Automatic Parallelism Tuning for Module Learning with Errors Based Post-Quantum Key Exchanges on GPUs

Tatsuki Ono
Graduate School of Informatics
Kyoto University
Kyoto, Japan

Song Bian
Graduate School of Informatics
Kyoto University
Kyoto, Japan

Takashi Sato
Graduate School of Informatics
Kyoto University
Kyoto, Japan

***Abstract*—The module learning with errors (MLWE) problem is one of the most promising candidates for constructing quantum-resistant cryptosystems. In this work, we propose an open-source framework to automatically adjust the level of parallelism for MLWE-based key exchange protocols to maximize the protocol execution efficiency. We observed that the number of key exchanges handled by primitive functions in parallel, and the dimension of the grids in the GPUs have significant impacts on both the latencies and throughputs of MLWE key exchange protocols. By properly adjusting the related parameters, in the experiments, we show that performance of MLWE based key exchange protocols can be improved across GPU platforms.**

I. Introduction

The rapid advancement of quantum computer technologies is making it possible to solve mathematical problems that have been difficult or intractable for conventional computers [1]. In particular, most of the public-key cryptosystems currently in use are expected to be broken by quantum computers. Therefore, there is a need for post-quantum cryptography which is secure against attacks from both quantum and classical computers. For this reason, the National Institute of Standards and Technology (NIST) is taking the lead in standardizing post-quantum public-key cryptography, where the last-round candidate algorithms were published in July 2020 [1]. While the standardization draft is expected to be released in 2022–2024, the Kyber [2], [3] key encapsulation mechanism (KEM) based on the learning with errors [4], [5] (in particular, the module learning with errors (MLWE) [6]) problem has already attracted attentions from across the academic fields, as Kyber is currently one of the four finalists in standardization process.

Recently, the use of graphics processing units (GPUs) as accelerators for applications outside the domain of computer graphics, has become widespread, known as general-purpose computing on the GPU (GPGPU). GPUs have thousands of arithmetic cores and show higher performance than central processing units (CPUs) for highly data-parallel computing. However, the computational power per core on GPUs is much lower than that of CPUs, and the latency for sequential operations is high. Therefore, in order to maximize the computing power of GPUs, we need to implement highly parallel algorithms that are suitable for the GPU architecture. As for conventional cryptographic systems such as RSA, implementations for GPUs have been proposed so far, but they have not yet achieved a higher performance than those on CPUs [7]. In contrast, learning-with-errors-based cryptography requires computationally large and complex parallel operations such as number theoretic transforms (NTT) and matrix multiplication. Such operations may take a considerable amount of time to execute on conventional CPUs, but can clearly be accelerated by parallelized implementations [8]–[11]. However, in order to run the application efficiently on GPUs, parameters such as the size of thread blocks allocated to functions and data placement in memory need to be configured appropriately. Currently, to the best of our knowledge, there exists no frameworks for parameter tuning on GPUs for cryptographic applications.

In this paper, we propose an automatic tuning framework for our full-scratch GPU implementation of Kyber. Different from the most relevant existing work [8], we develop both the open-source GPU implementation of Kyber and the parameter optimization framework [1]. In the proposed framework, we first measure the performance of the GPU implementation by varying parameters such as the grid and block size. Then, we determine the optimal parameters by applying a merit function over the measured performance figures, in this case the latency and throughput of Kyber key encapsulations. Finally, the optimized parameters will be fed into the compilation engine to create the actual running instance of the program. In the experiments, we show that by using our framework, we can achieve 85.5% better performance at the maximum over the baseline implementation.

II. Preliminaries

A. Notations for Lattice Cryptography

In this work, we use n to depict the order of $\mathcal{R}_q$, the quotient ring of polynomials modulo some integer q, and k the rank of the module $M \in \mathcal{R}_q^k$ over $\mathcal{R}_q$. We denote polynomials by

This work was partially supported by JSPS KAKENHI Grant No. 20K19799, 20H04156, and 20K21793. This work was also supported by JST, PRESTO Grant Number JPMJPR20M7, Japan.

