTT) for the case $n|(q-1)$, where q is the modulus and n is the dimension. Researchers have carried out many targeted optimizations. The previous works [26,34] used negative wrapped convolution (NWC) to avoid the zero-padding and eliminate the pre-processing and post-processing of NTT. Kyber [5], one of the NIST PQC finalists [21], even integrates the customized NTT into its algorithms to reduce the number of conversions. Because $2n \nmid (q-1)$ in Kyber, where $n = 256$ and $q = 3329 = 256 \cdot 13 + 1$ in its 3rd version, it is not possible to conduct NWC. On the contrary, Kyber absorbs the Chinese Remainder Theorem (CRT) form for its modular polynomial $(X^n + 1)$.

Meanwhile, many solutions have been proposed for the specific platforms to make full use of the hardware features and get better achievable performance. Taking the advantage of vector instructions, Lyubashevsky *et al.* [18] presented an AVX2 optimized NTT and applies it to NTRU. Similarly, Seiler [29] implemented NewHope with AVX2 optimized NTT. With the help of many-thread parallelism and high throughput of GPU (precisely, CUDA core), Gupta *et al.* [11] implements three different classes of post-quantum algorithms on NVIDIA Tesla V100. The main optimized technique of the work [11] is to reorganize the data storage sequence to facilitate continuous memory access. Gao *et al.* [9] also improved the performance of NewHope on NVIDIA MX150 and GTX1650. As for the resource-constrained devices, the proposed solutions might be more dedicated. Thanks to the flexibility of FPGA in programming, Xing and Li [33] presented a compact hardware implementation of Kyber on FPGA with many customized optimizations from the perspective of hardware. And Greconic *et al.* [10] presented implementations of the lattice-based digital signature scheme Dilithium for ARM Cortex-M3 and ARM Cortex-M4.

On the other hand, many manufacturers have designed high-performance AI accelerators, such as Google TPU [7], Apple M1 [12], and NVIDIA Tensor Core [13], to meet the needs of artificial intelligence applications. Compared with other general-purpose processors, AI accelerator generally focuses on low-precision arithmetic, novel data-flow architectures, or in-memory computing capability, and often has extremely stronger computing power. For instance, NVIDIA has claimed that Tesla V100's Tensor Cores can deliver up to 125 Tensor TFLOPS for training and inference applications. And NVIDIA Jetson Xavier NX brings supercomputer performance up to 21 TOPS while the power is up to 15W. However, little research has been done on how to expand the application to other fields, especially high-performance cryptographic computing. Our previous work [32] exploits Volta Tensor Core for byte-level modulus scheme LAC [16], but it does not involve module-lattice and NTT, which are more widely used.

Contributions. The primary motivation of the paper is to bring the extraordinary computing power of the AI accelerator to the area of cryptographic acceleration. In this paper, we further explore the feasibility of applying AI accelerators for LBC implementation. Since AI accelerators are dedicated to machine learning or neural networks, the biggest challenge is how to transform cryptographic operations into their workloads, while ensuring the correctness of the results and bringing convincing performance gains. The contributions of our work are as follows:

– Firstly, we propose a framework for an AI accelerator to accelerate module-lattice based cryptography. Through this framework, we can efficiently convert the workload of cryptographic primitives into the operation of the AI accelerator.
– Secondly, we present an NTT box based on NVIDIA AI accelerator, Tensor Core, under the proposed framework. The NTT box is efficient to perform NTT/INTT, especially when the dimension n is relatively small.
– Finally, we evaluate the proposed novel method for Kyber, a well-known PQC scheme, as a case study. To the best of our knowledge, it is the first attempt at implementing Kyber with an AI accelerator. Compared with the state-of-the-art implementation, our *polyvec_ntt* in Kyber can obtain a speedup of 8.1x.

Structure. The rest of this paper is organized as follows: Section 2 introduces the background knowledge, including NTT in Kyber and Tensor Core. Section 3 demonstrates the design of our NTT box. Section 4 presents some details of the implementation and case study. Section 5 shows the evaluation results and discusses the work. Finally, Section 6 concludes our work.

2 Preliminary

In this section, we give a basic background of Kyber, NTT and Tensor Core.

2.1 Notation & Definition

Notation. For a prime q, $\mathbb{Z}_q = \{0, 1, \ldots, q-1\}$ is the residue class ring modulo q. Define the ring of integer polynomials modulo (x^n+1) as $R = \mathbb{Z}_q/(x^n+1)$ for an integer $n \geq 1$, and the ring $R_q = \mathbb{Z}_q[x]/(x^n+1)$ means the coefficients are from $\mathbb{Z}_q$. Regular font letters denote elements in R or R_q (which includes elements in $\mathbb{Z}$ and $\mathbb{Z}_q$) and bold lower-case letters represent vectors with coefficients in R or R_q. By default, all vectors will be column vectors. Bold upper-case letters are matrices. For a vector $\mathbf{v}$ (or matrix $\mathbf{A}$), $\mathbf{v}^T$ (or $\mathbf{A}^T$) means its transpose. For a vector $\mathbf{v}$, $\mathbf{v}[i]$ denotes its i-th entry (with indexing starting at zero); for a matrix $\mathbf{A}$, $\mathbf{A}[i][j]$ denotes the entry in row i, column j (again, with indexing starting from zero). The rank k represents that a polynomial vector contains k polynomials, and a matrix contains $k \times k$ polynomials. For a finite field $F = \mathbb{Z}/q$, the primitive n-th root ω of unity exist whenever $n|(q-1)$, where $\omega^n \equiv 1 \mod q$.

Module-LWE. A lattice is the set of all integer linear combinations of some linearly independent vectors belonging to the euclidean space. Most lattice-based cryptographic schemes are built upon the assumed hardness of the Short Integer Solution (SIS) [1] and Learning With Errors (LWE) [25] problems. The LWE problem was popularized by Regev [25] who showed that solving a random LWE instance is as hard as solving certain worst-case instances of certain lattice problems. This assumption states that it is hard to distinguish from the uniform distribution ($\mathbf{A}$,$\mathbf{A}\mathbf{s}+\mathbf{e}$), where $\mathbf{A}$ is a uniformly-random matrix in $\mathbb{Z}_q^{m\times n}$, $\mathbf{s}$ is a uniformly-random vector in $\mathbb{Z}_q^n$, and $\mathbf{e}$ is chosen from some distribution. Later, Lyubashevsky *et al.* [17] introduced a similar adaptation for LWE, called Ring-LWE, which showed that it is also hard to distinguish a variant of the LWE distribution from the uniform one over certain polynomial rings. Combining the security advantages of LWE and the flexibility of Ring-LWE, Langlois *et al.* [15] demonstrated the worst-case to average-case reductions for module lattices. Intuitively, the size matrix $\mathbf{A}$ in Module-LWE is $k \times k$, where k is the rank. The elements in the matrix are vectors selected from $\mathbf{Z}_q^n$.

