Parallel Implementation of SPHINCS+ With GPUs

DongCheon Kim, *Student Member, IEEE*, HoJin Choi, *Student Member, IEEE*, and Seog Chung Seo, *Member, IEEE*

***Abstract*— SPHINCS+ was selected as one of NIST Post-Quantum Cryptography Digital Signature Algorithms (PQC-DSA). However, SPHINCS+ processes are slower compared to other PQC-DSA. When integrating it into protocols (*e.g.*, TLS and IPSec), optimization research from the server perspective becomes crucial. Therefore, we present highly parallel and optimized implementations of SPHINCS+ on various NVIDIA GPU architectures (Pascal, Turing, and Ampere). We discovered parts within the internal processes of SPHINCS+ that could be parallelized and optimized them (*e.g.*, leaf node generation and node merging process in MSS, subtree constructions in FORS, signature generation in WOTS+ and hypertree layer construction), leveraging the characteristics of GPU architecture (*e.g.*, warp-based execution and efficient memory access). As far as we know, this is the first SPHINCS+ implementations on GPUs. Our implementations achieve 44,391(resp. 24,997 and 11,401) signature generations, 725,118(resp. 354,309 and 100,168) key generations, and 285,680(resp. 155,800 and 106,280) verifications per second at security level 1(resp. 3 and 5) on RTX3090. Furthermore, on GTX1070, our SPHINCS+ shows an enhanced throughput of ×2.10 for signature generation, ×1.03 for key generation, and ×9.86 for verification at security level 1, surpassing the study conducted by Sun *et al*. (IEEE TPDS 2020) on the GTX1080 having 640 more cores than GTX1070.**

***Index Terms*— Post-quantum cryptography, hash-based digital signature algorithm, SPHINCS+, GPU architecture, CUDA, parallel processing.**

I. Introduction

RECENTLY, quantum computers have been rapidly developed by IBM and Google [1], [2]. It is widely known that currently used public key algorithms such as RSA and ECC can be broken by Shor's algorithm on a sufficiently large quantum computer [3]. In order to prepare secure IT environment in quantum computing era, NIST initiated a standardization competition for the development of Post-Quantum Cryptography (PQC) with respect to secure Key Encapsulation Mechanism (KEM) and Digital Signature Algorithm (DSA) since 2016 [4]. As the result of the round 3, in July 2022, NIST announced that they selected one KEM algorithm (Crystals-Kyber) and three DSA (Crystals-Dilithium, Falcon, and SPHINCS+) [5], [6], [7], [8]. Among them, Crystals-Kyber, Crystals-Dilithium, and Falcon are a kind of Lattice-Based Cryptosystems (LBCs) and only SPHINCS+ is a type of hash-based algorithm. Compared with LBCs, Hash-based digital signature methods include One Time Signature (OTS), Few-Time Signature (FTS) and Merkle hash tree Signature Scheme (MSS). OTS algorithms include Lamport signature, Winternitz One-Time Signature (WOTS), and WOTS-plus (WOTS+) [9], [10], [11]. In OTS method, if a private key is used to sign two different messages, the security degrades. The FTS method was designed to compensate for the key generation performance overhead and key recycling problems of the OTS method. There are several FTS methods: Hash to Obtain Random of Subset (HORS), HORS-plus (HORS+), and Forest Of Random Subset (FORS) methods [12].

SPHINCS+ is a stateless hash-based signature algorithm. SPHINCS+ uses existing WOTS+-based MSS, WOTS+-based XMSS and FORS hash-based digital signature methods as internal operation schemes [13]. The advantage of SPHINCS+ is that its security can be formally verified based on the security of underlying hash functions. It's another advantage is very short public key and private key size (For example, the sizes of public key and private key are 48 bytes and 96 bytes, respectively. Refer to Table I). However, SPHINCS+ generates long-sized signatures and its signature generation is quite slow. Based on NIST IR 8413-upd1 paper, the signature generation and verification times of SPHINCS+ are about ×150 and ×50 slower than those of Dilithium on x86-64 processor [14]. In uses of SPHINCS+ in cryptographic protocols such as Transport Layer Security (TLS), IPSec, Secure Shell (SSH) protocol, and so on, it is necessary to study optimizing the performance of SPHINCS+ signature algorithm.

A Graphic Processing Units (GPUs) device is an auxiliary device designed for computer graphics processing tasks, such as 3D design processing. A GPU device has a structure effective for task parallelization and data operation parallelization through many cores. As the hardware specifications of GPU devices have evolved, the use of GPU has been expanded to general applications including accelerating cryptographic operations, which is called General-Purpose computing on GPU (GPGPU) technology.NVIDIA developed the Computed Unified Device Architecture (CUDA), which allows designing computational processing methods for GPU devices using C/C++ and Python [15], [16], [17], [18]. Recently, there has been active research in the field of cryptography utilizing

Manuscript received 20 October 2023; revised 10 January 2024 and 10 February 2024; accepted 22 February 2024. Date of publication 6 March 2024; date of current version 30 May 2024. This work was partly supported by the Institute of Information and communications Technology Planning and Evaluation (IITP) Grant by the Korean Government (MSIT) (Research on Foundational Technologies for 6G Autonomous Security-by-Design to Guarantee Constant Quality of Security, 50%) under Grant 2021-0-00796 and supported by National Research Foundation of Korea (NRF) grant funded by (MSIT) (No. 2022R1C1C1013368, 50%). This article was recommended by Associate Editor W. Liu. *(Corresponding author: Seog Chung Seo.)*

The authors are with the Department of Financial Information Security, Kookmin University, Seoul 02707, South Korea (e-mail: kindongsy@kookmin.ac.kr; ondoli0312@kookmin.ac.kr; scseo@kookmin.ac.kr).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TCSI.2024.3370802.

Digital Object Identifier 10.1109/TCSI.2024.3370802

TABLE I
SPHINCS+ PARAMETER SETS (n: OUTPUT LENGTH OF HASH FUNCTION, h: HEIGHT OF HYPERTREE, d: NUMBER OF LAYERS IN HYPERTREE, $\log t$: HEIGHT OF SUBTREE IN FORS, k: NUMBER OF SUBTREE IN FORS, w: MESSAGE DIVISION UNIT FOR WOTS+)

Algorithm	n (byte)	h	d	$\log t$	k	w	Public Key Len (byte)	Private Key Len (byte)	Signature Len (byte)
SPHINCS+ - 128s	16	63	7	12	14	16	32	64	7,856
SPHINCS+ - 128f	16	66	22	6	33	16	32	64	17,088
SPHINCS+ - 192s	24	63	7	14	17	16	48	96	16,224
SPHINCS+ - 192f	24	66	22	8	33	16	48	96	35,664
SPHINCS+ - 256s	32	64	8	14	22	16	64	128	29,792
SPHINCS+ - 256f	32	68	17	9	35	16	64	128	49,856

GPGPU technology, such as performance improvements for blockchain systems and enhancing the speed of cryptographic operations required for TLS usage in cloud services [19], [20]. Additionally, GPU is also gaining attention as standalone signature server whose capability is delivered via high-speed network to tenants typically located in a well-protected enterprise private cloud [21].

The main motivation of this work is to enhance the throughput of signing and verification of SPHINCS+ with GPUs. There are a lot of applications requiring high-throughput of signing and verification. For example, domain name servers in DNSSEC (Domain Name System Security Extensions) need to sign numerous DNS record messages which are responses to the requests messages from clients [22]. Global online payment platform, such as Alipay, requires a huge number of simultaneous signing and verification for secure transaction [21]. To alleviate computational overhead of signing and verification from service providers, a model of *Signature as a Service* (SIGaaS) was proposed for ECDSA-based SIGaaS [21] and SPHINCS-based SIGaaS [23]. Since the computational overhead of signing and verification of SPHINCS+ is larger than that of ECDSA,[1] it is required to accelerate the throughput of signing and verification of SPHINCS+.

Until now, there have been several works optimizing public key algorithms and PQC by using NVIDIA GPUs [24], [25], [26], [27]. In addition, there is a study on the optimization of SPHINCS on NVIDIA GPU devices from Sun et al. [23]. Even through they achieved 5,152 signings per second on the NVIDIA GeForce GTX 1080, their methods cannot directly be used for optimizing SPHINCS+ because there are several differences between SPHINCS+ and SPHINCS. Furthermore, they ignored to parallelize the hypertree operations in their work, which is the main contribution in our work.

Our contribution can be summarized as follows:

1) Proposing Parallel Methods for FORS, WOTS+, MSS, and Hypertree Computation in SPHINCS+: Through analyzing the process of signing in SPHINCS+, we found that underlying signature schemes (FORS, WOTS+, and MSS) can be processed in parallel in GPU environment. We propose a suitable GPU configuration for parallelizing the computations of FORS, WOTS+, and MSS. Furthermore, we found that MSS constructions (including WOTS+ key generation) and WOTS+ signing in hypertree computation can be processed in parallel. Namely, MSS constructions in all layers of hypertree can be executed in parallel with GPU cores and then WOTS+ signings in all layers can also be executed in parallel. Thus, we change the execution order of hypertree computation such that all MSS constructions in hypertree are computed first and then, all WOTS+ signings are computed, and execute them in parallel, which results in maximizing the throughput.

2) Optimizing SPHINCS+ Performance by Taking Full Advantage of CUDA and Benchmarking on Various GPUs: We optimized the proposed parallel methods by taking advantage of GPU and CUDA characteristics. For example, we proposed an efficient shared memory access/store pattern for efficient merging steps in MSS. Through the proposed parallel methods and their efficient CUDA implementation, we achieved huge performance improvement. First, our SPHINCS+ signature generation implementation achieved performance improvements, denoted as $\times 1,305.61$ (resp. $\times 310.42$), $\times 1,136.22$ (resp. $\times 287.32$), and $\times 1,036.45$ (resp. $\times 259.11$) compared to the Reference C implementation (resp. AVX-2 version) across various security levels on RTX 3090. Subsequently, our SPHINCS+ key generation implementation demonstrated performance enhancements, achieving $\times 1,052.42$ (resp. $\times 232.03$), $\times 581.78$ (resp. $\times 141.72$), and $\times 426.24$ (resp. $\times 138.35$) on the same comparison basis. Finally, our SPHINCS+ verification implementation also achieved performance improvements denoted as $\times 462.26$ (resp. $\times 138.74$), $\times 362.33$ (resp. $\times 106.71$), and $\times 250.66$ (resp. $\times 75.75$). Furthermore, when comparing the SPHINCS implementation from Sun *et al.*'s [23] in the GTX 1080 with our implementation in the GTX 1070 having 640 fewer cores than GTX 1080, we achieved $\times 2.10$ the performance in signature generation, $\times 1.03$ in key generation, and $\times 9.86$ in verification. The comprehensive performance analysis on GPUs can be found in Section VI.

Organization The remainder of this article is organized as follows. Section II contains some backgrounds. Section III contains profiling results for SPHINCS+ for finding bottleneck of SPHINCS+. Section IV describes the proposed parallelization approaches to signature generation and key generation. Section V explains the proposed parallelization approaches to signature verification. Section VI presents the performance analysis results. Section VII concludes this paper.

