Efficient NTRU+ Implementations on Cortex-M

Subeen Cho[1], Minjoo Sim[1], Seungwon Lee[1], Anupam Chattopadhyay[2], Wai Kong Lee[3], and Hwajeong Seo[1]

[1] Department of Convergence Security,
Hansung University, Seoul (02876), South Korea
{chosubin1208, minjoos9797, dkajdfhd1, hwajeong84}@gmail.com
[2] School of Computer Science and Engineering,
Nanyang Technological University, Singapore
anupam@ntu.edu.sg
[3] Universiti Tunku Abdul Rahman, Malaysia
wklee@utar.edu.my

Abstract. Post-quantum cryptography on embedded devices requires implementations that perform cryptographic operations efficiently within limited computation and memory. Most lattice-based KEMs use 2-way NTTs over power-of-two cyclotomic rings, and their embedded implementations have been studied extensively. In contrast, implementations of schemes based on 3-way NTTs, which combine radix-3 and radix-2 layers, remain scarce. This work therefore presents efficient implementations of NTRU+, a KEM built on 3-way NTTs, on Arm Cortex-M0+, M3, M4, and M7. We configure the layer fusion, modular arithmetic, and instruction scheduling of the 3-way NTT for each parameter set and core. We also introduce ω-folded front layers, which compute the first two layers of the forward NTT on ternary inputs without modular reduction. Based on these techniques, we present two implementations. The speed-optimized implementation keeps operands as full polynomial arrays to prioritize execution time. The memory-optimized implementation reuses a single polynomial workspace and regenerates the required operands in pieces aligned with radix-3 blocks. On physical boards, the speed-optimized implementation reduces execution time by up to 52.43% compared with the official optimized C implementation. The memory-optimized implementation reduces peak stack usage by up to 82.60% while running faster than the official optimized C implementation in every KEM operation. A controlled NTT-replacement experiment and per-technique comparisons show that the same technique can have different effects depending on the instruction set and pipeline of each core.

Keywords: NTRU+ · 3-way NTT · Cortex-M · time-memory trade-off

1 Introduction

The emergence of quantum computers threatens the hardness assumptions on which widely used public-key cryptography relies [38]. This threat is accelerating the transition to post-quantum cryptography (PQC), which resists both

classical and quantum attacks. The transition is particularly important for key encapsulation mechanisms (KEMs), which establish shared secrets.

This transition is also needed on embedded devices with limited computation and memory. Post-quantum KEMs are often based on lattice problems, and their performance then depends heavily on the number-theoretic transform (NTT) used for polynomial multiplication. Most lattice-based KEMs, including ML-KEM standardized by NIST [34], use 2-way NTTs over power-of-two cyclotomic rings. Embedded implementation studies have therefore focused on layer fusion, modular arithmetic, and instruction scheduling for 2-way NTTs [20,5,2].

Other lattice-based KEMs use NTTs over rings whose dimension is not a power of two. Examples include NTTRU [29], CTRU and CNTR [27], and NTRU+ [25,37]. These rings allow more flexible parameter choices, but the factor of three in the dimension must be handled within the transform. NTTRU and CTRU therefore move this factor into base multiplication without radix-3 layers, whereas NTRU+ uses a 3-way NTT that combines radix-3 and radix-2 layers. The layer structure and final base blocks of such a 3-way NTT change across parameter sets, so techniques developed for 2-way NTTs do not carry over directly. For example, every layer of a 2-way NTT halves the coefficient distance, so the same kernel can be repeated. In NTRU+, however, the number and distances of the radix-3 layers and the size of the base blocks differ across parameter sets, so the boundaries of fused layers and the base multiplication must be chosen anew for each parameter set. Embedded implementation studies of such transforms remain scarce compared with those of 2-way NTTs.

NTRU+ is a lattice-based KEM that uses such a 3-way NTT and was selected in the Korean post-quantum cryptography competition KpqC [26]. It is based on the NTRU problem and uses SOTP, which encodes a message as a ternary polynomial using randomness [25,37]. Its ciphertext is a single polynomial, and its validity is checked without re-encryption [25]. This design keeps the ciphertext and the computation simple, which suits practical systems that must balance communication and computation costs. The three parameter sets of NTRU+ also differ in their radix-3 layers and base-block structures, which makes NTRU+ a suitable target for studying the difficulties of implementing 3-way NTTs.

Implementing a 3-way NTT-based KEM efficiently on embedded devices raises several issues. In 3-way NTTs, intermediate coefficients are repeatedly stored to memory and reloaded between transform layers. Fusing layers reduces these accesses, but it also increases the number of values that must stay in registers and can cause additional memory accesses [5]. A faster NTT alone also does not make the complete KEM sufficiently faster. As the transform becomes faster, the remaining operations and the way intermediate polynomials are stored dominate the execution time and memory usage. The official NTRU+ implementation provides optimized C code [36]. However, this portable C code does not target a specific core and therefore does not exploit the registers and instructions of Cortex-M cores. Its computation and dataflow must thus be reorganized for the limited registers and memory of Cortex-M devices.

These optimizations must also reflect the hardware of each Cortex-M core. Because the parameter sets of NTRU+ also differ in coefficient layout and base-block structure, a single arithmetic approach is unlikely to be efficient for every core and parameter set. A prior study implemented NTRU+ on Cortex-M4 [12], but it targets the previous revision and only M4, so implementations for other cores and for the latest revision have not yet been addressed.

This work therefore optimizes the 3-way NTT and the complete KEM dataflow of NTRU+ on Arm Cortex-M0+, M3, M4, and M7, four cores widely used in embedded systems. Starting from the official optimized C implementation [36], we present a speed-optimized implementation, which keeps operands as full polynomial arrays to prioritize execution time, and a memory-optimized implementation, which reuses one workspace to reduce peak stack usage. We also measure execution time and memory usage on physical boards and analyze the effect of the optimizations on each core and the resulting trade-offs between time and memory.

1.1 Contributions

The main contributions of this work are as follows.