[1] Source code available at https://github.com/tono-satolab/atpqc-cuda

Algorithm 1 Keypair() $\rightarrow (pk, sk)$

$z \leftarrow \{0,1\}^{32}$
$(publicseed \| noiseseed) \leftarrow \mathsf{SHA3\text{-}512}(\{0,1\}^{32})$
$\hat{\mathbf{A}} \leftarrow \mathsf{GenMatrix}(publicseed)$
$\hat{\mathbf{s}} \leftarrow \mathsf{NTTVec}(\mathsf{GenNoiseVec}(noiseseed, 0))$
$\hat{\mathbf{e}} \leftarrow \mathsf{NTTVec}(\mathsf{GenNoiseVec}(noiseseed, n))$
$\hat{\mathbf{t}} \leftarrow \hat{\mathbf{A}} \circ \hat{\mathbf{s}} + \hat{\mathbf{e}}$
$pk \leftarrow (\mathsf{EncodeVec}(\hat{\mathbf{t}}) \| publicseed)$
$sk' \leftarrow (\mathsf{EncodeVec}(\hat{\mathbf{s}}))$
$sk \leftarrow (sk' \| pk \| \mathsf{SHA3\text{-}256}(pk) \| z)$

Algorithm 2 $\mathsf{Enc}(pk) \rightarrow (c, K)$

$m \leftarrow \mathsf{SHA3\text{-}256}(\{0,1\}^{32})$
$(\bar{K} \| r) \leftarrow \mathsf{SHA3\text{-}512}(m \| \mathsf{SHA3\text{-}256}(pk))$
$c \leftarrow \mathsf{CPAPKE.Enc}(pk, m, r)$
$K \leftarrow \mathsf{SHAKE\text{-}256}(\bar{K} \| \mathsf{SHA3\text{-}256}(c))$

regular font lower-case letters (e.g., $a \in \mathcal{R}_q$), vectors by bold lower-case letters (e.g., $\mathbf{a} \in \mathcal{R}_q^k$), and matrices by bold upper-case letters (e.g., $\mathbf{A} \in \mathcal{R}_q^{k \times k}$). In addition, we denote a, $\mathbf{a}$, and $\mathbf{A}$ in NTT-domain by $\hat{a}$, $\hat{\mathbf{a}}$, and $\hat{\mathbf{A}}$ respectively.

B. The Kyber Scheme

Kyber [2] is a chosen ciphertext attack secure KEM based on the hardness of the MLWE problem. First, let $\mathcal{R}_q$ be the ring $\mathbb{Z}_q[X]/(X^n+1)$, where X^n+1 is the $2^{n'}$-th cyclotomic polynomial, and $n = 2^{n'-1}$. Then, let $\mathbf{A}$ be a random matrix following a uniform distribution on $\mathcal{R}_q^{k \times k}$, $\mathbf{s}$ be a secret module with uniform distribution on $\mathcal{R}_q^k$, and $\mathbf{e}$ be an error module following a certain distribution χ on $\mathcal{R}_q^k$. The MLWE problem is finding $\mathbf{s}$ from $\mathbf{A}$ and $\mathbf{As} + \mathbf{e}$, where $\mathbf{s}$ and $\mathbf{e}$ are unknown, and is assumed to be hard [6].

