Efficient Polynomial Multiplication for HQC on ARM Cortex-M4

Jihoon Jang[1], Myeonghoon Lee[2], Donggeun Kwon[3], Seokhie Hong[4], Suhri Kim[5*] and Sangjin Lee[1]

[1] Korea University, Seoul, South Korea, {jhw1031506,sangjin}@korea.ac.kr
[2] Samsung Electronics System LSI Business, Hwaseong, South Korea, ope2527@gmail.com
[3] Kunsan National University, Gunsan, South Korea, dgkwon@kunsan.ac.kr
[4] SmartM2M, Seoul, South Korea, shhong@smartm2m.co.kr
[5] Sungshin Women's University, Seoul, South Korea, suhrikim@sungshin.ac.kr

Abstract.
In this paper, we propose the Hybrid FAFFT-CRT method for accelerating HQC polynomial multiplication on the ARM Cortex-M4. The method uses the Chinese Remainder Theorem to map the polynomial ring to a product of an FAFFT-friendly ring of size 2^{d+1} and a small-degree residual ring, reducing the FAFFT transform length by half compared to the state-of-the-art. We also present the Hybrid Karatsuba-FAFFT method as an alternative hybrid based on a 2-way Karatsuba split, and apply radix-16 multiplication to HQC for the first time, with an operation-count cost model for selecting Karatsuba and Toom-Cook combinations. Additionally, we improve the core FAFFT butterfly through shortened XOR sequences, register scheduling, and `SWAPMOVE`-based bit swaps. On a NUCLEO-L4R5ZI board with a Cortex-M4 microcontroller, the Hybrid FAFFT-CRT method reduces polynomial multiplication cycles by 36.0% and 29.3% for HQC-1 and HQC-3, leading to cycle reductions for key generation, encapsulation, and decapsulation by 25.5%, 26.1%, and 23.5% for HQC-1, and 19.3%, 20.0%, and 19.1% for HQC-3. For HQC-5, our optimized butterfly FAFFT provides a consistent speedup of 1.4–1.6% across all KEM operations.

Keywords: Post-Quantum Cryptography, HQC, Binary polynomial multiplication, Cortex-M4, Additive FFT

1 Introduction

HQC [AAB+22] is a key encapsulation mechanism (KEM) that was officially selected as a NIST standard for post-quantum cryptography in March 2025. Since polynomial multiplication over $\mathbb{F}_2[x]$ is the core operation on HQC, the efficiency of HQC depends on how fast polynomial multiplication over $\mathbb{F}_2[x]$ can be performed. In particular, on embedded targets such as ARM Cortex-M4, which lack carry-less multiplication instructions, polynomial multiplication dominates the overall execution time. Figure 1 shows the cycle distribution for HQC-1 on the baseline FAFFT implementation [LJKH25] on the NUCLEO-L4R5ZI board. Polynomial multiplication accounts for 70.8%, 72.5%, and 65.4% of key generation, encapsulation, and decapsulation, respectively. The same dominance holds for the higher security levels, with polynomial multiplication taking 66–68% across key generation, encapsulation, and decapsulation in HQC-3 and 49–52% in HQC-5. This motivates polynomial multiplication as the primary optimization target on this platform. Two algorithmic frameworks for binary polynomial multiplication are particularly relevant

*Corresponding author.

on Cortex-M4. The first is the Frobenius Additive FFT (FAFFT) [LJKH25, CCK21], which reduces evaluation points by exploiting the Frobenius map. The second is a radix-16 method, proposed in [CC21], which turns $\mathbb{F}_2[x]$ polynomial multiplications into ordinary 32-bit integer multiplications that the Arm Cortex-M4 can execute efficiently. Motivated by these observations, we investigate FAFFT- and radix-16-based polynomial multiplication on the Cortex-M4 and propose several optimizations tailored to the HQC parameter sizes.

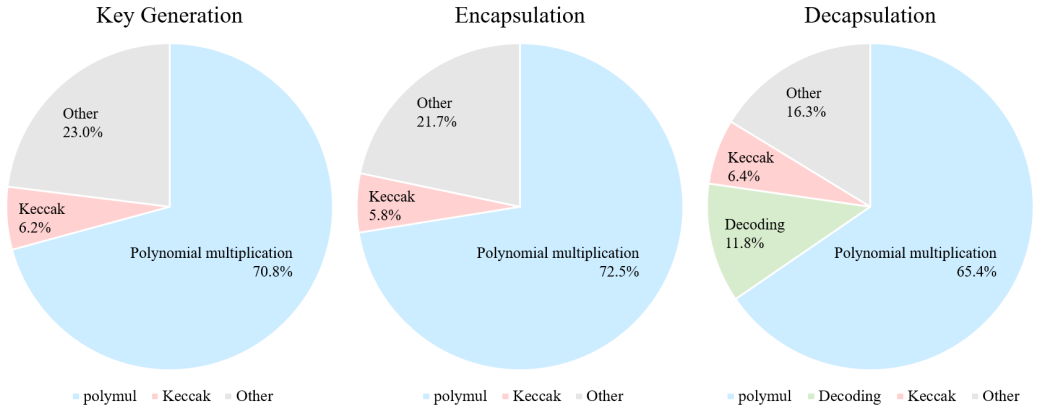

Figure 1: Cycle distribution of key generation, encapsulation, and decapsulation for HQC-1 on the baseline implementation [LJKH25] on Cortex-M4.

The ARM Cortex-M4 is the baseline embedded evaluation platform in the NIST PQC standardization project. As noted in NIST IR 8413 [AAC+22], the performance of PQC algorithms on embedded devices is one of the evaluation criteria. Specifically, the Cortex-M4 has been widely adopted in academic literature as a representative embedded platform because of its balance between resource constraints and computational capabilities. NIST has also formally acknowledged its prevalence and recommended it as an optimization target for the second round of the PQC project. As a result, many PQC schemes have been optimized for Cortex-M4 [BKS19, ABCG20, CHK+21, CC21, CCK21, GKS21, ACC+22, AHKS22, HZZ+22, HAZ+24, LJKH25]. In particular, pqm4 serves as the standard testing and benchmarking infrastructure for Cortex-M4 implementations [KPR+].

1.1 Related Work

For binary polynomial multiplication in code-based cryptography, baseline implementations typically rely on Toom-Cook and Karatsuba algorithms. Beyond this, two lines of work are particularly relevant to our approach: additive-FFT-based and radix-16-based methods.

The first approach is to use an additive FFT. Cantor [Can89] described how to compute length-$n = 2^m$ FFTs over $\mathbb{F}_{2^m}$ efficiently when m is a power of two, with further improvements in [VzGG96, GM10]. In particular, Gao and Mateer [GM10] introduced additive-FFT algorithms based on Taylor expansion over characteristic-two finite fields, significantly reducing the number of multiplications compared to earlier approaches such as [VzGG96]. Building on this line, Li et al. [LCK+18] proposed the Frobenius Additive FFT (FAFFT), which uses the Frobenius map to reduce the number of explicit evaluation points and thereby accelerates binary polynomial multiplication. Chen et al. [CCK21] first implemented FAFFT in a bit-sliced form on the Cortex-M4 for BIKE, processing 32 coefficients in parallel within 32-bit words via XOR operations. Lee et al. [LJKH25] subsequently applied the same approach to HQC. More recently, Chen et al. [CCPY26] proposed combining Kronecker segmentation (KS)-based additive FFT with the Chinese Remainder Theorem (CRT), referred to as KS+CRT, to handle HQC parameter sets whose

polynomial degrees are slightly larger than a power of two. Their construction uses the vanishing polynomial s_i as the CRT modulus and performs additive FFT on $\mathbb{F}_2[x]/s_i$ via Kronecker segmentation. The paper also considers FAFFT as a separate FFT-based multiplication backend. In their work, FAFFT is used on x86 AVX2/GFNI and ARM Neon with the cryptographic extension (`PMULL.P64`), whereas KS+CRT is adopted on ARM Neon platforms that lack the cryptographic extension and provide only `PMULL.P8`. Concurrently and independently, Chen et al. [CCC+26] also study FAFFT combined with CRT on the Cortex-M4. We provide a detailed comparison in Section 5.

The second approach is to use a radix-16 representation. This approach was proposed in [CC21], where the authors presented an optimized implementation of Classic McEliece on Cortex-M4. In particular, they introduced a radix-16 representation to accelerate the field multiplications required in the Berlekamp-Massey algorithm. By exploiting the structure of radix-16 polynomials and using the `umlal` instruction to emulate carry-less multiplications, the implementation achieves low-latency multiplication and inversion in $\mathbb{F}_{2^{12}}$ and $\mathbb{F}_{2^{13}}$, resulting in substantial speedups on 32-bit microcontrollers. Chen et al. [CGKT22] later applied this approach to BIKE, combining it with Karatsuba and Toom-Cook for large-degree binary polynomial multiplication and selecting the most efficient combination together with the `gf2x` library [BGTZ23]. Our work also builds on this line by incorporating radix-16-based multiplication for the residual non-FFT part, which complements the Hybrid FAFFT-CRT design on Cortex-M4.

1.2 Our Contributions

In this work, we present methods to optimize HQC polynomial multiplication on the ARM Cortex-M4. The main contributions of this paper are summarized as follows:

- We propose the Hybrid FAFFT-CRT method, which uses the Chinese Remainder Theorem to enable FAFFT to operate at size 2^{d+1} instead of 2^{d+2} for polynomial degrees $n = 2^d + r$ with small r. Chen et al. [CCPY26] combined CRT only with KS-based additive FFT, as the FAFFT evaluation domain lies outside the vanishing set of their CRT modulus $s_i(x)$, which prevents the reduction from preserving evaluation values. Our FAFFT-CRT method resolves this by using the vanishing polynomial of the FAFFT evaluation domain as the CRT modulus, combining FAFFT with CRT for the first time. Compared to the state-of-the-art [LJKH25], this reduces polynomial multiplication cycles by 36.0% and 29.3% for HQC-1 and HQC-3, leading to overall KEM improvements of 23–26% and 19–20%, respectively.

- We optimize the FAFFT butterfly for the Cortex-M4 by shortening the XOR sequences for fixed-coefficient multiplications, scheduling operations to fully utilize all 14 GP registers, and applying `SWAPMOVE` in the final five layers. The optimized butterfly reduces the cycle count by 3.2-3.5% across all FAFFT lengths. For HQC-5, where hybrid methods are not applicable, this yields a 1.4-1.6% improvement across all KEM operations.

- As complementary methods, we present the Hybrid Karatsuba-FAFFT method based on a 2-way Karatsuba split, and apply radix-16 multiplication to HQC for the first time. To optimize the Karatsuba and Toom-Cook combinations used inside radix-16, we develop an operation-count cost model that explores combinations of 2-, 3-, 5-Karatsuba and Toom-Cook 3-way. Since the radix-16 method omits the FAFFT code, it reduces the flash usage by 54–71% compared to the baseline FAFFT [LJKH25], making it the preferred choice for flash-constrained deployments. For HQC-1, it also improves polynomial multiplication cycles by 24.4% over the baseline.

The source code and implementation details of the proposed method are available at: https://github.com/jihoonjang98/Efficient-Polynomial-Multiplication-for-HQC-on-ARM-Cortex-M4

1.3 Organization

Section 2 provides background on HQC and reviews the additive FFT framework and FAFFT. Section 3 presents the Hybrid FAFFT-CRT method and the optimized FAFFT butterfly for HQC on the Cortex-M4. Section 4 covers radix-16 multiplication, its cost-model-based optimization, and the Hybrid Karatsuba-FAFFT method. Section 5 reports experimental results, and Section 6 concludes.

2 Preliminaries

2.1 HQC

HQC [AAB+22] is a code-based KEM selected by NIST for post-quantum cryptography standardization in 2025. Its IND-CPA security reduces to the decoding problem for random quasi-cyclic codes over $\mathbb{F}_2$, and IND-CCA2 security is obtained from the underlying PKE via the Fujisaki-Okamoto transform. The quasi-cyclic structure causes all algebraic operations of HQC to take place in the ring $R = \mathbb{F}_2[x]/(x^n - 1)$, where the polynomial degree n depends on the security level. The three parameter sets HQC-1, HQC-3, and HQC-5 target NIST security levels 1, 3, and 5, respectively. Their parameters are listed in Table 1.

Table 1: HQC parameter sets [AAB+22].

Parameter set	Security level	n	ℓ	w	$w_r = w_e$
HQC-1	NIST-1	17669	17664	66	75
HQC-3	NIST-3	35851	35840	100	114
HQC-5	NIST-5	57637	57600	131	149

The HQC public-key encryption scheme HQC-PKE is summarized in Algorithm 1. In the algorithm, R_w denotes the subset of R consisting of polynomials of Hamming weight w, and similarly for R_{w_r} and R_{w_e}. The hash function $I(\cdot)$ derives two sub-seeds from the input seed. The code $\mathcal{C}$ is a concatenated Reed-Muller and Reed-Solomon code, with $\mathcal{C}$.Encode and $\mathcal{C}$.Decode as its encoder and bounded-distance decoder, and $\mathsf{Truncate}(\cdot, \ell)$ returns the first ℓ coefficients of its input, where ℓ is the codeword length of $\mathcal{C}$ fixed by the parameter set. The polynomial multiplications in R are $h \cdot y$ in HQC-PKE.Keygen, both $h \cdot r_2$ and $s \cdot r_2$ in HQC-PKE.Encrypt, and $u \cdot y$ in HQC-PKE.Decrypt.

The HQC KEM, denoted HQC-KEM, is built from HQC-PKE through a Fujisaki-Okamoto-style transform with explicit rejection. Decapsulation runs HQC-PKE.Decrypt to recover the plaintext, re-derives the encryption randomness, runs HQC-PKE.Encrypt to re-compute the ciphertext, and outputs the shared key only if the recomputed ciphertext matches the received one. Counting the calls to the PKE procedures, HQC-KEM key generation performs one polynomial multiplication, encapsulation performs two, and decapsulation performs three. As shown in Figure 1, these polynomial multiplications dominate the cycle cost on the Cortex-M4 across all three parameter sets, which is the central motivation for the optimizations in this paper.

2.2 ARM Cortex-M4

The ARM Cortex-M4 is a 32-bit RISC processor based on the ARMv7-M architecture, widely used in embedded systems and adopted as the baseline platform for evaluating

Algorithm 1 HQC-PKE [AAB+22].

HQC-PKE.Keygen($seed_{\mathsf{PKE}}$)
  $(seed_{\mathsf{PKE.dk}}, seed_{\mathsf{PKE.ek}}) \leftarrow I(seed_{\mathsf{PKE}})$
  $(y, x) \xleftarrow{\$} (R_w, R_w)$ from $seed_{\mathsf{PKE.dk}}$
  $h \xleftarrow{\$} R$ from $seed_{\mathsf{PKE.ek}}$
  $s \leftarrow x + h \cdot y$
  return $ek_{\mathsf{PKE}} \leftarrow (seed_{\mathsf{PKE.ek}}, s)$, $dk_{\mathsf{PKE}} \leftarrow seed_{\mathsf{PKE.dk}}$

HQC-PKE.Encrypt($ek_{\mathsf{PKE}}, m, \theta$)
  Parse $ek_{\mathsf{PKE}} = (seed_{\mathsf{PKE.ek}}, s)$
  $h \xleftarrow{\$} R$ from $seed_{\mathsf{PKE.ek}}$
  $(r_2, e, r_1) \xleftarrow{\$} (R_{w_r}, R_{w_e}, R_{w_r})$ from θ
  $u \leftarrow r_1 + h \cdot r_2$
  $v \leftarrow \mathcal{C}.\mathsf{Encode}(m) + \mathsf{Truncate}(s \cdot r_2 + e, \ell)$
  return $c_{\mathsf{PKE}} \leftarrow (u, v)$

HQC-PKE.Decrypt($dk_{\mathsf{PKE}}, c_{\mathsf{PKE}}$)
  Parse $dk_{\mathsf{PKE}} = seed_{\mathsf{PKE.dk}}$
  $y \xleftarrow{\$} R_w$ from $seed_{\mathsf{PKE.dk}}$
  Parse $c_{\mathsf{PKE}} = (u, v)$
  return $m \leftarrow \mathcal{C}.\mathsf{Decode}(v - \mathsf{Truncate}(u \cdot y, \ell))$

PQC implementations [AAC+22, KPR+]. Its programmer-visible register file provides 14 general-purpose 32-bit integer registers, and 32 single-precision floating-point registers are available through the optional FPv4-SP extension. The core has no instruction cache and no out-of-order execution, so each instruction executes in a fixed number of cycles.

Several Cortex-M4 instructions are central to our implementations. The 32-bit unsigned multiplication `UMULL` and the multiply-accumulate `UMLAL` both produce 64-bit results in one cycle, which we use to emulate carry-less multiplication via the radix-16 representation. The single-cycle `vmov` instruction transfers data between general-purpose and floating-point registers, allowing the floating-point register file to act as low-latency scratch storage. The bit-swap subroutine `SWAPMOVE` [MPC00] reorders bits within a register using a four-instruction XOR-and-shift sequence. The single-cycle `ADCS` instruction (add with carry, set flags) propagates carry across 32-bit words, which we chain to perform efficient multi-word shifts. Notably, the Cortex-M4 does not provide a dedicated carry-less multiplication instruction such as `PCLMULQDQ` on AVX2 or `PMULL.P64` on ARMv8.

2.3 Frobenius Additive FFT

2.3.1 Additive FFT using novel polynomial basis

The Cantor basis [Can89] is an ordered set $\{v_0, v_1, \ldots, v_{m-1}\} \subset \mathbb{F}_{2^m}$ (m a power of two) satisfying $v_0 = 1$ and $v_{i+1}^2 + v_{i+1} = v_i$. The subspaces $W_k = \mathrm{span}_{\mathbb{F}_2}\{v_0, \ldots, v_{k-1}\}$ (with $W_0 = \{0\}$) and the vanishing polynomials $s_i(x)$, defined by $s_0(x) = x$ and $s_{i+1}(x) = s_i(x)^2 + s_i(x)$, satisfy the following properties:

- $\deg(s_i(x)) = |W_i| = 2^i$, $s_i(x + y) = s_i(x) + s_i(y)$, $s_{i+j}(x) = s_i(s_j(x))$.
- $s_k(x) = \prod_{a \in W_k}(x - a)$ (*vanishing polynomial* over W_k).
- $s_k(v_i) = v_{i-k}$ if $i > k$, $s_k(v_k) = 1$, $s_k(v_i) = 0$ if $i < k$.

Lin et al. [LCH14] introduced a novel polynomial basis $\{X_k(x)\}$ for efficient AFFT computations. Each basis polynomial $X_k(x)$ is defined as a product of vanishing polynomials:

$$X_k(x) = \prod_i (s_i(x))^{b_i},$$

where the index k is represented in binary as $k = \sum b_i 2^i$, with $b_i \in \{0, 1\}$. Since $b_i = 1$ if and only if the i-th bit of k is 1, the degree of $X_k(x)$ is computed as $\deg(X_k(x)) = \sum b_i \cdot 2^i = k$. Thus, the basis polynomial $X_k(x)$ has degree exactly k, matching the binary expansion of its index. Based on this property, any polynomial $f(x)$ over $\mathbb{F}_{2^m}$ of degree less than n can be uniquely represented in the form:

$$f(x) = g_0 + g_1 X_1(x) + \cdots + g_{n-1} X_{n-1}(x),$$

where each $g_i \in \mathbb{F}_{2^m}$.

Novel polynomial basis $\{X_k(x)\}$ satisfies the following properties:

- $X_{a\cdot 2^i}(x) \cdot X_b(x) = X_{a\cdot 2^i + b}(x)$ if $b < 2^i$.
- $X_i(s_j(x)) = X_{i\cdot 2^j}(x)$

The AFFT algorithm consists of two stages. First, BasisCvt [LANH16] converts $f(x)$ from the monomial basis to the novel polynomial basis $g(X)$ via recursive decomposition using vanishing polynomials. Second, Btfy recursively splits $g(X)$ into halves using $s_i(x)$ and evaluates at 2^l points of the form $\alpha + \omega$ ($\omega \in W_l$). At each butterfly step, the key operations are:

$$h_0(X) \leftarrow p_0(X) + s_i(\alpha) \cdot p_1(X),$$
$$h_1(X) \leftarrow h_0(X) + p_1(X),$$

where $s_i(v_i) = 1$ is used. Since all operations are over $\mathbb{F}_{2^m}$, the algorithm consists only of bitwise XORs and carry-less multiplications. The detailed procedures for BasisCvt, Btfy, and the complete AFFT algorithm are described in [LCH14, LANH16].

2.3.2 Frobenius Additive FFT

The Frobenius Additive FFT (FAFFT) [LCK+18, CCK21] exploits the Frobenius map $\phi_2(a) = a^2$ on $\mathbb{F}_{2^m}$ to reduce the number of evaluation points. For $f(x) \in \mathbb{F}_2[x]$ and $a \in \mathbb{F}_{2^m}$, the identity $f(\phi_2(a)) = \phi_2(f(a))$ implies that evaluations along Frobenius orbits are redundant.

Let $k = l - \log_2 m < \frac{m}{2}$ and define $\Sigma := v_{k+m/2} + W_k$. Then $\mathrm{Ord}_{\phi_2}(\Sigma) = m$, and the set $\Omega := \Sigma \cup \phi_2(\Sigma) \cup \cdots \cup \phi_2^{m-1}(\Sigma)$ forms a disjoint union of size $|\Omega| = 2^l$, making Σ a Frobenius partition of Ω with $|\Sigma| = 2^l/m$. Evaluating f at the $2^l/m$ points of Σ thus recovers the full 2^l evaluations via ϕ_2, allowing us to truncate the first $\ell_m = \log_2 m$ butterfly layers. We define the modulus polynomial $\mathcal{P}_l(x) = \prod_{\beta\in\Omega}(x - \beta)$ as the vanishing polynomial over Ω, with $\deg(\mathcal{P}_l) = 2^l$. Point-wise multiplication of evaluations over Ω corresponds to polynomial multiplication modulo $\mathcal{P}_l(x)$, so FAFFT effectively computes in the ring $\mathbb{F}_2[x]/\mathcal{P}_l(x)$.

FAFFT multiplies two polynomials $f, g \in \mathbb{F}_2[x]$ in three stages (Algorithm 2): (i) BasisCvt converts to the novel polynomial basis, (ii) Encode truncates the first $\ell_m = \log_2 m$ butterfly layers using the Frobenius partition, reducing the evaluation points from n to n/m, and (iii) Btfy evaluates at the remaining n/m points. The Encode step executes only one recursive call per butterfly node, with fixed multipliers $v_{m/2-\ell_m+1}, \ldots, v_{m/2}$, and can be represented as a matrix-vector multiplication without increasing memory usage.

Algorithm 2 Polynomial multiplication using FAFFT

Require: $A(x), B(x) \in \mathbb{F}_{2^m}[x]/\mathcal{P}_l(x)$.
Ensure: $C(x) = A(x) \cdot B(x) \in \mathbb{F}_{2^m}[x]/\mathcal{P}_l(x)$.
1: $A(X) \leftarrow \textsc{BasisCvt}(A(x))$.
2: $f(X) \leftarrow \textsc{Encode}(A(X))$.
3: $\left(\hat{a}_0, \cdots, \hat{a}_{2^k-1}\right) \leftarrow \textsc{Btfy}(f(X), v_{k+m/2})$.
4: $B(X) \leftarrow \textsc{BasisCvt}(B(x))$.
5: $g(X) \leftarrow \textsc{Encode}(B(X))$.
6: $\left(\hat{b}_0, \cdots, \hat{b}_{2^k-1}\right) \leftarrow \textsc{Btfy}(g(X), v_{k+m/2})$.
7: $\hat{C} \leftarrow \left(\hat{c}_0 = \hat{a}_0 \cdot \hat{b}_0, \cdots, \hat{c}_{2^k-1} = \hat{a}_{2^k-1} \cdot \hat{b}_{2^k-1}\right)$.
8: $h(X) \leftarrow \textsc{invBtfy}(\hat{C}, v_{k+m/2})$.
9: $C(X) \leftarrow \textsc{Decode}(h(X))$.
10: $C(x) \leftarrow \textsc{invBasisCvt}(C(X))$.
11: **return** $C(x)$

3 Hybrid FAFFT-CRT multiplication

On the Cortex-M4, which lacks carry-less multiplication instructions, FAFFT is well suited since its butterfly operations reduce to XOR sequences in a bit-sliced representation, enabling parallel processing of 32 coefficients per word [CCK21]. We therefore base our optimizations on the FAFFT backend.

In this section, we propose the Hybrid FAFFT-CRT method that applies FAFFT more efficiently for the HQC polynomial degrees. FAFFT can be applied only to polynomials of degree $2^l - 1$. To ensure that reduction does not affect the result, one chooses the smallest l such that $2n - 2 < 2^l$. For the degree of polynomials on HQC $n = 2^d + r$, this condition leads to $l = d + 2$. Consistent with this rule, the state-of-the-art implementation [LJKH25] selects $2^l = 65536$, 131072, and 131072 for HQC-1, -3, and -5, respectively, as shown in Table 2.

Table 2: FAFFT parameters for HQC used in [LJKH25].

Parameter set	n	2^l	m	k	Σ
HQC-1	17669	65536	32	11	$v_{27} + W_{11}$
HQC-3	35851	131072	32	12	$v_{28} + W_{12}$
HQC-5	57637	131072	32	12	$v_{28} + W_{12}$

Consequently, FAFFT is performed on polynomials whose degree is at least twice and less than four times the original degree. In particular, HQC-1, -3 have $n = 2^{14} + 1285$ and $n = 2^{15} + 3083$, which are slightly larger than powers of two. This mismatch introduces substantial overhead, and as a result HQC-3 uses the same 2^l as HQC-5. The observation in [LJKH25] that the polynomial-multiplication performance of HQC-3 and HQC-5 is very similar supports this explanation. To address this, we propose the Hybrid FAFFT-CRT method, which aligns the large subproduct with a power of two for FAFFT evaluation and handles the small residual part separately. This allows FAFFT to be applied more efficiently to the polynomial degrees used in HQC. A second variant based on Karatsuba splitting, the Hybrid Karatsuba-FAFFT method, is presented in Section 4.2. Lastly, to further optimize the FAFFT implementation, we improve the butterfly operation by shortening the XOR sequences in fixed-coefficient multiplications and carefully scheduling operations to fully saturate the 14 general-purpose registers available on the ARM Cortex-M4.

Table 3: Comparison between KS+CRT and the proposed FAFFT-CRT.

	KS+CRT [CCPY26]	Hybrid FAFFT-CRT
FFT variant	Additive FFT	Frobenius Additive FFT
FFT modulus	$s_i(x)$	$\mathcal{P}_{d+1}(x)$
Eval. domain	W_i	$\Sigma = v_{k+m/2} + W_k$ (actual) $\Omega = \bigcup_{j=0}^{m-1} \phi_2^j(\Sigma)$ (full)
Modulus gcd	$\gcd(s_i, x^c) = x \neq 1$	$\gcd(\mathcal{P}_{d+1}, x^{r_c}) = 1$
reverse CRT	$\begin{cases} k = I_s(c_0 + c_1)/x \bmod x^{c-1} \\ f = c_1 + k s_i \end{cases}$	$\begin{cases} t = (c_1 + c_2)\mathcal{P}_{d+1}^{-1} \bmod x^{r_c} \\ f = c_1 + t\mathcal{P}_{d+1} \end{cases}$

3.1 Ring decomposition with the Chinese Remainder Theorem

In the context of HQC, polynomial multiplication is performed in the ring $\mathbb{F}_2[x]/(x^n - 1)$. We compute the product $C(x) = A(x) \cdot B(x) \in \mathbb{F}_2[x]$ and reduce modulo $x^n - 1$ as a post-processing step. As discussed in Section 2.3.2, for the polynomial multiplication of degree $n = 2^d + r$ with $r > 0$, the previous approach using FAFFT operates over the ring $\mathbb{F}_{2^m}[x]/\mathcal{P}_{d+2}(x)$. For HQC-1 ($n = 17,669$), the previous approach typically employs $\mathcal{P}_{16}(x)$ of degree $2^{16} = 65,536$. This results in significant padding overhead since the required product length is only about $35,338$.

To address this inefficiency, our Hybrid FAFFT-CRT method uses the Chinese Remainder Theorem (CRT) to compute the product over a modulus $\mathcal{P}_{d+1}(x) \cdot x^{r_c}$. Here, $\mathcal{P}_{d+1}(x)$ corresponds to an FAFFT instance, and x^{r_c} covers the residual part. Since $\mathcal{P}_{d+1}(x)$ is not divisible by x, the moduli $\mathcal{P}_{d+1}(x)$ and x^{r_c} are coprime, and the CRT yields the ring isomorphism

$$\mathbb{F}_2[x]/(\mathcal{P}_{d+1}(x) \cdot x^{r_c}) \;\cong\; \mathbb{F}_2[x]/\mathcal{P}_{d+1}(x) \;\times\; \mathbb{F}_2[x]/x^{r_c}.$$

The first factor is handled by FAFFT, where the $\mathbb{F}_2$ coefficients are reused as elements of $\mathbb{F}_{2^m}$ via the subfield embedding without conversion, and the second factor by radix-16 multiplication.

Chen et al. [CCPY26] proposed a CRT-based decomposition (KS+CRT) using the vanishing polynomial $s_i(x)$ as the CRT modulus combined with Kronecker segmentation. However, their construction was not applied to FAFFT. The reason is that $s_i(x)$ vanishes on W_i, so $\text{addFFT}(f, W_i) = \text{addFFT}(f \bmod s_i, W_i)$, but FAFFT evaluates on points in $\Sigma = v_{k+m/2} + W_k$ that lie outside W_i. Reducing modulo $s_i(x)$ therefore alters the evaluation values on the FAFFT domain, making the combination of $s_i(x)$-based CRT with FAFFT infeasible. Our method overcomes this by using $\mathcal{P}_{d+1}(x)$ as the CRT modulus. Since $\mathcal{P}_{d+1}(x)$ is the vanishing polynomial over the FAFFT evaluation domain Ω, the reduction $f \bmod \mathcal{P}_{d+1}(x)$ preserves the evaluation values on Ω, enabling the direct combination of CRT with FAFFT. Moreover, since $\gcd(\mathcal{P}_{d+1}(x), x^{r_c}) = 1$, the standard CRT applies without a generalized version or consistency conditions. This is particularly advantageous on the Cortex-M4, where FAFFT has already been extensively optimized [CCK21, LJKH25] and is more efficient than KS-based multiplication. Although $\mathcal{P}_{d+1}(x)$ has a higher Hamming weight than $s_i(x)$, which increases the cost of CRT reconstruction, we demonstrate that this overhead can be effectively mitigated by optimizing the product $t(x) \cdot \mathcal{P}_{d+1}(x)$ as a sparse-dense multiplication with index grouping and batch processing tailored for the Cortex-M4 in Section 3.2.1. Table 3 summarizes the algorithmic differences between KS+CRT and our FAFFT-CRT.

For HQC-1 such that $d = 14$, instead of using $\mathcal{P}_{16}(x)$, we select the smaller modulus $\mathcal{P}_{15}(x)$ of degree $2^{15} = 32,768$. To ensure the total ring size covers the linear product without reduction, we choose $r_c = 2,592$ as a multiple of 32 for alignment. Since the

correctness condition $2n-2 < \deg(\mathcal{P}_{15}(x))+r_c$ is satisfied, the result in $\mathbb{F}_2[x]/(\mathcal{P}_{15}(x)\cdot x^{r_c})$ equals the product in $\mathbb{F}_2[x]$ without wrap-around. The computation is split into two parts:

$$C_1(x) = A(x)B(x) \pmod{\mathcal{P}_{15}(x)}, \qquad C_2(x) = A(x)B(x) \pmod{x^{r_c}}.$$

$C_1(x)$ is computed efficiently using the FAFFT of length 2^{15}. Since $\mathbb{F}_2$ is a subfield of $\mathbb{F}_{2^m}$ (with $m = 32$), the binary coefficients of the inputs $A(x)$ and $B(x)$ are reused as elements of $\mathbb{F}_{2^m}$ without any conversion, allowing FAFFT to operate over $\mathbb{F}_{2^m}[x]/\mathcal{P}_{15}(x)$. $C_2(x)$ is computed using radix-16 polynomial multiplication, which is described in Section 4. The final result is reconstructed using the explicit CRT formula. First, we compute the correction term $t(x)$ modulo x^{r_c}:

$$t(x) = (C_1(x) \pmod{x^{r_c}} + C_2(x)) \cdot \mathcal{P}_{15}^{-1}(x) \pmod{x^{r_c}}.$$

Here, $\mathcal{P}_{15}^{-1}(x)$ is the modular inverse of $\mathcal{P}_{15}(x)$ modulo x^{r_c}. Finally, the product $C(x)$ is recovered as:

$$C(x) = C_1(x) + t(x) \cdot \mathcal{P}_{15}(x) \in \mathbb{F}_2[x].$$

The final multiplication $t(x) \cdot \mathcal{P}_{15}(x)$ is performed efficiently by taking advantage of the sparsity of $\mathcal{P}_{15}(x)$. Although $\mathcal{P}_{15}(x)$ has a degree of 32,768, it contains only 501 non-zero terms in our parameter set, making it significantly sparse compared to a random dense polynomial. In standard HQC implementations, optimized sparse-dense multiplication is generally avoided because the sparse polynomial usually represents a secret key or error vector. Thus, the operation requires constant-time execution to prevent timing attacks. However, $\mathcal{P}_{15}(x)$ is a fixed and public polynomial determined by the FAFFT parameters. Therefore, we can employ a fast sparse-dense multiplication algorithm without the risk of leaking sensitive information.

For HQC-5 ($n = 57,637$), the required CRT residual degree must satisfy $r_c \geq 49,736$. Consequently, applying the hybrid FAFFT-CRT method to HQC-5 necessitates at least an FAFFT of size $65,536$ and a polynomial multiplication of degree r_c. To verify the feasibility of this approach, we estimated a conservative lower bound on the required clock cycles by summing the cycle counts of the FAFFT of size $65,536$ and a polynomial multiplication of degree $35,851$ (taken from our HQC-3 benchmark). Since the degree $35,851$ is strictly smaller than the required r_c and we explicitly exclude additional overheads, such as $t(x) \cdot \mathcal{P}_{16}(x)$, this estimate represents a strictly lower bound. Even this lower bound amounts to $7,328,549$ clock cycles, which already exceeds the $6,290,041$ cycles achieved by the state-of-the-art implementation [LJKH25]. Therefore, the hybrid FAFFT-CRT method offers no performance benefit for HQC-5.

Table 4: Parameters for Hybrid FAFFT-CRT method in HQC-1 ,-3. $\omega(\mathcal{P}_i)$ denotes the Hamming weight of the $\mathcal{P}_i(x)$.

Param.	n	FAFFT				CRT	
		$2^{l'}$	m	k	Σ	r_c	$\omega(\mathcal{P}_i)$
HQC-1	17669	32768	32	10	$v_{26}+W_{10}$	2592	501
HQC-3	35851	65536	32	11	$v_{27}+W_{11}$	6176	473

3.2 Efficient sparse-dense polynomial multiplication

The final step of the CRT reconstruction requires computing the product $t(x) \cdot \mathcal{P}_{d+1}(x)$. Since $\mathcal{P}_{d+1}(x)$ is a sparse polynomial and $t(x)$ is a dense polynomial, we employ an optimized shift-and-XOR strategy tailored to the 32-bit word architecture of the ARM Cortex-M4. As previously noted, since $\mathcal{P}_{d+1}(x)$ is a fixed and public polynomial, utilizing its sparsity does not incur side-channel leakage. We propose two variants of this sparse-dense multiplication that trade cycles against memory.

3.2.1 Input-stationary sparse-dense multiplication

In the input-stationary variant, the bit-shifted dense polynomial $t_s(x) = t(x) \cdot x^s$ is produced inside the inner loop by a 1-bit left shift chain applied to the working registers $\{v_1, \ldots, v_5\}$ linked with the carry word v_0 at each iteration over s. The implementation builds on two optimizations, index grouping and batch processing.

Index grouping A naive implementation would iterate through every non-zero term x^ι of $\mathcal{P}_{d+1}(x)$ and accumulate $t(x) \cdot x^\iota$. This approach requires shifting the dense polynomial $t(x)$ by an arbitrary bit index ι. On the ARM Cortex-M4, performing a bitwise shift on the polynomial is not a single instruction. It requires a sequence of a bitwise shift and a logical OR-shift (utilizing the barrel shifter) for every word of the dense polynomial. To reduce these costs, we minimize bitwise shift operations by grouping the indices of non-zero terms based on their offsets within a 32-bit word. Let $I_{\mathcal{P}_{d+1}}$ denote the set of indices corresponding to the exponents of the non-zero terms in $\mathcal{P}_{d+1}(x)$. We decompose each index $\iota \in I_{\mathcal{P}_{d+1}}$ into a quotient q_ι and a bitwise shift amount s such that $\iota = 32 \cdot q_\iota + s$, where $0 \leq s < 32$. Based on this decomposition, the multiplication can be reformulated as:

$$t(x) \cdot \mathcal{P}_{d+1}(x) = \sum_{\iota \in I_{\mathcal{P}_{d+1}}} t(x) \cdot x^\iota = \sum_{s=0}^{31} \left((t(x) \cdot x^s) \cdot \sum_{\substack{\iota \in I_{\mathcal{P}_{d+1}} \\ \iota \equiv s \pmod{32}}} (x^{32})^{q_\iota} \right).$$

In our implementation, we group the indices of $\mathcal{P}_{d+1}(x)$ by their bitwise shift amount s. The process is as follows:

(i) We compute the bit-shifted dense polynomial $t_s(x) = t(x) \cdot x^s$ within registers, reusing it for all terms with the same shift amount s.

(ii) For every term in the group with shift amount s, we accumulate $t_s(x)$ shifted by q_ι words. Since word-level shifts correspond to simple memory pointer adjustments, they incur negligible computational cost.

This optimization reduces the number of bitwise shift operations required, significantly accelerating the sparse-dense multiplication.

Batch processing To further optimize performance, we process consecutive words of the dense polynomial in batches per iteration, rather than iterating word-by-word. This strategy provides two advantages:

- **Reduction of load and shift overhead:** Computing the bit-shifted result for a batch of α target words requires loading (Line 8) and shifting (Line 13) $\alpha + 1$ words: the α target words $(v_1, \ldots, v_\alpha)$ and the preceding word (v_0) which supplies the carry bits. Consequently, the number of load and shift operations required per generated output word is proportional to $(\alpha + 1)/\alpha$, which decreases as the batch size α increases.
- **Utilization of `LDM` and `STM`:** Handling consecutive words enables the use of `LDM` (Load Multiple) and `STM` (Store Multiple) instructions. These instructions minimize the instruction fetch overhead and maximize memory bandwidth utilization compared to executing a sequence of single `LDR`/`STR` instructions.

In our specific implementation on the ARM Cortex-M4, we adopted a 5-word batch configuration ($\alpha = 5$). The Cortex-M4 provides 14 usable GP registers (R0-R12 and LR). Our innermost loop (lines 15-18) requires four registers to store the result array

Algorithm 3 Input-stationary Sparse-Dense Polynomial Multiplication

Require: R (result array), t (dense polynomial array),
Idx (sorted word offsets), Cnt (term counts per shift)
Ensure: $R \leftarrow R + t(x) \cdot \mathcal{P}_{d+1}(x)$

for $i = 0$ **to** $\text{len}(t) - 5$ **step** 5 **do**
  / Load 5 words ($v_1 \dots v_5$) and 1 carry word (v_0) */*
  if $i = 0$ **then**
    $v_0 \leftarrow 0$
  else
    $v_0 \leftarrow t[i-1]$
  end if
  $(v_5, v_4, v_3, v_2, v_1) \leftarrow (t[i+4], t[i+3], t[i+2], t[i+1], t[i])$
  $ptr \leftarrow Idx$
  / Unified Shift and XOR Processing */*
  for $s = 0$ **to** 31 **do**
    if $s > 0$ **then**
      / Shift chain including carry v_0 */*
      $(v_5, v_4, v_3, v_2, v_1, v_0) \leftarrow (v_5, v_4, v_3, v_2, v_1, v_0) \ll 1$
    end if
    for $j = 0$ **to** $Cnt[s] - 1$ **do**
      $off \leftarrow *ptr{+}{+}$
      $R[i + off, \cdots, i + off + 4] \oplus = (v_1, v_2, v_3, v_4, v_5)$
    end for
  end for
end for
Note: If $\text{len}(t) \not\equiv 0 \pmod 5$*, the remaining words must be processed.*

base address $\&R[i]$, the pointer ptr, the offset off, and the loop counter j. To maximize throughput of this loop using `LDM/STM`, we must allocate 5 registers to hold the pre-loaded dense polynomial batch $(v_1, \dots, v_5)$ and 5 additional registers as a result polynomial batch $R[i + off, \cdots, i + off + 4]$ for the accumulation (load, XOR, store) process. This requires a total of 14 registers, which efficiently uses all of the available GP registers and prevents unnecessary memory accesses. Note that the carry word v_0 is preserved on the stack as it is not required for the XOR accumulation step.

Algorithm 3 details the procedure. It iterates through the dense polynomial in 5-word batches. For each batch, the inner loop iterates through shift amounts s from 0 to 31, updating the working registers via a 1-bit left shift linked with v_0 to align with the current bit index, and then XORing the bit-shifted values into the positions of the result array R specified by $Cnt[s]$ and the corresponding segment of Idx.

3.2.2 Output-stationary sparse-dense multiplication

The output-stationary variant differs from the input-stationary scheme of Section 3.2.1 in two ways. First, all 31 bit-shifted copies of $t(x)$ are precomputed at the start into a table $Tsh[0..31]$ with $Tsh[0] = t$ using a chain of `ADCS` instructions that propagates carry across the 32-bit words, eliminating the per-shift 1-bit shift update of $\{v_1, \dots, v_5\}$ inside the inner loop. Second, the index grouping is reorganized per result block instead of per shift amount. The array $Voff$ stores offsets into Tsh for each contributing term, grouped by result block, and Cnt records the number of contributing terms per block. Batch processing is unchanged. The main loop iterates over 5-word result blocks and, for each block, XORs all contributing 5-word slices of Tsh into the registers holding the current block. Each row of Tsh is zero-padded so that 5-word reads at block boundaries handle

Algorithm 4 Output-stationary Sparse-Dense Polynomial Multiplication

Require: R (result array), t (dense polynomial array),
Voff (per-block offsets into *Tsh*), *Cnt* (term counts per result block)
Ensure: $R \leftarrow R + t(x) \cdot \mathcal{P}_{d+1}(x)$

```
/* Precompute 31 bit-shifted copies of t as 32 rows of Tsh */
Tsh[0][0, ··· , len(t)−1] ← t[0, ··· , len(t)−1]
for s = 1 to 31 do
    Tsh[s][0, ··· , len(t)−1] ← Tsh[s−1][0, ··· , len(t)−1] ≪ 1
end for
ptr ← Voff
/* Per-block accumulation */
for i = 0 to len(R) − 5 step 5 do
    /* Load 5 result words (v1 ... v5) */
    (v5, v4, v3, v2, v1) ← (R[i+4], R[i+3], R[i+2], R[i+1], R[i])
    for j = 0 to Cnt[i/5] − 1 do
        off ← *ptr++
        s ← ⌊off/len(t)⌋,    w ← off mod len(t)
        (v1, v2, v3, v4, v5) ⊕= Tsh[s][w, ··· , w + 4]
    end for
    R[i, ··· , i+4] ← (v1, v2, v3, v4, v5)
end for
Note: If len(R) ≢ 0 (mod 5), the remaining words must be processed.
```

partial overlaps via zero XORs. Algorithm 4 details the procedure.

The output-stationary variant is faster than the input-stationary one by 3.80% on HQC-1 and 2.85% on HQC-3 in polynomial multiplication cycles, at the cost of additional flash and RAM for the precomputed shift table *Tsh* and the rearranged offset arrays. Flash and RAM usage for both variants is reported in Table 12.

3.3 Optimizing the butterfly algorithm of FAFFT

During the butterfly stage, the term $s_i(\alpha) \cdot p_1(X)$ is evaluated as follows [LJKH25]. For $m = 32$, write $s_i(\alpha) = w + v$ with $w \in W_k$ and $v \in \{v_{17}, \ldots, v_{k+16}\}$. Then $s_i(\alpha) \cdot p_1(X) = w \cdot p_1(X) + v \cdot p_2(X)$. Since $k < 16$, w lies in a proper subspace $\mathbb{F}_2^{m'} \subsetneq \mathbb{F}_2^{32}$ with $m' \leq 16$, allowing the product $w \cdot p_1(X)$ over $\mathbb{F}_2^{32}$ to be computed as $(32/m')$ products over $\mathbb{F}_2^{m'}$. For the term $v \cdot p_2(X)$, [LJKH25] exploits that $v \in \{v_{17}, \ldots, v_{k+16}\}$ is fixed, converts the fixed part into a matrix-vector multiplication, and then searches XOR sequences for this operation using the method of [XZL+20]. This yields a 23-44% reduction in XORs compared with the previous implementation [CCK21].

We searched XOR sequences that use fewer XORs than [LJKH25] by running the heuristic of [XZL+20] on the same matrix-vector formulation. Moreover, by spilling the input and output registers to the stack and scheduling with all 14 GP registers and 32 FP registers of Cortex-M4, we improve register allocation and reduce `vmov` instructions. Table 5 compares our results with [LJKH25]. Although our sequence for v_{24} uses more XOR operations, it enables better register scheduling and results in a lower overall instruction count. Since performance depends on the instruction count rather than the XOR count, we select the implementation with fewer instructions.

In addition, we optimize the swap operations that occur during the butterfly using the `SWAPMOVE` subroutine proposed in [MPC00]. The `SWAPMOVE` operation applies a mask to select specific bits from register R_2 and swaps them with the bits of register R_1 shifted by n. We denote it as $\texttt{SWAPMOVE}(R_1, R_2, \text{mask}, n)$. On the Cortex-M4, this operation can be

Table 5: XOR and instruction counts for multiplication by v_i.

Multiplier	[LJKH25]		Ours	
	XOR	clock cycle	XOR	clock cycle
v_{17}	161	485	159	462
v_{18}	153	465	153	440
v_{19}	171	515	166	479
v_{20}	173	511	172	485
v_{21}	172	514	166	483
v_{22}	165	501	160	469
v_{23}	163	497	163	476
v_{24}	178	548	180	503
v_{25}	189	555	180	501
v_{26}	190	562	179	516
v_{27}	179	539	173	538
v_{28}	187	559	183	518

implemented using four instructions. Algorithm 5 represents the `SWAPMOVE` subroutine.

Algorithm 5 `SWAPMOVE`

Require: R_1, R_2, mask, n
Ensure: bit swap between the R_1, R_2
1: `EOR` tmp, R_2, R_1, `LSR` n
2: `AND` tmp, tmp, mask
3: `EOR` R_2, R_2, tmp
4: `EOR` R_1, R_1, tmp, `LSL` n

Our butterfly is implemented in a bitsliced manner on a Cortex-M4 with 32-bit registers. In the final five layers of BTFY, the offset n between the data corresponding to $p_0(X)$ and $p_1(X)$ is at most 16. In the bitsliced representation, the two slices reside within the same 32-bit word. For this reason, bit swaps in registers are required. There are $2^l/32$ such swaps for each layer, so a single FAFFT multiplication performs $2^l/32 \cdot 5 \cdot 2$ swaps in total. [LJKH25] implements this swap as Algorithm 6.

Algorithm 6 Bit swap in the butterfly stage [LJKH25]

Require: $d[i+j], d[i+32+j]$
Ensure: $d[i+j], d[i+32+j]$ s.t. the bits selected by M(Mask) are exchanged between the two words with shift offset n.
1: $d_l \leftarrow (d[i+j] \wedge \neg M) \,|\, ((d[i+32+j] \wedge \neg M) \ll n)$;
2: $d_h \leftarrow ((d[i+j] \wedge M) \gg n) \,|\, (d[i+32+j] \wedge M)$;
3: $d[i+j] \leftarrow d_l$;
4: $d[i+32+j] \leftarrow d_h$;

The implementation in Algorithm 6 requires eight instructions. Using `SWAPMOVE`, the same operation can be realized with four instructions as Algorithm 7, reducing four instructions for each swap.

4 Radix-16 multiplication and Hybrid Karatsuba-FAFFT

This section presents two additional polynomial multiplication methods. The first is radix-16 multiplication, a stand-alone polynomial multiplication method on Cortex-M4.

Algorithm 7 Bit swap in the butterfly stage with `SWAPMOVE`

Require: $d[i+j], d[i+32+j]$
Ensure: $d[i+j], d[i+32+j]$ s.t. the bits selected by M (mask) are exchanged between the two words with shift offset n.
1: $tmp \leftarrow d[i+32+j] \oplus (d[i+j] \gg n)$;
2: $tmp \leftarrow tmp \wedge M$;
3: $d[i+32+j] \leftarrow d[i+32+j] \oplus tmp$;
4: $d[i+j] \leftarrow d[i+j] \oplus (tmp \ll n)$;

The second is the Hybrid Karatsuba-FAFFT method, which uses radix-16 multiplication for the residual $r \times r$ product in its 2-way Karatsuba split.

4.1 Radix-16 multiplication

In [CC21], a radix-16 representation for degree-7 polynomials over $\mathbb{F}_2[x]$ was introduced. Given $A_0(x) = a_0 + a_1x + \cdots + a_7x^7$, define its radix-16 representation as the 32-bit integer $A'_0 = a_0 + a_1 2^4 + a_2 2^8 + \cdots + a_7 2^{28}$. Under this encoding, the product of two polynomials $A(x)$ and $B(x)$ can be computed by a single integer multiplication:

$$P_{0,0} = A'_0 \times B'_0 = a_0b_0 + (a_0b_1 + a_1b_0)2^4 + \cdots + a_7b_7 2^{56}$$

Since for each coefficient index k the sum $\sum_{i+j=k} a_ib_j$ contains at most seven terms (each equal to 0 or 1), no partial sum exceeds $8 < 16$, so no carries occur across 4-bit boundaries. Therefore, extracting the bits at positions $0, 4, 8, \ldots, 56$ from the 64-bit product recovers exactly the 15 coefficients of $A_0(x) \times B_0(x)$ in $\mathbb{F}_2[x]$.

Using this method, polynomial multiplication can be reduced to integer multiplication. Unlike high-performance architectures such as AVX2 and NEON, embedded systems like Cortex-M4 do not support carry-less multiplication instructions. Therefore, for Cortex-M4, radix-16 provides an efficient way to implement $\mathbb{F}_2[x]$ multiplication.

4.1.1 32-bit base multiplication in radix-16 representation

Using the method introduced in the previous section, 8-bit multiplications over $\mathbb{F}_2[x]$ can be performed. In this section, we describe the technique for 32-bit $\mathbb{F}_2[x]$ multiplication as proposed in [CGKT22].

For given 32-bit $\mathbb{F}_2[x]$ polynomial $A(x) = a_0 + a_1x + \cdots + a_{31}x^{31}$ and $B(x) = b_0 + b_1x + \cdots + b_{31}x^{31}$, we split each 32-bit polynomial into four 8-bit polynomials:

$$A_i(x) = \sum_{j=0}^{7} a_{8i+j}\, x^j, \qquad B_i(x) = \sum_{j=0}^{7} b_{8i+j}\, x^j.$$

Finally, $P_{i,j} = A_i(x) \times B_j(x)$ are summed at their appropriate offsets to compute $C'(x)$, the radix-16 representation of $C(x) = A(x) \times B(x)$. Each multiplication $A_i(x) \times B_j(x)$ is computed via integer multiplication on its radix-16 representation. We now introduce efficient methods for the radix-16 conversion and the summation of $P_{i,j}$.

Data conversion for the radix-16 representation. Storing each 8-bit polynomial $A_i(x)$ as a 32-bit integer A'_i in a 32-bit register incurs an increase in register usage by a factor of four. To eliminate this overhead, [CGKT22] proposes an efficient method that converts four 8-bit polynomials into their radix-16 representations within a single 32-bit register. We call it *packed radix-16 representation*.

The conversion method is based on the `SWAPMOVE` operation. Let each 8-bit polynomial $A_i(x) = \sum_{j=0}^{7} a_{i,j}x^j$ be stored in a 32-bit register as

$$[\, a_{0,0}, \ldots, a_{0,7}, \quad \ldots, \quad a_{3,0}, \ldots, a_{3,7} \,].$$

Then three `SWAPMOVE` operations with shifts of 12, 6, and 3 bits reorder these bits in place to

$$[\, a_{0,0}, a_{2,0}, a_{1,0}, a_{3,0}, \quad \ldots, \quad a_{0,7}, a_{2,7}, a_{1,7}, a_{3,7} \,].$$

The three `SWAPMOVE` operations use masks `0x0000f0f0`, `0x00cc00cc`, `0x0a0a0a0a` with the corresponding shifts of 12, 6, and 3 bits [CGKT22]. In total, only 12 instructions are required to construct A'_{pack}, the packed radix-16 representation of A.

After conversion, the 32-bit radix-16 representations for each 8-bit polynomial can be extracted by simple mask-and-shift operations. Let mask = `0x1111 1111`. Then each of the four radix-16 representations, A'_0, A'_1, A'_2, and A'_3, is obtained with a single instruction:

$$A'_0 = A'_{pack} \,\&\, \text{mask}, \qquad A'_1 = (A'_{pack} \gg 2) \,\&\, \text{mask},$$
$$A'_2 = (A'_{pack} \gg 1) \,\&\, \text{mask}, \qquad A'_3 = (A'_{pack} \gg 3) \,\&\, \text{mask}.$$

After multiplication, the inverse conversion restores the original format by applying the `SWAPMOVE` sequence in reverse order with shifts of 3, 6, and 12 bits using the same masks. This is possible since `SWAPMOVE` is self-inverse, i.e., applying it twice serves as the identity.

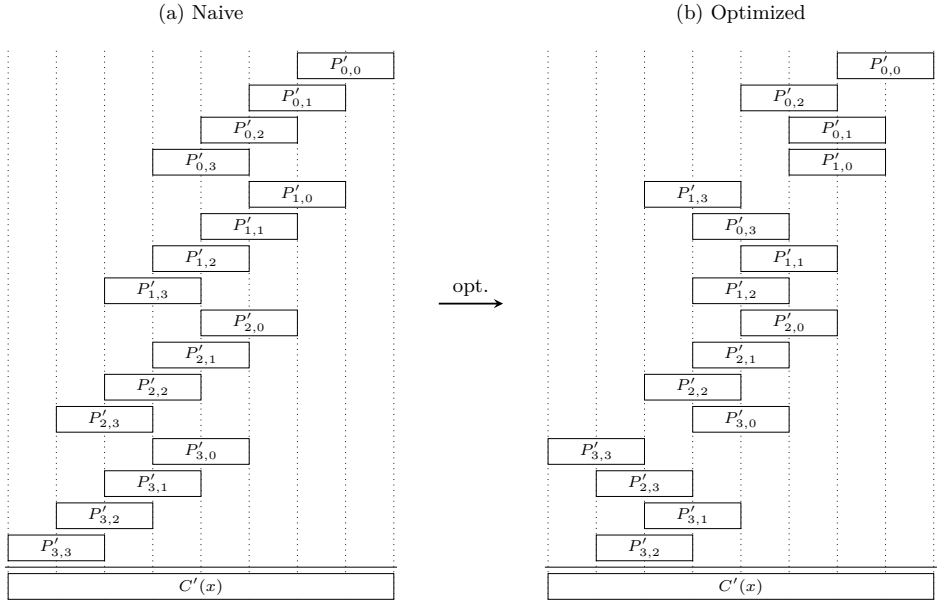

Figure 2: Constructing the 32-bit base multiplication: (a) naive accumulation order, (b) optimized order.

Construction of the 32-bit polynomial multiplication. $C'(x)$ which is the radix-16 representation of $C(x)$ can be obtained by summing the sixteen 64-bit integer products $P_{i,j} = A'_i \times B'_j$ as Figure 2. Each $P_{i,j}$ contains other set bits outside the positions 0, 4, 8, ..., 56. To ensure correctness, $P_{i,j}$ would be masked with $mask$ = 0x1111 1111 1111 1111 via AND operation before accumulation.

However, adding two upper 32-bit halves or an upper 32-bit half to a lower 32-bit half does not propagate carries beyond 4 bits. Hence, the masking step before addition can

be omitted in these cases, saving one instruction. In addition, the UMLAL instruction cannot be applied to uninitialized registers. If both the upper and lower 32-bit halves to be added are uninitialized, the UMULL must be used; otherwise, UMLAL can be used after initializing the register with a MOV instruction. The former requires one instruction, while the latter requires two. Consequently, the total number of instructions depends on the order of addition of the $P_{i,j}$s. Taking these factors and register scheduling into account, we perform accumulation in the optimized order shown in Figure 2(b).

After obtaining $C'(x)$ as in Figure 2(b), it must be converted from the radix-16 representation to the original format. This is done by applying the three `SWAPMOVE`s above in reverse order, requiring a total of 12 instructions. To apply this conversion, $C'(x)$ is converted to a packed radix-16 representation to get C'_{pack}, then converted to $C(x)$. In summary, 32-bit base multiplication in radix-16 representation proceeds as follows:

- **Step 1**: Apply the data conversion to $A(x)$ and $B(x)$ to obtain A'_{pack} and B'_{pack}.
- **Step 2**: Extract A'_i and B'_i $(0 \le i \le 3)$ from A'_{pack} and B'_{pack} .
- **Step 3**: Compute $P_{i,j} = A'_i \times B'_j$ $(0 \le i, j \le 3)$ by integer multiplication.
- **Step 4**: Accumulate $P_{i,j}$ in the order of Figure 2(b) to form $C'(x)$.
- **Step 5**: Construct C'_{pack} by repacking $C'(x)$.
- **Step 6**: Convert C'_{pack} to $C(x)$ by the inverse data conversion.

4.1.2 Karatsuba/Toom-Cook with the radix-16 representation

HQC-1, -3, and -5 contain $\mathbb{F}_2[x]$ polynomial multiplications of degree 17668, 35850, and 57636, respectively. Since Karatsuba and Toom-Cook over $\mathbb{F}_2[x]$ use only word-wise XOR operations except for the base multiplication, they apply directly to the radix-16 representation. We convert the inputs to radix-16 form, recursively apply Karatsuba and Toom-Cook until reaching 32-bit blocks, and then perform Steps 2-5 of Section 4.1.1.

To select the Karatsuba/Toom-Cook combination for each degree, [CGKT22] used the `gf2x` library [BGTZ23], whereas we adopt an operation-count cost model in the style of [JLK+24]. Since the Cortex-M4 has no out-of-order execution or cache, instruction counts are a reliable proxy for clock cycles. The cost model is independent of hardware-specific autotuning, and the resulting candidate set is consistent across platforms. We enumerate the candidate combinations under the model and select the best one by measurement on the target device.

We define an s-word multiplication as the product of two polynomials each stored in s 32-bit words, and set the cost function as

$$\mathrm{cost}(s) = \mathrm{cnt}_{\mathrm{XOR}}(s) + w_{\mathrm{mul}} \cdot \mathrm{cnt}_{\mathrm{mul}}(s),$$

where $\mathrm{cnt}_{\mathrm{XOR}}(s)$ and $\mathrm{cnt}_{\mathrm{mul}}(s)$ count 32-bit XOR operations and base $(s = 1)$ multiplications, and w_{mul} weights base multiplications relative to XORs. At the base, $\mathrm{cnt}_{\mathrm{XOR}}(1) = 0$ and $\mathrm{cnt}_{\mathrm{mul}}(1) = 1$. For a candidate algorithm $\mathcal{A}$ that performs α top-level XORs and recursively invokes β_i sub-multiplications of length s_i, the cost is

$$\mathrm{cost}^{\mathcal{A}}(s) = \alpha + \sum_i \beta_i \,\mathrm{cnt}_{\mathrm{XOR}}(s_i) + w_{\mathrm{mul}} \sum_i \beta_i \,\mathrm{cnt}_{\mathrm{mul}}(s_i).$$

The optimal algorithm at length s is $\mathcal{A}^* = \arg\min_{\mathcal{A} \in L} \mathrm{cost}^{\mathcal{A}}(s)$, which determines $\mathrm{cnt}_{\mathrm{XOR}}(s)$ and $\mathrm{cnt}_{\mathrm{mul}}(s)$ recursively. Table 6 lists the XOR and internal-multiplication counts for each Karatsuba and Toom-Cook variant, from which $\mathrm{cnt}_{\mathrm{XOR}}(s)$ and $\mathrm{cnt}_{\mathrm{mul}}(s)$ are derived.

In an ideal model with only XORs and base multiplications, w_{mul} would equal the cycle ratio between the two operations. In practice, surrounding `MOV`, `LDR`, and `STR` instructions shift this ratio, so we enumerate the optimal combinations across a range of w_{mul} and select the best one by measurement on the target device.

Table 6: The number of internal multiplications and XORs in the Karatsuba and Toom-Cook algorithms.

Algorithm	Input (words)	# of internal multiplications				XOR
		$(k-1)$-word	k-word	$(k+1)$-word	$(k+2)$-word	
2-Karat	$2k$	0	3	0	0	$7k$
	$2k+1$	0	1	2	0	$7k+3$
3-Karat	$3k$	0	6	0	0	$20k$
	$3k+1$	0	3	3	0	$20k+8$
	$3k+2$	0	1	5	0	$20k+14$
5-Karat	$5k$	0	15	0	0	$80k$
Toom-Cook 3-way	$3k$	0	3	0	2	$38k+18$
	$3k+1$	1	0	2	2	$38k+22$
	$3k+2$	0	1	2	2	$38k+32$

By enumerating w_{mul} in $[1, 500]$ with 0.01 increments, we obtained 4, 6, and 5 candidate combinations for HQC-1, HQC-3, and HQC-5, respectively. The combinations achieving the minimum clock cycles on the Cortex-M4 correspond to $w_{\text{mul}} \in [1, 29.08]$, $[2.50, 500]$, and $[1.60, 5.09]$, respectively, and the resulting cycle counts are reported in Section 5.

4.1.3 Comparison with `gf2x` library

We compare our cost-model-based selection with the `gf2x` library, an open-source library for binary polynomial multiplication. Since `gf2x` must handle arbitrary polynomial degrees at runtime, it dispatches the best algorithm via conditional branches and table lookups, whereas our HQC-specific approach selects combinations offline using the cost model of Section 4.1.2 and produces specialized branch-free implementations.

For a fair comparison, both approaches must consider the same candidate algorithms. In a radix-16 representation, shifts within a 32-bit word require more complex operations [CGKT22], so among the algorithms used by `gf2x` the practical candidates are Toom-Cook 3-way (TC3), Toom-Cook 4-way (TC4), and 2-Karatsuba. Although TC4 has the best asymptotic complexity, it is not used at the polynomial degrees of HQC due to poor practical performance [CGKT22], so we restrict both methods to TC3 and 2-Karatsuba and implement the resulting combinations under identical conditions. For HQC-1 and HQC-3, both approaches select the same algorithmic combinations, so the cycle counts in Table 7 coincide exactly. For HQC-5, our combination requires 21.7% fewer clock cycles than the `gf2x` combination. Our cost model additionally considers 3-Karatsuba and 5-Karatsuba, and the combinations we ultimately select with this broader candidate set yield further improvements.

Table 7: Clock cycle comparison on Cortex-M4 for `gf2x` library [BGTZ23] and our method, both restricted to Toom-Cook 3-way and 2-Karatsuba.

Algorithm	`gf2x` library	Our method
HQC-1	2,636,770	2,636,770
HQC-3	7,576,158	7,576,158
HQC-5	21,985,410	17,204,103

4.2 The hybrid Karatsuba-FAFFT method

For HQC-1 and HQC-3, we write the degree parameter as $n = 2^d + r$ with $0 < r < 2^d$ and split the inputs at 2^d:

$$A(x) = A_0(x) + x^{2^d} A_1(x), \quad B(x) = B_0(x) + x^{2^d} B_1(x),$$

where $\deg A_0, \deg B_0 < 2^d$ and $\deg A_1, \deg B_1 < r$. Then we compute

$$C_0 = A_0 B_0, \quad C_1 = (A_0 + A_1)(B_0 + B_1), \quad C_2 = A_1 B_1.$$

Finally, the result of multiplication is

$$C(x) = A(x)B(x) = C_0 + x^{2^d}(C_0 + C_1 + C_2) + x^{2 \cdot 2^d} C_2.$$

We compute C_0 and C_1 using FAFFT with $2^{l'} = 2^{d+1}$, and C_2 using radix-16 multiplication.

Algorithm 8 Hybrid Karatsuba-FAFFT method

Require: $A(x) = A_1 X + A_0$, $B(x) = B_1 X + B_0$, where $X = x^{2^d}$
Ensure: $C(x) = A(x) \times B(x)$
1: $C_0 \leftarrow A_0 \times B_0$ using `FAFFT` algorithm.
2: $C_2 \leftarrow A_1 \times B_1$ using radix-16 algorithm.
3: $C_1 \leftarrow (A_0 + A_1) \times (B_0 + B_1)$ using `FAFFT` algorithm.
4: $C(x) \leftarrow C_2 X^2 + (C_0 + C_1 + C_2)X + C_0$.
5: **return** $C(x)$

In our approach, we choose $X = x^{2^d}$ as the split point. While one could split the polynomials at $X = x^r$, which would only require modifying lines 1 and 2 of Algorithm 8, splitting at $X = x^{2^d}$ is more efficient for the following reason.

Let $X = x^{2^d}$. Given the degrees $\deg A_0 = \deg B_0 = 2^d - 1$ and $\deg A_1 = \deg B_1 = r - 1$, the Karatsuba components have degrees $\deg C_0 = \deg C_1 = 2 \cdot 2^d - 2$ and $\deg C_2 = 2r - 2$. With the split point $X = x^{2^d}$, the degrees of the individual terms in the final result $C(x) = C_0 + (C_0 + C_1 + C_2)X + C_2 X^2$ relate to the total degree $2n - 2$ as follows:

$$\deg C_0 = 2 \cdot 2^d - 2 < 2n - 2, \quad \deg(C_2 X^2) = 2r + 2 \cdot 2^d - 2 = 2n - 2.$$

Since the total degree is $\deg C(x) = 2n - 2$, the degree of the middle term must satisfy:

$$\deg(C_0 + C_1 + C_2) \leq 2n - 2 - \deg(X) = 2n - 2 - 2^d = 2^d + 2r - 2.$$

If the condition $2r < 2^d$ is satisfied, which holds for both HQC-1 and HQC-3, we obtain the following inequality:

$$\deg(C_0 + C_1 + C_2) \leq 2^d + 2r - 2 < 2 \cdot 2^d - 2 = \max\{\deg C_0,\ \deg C_1,\ \deg C_2\}. \quad (1)$$

Note that $2 \cdot 2^d - 2$ is the maximum degree supported by the FAFFT of length 2^{d+1} used to compute C_0 and C_1. Consequently, in line 4 of Algorithm 8, the upper $2^d - 2r$ bits of $C_0 + C_1 + C_2$ are zero and can be omitted from the computation.

In contrast, if the polynomials are split at $X = x^r$, the degree of the middle term is $\deg(C_0 + C_1 + C_2) = 2 \cdot 2^d + r - 2$. Since this exceeds $2 \cdot 2^d - 2$, the inequality in (1) does not hold. Thus, splitting at $X = x^{2^d}$ reduces the XOR operations required for reconstruction.

Table 8: parameters for Karatsuba-FAFFT in HQC-1, -3.

Param.	n	FAFFT				radix-16
		$2^{l'}$	m	k	Σ	r
HQC-1	17669	32768	32	10	$v_{26} + W_{10}$	1285
HQC-3	35851	65536	32	11	$v_{27} + W_{11}$	3083

5 Experimental results and comparisons

In this section, we evaluate the efficiency of the optimization methods proposed in Section 3 and Section 4 by implementing and benchmarking them on a NUCLEO-L4R5ZI evaluation board equipped with an STM32L4R5ZI microcontroller. The platform is based on an ARM Cortex-M4 core and features 2 MB of flash memory and 640 KB of SRAM. As specified by the `pqm4` framework [KPR+], we set the system clock to 20 MHz and configure the flash access latency to zero wait states. All code is compiled with `arm-none-eabi-gcc` version 11.3.1. Clock cycles are measured using the DWT cycle counter (`DWT_CYCCNT`). Since the Cortex-M4 has no cache or out-of-order execution and our implementations are constant-time, the cycle counts are deterministic across runs.

5.1 Side-channel resistance

We keep our implementations with a time constancy property, in which memory access and control flow are independent of secret values, to prevent timing side-channel leakage. All operations in our FAFFT implementation consist of bitwise XORs with data-independent access patterns. For the radix-16 multiplication, the Karatsuba and Toom-Cook algorithms operate in a word-wise manner using only XOR and integer multiplications without secret-dependent branches. In the Hybrid FAFFT-CRT method, the sparse-dense multiplication for the CRT reconstruction uses fixed access patterns determined by the public polynomial $\mathcal{P}_{d+1}(x)$. Although the secret values used in polynomial multiplications of HQC are sparse, our implementations treat all inputs as dense and do not exploit their sparsity.

5.2 FAFFT butterfly

Table 9 compares the performance of the optimized FAFFT butterfly described in Section 3.3 against the implementation in [LJKH25]. Our optimized butterfly reduces the cycle count by 3.2% to 3.5% across all lengths used in HQC.

Table 9: Clock cycle comparison of the FAFFT butterfly.

2^l	[LJKH25]	Our code
32768	314,494	303,740 (-3.4%)
65536	709,110	683,954 (-3.5%)
131072	1,589,934	1,539,345 (-3.2%)

5.3 Polynomial multiplication

Table 10 presents the benchmark results for various polynomial multiplication methods. In this comparison, the FAFFT column of our code refers to the implementation from [LJKH25] with our butterfly optimization applied.

For HQC-1 and HQC-3, the Hybrid FAFFT-CRT method achieves the highest efficiency. Specifically, the output-stationary variant reduces the cycle count by 36.0% and 29.3% for

HQC-1 and HQC-3, respectively, compared to the baseline. While the Hybrid Karatsuba-FAFFT and Radix-16 methods also outperform the baseline for these parameters, they are less efficient than the FAFFT-CRT approach.

For HQC-5, where hybrid splitting methods are not applied, our FAFFT code with our optimized butterfly algorithm reduces the cycle count by 3.0% compared to the baseline. The Radix-16 method is significantly less efficient for this degree due to the increased complexity of Karatsuba and Toom-Cook combinations required for the larger degree.

Table 10: Clock cycle comparison of HQC polynomial multiplication.

	Baseline	Ours					Concurrent
	FAFFT [LJKH25]	FAFFT	FAFFT-CRT (input-stat)	FAFFT-CRT (output-stat)	Karat-FAFFT	Radix-16	Chen et al. [CCC+26]
HQC-1	2,841,783	2,756,150	1,891,458	1,819,624	2,521,564	2,147,797	1,753,860
HQC-3	6,279,452	6,092,539	4,572,399	4,441,928	5,697,162	6,083,163	4,836,147
HQC-5	6,290,041	6,103,141	-	-	-	12,460,424	6,290,041

Remark (Comparison with concurrent work). During our submission, a concurrent and independent work [CCC+26] appeared, which also targets HQC on the Cortex-M4. While our work introduces batched and output-stationary variants of sparse-dense polynomial multiplication and optimizes the FAFFT butterfly, they select a CRT modulus of minimal Hamming weight and additionally optimize the Reed-Solomon decoder and apply transform caching during encryption. We measured their public implementation on the same NUCLEO-L4R5ZI board and include the polynomial multiplication cycle counts in the rightmost column of Table 10 and the KEM cycle counts in the rightmost column of Table 11. At the polynomial multiplication level, our FAFFT-CRT requires fewer cycles on HQC-3, while theirs is slightly faster on HQC-1. For HQC-5, Chen et al. retain the baseline FAFFT polynomial multiplication [LJKH25], while our optimized butterfly yields a 3.0% reduction over the same baseline. At the KEM level, their lower encapsulation and decapsulation cycle counts arise primarily from the Reed-Solomon decoder and transform caching, which are orthogonal to our polynomial-multiplication optimizations and could be combined with our methods. Full algorithmic details of their construction are available in [CCC+26].

5.4 Application to HQC

In this section, we evaluate the performance of the three HQC parameter sets across five distinct implementations: (i) the baseline FAFFT from [LJKH25], (ii) the FAFFT with our optimized butterfly algorithm, (iii) the hybrid FAFFT-CRT method, (iv) the hybrid Karatsuba-FAFFT method, and (v) the radix-16 implementation. Notably, both hybrid FAFFT approaches also integrate the optimized butterfly algorithm. The benchmark results are summarized in Table 11, where Keygen, Encap, and Decap denote key generation, encapsulation, and decapsulation, respectively.

For HQC-1, the output-stationary variant of our FAFFT-CRT method improves the performance of key generation, encapsulation, and decapsulation by 25.5%, 26.1%, and 23.5%, respectively, compared to the baseline. In the case of HQC-3, the same variant provides improvements of 19.3%, 20.0%, and 19.1%. For HQC-5, where hybrid methods are not applied, our optimized FAFFT achieves a consistent improvement of 1.4 - 1.6%. Conversely, the Radix-16 method at this security level is slower by 48-51%, which is consistent with the asymptotic advantage of FFT-based multiplication for larger polynomial degrees.

Based on these results, we recommend the Hybrid FAFFT-CRT method for HQC-1 and HQC-3 when sufficient flash is available, since it achieves the largest speedups across

Table 11: Clock cycles for HQC key generation, encapsulation, and decapsulation on Cortex-M4.

		Baseline	Ours					Concurrent
		FAFFT [LJKH25]	FAFFT	FAFFT-CRT (input-stat)	FAFFT-CRT (output-stat)	Karat-FAFFT	Radix-16	Chen et al. [CCC+26]
HQC-1	Keygen	4,015,835	3,930,193	3,065,459	2,993,695	3,692,595	3,322,032	2,925,125
	Encap	7,843,285	7,671,995	5,942,521	5,798,993	7,196,807	6,455,688	5,181,825
	Decap	13,026,321	12,769,358	10,175,173	9,959,877	12,056,601	10,944,987	8,739,392
HQC-3	Keygen	9,497,097	9,310,200	7,790,316	7,659,978	8,914,845	9,300,408	8,054,563
	Encap	18,341,826	17,968,035	14,928,198	14,667,541	17,177,322	17,948,454	14,427,433
	Decap	28,863,458	28,302,758	23,743,111	23,352,058	27,116,701	28,273,407	22,874,788
HQC-5	Keygen	12,700,457	12,513,535	-	-	-	18,874,866	12,701,983
	Encap	24,109,313	23,735,473	-	-	-	36,458,054	21,890,109
	Decap	38,752,685	38,191,910	-	-	-	57,275,806	34,769,380

all KEM operations. For HQC-5, the residual $r = 24{,}869$ in $n = 2^{15} + r$ is too large for the FAFFT-CRT split to apply, so the optimized FAFFT path with our improved butterfly is the appropriate choice and provides a 1.4–1.6% improvement over the baseline. The radix-16 method is the appropriate choice for deployments with limited flash, since it has the smallest flash usage, even though it is slower than FAFFT-CRT for HQC-1 and HQC-3.

5.5 Memory usage

Table 12 reports the flash and RAM usage for each implementation. The flash column corresponds to the `.text` section, which includes executable code and read-only data. The RAM column is the sum of the peak stack usage (maximum across key generation, encapsulation, and decapsulation) and the `.bss` and `.data` sections.

Table 12: Flash and RAM usage in bytes on Cortex-M4.

Method		HQC-1		HQC-3		HQC-5	
		Flash	RAM	Flash	RAM	Flash	RAM
Baseline	FAFFT [LJKH25]	195,678	63,116	196,866	125,064	201,950	154,916
Ours	FAFFT	191,518	63,108	192,706	124,972	197,790	155,000
	FAFFT-CRT (input-stat)	213,200	59,912	217,806	122,124	-	-
	FAFFT-CRT (output-stat)	246,250	71,496	291,938	147,996	-	-
	Karat-FAFFT	204,366	54,612	222,354	110,132	-	-
	Radix-16	56,502	54,180	87,486	111,532	93,610	176,328

Compared to the baseline FAFFT, the flash overhead is approximately 4–13% for Karatsuba-FAFFT, 9–11% for the input-stationary FAFFT-CRT, and 26–48% for the output-stationary FAFFT-CRT, due to the pre-computed arrays for the CRT reconstruction, the additional radix-16 multiplication code, and (for the output-stationary variant) the precomputed shift table *Tsh* and rearranged offset arrays. However, both methods reduce RAM usage because they use a smaller FAFFT of size 2^{d+1} instead of 2^{d+2}, requiring less intermediate buffer space. Among the two, the FAFFT-CRT method provides the best speed while the Karatsuba-FAFFT method achieves a slightly smaller RAM footprint. The radix-16 method has the smallest flash usage, with a 54–71% reduction compared to the baseline FAFFT, since it does not include the FAFFT code, but requires more RAM for the radix-16 conversion buffers. All implementations fit within the 640 KB SRAM of the target board.

6 Conclusion

In this paper, we presented optimized polynomial multiplication techniques for HQC on the ARM Cortex-M4, including hybrid FAFFT-CRT, hybrid Karatsuba-FAFFT, and radix-16 methods. These approaches effectively reduce the overhead of FAFFT, particularly for polynomial degrees of the form $n = 2^d + r$, with a small r. Furthermore, we improved the FAFFT butterfly through efficient register scheduling, shortened XOR sequences for fixed-coefficient multiplications, and the replacement of bit swaps with the `SWAPMOVE` instruction.

Also, to the best of our knowledge, we apply radix-16 multiplication to HQC for the first time. Radix-16 multiplication is well-suited for Cortex-M4, which lacks dedicated carry-less multiplication instructions but supports MAC operations, enabling efficient emulation via integer multiplication. We further optimize it using an operation-count cost model to identify efficient Karatsuba and Toom-Cook combinations.

Experimental results show that our methods significantly outperform the state-of-the-art on Cortex-M4. In particular, the output-stationary variant of the hybrid FAFFT-CRT method achieves up to 25.5%, 26.1%, and 23.5% speedups for key generation, encapsulation, and decapsulation in HQC-1, and similar improvements of around 19–20% for HQC-3. For HQC-5, the residual $r = 24{,}869$ in $n = 2^{15} + r$ is too large for our hybrid splits to apply, so the improvement comes from the optimized FAFFT butterfly alone and is limited to 1.4–1.6% across all KEM operations. For deployments with limited flash, the radix-16 method offers the smallest flash usage, although it is slower than FAFFT-CRT for HQC-1 and HQC-3. As HQC is selected for NIST standardization, our optimizations enable its practical deployment on resource-constrained embedded platforms.

However, achieving substantial improvements for HQC-5 remains challenging due to its large residual size, and further optimization for this parameter set is left for future work. In particular, exploring alternative approaches such as the KS+CRT method proposed by Chen et al. [CCPY26] on Cortex-M4 is a promising direction for improving performance. In addition, integrating the optimizations of the concurrent work of Chen et al. [CCC+26] with our polynomial-multiplication methods is another promising direction.

Our Hybrid FAFFT-CRT method consists of three computational components: FAFFT for the main product, radix-16 multiplication for the small residual via the `UMLAL` multiply-accumulate instruction, and sparse-dense multiplication for the CRT recombination via shift-and-XOR that exploits the sparsity of $\mathcal{P}_{d+1}(x)$. All three components are well suited to the Cortex-M4. On architectures providing dedicated carry-less multiplication instructions, for example `PCLMULQDQ` on AVX2 or `PMULL.P64` on ARMv8 with the cryptographic extension, divide-and-conquer methods such as Karatsuba and Toom-Cook become advantageous. Among FFT-based methods, the FAFFT path of Chen et al. [CCPY26], which splits inputs into a power-of-two part and a residual and uses FAFFT for the power-of-two product and recursive Karatsuba for the remaining cross and residual products, would be more efficient than our FAFFT-CRT. A detailed cross-platform evaluation is left for future work.

Acknowledgments

This work was supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. RS-2025-02263143, Development of Cybersecurity Threat Response Technologies for Satellite Ground Stations).

References

[AAB+22] Carlos Aguilar-Melchor, Nicolas Aragon, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Edoardo Persichetti, Gilles Zémor, Jurjen Bos, Arnaud Dion, Jerome Lacan, Jean-Marc Robert, and Pascal Veron. HQC. Technical report, National Institute of Standards and Technology, 2022. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/round-4-submissions.

[AAC+22] Gorjan Alagic, Daniel Apon, David Cooper, Quynh H. Dang, Thinh Dang, John Kelsey, Jacob Lichtinger, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Andrew Robinson, Daniel Smith-Tone, Diana Tonev, Apostol Vassilev, and William Whyte. Status report on the third round of the nist post-quantum cryptography standardization process. NIST IR 8413, National Institute of Standards and Technology, 2022.

[ABCG20] Erdem Alkim, Yusuf Alper Bilgin, Murat Cenk, and François Gérard. Cortex-M4 optimizations for {R,M}LWE schemes. *IACR TCHES*, 2020(3):336–357, 2020.

[ACC+22] Amin Abdulrahman, Jiun-Peng Chen, Yu-Jia Chen, Vincent Hwang, Matthias J. Kannwischer, and Bo-Yin Yang. Multi-moduli NTTs for Saber on Cortex-M3 and Cortex-M4. *IACR TCHES*, 2022(1):127–151, 2022.

[AHKS22] Amin Abdulrahman, Vincent Hwang, Matthias J. Kannwischer, and Amber Sprenkels. Faster Kyber and Dilithium on the Cortex-M4. In Giuseppe Ateniese and Daniele Venturi, editors, *ACNS 22International Conference on Applied Cryptography and Network Security*, volume 13269 of *LNCS*, pages 853–871. Springer, Cham, June 2022.

[BGTZ23] R. P. Brent, P. Gaudry, E. Thomé, and P. Zimmermann. gf2x library. https://gitlab.inria.fr/gf2x/gf2x, 2023. Accessed: Feb. 11, 2023. [Online].

[BKS19] Leon Botros, Matthias J. Kannwischer, and Peter Schwabe. Memory-efficient high-speed implementation of Kyber on Cortex-M4. In Johannes Buchmann, Abderrahmane Nitaj, and Tajje eddine Rachidi, editors, *AFRICACRYPT 19*, volume 11627 of *LNCS*, pages 209–228. Springer, Cham, July 2019.

[Can89] David G Cantor. On arithmetical algorithms over finite fields. *Journal of Combinatorial Theory, Series A*, 50(2):285–300, 1989.

[CC21] Ming-Shing Chen and Tung Chou. Classic McEliece on the ARM Cortex-M4. *IACR TCHES*, 2021(3):125–148, 2021.

[CCC+26] Ming-Shing Chen, Tun-You Chien, Chun-Ming Chiu, Han-Hsuan Lin, Chun-Tao Peng, and Bo-Yin Yang. Additive FFTs for HQC on ARM Cortex-M4, revisited. *IACR TCHES*, 2026(3), 2026.

[CCK21] Ming-Shing Chen, Tung Chou, and Markus Krausz. Optimizing BIKE for the intel haswell and ARM Cortex-M4. *IACR TCHES*, 2021(3):97–124, 2021.

[CCPY26] Ming-Shing Chen, Chun-Ming Chiu, Chun-Tao Peng, and Bo-Yin Yang. Accelerating HQC with additive FFT. *IACR TCHES*, 2026(2):520–544, 2026.

[CGKT22] Ming-Shing Chen, Tim Güneysu, Markus Krausz, and Jan Philipp Thoma. Carry-less to BIKE faster. In Giuseppe Ateniese and Daniele Venturi, editors, *ACNS 22International Conference on Applied Cryptography and Network Security*, volume 13269 of *LNCS*, pages 833–852. Springer, Cham, June 2022.

[CHK+21] Chi-Ming Marvin Chung, Vincent Hwang, Matthias J. Kannwischer, Gregor Seiler, Cheng-Jhih Shih, and Bo-Yin Yang. NTT multiplication for NTT-unfriendly rings. *IACR TCHES*, 2021(2):159–188, 2021.

[GKS21] Denisa O. C. Greconici, Matthias J. Kannwischer, and Amber Sprenkels. Compact Dilithium implementations on Cortex-M3 and Cortex-M4. *IACR TCHES*, 2021(1):1–24, 2021.

[GM10] Shuhong Gao and Todd Mateer. Additive fast fourier transforms over finite fields. *IEEE Transactions on Information Theory*, 56(12):6265–6272, 2010.

[HAZ+24] Junhao Huang, Alexandre Adomnicai, Jipeng Zhang, Wangchen Dai, Yao Liu, Ray C. C. Cheung, Çetin Kaya Koç, and Donglong Chen. Revisiting Keccak and Dilithium implementations on ARMv7-M. *IACR TCHES*, 2024(2):1–24, 2024.

[HZZ+22] Junhao Huang, Jipeng Zhang, Haosong Zhao, Zhe Liu, Ray C. C. Cheung, Çetin Kaya Koç, and Donglong Chen. Improved plantard arithmetic for lattice-based cryptography. *IACR TCHES*, 2022(4):614–636, 2022.

[JLK+24] Jihoon Jang, Myeonghoon Lee, Suhri Kim, Seogchung Seo, and Seokhie Hong. Optimized implementation of GF(2)[x] multiplication for HQC on AVX2. *Journal of the Korea Institute of Information Security & Cryptology*, 34(5):841–853, 2024.

[KPR+] Matthias J. Kannwischer, Richard Petri, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. PQM4: Post-quantum crypto library for the ARM Cortex-M4. https://github.com/mupq/pqm4.

[LANH16] Sian-Jheng Lin, Tareq Y Al-Naffouri, and Yunghsiang S Han. FFT algorithm for binary extension finite fields and its application to reed–solomon codes. *IEEE Transactions on Information Theory*, 62(10):5343–5358, 2016.

[LCH14] Sian-Jheng Lin, Wei-Ho Chung, and Yunghsiang S. Han. Novel polynomial basis and its application to reed-solomon erasure codes. In *55th FOCS*, pages 316–325. IEEE Computer Society Press, October 2014.

[LCK+18] Wen-Ding Li, Ming-Shing Chen, Po-Chun Kuo, Chen-Mou Cheng, and Bo-Yin Yang. Frobenius additive fast fourier transform. In *Proceedings of the 2018 ACM International Symposium on Symbolic and Algebraic Computation*, pages 263–270, 2018.

[LJKH25] Myeonghoon Lee, Jihoon Jang, Suhri Kim, and Seokhie Hong. Improved frobenius fft for code-based cryptography on cortex-m4. *IEEE Internet of Things Journal*, 12(17):36318–36327, 2025.

[MPC00] Lauren May, Lyta Penna, and Andrew J. Clark. An implementation of bitsliced DES on the Pentium MMX^{TM} processor. In Ed Dawson, Andrew Clark, and Colin Boyd, editors, *ACISP 00*, volume 1841 of *LNCS*, pages 112–122. Springer, Berlin, Heidelberg, July 2000.

[VzGG96] Joachim Von zur Gathen and Jürgen Gerhard. Arithmetic and factorization of polynomial over F2. In *Proceedings of the 1996 international symposium on Symbolic and algebraic computation*, pages 1–9, 1996.

[XZL+20] Zejun Xiang, Xiangyong Zeng, Da Lin, Zhenzhen Bao, and Shasha Zhang. Optimizing implementations of linear layers. *IACR Trans. Symm. Cryptol.*, 2020(2):120–145, 2020.