mathematics

MDPI

Article

Efficient Parallel Implementations of LWE-Based Post-Quantum Cryptosystems on Graphics Processing Units

SangWoo An [1] and Seog Chung Seo [2,*]

[1] Department of Financial Information Security, Kookmin University, Seoul 02707, Korea; pinksnail06@kookmin.ac.kr

[2] Department of Information Security, Cryptology, and Mathematics, Kookmin University, Seoul 02707, Korea

* Correspondence: scseo@kookmin.ac.kr; Tel.: +82-02-910-4742

Received: 13 September 2020; Accepted: 8 October 2020; Published: 14 October 2020

Abstract: With the development of the Internet of Things (IoT) and cloud computing technology, various cryptographic systems have been proposed to protect increasing personal information. Recently, Post-Quantum Cryptography (PQC) algorithms have been proposed to counter quantum algorithms that threaten public key cryptography. To efficiently use PQC in a server environment dealing with large amounts of data, optimization studies are required. In this paper, we present optimization methods for FrodoKEM and NewHope, which are the NIST PQC standardization round 2 competition algorithms in the Graphics Processing Unit (GPU) platform. For each algorithm, we present a part that can perform parallel processing of major operations with a large computational load using the characteristics of the GPU. In the case of FrodoKEM, we introduce parallel optimization techniques for matrix generation operations and matrix arithmetic operations such as addition and multiplication. In the case of NewHope, we present a parallel processing technique for polynomial-based operations. In the encryption process of FrodoKEM, the performance improvements have been confirmed up to 5.2, 5.75, and 6.47 times faster than the CPU implementation in FrodoKEM-640, FrodoKEM-976, and FrodoKEM-1344, respectively. In the encryption process of NewHope, the performance improvements have been shown up to 3.33 and 4.04 times faster than the CPU implementation in NewHope-512 and NewHope-1024, respectively. The results of this study can be used in the IoT devices server or cloud computing service server. In addition, the results of this study can be utilized in image processing technologies such as facial recognition technology.

Keywords: PQC; lattice-based; LWE; RLWE; FrodoKEM; NewHope; GPU; optimization

1. Introduction

The development of information communication technology such as the Internet of things (IoT) is rapidly increasing the amount of data exchanged in real time. As the number of communication users and volume of data increases, the risk of leakage of the user's personal information also increases, so encryption of transmitted information is required. To encrypt data, various modern cryptographic technologies such as symmetric-key and public-key encryption algorithms have been developed [1,2].

However, with the advent of quantum computers, it has been suggested that modern cryptographic algorithms such as Rivest–Shamir–Adleman (RSA), elliptic curve cryptography (ECC), and the digital signature algorithm that rely on mathematical computational complexity like discrete mathematics are no

longer secured for quantum algorithms such as the Shor algorithm [3]. Therefore, a new cryptographic method that is safe for quantum algorithms is needed, and various ideas based on computationally difficult problems have been proposed. A typical hard-problem-based idea is built on issues belonging to the non-deterministic polynomial complete class.

The National Institute of Standards and Technology (NIST) has been recruiting quantum-resistant cryptography that is safe for quantum algorithms through the post-quantum cryptography (PQC) standardization project since 2012 and is currently conducting round 2 screening [4]. The algorithms in the second round of the NIST PQC standardization contest are largely divided into multivariable, code, lattice, and isogeny-based cryptography and hash-based digital signature classifications. Lattice-based quantum-resistant cryptography is the most frequently proposed base among these round 2 target algorithms. Lattice-based cryptography has a fast operation speed compared with other types of quantum-resistant cryptography. However, the rate of increase in user data is getting faster. To keep up with the speed of user data growth, a method that can quickly encrypt large amounts of data is needed. From this need, optimization methods in various microprocessors have been proposed. However, it is still burdensome to actively use quantum-resistant cryptography from the standpoint of a server that needs to process data for multiple users.

To reduce the burden on a server that encrypts the data of multiple IoT devices or user information in a cloud computing service, this paper proposes several methods for optimizing some lattice-based cryptography using a graphics processing unit (GPU). Using the features of the GPU, certain operations or functions can be processed in parallel. By the parallel processing methods, data can be processed faster than it is when processed only by an existing central processing unit (CPU).

So far, various studies have been conducted to optimize cryptographic algorithms using GPU. In the case of block ciphers, GPU optimization studies on lightweight block cipher algorithms have been carried out recently [5]. In the case of public key cryptography, research has been conducted to optimize the Elliptic Curve Cryptography (ECC) in GPU [6]. In addition to encryption, various technologies are being optimized by utilizing the characteristics of the GPU, and studies are being conducted that can speed up tasks with a long processing time such as deep learning, metaheuristic algorithms, and sorting data using the GPU [7]. In the case of quantum resistant cryptography, optimization studies are mainly conducted in an embedded device environment, and optimization studies using GPUs are very few. Therefore, in this paper, we propose an optimization for PQC in a GPU environment.

The target algorithms presented in this paper are FrodoKEM [8], a learning with an error (LWE)-based algorithm, and NewHope [9], a ring-LWE (RLWE)-based algorithm, for lattice-based cryptography. FrodoKEM has major operations such as matrix generation functions and matrix multiplication and NewHope has number theoretic transform (NTT)-based multiplication [10] and big number operations taking up much of the algorithm's operation time. In this paper, we propose methods to speed up the main operation of these algorithms using a GPU. In addition, overall performance optimizations are undertaken using various optimization techniques within the GPU.

By using the GPU to speed up the encryption process, it is possible to reduce the encryption burden on the IoT server or cloud computing server. In addition, it can be used in technologies that need to deal with imaging data in real time, such as face recognition technology and eye tracking technology, or technologies that need to compute large amounts of data such as deep learning [11,12].

The contributions of our paper can be summarized as follows.

1. Optimizing the Latest Lattice-based PQCs
   We optimized performance for the round 2 candidates of the PQC standardization competition hosted by NIST. Among the round 2 competition algorithms, we selected the target algorithm from the lattice-based algorithms, which are the basis of many algorithms. In this paper, optimization methods

for FrodoKEM based on the LWE problem and NewHope based on the RLWE problem are proposed. By measuring the individual computational load for the two target algorithms and optimizing the computationally intensive computation with the GPU, a high level of performance improvement was achieved. In addition, we propose parallel optimization methods using the features of the GPU. However, in this case, parallel operation may be performed not just by executing the same function in multiple threads in parallel but also including cases in which multiple threads need to perform different operations. In this paper, we present possible problems or considerations in a situation in which each thread needs to perform a different role when performing matrix multiplication or the sampling process for error values.

2. Suggesting Optimization Methods for a GPU Called from a CPU
   We did not optimize the entire algorithm and suggest optimization methods for each operation by grasping the load for each operation inside the algorithm. Methods of optimizing only the internal time-loaded computation while using the CPU are proposed, and methods are proposed to improve the memory copy time that inevitably occurs between the CPU and the GPU through Nvidia's CUDA stream and the thread indexing method. In conclusion, the method of optimization using the CPU and the GPU at the same time can be actively run in an environment in which a large amount of data must be encrypted in real time, such as an IoT server.
3. Optimizing GPU Computation Using Shared Resources
   Threads that perform operations in parallel on the GPU share memory areas block by block and are called shared memory. Shared memory is faster than global memory, so it is easy to store and use the data and temporary results commonly used by each thread. Therefore, not only basic parallel optimization methods using the GPU but also shared memory were used to provide better performance results. In addition, a warp optimization method is proposed, because the threads are executed in batches according to the bundle of threads, called the warp, rather than being performed as independent operations. In addition, the problem of bank conflict that may occur while using shared memory has been resolved, and the coalesced memory access method is implemented.
4. Presenting performance measurement results on various GPU platforms
   GPUs are divided into various architectures by generation, and each architecture has different structural and performance characteristics. In this paper, we present the performance measurement results for the GPUs of Pascal and Turing, two architectures of Nvidia. The Pascal architecture GTX 1070 and Turing architecture RTX 2070 were used in the experiment, and the results of optimization and performance analysis in each environment are presented.

The remainder of this paper is as follows. Section 2 presents optimization trends for target quantum-resistant cryptography. In Section 3, we explain the basics of the target algorithms and the optimization environment. Section 4 introduces the various optimization methodologies presented in this paper. Section 5 presents the performance improvement for each optimization method, and the paper concludes in Section 6.

2. Previous Implementation of Target PQCs

The LWE problem was introduced by [13]. It is a generalization of the parity learning problems. In [13,14], the LWE problem was used to create public-key cryptography. In addition, the LWE-based public-key encryption scheme was developed more efficiently by [15] using a public matrix $\boldsymbol{A} \in \mathbb{Z}_q^{n \times n}$ instead of a rectangular one.

FrodoKEM modifies pseudorandom generation of the public matrix $\boldsymbol{A}$ from a small seed to make more balanced key and ciphertext sizes [15]. The Frodo algorithm first appeared at the 2016 ACM CCS

conference [8]. In [16], an optimization method through the GPU for general LWE-based algorithms was proposed.

Lattice-based algorithms have the advantage of being faster than other algorithms, but they are still slower than existing public-key cryptography. However, few optimization methods for FrodoKEM, which is an LWE-based algorithm, have been proposed. In [17], FrodoKEM was optimized by applying ECC and gray codes when encoding the bits converted from the encrypted symbol.

