High Throughput Lattice-based Signatures on GPUs: Comparing Falcon and Mitaka

Wai-Kong Lee, *Member, IEEE,* Raymond K. Zhao, Ron Steinfeld, *Member, IEEE,* Amin Sakzad, *Member, IEEE,* and Seong Oun Hwang, *Member, IEEE*

Abstract—The US National Institute of Standards and Technology initiated a standardization process for post-quantum cryptography in 2017, with the aim of selecting key encapsulation mechanisms and signature schemes that can withstand the threat from emerging quantum computers. In 2022, Falcon was selected as one of the standard signature schemes, eventually attracting effort to optimize the implementation of Falcon on various hardware architectures for practical applications. Recently, Mitaka was proposed as an alternative to Falcon, allowing parallel execution of most of its operations. These recent advancements motivate us to develop high throughput implementations of Falcon and Mitaka signature schemes on Graphics Processing Units (GPUs), a massively parallel architecture widely available on cloud service platforms. In this paper, we propose the first parallel implementation of Falcon on various GPUs. We develop an iterative version of the sampling process in Falcon, which is also the most time-consuming Falcon operation. This allows us to implement Falcon signature generation without relying on expensive recursive function calls on GPUs. In addition, we propose a parallel random samples generation approach to accelerate the performance of Mitaka on GPUs. We evaluate our implementation techniques on state-of-the-art GPU architectures (RTX 3080, A100, T4 and V100). Experimental results show that our Falcon-512 implementation achieves $58,595$ signatures/second and $2,721,562$ verifications/second on an A100 GPU, which is $20.03\times$ and $29.51\times$ faster than the highly optimized AVX2 implementation on CPU. Our Mitaka implementation achieves $161,985$ signatures/second and $1,421,046$ verifications/second on the same GPU. Due to the adoption of a parallelizable sampling process, Mitaka signature generation enjoys $\approx 2-20\times$ higher throughput than Falcon on various GPUs. The high throughput signature generation and verification achieved by this work can be very useful in various emerging applications, including the Internet of Things.

Index Terms—Post-quantum cryptography, lattice-based cryptography, and graphics processing units (GPU).

1 Introduction

Post-quantum cryptography (PQC) is an emerging research field with the aim of developing new cryptographic schemes that are capable to withstand the threat of scalable quantum computers. In the year 2017, the National Institute of Standards and Technology (NIST) of the United States initiated a standardization process involving worldwide participation [1]. Besides security concerns, the candidates are also evaluated on their implementation efficiency to ensure the possibility of widespread deployment. After almost five years of evaluation and public discussions, NIST has selected for standardization one key-encapsulation mechanism (KEM): Kyber [2], and three signature schemes: Crystals-Dilithium [3], Falcon [4] and SPHINCS+ [5]. The standardization process is still ongoing, currently in Round 4, with the aim of standardizing more post-quantum digital signatures. Besides the NIST standardization, there is also another ongoing effort to improve post-quantum signature and KEM. For instance, Mitaka [6] is a recent work that proposed a parallelizable variant of Falcon, which is simpler and allows masked implementation. Another interesting work is Scabbard [7], a suite of more efficient variants of the NIST finalist KEM, Saber [8].

During this evaluation period, many optimized implementations of NIST PQC candidates were presented. Most of them focused on Field Programmable Gate Arrays (FPGA) [9], x86 CPU (including AVX2) [10] and ARM Cortex-M4 CPU hardware architectures [11], [12]. The above are the official platforms suggested by NIST for evaluation. On top of that, some interesting works explored other advanced hardware architectures like ARM Cortex-A CPU for high-performance embedded systems [13], and graphics processing units (GPUs) that allow massively parallel computation [14].

GPUs were originally designed to accelerate graphics and video applications, but later on, opened up for general-purpose computing. For instance, GPUs were used to accelerate deep learning [15], medical imaging [16], cryptography [17], [18] and power grid simulation [19]. It is now considered a de-facto accelerator in many cloud services [20], [21]. Due to this reason, there are also some efforts in developing high throughput implementation of NIST PQC candidates on GPUs [14], [22], [23], [24], [25]. However, among the three signature schemes selected by NIST for standardization, only Crystals-Dilithium [26] and SPHINCS+ [14] were previously implemented on GPU platforms. The possibility of parallelizing the other scheme, Falcon, on GPU platforms, remains an open research problem.

High throughput signature generation and verification is beneficial to applications that have high volume and require fast response time. For instance, IoT applications

- *Wai-Kong Lee and Seong Oun Hwang are with the Department of Computer Engineering, Gachon University, Seongnam 13120, South Korea. E-mail: waikong.lee@gmail.com, sohwang@gachon.ac.kr.*
- *Raymond K. Zhao is with CSIRO's Data61, Marsfield, Australia. E-mail: raymond.zhao@data61.csiro.au.*
- *Ron Steinfeld and Amin Sakzad are with the Department of Software Systems and Cybersecurity, Faculty of Information Technology, Monash University, Clayton 3800, Victoria, Australia. E-mail: ron.steinfeld@monash.edu, amin.sakzad@monash.edu.*

Manuscript received ; revised .

require the cloud server to handle massive data collected from sensor nodes, which involves verifying thousands of signatures generated from the sensor nodes. A GPU-accelerated solution that can provide high throughput signature generation and verification would be very useful to such applications. Another typical use case can be found in e-commerce. Consider the case of Alibaba Single's Day [27], we see around $583,000$ orders per second during its peak time, and online payment adopts digital signature to secure the payments. Assumes that it conducts two signature verifications for each transaction to verify the buyer's certificate (identity) and his/her signature on the payment, followed by one signature generation (to confirm the transaction). In merely one second, the system needs to handle up to $583,000$ signature generations and $1,166,000$ verifications. This can be a very challenging task if all the signature generation and verification tasks are to be computed using only CPU, even for a very powerful server. To achieve high throughput signature generation and verification, one can scale the implementation horizontally since each signature is independent (coarse-grain parallelism). However, such an approach may not be optimal as it does not fully exploit the resources in the GPU. Exploiting the inner parallelism of the signature can ensure more efficient use of GPU resources, which was an approach adopted by previous work [24], [25].

In this paper, we focus on optimizing the implementation of the Falcon [4] and Mitaka [6] signature schemes on GPU platforms. The signature generation in Falcon utilizes the Fast Fourier sampling (*ffSampling*) algorithm, which turns out to be the most time-consuming operation in Falcon. This is because the *ffSampling* algorithm recursively traverses the Falcon tree until it reaches the leaves. A closer look into this reveals that it is non-trivial to parallelize *ffSampling* due to the data dependency between each sample. It is also challenging to implement such a recursive function call in GPU since the overhead introduced by dynamic parallelism [28] is not negligible and the maximum recursion depth supported is only 24. In contrast, Mitaka signature generation is more parallelizable on GPU platforms. In particular, the Gaussian sampling process does not have data dependency as in the case of *ffSampling*. Besides, Fast Fourier Transform (FFT) and Number Theoretic Transform (NTT) are frequently used in both signature schemes. They are considered embarrassingly parallel algorithms, which can be parallelized on the GPU platforms easily. These analyses show that both Falcon and Mitaka are potential candidates for parallel implementation. In order to achieve high throughput signature generation/verification on GPU platforms, we proposed several implementation techniques to address the issue in *ffSampling* and optimize other operations. We also analyse and compare Falcon and Mitaka from the parallel implementation aspects. The contributions of this paper are summarized below:

1) An iterative version of *ffSampling* is presented. We develop an iterative *ffSampling* algorithm by emulating the stack management during the recursive function call. This allows us to implement the *ffSampling* on a GPU without using the costly recursive function call (through dynamic parallelism). In addition, we optimize this iterative version of *ffSampling* by carefully managing the complicated memory operations during the tree traversal. This includes the placement of the emulated stack, pseudo-random number generation (PRG), and optimized memory copy within a thread. Finally, we apply the proposed *ffSampling* implementation technique on Falcon and evaluate its performance on four state-of-the-art GPUs: RTX 3080, A100, T4, and V100.
2) The first parallel implementation of Falcon signature on GPUs is presented. Other than *ffSampling*, we also parallelize most of the operations in Falcon, including FFT, NTT, HashToPoint and polynomial addition/subtraction. A kernel-fusion technique is also proposed to combine several operations in Falcon to reduce the overhead of kernel invocations. The proposed optimized implementation of Falcon-512 signature generation achieves a throughput of $27,908$ sign/s ($1,913,380$ verify/s) and $58,595$ sign/s ($2,721,562$ verify/s) on RTX 3080 and A100 respectively. The results from A100 are $16.34\times$ and $74.58\times$ faster than the AVX2 implementation [4], for signature generation and verification, respectively.
3) The first parallel implementation of Mitaka signature on GPUs is presented. Mitaka is a parallelizable variant of Falcon, but there is no implementation of this scheme on a parallel architecture to date. To close this gap, the first parallel implementation of Mitaka signature on GPUs is presented. Mitaka signature generation and verification are highly parallelizable as most of the operations are either polynomial arithmetic (e.g., multiplication, addition, subtraction) or coefficient-wise computation, which do not have data dependencies. However, the Gaussian sampling process illustrated in the reference implementation [6] consumes the random samples in a serial manner. To allow full parallel implementation, we propose to break the Gaussian sampling into two phases. The random samples are first generated in bulk[1] through the ChaCha20 stream cipher, followed by the rejection sampling process. The proposed optimized implementation of Mitaka signature generation on GPUs achieved $74,010$ sign/s ($695,931$ verify/s) and $161,985$ sign/s ($1,421,046$ verify/s) on RTX 3080 and A100, respectively. The results from A100 are $20.03\times$ and $29.51\times$ faster than the reference implementation [6], for signature generation and verification respectively. We open-sourced our implementation to https://github.com/benlwk/Falcon-Mitaka to encourage more future research on this topic.

2 Background

2.1 Falcon: NIST Standardized Signature

Falcon [4] is a hash-and-sign post-quantum signature scheme based on the NTRU lattice problem [29]. The signature generation process of Falcon requires sampling short

1. Our implementation allow high parallelism, and it is different from the original Mitaka implementation. Note that this proposed modification does not affect the security of Mitaka.

vectors from *discrete Gaussian* distributions over lattices [30]. To accelerate this process, Falcon utilizes the Fast Fourier sampling (*ffSampling*) algorithm [4], [31]. However, *ffSampling* is particularly "delicate to implement" and listed as a main shortcoming in the Falcon specification [4]. To the best of our knowledge, no reported attempt of implementing *ffSampling* on a platform with a high degree of parallelism has been made previously.

2.2 Mitaka: a Parallelizable Variant of Falcon

Recently, Mitaka [6], a variant of the Falcon signature scheme, overcame the aforementioned pitfalls of Falcon. Mitaka replaced *ffSampling* with a "hybrid" sampling procedure [32] based on a parallelizable discrete Gaussian sampler over lattices [33]. However, the vectors generated by the hybrid sampler are longer than the ones from *ffSampling*, which affects the security of the signature scheme. To compensate for the security loss, Mitaka modified the key generation algorithm and revised the scheme parameters. Although the Mitaka is designed to be parallelizable and maskable, very little study on the implementation aspects of Mitaka has been made. The only exception is the reference implementation on an Intel CPU from the authors [6].

2.3 Overview of NVIDIA GPU Architecture and CUDA

Volta and Ampere are the two representatives of advanced GPU architectures released by NVIDIA in 2017 and 2020, respectively. These GPUs consist of thousands of cores, which are ideal for computing on massively parallel data. For instance, the RTX3080 (Ampere architecture) consists of 128 cores. CUDA is the Software Development Kit released by NVIDIA to ease the programming of GPU for general-purpose computing. Under the CUDA programming model, multiple threads are grouped into a block, where multiple blocks form a GPU grid. This relationship is illustrated in Fig. 1, where each thread and block can be indexed individually for parallel computing. NVIDIA GPUs grouped 32 threads into one warp in order to allow efficient instruction scheduling and memory access. Warp divergence occurs if threads within a warp do not execute the same path, which may have a serious performance penalty. In the subsequent presentation, we refer to *tid* as a unique ID for parallel threads within a block.

2.4 Related Works

The use of GPU in accelerating cryptography algorithms has been common in the past decade. For instance, it was used in accelerating fully homomorphic encryption [17], [34], wherein heavy computations like NTT and residue number system (RNS) are offloaded to the GPU. Lee et al. [23] exploited the tensor cores in contemporary GPU architectures. They compute the polynomial convolution of NTRU, variants of FrodoKEM and LAC, but only support non-ephemeral keypairs. A follow-up work uses dot-product instructions on GPU [24] to speed up the polynomial convolution in Saber and FrodoKEM, with support to ephemeral keypair. GPU was also used as an accelerator to create a signature server [35] based on elliptic curve cryptography (ECC).