2.2 Description of CRYSTALS-Kyber

CRYSTALS-Kyber, or Kyber [5,28], whose security is based on the hardness of solving the LWE problem in module lattices, is an IND-CCA2-secure post-quantum key exchange mechanism. The submission to NIST PQC [24] lists three different parameter sets, Kyber-512, Kyber-768, and Kyber-1024, aiming at different security levels roughly equivalent to AES-128, AES-192, and AES-256, respectively. The parameters are listed in Table 1, where η_1 and η_2 are the

Table 1: Parameter sets for Kyber version 3

	n	k	q	η_1	η_2
Kyber-512	256	2	3329	3	2
Kyber-768	256	3	3329	2	2
Kyber-1024	256	4	3329	2	2

Algorithm 1 KYBER.CPAPKE.KeyGen(): key generation

```
Ensure: Secret key $sk$, Public key $pk$.
1: $d \leftarrow \boldsymbol{Random()}$
2: $(\rho, \sigma) := \mathrm{G}(d)$
3: $\hat{\mathbf{A}} \leftarrow Gen\_matrix\_\hat{\mathbf{A}}(\rho)$, $\hat{\mathbf{A}} \in R_q^{k \times k}$ in NTT domain
4: $\mathbf{s} \leftarrow Sample\_s(\sigma)$, $\mathbf{s} \in R_q^k$ from $B_{\eta_1}$
5: $\mathbf{e} \leftarrow Sample\_e(\sigma)$, $\mathbf{e} \in R_q^k$ from $B_{\eta_1}$
6: $\hat{\mathbf{s}} := NTT(\mathbf{s})$
7: $\hat{\mathbf{e}} := NTT(\mathbf{e})$
8: $\hat{\mathbf{t}} := \hat{\mathbf{A}} \circ \hat{\mathbf{s}} + \hat{\mathbf{e}}$
9: return $pk := Encode(\hat{\mathbf{t}}||\rho)$, $sk := Encode(\hat{\mathbf{s}})$
```

parameters of centered binomial distribution (CBD). The key generation, encryption, and decryption are described in Algorithm 1, 2, and 3.

In the KeyGen phase, d is a random number, ρ and σ are fixed-length intermediate variables generated by d through hash function G. The parameter $\hat{\mathbf{A}}$ is a $k \times k$ polynomial matrix generated by ρ. The parameters $\mathbf{s}$ and $\mathbf{e}$ are polynomial vectors generated through different sample functions but same distribution B_{η_1}. The final parameters need to be compressed and encode. In the Enc phase, the

Algorithm 2 KYBER.CPAPKE.Enc(): encryption

```
Require: Public key $pk$, Message $m$, Random seed $r$
Ensure: Ciphertext $c$
1: $(\hat{\mathbf{t}}, \rho) \leftarrow Decode(pk)$
2: $\hat{\mathbf{A}}^T \leftarrow Gen\_matrix\_\hat{\mathbf{A}}^T(\rho)$, $\hat{\mathbf{A}}^T \in R_q^{k \times k}$ in NTT domain
3: $\mathbf{r} \leftarrow Sample\_r(r)$, $\mathbf{r} \in R_q^k$ from $B_{\eta_1}$
4: $\mathbf{e}_1 \leftarrow Sample\_e_1(r)$, $\mathbf{e}_1 \in R_q^k$ from $B_{\eta_2}$
5: $e_2 \leftarrow Sample\_e_2(r)$, $e_2 \in R_q$ from $B_{\eta_2}$
6: $\hat{\mathbf{r}} := NTT(\mathbf{r})$
7: $\mathbf{u} := NTT^{-1}(\hat{\mathbf{A}} \circ \hat{\mathbf{r}}) + \mathbf{e}_1$
8: $v := NTT^{-1}(\hat{\mathbf{t}}^T \circ \hat{\mathbf{r}}) + e_2 + Decompress(m)$
9: return $c_1 := Encode_u(\mathbf{u})$, $c_2 := Encode_v(v)$
```

public key pk will be decoded first. Here, we need to emphasize that e_2 and v are polynomials rather than vectors. The ciphertext c consists of two parts: c_1 and c_2, which are obtained from $\mathbf{u}$ and v with different encode. Correspondingly, in the Dec phase, these two parts need to be decoded with different functions first. Then the NTT and the subsequent INTT are performed.

Algorithm 3 KYBER.CPAPKE.Dec(): decryption

```
Require: Secret key $sk$,Ciphertext $c$
Ensure: Message $m$
1: $\mathbf{u} := Decode_u(c)$
2: $v := Decode_v(c)$
3: $\hat{\mathbf{s}} := Decode(sk)$
4: return $m := Compress(v - NTT^{-1}(\hat{\mathbf{s}} \circ NTT(\mathbf{u})))$
```

2.3 Number Theoretic Transform

In general, Number Theoretic Transform (NTT) is one of the most prevailing approaches to improve polynomial multiplication over the ring. Simplemindedly, NTT is the finite field form of discrete Fourier transform (DFT), which transforms a sequence of n numbers $\mathbf{v} := \{v_0, v_1, \ldots, v_{n-1}\}$ into another sequence numbers $\mathbf{X} := \{X_0, X_1, \ldots, X_{n-1}\}$. That can be defined by:

$$X_k = \sum_{j=0}^{n-1} v_j \cdot \omega^{jk} \tag{1}$$

where ω is a primitive n-th root of unity, namely, $\omega^n \equiv 1 \mod q$. The inverse transform (INTT) is given as:

$$v_j = n^{-1} \sum_{k=0}^{n-1} X_k \cdot \omega^{-jk} \tag{2}$$

n^{-1} denotes the inverse of n, where $n \cdot n^{-1} \equiv 1 \mod q$.

The fast NTT is based on the idea of divide and conquer, similar to fast Fourier transform (FFT) [8], can perform the polynomial multiplication with the complexity of $O(n \log n)$. However, in practice, the usage of fast NTT can achieve acceleration only when n is relatively large.

NTT-based multiplication. Generally, NTT-based multiplication needs $q \equiv 1 \mod n$ to ensure the existence of the n-th roots of unity, where n is a power of 2. In a finite field, the NTT multiplication of two vectors $\mathbf{a}$ and $\mathbf{b}$ needs to append n zeros to each vector. Then, the product can be obtained by:

$$\mathbf{c} = INTT(NTT(\mathbf{a}_{padding}) \cdot NTT(\mathbf{b}_{padding})) \tag{3}$$

The zero-padding can be avoided to perform NTT-based polynomial multiplication over the ring $\mathbb{R}_q = \mathbb{Z}_q/f(x)$, with the well-known negative wrapped convolution (NWC). However, the NWC requires the existence of the $2n$-th roots of unity, namely, $q \equiv 1 \mod 2n$.

