CUSPX: Efficient GPU Implementations of Post-Quantum Signature SPHINCS$^+$

Ziheng Wang, Xiaoshe Dong, Heng Chen, Yan Kang, and Qiang Wang

***Abstract*—Quantum computers pose a serious threat to existing cryptographic systems. While Post-Quantum Cryptography (PQC) offers resilience against quantum attacks, its performance limitations often hinder widespread adoption. Among the three National Institute of Standards and Technology (NIST)-selected general-purpose PQC schemes, SPHINCS$^+$ is particularly susceptible to these limitations. We introduce CUSPX (CUDA SPHINCS$^+$), the first large-scale parallel implementation of SPHINCS$^+$ capable of running across 10,000 cores. CUSPX leverages a novel three-level parallelism framework, applying it to *algorithmic parallelism*, *data parallelism*, and *hybrid parallelism*. Notably, CUSPX introduces parallel Merkle tree construction algorithms for arbitrary parallel scales and several load-balancing solutions, further enhancing performance. By treating tasks parallelism as the top level of parallelism, CUSPX provides a four-level parallel scheme that can run with any number of tasks. Evaluated on a single GeForce RTX 3090 using the SPHINCS$^+$-SHA-256-128s-simple parameter set, CUSPX achieves a single task's signature generation latency of 0.67 ms, demonstrating a 5,105× speedup over a single-thread version and an 18.50× speedup over the previous fastest implementation.**

***Index Terms*—Post-quantum cryptography, stateless hash-based signatures, SPHINCS$^+$, parallel computing, CUDA, GPU**

I. Introduction

DIGITAL signatures are widely used in website certificates and email certificates to provide identification. However, the looming advent of large-scale quantum computers, capable of executing powerful quantum algorithms like Shor's and Grover's algorithm, casts a shadow over the security of the current public-key cryptographic framework.

Despite this looming quantum threat, cryptography remains a viable tool. Post-Quantum Cryptography (PQC) emerges as a resilient defence against quantum attacks. Among PQC, the Hash-Based Signature (HBS) is the most conservative scheme, relying for its security solely on the preimage resistance of its component cryptographic hash function [1]. Foreseeably, no known quantum algorithm poses a real threat to HBS [2].

PQC, or quantum-resistant or quantum-safe cryptography, encompasses two categories: Digital Signature Algorithms (DSAs) and Key Establishment Mechanisms (KEMs). HBSs are a type of DSA, invented in 1979. Due to the excessive computational complexity, it was not practical until the appearance of the eXtended Merkle Signature Scheme (XMSS) in 2011 which changed the situation. Leighton-Micali Signatures (LMS) later modified the mask method to further enhance computational performance. Nevertheless, both XMSS and LMS are stateful, their use is still limited, e.g., they require the synchronisation of states between devices [4]. In 2015, SPHINCS, a stateless HBS, was designed, which can be used anywhere a DSA is required. The difference between stateful HBS and stateless HBS lies in whether the state of the previous signature needs to be saved, which has different requirements for practical scenarios. To further enhance security and performance, SPHINCS further evolved into SPHINCS$^+$ [5] in 2019. The main distinguishing contributions of SPHINCS$^+$ are the new few-time signature scheme, Forest Of Random Subsets (FORS), and the way leaf nodes are chosen. Recently, the Stateless Hash-Based Digital Signature Algorithm (SLA-DSA) [6] has made small improvements to SPHINCS$^+$ in 2023, serving as a prototype for the PQC standard.

Nowadays, the National Institute of Standards and Technology (NIST) has selected three DSAs for standardization [7]: CRYSTALS-Dilithium, Falcon, and SPHINCS$^+$. Also, a KEM is selected for standardisation: CRYSTALS-KYBER. According to the NIST's recent presentation [8], the standardisation of these algorithms is expected to be finalised in 2024. Compared to the other three lattice-based algorithms, SPHINCS$^+$ [9] can offer more conservative security but with poorer performance. Therefore, it is necessary to optimise the performance of SPHINCS$^+$.

A. Motivation

The foundational operator in SPHINCS$^+$ is the hash function. Upon this foundation, Winternitz One-Time Signatures Plus (WOTS$^+$) constructs a series of independent Winternitz chains. A Winternitz chain of length w is constructed by running the hash function w times, with the output of each hash iteration serving as the input for the next iteration. The higher-level structure is the variant of the Merkle tree, including trees of XMSS and the Forest Of Random Subsets (FORS). In these trees, the value of a node is obtained by hashing the values of its left and right child nodes. Multiple XMSS trees, arranged in layers, constitute the SPHINCS$^+$ hypertree. SPHINCS$^+$ consists of a hypertree and FORS trees.

Several parallel strategies were considered for SPHINCS$^+$: vectorizing the underlying hash operations using AVX2 [5], AVX-512 [10] or NEON [11], reducing the number of instruction cycles by using hardware-specific instructions such as

Manuscript received ?? ??; revised ?? ??. accepted ?? ?? ??. This research has been supported by the China National Key R&D Program (Grant No. 2018YFB1700405). (Corresponding author: Heng Chen.)

The authors are with the School of Computer Science and Technology, Xi'an Jiaotong University, Xi'an 710049, China, and Q Wang is also with the Network Information Center, Xi'an Jiaotong University, Xi'an 710049, China (e-mail: {wzh009888, 3121351060}@stu.xjtu.edu.cn, {xsdong, hengchen, wangqiang1989}@xjtu.edu.cn)

Digital Object Identifier ??

TABLE I
DESIGN TARGETS AND THREAD DISTRIBUTIONS FOR PARALLELISM

	Latency	Throughput	Tasks	Threads per Task
AP	lowest	no need	1	max
DP	no need	highest	max	1
HP	reasonable low	reasonable high	t	max / t

SHA-NI [12] and accelerators such as FPGA [13], [14]. While these efforts have successfully parallelised the hash function and WOTS$^+$, they have yet to design efficient strategies for the large-scale structures of XMSS and FORS trees.

The use of SPHINCS$^+$ is still constrained by its performance notwithstanding optimisation efforts. The reported maximum throughput of SPHINCS$^+$-128s-simple is 80.65 signatures per second on an FPGA implementation [13], corresponding to 12.40 ms per signature and 14,285.71 verifications per second, or 0.07 ms per verification. Despite high throughputs, many applications still require more such as social networking software (e.g., TikTok) or a large e-commerce website. On Singles Day in 2017, the Alibaba company was processing 325,000 orders a second at one point [15]. A more efficient SPHINCS$^+$ parallel implementation is therefore urgently needed.

We now define three types of parallelism: (1) *Algorithmic Parallelism (AP)* uses multiple threads to speed up a single $task$ in parallel. (2) *Data Parallelism (DP)* refers to multiple threads performing multiple $tasks$ in parallel, one $task$ per thread. (3) *Hybrid Parallelism (HP)* is a hybrid of algorithmic parallelism and data parallelism. Threads are divided into multiple groups; each assigned a different $task$. Table I shows design targets and thread distributions for three types of parallelism, where max is the Maximum parallelism.

B. Challenges and Contributions

GPU-based SPHINCS$^+$ deployment is mainly used in two scenarios: web servers and cloud. Web servers are the most common use case, where SPHINCS$^+$ is used as a digital signature for web authentication. Digital signatures are composed of three operations: keypair generation, signature generation, and signature verification. In this case, keypair and signature generation only require low latency, while signature verification requires high throughput and low latency. Because the performance of signature verification directly affects the user experience. In addition, this high-performance scheme is also well-suited for cloud service [16]. The cloud can provide cloud services for the above three operations, so there are performance requirements for these three operations. Support for high throughput is a key design focus for cloud services.

Different scenarios have different performance requirements. In real-time scenarios, such as vehicular network [17], there are high demands for latency, making algorithmic parallelism a more appropriate scheme. In batch processing scenarios [18], for instance, when signing a large amount of data in the background, there are high demands for throughput, making data parallelism a suitable scheme. In stream processing scenarios or web server [19], there often exists a maximum latency requirement. Hybrid parallelism can provide high throughput based on this latency, to alleviate server pressure. Therefore, a variety of parallelism is necessary.

TABLE II
PARALLEL SCHEMES OF LARGE-SCALE HBS IMPLEMENTATIONS

Work	Inter-Tree	Inter-Node	Intra-Node	AP	DP	DP
LMS [20]	x	✓	✓	✓	✓	x
LMS [21]	x	✓	✓	✓	✓	x
XMSS [22]	x	✓	✓	✓	✓	x
SPHINCS [23]	✓	✓	✓	✓	✓	x
SPHINCS$^+$ [24]	✓	✓	✓	x	x	✓
CUSPX	✓	✓	✓	✓	✓	✓

Our design faces three challenges. (1) Large-scale parallel designs: Efficiently parallelizing SPHINCS$^+$ on GPUs, with their many cores and poor single-core performance, is a challenging task. (2) Hybrid parallel designs: Hybrid parallelism requires a trade-off between latency and throughput, which further complicates the design of parallel schemes. (3) GPU efficient implementation: It is a challenge to make SPHINCS$^+$ fully utilise the hardware resources on the GPU.

Table II shows parallel schemes of large-scale HBS implementations, which are based on LMS [20], [21], XMSS [22], SPHINCS [23] and SPHINCS$^+$ [24] algorithms. The vertical axis represents the parallel schemes involved, denoted as parallelism of inter-tree, inter-node and intra-node, which refer to the parallel construction of multiple Merkle trees, the parallel generation of multiple nodes of a single Merkle tree and the parallel generation of a single node, respectively. In general, our main contributions are:

C1. We propose CUSPX (CUDA SPHINCS$^+$), the first large-scale parallel implementation for standardised general-purpose HBS. Three parallel schemes are designed for various application scenarios: algorithmic parallelism, data parallelism and hybrid parallelism. Our code will be open-sourced after the article is accepted.

C2. We present a three-level parallel framework: multi-tree, inter-node and intra-node levels, which can be used individually or in combination. Merkle tree parallel construction algorithms for arbitrary parallel scales are proposed.

C3. We design a general-purpose hybrid parallel scheme for HBS. CUSPX understands task parallelism as a higher level existence than the three-level parallel framework, that is, hybrid parallelism of CUSPX is four-level parallel framework. Moreover, we design four kinds of load-balancing schemes for WOTS$^+$ and two for FORS trees.