In [18], an RLWE-based hard problem was proposed. In [19], an RLWE-based key exchange algorithm was proposed from a hardness assumption to create public-key cryptography. In [9], NewHope was suggested and modeled as a secure public-key encryption scheme in CPA resistance. In addition, research on efficient RLWE-based algorithms has been continuously conducted [20]. In [21], NewHope cryptography was optimized using a GPU on facial security.

3. Background

3.1. Overview of Lattice-Based Cryptography

Lattice-based cryptography refers to an algorithm that bases the basic components of cryptography on a computational lattice problem. Unlike the widely used and well-known public-key schemes such as RSA or ECC, which can theoretically be easily attacked by quantum computers, some lattice-based structures appear to be resistant to attack by classical and quantum computers. A lattice-based structure is considered safe under the assumption that it cannot effectively solve certain well-studied computational lattice problems.

The mathematical principle of the lattice-based cryptography is as follows. The LWE problem, with n unknowns, $m \geq n$ samples, modulo q and with error distribution X is as follows: for a random secret s uniformly chosen in $\mathbb{Z}_q^n$, and given m samples either all of the form ($\mathbf{a}$, b = <s,$\mathbf{a}$>+e mod q) where $e \overset{\$}{\leftarrow} X$, or from the uniform distribution ($\mathbf{a}$, b) $\overset{\$}{\leftarrow} U(\mathbb{Z}^n \times \mathbb{Z}_q)$, decide if the samples come from the former or the latter case [8]. Let R denote the ring $Z[X]/(X^n+1)$ for n a power of 2, and R_q the residue ring R/qR. The RLWE problem, with m unknowns, $m \geq 1$ samples, modulo q and with error distribution X is as follows: for a uniform random secret $\mathbf{s} \overset{\$}{\leftarrow} U(R_q)$, and given m samples either all of the form ($\mathbf{a}$, $\mathbf{b} = \mathbf{a} \cdot \mathbf{s} + \mathbf{e}$ mod q) where the coefficients of $\mathbf{e}$ are independently sampled following the distribution X, or from the uniform, distribution ($\mathbf{a}$, $\mathbf{b} \overset{\$}{\leftarrow} U(R_q \times R_q$), decide if the samples come from the former or the latter case [9].

The principle of lattice-based cryptography is as follows. When points are regularly arranged in a crisscross pattern in n-dimensional space R, this set of arranged points is called a lattice. The pattern of this lattice is determined by a specific basic vector. As the dimension increases, the texture of the lattice by the vector becomes more complicated. Grid-based cryptography is primarily based on the difficulties of the shortest vector problem (SVP) and the closest vector problem (CVP). SVP uses the principle that it is difficult to find the shortest vector in polynomial time using the basic vectors on the coordinates, and CVP uses the principle that it is difficult to find the nearest vector within the polynomial time using the basic vectors on the coordinates. It is easy to find the closest lattice point to an arbitrary position in a low-dimensional space, such as two dimensions, but it is difficult to efficiently find these lattice points on a quantum computer with hundreds of dimensions.

Lattice-based ciphers have the advantage of a faster operation speed compared with other PQC candidates but have the disadvantage that it is difficult to set parameters to satisfy security strength. Lattice-based cryptography began in 1996, when it was implemented in a public-key cryptosystem [22]. In 2005, public-key cryptography based on the LWE problem was proposed [13], and safety was proved under worst-case hardness assumptions. As research to improve lattice-based cryptography continued, RLWE-based cryptography was proposed, and in 2009, homogeneous cryptography was introduced.

Table 1 lists the lattice-based ciphers to be judged for round 2 of the NIST PQC standardization competition.

Table 1. List of lattice-based encryption and key encapsulation mechanisms (KEMs) [4].

Encryption/KEMs	Type (Lattice-Based)
Crystals-Kyber	MLWE
KINDI	MLWE
Saber	MLWR
FrodoKEM	LWE
Lotus	LWE
Lizard	LWE/RLWE
Emblem/R.emblem	LWE/RLWE
KCL	LWE/RLWE/LWR
Round 2	LWR/RLWR
Hila5	RLWE
Ding's key exchange	RLWE
LAC	RLWE
Lima	RLWE
NewHope	RLWE
Three Bears	IMLWE
Mersenne-756839	ILWE
Titanium	MP-LWE
Ramstake	LWE like
Odd Manhattan	Generic
NTRU Encrypt	NTRU
NTRU-HRSS-KEM	NTRU
NTRUprime	NTRU

3.2. Target NIST Round 2 LWE-Based PQC

LWE-based problems are based on hard problems using errors. From a linear algebra point of view, it is easy to find a matrix S that satisfies $B = AS$ given two matrices A and B. This is because the inverse matrix can be used to simply obtain $S = A^{-1}B$. However, if we want to find an S that satisfies $B = AS + E$ by adding a small error to the AS, those who know the error value can still calculate the S value easily. In contrast, those who do not know the error should count the number of all cases of the added error. In this case, it becomes difficult to find the S value. As the dimension of the matrix grows larger, the number of cases to be counted increases exponentially, even for small errors. This problem is known to be difficult to solve even with a quantum computer.

RLWE is an ideal lattice-based cryptographic technique made more efficient by reducing the key size or speeding up the cryptographic operation compared with the general lattice-based cryptographic technique. By applying the LWE technique on the ring, two integers A and B on the ring are searched for S according to the added error value. In RLWE, Toom–Cook, Karatsuba, or NTT-based multiplication is performed to efficiently calculate the multiplication of integers.

The LWE-based PQC candidates targeted in this paper are FrodoKEM and NewHope. FrodoKEM and NewHope are both round 2 target algorithms of the NIST PQC standardization competition, and many studies are actively being conducted, so this paper is expected to help other studies if these algorithms are optimized. In this section, we look at the operation structure of target algorithms.

3.2.1. FrodoKEM

The core of FrodoKEM is FrodoPKE, a public key encryption algorithm presented at the 2016 Association for Computing Machinery (ACM) Computer and Communications Security (CCS) conference [8]. FrodoKEM can be largely divided into three types: FrodoKEM-640, FrodoKEM-976, and FrodoKEM-1344. The overall process consists of key generation, encapsulation, and decapsulation. In this paper, we propose some methods to optimize the encryption process in FrodoPKE. Variable information according to the type of FrodoKEM is shown in Table 2. In Table 2, the unit of a key size is a byte. The notation of FrodoPKE is described in Table 3.

Table 2. Variable Information of FrodoKEM.

Type	FrodoKEM-640	FrodoKEM-976	FrodoKEM-1344
Dimension n	640	976	1344
Modulus q	2^{15}	2^{16}	2^{16}
Public key size	9616	15,632	21,520
Private key size	19,888	31,296	43,088
Hash function	SHAKE-128	SHAKE-256	SHAKE-256

Table 3. Notation of FrodoPKE.

Symbol	Description
$q = 2^D$	Powers of 2 with exponent $D \leq 16$
$n, \bar{n}, \bar{m}$	$n \equiv 0$ mod 8 integer matrix dimension
$B \leq D$	Number of bits encoded in each matrix component
$l = B \times \bar{m} \times \bar{n}$	Length of the bitstream encoded by the $\bar{m}$-by-$\bar{n}$ matrix
len_{seed_A}	Bit length of the seed used to generate a pseudorandom matrix.
$len_{seed_{SE}}$	Bit length of the seed used to generate a pseudo-random matrix for error sampling
$T_{\mathbf{x}}$	Cumulative distribution table for sampling

The encryption processes of FrodoPKE are described in Algorithm 1 and Figure 1. These algorithms and figures show the IND-CPA-secure public key encryption scheme process in FrodoPKE.

Algorithm 1 FrodoPKE Encryption [8].