2.4 Fast Modular Reduction

It is necessary to conduct modular reduction for the product of two coefficients or the sum of several products. The native module operation '%' is expensive, even if it might be optimized at the low level of the computer, but that is unspecified. In practice, fast modular reductions like Montgomery reduction [20], and Barrett reduction [4] are utilized, sometimes along with a lazy strategy which means that the reduction is done only before overflow.

Montgomery reduction. Montgomery reduction [20] allows modular arithmetic to be performed efficiently when the modulus is large. Let N be a positive integer, and let R and T be integers such that $R > n$, $gcd(n, R) = 1$, and $0 \leq T < NR$. The Montgomery reduction of $T \mod q$ with respect to R is defined as the value $TR^{-1} \mod q$, where R is a power of 2 and R^{-1} is the modular inverse of R. The calculation steps could be as (4).

$$\begin{aligned} m &:= (T \mod R)k \mod R, \\ t &:= (T + mN)/R \end{aligned} \tag{4}$$

$$\text{if } t \geq N \text{ return } t - N \text{ else return } t.$$

where $k = \frac{R(R^{-1} \mod N)-1}{N}$. Note that R is usually a power of 2, and multiplications and integer divides can be realized by shift, which is cheap.

Barrett reduction. Barrett reduction is another reduction algorithm introduced in 1986 by P.D. Barrett [4] to eliminate division operation in computer.

Let $s = 1/q$ be the inverse of q as a floating point number. Then

$$T \mod q = T - \lfloor Ts \rfloor q$$

where $\lfloor\rfloor$ denotes the floor function. Barrett reduction approximates $1/q$ with a value $m/2^k$ where $m = 2^k/q$. Then the reduction can be converted into (5) and becomes cheap. Since $\lfloor 2^k/q \rfloor$ can be pre-computed, and dividing T by 2^k is just a right-shift.

$$T \mod q = T - \lfloor T/2^k \rfloor \lfloor 2^k/q \rfloor \cdot q \tag{5}$$

2.5 AI Accelerator and Tensor Core

AI accelerator. Due to the explosive growth of AI applications, general-purpose CPUs are hard to meet the needs of neural network data processing. Therefore, a dedicated AI accelerator, an application-specific integrated circuit with a more specific design, may gain far more efficiency. The well-known AI accelerators include Google TPU, Apple M1, M1 MAX, M1 Pro, and ARM NPU. These accelerators mainly focus on optimized memory use and lower precision arithmetic to accelerate calculation and increase the throughput.

Tensor Core. In December 2017, Nvidia released the 1st generation Tensor Core (on Volta architecture) which is just for tensor calculations. Tensor Cores are designed to carry 64 GEMMs (General Matrix Multiplication) per clock cycle on 4×4 matrices, containing FP16 values (floating-point numbers 16 bits in size) or FP16 multiplication with FP32 addition. A year later, Nvidia launched the Turing architecture Tensor Core which has been updated to support other data formats, such as INT8 (8-bit integer values). In the latest Ampere architecture, Nvidia has improved the performance (256 GEMMs per cycle, up from 64), and added further data formats, shown in Table 2.

Table 2: Precision Supported by Multiple Generations of Tensor Core

	Volta	**Turing**	**Ampere**
Precision	*FP16*	*FP16*, *INT8*, *INT4*, *INT1*	*double*, *TF32*, *bfloat16*, *FP16*, *INT8*, *INT4*, *INT1*

Compared with other AI accelerators, Tensor Core exposes interfaces at different levels and has a certain degree of flexibility in its programming. CUDA has provided several tools to leverage Tensor Core, including library cuBLAS and cuDNN, and CUDA C++ WMMA (Warp Matrix Multiply Accumulate) API.

3 Design

As an AI accelerator, Tensor Core is much more powerful than normal CUDA Core. Almost all new CUDA devices come with this component, which means it can be easily accessible to the average developer. Additionally, CUDA provides some relatively low-level programming interfaces to manipulate the component. To this end, we decided to try to implement LBC using AI accelerators to fully utilize the computing resources in CUDA devices.

In this section, we explain the design of our proposal. First, we analyze the workload of Tensor Core. Next, we demonstrate the transformation from cryptographic primitives to workload of Tensor Core. Then, we illustrate the trade-off between performance and precision.

3.1 Analysis of Tensor Core Dedicated Workload

Warp level matrix operation. Up to now, Tensor Core can only support operations at the warp level, usually 32 threads. The warp matrix function requires co-operation from all threads in the warp, and perform the operation $\mathbf{D} = \mathbf{A} \times \mathbf{B} + \mathbf{C}$, where $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$, $\mathbf{D}$ are matrices with specific size, as shown in Fig. 1.

It is further complicated by threads holding only a fragment (a type of opaque architecture-specific ABI data structure) of the overall matrix, with the developer not allowed to make assumptions on how the individual parameters are mapped to the registers participating in the matrix multiply-accumulate. There are also some restrictions on matrix size. Generally, k is fixed to 16, and m can be 8, 16, or 32 (n corresponds to 32, 16, or 8).

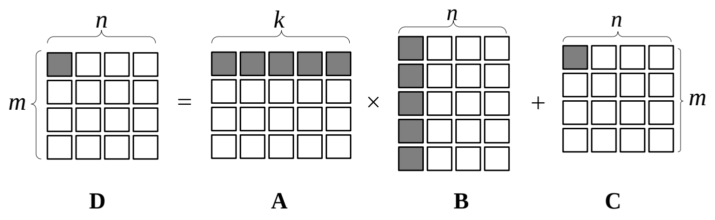


Fig. 1: A warp-level m-n-k matrix operation

FMA operation. Meanwhile, Tensor Core performs FMA mixed-precision operation, which means low-precision input and high-precision output, described in Fig. 2. For example, the FP16 (*half*) multiplication results in a full precision product that is then accumulated using FP32 (*float*) accumulation. Correspondingly, on the Ampere architecture, the input can be *int8* (*char*) and the output can be *int*. Table 3 represents the various combinations of element types of input

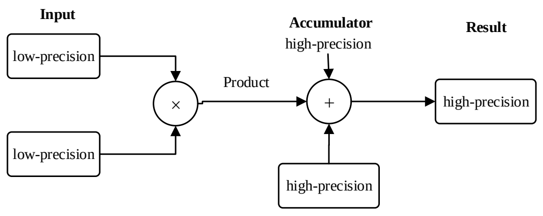


Fig. 2: Tensor Core mixed-precision operation

matrices and input/output accumulators.

Table 3: Precision combinations supported by Tensor Core

Matrix A	*FP16*	*unsigned char*	*signed char*	*__nv_bfloat16*	*precision::tf32*	*FP64*
Matrix B	*FP16*	*unsigned char*	*signed char*	*__nv_bfloat16*	*precision::tf32*	*FP64*
Accumulator C and D	*FP32*	*int*	*int*	*FP32*	*FP32*	*FP64*

3.2 Transformation from Cryptographic Primitive to Tensor Core Dedicated Workload

NTT in Kyber. Similar to NewHope-Compact [2], Kyber reduces its modulus from 12289 to 3329, which naturally improves the efficiency of the algorithm.