II. PRELIMINARIES

A. Digital Signature Mechanism

A digital signature mechanism is a tuple of cryptographic algorithms (KG, Sign, Verify), where Sign has two forms:

1. $(pk, sk) \leftarrow \mathsf{KG}(seed)$: A probabilistic ***key generation*** algorithm that generates pk and sk using $seed$.

2. $sig \leftarrow \mathsf{Sign}(sk, msg)$: A ***signature generation*** algorithm that signs msg with sk to output deterministic sig.

3. $sig \leftarrow \mathsf{Sign}(sk, msg, seed)$: A probabilistic ***signature generation*** algorithm that signs msg to output sig with sk and $seed$. SPHINCS$^+$ uses this form.

TABLE III
TERMS AND THEIR MEANINGS OF THE SPHINCS$^+$ ALGORITHM

Terms	Meanings
pk, sk, sig	Public key, private key and signature of SPHINCS$^+$
H, n	A hash operation and its output size in bytes
$wotsPk$, $wotsSk$	Secret key and public key of WOTS$^+$
$wotsSig$	Signature of WOTS$^+$
$forsPk$, $forsSk$	Secret key and public key of FORS$^+$
$forsSig$	Signature of FORS$^+$
h, d	Height and number of layers of the hypertree
h'=h/d	Height of a single-layer tree
w	Length of a Winternitz chain
len	Number of n-byte strings in OTS
msg, m	Message digest and its length in bytes for DSA
$hmsg$, md	Message and its digest for hash functions
a, k	Number and height of trees in FORS
$addr$	A 32-byte data structure
$OptRand$	A n-byte string used to counter side-channel attacks

4. $b \leftarrow$ Verify(pk, sig, msg): A ***signature verification*** algorithm that uses pk and msg to verify sig and output deterministic result b that can be *true* or *false*.

B. Review of SPHINCS$^+$

SPHINCS$^+$ is an only HBS that provides the most conservative scheme to resist the quantum threat. Here, we briefly introduce SPHINCS$^+$. The complete specification is available in the latest NIST submission [9].

Table III shows the terms and their meanings of the SPHINCS$^+$ algorithm. Let us illustrate the meaning of the parameter set with the example SPHINCS$^+$-SHA-256-192f-simple. "SPHINCS$^+$" is the prefix of all parameter sets. "SHA-256" indicates the hash function used. The number "128" represents the number of bits of parameter n. The letter after the number can be "f" or "s", indicating the fast version or the small signature version, respectively. The term "robust" means bitmasks are used when performing hash functions. When bitmasks are not used, "robust" replaces "simple". If some parts are ignored, all corresponding parameter sets are represented. For example, SPHINCS$^+$-SHA-256-128-simple represent its corresponding "f" and "s" versions.

FIPS 205 Compatibility. According to the new standard FIPS 205 (draft) [6], the robust parameter set is no longer permitted, i.e., only the simple parameter set is supported. Moreover, only SHA-256 and SHAKE-256 hash functions can be used. The parameter set in CUSPX, SPHINCS$^+$-SHA-256-192f-simple, corresponds to the SLH-DSA-SHA2-192f parameter set of FIPS 205. The parameter sets utilised in CUSPX are restricted to those that coincide with FIPS 205. Because the revisions of FIPS 205 to the latest NIST submission are only in terms of security, our approach is still applicable. Because FIPS 205 has no official code, we still base our description on the latest NIST submission.

Table IV shows the security levels and the sizes of the public key, secret key and signature for all parameter sets, where hf can be SHA-256 or SHAKE256. There is a total of 12 parameter sets. The requirements for levels 1–5 are: Any attack that breaks the relevant security definition must require computational resources comparable to or greater than those required for key/collision search on AES-128, SHA-256, AES-192, SHA-384 and AES-256 [25].

TABLE IV
SECURITY LEVELS (SL) AND SIZE (BYTES) FOR ALL PARMETER SETS

Parameter Sets	SL	pk bytes	sk bytes	sig bytes
SPHINCS$^+$-hf-128s-simple	1	32	64	7856
SPHINCS$^+$-hf-128f-simple	1	32	64	17088
SPHINCS$^+$-hf-192s-simple	3	48	96	16224
SPHINCS$^+$-hf-192f-simple	3	48	96	35664
SPHINCS$^+$-hf-256s-simple	5	64	128	29792
SPHINCS$^+$-hf-256f-simple	5	64	128	49856

The secret key, sk, consists of four consecutive n-byte strings: $sk.seed$, $sk.prf$, $pk.seed$ and $pk.root$. The first three strings are seeds in the SPHINCS$^+$ and $pk.root$ is the value of the root node of the top XMSS tree. The public key, pk, only includes $pk.seed$ and $pk.root$. In contrast to negligible sizes of sk and pk, the signature size is large. The signature size is affected by two aspects: (1) The output bit number of the hash function, which also affects the algorithm security. (2) Small signature (s) or fast execution (f), which also affects execution efficiency.

1) Hash Functions: The hash function is used to hash the message $hmsg$ of any length into a message digest md of a fixed value. The md is different from the message digest of a digital signature msg, which is the input of the signature generation. All operations of SPHINCS$^+$ are all based on six encapsulated hash functions, expressed as:

- T_l: $pk.seed \times addr \times hmsg(l) \rightarrow md.$
- H: $pk.seed \times addr \times hmsg(2) \rightarrow md.$
- F: $pk.seed \times addr \times hmsg(1) \rightarrow md.$
- PRF: $sk.seed \times addr \rightarrow md.$
- $\mathsf{PRF}_{\mathsf{msg}}$: $sk.prf \times Optrand \times msg \rightarrow R.$
- $\mathsf{H}_{\mathsf{msg}}$: $R \times pk.seed \times pk.root \times msg \rightarrow mhash.$

2) WOTS$^+$: WOTS$^+$ is the scheme of one-time signature used in SPHINCS$^+$. The overhead of generating $wotsSk$ is very low, so it is not stored and only needs to be generated when it is used. WOTS$^+$ consists of three main functions:

1. $wotsPk \leftarrow$ wotsPkGen($sk.seed$, $pk.seed$, $addr$). The WOTS$^+$ public key generation function takes $sk.seed$, $pk.seed$ and $addr$ to return $wotsPk$. The function is used when constructing the leaf node of the XMSS tree.

2. $wotsSig \leftarrow$ wotsSign(msg, $sk.seed$, $pk.seed$, $addr$). The WOTS$^+$ signature generation function uses msg, $sk.seed$ and $pk.seed$ to generate $wotsSig$. The function is used in XMSS signature generation.

3. $wotsPk \leftarrow$ wotsPkFromSig($wotsSig$, msg, $pk.seed$, $addr$). The WOTS$^+$ signature verification function computes $wotsPk$ from $wotsSig$, msg, $pk.seed$ and $addr$. The function is used in XMSS signature verification.

3) SPHINCS$^+$ Hypertree: SPHINCS$^+$ Hypertree (HT), a variant of XMSS^{MT}, has multiple layers of XMSS trees. The lowest layer is used to sign $forsPk$, while the XMSS trees of other layers are used to sign the root nodes of XMSS trees in the below layer. Consider an HT with a total height of h and d layers of XMSS trees with height h' = h/d. Layer d - 1 has a single XMSS tree, layer d - 2 has $2^{h'}$ XMSS trees, and so on. Layer 0 has $2^{h-h'}$ XMSS trees. The overhead of SPHINCS$^+$ HT mainly comes from the two XMSS algorithms:

1. $\langle root, authPath\rangle \leftarrow$ xmssTreehash(sk, idx, $addr$). Given sk and a leaf index idx, the function computes the authentication path $authPath$ and the $root$ node of the XMSS tree. An XMSS tree includes $2^{h'}$ leaf nodes. To generate a leaf, first generate $wotsPk$ and hash it to a leaf node. Then, starting from the leaf nodes, layer by layer, branch nodes are generated by hashing the concatenation of their two child nodes. Finally, the $root$ is obtained.

2. $root$ $\leftarrow$computeRoot($leaf$, idx, $authPath$, $pk.seed$, $addr$). Given a $leaf$, a leaf's index idx and an authentication path $authPath$, the computeRoot function computes a $root$ node when verifying an XMSS signature.

4) FORS: FORS is a few-time signature scheme, which is used to achieve statelessness. Three main functions are:

1. $root$ $\leftarrow$ forsTreehash(sk, idx, $addr$). This function is almost identical to xmssTreehash, except that the leaf node generation only uses the hash function F to hash $forsSk$. Since the cost of computing $forsSk$ is much less than that of computing $wotsPk$, The cost of leaf node generation of FORS is much less than that of the XMSS tree.

2. $\langle forsSig, forsPk\rangle$ $\leftarrow$ forsSign ($mhash$, $sk.seed$, $pk.seed$, $addr$). The FORS signature generation function takes $mhash$, $sk.seed$, $pk.seed$ and $addr$ to return $forsSig$ and $forsPk$. The $forsPk$ is obtained by splicing k root nodes of multiple FORS trees and hashing them. The reason why $forsSig$ and $forsPk$ are generated both in this function is that they need to construct the same FORS trees.

3. $forsPk \leftarrow$ forsPkFromSig ($forsSig$, $mhash$, $pk.seed$, $addr$). The FORS signature verification function computes $forsPk$ from $forsSig$, $mhash$, $pk.seed$ and $addr$, where root nodes of multiple FORS trees need to be computed.

5) SPHINCS$^+$: SPHINCS$^+$ consists of the above structures and includes three cryptographic functions (KG, Sign, Verify), which are described as follows:

1. SPX_KG: It is used to generate sk and pk. The most time-consuming part is to get the $pk.node$ because it needs to construct the top XMSS tree. The $sk.seed$, $sk.prf$ and $pk.seed$ are directly randomly generated.

2. SPX_SIGN: The SPHINCS$^+$ signature consists of R, $forsSig$ and an HT signature. The process of SPX_SIGN has four steps: (1) A random value R is generated by using $\mathsf{PRF}_{\mathsf{msg}}$. (2) $\mathsf{H}_{\mathsf{msg}}$ is performed to obtain outputs. Then the outputs are extracted to get md, the tree index $idxTree$ and the leaf index $idxLeaf$. (3) The md is signed to get $forsSig$ using the $idxLeaf$-th FORS keypair of the $idxTree$-th XMSS tree on the lowest HT layer. Then $forsPK$ is obtained using $forsSig$. (4) Use HT to sign $forsPK$.