Require: Message $\mu \in M$ and public key $pk = (\boldsymbol{seed_A}, \boldsymbol{B}) \in \{0,1\}^{len_{\boldsymbol{seed_A}}} \times \mathbb{Z}_q^{n \times n}$
Ensure: Ciphertext $c = (\boldsymbol{C}_1, \boldsymbol{C}_2) \in \mathbb{Z}_q^{\bar{m} \times n} \times \mathbb{Z}_q^{\bar{m} \times \bar{n}}$
1: $\boldsymbol{A} \leftarrow$ Frodo.Gen($\boldsymbol{seed_A}$)
2: $\boldsymbol{seed_{SE}} \leftarrow_{\$} U(\{0,1\}^{len_{seed_{SE}}})$
3: $(\boldsymbol{r}^{(0)}, \boldsymbol{r}^{(1)}, \ldots, \boldsymbol{r}^{(2\bar{m}n+\overline{mn}-1)}) \leftarrow \text{SHAKE}(0\text{x}96||\boldsymbol{seed_{SE}}, 2\bar{m}n + \overline{mn} \cdot len_x)$
4: $\boldsymbol{S'} \leftarrow \text{Frodo.SampleMatrix}((\boldsymbol{r}^{(0)}, \boldsymbol{r}^{(1)}, \ldots, \boldsymbol{r}^{(\bar{m}n-1)}), \bar{m}, n, T_x)$
5: $\boldsymbol{E'} \leftarrow \text{Frodo.SampleMatrix}((\boldsymbol{r}^{(\bar{m}n)}, \boldsymbol{r}^{(\bar{m}n+1)}, \ldots, \boldsymbol{r}^{(2\bar{m}n-1)}), \bar{m}, n, T_x)$
6: $\boldsymbol{E''} \leftarrow \text{Frodo.SampleMatrix}((\boldsymbol{r}^{(2\bar{m}n)}, \boldsymbol{r}^{(2\bar{m}n+1)}, \ldots, \boldsymbol{r}^{(2\bar{m}n+\overline{mn}-1)}), \bar{m}, \bar{n}, T_x)$
7: $\boldsymbol{B'} = \boldsymbol{S'}\boldsymbol{A} + \boldsymbol{E'}$ and $\boldsymbol{V} = \boldsymbol{S'}\boldsymbol{B} + \boldsymbol{E''}$
8: ***return*** ciphertext $c \leftarrow (\boldsymbol{C}_1, \boldsymbol{C}_2) = (\boldsymbol{B'}, \boldsymbol{V}+\text{Frodo.Encode}(\mu))$

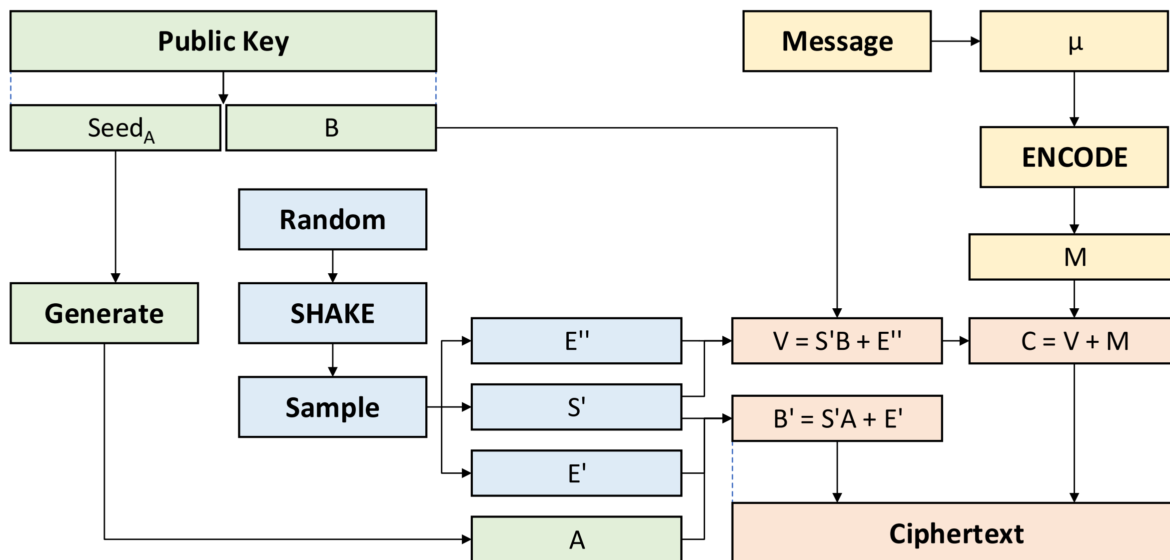

Figure 1. FrodoPKE encryption process.

The functions used consist of a *randombytes* function that generates random values, generates a matrix function that generates matrix A, the SHA-3-based extendable-output function $SHAKE$ [23], matrix multiplication and addition or subtraction operations, a Gaussian sampling function ($Frodo.SampleMatrix$), and an encoding and decoding function.

Looking at the FrodoKEM reference code provided by NIST, the *randombytes* function for generating random values uses the *BCryptGenRandom* function, an application programming interface (API) provided by Microsoft. When increasing or decreasing the data by a desired length, the SHA-3 extendable-output functions $SHAKE$-128 or $SHAKE$-256 are used depending on the key length. The *sample* function is responsible for generating errors on the LWE basis. In the case of FrodoKEM, sampling is performed using *CDF_Table* and the cumulative distribution function. The matrix A used for matrix operation $B = AS + E$ is created by selecting either the advanced encryption standard (AES) or $SHAKE$. The *encode* function replaces data with matrix data during encapsulation. Substituted matrix data are transformed into existing data through a *decode* function in a decapsulation process. In addition, there are various addition, subtraction, and multiplication operations on the matrix.

After the first key generation process of FrodoPKE, the public-key and the private key are generated. The matrix operation is also used in the encryption process. In this process, the $SHAKE$ function for hashing data is called continuously. The encryption process includes the processes of encoding data and generating a ciphertext.

3.2.2. NewHope

NewHope is an RLWE-based algorithm. Whereas FrodoKEM is a matrix-based operation, NewHope is mainly an integer operation on a ring. It can be largely divided into two types: NewHope-512 and NewHope-1024. The overall process consists of key generation, encryption, and decryption. Variable information according to the type of NewHope is shown in Table 4. In Table 4, the unit of a key size is a byte. The notation of FrodoPKE is described in Table 5. Since this paper aims to optimize the performance of encryption, NewHope, like FrodoKEM, is described based on the NewHope-CPA-PKE, which includes the encryption process.

Table 4. Variable information of NewHope.

Type (CPA-KEM)	NewHope-512	NewHope-1024
dimension n	512	1024
modulus q	12,289	12,289
NTT parameter γ	10,968	7
Public key size	928	1824
Private key size	896	1792

Table 5. Notation of NewHope.

Symbol	Description
n	Dimension of polynomial rings
μ	256-bit message represented as an array of 32 bytes
$coin$	True random generated value that used for sampling function
$\hat{b}, \hat{a}, \hat{s}, e', e'', \hat{t}, \hat{u}, v, v'$	Polynomial in $R_q = Z_q[\mathbf{X}]/(\mathbf{X}^n + 1)$
$publicseed$	Seed value that generates polynomial $\hat{a}$
PolyBitRev	Transform function to process NTT-based multiplication
Compress	Function that generates array h by switching between modulus q and modulus 8

The encryption processes of NewHope are described in Algorithm 2 and Figure 2. The main operation of NewHope is a ring-based polynomial operation. Among them, NTT-based multiplication is composed of NewHope's main operations.

Looking at the NewHope reference code provided by NIST, as in FrodoKEM, random values are generated through the $randombytes$ function. In NewHope, random values are generated using $CTR_DRBG(AES-256)$, which is a deterministic random bits generator using the AES-256 block cipher algorithm. The extendable-output function uses SHA-3's $SHAKE$-128 and $SHAKE$-256. The seeds generated through the $randombytes$ function and the $SHAKE$ function are used to generate $\hat{a}$, s, and e to calculate $\hat{b} = \hat{a} \circ \hat{s} + \hat{e}$, where NTT conversion is performed to quickly calculate $\hat{a} \circ \hat{s}$.

Algorithm 2 NewHope Encryption [9].

Require: $pk \in \{0,\ldots,255\}^{7\cdot n/4+32}$, $\mu \in \{0,\ldots,255\}^{32}$, $coin \in \{0,\ldots,255\}^{32}$
Ensure: Ciphertext c
1: $(\hat{\boldsymbol{b}}, publicseed) \leftarrow$ DecodePk(pk)
2: $\hat{\boldsymbol{a}} \leftarrow$ GenA$(publicseed)$
3: $\boldsymbol{s}' \leftarrow$ PolyBitRev(Sample$(coin, 0)$)
4: $\boldsymbol{e}' \leftarrow$ PolyBitRev(Sample$(coin, 1)$)
5: $\boldsymbol{e}'' \leftarrow$ Sample$(coin, 2)$
6: $\hat{\boldsymbol{t}} \leftarrow$ NTT$(\boldsymbol{s}')$
7: $\hat{\boldsymbol{u}} \leftarrow \hat{\boldsymbol{a}} \cdot \hat{\boldsymbol{t}} +$ NTT$(\boldsymbol{e}')$
8: $\boldsymbol{v} \leftarrow$ Encode(μ)
9: $\boldsymbol{v}' \leftarrow \mathrm{NTT}^{-1}(\hat{\boldsymbol{b}} \cdot \hat{\boldsymbol{t}}) + \boldsymbol{e}'' + \boldsymbol{v}$
10: h = Compress$(\boldsymbol{v}')$
11: ***return*** c = EncodeC$(\hat{\boldsymbol{u}}, h)$