In Kyber version 3, the security strength is regulated by the rank k with a fixed dimension $n = 256$ and a modulus $q = 3329 = 256 \cdot 13 + 1$. However, this means the $2n$-th roots do not exist and the negative wrapped convolution is not appliable. On the contrary, Kyber also absorbs the idea like the Chinese Remainder Theorem (CRT) for the modular polynomial, formally, $\mathbb{Z}_q/(f(x) \cdot g(x)) \cong \mathbb{Z}_q/f(x) \times \mathbb{Z}_q/g(x)$, and integrates the customized NTT in its algorithm to reduce conversion between different domain.

The defining polynomial $(X^{256} + 1)$ of R factors into 128 polynomials of degree 2 modulo q. And it can be written as

$$X^{256} + 1 = \prod_{i=0}^{127} (X^2 - \zeta^{2i+1}) = \prod_{i=0}^{127} (X^2 - \zeta^{2\boldsymbol{br}_7(i)+1})$$

where $\boldsymbol{br_7(i)}$ for $i = 0, 1, \cdots, 127$ is the bit reversal of the unsigned 7-bit integer i. Therefore, the NTT of a polynomial $f \in R_q$ is a vector of 128 polynomials of degree 1, and can be written as

$$(f \mod X^2 - \zeta^{2\boldsymbol{br}_7(0)+1}, \cdots, f \mod X^2 - \zeta^{2\boldsymbol{br}_7(127)+1})$$

Hence,

$$NTT(f) = \hat{f} = (\hat{f}_0 + \hat{f}_1 X, \hat{f}_2 + \hat{f}_3 X, \cdots, \hat{f}_{254} + \hat{f}_{255} X)$$

with

$$\hat{f}_{2i} = \sum_{j=0}^{127} f_{2j} \zeta^{(2\boldsymbol{br}_7(i)+1)j} \tag{6}$$

$$\hat{f}_{2i+1} = \sum_{j=0}^{127} f_{2j+1} \zeta^{(2\boldsymbol{br}_7(i)+1)j} \tag{7}$$

where ζ is the 256-th root of unity. The powers of ζ are also called twiddle factors. It is stressed that even though $\hat{f}$ is written as a polynomial in R_q, it has no algebraic meaning as such.

Computing NTT with matrix operation. The prevailing strategy of performing polynomial multiplication with NTT is to use the divide and conquer method. However, in practice, this approach has an advantage only when n is large enough. Moreover, it needs to manipulate each coefficient iteratively, which conflicts with the matrix operating mode.

As aforementioned, Kyber exploits a customized NTT in its algorithms like Equation (6) and (7). In fact, only $n/2$ coefficients of a vector are really involved in an NTT result. In addition, frequent interruptions during in-memory computing to access external memory will seriously increase the delay of the program. Based on the above observations, we decide to adopt a straightforward routine combined with techniques such as pre-computation. Therefore, we assemble several polynomials that need to be processed into a matrix (Matrix $\mathbf{A}$) and place the twiddle factors into another one (Matrix $\mathbf{B}$). The computing mode we adopt is shown in Fig. 3. In this way, this computing model can make full use of SIMT (Single Instruction Multiple Threads) to perform NTT on multiple polynomials at once.

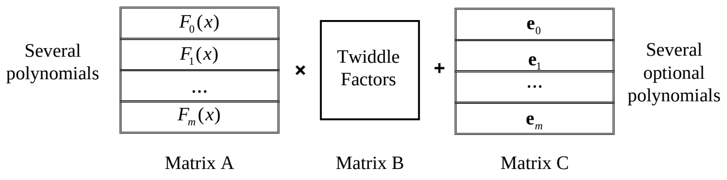


Fig. 3: The computing mode adopted

3.3 The Multiple Precision Representation

As mentioned in Table 3, the Ampere Tensor Core can support several precision combinations. We test the performance of different precision on NVIDIA RTX 3080 and list the results in Table 4. Generally speaking, low precision often corresponds to high computing speed. The choice of data type in cryptographic algorithm should be based on its accurate representation range and performance. For example, the bit length to exactly represent modulus $q = 3329$ (12289) is 12 (14). Then, only *double* of which the mantissa is 52 bits, can cover that case. However, the speed would be particularly slow. To this end, we suggest exploiting multiple-precision representation to make a trade-off, namely, using two or more lower-precision type elements to represent a coefficient.

Table 4: Performance of different precision combinations

	BFloat16 (or *bf16*)	*FP16* (or *half*)	*TensorFloat32* (or *tf32*)	*double*	*int8_t*
Exponent (bits)	8	5	8	11	-
Mantissa (bits)	7	10	10	52	7
Performance$^{\bullet}$	25.89×	28.69×	9.93×	1×	60.56×

The values are to compensate the performance difference caused by different precisions of Tensor Core. The evaluation is conducted with CUDA samples (without shared memory), and the results are based on the minimum value (the performance of *double*).

In the case study of Kyber, we split a 12-bit coefficient into two 6-bit parts represented by *int8_t*. Because the performance of *int8_t* is much higher than that of other floating-point types on Tensor Core.

Internal workflow of NTT box. With the multiple-precision representation, we make Tensor Core play the role of the NTT box as an individual module. The caller could simply load the sorted data into the box and get results quickly. The internal workflow of the NTT box is shown in Fig. 4. Several sorted polynomials are distilled into a matrix, which is then first loaded into the fragment matrix in the form of tiles.

Meanwhile, the pre-computed table will also be loaded into fragment matrix_b. Then, MMA is conducted. Next, the results are performed modular reduction to ensure that the coefficients of the target polynomial are less than q.

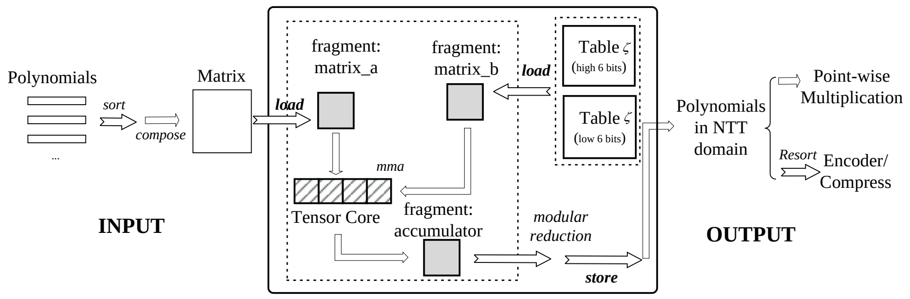


Fig. 4: The workflow of NTT box

4 Implementation Details

In this section, we elaborate on the technical details of our prototype implementation. First, we show the overall architecture of our system and the collaboration between the various modules. Next, we introduce two types of NTT: basic-NTT supports smaller modulus but achieves high performance, and split-NTT supports larger modulus but achieves relatively lower performance. Then, we explain some non-trivial optimization techniques we employ.