Alg. 1–4 describe the three primitives that construct Kyber KEM. In Alg. 1, Keypair generates a pair of public key pk and secret key sk. The public key pk contains $publicseed$ to generate a random matrix $\hat{\mathbf{A}}$, and $\hat{\mathbf{t}} = \hat{\mathbf{A}} \circ \hat{\mathbf{s}} + \hat{\mathbf{e}}$. Here, $\mathbf{s}$ is a secret module and $\mathbf{e}$ is a error module. The secret key sk contains $\hat{\mathbf{s}}$, pk, the hash of pk, and a random byte sequence z. In Alg. 2, Enc generates a ciphertext c by using CPAPKE. Enc described in Alg. 3, and derives a symmetric key K. CPAPKE. Enc generates c from $\mathbf{u} = \mathbf{A} \cdot \mathbf{r} + \mathbf{e}'$ and $v = \mathbf{t} \cdot \mathbf{r} + e'' + m$, where r is a secret module, $\mathbf{e}'$ and e'' are error polynomials, and m is some message. In Alg. 4, Dec verifies the received ciphertext c using CPAPKE. Enc, and derives K from c by calculating $v - \mathbf{s}^\top \cdot \mathbf{u}$. In Alg. 3–4, GenMatrix, GenNoiseVec, and GenNoisePoly generate random matrices, modules, and polynomials, respectively; NTTVec, INTTVec, INTTPoly performs NTT [12] for efficient polynomial multiplication; CompressVec, EncodeVec, FromMsg, etc. perform the transformation between polynomials and byte sequences. SHA3-256, SHA3-512, and SHAKE-256 are hashing algorithms of the Secure Hash Algorithm-3 (SHA-3) function family [13]. Among the procedures, random generation, NTT, and the hash algorithm are particularly demanding in terms of computation, and become the performance bottlenecks in running Kyber.

Algorithm 3 CPAPKE. $\mathsf{Enc}(pk, m, r) \rightarrow (c_1 \| c_2)$

$(pk_{\hat{\mathbf{t}}} \| publicseed) \leftarrow pk$
$\hat{\mathbf{t}} \leftarrow \mathsf{DecodeVec}(pk_{\hat{\mathbf{t}}})$
$\hat{\mathbf{A}}^\top \leftarrow \mathsf{GenMatrix}(publicseed)$
$\hat{\mathbf{r}} \leftarrow \mathsf{NTTVec}(\mathsf{GenNoiseVec}(r, 0))$
$\mathbf{e}' \leftarrow \mathsf{GenNoiseVec}(r, n)$
$e'' \leftarrow \mathsf{GenNoisePoly}(r, 2n)$
$\mathbf{u} \leftarrow \mathsf{INTTVec}(\hat{\mathbf{A}}^\top \circ \hat{\mathbf{r}}) + \mathbf{e}'$
$v \leftarrow \mathsf{INTTPoly}(\hat{\mathbf{t}}^\top \circ \hat{\mathbf{r}}) + e'' + \mathsf{FromMsg}(m)$
$c_1 \leftarrow \mathsf{CompressVec}(\mathbf{u})$
$c_2 \leftarrow \mathsf{CompressPoly}(v)$

Algorithm 4 $\mathsf{Dec}(sk, (c, sk)) \rightarrow K$

$(sk' \| pk \| h \| z) \leftarrow sk$
$(c_1 \| c_2) \leftarrow c$
$\hat{\mathbf{u}} \leftarrow \mathsf{NTTVec}(\mathsf{DecompressVec}(c_1))$
$v \leftarrow \mathsf{DecompressPoly}(c_2)$
$\hat{\mathbf{s}} \leftarrow \mathsf{DecodeVec}(sk')$
$m' \leftarrow \mathsf{ToMsg}(v - \mathsf{INTTPoly}(\hat{\mathbf{s}}^\top \circ \hat{\mathbf{u}}))$
$(\bar{K}' \| r') \leftarrow \mathsf{SHA3\text{-}512}(m' \| h)$
$c' \leftarrow \mathsf{CPAPKE.Enc}(pk, m', r')$
$\bar{K}'' \leftarrow (c == c')?\bar{K}' : z$
$K \leftarrow \mathsf{SHAKE\text{-}256}(\bar{K}'' \| \mathsf{SHA3\text{-}256}(c))$

As shown in Fig. 1, the key exchange protocol proceeds as follows. Alice first executes Keypair and transfers pk to Bob. Bob then runs Enc and returns c to Alice. Alice finalizes the key exchange protocol by running Dec.