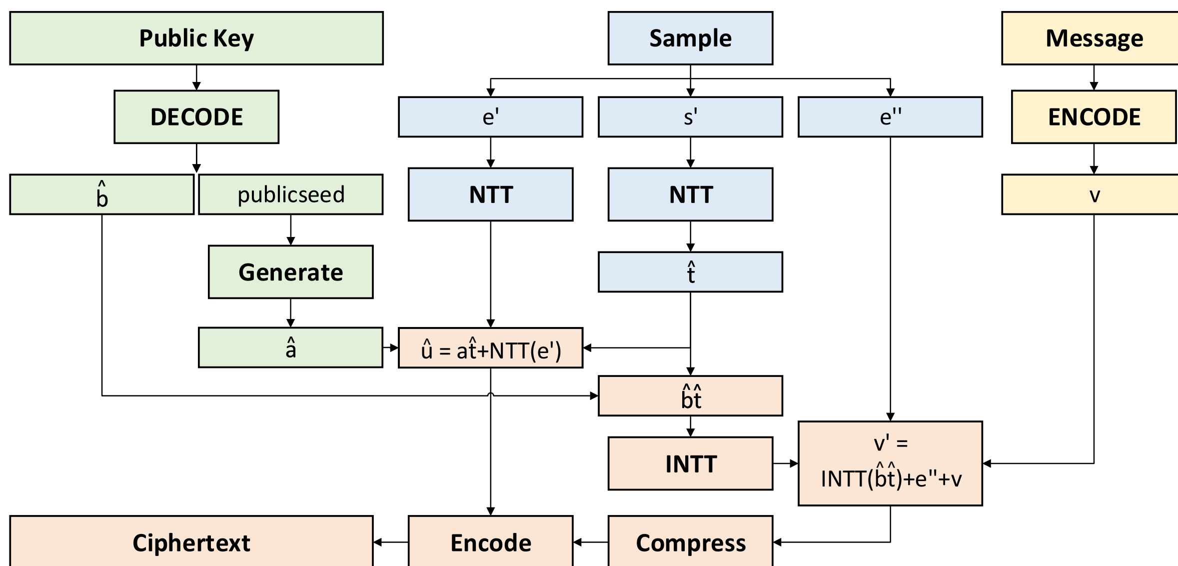

Figure 2. NewHope encryption process.

NTT-based multiplication [10] is an algorithm that can efficiently multiply two large numbers, such as Toom–Cook or Karatsuba multiplication. NTT is based on fast Fourier transform. NTT-based multiplication consists of obtaining $c = NTT^{-1}(NTT(a) \circ NTT(b))$ for $a, b, c \in \mathbb{R}$, and it is implemented by defining ideal lattice $\mathbb{R}_q = \mathbb{Z}_q[\mathtt{X}]/(\mathtt{X}^n + 1)$ for efficient NTT-based multiplication. In this way, the transformed coefficient-wise multiplication of two numbers through NTT conversion is the same as the multiplication result. However, it is necessary to process the reduction after the multiplication. NewHope uses the Montgomery reduction algorithm [24] for fast reduction.

The values generated in the key generation process are encoded and made into a private key and a public-key. In the encryption process, these values are first decoded to perform encryption.

3.3. Structure of Nvidia GPU Platform

In this paper, we propose several methods to optimize FrodoKEM and NewHope using a GPU. Therefore, the structure and characteristics of the GPU are explained to describe the principles of the proposed optimization methods.

A GPU is a kind of electronic circuit and is designed to accelerate the generation of images in the frame buffer to process and change memory quickly and output it to a screen. Modern GPUs handle computer graphics and image processing effectively. The reason for this performance is the structure of the GPU. If the CPU is performing calculations through a small number of high-performance cores, the GPU performs calculations using a number of generic cores. GPUs specialize in processing specific operations in parallel using multiple cores. The advantage of this GPU feature is that it can be performed quickly via parallel processing on the CPU through the GPU.

The GPU was originally developed to perform graphics tasks, but general-purpose computing on a GPU technique was later proposed that uses the GPU to compute applications for which the CPU does. This allows software developers to use the GPU for computations such as encryption rather than graphics tasks. CUDA is a parallel computing platform and API model developed by Nvidia [25]. As with programming written in C language using CUDA, GPU operations can be easily coded using CUDA libraries. Since CUDA was released in 2007, new CUDA versions have been developed as GPU technology evolved. With the development of a new GPU architecture and CUDA version, it is possible to use various memory controls and computations. However, previously developed GPUs do not support the latest CUDA version.

When programming a GPU using CUDA, the GPU receives commands from the CPU through a function called a kernel. To use the data used by the CPU in the GPU, the CPU data must be copied to the GPU before operation of the kernel. For example, when the block encryption algorithm is performed in parallel through the GPU, plaintext data existing in the CPU must be copied to the GPU in advance to encrypt the plaintext data in the GPU. In addition, if the encrypted data are to be used again in the CPU, an additional process of copying the ciphertext from the GPU to the CPU is required. The copying process between the CPU and the GPU is transmitted through peripheral component interconnect express (PCIe), but the PCIe transfer speed is slow compared with the individual speeds of the CPU and the GPU. Many studies have been conducted to reduce the data copying time between the CPU and the GPU through various optimization methods, because such a copying process causes a lot of load over time.

GPUs have some differences in architecture. The most recent Nvidia GPU architecture is the Turing architecture, which has 4352 CUDA cores for the RTX 2080 Ti, the flagship graphics card model of the Turing architecture. Table 6 shows the features of each Nvidia GPU architecture.

The GPU is composed of several streaming multiprocessors (SMs), and each SM is composed of several cores and memory. If the GPU operation structure is largely divided, it is composed of several grids, with several blocks to be drawn. These blocks are made of several threads. Because many of these threads perform individual operations in parallel, they have a fast operation speed. Each thread is used separately by dividing the register memory space, and threads in the block can be accessed jointly through shared memory. Global memory is a common memory that all threads can access. Global memory that uses the dynamic random-access memory space has a large memory size but has the disadvantage of being slow. Registers have fast access but small capacity. Furthermore, it consists of many memory types, such as GPU memory, constant memory, texture memory, and local memory. Figure 3 shows the Nvidia GPU memory structure.

Table 6. Features of the latest Nvidia graphics processing units (GPU) architectures.

Architecture	Maxwell	Pascal	Turing
Flagship GPU	GTX 980 Ti	GTX 1080 Ti	RTX 2080 Ti
SM count	24	28	68
Core count	2816	3584	4352
Memory Size	6 GB	11 GB	11 GB
Base clock	1000 MHz	1480 MHz	1350 MHz
CUDA compute capability	5.2	6.1	7.5

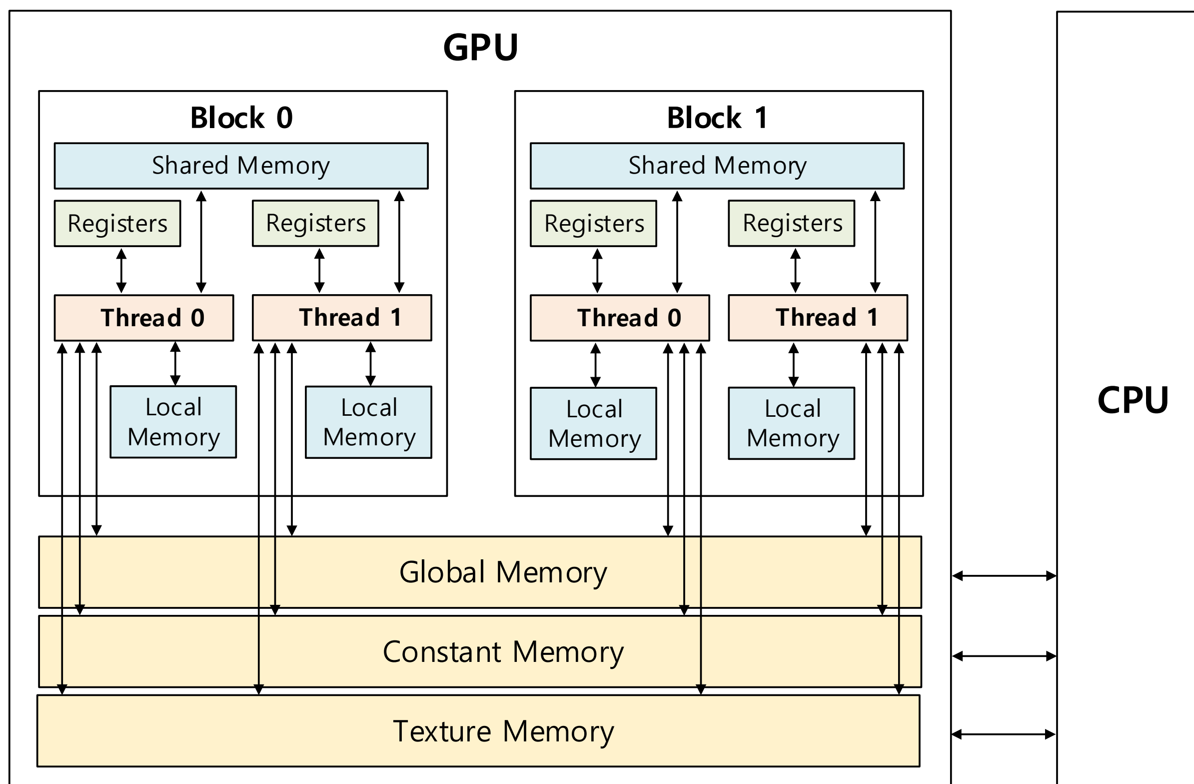

Figure 3. Nvidia GPU memory structure.

4. Proposed Optimization on Lattice-Based PQC in a GPU Environment

This section presents optimization methods for the main operations of the target algorithms FrodoKEM and NewHope. Using the characteristics of the GPU, we explain the parts to improve the performance by heavy operation in parallel and introduce the implementation method.