4.1 Overview

Our prototype is based on CUDA Toolkit 11.1. CUDA programming can support a large number of concurrent threads. In our implementation, each thread holds one instance, and these threads execute in SIMD mode. Although the specific procedures might be slightly different for key generation, encryption and decryption, the high-level overview could be like Fig. 5.

The collaboration between modules. The function of the RNG module is to extend the random seed to get the required parameters, just like the key derivation function (KDF). During key generation, Kyber uses SHA3-512 to extend a random number (usually 32 bytes) to obtain seed, then obtain key parameters. After obtaining the seed from a RNG module or decoder, Kyber

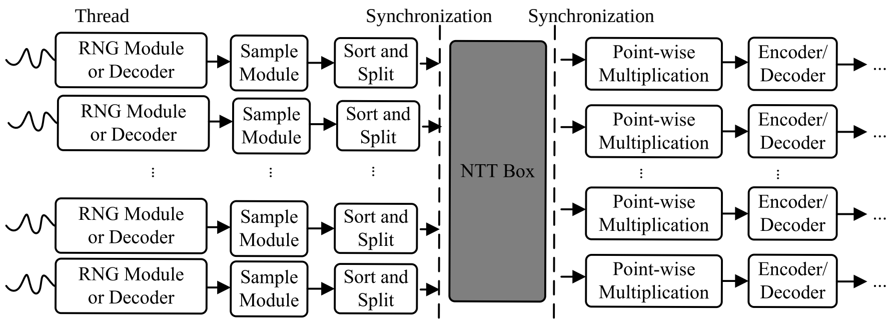


Fig. 5: General overview of implemented Kyber

will generate matrix or sample polynomial vectors based on the seed. On the basis of Equation (6) (7), for a polynomial, the elements with even (or odd) entries participate in the same NTT. Therefore, before entering the NTT box, we sort each polynomial so that even (or odd) entries are continuous in memory. When the program needs to perform NTT, it will synchronize between threads in the same thread block, and then input the data into the NTT box.

4.2 The Basic-NTT and Split-NTT

However, we can only load a fixed size tile into a fragment every time, while the target matrix is much larger. Therefore, we have made two scanning methods, according to the raw precision of the data to be processed. For the parameters of which the element value is less than 8 bits (256, or 128 for signed number), such as secret $\mathbf{s}$ and random noise $\mathbf{r}, \mathbf{e}$ generated from CBD (Centered Binomial Distribution), which have no more than 3 significant bits, we apply a basic-NTT method, shown in Fig. 6.

In this method, we only need to split the twiddle factors into T_h and T_l, and directly represent the input data with *int8_t* type. Both input and output are sorted according to parity items as M_e, M_o, R_e and R_o, to satisfy the requirement of contiguous memory access. Note that β in Fig. 6 represents the base of multiple precision representation, and the multiplication by b can be done by left shifting. As for the case that the coefficient is larger than 8 bits, such as INTT in Kyber, we employ a split-NTT scanning method and the details are shown in Fig 7. The input data is sorted first and then split. The temporary sums, like Tmp_e and Tmp_o in Fig. 7, can be used to reduce a shift operation.

All data matrices have n columns, while the number of rows can be adjusted according to the rank k and the configuration of thread block.

4.3 Pre-computed Table of Twiddle Factors

Since the root ζ is deterministic, and the powers $\zeta^{2\boldsymbol{br}_7(i)+1}$ can be known in advance, then all the twiddle factors can be pre-computed and stored in the

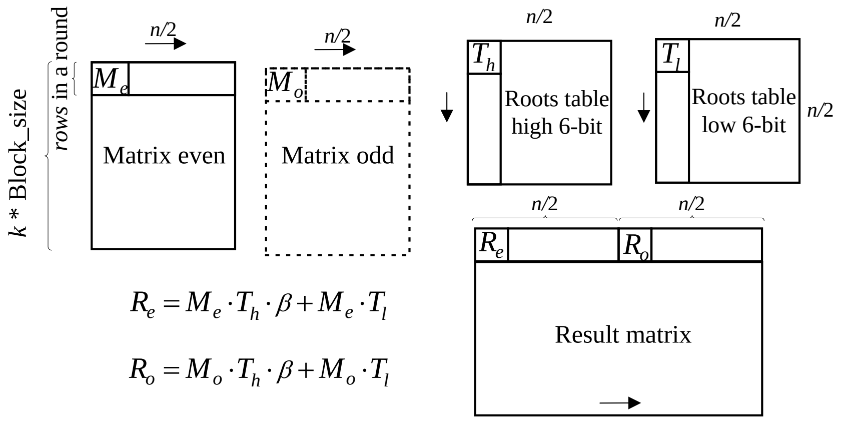


Fig. 6: Scanning of basic-NTT

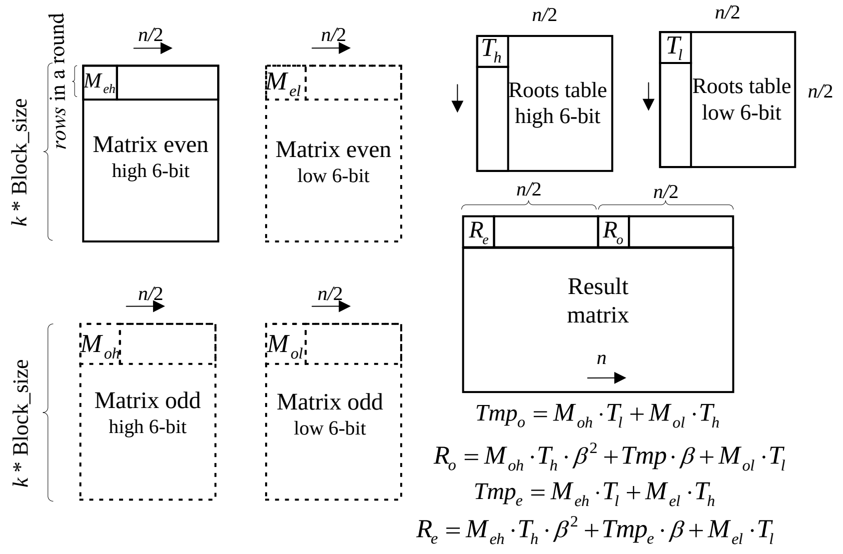

k * Block_size
rows in a round
n/2
M_{eh}
Matrix even high 6-bit
M_{el}
Matrix even low 6-bit
M_{oh}
Matrix odd high 6-bit
M_{ol}
Matrix odd low 6-bit
T_h
Roots table high 6-bit
T_l
Roots table low 6-bit
R_e
R_o
Result matrix
n
$Tmp_o = M_{oh} \cdot T_l + M_{ol} \cdot T_h$
$R_o = M_{oh} \cdot T_h \cdot \beta^2 + Tmp \cdot \beta + M_{ol} \cdot T_l$
$Tmp_e = M_{eh} \cdot T_l + M_{el} \cdot T_h$
$R_e = M_{eh} \cdot T_h \cdot \beta^2 + Tmp_e \cdot \beta + M_{el} \cdot T_l$

Fig. 7: Scanning of split-NTT

memory before the procedure. When NTT is executed, these values can be obtained by looking up the table instead of directly multiplying. This is also what the original author did..