- **Core-Specific 3-Way NTT Optimization.** We configure layer fusion, modular arithmetic, and instruction scheduling for each core according to the radix-3 and radix-2 layer structures of the three parameter sets. In a controlled experiment that replaces only the NTT, these optimizations reduce the execution time of the forward and inverse NTTs by up to 80.60% and 67.15%, respectively, relative to the reference implementation.
- **Techniques Specific to the 3-Way NTT.** We propose ω-folded front layers, which compute the first two layers of the forward NTT without modular reduction, and piece regeneration, which recomputes operands in pieces aligned with radix-3 block boundaries. Both techniques follow from conditions independent of the instruction set. Because the pieces are aligned with radix-3 blocks, each piece can use the core-specific kernels, which reduces the Decaps time of the memory-optimized implementation by 5.07% to 17.91% without increasing the stack.
- **Complete KEM Dataflow and Memory Optimization.** We reorganize the storage and reuse of intermediate data while producing the same outputs as the official implementation on the official known-answer tests and all measurement inputs. The speed-optimized implementation reduces execution time by up to 52.43% relative to the official optimized C implementation. The memory-optimized implementation reduces peak stack usage by up to 82.60% while remaining faster than the official optimized C implementation in every KEM operation. We also analyze the SRAM reservation within the same executable and the code size on physical boards.

Section 2 reviews the background and related work, Section 3 describes the implementation design, Section 4 presents the evaluation, and Section 5 concludes the paper.

2 Background

This section summarizes the structure and operations of NTRU+, the 3-way NTT, modular arithmetic, and the characteristics of the target Cortex-M cores, and then reviews related implementations.

2.1 NTRU+

NTRU+ [25] is a lattice-based key encapsulation mechanism (KEM) based on the NTRU problem. It combines a cyclotomic trinomial ring suitable for the NTT with the semi-generalized one-time pad (SOTP), which encodes a message as a ternary polynomial using randomness. A ciphertext is a single polynomial, and decapsulation validates it by checking the recovered randomness without re-encryption [37]. The operations of NTRU+ are based on the following ring [37].

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

Here, N denotes the number of polynomial coefficients, and q is the coefficient modulus. Each coefficient is represented modulo q, and polynomial products are reduced modulo $X^N - X^{N/2} + 1$. This ring structure allows polynomial multiplication to be decomposed into smaller polynomial operations in the NTT domain, providing the basis for the transform and inversion operations.

NTRU+ supports three parameter sets with $N = 768, 864, 1152$. NTRU+768 uses one radix-3 layer and yields 192 four-coefficient base blocks, whereas NTRU+864 and NTRU+1152 use two radix-3 layers and yield 288 base blocks with three and four coefficients, respectively [35]. The layer structure, coefficient distances, and base multiplication and inversion of the NTT therefore differ across parameter sets. Table 1 summarizes the parameter-specific transform structures and storage sizes.

Table 1. Transform structure and storage sizes of NTRU+ [37,36]. Distances are the coefficient distances of the corresponding transform layers. Storage sizes are object sizes and exclude function-frame stack usage.

Item	NTRU+768	NTRU+864	NTRU+1152
Factorization of N	$3 \cdot 2^8$	$3^3 \cdot 2^5$	$3^2 \cdot 2^7$
Radix-3 distances	128	144, 48	192, 64
Radix-2 tail	64, 32, 16, 8, 4	24, 12, 6, 3	32, 16, 8, 4
Base degree / blocks	4 / 192	3 / 288	4 / 288
16-bit polynomial	1536 B	1728 B	2304 B
Serialized poly. / pk / ct	1152 B	1296 B	1728 B
2-bit ternary state	192 B	216 B	288 B
Secret key	2336 B	2624 B	3488 B

The coefficient representation before the transform is called the coefficient domain, and the transformed representation is called the NTT domain. The

public key and ciphertext each consist of one NTT-domain polynomial serialized with 12 bits per coefficient [36], so each occupies $P = 3N/2$ bytes. In contrast, a 16-bit coefficient array used for computation requires $2N$ bytes, and a two-bit representation of a ternary polynomial requires $N/4$ bytes. These differences between representations determine the memory usage of the KEM.

NTRU+ consists of key generation (KeyGen), encapsulation (Encaps), and decapsulation (Decaps). KeyGen samples small polynomials f' and g' from the centered binomial distribution (CBD), constructs $f = 1 + 3f'$ and $g = 3g'$, and resamples a polynomial only if its inverse does not exist in the NTT domain. It then computes $h = gf^{-1}$ and $h^{-1} = fg^{-1}$ [36]. The public key stores h, and the secret key stores the NTT-domain representations of f and h^{-1} together with the public-key hash.

Encaps validates the public-key encoding and uses three SHAKE256-based hash functions F, G, and H [37,33] to derive the public-key hash, the SOTP-encoding values, and the values for the shared secret and polynomial generation. It samples a small polynomial r, encodes the message as a ternary polynomial m using the NTT-domain representation of r, and computes the ciphertext as $\hat{c} = \hat{h} \circ \hat{r} + \hat{m}$. Here, $\hat{\cdot}$ denotes the NTT-domain representation, and $\circ$ denotes multiplication within the base blocks.

Decaps deserializes the ciphertext and secret-key polynomials and validates their coefficients. It multiplies the ciphertext by f, applies the inverse NTT, and reduces the result modulo 3 to obtain a candidate message polynomial m_1. It then computes the candidate randomness $\hat{r}_2 = (\hat{c} - \mathrm{NTT}(m_1)) \circ \widehat{h^{-1}}$ [36], recovers the message by SOTP decoding with $r_2 = \mathrm{NTT}^{-1}(\hat{r}_2)$, regenerates r_1, and compares the serialized representations of the two polynomials. On verification failure, the shared secret is set to zero. Since all three operations repeatedly perform forward and inverse NTTs and base multiplications, the efficiency of the NTT determines the performance of the complete KEM.

2.2 3-Way NTT

NTRU+ decomposes polynomial multiplication into smaller polynomial operations through the NTT. It uses an incomplete NTT [35] that leaves the last few layers unevaluated, so multiplication and inversion are performed on the small polynomials that remain after the transform, called base blocks.

Figure 1 shows the structures of the radix-2 CT butterfly and the radix-3 butterfly, together with the NTT layer sequences of ML-KEM, a 2-way NTT that uses only radix-2 layers, and the three NTRU+ parameter sets, 3-way NTTs that include radix-3 layers.