4.1. Common Optimization Method Using the GPU

Even if the same operation is performed on a large amount of data, operating in series versus in parallel causes a significant speed difference. In particular, FrodoKEM and NewHope, which have large data sizes, need to perform the same calculation for a certain size of data over thousands of bytes. Therefore, if it is possible to perform a fast operation for each algorithm through parallel processing, the overall speed of the algorithm can be improved.

Tables 7 and 8 measure the computational load specificity for the target algorithms. The unit of time is microseconds, and the percentage value to the right of the time represents the percentage of the total computation time.

Table 7. Percentage of computational load in FrodoPKE encryption.

	FrodoKEM-640		**FrodoKEM-976**		**FrodoKEM-1344**	
Randombytes	20	0.12%	13	0.04%	12	0.02%
Generates $\boldsymbol{A}$ matrix	14,960	92.59%	33,794	92.87%	64,512	93.20%
Matrix Multiplication	1040	6.44%	2318	6.37%	4368	6.31%
Sampling	42	0.26%	78	0.21%	72	0.10%
SHAKE	96	0.59%	184	0.51%	252	0.36%

Table 8. Percentage of computational load in NewHope encryption.

	NewHope-512		**NewHope-1024**	
Randombytes	101	1.77%	151	1.16%
Generates $\boldsymbol{A}$	583	10.24%	1210	9.28%
NTT Multiplication	1728	30.36%	3713	28.49%
Sampling	2837	49.85%	6951	53.33%
encoding	232	4.08%	540	4.14%
SHAKE	210	3.69%	468	3.59%

It can be seen that FrodoKEM's time load is mostly used for the AES operation to generate matrix A. In addition, it was confirmed that the time consumption was large in matrix multiplication. Therefore, FrodoKEM focuses on optimizing the matrix A generation process. If the optimization for AES used for matrix A generation is performed, the overall operation speed may be increased.

In NewHope, the sampling and NTT-based multiplication processes took up most of the computation time. Therefore, in the processes of sampling and NTT-based multiplication, performance improvement can be achieved by optimizing the part that can process each component of the polynomial in parallel. The proposed GPU parallel optimization method was implemented to operate only the main operations with the GPU while processing the algorithm around the CPU.

4.2. Optimization Methods of FrodoKEM on GPU Platform

Because FrodoKEM operates on data with thousands to tens of thousands of bytes, the time load for each operation is high. The main operations of FrodoKEM are *randombytes*, $SHAKE$, AES, matrix multiplication, and sampling. Among them, AES operation for generating matrix A is the most time intensive. AES occupies the most FrodoKEM operating time because the size of the matrix to be generated reaches several thousand bytes, compared with generating a 16-byte ciphertext once encrypted. Therefore, if the AES operation is performed in parallel using a GPU, significant performance can be improved.

AES is used to generate pseudorandom matrix A from seed $seed_A$ during the key generation, encapsulation, and decapsulation processes of FrodoKEM.

In FrodoKEM's A matrix generation process, the number of AES calls to create matrix A is called $n^2/8$ times according to dimension parameter n, which means 51,200, 119,072, 225,792 times in FrodoKEM-640, FrodoKEM-976, and FrodoKEM-1344, respectively. However, on GPU, $n^2/8$ threads can create matrix A by calling AES each once.

In the parallel matrix generation process, the 16-byte data input to the AES contains only 2 bytes of row information and 2 bytes of column information; the remaining 12 bytes contain all 0-padded

data. Therefore, if each thread is responsible for a matrix component corresponding to a unique row and column component using a thread identifier value, which is the unique number of each thread, the process of creating a matrix A with AES can be parallelized. The GPU parallelization process using matrix components is shown in Figure 4.

By dividing the AES for creating an $n \times n$ matrix by 16 bytes, each thread performs AES once. Because the value that goes into the input of AES can be generated independently through the unique index of the thread and block, there is no need to copy inputs from the CPU to the GPU. Because the memory copy time between the CPU and the GPU is long, it is efficient to eliminate the memory copy time using the thread and block index. However, the result of AES performed on the GPU must be copied from the GPU to the CPU. The CUDA stream can be used to reduce the copy time as much as possible. Conventionally, when the process of copying the memory from the CPU to the GPU ends, the GPU kernel operates, and when the kernel ends, the GPU memory is copied to the CPU. However, the CPU is idle while the GPU is performing the operation. When using the CUDA stream, each stream performs memory copy and kernel operations in parallel. Because the CPU copies the memory of the second stream immediately after the memory copy of the first stream is finished, the idle state of the CPU generated while the GPU is operating can be reduced.

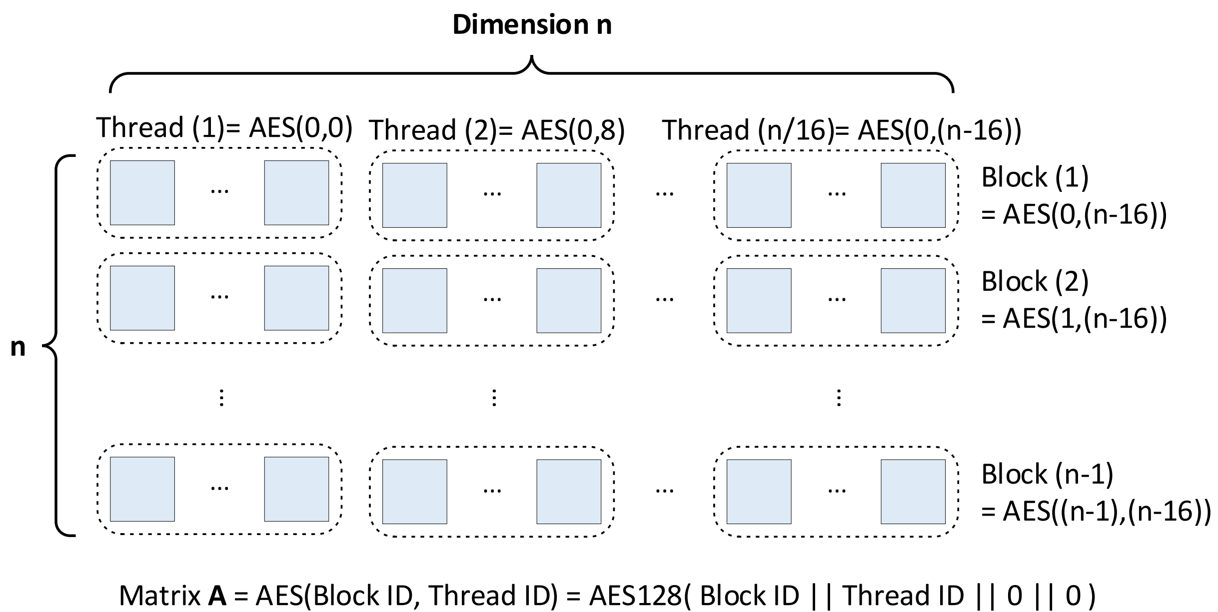

Figure 4. Matrix A parallel generation process.

Because the key value is $seed_A$, the AES of all matrices has the same round key; therefore, memory can be efficiently used if the pregenerated key is stored in the memory shared by threads. However, the global memory of the GPU has a disadvantage in that it is slow. Therefore, the key shared by the threads in the block is used to store the round key, and then the thread can access and use the shared memory. Basically, the higher the dimension of FrodoKEM, the slower it is proportionally, because the CPU has to call more AES. However, because the parallel optimization method on the GPU performs one AES per thread in parallel, the efficiency of the optimization increases: the GPU only needs to call the AES once, using as many threads as the AES needs to be called.

Another major time-consuming operation in FrodoKEM is the matrix multiplication operation. The matrix size is proportional to the dimension, so it takes a significant amount of time to multiply the $n \times n$ matrix A by the $n \times \bar{n}$ matrix S. On the GPU, this matrix multiplication process can be optimized using

threads. In FrodoKEM, $\bar{n}$ is always fixed at 8. In this case, to obtain the 8×8 block of the result matrix B, as much as $8 \times N$ data in matrix A and the entire S matrix are needed. That is, because matrix S is continuously used in all parts, if matrix S is stored in shared memory, a faster speed can be obtained than when it is accessed from global memory. By dividing $n \times \bar{n}$ matrix A by the number of blocks and threads, and multiplying and accumulating the results by referring to their matrix positions, each thread performs one multiplication process and one addition process. The parallel matrix product processes can be seen in Figure 5.