In addition, the intermediate products need to be performed Montgomery reduction. After that, the result of Equation (4) is in Montgomery format and needs to be converted into the normal format by multiplying by R. Therefore, the value R can be absorbed as $\zeta^{2\boldsymbol{br}_7(i)+1} \cdot R \mod q$ to save a multiplication.

According to Equation (1) and (2), our pre-computed table of NTT and INTT could be:

$$\begin{bmatrix} \zeta^{0\times 2\boldsymbol{br}_7(0)}R & \zeta^{0\times 2\boldsymbol{br}_7(1)}R & \cdots & \zeta^{0\times 2\boldsymbol{br}_7(127)}R \\ \zeta^{1\times 2\boldsymbol{br}_7(0)}R & \zeta^{1\times 2\boldsymbol{br}_7(1)}R & \cdots & \zeta^{1\times 2\boldsymbol{br}_7(127)}R \\ \vdots & \vdots & \ddots & \vdots \\ \zeta^{127\times 2\boldsymbol{br}_7(0)}R & \zeta^{127\times 2\boldsymbol{br}_7(1)}R & \cdots & \zeta^{127\times 2\boldsymbol{br}_7(127)}R \end{bmatrix}_{128\times 128} \mod q,$$

.

$$\begin{bmatrix} n^{-1}\zeta^{-0\times 2\boldsymbol{br}_7(0)}R & n^{-1}\zeta^{-0\times 2\boldsymbol{br}_7(1)}R & \cdots & n^{-1}\zeta^{0\times 2\boldsymbol{br}_7(127)}R \\ n^{-1}\zeta^{-1\times 2\boldsymbol{br}_7(0)}R & n^{-1}\zeta^{-1\times 2\boldsymbol{br}_7(1)}R & \cdots & n^{-1}\zeta^{-1\times 2\boldsymbol{br}_7(127)}R \\ \vdots & \vdots & \ddots & \vdots \\ n^{-1}\zeta^{-127\times 2\boldsymbol{br}_7(0)}R & n^{-1}\zeta^{-127\times 2\boldsymbol{br}_7(1)}R & \cdots & n^{-1}\zeta^{-127\times 2\boldsymbol{br}_7(127)}R \end{bmatrix}_{128\times 128} \mod q.$$

Note that the transpose of the matrix can be determined by the flag parameter of the built-in function. In addition, the NTT results are:

$$\begin{aligned} \tilde{f}_{2i} &= \sum_{j=0}^{127} f_{2j}\zeta^{(2\boldsymbol{br}_7(i)+1)j}R \\ \tilde{f}_{2i+1} &= \sum_{j=0}^{127} f_{2j+1}\zeta^{(2\boldsymbol{br}_7(i)+1)j}R \end{aligned} \tag{8}$$

4.4 Point-wise Multiplication and Modular Reduction

Point-wise multiplication. In Kyber, after applying NTT or INTT to a polynomial, the polynomial multiplication $h(x) = f(x) \cdot g(x)$ has also been redefined. $NTT(f) \circ NTT(g) = \hat{f} \circ \hat{g} = \hat{h}$ denotes the basecase multiplication consisting of the 128 products.

$$\hat{h}_{2i} + \hat{h}_{2i+1}X = (\hat{f}_{2i} + \hat{f}_{2i+1}X)(\hat{g}_{2i} + \hat{g}_{2i+1}X) \mod (X^2 - \zeta^{2\boldsymbol{br}_7(i)+1})$$

Specifically, the product coefficients can be written as:

$$\begin{aligned} \hat{h}_{2i} &= \hat{f}_{2i}\hat{g}_{2i} + \hat{f}_{2i+1}\hat{g}_{2i+1}\zeta^{2\boldsymbol{br}_7(i)+1} \\ \hat{h}_{2i+1} &= \hat{f}_{2i}\hat{g}_{2i+1} + \hat{f}_{2i+1}\hat{g}_{2i} \end{aligned} \tag{9}$$

Similar to the multiplication of multi-precision representation, point-wise multiplication can also be combined with the Karatsuba algorithm. Then, we can utilize Karatsuba algorithm [14] to decrease the times of multiplication, and the calculation form of results are listed in (10).

$$\begin{aligned} \hat{h}_{2i} &= \hat{f}_{2i}\hat{g}_{2i} + \hat{f}_{2i+1}\hat{g}_{2i+1}\zeta^{2\boldsymbol{br}(i)+1} \\ \hat{h}_{2i+1} &= (\hat{f}_{2i} + \hat{f}_{2i+1})(\hat{g}_{2i} + \hat{g}_{2i+1}) - (\hat{f}_{2i}\hat{g}_{2i} + \hat{f}_{2i+1}\hat{g}_{2i+1}) \end{aligned} \tag{10}$$

One round lazy modular reduction. For CBD generated vectors, the biggest sum in NTT should be less than $n'q \cdot 2^3$ (where $n' = 128$), which is 22 bits. As mentioned earlier, Tensor Core performs FMA operation, and the sum of intermediate products is still in the representation range of accumulator, such as *int* (32 bits). For a polynomial whose coefficient can be up to $q - 1$, we use two 6-bit data to represent the value. Therefore, $n'q \cdot 2^6$ (25 bits) can also be represented without overflow. Then, only a round fast modular reduction is needed for the final NTT result.

5 Performance Evaluation and Discussion

In this section, we present our evaluation results and perform a comparative analysis with related works, including the performance of the NTT box (especially *polyvec_ntt*), and Kyber-512, Kyber-768, Kyber-1024. Finally, we discuss the scalability and security of our solution.

5.1 Results of NTT/INTT

Firstly, we test the performance of the two types of NTT. There is no significant discriminative between NTT and INTT except for the pre-computed twiddle factor tables. Since INTT does not involve small coefficients in Kyber, we evaluate the split-INTT (for *int16_t*) more. The results are listed in Table 5. For split-NTT, when the thread block size is 128, the performance can reach 247.2 MOPS.