C. CUDA Programming

Compute Unified Device Architecture (CUDA) is a parallel computing framework designed for GPU environment by NVIDIA. Here, we introduce some important concepts in CUDA programming. A series of parallel processing on the GPU is treated as a function called kernel, which is launched by instructions from the host (e.g., CPU). The collection of threads generated by each kernel is referred to as a grid, and the threads form a group known as a block. The size of grid and blocks can be specified when the kernel is booted. The threads in a block that execute the same instructions simultaneously in units of 32 constitute a warp.

III. Tuning Framework

A. Overview

We have developed a framework shown in Fig. 2, which automatically tunes cryptographic applications on GPUs for the given cryptographic protocol, objective function, and execution environment. The proposed framework takes three inputs:

- Host code of primitive functions,
- Device code of kernel functions used in the primitive functions, and

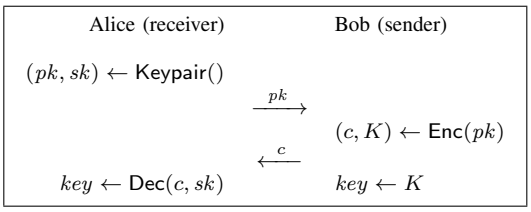

Fig. 1. Key exchange protocol based on Kyber [2].

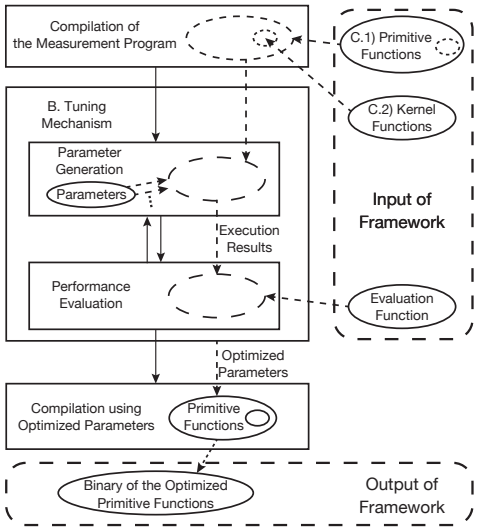

Fig. 2. Concept of the proposed framework.

- Evaluation function.

The details of primitive functions and kernel functions will be explained again in Section III-C. In this framework, first, the measurement program is compiled using the given primitive function and kernel functions. Then, the tuning mechanism described in Section III-B determines the optimal parameters that minimize the evaluation function. Finally, the framework compiles the primitive functions using the output parameters and output the optimized executable binary.

B. Tuning Mechanism

The parameter tuning mechanism is implemented as follows. First, the mechanism gives parameters to the primitive function, which can be custom designed, and measures the performance. Then, the execution results are passed to the evaluation function, and the performance score is recorded. The above procedure is repeated for all parameters in some predefined range, and the final output is the optimal parameter in terms of the performance scores.

In this work, we set the number of key exchanges handled by primitive functions in parallel, denoted as N, as the parameter to be optimized. We use l/c as the performance evaluation function, where l is the execution latency and c is the throughput of the primitive function. The (average) latency is calculated according to

$$l = \frac{1}{I_\mathrm{l}} \sum_{i}^{I_\mathrm{L}} t_i, \tag{1}$$

where I_l is the number of measurement trials and t_i is the time between the start and the end of one primitive function for the i-th trial. The throughput is calculated as

$$c = \frac{N I_\mathrm{t}}{T}, \tag{2}$$

where T is the time taken to complete the primitive function after I_t consecutive executions. We use CUDA streams and CUDA events to measure execution times.