There is limited prior work in the literature that implements Falcon [4]. Thomas Pornin presented an optimized version of Falcon on CPU utilizing AVX2 instructions [10]. In the same work, they also presented the implementation of Falcon on the Cortex-M4 microcontroller. Oder et al. [36] managed to reduce the dynamic memory consumption of Falcon by 43%, through revision of the memory layout targeting Cortex-M4 microcontroller. Besides that, there are two prior works that optimized the implementation of Falcon on more advanced processor architectures. Nguyen and Gaj [37] proposed techniques to compress the size of the twiddle-factor table and optimize the memory access pattern in FFT. They achieved a fast and memory-efficient implementation on Cortex-A72. Kim et al. [38] demonstrated techniques for parallelizing the FFT and NTT operations. They utilized the NEON instructions found in the NVIDIA Carmel with ARMv8 architecture. Recently, a hardware architecture for Falcon signature verification was proposed in [39], but the implementation does not include the signature generation. Another notable work [40] parallelized some of the inner operations in *ffSampling* algorithm by using a software/hardware co-design approach. There is no prior work reported on the implementation of the Mitaka signature, except the reference implementation presented by the authors [6].

In summary, most of these prior implementations focus on optimizing the memory and computational aspects of FFT and NTT. However, the most time-consuming part of Falcon signature generation is the *ffSampling* algorithm. It involves expensive memory operations during the tree-traversal process. In addition, there is no parallel implementation of Falcon and Mitaka on a massively parallel architecture like GPU. The previous parallel implementation techniques reported only target a small degree of parallelism supported by NEON [37], [38] and AVX [4]. However, the latter techniques are not directly applicable to the GPU architecture. Hence, we are motivated to close this gap and explore techniques to optimize the implementation of Falcon and Mitaka on GPUs with high throughput performance.

3 Proposed GPU Implementation Techniques

This section first presents the overview of GPU-based signature servers and their potential applications. This is followed by the descriptions of the proposed techniques to optimize the performance of Falcon and Mitaka on GPUs. The design choices and trade-offs made in our implementation are also presented in this section.

3.1 Overview of the Parallel Signature Generation and Verification on GPUs

There are two commonly used strategies to parallelize an algorithm on a GPU targeting server environment: coarse-grain and fine-grain [22]. Coarse-grain implementation completes the entire algorithm within one thread, which is essentially a serial implementation. The parallelism is achieved by processing many threads concurrently, wherein sufficient workload is required to fully exploit the computing power in GPU. On the other hand, the fine-grain

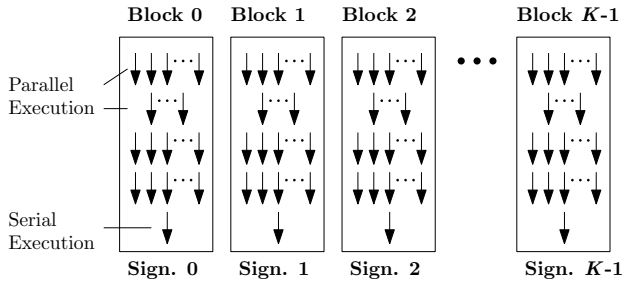


Fig. 1: GPU-based signature server: parallelizing the implementation of signatures on a GPU.

approach assigns several threads to compute one algorithm in parallel. This allows us to fully harness the GPU even though the workload is relatively low. The coarse-grain approach relies on a large amount of workload to achieve a high throughput performance, but this may not be always attainable. Moreover, the coarse-grain approach can also be slower as it requires a lot of memory to hold a large amount of workload. It also has high latency, which is not desirable for applications that require fast response time. Fine-grain approach has low latency but the throughput performance is rather low. This relationship was also observed in prior work that implemented the post-quantum KEMs [22].

Taking these into consideration, we took an intermediate approach. Referring to Fig. 1, we compute one signature (generation or verification) on a block in a fine-grain approach and instantiate K blocks to process K signatures concurrently. This can be viewed as a combination of coarse-grain (processing multiple signatures) and fine-grain (computing one signature with multiple threads) parallelism. It allows us to achieve a balance between throughput and latency. Within each block, several GPU kernels were developed to compute all the signing or verification processes using different numbers of parallel threads. For instance, polynomial arithmetic can be parallelized with a high number of threads, but the hash function that has a smaller degree of parallelism uses a smaller number of threads. This approach was also observed in other GPU implementations of post-quantum cryptography [23], [24], [25].

3.2 Parallelizing Falcon Signature

Falcon [4] allows the pre-expanded LDL tree as part of the private key to be used in the signature generation. However, in this paper, we have implemented a more generic version. Our version rebuilds the LDL tree dynamically for every signature generation, following closely the reference implementation. Table 1 shows the breakdown of major computational steps in Falcon signature generation and verification. This was evaluated on an Intel i9-10900K CPU, based on the reference implementation submitted to NIST. Referring to the Table, *ffSampling* and FFT/IFFT are the most time-consuming operations in Falcon, accounting for more than 70% of the entire signature generation process. This shows that parallelizing and optimizing these two operations can greatly improve the performance of Falcon. *ffSampling* is a memory-bound algorithm as it traverse the entire Falcon tree in a serial manner. The FFT/IFFT algorithm is considered balanced. This is because it combines arithmetic

TABLE 1: Breakdown of the Execution Time of Major Computational Steps in Falcon. [1]

Functions	Percentage (%)			
	Falcon-512		Falcon-1024	
	Sign	Verify	Sign	Verify
FFT/IFFT	6.39	–	5.29	–
FP64 polynomial arithmetic [2] (add, sub, negate, etc.)	6.20	3.44	6.09	3.41
ffSampling	65.78	–	65.66	
HashToPoint[2]	3.98	30.81	4.28	30.81
Encode	1.86	–	1.92	–
Decode	–	16.45	–	16.46
NTT/INTT [2]	6.89	46.49	5.18	46.50
Others	8.95	2.81	8.91	2.82

[1] Serial reference implementation [4] evaluated on an Intel i9-10900K CPU.
[2] The same function is used in both sign and verify.

(predominantly floating point multiply and add) and memory (load/store inputs/outputs and reading twiddle factors) operations.

We also focus on optimizing NTT/INTT which accounted for almost half of the execution time in signature verification. Similar to FFT/IFFT, NTT/INTT algorithm is also considered balanced, but it operates on the integer domain. HashToPoint algorithm is used in both signature generation and verification; it is inherently a serial process, and the parallelism is limited to the hash function (i.e., SHAKE) only. Hence, we exploit the inner parallelism of the hash function in our implementation to achieve a lower latency HashToPoint. The polynomial arithmetic is lightweight in computation and easily parallelizable. The encode/decode function is not parallelizable, lightweight and memory-bound. Note that the NTT/INTT, FP64 polynomial arithmetic and HashToPoint functions are used in both signature generation and verification. However, the input/output size and frequency of invocation are different. For example, signing requires three NTTs while verification requires only two.

3.2.1 Original Recursive ffSampling Algorithm

The original *ffSampling* algorithm was implemented in a recursive manner [4]; it is reproduced in Algorithm 1. It is given an input vector t, parameter σ, and Gram matrix G associated with matrix B. The output of the algorithm is a vector z such that $(t - z)B$ is a discrete Gaussian vector with standard deviation σ. To realize the sampling procedure, the algorithm generates a Falcon tree (see Fig. 2) with the LDL decomposition (*poly_LDL_fft*, line 9 in Algorithm 1). It samples discrete Gaussian values on each leaf ($\mathrm{SamplerZ}$, lines 4, 5 in Algorithm 1). The function *poly_split_fft* computes the Gentleman-Sande inverse FFT butterflies [41], while *poly_merge_fft* computes the Cooley-Tukey FFT butterflies [42]. Functions *poly_add*, *poly_sub*, and *poly_mul_fft* compute the polynomial addition, subtraction, and pointwise multiplication, respectively.

It is challenging to implement this algorithm on a GPU, due to the following reasons:

1) Referring to Fig. 2 and Algorithm 1 lines 15 and 23, the samples in Falcon are generated by each leaf in Falcon tree in a sequential manner. There is a data

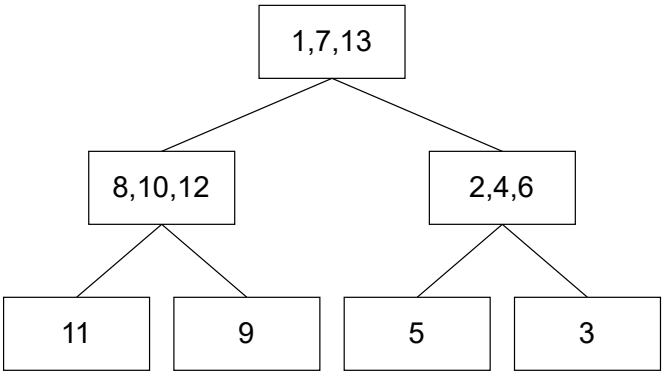


Fig. 2: A Falcon tree of height 3. Labels on the tree nodes indicate the order in which the tree nodes are traversed by the ffSampling algorithm (where 1 is visited first, 2 second, etc).

dependency between each sample in the *ffSampling* algorithm. This makes it impossible to compute multiple samples in parallel. For instance, we need to first obtain the sample on the right-most leaf (labelled as 3), traverse backward and proceed to the next leaf (labelled as 5). In other words, one cannot execute the *ffSampling* to generate multiple samples at a time. This prevents parallelizing *ffSampling*, and the only way to implement this is through a coarse-grain method, wherein one thread computes one *ffSampling* algorithm.

2) NVIDIA GPU allows recursive function calls to be implemented through the dynamic parallelism [28] feature, but it is relatively expensive. Each recursive function call needs to configure the kernel launch parameters (e.g., number of blocks and threads), introducing significant overhead. Take Falcon-512 as an example. There are $N = 512$ leaves in the LDL tree, so it takes one function call from the host (CPU) and $N \times 2 - 2 = 1022$ calls from the kernel to fully traverse the entire tree. Hence, it is very expensive to implement *ffSampling* algorithm on a GPU in a recursive manner.
3) Some operations in Algorithm 1 exhibits high level of parallelism. For instance, *poly_split_fft* (line 10) and *poly_mul_ fft* (line 19) can be executed in parallel. One way to exploit this is to run the Algorithm 1 with a single thread, and then launch another child kernel on the GPU with multiple threads to execute *poly_split_fft* and *poly_mul_ fft*. This approach also requires the use of dynamic parallelism [28] which has high overheads. Moreover, after each parallel computation in *poly_split_fft* and *poly_mul_ fft*, we need to synchronize all the child threads before returning to the parent thread and moving on to the next recursion. This kind of synchronization is also very expensive.

Note that dynamic parallelism is widely used to facilitate workload management within GPU kernels without relying on the CPU. This does not require the size of the workload to be known a priory. However, such an approach should have sufficient parallelism and low recursive depth in order to enjoy a performance benefit. Unfortunately, *ffSampling* is a serial algorithm, which means that each recursive call can only launch one thread at a time. Due to these reasons, we found that implementing *ffSampling* in recursive form could be inefficient for GPU architectures. Hence, in this paper, we have converted Algorithm 1 into its iterative version, which is detailed in Algorithm 2. With the proposed iterative *ffSampling* algorithm, we do not have to rely on dynamic parallelism, avoiding all the expensive overhead in launching a kernel within a kernel.

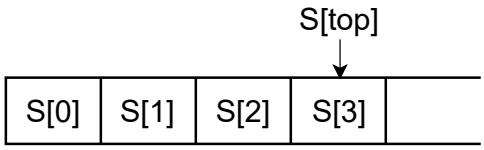


Fig. 3: A stack array S. Each array element contains a stack frame.

3.2.2 Proposed Iterative ffSampling Algorithm

We now present the proposed iterative *ffSampling* algorithm. It uses an array S to record the stack frames $(t_0, t_1, G_{00}, G_{01}, G_{11}, tmp, N, z_0, z_1)$ (see Fig. 3). The size of S is $\log_2 N+1$ since *ffSampling* is essentially a reverse depth-first tree traversal of a perfect binary tree with depth $\log_2 N$ [4]. The traversl accesses the right sub-tree first, then the left sub-tree, as illustrated in Fig. 2. During each iteration, the algorithm will access the top stack frame $S[top]$ and make updates based on its local state. The values z_0 and z_1 in a stack frame can indicate the local state of *ffSampling*. This is because z_1 is initialized before the first recursive call and z_0 is initialized before the second recursive call (after the first recursion). Recursive calls in Algorithm 1 (lines 15 and 23) become pushing new stack frames into S in Algorithm 2 (lines 20 and 29). Statements after recursions in Algorithm 1 (lines 16 and 24) are converted to the updateSt function in Algorithm 3, which calls *poly_merge_fft* to update $S[top]$ based on its local state.

Both Algorithm 1 and 2 produce the same outputs and have the same time complexity, since their only difference is the stack management. Moving stack frames to an array does not change the computational steps or the data flow of the algorithm.

3.2.3 FFT and NTT

FFT involves complex numbers, which are stored in the same array separately. A polynomial with length N is stored in an array of $2 \times N$; the first N elements are real numbers, while the next N elements are imaginary numbers. The FFT in-place implementation in Falcon is detailed in Algorithm 4. The algorithm takes the polynomial $\mathbf{a}$ in natural format and produces the results in FFT format, replacing the original data in polynomial $\mathbf{a}$. The FFT first level is skipped because the twiddle factor used is i, resulting in free operation [4]. There are two **for** loops involved in the FFT algorithm. The size of i loop doubles in every level (controlled by m in lines 8 and 21), while the size of j loop halves in every level (controlled by ht in lines 7 and 20). The twiddle factors are pre-computed into a lookup table, where

Algorithm 1 The ffSampling_dyntree algorithm: original recursive version [4].

Input: $t = (t_0, t_1)$, standard deviation σ, Gram matrix $G = \begin{pmatrix} G_{00} & G_{01} \\ G_{01}^* & G_{11} \end{pmatrix}$, buffer $tmp[0\colon 4N]$.
Output: Samples $z = (t_0, t_1)$ (in-place).

1: **function** ffSampling($t_0, t_1, G_{00}, G_{01}, G_{11}, tmp, N$)
2: **if** $N = 1$ **then**
3: $R \leftarrow \sigma/\sqrt{G_{00}[0]}$.
4: $t_0[0] \leftarrow \text{SamplerZ}(t_0[0], R)$.
5: $t_1[0] \leftarrow \text{SamplerZ}(t_1[0], R)$.
6: **return**
7: **end if**
8: $hn \leftarrow N/2$.
9: $G_{00}, tmp[0\colon N], G_{11} \leftarrow$ poly_LDL_fft(G_{00}, G_{01}, G_{11}).
10: $G_{00}[0\colon hn], G_{00}[hn\colon N] \leftarrow$ poly_split_fft(G_{00}).
11: $G_{11}[0\colon hn], G_{11}[hn\colon N] \leftarrow$ poly_split_fft(G_{11}).
12: $G_{01}[0\colon hn], G_{01}[hn\colon N] \leftarrow G_{00}[0\colon hn], G_{11}[0\colon hn]$.
13: Let z_1 be alias to $tmp[N\colon 2N]$.
14: $z_1[0\colon hn], z_1[hn\colon N] \leftarrow$ poly_split_fft(t_1).
15: ffSampling$(z_1[0\colon hn], z_1[hn\colon N], G_{11}[0\colon hn], G_{11}[hn\colon N], G_{01}[hn\colon N], tmp[2N\colon 4N], hn)$.
16: $tmp[2N\colon 3N] \leftarrow$ poly_merge_fft$(z_1[0\colon hn], z_1[hn\colon N])$.
17: $z_1 \leftarrow$ poly_sub$(t_1, tmp[2N\colon 3N])$.
18: $t_1 \leftarrow tmp[2N\colon 3N]$.
19: $tmp[0\colon N] \leftarrow$ poly_mul_fft$(tmp[0\colon N], z_1)$.
20: $t_0 \leftarrow$ poly_add$(t_0, tmp[0\colon N])$.
21: Let z_0 be alias to $tmp[0\colon N]$.
22: $z_0[0\colon hn], z_0[hn\colon N] \leftarrow$ poly_split_fft(t_0).
23: ffSampling$(z_0[0\colon hn], z_0[hn\colon N], G_{00}[0\colon hn], G_{00}[hn\colon N], G_{01}[0\colon hn], tmp[N\colon 3N], hn)$.
24: $t_0 \leftarrow$ poly_merge_fft$(z_0[0\colon hn], z_0[hn\colon N])$.
25: **end function**

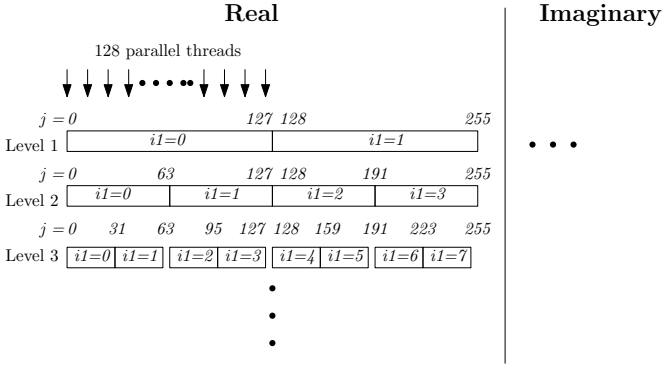


Fig. 4: Parallelizing the FFT implementation in Falcon-512

they are accessed differently at each level (lines 8 and 9). The main computation in FFT is the butterfly operation. In this operation, two coefficients from polynomial **a** are multiplied with the twiddle factors (line 17), and one of them is added up (line 18). Note that the multiplication in the complex domain is the most time-consuming, as it involves four multiplications, one subtraction and one addition.

The FFT algorithm is inherently parallel because there are always $N/2$ pairs of work items (butterfly operations)

Algorithm 2 The iterative ffSampling_dyntree algorithm.

Input: $t = (t_0, t_1)$, standard deviation σ, Gram matrix $G = \begin{pmatrix} G_{00} & G_{01} \\ G_{01}^* & G_{11} \end{pmatrix}$, buffer $tmp[0\colon 4N]$, array $S[0\colon \log_2 N{+}1]$ with tuples $(t_0, t_1, G_{00}, G_{01}, G_{11}, tmp, N, z_0, z_1)$.
Output: Samples $z = S[0].(t_0, t_1)$.

1: **function** It_ffSampling($t_0, t_1, G_{00}, G_{01}, G_{11}, tmp, N$)
2: $top \leftarrow 0$.
3: $S[0] \leftarrow (t_0, t_1, G_{00}, G_{01}, G_{11}, tmp, N,$ **null**, **null**$)$.
4: **loop**
5: Let St be alias to $S[top]$.
6: $n \leftarrow St.N, hn \leftarrow n/2$.
7: **if** $n = 1$ **then**
8: $R \leftarrow \sigma/\sqrt{St.G_{00}[0]}$.
9: $St.t_0[0] \leftarrow \text{SamplerZ}(St.t_0[0], R)$.
10: $St.t_1[0] \leftarrow \text{SamplerZ}(St.t_1[0], R)$.
11: $top \leftarrow top - 1$, updateSt$(S[top])$.
12: **else**
13: **if** $St.z_1 =$ **null then**
14: $St.(G_{00}, tmp[0\colon n], G_{11}) \leftarrow$ poly_LDL_fft$(St.(G_{00}, G_{01}, G_{11}))$.
15: $St.(G_{00}[0\colon hn], G_{00}[hn\colon n]) \leftarrow$ poly_split_fft$(St.G_{00})$.
16: $St.(G_{11}[0\colon hn], G_{11}[hn\colon n]) \leftarrow$ poly_split_fft$(St.G_{11})$.
17: $St.(G_{01}[0\colon hn], G_{01}[hn\colon n]) \leftarrow St.(G_{00}[0\colon hn], G_{11}[0\colon hn])$.
18: Let $St.z_1$ be alias to $St.tmp[n\colon 2n]$.
19: $St.(z_1[0\colon hn], z_1[hn\colon n]) \leftarrow$ poly_split_fft$(St.t_1)$.
20: $S[top{+}1] \leftarrow (St.(z_1[0\colon hn], z_1[hn\colon n], G_{11}[0\colon hn], G_{11}[hn\colon n], G_{01}[hn\colon n], tmp[2n\colon 4n]), hn,$ **null**, **null**$)$.
21: $top \leftarrow top + 1$.
22: **else if** $St.z_0 =$ **null then**
23: $St.z_1 \leftarrow$ poly_sub$(St.(t_1, tmp[2n\colon 3n]))$.
24: $St.t_1 \leftarrow St.tmp[2n\colon 3n]$.
25: $St.tmp[0\colon n] \leftarrow$ poly_mul_fft$(St.(tmp[0\colon n], z_1))$.
26: $St.t_0 \leftarrow$ poly_add$(St.(t_0, tmp[0\colon n]))$.
27: Let $St.z_0$ be alias to $St.tmp[0\colon n]$.
28: $St.(z_0[0\colon hn], z_0[hn\colon n]) \leftarrow$ poly_split_fft$(St.t_0)$.
29: $S[top{+}1] \leftarrow (St.(z_0[0\colon hn], z_0[hn\colon n], G_{00}[0\colon hn], G_{00}[hn\colon n], G_{01}[0\colon hn], tmp[n\colon 3n]), hn,$ **null**, **null**$)$.
30: $top \leftarrow top + 1$.
31: **else**
32: **if** $n = N$ **then**
33: **return**
34: **else**
35: $top \leftarrow top - 1$, updateSt$(S[top])$.
36: **end if**
37: **end if**
38: **end if**
39: **end loop**
40: **end function**

to be computed in each FFT level, and these work items are not dependent on each other. In particular, one can execute the *i* and *j* loops in parallel by assigning the correct indices. Referring to Fig. 4, the polynomial length in Falcon-512 is $N = 512$, so we can utilize 128 threads to parallelize the FFT implementation. At level 1, 128 threads can load the first portion (j =0–127) of real values in parallel, followed by the

Algorithm 3 The updateSt function.

Input: Tuple $St = (t_0, t_1, G_{00}, G_{01}, G_{11}, tmp, N, z_0, z_1)$.
1: **function** updateSt(St)
2: $n \leftarrow St.N, hn \leftarrow n/2$.
3: **if** $St.z_0 =$ **null then**
4: $St.tmp[2n{:}3n] \leftarrow$ poly_merge_fft$(St.(z_1[0{:}hn], z_1[hn{:}n]))$.
5: **else**
6: $St.t_0 \leftarrow$ poly_merge_fft$(St.(z_0[0{:}hn], z_0[hn{:}n]))$.
7: **end if**
8: **end function**

Algorithm 4 FFT in-place implementation in Falcon [4].

Input: Polynomial **a** in natural format.
Output: Polynomial **a** in FFT format.
1: **function** FFT(**a**)
2: $hn \leftarrow N/2$.
3: $t \leftarrow hn$.
4: $m \leftarrow 2$.
5: $j_1 \leftarrow 0$.
6: **for** $u = 1; u < log_2 N; u$++ **do**
7: $ht \leftarrow N/2$.
8: **for** $i = 0; i < m; i$++ **do**
9: $j_2 \leftarrow j_1 + ht$.
▷ Load twiddle factors
10: $s_{re} \leftarrow tf[((m+i)/2)]$.
11: $s_{im} \leftarrow tf[((m+i)/2)+1]$
12: **for** $j = j_1; u < j_2; j$++ **do**
13: $x_{re} \leftarrow \mathbf{a}[j]$. ▷ Load real parts
14: $y_{re} \leftarrow \mathbf{a}[j+ht]$.
15: $x_{im} \leftarrow \mathbf{a}[j+hn]$. ▷ Load imag. parts
16: $y_{im} \leftarrow \mathbf{a}[j+hn+ht]$.
▷ Butterfly operations
17: C_MUL$(y_{re}, y_{im}, x_{re}, x_{im}, s_{re}, s_{im})$.
18: C_ADD$(\mathbf{a}[j], \mathbf{a}[j+hn], x_{re}, x_{im}, y_{re}, y_{im})$.
19: **end for**
20: $t \leftarrow ht$.
21: $m \leftarrow m \times 2$.
22: $j_1 \leftarrow j_1 + t$.
23: **end for**
24: **end for**
25: **end function**

second portion (j =128–255); subsequent portions (j =128–255 and j =128–255) represents the imaginary values. In this way, we can eliminate the *i* and *j* loops, and execute the FFT in a parallel fashion. Note that the parallelism achieved by the FFT algorithm is limited to $N/4$ because we are working with complex numbers, so we need to load the real and imaginary (total four parts) numbers separately.

Due to limited parallelism offered by NEON and AVX2 instructions, previous implementation of Falcon FFT [4], [37], [38] can only parallelize part of the FFT operations. Algorithm 5 shows the fully parallel version of FFT implementation that utilizes a large degree of parallelism offered by GPU architecture to achieve high performance. We launch $N/4$ threads in parallel to compute the FFT. Firstly, the polynomials are loaded from the global memory and stored in shared memory (lines 6 – 8). $BDim$ refers

Algorithm 5 Our proposed parallel FFT in-place implementation in Falcon.

