Contents lists available at ScienceDirect

ICT Express

journal homepage: www.elsevier.com/locate/icte

Optimized implementation of HQC on Cortex-M4

DongCheon Kim [a], JunHyeok Choi [a], SeungYong Yoon [b], Seog Chung Seo [a,*]

[a] Department of Cybersecurity, Kookmin University, Seoul, South Korea
[b] Cyber Security Research Division, Electronics and Telecommunications Research Institute, Daejeon, South Korea

ARTICLE INFO

Keywords:
HQC
Cortex-M4
Optimization
Binary polynomial
Multiplication

ABSTRACT

In March 2025, NIST selected HQC as a standardized PQC algorithm. Since HQC relies on binary polynomial operations, optimizations for prime-field schemes like Kyber are not directly applicable. Furthermore, optimizing HQC on Cortex-M4 involves constraints that complicate objective performance evaluation, which has hindered active research in this area. We address these issues and optimize dense-dense polynomial multiplication, HQC's main computational bottleneck. Using the PQM4 benchmark framework, our implementation achieves speedups of 1139.53–1347.69% in key generation, 1139.53–1253.73% in encapsulation, and 1042.09–1198.78% in decapsulation over PQClean, and 38.78–45.81%, 38.18–45.58%, and 34.76–43.56% improvements over the NTL-based reference, depending on the security level.

1. Introduction

To establish secure cryptographic communication in the rapidly advancing quantum computing era, NIST (National Institute of Standards and Technology) launched the PQC (Post-Quantum Cryptography) standardization process in 2016. After three rounds, four algorithms were selected in 2022 [1,2]. To ensure diversity in the mathematical hardness assumptions underlying the security of PQC algorithms, NIST proceeded with Round 4 and, in March 2025, officially selected HQC (Hamming Quasi-Cyclic) as a standardized KEM (Key Encapsulation Mechanism) [3].

HQC is a code-based cryptography designed using binary field operations, unlike previously standardized PQC algorithms that primarily rely on prime field arithmetic. Binary field operations, which utilize simple logical operations such as XOR (Exclusive-OR) and shift without carry propagation, offer advantages in terms of hardware implementation efficiency. Accordingly, the HQC reference implementation includes an optimized version for Artix-7 FPGA (Field Programmable Gate Array) [4].

However, in edge-network environments such as IoT devices and low-power systems, most platforms are resource-constrained embedded devices [5]. Using a basic software implementation or integrating dedicated cryptographic hardware into embedded devices for such environments would be inefficient in terms of both cost and deployment time. Furthermore, rapid adoption of PQC is crucial to counter security threats like harvest now, decrypt later attacks [6].

However, executing computationally intensive PQC algorithms on embedded devices incurs significant performance overhead, which may hinder their practical adoption in such constrained environments. Recent studies have investigated the integration of PQC into TLS (Transport Layer Security) 1.3 and conducted benchmark evaluations on embedded platforms [7]. The results indicate that most PQC algorithms demonstrate inferior performance compared to conventional public-key cryptographic schemes. These findings emphasize the necessity of optimizing PQC not only at the algorithmic level, but also in terms of protocol-level integration for real-world applications. To achieve protocol performance comparable to or exceeding that of existing solutions, further improvements in the efficiency of PQC implementations are essential.

To address this, NIST recommends evaluating the feasibility and efficiency of PQC implementations in resource-constrained environments, such as micro-controllers, with Cortex-M4 as a reference platform [8]. Since HQC has been selected as a standardized KEM, optimizing its implementation for resource-constrained environments is essential. In this study, we analyze the constraints of implementing HQC on Cortex-M4, identify the most performance-critical operations, and optimize these computations to reduce overhead.

1.1. Related work

As mentioned earlier, research on deploying HQC in embedded systems is crucial. However, to the best of our knowledge, there has been only one study on HQC implementation for Cortex-M, conducted by R. Aissaoui et al4 [9]. This study leveraged the architectural characteristics of Cortex-M4 and proposed addition and multiplication techniques

* Corresponding author.
E-mail addresses: kindongsy@kookmin.ac.kr (D. Kim), c981126@kookmin.ac.kr (J. Choi), syyoon@etri.re.kr (S. Yoon), scseo@kookmin.ac.kr (S.C. Seo).

https://doi.org/10.1016/j.icte.2025.07.001
Received 25 March 2025; Received in revised form 6 June 2025; Accepted 3 July 2025
Available online 26 July 2025

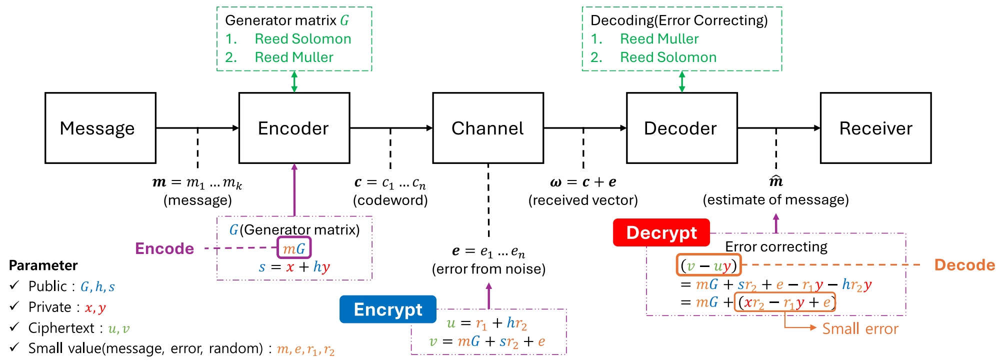

Fig. 1. HQC PKE structure overview.

that were not included in the reference implementation [4]. R. Aissaoui et al.'s method stores the positions (indexes) of 1s in a sparse binary secret vector within an array and aligns the bit positions using shift and XOR operations based on these indices. Since this approach relies solely on XOR and shift operations using an index array, it is highly efficient in terms of both computation and memory usage.

However, in actual implementation, conditional statements dependent on the indices of 1s are required to determine the shift operations, as both array indices and bit positions must be considered. This results in variable execution times based on input data, making it vulnerable to timing attacks that could reveal the secret vector's 1s positions. Therefore, to ensure resistance against side-channel attacks, an optimized constant-time implementation remains essential [10–12].

1.2. Our contributions

This paper presents optimization techniques for HQC on Cortex-M4 and provides an objective performance evaluation based on our implementation. For a fair comparison, we evaluate both cycle count and stack usage. We apply suitable optimizations for all security levels of HQC, and our contributions are summarized as follows:

- ***Optimized Dense-dense Multiplication and Their Comprehensive Analysis on Cortex-M4***
  To avoid some possible leakage of secret information, as noted in the HQC specification for non-constant-time implementations, we employed the Karatsuba n-way algorithm (KAn) and Toom–Cook n-way algorithm (TCn) for dense-dense multiplication which does not depend on sparsity of polynomial [4]. We explored and comprehensively compared higher-degree binary polynomial multiplication methods based on dense-dense multiplication algorithms across all security levels of HQC, identifying the most efficient approach.
- ***Objective Performance Benchmark***
  To ensure our results serve as objective performance benchmarks, we conducted a performance analysis using PQM4, a well-established Cortex-M4 PQC implementation framework. As a result, our implementation achieves performance improvements of 1139.53–1347.69% in KG (key generation), 1139.53–1253.73% in ENC (encapsulation), and 1042.09–1198.78% in DEC (decapsulation) compared to the PQClean baseline in PQM4, and 38.78–45.81%, 38.18–45.58%, and 34.76–43.56% improvements compared to the reference implementation utilizing the NTL (Number Theory Library), depending on the security level.
- ***HQC Reference Implementation on PQM4***
  We resolved the dependency-related constraints that had previously made it difficult to integrate the HQC reference implementation, which was finalized as a standardized algorithm in NIST PQC Round 4, into Cortex-M4. We then extracted only the optimized components from the dependent libraries and merged them into the PQM4 framework to enable a fair performance comparison and conduct our performance evaluation.

Table 1
Parameters of the HQC.

Security level	n	pk size	sk size	ct size	ss size
HQC-128	17,669	2249	56	4,497	64
HQC-192	35,851	4522	64	9,042	64
HQC-256	57,637	7245	72	14,485	64

The source code is available at https://github.com/kindongsy/ICTE-HQC_Cortex-M4.

2. Preliminaries

2.1. HQC

HQC is a KEM algorithm based on the hardness of the syndrome decoding problem. It employs a concatenated code structure, combining two distinct codes in series: Reed–Muller code (RM) and Reed–Solomon code (RS) [13,14]. Table 1 presents the polynomial degree (n) and the corresponding parameter sizes for each NIST security level of HQC.

Algorithm 1 Karatsuba 2-way Algorithm

```
1: Input: $A$ and $B$ on $m$ words($W$).
2: Output: $R = A \times B$
3: If $m = 1$ then return BaseMultiplication($A, B$)
4: else
5:     // Split in two halves of word size $m/2$.
6:     $A \leftarrow A_0 + x^{Wm/2}A_1, \quad B \leftarrow B_0 + x^{Wm/2}B_1$
7:     // Recursive multiplications
8:     $R_0 \leftarrow$ KA2($A_0, B_0, m/2$)
9:     $R_1 \leftarrow$ KA2($A_1, B_1, m/2$)
10:    $R_2 \leftarrow$ KA2($A_0 + A_1, B_0 + B_1, m/2$)
11:    // Reconstruction
12:    $R \leftarrow R_0 + (R_0 + R_1 + R_2)x^{Wm/2} + R_1x^{Wm}$
13:    return $R$
14: end if
```

HQC achieves IND-CCA2 security by applying the Fujisaki–Okamoto transform to a Public-Key Encryption (PKE) scheme [15]. The HQC PKE algorithm structure is illustrated in Fig. 1. The input message undergoes a serial encoding process using RS and RM codes, followed by the encryption step, where errors are injected. The decryption process then applies RM and RS codes in reverse order, followed by error correction to recover the original message.

2.2. Binary field arithmetic

HQC is a code-based cryptography designed to operate over a binary field. Since the coefficients of polynomials in a binary field are restricted to 0 and 1, addition and subtraction are performed using XOR operations, while multiplication is carried out as carry-less multiplication due to the absence of carry propagation in this arithmetic.

Algorithm 2 Toom-Cook 3-Way Algorithm

```
1: Input: Polynomials $A = a_2X^2 + a_1X + a_0$, $B = b_2X^2 + b_1X + b_0$ in $GF(2)[x]$
2: Output: $C(X) = A(X) \cdot B(X)$
3: Let $W = x^w$
4: $c_0 \leftarrow a_1W + a_2W^2$, $c_4 \leftarrow b_1W + b_2W^2$, $c_5 \leftarrow a_0 + a_1 + a_2$, $c_2 \leftarrow b_0 + b_1 + b_2$
5: $c_1 \leftarrow c_2 \times c_5$, $c_5 \leftarrow c_5 + c_0$, $c_2 \leftarrow c_2 + c_4$, $c_0 \leftarrow c_0 + a_0$
6: $c_4 \leftarrow c_4 + b_0$, $c_3 \leftarrow c_2 \times c_5$, $c_2 \leftarrow c_0 \times c_4$, $c_0 \leftarrow a_0 \times b_0$
7: $c_4 \leftarrow a_2 \times b_2$, $c_3 \leftarrow c_3 + c_2$, $c_2 \leftarrow c_2 + c_0$, $c_2 \leftarrow c_2/W + c_3$
8: $c_2 \leftarrow \left(c_2 + (1 + W^3)c_4\right)/(1 + W)$, $c_1 \leftarrow c_1 + c_0$, $c_3 \leftarrow c_3 + c_1$
9: $c_3 \leftarrow c_3/(W^2 + W)$, $c_1 \leftarrow c_1 + c_2 + c_4$, $c_2 \leftarrow c_2 + c_3$
10: Return $c_4X^4 + c_3X^3 + c_2X^2 + c_1X + c_0$
```

Not only in HQC but also in most PQC schemes operating over a binary field, polynomial multiplication remains the primary computational bottleneck. Therefore, optimizing polynomial multiplication continues to be the top priority in enhancing efficiency [16].

2.3. Karatsuba algorithm

The KA is a divide-and-conquer based method designed to accelerate polynomial multiplication. It is classified as KAn depending on how many parts the input polynomials are divided into. While the traditional school-book multiplication has a time complexity of $O(n^2)$, the standard $KA2$ leverages its recursive structure to achieve a complexity of $O(n^{\log_2 3})$, approximately $O(n^{1.585})$, making it more efficient as the polynomial degree increases [17].

The $KA2$, as shown in Algorithm 1, each polynomial is divided into two halves. For a polynomial of length n, this method requires three multiplications on polynomials of size $n/2$ and $3n-4$ additions.

2.4. Toom–Cook algorithm

The TCn algorithm is a generalized form of KAn, where a polynomial is divided into n ($n \geq 3$) parts and undergoes four major steps: Evaluation, Pointwise Multiplication, Interpolation, and Recomposition to compute the coefficients corresponding to the product polynomial's degree, achieving a time complexity of $O(n^{\log_3 5})$ (approximately $O(n^{1.465})$) [18].

To apply the $TC3$, five evaluation points are required. However, when multiplying binary polynomials such as in HQC, where coefficients are restricted to 0 and 1, these evaluation points cannot be directly utilized. Instead, word-based operations are considered using W, where $W = x^{wordsize}$. Since W can be efficiently processed via shift operations, the evaluation points W and $W+1$ are used. The optimized algorithm for binary polynomial multiplication based on this approach is presented in Algorithm 2 [19].

At first glance, the TC may seem more efficient than the KA for high-degree binary polynomial multiplication. However, it is essential to consider the additional addition/subtraction operations introduced in TC's interpolation step, which are necessary to reduce the number of multiplications. Furthermore, factors such as how the polynomial is partitioned (n-way division), the recursive operations required for different degrees, and the characteristics of the computing environment must be taken into account. Given these considerations, an objective comparison should be conducted by directly integrating the algorithm into architectures where it will be deployed.

2.5. Challenges in implementation on Cortex-M4

- ***HQC Implementation on PQM4***
  The PQM4 project is a framework for benchmarking PQC implementations on Arm Cortex-M4. It is designed to provide a fair comparison of various PQC algorithms under the same criteria. Due to its reliance on external libraries such as NTL for polynomial multiplication, the HQC reference implementation has not yet been incorporated into PQM4 [8]. Instead, PQM4 leverages PQClean implementations for benchmarking HQC, as they eliminate dependencies on external libraries. However, PQClean differs from the HQC reference implementation, which utilizes optimized multiplication algorithms from the previously mentioned libraries.
- ***Carry-less Arithmetic on Cortex-M4***
  As mentioned earlier, code-based cryptography, such as HQC, is primarily designed based on binary field arithmetic. While high-performance CPUs support carry-less operations through specialized architectures, such as the AVX instruction set, this is not the case for Cortex-M4 [20]. Furthermore, due to these characteristics, optimization techniques developed for prime-field-based cryptographic algorithms cannot be directly applied to $GF2[x]$ operations. Additionally, research on optimizing computational bottlenecks on Cortex-M4 has been relatively limited.
- ***Optimized and Secure Implementation***
  Examining the multiplication process, we observe that HQC's secret values (x, y) and small values (e, r_1, r_2) participate in the computation. These binary vectors exhibit a sparse property, meaning that the number of 1s is relatively low. Multiplication leveraging the sparsity of vectors can, from an implementation perspective, lead to vulnerabilities in constant-time execution and memory access safety. Therefore, a dense-dense multiplication approach is recommended in this context [4]. However, due to the high polynomial degree, this approach also comes with a significant computational overhead [9,10].

Table 2
HQC-256 performance profiling.

Alg.	Function	Ratio
KG	`shake_prng`	0.008
	`seedexpander_init`	0.002
	`set_random_fixed_weight`	1.913
	`vect_set_random`	0.462
	`vect_mul`	**97.576**
	`vect_add`	0.004
	`hqc_public_key_to_string`	0.017
	`hqc_secret_key_to_string`	0.017
Total	296,502,394 (Clock Cycle)	
ENC	`vect_set_random_from_prng`	0.04
	`shake256_512_ds`	0.685
	`hqc_pke_encrypt`	**99.294**
	`hqc_ciphertext_to_string`	0.017
Total	594,750,705 (Clock Cycle)	
DEC	`hqc_ciphertext_from_string`	0.011
	`hqc_pke_decrypt`	**33.470**
	`shake256_512_ds`	0.456
	`hqc_pke_encrypt`	**66.049**
	`vect_compare`	0.015
Total	894,101,373 (Clock Cycle)	

3. Our optimized implementation

In this section, we describe the optimization strategies applied to HQC on Cortex-M4. An overview of the optimization process is shown in Fig. 2.

3.1. HQC performance profiling

The emphasis on binary polynomial multiplication as the primary optimization target for HQC is supported by profiling results. Table 2 presents the profiling data for HQC-256 on Cortex-M4, obtained using the PQClean implementation (2023-04-30 submission) provided by PQM4. The profiling was conducted by measuring the

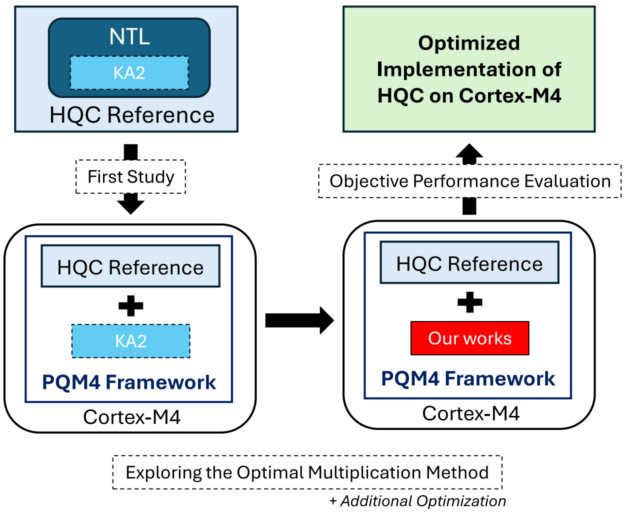
NTL
KA2
HQC Reference
First Study
HQC Reference
KA2
PQM4 Framework
Cortex-M4
Optimized Implementation of HQC on Cortex-M4
Objective Performance Evaluation
HQC Reference
Our works
PQM4 Framework
Cortex-M4
Exploring the Optimal Multiplication Method
+ Additional Optimization

Fig. 2. Optimization Process Overview.

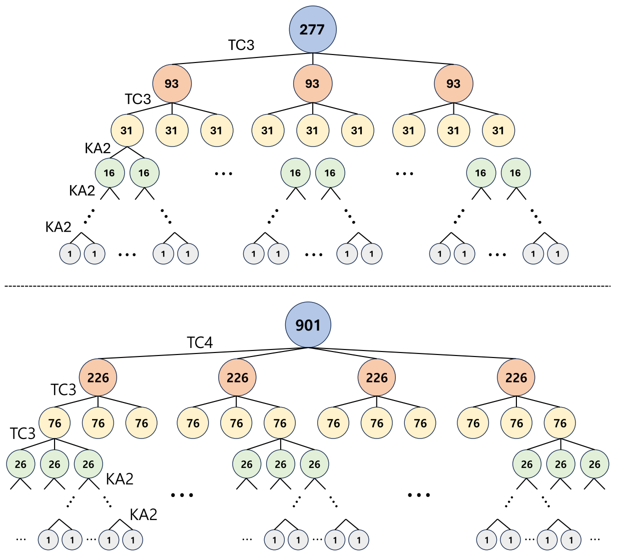

Fig. 3. Configurations of polynomial multiplication (Top: HQC-128, Bottom: HQC-256).

Table 3
Optimal polynomial multiplication for HQC (Clock Cycle).

Poly Mul	HQC-128	HQC-192	HQC-256
$KA2$ (NTL)	4,649,423	14,111,027	27,299,953
$TC3 + KA2$	3,701,811	11,005,380	26,379,978
$TC4 + KA2$	4,020,190	11,778,922	22,510,524
$TC3 + TC3 + KA2$	**3,388,429**	9,693,627	22,125,878
$TC3 + KA3 + KA2$ (Opt.)	3,920,622	11,636,597	30,205,574
$TC4 + TC3 + TC3 + KA2$	3,633,517	**9,581,246**	**19,235,636**
$TC4 + KA3 + KA3 + KA2$	4,123,510	12,054,795	28,887,498
PQClean	51,505,609	158,250,520	289,320,261
Improvement (vs. PQClean)	**1420.04%↑**	**1551.66%↑**	**1404.08%↑**
NTL	4,649,423	14,111,027	27,299,953
Improvement (vs. NTL)	**37.21%↑**	**47.27%↑**	**41.92%↑**
Additional Optimization	3,141,544	8,883,542	17,857,915
Improvement (vs. C)	**7.86%↑**	**7.86%↑**	**7.71%↑**

performance of all internal functions. As seen in the KG results, the `vect_mul` function, which performs binary polynomial multiplication, accounts for over 97% of the total computational overhead in KG. Similarly, in the ENC and DEC processes, the `hqc_pke_encrypt` and `hqc_pke_decrypt` functions exhibit the highest computational burden, with `vect_mul` contributing 98% and 96% of the total operations, respectively. These results clearly indicate that for efficient HQC optimization, improving the performance of binary polynomial multiplication must be the primary focus.

3.2. Adopting existing implementations

The HQC reference implementation utilizes NTL to optimize $GF2[x]$ multiplication by employing a $KA2$ recursive algorithm tailored to polynomial length. To integrate it into PQM4, we eliminated its dependency on external libraries and adapted the extracted NTL-based optimization for Cortex-M4. As shown in Table 3, this NTL-based optimization alone results in significant performance improvements.

We further developed the optimized HQC software by implementing the $TC3$ + $KA3$ ($KA5$ for HQC-256) + $KA2$ algorithm in C, which was originally proposed in AVX2 format. Performance evaluations across all security levels showed a performance improvement of 18.58% for HQC-128 and 21.49% for HQC-192 compared to the NTL-based reference implementation, whereas HQC-256 showed a 4.08% performance degradation. These results highlight the need to explore more suitable optimization techniques across different security levels.

3.3. Exploring the optimal multiplication method

Building upon the multiplication algorithms proposed in the HQC submission, we explored the most efficient multiplication configurations for each security level by combining $TC3$ and $TC4$ with $KA2$ and $KA3$. Fig. 3 illustrates the polynomial decomposition method for HQC-128 and HQC-256. In the case of HQC-128, the polynomial array consisting of 277 64-bit words is decomposed using the TC3+TC3+KA2 combination. For HQC-256, the polynomial array consists of 901 64-bit words and is decomposed using the TC4+TC3+TC3+KA2 configuration.

As shown in Table 3, we conducted a comprehensive performance analysis on algorithm configurations that could be efficiently arranged based on polynomial length. This approach led to the maximum performance improvements achieved in our study.

For HQC-128, the highest performance was achieved using the recursive $TC3$ + $TC3$ + $KA2$ multiplication strategy, resulting in a 37.21% improvement over the NTL-based implementation. For HQC-192 and HQC-256, the most efficient approach was the recursive $TC4$ + $TC3$ + $TC3$ + $KA2$ strategy, which achieved 47.27% and 41.92% improvements, respectively.

To determine the optimal multiplication strategy, we also considered algorithms beyond TC5. While increasing the number of polynomial splits can theoretically reduce computational complexity, it also introduces additional overhead from auxiliary operations, which may ultimately degrade overall performance [18]. Moreover, previous studies suggest that for binary polynomials of the degree used in HQC, such higher-degree divide-and-conquer algorithms may not necessarily offer performance advantages [21,22]. Therefore, we consider algorithms up to TC4, as they are sufficient to achieve a favorable balance between complexity and performance.

3.4. Selection of base multiplication method

Although we applied suitable divide-and-conquer algorithms for each HQC security level, the primary computational bottleneck remains the final word-level multiplication. To optimize this, we compared Window-Table-based multiplication, used in HQC and BIKE via NTL, with Bitwise serial multiplication, adopted in Classic McEliece and McBits [19,23,24].

Window-Table-based multiplication achieved 394cc, significantly outperforming Bitwise serial multiplication (3112cc). To ensure fairness, we excluded the reduction step embedded in McBits and Classic McEliece. Based on this result, we adopted Window-Table-based multiplication as the base method for all implementations.

Table 4
HQC Performance Comparison (cc : Clock Cycle, ss : Stack Size, K : $Kilo(=10^3)$, M : $Mega(=10^6)$)

Algorithm	HQC-128			HQC-192			HQC-256		
	KG	ENC	DEC	KG	ENC	DEC	KG	ENC	DEC
Our work (C) (cc)	4.38 M (0.22 s)	8.95 M (0.45 s)	14.63 M (0.73 s)	12.41 M (0.62 s)	24.99 M (1.25 s)	39.10 M (1.96 s)	29.83 M (1.49 s)	59.65 M (2.98 s)	93.15 M (4.65 s)
Our work (ASM) (cc)	4.10 M (0.21 s)	8.38 M (0.42 s)	13.78 M (0.69 s)	11.70 M (0.59 s)	23.56 M (1.18 s)	36.96 M (1.85 s)	23.55 M (1.18 s)	47.10 M (2.36 s)	74.31 M (3.72 s)
Improvement (vs. C)	**6.83% ↑**	**6.80% ↑**	**6.16% ↑**	**6.07% ↑**	**6.07% ↑**	**5.79% ↑**	**26.66% ↑**	**26.64% ↑**	**25.35% ↑**
PQClean (cc)	51.96 M	104.12 M	157.38 M	169.38 M	318.94 M	480.03 M	291.91 M	583.82 M	879.41 M
Improvement (vs. PQClean)	**1167.32% ↑**	**1142.48% ↑**	**1042.09% ↑**	**1347.69% ↑**	**1253.73% ↑**	**1198.78% ↑**	**1139.53% ↑**	**1139.53% ↑**	**1083.43% ↑**
NTL (cc)	5.69 M	11.58 M	18.57 M	17.06 M	34.30 M	53.06 M	33.16 M	66.31 M	103.13 M
Improvement (vs. NTL)	**38.78% ↑**	**38.18% ↑**	**34.76% ↑**	**45.81% ↑**	**45.58% ↑**	**43.56% ↑**	**40.80% ↑**	**40.78% ↑**	**38.78% ↑**
Our work (ASM) (ss)	37.0 K	54.9 K	61.5 K	73.5 K	109.5 K	123.1 K	117.1 K	175.0 K	196.6 K
NTL (ss)	37.0 K	54.8 K	61.5 K	73.5 K	109.5 K	123.0 K	117.3 K	175.1 K	196.8 K
Improvement (vs. NTL)	–	**0.18% ↑**	–	–	–	**0.08% ↑**	**0.17% ↓**	**0.05% ↓**	**0.10% ↓**

IDE : STM32CubeIDE 1.15.1, NTL : NTL-based reference implementation.
Improvement : $((Other - OurWork)/OurWork) \times 100$, Execution time : $ConsumedClockCycles/ClockConfiguration$.

3.5. *Further optimization with assembly*

To further optimize the algorithm in the Cortex-M4, we implemented the base multiplication of KA in assembly. In the assembly implementation, it is crucial to pipeline the instructions without stalls in order to achieve performance improvements. For example, the ldr instruction typically takes 2 cycles when used individually, but when n ldr instructions are arranged consecutively without dependencies, they can be processed in $n+1$ cycles due to pipeline. By applying this approach, we carefully arranged memory access instructions. Additionally, with regard to memory access instructions, stalls can occur when the destination register of one instruction becomes the source register for the next. To avoid this, we eliminated such dependencies, ensuring smooth execution without pipeline stalls. As a result, compared to the C implementation, we achieved performance improvements of 7.86%, 7.85%, and 7.71% in multiplication for HQC-128, HQC-192, and HQC-256, respectively [25].

4. Implementation results

We conducted our performance benchmarking on the NUCLEO-L4R5ZI, a 32-bit ARM Cortex-M4 board designated as the default platform in PQM4 for NIST PQC Round 4, using the same 20MHz clock configuration. Additionally, to evaluate the security of sparse binary vectors, which is a key contribution of this study, we compared the performance of constant-time dense-dense multiplication.

The comprehensive performance results for all HQC parameter sets are summarized in Table 4. For HQC-128, the $TC3+TC3+KA2$ multiplication method resulted in 1167.32% improvement in KG, 1142.48% in ENC, and 1042.09% in DEC compared to the PQClean implementation. Even compared to the NTL-based reference implementation, our approach achieved 38.78%, 38.18%, and 34.76% performance improvements, respectively.

For HQC-192 and HQC-256, the most efficient configuration was $TC4 + TC3 + TC3 + KA2$, which resulted in 1347.69% (1139.53%) improvement in KG, 1253.73% (1139.53%) in ENC, and 1198.78% (1083.43%) in DEC compared to PQClean. Against the NTL-based implementation, the performance gains were 45.81% (40.80%) in KG, 45.58% (40.78%) in ENC, and 43.56% (38.78%) in DEC, where values for HQC-256 are in parentheses.

Although each instruction on the Cortex-M4 has a fixed number of cycles defined by ARM documentation, these values do not account for practical overheads such as memory stalls or register pressure that may occur due to the processor's pipelined architecture [26]. In addition, since the Cortex-M4 supports a barrel shifter and allows for optimized memory access (e.g., consecutive `ldr` instructions), the actual number of executed instructions and their performance impact can vary depending on the compiler version, optimization level, and firmware environment. For these reasons, we believe that reporting empirical clock cycles measured on actual hardware, using profiling methods officially supported by Cortex-M4 and PQM4, provides a more realistic basis for performance comparison. We also make our source code publicly available, which may assist others in verifying instruction-level behavior and extending our results under various conditions.

We analyze stack usage to identify potential drawbacks that may arise when incorporating optimization techniques into embedded environments. When comparing stack usage with the NTL-based reference implementation, we find that the difference at the algorithmic level remains minimal, and in practice, our optimizations result in similar or even reduced memory usage.

Comparison to FAFFT To the best of our knowledge, no FFT (Fast Fourier Transform)-based techniques have been proposed so far for binary polynomial multiplication on the Cortex-M4, which constitutes the primary computational bottleneck in HQC. However, for BIKE, which shares similar requirements for efficient polynomial multiplication, Chen et al. introduced an approach applying FAFFT (Frobenius Additive FFT) to the scheme in [27]. This FAFFT implementation is publicly available through the PQM4 platform, and we were able to adapt the BIKE-3 implementation from the optimized FAFFT code presented by Chen et al. to suit HQC-128.

In our evaluation, the FAFFT-based multiplication for HQC-128 on Cortex-M4 recorded 3,508,854cc. In comparison, our proposed method, using the $TC3 + TC3 + KA2$ configuration, achieved 3,141,544cc, representing an 11.69% improvement. These results demonstrate that, despite the lower theoretical complexity of FFT-based multiplication, the overhead introduced by basis conversion and butterfly operations can be substantial in practice. This explains why our HQC-128 multiplication approach achieves better performance than the FAFFT-based method in actual Cortex-M4 environments.

While our method outperforms FAFFT under the current setting, the development of efficient basis conversion techniques and butterfly structures could potentially improve the practicality of FFT-like methods for use in constrained environments. We expect that the performance results provided in this work may serve as a useful baseline for future FFT-oriented optimizations of HQC.

5. Conclusion

This paper presents optimization results for HQC, which was selected as a final NIST PQC standardized algorithm, on the Cortex-M4 platform. For a fair performance evaluation, we compared the

NTL-based reference implementation and our work within the PQM4 framework. To mitigate potential implementation vulnerabilities, we employed dense-dense multiplication and constant-time algorithms.

CRediT authorship contribution statement

DongCheon Kim: Writing – review & editing, Methodology, Writing – original draft, Investigation, Validation, Conceptualization. **Jun-Hyeok Choi:** Writing – original draft, Validation, Writing – review & editing, Investigation. **SeungYong Yoon:** Conceptualization. **Seog Chung Seo:** Project administration, Writing – review & editing, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was partly supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government, Ministry of Science and ICT (MSIT) (No. 2022R1C1C1013368, 50%) and by the Institute of Information and Communications Technology Planning and Evaluation (IITP) grant funded by MSIT (No. RS-2023-00225201, Development of Control Rights Protection Technology to Prevent Reverse Use of Military Unmanned Vehicles, 50%).

References

[1] National Institute of Standards and Technology, Submission Requirements and Evaluation Criteria for the Post-Quantum Cryptography Standardization Process, Tech. Rep., National Institute of Standards and Technology, 2016.

[2] National Institute of Standards and Technology, NIST IR 8413 : Status Report on the Third Round of the NIST Post-Quantum Cryptography Standardization Process, Tech. Rep., National Institute of Standards and Technology, 2022.

[3] National Institute of Standards and Technology, NIST IR 8545 : Status Report on the Fourth Round of the NIST Post-Quantum Cryptography Standardization Process, Tech. Rep., National Institute of Standards and Technology, 2025.

[4] C.A. Melchor, N. Aragon, S. Bettaieb, L. Bidoux, O. Blazy, J.-C. Deneuville, P. Gaborit, E. Persichetti, G. Zémor, I. Bourges, Hamming quasi-cyclic (HQC), in: NIST PQC Round 4, 2025.

[5] F. Medina, Analysis and Contributions to a Post-Quantum Cryptography Library written in Rust for a ARM Cortex-M4 board (Ph.D. thesis), Politecnico di Torino, 2024.

[6] A.T. Olutimehin, S. Joseph, A.J. Ajayi, O.C. Metibemu, A.Y. Balogun, O.O. Olaniyi, Future-proofing data: Assessing the feasibility of post-quantum cryptographic algorithms to mitigate 'harvest now, decrypt later'attacks, in: Decrypt Later' Attacks (February 17, 2025), 2025.

[7] G. Tasopoulos, J. Li, A.P. Fournaris, R.K. Zhao, A. Sakzad, R. Steinfeld, Performance evaluation of post-quantum TLS 1.3 on resource-constrained embedded systems, in: International Conference on Information Security Practice and Experience, Springer, 2022, pp. 432–451.

[8] M.J. Kannwischer, J. Rijneveld, P. Schwabe, K. Stoffelen, pqm4: Testing and benchmarking NIST PQC on ARM cortex-M4, 2019.

[9] R. Aissaoui, J.-C. Deneuville, C. Guerber, A. Pirovano, A performant quantum-resistant KEM for constrained hardware: optimized HQC, in: 21st International Conference on Security and Cryptography, SCITEPRESS-Science and Technology Publications, 2024, pp. 668–673.

[10] J. Dong, Y. Fu, X. Qin, Z. Dong, F. Xiao, J. Lin, Eco-bike: Bridging the gap between PQC BIKE and GPU acceleration, IEEE Trans. Inf. Forensics Secur. (2024).

[11] L. Demange, BIKE Implementation: Vulnerabilities and Countermeasures (Ph.D. thesis), Sorbonne Université, 2024.

[12] G. Barthe, B. Grégoire, V. Laporte, Secure compilation of side-channel countermeasures: the case of cryptographic "constant-time", in: 2018 IEEE 31st Computer Security Foundations Symposium, CSF, IEEE, 2018, pp. 328–343.

[13] I.S. Reed, G. Solomon, Polynomial codes over certain finite fields, J. Soc. Ind. Appl. Math. 8 (2) (1960) 300–304.

[14] D.E. Muller, Application of Boolean algebra to switching circuit design and to error detection, Trans. IRE Prof. Group Electron. Comput. (3) (1954) 6–12.

[15] D. Hofheinz, K. Hövelmanns, E. Kiltz, A modular analysis of the Fujisaki-Okamoto transformation, in: Theory of Cryptography Conference, Springer, 2017, pp. 341–371.

[16] 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, 2024.

[17] C. Negre, Efficient binary polynomial multiplication based on optimized Karatsuba reconstruction, J. Cryptogr. Eng. 4 (2014) 91–106.

[18] Y. Spiizer, Toom-Cook with Base Change (Ph.D. thesis), The Hebrew University of Jerusalem, 2022.

[19] R.P. Brent, P. Gaudry, E. Thomé, 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.

[20] W. Beullens, F. Campos, S. Celi, B. Hess, M.J. Kannwischer, Nibbling MAYO: Optimized implementations for AVX2 and cortex-M4, IACR Trans. Cryptogr. Hardw. Embed. Syst. 2024 (2) (2024) 252–275.

[21] T. Granunland, GNU multiple precision arithmetic library (GMP), 2023, Online, URL https://gmplib.org/ (Accessed 20 March 2025).

[22] D.J. Bernstein, T. Lange, B. van Gastel, GF2x: A c library for fast arithmetic in gf(2)[x], 2024, https://gitlab.inria.fr/gf2x/gf2x (Accessed 04 June 2025).

[23] T. Chou, McBits revisited: toward a fast constant-time code-based KEM, J. Cryptogr. Eng. 8 (2) (2018).

[24] 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, NIST Submiss. 1 (1) (2017) 1–25.

[25] M. Fisher, ARM® Cortex® M4 Cookbook, Packt Publishing Ltd, 2016.

[26] Arm Ltd., Cortex-M4 devices generic user guide, 2010, https://developer.arm.com/documentation/ddi0439/b/CHDDIGAC (Accessed 5 June 2025).

[27] M.-S. Chen, T. Chou, M. Krausz, Optimizing bike for the intel haswell and arm cortex-m4, IACR Trans. Cryptogr. Hardw. Embed. Syst. (2021) 97–124.