cuFalcon: An Adaptive Parallel GPU Implementation for High-Performance Falcon Acceleration

Wenqian Li, Hanyu Wei, Shiyu Shen, Hao Yang, Wangchen Dai, and Yunlei Zhao

***Abstract*—The rapid advancement of quantum computing has ushered in a new era of post-quantum cryptography, urgently demanding quantum-resistant digital signatures to secure modern communications and transactions. Among NIST-standardized candidates, Falcon—a compact lattice-based signature scheme—stands out for its suitability in size-sensitive applications. In this paper, we present cuFalcon, a high-throughput GPU implementation of Falcon that addresses its computational bottlenecks through adaptive parallel strategies. At the operational level, we optimize Falcon key components for GPU architectures through memory-efficient FFT, adaptive parallel ffSampling, and a compact computation mode. For signature-level optimization, we implement three versions of cuFalcon: the raw key version, the expanded key version, and the balanced version, which achieves a trade-off between efficiency and memory usage. Additionally, we design batch processing, streaming mechanisms, and memory pooling to handle multiple signature tasks efficiently. Ultimately, performance evaluations show significant improvements, with the raw key version achieving 172k signatures per second and the expanded key version reaching 201k. Compared to the raw key version, the balanced version achieves a 7% improvement in throughput, while compared to the expanded key version, it reduces memory usage by 70%. Furthermore, our raw key version implementation outperforms the reference implementation by 36.75 $\times$ and achieves a 2.94$\times$ speedup over the state-of-the-art GPU implementation.**

***Index Terms*—Post-Quantum Cryptography, Falcon, Fast Fourier Sampling, GPU acceleration.**

I. Introduction

Digital signatures play a critical role in ensuring data integrity, authenticity, and non-repudiation for applications such as e-commerce, financial transactions, and identity verification. However, the advent of quantum computing [1] poses significant threats to traditional cryptographic systems (e.g., RSA and ECC), necessitating an urgent transition to post-quantum cryptography (PQC). Since 2016, the National Institute of Standards and Technology (NIST) has actively standardized PQC algorithms. Among the selected candidates [2], Falcon [3], a lattice-based scheme, stands out for its compact design and suitability for size-sensitive applications such as DNSSEC [4]. Falcon has also been adopted in real-world applications; for instance, its integration into the Open Quantum Safe (OQS) library [5] enables quantum-safe TLS protocols. Despite these advantages, Falcon's reliance on complex floating-point arithmetic and recursive tree traversal poses significant challenges for GPU acceleration. These limitations motivate cuFalcon's design, which rethinks Falcon computational patterns for massively parallel architectures while preserving its security guarantees.

Corresponding author: Yunlei Zhao.

Wenqian Li and Hanyu Wei are with School of Computer Science, Fudan University, Shanghai 200438, China (e-mail: liwq24@m.fudan.edu.cn; hywei24@m.fudan.edu.cn).

Shiyu Shen and Hao Yang are with Department of Electrical Engineering, City University of Hong Kong, Hong Kong, China (e-mail: crypto@sher1e.dev; crypto@d4rk.dev).

Wangchen Dai is with School of Cyber Science and Technology, Sun Yat-sen University, Shenzhen, China (e-mail: daiwch@mail.sysu.edu.cn).

Yunlei Zhao is with School of Computer Science, Fudan University, Shanghai 200438, State Key Laboratory of Cryptology, Beijing 100878, China (e-mail: ylzhao@fudan.edu.cn).

A. Related Works

Existing studies have explored optimizing Falcon using software instruction sets and hardware design. For resource-constrained devices, Thomas et al. [6] optimized Falcon on ARM Cortex-M4, while Nguyen et al. [7] leveraged NEON SIMD instructions on ARM Cortex-A processors for improved performance. On the hardware, Schmid et al. [8] implemented Falcon on dedicated hardware for both key and signature generation, and Lee et al. [9] proposed a Falcon accelerator using HW/SW co-design with fine-grained optimizations to enhance performance and minimize silicon area. While these implementations demonstrate Falcon's adaptability to specialized hardware, expanding it to GPU platforms opens new opportunities for higher throughput and parallelism.

Leveraging its powerful parallel computing capabilities, the GPU is increasingly being employed to accelerate PQC. Currently, extensive research focuses on GPU implementations of NIST PQC candidate schemes, including the standardized signature schemes CRYSTALS-Dilithium [10], [11], SPHINCS+ [12], and Falcon [13]. Lee et al. [13] introduced the first parallel GPU implementation of Falcon, featuring an iterative Fast Fourier Sampling (ffSampling) but its coarse-grained task partitioning limits parallelism.

B. Our Contributions

In this paper, we propose cuFalcon, which is designed to enhance the computational efficiency of the Falcon digital signature algorithm on GPUs. Our contributions are summarized as follows:

- **Compact and efficient design at operation level.** We optimize the core operations in signature generation, such as FFT and ffSampling, to better align with GPU architectures. First, we design a memory-efficient FFT

implementation that improves memory access patterns by minimizing memory access conflicts and merging memory accesses. Next, we propose an adaptive parallel ffSampling implementation, featuring a highly parallel design to enhance computational efficiency. Additionally, we apply a compact computation model to other operations to reduce unnecessary memory accesses and minimize space usage.

- **High-throughput and space-saving optimization at signature level.** Based on the optimized operations, we implement cuFalcon, which includes three signature implementations: the raw key version, the expanded key version, and a balanced version proposed to balance efficiency and memory usage. The raw key version dynamically generates the Falcon tree during signing, while the expanded key version offloads key-related operations, such as Falcon tree generation, to offline processing. The balanced version reduces memory usage by parallelizing Falcon tree generation and offloading other key operations to offline computation. Additionally, to support scenarios with multiple signing tasks, we design batching and streaming mechanisms, along with a secure and efficient memory pool.
- **Performance evaluation.** To validate the effectiveness of cuFalcon, we conduct comprehensive experiments across three dimensions: operation-level optimizations, signature-level implementations, and comparisons with related works. Benchmark results demonstrate the significant efficiency of our optimized operation and signature workflow designs. For Falcon-512, the raw key version achieves a throughput of 172k signings per second, while the expanded key version reaches 201k signings per second. The proposed balanced version improves throughput by 7% compared to the raw key version and reduces memory usage by 70% compared to the expanded key version. Furthermore, the raw key version signing implementation shows a $36.75\times$ speedup compared to the reference implementation [3] and a $2.94\times$ speedup compared to the state-of-the-art implementation [13].

II. Preliminaries

A. Notation

Let $\mathbb{Q}$ be the field of rational numbers. Let n be a power-of-two, $\phi = x^n + 1$ is a cyclotomic polynomial. We denote the number field $\mathcal{Q} = \mathbb{Q}/(\phi)$, $\boldsymbol{f} = \sum_{i=0}^{n-1} f_i x^i$ be arbitrary elements of it. Matrices will be in bold uppercase (e.g. $\mathbf{B}$), vectors in bold lowercase (e.g. $\mathbf{v}$), and polynomials in bold italic (e.g. $\boldsymbol{f}$). The transpose of a matrix $\mathbf{B}$ will be noted $\mathbf{B^T}$. $\hat{\boldsymbol{f}}$ indicates polynomial $\boldsymbol{f}$ in FFT/NTT domain.

B. Falcon

Falcon is a compact lattice-based signature scheme based on the hash-and-sign paradigm, leveraging the GPV framework [14] for security in both classical and quantum random oracle models [15], [16], along with message-recovery capabilities [17]. Its compactness makes it ideal for practical deployment. Falcon includes two parameter sets [3], Falcon-512 and Falcon-1024, targeting security levels I and V, with modulus $q = 12289$.

The signature generation process (Algorithm 1) begins by decoding the secret key sk to obtain the Falcon tree T and $(\boldsymbol{f}, \boldsymbol{g}, \boldsymbol{F}, \boldsymbol{G})$, transforming them into the FFT domain. After generating salt $\boldsymbol{r}$, the hash of $(\boldsymbol{r}||\mathsf{m}, q, n)$ maps to $\boldsymbol{c} \in \mathbb{Z}_q[x]/(\phi)$. The target vector $\mathbf{t}$ is computed, and the ffSampling algorithm generates $\mathbf{z}$. The resulting vector $\mathbf{s}$ is verified against the norm bound $\lfloor \beta^2 \rfloor$. Finally, the lattice point $(\boldsymbol{s_1}, \boldsymbol{s_2})$ is derived, and $\boldsymbol{s_2}$ is compressed into a bit-string str. The signature is represented as $(\boldsymbol{r}, \mathsf{str})$.

Algorithm 1 $\mathsf{Falcon.Sign}(\mathsf{m}, \mathsf{sk}, \lfloor \beta^2 \rfloor)$

Input: A message m, a secret key sk, a bound $\lfloor \beta^2 \rfloor$
Output: A signature sig of m
1: $((\boldsymbol{f}, \boldsymbol{g}, \boldsymbol{F}, \boldsymbol{G}), \mathsf{T}) \leftarrow \mathsf{Decode}(\mathsf{sk})$
2: $\boldsymbol{r} \leftarrow \{0,1\}^{320}$ uniformly
3: $\boldsymbol{c} \leftarrow \mathsf{HashToPoint}(\boldsymbol{r}\|\mathsf{m}, q, n)$
4: $\mathbf{t} \leftarrow (-\frac{1}{q}\mathsf{FFT}(\boldsymbol{c}) \odot \mathsf{FFT}(\boldsymbol{F}), \frac{1}{q}\mathsf{FFT}(\boldsymbol{c}) \odot \mathsf{FFT}(\boldsymbol{f}))$
5: **while** $\mathsf{str} = \perp$ **do**
6:     **while** $\|\mathbf{s}\|^2 > \lfloor \beta^2 \rfloor$ **do**
7:         $\mathbf{z} \leftarrow \mathsf{ffSampling}_n(\mathbf{t}, \mathsf{T})$
8:         $\mathbf{s} = (\mathbf{t} - \mathbf{z})\hat{\mathbf{B}}$
9:     $(\boldsymbol{s_1}, \boldsymbol{s_2}) \leftarrow \mathsf{invFFT}(\mathbf{s})$
10:     $\mathsf{str} \leftarrow \mathsf{Compress}(\boldsymbol{s_2}, 8 \cdot \mathsf{sbytelen} - 328)$
11: **return** $\mathsf{sig} = (\boldsymbol{r}, \mathsf{str})$

C. Fast Fourier Transform

Fast Fourier Transform (FFT) is a rapid algorithm for computing discrete convolutions with a time complexity of $O(n \log n)$. For the field $\mathbb{Q}[x]/(x^n + 1)$ used in Falcon, where $x^n + 1 = x^n - \zeta^n$ (ζ is a primitive $2n$-th root of unity), the process involves the Chinese remainder map: $\boldsymbol{f} \mapsto (\boldsymbol{f} \bmod x^{n/2} - \zeta^{n/2}, \boldsymbol{f} \bmod x^{n/2} + \zeta^{n/2})$, $\mathbb{Q}[x]/(x^n - \zeta^n) \mapsto \mathbb{Q}[x]/(x^{n/2} - \zeta^{n/2}) \times \mathbb{Q}[x]/(x^{n/2} + \zeta^{n/2})$. Compute the polynomials in these two fields, decomposing the original field into lower-degree polynomial fields based on $\phi(x) = \prod_{k \in Z_m^\times}(x - \zeta^k)$. This is achieved using the Cooley-Tukey (CT) butterfly operation [18], with the inverse typically handled by the Gentleman-Sande (GS) butterfly operation [19]. For $\boldsymbol{f}$ with coefficients c_i and $c_{n/2+i}$, the i-th coefficients of the reduced polynomials are $c'_i = c_i + \zeta^{n/2} c_{n/2+i}$ and $c''_i = c_i - \zeta^{n/2} c_{n/2+i}$. After the k-th layer ($0 \le k < \log_2 n$), pairs of coefficients are generated as $(\boldsymbol{f} \bmod x^{n/2^{k+1}} - \zeta^{brv(2^k+i)}, \boldsymbol{f} \bmod x^{n/2^{k+1}} + \zeta^{brv(2^k+i)})$, where $brv()$ is the bit-reversal function over $k+1$ bits. The GS operation for inverse FFT is $c_i = \frac{1}{2}(c'_i + c''_i)$ and $c_{n/2+i} = \frac{1}{2}\zeta^{-n/2}(c'_i - c''_i)$. The Number Theoretic Transform (NTT), an integer-domain variant of FFT, computes the product $\boldsymbol{f}, \boldsymbol{g} \in R_q$ as $\boldsymbol{f} \cdot \boldsymbol{g} = \mathsf{INVNTT}(\mathsf{NTT}(\boldsymbol{f}) \odot \mathsf{NTT}(\boldsymbol{g}))$ where $\odot$ denotes point-wise multiplication.

D. Fast Fourier Sampling

Fast Fourier Sampling (ffSampling) is a key component of the Falcon signature algorithm, designed to efficiently generate short vectors $(\boldsymbol{s_1}, \boldsymbol{s_2})$ critical for signature generation. It leverages the trapdoor structure to optimize lattice operations

and combines FFT with recursion for efficient sampling. The input vector is decomposed via the Falcon tree's nodes, with a rightward depth-first traversal using trapdoor information to compute the target vector. Discrete Gaussian sampling occurs at the leaf nodes, and the error is propagated upwards to refine the result. The recursive implementation of ffSampling is shown in Algorithm 2. FFT optimizes vector splitting and merging, reducing high-dimensional operations to lower-dimensional subproblems, and significantly cutting computational complexity. splitfft decomposes a high-dimensional FFT result into two lower-dimensional FFTs using the GS butterfly transform, while mergefft combines two lower-dimensional FFT results into a complete FFT using the CT butterfly transform [3]. ffSampling efficiently utilizes trapdoor information and tree structures, ensuring the security and accuracy of the generated short vectors while preventing private key leakage.

Algorithm 2 The recursive version of ffSampling [3]

Input: $\mathbf{t} = (\boldsymbol{t_0}, \boldsymbol{t_1}) \in \mathsf{FFT}(\mathbb{Q}[x]/(x^n+1))^2$, a Falcon tree T
Output: $\mathbf{z} = (\boldsymbol{z_0}, \boldsymbol{z_1}) \in \mathsf{FFT}(\mathbb{Z}[x]/(x^n+1))^2$
1: **if** $n = 1$ **then**
2: $\quad \sigma \leftarrow \mathsf{T.value}$
3: $\quad \boldsymbol{z_0} \leftarrow \mathsf{SamplerZ}(\boldsymbol{t_0}, \sigma)$
4: $\quad \boldsymbol{z_1} \leftarrow \mathsf{SamplerZ}(\boldsymbol{t_1}, \sigma)$
5: $\quad$ **return** $\mathbf{z} = (\boldsymbol{z_0}, \boldsymbol{z_1})$
6: $(l, \mathsf{T}_0, \mathsf{T}_1) \leftarrow (\mathsf{T.value}, \mathsf{T.leftchild}, \mathsf{T.rightchild})$
7: $\mathbf{t_1} \leftarrow \mathsf{splitfft}(\boldsymbol{t_1})$
8: $\mathbf{z_1} \leftarrow \mathsf{ffSampling}_{n/2}(\mathbf{t_1}, \mathsf{T}_1)$ ▷ first recursive call
9: $\boldsymbol{z_1} \leftarrow \mathsf{mergefft}(\mathbf{z_1})$
10: $\boldsymbol{t_0'} \leftarrow \boldsymbol{t_0} + (\boldsymbol{t_1} - \boldsymbol{z_1}) \odot l$
11: $\mathbf{t_0} \leftarrow \mathsf{splitfft}(\boldsymbol{t_0'})$
12: $\mathbf{z_0} \leftarrow \mathsf{ffSampling}_{n/2}(\mathbf{t_0}, \mathsf{T}_0)$ ▷ second recursive call
13: $\boldsymbol{z_0} \leftarrow \mathsf{mergefft}(\mathbf{z_0})$
14: **return** $\mathbf{z} = (\boldsymbol{z_0}, \boldsymbol{z_1})$

E. Target Platform: GPU

GPU is a widely recognized parallel platform for accelerating computations. GPU features a higher number of computing cores, which affords them superior parallel computing capabilities. Instructions are executed through a thread flow, where threads are organized into thread blocks, and multiple independent blocks form a grid. The Streaming Multiprocessor (SM) serves as the primary unit responsible for executing the blocks of kernels. Additionally, GPUs encompass various types of memory, including register file (RF), constant memory (CMEM), shared memory (SMEM), global memory (GMEM), and local memory (LMEM). Efficient utilization of the GPU memory hierarchy facilitates more effective data access. The access speed is fastest for memory regions closest to the CUDA cores, including RF, L1 Cache, SMEM, and constant cache.

III. Compact and Efficient Operation-Level Design

In this section, we provide a comprehensive analysis of the current GPU implementation of Falcon and propose operation-level optimization strategies based on this analysis. First, we quantitatively identify issues such as suboptimal resource utilization and insufficient parallel design in existing implementations. To address these challenges, we propose a memory-efficient FFT implementation, a fully parallelized ffSampling strategy, and a compact computational design. Finally, these optimizations are integrated into the Falcon implementation to enhance its execution performance on GPU.

A. Analysis of Existing Implementation

To evaluate the performance of GPU-accelerated Falcon implementations, we conduct benchmark tests on the state-of-the-art implementation [13] using a desktop equipped with a 12th Gen Intel(R) Core(TM) i5-12400F CPU (with 2.5 GHz base frequency) and an NVIDIA GeForce RTX 4090 24GB GPU. In these tests, we batch process 1,024 tasks to analyze the time distribution of each module, record execution times in microseconds (μs), and evaluate the resource usage of each operation in the signing process. The time distribution results are shown in Fig. 1, where the computation of each kernel is labeled. The resource usage details are presented in Table I. Based on these test results, we present the following analysis.

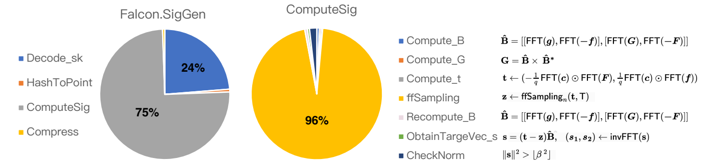


Fig. 1: The time distribution of Falcon-512 signature generation implemented in [13].

*1) **Inefficient Memory Access**:* Based on the annotations in Fig. 1, modules like Compute_B and ffSampling are frequently accelerated by FFT, making it a critical operation in Falcon signature generation. While FFT's computational operations, such as multiplications and additions, exhibit GPU-friendly parallelism, its hierarchical structure leads to frequent memory accesses, which create performance bottlenecks.

According to the data in Table I, the FFT_SMx4 kernel consumes 4,096 bytes of shared memory—significantly higher than other kernels (e.g., poly_copy uses 0 bytes). However, its memory throughput (22.95%) remains suboptimal, indicating potential for access pattern optimization. Therefore, we performed a micro-architecture level analysis of the FFT_SMx4 kernel using NVIDIA Nsight Compute. The analysis reveals 524,288 SMEM bank conflicts and uncoalesced GMEM accesses, the latter resulting in 12% of GMEM sectors being overutilized. These issues lead to inefficient memory access, creating performance bottlenecks during FFT computations. To address these issues, we propose an efficient memory access design for FFT, aiming to reduce memory access latency and thereby improve overall performance.

*2) **Suboptimal Design for GPU Architecture**:* As shown in Fig. 1, the ffSampling module accounts for 96% of the signature generation time, making it the most time-consuming component. Despite adopting an iterative approach to replace the original recursive implementation, which makes it more

TABLE I: Profiling of kernels in ComputeSig of [13].

Function	Count	Throughput (%)		Execution Time (µs)	SMEM (byte)	Register Number
		Compute	Memory			
smallints_to_fpr	8	23.1	49.92	4.16	0	16
FFT_SMx4	2	78.01	**22.95**	75.14	4096	40
poly_neg	4	14.38	66.72	6.56	0	16
poly_copy	13	5.43	66.45	6.53	0	16
poly_mulselfadj_fft	4	14.16	65.68	6.66	0	16
poly_muladj_fft	2	16.5	77	11.17	0	18
poly_add	5	8.57	78.93	10.98	0	16
poly_mul_fft	6	16.88	78.49	10.94	0	18
poly_mulconst	2	14.18	65.68	6.62	0	16
ffSampling_dyntree	1	67.8	52.33	**15750**	1000	196
iFFT	2	52.29	65.36	30.24	0	36
check1	1	90.2	14.19	227.55	0	30
check2	1	66.39	44.13	9.82	0	16
is_short_half	1	12.68	17.33	56.74	0	38

suitable for GPU architectures, there are still certain shortcomings:

- Coarse-grained task partitioning reduces parallel efficiency in ffSampling. It relies on single-threaded iterative calculations, with a task granularity that doesn't fully utilize the GPU parallel capabilities. Operations within each tree node, such as splitfft and mergefft, show high parallelism, but the tasks are not designed for parallel execution. Instead, operations are sequentially performed on the polynomial coefficients. This leads to large computational tasks within each node, which reduces the efficiency of tree node sampling and creates a bottleneck in the process.
- The varying polynomial sizes across tree levels cause imbalanced thread allocation in ffSampling. At the root node, the polynomial size is n, while at the leaf nodes, it reduces to 1. This disparity makes it difficult to set a uniform thread count for parallel computations. Excess threads at the leaf nodes result in idle resources, while insufficient threads at the root node limit parallel efficiency. Consequently, this imbalance hinders performance and reduces overall computational efficiency.

Therefore, we propose an adaptive parallel ffSampling scheme that is better suited to the GPU architecture.

3) ***Fragmented Computing Mode:*** From Table I, it can be seen that the signature computation process involves many small kernels, such as poly_neg, poly_copy and poly_add. These kernels have runtimes of less than 10 μs, do not use SMEM, and are primarily responsible for computations between GMEM or data exchanges between GMEM and registers. This indicates that their operations are simple, without involving intermediate results or complex computation processes. Among them, the 13 poly_copy kernels are used solely for data movement within GMEM, with simple operations.

However, this design has some drawbacks. First, since each small kernel needs to be launched and scheduled independently, it leads to frequent context switching and kernel invocations, which increases scheduling overhead on the GPU. Secondly, these small kernels perform relatively simple tasks and frequently access GMEM, which may become a performance bottleneck, especially when dealing with high-latency GMEM accesses that can reduce computational efficiency. Therefore, we propose a compact computing model that utilizes kernel fusion and optimized memory access strategies to mitigate the disadvantages of small kernels.

B. Fast Fourier Transform

This subsection presents key optimizations applied to the GPU-based FFT implementation, including loop unrolling, constant-time modular reduction, and efficient memory access techniques, aimed at enhancing memory access efficiency and computational performance.

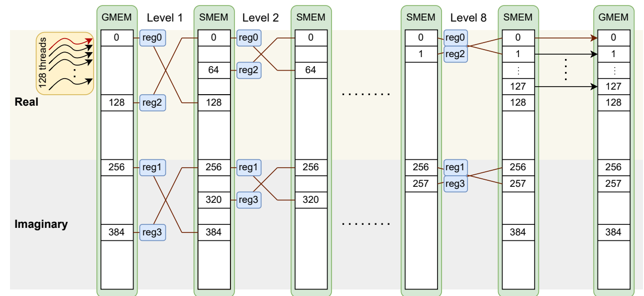
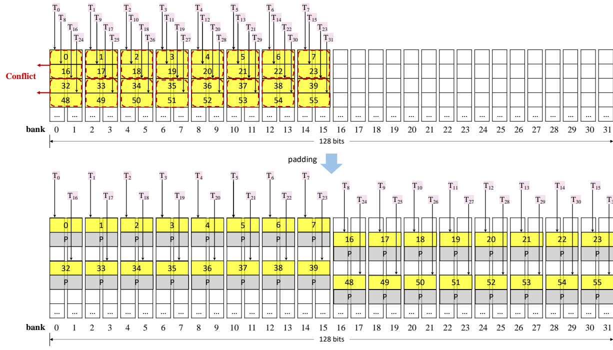


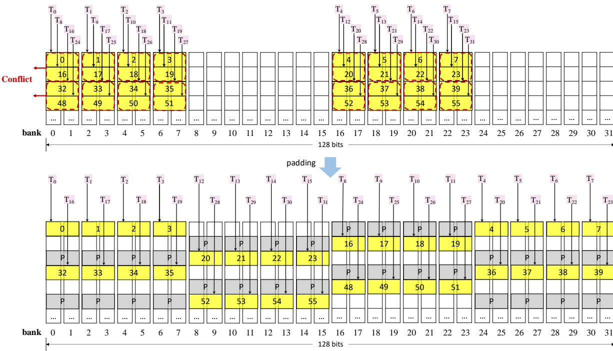
Fig. 2: The optimized implementation of FFT-512

1) ***Computational Workflow Design:*** In our FFT implementation for Falcon-512, the polynomial's real and imaginary parts each have a length of $n/2$, where n is the total length of the polynomial. The coefficients are first loaded into registers, where the butterfly transformation is performed. After completing each level of the butterfly transformation, data is exchanged between the registers and SMEM. This process is repeated iteratively until all levels are processed. To optimize performance, we implement several techniques, such as loop unrolling, which significantly enhances computational efficiency.

*2) **Conflict-Minimal Inter-Layer Data Shuffle**:* Bank conflicts occur when multiple threads within a warp access the same SMEM bank, leading to serialized memory accesses. This issue arises during the data shuffle phase of the FFT computation, especially with thread span access. To resolve this, we use blank unit padding to ensure threads access different SMEM banks. For the i-th level of FFT ($i \geq 3$), the thread stride is 2^{8-i}, and padding of 2^{7-i} blank units is introduced at intervals of 2^{8-i} to prevent conflicts. For Falcon with $n = 1024$ and $n = 512$, this padding requires 256 blank SMEM units. This optimization maximizes parallelism by avoiding bank conflicts and ensuring efficient memory access. Fig. 3 illustrates the data exchange in the 5th level of FFT-512, demonstrating how padding effectively resolves bank conflicts.



(a) Load (4th level).



(b) Store (4th level).

Fig. 3: Solving bank conflicts in SMEM. (P represents blank units.)

*3) **Burst Data Loading and Storing**:* In the FFT computation, loading from and storing to GMEM incurs high latency due to its slower speed. To improve efficiency, we align the memory of FFT polynomials to 128 bytes during device memory allocation, with the FFT data type set to double (64-bit). This enables merged memory accesses, improving bandwidth utilization by allowing hardware to transmit data in bursts. During FFT computation, data is read sequentially according to thread IDs for efficient loading. To prevent out-of-order storage when writing back to GMEM, we reorder the data in SMEM to enable merged memory access, as shown in Figure 2. This approach enables merged memory accesses, allowing data to be loaded or stored in bursts, which reduces GMEM access time.

C. *Adaptive Parallel* ffSampling *Algorithm*

In this subsection, we propose the adaptive parallel ffSampling algorithm, which uses a stack-based iterative strategy for tree traversal, overcoming the recursion limitations of the reference implementation. By leveraging intra-node parallelism and a layer-specific thread allocation method, we improve parallelism and adaptability across different tree layers. The design overview of the algorithm is shown in Fig. 4, using Falcon-512 as an example.

*1) **Stack-Based Explicit Iteration Strategy**:* To overcome recursion limitations, we use the iterative approach from [13] with explicit stack management for the stack-based ffSampling design. We initialize a stack of size $\log_2 n + 1$, corresponding to the maximum depth of the Falcon tree. The traversal starts at the root node and follows a depth-first search, prioritizing the right child of each node. As nodes are visited, their state, including associated polynomials and relevant information, is pushed onto the stack. Each node is processed from the top of the stack and popped after computation. Gaussian sampling occurs at leaf nodes, while polynomial operations are performed in FFT form at internal nodes. This iterative process continues, propagating data upward, sequentially updating each node's state, until reaching the root. By managing stack operations, the iterative implementation avoids recursion stack depth limitations and reduces function call overhead, while maintaining a time complexity of $O(n \log n)$.

*2) **Node-Level Parallel Computation**:* As described in Section III-C1, the tree traversal in the ffSampling process accesses nodes sequentially, making parallelization across nodes challenging. However, polynomial operations within each node, such as splitfft, mergefft, and sub, exhibit good parallelism, enabling parallel computation within individual nodes. On the GPU, we accelerate operations such as splitfft, mergefft, LDL_fft, mul_fft, and poly_sub using parallel design and memory optimization strategies. The splitfft and mergefft operations use FFT and inverse FFT subroutines to efficiently manage polynomial splitting and merging. As detailed in Section III-B, we implement an efficient memory access design for them. The LDL_fft and mul_fft operations convert polynomial decomposition and multiplication into point-to-point tasks, with each thread computing different polynomial coefficients. Polynomial addition and subtraction are also parallelized element-wise, where each thread handles a single coefficient. This parallelization enhances the execution of polynomial computations in ffSampling on the GPU, significantly reducing computational time. By decomposing tasks into independent units, this scheme enables efficient thread coordination and parallel execution, accelerating the tree node sampling process. By decomposing tasks into independent units, this scheme enables efficient thread coordination and parallel execution, accelerating the tree node sampling process.

*3) **Layer-Specific Thread Allocation**:* In the ffSampling implementation, the polynomial size reduces progressively with tree depth, requiring dynamic thread allocation adjustments at each level. As tree depth increases, the polynomial

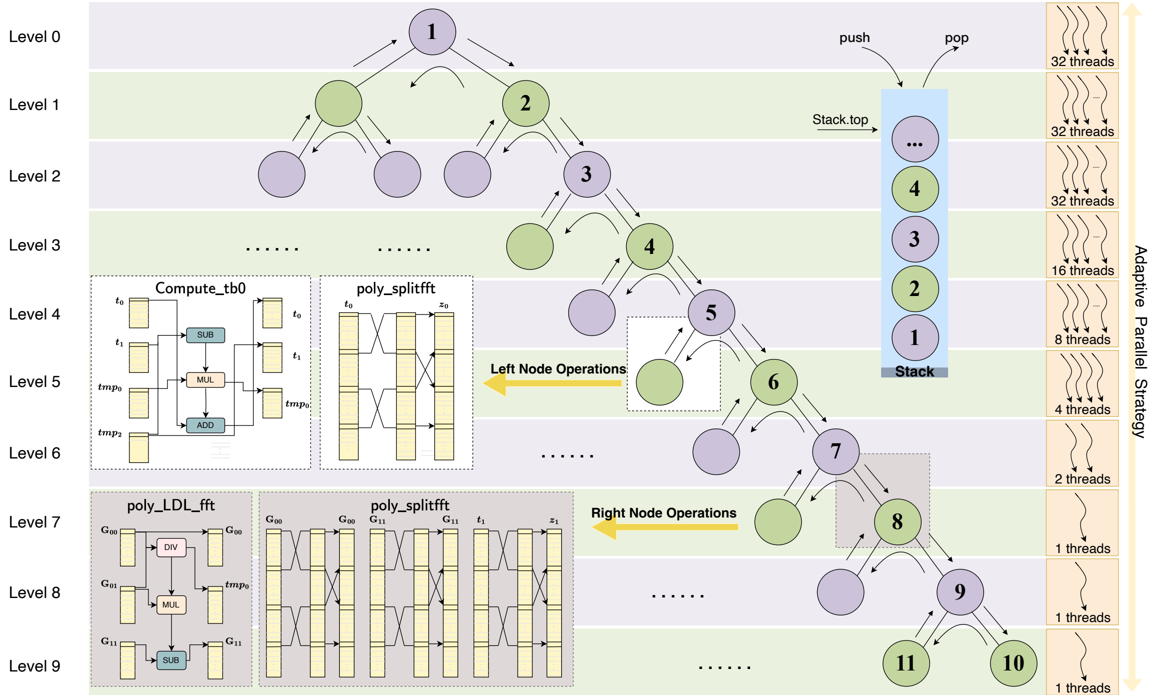


Fig. 4: Adaptive parallel strategy for traversing the Falcon tree in ffSampling.

size halves, reducing the parallelizable dimensions. For instance, the FFT subroutines in splitfft and mergefft utilize multi-threaded cooperation; assuming the polynomial size at the current level is k, up to $k/4$ threads can be allocated, with each thread responsible for computing the value of a single point. In contrast, the sub operation involves element-wise parallel subtraction, where up to k threads are allocated, each handling a single coefficient. Consequently, the maximum thread allocation for the current level is $k/4$.

We implement ffSampling as a single GPU kernel, with each thread required to execute tasks across multiple tree levels sequentially. Therefore, the thread allocation strategy must balance computational intensity and hardware resource utilization. Excessive thread allocation at smaller dimensions, such as at leaf nodes, would result in many idle threads, while insufficient allocation at larger dimensions, such as at the root node, would limit performance due to inadequate parallelism. Based on experimental results (see Section V-B2), the optimal configuration for the ffSampling kernel allocates one warp (32 threads, which is the minimum execution unit). As shown in Fig. 4, the number of active threads adapts with tree depth, optimizing resource utilization and computational efficiency.

D. Compact Computing Mode

Based on the analysis of the current Falcon GPU implementation [13], we identified inefficiencies due to a loose computing mode, leading to unnecessary kernel launch overhead and under-utilization of memory. To address these issues, we propose a compact computing model that utilizes kernel fusion and optimized memory management. This model is applied to three kernels: Compute_B, Compute_G_t, and TargetVec_s.

*1) **Kernel Fusion**:* In the implementation of signature generation, there are numerous invocations of small kernels, which will cause some kernel launch overhead. To mitigate this, we use kernel fusion, combining multiple kernels into one to reduce launch overhead. Data-related operations are merged into a single kernel to minimize redundant data access. For example, in Compute_B, small kernels like smallints_to_fpr, FFT_SMx4, and poly_neg are fused to compute matrix $\mathbf{B}$. Similar fusion is applied to other operations, as detailed in Table II. For the fused kernel, resource usage and execution time are optimized by adjusting the computation order and utilizing memory more efficiently, rather than simply summing the individual kernels' costs.

TABLE II: The proposed kernel fusion design.

Our work	Small kernels in [13]
compute_B_kernel	smallints_to_fpr, FFT_SMx4, poly_neg
compute_G_t_kernel	Compute_Gram_G: poly_mulselfadj_fft, poly_muladj_fft,poly_add,poly_copy
	Compute_t: poly_set, FFT_SM, poly_copy, poly_mul_fft, poly_mulconst
TargeVec_s_kernel	poly_mul_fft, poly_add, poly_copy, iFFT

*2) **Optimized Memory Utilization**:* After completing kernel fusion, we optimized the memory usage of the fused kernel. By consolidating data-dependent operations within a single kernel, intermediate results can be stored in SMEM or registers, thereby reducing GMEM access and improving memory access speed, as illustrated in Fig. 5. Additionally, within the same kernel, allocated SMEM or registers can be reused, eliminating the need for repeated allocation and deallocation across different kernels. Taking the poly_copy kernel as an

example, instead of performing data copying in GMEM, we copy data to temporary SMEM and directly compute within the device memory. This approach reduces GMEM usage and access frequency.

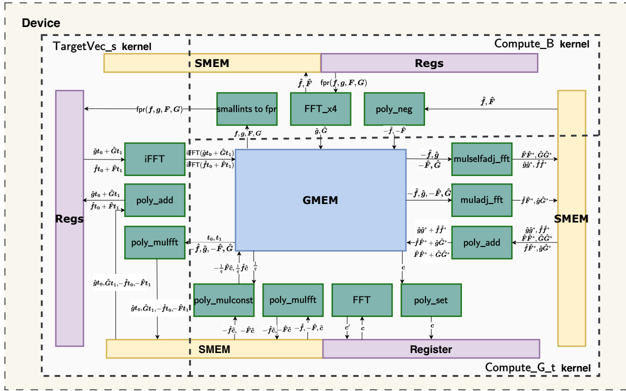


Fig. 5: The target vector $\mathbf{s}$ is computed in compact mode.

IV. High-Throughput and Space-Saving Signature-Level Implementation

In this section, we present three implementation strategies for Falcon signature generation, as shown in Fig. 6, and examine the parallel techniques used in handling multiple signature tasks. Existing methods, as elaborated in [3], [6], include a dynamic computation strategy (sign_dyn), which reduces storage overhead but increases computation costs, and a precomputation-based strategy (sign_tree), which precomputes private key operations for faster signing at the expense of higher storage. To balance storage requirements and computational efficiency, we propose a balanced implementation (sign_balance) that selectively precomputes key components while dynamically computing others, thereby reducing storage demands while maintaining high signing efficiency. Furthermore, for multi-signature task scenarios, we introduce a multi-task parallel processing strategy to improve throughput.

A. Dynamic computation-based implementation

To save storage space, we follow the reference implementation by adopting the raw key and dynamically reconstructing the LDL tree during private key loading. This way, we only need to store the path from the root to the current leaf, without keeping the entire tree in RAM. Fig. 6a presents an overview of the dynamic computation-based Falcon GPU acceleration implementation. This implementation follows a similar process to the reference implementation of the Falcon algorithm [20], but the original modules have been replaced with the optimized ones proposed in Section III. In this implementation, the key pair is dynamically generated, and all operations are computed in real-time, enhancing security. To optimize it for the GPU architecture, we significantly improve the signing computation speed and overall performance through parallel computing, memory access optimization, and efficient data transfer. The implementation details of the signature generation are as follows:

Message Hashing: In the message hashing phase, the message m is concatenated with a random salt $\boldsymbol{r}$ to form $(\boldsymbol{r}\|\mathsf{m})$, which is then transformed into the target polynomial $\boldsymbol{c} \in \mathbb{Z}_q[x]/(\phi)$ using the HashToPoint function. To accelerate this process on the GPU, we employ single warp parallelism, where different threads process different parts of the input message simultaneously. This design eliminates inter-thread synchronization overhead and ensures efficient computation. Intermediate results are stored in SMEM, reducing GMEM access latency. By combining warp-level parallelism with optimized memory access patterns, the GPU parallel computing capabilities are fully utilized, significantly reducing hash computation latency compared to traditional CPU-based serial operations.

Short vector computation: Short vector computation is a key part of signature generation, with the ffSampling algorithm playing a crucial role. First, we compute the preimage $\mathbf{t}$ of the message hash $\boldsymbol{c}$, utilizing the FFT acceleration proposed in Section III-B. Then, using private key information, the ffSampling algorithm recursively processes $\mathbf{t}$ in the Falcon tree structure to generate two short polynomials $\boldsymbol{s_1}$ and $\boldsymbol{s_2}$, which satisfy the relationship $\boldsymbol{s_1} + \boldsymbol{s_2}h = \boldsymbol{c} \mod q$. Finally, the adaptive parallel ffSampling approach from Section III-C generates the short vector. Additionally, by incorporating the compact computing mode from Section III-D, unnecessary data transfers and memory accesses are minimized during the computation.

Signature compression: The short polynomial $\boldsymbol{s_2}$ is compressed into a bitstring $\mathbf{str}$ using the Compress function. The compression process is designed in parallel, in which each thread maps individual coefficients of $\boldsymbol{s_2}$ to their respective positions in the target bitstring $\mathbf{str}$. Ultimately, the signature consists of the salt $\boldsymbol{r}$ and the signature bitstring $\mathbf{str}$, forming the signature pair $sig = (\boldsymbol{r}, \mathbf{str})$.

B. Precomputation-based Implementation

To improve the efficiency of the signature generation process, we designed a GPU-accelerated implementation based on a precomputation approach, as outlined in Figure 6b. This implementation follows the expanded key reference workflow of the Falcon algorithm but adopts an online-offline computation model with a fixed key design. To optimize the signature generation, computations related to the private key (such as the polynomial $\boldsymbol{G}$ calculation, matrix $\mathbf{B}$ construction, and LDL tree generation) are precomputed and used to generate the expanded key. The expanded key is stored in double-precision format and loaded on demand, reducing runtime computation overhead. In addition, we ensure efficient handling of complex operations in private key extension, such as FFT and LDL decomposition, further optimizing the signature process [21]. All other operations are still performed in real time to maintain the security of the implementation and the flexibility of the application.

Although the precomputation-based implementation significantly reduces the real-time computational workload and improves signing efficiency, it substantially increases memory usage, requiring $(8 \times \log n + 40) \times n$ bytes to store the expanded

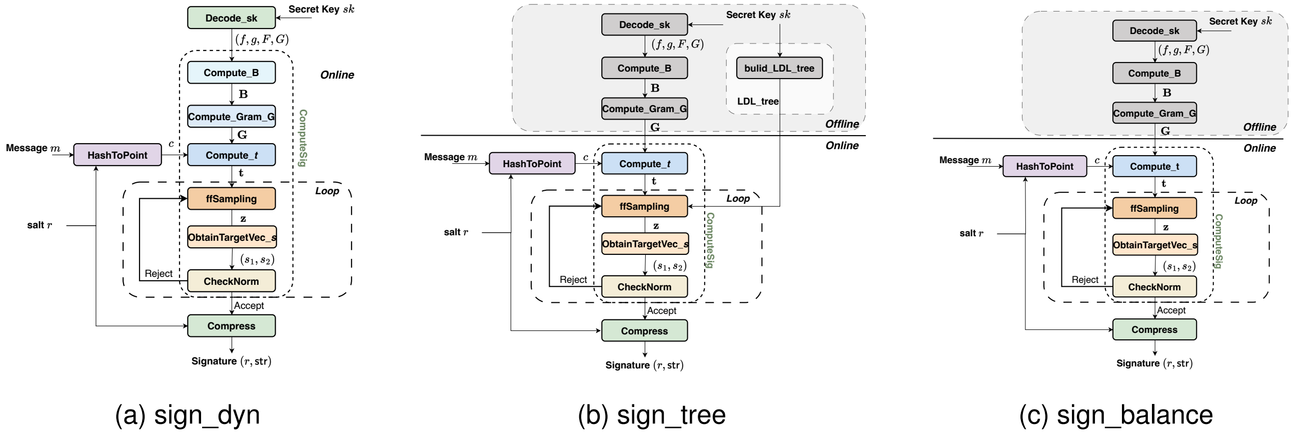


Fig. 6: The overview of the three designed Falcon signature generations.

key. For Falcon-512, one expanded key requires approximately 56 KB of storage space. Notably, the precomputed key-related data is stored in CPU memory and dynamically loaded into the GPU kernel as RAM input during runtime. This design ensures that sensitive private key-related data is securely retained in memory, preventing direct processing or storage by the compiler or disk, which aligns with standard security practices for handling cryptographic information. While this method may introduce a performance overhead due to additional memory access compared to embedding the data as constants within the kernel, it prioritizes security, offering a reliable solution for managing key-dependent data in cryptographic computations.

C. Balanced Efficiency-Space Implementation

To optimize memory usage and improve throughput, we propose a more balanced approach: precomputing the FFT representation of the $\mathbf{B}_0$ matrix and dynamically reconstructing the LDL tree during private key loading, as shown in Fig. 6c. The expanded key consists of the FFT representation of the $\mathbf{B}_0$ matrix and the LDL tree, where the $\mathbf{B}_0$ matrix occupies $4 \times 8 \times n$ bytes, and the LDL tree occupies $(\log n + 1) \times 8 \times n$ bytes, accounting for approximately 72% of the expanded key memory. Hence, the LDL tree is the primary source of memory overhead in the expanded key. During the signature generation process, the LDL tree is mainly used for ffSampling computations, while the primary computational overhead of ffSampling arises from the node-by-node traversal of the Falcon tree and node sampling calculations. Notably, the LDL tree contributes only a portion of the node sampling operations in ffSampling, and its construction is well-suited for GPU parallel computing. Therefore, dynamically loading the LDL tree onto the GPU enhances computational efficiency.

By dynamically reconstructing the LDL tree, multiple signature generation instances can reuse the tree, significantly reducing memory overhead. Only the path from the root to the current leaf node needs to be stored, eliminating the need to keep the entire tree structure in memory. Compared to the sign_tree method, this approach saves approximately 70% of the space, reducing the storage requirement for precomputed data to $4 \times 8 \times n$ bytes. As shown in Section V-C2, experimental results indicate that this method improves performance by 8% compared to sign_dyn, demonstrating an excellent balance between efficiency and memory usage. Most importantly, The expanded key version precomputes sensitive components (e.g., LDL tree), which risks exposure if keys are compromised. In contrast, the balanced version dynamically reconstructs critical structures during signing, reducing memory usage by 70% while maintaining 95% of the expanded key version's throughput.

D. Secure and Efficient Multi-Task Processing

In the scenario of multiple signature tasks, achieving efficient processing and secure access is the key design challenge. To address this, we adopt batching and streaming strategies to enhance the efficiency of handling multiple signature tasks. Additionally, we design an efficient and secure memory pool mechanism to ensure the correctness and security of multi-task computations.

*1) **Batching and Streaming**:* When batch processing multiple tasks, each signature is assigned to a GPU block, with multiple blocks running in parallel. During the computation of each signature task, we apply the optimization techniques mentioned earlier to ensure the full utilization of GPU resources within the block, achieving efficient computation.

Besides, data transfer between the CPU and GPU is a significant factor, as it incurs considerable latency. Therefore, we use multiple CUDA streams to hide the data transfer delay, as shown in Fig. 7, and asynchronously manage the data transfer. This approach ensures concurrent operation of the CPU and GPU, reducing overall data transfer latency.

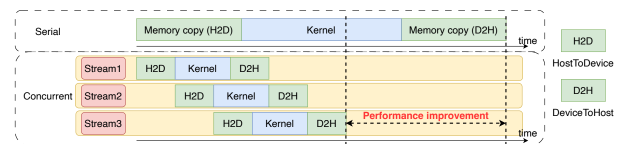


Fig. 7: Multiple streams can mask the time spent on data transmission.

*2) **Efficient and Secure Memory Pool:*** In order to ensure efficient and secure memory access during parallel multi-task processing, we employ the memory pool for memory management. The memory pool has two key features: the rational storage order of parameters and memory allocation alignment. Firstly, placing parameters of the same type and hash function input parameters in contiguous locations reduces the unnecessary concatenation overhead of bit streams. Second, we utilize pitch allocation to ensure block alignment, which helps reduce memory fragmentation and improve memory access efficiency. Additionally, we align memory blocks to 128 bytes, which is the size of L1 cache lines, enabling thread access to be merged into as few cache lines as possible, thus reducing the impact of stride access on memory access throughput. The memory pool designed for signature generation is illustrated in Fig. 8.

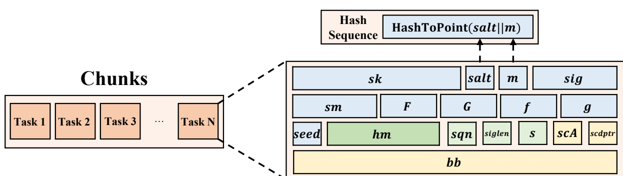


Fig. 8: The design of the memory pool for signature generation.

V. Performance Evaluation

In this section, we evaluate the operation optimization proposed in Section III to demonstrate the effectiveness of optimization techniques. Next, we assess the performance of the three implementations of cuFalcon introduced in Section IV, analyzing their throughput and latency, and comparing the results with related works.

A. Experimental Setup

The C/C++ source code is compiled using g++ 11.4.0, while the GPU implementation is compiled using CUDA 12.0. All compilation and execution processes are conducted on the Ubuntu 22.04 operating system. The CPU benchmark performance is based on the 12th generation Intel(R) Core(TM) i5-12400F CPU. GPU performance benchmarking was conducted on desktop-grade GPU NVIDIA GeForce RTX 4090 and server-grade GPU NVIDIA Tesla A100 80G PCIe. The RTX 4090 is based on the Ada Lovelace GPU architecture, while the A100 is based on the Ampere architecture.

B. Evaluating Operation Optimization Techniques

This subsection mainly focuses on testing the operation-level implementations in the Falcon signature generation, aiming to evaluate the effectiveness of the proposed optimization techniques.

*1) **Evaluation of FFT:*** To validate the effectiveness of our proposed FFT optimization techniques, we conducted step-by-step optimization tests, progressively applying fine-tuning of the computational workflow, conflict-minimal inter-layer data shuffle, and burst data loading and storing. We conducted benchmarking for FFT computation with a batch size of 1024, which included both FFT512 and FFT1024. The performance results are illustrated in Fig. 9. It can be observed that with the application of each optimization technique, the computation time of the FFT decreases. The final execution time of FFT-512 decreased by 11.73%, while FFT-1024 decreased by 4.21%.

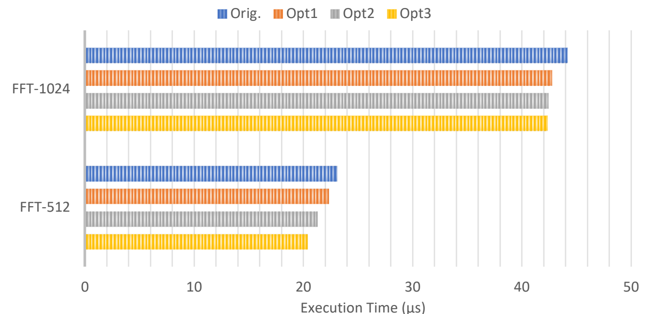


Fig. 9: The comparison results of FFT execution time between t the baseline and the step-by-step optimizations.

*2) **Evaluation of Adaptive Parallel** ffSampling:* We test the adaptive parallel ffSampling algorithm with a batch size of 1024. First, we explored the optimal thread allocation by adjusting the number of parallel threads, and the results are shown in Fig. 10. Since a warp is the smallest unit of scheduling by the GPU hardware, it facilitates unified management and parallel computation. As the number of threads within a warp increases, the kernel throughput continues to improve. However, once the number of threads exceeds one warp, it typically leads to expensive synchronization overhead.

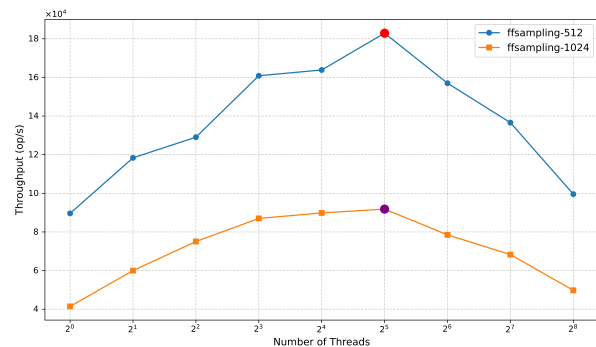


Fig. 10: The throughput results of the adaptive parallel ffSampling algorithm with different thread configurations.

Next, we test the throughput, execution time, and resource utilization of the adaptive parallel ffSampling kernel, and the results are provided in Table III. Since the baseline implementation of Falcon-1024 is not open-sourced, we only test ffSampling-512 from [13]. Taking Falcon-512 as an example, our work improves throughput by 16.89%, indicating that

the parallel design makes ffSampling more efficient while reducing unnecessary memory accesses, leading to a 17.10% decrease in memory throughput. Moreover, the execution time is reduced by 59.88%, demonstrating a significant improvement in overall computational efficiency and better alignment with the GPU architecture. Regarding resource utilization, the change is minimal because no additional GPU memory resources are used and the ffSampling algorithm is simply adapted structurally to the GPU. Notably, both our ffSampling-512 and ffSampling-1024 achieve a computation throughput of approximately 80%, demonstrating high-throughput performance. In summary, the adaptive parallel ffSampling algorithm enhances computational efficiency through parallel design, reduces unnecessary memory access, and does not increase memory usage, fully leveraging the GPU computational power.

TABLE III: Evaluation of adaptive parallel ffSampling compared with the work [13].

		Throughput (%)		Execution	Occupancy (%)	
		Compute (SM)	Memory	Time(ms)	Theoretical	Achieved
Lee et al. [12]	ffSampling-512	68.03	52.52	15.78	17.67	16.52
Our work	ffSampling-512	79.52 **(+16.89%)**	43.54 **(-17.10%)**	5.59 **(-64.58%)**	16.67 **(-5.66%)**	16.45 **(-0.42%)**
	ffSampling-1024	80.48	44.83	11.15	16.67	16.5

*3) **Evaluation of Compact Computing Mode**:* We test the individual operations of the compact computing design (batch_size=1024), with the execution time results for each kernel listed in Table IV. As shown in Table IV, the compact design significantly improves performance compared to the previous implementation. Specifically, the computation matrix B achieves a 1.81× performance improvement, the efficiency of compute_G_t is enhanced by 3.72×, and the computation performance of vector s improves by 2.61×. These operations cover key steps in Falcon signing, thereby effectively reducing the overall signing time.

TABLE IV: Evaluation of compact computing model

Lee et al. [13]				Our Work	
Small Kernels	Invocation Count	Execution Time (μs)	Total Time (μs)	Kernel	Total Time (μs)
smallints_to_fpr	4	8.0	126.3	compute_B	**69.6 (1.81×)**
FFT_SMx4	1	71.7			
poly_neg	2	11.3			
poly_copy	5	6.56	199.45	compute_G_t	**53.66 (3.72×)**
poly_mulselfadj_fft	4	6.75			
poly_muladj_fft	4	11.01			
poly_add	3	10.94			
poly_set	1	4.13			
FFT_SM	1	23.14			
poly_mul_fft	2	11.07			
poly_mulconst	2	6.69			
poly_mul_fft	4	11.26	80.04	TargeVec_s	**30.66 (2.61×)**
poly_add	2	10.91			
poly_copy	2	6.59			

C. Performance of cuFalcon

In this section, we evaluate the performance of the three signature implementations proposed in Section IV, testing both resource usage and throughput to validate the effectiveness of our proposed approach.

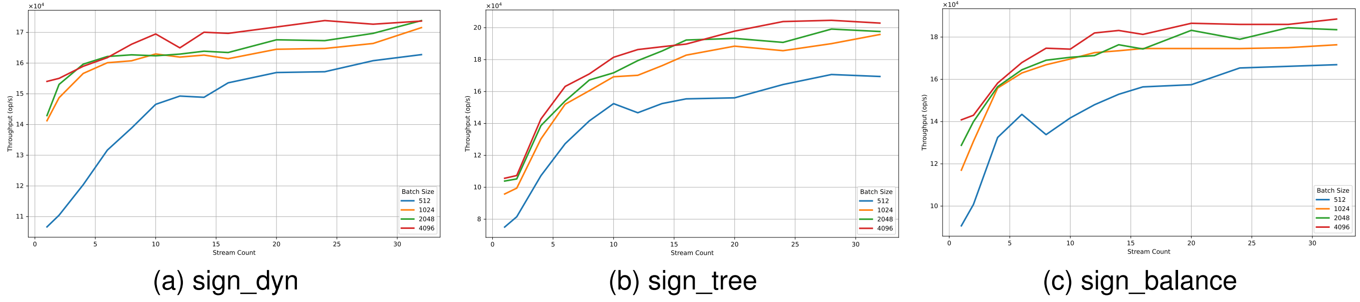
*1) **Resource Utilization Analysis**:* We perform kernel profiling of the three signature design implementations proposed in Section IV to verify the rationality of kernel resource allocation during the signature generation process. The number of signature tasks is set to 10,000, and the profiling data for key kernels are shown in Table V. The results indicate that the memory or computation throughput of these four kernels exceeds 80%, indicating high-throughput implementations. Except for ffSampling, which is constrained by the complex data structures of the Falcon tree, the other core kernels achieve balanced resource utilization, with theoretical utilization reaching 100% and actual occupancy exceeding 80%, suggesting well-planned resource allocation.