Input: Polynomial **a** in natural format.
Output: Polynomial **a** in FFT format.
1: **function** FFT(**a**)
2: $hn \leftarrow N/2$.
3: $t \leftarrow hn$.
4: $m \leftarrow 2$.
5: _shared_ sa[N]; ▷ Initialize shared memory
▷ Copy from global to shared memory
6: **for** $u = 0; u < N/BDim; u$++ **do**
7: $sa[u \times BDim + tid] \leftarrow a[bid \times N + u \times BDim + tid]$.
8: **end for**
9: **for** $u = 1; u < log_2 N; u$++ **do**
10: $i \leftarrow j_1 \leftarrow 0$.
11: $ht \leftarrow t/2$.
12: $j_2 \leftarrow j_1 + ht$.
13: $i \leftarrow tid/j_2$.
14: $j \leftarrow tid \,\%\, j_2 + (tid/j_2) \times 2 \times j_2$.
▷ Load twiddle factors
15: $s_{re} \leftarrow tf[((m+i)/2)]$.
16: $s_{im} \leftarrow tf[((m+i)/2)+1]$
17: $x_{re} \leftarrow sa[j]$. ▷ Load real parts
18: $x_{im} \leftarrow sa[j+hn]$. ▷ Load imag. parts
19: $y_{re} \leftarrow sa[j+ht]$.
20: $y_{im} \leftarrow sa[j+hn+ht]$.
▷ Butterfly operations
21: C_MUL$(y_{re}, y_{im}, x_{re}, x_{im}, s_{re}, s_{im})$.
22: C_ADD$(sa[j], sa[j+hn], x_{re}, x_{im}, y_{re}, y_{im})$.
23: $j_1 \leftarrow j_1 + t$.
24: $m \leftarrow m \times 2$.
25: $t \leftarrow ht$.
26: Synchronize threads.
27: **end for**
▷ Write back to global memory
28: **for** $u = 0; u < N/BDim; u$++ **do**
29: $a[bid \times N + u \times BDim + tid] = sa[u \times BDim + tid]$.
30: **end for**
31: **end function**

to the number of threads in a block, *tid* is the ID of each parallel thread, and *bid* is the block ID. Next, the indices to compute the *i* (line 13) and *j* loop (line 14) are computed respectively. Then, the twiddle factors and two coefficients from polynomial **a** are loaded into registers (lines 15 – 20). This is followed by the butterfly operations (lines 21 – 22). Finally, synchronization across all parallel threads (line 26) is required before proceeding to the next level. This is crucial because some threads may be executing faster than others, resulting in a race condition [43] (i.e., read before write). After the synchronization process, it proceeds to the next level until all FFT levels are completed (line 9). Finally, the stored results in shared memory are written back into the global memory (lines 28 - 30). *bid* here refers to the ID of parallel blocks launched. Note that we do not store the twiddle factors in shared memory because each factor is only accessed once. It does not bring any performance gain by caching them onto the shared memory.

The NTT is frequently used in Falcon for both signature generation and verification; it is detailed in Algorithm 6. The

Algorithm 6 NTT implementation in Falcon [4].

Input: Polynomial $\mathbf{x}$ in natural format.
Output: Polynomial $\mathbf{x}$ in NTT format.

1: **function** NTT($\mathbf{t}, T$)
2: $t \leftarrow N$;
3: **for** $m = 1; m < N; m = m \times 2$ **do**
4: $ht \leftarrow t/2$;
5: $j_1 \leftarrow 0$;
6: **for** $i = 0; i < m; i$++ **do**
▷ Load twiddle factors
7: $s \leftarrow tf[m + i]$
8: $j_2 \leftarrow j_1 + ht$;
9: **for** $j = j_1; j < j_2; j$++ **do**
▷ Butterfly operations
10: $u \leftarrow a[j]$
11: $v \leftarrow MQ\_MUL(a[j + ht], s)$
12: $a[j] = MQ\_ADD(u, v)$
13: $a[j + ht] = MQ\_SUB(u, v)$
14: **end for**
15: $j_1 \leftarrow j_1 + t$;
16: **end for**
17: $t \leftarrow ht$;
18: **end for**
19: **end function**

computational patterns of NTT are very similar to the FFT described in Algorithm 4, except that it is operating in the integer domain (INT32) instead of FP64. Unlike FFT, the NTT algorithm has to be executed for $\log_2 N$ levels (line 3), since the first level in NTT cannot be skipped. It reads the precomputed twiddle factors (line 7) and performs the butterfly operations (lines 10 – 13) similar to the FFT. The modular multiplication (MQ_MUL in line 11) is implemented using the Montgomery algorithm [4].

The NTT can be parallelized similarly to FFT. Unlike FFT which deals with complex numbers, NTT can achieve more parallelism (bound by $N/2$) because it does not contain the imaginary part. Referring to Algorithm 7, the polynomial **a** is first loaded onto the shared memory (lines 6 – 8). The *i* and *j* loops in Algorithm 6 are parallelized in the same way as described in Fig. 4. The twiddle factors are loaded (line 9) and the index (*j*) for accessing the polynomial (line 10) is calculated. It is followed by the butterfly operations (lines 13 – 16) and synchronization across all parallel threads. After completing all NTT levels, the results are copied from shared memory to the global memory (lines 19 – 21).

Algorithm 7 Parallel implementation of NTT on a GPU.

Input: Polynomial $\mathbf{x}$ in natural format.
Output: Polynomial $\mathbf{x}$ in NTT format.

1: **function** NTT($\mathbf{t}, T$)
2: _shared_ sa[N]; ▷ Initialize shared memory
▷ Copy from global to shared memory
3: **for** $u = 0; u < N/BDim; u$++ **do**
4: $sa[u \times BDim + tid] = a[bid \times N + u \times BDim + tid]$.
5: **end for**
6: $t \leftarrow N$;
7: **for** $m = 1; u < N; m = m \times 2$ **do**
8: $ht \leftarrow t/2$;
9: $s \leftarrow tf[m + tid/ht]$
10: $j \leftarrow tid \,\%\, ht + (tid/ht) \times t$;
▷ Load twiddle factors
11: $s \leftarrow tf[m + i]$
12: $j \leftarrow tid \,\%\, ht + (tid/ht) \times t$;
▷ Butterfly operations
13: $u \leftarrow sa[j]$
14: $v \leftarrow MQ\_MUL(sa[j + ht], s)$
15: $sa[j] = MQ\_ADD(u, v)$
16: $sa[j + ht] = MQ\_SUB(u, v)$
17: Synchronize threads.
18: **end for**
▷ Write back to global memory
19: **for** $u = 0; u < N/BDim; u$++ **do**
20: $a[bid \times N + u \times BDim + tid] = sa[u \times BDim + tid]$.
21: **end for**
22: **end function**

Algorithm 8 HashToPoint in Falcon [4].

Input: A string str, a modulus $q \leq 2^{16}$, polynomial length N.
Output: A polynomial $c = \sum_{n-1}^{i=0} c_i x^i$ in $\mathbb{Z}_q[x]$.

1: **function** HashToPoint(str, q, N)
2: $k \leftarrow \lfloor 2^{16}/q \rfloor$
3: $ctx \leftarrow$ SHAKE-256-Init()
4: SHAKE-256-Inject(ctx, str)
5: $i \leftarrow 0$
6: **while** $i < N$ **do**
7: $t \leftarrow$SHAKE-256-Extract(ctx, 16)
8: **if** $t < kq$ **then**
9: $c_i \leftarrow t \,\%\, q$
10: $i \leftarrow i + 1$
11: **end if**
12: **end while**
13: **end function**

3.3 HashToPoint

Algorithm 8 shows the HashToPoint algorithm in Falcon, which utilizes the SHAKE-256 as an extendable-output hash function (XOF). It is found that the **while** loop (lines 6 – 12) is a serial process because the coefficients in output polynomial c_i are extracted from SHAKE-256 serially. In other words, it is not possible to parallelize this **while** loop to have any performance gain. However, a closer look into SHAKE-256 reveals that the inner states can be parallelized by using 25 parallel threads. This is an approach exploited to implement SHA3 in a fine-grain parallel manner, which was first proposed by Lee et. al. [44]. In this paper, we adopted the same idea to parallelize SHAKE-256 operations (lines 3, 4 and 7) on the GPU; the **while** loop remains in serial execution.

3.3.1 Other Operations

Polynomial arithmetic (e.g., addition, subtraction, negation, etc.) can be parallelized easily as they are coefficient-wise operations. We launched N threads to compute these operations in a fine-grain parallel manner. The remaining operations like encode/decode are serial processes, so we implemented them in a coarse-grain parallel manner.

TABLE 2: Breakdown of the execution time of major computational steps in Mitaka

Functions	Percentage (%)			
	Mitaka-512		Mitaka-1024	
	Sign	Verify	Sign	Verify
Hash [2]	2.85	12.83	2.86	12.83
FFT/IFFT [2]	15.69	8.95	15.63	8.91
FP64 polynomial arithmetic [2] (add, sub, negate, etc.)	22.39	74.41	22.46	74.42
Sampler: Discrete Gaussian	22.67	–	22.67	–
Sampler: Normaldist	33.87	–	33.86	–
Others	2.53	3.81	2.52	3.84

[1] Serial reference implementation [6] evaluated on an Intel i9-10900K CPU.
[2] The same function is used in both sign and verify.

3.4 Parallelizing Mitaka Signature

Table 2 shows the breakdown of major computational steps in Mitaka signature generation and verification. It was evaluated on an Intel i9-10900K CPU, based on the reference implementation from the authors [6]. The sampler in Mitaka is the most time-consuming operation; it consumes around 57% of time in signature generation. It consists of two parts: Normaldist and the discrete Gaussian sampler (sample_discrete_gauss). Normaldist is a function used in the sampler to generate a centred normal polynomial using the Box-Muller algorithm. It can be parallelized directly on a GPU as it involves only coefficient-wise operations with no data dependency between the coefficients. The discrete Gaussian sampler accounts for 22% of the entire execution time in Mitaka signature generation. Both functions in the Mitaka sampler are computationally heavy with insignificant memory operations.

Followed by this are the FFT/IFFT and FP64 arithmetic, which are coefficient-wise operations that can be parallelized. For signature verification, most of the time was spent in the FP64 arithmetic (74%). Since Mitaka was designed to be easily parallelizable, most of the operations can be directly implemented on a GPU. In this section, we focus on describing the optimization techniques to improve the performance of Mitaka implemented on a GPU. Note that the FFT/IFFT and FP64 arithmetic in Mitaka share the same implementation methodology as in Falcon [4].

3.4.1 Generating polynomial with normal distribution

Algorithm 9 shows the Box-Muller algorithm used to generate a random polynomial before passing it to the discrete Gaussian sampler. Three random vectors with *N* bytes long are generated and passed to Algorithm 9. The algorithm then generates $N/2$ random samples (*vf[i]*) from these three random vectors (i.e., *u*, *v* and). Lines 2 – 9 describe the detailed steps to perform the Box-Muller transform to obtain these random samples. Finally, the random polynomial *vec* is obtained by taking the cosine and sine values (lines 11 and 12) from the random samples. Note that Algorithm 9 only involves coefficient-wise operations, so it can be easily parallelized. We launched $N/2$ threads in our implementation, each thread computes one item in the *i* loop (line 2 and line 10), thus achieving a fine-grain parallel implementation of Algorithm 9.

Algorithm 9 Generate Normal Distribution Random Polynomial: Box-Muller.

Input: Three random vectors *u*, *v* and *e* of *N* bytes.
Output: A centered normal polynomial vec.

1: **function** Normdist(*u*, *v*, *e*)
2: **for** $i = 0; i < N/2; i$++ **do**
3: $uf[i] \leftarrow 2 \times PI \times (u[i]\&\texttt{0x1FFFFFFFFFFFFF}) \times 2^{-53}$
4: $vf[i] \leftarrow 0.5 + (v[i]\&\texttt{0x1FFFFFFFFFFFFF}) \times 2^{-54}$
5: $b0 \leftarrow \mathsf{ffsll}(e[2 \times i + 1])$ ▷ Find first bit set
6: $b1 \leftarrow \mathsf{ffsll}(e[2 \times i])$
7: $\mathsf{geom}\,[i] \leftarrow CMUX(63 + b0, b1 - 1, CZ(e[2 \times i]))$
8: $vf[i] \leftarrow \sqrt{(N \times (\ln(2) \times \mathsf{geom}[i] - \log(vf[i])))}$
9: **end for**
▷ Write the results onto vec
10: **for** $i = 0; i < N/2; i$++ **do**
11: $vec[2 \times i] \leftarrow vf[i] \times \cos(uf[i])$
12: $vec[2 \times i + 1] \leftarrow vf[i] \times \sin(uf[i])$
13: **end for**
14: **end function**

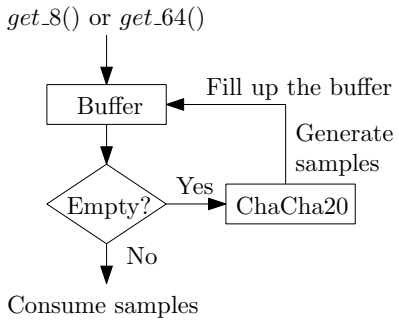


Fig. 5: Random samples generation in Mitaka

3.4.2 Batch Random Samples Generation

The discrete Gaussian sampler requires a lot of random samples, which are generated from the ChaCha20 stream cipher in the reference implementation of Mitaka. In a serial implementation presented by the authors of [6], the random samples are generated on an on-demand basis, which is not friendly to the parallel implementation. Algorithm 10 shows the serial discrete Gaussian sampler implemented by the authors of [6]. Line 3 can be parallelized as there is no data dependency between each coefficient. SamplerZ and base_sampler invokes several instances (lines 9, 16 and 25) to obtain random bytes on an on-demand basis. Note that the random samples are generated through the ChaCha20 stream cipher and stored in a buffer. Referring to Fig. 5, every time $get8()$ or $get64$ are called, it reads the remaining random samples in this buffer. If the random samples in the buffer are completely consumed, ChaCha20 will be invoked to refill it. To compute this in parallel, we can instantiate *N* structures to hold the random samples, so that *N* threads can compute line 3 in parallel. However, we need to perform housekeeping on these structures to keep track of the number of random samples consumed, which is a non-trivial overhead.