Each NTT layer combines coefficients at a specified distance through butterfly operations. Most lattice-based KEMs use 2-way NTTs that consist only of radix-2 layers. For example, ML-KEM [34] applies seven radix-2 layers over $\mathbb{Z}_q[X]/(X^{256} + 1)$, halving the coefficient distance from 128 to 2, and leaves 128 two-coefficient base blocks (top row of Figure 1(c)). In contrast, a 3-way NTT is a mixed-radix NTT that combines radix-3 and radix-2 layers. It also supports

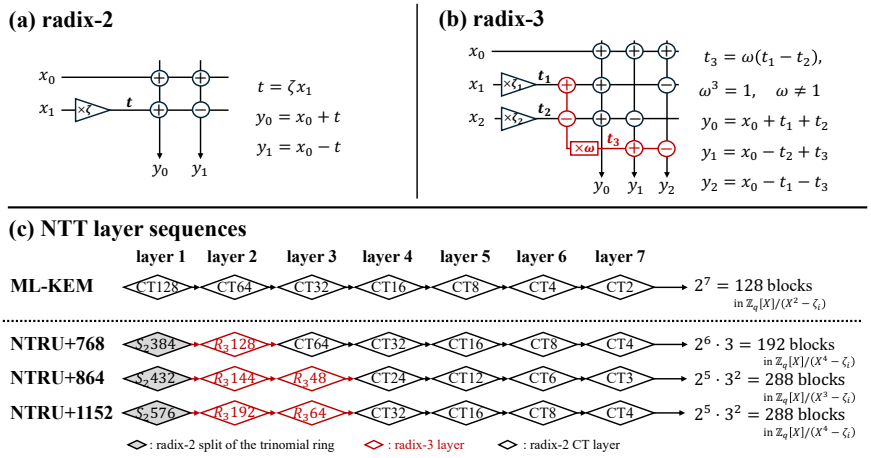

Fig. 1. (a) Radix-2 CT butterfly. (b) Radix-3 butterfly. In (a) and (b), circled signs add or subtract the row value into the output column. (c) NTT layer sequences of ML-KEM and the three NTRU+ parameter sets, with the coefficient distance after each layer name and the base blocks after seven layers on the right.

rings whose dimension is not a power of two, but its layer distances and base blocks vary across parameter sets.

A radix-2 butterfly combines two coefficients a_i and a_{i+s} at distance s with a twiddle factor ζ to produce two outputs. Depending on when ζ is applied, it takes the Cooley-Tukey (CT) [14] or Gentleman-Sande (GS) [15] form. CT multiplies one input by ζ before computing the sum and difference, whereas GS computes the sum and difference first and then multiplies the difference by ζ (Figure 1(a)). A radix-3 butterfly takes three coefficients x_0, x_1, and x_2 at distance s, computes $t_1 = \zeta_1 x_1$, $t_2 = \zeta_2 x_2$, and $t_3 = \omega(t_1 - t_2)$, and outputs $y_0 = x_0 + t_1 + t_2$, $y_1 = x_0 - t_2 + t_3$, and $y_2 = x_0 - t_1 - t_3$ [35] (Figure 1(b)). Here, ω is a primitive cube root of unity, which splits a polynomial into three factors and distinguishes the three outputs.

From an implementation perspective, the 3-way NTT differs from the 2-way NTT in three ways. First, a radix-2 CT butterfly requires one multiplication and two additions or subtractions, whereas a radix-3 butterfly requires three multiplications and seven additions or subtractions, which increases the cost per butterfly. Handling three inputs and the multiplication by ω together also keeps more values live at once and increases register pressure. Second, the layer distances do not halve uniformly, so the same kernel cannot be repeated for every layer, and the boundaries for fusing adjacent layers must be chosen again for each parameter set. Third, base blocks contain three or four coefficients instead of two, so base multiplication and base inversion are computed differently.

NTRU+ leaves base blocks of d coefficients and thus decomposes a polynomial into N/d blocks [36]. For the three parameter sets, this number is $768/4 = 3 \cdot 2^6$,

$864/3 = 3^2 \cdot 2^5$, and $1152/4 = 3^2 \cdot 2^5$, and each contains a factor of three. As shown in Figure 1(c), the forward NTT of NTRU+ is therefore a 3-way NTT that starts with a radix-2 split of the trinomial ring, proceeds through one or two parameter-specific radix-3 layers [35], and then applies the remaining radix-2 layers. The final base blocks have the form $X^4 - \zeta$ for NTRU+768 and NTRU+1152 and $X^3 - \zeta$ for NTRU+864.

2.3 Modular Arithmetic

NTT butterflies and base-block operations repeatedly involve coefficient multiplications and multiplications by constants such as twiddle factors. However, division-based modular reduction is costly on microcontrollers, so methods that replace division with multiplication and bitwise operations are needed. Montgomery reduction [30] is a representative method that uses a power of two to perform modular reduction without division. NTRU+ uses $q = 3457$ and $\beta = 2^{16}$ [36], and signed Montgomery reduction is defined as follows.

$$t = \mathrm{signed}_{16}(uq^{-1}), \qquad \mathrm{MRed}(u) = \frac{u - tq}{\beta} \equiv u\beta^{-1} \pmod{q}. \tag{2}$$

Here, signed_{16} interprets the lower 16 bits as a signed integer, and $q^{-1} \bmod \beta = 12929$ is a precomputed constant. The output of Montgomery reduction is congruent to the input multiplied by β^{-1} modulo q. Thus, the arithmetic scales of intermediate values and precomputed constants must be maintained consistently across consecutive NTT butterflies and base-block operations.

In addition, since NTT twiddle factors are predetermined constants, Plantard arithmetic [17] can be used for their repeated multiplication. Plantard arithmetic precomputes values associated with the modulus and multiplication constants and uses integer multiplication and bitwise operations to perform modular multiplication. In particular, improved Plantard arithmetic extends its applicability to signed inputs and reduces final correction operations when the permitted input ranges are satisfied.

Not only multiplications but also the additions and subtractions of butterflies require reduction, because sums and differences increase the magnitudes of intermediate coefficients. Barrett reduction [10] precomputes an approximation to the reciprocal of q, approximates the quotient t of an input x, and computes $x - tq$, thereby reducing intermediate coefficients to a small range without division. Since the resulting range depends on the input range, the permitted ranges of subsequent operations must be considered.

Moreover, the cost of modular arithmetic depends not only on the reduction method but also on when reductions are performed. Lazy reduction [5] does not reduce the result of every butterfly immediately. Instead, it postpones reduction while intermediate values remain within both the input range of the next multiplication and the range of 16-bit signed integers, and reduces them once when needed, which decreases the number of reductions. Combined with the fusion of adjacent NTT layers, it can reduce both intermediate memory accesses and some

reductions. However, successive operations increase intermediate magnitudes, so the input range of the next multiplication and the representable range of the integer type must be preserved. Since reductions or subtractions are repeated for every butterfly output in each NTT layer, these operations account for a large part of the per-layer cost.

2.4 Cortex-M Cores and Implementation Constraints

Cortex-M cores differ in their instructions and execution architectures, so even the same NTT operation incurs different costs for modular arithmetic, register use, and memory access. This work targets Cortex-M0+, M3, M4, and M7.

Cortex-M0+ uses the restricted Thumb instruction set of Armv6-M, and some arithmetic and addressing instructions can access only the low registers [9]. Its costs are therefore dominated by instruction count and repeated memory accesses. Cortex-M3 supports Armv7-M Thumb-2 instructions but lacks DSP extensions [2], so polynomial arithmetic is built from scalar instructions, and instruction dependencies and register reuse become important. Cortex-M4 provides the Armv7E-M DSP extensions, which allow packed-halfword operations on two 16-bit coefficients held in one 32-bit register [28]. These operations process the addition, subtraction, or multiplication of two coefficients with a single instruction and thus reduce the instruction count, but coefficient rearrangement or additional memory accesses can offset this benefit [5]. Cortex-M7 supports the same DSP extensions as M4 but differs in that it has a dual-issue pipeline and caches [28,3]. Independent butterflies must therefore be interleaved to reduce dependency stalls [3]. As a result, a layer-fusion or modular-arithmetic technique that is effective on one core is not guaranteed to perform equally well on another, and implementations must account for both the parameter-specific transform structure and the instruction and memory constraints of each core.

2.5 Related Implementations

Embedded implementations of lattice-based KEMs have mainly optimized NTT arithmetic and transform structure. Huang et al. proposed improved Plantard arithmetic [17], and Becker et al. optimized polynomial multiplication on embedded vector architectures [11]. These techniques form a common basis for NTT implementations. For 3-way NTTs, NTTRU [29] and CTRU [27] avoid radix-3 layers and move the factor 3 into base or sub-polynomial multiplication. For implementations that keep radix-3 layers, Alkim et al. compared mixed-radix NTTs that use smaller multipliers and less memory [6], Alkim, Hwang, and Yang fused radix-2 and radix-3 layers [7], and Chung et al. grouped two radix-3 layers into a size-9 NTT [13]. However, these techniques are designed for transforms with a fixed NTT size and fixed positions of the radix-3 layers, so the fusion units and reduction points must be redetermined when the layer structure changes.

For core-specific optimization, Alkim et al. optimized NewHope and Kyber on Cortex-M4 [5], Abdulrahman et al. optimized Saber with multi-moduli NTTs on Cortex-M3 and M4 [2], and ML-KEM and ML-DSA were optimized on Cortex-M7

with SLOTHY [3]. These studies all target schemes based on 2-way NTTs, and schemes based on 3-way NTTs have rarely been compared across several cores. For NTRU+ optimization, the official C, AVX2, and NEON implementations [36] and a faster inversion for key generation [23] are available. Cho et al. presented an optimized implementation on a Cortex-M board [12], but it targets the previous revision and is limited to M4.

3 NTRU+ Implementation Design on Cortex-M

This section describes how the 3-way NTT and the KEM dataflow of NTRU+ are implemented on Cortex-M cores. The speed-optimized and memory-optimized implementations share the same NTT, base-multiplication, and hash kernels but have different goals. The speed-optimized implementation keeps operands as full polynomial arrays to reduce the execution time of complete KEM operations. The memory-optimized implementation manages the lifetime of intermediates to reduce peak stack usage while remaining faster than the official optimized C implementation. Table 2 divides the techniques into those independent of the instruction set and those tailored to the instructions or execution structure of a specific core. It also marks the techniques specific to the 3-way NTT and the cores on which each technique is applied. Techniques applied to all cores, such as lazy reduction and the fusion of Barrett reduction into the last CT layer, are omitted from the table.

Table 2. Techniques applied on each core. • means applied, and a list of parameter sets means applied only to those sets. ○ means not adopted because it gave no meaningful benefit for the complete KEM. − means not applicable because the required instructions or execution structure, such as DSP extensions or dual issue, are absent, or not needed because the Flash memory has no wait states. "Used in" indicates whether the technique is used in both implementations or only in the memory-optimized implementation.

Technique	Used in	3-way	M0+	M3	M4	M7
Independent of the instruction set						
Layer fusion	Both	✓	○	•	•	•
ω-folded front layers	Both	✓	•	768, 864	•	•
Workspace reuse, batch inversion	Memory		•	•	•	•
Radix-3 block piece regeneration	Memory	✓	•	•	•	•
Core-dependent						
Modular multiplication	Both		Montgomery	Scalar Plantard	Packed Plantard	Packed Plantard
Packed-halfword arithmetic	Both		−	−	•	•
Four-multiplication radix-3 butterfly	Both	✓	−	−	○	○
SLOTHY dual-issue scheduling	Both	✓	−	−	−	•
Code execution from SRAM	Both		•	−	−	−

3.1 3-Way NTT Layer Fusion

Each NTT layer consumes the coefficients produced by the previous layer. Executing the layers separately therefore requires intermediate coefficients to be stored and reloaded, so our implementation applies layer fusion, which computes adjacent layers consecutively [5,13]. However, keeping all coefficients and twiddle factors of the fused layers in registers can exhaust the register set and spill intermediates to memory, which offsets the benefit of fusion. We therefore choose fusion boundaries according to the coefficients consumed together by the next layer and the number of available registers.