TABLE V: Resource utilization of core functions

Function	Throughput (%)		Occupancy (%)		SMEM	Register
	Compute	Memory	Theoretical	Achieved	(byte)	Number
FFT_polyx4	**85.82**	45.39	**100**	96.7	6144	40
Compute_G_t	59.1	**87.26**	**100**	97.71	4096	40
ffsampling	**81.95**	45.75	16.67	16.53	1000	198
Targevec_s	23.4	**93.16**	**100**	86.96	0	36

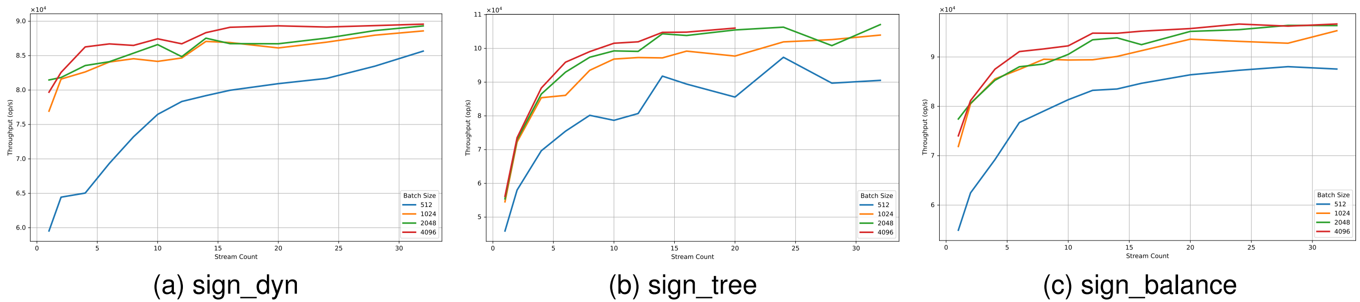
*2) **Execution Efficiency Analysis**:* First, we experimentally evaluate the effect of varying CUDA stream counts on the signature scheme throughput to determine the optimal number of streams. Fig. 11 and 12 show the throughput of the multi-CUDA-stream signature implementation under different batch sizes. As observed from the results, throughput increases with the number of CUDA streams across different batch sizes and eventually stabilizes when the number of streams reaches 16. Consequently, we set the number of CUDA streams for our implement to 16. Additionally, we observe that the throughput of sign_tree is slightly lower than that of sign_dyn with fewer CUDA streams. However, as the number of streams increases, the data transfer between the CPU and GPU in sign_tree becomes better hidden, gradually improving throughput, eventually surpassing that of sign_dyn.

Next, with the CUDA stream count set to 16, we test the three signature implementations proposed in Section IV and verify on the desktop GPU RTX4090 and the server-level GPU 100. We compared the results with the latest implementation [13]. The test results, including the time for CPU-GPU data transfer, are shown in Table VI and Table VII. Based on Table VI and Table VII, our proposed implementations demonstrate significant throughput improvements as the batch size increases. On the A100 platform, the sign_dyn implementation achieves a throughput of 172k signatures per second for Falcon-512 and 90k signatures per second for Falcon-1024, with speedups of 2.94× and 2.41× over [13]. For the sign_tree implementation, the throughput reaches 201k and 100k signatures per second for Falcon-512 and Falcon-1024, respectively, reflecting approximately 17% and 11% improvements over the sign_dyn implementation. In the case of sign_balance, the throughput achieves 184k and 96k signatures per second for Falcon-512 and Falcon-1024, respectively. Although slightly lower than the sign_tree implementation, the throughput still improves by approximately 7% for both Falcon-512 and Falcon-1024 compared to sign_dyn. Additionally, the sign_balance implementation



(a) sign_dyn (b) sign_tree (c) sign_balance

Fig. 11: Research on the impact of varying stream counts on cuFalcon-512 throughput across multiple batch sizes.



(a) sign_dyn (b) sign_tree (c) sign_balance

Fig. 12: Research on the impact of varying stream counts on cuFalcon-1024 throughput across multiple batch sizes.

TABLE VI: Throughput of cuFalcon compared with the work [13] on RTX 4090.

Algorithm	Batch Size	Our Work (op/s)				Lee et al. [13][1](op/s)	
		Sign_dyn[2]	Sign_tree[3]	Sign_balance[3]	Verify[2]	Sign	Verify
Falcon-512	256	130695 (8.46×)	120826(-7.55%)	137099 (+4.90%)	1133245 (1.46×)	15454	773648
	512	153580 (5.28×)	155416(+1.20%)	156421 (+1.85%)	1843053 (1.56×)	29042	1184637
	1024	161434 (3.25×)	182795 (+13.23%)	174581 (+8.14%)	2666667 (1.66×)	49667	1606527
	2048	163447 (3.02×)	192267 (+17.63%)	174360 (+6.68%)	3311772 (1.52×)	54080	2172944
	4096	169690 (2.64×)	189745(+11.82%)	181263 (+6.82%)	3535912 (1.47×)	64294	2400375
	8192	172223 (2.71×)	**195732 (+13.65%)**	**186043 (+8.02%)**	3846551 (1.45×)	63654	2650104
	16384	**172111 (2.54×)**	—	185228 (+7.62%)	**3879247 (1.39×)**	67686	2784216
Falcon-1024	256	73471	71309 (-2.94%)	81025 (+10.28%)	131430 (2.49×)	—	—
	512	79969	89432 (+11.83%)	84670 (+5.88%)	254777 (4.83×)	—	—
	1024	86899	99195 (+14.15%)	91273 (+5.03%)	4780356(9.05×)	—	—
	2048	86738	103828 (+19.70%)	92493 (+6.63%)	876074 (16.59×)	—	—
	4096	**89115**	**104822 (+17.63%)**	95229 (+6.86%)	1556881 (29.49×)	—	—
	8192	89002	—	**95597 (+7.41%)**	**1925128 (36.46×)**	—	—

[1] The Falcon-512 implementation from [13] is open source, and the data in this table are obtained by running it on our test platform. The code is available at https://github.com/benlwk/Falcon-Mitaka.
[2] The values in parentheses indicate the performance improvement relative to the implementation in [13].
[3] The values in parentheses represent the performance improvement relative to our sign_dyn implementation.

significantly reduces memory usage for larger batch sizes, showcasing higher resource efficiency. Furthermore, the verify achieves throughput of 3259k and 2372k signatures per second for Falcon-512 and Falcon-1024, respectively, with speedups of 1.21× and 1.19× over [13]. The improved verification throughput is mainly due to the multi-stream design, which effectively hides CPU-GPU memory transfer latency.

*3) **Comparison of Related Work**:* Table VIII lists the throughput results derived from several related studies [3], [6]–[9], targeting different platforms. The specification for Falcon provides a C implementation and an AVX2-optimized implementation. Compared to the reference implementation, our work achieves speedups of 36×, 26×, and 73× for sign_dyn, sign_tree, and verify, respectively in Falcon-512. For Falcon-1024, the speedups are 38×, 28×, and 74× for sign_dyn, sign_tree, and verify, respectively. The study [7] presents a speed record on ARMv8 architecture, using the SIMD extension NEON. The work [6] proposes optimized implementations on a small microcontroller. The study [8] focuses on FPGA-centric hardware design, dedicated to achieving enhanced performance with fewer hardware resources. The study [9] proposes an efficient FALCON accelerator, EFX,

TABLE VII: Throughput of cuFalcon compared with the work [13] on A100.

Algorithm	Batch Size	Our Work (op/s)				Lee et al. [13][1] (op/s)	
		Sign_dyn[2]	Sign_tree[3]	Sign_balance[3]	Verify[2]	Sign	Verify
Falcon-512	256	99708 (3.83×)	106694 (+7.01%)	107547 (+7.86%)	657084 (1.04×)	26037	629287
	512	130094 (2.94×)	133681 (+2.76%)	132557 (+1.89%)	1163108 (1.08×)	44229	1078854
	1024	149190 (3.17×)	171740 (+15.12%)	159496 (+6.91%)	1907956 (1.13×)	47029	1688585
	2048	162017	190844 (+17.79%)	172532 (+6.49%)	2704702	—	—
	4096	168964 (3.04×)	198498 (+17.48%)	180739 (+6.97%)	2908471 (1.21×)	55505	2399110
	8192	172022	201062 (+16.88%)	184050 (+6.99%)	**3259460**	—	—
	16384	**172263 (2.94×)**	**201545 (+17.00%)**	**184371 (+7.03%)**	2787295 (1.02×)	58595	2721562
Falcon-1024	256	62544 (4.98×)	63396 (+1.36%)	61493 (-1.68%)	347420 (1.02×)	12556	341587
	512	76070 (2.98×)	80298 (+5.56%)	79052 (+3.92%)	684372 (1.18×)	25513	578306
	1024	82373 (3.26×)	92586 (+12.40%)	87286 (+5.96%)	1282579 (1.34×)	25272	956023
	2048	87789	96325 (+9.72%)	93620 (+6.64%)	1573448	—	—
	4096	89480 (2.00×)	99647 (+11.36%)	96529 (+7.88%)	2023367 (1.19×)	44680	1916189
	8192	90511	**100292 (+10.81%)**	**96997 (+7.17%)**	2282225	—	—
	16384	**90636 (2.41×)**	—	96731 (+6.73%)	**2372579 (1.13×)**	37550	2092758

[1] The data in this table are sourced from the results in [13].
[2] The values in parentheses indicate the performance improvement relative to the implementation in [13].
[3] The values in parentheses represent the performance improvement relative to our sign_dyn implementation.