In this paper, we proposed to divide the discrete Gaussian sampler into two parts. Firstly, a large number of

Algorithm 10 Serial discrete Gaussian sampler implementation [6].

```
1: function sample_discrete_gauss(p)
2:     for i = 0; i < N; i++ do
3:         p[i] ← SamplerZ(p[i])
4:     end for
5: end function
6: function SamplerZ(u)
7:     uf ← floor(u)
8:     while (1) do
9:         entropy ← get8()                ▷ Get one random byte
10:        for i = 0; i < 8; i++ do
11:            z0 ← base_sampler()
12:            b ← (entropy ≫ i)&1
13:            z ← (2 × b − 1) × z0 + b + uf
14:            x ← (z0^2 − (z − u)^2)/(2 × R^2)
15:            p ← e^x
16:            r ← (get64()&0x1FFFFFFFFFFFFF) × 2^−53
17:            if r < p then
18:                return z
19:            end if
20:        end for
21:    end while
22: end function
23: function base_sampler()
24:    r ← get64()                         ▷ Get eight random bytes
25:    res ← 0
26:    for i = 0; i < 13; i++ do
   ▷ +1 if r ≥ CDT[i]; +0 otherwise.
27:        res = res + (r ≥ CDT[i])
28:    end for
29:    return res
30: end function
```

Algorithm 11 Batch random samples generation in Mitaka for signature generation.

Input: Buffer *dst* with $N \times 512$ bytes, encryption key *prng_k*, samples per thread N_{samp}

Output: $N \times 512$ bytes of random samples generated through ChaCha20 stream cipher.

```
1: function PRNG_batch(u, v, e)
   ▷ Each ChaCha20 encryption produces 64B.
   ▷ This is repeated for N_samp/64 times to fill up dst.
2:     for j = 0; j < N_samp/64; j++ do
   ▷ Generate the counter for encryption.
3:         cc ← bid × N × N_samp + tid × N_samp + j ∗ 8
   ▷ Load 16B of Initialization Vector (IV).
4:         for i = 0; i < 4; i++ do
5:             state[i] = CW[i]
6:         end for
   ▷ Load 32B of the key.
7:         for i = 0; i < 8; i++ do
8:             state[4 + i] = prng_k[bid × 56 + 4 × i]
9:         end for
10:        state[14] = state[14] ⊕ cc      ▷ XOR with counter
11:        state[15] = state[15] ⊕ (cc ≫ 32)
12:        Compute ChaCha20 encryption.
13:        ...                             ▷ Removed for brevity.
   ▷ Store the random samples onto global memory.
14:        for i = 0; i < 8; i++ do
15:            dst[bid × N × N_samp/4 + j × N × 8 + i × N + tid] = state[i]
16:        end for
17:    end for
18: end function
```

random samples are generated on the GPU with batch processing and stored in a large buffer on the global memory. Then, we invoke the sample_discrete_gauss in Algorithm 10. Note that this only changes the sequence of how the random samples are generated and consumed; we believe this does not create any security issues. With the proposed technique, we no longer need to generate random samples through ChaCha20 on an on-demand basis. This greatly reduces the housekeeping overhead. In particular, lines 9, 16 and 25 can obtain the pre-computed random samples directly.

From Algorithm 10, we know that each iteration of **while** loop (lines 8 – 21) consumes 129 bytes. From our experiments, we found that most of the time, this **while** loop only executes once or twice, because the chance to reject *r* (line 16) is low. In other words, the consumed random bytes are between 129 – 258 bytes. In our implementation, we instantiate *K* blocks and *N* threads, each thread generates $N_{samp} = 512$ bytes of random samples. Algorithm 11 shows the proposed batch random samples generation implemented on a GPU. Note that each ChaCha20 encryption in counter mode produces 64 bytes of random samples. We repeat this for $N_{samp}/64$ (line 2) in each thread to generate sufficient random samples. The unique counter value is generated through the thread ID (line 3, *tid*), which is a unique identifier for each thread. Since there are *K* signatures (blocks) generated, we use the block ID (*bid*) to create the unique counter for each block. Next, the initialization vector (IV), which is constant, is loaded onto the *state* buffer (line 5), followed by the encryption key (line 8). The counter value is XORed with the last 8 bytes of the *state* buffer (lines 10 – 11). The ChaCha20 stream cipher is executed to encrypt the counter value and generate 64 bytes of random samples. Note that the details of ChaCha20 encryption are not shown in Algorithm 11 for brevity reasons; it can be found in our source code shared on the public domain. Finally, the results in *state* buffer are copied to the *dst*, which will be consumed by the discrete Gaussian sampler (Algorithm 10, lines 9, 16 and 25).

3.4.3 Other Operations

We implemented the FFT/IFFT and FP64 arithmetic in a fine-grain parallel manner, following the same techniques described in Falcon (Section 3.2.3 and 3.3.1). Similarly, the hash function (SHAKE) used in Mitaka can be parallelized through the technique described in [44]. The process similar to the HashToPoint algorithm in Falcon is also used to hash the input message; we only parallelize SHAKE and leave the hash process executed in serial. The remaining operations like encode/decode are implemented in a coarse-grain parallel manner.

TABLE 3: Micro-benchmark of the proposed kernel fusion technique.

Falcon: Sign	Invocation	Total Time (μs)	Remarks
mq_NTT	3	56.07	NTT
mq_poly_tomonty	1	5.63	Convert to Montgomery representation
mq_poly_montymul_ntt	1	7.74	Point-wise multiplication for two polynomials
mq_conv_small	3	6.27	Reduce a small signed integer modulo q
mq_div_12289	1	12.35	q = 12289 Divide x by y modulo
mq_iNTT	1	19.01	INTT
recompute_G	1	6.40	Recompute the private key G
Total		**113.47**	
complete_private_comb	1	**57.28**	Kernel fusion: combined these kernels into one.
Falcon: Verify			
mq_NTT	1	18.69	NTT
reduce_s2	1	6.50	Reduce s2 elements modulo q
mq_poly_montymul_ntt	1	7.74	Point-wise multiplication for two polynomials
mq_iNTT	1	19.01	INTT
mq_poly_sub	1	7.42	Point-wise subtraction for two polynomials
norm_s2	1	6.40	Normalize -s1 elements into the [-q/2..q/2] range.
Total		**65.76**	
comb_all_kernels	1	**35.31**	Kernel fusion: combined these kernels into one.
Mitaka: Sign	Invocation	Total Time (us)	Remarks
normaldist	1	196.42	NTT
poly_mul_fft	1	11.55	Point-wise multiplication for two polynomials (complex domain)
Total		**207.97**	
normaldist_mul_fft	1	**199.81**	Kernel fusion: combined these two kernels into one.
poly_mul_fft	1	11.55	Point-wise multiplication for two polynomials (complex domain)
poly_add	1	11.46	Point-wise addition for two polynomials (complex domain)
Total		**23.01**	
poly_mul_fft_add	1	**15.20**	Kernel fusion: combined these two kernels into one.

3.5 Kernel Fusion

Implementation of Falcon and Mitaka requires many GPU kernels to handle different function calls. Each of the functions is implemented as a GPU kernel and called from the CPU. Each function's GPU kernel may have a different configuration (number of threads) according to the exploitable parallelism of the function. For instance, all FP64 polynomial arithmetic can use N threads, NTT/INTT can use $N/2$ threads, but FFT/IFFT may only use $N/4$ threads. However, each instantiation of the GPU kernel requires additional steps to configure the number of blocks/threads, prepare the stack memory and perform a context switch. All these steps introduce some overheads. To reduce these overheads, we proposed to fuse multiple kernels that have the same parallelism (i.e., the same number of threads) into one kernel. By performing kernel fusion, we can effectively reduce the number of kernel calls and potentially reuse some of the intermediate results.

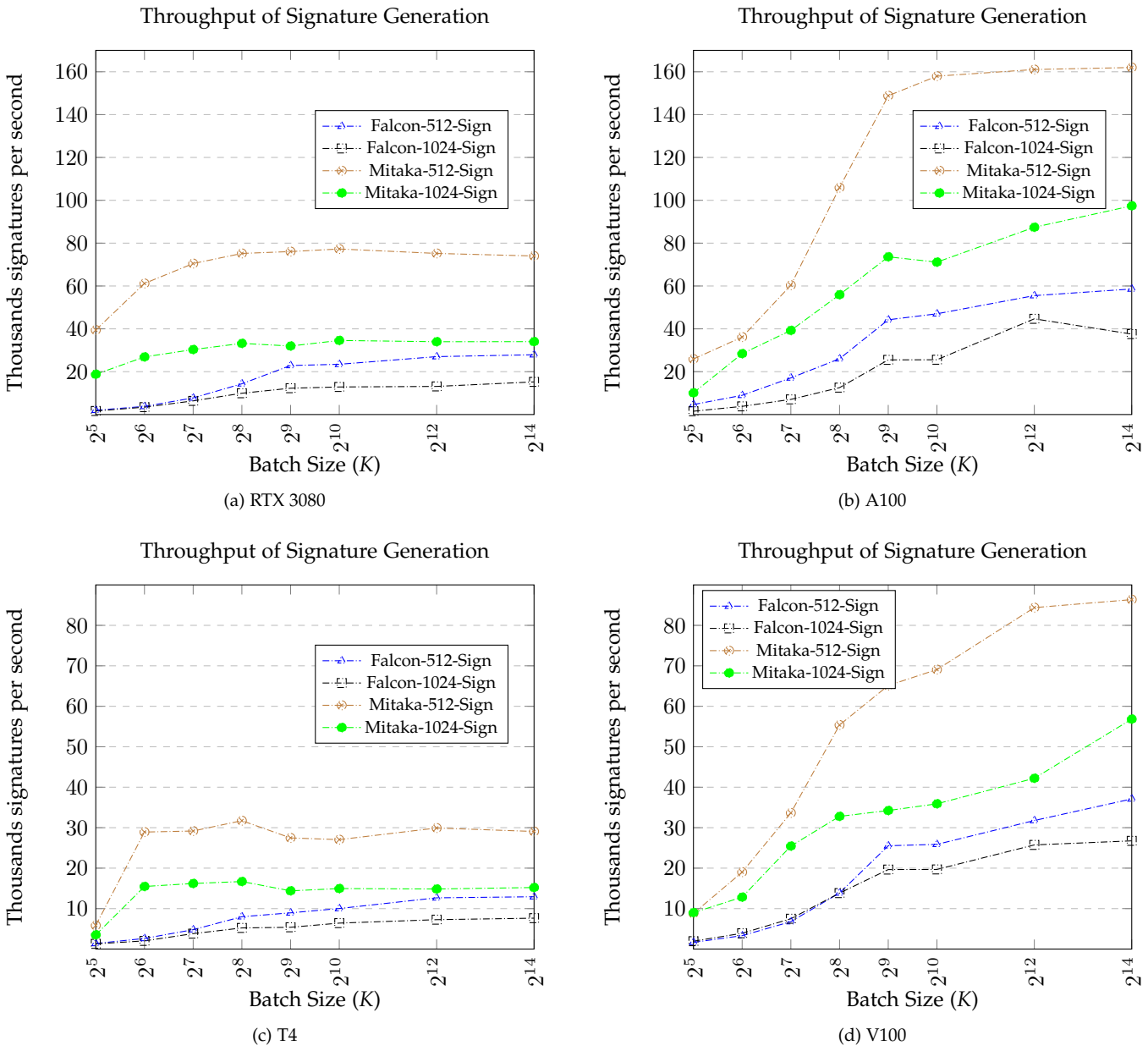
Table 3 shows the micro-benchmark of the proposed kernel fusion technique for both Falcon and Mitaka implementation. Referring to Falcon signature generation, we managed to combine seven kernels into one (complete_private_comb), effectively reducing the computation time from $113.47\mu s$ to $57.28\mu s$. For Falcon signature verification, combining six kernels into one (comb_all_kernels), reduces the computation time from $65.76\mu s$ to $35.1\mu s$. For both cases, the computation time is almost reduced by half. This shows that reducing the number of kernel invocations does improve the performance significantly. In contrast to Falcon, in Mitaka signature generation we did not find many kernels that have the same degree of parallelism. In the first case, we managed to combine the kernels normaldist and poly_mul_fft, but the performance gain was not significant (4% reduction), because normaldist is a heavy-weight kernel compared to poly_mul_fft. The second case which combines poly_mul_fft and poly_add, managed to reduce the computation time from $23.01\mu s$ to $15.20\mu s$. Unfortunately, we did not find any opportunity to apply the kernel fusions to the Mitaka signature verification.

TABLE 4: Experimental Platforms Used

	Platform-1	Platform-2		
	Desktop Workstation	Cloud System		
GPU	RTX 3080	V100	T4	A100
CUDA Cores	8704	5120	2560	8192
Architecture	Ampere	Volta	Turing	Ampere
Compute capability	8.6	7.0	7.5	8.0
Clock (GHz)	1.710	1.246	0.585	1.410
Memory bandwidth (GB/s)	760	900	300	1935
No. Streaming Multiprocessor (SM)	68	80	40	64
Compiler	CUDA 11	CUDA 11		
CPU	Intel i9-10900K	Intel Xeon Gold 6150		
Clock	3.70 GHz	2.2 GHz		

4 Experimental Results and Discussions

The evaluation platforms used in our experiments are detailed in Table 4. Platform-1 is a desktop workstation consisting of Intel(R) Core(TM) i9-10900K CPU operating at 3.70



(a) RTX 3080

(b) A100

(c) T4

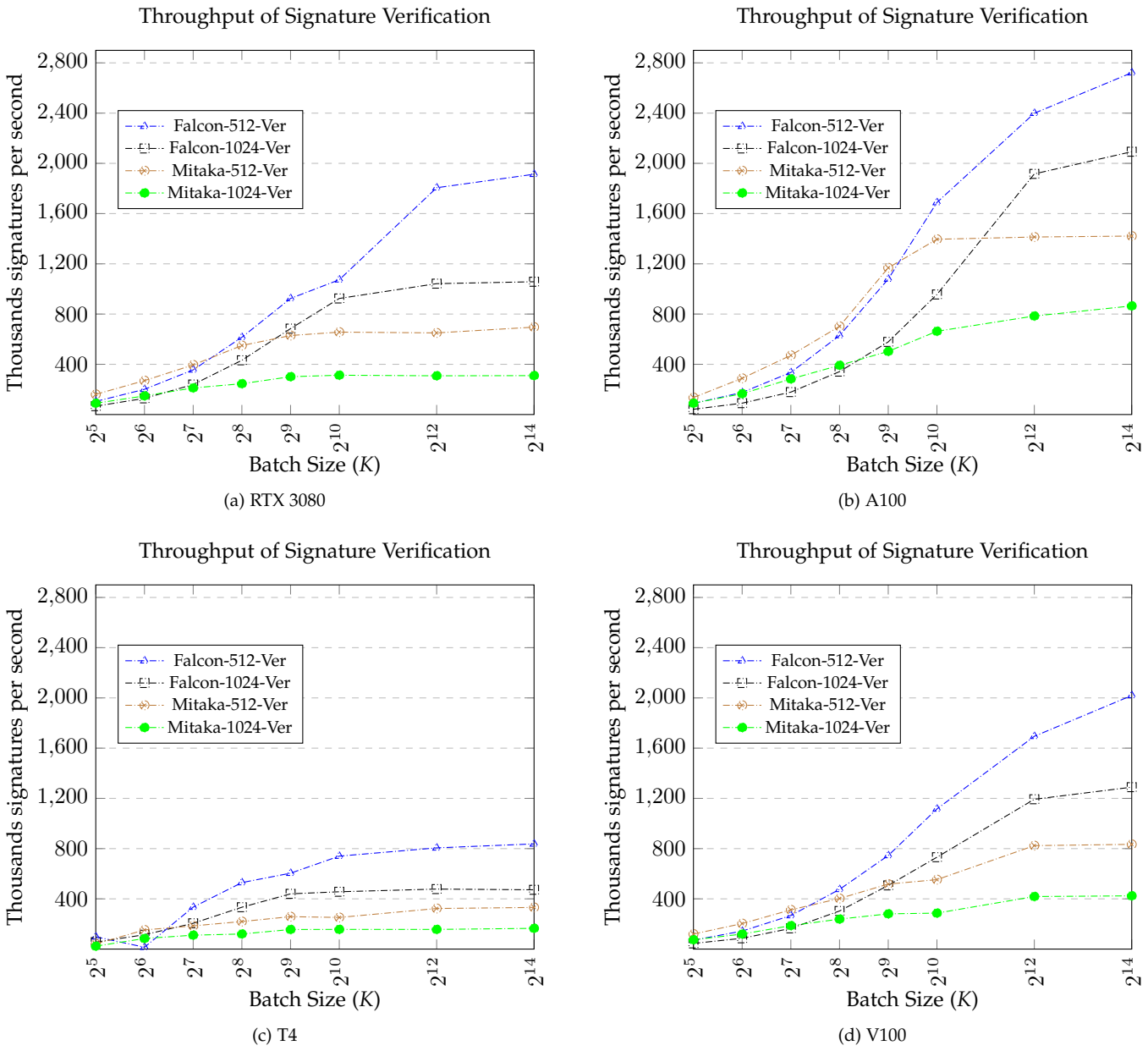
(d) V100

Fig. 6: Throughput of Falcon and Mitaka signature generation on various GPU devices

GHz clock, an RTX 3080 GPU and 32 GB RAM. Platform-2 is the ARDC Nectar Research Cloud system [45] that allows flexible configurations on computing resources. The GPU devices that we used in Platform-2 are A100, T4 and V100. Note that RTX 3080 is a consumer-grade GPU commonly found in desktop workstations, while the A100, T4 and V100 GPUs are server-grade GPUs with higher performance. These GPUs represent the four state-of-the-art NVIDIA GPU architectures: Volta (V100, from the year 2017), Turing (T4, from the year 2018) and Ampere (A100, from the year 2020; RTX 3080, from the year 2021). Following the parallelization strategy described in Section 3.1, K GPU blocks are launched to generate/verify K signatures in parallel. Within each block, multiple threads are used to compute one signature.

4.1 Performance of Falcon and Mitaka Signature Schemes on GPUs

Fig. 6 shows the performance of Falcon and Mitaka signature generation on four selected GPU platforms. The throughput of signature generation increases when the size of the workload increases (i.e., larger K). In general, the throughput saturates when the batch size K is between 1024 to $16,384$. However, for T4 with a smaller number of cores, the throughput saturates earlier when $K \geq 64$. Note that throughput saturation indicates that the GPU is already fully loaded, giving additional workload (i.e., increasing K) will not produce higher throughput anymore. Experimental results also show that Mitaka-512 and Mitaka-1024 are always faster than Falcon-512 and Falcon-1024. The speed-up can range from $1.99\times$ to $19.72\times$ depending on the batch size and GPU platforms. This is mainly because



(a) RTX 3080

(b) A100

(c) T4

(d) V100

Fig. 7: Throughput of Falcon-512 and Mitaka-512 signature verification on various GPU devices.

Mitaka uses a parallelizable sampler, which benefits from the parallel architecture in a GPU. In contrast, the Falcon *ffSampling* is not parallelizable, and only a coarse-grained implementation on GPUs is possible. This shows that the design choice of a signature scheme can greatly affect its performance on parallel hardware architectures.

Fig. 7 shows the performance of Falcon and Mitaka signature verification on various GPU platforms. Similar to signature generation, the throughput of verification increases when the size of workload increases (i.e., larger K), but it takes more workload to reach the saturated state. The throughput saturation happens in all selected GPUs when the batch size K is $\geq$ 4096. In contrast to signature generation, Falcon has a remarkably high verification throughput across all the selected GPUs, which is higher than Mitaka for most of the test cases. We note that this performance difference is due to the efficient verification process in Falcon which only involves simple polynomial arithmetic in the integer domain using number theoretic transform (NTT). Hence, all the computations can be carried out using integer units in GPUs. On the other hand, Mitaka signature verification was computed over double precision involving the fast Fourier transform (FFT), which is executed on floating point units in GPUs. For A100 and V100, the throughput of 32-bit integer units is 2$\times$ higher than 64-bit floating point units [28], but the gap is higher for T4 and RTX 3080 (32$\times$ higher [28]). This explains that Mitaka signature verification can be significantly slower if it is implemented on double precision.

Table 5 shows the throughput and latency of Falcon/Mitaka-512 and Falcon/Mitaka-1024, respectively. The experiments were carried out 30 times and the average results were reported. Experimental results show that the signature generations have low latency and high through-

put performance. For instance, the slowest GPU in our experiment, T4, can produce 16,384 Falcon-1024 signatures in 2144.22 ms; the fastest GPU, A100 can complete the same task in only 279.61 ms. The performance of signature verification is equally impressive, wherein all verifications can be completed within 100 ms regardless of the batch size and the GPU used.

4.2 Comparison with Existing Works

Table 6 shows the comparison of our work against the state-of-the-art Falcon and Mitaka implementation on CPU. We also compared our implementation results with three recently published works on the GPU implementation of post-quantum signature schemes. FP64 refers to the reference implementation provided by the Falcon and Mitaka authors, which is implemented using double precision floating point. AVX2 refers to the optimized implementation utilizing AVX2 instructions available on the CPU. Our Falcon-512 implementation on RTX 3080 is $7.78\times$ and $52.43\times$ faster than the AVX2 implementation for sign and verify, respectively. For the case of Falcon-1024, the speed-up is higher ($20.56\times$ and $148.56\times$), due to the high parallelism available in GPU. Since there is no AVX2 implementation of Mitaka available, we compare our work with the reference implementation (FP64) on CPU. Our GPU implementation is $9.15\times$ (sign)/$14.45\times$ (verify) and $39.07\times$ (sign)/$69.1\times$ (verify) faster than FP64 for Mitaka-512 and Mitaka-1024 respectively.

Wang et al. [46] had reported the first implementation of XMSS signature on GPU devices. For the parameter set XMSS_10, they reported a throughput of 225396 sign/s and 730450 verify/s on RTX 3090. Note that this GPU has more cores compared to the one we use, so we scale the results accordingly. XMSS_10 [46] has a faster signature generation compared to Falcon-512 and Mitaka-512. On the other hand, Falcon and Mitaka verification throughputs are $3.16\times$ and $1.14\times$ faster than XMSS_10 (scaled). Similar results are also observed in the recent GPU implementation of Dilithium [26]. Dilithium achieved throughput of $717,306$ sign/s and $1,960,182$ verify/s on RTX 3090 Ti, which is $20.8\times$ and $7.84\times$ (scaled) faster than Falcon-512 and Mitaka-512, respectively. In contrast, the verification throughput of Falcon-512 and Mitaka-512 is $3.2\times$ and $1.18\times$ (scaled) faster than Dilithium. Sun et al. [14] showed that throughput of $5,152$ sign/s and $106,390$ verify/s can be achieved by SPHINCS on an older GPU, GTX $1,080$. Our implementation of Falcon-512 and Mitaka-512 are $1.59\times$ and $4.22\times$ faster in signature generation; it is also $5.28\times$ and $1.92\times$ faster in signature verification, compared to SPHINCS [14].

4.3 Porting Mitaka Verification to the Integer Domain

From the experimental results, we note that Mitaka signature verification is slower than Falcon, due to the use of double-precision arithmetic. The original Mitaka scheme [6] does not restrict that the verification process must reside on the floating point domain. By porting it over to the integer domain, the Mitaka verification throughput on GPUs can be greatly improved. Note that the Mitaka verification process is very similar to the Falcon verification [4] with the same polynomial degree and modulus in the arithmetic. Hence, the FFT used by the Mitaka verification can be easily replaced with the NTT in the integer domain. In this subsection, we show two versions of Mitaka verification on the Mitaka-512 parameter set. The first version follows strictly the reference implementation provided by the authors [6] utilizing double precision arithmetic. The second version replaced the FFT/IFFT with NTT/INTT and computes all operations using a 32-bit integer unit.

Referring to Table 7, the throughput of Mitaka-512 verification is greatly improved when it is implemented on the integer domain. On an A100 GPU, the integer version (INT32) of Mitaka verification is $2.67\times$ faster than the floating point version (FP64). This result is also $1.39\times$ faster than Falcon-512 verification on the same GPU device.

4.4 Coarse-grained Implementation

This subsection presents the implementation of Falcon-512 and Mitaka-512 in a coarse-grained parallel method (refer to Section 3.1 for details) on an RTX 3080 GPU. Referring to Table 9, the coarse-grained implementation of both schemes is also able to produce high throughput performance. However, the proposed implementation that mixes fine- and coarse-grain parallelism, consistently performs better than the coarse-grained approach. For instance, when K=16384, the mixed approach is $1.38\times$ and $1.80\times$ faster than the coarse-grain approach for Mitaka signature generation and verification, respectively. Similarly, the mixed approach is $1.51\times$ and $3.81\times$ faster than the coarse-grain approach for Falcon signature generation and verification, respectively.

4.5 *ffSampling*: Recursive vs Iterative

The *ffSampling* process can be implemented in a recursive manner, as originally proposed by the authors of Falcon [4]. However, it is very slow when implemented on a GPU, due to the complex tree-traversal using recursive function calls. Referring to Table 8, the throughput of Falcon signature generation is low when the *ffSampling* is implemented in a recursive manner. The proposed iterative *ffSampling* can offer a much higher throughput, in the range of $11.44\times$ – $14.39\times$. This phenomenon can be observed across different GPU architectures, which proves that the iterative *ffSampling* is more efficient compared to the recursive version.

4.6 Discussions

Referring to the use case in e-commerce that we discussed in Section 1, we observed that an A100 GPU can process the required signature generations (583,000) and verifications (1,166,000) in $\approx 10s$ and $\approx 0.42s$ respectively, using Falcon-512. Similarly, Mitaka also shows similar performance wherein $\approx 3.5s$ and $\approx 0.31s$ are required to process the same amount of signature generations and verification using Mitaka-512 (integer verification). This shows that by offloading these computations to a GPU accelerator [35], the response time can be greatly reduced.

The proposed iterative *ffSampling* (Algorithm 2) is a generic solution that can be used in many other computing architectures, including new GPU architectures. Stack management in a recursive function is generally regarded as a complicated task that requires complex control hardware.

TABLE 5: Throughput and latency performance of Falcon and Mitaka on four selected GPU platforms.

RTX 3080

K	Falcon-512				Mitaka-512				Falcon-1024				Mitaka-1204			
	Sign		Verify		Sign		Verify		Sign		Verify		Sign		Verify	
	Op/s	ms	Op/s	ms	Op/s	ms	Op/s	ms	Op/s	ms	Op/s	ms	Op/s	ms	Op/s	ms
32	2000	16	104112	0.31	39448	0.81	160708	0.2	1655	19.34	66653	0.48	18786	1.7	89815	0.36
64	3796	16.86	200723	0.32	61188	1.05	270047	0.24	3438	18.62	128667	0.5	26889	2.38	148258	0.43
128	7751	16.51	355587	0.36	70473	1.82	395260	0.32	6366	20.11	239006	0.54	30335	4.22	211786	0.6
256	14351	17.84	614959	0.42	75198	3.4	549508	0.47	9940	25.75	431779	0.59	33211	7.71	245399	1.04
512	22879	22.38	924375	0.55	76055	6.73	629086	0.81	12287	41.67	684112	0.75	31975	16.01	301159	1.7
1024	23454	43.66	1073105	0.95	77275	13.25	656274	1.56	12867	79.58	924214	1.11	34577	29.62	313087	3.27
4096	27025	151.56	1805872	2.27	75200	54.47	649951	6.3	14924	274.46	1140358	3.59	33987	120.52	308443	13.28
16384	**27908**	587.07	**1913380**	8.56	**74010**	221.38	**695931**	23.54	**15239**	1075.14	**1217317**	13.46	**34025**	481.53	**309841**	52.88

V100

K	Sign		Verify		Sign		Verify		Sign		Verify		Sign		Verify	
	Op/s	ms	Op/s	ms	Op/s	ms	Op/s	ms	Op/s	ms	Op/s	ms	Op/s	ms	Op/s	ms
32	1668	19.18	70801	0.45	8808	3.63	119904	0.27	1948	16.43	45094	0.71	9006	3.55	73975	0.43
64	3318	19.29	143328	0.45	19012	3.37	203066	0.32	3903	16.4	85852	0.75	12838	4.99	119596	0.54
128	6782	18.87	266987	0.48	33643	3.8	312402	0.41	7500	17.07	164908	0.78	25455	5.03	186689	0.69
256	13861	18.47	475483	0.54	53379	4.8	402962	0.64	13834	18.51	303398	0.84	32796	7.81	238671	1.07
512	25534	20.05	745156	0.69	65128	7.86	517649	0.99	19650	26.06	507099	1.01	34239	14.95	280613	1.82
1024	25876	39.57	1115799	0.92	69077	14.82	552935	1.85	19282	53.11	731529	1.4	35901	28.52	286369	3.58
4096	31771	128.92	1694377	2.42	84384	48.54	823843	4.97	25741	159.12	1192980	3.43	42208	97.04	418551	9.79
16384	**37100**	441.62	**2019222**	8.11	**86361**	189.72	**843035**	19.43	**26745**	612.6	**1289210**	12.71	**56829**	288.3	**423898**	38.65

T4

K	Sign		Verify		Sign		Verify		Sign		Verify		Sign		Verify	
	Op/s	ms	Op/s	ms	Op/s	ms	Op/s	ms	Op/s	ms	Op/s	ms	Op/s	ms	Op/s	ms
32	1380	23.19	97924	0.33	5844	5.48	41235	0.78	1299	24.63	55401	0.58	3521	9.09	23114	1.38
64	2627	24.36	147922	0.43	28921	2.21	152549	0.42	2027	31.57	114240	0.56	15471	4.14	84425	0.76
128	4793	26.71	333985	0.38	29178	4.39	184732	0.69	3814	33.56	205351	0.62	16207	7.9	110647	1.16
256	7996	32.02	529447	0.48	31734	8.07	219441	1.17	5225	49	333150	0.77	16694	15.33	120143	2.13
512	8954	57.18	604334	0.85	27472	18.64	258165	1.98	5387	95.04	439986	1.16	14389	35.58	155919	3.28
1024	10024	102.15	740110	1.38	27045	37.86	251786	4.07	6417	159.58	456454	2.24	14956	68.47	156734	6.53
4096	12666	323.39	805956	5.08	**29917**	136.91	323152	12.68	7270	563.41	**477821**	8.57	14847	275.88	156544	26.17
16384	**12934**	1266.74	**837496**	19.56	29066	563.68	**331619**	49.41	**7641**	2144.22	472051	34.71	**15213**	1076.97	**165105**	99.23

A100

K	Sign		Verify		Sign		Verify		Sign		Verify		Sign		Verify	
	Op/s	ms	Op/s	ms	Op/s	ms	Op/s	ms	Op/s	ms	Op/s	ms	Op/s	ms	Op/s	ms
32	4674	6.85	88523	0.36	25907	1.24	135281	0.24	1542	20.75	41976	0.76	10122	3.16	89741	0.36
64	8963	7.14	175764	0.36	36156	1.77	288472	0.22	3811	16.79	90440	0.71	28412	2.25	165736	0.39
128	17090	7.49	335189	0.38	60355	2.12	471454	0.27	7060	18.13	178508	0.72	39257	3.26	283222	0.45
256	26037	9.83	629287	0.41	105923	2.42	702309	0.36	12556	20.39	341587	0.75	55935	4.58	392042	0.65
512	44229	11.58	1078854	0.47	148803	3.44	1168452	0.44	25513	20.07	578306	0.89	73614	6.96	502814	1.02
1024	47029	21.77	1688585	0.61	157948	6.48	1395524	0.73	25272	40.52	956023	1.07	71137	14.39	662993	1.54
4096	55505	73.8	2399110	1.71	161044	25.43	1413547	2.9	44680	91.67	1916189	2.14	87413	46.86	784442	5.22
16384	**58595**	279.61	**2721562**	6.02	**161985**	101.15	**1421046**	11.53	**37550**	436.32	**2092758**	7.83	**97413**	168.19	**864741**	18.95

However, this is not the strength of GPU cores, because they are designed based on the single instruction multiple data (SIMD) paradigm. In this paradigm, each core has a small control logic and only computes simple tasks [28]. Such GPU architecture has been the same since its inception and there is no clear indication that the manufacturers are going to make a drastic change on it in the near future. Hence, we believe that slow recursive function calls will remain a problem on GPU architectures released in the near future, and thus, Algorithm 2 is likely to remain useful for such architectures. On top of that, all the parallel implementation techniques proposed in this paper are also generic and independent of the generation of GPU architecture. This is illustrated by the experimental results, which exhibit a similar speed-up pattern for Falcon and Mitaka, regardless of the GPU architectures.

In addition to GPU, Algorithm 2 can also be used for an FPGA implementation. For instance, [40] is the state-of-the-art Falcon implementation on an FPGA. It only parallelizes the inner operations of *ffSampling*, while the tree-traversal process mainly relies on the software approach. This is mainly because the tree-traversal requires stack management which is also expensive on the FPGA. By adopting the proposed iterative solution in Algorithm 2, one can emulate the stack management more efficiently on an FPGA. It allows more parallelism and potentially improving the throughput performance.

This paper only evaluated the throughput of these signatures but does not consider the entire communication protocols (e.g., TLS handshake). Therefore, the performance

TABLE 6: Comparing with CPU and state-of-the-art implementations.

		Fal-512	Mit-512	Fal-1024	Mit-1024
		CPU, Op/s			
AVX2	Sign	3167	-	2850	-
	Verify	18230	-	18319	-
FP64	Sign	3587	8087	2045	4146
	Verify	36491	48153	17694	20565
		GPU, Op/s			
This work [1] (RTX 3080)	Sign	27908	74010	15239	34025
	Verify	1913380	695931	1217317	309841
XMSS_10 [46] (SHA-256)	Sign	225396/186913[2]			
	Verify	730450/605739[2]			
Dilithium [26]	Sign	717306/580676[3]			
	Verify	1960182/1586814[3]			
SPHINCS [14] (ChaCha)	Sign	5152/17516[4]			
	Verify	106390/361726[4]			

[1] The highest throughput with $K = 16384$.
[2] Performance scaled by the number of cores, 10496/8704. RTX 3090 was used in [46].
[3] Performance scaled by 10752/8704 RTX 3090 Ti was used in [26].
[4] Performance scaled by 2560/8704. GTX 1080 was used in [14].

TABLE 7: Throughput of Mitaka-512 verification on floating point and integer units

	GPU	Mitaka-512 FP64		Mitaka-512 INT32	
		Op/s	ms	Op/s	ms
K=16384	RTX 3080	695931	23.54	2007649	8.16
	V100	843035	19.44	2130201	7.69
	T4	331619	49.41	918191	17.84
	A100	1421046	11.53	3790838	4.32

TABLE 8: Throughput of Falcon-512 signature generation using recursive and iterative *ffSampling*

	GPU	Falcon-512 Recursive		Falcon-512 Iterative	
		Op/s	ms	Op/s	ms
K=16384	RTX 3080	2439	6717.51	27908	587.07
	V100	2898	5653.36	37100	441.62
	T4	1009	16237.86	12934	1266.74
	A100	4025	4070.56	58595	279.61

reported here may not represent the entire communication performance. In future, we aim to integrate the PQC signature schemes with existing software libraries like OpenSSL to evaluate the end-to-end signature performance. Another interesting research direction is to apply the proposed iterative *ffSampling* (Algorithm 2) on FPGA devices to completely remove the reliance on software control. This can potentially improve the performance of [40] and achieve high performance Falcon signing and verification.

5 Conclusions

A high throughput implementation of Falcon and Mitaka was presented in this article. Experimental results show that Mitaka has a much higher signature generation throughput compared to Falcon, due to the parallelizable sampling process. On the other hand, Falcon enjoys a higher verification throughput as all the computations can be performed in the integer domain. This shows that the choices made in designing a signature scheme can greatly affect its performance on various hardware architectures, including parallel architecture like GPU. Further analysis on the Mitaka verification process shows that its performance on GPUs is improved by porting the existing implementation from FP64 to the integer domain. Given that the NIST round 4 standardization process for signature schemes is still ongoing, parallelizing the selected candidates on GPUs would be a good research direction.

References

[1] "Post-quantum cryptography: Round 1 submissions." [Online]. Available: https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-1-submissions

[2] R. Avanzi, J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé, "Crystals-kyber," 2021. [Online]. Available: https://csrc.nist.gov/CSRC/media/Projects/post-quantum-cryptography/documents/round-3/submissions/Kyber-Round3.zip

[3] V. Lyubashevsky, L. Ducas, E. Kiltz, T. Lepoint, P. Schwabe, G. Seiler, D. Stehlé, and S. Bai, "Crystals-dilithium," 2021. [Online]. Available: https://csrc.nist.gov/CSRC/media/Projects/post-quantum-cryptography/documents/round-3/submissions/Dilithium-Round3.zip

[4] P.-A. Fouque, J. Hoffstein, P. Kirchner, V. Lyubashevsky, T. Pornin, T. Prest, T. Ricosset, G. Seiler, W. Whyte, and Z. Zhang, "Falcon: Fast-fourier lattice-based compact signatures over ntru," 2021. [Online]. Available: https://csrc.nist.gov/CSRC/media/Projects/post-quantum-cryptography/documents/round-3/submissions/Falcon-Round3.zip

[5] J.-P. Aumasson, D. J. Bernstein, C. Dobraunig, M. Eichlseder, S. Fluhrer, S.-L. Gazdag, A. Hülsing, P. Kampanakis, S. Kölbl, T. Lange *et al.*, "Sphincs+." [Online]. Available: https://csrc.nist.gov/CSRC/media/Projects/post-quantum-cryptography/documents/round-3/submissions/SPHINCS-Round3.zip

[6] T. Espitau, P.-A. Fouque, F. Gérard, M. Rossi, A. Takahashi, M. Tibouchi, A. Wallet, and Y. Yu, ": A simpler, parallelizable, maskable variant of," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2022, pp. 222–253.

[7] J. M. B. Mera, A. Karmakar, S. Kundu, and I. Verbauwhede, "Scabbard: a suite of efficient learning with rounding key-encapsulation mechanisms," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 474–509, 2021.

[8] J.-P. D'Anvers, K. Angshuman, R. Sujoy Sinha, F. Vercauteren, J. Maria Bermudo Mera, M. Van Beirendonck, and A. Basso, "Saber: Mod-LWR based kem," 2020. [Online]. Available: https://csrc.nist.gov/CSRC/media/Projects/post-quantum-cryptography/documents/round-3/submissions/SABER-Round3.zip

[9] Y. Zhu, M. Zhu, B. Yang, W. Zhu, C. Deng, C. Chen, S. Wei, and L. Liu, "Lwrpro: An energy-efficient configurable crypto-processor for module-lwr," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 68, no. 3, pp. 1146–1159, 2021.

[10] T. Pornin, "New efficient, constant-time implementations of falcon," *Cryptology ePrint Archive*, 2019. [Online]. Available: https://eprint.iacr.org/2019/893

[11] M. J. Kannwischer, J. Rijneveld, P. Schwabe, and K. Stoffelen, "PQM4: Post-quantum crypto library for the ARM Cortex-M4," https://github.com/mupq/pqm4.

[12] E. Alkim, Y. A. Bilgin, M. Cenk, and F. Gérard, "Cortex-m4 optimizations for {R, M} lwe schemes," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 336–357, 2020.

[13] H. Seo, P. Sanal, A. Jalali, and R. Azarderakhsh, "Optimized implementation of sike round 2 on 64-bit arm cortex-a processors," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 8, pp. 2659–2671, 2020.

[14] S. Sun, R. Zhang, and H. Ma, "Efficient parallelism of post-quantum signature scheme sphincs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 11, pp. 2542–2555, 2020.

[15] S. Dong, Y. Sun, N. B. Agostini, E. Karimi, D. Lowell, J. Zhou, J. Cano, J. L. Abellán, and D. Kaeli, "Spartan: A sparsity-adaptive framework to accelerate deep neural network training on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 10, pp. 2448–2463, 2021.

TABLE 9: Throughput of Coarse-grain Parallel Implementation of Mitaka-512 and Falcon-512 on an RTX 3080 GPU

	Mitaka-512				Falcon-512			
K	Coarse-grain		Mixed (This Work)		Coarse-grain		Mixed (This Work)	
	Sign/s	Verify/s	Sign/s	Verify/s	Sign/s	Verify/s	Sign/s	Verify/s
32	2058	19547	39448	160708	1189	18190	2000	104112
64	8001	34721	61188	270047	2231	26741	3796	200723
128	10222	41169	70473	395260	3176	29654	7751	355587
256	10398	45800	75198	549508	4160	32975	14351	614959
512	12445	48487	76055	629086	4696	34242	22879	924375
1024	18064	87583	77275	656274	7995	66708	23454	1073105
4096	52088	358953	75200	649951	17620	246050	27025	1805872
16384	53710	386726	74010	695931	18421	501752	27908	1913380
32768	45696	313306	73489	684112	16580	318554	27647	1900473

[16] N. Tahmasebi, P. Boulanger, J. Yun, G. Fallone, M. Noga, and K. Punithakumar, "Real-time lung tumor tracking using a CUDA enabled nonrigid registration algorithm for MRI," *IEEE journal of translational engineering in health and medicine*, vol. 8, pp. 1–8, 2020.

[17] A. Al Badawi, B. Veeravalli, J. Lin, N. Xiao, M. Kazuaki, and A. K. M. Mi, "Multi-GPU design and performance evaluation of homomorphic encryption on GPU clusters," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 2, pp. 379–391, 2020.

[18] W.-K. Lee, B.-M. Goi, and R. C.-W. Phan, "Terabit encryption in a second: Performance evaluation of block ciphers in GPU with Kepler, Maxwell, and Pascal architectures," *Concurrency and Computation: Practice and Experience*, vol. 31, no. 11, p. e5048, 2019.

[19] X. Su, C. He, T. Liu, and L. Wu, "Full parallel power flow solution: A GPU-CPU-based vectorization parallelization and sparse techniques for newton–raphson implementation," *IEEE Transactions on Smart Grid*, vol. 11, no. 3, pp. 1833–1844, 2019.

[20] "NVIDIA GPUs on IBM cloud servers," https://www.ibm.com/cloud/gpu, 2021, accessed: 2021-10-10.

[21] "Amazon ec2 p4d instances," https://aws.amazon.com/ec2/instance-types/p4/, 2021, accessed: 2021-10-10.

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

[23] W.-K. Lee, H. Seo, Z. Zhang, and S. O. Hwang, "Tensorcrypto: High throughput acceleration of lattice-based cryptography using tensor core on gpu," *IEEE Access*, vol. 10, pp. 20 616–20 632, 2022.

[24] W.-K. Lee, H. Seo, S. O. Hwang, R. Achar, A. Karmakar, and J. M. B. Mera, "Dpcrypto: Acceleration of post-quantum cryptography using dot-product instructions on gpus," *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2022.

[25] W. K. Lee and S. O. Hwang, "High throughput implementation of post-quantum key encapsulation and decapsulation on GPU for Internet of Things applications," *IEEE Transactions on Services Computing*, 2021.

[26] S. Shen, H. Yang, W. Dai, Z. Liu, and Y. Zhao, "High-throughput gpu implementation of dilithium post-quantum digital signature," *arXiv preprint arXiv:2211.12265*, 2022.

[27] Alibaba, "Alibaba's 11.11 signals china retail health, a boon for international brands," https://www.alizila.com/alibabas-11-11-shows-retail-boom-in-china/, 2023, (accessed on Feb. 17, 2023).

[28] C. NVIDIA, "CUDA C programming guide, version 11.6," *NVIDIA Corp*, 2022.

[29] D. Stehlé and R. Steinfeld, "Making NTRU as secure as worst-case problems over ideal lattices," in *EUROCRYPT*, ser. Lecture Notes in Computer Science, vol. 6632. Springer, 2011, pp. 27–47.

[30] C. Gentry, C. Peikert, and V. Vaikuntanathan, "Trapdoors for hard lattices and new cryptographic constructions," in *STOC*. ACM, 2008, pp. 197–206.

[31] L. Ducas and T. Prest, "Fast fourier orthogonalization," in *ISSAC*. ACM, 2016, pp. 191–198.

[32] T. Prest, "Gaussian sampling in lattice-based cryptography," 2015.

[33] C. Peikert, "An efficient and parallel Gaussian sampler for lattices," in *CRYPTO*, ser. Lecture Notes in Computer Science, vol. 6223. Springer, 2010, pp. 80–97.

[34] A. Al Badawi, B. Veeravalli, C. F. Mun, and K. M. M. Aung, "High-performance FV somewhat homomorphic encryption on GPUs: An implementation using CUDA," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 70–95, 2018.

[35] W. Pan, F. Zheng, Y. Zhao, W.-T. Zhu, and J. Jing, "An efficient elliptic curve cryptography signature server with gpu acceleration," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 1, pp. 111–122, 2016.

[36] T. Oder, J. Speith, K. Höltgen, and T. Güneysu, "Towards practical microcontroller implementation of the signature scheme falcon," in *International Conference on Post-Quantum Cryptography*. Springer, 2019, pp. 65–80.

[37] D. T. Nguyen and K. Gaj, "Fast falcon signature generation and verification using armv8 neon instructions," *PQC2022*, 2022. [Online]. Available: https://csrc.nist.gov/csrc/media/Events/2022/fourth-pqc-standardization-conference/documents/papers/fast-falcon-signature-generation-and-verification-pqc2022.pdf

[38] Y. Kim, J. Song, and S. C. Seo, "Accelerating falcon on armv8," *IEEE Access*, vol. 10, pp. 44 446–44 460, 2022.

[39] L. Beckwith, D. T. Nguyen, and K. Gaj, "High-performance hardware implementation of lattice-based digital signatures," *Cryptology ePrint Archive*, 2022. [Online]. Available: https://eprint.iacr.org/2022/217

[40] E. Karabulut and A. Aysu, "A hardware-software co-design for the discrete gaussian sampling of falcon digital signature," *Cryptology ePrint Archive*, 2023.

[41] W. M. Gentleman and G. Sande, "Fast fourier transforms: for fun and profit," in *AFIPS Fall Joint Computing Conference*, ser. AFIPS Conference Proceedings, vol. 29. AFIPS / ACM / Spartan Books, Washington D.C., 1966, pp. 563–578.

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

[43] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux device drivers*. " O'Reilly Media, Inc.", 2005.

[44] W.-K. Lee, G. Wong, Xian-Fu, Bok-Min, and R. C.-W. Phan, "Cuda-ssl: Ssl/tls accelerated by gpu," in *2017 International Carnahan Conference on Security Technology (ICCST)*. Springer, 2019, pp. 1–6.

[45] "Australian research data commons nectar research cloud system," 2023. [Online]. Available: https://ardc.edu.au/services/ardc-nectar-research-cloud/

[46] Z. Wang, X. Dong, H. Chen, and Y. Kang, "Efficient gpu implementations of post-quantum signature xmss," *IEEE Transactions on Parallel and Distributed Systems*, 2023.

ACKNOWLEDGMENTS

The experiments were carried out on the Nectar Research Cloud system supported by the Australian Research Data Commons (ARDC). Wai-Kong Lee was supported by the Brain Pool Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science and Information Communication Technology (ICT) under Grant 2019H1D3A1A01102607. The work of Seong Oun Hwang was supported by the Gachon University research fund under Grant GCU-202304050001 and was conducted by funding from The Circle Foundation (Republic of Korea)

for 1 years since December 2023 as Quantum Security Research Center selected as the 2023 The Circle Foundation Innovative Science Technology Center under Grant 2023 TCF Innovative Science Project-05.

Wai-Kong Lee received the B.Eng. degree in electronics and the M.Sc. degree from Multimedia University in 2006 and 2009, respectively, and the Ph.D. degree in engineering from Universiti Tunku Abdul Rahman, Malaysia, in 2018. He was a Visiting Scholar with Carleton University, Canada, in 2017, Feng Chia University, Taiwan, in 2016 and 2018, and OTH Regensburg, Germany, in 2015, 2018 and 2019. Prior to joining academia, he worked in several multinational companies including Agilent Technologies (Malaysia) as R&D engineer. His research interests are in the areas of cryptography, numerical algorithms, GPU computing, the Internet of Things, and energy harvesting. He is currently a post-doctoral researcher in Gachon University, South Korea.

Raymond K. Zhao received the BEng degree in computer science and technology from Zhejiang University, China, in 2015, the master's degree in network and security from Monash University, Australia, in 2017, and the PhD degree from the Faculty of Information Technology (FIT), Monash University, Australia, in 2022. He was a research fellow in the Department of Software Systems and Cybersecurity, FIT, Monash University, Australia, in 2022. Since November 2022, he has been a postdoctoral fellow with CSIRO's Data61. His main research interests include efficient and secure implementation techniques for post-quantum cryptographic applications and protocols.

Ron Steinfeld (S'99-M'04) received the BSc degree in mathematics and physics from Monash University, Australia, in 1998, the BE (First Class Hons) degree in electrical and computer systems engineering from Monash University, in 2000, and the PhD degree in computer science from Monash University, in 2003. Since 2020, he is an Associate Professor with the Department of Software Systems and Cybersecurity, Monash University, Australia. From 2003 to 2006, he was a postdoctoral research fellow in cryptography and information security with Macquarie University, Australia. From 2007 to 2009, he was a Macquarie University research fellow in cryptography and information security. From 2009 to 2014, he was an ARC Australian research fellow in cryptography and information security with Macquarie University (until 2012) and then with Monash University (2012-2014) and a senior lecturer at Monash University from 2015 to 2019. His main research interests include the design and analysis of cryptographic algorithms and cybersecurity protocols with a focus on post-quantum cryptography. He is a member of the IEEE.

Amin Sakzad (M'12) received the PhD degree in applied mathematics from the Amirkabir University of Technology (Tehran Polytechnique), Tehran, Iran, in 2011. He was a research fellow with the Software Defined Telecommunications (SDT) Laboratory, Department of Electrical and Computer Systems Engineering (ECSE), Monash University, Melbourne, Australia, from 2012-2015. Starting from 2016, he held a postdoctoral research fellowship position with the Faculty of Information Technology (FIT), Monash University, Melbourne, Australia. Since May 2017, he has been appointed as a lecturer with FIT, Monash University, Melbourne, Australia. As of 2020, he is a senior lecturer at the Department of Software Systems and Cybersecurity at Monash University. His research interests include Euclidean lattices, lattice-based cryptography, and wireless network coding.

Seong Oun Hwang (Senior Member, IEEE) received the B.S. degree in mathematics from Seoul National University, in 1993, the M.S. degree in information and communications engineering from the Pohang University of Science and Technology, in 1998, and the Ph.D. degree in computer science from the Korea Advanced Institute of Science and Technology, in 2004, South Korea. He worked as a Software Engineer with LG-CNS Systems, Inc., from 1994 to 1996. He worked as a Senior Researcher with the Electronics and Telecommunications Research Institute (ETRI), from 1998 to 2007. He worked as a Professor with the Department of Software and Communications Engineering, Hongik University, from 2008 to 2019. He is currently a Professor with the Department of Computer Engineering, Gachon University. His research interests include cryptography, cybersecurity, and artificial intelligence. He is an Editor of *ETRI Journal*.