C. Detailed Components

1) Primitive Functions: The three primitives, Keypair, Enc, Dec, are implemented as host functions as follows. First, at program launch, global memory allocation on the GPU and initialization of CUDA streams and CUDA events are completed. When each primitive function is invoked, the framework launches each kernel function with statically given parameters at compile time. Each kernel function is launched in parallel with appropriate orders using CUDA streams and CUDA events. Finally, when all the kernel functions have been executed, the event is recorded in the specified stream to notify the completion of the primitive functions.

2) Kernel Functions: We based our implementation upon the specification provided in [3]. The size of the grid for each kernel function is set to N, where we assume that each block processes a single key exchange protocol.

Symmetric Primitives We implemented SHA-3 function family using warp shuffle instructions based on [14]. Therefore, each block gets assigned to one warp.

Generating Random Matrices GenMatrix assigns $k \times k$ warps to a block, and each warp extends the random 32-bit seed to the width that is long enough to generate a matrix using SHAKE-128. Then, each warp converts the output of SHAKE-128 into matrix $\mathbf{A}$ by rejection sampling [15].

Generating Noise GenNoiseVec assigns k warps to a block, while GenNoisePoly assigns one warp to one block. Each warp extends the random 32-bit width seed by SHAKE-256, and converts the output numbers into a polynomial representation.

Number Theoretic Transform We implemented NTTVec, INTTVec, and INTTPoly using the Cooley-Tukey algorithm [16] for forward NTT and the Gentleman-Sande algorithm [17] for inverse NTT [18]. On the GPU, $n/2$ threads are assigned to apply the transformations to a single polynomial.

Addition, Subtraction, and Multiplication We implemented functions for the following six operations: $\hat{\mathbf{A}} \circ \hat{\mathbf{b}} + \hat{\mathbf{c}}$, $\hat{\mathbf{A}} \circ \hat{\mathbf{b}}$, $\hat{\mathbf{a}} \circ \hat{\mathbf{b}}$, $\mathbf{a} + \mathbf{b}$, $a + b + c$, and $a - b$. The size of the blocks are $kn/2$ threads for the former three, kn threads for $\hat{\mathbf{a}} \circ \hat{\mathbf{b}}$, and n threads for $a + b = c$ as well as $a - b$.

TABLE I
TEST ENVIRONMENT AND IMPROVEMENT PERFORMANCES THROUGH THE PROPOSED FRAMEWORK

			NVIDIA K80	NVIDIA GTX TITAN X	NVIDIA GTX1080Ti	NVIDIA V100	NVIDIA RTX2080Ti
Core Count			2496	3072	3584	5120	4352
Base Clk. Freq.		(MHz)	560	1000	1480	1245	1350
Mem. BW		(GB/s)	480	336.5	484	900	616
Alice	l	(ms)	1.72 (-38.1%)	0.542 (-54.6%)	0.777 (-79.0%)	3.99 ($+19.2\%$)	3.41 ($+22.8\%$)
	c	(KEX/ms)	26.7 (-10.2%)	83.1 (-23.6%)	136 ($+44.6\%$)	30.4 ($+8.65\%$)	64.8 ($+25.9\%$)
	l/c	(10^3ms^2/KEX)	645 (-31.1%)	65.2 (-40.6%)	57.3 (-85.5%)	1310 ($+9.73\%$)	527 (-2.46%)
Bob	l	(ms)	1.39 (-50.7%)	0.465 (-35.4%)	0.503 ($+1.88\%$)	6.50 ($+58.7\%$)	3.66 ($+57.1\%$)
	c	(KEX/ms)	48.2 (-40.8%)	149 (-26.7%)	324 ($+15.4\%$)	81.3 ($+83.9\%$)	110 ($+97.3\%$)
	l/c	(10^3ms^2/KEX)	290 (-16.7%)	31.3 (-11.9%)	15.5 (-11.7%)	800 (-13.7%)	332 (-20.4%)

TABLE II
OPTIMIZED N FOR EACH GPU ARCHITECTURE

GPU	Keypair	Enc	Dec
K80	24	48	88
GTX TITAN X	48	64	48
1080 Ti	72	136	144
V100	80	296	216
2080 Ti	136	376	256

Other Routines Encoding, decoding, verification and constant-time copying are implemented by unrolling the loops in the implementation of [3] and assigning a thread to each of the unrolled loop iterations.