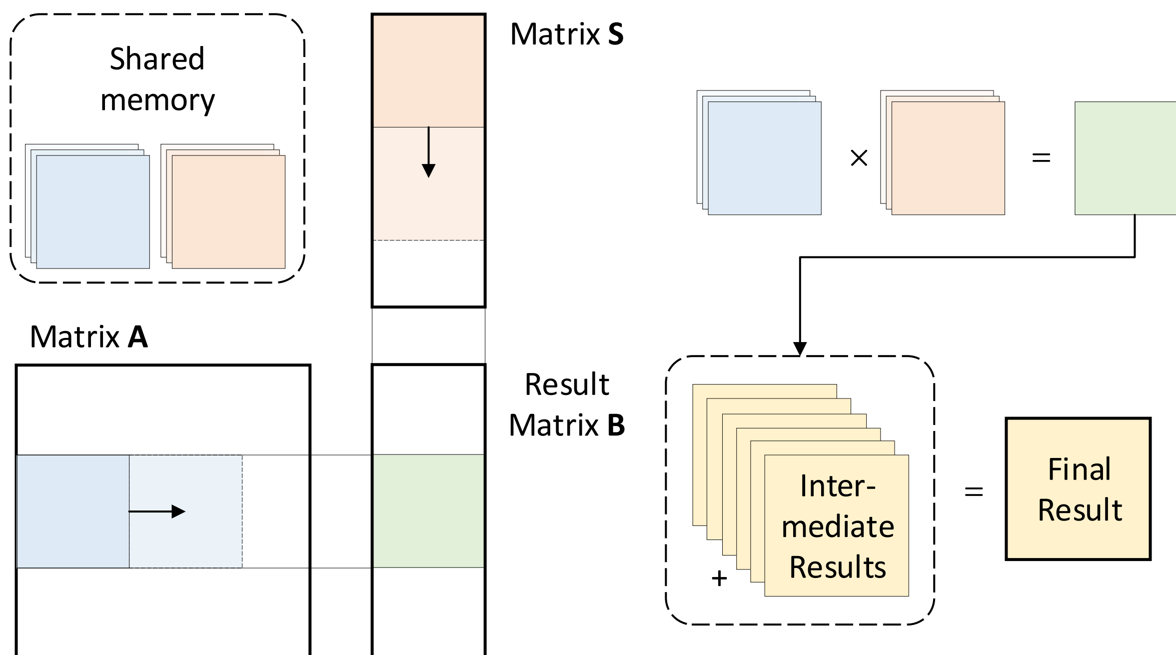

Figure 5. Parallel matrix product process.

To perform matrix multiplication for an $n \times n$ matrix and an $n \times 8$ matrix in the process of FrodoKEM's matrix multiplication, component multiplication and addition by $n^2/8$ times are required. In addition, $8n$ times of component additions are required when performing matrix addition on the result of matrix multiplication. In GPU, this process is performed by a bundle of 64 threads independently performing a matrix multiplication operation every 8 rows of an $n \times n$ matrix. Accordingly, matrix multiplication can be processed by performing $n/8$ times of component multiplication and addition operations for each thread.

However, this implementation creates a problem in that each thread must access the memory where the result is stored at every calculation. This creates serialization between threads, which slows the implementation considerably. Therefore, access to the same result memory can be reduced by creating 8×8 shared memory to temporarily store the result and adding all data in the shared memory after the calculation is complete.

4.3. Optimization Methods of NewHope on the GPU Platform

The main operation of NewHope, NTT, is a multiplication method that can reduce the time complexity for performing $n \times n$ multiplication from $O(n^2)$ to $O(n \log n)$. The NTT-converted data are calculated in a manner similar to dot product between coefficients, where the multiplication, addition, and reduction processes between coefficients can be performed in parallel. Figures 6 and 7 show the polynomial operation that can be performed in parallel.

Multiplication in polynomials with NTT transformation is calculated through dot multiplication. In this process, a dot product operation is performed on an array of 512 and 1024 polynomial coefficients for each of NewHope-512 and NewHope-1024 according to dimension n. In GPU, each thread can compute an operation on each array. By using n threads, it is possible to speed up the operation with one calculation.

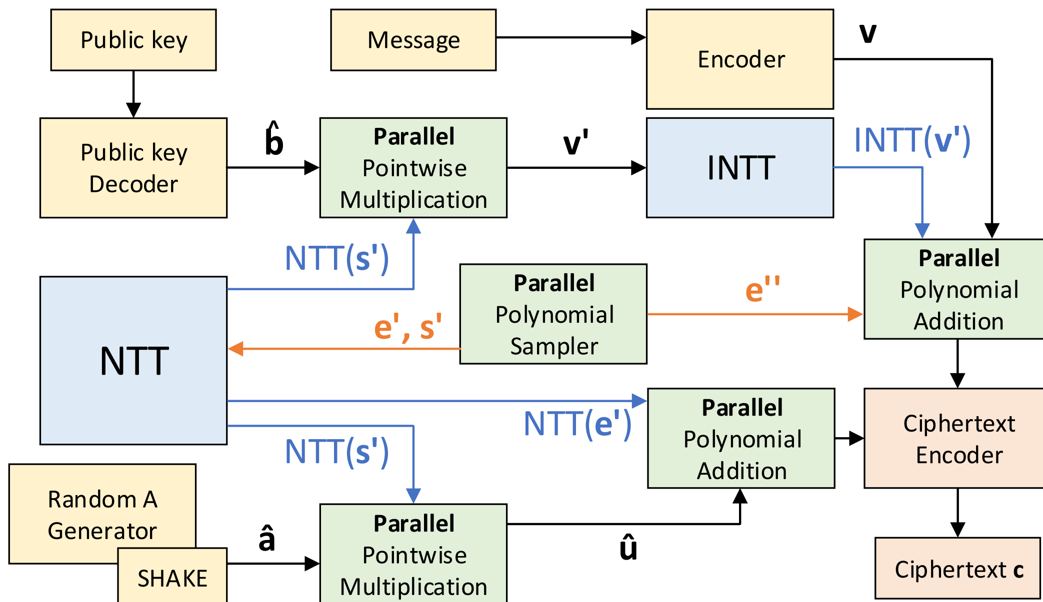

Figure 6. NewHope encryption with partial parallelism.

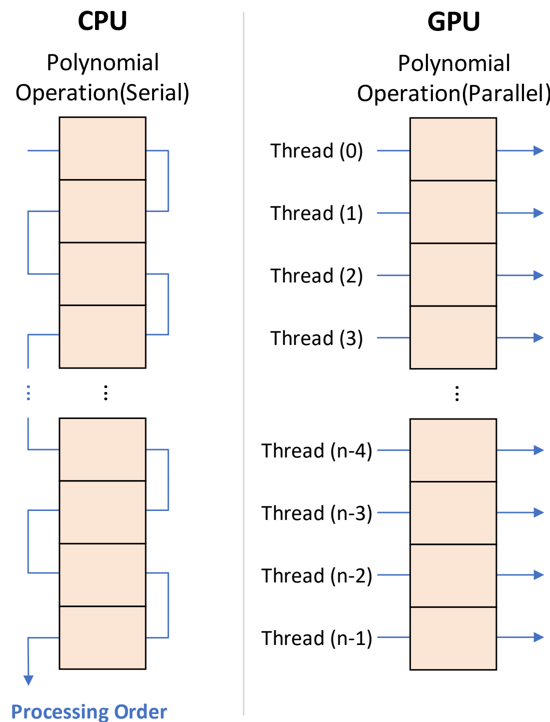

Figure 7. Parallel polynomial operation process.

Inside the sampling process at NewHope, the extractor seed is set as the input of the $SHAKE$ function. The extractor seed consists of the input seed, nonce value, and i value padded, and i is sequentially increased from 0 to 7 or 15 depending on the NewHope parameter. Because the input seed and nonce values are the same, the value of extract seed varies depending on the value of i, and these values are set as input values of the $SHAKE$ function. Therefore, the $SHAKE$ function that does not need to be calculated by all threads is calculated by the warp, which is a bundle of threads, and shared in shared memory. For data stored in shared memory, each thread takes a polynomial coefficient and can sample using the result of $SHAKE$ stored by the warp.

In addition, each thread of the GPU performs functions to calculate each term of the polynomial in parallel, reducing computation time. Polynomial point-wise multiplication, addition, and Montgomery subtraction can be implemented to ensure that all threads perform the same instruction. However, in the sampling function, in which threads must perform different operations, threads in the same warp are implemented to perform the same operation and thus prevent the thread serialization problem because of control divergence. Control divergence refers to a situation in which different threads wait until the end of one conditional statement, when threads execute different operations because of a branch such as an if statement in a warp. Therefore, in the case of the sampling $SHAKE$ function, 32 threads in one warp perform calculations only with different parameter values and store the result value in shared memory, and then the entire thread in the block performs parallel sampling using the value to be implemented.

In these optimization methods, memory storage and access coalesce to allow multiple memory types to have effective memory access. When multiple threads access memory, the necessary threads access the memory by referring to an area, not by thread. However, if the data are not coalesced and stored, and if the thread does not have access to the desired data in one instance of accessing the memory area, it causes a time decrease because additional memory area access is required.

5. Experimental Results

The platform environment in which the experiment was measured is shown in Table 9. Performance was measured, including memory time between the CPU and the GPU. The experimental results show performance improvement when the GPU optimization method is applied based on the result of running the reference source code of the submission in the experimental environment in round 2 of the existing NIST PQC standardization competition.

Table 9. GPU optimization implementation test environment.

CPU	**Intel Core i7-9700K**	**Intel Core i7-9700K**
GPU	**GTX 1070**	**RTX 2070**
GPU Architecture	Pascal	Turing
CUDA Core count	1920	2340
GPU Memory Size	GDDR5 8 GB	GDDR6 8 GB
GPU Base clock	1506 MHz	1410 MHz
OS	Windows 10	Windows 10
CUDA Version	10.2	10.2

The experimental results were presented in units of microseconds and were based on the average of the times measured while repeating the entire encryption process 1000 times. The time for the major operations was measured from before data is copied from the CPU to the GPU until the data is copied again after the GPU operation is completed. Each experiment result was measured by replacing only two GPUs on the same CPU and OS. During the experimental measurement, we used the test vector to verify the optimization implementation result. If there is interference from the outside such as jamming or an

error occurs inside due to an abnormality in the device, it is implemented so that the presence or absence of an error can be determined using an error correction code.