3. SPX_VERIFY: The signature verification consists of three steps: (1) Recompute md, $idxTree$ and $idxLeaf$. (2) Compute a candidate FORS public key. (3) Verify the HT signature using the candidate FORS public key. The signature verification succeeds if the generated root node is equal to the public key's root node. Otherwise, it fails.

C. Related Work

Table II summarises large-scale parallel work on HBSs. XMSS, LMS and SPHINCS exhibit comparable parallelisation approaches due to their similar schemes. Wang et al. [22] apply a two-level parallel framework to XMSS on the GPU. The first level is inter-node parallelism, which distributes the computation across multiple nodes. The second level is intra-node parallelism, which distributes the computation within each node. Kang et al. [20] designed a three-level parallel framework for LMS on the CPU. The first two levels, inter-node and intra-node, are similar to those of Wang et al. The third level, vector parallelism, uses CPU-specific implementation techniques to further improve performance. Wang et al. [21] designed a parallel Merkle tree traversal and implement it on LMS. For SPHINCS, Sun et al. [23] employed parallelism of inter-tree, inter-node, and intra-node for CPU optimisation. Their GPU implementation only incorporated data parallelism, lacking a dedicated parallel framework. For SPHINCS$^+$, Kim et al. [24] introduced a hybrid parallel implementation. Their implementation does not support algorithmic and data parallelism; that is, it does not facilitate parallel computation across blocks or employ multi-streaming techniques. Additionally, it does not support "s" parameter sets due to the limitation that tasks can only be parallelised within a block, resulting in excessively long execution times.

Although current work on SPHINCS$^+$ is limited, we can draw on other aspects. SPHINCS$^+$ realises security and performance improvement based on SPHINCS [26], so we can refer to the performance optimisation methods developed for SPHINCS. Specifically, SPHINCS has been optimised for special platforms, such as FPGA [27] and embedded ARM [28]. This scheme does not improve the performance of a single signature, and the signing time of some parameter sets exceeds 30 seconds. As such, this approach is not practical for real-world applications.

In addition, SPHINCS$^+$ has an important structure, the XMSS tree. Optimisation schemes for the XMSS tree are still of reference significance. Overall, optimisation schemes can be divided into three categories: vectorization technology such as AVX2 [29], [30] and NEON [31], optimization of hardware-specific instructions such as those on Intel [32] and Cadence [33] processors, and accelerator-specific optimizations such as those on FPGA [34]–[37], ASIC [38] and GPU [22].

Extensive research has been conducted on the use of GPUs to accelerate PQC algorithms, i.e., AES, Falcon and Dilithium. Hajihassani et al. [39] exploit the high parallelism of GPUs by deploying data parallelism of AES, which is based on the idea that the computation of each data block in a single data stream is independent. Lee et al. [40] presented the first parallel implementation of Falcon on various GPUs. They developed an iterative version of the sampling process, which is the most time-consuming Falcon operation. Shen et al. [41] presented a high-throughput GPU implementation of Dilithium. From the aspects of individual operations, concurrent task processing, dynamic task scheduling and data transfer hiding, they performed comprehensive optimization.

III. DESIGN FOR PARALLELISED SPHINCS$^+$

This section studies parallel methods of SPHINCS$^+$. Three parallel schemes are provided: algorithmic, data and hybrid

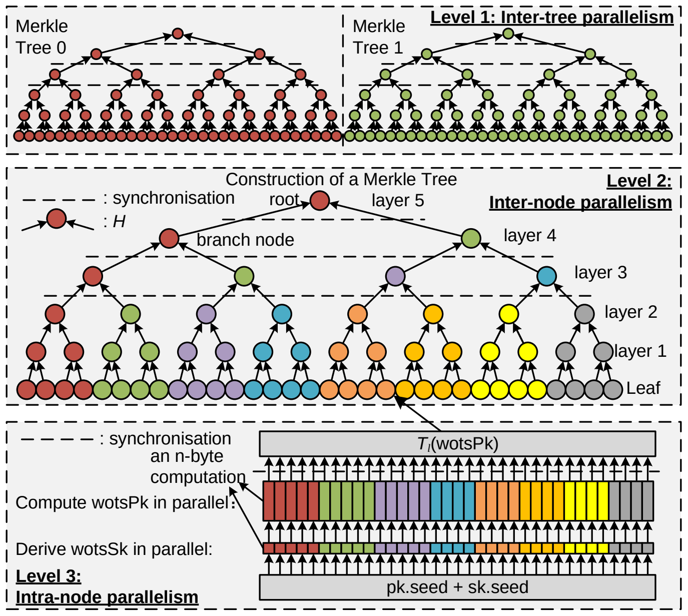
Merkle Tree 0
Merkle Tree 1
Level 1: Inter-tree parallelism
Construction of a Merkle Tree
: synchronisation
: H
root
layer 5
Level 2: Inter-node parallelism
branch node
layer 4
layer 3
layer 2
layer 1
Leaf
: synchronisation
an n-byte computation
Compute wotsPk in parallel:
Derive wotsSk in parallel:
Level 3: Intra-node parallelism
T_l(wotsPk)
pk.seed + sk.seed

Fig. 1. The three-level parallel framework of CUSPX.

parallelism. We call a single execution of keypair generation, signature generation or signature verification a "$task$".

A. Algorithmic Parallelism

In this section, we address the challenge of large-scale parallel designs. First, we present a three-level parallel framework. Then, we illustrate the parallel schemes of internal designs.

1) Three-Level Parallel Framework: The three-level parallel framework of CUSPX is shown in Fig. 1. Level 1 is inter-tree parallelism, in which multiple Merkle trees are constructed in parallel. Level 2 is inter-node parallelism, in which nodes of the same layer within a single Merkle tree are generated in parallel. Level 3 is intra-node parallelism, in which a leaf node is generated in parallel. The Three levels can be used independently or in combination.

Only level 1 is used. To construct multiple Merkle trees on the same layer, level 1 can be used. Each thread can independently construct a Merkle tree in parallel, without requiring synchronization, as the procedure does not require data correlation.

Only level 2 is used. Consider eight threads constructing a Merkle tree with 32 leaf nodes and five layers. First, each thread is responsible for constructing the subtree from a leaf node to layer 2. At layer 3, only four nodes are generated. Therefore, four threads then acquire data from other threads and generate the four nodes in parallel. To ensure that all threads have completed their computations and that the data is correct, a synchronization operation is required. Similarly, threads need to be synchronised before the parallel generation of 2 branch nodes at layer 4. Finally, the root node of the Merkle tree is obtained at layer 5.

Only level 3 is used. Assuming eight threads are used to generate a leaf node, the computation of $wotsPk$ is parallelised. The procedure of generating $wotsPk$ comprises len n-byte computations. These n-byte computations are independent of each other, so they are distributed equally among eight threads. After each thread generates a part of $wotsPk$, the synchronisation operation is performed to obtain the whole $wotsPk$. Then $wotsPk$ is then hashed by the T_l hash function to obtain a single leaf node. All the leaf nodes are generated in this way. After all the leaf nodes have been generated, the Merkle tree is constructed using a single thread.

Parallel efficiency. Parallel efficiency decreases from level 1 to level 3 for two reasons. First, the proportion of inter-thread synchronisation overhead increases as the number of levels increases. Second, the proportion of parallelizable parts and the parallel scale decreases. Level 1 has no non-parallelizable parts. Level 2 has two types of nodes, branch nodes and leaf nodes. Although the overhead of branch nodes is much less than that of leaf nodes, the parallelism of branch nodes decreases from bottom to top. In level 3, the hash function $T_l(wotsPk)$ cannot be parallelised and it takes a lot of time. Thus, the parallel efficiency is lower.

Memory usage. Levels 3 have the same memory footprint as the serial implementation, while level 2 requires more memory because all node values at a layer must be stored in memory before computation. When level 1 is used alone, the memory footprint is unchanged. But when level 1 is used with level 2, the memory footprint becomes more. If memory is constrained, levels 1 and 3 are recommended for use alone. Otherwise, level 2 can be used to achieve higher parallel efficiency and scale.

Multi-levels parallelism. When single-level parallelism cannot meet the parallelism requirements, multi-level parallelism should be considered. Consider the example of using levels 2 and 3. level 2 offers a maximum parallelism of $2^{h'}$, corresponding to the number of leaf nodes. Level 3 exhibits a maximum parallelism of len. When both levels are used concurrently, the maximum parallelism is $2^{h'} \times len$. Denoting the number of threads in levels 2 and 3 as t_1 and t_2 respectively, the total number of threads used is $t_1 \times t_2$. First, each leaf node is constructed using t_2 threads and t_1 leaf nodes are constructed in parallel, that is, all threads construct leaf nodes at the same time. Then the parallel construction method of level 2 is used to generate branch and root nodes.

Suitable solution for GPU. As the GPU's memory is typically large, all three levels of parallelism can be used. However, parallel efficiency decreases as the number of levels increases. Therefore, low levels are preferred. To increase parallelism, multi-level parallelism is also employed. In summary, the priorities of parallel schemes are: level 1 > level 1+2 > level 1+2+3. Table V summarises the parallel methods used by the three cryptographic functions, where the underscore represents levels of parallelism used, NF and SF stands for node-first and subtree-first scheme of parallelised Merkle tree. In CUSPX, these parallel algorithms are executed sequentially, one after the other.

2) Parallelised WOTS$^+$: WOTS$^+$ consists of three functions: wotsPkGen, wotsSign and wotsPkFromSig. The parallel scheme for these functions is similar: len n-byte computations can be computed in parallel. The standalone WOTS$^+$ exists only in SPHINCS$^+$ verifying. (WOTS$^+$_3). The other two functions are used with multiple levels of parallelism.

TABLE V
PARALLEL METHODS FOR ALGORITHMIC PARALLELISM

	Level 1	Level 1+2	Level 1+2+3
SPX_KG		XMSS_2 (SF)	XMSS_23 (SF)
SPX_Sign	FORS_1	FORS_12 (NF)	FORS_12 (NF)
	HT_XMSS_1	HT_XMSS_12 (NF)	HT_XMSS_123 (NF)
	HT_WOTS$^+$_1	HT_WOTS$^+$_1	HT_WOTS$^+$_13
SPX_Verify	FORS_1	FORS_1	FORS_1; WOTS$^+$_3

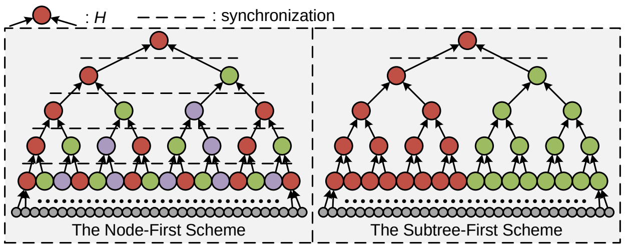

Fig. 2. Two schemes of parallel Merkle tree construction with three threads.

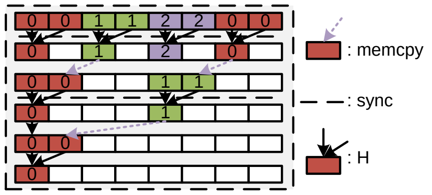

Fig. 3. Data flow in memory of the node-first scheme with 3 threads.

3) Parallelised Merkle tree: Both FORS and XMSS trees are instances of Merkle trees. We consider schemes with an arbitrary number of threads. Since the number of threads may not be a power of two, the solution in Fig. 1 may not be optimal. Fig. 2 shows two schemes for parallel Merkle tree construction with three threads, where one colour represents a thread. The subtree-first scheme constructs a power of 2 subtrees to minimise synchronization overhead. The node-first scheme evenly distributes the node generation of each layer to achieve load balancing.

The node-first scheme is described first. When constructing a Merkle tree, branch nodes are generated layer-by-layer from the bottom up. The computation of nodes in a layer is divided equally among three threads. To avoid memory overlap when computing in parallel, the computation is performed as shown in Fig. 3. Eight blocks are used, each representing a contiguously stored $leaf$ array. The computation proceeds as follows: first, the thread synchronises; second, the memory is copied; third, the hash function is applied. Finally, the first node stored in the $leaf$ array is the root node.

The subtree-first scheme first determines the number of subtrees, $stNum$, as the maximum power of 2 less than or equal to the maximum parallelism, $para$. Then, it uses $stNum$ threads to construct the subtrees in parallel, using the serial Merkle tree algorithm. Once the subtrees are constructed, the node-first scheme uses the root nodes of the subtrees to complete the construction of the Merkle tree in parallel.

Both schemes have their advantages and disadvantages. The subtree-first scheme requires only a small amount of memory, proportional to the number of subtrees. It also requires fewer synchronization operations, which can improve performance. The node-first scheme is more efficient in load balancing,

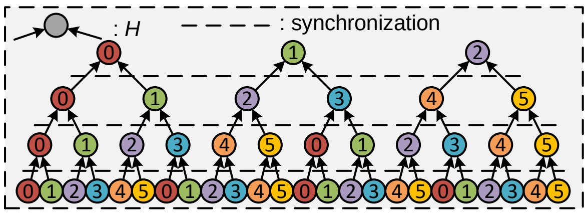

Fig. 4. Parallel construction of 3 FORS trees with 6 threads.

easier to implement for irregular Merkle trees, and easier to leverage GPU coalesced access technology (introduced in Section IV-B). We will describe the application scheme according to the specific situation in the later parts.

4) Parallelised XMSS: When performing SPX_KG, only top XMSS tree can be parallelised using parallelism of 2 and 3 (XMSS). We recommended using the subtree-first scheme because the number of nodes at every layer is power of 2.

5) Parallelised FORS: FORS is used in both SPHINCS$^+$ signing and verifying. The multiple FORS trees are located in the same layer, so inter-tree parallelism is possible. Inter-node parallelism can be considered for high parallelism. In FORS signing, parallelism of levels 1 and 2 can be used (FORS_1, FORS_12). Level 3 cannot be used because the leaf nodes of FORS trees do not include WOTS$^+$ and cannot be parallelised. In FORS verifying, only levels 1 can be used because no Merkle trees are constructed (FORS_1).

We illustrate the parallel FORS scheme using FORS signing using parallelism of levels 1+2. Fig. 4 shows our parallel design using node-first scheme, in which threads generate nodes sequentially and synchronise at each layer. The reason for choosing node-first scheme is that the number of FORS trees is not a power of 2, so load balancing is more important.

Algorithm 1. details the parallel FORS signing, which signs m to generate $forsSig$. Leaf node generation (lines 2-16) and Merkle tree construction (lines 18-32) are executed separately to facilitate Merkle tree construction. Line 1 initialises thread ID to tid and converts the message to an index array, $indices$. The number of leaves in a single FORS tree and across all FORS trees is then retrieved. Line 4 guarantees that only para threads are employed. Leaf node height and row indices are set in lines 5 and 8, respectively. Lines 6 and 7 distribute leaf generation across threads using the parallel scheme illustrated in Fig. 4. Following the index setting, the $forSk$ of $forsSig$ is generated (line 9). Finally, all leaf nodes are generated and stored in the shared array $leaf$ (line 10).

After leaf nodes are generated, Merkle trees are constructed. There are three methods to build Merkle trees in parallel. The first method is to have threads build FORS trees in parallel and one-to-one (level 1). This method has low maximum parallelism. The second method is to have threads build FORS trees in parallel, one by one (level 2). This method has poor parallel efficiency, so we do not recommend it. This method has low maximum parallelism, so we only consider the third method. The third method has threads build multiple FORS trees in parallel (level 1+2). We treat multiple FORS trees as multiple subtrees and do not process them separately.

The FORS trees are constructed from lines 18 to 32. We begin by describing the parallel construction of a Merkle

Algorithm 1. parallelForsSign

```
Input: $m$, $addr$, $pkAddr$, $para$; shared: $leaf$, $authPath$, $roots$
Output: $forsSig$, $forsPk$
1: $tid \leftarrow$ getThreadId(); $indices \leftarrow$ messageToIndices($m$)
2: $num \leftarrow (1 \ll$ a)    ▷ number of leaves in a FORS tree
3: $maxPara \leftarrow num \cdot k$    ▷ All number of leaves
4: if $tid < para$ then
5:     $addr \leftarrow$ setHeight(0)
6:     for $i \leftarrow tid$ to $maxPara$ step $para$ do
7:         $id \leftarrow i$ / $num$; $idx \leftarrow id \cdot num$
8:         $addr \leftarrow$ setIndex($indices[id]$ + $idx$)
9:         $forsSig$+$id{\cdot}n$+$id{\cdot}n{\cdot}a \leftarrow$ forsGenSK($sk.seed$, $addr$)
10:        $leaf$+$i{\cdot}n \leftarrow$ forsGenLeaf($sk.seed$, $pk.seed$, $i$, $addr$)
11:    end for
12: end if
13: allthreads.sync(); $ii \leftarrow 1$
14: if $tid < k$ then
15:     memcpy($authPath$+$tid{\cdot}a{\cdot}n$,
16:             $leaf$+$tid{\cdot}num{\cdot}n{\cdot}(indices[tid]{\oplus}$0x1$){\cdot}n$, $n$)
17: end if
18: for i $\leftarrow 1$ to a do
19:     allthreads.sync(); $addr \leftarrow$ setHeight($i$)
20:     if $tid < para$ then
21:         for $j \leftarrow tid$ to $k({\cdot}h{\gg}i)$ step $para$ do
22:             $off \leftarrow 2{\cdot}j{\cdot}ii{\cdot}n$; $addr \leftarrow$ setIndex($j$)
23:             memcpy($leaf$+$off$+$n$, $leaf$+$off$+$ii \cdot n$, $n$)
24:             $leaf$+$off \leftarrow$ H($leaf$+$off$, $pk.seed$, $addr$)
25:             $t \leftarrow j$ / ($num{\gg}i$)
26:             if $j\%(num{\gg}i)$ = (($indices[t]{\gg}i){\oplus}$0x1) then
27:                 memcpy($authPath$+$i{\cdot}n$+$t{\cdot}a{\cdot}n$, $leaf$+$off$, $n$)
28:             end if
29:         end for
30:     end if
31:     $ii \leftarrow ii \cdot 2$
32: end for
33: allthreads.sync(); $p \leftarrow tid{\cdot}n$; $q \leftarrow n{\cdot}a$; $r \leftarrow tid{\cdot}n{\cdot}a$
34: if $tid < k$ then
35:     memcpy($forSig$+$n$+$p$+$r$, $authPath$+$tid{\cdot}r$, $q$)
36:     memcpy($roots$+$p$, $leaf$+$p{\cdot}num$, $n$)
37: end if
38: $forsPk \leftarrow T_k$($roots$, $pk.seed$, $pkAddr$)
```

tree. First, line 19 synchronises all $para$ threads before the computation of each layer begins. Then, line 21 distributes the computations of all nodes to all $para$ threads in a round-robin way. After setting the index (lines 19 and 22) and obtaining the data (line 23), the hash function H is performed. Finally, the loop terminates when the number of layers reaches the highest layer, a (line 18). At each layer i, the variable off is used to avoid memory overlap (lines 22, 23, 24 and 27).

Unlike parallel Merkle tree construction, parallel FORS tree construction requires storing the authentication path. Lines 14-17 and 25-28 show how to store authentication paths on the leaf layer and non-leaf layers, respectively. Once k FORS trees have been constructed, the authentication path and root node are stored in $forsSig$ (line 35) and $roots$ (line 36), respectively. Finally, the hash function T_k is applied to the roots array to obtain $forsPk$ (line 39).

6) Parallelised HT: When performing SPX_SIGN, HT and multiple layers of WOTS$^+$ signatures should be generated. HT consists of d layers of XMSS trees, which can be constructed in parallel (HT_XMSS). The HT can be constructed using parallelism of levels 1, 2 and 3. The parallelization of HT can be divided into two steps: parallelizing the generation of leaf nodes and branch nodes. The process of parallelizing the generation of leaf nodes is shown in Figure 1. The method of generating branch nodes from multiple XMSS trees is consistent with the FORS parallelization process. The multiple layers of WOTS$^+$ signatures can also be generated in parallel using parallelism of levels 1 and 3 (HT_WOTS$^+$).

TABLE VI
PARALLEL METHODS FOR HYBRID PARALLELISM

	Scheme
KG	XMSS_23 (NF)
Sign	FORS_12 (NF); HT_XMSS_12 (NF); HT_WOTS$^+$_13
Verify	FORS_1; WOTS$^+$_3

B. Data Parallelism

Data parallelism refers to multiple threads performing multiple $tasks$ in parallel, one $task$ per thread. Data parallelism is divided into two categories: single-keypair and multi-keypairs. The distinction between the two categories is negligible. The main difference is the amount of data transferred between CPUs and GPUs. Data parallelism can achieve higher throughput than algorithmic parallelism for large-scale tasks, due to its high parallel efficiency. We also design a multi-stream scheme to further improve the throughput, see Section IV-C.

C. Hybrid Parallelism

Hybrid parallelism refers to multiple groups performing multiple $tasks$ in parallel, one $task$ per group. Here, we address the challenge of designing algorithms of hybrid parallelism with high parallel efficiency. We understand the parallelism of tasks as a higher level existence than the three-level parallel framework, that is, hybrid parallelism of CUSPX is four-level parallel framework. CUSPX only needs information about the number of tasks, and it can adjust the parallel scheme based on its load-balancing schemes.

Table VI presents parallel methods for hybrid parallelism, where the underscore represents the level of parallelism used, NF and SF stand for node-first and subtree-first schemes. In the design of this subsection, task parallelism is always present. Unlike algorithmic parallelism, we do not design a hybrid parallel scheme for HT_XMSS_123, because the parallel efficiency is low and much memory is used.

1) Parallelised WOTS$^+$: WOTS$^+$ consists of multiple n-byte computations. The largest overhead comes from the execution of the Winternitz chain, which is the continuous execution of the hash function. In WOTS$^+$ public key generation, all the chain lengths are the same, but not in signature generation and signature verification. The maximum parallelism of the two functions is len. If the maximum parallelism is insufficient to support multiple $tasks$ with len parallelism, load balancing schemes are needed. We consider four kinds of schemes.

Local load-balancing scheme. The len n-byte computations of each group are divided equally among the threads. This allows each group to compute independently, with maximum load imbalance but no synchronization overhead.

Global load-balancing scheme. All n-byte computations are evenly distributed to all threads across all groups. This scheme requires global synchronization, but it effectively avoids load imbalance.

Working pool scheme. The scheme works by placing all n-byte computations of a group in a working pool. When a thread completes a chain, it checks the pool for unfinished chains. If it finds one, it computes that chain. Otherwise, the thread terminates. The working pool is shared across groups, which introduces an intra-group synchronization issue. Compared to global sharing, this scheme still suffers from load imbalance.

Pre-allocation scheme. The core of this scheme is to determine which n-byte computations should be executed by which thread through pre-calculation. By counting the accumulated length of hash chains for each thread, it can be determined which n-byte computations each thread should execute.

The scheme suitable for GPU. In large-scale parallel computing, synchronization overhead is often the dominant factor affecting performance. Local load-balancing schemes, which do not require synchronization, therefore typically offer the best performance in most use cases. However, if local load-balancing leads to extremely unbalanced load, global load-balancing should be used.

2) Parallelised XMSS: Since the majority of task numbers are not powers of 2, we use a node-priority scheme here to improve load-balancing capability. For leaf nodes, we evenly distribute the generation of all nodes in one layer of all tasks to all threads. For the generation of branch nodes, we treat the XMSS trees of multiple tasks as multiple subtrees and use the method describe in Section III-A5 to construct.

3) Parallelised FORS: FORS parallelisation can utilise parallelism of levels 1 and 2, with level 1 having the highest parallel efficiency. FORS Signing use the same parallel method of parallelised XMSS. FORS Verifying only use level 1, so the parallel method is difference. Unlike WOTS$^+$, the global load-balancing method is effective for FORS verifying because the computation overhead of a single FORS tree construction is large. Synchronization overhead is no longer the bottleneck, and performance is dominated by load distribution.

Fig. 5 depicts the local and global load-balancing schemes employed within FORS verifying, where each block represents a FORS tree. The construction of seven FORS trees constitutes a single $task$ and seven such tasks are executed by 25 threads. In the local load-balancing scheme, the 25 threads are partitioned into seven groups, each responsible for constructing 7 FORS trees (a single task). Threads are assigned uniformly to each group, with surplus threads remaining idle (21-24). This results in three threads per group, distributing the FORS trees evenly. Consequently, one thread processes three trees, while the remaining two handle two trees each. In the global load-balancing scheme, all FORS trees are distributed across the 25 threads, with each thread processing a maximum of two trees. This scheme surpasses the local scheme as it eliminates the requirement for any thread to handle three trees.

4) Parallelised HT: Similarly, CUSPX uses the node-first method to construct multiple HT trees (HT_XMSS). A clear distinction of tasks, XMSS trees and nodes is all that is needed to design. For the WOTS$^+$ signing involved in HT (HT_WOTS$^+$), we recommend using global load-balancing scheme. The reason is that this method is more scalable and facilitates real-time adjustments to the GPU implementation.

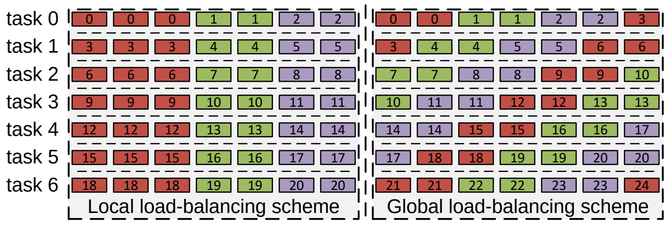

Fig. 5. Two load-balancing schemes of FORS for hybrid parallelism.

IV. Implementations on the GPU

CUSPX contains over 10,000 lines of CUDA code. In this section, we address the challenge of GPU efficient implementation. Many GPU optimizations have been performed in CUSPX, which can be roughly divided into three parts:

1. *Instruction Optimisations*. It stands for instruction cycle reduction and instruction order optimisation.

2. *Memory Access Optimisations*. It describes how to access various levels of GPU memory more efficiently.

3. *Parallel Optimisations*. It describes how to efficiently coordinate computing components for parallel computing.

The hash function, as the core of HBS, is a key point affecting computational performance. We optimise according to [22] and only describe the rest of the content here.

A. Instruction Optimisations

Synchronization. We use cooperative_groups for global memory synchronization and syncthreads for shared memory synchronization. To reduce the number of synchronizations, we analyze whether data needs to be synchronised, and try to use shared memory synchronization as much as possible, because the overhead of shared memory synchronization is smaller than that of global one.

Kernel function fusion. We merge multiple kernels of a cryptographic function into one kernel, thereby reducing overheads such as startup and data transfer. For example, when performing SPX_SIGN, only the private key and message need to be input, and the signature is output.

B. Memory Access Optimisations

Coalesced access. As mentioned in Section III-A5, the working theory of coalesced access is that if threads in a warp handle the same 128-byte chunk and the chunk is aligned to the 128-byte cache line, fewer cache transactions are needed compared to random access memory. For example, coalesced access can be used if four threads in a warp access the same 128-byte chunk when n = 32 (32 bytes). We extensively apply this technique in the process of parallelizing WOTS$^+$ and branch nodes, and this scheme is very compatible with the global load balancing scheme. The idea of applying this technique is that all underlying operators (WOTS$^+$ chain or nodes generation) are evenly distributed to threads in order, and the data they read are always stored continuously.

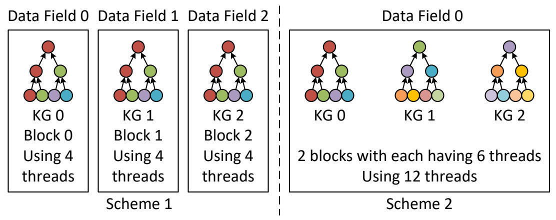
Data Field 0
Data Field 1
Data Field 2
Data Field 0
KG 0 Block 0 Using 4 threads
KG 1 Block 1 Using 4 threads
KG 2 Block 2 Using 4 threads
KG 0
KG 1
KG 2
2 blocks with each having 6 threads Using 12 threads
Scheme 1
Scheme 2

Fig. 6. Two schemes for implementing hybrid parallelism on the GPU, where three KG tasks are assigned to 12 threads.

C. Parallel Optimisations

The core of parallel optimisations is how to make fuller use of the GPU's computing resources.

Scheme of hybrid parallelism. Fig. 6 shows two schemes for implementing hybrid parallelism on the GPU. the left one is the simplified scheme provided by GPU, which is to let a GPU block handle a $task$ in its data field. The parallel scheme is only implemented on shared memory. This solution hands over the specific parallel scheduling to the GPU platform, which is highly related to hardware. The advantage is that it is simple to program. However, there are also many disadvantages: 1. The parallel scale of a task cannot be larger than the maximum number of threads per block (1024 in RTX 3090). Therefore, this design cannot cope with low-latency scenarios. 2. If too many threads are used within a block, the parallel efficiency drops significantly. 3. It depends on the platform. The conventional way of porting to the CPU platform using this method is to have one task run on one node, which seriously restricts practical use.

CUSPX uses the general scheme described in the right figure, treating multiple tasks as one large task to be processed in a data domain. The only difference between different tasks is that the inputs are different. We distinguish these inputs of threads so that they can be processed uniformly. The disadvantages of this scheme are: 1. The programming logic is complex and difficult to maintain; 2. It uses a lot of global memory. But there are many advantages: 1. There is no parallel limit for a single task; 2. Coalesced memory access can be utilised at any time; 3. It facilitates load balancing, handing over the process of parallel scheduling to the code; 4. It is platform-independent, and all platforms can use this scheme.

The number of threads and blocks. It is required to specify the number of blocks ($nBlocks$) and the number of threads in a block ($nThreads$) for GPU execution. The product of $nBlocks$ and $nThreads$ is the total number of threads. Since most of our work uses a global load-balancing scheme, we don't need to consider the issue of the number of threads in shared memory. In our tests, the optimum number of threads is 32 in most cases. The optimal number of blocks, however, needs to be obtained through testing. When the maximum parallelism mp does not exceed the number of GPU cores nc, the number of blocks is mp/32. When it exceeds, the optimal number of blocks in most cases is nc/32 for algorithmic parallelism. While in hybrid parallelism, the optimal number of blocks in most cases is 2*nc/32. When displaying data, we obtain the optimal solution by testing various number of blocks.