IV. EXPERIMENTS

Here, we first describe the test environment and show the optimization results of the proposed framework. We choose the Kyber1024 [8] parameter suite specified for benchmarking. Kyber1024 corresponds to the NIST Security level 5, which is equivalent to a block cipher with a symmetric 256-bit key [3], [19]. For the complete experiment setup, please refer to our open-source implementation. For baseline comparison, we set the parameter to be optimized, N to be a fixed integer of 128, according to [9]. We assert that since existing works offered no automatic parameter tuning mechanisms, a fixed-parameter approach will always be equally or less efficient than our optimized parameter set.

Using the Kyber1024-based key exchange protocol, we measured l, c, and l/c on both Alice and Bob sides. We benchmarked the framework in several GPU environments, summarized in Table I. Table I details each GPU environment, the performance after optimization, and the performance change before and after our optimization. Here, $\mathrm{KEX/ms}$ is the number of keys that can be exchanged in one millisecond. The parameters after optimization under each GPU environment are shown in Table II. From Table I, we see that the optimization resulted in the improvements of l, c, and l/c on the Alice's side by $2.09\mathrm{ms}$, $68.1\,\mathrm{KEX/ms}$, and $52.2\times10^{-3}\,\mathrm{ms}^2/\mathrm{KEX}$, respectively, on average. Similarly, on Bob's side, $2.51\mathrm{ms}$, $142\,\mathrm{KEX/ms}$, and $29.4\times10^{-3}\,\mathrm{ms}^2/\mathrm{KEX}$ improvements for the same metrics are observed. The biggest l/c improvement was seen on 1080Ti, where l/c can be reduced by 85.5% for Alice and 11.7% for Bob. The mean improvements in l/c for Alice and Bob are 30.0% and 14.9%, respectively.

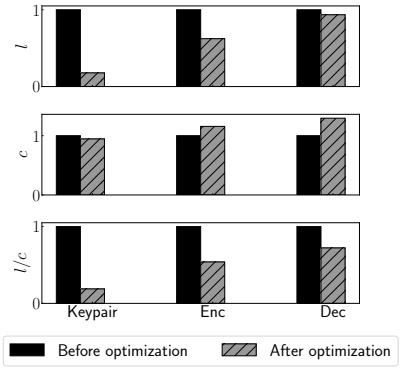

Fig. 3. Comparison of normalized performance scores for primitive functions on 1080Ti.

Fig. 3 shows a detailed comparison of normalized l, c, and l/c for each primitive function before and after the optimization process on 1080Ti, where l/c improved the most. Particularly in Keypair, we see that although the throughput is slightly reduced as a result of reducing N, the latency is reduced significantly. As a result, the performance metric l/c is greatly improved.

V. CONCLUSIONS

In this paper, we propose an automatic tuning framework for post-quantum key exchange scheme implementations on GPUs. We applied our framework to a full-scratch GPU implementation of Kyber, a MLWE-based post-quantum KEM. We explored how parameters, specifically the number of parallel key exchanges handled by the primitive functions at one time, impacts the latency l and throughput c of the KEM. Automatic tuning results showed that, depending on the GPU architecture, the latency-throughput product can be improved by up to 85.5% and 20.4% on Alice's and Bob's sides, respectively, compared to unoptimized versions.

ACKNOWLEDGMENT

The authors acknowledge support from VLSI Design and Education Center (VDEC), the University of Tokyo in collaboration with Synopsys, Inc.