Table 5: The performance of NTT, **Total_case**=69632, **Grid_size**=136, $n = 256$

Operation	**Input Type**	**Block size**	**Time elapsed (ms)**	**Performance (MOPS)**
split-NTT	int16_t	128	0.281632	247.2
		256	0.356992	195.1
basic-NTT	int8_t	128	0.183296	379.9
		256	0.217088	320.8
split-INTT	int16_t	128	0.277376	251.0
		256	0.357376	194.8

Related work comparison. We also compare the defined NTT of polynomial vector (*poly_vec*, $n = 256$, $k = 4$) implemented with our NTT box and the counterparts on CPU and GPU, and can obtain a speedup of at least 8.1x. Furthermore, we test the provided source code on our machine and still get about 6.47x improvement. The results are shown in Table 6. In fact, Tensor Core is

Table 6: Comparison of *polyvec_ntt* in KYBER, $n = 256, k = 4$

	Device	Architecture	Time (ns)°
Ref	W2123	Skylake-W	6,464
Gupt *et al.* [11]	G1060	Pascal	378.1
	P6000	Pascal	202.3
	V100	Volta	135
	R3080	Ampere	107.81 *
Ours	R3080	Ampere	16.65 •

° The average time spent on each instance obtained by processing a given number of instances.

* The code in [11] is downloaded from https://github.com/nainag/PQC and is tested on RTX3080.

• The same *polyvec_ntt* in KYBER with our NTT box, where the best blockSize is 128.

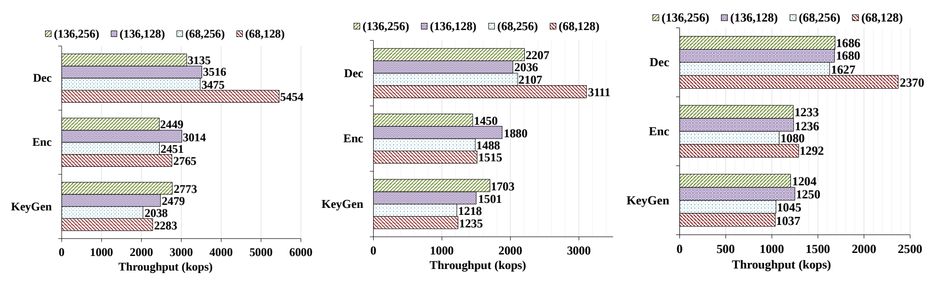
also supported by V100, but not exploited in [11]. Although the gain mainly comes from AI accelerator hardware, the key lies in the advantage of relatively small precision and our fine manipulation to adapt the cryptographic primitives perfectly into its operating mode. Our Tensor Core based NTT box involves the pre-computed tables of twiddle factors instead of the idea of divide and conquer. Because the initial control granularity of butterfly operation is in single element level, which conflicts with the matrix mode and might make the control very complicated. More importantly, interrupting computation frequently to access memory can severely impact performance when utilizing Tensor Cores.

5.2 Results of Kyber

The security strength recommended by the original author is Kyber-768 ($k = 3$) [28]. In addition, we also test Kyber-512 ($k = 2$) and Kyber-1024 ($k = 4$) for the convenience to compare with other existing solutions, and the results are shown in Fig. 8.

Related work comparison. The previous implementations of Kyber are based on various platforms, following different design ideas and targeting different scenarios. The FPGA based implementation such as [33] is mainly committed to using fewer hardware resources to reach more achievable performance. Some platforms (e.g., Apple A12 [27]) have special instructions for hash functions. The CPU based optimization [5] uses vector set instructions for acceleration. Unlike FPGA solutions, in which the improved algorithms are mainly conducted through hardware programming, the hardware circuit of our proposal can no longer be changed, and accelerations can only be carried out around the characteristics it exposes.

Table 7 lists the average time cost on Kyber-1024 of related works. Compared to resource-constrained devices, we can achieve two orders of magnitude



(a) Kyber-512 (b) Kyber-768 (c) Kyber-1024

Fig. 8: The performance results of our prototype.

Table 7: Comparison of average time cost on Kyber-1024 with related works.

	Platform	**KeyGen** (μs)	**Enc** (μs)	**Dec** (μs)	**KX (k/s)**$^\circ$
Pakize Sanal et al. [27]	Apple A12 @2.49 GHz (AES accelerator)	38.23	37.35	36.55	13.4
PQClean [19]	ARM Cortex-A75 @2.8 GHz	137.54	170.25	195.0	3.0
Xing, Y et al. [33]	Xilinx Artix-7	58.2	67.9	86.2	6.93
C-Ref [28]	Intel Core i7-4770K @3.5 GHz (Haswell)	87.8	99.0	113.3	4.97
AVX2-Ref [28]	Intel Core i7-4770K @3.5 GHz (Haswell)	21.01	27.81	22.61	22.9
This work	NVIDIA GeForce RTX 3080	**0.80**	**0.77**	**0.42**	**819.7**

$^\circ$ computed by $\frac{ab}{a+b}$, where a, b are the throughput of KeyGen and Dec.

performance improvement. When compared with the optimized AVX2 version Kyber-1024, our prototype can obtain a speedup of approximately 26x, 36x, and 35x for KeyGen, Enc, and Dec respectively. Note that we have not optimized the hash algorithm yet.

5.3 Discussion

Scalability. Our proposal also illustrates the tremendous potential of Tensor Core in future LBC acceleration. With the upgrade of hardware products, we believe the restrictions would be fewer, and the control interfaces provided could be with finer granularity, which means that they would become more versatile. Although the study case in this paper is the PQC scheme, the proposed solution and techniques can also be applied for other computation-sensitive schemes like homomorphic encryption, or of which the polynomial multiplication is a time-consuming part. Furthermore, in practice, the implementation would be more solid with the optimizations for CUDA hardware, such as multiple working streams, shared memory, and multi-threaded cooperation. Cloud computing service providers, data centers, etc., all require high-performance cryptographic computing. We think that such AI accelerator-based cryptographic optimization solution would appear in their business shortly.

Security. The security issue is also an important aspect of cryptographic implementation. For Tensor Core itself, as far as we know, it can be treated as an atomic instruction parallel execution unit for calculating with a fixed amount of cycles. According to [22], it is almost impossible to perform a timing attack on parallel AI accelerators so far. For example, suppose that the attacker can construct any attack vector for the responder to execute, and knows the time cost (t_v). In general, the execution time t_1 for vectors containing large values is more than the execution time t_2 for vectors with small values. Now that the attacker knows the time t_s for the execution of the secret vector s. The attacker can exploit the information, numerical difference between t_v and t_s, to construct a new attack vector closer to s. However, in our prototype, multiple parameters are executed simultaneously due to the parallel feature of Tensor Core. Only the time $t_s^\star$ cost by the same group as the secret s can be measured, which makes it difficult for the attacker to guess the real t_s. Meanwhile, our implementations contain no conditional statements. Tensor Core related operations involve no secret-related conditional branch, and the related memory access (pre-computed tables) is secret-irrelevant. Techniques against side-channel timing leakage, such as eliminating conditional statements in CUDA kernel functions, are also involved in our work, even though they are not the main focus. In a nutshell, the AI accelerator we introduce will not bring additional security risks.