TABLE VII
THE DETAILS OF THE TESTBED

Item	Value
GPU (Architecture)	Geforce RTX 3090 (Ampere)
CUDA cores (SMs)	10,496 (80)
Base memory bandwidth (clock)	784.4 GB/s (1395 MHz)
Copy engines	2
Compute capability	8.6
Compiler	CUDA-11.2
CPU	Intel Xeon Gold 5218R @ 2.1 GHz
CPU to GPU bandwidth	12.4 GB/s
GPU to CPU bandwidth	13.2 GB/s

The number of streams. As multi-streaming can lead to increased latency, we only use multi-streaming in data parallelism, because it does not care about latency. Hybrid parallelism is more sensitive to latency, and the latency caused by multi-streaming cannot be ignored. The design of multiple streams needs to consider two points, one is the number of streams, and the other is how tasks are distributed on a specified number of streams. CUSPX uses the method of [22]. Assuming the number of GPU cores is nc, and the number of tasks is nt, then the number of streams is $ns = nt/nc$, and the distribution of tasks on ns streams is nc, nc, ..., (nt % nc).

V. EVALUATIONS

A. Evaluation Criteria and Setups

Performance is evaluated in terms of throughput and latency. Latency, expressed in milliseconds (ms), encompasses both data transfer time between CPUs and GPUs and GPU kernel execution times. Throughput is measured as the number of completed tasks per second (tasks/sec). While some studies employ cycle counts for evaluation, we provide only the main frequency. This is because latency is equal to the product of cycle count and main frequency, provided the device frequency remains constant. This condition was invariably met in our tests. Parallel efficiency and speedup, common metrics in parallel studies, are also considered. All reported results are averages of the first 64 tests.

Table VII outlines the testbed configuration. The base clock and base memory bandwidth are provided due to the deactivation of GPU boost, as enabling it can diminish GPU longevity. Constraints on data transfers between CPUs and GPUs exist due to the mainboard, PCI-E, and CPU memory. Therefore, only the transfer bandwidth directly linked to performance is presented. The Cryptographic Algorithm Validation Program (CAVP) from NIST validates the correctness of all SHA-256 and SHAKE256 implementations [42]. The optimised submission to NIST serves as the basis for our SPHINCS$^+$ implementations [9]. The reference implementation's correctness verification functions are employed to examine all implementations. Furthermore, seeds are manually set to ensure consistency between keys, signatures, and verification outcomes produced by the reference and our implementations. GPU optimisation techniques from [22] are utilised.

B. Tests of Components

Table VIII shows the latency of hash functions, which is represented in μs. The hash function is a basic operator in

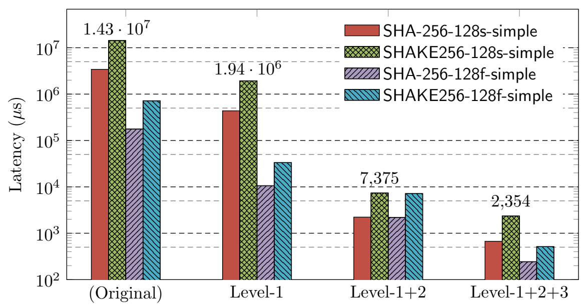

Fig. 7. Latency (ms) of signature generation for algorithmic parallelism.

TABLE VIII
LATENCY (μS) OF HASH FUNCTIONS FOR SPHINCS$^+$-128F-SIMPLE

	T_l(wots)	T_l(fors)	H	F	PRF	PRF_{msg}	H_{msg}
SHA-256	19.9	18.7	3.2	2.8	1.6	5.9	4.8
SHAKE256	31.0	30.1	6.9	6.5	5.1	5.2	6.3

HBS. There are six hash functions in SPHINCS$^+$, among which the message length of the T_L function has two types, which are used for WOTS and FORS respectively. The two have different lengths, hence different performances. Results show that the latency of the T_l function is the highest, and the performance using SHAKE256 is slightly lower.

Table IX shows the performance of each component under algorithmic parallelism in CUSPX, which corresponds to the components used in Table V. The number of cores used for testing is the maximum parallel scale of the components, up to 10,496, which is the number of GPU cores. Cores do not mean the number of threads used. When it's less than 10,496, the number of threads is equal to the number of cores. However, when more than 10,496 threads need to be used, this is not the case. For example, the number of threads running HT_XMSS is 10,496×2 threads, the reason is that using more than 10,496 threads can improve performance. The parallel efficiency is compared with naive, which uses no parallel method. Generally speaking, the parallel efficiency won't be more than 1. The reason why the parallel efficiency of some components exceeds 1 may be due to the increase in cache hit rate, coalesced access and other GPU hardware reasons. The results show that most components have high parallel efficiency, an important reason is that the parallel scale has not yet reached the hardware limit. As it approaches the number of GPU cores, the parallel efficiency begins to decline.

C. Tests of Algorithmic Parallelism

The goal of algorithmic parallelism is to reduce the latency of a single $task$. In this subsection, we first evaluate the performance benefits of different parallelism levels. Then, we present the latency for different parameter sets.

Fig. 7 shows the latency of SPHINCS$^+$ signature generation for 128s parameter sets. The optimisations are gradually added, with "(Original)" "Level_1" "Level_1+2" and "Level_1+2+3" denoting the original serial code, schemes of the three-level parallel structure, respectively. The "(Original)" code only performed necessary operations for porting based on the official single-thread code, such as explicitly transferring data between the CPU and GPU, and changing dynamic allocation to static methods. Evaluated on a GeForce RTX 3090 using All 10,496 cores with SPHINCS$^+$-SHA-256-128s-simple, the speedup brought by algorithmic parallelism is 3,420.92 / 0.67 = 5105.84×, with a parallel efficiency is 5,105.84 / 10,496 = 48.65%.

TABLE IX
LATENCY (MS), NUMBER OF CORES AND PARALLEL EFFICIENCY (PE) OF COMPONENTS IN CUSPX

Components	Strategies	SHA-256-128s-simple			SHA-256-128f-simple		
		Latancy	Cores	PE	Latancy	Cores	PE
WOTS$^+$	Naive	1.064	1	1.00	1.053	1	1.00
	Level 3	0.033	35	0.92	0.032	35	0.94
HT_WOTS$^+$	Naive	6.996	1	1.00	21.825	1	1.00
	Level 1	1.115	7	0.90	1.260	22	0.79
	Level 13	0.033	7*35	0.87	0.033	22*35	0.86
FORS	Naive	351.901	1	1.00	13.710	1	1.00
	Level 1	25.658	14	0.98	0.378	33	1.10
	Level 12	0.157	10,496	0.21	0.060	33*64	0.11
XMSS	Naive	501.739	1	1.00	7.744	1	1.00
	Level 2	0.950	512	1.03	0.933	8	1.04
	Level 23	0.114	10,496	0.42	0.059	8*35	0.47
HT_XMSS	Naive	4,327.395	1	1.00	157.073	1	1.00
	Level 1	508.278	7	1.22	8.071	22	0.88
	Level 12	0.934	7*512	1.29	0.902	8*22	0.99
	Level 123	0.457	10,496	2.64	0.114	22*8*35	0.22

TABLE X
PERFORMANCE OF ALL SHA-256 PARAMETER SETS FOR AP

Parameter Sets	Speedup			Latency		
	KG	Sign	Verify	KG	Sign	Verify
128s-simple	3,482	5,106	6.49	0.13	0.67	0.50
128f-simple	107	726	8.62	0.08	0.24	1.12
192s-simple	3,382	4,717	6.51	0.22	1.58	0.79
192f-simple	114	762	8.77	0.10	0.45	1.66
256s-simple	1,802	2,671	7.50	0.24	2.65	1.13
256f-simple	168	1,058	8.39	0.16	0.70	1.94

Table X shows the speedup and latency of algorithmic parallelism for SHA-256 parameter sets. Signature verification achieves a much lower speedup than keypair and signature generation. This is because signature verification has lower parallelism than the other two operations. The speedup of keypair generation and signature generation for parameter sets with f is much lower than that with s. This is because the parameter sets with f have less parallelism than those with s. The speedup of keypair generation is higher than that of signature generation for the parameter sets with s, while lower for the parameter sets with f. This is because the internal structure of the parameter sets varies for different parallelisms.

All latencies are effectively reduced to less than 3 ms. The maximum elapsed times of KG, Sign and Verify are 0.24 ms, 2.65 ms and 1.94 ms, respectively. The following patterns can be observed. The elapsed time of KG of parameter sets with f is shorter than that with s, while the elapsed time of Sign and Verify are larger. The latency increases as the output bits of the hash function increase.

D. Tests of Data Parallelism

We believe that latency does not need to be considered in the case of data parallelism. Therefore, CUSPX uses a multi-streaming scheme to accelerate data parallelism. Fig. 8 shows the throughput of signature verification. We test the scenario

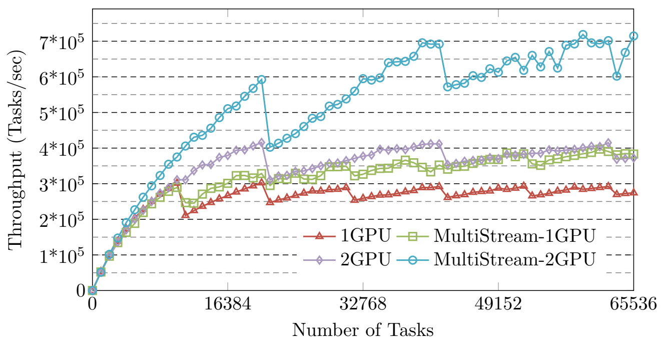

Fig. 8. Throughput of SPHINCS$^+$-SHA-256-128f-simple for data parallelism.

TABLE XI
PERFORMANCE OF DIFFERENT NUMBER OF TASKS FOR HYBRID PARALLELISM WITH SPHINCS$^+$-SHA-256-128F-SIMPLE

Tasks	Speedup			Latency			Throughput		
	KG	Sign	Verify	KG	Sign	Verify	KG	Sign	Verify
32	74.3	152.0	14.6	0.09	1.54	1.29	337,561	20,733	24,847
64	56.8	105.1	14.1	0.12	2.24	1.34	511,123	28,442	47,878
128	40.8	67.9	13.1	0.17	3.45	1.44	734,143	36,755	88,719
256	27.7	42.0	11.1	0.26	5.64	1.75	993,442	45,209	147,618
512	16.1	21.7	7.3	0.44	10.87	2.66	1,151,348	46,587	190,963
1,024	8.9	10.8	4.5	0.80	21.97	4.49	1,271,577	46,137	228,552
2,048	4.7	5.4	2.8	1.52	44.66	7.66	1,337,560	45,626	266,459

with and without the use of multiple streams in the case of using one GPU and two GPUs on a single node. The reason for the sawtooth-shape results is due to the changing number of streams. The results show that when the number of tasks is 41,984, the throughput almost reaches saturation. At this point, the throughput of 2 GPUs with multi-stream is approximately 2.37 times that of a single GPU without multi-stream.

