HIGH: Harnessing GPU Parallelism for Optimized HQC Performance

Jiankuo Dong*, Yuze Hou*, Shiqin Wang*, Letian Sha*, Fu Xiao*, Zhenjiang Dong* and Jingqiang Lin†
*School of Computer Science, Nanjing University of Posts and Telecommunications
†School of Cyber Science and Technology, University of Science and Technology of China

***Abstract*—Hamming Quasi-Cyclic (HQC) was a candidate algorithm in the fourth round of the National Institute of Standards and Technology (NIST) post-quantum cryptography (PQC) standardization process and was ultimately selected as a standardized cryptographic scheme in the latest round. To date, although HQC has been optimized for FPGA, CPU, and other platforms, research on GPU-based parallel acceleration remains significantly underexplored. Given this, our research aims to investigate the feasibility and efficiency of hand-optimized HQC implementations on GPU, addressing the current research gap in GPU-based implementations of code-based cryptographic algorithms. In this paper, we introduce a High-performance Implementation of GPU-based HQC, named HIGH. First, we propose a novel architecture for coding-based PQC implementations, significantly reducing redundant global memory access through core fusion. Second, we design a HIKD structure for HIGH, combining a high-order multiplication scheme (HIK) and a low-order multiplication scheme (HID), achieving an impressive 85.6% improvement over Official fourth round Optimized Implementation. Third, through extensive experimentation, we sought the optimal HIKD combination and identified the optimal parallel parameters, leveraging these optimizations, HIGH surpasses current state-of-the-art benchmarks, with Key Generation, Encapsulation, and Decapsulation performance increased by 20×, 32×, and 39×, respectively.**

***Index Terms*—PQC, HQC, Karatsuba, Polynomial Multiplication, GPU**

I. Introduction

The rapid progression of quantum computing technology poses a formidable challenge to the security of classical cryptographic systems. Quantum algorithms, such as Shor algorithm [1], are capable of efficiently solving the integer factorization and discrete logarithm problems upon which RSA and ECC cryptosystems rely, thereby compromising their security in a post-quantum context. This looming threat has galvanized the cryptographic community and institutions like the National Institute of Standards and Technology (NIST) to initiate efforts towards developing and standardizing post-quantum cryptographic (PQC) algorithms [2]. During the fourth round of the NIST PQC standardization process, several candidate algorithms—including Classic McEliece [3], BIKE [4], HQC [5], and SIKE [6]—were closely evaluated for their resistance to quantum attacks and practical performance. As the assessment progressed, distinct limitations emerged: SIKE was compromised by key-recovery attacks, leading its team to declare it insecure [7]; BIKE lacked sufficient maturity in decryption failure rate analysis, undermining its security foundation; and Classic McEliece, despite its theoretical strength, was hindered by impractically large public key sizes. After multiple rounds of rigorous evaluation, NIST officially selected the Hamming Quasi-Cyclic (HQC) scheme as a standardized key encapsulation mechanism in March 2025.

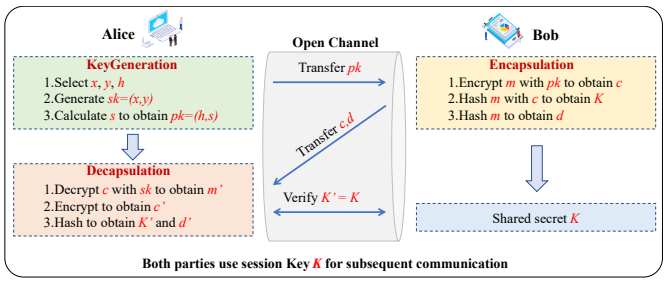


Fig. 1: The KEM Structure of HQC

As shown in Figure 1, HQC is primarily utilized for Key Encapsulation Mechanisms (KEM), a cryptographic technique designed for the secure distribution of symmetric keys. Despite its strong security guarantees, a significant bottleneck in the practical implementation of HQC is its performance. The computational complexity of polynomial multiplication, particularly with large key sizes, presents substantial challenges. With the rapid growth of IoT devices and user terminals, the demand for large-scale request processing, especially in cloud computing, is intensifying. This surge underscores the need for high-performance cryptographic implementations that prioritize high throughput to efficiently handle massive concurrent operations. This paper focuses on improving the throughput performance of cryptographic algorithms based on multi-core GPU architectures. The term "performance" mentioned in this paper primarily refers to the throughput of the HQC.

A. Related Works

In the process of standardization, the performance and implementation efficiency of PQC algorithms are critical in their selection [8]. To accelerate the HQC cryptographic algorithm, researchers pursued optimizations across hardware and software levels, with a particular focus on the computational bottleneck of polynomial multiplication in the ring. Sanjay Deshpande et al. [9] presented a hardware design for HQC on the Xilinx Artix 7 FPGA, showcasing a constant-time implementation that optimizes key components such as polynomial multiplication and polynomial addition/subtraction. In parallel, Schöffel et al. [10] introduced a comprehensive

hardware/software (HW/SW) co-design for HQC, targeting both ASIC and FPGA platforms. Their methodology involved profiling the reference code, extending a custom RISC-V processor with instruction set extensions, and integrating loosely-coupled accelerators for functions like polynomial multiplication, demonstrating viability for low-power IoT applications. Beyond dedicated hardware and co-design, other implementation strategies have been explored. Carlos Aguilar-Melchor et al. [11] demonstrated rapid prototyping of the HQC algorithm using high-level synthesis(HLS) tools. In the realm of software-level optimizations, Robert et al. [12] addressed polynomial multiplication over binary fields by implementing a recursive Karatsuba variant combined with a Toom-Cook-3 scheme, leveraging the AVX512 VPCLMULQDQ instruction to parallelize multiple 64-bit polynomial multiplications.

In HQC, large-degree polynomial multiplication is the most time-consuming step. The Schoolbook algorithm [13], while applicable to any parameter, has a time complexity of $O(n^2)$, which is inefficient for large-degree polynomials. In contrast, Number Theoretic Transform(NTT) [14] offers faster computation over a ring through modular reduction using algorithms like Barrett, with a time complexity of $O(n \log n)$. Wan et al. [15] achieved a 6.47$\times$ performance improvement for the post-quantum cryptographic algorithm Kyber through the optimization of NTT. However, it is only suitable for specific parameters and requires significant computational resources. For HQC in $\mathbb{F}_2$, transform-based methods such as Frobenius Additive FFT also achieve $O(n \log n)$ complexity [16]. Robert et al. [12] implemented various Toom–Cook variants. For instance, the asymptotic time complexity of Toom-Cook-3-5 multiplication is $O(t^{\log_3 5})$, that of Toom-Cook-4-7 is $O(t^{\log_4 7})$, and that of Toom-Cook-5-9 is $O(t^{\log_5 9})$.

Sparse-dense multiplication [17], such as fast convolution, can effectively handle zero elements in polynomials, enhancing the performance of HQC. For a sparse polynomial with d non-zero coefficients, its complexity can be as low as $O(dn)$. Nevertheless, as reported in [18], constant-time implementation is an ideal security feature for NIST PQC candidates to ensure resistance to side-channel attacks [19]. Any multiplication algorithm that exploits the sparsity of secret data may leak some information. The core vulnerability stems from the dependence of execution time or power consumption on the Hamming weight of sparse secret operands, which adversaries can exploit via side-channel analysis to recover structural information and potentially the full key. Consequently, achieving constant-time execution—wherein runtime is independent of secret data—necessitates treating sparse operands as dense, thereby negating the performance gains of sparse-aware algorithms in secure implementations.

The Karatsuba algorithm [20], as a dense-dense multiplication, reduces multiplication operations by cleverly using addition and subtraction. This approach achieves a complexity of $O(n^{\log_2 3})$. The sliding window multiplication technique proposed in 2008 [21] effectively addresses the efficiency issues of the Karatsuba algorithm in the underlying 64-bit multiplication operations. Therefore, combining the advantages of Karatsuba and sliding window multiplication, they were selected as the official additional implementation for polynomial multiplication in HQC [5]. From an algorithmic structure perspective, Karatsuba represents the most fundamental instance of the Toom–Cook method. The UKaratsuba approach and its variants forms adopted in this work align with the core idea of Toom–Cook, yet retain a more regular and controllable recursive structure at the implementation level. Compared to a general Toom–Cook scheme, our method avoids complex coefficient padding, the construction of generic interpolation matrices, and additional boundary processing, thereby making the algorithm better suited to the actual polynomial lengths in HQC. In terms of parameter compatibility and implementation efficiency, FFT-based methods typically incur significant constant overhead, impose additional requirements on the modulus structure and numerical precision, and necessitate extra transformations to adapt HQC to the FFT algorithm. Furthermore, the upadded strategy proposed in this paper enables the Karatsuba algorithm to flexibly accommodate polynomials of non-ideal lengths in HQC, eliminating the zero-padding operations common in traditional Toom–Cook implementations and thus reducing redundant computations.

In 2006, NVIDIA introduced the Single Instruction Multiple Threads (SIMT) architecture and Compute Unified Device Architecture (CUDA) [22] in its GPU. This architecture features thread-level parallelism, allowing multiple independent threads to execute a single instruction simultaneously, which can significantly enhance performance. The inherent parallelism of GPU architectures is particularly suited to optimizing throughput, making them ideal for handling the intensive demands of cryptographic workloads under peak conditions.

The inherent parallelism of GPUs, characterized by their many-core architecture, makes them particularly well-suited for cryptographic acceleration, especially in applications requiring high throughput. Numerous studies have leveraged this capability to achieve substantial improvements in cryptographic performance [8], [23]–[26]. For lattice-based cryptography, schemes like Kyber and Dilithium, which are amenable to the Number Theoretic Transform (NTT), have shown excellent acceleration on GPUs. For instance, Lee et al. [27] demonstrated significant throughput gains by employing tensor cores in NVIDIA GPUs to accelerate polynomial convolution in lattice-based cryptosystems. Similarly, Ji et al. [28] introduced HI-Kyber, a high-performance implementation of the CRYSTALS-Kyber algorithm on NVIDIA GPUs, achieving a key exchange performance of 1,664 ops/s. Regarding hash-based signatures, the highly parallel nature of hash computations aligns well with the GPU architecture. Bernstein et al. [29] demonstrated that a GPU-based implementation of SPHINCS+ could outperform its CPU counterpart by an order of magnitude in batch verification scenarios. Recent advances have also explored GPU acceleration for code-based cryptography. While lattice-based cryptographic schemes have emerged as a dominant research strand in post-quantum cryptography due to their versatile efficiency and compact key sizes, spurring extensive optimization efforts, code-based schemes—as one of the oldest security assumptions—have also seen significant advances in recent years. For BIKE, Guerreau et al. [30] leveraged CUDA warp-level primitives to parallelize sparse matrix-vector multiplication, achieving

an 8.7× speedup and a throughput of 12,000 operations per second on an NVIDIA A100 GPU while maintaining constant-time execution. Further optimization was presented by Fu et al. [31] with ECO BIKE, introducing a three-thread parallel architecture and optimized polynomial multiplication. For Classic McEliece, Wu et al. [32] proposed the first high-performance GPU implementation for Classic McEliece, employing a heterogeneous CPU-GPU execution model, kernel fusion, and optimized AFFT-based operations, which led to significant acceleration for both encapsulation and decapsulation.

Despite these advancements, research on GPU acceleration for HQC remains limited, especially since its recent standardization. In this paper, we try to address this gap by proposing a novel GPU-based solution for accelerating HQC, aiming to overcome its performance limitations and enhance the efficiency of high-throughput cryptographic operations.

B. Our Contribution

To date, most post-quantum cryptographic hardware research has focused on lattice-based candidates, with code-based algorithms receiving much less attention. Existing hardware implementations for HQC primarily rely on high-level synthesis (HLS) or hardware-software co-design approaches. The motivation for our work is to understand how well hand-optimized HQC implementation can be designed and realized on GPUs, addressing the current research gap in GPU-based implementations of code-based cryptographic algorithms. In this paper, Our research presents the first fully **H**igh-performance **I**mplementation **G**PU-based **H**amming Quasi-Cyclic (short for HIGH). The main innovation points are as follows:

- First, We innovatively proposed a High-performance Implementation scheme of GPU-based HQC (HIGH). This scheme comprehensively covers the key aspects of HQC, encompassing KeyGeneration, Encapsulation, and Decapsulation. Our design incorporates an efficient core fusion strategy for HQC, integrating fundamental operations such as UKaratsuba, reduction and SHAKE256. This fusion strategy dramatically reduces global memory access frequency during computation. To the best of our knowledge, this work presents the first complete GPU-based implementation of HQC, covering all cryptographic stages from KeyGeneration to Decapsulation.
- Then, Experimental results have demonstrated that polynomial multiplication is central to the performance of HIGH. We proposed an innovative polynomial multiplication architecture HIKD, we designate the Karatsuba multiplication for higher-order components as HIK, and the Dual-track Parallel Windowing multiplication for lower-order components as HID. In HIK, we improved and implemented UKaratsuba, its variants, and their combinations, based on Karatsuba. This approach eliminates the need for padding and reduces the number of multiplication operations. In HID, we designed a sliding window parallel strategy that supports both 64-bit and 128-bit polynomial operations. We identified the optimal multiplication scheme for HQC on GPU architectures. Experimental results have shown that HIKD increases throughput by 85.6% compared to native multiplication.
- Finally, All performance evaluations were conducted on a system equipped with an NVIDIA GeForce RTX 4090 GPU (24 GB VRAM) and CUDA 12.1. Our HIKD was implemented in HIGH and rigorously tested in various thread configurations within the GPU block. Our goal is to systematically explore Karatsuba variants, windowing strategies, and GPU execution configurations, and to identify combinations that achieve near-peak performance for HQC on modern GPUs. Through extensive comparative experiments on GPU, our HIGH can achieve key generation of 917488 ops/s, encapsulation of 586560 ops/s, and unblocking of 374160 ops/s.

The rest of the paper is organized as follows. Section II introduces some basic preparatory knowledge. Section III presents our specific optimization implementation scheme on GPU. Section IV shows the experimental results of our scheme. Section V summarizes our work.

II. Preliminary knowledge

This section introduces the preparatory knowledge related to our work. Table I displays the symbols used along with their detailed explanations.

TABLE I: Symbol Explanation

Symbol	Explanation
$\mathbb{F}_2$	Binary finite field
$\mathbb{F}_2^k$	The k-dimensional vector space over the binary field
R	Cyclic polynomial ring $\mathbb{F}_2[X]/(X^r-1)$
$\mathcal{G}$, $\mathcal{K}$, $\mathcal{H}$	Hash functions
$parameters$	The vector length and Hamming weight information are specified
m	The plaintext to be encrypted, a message $m \in Z_n$
c	The encrypted ciphertext c
$w(\cdot)$	The Hamming weight of a vector *i.e.* the number of its non-zero coordinates

A. GPU Architecture

Unlike CPUs, which are optimized for sequential task processing, modern Graphics Processing Units (GPUs) are massively parallel architectures containing thousands of smaller, energy-efficient cores designed for high-throughput computation [33]. The CUDA programming model organizes these cores hierarchically: multiple threads are grouped into a thread block, and multiple blocks form a grid [34]. In terms of memory hierarchy, each thread has its own private local memory and registers, while threads within the same block can communicate efficiently through fast, on-chip shared memory [35]. At the hardware level, a GPU consists of multiple Streaming Multiprocessors (SMs). Each SM schedules and executes multiple thread blocks concurrently, with the fundamental execution unit being a warp—typically a group of 32 threads that execute the same instruction in a Single Instruction, Multiple Threads (SIMT) fashion [36]. This SIMT architecture achieves peak computational efficiency when all threads in a warp follow the same control flow, operating on different data elements. Through frameworks like CUDA,

the host (CPU) can offload parallel workloads to the device (GPU), harnessing the power of heterogeneous computing to dramatically accelerate compute-intensive tasks.

B. Code-Based and Dependency Problem

Code-based cryptography, pioneered by McEliece in 1978 [37], constitutes one of the major and longest-studied families of post-quantum cryptography (PQC). Its security relies on the hardness of problems from coding theory, which are believed to resist attacks by both classical and quantum computers [2]. A key advantage is that the core problem—Syndrome Decoding—has been extensively analyzed for decades, providing a high confidence level in its asymptotic security.

Syndrome Decoding(SD) problem. The security of most code-based PQC schemes, including HQC, primarily relies on the SD problem, which is known to be NP-complete [38] and is considered intractable for random codes. Specifically, given a parity-check matrix $H \in \mathbb{F}_2^{(n-k)\times n}$ for a binary linear code C, a syndrome vector $y \in \mathbb{F}_2^{n-k}$, and an integer z, the problem is to find an error vector $x \in \mathbb{F}_2^n$ with Hamming weight $\mathrm{wt}(x) \leq z$ such that $Hx = y$. Practical schemes like HQC use *structured* codes (e.g., quasi-cyclic) to achieve efficiency, under the assumption that the SD problem remains hard even with this structure [39].

HQC is a code-based encryption scheme whose security relies on the hardness of the Syndrome Decoding problem for structured codes. As an efficient Key Encapsulation Mechanism (KEM), HQC naturally integrates into the KEM-DEM paradigm, enabling the construction of a hybrid encryption system that achieves IND-CCA2 security.

C. Key Encapsulation Mechanism of HQC

HQC constructs the KEM based on the aforementioned hard problems, thereby enabling secure key exchange. The HQC family consists of three different algorithms: HQC128, HQC192, and HQC256, which differ in their security levels of 1, 3, and 5. To introduce HQC.KEM [40], we first introduce HQC.PKE.

The HQC.PKE consists of three core parts: KeyGeneration, Encryption and Decryption, as described in Algorithm 1. In KeyGeneration, a vector h is sampled uniformly random. The secret key is composed of two vectors x, y that are sampled with a specified weight w. The public key is composed of h and the syndrome s. In Encryption, three vectors r_1, r_2, e are sampled with a specified weight w_r and w_e. Meanwhile, the message m is encoded first by a Reed-Solomon (RS) encoder and then by a Reed-Muller (RM) encoder to get t. Finally, the ciphertext pair (u, v) can be calculated by $r_1 + h \cdot r_2$ and $t + s \cdot r_2 + e$. In Decryption, the message decoded from $v - u \cdot y$ to get message m'.

HQC.KEM is also composed of three primitives: KeyGeneration, Encapsulation and Decapsulation. KeyGeneration in HQC.KEM is the same as KeyGeneration in HQC.PKE. The message m is then sampled and encrypted into ciphertext c. Simultaneously, m and c are input into a hash function to generate the shared secret K. The message m alone is input into a different hash function to generate d. During communication, K is retained, and c and d are transmitted to the other party. Concerning Decapsulation, the c in ciphertext is used in the decryption to retrieve the message m' and it is necessary to check whether the received ciphertext is correct.

Algorithm 1: HQC.PKE

Setup(1^λ)**:** message m, parameter = (n, k, w, w_r, w_e)
1: **HQC.PKE.KeyGeneration():**
2: $h \leftarrow R$
3: $(\mathbf{x}, \mathbf{y}) \xleftarrow{w} R^2$
4: $s = x + h \cdot y$
5: **return** $pk = (h, s),\ sk = (x, y)$
6: **HQC.PKE.Encryption**(pk, m, θ)**:**
7: $\mathbf{r_2} \xleftarrow{w_r} R$
8: $\mathbf{e} \xleftarrow{w_e} R$
9: $\mathbf{r_1} \xleftarrow{w_r} R$
10: $u = r_1 + h \cdot r_2$
11: $t = Encode(m)$
12: $v = t + s \cdot r_2 + e$
13: **return** $c = (u, v)$
14: **HQC.PKE.Decryption**(sk, c)**:**
15: $m' = Decode(v - u \cdot y)$
16: **return** m'

Algorithm 2: HQC.KEM.Encapsulation(pk)

Input:
Public key pk.
Output:
Ciphertext $(c, d), K$.
1: $m \leftarrow \mathbb{F}_2^k$
2: $salt \leftarrow \mathbb{F}_2^{128}$
3: parse pk as (seed2, s)
4: $\theta = \mathcal{G}(m, seed2, salt)$
5: $c = (u, v) =$ **HQC.PKE.Encryption**(pk, m, θ)
6: $K = \mathcal{K}(m, c)$
7: $d = \mathcal{H}(m)$
8: **return** $(c, d, salt)$

Algorithm 3: HQC.KEM.Decapsulation$(sk, c, d, salt)$

Input:
Secret key sk, c, d.
Output:
$(K', 0)$ or $(K', 1)$.
1: $m' =$ **HQC.PKE.Decryption**(sk, c)
2: parse pk as (seed2, s)
3: $\theta' = \mathcal{G}(m', seed2, salt)$
4: $c' = (u', v') =$ **HQC.PKE.Encryption**(pk, m', θ')
5: $d' = \mathcal{H}(m')$
6: $K' = \mathcal{K}(m', c')$
7: if $c \neq c'$ or $d \neq d'$ then
8: result $(K', 0)$
9: else
10: result $(K', 1)$
11: end if

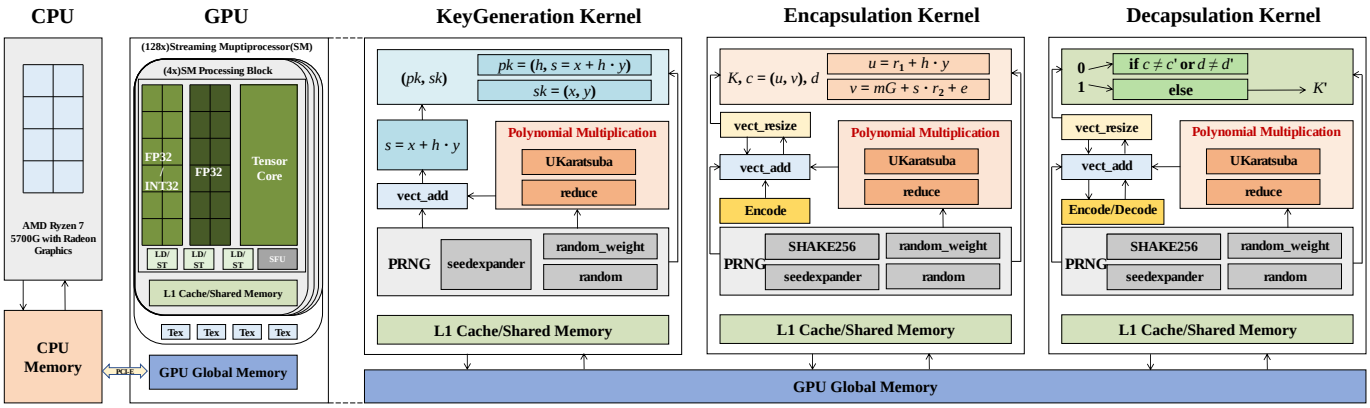


Fig. 2: The Kernel-fusion Architecture of HQC

Therefore, the Encapsulation needs to be done again using the result m' to get the new ciphertext c' and hash value d'. Finally, it is easy to know if there is an error by verifying whether c and c', and d and d' are identical. Algorithm 2 and 3 show the process of HQC.KEM.

D. Karatsuba Multiplication

HQC operates in the finite field $\mathbb{F}_2^m$ using polynomial multiplication as a key operation [5]. The finite field $\mathbb{F}_2^m$ has only the elements 0 and 1. In this binary field, the addition and the subtraction are defined as XOR operation.

In $\mathbb{F}_2^m$, elements are represented as polynomials:

$$\begin{aligned} a(x) &= a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + \ldots + a_1 x + a_0, \\ b(x) &= b_{m-1}x^{m-1} + b_{m-2}x^{m-2} + \ldots + b_1 x + b_0. \end{aligned} \tag{1}$$

Here, $a(x)$ and $b(x)$ are elements of $\mathbb{F}_2^m$, and their product $c(x) = a(x)b(x)\ mod\ w(x)$, $w(x)$ an irreducible polynomial chosen for constructing the field.

In Karatsuba multiplication, A and B are two binary polynomials with a maximum degree of $N-1$. These polynomials are packed into a table of 64-bit words, where the size of the table is $\lceil N/64 \rceil$. The result R of the algorithm is a polynomial with a maximum degree of $2 \cdot N - 2$. Let $t = 2^r$ with r the minimum value ensuring $t \geqslant \lceil N/64 \rceil$. Now A and B are considered as polynomials of degree at most $64 \cdot t - 1$, and A_L/A_H and B_L/B_H denote the lower/higher halves of A and B, respectively. As shown in Equation 2, Karatsuba efficiently diminishes the computational burden of performing an t-words multiplication by breaking it down into three smaller multiplications of $t/2$-words, along with four t-words addition steps. This approach involves three intermediary multiplication outcomes, symbolized by α, β, and γ.

$$\begin{aligned} \alpha &= A_L \times B_L, \\ \beta &= A_H \times B_H, \\ \gamma &= (A_L + A_H) \times (B_L + B_H), \\ R &= A \times B = \alpha + (\gamma - \alpha - \beta) \times 2^{\frac{t}{2}} + \beta \times 2^t. \end{aligned} \tag{2}$$

HQC employs the unpadded variant of the Karatsuba algorithm (UKaratsuba). Unlike the standard Karatsuba, which requires operand sizes to be a power of two and introduces padding overhead, UKaratsuba operates on arbitrary sizes, eliminating redundant multiplications and enhancing computational efficiency.

III. Optimized Implementation of HQC on GPU

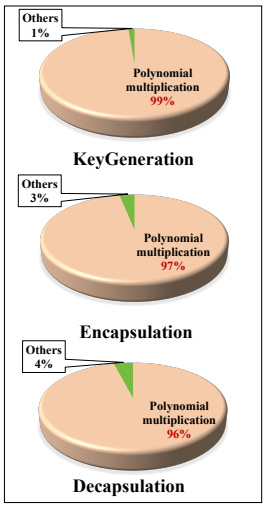
In this section, we present the implementation details and optimization techniques of HQC.

A. The Kernel-fusion Scheme of HQC Implementation

When implementing HQC on GPU, we are committed to reducing the memory access from global memory as off-chip memory has higher latency and lower access speed. Before considering the implementation of HQC on GPU, it is essential to address a potential warning that may arise when porting CPU code to GPU code: Stack size for entry function δ cannot be statically determined. In HQC, the function δ operates as the key generation, encryption, and decryption function. This issue occurs due to the presence of a recursive structure in the code, with function δ being the recursive function in question. In fact, according to the official explanation of the CUDA, when recursive structures are present in GPU code, the compiler typically issues a warning to alert developers about potential stack usage. If a stack overflow actually occurs at runtime, developers must manually adjust the stack size. Since the critical multiplication part of our solution employs the Karatsuba recursive multiplication algorithm, we manually set the stack size using the CUDA-provided API to ensure the recursion executes smoothly. Unlike CPUs, GPU handle stack and memory management differently. GPU are initially designed to schedule a large number of threads for parallel computation, prioritizing efficiency and hardware cost.

As a result, GPU stack space is smaller and lacks the flexibility for dynamic adjustment, which significantly limits GPU support for recursive functions. To resolve this issue, three approaches can be considered: The first approach is to convert the recursive function into an equivalent iterative implementation. The second approach is to unroll the recursive function. The final approach involves using the CUDA programming function “cudaDeviceSetLimit” to increase the

stack size on the GPU. Our implementation utilizes the last methods to address the stack size issue. We chose not to adopt an iterative version of the algorithm primarily because the recursive structure of Karatsuba not only directly reflects its mathematical principle but also yields a more concise and transparent implementation.



Type	Operation Name
Other Operations	SHAKE256
	Random_weight
	Seedexpander
	Random
	Compare resize
	Encode\Decode
	… …
Multiplication	Polynomial Multiplication

Polynomial Multiplication operations constitute over **96%** of the computational workload in the Key Generation, Key Encapsulation, and Key Decapsulation processes.

Fig. 3: The Computational Workload

Accordingly, as shown in Figure 2, we show the overall algorithm composition kernel-fusion framework of KeyGeneration, Encapsulation, and Decapsulation. By employing the kernel fusion technique, we enhance the performance of parallel computations on the GPU by merging multiple kernel functions into a single kernel. This approach minimizes the time spent transferring data between the CPU and GPU, and it also reduces the overhead associated with launching and managing multiple kernels.

In the evaluation of the four official HQC implementations, the additional implementation was chosen for in-depth analysis considering the hardware characteristics of the GPU. On the GPU platform, our pioneering kernel fusion approach achieves a throughput of 445,400 ops/s for KeyGeneration, 350,024 ops/s for Encapsulation, and 227,066 ops/s for Decapsulation. Compared with CPU, the performance is improved by 585×, 924×, and 912× respectively. All our subsequent work is based on kernel-fusion as the baseline for research.

B. Polynomial Multiplication Architecture on GPU

Before optimizing the algorithm layer, it is essential to analyze the bottlenecks in the implementation of HQC. As shown in Figure 3, in the additional implementation of HQC, polynomial multiplication is the most time-consuming part, accounting for more than 96% of the overall performance of HQC. Therefore, our approach focuses on optimizing polynomial multiplication, specifically by reducing the number of multiplications to decrease the overall multiplication time. Next, we will focus on introducing the optimization improvements for polynomial multiplication in HQC.

When designing an efficient and flexible polynomial multiplication architecture, considering the significant impact of

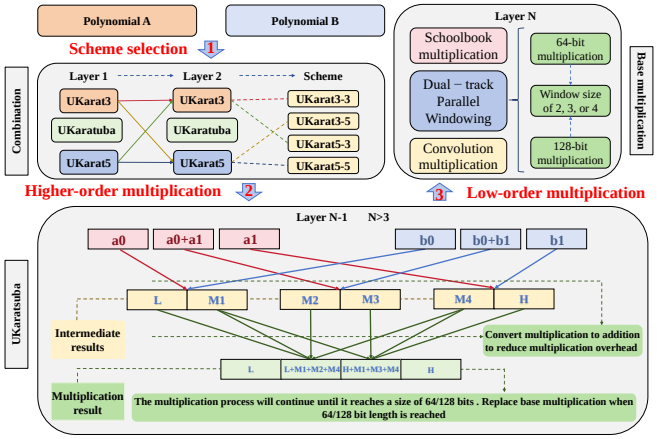


Fig. 4: HIKD Architecture

polynomial orders on computational complexity and resource requirements, we divide the architecture into two parts: the high-order multiplication scheme and the low-order multiplication scheme, as shown in Figure 4. In the context of our HIGH scheme, high-order multiplication refers to polynomial multiplication exceeding 128 bits, while both 128-bit and 64-bit multiplications are classified as low-order.

In the high-order multiplication scheme, We optimized Karatsuba and implemented the UKarat3 and UKarat5 algorithms, specifically avoiding padding steps in Karatsuba and focusing on reducing multiplication operations. Building upon this foundation, we have implemented several novel multiplication combinations (UKarat3-3, UKarat3-5, UKarat5-3, and UKarat5-5). These combinations correspond to distinct multiplication-splitting strategies. For instance, the notation UKarat3-3 indicates that a tripartite splitting strategy is applied in the first two partitions of the high-order polynomial. We compare these methods with UKaratsuba to highlight their advantages, as detailed in Section III-C.

In our low-order multiplication scheme, we implemented three methods: schoolbook, convolution, and a dual-track parallel windowing approach, aiming to identify the optimal configuration for varying GPU thread counts. We also analyzed sliding window multiplication and determined suitable parameters for the dual-track method, as detailed in Section III-D.

Although the performance data is based on HQC-128 (both original and optimized results), our approach can be extended to HQC-192 and HQC-256, where it is expected to deliver significant performance gains over their respective baselines.

C. Optimized Karatsuba Multiplication

We will provide a detailed explanation of the aforementioned HIKD multiplication scheme. Before introducing HID, we will first discuss high-order multiplication scheme HIK and describe all multiplication operations from two perspectives.

1) UKaratsuba Variants Optimization: We have implemented UKaratsuba on GPU for the first time, establishing it as a benchmark for implementing other algorithms. We refer to the Karatsuba algorithm with the padding step removed

as the UKaratsuba algorithm. When multiplying polynomials of 17,669 bits using Karatsuba algorithm, the operands must first be padded to the nearest power-of-two length (i.e., 32,768 bits). Based on its complexity, this padding introduces approximately 46% of extra zero coefficients and increases the multiplication count by about 41%. In contrast, UKaratsuba recursively partitions the operands directly according to their actual bit-length, thereby avoiding the redundant computation and storage overhead caused by padding.

Building upon UKaratsuba, we have implemented two classical variants of the Karatsuba algorithm: Karat3 and Karat5, as shown in Figure 7. Karat3 divides polynomials into three parts, while Karat5 divides them into five parts and employs different merging strategies. Although Karat3 and Karat5 offer lower theoretical time complexity, their practical efficiency depends on multiple factors, such as the increased number of multiplication and addition operations, as well as the characteristics of polynomials with varying lengths. Furthermore, the padding operations required in Karatsuba-3 and Karatsuba-5 introduce additional multiplicative overhead, an effect that becomes especially pronounced in large-scale polynomial multiplication. To address this, we innovatively proposed unpadded versions of Karat3 and Karat5, and have implemented UKarat3 and UKarat5 on GPU for the first time. We provided the implementations of UKarat3 and UKarat5 as shown in Algorithm 4 and Algorithm 5.

The advantage of these two types of multiplication is that, on the one hand, UKarat3 and UKarat5 eliminate unnecessary computational overhead by avoiding polynomial padding. On the other hand, the idea of unpadded multiplication eliminates the need to search for algorithm adapted lengths based on the size of the operands. Instead, the operands can be directly applied to multiplication, and internal processing has been made for situations where the length of the polynomial cannot be evenly divided, such as evenly distributing the undifferentiated part to the lower bits, and separating the number of items in the final combination due to different lengths. Specifically, for HQC-128, when processing the multiplication of two polynomials with a length of 17669 bits, Karat3 and Karat5 would require padding operands to 17,856 and 17,920 bits, respectively. The reduction in the number of multiplications is quantified by comparing the number of leaf multiplications in the Karatsuba recursion tree: a padded implementation performs 3^r multiplications for length $t = 2^r$, whereas the padding-free version only generates leaf multiplications corresponding to non-zero subproblems. As shown in Table II, compared to their padded versions, UKarat3 reduces the number of multiplications by approximately 0.48%, while UKarat5 reduces it by 0.24%. This enhanced adaptability improves compatibility with GPU parallel processing capabilities and significantly accelerates the multiplication process.

The challenge of unpadded versions lies in the increased complexity of length alignment issues as polynomial partitioning becomes more refined, particularly noticeable in UKarat5 compared to UKarat3 during final merging. This occurs because UKarat5 employs a finer polynomial partitioning, and the upadded strategy results in segments of unequal lengths, leading to additional offset handling during the final recombination stage. But overall, the improvements from unpadded methods enhance algorithm efficiency.

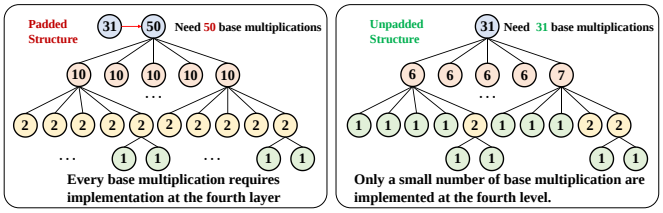


Fig. 5: Upadd.vs.Padd

TABLE II: Number of 64 bit Multiplication

Padded Scheme	Count	Upadded Scheme	Count	Improvement
Karatsuba	—	UKaratsuba	9857	—
Karat3	10086	UKarat3	10038	0.48%
Karat5	10125	UKarat5	10101	0.24%
Karat3-3	8676	UKarat3-3	8664	0.14%
Karat5-3	13050	UKarat5-3	12618	3.3%
Karat3-5	13050	UKarat3-5	12618	3.3%
Karat5-5	14175	UKarat5-5	13455	5.1%

2) UKaratsuba Combinations Optimization: As mentioned earlier, UKaratsuba, UKarat3, and UKarat5 employ a recursive divide-and-conquer approach to partition the original problem into smaller sub-problems, utilizing UKaratsuba recursively to compute and combine results. These three multiplication methods can be abstracted into a tree structure, where the N-dimensional vector of UKaratsuba can be viewed as a binary tree. Assuming polynomial recursion splits evenly each time, *i.e.*, $N = 2^r$, the minimum depth of the binary tree is $\log_2^N + 1$. In this structure, except for the last layer, each layer doubles the number of nodes from the previous layer, with each node's weight being half of its parent node's weight, where weight represents the length of operands at that layer. For smaller or moderately sized operands, UKaratsuba, UKarat3, and UKarat5 effectively solve the problem. However, for larger operands requiring deeper recursion, we introduce four related multiplication combinations starting from the second layer of multiplication.

Moving beyond the traditional binary splitting of Karatsuba's algorithm, we consider re-evaluating polynomial splitting schemes. Employing a higher splitting factor leads to a recursion tree with increased branching factor and reduced depth. This implies that large problems reach the low-order base case faster, offering higher potential parallelism, albeit at the cost of a greater number of sub-multiplications per level, which may introduce additional overhead. We measure four specific multiplication combinations: UKarat3-3, UKarat3-5, UKarat5-3, and UKarat5-5. Here, the notation UKaratN-M denotes an N-way split in the first stage followed by an M-way split in the second stage of the polynomial. Assuming a total of N layers in recursion, the specific combinations are illustrated in Figure 4, with UKarat3 and UKarat5 used in the first and second layers, UKaratsuba used from the third layer to the $N-1$th layer, and the subsequent HID multiplication scheme used in the Nth layer.

Table II demonstrates the efficiency of unpadded multiplication over padded multiplication for a polynomial of length

Algorithm 4: HIK-UKarat3(A, B, t)

input : A and B on t computer words.
output: $R = A \times B$
$s_0 = s_1 = s_2 \leftarrow t/3$
Step (1): Segmentation and compensation
if $t \mod 3 = 1$ **then**
  $s_0 \leftarrow t/3 + 1$
  else if $t \mod 3 = 2$ **then**
    $s_0 = s_1 \leftarrow t/3 + 1$
$A = A_0 + x^{64s_0}A_1 + x^{64(s_0+s_1)}A_2$
$B = B_0 + x^{64s_0}B_1 + x^{64(s_0+s_1)}B_2$
Step (2): Perform recursive multiplication
for $i \leftarrow 0$ **to** 2
  $R_i \leftarrow \text{UKaratsuba}(A_i, B_i, s_i)$
/* the addition for UnpaddedKarat3 */
for $i \leftarrow 0$ **to** 1
  for $j \leftarrow i + 1$ **to** 2
    for $z \leftarrow 0$ **to** $s_2 - 1$
      $A_{ij}[z] \leftarrow A_i[z] + A_j[z]$
      $B_{ij}[z] \leftarrow B_i[z] + B_j[z]$
if $t \mod 3 = 1$ **then**
  $A_{01}[s_1] = A_{02}[s_1] \leftarrow A_0[s_1]$
  $B_{01}[s_1] = B_{02}[s_1] \leftarrow B_0[s_1]$
  else if $t \mod 3 = 2$ **then**
    $A_{01}[s_2] \leftarrow A_0[s_2] + A_1[s_2]$
    $B_{01}[s_2] \leftarrow B_0[s_2] + B_1[s_2]$
    $A_{02}[s_2] \leftarrow A_0[s_2]$
    $B_{02}[s_2] \leftarrow B_0[s_2]$
    $A_{12}[s_2] \leftarrow A_1[s_2]$
    $B_{12}[s_2] \leftarrow B_1[s_2]$
for $i \leftarrow 0$ **to** 1
  for $j \leftarrow i + 1$ **to** 2
    $R_{ij} \leftarrow \text{UKaratsuba}(A_{ij}, B_{ij}, s_i)$
Step (3): Final result of restructuring
$R \leftarrow R_0 + (R_0 + R_1 + R_{01})X^{64s_0} + (R_0 + R_2 + R_{02})X^{64(s_0+s_1)} + R_1X^{64(2s_0)} + (R_1 + R_2 + R_{12})X^{64(2s_0+s_1)} + R_2X^{64(2s_0+2s_1)}$
return (R)

17669, with notable reductions in multiplication operations. Using a length 31 polynomial as an example for UKarat5-5, we analyze the advantages of unpadded methods as shown in Figure 5. For Karat5-5, the polynomial length must be divisible by 25 to accommodate the division by 5 in the first and second layers, whereas UKarat5-5 requires no padding, distributing the remainder evenly to the lower bits and maintaining this principle in subsequent multiplications. Compared to Karat5-5, UKarat5-5 significantly reduces the number of multiplications.

In order to demonstrate the UKaratsuba combinations in more detail, we take UKarat3 as an example, as shown in Figure 7. For the 17,669-bit polynomial multiplication, the scheme employs the UKarat3 algorithm for the first two recursive layers, followed by UKaratsuba down to 64-bit or 128-bit operands. As evidenced by the statistics in Table II, the UKarat3-3 scheme is better suited for the 17,669-bit scenario, requiring only 8,664 base multiplications—significantly fewer than other Karatsuba combinations.

3) Comparison: These combinations break away from the recursive approach of UKaratsuba by dividing the polynomials into three or five parts in the first multiplication layer, corresponding to UKarat3 or UKarat5, respectively. In the second multiplication layer, UKarat3 or UKarat5 is continued instead of UKaratsuba, and then UKaratsuba recursion resumes from the third layer onwards. The design aims to explore the impact of different combinations on the number of polynomial multiplications, seeking the optimal solution with the minimum multiplication count.

As previously discussed, UKarat3-3 proves most suitable for this polynomial length, reducing 1193 multiplications compared to native UKaratsuba. Our approach can extend UKaratsuba combinations into higher layers to explore additional combined multiplication strategies. For other cryptographic schemes, our HIK provides an evaluation standard and design approach to explore more suitable polynomial multiplication combinations across any polynomial length.

D. Base Multiplication

As low-order multiplication scheme, HID plays a supporting role as the most HIK of base multiplication. Our HID implements schoolbook multiplication and convolution multiplication, alongside a refined strategy tailored for parallel processing: the dual-track parallel windowing strategy polynomial multiplication.

Schoolbook Multiplication: This multiplication [41] is a classic implementation method. For two polynomials of length n, the multiplication operation requires n^2 multiplicative operations. The 64-bit Schoolbook multiplication is a specific instance of this method, tailored for 64-bit numbers. The Equation 3 for the multiplication in this case would involve the traditional multiplication used for binary numbers.

$$ab = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} a_i b_j x^{i+j} \tag{3}$$

Convolution Multiplication: Convolution is actually a kind of multiplication with 'location information'. Its operation steps: reverse sequence, displacement, multiplication and add. The convolution of two vectors u and v represents the area of the overlapping part determined by these points when $v$slides over u. Algebraically speaking, convolution is the same operation as the multiplication of polynomials with coefficients of u and v elements. Then w is of length $m+n-1$ and the k-th element is $w(k)$, where $m = length(u)$ and $n = length(v)$.

We extract polynomial multiplication from HQC and replace the native sliding window multiplication with schoolbook multiplication and convolution multiplication in the lower-order multiplication. The resulting polynomial multiplication performance is shown in Table III. Given the superior performance of sliding window multiplication, we further optimize the lower-order multiplication based on this.

$$w(k) = \sum_{j} u(j)v(k - j + 1) \tag{4}$$

Algorithm 5: HIK-UKarat5(A, B, t)

input : A and B on t computer words.
output: $R = A \times B$
$s_0 = s_1 = s_2 = s_3 = s_4 \leftarrow t/5$
Step (1): Segmentation and compensation
if $t \mod 5 = 0$ **then**
  $s_0 = s_1 = s_2 = s_3 = s_4 \leftarrow t/5$
  else if $t \mod 3 = 1$ **then**
    $s_0 \leftarrow t/5 + 1$
  else if $t \mod 5 = 2$ **then**
    $s_0 = s_1 \leftarrow t/5 + 1$
  else if $t \mod 5 = 3$ **then**
    $s_0 = s_1 = s_2 \leftarrow t/5 + 1$
  else if $t \mod 5 = 4$ **then**
    $s_0 = s_1 = s_2 = s_3 \leftarrow t/5 + 1$

$A = A_0 + x^{64s_0}A_1 + x^{64(s_0+s_1)}A_2 + x^{64(s_0+s_1+s_2)}A_3 + x^{64(s_0+s_1+s_2+s_3)}A_4$
$B = B_0 + x^{64s_0}B_1 + x^{64(s_0+s_1)}B_2 + x^{64(s_0+s_1+s_2)}B_3 + x^{64(s_0+s_1+s_2+s_3)}B_4$
Step (2): Perform recursive multiplication
for $i \leftarrow 0$ **to** 4
  $R_i \leftarrow \text{UKaratsuba}(A_i, B_i, s_i)$
$UnpaddedKarat5_add()$
  `/* the addition for UnpaddedKarat5, similar to the addition handling in Karat3 */`
for $i \leftarrow 0$ **to** 3
  for $j \leftarrow i + 1$ **to** 4
    $R_{ij} \leftarrow \text{UKaratsuba}(A_{ij}, B_{ij}, s_i)$
Step (3): Final result of restructuring
$R \leftarrow R_0 + R_4X^{64\times2(s_0+s_1+s_2+s_3)} + (R_{01} + R_0 + R_1)X^{64s_0} + R_1X^{64\times2s_0} + (R_{02} + R_0 + R_2)X^{64(s_0+s_1)} + (R_{12} + R_1 + R_2)X^{64(2s_0+s_1)} + (R_{03} + R_0 + R_3)X^{64(s_0+s_1+s_2)} + R_2X^{64\times2(s_0+s_1)} + (R_{13} + R_1 + R_3)X^{64(2s_0+s_1+s_2)} + (R_{04} + R_0 + R_4)X^{64(s_0+s_1+s_2+s_3)} + (R_{23} + R_2 + R_3)X^{64(2s_0+2s_1+s_2)} + (R_{14} + R_1 + R_4)X^{64(2s_0+s_1+s_2+s_3)} + R_3X^{64\times2(s_0+s_1+s_2)} + (R_{24} + R_2 + R_4)X^{64(2s_0+2s_1+s_2+s_3)} + (R_{34} + R_3 + R_4)X^{64(2s_0+2s_1+2s_2+s_3)}$
return (R)

TABLE III: The performance of classical base multiplications

Scheme	Throughput(ops/s)	Latency(ms)
Schoolbook multiplication	57	17.627
Convolution multiplication	553	1.808
Sliding Window multiplication	782	1.278

The Dual – track Parallel Windowing Strategy: In our research, building on the generic sliding-window algorithm introduced in [42], we propose an optimized configuration referred to as the dual-track parallel windowing strategy. In contrast to the original approach, which relies on a uniform window configuration, our method integrates two parallel execution tracks using 128-bit and 64-bit word representations, respectively, and assigns them to different computation stages. In native multiplication scheme, UKaratsuba stops recursion and switches to base multiplication when handling polynomial multiplication of length 64 ($i.e.$, array length 1). To significantly reduce the number of multiplication operations and optimize the storage and utilization of intermediate results, our strategy extends the base polynomial multiplication to 128 bits ($i.e.$, array length of 2). This means that when multiplying polynomials with an array length of 2, we no longer split them into polynomials with an array length of 1, but instead implement it using 128-bit sliding window polynomial multiplication ($w = 128$). This approach, combined with sliding window polynomial multiplication for array lengths of 1, forms the dual-track parallel windowing strategy.

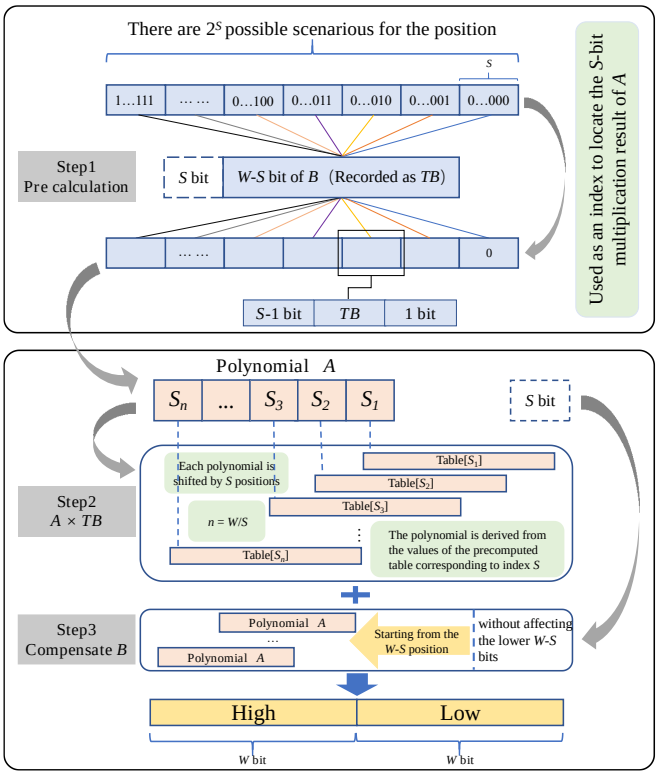


Fig. 6: The Sliding Windowing Strategy

In sliding window multiplication, a and b are multiplicands, each with degrees up to $w-1$ (where $w = 64$ or 128), employs a "window" method analogous to base-2^s exponentiation. This approach optimizes computational efficiency by precomputing multiples of b for all polynomials of degree less than s and leveraging them during computation. By scanning a in chunks of s bits, the method accumulates and shifts corresponding precomputed data from the table into the result, as shown in Figure 6. In this architecture, the constant s, denoted as the window size, is carefully selected and optimized, typically set to $s = 2, 3, 4$.

To systematically define this strategy, we formalize it as the DP(s_1, s_2) model, where s_1 denotes the multiplication operation count for array length 1 (i.e., 64-bit sliding-window multiplication), and s_2 for array length 2 (i.e., 128-bit sliding-window multiplication). We test $s_1 \in 2, 3, 4$ and $s_2 \in$

Algorithm 6: DP(s_1, s_2)

Input:

a and b /* For 64-bit a and b, $w = 64$ and $s = s_1$; for 128-bit, $w = 128$ and $s = s_2$ */

Output:

The result l, h

Step (1): Precomputation

$u[2^s] = \{0, b, 0, \ldots\}$;
for $(i = 2; i < 2^s; i+ = 2)$
  $u[i] = u[i >> 1] << 1$;
  $u[i + 1] = u[i] \wedge b$;

Step (2): Multiplication

$g = u[a \ \& \ (2^s - 1)]$, $1 = g$, $h = 0$;
for $(int \ i = s; i < w; i+ = s)$
  $g = u[a >> i \ \& \ (2^s - 1)]$
  $l \wedge = g << i$;
  $h \wedge = g >> (w - i)$);

Step (3): Handling the High Bits

$m = (2^s - 2) \times (1 + 2^s + 2^{2s} + 2^{3s} + \cdots) \ mod \ 2^w)$;
for $(j = 1; j < s; j + +)$
  $a = (a << 1) \ \& \ m$;
  if (bit $w - j$ of b is set)
    $h \wedge = a$;
return l, h;

$0, 2, 3, 4$, with $s_2 = 0$ indicating that 128-bit multiplication is not used. Algorithm 6 presents the corresponding pseudocode. The selection of s is critical to optimizing thread-level performance within a block, minimizing computational overhead while enhancing parallel scalability.

This innovative dual-track parallel windowing strategy combines 64-bit and 128-bit basic multiplication methods. In 128-bit multiplication, efficiency is enhanced through precomputation and reuse of intermediate results. Unlike UKaratsuba, this strategy directly employs 128-bit basic multiplication, eliminating the need to recursively reduce to 64-bit multiplication before achieving to basic multiplication. This reduction in basic multiplication calls significantly optimizes computational efficiency. For instance, for polynomial multiplication involving 31 operands, results are obtained after recursion to the third layer, bypassing the need for calculations in the fourth layer. This optimization reduces the depth of multiplication recursion, thereby markedly improving overall algorithm performance, as illustrated in Figure 5.

Our optimized implementation is an improvement over existing algorithms and holds significant practical value in the processing of large-scale data. This strategy provides a new perspective and solution for efficient polynomial multiplication, while also offering new insights for other PQC schemes that utilize polynomial multiplication.

E. HIKD Multiplication Architecture

Here we summarize all the optimization strategies for multiplication, comprehensively covering the processing flow of two polynomial multiplications from high-order to low-order, and introducing specific polynomial multiplications.

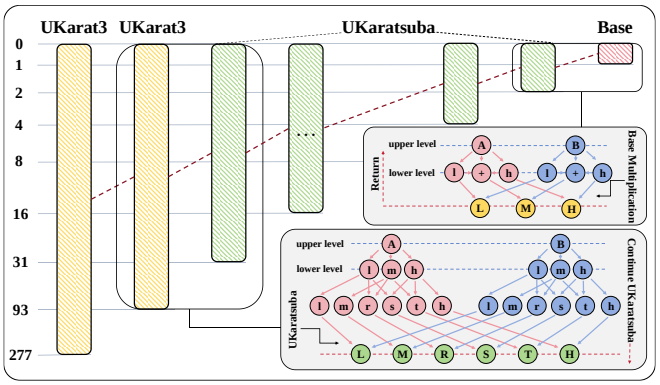


Fig. 7: Overview of UKarat3-3 Multiplication

For high-order multiplication, due to the insecurity of fast convolution multiplication [12], we adopted UKaratsuba and its variants and combinations. This algorithm offers higher security and is easier to implement. More importantly, it is suitable for most polynomial multiplications. In terms of algorithm efficiency, UKaratsuba-related algorithms exhibit superior performance, significantly boosting schemes like HQC, where polynomial multiplication accounts for the majority of the time consumption.

For low-order multiplication, we implemented three basic multiplication schemes and proposed a dual-track parallel windowing strategy. This strategy effectively reduces the recursion depth of UKaratsuba, reaching the recursive exit earlier under certain conditions, thus enabling faster 128-bit multiplication in low-order scenarios. This idea is easy to extend, allowing for the setting of more recursive exits based on specific situations to further shorten the recursion depth and seek an efficiency balance between low-order multiplication and UKaratsuba.

IV. Performance Evaluation

In this section, we evaluate the performance of HIGH through multiple rounds of testing, select the optimal configuration parameters, and ultimately compare our results with those of other works. Furthermore, we designate the official fourth round optimized implementation as the native implementation and conduct a comparative analysis against it.

A. Experimental Environment and Evaluation Criteria

TABLE IV: Experimental Platform Configuration

Type	Model
OS & Tool Chain	Ubuntu 20.04 & CUDA 12.1
CPU	8-core, AMD Ryzen 7 5700G, CPU 3.8GHz
GPU	NVIDIA GeForce RTX 4090 SM×128 (16384 CUDA cores)
Memory	24GB, 2 copy engine(s)

As shown in Table IV, the detailed test environment hardware configuration and corresponding platforms are as follows: the GPU used is the NVIDIA GeForce RTX 4090, featuring 16,384 cores with a frequency of 2.52 GHz, 128 SMs, and 24 GB of memory. The GPU compute capability is 8.9. The

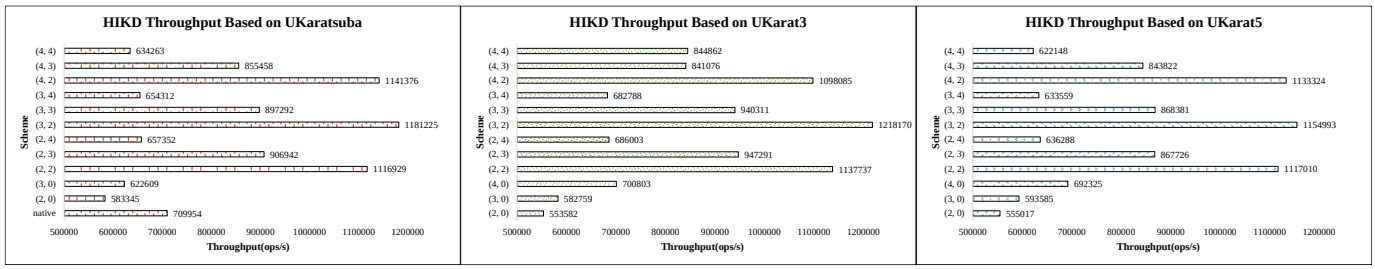


Fig. 8: Performance of HIKD

operating system is Linux Ubuntu 20.04.1, and the CUDA version used is 12.1. The programs were compiled using CMake 3.23.1. And We evaluates our HIGH performance from two perspectives:

- *Throughput*: The number of requests completed by the host and the device per second, measured in thousand operations per second (ops/s).
- *Latency*: The time taken by the host and the device to complete a computation from the moment a request is received, measured in milliseconds (ms).

During experiments on the GPU platform, related parameters significantly impact algorithm performance. These include Block Number, which denotes the quantity of blocks within a CUDA kernel, Threads/Block representing the number of threads per CUDA block, and Threads signifying the total threads launched by the GPU kernel—computed by multiplying the number of thread blocks with the threads per block.

B. Performance of HIGH

The core performance metric of HIGH is throughput, and all our work presented above is conducted based on the security level of HQC-128. The performance of native polynomial multiplications on GPU reaches 709,954 ops/s. We first extract the polynomial multiplication module from the entire cryptographic algorithm and perform a performance comparison of the polynomial multiplication. Finally, the optimal performance of the entire HIGH scheme is presented.

In evaluating the performance of the HIKD structure within the additional HQC-128 implementation, Figure 8 illustrates the effectiveness of selecting the most suitable scheme for HIGH, where (s_1, s_2) denotes DP(s_1, s_2), the abscissa represents throughput. Delving into the performance of UKaratsuba, UKarat3, UKarat5 schemes in HIGH, we have uncovered a remarkable optimization strategy—DP(3,2) configuration mode, which has demonstrated outstanding efficiency. At this point, we determine the optimial parallel paramenters, 384 threads within a block for HIGH. Specific analysis is as follows:

UKaratsuba Scheme: In this scheme, two polynomials of degree 17669 are split into two parts and recursively processed. Using DP(3, 2) for base multiplication, it performs 824 64-bit and 3011 128-bit multiplications, thereby replacing the 9857 64-bit multiplications required by the standard UKaratsuba. Performance depends on polynomial length and partitioning strategy. By eliminating padding, UKaratsuba shortens effective polynomial length and reduces multiplication count

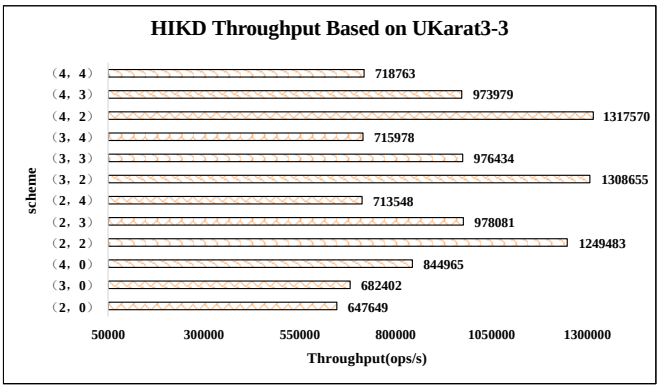


Fig. 9: The Optimal Performance of HIKD

(Table II). Further reduction is achieved by halting recursion at 128 bits: rather than decomposing a 128-bit multiplication into three separate 64-bit calls, a dedicated 128-bit routine is invoked, saving three leaf-level multiplications, two additive constructions, and recursion overhead. Consequently, throughput improves by 66.4%.

UKarat3 scheme: Expands the recursion depth of UKaratsuba by partitioning first-level operands into three parts, thereby reducing the number of recursive layers. Using the DP(3,2) configuration, it replaces the original 10,038 64-bit multiplications with 1,416 64-bit and 2,874 128-bit multiplications, achieving a 71.6% throughput gain.

UKarat5 scheme: Follows a similar approach with DP(3,2). It substitutes 10,101 64-bit multiplications with 417 64-bit and 3,228 128-bit multiplications, yielding a 62.7% improvement in throughput.

Among the UKarat3-3, UKarat5-3, UKarat3-5, and UKarat5-5 schemes, UKarat3-3 delivers the highest performance, surpassing not only the native UKaratsuba but also the other three variants, as shown in Figure 9. Compared to the standard UKaratsuba scheme, which requires 8,664 64-bit multiplications, UKarat3-3 inherently employs fewer multiplications. Furthermore, under the DP(4,2) configuration, it reduces 128-bit multiplications by 4.5% and 64-bit multiplications by 94.9% relative to UKaratsuba with DP(3,2). These structural and arithmetic optimizations collectively enhance peak polynomial multiplication throughput by 85.6%. The performance gains stem from both algorithmic refinement and reduced multiplication counts, offering an efficient reference for high-degree polynomial multiplication. On the HIGH setting, as shown in Figure 10, throughput reaches 917,488

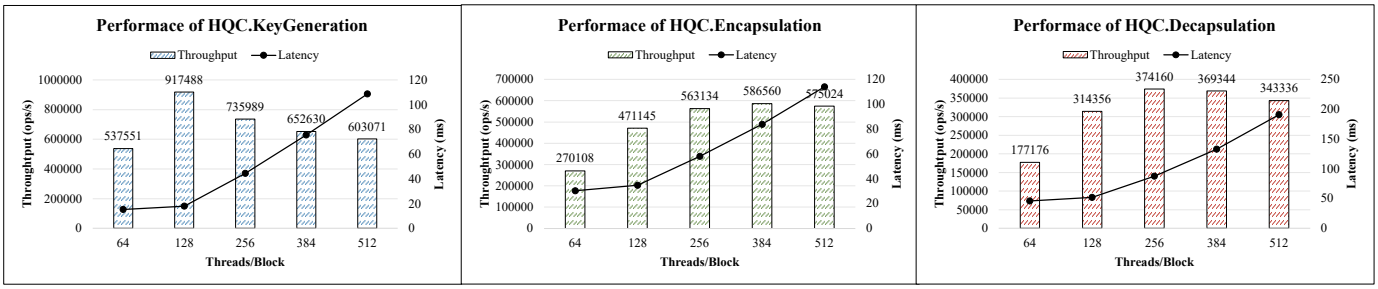


Fig. 10: Performance of HIGH

ops/s for KeyGeneration, 586,560 ops/s for Encapsulation, and 374,160 ops/s for Decapsulation—improvements of 106%, 67.6%, and 64.8 over the baseline, respectively. To highlight the contribution of algorithmic configuration, Table V also includes the original GPU-based HQC results. The overall acceleration thus arises from a combination of GPU hardware capabilities and systematic algorithmic optimization.

C. Comparison with Related Works

It is important to note that the majority of existing works leveraging CPUs or FPGAs are designed with the primary objective of achieving low-latency performance. C-Ref and AVX2-Ref [5] implementations were conducted on the Intel Core i7-7820X processor running at 3.6GHz for the three parameters of HQC. The former is a C++ implementation, while the latter is a fixed-time optimization using AVX2 instructions. As shown in Table V, AVX2-Ref achieves approximately twice the Encapsulation performance compared to C-Ref. Jean-Marc Robert et al. [12] leveraged the features of the AVX512 instruction set, employing the VPCLMULQDQ instruction for parallel computation of fourth-degree polynomial multiplication over $GF(2^m)$. Even considering the multi-core performance of CPU (Intel Core i7-1165G7, 4-core), under ideal conditions where resources are adequately available across cores, our HIGH implementation still demonstrates a significant throughput advantage over the multi-core CPU performance. Due to the lack of clock frequency data in [16], its performance cannot be accurately converted to throughput and is therefore excluded from Table V.

Carlos et al. [11] proposed an FPGA design for HQC utilizing HLS, demonstrating the impact of parameter modifications on achieving varying degrees of comparable results. Despite aiming for low hardware footprint and maintaining performance, particularly on low-cost FPGA devices, the observed performance improvements were modest. However, their implementation performance remains constrained to the range of tens of thousands of operations, while our throughput for the same parameter set exceeds by dozens times. To further contextualize our performance gains, we directly compare with the state-of-the-art FPGA implementation by [9], which is recognized for its high efficiency. As evidenced in Table V, our HIGH design demonstrates a substantial advantage, achieving approximately 100×, 82×, and 86× higher throughput in Key Generation, Encapsulation, and Decapsulation, respectively, compared to [9]. This significant leap underscores the transformative potential of leveraging massive GPU parallelism for code-based cryptography when high throughput is the primary objective. It is worth noting that GPU-based computations have inherent disadvantages in latency compared to CPUs, primarily due to the differences in single-core computational resources between GPUs and CPUs. However, GPUs excel in achieving high-throughput performance. Current GPU optimizations [15], [23], [28] typically operate within millisecond-level latencies, which are acceptable for most application scenarios, such as mobile apps and web authentication, where such delays are imperceptible to users. Therefore, the primary optimization goal for GPUs is to achieve higher throughput within an acceptable latency range.

BIKE and Classic McEliece [2], the other two code-based candidates in NIST's fourth round, were not progressed. In contrast, HQC stands as the sole code-based key encapsulation mechanism to be standardized. Moreover, our proposed HIGH optimization substantially outperforms the current state-of-the-art HQC implementation [12], achieving speedups of 20× in Key Generation, 32× in Encapsulation, and 39× in Decapsulation. While previous GPU-based accelerated research on post-quantum cryptography has predominantly focused on lattice-based schemes, this work fills a critical gap by pioneering the high-performance implementation of code-based cryptography on GPUs, thereby providing a crucial foundation for its practical engineering applications.

V. Conclusion

In this paper, we explore HIKD structures implemented on GPU, integrating high-order and low-order multiplication schemes. For HIKD, we investigate various integration schemes, leveraging the powerful parallel processing capabilities of GPU. Experimental results demonstrate that our HIGH achieves a new throughput record compared to existing implementations, with improvements of 20×, 32×, and 39× in KeyGeneration, Encapsulation, and Decapsulation, respectively. The HIKD structure not only significantly accelerates the HQC algorithm, but also, due to its versatility, can be widely applied in other polynomial multiplication scenarios, thus enhancing computational efficiency in related fields.

References

[1] P. W. Shor, "Polynomial time algorithms for discrete logarithms and factoring on a quantum computer," in *Algorithmic Number Theory: First International Symposium, ANTS-I Ithaca, NY, USA, May 6–9, 1994 Proceedings 1*. Springer, 1994, pp. 289–289.

TABLE V: Comparison of Throughput on HQC with Related Works

Implementation	PQC Algorithm	Platform		KeyGeneration (ops/s)	Encapsulation (ops/s)	Decapsulation (ops/s)
Melchor et al. [5] (NIST Round 4)	HQC	Intel Core i7-7820X CPU @ 3.6GHz (one-core*)	C	761	379	249
			C++	20,000	8,823	4,114
			AVX2	41,379	17,647	9,944
Robert et al. [12]		Intel Core i7-1165G7 CPU @ 2.80GHz (one-core)	AVX512	43,193	17,679	9,313
Sanjay Deshpande et al. [9]		Xilinx Artix-7 FPGA	Verilog	9,091	7,143	4,348
Carlos et al. [11]		Xilinx Artix-7 xc7a100t-ftg256-1	HLS(C++)	3,711	1,705	787
Schoffel et al. [10]		ASIC (22nm)	-	12,500	5,340	1,260
Schoffel et al. [10]		FPGA(XilinxArtix-7)	-	1,790	763	179
Nicolas et al. [4]	BIKE	Intel Xeon Platinum 8175M CPU @ 2.50GHz (one-core)	C	6,827	33,811	2,123
Fu et al. [31]	BIKE	NVIDIA RTX 4090	PTX ISA	24,033	277,789	5,817
Tung et al. [3]	McEliece	Intel Haswell CPU @ 3.10GHz (one-core)	C	51	82,480	24,282
Wu et al. [32]	McEliece348864	NVIDIA RTX 4090	PTX ISA	-	28,628,195	3,051,701
Seo et al. [43]	SIKE	NVIDIA GeForce RTX 2080 Ti	CUDA PTX	36,376	25,603	22,211
Ji et al. [28]	Kyber	nvidia tESLA v100 Volta GV100, 5120,CUDA cores	CUDA PTX	2,022,552	2,105,977	9,378,937
Our HIGH	HQC	NVIDIA GeForce RTX 4090	CUDA PTX <<<128,384>>>	**917,488**	**586,560**	**374,160**

* "one-core" refers to the single-core performance based on the platform.

[2] NIST. (2023) Post-Quantum Cryptography Standardization. [Online]. Available: https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization

[3] D. J. Bernstein, T. Chou, T. Lange, I. von Maurich, R. Misoczki, R. Niederhagen, E. Persichetti, C. Peters, P. Schwabe, N. Sendrier *et al.*, "Classic mceliece: conservative code-based cryptography," *Project documentation: https://classic. mceliece. org/nist/mceliece-20190331. pdf,24.12. 2021)*, 2017.

[4] N. Aragon, P. Barreto, S. Bettaieb, L. Bidoux, O. Blazy, J.-C. Deneuville, P. Gaborit, S. Ghosh, S. Gueron, T. Güneysu *et al.*, "Bike: bit flipping key encapsulation," *NIST*, 2022.

[5] C. A. Melchor, N. Aragon, S. Bettaieb, L. Bidoux, O. Blazy, J.-C. Deneuville, P. Gaborit, E. Persichetti, G. Zémor, and I. Bourges, "Hamming quasi-cyclic (HQC)," *NIST PQC Round*, vol. 2, no. 4, p. 13, 2018.

[6] R. Azarderakhsh, M. Campagna, C. Costello, L. Feo, B. Hess, A. Jalali, D. Jao, B. Koziel, B. LaMacchia, P. Longa *et al.*, "Supersingular isogeny key encapsulation-submission to the nist's post-quantum cryptography standardization process, round 2 (2019)."

[7] L. Maino and C. Martindale, "An attack on sidh with arbitrary starting curve," *Cryptology ePrint Archive*, 2022.

[8] T. Liu, G. Ramachandran, and R. Jurdak, "Post-quantum cryptography for internet of things: a survey on performance and optimization," *arXiv preprint arXiv:2401.17538*, 2024.

[9] S. Deshpande, C. Xu, M. Nawan, K. Nawaz, and J. Szefer, "Fast and efficient hardware implementation of hqc," in *International Conference on Selected Areas in Cryptography*. Springer, 2023, pp. 297–321.

[10] M. Schöffel, J. Feldmann, and N. Wehn, "Code-based cryptography in iot: A hw/sw co-design of hqc. arxiv 2023," *arXiv preprint arXiv:2301.04888*, 2023.

[11] C. Aguilar-Melchor, J.-C. Deneuville, A. Dion, J. Howe, R. Malmain, V. Migliore, M. Nawan, and K. Nawaz, "Towards automating cryptographic hardware implementations: a case study of hqc," in *Code-Based Cryptography Workshop*. Springer, 2022, pp. 62–76.

[12] J.-M. Robert and P. Véron, "Faster multiplication over f 2 [x] using avx512 instruction set and vpclmulqdq instruction," *Journal of Cryptographic Engineering*, vol. 13, no. 1, pp. 37–55, 2023.

[13] T. Güneysu, V. Lyubashevsky, and T. Pöppelmann, "Practical lattice-based cryptography: A signature scheme for embedded systems," in *Cryptographic Hardware and Embedded Systems – CHES 2012*, E. Prouff and P. Schaumont, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 530–547.

[14] 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.

[15] L. Wan, F. Zheng, G. Fan, R. Wei, L. Gao, Y. Wang, J. Lin, and J. Dong, "A novel high-performance implementation of crystals-kyber with ai accelerator," in *European Symposium on Research in Computer Security*. Springer, 2022, pp. 514–534.

[16] A. Ras, A. Loiseau, M. Carmona, S. Pontié, G. Renault, B. Smith, and E. Valea, "Optimizing HQC using Frobenius Additive FFT on a RISC-V-based System-on-Chip," in *DSD/SEAA 2025 - 28th Euromicro Conference Series on Digital System Design*, Salerne, Italy, Sep. 2025. [Online]. Available: https://cea.hal.science/cea-05318642

[17] D. S. Roche, "What can (and can't) we do with sparse polynomials?" in *Proceedings of the 2018 ACM International Symposium on Symbolic and Algebraic Computation*, 2018, pp. 25–30.

[18] A. Guimarães, D. Aranha, and E. Borin, "Secure and efficient software implementation of qc-mdpc code-based cryptography," in *Simpósio em Sistemas Computacionais de Alto Desempenho (SSCAD)*. SBC, 2019, pp. 116–117.

[19] P. Gaborit, C. Aguilar-Melchor, N. Aragon *et al.*, "Hamming quasi-cyclic (hqc) specification," NIST PQC Submission, 2025, version dated 2025/08/22. [Online]. Available: https://pqc-hqc.org

[20] A. A. Karatsuba and Y. Ofman, "Multiplication of multidigit numbers on automata," *Soviet physics. Doklady*, vol. 7, pp. 595–596, 1963. [Online]. Available: https://api.semanticscholar.org/CorpusID:117858583

[21] R. P. Brent, P. Gaudry, E. Thomé, and P. Zimmermann, "Faster multiplication in gf(2)[x]," in *Algorithmic Number Theory*, A. J. van der Poorten and A. Stein, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 153–166.

[22] NVIDIA, "CUDA C programming guide 9.0," https://docs.nvidia.com/cuda/cuda-c-programming-guide/, 2017.

[23] J. Dong, F. Zheng, N. Emmart, J. Lin, and C. Weems, "sDPF-RSA: Utilizing floating-point computing power of GPUs for massive digital signature computations," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2018, pp. 599–609.

[24] W. Pan, F. Zheng, W. Zhu, and J. Jing, "An efficient elliptic curve cryptography signature server with GPU acceleration," *IEEE Transactions on Information Forensics and Security*, 2017.

[25] L. Gao, F. Zheng, N. Emmart, J. Dong, J. Lin, and C. Weems, "DPF-ECC: Accelerating Elliptic Curve Cryptography with Floating-Point Computing Power of GPUs," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2020, pp. 494–504.

[26] Z. Wang, X. Dong, H. Chen, and Y. Kang, "Efficient gpu implementations of post-quantum signature xmss," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 3, pp. 938–954, 2023.

[27] 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.

[28] X. Ji, J. Dong, T. Deng, P. Zhang, J. Hua, and F. Xiao, "Hi-kyber: A novel high-performance implementation scheme of kyber based on gpu," *IEEE Transactions on Parallel and Distributed Systems*, 2024.

[29] D. J. Bernstein, T. Chen, C.-M. Cheng, and T. Lange, "Post-quantum cryptography on gpus," *Cryptology ePrint Archive*, 2017, report 2017/761.

[30] G. Guerreau and F. Dahan, "High-throughput bike implementations on gpus," *IEEE Transactions on Computers*, vol. 72, no. 8, pp. 1124–1137, 2023.

[31] J. Dong, Y. Fu, X. Qin, Z. Dong, F. Xiao, and J. Lin, "Eco-bike: Bridging the gap between pqc bike and gpu acceleration," *IEEE Transactions on Information Forensics and Security*, vol. 19, pp. 8952–8965, 2024.

[32] W. Wu, J. Dong, Z. Xu, Z. Dong, D. Duong, F. Xiao, and J. Lin, "Symphony of speeds: Harmonizing classic mceliece cryptography with gpu innovation," *IEEE Transactions on Information Forensics and Security*, vol. 20, pp. 8746–8759, 2025.

[33] NVIDIA Corporation, *CUDA C++ Programming Guide*. NVIDIA Corporation, 2025, available at: https://docs.nvidia.com/cuda/cuda-c-programming-guide.

[34] M. Harris, S. Sengupta, and J. D. Owens, "Parallel prefix sum (scan) with cuda," in *GPU gems*, vol. 3, no. 39. Addison-Wesley, 2007, pp. 851–876.

[35] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "Gpu computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

[36] NVIDIA Corporation, "Nvidia ampere ga102 gpu architecture," NVIDIA, Tech. Rep., 2020.

[37] R. J. McEliece, "A public-key cryptosystem based on algebraic coding theory," *DSN progress report*, vol. 42, no. 44, pp. 114–116, 1978.

[38] E. R. Berlekamp, R. J. McEliece, and H. C. A. van Tilborg, "The inherent intractability of certain coding problems (corresp.)," *IEEE Transactions on Information Theory*, vol. 24, no. 3, pp. 384–386, 1978.

[39] T. P. Berger and P. Loidreau, "Reducing key length of the mceliece cryptosystem," in *International Workshop on Coding and Cryptography*. Springer, 2009, pp. 1–17.

[40] D. Hofheinz, K. Hövelmanns, and E. Kiltz, "A modular analysis of the fujisaki-okamoto transformation," in *Theory of Cryptography Conference*. Springer, 2017, pp. 341–371.

[41] M. B. İlter, "An analysis on efficient polynomial multiplication algorithms for cryptographic purposes," Ph.D. dissertation, Ph. D. thesis, Middle East Technical University, 2016.

[42] R. P. Brent, P. Gaudry, E. Thomé, and P. Zimmermann, "Faster multiplication in gf (2)[x]," in *Algorithmic Number Theory: 8th International Symposium, ANTS-VIII Banff, Canada, May 17-22, 2008 Proceedings 8*. Springer, 2008, pp. 153–166.

[43] S. C. Seo, "Sike on gpu: Accelerating supersingular isogeny-based key encapsulation mechanism on graphic processing units," *Ieee Access*, vol. 9, pp. 116 731–116 744, 2021.



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



Yuze Hou received the B.E. degree from Linyi University, and is pursuing master's degree in electronic information from Nanjing University of Posts and Telecommunications. His research interests include cryptographic engineering, public key cryptography and applied cryptography.



Shiqin Wang received the B.E. degree from Nanjing University of Posts and Telecommunications, and is pursuing master's degree in computer science and technology from Nanjing University of Posts and Telecommunications. Her research interests include cryptographic engineering, post quantum cryptography and applied cryptography.



Letian Sha (Member, IEEE) received the BS degree in information security from Southwest Jiaotong University, China, in 2007, the MS degree in information security from the University of Electronic Science and Technology, China, in 2010, and the PhD degree in information security from Wuhan University, China, in 2014. He is currently a professor with the School of Computer Science, Nanjing University of Posts and Telecommunications. His research interests include software security, cyber security and defense in Internet of Things.



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



Zhenjiang Dong received the Ph.D. degree from Shanghai Jiao Tong University, Shanghai, China. He is currently a full professor at School of Computer Science in in Nanjing University of Posts and Telecommunications. His current research interests are mainly in security of data element flows, privacy computing, cyberspace security and blockchain.



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