6 Conclusion and Future Work

In this paper, we propose an NTT box based on NVIDIA AI accelerator, Tensor Core. After that, we present a high performance implementation of CRYSTALS-Kyber with our NTT box and achieve considerable performance improvement. Our work further explores the practicality of applying AI accelerator to LBC. We believe that AI accelerators will become more versatile, and support more operations and precisions. In the future, the subsequent work would cover more lattice-based cryptographic schemes, especially homomorphic encryption (HE) which urgently requires high efficiency for the wider application.

Acknowledgements. We would like to thank the anonymous reviewers for their careful reading of our manuscript and their many insightful comments and suggestions. We are grateful to Massimiliano Albanese for helping us to improve our paper. This work is supported in part by National Key RD Plan of China under Grant No. 2020YFB1005803, the National Natural Science Foundation of China No. 61902392 and CCF-Tencent Open Fund under Grant No. RAGR20210131.

References

1. Ajtai, M.: Generating hard instances of lattice problems. In: Proceedings of the twenty-eighth annual ACM symposium on Theory of computing. pp. 99–108 (1996)
2. Alkım, E., Bilgin, Y.A., Cenk, M.: Compact and simple RLWE based key encapsulation mechanism. In: International Conference on Cryptology and Information Security in Latin America. pp. 237–256. Springer (2019)
3. Banerjee, A., Peikert, C., Rosen, A.: Pseudorandom functions and lattices. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 719–737. Springer (2012)
4. Barrett, P.: Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In: Conference on the Theory and Application of Cryptographic Techniques. pp. 311–323. Springer (1986)
5. Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J.M., Schwabe, P., Seiler, G., Stehlé, D.: CRYSTALS-Kyber: a CCA-secure module-lattice-based KEM. In: 2018 IEEE European Symposium on Security and Privacy (EuroS&P). pp. 353–367. IEEE (2018)
6. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (leveled) fully homomorphic encryption without bootstrapping. ACM Transactions on Computation Theory (TOCT) **6**(3), 1–36 (2014)
7. Cloud, G.: Cloud tpu. `https://cloud.google.com/tpu/`, accessed 19 May 2021
8. Cooley, J.W., Tukey, J.W.: An algorithm for the machine calculation of complex fourier series. Mathematics of computation **19**(90), 297–301 (1965)
9. Gao, Y., Xu, J., Wang, H.: cuNH: Efficient GPU Implementations of Post-Quantum KEM NewHope. IEEE Transactions on Parallel and Distributed Systems **33**(3), 551–568 (2021)
10. Greconici, D.O., Kannwischer, M.J., Sprenkels, D.: Compact dilithium implementations on Cortex-M3 and Cortex-M4. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 1–24 (2021)
11. Gupta, N., Jati, A., Chauhan, A.K., Chattopadhyay, A.: PQC acceleration using GPUs: FrodoKEM, NewHope, and Kyber. IEEE Transactions on Parallel and Distributed Systems **32**(3), 575–586 (2020)
12. Inc, A.: Apple unleashes m1. `https://www.apple.com/newsroom/2020/11/apple-unleashes-m1/`, accessed 19 May 2021
13. Inc, N.: Nvidia tensor cores–unprecedented acceleration for hpc and ai. `https://www.nvidia.com/en-us/data-center/tensor-cores/`, accessed 19 May 2021
14. Karatsuba, A.: Multiplication of multidigit numbers on automata. In: Soviet physics doklady. vol. 7, pp. 595–596 (1963)
15. Langlois, A., Stehlé, D.: Worst-case to average-case reductions for module lattices. Designs, Codes and Cryptography **75**(3), 565–599 (2015)
16. Lu, X., Liu, Y., Zhang, Z., Jia, D., Xue, H., He, J., Li, B., Wang, K.: Lac: Practical ring-lwe based public-key encryption with byte-level modulus. Cryptology ePrint Archive (2018)
17. Lyubashevsky, V., Peikert, C., Regev, O.: On ideal lattices and learning with errors over rings. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 1–23. Springer (2010)
18. Lyubashevsky, V., Seiler, G.: Nttru: truly fast NTRU using NTT. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 180–201 (2019)
19. M, K., J, R., P, S., D, S., Wiggers: The pqclean project. `https://github.com/PQClean/PQClean`, accessed 8 Apr 2022

20. Montgomery, P.L.: Modular multiplication without trial division. Mathematics of computation **44**(170), 519–521 (1985)
21. Moody, D., Alagic, G., Apon, D.C., Cooper, D.A., Dang, Q.H., Kelsey, J.M., Liu, Y.K., Miller, C.A., Peralta, R.C., Perlner, R.A., et al.: Status report on the second round of the nist post-quantum cryptography standardization process (2020)
22. Nakai, T., Suzuki, D., Fujino, T.: Timing black-box attacks: Crafting adversarial examples through timing leaks against DNNs on embedded devices. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 149–175 (2021)
23. NIST: Post-quantum cryptography, call for proposals. `https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/Call-for-Proposals`, accessed 31 Mar 2022
24. NIST: Post-quantum cryptography, round 3 submissions. `https://csrc.nist.gov/Projects/post-quantum-cryptography/round-3-submissions`, accessed 12 Apr 2022
25. Regev, O.: On lattices, learning with errors, random linear codes, and cryptography. Journal of the ACM (JACM) **56**(6), 1–40 (2009)
26. Roy, S.S., Vercauteren, F., Mentens, N., Chen, D.D., Verbauwhede, I.: Compact ring-LWE cryptoprocessor. In: International workshop on cryptographic hardware and embedded systems. pp. 371–391. Springer (2014)
27. Sanal, P., Karagoz, E., Seo, H., Azarderakhsh, R., Mozaffari-Kermani, M.: Kyber on ARM64: Compact implementations of Kyber on 64-bit ARM Cortex-A processors. In: International Conference on Security and Privacy in Communication Systems. pp. 424–440. Springer (2021)
28. Schwabe, P.: Crystals-cryptographic suite for algebraic lattices. `https://pq-crystals.org/kyber/index.shtml`, accessed 18 May 2021
29. Seiler, G.: Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography. IACR Cryptol. ePrint Arch. **2018**, 39 (2018)
30. Shor, P.W.: Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM review **41**(2), 303–332 (1999)
31. Toom, A.L.: The complexity of a scheme of functional elements realizing the multiplication of integers. In: Soviet Mathematics Doklady. vol. 3, pp. 714–716 (1963)
32. Wan, L., Zheng, F., Lin, J.: TESLAC: Accelerating lattice-based cryptography with AI accelerator. In: International Conference on Security and Privacy in Communication Systems. pp. 249–269. Springer (2021)
33. Xing, Y., Li, S.: A compact hardware implementation of CCA-secure key exchange mechanism CRYSTALS-KYBER on FPGA. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 328–356 (2021)
34. Zhang, N., Yang, B., Chen, C., Yin, S., Wei, S., Liu, L.: Highly efficient architecture of NewHope-NIST on FPGA using low-complexity NTT/INTT. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 49–72 (2020)