E. Tests of Hybrid Parallelism

Hybrid parallelism uses multiple groups to process multiple $taks$ one-to-one. A group with a single thread is data parallelism, while a group with multiple threads is hybrid parallelism. We only show the case of multiple keypairs, as the case of a single-keypair is similar.

Table XI shows the performance of different tasks for Hybrid Parallelism, where the speedup is compared to data parallelism with an intra-group parallelism of 1. When the number of tasks is low, there is a high parallelism per task, so the speedup is quite significant. As the number of tasks increases, constrained by the GPU's parallelism, the speedup per task decreases. When the number of tasks reaches 512, the throughput barely changes, indicating it has reached the peak performance of CUSPX on the GPU.

TABLE XII
PERFORMANCE OF ALL SHA-256 PARAMETER SETS FOR HP

Parameter Sets	Tasks	Latency (ms)			Throughput (tasks/s)		
		KG	Sign	Verify	KG	Sign	Verify
128f-simple	512	0.44	10.87	2.68	1,175,716	47,106	190,988
192f-simple	512	0.77	22.22	5.53	662,638	23,038	92,620
256f-simple	512	2.84	59.55	7.44	180,158	8,598	68,840
128s-simple	128	6.10	47.58	0.65	20,968	2,690	196,360
192s-simple	128	10.42	120.32	1.24	12,280	1,064	103,128
256s-simple	128	10.82	184.98	2.14	11,830	692	59,791

TABLE XIII
PROFILING RESULTS OF SPHINCS$^+$-SHA-256-128F-SIMPLE

	Throughput		Pipe Utilization		Occupancy		Registers
	Compute	Memory	ALU	LSU	Theoretical	Achieved	
KG	78.37%	3.51%	79.76%	3.07%	33.33%	16.40%	72
Sign	78.85%	19.25%	79.22%	9.55%	33.33%	15.91%	96
Verify	61.56%	12.74%	61.61%	11.13%	33.33%	7.31%	128

Table XII shows the performance of hybrid parallelism for all SHA-256 parameter sets with tasks = 512. We selected an appropriate number of tasks to ensure that the longest task running time does not exceed 0.2 seconds. Overall, the performance of KG and Sign is better for the “f” data set, while the performance of Verify is slightly worse.

Table XIII shows the profiling results of SPHINCS$^+$-SHA-256-128f-simple produced using Nsight Compute. Throughput reports the achieved percentage of utilization concerning the theoretical maximum, reflecting that CUSPX's compute is more heavily utilised than its Memory. Pipeline Utilization of the Arithmetic Logic Unit (ALU) and Load Store Unit (LSU) reflects the utilization rate of computing and memory instruction pipelines. Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. Large discrepancies between the theoretical and the achieved occupancy during execution typically indicate highly imbalanced workloads. CUSPX experiences a significant reduction in parallel scale during execution, resulting in these large discrepancies.

F. Comparison with Related Works

CUSPX is the first to accelerate SPHINCS$^+$ on the GPU, thus we mainly compare works on other platforms. In most scenarios, the parameter sets of SPHINCS$^+$-128s-simple and SPHINCS$^+$-128f-simple provide sufficient security. We use these to compare performance. Table XIV shows the performance comparison with related works on a single device. For each work, we only list the performance using the best-performing hash function in their paper. For CUSPX, AP, DP and HP represent the algorithmic, data and hybrid parallelism. The throughput of data parallelism for CUSPX is given as 100% to represent the throughput of other work.

The works on FPGA [13], [14] / CPUs [9], [10], [12] usually have lower latency than that on GPUs because GPUs have poor single-core performance. Therefore, they are suitable for applications that require low latency. The best performance of related work so far is achieved by [13]. We use algorithmic parallelism to reduce latency, achieving a speedup of 12.40 / 0.67 = 18.50$\times$ for signature generation with SPHINCS$^+$-128s-simple parameter sets compared to [13]. However, performance of CUSPX verifying is lower due to the limitation of single-core performance and parallelism.

Data parallelism and hybrid parallelism effectively utilise the high parallelism of the GPU. All data parallelism experiments use 41,984 $tasks$ with four streams. The number of $tasks$ for SPHINCS$^+$-128s-simple and SPHINCS$^+$-128f-simple in hybrid parallelism are 128 and 512, respectively.

TABLE XIV
PERFORMANCE COMPARISON WITH RELATED WORK ON A SINGLE DEVICE FOR SPHINCS$^+$-128S/128F-SIMPLE

Parameter Sets	Hash Function, Year [Work]	Latency (ms)			Throughput (tasks/sec)			Device
		KG	Sign	Verify	KG	Sign	Verify	
128s-simple	Haraka, 2020 [9]	9.70	77.67	0.10	103 (0.4%)	13 (0.5%)	10,040 (1%)	Xeon E3-1220
	SHAKE256, 2020 [13]	-	12.40	0.07	-	81 (3%)	14,286 (1%)	Artix-7
	SHA-256, 2021 [10]	11.68	89.00	0.21	86 (0.4%)	11 (0.4%)	4,763 (0.5%)	Xeon Silver 4215R
	SHA-256, 2021 [14]	65.77	985.16	1.12	15 (0.1%)	1 (0.0%)	893 (0.1%)	XILINX XZU3EG
	SHA-256 (AP, CUSPX)	0.13	0.67	0.50	7,692 (33%)	1,493 (55%)	2,000 (0.2%)	RTX 3090
	SHA-256 (DP, CUSPX, 41,984 tasks)	1,815.85	15,423.30	40.54	23,120 (100%)	2,722 (100%)	1,035,619 (100%)	RTX 3090
	SHA-256 (HP, CUSPX, 128 tasks)	6.1	47.58	0.65	20,984 (91%)	2,690 (99%)	196,923 (19%)	RTX 3090 (19%)
128f-simple	Haraka, 2020 [9]	0.16	3.93	0.26	6,296 (0.5%)	254 (0.6%)	3,876 (1%)	Xeon E3-1220
	SHAKE256, 2020 [13]	-	1.01	0.16	-	990 (2%)	6,250 (2%)	Artix-7
	SHA-256, 2021 [10]	0.23	5.39	0.51	4,371 (0.3%)	186 (0.4%)	1,972 (0.6%)	Xeon Silver 4215R
	SHA-256, 2021 [14]	2.02	64.34	2.51	495 (0.0%)	16 (0.0%)	398 (0.1%)	XILINX XZU3EG
	SHA-256, 2022 [12]	0.12	2.73	0.20	8,333 (0.6%)	366 (0.8%)	5,000 (1%)	Core i7-1185G7
	SHA-256 (2024, [24]*, 512 tasks)	0.71	11.53	1.79	725,118 (55%)	44,391 (97%)	285,681 (81%)	RTX 3090
	SHA-256 (HP, CUSPX*, 512 tasks)	0.43	10.34	0.80	1,116,066 (90%)	49,531 (109%)	642,881 (182%)	RTX 3090
	SHA-256 (AP, CUSPX)	0.08	0.24	1.12	12,500 (1%)	4,167 (9%)	893 (0.3%)	RTX 3090
	SHA-256 (DP, CUSPX, 41,984 tasks)	32.07	924.24	119.16	1,309,136 (100%)	45,425 (100%)	352,333 (100%)	RTX 3090
	SHA-256 (HP, CUSPX, 512 tasks)	0.44	10.87	2.68	1,163,636 (89%)	47,102 (104%)	191,045 (54%)	RTX 3090
ChaCha	2024, SPHINCS, DP [23]	126.00	616.00	119.00	129,897 (10%)	6,651 (15%)	137,435 (39%)	TITAN Xp
level 2	2024, Dilithium, HP [41]	–	–	–	1,418,283 (108%)	562,534 (1238%)	1,484,420 (421%)	RTX 3090 Ti
Falcon-512	2024, Falcon, DP [40]	–	587.07	8.56	–	27,908 (61%)	1,913,380 (543%)	RTX 3080
SHA2_20/2	2022, XMSS, SDP [22]	3.20	52.61	39.47	313 (0.0%)	311,424 (686%)	415,100 (118%)	RTX 3090
H20(10,10)	2024, LMS, SDP [21]	1.11	20.86	7.40	9,225,225 (705%)	490,892 (1081%)	1,383,784 (393%)	RTX 3090

If users can tolerate tolerable latency, our work can offer performance that is significantly better than other works.

For Kim et al. [24], we read their code and found that their timing standards differ from those in our paper. Specifically, we report both kernel time and end-to-end time. However, their method does not include the time taken for data transfer to the GPU and includes only the parallelizable parts, which overly inflated execution efficiency. To ensure fairness, we test the data using their timing method (marked as *). For 128f, 192f and 256f, CUSPX has KG throughput of 1177,993, 639,849 and 181,556, Sign throughput of 49,532, 24,262 and 9,427, and verify throughput are 642,882, 315,804 and 226,324, respectively. Kim et al. exhibits KG performances of 725,118, 305,881 and 100,168, sign performances of 44,391, 24,997 and 11,401, and verify performances of 285,681, 155,803 and 106,282, respectively. CUSPX only shows poorer performance in specific signature generation scenarios due to the high computational load. CUSPX uses global memory with global synchronization, while Kim et al. employs shared memory and block-level synchronization, resulting in lower computational overhead. For better signing performance, one could consider adopting Kim et al.'s implementation, which is the hybrid parallel scheme 1 of the Section IV. However, for overall improved performance and versatility, we recommend CUSPX's method.

Finally, we show the performance of PQC DSA on GPU. We used the data corresponding to the highest performance parameter set in the paper. Among them, the work of Dilithium is shown using 10 streams. If this data is used to calculate the delay, it is quite unfair and will lead to excessive delay, so it is not shown. The parallel method of XMSS and LMS needs to distinguish the number of keypairs. We have shown the case of single-keypair data parallelism (SDP), which has higher performance. We compared the performance with CPSPX(DP)'s 128f-simple parameter set. The results show that the performance of KG, Sign, and Verify ranks third, fourth and fifth, respectively.