5.1. Performance Improvement on FrodoKEM

This section presents the optimization results of matrix A generation and matrix multiplication, which are the main operations of FrodoKEM, and the optimization results of the entire FrodoPKE encryption process according to GPU optimization implementation.

Tables 10 and 11 show the results of optimizing the main operation of FrodoKEM using the GPU. Each figure shown in the table represents microseconds, and the percentage represents a performance improvement in the GPU compared with the CPU.

Table 10. Operation time (µs) and performance improvements by optimized matrix A generation and matrix multiplication in FrodoKEM (GTX 1070).

	Matrix A Generation			Matrix Multiplication		
	CPU	**GPU**	**Improvement**	**CPU**	**GPU**	**Improvement**
FrodoKEM-640	14,960	3877	286%	1040	873	19%
FrodoKEM-976	33,794	7440	354%	2318	1911	21%
FrodoKEM-1344	64,512	13,254	387%	4368	3125	39%

Table 11. Operation time (µs) and performance improvements by optimized matrix A generation and matrix multiplication in FrodoKEM (RTX 2070).

	Matrix A Generation			Matrix Multiplication		
	CPU	**GPU**	**Improvement**	**CPU**	**GPU**	**Improvement**
FrodoKEM-640	14,960	2167	590%	1040	785	32%
FrodoKEM-976	33,794	4463	657%	2318	1554	49%
FrodoKEM-1344	64,512	7847	722%	4368	2519	73%

In the matrix A generation operation, the GPU calls AES in parallel, resulting in a significant performance improvement. In terms of the computation speed inside the GPU kernel, it shows thousands of times faster than the CPU implementation. However, to perform calculations on the GPU, data must be copied from the CPU to the GPU in advance. This additional computational load has the disadvantage of greatly reducing the GPU computational efficiency in a real environment.

In RTX 2070, as a result of implementing the operation to generate the matrix A in parallel through the GPU and optimizing it using shared memory, it was confirmed that it was 6.9, 7.57, and 8.22 times faster than the existing CPU for FrodoKEM-640, FrodoKEM-976, and FrodoKEM-1344, respectively. This speed was measured based on time including the memory copy time between the CPU and the GPU. In addition, as a result of performing the matrix multiplication operation using a GPU, performance improvement of 32%, 49%, and 73% was confirmed for FrodoKEM-640, FrodoKEM-976, and FrodoKEM-1344, respectively, compared with the existing CPU implementation. In a matrix multiplication kernel, 5120, 7808, 10,752 threads multiply each block stored in shared memory for FrodoKEM-640, FrodoKEM-976, and FrodoKEM-1344, respectively. In GTX 1070, 3.86, 4.54, and 4.87 times faster speed was confirmed for matrix A generation function in FrodoKEM-640, FrodoKEM-976, and FrodoKEM-1344, respectively. In a matrix multiplication function, performance improvement of 19%, 21%, and 39% has been shown. When only kernel time was measured, excluding memory copy time, FrodoKEM-640's matrix A generation kernel was confirmed that a throughput of 164 and 279 Gbps was achieved in GTX 1070 and RTX 2070, respectively. In Figure 8, FrodoPKE's total encryption time is

presented by each platform. In RTX 2070, 5.20, 5.78, and 6.47 times faster speed than CPU implementation was confirmed for the whole encryption process in FrodoKEM-640, FrodoKEM-976, and FrodoKEM-1344, respectively. In GTX 1070, 3.29, 3.78, and 4.14 times faster speed than CPU implementation was confirmed for the whole encryption process in FrodoKEM-640, FrodoKEM-976, and FrodoKEM-1344, respectively.

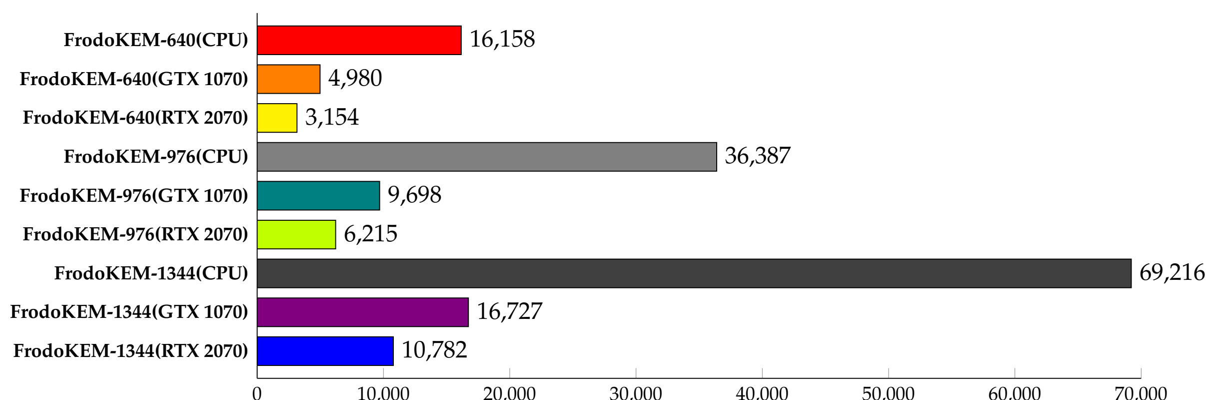

Figure 8. Comparison of operation time (μs) for FrodoPKE encryption between CPU implementation and GPU implementation including memory copy time.

The larger the dimension n, the longer the length that the CPU needs to process serially, but the GPU only needs to increase the available threads, so the performance improvement increases even more.

In the case of matrix A generation operation, the performance improvement number was high because it was optimized so that all threads perform the operation only once. However, in the matrix multiplication optimization implementation, each thread was implemented to perform $n/8$ component multiplication operations, so the performance improvement was not greater than the matrix A generation operation. It could have been implemented so that each thread could perform component multiplication only once by dividing $n/8$ operations again, but in that case, serialization problems caused by bank conflict that occur when different threads access the same memory bank.

5.2. Performance Improvement on NewHope

Like FrodoKEM, NewHope also has a very fast computing speed inside the GPU kernel, but due to the time load due to memory copying, it did not show a significant performance improvement in the actual encryption performance measurement. However, as the dimension n increases, the GPU only increases the number of threads that perform the same operation. Therefore, optimization performance improvements increase as dimension n increases.

In the case of the sampling function, a part of the inside of the sampling function has been optimized, but this part has a fixed ratio of the whole sampling function. Therefore, the sampling optimization performance improvement is not significantly affected by the GPU architecture type and dimension n.

This section presents the optimization results of NTT-based multiplication and polynomial operations, which are major operations of NewHope, and the optimization results of the entire encryption process according to GPU optimization implementation. Table 12 shows the results of optimizing the main operation of NewHope using the GPU. NewHope compared performance based on the time during which the same operation was repeated 100 times. In RTX 2070, NTT-based transformation and the point-wise multiplication process recorded 3.12 and 4.67 times faster speed in NewHope-512 and NewHope-1024, respectively, compared with the CPU implementation. In GTX 1070, 2.2 and 3.57 times faster speed in NewHope-512 and NewHope-1024 was confirmed. The 512 and 1024 byte arrays of polynomial work once each in the same number of threads for each NewHope-512 and NewHope-1024 encryption process.

NewHope's polynomial operation increases in proportion to the time load as the parameter grows larger on the CPU, but on the GPU, it only needs to increase the number of threads running simultaneously, so the time does not change significantly. However, the polynomial NTT conversion process could not be parallelized. Because NTT multiplication of the experimental results is the time including the NTT conversion, the simple point-wise multiplication speed is faster than the existing CPU implementation.

Table 12. Operation time (μs) and performance improvements by optimized major operation in NewHope.

	NewHope-512				
		GTX 1070		**RTX 2070**	
	CPU	**GPU**	**Improvement**	**GPU**	**Improvement**
NTT Multiplication	1618	734	120%	518	212%
Poly Addition	110	68	62%	65	69%
Poly Sampler	2837	941	201%	820	246%
	NewHope-1024				
		GTX 1070		**RTX 2070**	
	CPU	**GPU**	**Improvement**	**GPU**	**Improvement**
NTT Multiplication	3494	978	257%	748	367%
Poly Addition	219	120	82%	107	104%
Poly Sampler	6951	2163	221%	1915	263%

In the parallel sampling process, when implementing to perform different roles for threads in the warp without taking control divergence into account, serialization occurs, and CPU performance is better than GPU performance. Accordingly, we solved the problem by changing the threads in the warp to perform the same role in consideration of the warp of the thread. The sampling optimization result in the table is the result of measuring the version that solved the problem. When only kernel time was measured, excluding memory copy time, NewHope-512's polynomial multiplication kernel was confirmed that a throughput of 604 and 1435 Gbps was achieved in GTX 1070 and RTX 2070, respectively. In Figure 9, NewHope's total encryption time is presented by each platform. In RTX 2070, 3.33 and 4.04 times faster speed than CPU implementation was confirmed for the whole encryption process in NewHope-512 and NewHope-1024, respectively. In GTX 1070, 2.95 and 3.76 times faster speed was confirmed for the whole encryption process in NewHope-512 and NewHope-1024, respectively.

