A Memory-Efficient and Assembly-Optimized Implementation of NTRU+

SuBeen Cho[1][0009-0004-4352-8021], Jiwon Bang[1][0009-0006-5453-1722], Minjoo Sim[2][0000-0001-5242-214X], and Hwa-Jeong Seo[1][0000-0003-0069-9061]*

[1] Department of Convergence Security,
Hansung University, Seoul (02876), South Korea
{chosubin1208, bgjiiw0n754}@gmail.com
[2] Department of Information Computer Engineering,
Hansung University, Seoul (02876), South Korea
minjoos9797@gmail.com

Abstract. This paper presents a memory-efficient and high-speed implementation of NTRU+, one of the key encapsulation mechanisms (KEMs) selected by Korea's post-quantum cryptography project (KpqC), on the ARM Cortex-M4. NTRU+ is small enough to run on its own on a Cortex-M4 class microcontroller, yet in real embedded environments, the peak stack occupied by polynomial buffers and the running time dominated by the NTT become key constraints. To address this, in the proposed technique, we reduce memory by analyzing when each polynomial buffer is actually live and restructuring the algorithm so that only a single buffer is resident during computation, and we significantly improve speed by rewriting the NTT for the mixed-radix structure of NTRU+ in Cortex-M4 assembly. By doing so, the assembly NTT acceleration compensates for the recomputation overhead introduced by the memory optimization, so that the combined implementation is both smaller and faster than the reference. Experimental results show that, relative to the KpqClean_ver2[1] reference implementation, the proposed technique reduces the peak stack by 83–84% while improving the speed of all three operations by up to 1.8×.

Keywords: Memory Optimization · KpqC · NTRU+ · Lattice-based KEM · Assembly Optimization · Cortex-M4

1 Introduction

Advances in quantum computing pose a fundamental threat to classical public-key cryptography based on the integer factorization and discrete logarithm problems. In response, standardization bodies worldwide are accelerating the transition to post-quantum cryptography [2], and through the KpqC competition [3]

* Corresponding author: hwajeong84@gmail.com.

Korea is selecting its own cryptographic algorithms and advancing their official standardization.

Among them, the lattice-based scheme NTRU+ is a relatively small KEM, and running it in isolation is already feasible on a Cortex-M4. However, in real deployments the peak stack remains a bottleneck. This is the case in particular when the KEM shares SRAM with a protocol stack or certificate processing, on low-cost nodes with only tens of kilobytes, in security modules handling multiple concurrent sessions, or in environments that load several algorithms together. Speed is also a constraint. Since most of the computational cost is taken by the NTT and Inverse NTT, NTT acceleration governs practicality in environments where handshake latency, throughput, and power matter. Yet structural rearrangements that reduce memory often increase recomputation and thus computation time, so memory and speed tend to conflict. Achieving both goals together is therefore the central challenge for embedded NTRU+.

Previous work on memory optimization of embedded lattice cryptography has mostly targeted schemes that use power-of-two NTTs, such as ML-KEM [4] and ML-DSA [5]. NTRU+ differs from these in that it operates with a mixed-radix NTT structure that combines radix-2 and radix-3, and memory optimization specialized to such a structure remains underexplored. There is therefore a need for research that improves both memory and speed in a way tailored to the structural characteristics of NTRU+.

This paper proposes an optimized implementation that enables NTRU+ to run efficiently on the Cortex-M4.

1.1 Contributions

The contributions of this paper are as follows:

- **Structure-aware phased stack reduction for NTRU+.** Exploiting NTRU+'s mixed-radix ring, ternary secrets, and interleaved serialization, we apply lifetime analysis and fusion techniques to reduce simultaneously active polynomials to just one. Notably, fusing the unpacking of serialization directly inside the multiplication loop resolves a challenge specific to NTRU+.
- **Assembly acceleration of the NTRU+ mixed-radix NTT.** Overcoming the limitations of strictly power-of-two NTTs [6], we optimize the NTRU+ NTT/INVNTT—including radix-3 butterflies and $x^4 - \zeta/x^3 - \zeta$ block products—alongside basemul and baseinv in Cortex-M4 assembly to reduce hot-loop cycles.

2 Background

2.1 KpqC

Launched in 2022 with the goal of establishing an independent national cryptographic standard, Korea's post-quantum cryptography competition (KpqC) [3]

Table 1: NTRU+ parameters [13].

Item	KEM768	KEM864	KEM1152
Security level (NIST)	1	3	5
Polynomial degree n	768	864	1152
Modulus q	3457	3457	3457
basemul unit	$x^4 - \zeta$	$x^3 - \zeta$	$x^4 - \zeta$
Polynomial serialization size (B)	1152	1296	1728
One polynomial (runtime, $2n$ B)	1536	1728	2304
Public key / secret key (B)	1152 / 2384	1296 / 2592	1728 / 3456
Ciphertext / shared key (B)	1152 / 32	1296 / 32	1728 / 32

selected its standardization targets through a three-year evaluation. In the key encapsulation mechanism (KEM) category, NTRU+[7] and SMAUG-T[8] were selected, and in the digital signature category, AIMer[9] and HAETAE[10].

2.2 NTRU+

NTRU+ polynomial operations take place in the following ring [7]:

$$R_q = \mathbb{Z}_q[x]/(x^n - x^{n/2} + 1), \qquad q = 3457. \tag{1}$$

The modulus polynomial offers a narrow algebraic attack surface and parameter flexibility without restricting n to a power of two. Since $q \equiv 1 \pmod 6$, $\mathbb{Z}_q$ contains a 6th root of unity, enabling multiplication acceleration via a mixed-radix (radix-2 and radix-3) NTT [11].

Table 1 summarizes the parameters of the three security levels provided by NTRU+. A runtime polynomial buffer occupies 16 bits per coefficient (2n bytes) and heavily dominates stack usage. The KEM operations heavily utilize the NTT domain. Key generation computes the public key $\hat{h} = \hat{g} \cdot \hat{f}^{-1}$ from ternary secrets f, g. Encapsulation encodes a message into a ternary polynomial $\hat{m}$ and produces ciphertext $\hat{c} = \hat{m} + \hat{h} \cdot \hat{r}$. Decapsulation recovers the message by inverse-transforming $\hat{c} \cdot \hat{f} \pmod 3$ and verifies ciphertext honesty via the FO transform's re-encryption check [12].

2.3 Related Work on Memory Optimization

Research on memory reduction for embedded lattice cryptography has reduced transient storage through on-the-fly generation, representation compression, buffer reuse, and streaming.

Table 2 summarizes representative prior work. Botros et al. [4] generate the public matrix A on the fly and stream serialization in Kyber, reducing the memory taken by matrix A and the NTT buffers. Bos et al. [14] reduce the polynomial workspace in Dilithium through polynomial compression, alternative NTTs, a schoolbook fallback for some operations, and recomputation. Greconici et al.

Table 2: Comparison of representative embedded memory-optimization techniques.

Related work	Target	OTF	Stream	Recomp	Reuse	Dominant bottleneck
Botros et al. [4]	Kyber	✓	✓		✓	matrix A / NTT buffers
Bos et al. [14]	Dilithium	✓		✓	✓	polynomial workspace
Greconici et al. [5]	Dilithium	✓			✓	simultaneously active polys
Hou et al. [15]	ML-KEM	✓	✓		✓	encaps/decaps buffers
This work	**NTRU+**	✓	✓	✓	✓	NTT working poly / ciphertext

OTF: on-the-fly generation, Stream: serialization streaming, Recomp: recomputation, Reuse: buffer reuse. ✓ marks the technique each work primarily emphasizes.

[5] minimize the number of simultaneously active polynomials in Dilithium by reordering operations and reusing buffers. Hou et al. [15] reduce the encapsulation and decapsulation buffers in ML-KEM through split loading and streaming output. This work integrates these techniques tailored to NTRU+, which is NTT-friendly and has a mixed-radix structure, and additionally shows that algorithm-structure-based memory optimization is orthogonal to Cortex-M4 assembly NTT acceleration, distinguishing it from prior work.

2.4 Cortex-M4

The ARM Cortex-M4 [16] is a 32-bit microcontroller based on ARMv7E-M [17] that has become the de facto reference platform for embedded post-quantum cryptography benchmarking. It provides DSP extensions such as single-cycle multiply-accumulate (MAC) and the 2×16-bit SIMD instructions `sadd16`, `ssub16`, and `smuad`, efficiently handling integer-centric cryptographic computation, and its lack of a data cache makes access latency deterministic, which is favorable for constant-time implementation. pqm4 [18], our measurement baseline, established the practice of measuring cycles, stack, and code size on this platform.

3 Proposed Method

In key generation, encapsulation, and decapsulation of NTRU+, polynomial buffers dominate the stack and the NTT dominates the running time, so making NTRU+ practical on embedded targets requires reducing both costs together. We attack the two costs at different layers. The memory optimization restructures the computation flow above the NTT to reduce the number of polynomial buffers resident at the same time, while the speed optimization rewrites the hot loops inside the NTT in Cortex-M4 assembly to reduce cycles. Because the two techniques operate respectively above and inside the NTT, they combine orthogonally without interfering with each other, and every step is implemented in constant time to guarantee side-channel safety.

Table 3: Per-operation application of the memory-optimization techniques.

Technique	KG	Enc	Dec
Lifetime-based reuse	✓	✓	✓
On-the-fly generation	✓	✓	✓
Serialization–hash streaming		✓	✓
Direct-byte multiply	✓	✓	✓
Union overlay		✓	✓
Representation compression			✓
Hash-based re-encryption check			✓

KG: key generation, Enc: encapsulation, Dec: decapsulation.

3.1 Optimizing for Memory

Most of the stack of a lattice-based KEM is taken by polynomial buffers, so the goal of optimization is to reduce the number of simultaneously live buffers. To this end, we exploit three properties common to lattice KEMs—block-wise independence of the matrix–vector product, regenerability of pseudorandom coefficients from a seed, and non-overlapping lifetimes of the main-stage variables—and apply seven techniques. *Lifetime-based reuse* overwrites a buffer whose lifetime has ended with the next polynomial (its strongest form being in-place computation), and *union overlay* overlaps buffers with disjoint lifetimes at the same address. *Representation compression* compresses ternary coefficients to 2 bits, and *on-the-fly generation* generates the public matrix and randomness from a seed on demand. *Direct-byte multiply* multiplies serialized bytes block-wise without deserialization, *serialization–hash streaming* streams the output into the hash in chunks, and *hash-based re-encryption check* compares only hash values instead of the two random polynomials in the FO check of decapsulation. The per-operation application of each technique is shown in Table 3.

Key Generation Key generation computes the inverse of the secret polynomial f, forms the public polynomial $h = g \cdot f^{-1}$, and then computes the inverse of h. The reference implementation uses five variables f, f^{-1}, g, h, h^{-1}, whose overlapping lifetimes leave up to four resident when $h = g \cdot f^{-1}$ is computed. Exploiting the fact that the inversion reads its entire input before writing any output, our implementation keeps the input in the secret-key buffer and overwrites the inverse in place in the same buffer. The secret polynomials f and g are generated on the fly from the seed, the inverse input of h is read block-wise from the secret-key bytes rather than deserialized into a polynomial, and each result is written out to the key buffer immediately. As a result a single buffer a is reused sequentially as $f \to f^{-1} \to g \to h \to h^{-1}$, the number of simultaneously active polynomials drops to one, and the peak stack of key generation falls from 9,440 B to 2,032 B. The buffer lifetimes are illustrated in Figure 1.

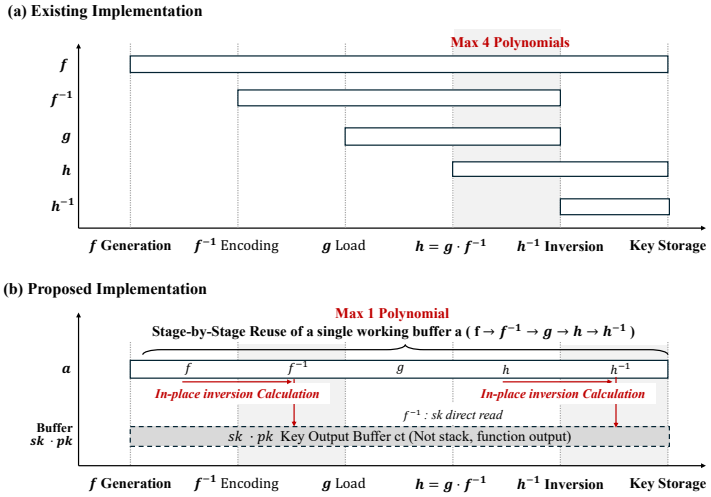

Fig. 1: Polynomial-buffer lifetimes in key generation. (a) the original implementation and (b) the proposed implementation. A single buffer a is reused sequentially as $f \to f^{-1} \to g \to h \to h^{-1}$.

Encapsulation Encapsulation forms the ciphertext $\hat{c} = \hat{m} + \hat{h} \cdot \hat{r}$. The reference implementation simultaneously holds four polynomials, the random value, the message, the public key, and the accumulator, together with a serialization buffer of about 1.7 KB. Our implementation derives the shared key and seed by hashing, temporarily packs the random $\hat{r}$ into the result buffer, and then overwrites that buffer with the message $\hat{m}$. The serialized bytes of the public key and the random value are read directly inside the multiplication loop and accumulated, so no separate temporary polynomial is needed, and the serialization is streamed into the hash in chunks, removing the large intermediate buffer. As a result the number of simultaneously active polynomials drops from four to one, and the peak stack of encapsulation falls from 9,216 B to 2,320 B. The buffer lifetimes are shown in Figure 2.

Decapsulation Because decapsulation recovers the message and then performs the FO re-encryption check, it is the maximum-bottleneck region with the most simultaneously active polynomials, and the reference implementation holds seven polynomials. Our implementation reads the ciphertext and secret key directly from bytes and multiplies, stores the recovered ternary message compressed to 2 bits per coefficient, and fuses the block-wise subtraction of the re-transformed message from the ciphertext with the multiplication by the key inverse, overwriting the same buffer. The re-encryption check further compares only the hash values of the two random polynomials instead of comparing them coefficient by coefficient, removing the comparison buffer as well. On the honest path the two random values are equal, so the hashes always match and the output

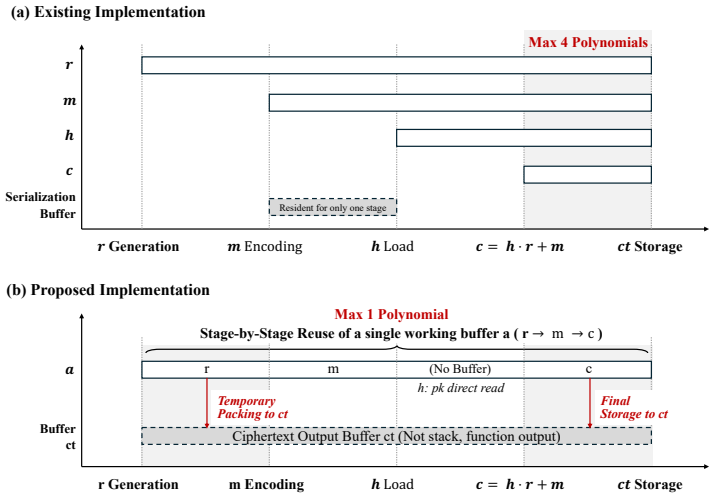

Fig. 2: Polynomial-buffer lifetimes in encapsulation. (a) the original implementation and (b) the proposed implementation. A single buffer a is reused sequentially as $r \to m \to c$.

is unchanged, and the probability that a tampered ciphertext passes is at most the hash-collision probability 2^{-1536}, which is negligible, so IND-CCA2 [19,20] security is preserved. As a result the number of simultaneously active polynomials drops from seven to one, and the peak stack of decapsulation falls from 15,880 B to 2,728 B. The full flow is given in Algorithm 1.

3.2 Optimizing for Speed

While the memory optimization targets the polynomial buffers that dominate the stack, the speed optimization targets the NTT that dominates the running time. The NTT and INVNTT form the core kernel shared by all three operations and account for most of the computation time. However, since NTRU+ has a mixed-radix structure combining radix-2 and radix-3, the existing Cortex-M4 NTT assembly kernels [6,21] that assume a power-of-two degree cannot be applied directly. This section reduces the hot-loop cycles of the NTT and INVNTT by exploiting the DSP extensions of the Cortex-M4, namely 2×16-bit SIMD, single-cycle MAC, and the barrel shifter, and by keeping coefficients resident in registers. We describe it along three aspects, the decomposition pipeline, the cycle-reduction levers, and what distinguishes it from power-of-two NTT.

Decomposition pipeline The forward transform decomposes the coefficient representation into the $x^4 - \zeta$ block representation through radix-2 splitting, the radix-3 butterfly, radix-2 Cooley–Tukey layers, and Barrett alignment. The

Algorithm 1 Memory-efficient decapsulation with hash-based verification

Require: ciphertext ct, secret key $(\hat{f},\ \hat{h}^{-1},\ H(pk))$, hash H
Ensure: shared key K
$m_1 \leftarrow \mathrm{crepmod3}(\mathrm{INVNTT}(ct \odot \hat{f}))$ ▷ direct multiply of ct/sk bytes
store m_1 compressed to 2 bits/coefficient ▷ ternary representation compression
$\hat{r}_2 \leftarrow (ct - \mathrm{NTT}_{\mathrm{tern}}(m_1)) \odot \hat{h}^{-1}$ ▷ fused subtract/multiply, in place
$mask \leftarrow H(\mathrm{ser}(\hat{r}_2))$; $\mu \leftarrow \mathrm{SOTP}^{-1}(m_1, mask)$ ▷ recover message via hash of r_2
$\hat{r}_1 \leftarrow \mathrm{NTT}_{\mathrm{tern}}(\mathrm{CBD1}(H(\mu \,\|\, H(pk))))$ ▷ re-derive, reuse buffer
if $H(\mathrm{ser}(\hat{r}_1)) = mask$ **then** ▷ compare hash values instead of polynomials
    $K \leftarrow \mathrm{SHAKE256}(\mu)$
else
    $K \leftarrow$ pseudorandom ▷ implicit rejection

inverse transform reconstructs the original representation through Gentleman–Sande layers, the inverse radix-3, $1/n$ scaling, and the $x^{n/2}$ merge, as illustrated in Figure 3. Modular multiplication uses signed Montgomery reduction [22], while layer alignment uses signed Barrett reduction [21]. As a result, a degree-n polynomial product is reduced to several degree-4 block products, or degree-3 products for KEM864, allowing the transform, multiplication, and inverse path to share a single assembly route.

Cycle-reduction levers Cycle reduction is achieved through six levers. The coefficients and twiddles within a layer are kept resident in registers to reduce memory round-trips, and two 16-bit coefficients packed into a 32-bit register let `sadd16`/`ssub16` process the additions and subtractions of two butterflies in one instruction. Multiply-accumulate is performed in a single cycle and the sum of two products is fused into one `smuad`. Layer reduction is omitted as long as intermediate values stay within the 16-bit range (lazy reduction), and adjacent layers are merged to reduce loads/stores and reductions together (layer merging). Finally, the block-wise product basemul and the block inverse baseinv for key generation are also handled in assembly. Together these six levers cut the instruction count and memory accesses per butterfly, lowering the hot-loop cycles of NTT and INVNTT, which carries over to a speedup in all three operations.

Beyond power-of-two NTT The six cycle-reduction levers presented above were established in the existing Cortex-M4 NTT assembly [6,21] that assumes power-of-two degree. This work restructures them for the mixed-radix structure of NTRU+. Since NTRU+ decomposes not only through radix-2 layers but also through a radix-3 butterfly into $x^4 - \zeta$ (KEM768/1152) and $x^3 - \zeta$ (KEM864) block products, implementing both the radix-3 butterfly and the two kinds of block products in assembly is what distinguishes it from power-of-two degree NTT implementations.

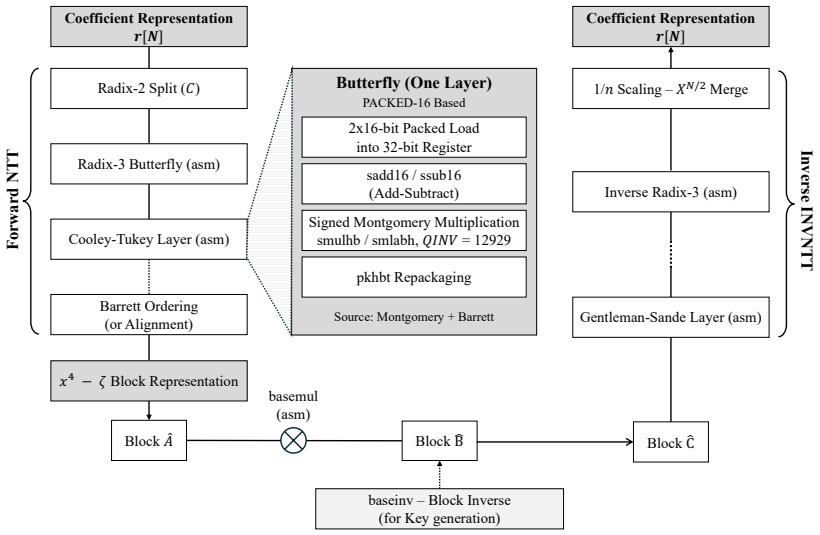

Fig. 3: The assembly NTT pipeline. The forward NTT decomposes the coefficient representation into the block representation ($x^4 - \zeta$ for KEM768/1152, $x^3 - \zeta$ for KEM864), and the inverse INVNTT reverses it.

4 Evaluation

4.1 Experimental Environment

Experiments are conducted on a real NUCLEO-L4R5ZI board (MCU: STM32L4R5), an ARM Cortex-M4 platform, using the pqm4 [18] benchmark framework; the platform specifications are given in Table 4. In this section, Ref denotes the clean implementation of KpqClean_ver2 [1], and Opt denotes the proposed implementation that jointly applies memory optimization and the assembly NTT.

Cycle counts are measured from builds using `arm-none-eabi-gcc -O3`, whereas stack usage and code size are measured from builds using `-Os`. Cycles are reported as the median over 10,000 executions of each operation. Stack usage is obtained by filling the unused stack region with a sentinel value before execution and counting the overwritten bytes afterward. Code size measures the `.text` size of the KEM core and assembly routines.

4.2 Evaluation of Cycle Performance and Memory

To comprehensively evaluate the independent effects and synergy of the proposed optimizations, we comparatively analyze the peak stack usage, cycle counts, and code size across four versions (Ref C, Memory-only C, Assembly-only, and Memory + Assembly). The experimental results are summarized in Table 5.

Table 4: Experimental platform specifications.

	NUCLEO-L4R5ZI
MCU	STM32L4R5ZI
Core	ARM Cortex-M4F
ISA	ARMv7E-M
FPU	FPv4-SP-D16
Flash	2 MB
SRAM	640 KB
Clock (speed)	20 MHz (`CLOCK_BENCHMARK`)
Clock (stack)	120 MHz (`CLOCK_FAST`)

Memory Optimization Performance Restructuring the activation timing of polynomial buffers reduces the peak stack by 83–84% across all three parameter sets compared to the reference implementation [1]. This consistency indicates that the buffer-lifetime restructuring is effective regardless of the security level. Specifically, the peak stack of KEM1152 decreases from 23,608 B to 3,832 B, significantly enhancing applicability in memory-constrained embedded devices. Conversely, the code size increases by approximately 67–72%. This expansion is driven by the restructuring logic and recomputations required to eliminate large temporary buffers, alongside the integration of core assembly kernels. Consequently, a clear trade-off emerges between stack memory reduction and code size footprint.

Cycle Performance The proposed implementation addresses this by jointly applying assembly acceleration tailored to the mixed-radix structure of NTRU+. As a result, the optimized implementation achieves up to a 1.83× speedup in key generation, 1.54× in encapsulation, and 1.30× in decapsulation relative to the reference design. The impact of this assembly acceleration is particularly pronounced in key generation, where polynomial operations heavily dominate the workload.

Furthermore, key generation can exhibit variance in cycle distribution due to the rejection sampling step involved in finding the secret-polynomial inverse. To mitigate the influence of extreme values and provide a more objective comparison, the computational costs in this evaluation are reported as medians.

5 Conclusion

This paper proposed an optimization that enables NTRU+, a KpqC KEM standardization target, to run practically in memory constrained embedded environments. The key idea is to combine memory optimization above the NTT with assembly optimization inside the NTT, reducing memory usage without the usual speed penalty. As a result, the peak stack is reduced by 83–84% relative to the official reference implementation, reaching a level operable within

Table 5: NTRU+ performance and memory evaluation on pqm4 (memory: `-Os`; cycles: `-O3`).

Parameter	Version	Stack	Cycles			Code size
			KeyGen	Enc	Dec	
KEM768	Ref C [1]	15,880	662,103	556,926	709,579	3,956
	Memory-only C	2,728	659,543	534,598	743,287	6,108
	Assembly-only	15,880	346,287	344,607	410,895	5,300
	Memory + Assembly	2,728	361,977	362,019	544,747	6,696
KEM864	Ref C [1]	17,816	767,950	625,560	835,673	4,632
	Memory-only C	3,056	781,310	620,890	891,592	7,552
	Assembly-only	17,816	446,897	447,068	535,811	5,972
	Memory + Assembly	3,056	485,023	481,172	708,331	7,988
KEM1152	Ref C [1]	23,608	978,949	831,399	1,080,465	4,502
	Memory-only C	3,832	978,196	808,330	1,155,435	7,004
	Assembly-only	23,608	578,204	585,890	694,140	5,898
	Memory + Assembly	3,832	604,791	611,965	897,616	7,508

an SRAM budget of tens of kilobytes, while all three operations are accelerated. These results show that memory restructuring and assembly acceleration can be combined effectively even in a mixed-radix lattice KEM.

The proposed optimizations, however, increase the code size by 67–72% due to recomputation and added assembly. This is an acceptable trade-off on embedded targets where flash is less constrained than SRAM. In addition, the assembly acceleration relies on the DSP extensions of ARMv7E-M and is therefore confined to the Cortex-M4 family as is, whereas the memory optimization is implemented as an algorithm-level restructuring in C and can be ported across platforms. Future work includes further cycle reduction using Plantard arithmetic [23] or incomplete NTT with batch inversion, and extension to other platforms and KpqC lattice schemes.

References

1. KpqC-CryptoCraft, "KpqClean_ver2: Benchmark on korean post-quantum cryptography, version 2." GitHub repository, 2024. https://github.com/kpqc-cryptocraft/KpqClean_ver2.
2. D. Moody, R. Perlner, A. Regenscheid, A. Robinson, and D. Cooper, "Transition to post-quantum cryptography standards," Tech. Rep. NIST Internal Report (IR) 8547 (Initial Public Draft), U.S. Department of Commerce, NIST, 2024.
3. Korean Post-Quantum Cryptography (KpqC), "KpqC: Korean post-quantum cryptography competition." https://kpqc.or.kr. Accessed: 2026-06-08.
4. L. Botros, M. J. Kannwischer, and P. Schwabe, "Memory-efficient high-speed implementation of kyber on cortex-m4," in *Progress in Cryptology — AFRICACRYPT 2019*, vol. 11627 of *Lecture Notes in Computer Science*, pp. 209–228, Springer, 2019.

5. D. O. C. Greconici, M. J. Kannwischer, and A. Sprenkels, "Compact dilithium implementations on cortex-M3 and cortex-M4," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2021, no. 1, pp. 1–24, 2021.
6. E. Alkim, Y. A. Bilgin, M. Cenk, and F. Gérard, "Cortex-M4 optimizations for {R,M}lwe schemes," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2020, no. 3, pp. 336–357, 2020.
7. J. Kim and J. H. Park, "NTRU+: Compact construction of NTRU using simple encoding method," *IEEE Transactions on Information Forensics and Security*, vol. 18, pp. 4760–4774, 2023. https://eprint.iacr.org/2022/1664.
8. J. H. Cheon, H. Choe, D. Hong, J. Hong, H. Seong, J. Shin, and M. Yi, "SMAUG-T: The key exchange algorithm based on Module-LWE and Module-LWR," tech. rep., KpqC Competition, 2023. Round-2 specification, Version 2.0.
9. S. Kim, J. Ha, M. Son, B. Lee, D. Kwon, M. Jang, H. Lee, H. Chung, H. Cho, H. Yoon, J. Cho, and J. Lee, "The AIMer signature scheme," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, ACM, 2023.
10. J. H. Cheon, H. Choe, J. Devevey, T. Güneysu, D. Hong, M. Krausz, G. Land, M. Möller, D. Stehlé, and M. Yi, "HAETAE: Shorter lattice-based Fiat–Shamir signatures," tech. rep., KpqC Competition, 2023. Round-2 specification, Version 2.0.
11. E. Alkim, V. Hwang, and B.-Y. Yang, "Multi-parameter support with NTTs for NTRU and NTRU prime on cortex-M4," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2022, no. 4, pp. 349–371, 2022. https://eprint.iacr.org/2022/930.
12. E. Fujisaki and T. Okamoto, "Secure integration of asymmetric and symmetric encryption schemes," in *Advances in Cryptology — CRYPTO '99*, vol. 1666 of *Lecture Notes in Computer Science*, pp. 537–554, Springer, 1999.
13. J. Kim and J. H. Park, "NTRU+: KpqC round 2 algorithm specification." Korean Post-Quantum Cryptography (KpqC) Competition, https://kpqc.or.kr/images/pdf/NTRU+.pdf, 2024. Accessed: 2026-06-08.
14. J. W. Bos, J. Renes, and A. Sprenkels, "Dilithium for memory constrained devices," in *Progress in Cryptology — AFRICACRYPT 2022*, vol. 13503 of *Lecture Notes in Computer Science*, pp. 217–235, Springer, 2022.
15. R. Hou, Y. Gao, Y. Liu, J. Ming, and Y. Zhou, "MEML-KEM: A memory-efficient implementation of ML-KEM for IoT devices," in *Algorithms and Architectures for Parallel Processing — ICA3PP 2025*, vol. 16387 of *Lecture Notes in Computer Science*, Springer, 2025.
16. Arm Limited, *Cortex-M4 Technical Reference Manual*. Arm Limited, ddi 0439 ed. https://developer.arm.com/documentation/ddi0439/latest/.
17. Arm Limited, *ARMv7-M Architecture Reference Manual*. Arm Limited, ddi 0403 ed. https://developer.arm.com/documentation/ddi0403/latest/.
18. M. J. Kannwischer, J. Rijneveld, P. Schwabe, and K. Stoffelen, "pqm4: Testing and benchmarking NIST PQC on ARM cortex-M4." Cryptology ePrint Archive, Paper 2019/844; Second NIST PQC Standardization Conference, 2019. https://github.com/mupq/pqm4.
19. C. Rackoff and D. R. Simon, "Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack," in *Advances in Cryptology — CRYPTO '91*, vol. 576 of *Lecture Notes in Computer Science*, pp. 433–444, Springer, 1991.
20. M. Bellare, A. Desai, D. Pointcheval, and P. Rogaway, "Relations among notions of security for public-key encryption schemes," in *Advances in Cryptology*

— *CRYPTO '98*, vol. 1462 of *Lecture Notes in Computer Science*, pp. 26–45, Springer, 1998.
21. A. Abdulrahman, V. Hwang, M. J. Kannwischer, and A. Sprenkels, "Faster kyber and dilithium on the cortex-M4," in *Applied Cryptography and Network Security — ACNS 2022*, vol. 13269 of *Lecture Notes in Computer Science*, pp. 853–871, Springer, 2022.
22. G. Seiler, "Faster AVX2 optimized NTT multiplication for ring-LWE lattice cryptography." Cryptology ePrint Archive, Paper 2018/039, 2018. https://eprint.iacr.org/2018/039.
23. T. Plantard, "Efficient word size modular arithmetic," *IEEE Transactions on Emerging Topics in Computing*, vol. 9, no. 3, pp. 1506–1518, 2021.