TABLE VIII: Throughput of Falcon implementation on different platforms.

Related Work	Platform	Falcon-512 (op/s)			Falcon-1024 (op/s)		
		sign_dyn	sign_tree	verify	sign_dyn	sign_tree	verify
Reference Implementation [3]	Intel i5-12400F CPU (C)	4687	7335	52798	2343	3698	25860
	Intel i5-12400F CPU (AVX2)	5350	8167	55772	2668	4113	27412
Nguyen et al. [7]	Jetson AGX Xavier	5645	—	99476	2831	—	51771
	Apple M1	7240	—	140969	3628	—	74592
	Raspberry Pi 4 with Cortex-A72	1797	—	30612	879	—	13804
Pornin et al. [6]	ARM Cortex-M4	4	8	333	2	4	163
Schmid et al. [8]	Zynq UltraScale+ FPGA	—	238	1618	—	114	795
Lee et al. [9]	Cortex-M4 + ASIC	26	—	—	12	—	—

based on an HW/SW co-design, optimizing operations with granular hardware and software improvements to enhance performance and reduce silicon area usage.

VI. Conclusion

This paper introduces cuFalcon, a high-throughput GPU implementation of the NIST post-quantum signature algorithm Falcon. Through quantitative and qualitative analyses of existing implementations, we identify key optimization opportunities and design compact and efficient solutions for critical operations in signature generation. Besides, building on a detailed analysis of the signature generation process, we propose the balanced efficiency-space implementation method, which significantly reduces memory usage while maintaining high throughput. Then, we develop three GPU-based signature generation schemes to cater to different requirements. Finally, experimental results demonstrate that the proposed techniques effectively improve throughput, validating the effectiveness of our approach. cuFalcon achieves 201k signatures per second for Falcon-512 on A100 GPUs—a 3.44× speedup over the state-of-the-art [13]. The balanced version further reduces memory usage by 70%, making it viable for memory-constrained edge devices.

References

[1] P. W. Shor, "Algorithms for quantum computation: Discrete logarithms and factoring," in *35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, 20-22 November 1994*. IEEE Computer Society, 1994, pp. 124–134.

[2] G. Alagic, D. Cooper, Q. Dang, T. Dang, J. M. Kelsey, J. Lichtinger, Y.-K. Liu, C. A. Miller, D. Moody, R. Peralta, R. Perlner, A. Robinson, D. Smith-Tone, and D. Apon, "Status report on the third round of the nist post-quantum cryptography standardization process," 2022-07-05 04:07:00 2022.

[3] P.-A. Fouque, J. Hoffstein, P. Kirchner, V. Lyubashevsky, T. Pornin, T. Prest, T. Ricosset, G. Seiler, W. Whyte, Z. Zhang *et al.*, "Falcon: Fast-fourier lattice-based compact signatures over ntru," *Submission to the NIST's post-quantum cryptography standardization process*, 2022. [Online]. Available: https://falcon-sign.info/falcon.pdf

[4] N. W. Matthieu Grillere, Peter Thomassen, "Falcon-512 in powerdns," Online, April 07 2022. [Online]. Available: https://blog.powerdns.com/2022/04/07/falcon-512-in-powerdns

[5] O. Q. S. O. project, "liboqs (release 0.7.2)," Online, 2022. [Online]. Available: https://github.com/open-quantum-safe/liboqs

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

[7] D. T. Nguyen and K. Gaj, "Fast falcon signature generation and verification using armv8 NEON instructions," in *Progress in Cryptology - AFRICACRYPT 2023 - 14th International Conference on Cryptology in Africa, Sousse, Tunisia, July 19-21, 2023, Proceedings*, ser. Lecture Notes in Computer Science, N. E. Mrabet, L. D. Feo, and S. Duquesne, Eds., vol. 14064. Springer, 2023, pp. 417–441.

[8] M. Schmid, D. Amiet, J. Wendler, P. Zbinden, and T. Wei, "Falcon takes off - A hardware implementation of the falcon signature scheme,"

IACR Cryptol. ePrint Arch., p. 1885, 2023. [Online]. Available: https://eprint.iacr.org/2023/1885

[9] Y. Lee, J. Youn, K. Nam, H. H. Jung, M. Cho, J. Na, J.-Y. Park, S. Jeon, B. G. Kang, H. Oh, and Y. Paek, "An efficient hardware/software co-design for falcon on low-end embedded systems," *IEEE Access*, vol. 12, pp. 57 947–57 958, 2024.

[10] S. Shen, H. Yang, W. Dai, H. Zhang, Z. Liu, and Y. Zhao, "High-throughput gpu implementation of dilithium post-quantum digital signature," *IEEE Trans. Parallel Distrib. Syst.*, vol. 35, no. 11, p. 1964–1976, Sep. 2024. [Online]. Available: https://doi.org/10.1109/TPDS.2024.3453289

[11] S. Shen, H. Yang, W. Li, and Y. Zhao, " cuML-DSA: Optimized Signing Procedure and Server-Oriented GPU Design for ML-DSA ," *IEEE Transactions on Dependable and Secure Computing*, no. 01, pp. 1–12, Nov. 5555. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/TDSC.2024.3494835

[12] 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.

[13] W. Lee, R. K. Zhao, R. Steinfeld, A. Sakzad, and S. O. Hwang, "High throughput lattice-based signatures on gpus: Comparing falcon and mitaka," *IEEE Trans. Parallel Distributed Syst.*, vol. 35, no. 4, pp. 675–692, 2024.

[14] C. Gentry, C. Peikert, and V. Vaikuntanathan, "Trapdoors for hard lattices and new cryptographic constructions," in *Proceedings of the 40th Annual ACM Symposium on Theory of Computing, Victoria, British Columbia, Canada, May 17-20, 2008*, C. Dwork, Ed. ACM, 2008, pp. 197–206.

[15] D. Boneh, Ö. Dagdelen, M. Fischlin, A. Lehmann, C. Schaffner, and M. Zhandry, "Random oracles in a quantum world," in *Advances in Cryptology - ASIACRYPT 2011 - 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings*, ser. Lecture Notes in Computer Science, D. H. Lee and X. Wang, Eds., vol. 7073. Springer, 2011, pp. 41–69.

[16] A. Chailloux and T. Debris-Alazard, "Tight and optimal reductions for signatures based on average trapdoor preimage sampleable functions and applications to code-based signatures," in *IACR International Conference on Public-Key Cryptography*. Springer, 2020, pp. 453–479.

[17] R. del Pino, V. Lyubashevsky, and D. Pointcheval, "The whole is less than the sum of its parts: Constructing more efficient lattice-based akes," in *Security and Cryptography for Networks - 10th International Conference, SCN 2016, Amalfi, Italy, August 31 - September 2, 2016, Proceedings*, ser. Lecture Notes in Computer Science, V. Zikas and R. D. Prisco, Eds., vol. 9841. Springer, 2016, pp. 273–291.

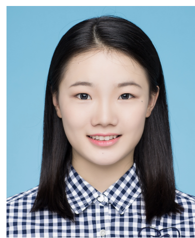
[18] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," *Mathematics of computation*, vol. 19, no. 90, pp. 297–301, 1965.

[19] W. M. Gentleman and G. Sande, "Fast fourier transforms: For fun and profit," in *Proceedings of the November 7-10, 1966, Fall Joint Computer Conference*, ser. AFIPS '66 (Fall). New York, NY, USA: Association for Computing Machinery, 1966, p. 563–578.

[20] F. Signature, "Falcon signature algorithm," 2024, accessed: 2024-11-26. [Online]. Available: https://falcon-sign.info/

[21] D. Micciancio and O. Regev, "Lattice-based cryptography," in *Post-quantum cryptography*. Springer, 2009, pp. 147–191.

Wenqian Li is currently a PhD candidate at the School of Computer Science, Fudan University. Her research focuses on post-quantum cryptography and cryptographic engineering.



Hanyu Wei is currently a PhD candidate at the School of Computer Science, Fudan University. Her research focuses on post-quantum cryptography and cryptographic engineering.

Shiyu Shen received the PhD degree from School of Computer Science, Fudan university in 2024. She is currently a postdoctoral fellow at City University of Hong Kong. Her research interests include lattice-based cryptography, homomorphic encryption, and cryptographic engineering. Her email address is crypto@sher1e.dev.

Hao Yang received the PhD degree from College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics in 2024. He is currently a postdoctoral fellow at City University of Hong Kong. His research interests include homomorphic encryption, lattice-based cryptography, and cryptographic engineering. His email address is crypto@d4rk.dev.

Wangchen Dai received the B.Eng. degree in electrical engineering and automation from Beijing Institute of Technology, China, in 2010, the M.A.Sc. degree in electrical and computer engineering from the University of Windsor, Canada, in 2013, and the Ph.D. degree in electronic engineering from the City University of Hong Kong in 2018. After completing the Ph.D. study, he had appointments at Hardware Security Lab, Huawei Technologies Company Ltd., in 2018, and the Department of CSSE, Shenzhen University in 2020, respectively. His research interests include cryptographic hardware and embedded systems, fully homomorphic encryption, and reconfigurable computing.

Yunlei Zhao received his PhD at Fudan University in 2004. He is now a distinguished professor at Fudan university. His main research interests include post-quantum cryptography, cryptographic protocols, theory of computing. His email address is ylzhao@fudan.edu.cn.