In this paper, these experiments present the result of optimizing only the internal operation by calling one cryptographic algorithm. Therefore, in a real environment, since numerous threads can call not only major operations in the cryptographic algorithm, but also the cryptographic algorithm itself in parallel, the actual efficiency is much higher.

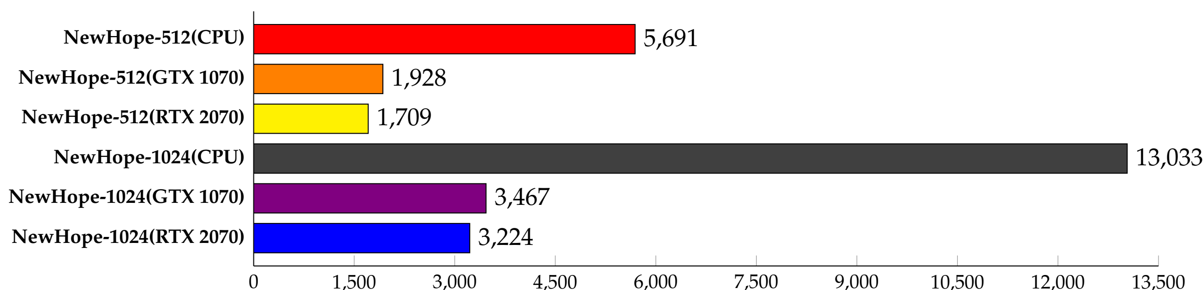

Figure 9. Comparison of operation time (μs) for NewHope encryption between CPU implementation and GPU implementation including memory copy time.

6. Conclusions

In this paper, we propose an optimization method for FrodoKEM and NewHope, which are NIST PQC standardized competition round 2 competition algorithms. The LWE-based algorithm, FrodoKEM, and the RLWE-based algorithm, NewHope, are lattice-based PQC candidates. By analyzing the main operation time for each algorithm, we propose several methods for parallel optimization of computations with a large time load in the CPU using the GPU. These are implemented to perform the main operation of algorithms in parallel using the GPU thread and solve the memory problems in the GPU that occur during the implementation process. As a result of implementation using RTX 2070, in the process of FrodoPKE, which is the encryption part of FrodoKEM, the performance results can be confirmed as up to 5.2, 5.75, and 6.47 times faster than CPU implementation in FrodoKEM-640, FrodoKEM-976, and FrodoKEM-1344, respectively. In the process of NewHope's encryption part, the performance results can be confirmed as up to 3.33 and 4.04 times faster than CPU implementation in NewHope-512 and NewHope-1024, respectively. Experimental measurements are taken on Nvidia GeForce GTX 1070 GPU and Nvidia GeForce RTX 2070 GPU.

The process can be actively used in a situation such as a server equipped with a GPU in an actual operating environment. Servers that manage IoT devices or servers that provide cloud computing services can reduce the burden of encryption operations. In addition, this optimization method can be used to encrypt data in real-time image processing technologies such as face recognition and eye tracking.

In this paper, only FrodoKEM and NewHope, which are candidates for NIST PQC standardization round 2, are proposed, but future studies will study optimization for more diverse algorithms, such as NTRU, Saber, and Crystals–Kyber. In addition, we plan to study the differences for each architecture while measuring various experimental results on more diverse GPU platforms.

Author Contributions: Investigation, S.A.; Software, S.A. and S.C.S.; Supervision, S.C.S.; Writing–original draft, S.A.; Writing–review and editing, S.C.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by National Research Foundation of Korea: 2019R1F1A1058494.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Federal Information Processing Standards Publications 197 (FIPSPUBS). *Announcing the ADVANCED ENCRYPTION STANDARD(AES)*; Technical Report; National Institute of Standards and Technology (NIST): Gaithersburg, MD, USA, 2001.
2. Menezes, A.J.; Oorschot, P.C.; Vanstone, S.A. *Handbook of Applied Cryptography*; CRC Press: Boca Raton, FL, USA, 1996.
3. Shor, P.W. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Sci. Statist. Comput.* **1997**, *26*, 1484–1509. [CrossRef]
4. Moody, D. *The Ship Has Sailed: The NIST Post-Quantum Crypto Competition*; AsiaCrypt: HongKong, 2017.
5. Li, P.; Zhou, S.; Ren, B.; Tang, S.; Li, T.; Xu, C.; Chen, J. Efficient implementation of lightweight block ciphers on volta and pascal architecture. *J. Inf. Secur. Appl.* **2019**, *47*, 235–245.
6. Pan, W.; Zheng, F.; Zhao, Y.; Zhu, W.T.; Jing, J. An Efficient Elliptic Curve Cryptography Signature Server with GPU Acceleration. *IEEE Trans. Inf. Forens. Secur.* **2016**, *12*, 111–122. [CrossRef]
7. Coelho, I.M.; Coelho, V.N.; Luz, E.J.d.S.; Ochi, L.S.; Guimaraes, F.G.; Rios, E. A GPU deep learning metaheuristic based model for time series forecasting. *J. Inf. Secur. Appl.* **2019**, *47*, 412–418. [CrossRef]
8. Bos, J.; Costello, C.; Ducas, L. Frodo: Take off the Ring! Practical, Quantum-Secure Key Exchange from LWE. In Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, 24–28 October 2016; pp. 1006–1018.

9. Alkim, E.; Ducas, L.; Pöppelmann, T.; Schwabe, P. Post-quantum Key Exchange—A New Hope. In Proceedings of the 25th USENIX Security Symposium (USENIX Security 16), Austin, TX, USA, 10–12 August 2016; pp. 327–343.
10. Gathen, J.v.z.; Gerhard, J. *Modern Computer Algebra*; Cambridge University Press: Cambridge, UK, 2003.
11. Octavian, D.; Iulian, P. Face Detection and Face Recognition in Android Mobile Applications. *Inform. Econ. J.* **2016**, *20*, 20–28.
12. Dospinescu, O.; Perca-Robu, A.E. The Analysis of E-Commerce Sites with Eye-Tracking Technologies. *Board Res. Artif. Intell. Neurosci.* **2017**, *8*, 85–100.
13. Regav, O. On lattices, learning with errors, random linear codes, and cryptography. In Proceedings of the STOC'05: Thirty-Seventh Annual ACM Symposium On Theory of Computing, Baltimore, MD, USA, 22–24 May 2005; pp. 84–93.
14. Peikert, C. Public-key cryptosystems from the worst-case shortest vector problem. In Proceedings of the STOC'09: Forty-First Annual ACM Symposium On Theory of Computing, Bethesda, MD, USA, 31 May–2 June 2009; pp. 333–342.
15. Lindner, R.; Peikert, C. *Better Key Sizes (and Attacks) for LWE-Based Encryption*; Springer: Berlin, Germany, 2011; pp. 319–339.
16. Badawi, A.; Veeravalli, B.; Aung, K.; Hamadicharef, B. Accelerating subset sum and lattice based public-key cryptosystems with multi-core CPUs and GPUs. *J. Parallel Distrib. Comput* **2019**, *119*, 179–190. [CrossRef]
17. Lee, E.; Kim, Y.; No, J.; Song, M.; Shin, D. Modification of Frodokem Using Gray and Error-Correcting Codes. *IEEE Access* **2019**, *7*, 179564–179574. [CrossRef]
18. Lyubashevsky, V.; Peikert, C.; Regev, O. *On Ideal Lattices and Learning with Errors Over Rings*; Eurocrypt/Springer: Berlin, Germany, 2010.
19. Peikert, C. Lattice Cryptography for the Internet. In Proceedings of the International Workshop on Post-Quantum Cryptography, Waterloo, ON, Canada, 1–3 October 2014; pp.197–219.
20. Tan, T.N.; Lee, H. Efficient-Scheduling Parallel Multiplier-Based Ring-LWE Cryptoprocessors. *Electronics* **2019**, *8*, 413. [CrossRef]
21. Duong-Ngoc, P.; Tan, T.N.; Lee, H. Efficient NewHope Cryptography Based Facial Security System on a GPU. *IEEE Access* **2020**, *8*, 108158–108168. [CrossRef]
22. Ajtai, M. Generating hard instances of lattice problems. In Proceedings of the STOC'96: Twenty-Eighth Annual ACM Symposium On Theory of Computing, Philadelphia, PA, USA, 22–24 May 1996; pp. 99–108.
23. Federal Information Processing Standards Publications 202 (FIPSPUBS). *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*; Technical Report; National Institute of Standards and Technology (NIST): Gaithersburg, MD, USA, 2015.
24. Montgomery, P. Modular Multiplication Without Trial Division. *Math. Comput.* **1985**, *44*, 519–521. [CrossRef]
25. NVIDIA. CUDA Toolkit-"Develop, Optimize and Deploy GPU-Accelerated Apps". Available online: https://developer.nvidia.com/cuda-toolkit (accessed on 13 September 2020).

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

© 2020 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/).