II. BACKGROUNDS

A. Description of SPHINCS+

SPHINCS+ consists of existing hash-based digital signatures. Therefore, to understand SPHINCS+, one must first

[1]Signing and verification in SPHINCS+ are about 348 and 7 times slower than those in ECDSA on x84-64-bit system based on https://openquantumsafe.org/benchmarking/visualization/openssl_speed.html

have an understanding of the MSS (or XMSS), FORS, WOTS+, and Hyper-Tree algorithms [28].

1) MSS & XMSS: MSS is a structure that combines the operation of a conventional tree structure and hash function [29]. In the case of a binary tree MSS, with a total height of h, it can manage 2^h OTS key pairs by storing the digests of OTS keys in the leaves of a Merkle tree. The MSS method uses a different hash-based signature algorithm in the leaf node generation process. The signature includes the index of the leaf, the OTS public key, the digest of the OTS public key (leaf), and the authentication path of that leaf. The structure of XMSS is similar to that of MSS, but it includes a process to XOR a random mask value during node merging [30].

2) FORS: FORS is an improved version of HORS designed by the SPHINCS+ team. Defined in terms of integers k and $t = 2^a$, it can be used to sign a string of ka bits. The message digest is split into k bitstrings of length a each, and k of the divided message blocks are allocated to each FORS-subtree. The message block is used as the selection index for the created leaf node. The FORS private key consists of kt random n-bit values grouped into k sets of t values each.

3) WOTS+: The Lamport signature scheme generates private and public key pairs for each bit of a message, whereas WOTS+ divides a hashed message into segments [11]. WOTS+ first transforms the message m into a new form using a base w representation and then breaks it down into blocks of length $\log w$. For each block, it applies a chosen function up to a maximum of $w-1$ times, and the output of this becomes the signature for that block. Increasing the value of w reduces the length of the signature but increases the time required for signature calculation.

4) Hypertree: SPHINCS+ utilizes a hypertree structure to efficiently handle the generation of leaf nodes by subdividing the entire tree height. The total height of the hypertree is specified as h, and it is divided into a total of d layers. The hypertree is designed to minimize the leaf node generation process in the MSS structure. The total tree height of SPHINCS+ is composed of 66, 66, and 68 for each of security levels 1, 3, and 5, respectively.

The overall process of signing in SPHINCS+ is shown in Fig. 1. SPHINCS+ uses the FORS method for signing the original input message at the lowest level. Then, it utilizes a hypertree structure to alleviate the overhead of generating leaf nodes from 2^h to $d \times 2^{h/d}$ in the MSS like SPHINCS. With the hypertree mechanism, single MSS of height h is divided into $d-1$ layers containing a subtree of height $\lceil h/d \rceil$. Namely, each layer of the hypertree contains WOTS+ signing and MSS construction. Note that MSS construction has two steps: generation of leaf nodes (WOTS+ key pair generation) and authentication path corresponding to select a WOTS+ key pair. Through the signing process in Fig. 1, the final signature consists of randomness information, FORS signature, d WOTS+ signatures, and the public key which is the root node of the MSS in the layer $d-1$. The details of FORS, WOTS+, MSS, and hypertree can be found in the specification document [13].

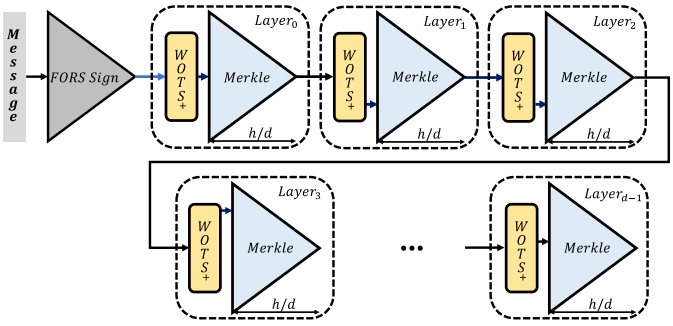

Fig. 1. SPHINCS+ structure.

B. SPHINCS+ Modes and Parameters

SPHINCS+ has two modes: small mode (SPHINCS+s), which aims to minimize the signature length, and fast mode (SPHINCS+f), which aims to maximize operation speed. In addition, SPHINCS+ is subdivided into robust model and simple model, depending on whether or not random values ($mask$) are used in MSS computations. Simple model uses MSS tree structure, and robust model uses XMSS tree structure. Table I shows the parameter sets of SPHINCS+. n is the output length of the available hash function. SHA-256, SHAKE and Haraka can be used for the internal hash function in SPHINCS+. All output values of the hash function used in SPHINCS+ are 32 bytes. Note that SPHINCS+ of security level 1 (resp. 3 and 5) uses 16-bytes (resp. 24-bytes and 32-bytes) out of 32-bytes hash function output. d is the number of layers in the hypertree. Therefore, the height of each layer satisfies $\lceil h/d \rceil$. Each of $\log t$ and k is the height of each subtree and the number of subtrees in FORS method, respectively. w is a message division unit used in WOTS+ method. The signature length and the number of operations depends on security parameters. In the SPHINCS+ reference code submitted to NIST competition, SPHINCS+f mode on security level 5 is faster than SPHINCS+s mode as much as $\times 16$, $\times 9$, and $\times 2$ for key generation, signing, and signature verification, respectively [8]. However, the signature size of SPHINCS+f is much longer than that of SPHINCS+s mode. Regarding performance of signing and verification of SPHINCS+, signing and verification are about $\times 150$ and $\times 50$ slower than those of Dilithium on x86/64-bit CPU according to [14]. Thus, the performance of SPHINCS+ (especially, signing) needs to be improved for smoothly applying it to various security protocols such as TLS, DNSSEC, and so on.

C. Differences Between SPHINCS+ and NIST FIPS 205 Draft

NIST announced the three FIPS drafts for Post-Quantum Cryptography [31] on August 2023, based on the selected algorithms from the competition. Among them, FIPS 204 and 205 serve as Digital Signature Standard (DSS): FIPS 204 (Module-Lattice-Based DSS) and FIPS 205 (Stateless Hash-Based DSS) for Dilithium and SPHINCS+, respectively. Even though the algorithm drafted in FIPS 205 is based on SPHINCS+ version 3.1, there exist only a few minor differences compared with SPHINCS+ version 3.0 which is our target algorithm as follows.

- Two new address types were defined, $WOTS_{PRF}$ and $FORS_{PRF}$, which are used for WOTS+ and FORS secret key value generation.
- PK.seed was added as an input to PRF in order to mitigate multi-key attacks.
- For the security level 3 and 5 parameter sets that use SHA-2, SHA-256 was replaced with SHA-512 in H_{msg}, PRF_{msg}, H and T_l based on weakness that were discovered when using SHA-256 to obtain security level 5.
- R and PK.seed were added as inputs to $MGF1$ when computing H_{msg} for the SHA-2 parameter sets in order to mitigate against multi-target long-message second preimage attacks.

In addition, modifications were made to the specification based on the reference implementation of SPHINCS+ version 3. For standardization, among the 36 parameter sets, only the 'simple' instances instantiated with SHA-2 or SHAKE cryptographic functions are approved. The primary difference between the SPHINCS+ version 3 targeted in our research and the approved version lies in the input/output parameters of the algorithm and the type of hash function (for security level 3 and 5). Nevertheless, since our proposed optimization strategies are not dependent on specific hash functions, they can be applied universally.

D. GPU and CUDA

Recently, GPUs have been widely used for general-purpose tasks including artificial intelligence, deep-learning, cryptographic accelerators, and so on. As a development platform, we select NVIDIA GPUs and CUDA framework [32] for its popularity. CUDA-enabled GPUs include independent multiple Streaming Multiprocessors (SM). An SM is the basic unit that can execute GPGPU programs. SM consists of several Stream Processors (SPs or CUDA cores), Special Function Unit, L1 cache, and shared memory. Application developed with CUDA executes in a way of Single Instruction Multiple-Thread (SIMT) operation model. In other words, threads in a GPU application execute the same instruction as a unit of warp. However, if there are codes related with conditional branches, warp divergence occurs, which results in serial execution. A GPU application is called a grid and a grid has several logical GPU blocks, and also each block has several threads. There are built-in variables for identifying a block in a grid and a thread in a block. The unit of parallel processing in the GPU CUDA is a warp and threads in a block are scheduled by a warp unit (typically, 32). GPU CUDA memory is generally divided into constant memory, shared memory, and global memory. Each memory area has a different memory access speed, memory size, and memory bandwidth. The constant memory cannot modify the stored data value and has a small memory capacity, but the memory access speed is fast. Global memory has a large memory capacity, but memory access speed is slower than other memory areas. Therefore, when designing a parallel computing operation method using GPU CUDA, an efficient memory utilization method is required. The details of NVIDIA GPU and CUDA programming method can be found in [32].

E. Existing Implementations of NIST PQC DSA on GPUs

1) SPHINCS and SPHINCS+: There is a study on the optimization of SPHINCS, a hash-based DSA, on GPU devices [23]. In Sun et al.'s [23] paper, parallelization of SPHINCS MSS merging process, parallelization of HORST digital signature process, and parallelization of WOTS+ digital signature process were proposed, and optimization methods applicable to GPU devices were provided. Their software performs 5,152 (resp. 6,651 and 27,052) SPHINCS-256 (ChaCha) signatures generation per second in the GTX 1080 environment (resp. TITAN Xp and 4 TITAN Xp). There is also SPHINCS optimization study results in the hardware device environment [33]. In [33], a method for accelerating the ChaCha and Blake functions in an FPGA device and a pipeline method for the HORST operation structure of SPHINCS were proposed. The optimization result in [33] resulted in a performance measurement of 1.53 milliseconds per signature generation and 65 microseconds in signature verification using about 19,000 LUT, 38,000 FF, and 36 BRAM on the Kintex-7 Xilinx FPGA board. There are also optimization research results related to SPHINCS+ in hardware environment [34], [35]. Quentin Berthet et al. [34] proposed a method for designing digital signature WOTS+ and FORS calculations used internally for the signature and verification of SPHINCS+ in FPGA hardware devices. The method also focuses on minimizing hardware resource utilization for digital signature built-in functions. Quentin Berthet et al.'s [34] implementation methods consumed 64.34 ms per signature and 2.51 ms per verification based on the SPHINCS+SHA256-128f-simple mode on the Xilinx XZU3EG FPGA board, and used 5,917 LUTs, 210 LUTRAMs, 4,933 FFs, and 0.5 BRAM. Dorian Amiet et al. [35] proposed an optimization study for SPHINCS+, which includes speeding up SHA-3 and SHAKE functions on FPGA devices. Their implementation achieved 1.01 ms per signature using SPHINCS+SHAKE-128f-simple mode on 7-series Xilinx FPGAs, with resource usage of 47,991 LUTs, 72,505 FFs, and 11.5 BRAM [35]. In most previous research, the focus was primarily on accelerating the hash function itself, and there were no studies aimed at parallelizing the SPHINCS+ structure. We fill this gab by processing several optimization approaches in our work.

2) Dilithium: Dilithium, announced as the PQC DSS alongside SPHINCS+, has undergone several studies in GPU environments, in contrast to SPHINCS+. The algorithms that primarily contribute to computational burdens during Dilithium operations are NTT (Number-Theoretic Transform) and rejection sampling. The papers up to now have focused on optimizing these algorithms using GPU threads. In [36], detailed algorithms used in Dilithium operations were optimized from the perspective of latency by varying the batch mode (or batch-size) using GPU threads. For the Dilithium scheme with a batch size of $15,360$, the authors achieved reduction in computation time compared to the reference C code on an Intel Xeon Gold 6133 CPU. Specifically, on P2000, V100, and T4 GPUs, they achieved time reductions of $\times 3.51$, $\times 11.18$, and $\times 4.92$ respectively. On the other hand, in [37], optimization studies from the perspective of throughput were conducted targeting NTT and rejection sampling in

a GPU environment. In the GPU environment of NVIDIA Jetson AGX Xavier, using the reference C code, the authors achieved throughput improvements compared to Jetson AGX Xavier's CPU (8-core ARM v8.2) at security level 1 (resp. 3 and 5). Signature generation achieved throughput improvements of ×14.31 (resp. ×11.73 and ×20.41), key generation ×16.42 (resp. ×17.33 and ×17.22), and verification ×15.66 (resp. ×16.92 and ×17.20). In particular, the most recent GPU optimization paper for Dilithium, namely Shiyu Shen et al.'s [38] work, conducted optimization research in GPU environments not only for the aforementioned algorithms but also for hash algorithms. In addition, unlike previous studies, optimization based on the analysis of resource utilization from Nsight was also considered. Similarly, this study was conducted from the perspective of throughput, and in the RTX 3090ti environment, using the reference C code, the authors achieved throughput improvements compared to Intel i9-12900KS at security level 1 (resp. 3 and 5). Signature generation achieved throughput improvements of ×108.55 (resp. ×109.79 and ×101.84), key generation ×64.67 (resp. ×63.24 and ×55.90), and verification ×72.32 (resp. ×64.46 and ×60.62). We applied Shiyu Shen et al.'s [38] analytical methods to analyze the optimality of the proposed optimization methodologies in Section VI-D.

III. Profiling of SPHINCS+ Operations

In this section, we present profiling results for SPHINCS+ operations. Through SPHINCS+ profiling, we identified performance workload and bottlenecks in the signing process of SPHINCS+. For profiling, we make use of SPHINCS+ reference code submitted to NIST PQC competition. Based on the SPHINCS+ profiling result, we concentrate on solving the performance bottleneck for maximum throughput.

Table II provides performance profiling results of SPHINCS+ by security level. In this experiment, the C code of SPHINCS+SHA256f-simple mode was used. The CPU used in this experiment was AMD Ryzen 9 5900X, and a single core is used for measurement. FORS digital signature increases in the rate as the security level increases. For a security level of 3 and 5, FORS digital signature process accounts for approximately 20% of overall running. The operation that causes the bottleneck in SPHINCS+ is hypertree operation. Hypertree's process handles WOTS+ based MSS operations. This involves generating leaf nodes in WOTS+ to handle MSS operations. In all security level, the WOTS+ leaf node generation process showed the largest workload point. And in the case of security level 1, WOTS+ leaf node generating process is approximately 89% of overall performance. Therefore, the maximum computational bottleneck occurs in the WOTS+ leaf node generation process, and it needs to be selected as a primary point for parallelization. In addition, the leaf node generation process that performs MSS-based operations also causes performance overhead in all process. In the case of FORS signature, the performance load in the leaf node generation process occurring in the FORS-subtree accounts for more than 50% of the entire FORS digital signature operation. Therefore, parallelization of the MSS leaf node generation process should be performed first.

TABLE II
Performance Profile of SPHINCS+SHA256f-Simple Mode Reference Code

Security Level	Algorithm	Detail	Clock	Percentage
Level 1	FORS Signature	FORS Leaf Node Gen	3,937,177	3.67
		FORS MSS Process	1,813,930	1.69
	WOTS+ Signature	WOTS+ Sign Process	5,847,626	5.45
	Hypertree MSS	WOTS+ Leaf Node Gen	95,566,010	89.01
		WOTS+ MSS Process	193,573	0.18
Total	SPHINCS+ - 128		107,358,316	100
Level 3	FORS Signature	FORS Leaf Node Gen	15,892,287	9.38
		FORS MSS Process	15,299,306	9.04
	WOTS+ Signature	WOTS+ Sign Process	8,083,345	4.78
	Hypertree MSS	WOTS+ Leaf Node Gen	129,818,537	76.64
		WOTS+ MSS Process	275,434	0.16
Total	SPHINCS+ - 192		169,368,909	100
Level 5	FORS Signature	FORS Leaf Node Gen	33,075,282	9.26
		FORS MSS Process	31,347,781	8.79
	WOTS+ Signature	WOTS+ Sign Process	8,959,695	2.50
	Hypertree MSS	WOTS+ Leaf Node Gen	283,273,204	79.31
		WOTS+ MSS Process	511,090	0.14
Total	SPHINCS+ - 256		357,167,052	100

IV. Parallel Implementation of SPHINCS+ Signature Scheme

In this section, we present our proposal for parallelizing the signing process of SPHINCS+ for maximizing the throughput. Note that key generation in SPHINCS+ is composed of WOTS+ leaf node generation and the corresponding hash tree (MSS) operation, which is the same as a single-layer operation in signing process (Refer to Fig. 1). Thus, we focus on describing the proposed optimization methods for the signing process of SPHINCS+ rather than additionally describing optimization details of the key generation.

A. Overall Configuration for Maximum Throughput

The goal of our implementation is to improve the throughput of SPHINCS+ signing. Thus, our software processes a number of messages at once by using the characteristics of GPUs. CUDA software is executed as a grid on a GPU and a grid is configured with CUDA blocks (Each CUDA block has multiple threads). Since such a GPU configuration heavily affects the performance, it is required to find the optimal configuration. Our SPHINCS+ GPU kernel structure is shown in Fig. 2. We design a parallel implementation of SPHINCS+ signature generation for multiple messages. We propose a parallel implementation of FORS digital signature, hypertree MSS digital signature, and hypertree WOTS+ digital signature among SPHINCS+ internal algorithms. In our proposed parallel scheme, each DSA uses different threads configuration. Thus, we call each kernel for each DSA parallel implementation operation.

Table III shows the GPU configuration of our SPHINCS+ implementation according to the security level. Since we assign each CUDA block to process a message, the number of blocks is the same as the number of messages to be processed. Namely, threads in a block cooperatively sign a message and each CUDA block in a grid independently processes a message. Since the major operations in a signing process are FORS, WOTS+, and MSS and their operational structures are

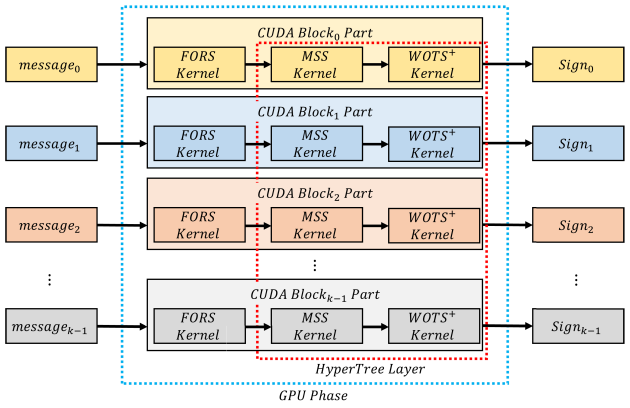

Fig. 2. The proposed SPHINCS+ GPU kernel structure.

TABLE III
GPU BLOCK AND THREAD CONFIGURATION FOR SPHINCS+ SIGNATURE GENERATIONS

Security Level	Algorithm	#CUDA Blocks	#CUDA Threads
Level 1	FORS	Number of Messages	64
	Hypertree MSS		21×8
	Hypertree WOTS+ Sign		22×35
Level 3	FORS	Number of Messages	256
	Hypertree MSS		21×8
	Hypertree WOTS+ Sign		11×51
Level 5	FORS	Number of Messages	512
	Hypertree MSS		16×16
	Hypertree WOTS+ Sign		8×67

different, we assign the different number of threads for each operation. Regarding thread assigning in FORS, we assign 2^h threads for constructing a subtree in parallel manner. Thus, since the heights of FORS in security level 1, 3, and 5 are 6, 8, and 9, respectively, we assign 2^6, 2^8, and 2^9 threads for each security level. Hypertree in SPHINCS+ consists of d layers and each layer consists of WOTS+ signing and MSS construction (MSS construction has two steps: WOTS+ key pair generation and computing authentication path). Since hypertree computation is the most time consuming in a signing process according to our profiling result, we parallelize not only each operation (WOTS+ signing and MSS construction) in a layer, but also entire layers. At first, regarding MSS construction, since the height of a tree in a layer is l and there are d layers, there are $l \times d - 1$ threads in a block. For example, the number of threads in a block is 21×8 for security level 1 and 3. Regarding WOTS+ sign in the hypertree computation, since the lengths of messages are 128-bit, 192-bit, and 256-bit for security level 1, 3, and 5, the numbers of divided parts (e) including their checksum part to be signed are 35, 51, and 67, respectively. Thus, $d \times e$ threads are required to compute the process of WOTS+ sign in all layers in parallel. However, in case of security level 3 and 5, the required number of threads (resp. 1,122 and 1,139) is more than the maximum number of threads (1,024) that can be created in a block. Thus, in case of security level 3 and 5, we make each thread to compute two divided parts.

B. Proposed Hypertree Parallel Methods

1) Analysis for Parallelizing Hypertree Operations: We analyze the hypertree process in detail in order to identify the possibility of parallel processing.

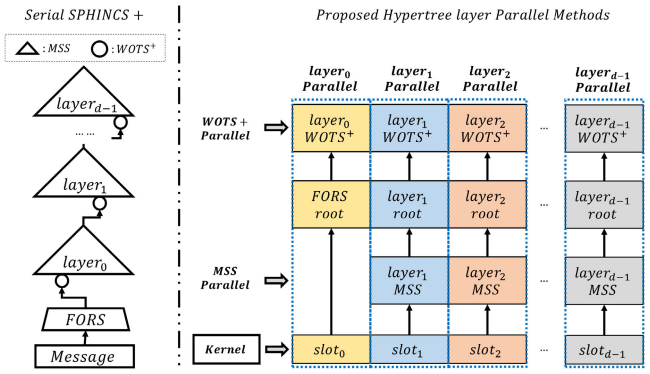

Fig. 3. Proposed parallel scheme for hypertree computation.

The process of hypertree computation in SPHINCS+ signing is shown in Fig. 3. Our goal is to parallelize processing all layers in the hypertree computation rather than processing each layer one-by-one. Parallelizing hypertree computation is not possible if the leaf node selection of MSS in a layer depends on the root node of MSS in the below layer. Fortunately, the leaf node selection of MSS in each layer does not depend on the root node in the below layer. Namely, the leaf node of MSS in each layer is predetermined before starting the process of hypertree computation. Followings are our code analysis for finding the independent execution of each layer in the hypertree computation.

- In the SPHINCS+ reference code, the MSS leaf node index is `idx_leaf`. `idx_leaf` which depends on `tree`. `idx_leaf` is changed in two processes: the input message compression function `hash_message()` and the hypertree layer process `idx_leaf = tree & ( (1 ≪ SPX_TREE_HEIGHT) - 1)`. `hash_message()` is called before the hypertree layer process. Therefore, the `idx_leaf` value can be pre-computed after the `hash_message()`. `hash_message()` is called before the hypertree layer process. `tree` is set to `hash_message()` and it is independent of the hypertree layer process. Therefore, the `idx_leaf` value can also be pre-computed after the `hash_message()`.
- In the WOTS+ signature process, checksum blocks and signature blocks are generated from input messages. At security level 1 (resp. 3 and 5), the number of WOTS+ signature blocks is 35 (resp. 51 and 67). In WOTS+ signatures, signature blocks are processed independently. Therefore, WOTS+ signatures are parallelizable.
- The hypertree MSS uses the WOTS+ leaf node generation function. In the WOTS+ leaf node generation function, the internal operation process is divided by the number of WOTS+ signature blocks. They can be parallelizable.
- WOTS+ signature generation is affected by the MSS root node. In the 0-th hypertree layer, WOTS+ signature input message is FORS signature value. All other hypertree layers perform MSS root node signing of lower layers.

2) Idea for Parallelizing Hypertree Operations: We propose a computation method in order to process all layers in the hypertree in parallel. Fig. 3 depicts of our method for parallelizing hypertree layer computation. Node selection and

computing authentication path in MSS are not affected by the root node of below MSS. However, WOTS+ signature depends on below MSS's root node. Thus, we need to change the order of hypertree computation: first computing MSS in all layers and then computing WOTS+ signatures of all layers by using the computed root nodes. From Fig. 3, MSS operations from layer 1 to layer $d-1$ are executed in parallel. As the result of MSS execution, the root nodes of all MSS are available, which are used by WOTS+ signing operation. WOTS+ signings in all layers are also executed in parallel.

3) Hypertree Layer Parallel Implementation Using CUDA: We explain our CUDA implementation about the proposed idea for parallelizing the hypertree computation regarding MSS parallel computation and WOTS+ signing parallel computation.

a) MSS parallel methods: In SPHINCS+ hypertree at security level 1 and 3, the layer is 22 and the MSS height is 3. Therefore, 21×2^3 MSS leaf nodes are required to process hypertree layer operations. At security level 5, 16×2^4 leaf nodes are required. Fig. 4 depicts the hypertree MSS parallel computation method in our implementation. In our SPHINCS+ implementation, one CUDA block performs operations on all layers of the hypertree. Therefore, we designed MSS parallel computation method for all layers, in one CUDA block. In our implementation of security level 1 and 3, the CUDA block uses $2^3 \times 21$ threads. Each thread processes one MSS leaf node generation (WOTS+ leaf node generation function). After that, all threads perform a total of 21 MSS root node merging processes in units of 2^3 threads. In our implementation of security level 5, the CUDA block uses $2^4 \times 16$ threads. In our security level 5 implementation, all threads perform a total of 16 MSS root node merging processes in units of 2^4 threads. As a result, our hypertree parallel MSS implementation applied the MSS leaf node generation parallel method and the MSS parallel node merging method.

b) WOTS+ Signature parallel methods: At security level 1 (resp. 3 and 5), a WOTS+ signature consists of 35 (resp. 51 and 67) message blocks and the number of layers is 22 (resp. 22 and 17). In our kernel, each thread processes one WOTS+ message block. Therefore, at security level 1, 35 threads collectively perform the generation of a single hypertree layer WOTS+ signature. We define the threads responsible for generating a 1-layer WOTS+ signature as one slot. That is, at security level 1, the 35 threads are organized into one slot, and one slot is responsible for generating WOTS+ signatures for a single layer. One CUDA block performs WOTS+ signature generation for all layers. Thus, in our security level 1 implementation, one block uses 35×22 threads and each thread computes a WOTS+ signature part. For security level 3 and 5, the total number of signature parts exceeds 1024 which is the maximum possible threads in a CUDA block. Thus, we make each thread to process two WOTS+ message block signature generation process in security level 3 and 5. Fig. 5 depicts the parallelization method for hypertree WOTS+ signing in security level 1.

C. Proposed FORS Signature Parallel Methods

At security level 1 (resp. 3 and 5), the process of FORS signing consists of 33 (resp. 33 and 35) subtree constructions.

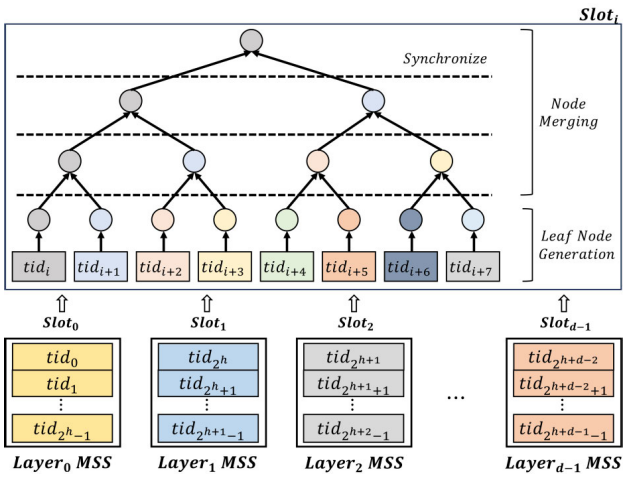

Fig. 4. Parallel computation of MSS in hypertree (MSS in all layers are executed in parallel).

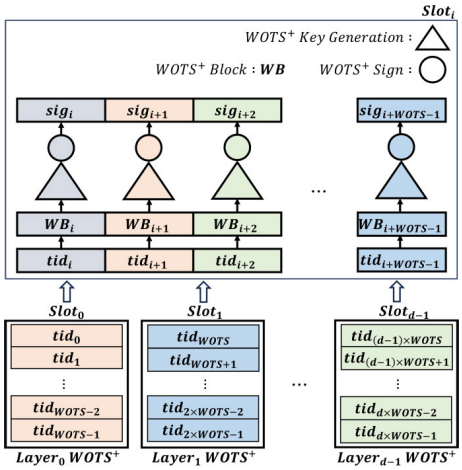

Fig. 5. Parallel computation of WOTS+ signing in hypertree (WOTS+ signings in all layers are executed in parallel).

Since each subtree is constructed independently, constructing subtrees can be executed in parallel. In our implementation, we take full advantage of shared memory for efficient subtree constructions. However, parallelizing all subtree construction will run out of shared memory. Thus, we decide to parallelize a subtree construction with t threads (2^6, 2^8, and 2^9 for security level 1, 3, and 5, respectively). Namely, t threads cooperatively construct a subtree and all subtrees are constructed by iterating building a subtree. Fig. 6 depicts the process of computing FORS with t threads. Note that t threads construct a subtree in parallel and this process is iterated k times. The construction of subtree (namely, generating leaf nodes and merging nodes for computing a root node) with t threads is efficiently implemented with using the shared memory.

D. Proposed MSS Parallel Method

MSS are extensively used in the FORS signatures generation and Hypertree operations. MSS primarily consists of two major phases: firstly, the generation of leaf nodes (FORS key generation or WOTS+ key generation), and secondly, the computation of the root node based on the generated leaf nodes. In particular, the generation of the MSS leaf node accounts for a significant portion of the overhead in

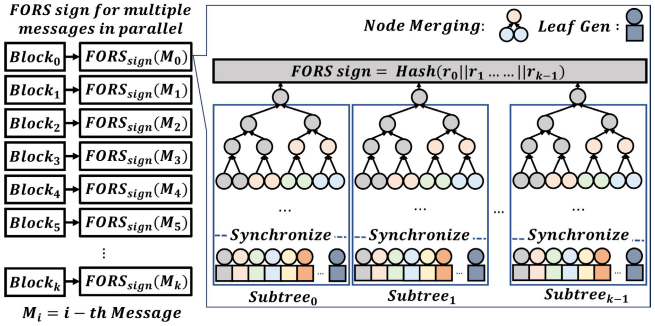

Fig. 6. FORS Parallel implementation using CUDA.

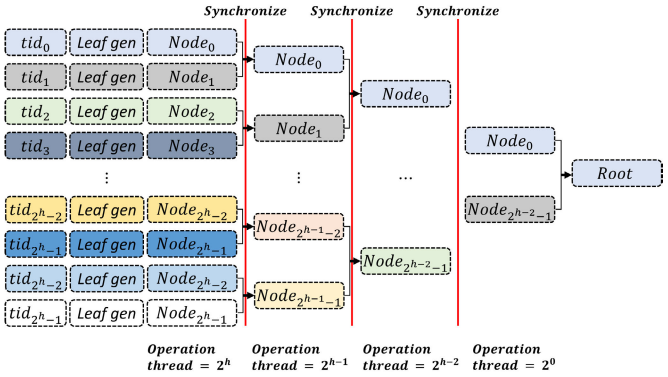

Fig. 7. Parallel methods of MSS.

SPHINCS+ signature generation. Therefore, in this section, we propose an efficient parallel method for MSS operations.

1) Parallel Method for MSS: Fig. 7 illustrates the proposed parallel method for MSS operations. In our parallel methods, a number of threads, corresponding to the number of leaves in the FORS-subtree (t) and the hypertree (h), are allocated. Each thread is responsible for generating a single leaf node. Subsequently, half of the threads used for leaf node generation are operated in the merging process of the two child nodes to generate the parent node. Synchronization (`__syncthreads()`) among threads is required at this stage since the values computed after this operation need to be used in the subsequent calculations. This process is repeated iteratively until, finally, the root node can be computed.

2) Efficient Shared Memory Utilization Approach for Merging Process in MSS: In our MSS merging nodes process implementation, each parent node is computed by concatenating its two child nodes and hashing it. Note that the computed leaf nodes need to be shared by threads for merging process for efficiency. In CUDA, the data in a block can be shared with the shared memory. When using the shared memory, the important thing is to minimize the bank conflict. For example, when each thread accesses the same memory area in the shared memory at the same time, bank conflict occurs, which results in serialized memory accesses. In addition, in CUDA memory access is performed in units of warps and the memory access range is 128-byte. Namely, the interval of memory accesses should be smaller than the warp memory access unit. Fig. 8 is our strategy for optimizing the usage of the shared memory for merging process. In our implementation, each thread stores the computed node value in even and odd-unit memory areas

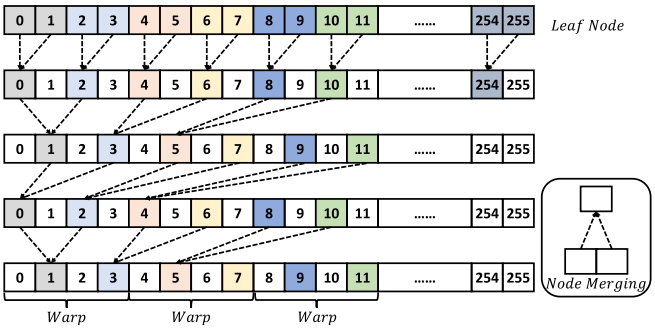

Fig. 8. Proposed shared memory access/store method.

of shared memory. For example, when a thread uses a node value stored in even-unit position of the shared memory, the thread stores the resulting value in odd-unit position. Our shared memory storage method can solve the memory conflict problem, which results in efficient usage of the shared memory. Thus, the memory accesses by threads are accomplished within the range of warp memory unit.

V. Parallel Implementation of SPHINCS+ Verify Scheme

We design a parallel implementation of SPHINCS+ digital signature verification for multiple messages. We propose a parallel implementation of FORS signature verification, MSS signature verification and WOTS+ signature verification among SPHINCS+ internal algorithms.

1) Comparison With Signature Generation: For SPHINCS+ signature verification, the verifier does not have a private key. Therefore, the verifier checks the signature success/failure of the algorithms through the data included in the SPHINCS+ signature. In Section IV-B.3, MSS leaf nodes are generated via a private key seed. Therefore, signature verification does not perform the MSS leaf node generation process. Similar to MSS, FORS's subtree leaf nodes are generated via SPHINCS+ private keys. In the WOTS+ signature generation process, the WOTS+ signing key is generated via SPHINCS+ private key. Therefore, the verifier does not perform the leaf node generation & key generation. And, the overall ratio of FORS leaf node generation & hypertree MSS key generation at security level 1 (resp. 3 and 5) is 92.68% (resp. 86.02% and 88.57%) (in Table II). Therefore, SPHINCS+ signature verification is faster than signature verification.

2) Limitations of the Hypertree Parallel Schemes in Verification: SPHINCS+ signature generation can be applied in the parallel scheme of the hypertree layer. However, in the SPHINCS+ signature verification process, the possibility of hypertree layer parallel scheme should be considered.

- In the SPHINCS+ reference code, the MSS leaf node index is `idx_leaf`. `idx_leaf` depends on `tree`. There is no difference between `tree` and `idx_leaf` in the SPHINCS+ signature generation/verification process.
- SPHINCS+ hypertree signature verification is performed as a WOTS+ verification → MSS root node verification process. The WOTS+ signature verification key is generated through the SPHINCS+ signature value and the

MSS root node of the lower layer. We analyzed the NIST SPHINCS+ verify code and the results are as follows [8]. The WOTS+ verification key generation process uses the `WOTS_pk_from_sig` function, and the hypertree layer MSS root node process uses `compute_root`. However, the i-th layer's WOTS+ verification key should be used in the $(i-1)$-th layer's MSS root node. Therefore, the `WOTS_pk_from_sig` function takes the *root* variable as an input argument. The `compute_root` function stores the MSS root value in root.

- As a result, each layer of SPHINCS+ signature generation does not affect each other. However, SPHINCS+ signature verification requires deriving the WOTS+ verification key through the root node of the lower layer. Therefore, SPHINCS+ signature verification requires another parallel implementation approach.

3) Hypertree Verification Parallel Implementation Method Using CUDA: SPHINCS+ hypertree verification process is subdivided into WOTS+ signature verification and MSS verification process. The MSS verification process is performed by calling the hash function h/d times. That is, the number of hash functions called in each layer of the hypertree in security level 1 (resp. 3 and 5) is 3 (resp. 3 and 4) times. The signature verification process in WOTS+ is similar to the signature generation process. WOTS+ signature verification splits the input message into WOTS+ message blocks. And it performs signature block verification for each block. The number of blocks in WOTS+ at security level 1 (resp. 3 and 5) is 35 (resp. 51 and 67).

Fig. 9 is a WOTS+ signature verification parallel scheme. In our SPHINCS+ signature verification implementation, one thread performs one WOTS+ message block signature verification. That is, our security level 1 (resp. 3 and 5) WOTS+ verification implementation performs WOTS+ signature verification through 35 (resp. 51 and 67) threads. Furthermore, our SPHINCS+ verification scheme is designed to process multiple messages in one kernel call. Therefore, our WOTS+ signature verification at security level 1 (resp. 3 and 5) defines 35 (resp. 51 and 67) threads as slots. We performed multiple WOTS+ signature verifications in one kernel through multiple slots. As a result, in our kernel, each slot performs parallelization of WOTS+ signature verification. In addition, each slot performs MSS verification and merging of signature verification values through shared memory.

4) FORS Verification Parallel Implementation Method Using CUDA: The signature verification of the FORS-subtree calls the hash function h times. As a result, performing a one FORS signature on security level 1 (resp. 3 and 5) signature calls the hash function a total of 33×6 (resp. 33×8 and 35×9) times. Our FORS signature verification parallel scheme exploits the independence of FORS-subtree. That is, each FORS-subtree does not affect other subtree. Our FORS signature verification kernel calls as many threads as there are FORS-subtree. Similarly, our FORS signature verification implementation verifies multiple signatures simultaneously. We configure slots as many as the number of FORS-subtree (33, 33, and 35 for each security level, respectively). We configure slots as many as the number of FORS-subtree. One slot performs the verification process for one FORS signature. As a result, our verification parallel method calls the CUDA kernel once and inputs information for multiple signatures. We construct a plot as many as the number of signatures received. The CUDA kernel processes SPHINCS+'s FORS and WOTS+ signature verification algorithms in parallel through slots.

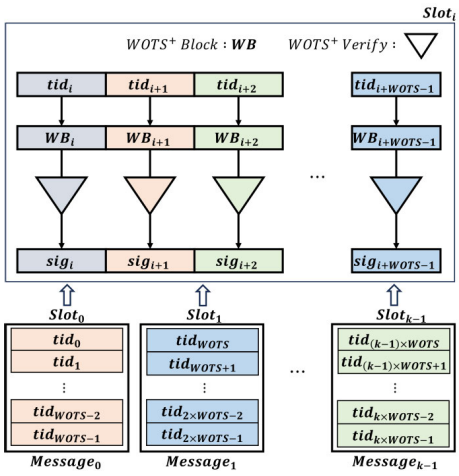

Fig. 9. WOTS+ Signature Verification Parallel Scheme.

VI. Evaluation

In this section, we compare our implementation with the previous implementations. We present our comparison experimented environment and setup for performance. We present performance measurements of our SPHINCS+ implementation and compare them with the previous implementations: SPHINCS implementation on GPU and SPHINCS+ FPGA implementations. First, we conducted a performance comparison of our implementation with the NIST PQC project SPHINCS+ reference code (version 3.0), which includes a C-language-based implementation and an AVX-2 based implementation, on both single-threaded and multi-threaded CPU perspectives [8]. We also compare SPHINCS+ the previous two implementations on FPGA [34], [35]. Sun et al. [23] have proposed SPHINCS parallelization schemes on various GPU architectures. Although SPHINCS and SPHINCS+ have different internal structures, the performance bottlenecks such as hypertree-based signature structure and WOTS+-based MSS are the same. Therefore, we conduct performance evaluations of implementations and Sun et al. [23].

We present clocks generated from SPHINCS+ reference code operations. Our CUDA implementation includes kernel function call time and memory copy time. The naive version is a simple porting of the C-language-based reference code, where each CUDA thread in CUDA handles a single algorithm. In our performance test, each experiment was performed 1000 times and the average value is presented. We used AMD Ryzen9 5900X (3.70GHz, single core) CPU environment, NVIDIA GeForce GTX 1070 (compute capability 6.1), RTX 2080ti (compute capability 7.5) and RTX 3090 (compute capability 8.6) GPU devices. Table V is a summary of GPU device specifications used in this paper. We used the Windows 10 operating system, the compiler used Visual

TABLE IV

PERFORMANCE COMPARISON OF SPHINCS+ SIGNING ACCORDING TO THE PROPOSED PARALLEL METHODS ON RTX 3090 (UNIT : 100 SIGN/SEC, **COM** : MSS LEAF NODE GENERATION AND NODE MERGING PROCESS PARALLEL METHODS. **HL** : HYPERTREE LAYER PARALLEL METHOD.**WP** : WOTS+ SIGNATURE GENERATION PARALLEL METHODS. **RATIO** : PERFORMANCE COMPARISON IN 512 BLOCK CASES)

Security Level	Algorithm		Number of Blocks							Ratio
			8	16	32	64	128	256	512	
Level 1	FORS	Naive	7.44	14.95	29.88	59.68	118.16	120.64	124.67	1
		Com	66.06	138.06	274.73	539.09	1051.77	1961.12	2010.78	16.25
	Hypertree MSS	Naive	0.62	1.25	2.50	5.04	10.15	20.05	20.67	1
		Com	4.86	9.98	19.80	39.05	77.09	149.29	158.59	7.67
		HL+Com	55.72	110.64	216.27	414.05	548.99	571.45	581.45	28.50
	Hypertree WOTS+	Naive	9.72	19.12	39.14	75.09	155.79	306.48	297.44	1
		WP	144.44	283.73	544.67	1023.82	1874.56	2119.82	2182.72	7.33
		HL+WP	400.19	747.65	1397.12	2376.75	2561.25	2849.51	3240.41	10.89
Level 3	FORS	Naive	1.58	3.17	6.00	11.54	22.96	41.94	42.05	1
		Com	42.61	87.89	174.63	343.19	647.37	664.80	670.78	15.95
	Hypertree MSS	Naive	0.42	0.83	1.68	3.35	6.63	13.16	13.28	1
		Com	3.09	6.23	12.27	24.62	46.88	88.85	101.45	7.64
		HL+Com	37.15	72.98	142.33	276.03	365.10	372.12	398.95	30.04
	Hypertree WOTS+	Naive	6.73	13.35	26.58	52.87	104.82	106.46	105.32	1
		WP	131.41	252.60	472.14	868.27	1365.96	1444.20	1674.19	15.89
		HL+WP	285.50	524.98	928.08	1525.14	1607.97	1648.81	1822.36	17.30
Level 5	FORS	Naive	0.31	0.64	1.25	2.55	5.06	10.00	19.70	1
		Com	31.86	64.92	128.33	248.03	300.49	345.89	376.61	19.11
	Hypertree MSS	Naive	0.20	0.40	0.81	1.63	3.25	6.45	6.65	1
		Com	3.11	6.29	12.45	24.27	46.17	90.16	98.78	14.85
		HL+Com	18.63	36.87	72.73	141.93	149.97	158.11	160.11	24.07
	Hypertree WOTS+	Naive	5.68	11.33	22.59	44.81	89.27	171.64	187.90	1
		WP	180.45	337.37	566.61	913.74	1084.79	1387.55	1389.30	7.39
		HL+WP	259.66	471.34	851.09	1281.16	1368.46	1575.61	1600.15	8.51

TABLE V

GPU ARCHITECTURE SPECIFICATIONS [39]

Classification	GTX 1070	GTX 1080	TITAN Xp	RTX 2080ti	RTX 3090
CUDA cores	1,920	2,560	3,840	4,352	10,496
Architecture	Pascal	Pascal	Pascal	Turing	Ampere
Graphic Processor	GP104	GP104	GP102	TU102	GA102
Streaming Multiprocessor	15	20	30	68	82
Bandwidth	256.3 GB/s	320.3 GB/s	547.6 GB/s	616.0 GB/s	936.2 GB/s
Base Clock	1,506 MHz	1,607 MHz	1,405 MHz	1,350 MHz	1,395 MHz
Compute Capability	6.1	6.1	6.1	7.5	8.6
Power Consumption	150W	180W	250W	250W	350W

Studio 2019 Release mode (x64) and CUDA version used 10.2. The performance of CPU reference code is measured on a single core.

A. SPHINCS+ Signature Generation Performance Analysis

Table IV is an analysis table of the performance improvement ratio for our parallel scheme. The performance improvement comparison (Ratio) measures the performance maximum point of the GPU architecture in each algorithm's operation, and is the performance ratio for that value. Hypertree MSS applies parallel implementation method for MSS and hypertree layer parallel implementation method. At security level 1 (resp. 3 and 5), our MSS parallel implementation method has a performance improvement of $\times$7.67 (resp. $\times$7.64 and $\times$14.85) compared to the Naive version, hypertree layer parallel implementation and MSS parallel implementation have performance improvements in $\times$28.13 (resp. $\times$30.04 and $\times$24.07). Our WOTS+ parallel implementation method applies WOTS+ signature parallelization method and hypertree parallel implementation method. At security level 1, our WOTS+ parallel implementation has a performance improvement of $\times$7.33 (resp. $\times$15.89 and $\times$7.39) compared to the Naive version. The result of applying our hypertree layer parallel scheme and WOTS+ parallel implementation at security level 1 is a performance improvement of $\times$10.89 (resp. $\times$17.30 and $\times$8.51).

Table VI shows the performance results of signature generation on GPU devices. The performance measurement results of Table VII are all parallel methods (FORS, Hypertree MSS, and WOTS+) specified in Table IV applied. At security level 1 (resp. 3 and 5) our SPHINCS+ generates up to 44,391 (resp. 24,997 and 11,401) signatures on the RTX 3090.

Table VII compares the performance of our SPHINCS+ sign implementation with other GPU and CPU SPHINCS+ implementations. Table VIII compares the performance of our SPHINCS+ sign implementation with other GPU and hardware SPHINCS+ implementations.

First, we compared the performance of our SPHINCS+ signature generation on an RTX 3090 with CPU performance. In this case, the Ref-AVX2 implementation among the existing CPU implementations showed the highest performance. To maximize this performance, we conducted a comparison in a multi-threaded environment. Ultimately, our SPHINCS+ signature generation demonstrated a performance enhancement of $\times$53.61, $\times$44.64 and $\times$32.39 (for security level 1, 3, and 5) compared to the multi-threaded environment.

Amiet et al. [35] published their SPHINCS+ optimization study on FPGA Artix-7, where the SPHINCS+ instance function is SHAKE. In contrast, our SPHINCS+ instance function is SHA-256. Despite this difference, the results of the performance analysis focusing on SPHINCS+ signature generation throughput are remarkable. Our SPHINCS+ implementation in the RTX 3090 environment has performance

TABLE VI
PERFORMANCE OF SIGNATURE GENERATION IN GPU DEVICES (UNIT : 1,000 SIGN/SEC)

Security Level	Platform	Number of Blocks									
		1	2	4	8	16	32	64	128	256	512
Level 1	GTX 1070	0.23	0.52	1.04	2.12	3.69	5.48	7.55	9.49	10.72	10.85
	RTX 2080ti	0.30	0.82	1.64	3.29	6.57	12.98	25.17	27.01	32.24	33.47
	RTX 3090	0.34	0.71	1.43	2.90	5.88	11.46	22.30	33.76	40.87	44.39
Level 3	GTX 1070	0.15	0.32	0.64	1.30	2.58	4.10	4.88	4.97	5.03	5.18
	RTX 2080ti	0.18	0.41	0.83	1.66	3.39	6.58	13.06	17.70	18.13	18.14
	RTX 3090	0.23	0.48	0.92	1.87	3.86	7.79	15.53	23.32	23.61	24.99
Level 5	GTX 1070	0.11	0.23	0.46	0.96	1.34	1.59	1.99	2.29	2.43	2.48
	RTX 2080ti	0.12	0.26	0.52	1.04	2.11	4.10	7.88	7.74	7.98	8.05
	RTX 3090	0.14	0.29	0.56	1.11	2.24	4.54	9.23	9.80	10.19	11.40

TABLE VII
SPHINCS+ SIGNATURE GENERATION PERFORMANCE COMPARISON (GPU & CPU ARCHITECTURES, UNIT: THROUGHPUT PER SECOND, MULTI THREADS : MAXIMUM THROUGHPUT)

Security Level	Algorithm	Version	Platform	Maximum Throughput	Ratio
Level 1	SPHINCS+ SHA256-128f simple	Ref-C	Ryzen9 5900X	34	1
		Ref-AVX-2 (Single Thread)		143	4.20
		Ref-AVX-2 (Multi Threads)		828 (12 threads)	24.35
		Our Works	**GTX 1070**	**10,851**	**319.14**
			RTX 2080ti	**33,470**	**984.41**
			RTX 3090	**44,391**	**1,305.61**
	SPHINCS-256 (ChaCha)	Sun *et al.* [23]	GTX 1080	5,152	-
			TITAN Xp	6,651	
			4 TITAN Xp	27,052	
Level 3	SPHINCS+ SHA256-192f simple	Ref-C	Ryzen9 5900X	22	1
		Ref-AVX-2 (Single Thread)		87	3.95
		Ref-AVX-2 (Multi Threads)		560 (16 threads)	25.45
		Our Works	**GTX 1070**	**5,186**	**235.72**
			RTX 2080ti	**18,141**	**824.59**
			RTX 3090	**24,997**	**1,136.22**
Level 5	SPHINCS+ SHA256-256f simple	Ref-C	Ryzen9 5900X	11	1
		Ref-AVX-2 (Single Thread)		44	4.00
		Ref-AVX-2 (Multi Threads)		352 (16 threads)	32.00
		Our Works	**GTX 1070**	**2,486**	**226.00**
			RTX 2080ti	**8,050**	**731.81**
			RTX 3090	**11,401**	**1,036.45**

TABLE VIII
SPHINCS+ SIGNATURE GENERATION PERFORMANCE COMPARISON (GPU & HARDWARE ARCHITECTURES, PPS : POWER CONSUMPTION PER A SIGNATURE)

Security Level	Version	Hash Algorithm	Platform	Maximum Throughput	PPS (Watt)
Level 1	Berthet *et al.* [34]	SHA-256	Xilinx XZU3EG	15.54	0.4
	Amiet *et al.* [35]	SHAKE-256	Artix-7	990.10	9.76
	Our Works	**SHA-256**	**GTX 1070**	**10,851**	**0.0138**
			RTX 2080ti	**33,470**	**0.0074**
			RTX 3090	**44,391**	**0.0078**
Level 3	Berthet *et al.* [34]	-			
	Amiet *et al.* [35]	SHAKE-256	Artix-7	854.70	9.69
	Our Works	**SHA-256**	**GTX 1070**	**5,186**	**0.0289**
			RTX 2080ti	**18,141**	**0.0138**
			RTX 3090	**24,997**	**0.0140**
Level 5	Berthet *et al.* [34]	SHA-256	Xilinx XZU3EG	0.57	0.474
	Amiet *et al.* [35]	SHAKE-256	Artix-7	396.82	9.80
	Our Works	**SHA-256**	**GTX 1070**	**2,486**	**0.0603**
			RTX 2080ti	**8,050**	**0.0310**
			RTX 3090	**11,401**	**0.0306**

improvements of ×44.83, ×29.24, and ×28.73 (for each security level) compared to the Amiet et al.'s [35] implementation, respectively. In the RTX 2080ti device, our SPHINCS+ implementation has performance improvements of ×33.80, ×21.22, and ×20.28 (for each security level) compared to the Amiet et al.'s [35] implementation, respectively. In the GTX 1070 device, our SPHINCS+ implementation has performance improvements of ×10.95, ×6.06, and ×6.26 (for each security level) compared to the Amiet *et al.*'s [35] implementation, respectively.

Quentin Berthet et al. [34] proposed a SPHINCS+ implementation method focusing on minimizing resource usage in Xilinx XZU3EG devices. An implementation by Quentin Berthet et al. [34] uses 0.4W at security level 1 and 0.474W at security level 5. Power consumption for the GPU architecture we used is 150W (GTX 1070), 250W (RTX 2080ti) and 350W (RTX 3090). We presented the power used per SPHINCS+ signature for power consumption analysis. On the GTX 1070 (resp. RTX 2080ti and RTX 3090) architecture, our security level 1 implementation is 0.0138W (resp. 0.0074W and 0.0078W) per signature, our security level 3 implementation is 0.0289W (resp. 0.0138W and 0.0140W) per signature, our secure level 5 implementation used 0.0603W (resp. 0.0310 and 0.0306W) per signature.

Sun et al. [23] proposed SPHINCS parallelization schemes on various GPU architectures. SPHINCS-256 provides security level 2^{128} on quantum computer [40]. The parameters of SPHINCS-256 are as follows [40]: SPHINCS-256 use a 256-bit hash digest length. The height of the hypertree (h) is 60, and d is 12. WOTS parameter w uses 16. The signature length is 41,000 bytes, the public key length is 1,056 bytes, and the private key length is 1,088 bytes. Our implementation performance compare with SPHINCS-256 at security level 1, which has the same security level as SPHINCS+ at security level 1. On the GTX 1070 architecture, our SPHINCS+ security level 1 implementation performance is up to 10,851 sign/sec. On the GTX 1080 (resp. TITAN Xp) architecture, Sun et al.'s [23] SPHINCS-256 (ChaCha) implementation performance is up to 5,152 (resp. 6,651) sign/sec [23]. According to Table V, it is expected that GTX 1080 provides almost 30% of improved performance compared with GTX 1070 since it has about 30% larger CUDA cores and memory bandwidth. Additionally, they boast faster bandwidth and base clocks when compared to the GTX 1070. Based on the provided information, we estimate that our SPHINCS+ Signature generation implementation, when benchmarked on the GTX 1080, will exhibit a maximum

TABLE IX
SPHINCS+ KEY GENERATION PERFORMANCE IN GPU DEVICES (UNIT : 1,000 KEYGEN/SEC)

Security Level	Platform	Number of Blocks									
		1	2	4	8	16	32	64	128	256	512
Level 1	GTX 1070	7.65	15.91	31.82	65.38	67.12	90.76	109.61	121.59	121.58	124.87
	RTX 2080ti	8.62	18.69	37.08	72.90	147.00	297.47	597.11	598.64	599.84	588.37
	RTX 3090	10.09	20.62	41.25	82.37	165.96	328.90	659.14	688.81	723.44	725.11
Level 3	GTX 1070	4.60	9.39	18.80	38.50	39.72	52.25	62.05	68.29	67.56	66.81
	RTX 2080ti	5.41	11.35	22.70	45.39	90.77	177.72	304.68	306.50	301.61	305.88
	RTX 3090	5.87	11.90	23.69	47.66	96.89	191.26	344.51	346.98	347.71	354.30
Level 5	GTX 1070	0.97	2.08	4.17	8.06	8.63	11.55	13.16	13.50	13.59	13.66
	RTX 2080ti	1.47	2.86	5.65	11.16	21.46	40.03	66.54	69.60	69.01	69.50
	RTX 3090	1.86	3.29	6.23	12.79	24.88	49.72	90.61	91.44	92.42	100.16

TABLE X
SPHINCS+ VERIFICATION PERFORMANCE IN GPU DEVICES (UNIT : 1,000 VERIFY/SEC)

Security Level	Platform	Number of Blocks									
		1	2	4	8	16	32	64	128	256	512
Level 1	GTX 1070	4.94	9.88	18.80	28.06	31.87	39.86	45.78	49.20	49.90	50.82
	RTX 2080ti	3.55	7.09	14.09	28.17	56.03	107.91	184.39	190.27	196.91	199.95
	RTX 3090	3.72	7.44	14.71	29.45	58.77	116.22	231.97	260.35	278.22	285.68
Level 3	GTX 1070	2.48	4.94	8.27	13.73	15.42	18.07	22.19	24.09	24.36	24.66
	RTX 2080ti	1.84	3.67	7.29	14.01	29.04	53.00	90.07	92.18	90.42	85.11
	RTX 3090	1.89	3.83	7.64	15.12	30.04	58.96	120.02	137.53	150.80	155.80
Level 5	GTX 1070	1.57	2.98	5.58	11.06	11.65	14.79	17.38	19.33	19.55	20.02
	RTX 2080ti	1.05	2.11	4.18	8.17	16.58	31.74	59.21	58.88	62.22	59.70
	RTX 3090	1.32	2.65	5.22	10.58	21.28	42.56	83.83	96.05	103.90	106.28

improvement of up to 30% compared to the performance on the GTX 1070, denoted as 14,106 sign/sec at security level 1, which is the ×2.72 of performance improvement compared to Sun et al.'s [23] work.

B. SPHINCS+ Key Generation Performance Analysis

The key generation implementation of SPHINCS+ applies the optimization methodologies for WOTS+ leaf node generation and hash tree used in the signature generation. The performance measured in the architecture proposed in the paper is presented in Table IX. As a result, our SPHINCS+ key generation implementation at security level 1 (resp. 3 and 5) achieved a throughput of up to 725,118 (resp. 354,309 and 100,168) on RTX 3090. Performance of SPHINCS+ key generation, as shown in Table XI, was also compared between GPU and CPU environments. Compared to the Ref-AVX2 implementation in a multi-threaded environment, our SPHINCS+ key generation at security level 1 (resp. 3 and 5) achieved processing speeds of ×121.10, ×64.84 and ×27.89 higher on the CPU's peak throughput. We estimate that our SPHINCS+ Key generation implementation, when benchmarked on the GTX 1080, will exhibit a maximum improvement of up to 30% compared to the performance on the GTX 1070, denoted as 162,337 keygen/sec at security level 1, which is the ×1.34 of performance improvement compared to Sun et al.'s [23] work.

C. SPHINCS+ Verification Performance Analysis

Table X is the performance test result of our SPHINCS+ verification implementation. The unit in Table X consists of 1,000 verify, and performance experiments were conducted on the RTX 3090 GPU architecture. Our SPHINCS+ hypertree verification implementation performed message signing

TABLE XI
SPHINCS+ KEY GENERATION PERFORMANCE COMPARISON (GPU & CPU ARCHITECTURES, UNIT: THROUGHPUT PER SECOND, MULTI THREADS : MAXIMUM THROUGHPUT)

Security Level	Algorithm	Version	Platform	Maximum Throughput	Ratio
Level 1	SPHINCS+ SHA256-128f simple	Ref-C	Ryzen9 5900X	689	1
		Ref-AVX-2 (Single Thread)		3,125	4.53
		Ref-AVX-2 (Multi Threads)		5,988 (4 threads)	8.69
		Our Works	**GTX 1070**	**124,875**	**181.24**
			RTX 2080ti	**599,841**	**870.59**
			RTX 3090	**725,118**	**1,052.42**
	SPHINCS-256 (ChaCha)	Sun *et al.* [23]	GTX 1080	120,471	-
			TITAN Xp	129,897	
			4 TITAN Xp	521,215	
Level 3	SPHINCS+ SHA256-192f simple	Ref-C	Ryzen9 5900X	609	1
		Ref-AVX-2 (Single Thread)		2,500	4.10
		Ref-AVX-2 (Multi Threads)		5,464 (8 threads)	8.97
		Our Works	**GTX 1070**	**68,293**	**112.13**
			RTX 2080ti	**305,881**	**502.26**
			RTX 3090	**354,309**	**581.78**
Level 5	SPHINCS+ SHA256-256f simple	Ref-C	Ryzen9 5900X	235	1
		Ref-AVX-2 (Single Thread)		724	3.08
		Ref-AVX-2 (Multi Threads)		3,592 (8 threads)	15.28
		Our Works	**GTX 1070**	**13,664**	**58.14**
			RTX 2080ti	**69,507**	**295.77**
			RTX 3090	**100,168**	**426.24**

through slots and CUDA blocks. Therefore, the number of signatures is equal to $Number\_of\_blocks \times slot$. That is, at security level 1 (resp. 3 and 5) $Number\_of\_blocks \times 28$ (resp. 12 and 6) signatures are verified in parallel. Unlike hypertree signature verification, our FORS signature verification implementation does not have slots due to the limitation of CUDA shared memory. However, our FORS signature verification performs verification on FORS-subtree in parallel.

TABLE XII

SPHINCS+ SIGNATURE VERIFICATION PERFORMANCE COMPARISON (GPU & CPU ARCHITECTURES, UNIT: THROUGHPUT PER SECOND, MULTI THREADS : MAXIMUM THROUGHPUT)

Security Level	Algorithm	Version	Platform	Maximum Throughput	Ratio
Level 1	SPHINCS+ SHA256-128f simple	Ref-C	Ryzen9 5900X	618	1
		Ref-AVX-2 (Single Thread)		2,059	3.33
		Ref-AVX-2 (Multi Threads)		5,312 (8 threads)	8.59
		Our Works	**GTX 1070**	**50,821**	**82.23**
			RTX 2080ti	**199,952**	**323.54**
			RTX 3090	**285,681**	**462.26**
	SPHINCS-256 (ChaCha)	Sun *et al.* [23]	GTX 1080	5,152	-
			TITAN Xp	6,651	
			4 TITAN Xp	27,052	
Level 3	SPHINCS+ SHA256-192f simple	Ref-C	Ryzen9 5900X	430	1
		Ref-AVX-2 (Single Thread)		1,460	3.39
		Ref-AVX-2 (Multi Threads)		4,288 (8 threads)	9.97
		Our Works	**GTX 1070**	**24,665**	**57.36**
			RTX 2080ti	**92,181**	**214.37**
			RTX 3090	**155,803**	**362.33**
Level 5	SPHINCS+ SHA256-256f simple	Ref-C	Ryzen9 5900X	424	1
		Ref-AVX-2 (Single Thread)		1,403	3.30
		Ref-AVX-2 (Multi Threads)		4,200 (8 threads)	9.90
		Our Works	**GTX 1070**	**20,026**	**47.23**
			RTX 2080ti	**62,223**	**146.75**
			RTX 3090	**106,282**	**250.66**

And, our implementation leverages CUDA blocks to perform multiple signature validation. As a result, our SPHINCS+ verification implementation at security level 1 (resp. 3 and 5) performs up to 285,680 (resp. 155,800 and 106,280) SPHINCS+ signature verifications per second on RTX 3090. As illustrated in Table XII, our SPHINCS+ verification performance results on the RTX 3090 environment demonstrated performance improvements of $\times$53.78 (resp. $\times$36.33 and $\times$25.30) at security level 1 (resp. 3 and 5), compared to the multi-threaded CPU verification. We estimate that our SPHINCS+ verification implementation, when benchmarked on the GTX 1080, will exhibit a maximum improvement of up to 30% compared to the performance on the GTX 1070, denoted as 66,067 verify/sec at security level 1, which is the $\times$12.82 of performance improvement compared to Sun et al.'s [23] work.

D. Detailed Profiling Results of the SPHINCS+ Implementation on GPU

Table XIII provides detailed results analyzed using NVIDIA Nsight Compute and System tools Ver. 2023.2.1, similar to the study in Shen *et al.*'s [38] paper, to analyze GPU resource utilization. From this table, we observe that our implementations almost reach the theoretical maximum throughput, except for FORS. Although FORS-subtree is designed with a parallel structure at the thread block level, the nature of the hash tree structure leads to a decrease in resource usage relative to input as the operations progress (*i.e.*, the number of threads used decreases by half as we process the upper layers of the hash tree). Therefore, achieved occupancy and throughput may be lower. Moreover, as evident in Table II, FORS constitutes a small proportion of the overall operations. Since FORS operates only once during the initial execution of SPHINCS+, its impact on Occupancy and Throughput is insignificant in the overall processing. In our study, we proposed optimization methodologies for the hypertree structure of SPHINCS+ and simultaneously presented optimal/parallel perspectives in terms of throughput for FORS, MSS, and WOTS+. These constitute the main contributions of our research, and we want to emphasize these contributions.

TABLE XIII

PROFILING RESULTS FOR IMPLEMENTATION OF SPHINCS+ ON GPU (SPHINCS+ SHA256-128F SIMPLE, **GPU** : RTX 3090, **SMEM** : SHARED MEMORY)

Classification	Occupancy (%)		Throughput (%)		Pipe Utilization (%)		Register
	Theoretical	Achieved	Compute	Memory	LSU	ALU	Numbers
FORS	50.00	23.24	34.49	9.58	11.73	42.26	79
WOTS+	52.08	46.49	79.45	1.27	1.43	89.24	64
MSS	25.00	23.30	83.71	2.08	1.11	96.47	120

Classification	Memory instruction			L1/TEX Cache Request		L2 Cache (MB)	
	Global	Local	SMEM	Loads	Stores	L1/TEX	Device
FORS	2.11M	7.84M	8.21M	7.25M	2.85M	48.77	16.87
WOTS+	960.0K	0	0	755.20K	204.80K	9.93	8.21
MSS	286.21K	5.24M	272.90K	3.47M	2.70M	409.79	200.53

VII. CONCLUSION

This paper presents an optimized SPHINCS+ implementation with high parallelism on GPU. Enhancing the throughput of signing and verification processes is crucial for global online payment platforms like Alipay, where extensive signing and verification procedures are imperative to ensure secure transactions. We discovered multiple points within the internal processes of SPHINCS+ that could be parallelized, and optimized them while leveraging GPU and CUDA features. Through comprehensive performance analysis, our proposed implementation significantly enhances the signing performance in SPHINCS+ compared to various previous works, including SPHINCS+ CPU reference implementation, GPU implementation, and FPGA implementation. Therefore, our study contributes to enhancing the efficiency of service providers and security protocols by improving the performance of SPHINCS+ in diverse application domains. Additionally, we have uploaded the source code for our SPHINCS+ implementation to 'https://github.com/kindongsy/TCAS-SPHINCSp_GPU' for you to review our research findings. Please refer to it.

REFERENCES

[1] J. Kelly. (2018). *Preview of Bristlecone Google's New Quantum Processor*. Accessed: Jan. 2, 2023. [Online]. Available: https://ai.googleblog.com/2018/03/a-preview-of-bristlecone-googles-new.html

[2] J. Chow, O. Dial, and J. Gambetta, "IBM quantum breaks the 100-qubit processor barrier," *IBM Res. Blog*, vol. 2, Nov. 2021.

[3] P. W. Shor, "Algorithms for quantum computation: Discrete logarithms and factoring," in *Proc. 35th Annu. Symp. Found. Comput. Sci.*, Santa Fe, NM, USA, 1994, pp. 124–134.

[4] The US National Institute of Standards and Technology. *Post-Quantum Cryptography Standardization Project*. Accessed: Oct. 19. [Online]. Available: https://csrc.nist.gov/Projects/post-quantum-cryptography

[5] R. Avanzi et al., "CRYSTALS-Kyber. Submission to the NIST post-quantum cryptography standardization project," Nat. Inst. Standards Technol. (NIST), Gaithersburg, MD, USA, Tech. Rep., 2020.

[6] L. Ducas et al., "CRYSTALS-Dilithium. Submission to the NIST post-quantum cryptography standardization project," Nat. Inst. Standards Technol. (NIST), Gaithersburg, MD, USA, Tech. Rep., 2020.
[7] P.-A. Fouque et al., "Falcon: Fast-Fourier lattice-based compact signatures over NTRU NIS," Nat. Inst. Standards Technol. (NIST), Gaithersburg, MD, USA, Tech. Rep., 2020.
[8] J.-P. Aumasson et al., "SPHINCS+—Submission to the 3rd round of the NIST post-quantum project NIS," Nat. Inst. Standards Technol. (NIST), Gaithersburg, MD, USA, Tech. Rep., 2020.
[9] L. Lamport. (1979). *Constructing Digital Signatures From a One Way Function*. Accessed: Aug. 7, 2023. [Online]. Available: https://www.microsoft.com/en-us/research/uploads/prod/2016/12/Constructing-Digital-Signatures-from-a-One-Way-Function.pdf
[10] J. Buchmann, E. Dahmen, S. Ereth, A. Hülsing, and M. Rückert, "On the security of the winternitz one-time signature scheme," *Int. J. Appl. Cryptogr.*, vol. 3, no. 1, pp. 84–96, 2013.
[11] A. Hülsing, "W-OTS+—Shorter signatures for hash-based signature schemes," in *Progress in Cryptology—AFRICACRYPT*, vol. 7918, A. M. Youssef, A. Nitaj, and A. E. Hassanien, Eds., Cairo, Egypt. Berlin, Germany: Springer, 2013, pp. 173–188.
[12] J. Pieprzyk, H. Wang, and C. Xing, "Multiple-time signature schemes against adaptive chosen message attacks," in *Selected Areas in Cryptography*, vol. 3006, M. Matsui and R. J. Zuccherato, Eds., Ottawa, ONT, Canada. Berlin, Germany: Springer, 2003, pp. 88–100.
[13] D. J. Bernstein, A. Hülsing, S. Kölbl, R. Niederhagen, J. Rijneveld, and P. Schwabe, "The SPHINCS$^+$ signature framework," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, L. Cavallaro, J. Kinder, X. Wang, and J. Katz, Eds. London, U.K., 2019, pp. 2129–2146.
[14] M. V. Yesina, Y. V. Ostrianska, and I. D. Gorbenko, "Status report on the third round of the NIST post-quantum cryptography standardization process," *Radiotekhnika*, no. 210, pp. 75–86, Sep. 2022.
[15] A. A. Badawi, B. Veeravalli, C. F. Mun, and K. M. M. Aung, "High-performance FV somewhat homomorphic encryption on GPUs: An implementation using CUDA," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2018, no. 2, pp. 70–95, 2018.
[16] W. Jung, S. Kim, J. H. Ahn, J. H. Cheon, and Y. Lee, "Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with GPUs," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2021, pp. 114–148, Aug. 2021.
[17] T. Baji, "Evolution of the GPU device widely used in AI and massive parallel processing," in *Proc. IEEE 2nd Electron Devices Technol. Manuf. Conf. (EDTM)*, Mar. 2018, pp. 7–9.
[18] D. Merrill, M. Garland, and A. Grimshaw, "High-performance and scalable GPU graph traversal," *ACM Trans. Parallel Comput.*, vol. 1, no. 2, pp. 1–30, Feb. 2015.
[19] 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. Secur. Technol. (ICCST)*, Oct. 2017, pp. 1–6.
[20] K. Iliakis, K. Koliogeorgi, A. Litke, T. Varvarigou, and D. Soudris, "GPU accelerated blockchain over key-value database transactions," *IET Blockchain*, vol. 2, no. 1, pp. 1–12, Mar. 2022.
[21] W. Pan, F. Zheng, Y. Zhao, W.-T. Zhu, and J. Jing, "An efficient elliptic curve cryptography signature server with GPU acceleration," *IEEE Trans. Inf. Forensics Security*, vol. 12, no. 1, pp. 111–122, Jan. 2017.
[22] *Dns Security Introduction and Requirements, Rfc 4033*. Accessed: Aug. 7, 2023. [Online]. Available: https://www.rfc-editor.org/rfc/pdfrfc/rfc4033.txt.pdf
[23] S. Sun, R. Zhang, and H. Ma, "Efficient parallelism of post-quantum signature scheme SPHINCS," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 11, pp. 2542–2555, Nov. 2020.
[24] N. Gupta, A. Jati, A. K. Chauhan, and A. Chattopadhyay, "PQC acceleration using GPUs: FrodoKEM, NewHope, and kyber," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 3, pp. 575–586, Mar. 2021.
[25] K. Lee, M. Gowanlock, and B. Cambou, "SABER-GPU: A response-based cryptography algorithm for SABER on the GPU," in *Proc. IEEE 26th Pacific Rim Int. Symp. Dependable Comput. (PRDC)*, Dec. 2021, pp. 123–132.
[26] W.-K. Lee, H. Seo, Z. Zhang, and S. O. Hwang, "TensorCrypto: High throughput acceleration of lattice-based cryptography using tensor core on GPU," *IEEE Access*, vol. 10, pp. 20616–20632, 2022.
[27] S. C. Seo, "SIKE on GPU: Accelerating supersingular isogeny-based key encapsulation mechanism on graphic processing units," *IEEE Access*, vol. 9, pp. 116731–116744, 2021.
[28] V. Srivastava, A. Baksi, and S. K. Debnath, "An overview of hash based signatures," Cryptol. ePrint Arch., Tech. Rep. 2023/411, 2023. [Online]. Available: http://eprint.iacr.org
[29] R. C. Merkle, "A certified digital signature," in *Advances in Cryptology—CRYPTO*, vol. 435, G. Brassard, Ed., Santa Barbara, CA, USA. New York, NY, USA: Springer, Aug. 1989, pp. 218–238.
[30] J. Buchmann, E. Dahmen, and A. Hülsing, "XMSS—A practical forward secure signature scheme based on minimal security assumptions," in *Post-Quantum Cryptography*, vol. 7071, B. Yang, Ed., Taipei, Taiwan. Berlin, Germany: Springer, 2011, pp. 117–129.
[31] NIST. (2023). *Three Draft Fips for Post-Quantum Cryptography*. [Online]. Available: https://csrc.nist.gov/news/2023/three-draft-fips-for-post-quantum-cryptography
[32] CUDA NVIDIA. (2007). *Compute Unified Device Architecture Programming Guide*. [Online]. Available: https://www.geeksforgeeks.org/introduction-to-cuda-programming/
[33] D. Amiet, A. Curiger, and P. Zbinden, "FPGA-based accelerator for post-quantum signature scheme SPHINCS-256," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2018, pp. 18–39, Feb. 2018.
[34] Q. Berthet, A. Upegui, L. Gantel, A. Duc, and G. Traverso, "An area-efficient SPHINCS$^+$ post-quantum signature coprocessor," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW)*, Portland, OR, USA, Jun. 2021, pp. 180–187.
[35] D. Amiet, L. Leuenberger, A. Curiger, and P. Zbinden, "FPGA-based SPHINCS$^+$ implementations: Mind the glitch," in *Proc. 23rd Euromicro Conf. Digit. Syst. Design (DSD)*, Kranj, Slovenia, Aug. 2020, pp. 229–237, doi: 10.1109/dsd51259.2020.00046.
[36] X. Zhao, B. Wang, Z. Zhao, Q. Qu, and L. Wang, "Highly efficient parallel design of Dilithium on GPUs," Cornell Univ., Ithaca, NY, USA, Tech. Rep. 2211.12265, 2022.
[37] S. C. Seo and S. An, "Parallel implementation of CRYSTALS-Dilithium for effective signing and verification in autonomous driving environment," *ICT Exp.*, vol. 9, no. 1, pp. 100–105, 2022.
[38] S. Shen, H. Yang, W. Dai, H. Zhang, Z. Liu, and Y. Zhao, "High-throughput GPU implementation of Dilithium post-quantum digital signature," 2022, *arXiv:2211.12265*.
[39] N. Cooperation. (2023). *Nvidia Gpu Specs*. Accessed: Aug. 7, 2023. [Online]. Available: https://www.nvidia.com/en-us/geforce/graphics-cards/compare/?section=compare-20
[40] D. J. Bernstein et al., "SPHINCS: Practical stateless hash-based signatures," in *Advances in Cryptology—EUROCRYPT*, vol. 9056, Sofia, Bulgaria. Berlin, Germany: Springer, 2015, pp. 368–397.

DongCheon Kim (Student Member, IEEE) received the B.S. degree from the Department of Information Security, Cryptology, and Mathematics, Kookmin University, where he is currently pursuing the master's degree in financial information security. His research interests include parallel programming environments, such as GPUs, post-quantum cryptography (PQC), and cryptographic module validation programs (CMVP).

HoJin Choi (Student Member, IEEE) received the B.S. degree from the Department of Information Security, Cryptology, and Mathematics, Kookmin University, Seoul, South Korea, and the M.S. degree in financial information security from Kookmin University. His research interests include the efficient implementation of cryptographic hash functions in high-end processes.

Seog Chung Seo (Member, IEEE) received the Ph.D. degree from Korea University, Seoul, South Korea, in 2011. From September 2011 to April 2014, he was a Research Staff Member with Samsung Electronics. From 2014 to 2018, he was a Senior Research Member with the Affiliated Institute, ETRI. He is currently an Associate Professor with Kookmin University, South Korea. His research interests include public-key cryptography, its efficient implementations on various IT devices, cryptographic module validation programs, network security, and data authentication algorithms.