VI. CONCLUSION

In this work, we propose CUSPX, an efficient GPU implementation of SPHINCS$^+$. Our design includes three parallel schemes: algorithmic, data and hybrid parallelism. For these parallel schemes, we propose a three-level parallel structure and some load-balancing schemes to run the SPHINCS$^+$ algorithms with high parallelism and high parallel efficiency. The results show that our work outperforms other work. We plan to conduct targeted parallel design for CPU in the future.

REFERENCES

[1] D. A. Cooper et al., "Recommendation for stateful hash-based signature schemes," *NIST Special Pub.*, vol. 800, 2020, Art. no. 208.

[2] NIST, "Report on post-quantum cryptography," *NISTIR*, vol. 8105, 2016. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/ir/2016/nist.ir.8105.pdf

[3] NIST, "Stateful hash-based signatures," 2022. [Online]. Available: https://csrc.nist.gov/projects/stateful-hash-based-signatures

[4] D. A. McGrew, P. Kampanakis, S. R. Fluhrer, S. Gazdag, D. Butin, and J. Buchmann, "State management for hash-based signatures," in *Proc. 3rd Int. Conf. Res. Secur. Standardisation*, 2016, pp. 244–260.

[5] 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.*, 2019, pp. 2129–2146.

[6] NIST, "Stateless Hash-Based Digital Signature Standard," *FIPS*, vol. 205, 2023. [Online]. Available: https://doi.org/10.6028/NIST.FIPS.205.ipd

[7] NIST, "Status report on the third round of the NIST post-quantum cryptography standardization process," *NISTIR*, vol. 8413, 2022. [Online]. Available: https://doi.org/10.6028/NIST.IR.8413

[8] NIST, "The beginning of the end: The first NIST PQC standards," 2022. [Online]. Available: https://csrc.nist.gov/Presentations/2022/the-beginning-of-the-end-the-first-nist-pqc-standa

[9] A Hülsing, "SPHINCS$^+$ Submission to the NIST post-quantum project, v.3," 2020. [Online]. Available: https://sphincs.org/data/sphincs+-round3-specification.pdf

[10] D. M. Alter, P. Schwabe, and J. Daemen, "Optimizing the NIST post quantum candidate SPHINCS$^+$ using AVX-512," 2021. [Online]. Available: https://www.cs.ru.nl/bachelors-theses/2021/Dor_Mariel_Alter___1027021___Optimizing_the_NIST_Post_Quantum_Candidate_SPHINCS+_using_AVX-512.pdf

[11] H. Becker and M. J. Kannwischer, "Hybrid scalar/vector implementations of Keccak and SPHINCS$^+$ on AArch64," in *Proc. 23rd International Conference on Cryptology in India*, 2022, pp. 272–293.

[12] T. Hanson, Q. Wang, S. Ghosh, F. Virdia, A. Reinders, and M. R. Sastry, "Optimization for SPHINCS$^+$ using intel secure hash algorithm extensions," *IACR Cryptol. ePrint Arch.*, 2022, Art. no. 1726.

[13] D. Amiet, L. Leuenberger, A. Curiger, and P. Zbinden, "Fpga-based SPHINCS$^+$ implementations: Mind the glitch," in *Proc. 23rd Euromicro Conference on Digital System Design*, 2020, pp. 229–237.

[14] 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,* 2021, pp. 180–187.

[15] K. Kyung-Hoon, "Alibaba is slipping despite a record-setting singles day (BABA)," 2017. [Online]. Available: https://finance.yahoo.com/news/alibaba-slipping-despite-record-setting-155200717.html

[16] V. Varadharajan and U. Tupakula, "Security as a service model for cloud environment," *IEEE Trans. Netw. Service Manag.*, vol. 11, no. 1, pp. 60–75, 2014.

[17] A. A. Yavuz, A. Mudgerikar, A. Singla, I. Papapanagiotou and E. Bertino, "Real-Time Digital Signatures for Time-Critical Networks," *IEEE Trans. Inf. Forensics Secur.*, vol. 12, no. 11, pp. 2627–2639, 2017.

[18] A. S. Kittur and A. R. Pais, "Batch verification of Digital Signatures: Approaches and challenges," *J. Inf. Secur. Appl.*, vol. 37, pp. 15–27, 2017.

[19] H. Isah, T. Abughofa, S. Mahfuz, D. Ajerla, F. H. Z. and S. Khan, "A Survey of Distributed Data Stream Processing Frameworks," *IEEE Access*, vol. 7, pp. 154300–154316, 2019.

[20] 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, 2023.

[21] 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.*, 2024. [Online]. Available: https://doi.org/10.1145/3659209

[22] 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, 2023.

[23] S. Sun, R. Zhang, and H. Ma, "Efficient parallelism of postquantum signature scheme SPHINCS," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 11, pp. 2542–2555, 2020.

[24] D. Kim, H. Choi, and S. C. Seo, "Parallel implementation of SPHINCS+ with GPUs," *IEEE Trans. Circuits Syst. I Regul. Pap.*, vol. 71, no. 6, pp. 2810–2823, 2024.

[25] NIST, "Call for additional digital signature schemes for the post-quantum cryptography standardization process," 2022. [Online]. Available: https://csrc.nist.gov/csrc/media/Projects/pqc-dig-sig/documents/call-for-proposals-dig-sig-sept-2022.pdf

[26] D. J. Bernstein et al., "SPHINCS: practical stateless hash-based signatures," in *Proc. 34th Annu. Int. Conf. Theory Appl. Cryptogr. Techn.*, 2015, pp. 368–397.

[27] D. Amiet, A. Curiger, and P. Zbinden, "Fpga-based accelerator for post-quantum signature scheme SPHINCS-256," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2018, no. 1, pp. 18–39, 2018.

[28] Andreas Hülsing, Joost Rijneveld, and Peter Schwabe, "Armed SPHINCS - computing a 41 KB signature in 16 KB of RAM," in *Proc. 19th IACR Int. Conf. Pract. Theory Public-Key Cryptogr.*, 2016, pp. 446–470.

[29] A. K. D. S. de Oliveira and J. C. López-Hernández, "An efficient software implementation of the hash-based signature scheme MSS and its variants," in *Proc. 4th Int. Conf. Cryptol. Inf. Secur.*, 2015, pp. 366–383.

[30] A. K. D. De Oliveira, J López, and R. Cabral, "High performance of hash-based signature schemes," *Int. J. Adv. Comput. Sci. Appl.*, vol. 8, no. 3, pp. 421–432, 2017.

[31] M. Sim et al., "K-XMSS and K-SPHINCS$^+$: Hash based signatures with Korean cryptography algorithms," *IACR Cryptol. ePrint Arch.*, 2022, Art. no. 152.

[32] A. Faz-Hernández, J. C. López-Hernández, and A. K. D. S. de Oliveira, "SoK: A performance evaluation of cryptographic instruction sets on modern architectures," in *Proc. 5th ACM ASIA Public-Key Cryptography Workshop*, 2018, pp. 9–18.

[33] F. Pauls, R. Wittig, and G. P. Fettweis, "A latency-optimized hash based digital signature accelerator for the tactile internet," in *Proc. 19th Int. Conf. Embedded Comput. Syst.: Architect. Model. Simul.*, 2019, pp. 93–106.

[34] W. Wang et al., "XMSS and embedded systems," in *Proc. 26th Int. Conf. Sel. Areas Cryptogr.*, 2019, pp. 523–550.

[35] S. Ghosh, R. Misoczki, and M. R. Sastry, "Lightweight post-quantum-secure digital signature approach for IoT motes," *IACR Cryptol. ePrint Arch.*, 2019, Art. no. 122.

[36] J. P. Thoma and T. Guneysu, "A configurable hardware implementation of XMSS," *IACR Cryptol. ePrint Arch.*, 2021, Art. no. 352.

[37] Y. Cao et al., "An efficient full hardware implementation of extended Merkle signature scheme," *IEEE Trans. Circuits Syst. I Regul. Pap.*, vol. 69, no. 2, pp. 682–693, 2022.

[38] P. Mohan, W. Wang, B. Jungk, R. Niederhagen, J. Szefer, and K. Mai, "ASIC accelerator in 28 nm for the post-quantum digital signature scheme XMSS," in *Proc. IEEE 38th Int. Conf. Comput. Des.*, 2020, pp. 656–662.

[39] O. Hajihassani, S. K. Monfared, S. H. Khasteh, and S. Gorgin, "Fast AES implementation: A high-throughput bitsliced approach," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 10, pp. 2211–2222, 2019.

[40] W. K. Lee, R. K. Zhan, R. Steinfeld, A. Sakzad, and S. O. Hwang, "High Throughput Lattice-Based Signatures on GPUs: Comparing Falcon and Mitaka," in *IEEE Trans. Parallel Distrib. Syst.*, vol. 35, no. 4, pp. 675–692, 2024.

[41] S. Shen, H. Yang, W. Dai, H. Zhang, Z. Liu, and Y. Zhao, "High-Throughput GPU Implementation of Dilithium Post-Quantum Digital Signature," in *ArXiv*, vol. 2011, no. 12265, 2022.

[42] NIST, "Cryptographic algorithm validation program," 2022. [Online]. Available: https://csrc.nist.gov/projects/cryptographic-algorithm-validation-program/secure-hashing#Testing

Ziheng Wang received the BE degree in internet of things engineering from Hefei University of Technology, China, in 2018 and the ME degree in School of Computer Science Technology at Xi'an Jiaotong University, China, in 2021. He is currently working toward a Ph.D. degree in the School of Computer Science and Technology at Xi'an Jiaotong University. His major research interests include parallel computing, communication modeling and network security.

Xiaoshe Dong is currently a Professor and Ph.D. supervisor in the School of Computer Science and Technology at Xi'an Jiaotong University. His current research interests mainly include computer architecture, high-performance computing, cloud computing and parallel programming.

Heng Chen received a computer science engineering degree and a Ph.D. degree in computer science from Xi'an Jiaotong University. He is currently an associate Professor in the School of Computer Science and Technology at Xi'an Jiaotong University. His major research interests include computer architecture, high-performance computing, cloud computing and parallel programming.

Yan Kang received the BE degree in Computer Science and Technology from Xi'an Shiyou University, China, in 2021. She is currently working toward the ME degree in the School of Computer Science and Technology at Xi'an Jiaotong University. Her major research interests include high-performance computing and network security.

Qiang Wang received the MS degree in Computer Science from Xi'an Jiaotong University in 2015. He is currently a senior engineer with the Network Information Center, Xi'an Jiaotong University. His research interests include cloud computing, storage systems and parallel computing systems.