Parameter-specific fusion boundaries. Fusion boundaries differ between parameter sets. In NTRU+768, the radix-3 layer is followed by CT64 and CT32, so computing the outputs needed by both CT layers consecutively reduces intermediate memory accesses. The three outputs of each radix-3 butterfly are consumed at different points in the following CT computation. Some outputs used immediately by CT64 are therefore kept in registers, while those used later are selectively stored and combined in CT32. Such a small group of coefficients processed consecutively is called a tile, and its size is chosen from the last use of each value and the number of available registers. In NTRU+864, the two radix-3 layers have coefficient distances of 144 and 48 and the base blocks have three coefficients, so the four-coefficient processing order of NTRU+768 cannot be reused. In NTRU+1152, the two radix-3 layers have distances of 192 and 64, so the outputs of the earlier layer stay live longer, and the fusion depth is chosen by separating values needed immediately by the next CT layer from those needed later. Continuing without storing intermediates increases coefficient magnitudes. We therefore check that the input range of the next multiplication and the signed 16-bit range are not exceeded, and pass coefficients to the next layer without reduction only where both hold.

ω-folded front layers. If the input coefficients of the forward NTT are ternary, the first two layers can be computed without modular reduction while intermediates stay within the 16-bit range. Every polynomial entering the forward NTT inside the KEM satisfies this condition. The polynomials r and m in Encaps and Decaps are ternary with coefficients in $\{-1, 0, 1\}$, and f and g in KeyGen have the forms $3t + 1$ and $3t$ for a ternary polynomial t. Using this property, we design ω-folded front layers that compute the first two layers of the forward NTT, namely the trinomial splitting and the first radix-3 layer, without reduction. They differ from skipping reductions in the first layers for small inputs [13] and from replacing some Montgomery multiplications with unreduced multiplications [22] in that the ω products inside the radix-3 butterfly are also folded into precomputed constants. This is possible because, in the NTRU+ ring, the root ζ_6 of the trinomial splitting and the primitive cube root of unity ω of the radix-3 layer satisfy $\zeta_6 = -\omega^2$. For ternary inputs, each output of the first splitting can be written as $u + Zv$ with small integers $|u| \leq 2$ and $|v| \leq 1$ and a fixed representative Z of ζ_6, which has the form of an Eisenstein integer [19]. The twiddle product $t(u + Zv)$ of the next

radix-3 layer then becomes $tu + (tZ \bmod q)v$, and since 18 divides $q - 1$, the ω products inside the butterfly are also absorbed into precomputed constants. The first two layers are therefore computed only with multiply-accumulate operations on small integers and precomputed constants. The factor of three in KeyGen is absorbed into the constants, and the term +1 is handled by adding NTT(1) to the output. Without reduction, the front-layer outputs are larger than those of the official implementation. However, even after they grow through the following layers up to the existing reduction points, they reach at most 26,430, which is below the largest signed 16-bit value of 32,767. Since no reduction has to be added elsewhere, all instructions spent on modular reduction in the first two layers are removed. The front layers are called through a separate entry point only at the KEM call sites where the input condition holds, and all other calls use the general forward NTT. Because the method depends only on the root of the first splitting, it is not limited to NTRU+. It applies to other rings whose first splitting root lies in $\mathbb{Z}_q$ and whose NTT inputs are small, provided that the resulting bound fits the integer type.[4] The front layers are also implemented with the multiply instructions of each core, as described in Section 3.2.

3.2 Core-Specific Kernels

Layer fusion is mapped to instructions differently on each core. Figure 2 shows how the forward 3-way NTT of NTRU+768 is implemented on the four cores. In the figure, the number after CT is the coefficient distance between butterfly inputs, A, B, and C are the three outputs of a radix-3 butterfly, and red paths mark where intermediates stay in registers or where consecutive butterflies and reductions are combined.

Most of the time of NTRU+ KEM operations is spent in the NTT and in SHAKE256 hashing. Both are independent of the core at the algorithm level, but the most efficient arithmetic, fusion depth, and instruction order depend on the instruction set, the number of registers, and the pipeline, so we optimize them separately for each core. On all cores, the implementation also ensures that no branch, memory address, or operand of a variable-latency instruction depends on secret values. We checked this property by comparing the instruction and memory-access traces of each KEM operation over different inputs, including rejected ciphertexts, in an instruction-level emulator. On M0+, M3, and M4, Decaps also took the same number of cycles for all 100 measured inputs.

For hashing, all cores keep the Keccak state in the bit-interleaved representation suited to 32-bit cores throughout the sponge [4,16], which removes the conversion cost at every absorption and squeeze. The copying of hash inputs into a separate buffer is also removed. For the Keccak permutation itself, we reuse existing assembly without changing its instructions. M0+ uses the Armv6-M implementation of XKCP [21], M3 and M4 use the bit-interleaved Armv7-M

[4] If the next layer is radix-3 and 18 divides $q - 1$, the whole butterfly can be folded as in NTRU+. If it is radix-2, only the twiddle product is folded. For example, the bound stays within the 16-bit range for CTRU, which uses the same $q = 3457$.