REFERENCES

[1] NIST, "Post-quantum cryptography | CSRC," accessed: Nov. 15, 2020. [Online]. Available: https://csrc.nist.gov/Projects/Post-Quantum-Cryptography

[2] J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé, "CRYSTALS-Kyber: a CCA-secure module-lattice-based KEM," in *Proc. European Symp. Security and Privacy*. IEEE, Apr. 2018, pp. 353–367.

[3] R. Avanzi, J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé, *CRYSTALS-Kyber —Algorithm specifications and Supporting Documentation*, Apr. 2019. [Online]. Available: https://pq-crystals.org/kyber/data/kyber-specification-round2.pdf

[4] O. Regev, "On lattices, learning with errors, random linear codes, and cryptography," *Journal of the ACM*, vol. 56, no. 6, pp. 34:01–34:40, Sep. 2009.

[5] C. Peikert, "Public-key cryptosystems from the worst-case shortest vector problem," in *Proc. Forty-First Annu. ACM Symp. Theory of Computing*. New York, NY, USA: ACM, May 2009, pp. 333–342.

[6] A. Langlois and D. Stehlé, "Worst-case to average-case reductions for module lattices," *Designs, Codes and Cryptography*, vol. 75, no. 3, pp. 565–599, Jun. 2015.

[7] E. Ochoa-Jiménez, L. Rivera-Zamarripa, N. Cruz-Cortés, and F. Rodríguez-Henríquez, "Implementation of RSA signatures on GPU and CPU architectures," *IEEE Access*, vol. 8, pp. 9928–9941, Jan. 2020.

[8] N. Gupta, A. Jati, A. K. Chauhan, and A. Chattopadhyay, "PQC acceleration using GPUs: FrodoKEM, NewHope, and Kyber," *IEEE Trans. Parallel and Distributed Systems*, vol. 32, no. 3, pp. 575–586, Mar. 2021.

[9] X.-F. Wong, B.-M. Goi, W.-K. Lee, and R. C.-W. Phan, "Performance evaluation of RSA and NTRU over GPU with Maxwell and Pascal architecture," *Software Networking*, vol. 2017, no. 1, pp. 201–220, Jan. 2018.

[10] S. An and S. C. Seo, "Efficient parallel implementations of LWE-based post-quantum cryptosystems on graphics processing units," *Mathematics*, vol. 8, no. 10, p. 1781, Oct. 2020.

[11] K. Matsuoka, R. Banno, N. Matsumoto, T. Sato, and S. Bian, "Virtual secure platform: A five-stage pipeline processor over TFHE," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[12] J. M. Pollard, "The fast Fourier transform in a finite field," *Mathematics of Computation*, vol. 25, no. 114, pp. 365–374, 1971.

[13] NIST, *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*, Aug. 2015. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf

[14] W.-K. Lee, X.-F. Wong, B.-M. Goi, and R. C.-W. Phan, "CUDA-SSL: SSL/TLS accelerated by GPU," in *Proc. Int. Carnahan Conf. Security Technology*, Oct. 2017, pp. 1–6.

[15] S. Gueron and F. Schlieker, "Speeding up R-LWE post-quantum key exchange," in *Proc. Nordic Conf. on Secure IT Systems*. Springer, Nov. 2016, pp. 187–198.

[16] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Mathematics of Computation*, vol. 19, no. 90, pp. 297–301, Apr. 1965.

[17] W. M. Gentleman and G. Sande, "Fast Fourier transforms: for fun and profit," in *Proc. Fall Joint Computer Conf.* ACM, Nov. 1966, pp. 563–578.

[18] T. Pöppelmann, T. Oder, and T. Güneysu, "High-performance ideal lattice-based cryptography on 8-bit ATxmega microcontrollers," in *Proc. Int. Conf. Cryptology and Information Security in Latin America*. Springer, Aug. 2015, pp. 346–365.

[19] NIST, "Submission requirements and evaluation criteria for the post-quantum cryptography standardization process," accessed: Nov. 15, 2020. [Online]. Available: https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf