

GRASP: Accelerating Hash-Based PQC Performance on GPU Parallel Architecture

Yijing Ning , Jiankuo Dong , Jingqiang Lin , Senior Member, IEEE, Fangyu Zheng , Yu Fu , and Fu Xiao , Senior Member, IEEE

Abstract—SPHINCS⁺, one of the Post-Quantum Cryptography Digital Signature Algorithms (PQC-DSA) selected by NIST in the third round, features very short public and private key lengths but faces significant performance challenges compared to other post-quantum cryptographic schemes, limiting its suitability for real-world applications. In scenarios involving a large number of concurrent signing or verification tasks, these performance bottlenecks become particularly critical. To address these challenges, we propose the GPU-based paRallel Accelerated SPHINCS⁺ (GRASP), which leverages GPU technology to enhance the efficiency of SPHINCS⁺ signing and verification processes. We propose an adaptable parallelization strategy for SPHINCS⁺, analyzing its signing and verification processes to identify critical sections for efficient parallel execution. Utilizing CUDA, we perform bottom-up optimizations, focusing on memory access patterns and hypertree computation, to enhance GPU resource utilization. These efforts, combined with kernel fusion technology, result in significant improvements in throughput and overall performance. Compared to previous works, our approach achieves the highest occupancy. Extensive experimentation demonstrates that our optimized CUDA implementation of SPHINCS⁺ achieves superior performance. Specifically, our GRASP scheme delivers throughput improvements ranging from 1.09× to 3.45× compared to state-of-the-art GPU-based solutions and surpasses the NIST reference implementation by over three orders of magnitude, highlighting a significant performance advantage.

Index Terms—PQC, hash-based digital signature, SPHINCS⁺, GPU, CUDA.

Received 1 April 2025; revised 9 December 2025; accepted 11 January 2026. Date of publication 16 January 2026; date of current version 13 March 2026. This work was supported in part by the National Cryptologic Science Fund of China under Grant 2025NCSF02005; in part by the National Natural Science Foundation of China under Grant 62572255 and Grant 62302238; in part by China Postdoctoral Science Foundation under Grant 2025T180411; and in part by Jiangsu Province Frontier Technology Research and Development under Grant BF2025004. Recommended for acceptance by S. Ghosh. (Corresponding author: Jiankuo Dong.)

Yijing Ning, Jingqiang Lin, and Yu Fu are with the School of Cyber Science and Technology, University of Science and Technology of China, and Anhui Province Key Laboratory of Digital Security, Hefei 230026, China.

Jiankuo Dong is with the School of Cyber Science and Technology, University of Science and Technology of China, and Anhui Province Key Laboratory of Digital Security, Hefei 230026, China, and also with the School of Computer Science, Nanjing University of Posts and Telecommunications, Nanjing 210023, China (e-mail: djiankuo@gmail.com).

Fangyu Zheng is with the School of Cryptology, University of Chinese Academy of Sciences, Beijing 100049, China.

Fu Xiao is with the School of Computer Science, Nanjing University of Posts and Telecommunications, Nanjing 210023, China.

Digital Object Identifier 10.1109/TC.2026.3654650

I. INTRODUCTION

PUBLIC key cryptography is a fundamental component of secure communication. However, it is widely known that current public key algorithms, such as RSA and ECC, can be broken by Shor's [1] and Grover's [2] algorithms on quantum computers in polynomial time. Recently, many research institutions and enterprises, including Google and IBM [3], have made significant progress in the field of quantum computing. To address this challenge, NIST has initiated a process to solicit quantum-safe cryptographic algorithms from around the world [4], which can replace traditional public-key cryptographic algorithms.

In 2022, NIST announced the results of Round 3 of the Post-Quantum Cryptography Standardization Process, which included four selected algorithms (CRYSTALS-Kyber [5], CRYSTALS-Dilithium [6], Falcon [7], and SPHINCS⁺ [8]) and four candidate algorithms (BIKE [9], Classic McEliece [10], HQC [11], and SIKE). Among the selected algorithms, CRYSTALS-Kyber, CRYSTALS-Dilithium, and Falcon are lattice-based, while SPHINCS⁺ is the only hash-based digital signature algorithm. SPHINCS⁺ is a stateless hash-based signature scheme, which means it does not require maintaining any state information. One of the primary advantages of SPHINCS⁺ is the relatively short lengths of its public and secret keys, which simplifies key management. However, a significant drawback is its performance; the signing and verification processes for SPHINCS⁺ are approximately 150 times and 50 times slower, respectively, than those for Dilithium on an x86/64-bit CPU [12]. Consequently, to facilitate the deployment of SPHINCS⁺ in various security protocols such as TLS, DNSSEC, and others, performance optimizations, particularly for the signing process, are essential. This need is especially pronounced in high-throughput environments, such as blockchain validation or cloud-based authentication services, where a large number of concurrent cryptographic operations must be handled efficiently.

Graphics Processing Units (GPUs) are auxiliary devices designed for computer graphics processing tasks, such as 3D design rendering. The architecture of a GPU is highly effective for parallelizing tasks and data operations due to its numerous cores. NVIDIA developed the Compute Unified Device Architecture (CUDA), which enables the design of computational processing methods for GPUs using C, C++, and Python [13]. Recently, substantial research has focused on utilizing GPUs to

enhance performance in cryptography. This includes improving the performance of blockchain systems [14] and accelerating cryptographic operations required for TLS in cloud services [15]. GPUs are now widely used for cryptographic acceleration, including homomorphic encryption [16], [17], public-key cryptography [18], [19] and so on.

A. Related Works

Sun et al. [20] proposed parallelization techniques for MSS, HORST, and WOTS⁺ to enhance the throughput of SPHINCS on GPU devices. Kim et al. [21] introduced parallel methods for FORS, WOTS⁺, MSS, and hypertree computation in SPHINCS⁺ on GPU platforms. However, their use of multiple CUDA kernels resulted in inefficient GPU utilization. Wang et al. [22] optimized SPHINCS⁺ based on kernel fusion. However, their design, where all threads within a CUDA kernel execute the same task and the number of computation rounds is dynamically adjusted based on the workload, results in low occupancy as well as high synchronization and memory access overheads. Several other studies have concentrated on the optimization of stateful hash-based digital signature algorithms on GPU [23], [24], [25].

Furthermore, significant research efforts have focused on optimizing SPHINCS⁺ performance across diverse hardware platforms. Quentin Berthet et al. proposed a methodology for implementing internal digital signature computations of WOTS⁺ and FORS on FPGA hardware [26]. Their approach prioritized optimizing resource utilization for embedded digital signature functions. In their implementation, SPHINCS⁺ SHA256-128f-simple mode on a Xilinx XZU3EG FPGA achieved a signature generation time of 64.34 ms and a verification time of 2.51 ms, utilizing 5,917 LUTs, 210 LUTRAMs, 4,933 FFs, and 0.5 BRAM. Amiet et al. conducted an optimization study aimed at accelerating SHA-3 and SHAKE functions specifically for SPHINCS⁺ on FPGA devices [27]. Their implementation of SPHINCS⁺ SHAKE-128f-simple mode on 7-series Xilinx FPGAs achieved a remarkable signature generation time of 1.01 ms, utilizing 47,991 LUTs, 72,505 FFs, and 11.5 BRAM.

Additionally, significant research has focused on accelerating various Post-Quantum Cryptography (PQC) algorithms using GPUs. Zhao et al. [28] meticulously optimized the Dilithium algorithm by adjusting batch sizes to leverage GPU thread efficiency. With a batch size of 15,360, they achieved substantial reductions in computation time compared to the reference C code running on an Intel Xeon Gold 6133 CPU. Specifically, on the P2000, V100, and T4 GPUs, time reductions of 3.51 $\times$, 11.18 $\times$, and 4.92 $\times$ were reported, respectively. Shen et al. [29] expanded on this optimization within GPU environments, targeting not only Dilithium but also hash algorithms. Their research, unlike previous studies, included optimization based on resource utilization analysis from Nsight. Similar to Zhao et al. [28], this study emphasized throughput improvements. On an RTX 4090, significant throughput enhancements were achieved compared to an Intel(R) XEON W7-2495X across security levels 2, 3, and 5. Specifically, signature generation improved by 213 $\times$, 228 $\times$, and 204 $\times$, respectively. Key gener-

ation showed improvements of 132 $\times$, 136 $\times$, and 117 $\times$, while verification exhibited enhancements of 152 $\times$, 142 $\times$, and 131 $\times$.

B. Our Contribution

Compared to other post-quantum cryptographic schemes, SPHINCS⁺ faces significant challenges regarding performance efficiency. The inherent complexity and computational demands of SPHINCS⁺ make it less performant than its counterparts, which is a critical issue for its adoption in real-world applications. GPUs, with their highly parallel architecture and numerous computational cores, offer substantial advantages in addressing these performance issues. They are particularly well-suited for tasks requiring extensive parallel processing, such as cryptographic operations. Despite these advantages, existing research has not fully tapped into the computational potential of SPHINCS⁺ on GPU platforms. To bridge this gap, we propose the GPU-based parallel Accelerated SPHINCS⁺ (GRASP). GRASP aims to harness the power of GPU technology to significantly enhance the efficiency of SPHINCS⁺ signing and verification processes. By fully leveraging the parallel processing capabilities of GPUs, GRASP targets large-task-count workloads, enabling efficient batch processing of SPHINCS⁺ operations in high-throughput environments such as cloud services and blockchain systems.

Our contributions are as follows:

- Firstly, through a comprehensive analysis of the signing and verification processes of SPHINCS⁺, GRASP explores parallelization strategies across various dimensions and proposes adaptable methodologies tailored to specific application requirements. This approach enables practitioners to select the optimal GPU thread configuration based on their specific needs for throughput and latency. In particular, GRASP identifies critical sections within the SPHINCS⁺ algorithm that benefit most from parallel execution, ensuring that the parallelization is both efficient and scalable. Furthermore, we provide a detailed explanation of the GPU thread configuration that achieves the highest throughput, demonstrating its superior performance through an in-depth analysis of thread utilization and synchronization overhead.
- Secondly, we leverage the capabilities of CUDA to perform a comprehensive, bottom-up optimization of the SPHINCS⁺ implementation on the GPU, targeting various performance aspects. A key focus is on optimizing memory access patterns, resulting in substantial performance gains. By minimizing memory latency and maximizing throughput, GRASP enhances the overall efficiency of cryptographic operations. Additionally, we refine the hypertree computation process, reducing the total number of WOTS⁺ signatures required in SPHINCS⁺. Our approach employs kernel fusion technology, which, compared to using multiple kernels, significantly enhances the efficiency of GPU resource utilization. This method consolidates multiple computational steps into a single kernel, reducing the overhead associated with kernel launches and improving the overall speed.

- Finally, by applying our optimization strategy, we conduct extensive experiments to achieve an efficient CUDA implementation of SPHINCS⁺ for signature generation and verification, leading to significant performance improvements. For signature generation, our implementation improves throughput by $1.37\times$ (resp. $1.79\times$ and $1.58\times$) on the RTX 4090 for 128f (resp. 192f and 256f) compared to [21]. On the RTX 3090, we observe throughput improvements of $1.09\times$ (resp. $1.29\times$ and $1.57\times$) for 128f (resp. 192f and 256f) compared to the latest work [22]. Additionally, for low-latency signature generation at security level 1 on the RTX 4090, our implementation achieves a $5.5\times$ reduction in throughput with a latency approximately $9.4\times$ lower than [21]. For signature verification, our implementation enhances throughput by $1.53\times$, $1.80\times$, and $3.45\times$ on the RTX 4090 across different security levels. Compared to the NIST official reference implementation, our performance improvement exceeds three orders of magnitude, demonstrating a significant performance advantage.

The rest of the paper is organized as follows. Section II introduces the background related to this paper. Section III presents our specific optimization implementation scheme on GPU. Section IV shows the experimental results. Section V concludes this paper.

II. PRELIMINARY KNOWLEDGE

This section introduces the preparatory knowledge related to our work. Firstly, we briefly introduce the GPU architecture. Then, we present a detailed description of the SPHINCS⁺ signature scheme.

A. GPU Architecture

As a computing platform, GPU has extremely broad application scenarios, such as machine learning, deep learning and artificial intelligence. In the field of cryptographic engineering, the utilisation of general-purpose computing on GPUs offers a significant computational advantage in accelerating cryptographic algorithms. In the context of new parallel computing architectures, the GPU can be regarded as a kind of parallel data processing device. As a novel parallel computing architecture, the GPU can be conceptualised as a parallel data-processing device, comprising thousands of stream processors that can perform massively parallel computations by making optimal use of various memories on the GPU to complete diverse computational tasks at high speeds. The GPU can be employed as a high-performance realisation platform for cryptographic computation.

In this paper, we select NVIDIA GPUs and CUDA framework for its popularity. Table shows the specifications of the GPU/DCU architecture used in our paper. NVIDIA GPUs have thousands of Streaming Multiprocessors (SM). An SM is the basic unit that can execute GPGPU programs. SM consists of several CUDA cores. Usually, dozens of cores form a group, and the group corresponds to the block, which has its own shared memory. Within a block, many threads can be launched. Each

thread corresponds to a core. In the hardware, there are registers that can only be used by blocks. Threads in the same block will share registers. So the more threads are launched, the fewer registers can be used, which constrains the scale of parallelism. By CUDA programming, the kernel function can call GPU resources. Due to the parallelism principle in hardware design, the smallest execution unit in SM is called a warp. All threads in a warp execute the same instruction, which maximizes GPU efficiency when the instructions executed by all threads remain consistent.

B. The Structure of SPHINCS⁺

SPHINCS⁺ is a stateless hash-based signature scheme comprising a number of sub-algorithms, including WOTS⁺, XMSS, hypertree and FORS.

1) *WOTS⁺*: Winternitz One-Time Signature (WOTS) is an instance of the One-Time Signature (OTS) scheme [30]. OTS restricts a private key to be used for exactly one message, as its security quickly decreases with reuse. WOTS⁺ is a variants of WOTS [31]. The security parameter and message length of WOTS⁺ is denoted by n . The Winternitz parameter is w . As the value of w increases, the length of the signature decreases, while the time required for signature generation increases. Let l be the number of blocks in an uncompressed WOTS⁺ private key, public key, and signature, where

$$l = l_1 + l_2, l_1 = \left\lceil \frac{n}{\log(w)} \right\rceil, l_2 = \left\lceil \frac{\log(l_1(w-1))}{\log(w)} \right\rceil + 1.$$

The core idea of WOTS⁺ is to use hash function chains starting from random values. These random values together act as the secret key. The public key consists of the ends of all chains. The signature is computed by mapping the message to one intermediate value of each function chain.

The WOTS⁺ private key sk consists of l random blocks ($sk = (sk_1, \dots, sk_l), sk_i \in \{0, 1\}^n$). The uncompressed WOTS⁺ public keys $\{pk_i\}_{i=1}^l$ is derived by applying hash function $\mathbf{H}$ iteratively for $w-1$ times to each of the blocks in the private key ($pk_i = \mathbf{H}^{w-1}(sk_i)$), then the public keys are compressed to a block using $\mathbf{H}$ ($pk = \mathbf{H}(pk_1, \dots, pk_l)$). In the process of signing, WOTS⁺ initially transforms the message m into l_1 integers $m_i \in \{0, 1, \dots, w-1\}$ using the base w representation. This is followed by compute a checksum $c = \sum_{i=1}^{l_1} (w-1-m_i)$, represented as a string of l_2 base- w values $c = (c_1, \dots, c_{l_2})$. This yields the following result: $\theta = (m_1, \dots, m_{l_1}, c_1, \dots, c_{l_2})$. Apply $\mathbf{H}$ to each private key sk_i for θ_i times ($s_i = \mathbf{H}^{\theta_i}(sk_i)$), the WOTS⁺ signature of message m is $s = (s_1, \dots, s_l)$. The verifier can then recompute the checksum and apply $\mathbf{H}$ to each block for $w-1-\theta_i$ times ($pk'_i = \mathbf{H}^{w-1-\theta_i}(s_i)$). Finally compress those values to an n -bit public key pk' using $\mathbf{H}$ ($pk' = \mathbf{H}(pk'_1, \dots, pk'_l)$). The verifier accepts this signature if $pk = pk'$.

2) *MSS and XMSS*: MSS (Merkle Signature Scheme) is a structure that combines Merkle tree and hash function [32]. In order to sign $2^{h'}$ messages, the signer generates $2^{h'}$ WOTS⁺ key pairs and constructs a binary tree of height h' using these

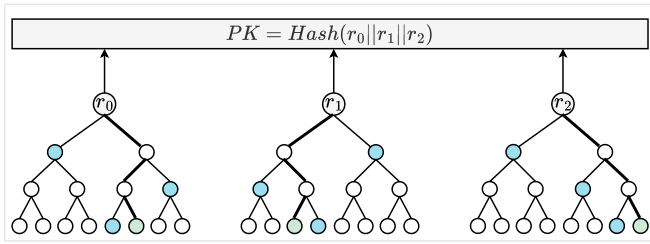


Fig. 1. An example of a FORS signature with $k = 3$ and $a = 3$, on message 101 011 111.

$2^{h'}$ public keys as leaf nodes. Then, signer repeatedly applies **H** to each pair of child nodes to generate the corresponding parent node until reaching the root of the tree, which is the public key for the MSS. XMSS (eXtended Merkle Signature Scheme) is a variant of MSS [33]. The structure of XMSS is similar to that of MSS, but it includes a process to XOR a random mask value during node merging.

To sign a message, the signer picks one of the WOTS⁺ leaf nodes and publishes the WOTS⁺ signature as well as all siblings of the nodes on the path from the leaf to the root, which is referred to as the “authentication path”. The verifier first derives the WOTS⁺ public key from the signature and then uses the nodes on the authentication path and their siblings to reconstruct the root.

3) *Hypertree*: Hypertree is a tree of MSS trees. A hypertree consists of d layers. The leaf nodes of the trees on the bottom layer are used to sign messages (in SPHINCS⁺, the message is a FORS public key), while the leaf nodes of trees on other layers are used to sign the root nodes of the trees immediately beneath them. The total height of the hypertree is specified as h , therefore, the height of MSS tree at each layer is $\frac{h}{d}$. During key generation, only the top-most tree is generated to derive the public key. The rest of the trees can be generated when needed.

4) *FORS*: Forest of Random Subset (FORS) is a few-time signature scheme that allows a private key to sign multiple messages, with security decreasing as the number of signatures increases [34]. It is an improved version of HORS proposed in SPHINCS⁺. FORS includes k Merkle trees with height a , and can be used to sign messages of $k \cdot a$ bits. To construct the FORS public key, the signer first construct k Merkle trees with height a from $k \cdot 2^a$ leaf and then compress the root nodes using **H**. The compressed result is the FORS public key. Given a message of $k \cdot a$ bits, the signer split it into k blocks of a bits. Each block value is used as the selection index of leaf node in the corresponding Merkle tree. The signature consists of these nodes and their respective authentication paths. The verifier reconstructs all the root nodes from the signature, compresses the root nodes using **H**, and compares the result value against the public key. Fig. 1 presents a simple diagrammatic representation of FORS.

The overall process of signing in SPHINCS⁺ is shown in Fig. 2. 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 $d \cdot 2^{\frac{h}{d}}$ in the MSS like SPHINCS. With the

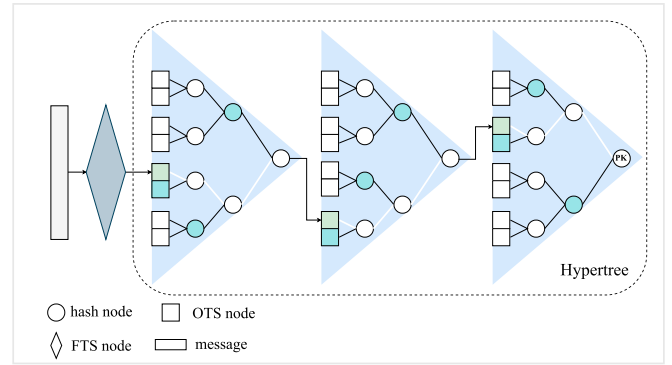


Fig. 2. An overview of SPHINCS⁺ structure with $h = 9$ and $d = 3$.

hypertree mechanism, single MSS of height h is divided into d layers containing a subtree of height $\frac{h}{d}$. 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 computation of authentication path. Through the signing process, 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 highest layer. The details of FORS, WOTS⁺, MSS, and hypertree can be found in the specification document [35].

C. SPHINCS⁺ Modes and Their Parameters

Table II shows the parameter sets of SPHINCS⁺. SPHINCS⁺ has three distinct security levels: level 1, level 3 and level 5. Each level offers a different strength of security, with the strength increasing as the level rises. However, the time required for operation also increases. Furthermore, 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. For instance, SPHINCS⁺-f mode on security level 5 is faster than SPHINCS⁺-s mode as faster as 16 $\times$, 9 $\times$, and 2 $\times$ for key generation, signing, and signature verification, respectively [35]. However, the signature size of SPHINCS⁺-f is much longer than that of SPHINCS⁺-s mode. In addition, SPHINCS⁺ is subdivided into robust model and simple model, depending on whether or not random values are used as mask to XOR the input message of the hash function. The definitions of all parameters are provided in the preceding section. A variety of hash functions can be employed to construct distinct SPHINCS⁺ instantiations. In the SPHINCS⁺ reference code, the underlying hash functions selected are SHA-256, SHAKE-256 and Haraka. In this paper, we select the simple and fast mode for our implementation, and the underlying hash function is SHA-256. However, our scheme is equally effective in all modes.

III. OPTIMIZED IMPLEMENTATION OF SPHINCS

The detailed optimization strategies of SPHINCS⁺ on GPU are discussed in this section. Firstly, parallel strategy analyses were conducted in various aspects based on the SPHINCS⁺

TABLE I
GPU ARCHITECTURE SPECIFICATIONS

	GTX 1080	RTX 3090	RTX 4090
Multiprocessors	20	82	128
CUDA Cores	2560	10496	16384
Memory clock	5005 MHz	1395 MHz	10501 MHz
Memory bus	256-bit	384-bit	384-bit
Power consumption	180W	350W	450W

signing process. Then, based on previous analyses, parallel strategies for the SPHINCS⁺ signing process aimed at maximizing throughput and minimizing latency are proposed. Subsequently, we analyzed the process of SPHINCS⁺ signature verification and proposed our parallel scheme for signature verification. Finally, we introduce the other optimization strategies we employed.

A. Analysis of Parallel Strategies in Different Aspects for SPHINCS⁺ Signature Generation

When utilizing GPUs to execute large-scale computing tasks, an intuitive strategy is to assign an independent task to each thread for processing. The advantage of this approach lies in its ability to fully utilize GPU thread resources without causing thread wastage, thereby achieving exceptionally high throughput. However, due to the limited computational capability of individual threads, this method often results in significant task processing latency. This issue is particularly pronounced in tasks like SPHINCS⁺ signature generation, which inherently have low computational efficiency. The substantial latency caused by single-threaded processing is often unacceptable in practical applications. Therefore, to achieve both high throughput and low processing latency, it is essential to adopt a parallel computing approach where a single task is divided among multiple threads for collaborative processing. In this section, we explore the feasibility of partitioning tasks across multiple aspects for parallel computation during the SPHINCS⁺ signature generation process.

1) *FORS*: As illustrated in Fig. 1, FORS is composed of k independent subtrees that can be computed in parallel. A natural approach is to utilize t threads for the computation of each subtree. Fig. 3 presents a simple example of multiple threads computing a single tree of height 3 in parallel. It can be observed that for a tree of height h , during the computation of the root node, the number of node merging processes per layer decreases incrementally from 2^{h-1} at the lowest layer to 1 at the topmost layer. Thus, we can infer the conclusion that, for $t > 1$, the process of root node computation (node merging) leads to inefficient thread utilization. Furthermore, Additionally, to ensure the correctness of the computation process, h synchronizations are required between different threads during the computation, which introduce additional overhead and affect overall computational efficiency. Therefore, to maximize throughput, we should set $t = 1$, meaning that each subtree is computed by a single thread. However, if the objective is to minimize latency,

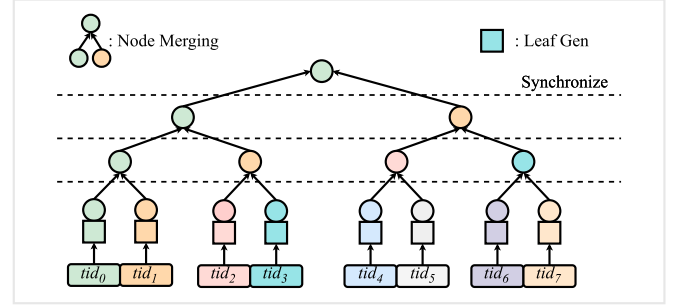


Fig. 3. Parallel strategy for Merkle tree.

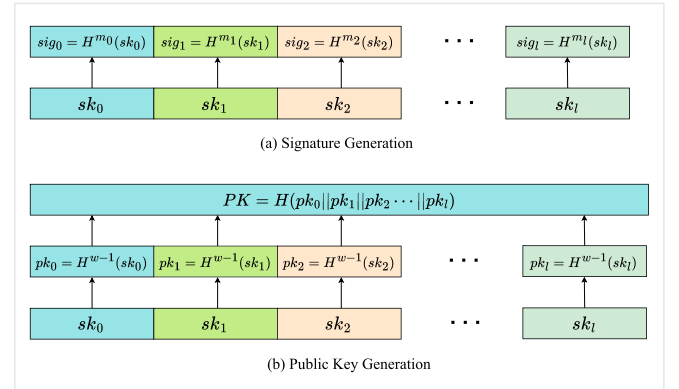


Fig. 4. Computation process of WOTS⁺.

t should be as large as possible to reduce the computational workload for each thread.

2) *Hypertree*: As illustrated in Fig. 2, the hypertree consists of d layers. The selected MSS leaf nodes at the upper layer sign the MSS root nodes located in the layer below using WOTS⁺. We explored possible parallel strategies from two different perspectives.

- In WOTS⁺, there are l blocks, as shown in Fig. 4, in the context of the WOTS⁺ key generation and signing process, each WOTS⁺ block can be processed by a single thread. However, since the WOTS⁺ public key is derived by compressing the public keys of each block, and this compression process requires only one thread, it results in thread wastage. Furthermore, as shown in Fig. 3, thread wastage also occurs when computing the MSS tree. The parallel strategy in this dimension is not suitable for maximizing throughput.
- According to the analysis in [21], we can achieve parallel computation across the layers by first computing the MSS trees at each layer of the hypertree and then generating WOTS⁺ signatures based on the root node of MSS tree. As depicted in Fig. 3, and similarly to FORS, we can utilize t threads to compute each layer, setting $t > 1$ leads to thread wastage. To achieve maximum throughput, t should be set to 1 in this context as well.

TABLE II
PARAMETERS OF SPHINCS⁺ IN DIFFERENT VERSIONS

Version	n (byte)	w	h	d	a	k	Public Key Len (byte)	Private Key Len (byte)	Signature Len (byte)
SPHINCS ⁺ - 128s	16	16	63	7	12	14	32	64	7856
SPHINCS ⁺ - 128f	16	16	66	22	6	33	32	64	17088
SPHINCS ⁺ - 192s	24	16	63	7	14	17	48	96	16224
SPHINCS ⁺ - 192f	24	16	66	22	8	33	48	96	35664
SPHINCS ⁺ - 256s	32	16	64	8	14	22	64	128	29792
SPHINCS ⁺ - 256f	32	16	68	17	9	35	64	128	49856

B. The SPHINCS⁺ Signing Implementation to Maximize Throughput

1) *Differences From the Prior Work:* To maximize the throughput of parallel computing, it is crucial to design parallelization strategies tailored to the specific characteristics of each computational module, thereby minimizing idle threads during execution. However, as discussed in section II-B, the computational patterns of the individual submodules in SPHINCS⁺ vary significantly, making it highly challenging to develop a unified and efficient parallelization strategy that works across all submodules. To address this issue, Kim et al. [21] proposed a method that employs multiple CUDA kernels, with each kernel dedicated to handling the computational tasks of a specific submodule. Their approach allows for the parallelization strategy to be customized to the unique requirements of each submodule. Specifically, for the k Merkle trees in FORS, [21] employs 2^a threads to compute each Merkle tree in parallel, iterating k times as illustrated in Fig. 3. For the d -layer Merkle trees within the hypertree, [21] first utilizes $d \cdot 2^{\frac{h}{2}}$ threads to compute the leaf node values and root node values of each layer in parallel. Subsequently, $d \cdot l$ threads are employed to compute the WOTS⁺ signatures for each layer concurrently. While effective in optimizing the performance of individual submodules, this method requires intermediate results to be exchanged between submodules, necessitating data transfers across different CUDA kernels. [21] implemented these data exchanges using global memory, the high latency of global memory access inevitably introduces additional overhead.

Wang et al. [22] proposed a parallel scheme based on kernel fusion. In contrast to [21], where multiple threads form a thread group and each thread group computes a single Merkle tree, they introduced a so-called “node first” approach. In their implementation, a CUDA kernel contains only one thread group composed of a fixed number of threads, which sequentially computes nodes at the same height across multiple Merkle trees. During each iteration, different threads may process nodes from different Merkle trees. The key advantage of [22] is its strategy of aggregating nodes from multiple Merkle trees into the same thread group when computing higher-level nodes, effectively mitigating thread underutilization and improving overall parallel efficiency. This strategy is effective in low-task-count scenarios. However, it also entails certain trade-offs. First, the approach used to mitigate thread idling leads to a small number of active threads in the CUDA kernel. Second, because

threads within the same thread group are distributed across different thread blocks, intermediate data must be stored in global memory, and global synchronization becomes necessary. Under workloads with a large number of parallel tasks, these design choices may result in suboptimal utilization of GPU resources and introduce additional overhead.

To address these limitations, based on the analysis presented in Section III-A, we propose a novel parallel strategy based on kernel fusion. Furthermore, we provide an optimal thread configuration tailored to the parameters of SPHINCS⁺. Our approach eliminates the need for data transfer across kernels, minimizes the number of idle threads, and allows the number of launched threads to be dynamically adjusted according to the amount of tasks to be processed. As a result, our scheme significantly improves the utilization of GPU resources.

2) *Our Parallel Computing Strategy:* CUDA software is executed as a grid on a GPU and a grid is configured with CUDA blocks (Each CUDA block has multiple threads), which means we can generate signatures for a number of messages at once and each message can be processed with multiple threads. The number of threads used to process each message, which we refer to as the thread configuration, can significantly impact performance. Therefore, we need to find the optimal configuration to achieve the best performance. It is important to emphasize again that our implementation utilizes only a single CUDA kernel, which means that the same thread configuration is used across all submodules throughout the entire SPHINCS⁺ signing process. Based on the analysis results from section III-A, we designed a parallel strategy for SPHINCS⁺ that can achieve the highest throughput.

Table III shows the thread configurations for our high-throughput SPHINCS⁺ signing implementation at various security levels. As previously analyzed, to maximize signature generation throughput, we allocate one thread to each layer in the hypertree and each subtree in FORS. Assuming the number of threads used to generate a single signature is t , as illustrated in Fig. 5, it can be deduced that, to minimize thread wastage, a multiple of t should approximate the number of layers (d) in the hypertree. Similarly, as depicted in Fig. 6, a multiple of t should approximate the number of subtrees (k) in FORS. It is worth noting that due to the hypertree being the most time-consuming step in the SPHINCS⁺ signature generation, our thread configuration prioritizes satisfying the requirements of the hypertree.

TABLE III
 THREAD CONFIGURATION FOR SPHINCS⁺
 SIGNATURE GENERATION

Version	d	k	CUDA Threads
128f	22	33	11
192f	22	33	11
256f	17	35	18
128s	7	14	7
192s	7	17	9
256s	8	22	8

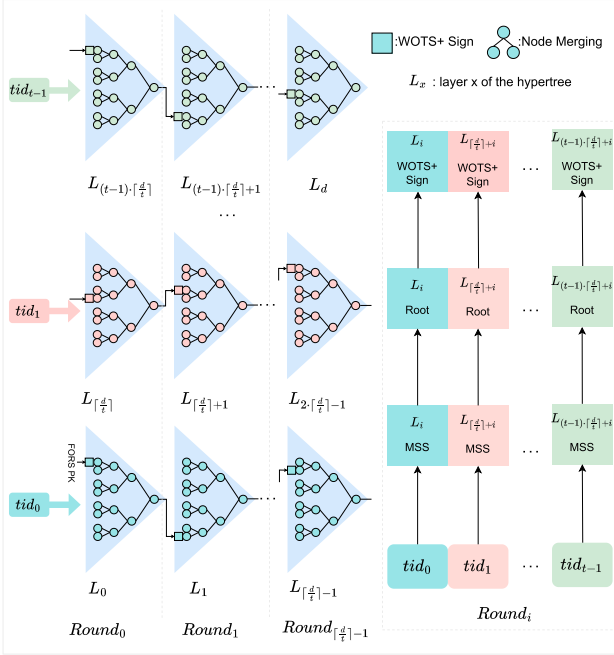


Fig. 5. Parallel strategy for hypertree.

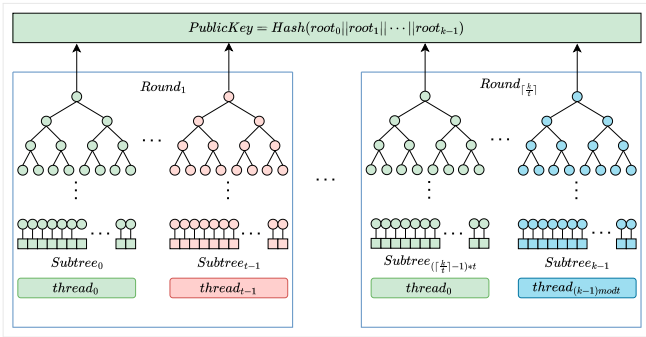


Fig. 6. Parallel strategy for FORS.

Fortunately, for 128f and 192f, the values of d and k are 22 and 33 respectively, with a greatest common divisor of 11. This allows us to use 11 threads to perform two rounds and three rounds of computation for hypertree and FORS, respectively, without any thread wastage. However, the values of d and k are 17 and 35 for 256f, we use 18 threads to generate a signature,

 TABLE IV
 THREAD CONFIGURATION FOR LOW-LATENCY SIGNING

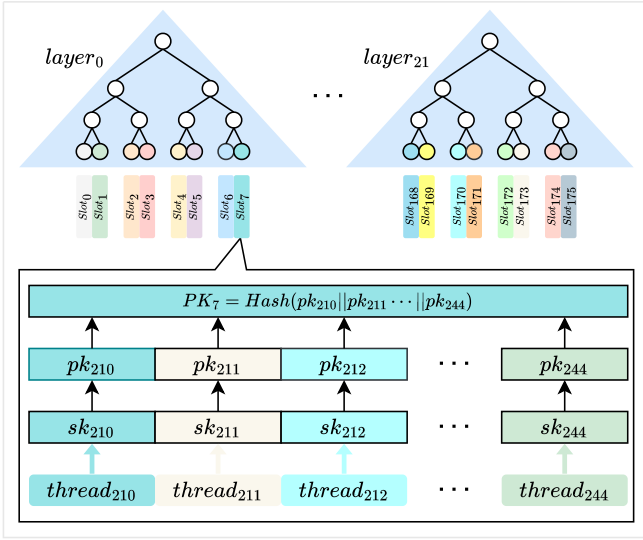
Submodule	CUDA Threads
WOTS ⁺ key pair generation	$d \cdot l \cdot 2^{\frac{h}{d}}$
WOTS ⁺ signature generation	$d \cdot l$
MSS authentication path computation	$d \cdot 2^{\frac{h}{d}}$
FORS key pair generation	$k \cdot t$
FORS authentication path computation	$k \cdot 16$

which will result in a wasted thread in the computation of the hypertree. The advantage of our parallel strategy is that it ensures that latency is within an acceptable range while maximizing throughput. Similarly, for 128s, 192s, and 256s, we also provide the corresponding thread configurations.

C. The SPHINCS⁺ Signing Implementation to Minimize Latency

In contrast to the preceding section, the objective of this section is to minimize the latency. Rather than focusing on preventing thread wastage, the optimisation strategy will seek to maximise parallelism. As previously mentioned, the hypertree is the most time-consuming step of the entire SPHINCS⁺ signature generation process. The computation of the hypertree involves two steps: first, the MSS tree computation, which includes the highly time-consuming generation of Merkle tree leaves (i.e., WOTS⁺ key pair generation) and the less time-consuming computation of authentication path; second, the WOTS⁺ signature generation, which is relatively less time-consuming compared to the MSS tree computation. Therefore, it is crucial to utilize a sufficient number of threads to ensure that the generation of all Merkle tree leaves in the hypertree can be completed within a single round of computation. Table IV shows the thread configuration for each part of low-latency SPHINCS⁺ signing implementation.

1) *Parallel Method for Hypertree:* As previously stated, hypertree is comprised of two distinct components: MSS and WOTS⁺ signing. Consider the structure of MSS tree. Since each selected MSS tree in each layer of the hypertree has a height of $\frac{h}{d}$ and there are d layers, a total of $d \cdot 2^{\frac{h}{d}}$ leaf nodes need to be computed. The message lengths for security levels 1, 3, and 5 are 128 bits, 192 bits, and 256 bits. Thus, the number of divided parts (l) that need to be computed in each leaf are 35, 51, and 67, respectively. We need to simultaneously compute WOTS⁺ key pair for the leaf nodes of the selected MSS tree at each layer in the hypertree. Thus, $d \cdot l \cdot 2^{\frac{h}{d}}$ threads are required to compute the process of WOTS⁺ key pair generation in all layers in parallel, to illustrate, for security level 1, a total of 6160 threads were utilized. Fig. 7 depicts of our method for parallelizing WOTS⁺ key pair generation at security level 1. We also aim to achieve maximum parallelism in the remaining steps. Therefore, for the authentication path computation in the MSS and the generation of WOTS⁺ signatures, we adopted the same parallelization strategy as described in [21].

Fig. 7. The key pair generation of WOTS⁺.

2) *Parallel Method for FORS*: Consider the structure of FORS, containing k subtrees with heights of a . The computation of each subtree is independent of each other, so each subtree can be computed in parallel. The computation process of subtree is similar to that of MSS tree, the difference lies in the way leaf node key pairs are generated. But in the same way, The generation of leaf nodes and authentication path computation is applicable to a range of parallel strategies. FORS contains $k \cdot t$ leaf nodes, for example, at security level 1, there are 2112 leaf nodes. Given that we have sufficient threads, we can generate key pairs for all leaf nodes simultaneously in a single round of computation. In contrast to the generation of leaf nodes, the computation of the authentication path is less time-consuming and fewer threads are required. To enable more efficient intra-block synchronization instead of inter-block synchronization, the computation of the authentication path is performed by a single block. In our implementation, we use $16 \cdot k$ threads to compute the verification paths. That is, every 16 threads are grouped to compute authentication path of a subtree in FORS, and k groups of threads are performed simultaneously. Fig. 8 depicts the process of computing FORS.

D. The SPHINCS⁺ Signature Verification Implementation for Maximizing Throughput

This section begins by analysing the differences between the SPHINCS⁺ signature generation process and the signature verification process. It then demonstrates that the parallel strategy employed for the signature generation process is not applicable to the signature verification. Finally, we analyzed the feasibility of parallelizing SPHINCS⁺ signature verification and introduce our implementation for signature verification with maximum throughput.

1) *Differences From the Signature Process*: In the context of SPHINCS⁺ signature verification, the verifier does not possess the private key. The verifier derives the public key from the message and the signature, then compares it with the public

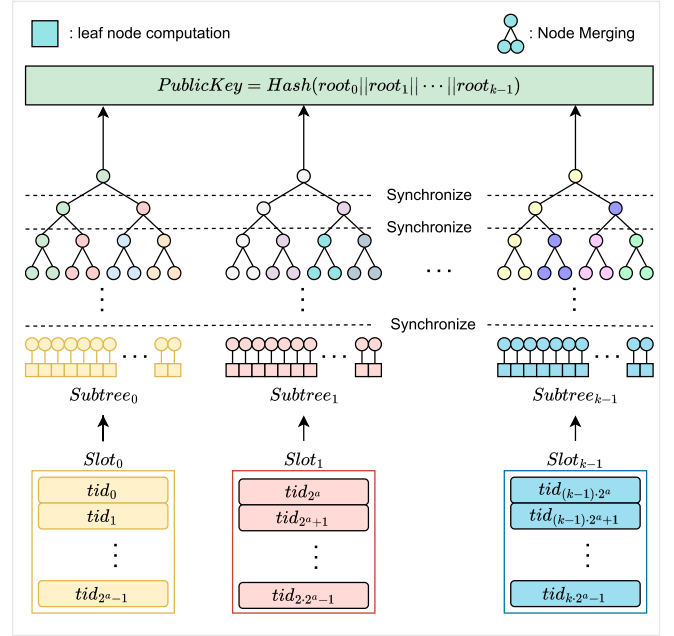


Fig. 8. Low latency parallel strategy for FORS.

key of the signer to verify the authenticity of the signature. During the SPHINCS⁺ signing process, the signer utilizes the SPHINCS⁺ private key seed to generate all leaf nodes for the FORS subtrees and MSS trees and subsequently computes the root node values in a bottom-up manner. In contrast, the verifier does not generate all leaf nodes; rather, it derives the necessary leaf node value from the signature and utilizes the authentication path to compute the root node value. Given that the generation of FORS and MSS tree leaf nodes consumes approximately 90% of the total time for the SPHINCS⁺ signing process, the verification of a SPHINCS⁺ signature is considerably more efficient than its generation.

2) *The Feasibility of Parallelizing SPHINCS⁺ Signature Verification*: Given that Merkle tree constitutes the fundamental structure of FORS and hypertree, We first analyze the signature verification process of Merkle tree. This process comprises two distinct phases. Initially, the public key value of the leaf node is derived based on the signature. Subsequently, the root node value of Merkle tree is determined through the application of the aforementioned public key value in conjunction with the authentication path. Each subtree of FORS and each selected tree of each layer of hypertree are Merkle tree structure. We can analyze the potential for parallelizing signature verification from three different perspectives: intra-leaf parallelism, intra-tree parallelism, and inter-tree parallelism.

- For hypertree, as shown in Fig. 9, public key of the leaf node is derived from l data blocks in the signature through hash function computation. This allows for the utilization of multiple threads to compute the leaf node, with each thread responsible for processing one of the data blocks. Nevertheless, this approach requires the utilization of global memory to temporarily store the computation results of each thread, enabling the derivation of the public

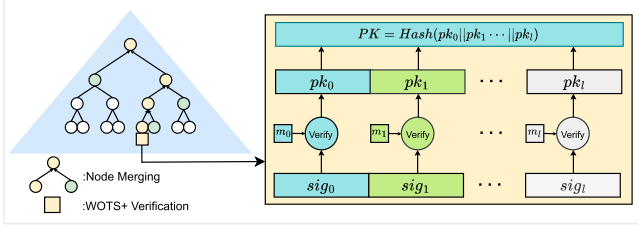


Fig. 9. The verification of MSS in hypertree.

key. This, in turn, increases the memory access time. Additionally, within each MSS tree, the root node is computed by sequentially combining leaf nodes and the data blocks along the authentication path, which is inherently a serial process and does not lend itself to parallel computation. As previously mentioned, SPHINCS⁺ signature generation can be applied in the parallel scheme of the hypertree layer. However, in the SPHINCS⁺ signature verification process, the feasibility of hypertree layer parallel scheme should be considered. In i^{th} the layer of the hypertree, the computation of leaf node public key values depends on the signature and the root node values of the $(i-1)^{th}$ layer MSS tree. The root node values of the $(i-1)^{th}$ layer MSS tree, subsequently, hinge on the public key values of the $(i-1)^{th}$ layer leaf nodes, and this dependency continues recursively. This inherent sequential nature precludes parallel computation across different layers or trees within the hypertree.

- For FORS, the public key value of leaf node is computed by hashing a single data block from the signature, eliminating the necessity for parallel computation using multiple threads. Given that each subtree is a Merkle tree, parallel computation within each subtree is similarly infeasible, as analyzed in the context of hypertree. However, FORS is constructed from k independent subtrees, allowing for the utilization of multiple threads across subtrees, with each thread responsible for the computation of a single subtree.

3) *Parallel Method for Signature Verification:* Similar to the signature generation process, the signature verification is also performed within a single CUDA kernel. To maximize throughput, each signature verification process is handled by a single thread. This approach represents a simple yet highly efficient parallel strategy.

As analysed in the previous section, there are k MSS structures in FORS that can be computed in parallel, and d MSS structures in hypertree that cannot be computed in parallel. Within the MSS of hypertree, the generation of the public key for leaf nodes can be computed in parallel using multiple threads. However, this approach cannot be applied to the computation of the root node, as the computation of the root node is a serial process. To elaborate further, utilizing multiple threads for the computation of MSS leaf nodes will lead to inefficient resource usage during the root node computation. Moreover, synchronization between different threads entails additional performance overhead. Consequently, it can be anticipated that an increase in the number of threads will result in a decrease

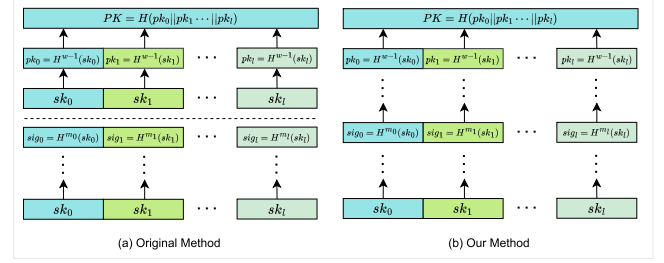


Fig. 10. Comparison of the original method and our method.

in throughput. Therefore, we use only one thread to process a signature verification process to achieve maximum throughput. However, this approach results in higher latency than simultaneous computation by multiple threads. Fortunately, as previously mentioned, the latency of the signature verification process in SPHINCS⁺ is considerably lower than that of the signing process, and we consider the latency of using a single thread for signature verification to be acceptable.

E. Other Optimization Methods

1) *Optimization of the Hypertree Computation Process:* As depicted in Fig. 2, for each layer of the hypertree, the computation begins with the selection of leaf node, followed by the generation of WOTS⁺ signature for the root node of its underlying layer. Subsequently, the public key of the leaf node is computed, thus deriving the root node value of the MSS tree for the current layer. Fig. 4 elucidates that the generation process of WOTS⁺ signature can be considered an intermediate step in the public key generation process. Consequently, if the message to be signed by the current layer (i.e., the root node of the underlying layer) can be determined beforehand, the generation of leaf node public keys and the WOTS⁺ signing process can be combined. This integration obviates the need for separate WOTS⁺ signature generation, thereby enhancing efficiency.

Fig. 10 compares the original computation method with our proposed method. On average, the generation of a WOTS⁺ signature requires $\frac{w-1}{2} \cdot l$ hash operations. By implementing our method, this part of the operations can be eliminated, leading to a reduction in the overall number of hash operations required at each layer of the hypertree. For the leaf nodes executing WOTS⁺ signature generation, this method can reduce the number of hash operations by approximately 33% across different security levels. Our method is effective for both CPU and GPU implementations. On the CPU, hypertree computation follows a sequential process. As each layer is computed, its respective underlying root nodes have already been derived. This enables us to integrate the WOTS⁺ signature generation with the leaf node public key generation, reducing the frequency of WOTS⁺ signature generation from d to 0. For GPU implementation, as shown in Fig. 5, each thread processes consecutive layers in the hypertree. With the exception of the initial layer, every thread can access the root nodes of the underlying layer during its computation. Each thread can employ our method, decreasing

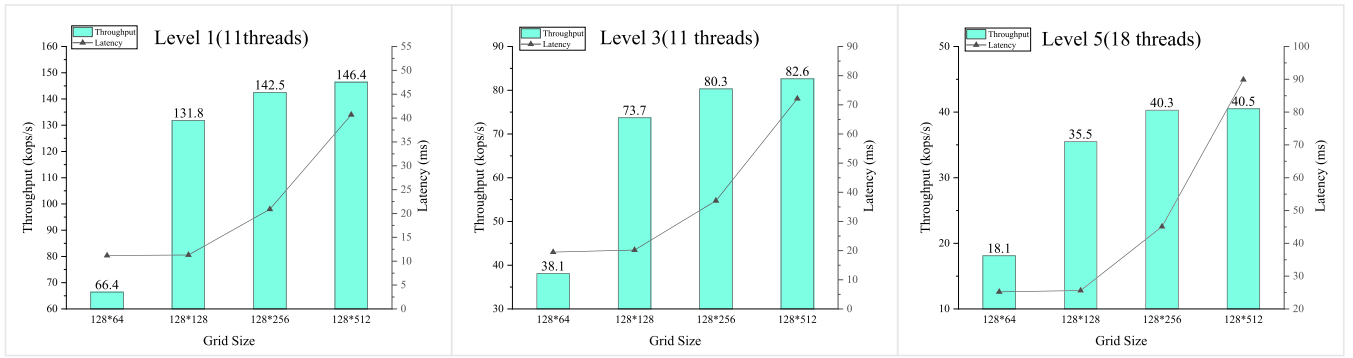


Fig. 11. Peak performance of signature generation across different grid size.

the total number of WOTS⁺ signature generations from d times to t times.

2) *Optimized Memory Access*: Ensuring coalesced memory access is vital for optimizing performance in applications developed on GPU architectures. It involves efficiently using the memory bus to minimize the number of memory accesses, thereby significantly reduce the time spent on memory read and write operations. In GPUs, conventional memory access operations read or write only one byte at a time, leading to significant underutilization of the memory bus. For the three security levels of SPHINCS⁺, the sizes of the data blocks are 16 bytes, 24 bytes, and 32 bytes, respectively. If conventional memory access operations are used, it would require 16, 24, and 32 operations to read or write a single data block, respectively, resulting in substantial time wastage. CUDA provides special instructions to facilitate data access, including INT, INT2 and INT4 instructions. By using the INT, INT2 and INT4 instructions, the computing program can read or write 4, 8, and 16 bytes of data at a time. These particular instructions, which read multiple bytes at a time, are faster than ordinary data access instructions. For security levels 1 and 5, we employ the INT4 instruction for memory access, allowing each data block to be read or written in 1 and 2 operations, respectively. However, for security level 3, due to the 24-byte size of the data blocks, utilizing the INT4 instruction would result in memory misalignment. To address this, we use the INT2 instruction, which requires 3 operations for each data block read or write.

IV. RESULT AND ANALYSIS

A. Implementation Results

In this section, we present a comprehensive performance analysis, comparing our implementation with previous implementations. We begin by detailing the experimental environment and setup used for our performance evaluations. Subsequently, we exhibit the throughput of our high-throughput SPHINCS⁺ implementation, benchmarked against existing implementations: NIST PQC project SPHINCS⁺ reference C-language-based code (version 3.0) [35], SPHINCS implementation on GPU [20], SPHINCS⁺ implementations on GPU [21] and FPGA [26], [27]. Additionally, we analyze the latency of our low-latency SPHINCS⁺ implementation, comparing it

with the CPU reference code and the scheme proposed by Kim et al. [21]. Lastly, we present the throughput of our SPHINCS⁺ signature verification implementation and compare it with the prior works.

B. Our Experimental Environment

A variety of hash algorithms can be employed to generate distinct SPHINCS⁺ instances, the hash function of our implementation is SHA-256 and SHAKE-256. In our performance test, each experiment was performed 1000 times and the average value is presented. The hardware environment of the GPU is an RTX 4090. Given that the RTX 4090 platform contains 128 multiprocessors, we set the size of block to 128 in order to fully utilize the computational resources of GPU. The software environment is Linux operating system and NVCC compiler. The hardware environment of the CPU is a AMD Ryzen5 5600G. The performance of CPU reference code is measured on a single core. In our analysis, two performance metrics are considered: throughput and latency. Throughput is the amount of computed instances within a time unit, e.g., how many signatures can be generated in one second. Hereafter, let operations per second (ops/s) denote this metric. The latency of our CUDA implementation includes only the CUDA kernel execution time, excluding the memory copy time between the CPU and GPU. This is because the copy time is influenced by various factors such as CPU memory and PCI-E, which are unrelated to the parallel optimization strategies of the GPU.

C. SPHINCS⁺ Signature Generation Performance Analysis

Fig. 11 presents the peak performance of our implementation for SPHINCS⁺-SHA-256-f signature generation on the RTX 4090 across varying grid sizes at different security levels. For 128f (resp. 192f and 256f), our implementation generates 146,363 (resp. 82,621 and 40,489) SPHINCS⁺-SHA-256 signatures and 43,102 (resp. 26,404 and 14,065) SPHINCS⁺-SHAKE-256 signatures per second on the RTX 4090. It is evident that as the grid size increases, the latency gradually rises while the throughput correspondingly improves, reaching its peak at a grid size of 128 * 512. Furthermore, we conducted experiments under “s” mode. For 128s (resp. 192s and 256s),

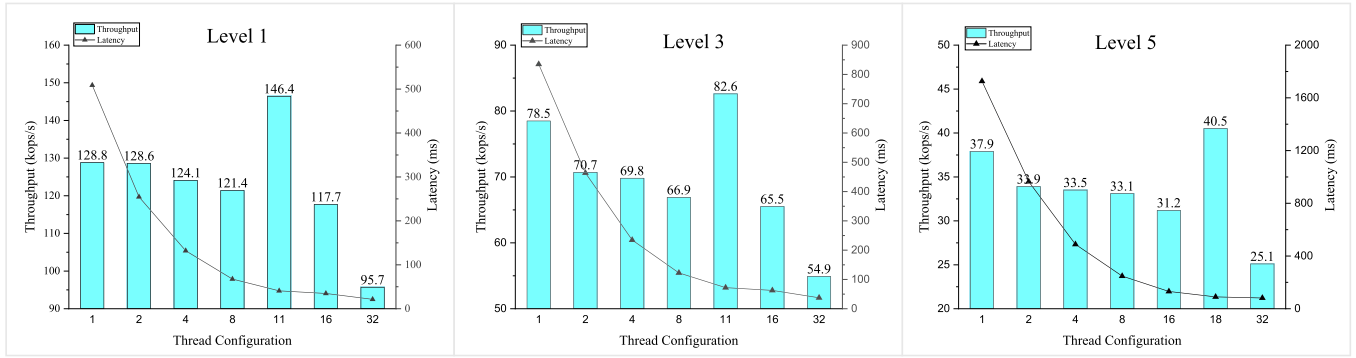


Fig. 12. Performance of signature generation in different thread configuration.

TABLE V
COMPARISON OF SIGNATURE GENERATION THROUGHPUT ON OUR WORK WITH THE RELATED WORK

Version	Hash Algorithm	Platform	Security Level					
			Level 1		Level 3		Level 5	
			Maximum Throughput	Ratio	Maximum Throughput	Ratio	Maximum Throughput	Ratio
[35]	SHA-256	Ryzen5 5600G	40	1	26	1	13	1
[26]	SHA-256	Xilinx XZU3EG	15.54	0.39	-	-	5.02	0.39
[27]	SHAKE-256	Artix-7	990.10	24.75	854.70	32.87	396.82	30.52
[20]	ChaCha	GTX 1080	5152	128.8	-	-	-	-
[21]	SHA-256	RTX 3090	44391*	1109.78	24997*	964.42	11401*	877.00
		RTX 4090	106631	2665.78	46127	1774.12	25578	1967.54
[22]	SHA-256	RTX 3090	49532*	1238.3	24262*	933.15	9427*	725.15
Ours	SHA-256	GTX 1080	19405	485.12	9544	367.08	4481	344.69
		RTX 3090	53961*	1349.10	31257*	1202.19	14824*	1140.31
		RTX 4090	146363	3659.08	82621	3177.73	40489	3114.54

* means that the memory copy time is calculated.

our implementation generates 8,081 (resp. 5,028 and 4,314) SPHINCS⁺-SHA-256 signatures per second on the RTX 4090.

Fig. 12 illustrates the throughput and latency of SPHINCS⁺-SHA-256-f signature generation for various thread configurations on the RTX 4090 at three security levels within a grid size of 128 * 512. It can be observed that the algorithmic processing with 11 (resp. 11 and 18) threads exhibited the highest throughput performance for 128f (resp. 192f and 256f). As previously indicated, these thread configurations result in minimal thread wastage during the signing process. Employing diverse thread configurations for signature generation induces variations in both latency and throughput. An increase in the number of threads correlates with reduced latency. Therefore, appropriate thread configurations can be selected based on the specific requirements for throughput and latency in real-world application scenarios.

Table V presents the peak throughput of various implementations of SPHINCS/SPHINCS⁺-f signature generation across diverse hardware platforms. The reference code running on the CPU is utilized as the baseline, and the Ratio represents the ratio of peak throughput to baseline throughput for each implementation. First, we compared the performance of our SPHINCS⁺-SHA-256 signature generation on the RTX 4090 to the baseline. It was observed that the SPHINCS⁺ signature generation performance improved by a factor of 3659.08×

(resp. 3177.73× and 3114.54×) for 128f (resp. 192f and 256f) compared to the baseline.

Berthet et al. [26] proposed a SPHINCS⁺-SHA-256-f implementation method in Xilinx XZU3EG devices. Compared to their implementation, our approach on the RTX 4090 achieves performance enhancements of 9418.47× and 8065.53× at security levels 1 and 5. Amiet et al. [27] optimized SPHINCS⁺-SHAKE-256-f on the FPGA Artix-7. On the RTX 4090, our implementation achieves performance improvements of 43.53×, 30.89×, and 35.44× across the three different security levels, respectively, compared to [27].

Sun et al. [20] proposed a parallel scheme for SPHINCS on GPU, implementing SPHINCS-256, which offers 128-bit security strength against quantum computers [36], equivalent to the security strength of SPHINCS⁺ at security level 1. Despite structural differences between SPHINCS and SPHINCS⁺, the performance bottleneck in both is rooted in the WOTS⁺-based MSS. Therefore, we also compared our implementation with theirs. On the GTX 1080, our implementation achieved a performance improvement of 3.77× at security level 1 compared to their work [20].

Kim et al. [21] proposed a parallel scheme specifically designed to maximize throughput on various GPU architectures. Their performance metrics diverge from ours because the latency in their CUDA implementation encompasses not only

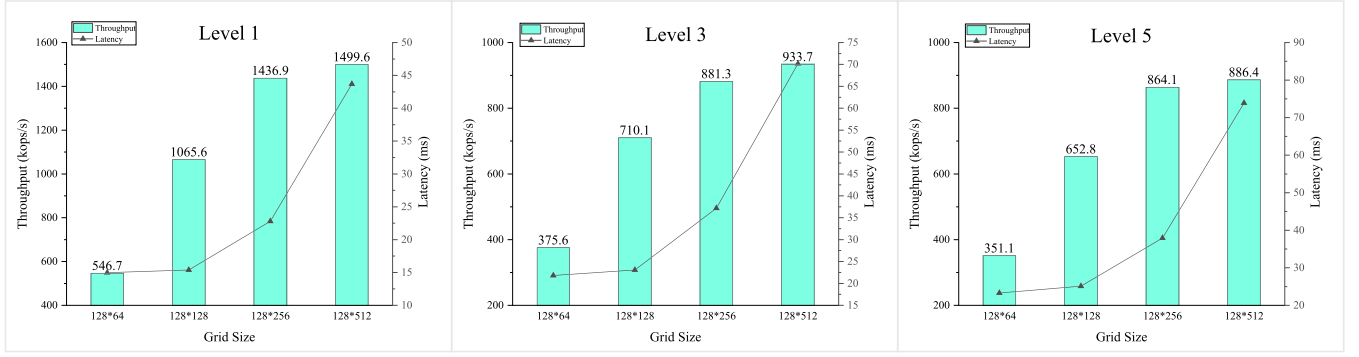


Fig. 13. Peak performance of signature verification across different grid size.

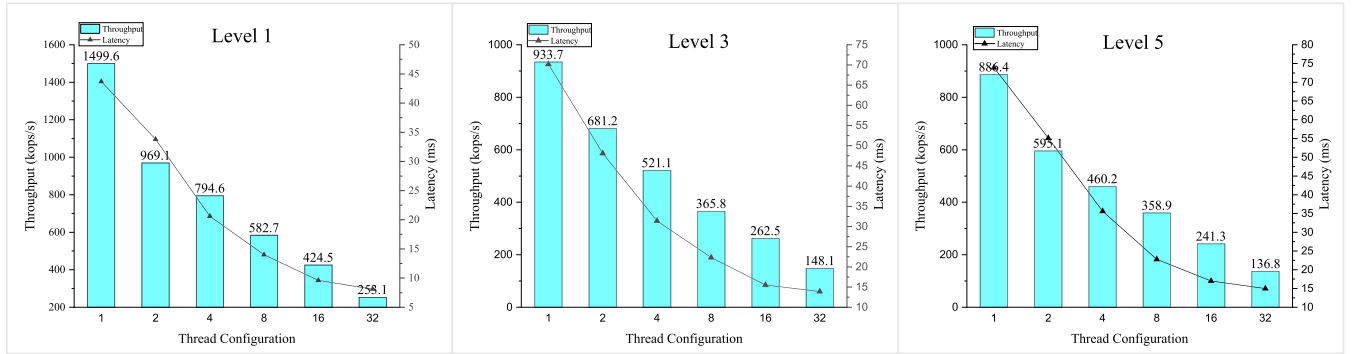


Fig. 14. Performance of signature verification in different thread configuration.

kernel execution time but also includes memory copy time. Although a direct comparison with the data presented in their paper is not feasible, their implementation is open source, allowing us to conduct our own evaluations. On the RTX 4090, excluding memory copy time, their implementation achieves a throughput of 106,631 (resp. 46,127 and 25,578) ops/s for 128f (resp. 192f and 256f), whereas our implementation attains a throughput of 146,363 (resp. 82,621 and 40,489) ops/s, reflecting a $1.37\times$ (resp. $1.79\times$ and $1.58\times$) improvement. When accounting for memory copy time, for 128f (resp. 192f and 256f), the throughput of their implementation is 86,412 (resp. 39,483 and 23,876) ops/s, compared to our throughput of 117,723 (resp. 65,128 and 35,510) ops/s, representing a $1.36\times$ (resp. $1.65\times$ and $1.49\times$) enhancement. Furthermore, on the RTX 3090, to facilitate comparison with the data reported in their paper, we evaluated the performance of our implementation including memory copy time. For 128f (resp. 192f and 256f), their implementation yields a throughput of 44,391 (resp. 24,997 and 11,401) ops/s, while our implementation achieves 53,961 (resp. 31,257 and 14,824) ops/s, marking an improvement of $1.22\times$ (resp. $1.25\times$ and $1.30\times$). Wang et al. [22] also conducted an optimized implementation of SPHINCS⁺-SHA-256 on the GPU. Regarding “f” mode, At security level 1 (resp. 3 and 5), their implementation yields a throughput of 49,532 (resp. 24,262 and 9,427) ops/s on the RTX 3090, our approach achieves performance improvements of $1.09\times$ (resp. $1.29\times$, and $1.57\times$) over their implementation. Table VI presents the occupancy analysis of SPHINCS⁺-SHA-256-128f obtained using Nsight Compute.

TABLE VI
OCCUPANCY COMPARISON OF SPHINCS⁺-SHA-256-128F-SIMPLE

Version	Kernel	Theoretical	Achieved
[21]	FORS	33.33%	4.10%
	Hypertree MSS	25.00%	12.42%
	WOTS ⁺ sign	52.10%	46.70%
[22]	SPHINCS ⁺ sign	33.33%	15.91%
Ours	SPHINCS ⁺ sign	33.33%	32.80%

Compared to [21] and [23], our implementation achieves occupancy that is closest to the theoretical peak.

The low-latency implementation, at security level 1, with a grid size of 128×512 , has a throughput of 19,521 ops/s and a latency of 0.512 ms. The signature latency of the CPU reference code for SPHINCS⁺-SHA-256-f is 25 ms, and the throughput is 40 ops/s. The performance of the system was enhanced by $48.8\times$ and $488.03\times$ in terms of latency and throughput, respectively. Compared to Kim et al. [21] on the RTX 4090. Their SPHINCS⁺SHA-256-f signature generation has a throughput of 106,631 ops/s and a latency of 4.8 ms at security level 1. Our implementation demonstrates a $5.5\times$ decrease in throughput and a $9.4\times$ reduction in latency.

D. SPHINCS⁺ Verification Performance Analysis

Fig. 13 illustrates the peak performance of our SPHINCS⁺-SHA-256-f signature verification implementation on the RTX 4090, evaluated across various grid sizes at distinct security

TABLE VII
COMPARISON OF SIGNATURE VERIFICATION THROUGHPUT
ON OUR WORK WITH THE RELATED WORK

Security level	Version	Hash Algorithm	Platform	Maximum Throughput	Ratio
Level 1	[35]	SHA-256	Ryzen5 5600G	739	1
	[20]	ChaCha	GTX 1080	106390	143.96
	[21]	SHA-256	RTX 4090	980092	1326.24
	Ours	SHA-256	GTX 1080	127034	171.90
			RTX 4090	1499590	2029.22
Level 3	[35]	SHA-256	Ryzen5 5600G	492	1
	[21]	SHA-256	RTX 4090	518044	1052.93
	Ours	SHA-256	GTX 1080	71298	144.91
			RTX 4090	933696	1897.76
	[35]	SHA-256	Ryzen5 5600G	487	1
Level 5	[21]	SHA-256	RTX 4090	257286	528.31
	Ours	SHA-256	GTX 1080	70547	144.86
			RTX 4090	886362	1820.05

levels. Specifically, at security level 1 (resp. levels 3 and 5), our implementation can verify 1499,590 (resp. 1408,846 and 886,361) SPHINCS⁺-SHA-256 signatures and 643,195 (resp. 501,327 and 427,003) SPHINCS⁺-SHAKE-256 signatures per second on the RTX 4090. The data demonstrates a clear trend: as the grid size expands, the latency exhibits a gradual increase, while the throughput concurrently enhances, ultimately reaching its peak at a grid size of 128 * 512. We also conducted experiments under “s” mode. For 128s (resp. 192s and 256s), our implementation can verify 3743,451 (resp. 2240,868 and 1561,818) SPHINCS⁺-SHA-256 signatures per second on the RTX 4090.

Fig. 14 presents the throughput and latency for SPHINCS⁺-SHA-256-f signature verification under various thread configurations across three security levels on the RTX 4090. The data reveal that single-threaded execution consistently achieves the highest throughput across all security levels. Conversely, increasing the number of threads results in decreased latency and reduced throughput. As previously demonstrated, single-threaded verification optimizes efficiency by minimizing thread wastage. One intriguing phenomenon is that the peak throughput of signature verification at Level 3 and Level 5 does not significantly differ. Upon analysis, during the FORS signature verification stage, each subtree requires running the hash function once to compute the public key of leaf from the signature and a times to compute the root node. Thus, FORS requires running the hash function $(a + 1) \cdot k$ times in total. During the hypertree verification stage, for each layer, calculating the MSS tree leaf’s public key from the signature requires running the hash function an average of $(\frac{w-1}{2} \cdot l + 1)$ times, and computing the root node requires $\frac{h}{d}$ hash operations. Therefore, the hypertree requires a total of $((\frac{w-1}{2} \cdot l + 1) \cdot d + h)$ hash operations. According to the parameters provided in Table II, verifying at Level 3 requires an average of 8800 hash operations, while verification at Level 5 necessitates an average of 8977.5 hash operations, which are very close.

As illustrated in Table VII, our SPHINCS⁺-SHA-256-f verification performance results on the RTX 4090 environment demonstrated performance improvements of 2029.22 $\times$ (resp. 1897.76 $\times$ and 1820.05 $\times$) at security level 1 (resp. 3 and 5), compared to the C reference code on CPU. On the GTX 1080, our implementation achieved a performance improvement of 1.19 $\times$ at security level 1 compared to the work of Sun et al. [20].

Compared to Kim et al.’s work [21], our SPHINCS⁺-SHA-256-f verification performance results on the RTX 4090 environment demonstrated performance improvements of 1.53 $\times$ (resp. 1.80 $\times$ and 3.45 $\times$) at security level 1 (resp. 3 and 5).

V. CONCLUSION

This paper presents an optimized implementation of SPHINCS⁺ using GPU acceleration, primarily aimed at enhancing the throughput of the signing and verification processes. We have thoroughly investigated the SPHINCS⁺ signing and verification procedures, analyzing the feasibility of parallelization from multiple perspectives. A set of parallel strategies, adjustable according to practical requirements, has been proposed and further optimized by leveraging the capabilities of GPU and CUDA. Extensive experiments and comprehensive performance analyses demonstrate that, compared to previous works, including the SPHINCS⁺ CPU reference implementation, GPU implementation, and FPGA implementation, our proposed parallel strategies significantly enhance the performance of SPHINCS⁺ signing and verification. Consequently, our research meets the demand for improved SPHINCS⁺ performance in various application scenarios, benefiting service providers and security protocols. Furthermore, our approach is generalizable to other hash-based signature algorithms. Specifically, the low-latency parallelism is well-suited for algorithms such as XMSS and LMS, which rely on a single Merkle tree structure. In contrast, the high-throughput parallel strategy can be effectively applied to multi-tree constructions like XMSS^{MT}.

REFERENCES

- [1] P. W. Shor, “Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer,” *SIAM Rev.*, vol. 41, no. 2, pp. 303–332, 1999.
- [2] L. K. Grover, “Quantum computers can search rapidly by using almost any transformation,” *Phys. Rev. Lett.*, vol. 80, no. 19, 1998, Art. no. 4329.
- [3] “IBM quantum computing,” Armonk, NY, USA: IBM Company®, 2023. [Online]. Available: <https://www.ibm.com/quantum/>
- [4] “Post-quantum cryptography standardization,” Gaithersburg, MD, USA: NIST, 2023. [Online]. Available: <https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization>
- [5] J. Bos et al., “CRYSTALS-Kyber: A CCA-secure module-lattice-based KEM,” in *Proc. IEEE Eur. Symp. Secur. Privacy (EuroS&P)*, Piscataway, NJ, USA: IEEE Press, 2018, pp. 353–367.
- [6] L. Ducas et al., “Crystals-dilithium: A lattice-based digital signature scheme,” *IACR Trans. Cryptographic Hardware Embedded Syst.*, 2018, pp. 238–268.
- [7] P.-A. Fouque et al., “Falcon: Fast-Fourier lattice-based compact signatures over NTRU,” *Submission to NIST’s Post-Quantum Cryptography Standardization Process.*, vol. 36, no. 5, 2018, pp. 1–75.
- [8] D. J. Bernstein, A. Hülsing, S. Kölbl, R. Niederhagen, J. Rijneveld, and P. Schwabe, “The SPHINCS⁺ signature framework,” in *Proc. ACM SIGSAC Conf. Computer Commun. Security*, 2019, pp. 2129–2146.
- [9] N. Aragon et al., “BIKE: Bit flipping key encapsulation,” 2022. [Online]. Available: https://bikesuite.org/files/v5.2/BIKE_Spec.2024.10.10.1.pdf
- [10] M. R. Albrecht et al., “Classic McEliece: Conservative code-based cryptography,” 2022. [Online]. Available: <https://classic.mceliece.org/mceliece-spec-20221023.pdf>
- [11] C. A. Melchor et al., “Hamming quasi-cyclic (HQC),” *NIST PQC Round*, vol. 2, no. 4, 2018, Art. no. 13.
- [12] G. Alagic et al., “Status report on the third round of the NIST post-quantum cryptography standardization process,” 2022. [Online]. Available: <https://doi.org/10.6028/NIST.IR.8413-upd1>

- [13] "CUDA C programming guide 9.0," Santa Clara, CA, USA: NVIDIA, 2017. [Online]. Available: <https://docs.nvidia.com/cuda/cuda-c-programming-guide/>
- [14] 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, 2022.
- [15] 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)*, Piscataway, NJ, USA: IEEE Press, 2017, pp. 1–6.
- [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. Cryptographic Hardware Embedded Syst.*, vol. 2021, no. 4, pp. 114–148, Aug. 2021, doi: 10.46586/tches.v2021.i4.114-148.
- [17] S. Shen, H. Yang, Y. Liu, Z. Liu, and Y. Zhao, "Cuda-accelerated RNS multiplication in word-wise homomorphic encryption schemes," *Cryptology ePrint Archive*, Paper 2022/633, 2022. [Online]. Available: <https://eprint.iacr.org/2022/633>
- [18] F. Zheng, W. Pan, J. Lin, J. Jing, and Y. Zhao, "Exploiting the floating-point computing power of GPUS for RSA," in *Proc. 17th Int. Conf. Inf. Secur. (ISC)*, Cham, Switzerland: Springer, 2014, pp. 198–215.
- [19] J. Dong, F. Zheng, J. Lin, Z. Liu, F. Xiao, and G. Fan, "EC-ECC: Accelerating elliptic curve cryptography for edge computing on embedded GPU tx2," *ACM Trans. Embedded Comput. Syst. (TECS)*, vol. 21, no. 2, pp. 1–25, 2022.
- [20] 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.
- [21] D. Kim, H. Choi, and S. C. Seo, "Parallel implementation of SPHINCS+ with GPUs," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 71, no. 6, pp. 2810–2823, Jun. 2024.
- [22] Z. Wang, X. Dong, H. Chen, Y. Kang, and Q. Wang, "CUSPX: Efficient GPU implementations of post-quantum signature SPHINCS+," *IEEE Trans. Comput.*, vol. 74, no. 1, pp. 15–28, Jan. 2025.
- [23] Z. Wang, X. Dong, H. Chen, and Y. Kang, "Efficient GPU implementations of post-quantum signature XMSS," *IEEE Trans. Parallel Distrib. Syst.*, vol. 34, no. 3, pp. 938–954, Mar. 2023.
- [24] Z. Wang, X. Dong, Y. Kang, H. Chen, and Q. Wang, "An example of parallel Merkle tree traversal: Post-quantum Leighton-Micali signature on the GPU," *ACM Trans. Archit. Code Optim.*, vol. 21, no. 3, pp. 1–25, 2024.
- [25] Y. Kang, X. Dong, Z. Wang, H. Chen, and Q. Wang, "Parallel implementations of post-quantum Leighton-Micali signature on multiple nodes," *J. Supercomput.*, vol. 80, no. 4, pp. 5042–5072, 2024.
- [26] 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)*, Piscataway, NJ, USA: IEEE Press, 2021, pp. 180–187.
- [27] D. Amiet, L. Leuenberger, A. Curiger, and P. Zbinden, "FPGA-based SPHINCS+ implementations: Mind the glitch," in *Proc. 23rd Euromicro Conf. Digit. Syst. Des. (DSD)*, Piscataway, NJ, USA: IEEE Press, 2020, pp. 229–237.
- [28] X. Zhao, B. Wang, Z. Zhao, Q. Qu, and L. Wang, "Highly efficient parallel design of dilithium on GPUs," Research Square, Preprint, Dec. 2022. [Online]. Available: <https://doi.org/10.21203/rs.3.rs-2337597v1>
- [29] S. Shen, H. Yang, W. Dai, H. Zhang, Z. Liu, and Y. Zhao, "High-throughput GPU implementation of dilithium post-quantum digital signature," *IEEE Trans. Parallel Distrib. Syst.*, vol. 35, no. 11, pp. 1964–1976, Nov. 2024.
- [30] J. Buchmann, E. Dahmen, S. Ereth, A. Hülsing, and M. Rückert, "On the security of the Winternitz one-time signature scheme," *Int. J. Appl. Cryptography*, vol. 3, no. 1, pp. 84–96, 2013.
- [31] A. Hülsing, "W-OTS+—shorter signatures for hash-based signature schemes," in *Proc. 6th Int. Conf. Cryptol. Prog. Cryptol. (AFRICACRYPT) Africa*, Berlin, Germany: Springer, 2013, pp. 173–188.
- [32] R. C. Merkle, "A certified digital signature," in *Conf. Theory Appl. Cryptol.*, New York, NY, USA: Springer, 1989, pp. 218–238.
- [33] J. Buchmann, E. Dahmen, and A. Hülsing, "XMSS-A practical forward secure signature scheme based on minimal security assumptions," in *Proc. 4th Int. Workshop Post-Quantum Cryptogr. (PQCrypto)*, Berlin, Germany: Springer, 2011, pp. 117–129.
- [34] J. Pieprzyk, H. Wang, and C. Xing, "Multiple-time signature schemes against adaptive chosen message attacks," in *Proc. 10th Annu. Int. Workshop Sel. Areas Cryptogr. (SAC)*, Berlin, Germany: Springer, 2004, pp. 88–100.
- [35] J.-P. Aumasson et al., "SPHINCS+ – Submission to the 3rd round of the NIST post-quantum project. v3.1," Tech. Rep., 2022. [Online]. Available: <https://sphincs.org/data/sphincs+-r3.1-specification.pdf>
- [36] D. J. Bernstein et al., "SPHINCS: Practical stateless hash-based signatures," in *Proc. Annu. Int. Conf. Theory Appl. Cryptogr. Techn.*, Berlin, Germany: Springer, 2015, pp. 368–397.



Yijing Ning received the B.E. degree in 2023 from the University of Science and Technology of China, where he is currently working toward the M.S. degree with the School of Cyber Science and Technology. His research interests include cryptographic engineering, postquantum cryptography, and high-performance computing.



Jiankuo Dong received the B.E. degree from Xi'an Jiaotong University, and the Ph.D. degree from the University of Chinese Academy of Sciences, in 2014 and 2019, respectively. He is currently a Research Professor with the University of Science and Technology of China, and also with the School of Computer Science, Nanjing University of Posts and Telecommunications. His research interests include cryptographic engineering, public key cryptography, and applied cryptography.



Jingqiang Lin (Senior Member, IEEE) received the M.S. and Ph.D. degrees from the University of Chinese Academy of Sciences, in 2004 and 2009, respectively. He is with the School of Cyber Security, University of Science and Technology of China. He is a Full Professor with the School of Cyber Science and Technology, University of Science and Technology of China. His research interests include applied cryptography and system security.



Fangyu Zheng received the B.E. degree from the University of Science and Technology of China, and the Ph.D. degree from the University of Chinese Academy of Sciences, in 2011 and 2016, respectively. He is currently an Associate Professor with the School of Cryptology, University of Chinese Academy of Sciences. His research interests include applied cryptography and high performance computing.



Yu Fu received the M.S. degree from the University of Chinese Academy of Sciences, in 2022, and is currently working toward the Ph.D. degree with the School of Cyber Science and Technology, University of Science and Technology of China. His research interests include applied cryptography and privacy-preserving machine learning.



Fu Xiao (Senior Member, IEEE) received the Ph.D. degree in computer science and technology from Nanjing University of Science and Technology, Nanjing, China, in 2007. He is currently a Professor and the Ph.D. Supervisor with the School of Computer Science and Technology, Nanjing University of Posts and Telecommunications. His research interest includes wireless sensor networks. He is a member of the IEEE Computer Society and the Association for Computing Machinery.