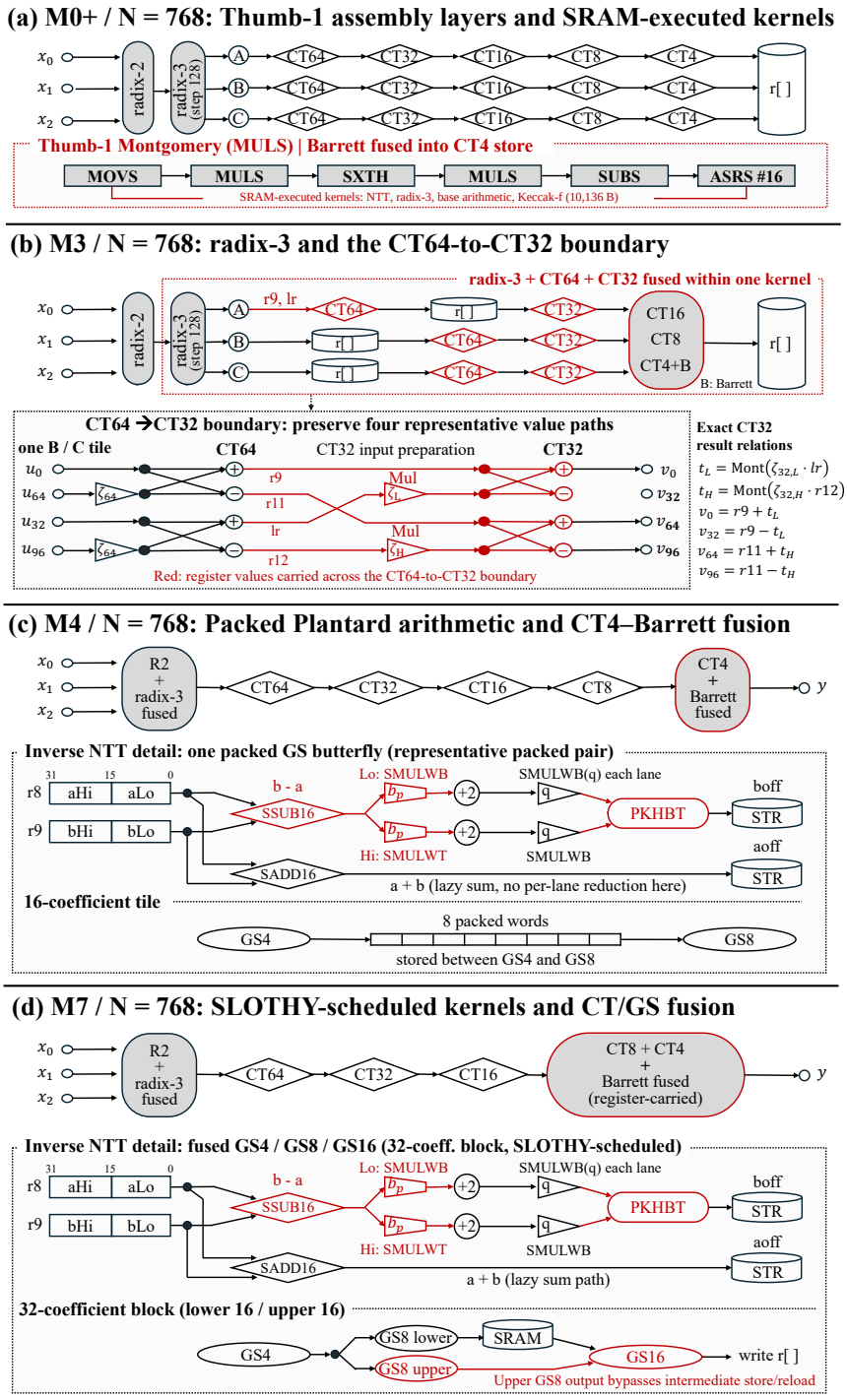

Fig. 2. Forward 3-way NTT dataflow for NTRU+768 on Cortex-M0+, M3, M4, and M7. (a) M0+ executes Thumb-1 kernels from SRAM. (b) M3 fuses the radix-3, CT64, and CT32 layers. (c) M4 uses packed Plantard arithmetic and CT4-Barrett fusion. (d) M7 uses SLOTHY scheduling and CT8-CT4-Barrett fusion. Panels (c) and (d) also show inverse NTT details.

implementation of XKCP with the optimizations of [4], and M7 uses the version of the latter rescheduled with SLOTHY and distributed with it [39].

Cortex-M0+. On M0+, the NTT and the base inversion in KeyGen are written in Thumb-1 assembly, with a focus on the instruction count of multiplication and modular reduction. Montgomery and Barrett reductions are built from Armv6-M multiply, subtract, and arithmetic-shift instructions and inlined into the NTT, and the coefficient distance of each layer is fixed at compile time to simplify address calculation. The ω-folded front layers compute two 16-bit values with one 32-bit multiplication, and their benefit was largest on M0+, where the instruction count dominates the cost. Layer fusion is not applied on M0+ because, with only eight low registers, reloading constants and offsets costs more than the saved memory passes. In addition, our M0+ board runs its Flash with two wait states and prefetch disabled, so every instruction fetch from Flash incurs a wait and frequently executed code is placed in SRAM. Only the functions that are actually executed are placed there, because placing the whole Keccak file, including unused functions, added branches between Flash and SRAM code and contention on the SRAM bus, which slowed execution. The speed-optimized implementation therefore executes from SRAM only the NTT, base-arithmetic, serialization, and Keccak functions that are actually used, which reduces instruction-fetch waits. The memory-optimized implementation executes only the Keccak functions from SRAM to limit the increase in SRAM usage.

Cortex-M3. M3 fuses the radix-3 outputs into CT64 and CT32 while limiting the intermediates held in registers to reduce memory accesses, as shown in Figure 2(b). When one operand is a fixed constant, Plantard multiplication uses a precomputed value to multiply and reduce with fewer instructions than Montgomery multiplication [17,18,40]. Selected CT layers that multiply by fixed twiddle factors therefore use scalar Plantard arithmetic, whereas general coefficient products and base arithmetic, whose operands both vary, use Montgomery arithmetic. Independent products are scheduled first to relax instruction dependencies, and the ω-folded front layers use the 32-bit-result multiply instructions MUL and MLA. On M3, the latency of multiply instructions can also depend on their operands. For constant-time execution, all modular arithmetic and base multiplication therefore use only MUL, MLA, and MLS and avoid the 64-bit-result multiplies UMULL and SMULL, which may terminate early depending on operand values [8].

Cortex-M4. M4 uses DSP instructions for packed-halfword arithmetic, which holds two 16-bit coefficients in one 32-bit register and processes their sum, difference, and per-halfword constant products together to reduce the instruction count [28]. The Montgomery scale correction after base multiplication is also performed in packed assembly, and the ω-folded front layers accumulate two products in one instruction with SMUAD and SMLAD. Packing is used only where two coefficients naturally form a pair, because rearranging coefficients adds loads and stores. This is also why scalar processing is retained at some NTRU+864 boundaries.

Cortex-M7. M7 supports the same DSP extensions as M4 but has a dual-issue pipeline, so independent operations must be interleaved to reduce dependency stalls. Lower- and upper-halfword products are therefore interleaved, and CT8 and CT4 as well as the inverse GS layers are connected to reduce intermediate memory accesses. The instruction order is determined with SLOTHY, which changes only the instruction order and register allocation while keeping the operations and dataflow [1,3]. We apply it to the NTT kernels, including the radix-3 layers, and use the SLOTHY-scheduled Keccak permutation described above for SHAKE256 [33]. Since the benefit of SLOTHY comes from hiding dependency stalls through interleaving, it is applied only to M7 and not to the single-issue M4. NTRU+864 has a layer structure not found in the other parameter sets, in which CT24 follows the second radix-3 layer, so it uses dedicated kernels for this part and for the inverse radix-3 layers. These dedicated kernels are also scheduled with SLOTHY, and the separate Barrett pass is removed.

Some techniques could be applied to several cores but are used only on some of them because their effect depends on the execution structure of each core. For example, the four-multiplication radix-3 butterfly of [7] made the kernel faster on M4, where SMUAD performs two multiplications in one instruction, but its gain in the complete KEM was below 1%, so it was not adopted. It was slower on M7, where the SLOTHY-scheduled kernel already fills the multiply-accumulate pipeline. The ω-folded front layers likewise differ in effect between cores, as Section 4.3 examines. The Keccak permutation scheduled for M7 was slower on M3 and M4, because interleaving loads and arithmetic for dual issue does not suit their single-issue pipelines. An assembly implementation of base inversion was faster on M0+, but on M4 it required more instructions than the official optimized C implementation and was not adopted. The effect of the same technique thus depends on the instruction set and pipeline of each core.

After the NTT and hashing are optimized, the remaining processing inherited from the official implementation becomes the bottleneck, so three techniques are applied to all cores. First, we remove the passes that the official optimized C implementation places before and after the NTT to adjust coefficient ranges. In the forward NTT, the last CT layer already stores coefficients after Barrett reduction. In the inverse NTT, the final Plantard output lies in $[-1728, 1728]$, and the input scale factor can be absorbed into the normalization constant of the output. Second, auxiliary functions such as base multiplication, serialization, sampling, and SOTP encoding are reimplemented for the instructions of each core. Third, the byte-wise erasure of secret intermediates in the official implementation is replaced with word-wise erasure of the same memory.

3.3 Memory-Optimized KEM Dataflow

The memory-optimized implementation reuses one polynomial workspace and the output buffers that have not yet been written, according to the last use of each value. It accepts the cost of restoring operands block by block instead of keeping several intermediate polynomials at once, which reduces peak stack usage. This dataflow is applied identically on all four cores.

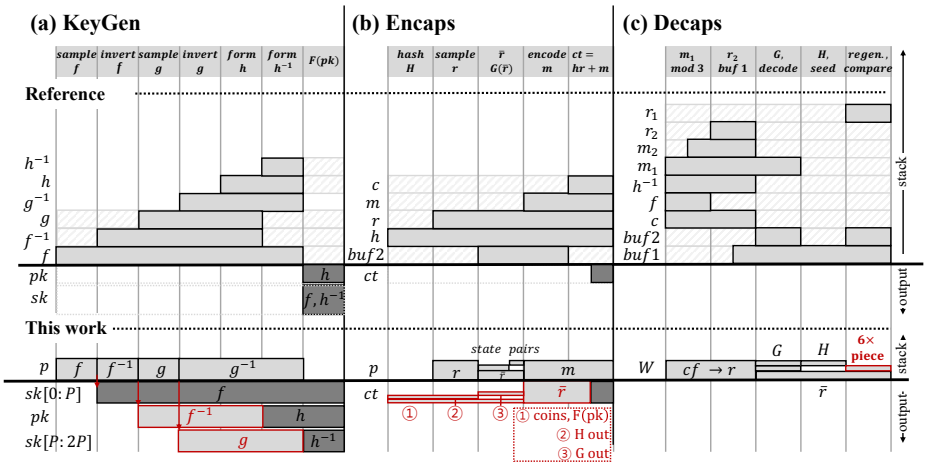

Fig. 3. Buffer usage in each step of the reference implementation (top) and the memory-optimized implementation (bottom) for NTRU+768 on M4. (a) KeyGen, (b) Encaps, (c) Decaps. Bar lengths are proportional to bytes. Hatched areas hold values not needed in that step, and red boxes mark areas reused to store other values.

Figure 3 compares the buffer usage of the reference implementation and the memory-optimized implementation for NTRU+768 on M4, using the same steps. The reference implementation declares a separate stack array for each intermediate polynomial, so arrays occupy space even in steps where their values are no longer needed (hatched). The memory-optimized implementation uses one 16-bit polynomial workspace p and the output buffers pk, sk, and ct to store different values at each step (red boxes).

KeyGen. After f is transformed to the NTT domain, it is serialized to its final location in sk, and the inverse of f is stored temporarily in pk, which has not yet been written. The workspace p is then reused for g, and the transformed g is stored temporarily in the area of sk that will later hold h^{-1}. The products $h = g \cdot f^{-1}$ and $h^{-1} = f \cdot g^{-1}$ are computed by restoring base blocks from the serialized operands, and each temporary value is overwritten only after its last read. If an inverse does not exist, only the failed polynomial is resampled. Base-block inverses use batched inversion as in the official optimized C implementation [23,24], but the base blocks are processed in groups of 24 rather than all at once to reduce the workspace for prefix products and numerators.

Encaps. The unwritten ct buffer stores the hash inputs and outputs and the serialized r, and p is reused after the NTT of r for the encoding and NTT of m. Serialization and SOTP encoding are performed in place, hash inputs are absorbed while they are serialized, and the hash state is kept only inside the hash function. Absorbing during serialization gave no benefit in the speed-optimized implementation because the Keccak permutation dominates the hashing time. In the memory-optimized implementation, however, it removes an intermediate

byte array and thus reduces stack usage. The ciphertext is computed by restoring only the required base blocks of h from pk and of r from ct, and each block of r is overwritten with the final ciphertext only after its last read. Accepting this restoration cost avoids keeping both operands as full arrays at once and removes the stack space they would occupy together.

Decaps. Decaps uses one workspace and an $N/4$-byte buffer m, which is 192 bytes for NTRU+768. After $c \cdot f$ is computed, the inverse NTT and reduction modulo three yield m_1, which is ternary and is therefore stored in m with two bits per coefficient, that is, $N/4$ bytes. The workspace is reused for the forward NTT of m_1 and the computation of $\hat{r}_2$, and the seed obtained by hashing is also stored in m. The final check regenerates r_1 and compares it with $\hat{r}_2$. To reduce stack usage, r_1 is regenerated and compared in pieces rather than as a whole, and the piece size follows the structure of the 3-way NTT. In the 3-way NTT, all layers after the first radix-3 layer stay within blocks of $N/6$ coefficients, that is, 128, 144, and 192 coefficients for NTRU+768, NTRU+864, and NTRU+1152, so pieces of this block size can be computed independently of each other. In NTRU+768, 192-coefficient pieces that are not aligned with these blocks would compute some layers repeatedly in two pieces, whereas 128-coefficient pieces remove this repetition. In NTRU+864 and NTRU+1152, Decaps time is reduced by computing the radix-3 layers inside each piece with core-specific kernels instead of C. In either case, peak stack usage does not increase. The first two layers of each piece are computed directly from the seed bits with the ω-folded constants.

4 Evaluation

This section evaluates the execution time and memory cost of complete KEM operations on four Cortex-M cores. It then isolates the contribution of the 3-way NTT kernels through controlled NTT replacement and examines the ω-folded front layers and radix-3 block piece regeneration by comparing configurations with and without each technique. Finally, a same-executable analysis shows how the reduction in peak stack usage translates into SRAM and code-size changes.

4.1 Evaluation Setup and Measurement

Table 3 lists the board, clock, and timing source of each core. The board and clock settings are informed by the board-level benchmarking environments of pqm3, pqm4, and pqm7 [31,20,32]. M0+ is not covered by these environments, so the STM32G0B1RE is configured at its maximum clock of 64 MHz with the two Flash wait states required at this clock and the Flash prefetch left disabled. The evaluation focuses on changes between implementations under fixed conditions on the same core. M3, M4, and M7 use DWT CYCCNT for direct cycle counting and to keep periodic SysTick interrupts out of the timed region. M0+ does not provide the same DWT cycle counter, so it uses TIM2 at the same 64 MHz CPU clock without prescaling. M7 enables I/D caches, so for each input, timing

Table 3. Boards and measurement conditions. Basis denotes the benchmarking framework from which the build and HAL of each core were derived. All implementations were compiled with gcc 15.2.1 and `-O3`.

Item	M0+	M3	M4	M7
Basis	–	pqm3 [31]	pqm4 [20]	pqm7 [32]
Board	NUCLEO G0B1RE	NUCLEO F207ZG	NUCLEO L4R5ZI	NUCLEO F767ZI
MCU	STM32G0B1RE	STM32F207ZG	STM32L4R5ZI	STM32F767ZI
Clock	64 MHz	30 MHz	24 MHz	24 MHz
Counter	TIM2	DWT CYCCNT	DWT CYCCNT	DWT CYCCNT
Flash wait states	2	0	0	0

starts after KeyGen, Encaps, and Decaps have each been executed once to reduce variation from the initial cache state.

On each core, we compare the speed-optimized and memory-optimized implementations of NTRU+768, NTRU+864, and NTRU+1152 with the reference implementation (Ref.) and the official optimized C implementation (Opt.) of the same NTRU+ revision. Using the same 100 deterministic inputs, we report the median execution time and the peak stack usage, which is the maximum stack usage during one KEM operation. The two quantities are measured in separate ELFs so that the stack instrumentation does not change the code layout. Initialization of the random number generator, UART output, result checks, and stack scanning are excluded from the measured region. KeyGen and Encaps call the deterministic random-number generator of the harness, an AES-256-CTR DRBG written in C, and this call is included in the measured region identically for all implementations, whereas Decaps does not call it. Peak stack usage is measured from the top of the stack including the caller on M0+ and as the increase from the call site on the other cores, so stack values are compared only within each core. Every measured ELF passed the 100 official known-answer tests, produced the same outputs as the official implementation for the 100 measurement inputs, and rejected non-canonical public keys, ciphertexts, and secret keys as well as 50 ciphertexts with one flipped bit in the same way as the official implementation.

4.2 Complete KEM Timing and Peak Stack Usage

Table 4 compares the KeyGen, Encaps, and Decaps execution time and peak stack usage of each implementation on the four cores. For the speed-optimized and memory-optimized implementations, the % column next to each value gives the reduction relative to Opt.

Speed-optimized implementation. On all four cores, the speed-optimized implementation reduced execution time by 29.42% to 62.09% relative to Ref. and by 18.02% to 52.43% relative to Opt. The reduction differed by KEM operation according to the share of the optimized transform arithmetic in the total

Table 4. Complete KEM timing and peak stack usage. Speed and Memory denote the speed-optimized and memory-optimized implementations of this work, and K, E, and D denote KeyGen, Encaps, and Decaps. Time is the median of 100 inputs in k cycles for M3, M4, and M7 and in k ticks for M0+, where TIM2 runs at the CPU clock without prescaling so that ticks equal cycles. Stack is the observed maximum in B from a separate ELF. Columns marked % give reductions relative to Opt., and a negative value denotes an increase. NTRU+1152 KeyGen retries depend only on the input and are identical in every implementation.

Core			NTRU+768				NTRU+864				NTRU+1152			
			Time	%	Stack	%	Time	%	Stack	%	Time	%	Stack	%
M0+	Ref. [36]	K	2,629.0		11,784		2,982.3		13,104		4,632.3		17,064	
		E	2,379.9		10,088		2,832.5		11,200		3,603.9		14,504	
		D	1,517.1		15,768		1,888.9		17,584		2,514.2		23,016	
	Opt. [36]	K	2,447.1		10,240		2,672.3		11,656		4,139.6		14,752	
		E	2,282.5		8,936		2,599.7		9,904		3,293.1		12,776	
		D	1,298.5		11,144		1,521.5		12,384		2,024.1		16,088	
	Speed	K	1,714.7	29.93	9,980	2.54	1,826.6	31.65	11,692	−0.31	2,805.7	32.22	14,644	0.73
		E	1,603.2	29.76	7,376	17.46	1,809.1	30.41	8,200	17.21	2,262.5	31.30	10,632	16.78
		D	617.7	52.43	9,584	14.00	726.6	52.25	10,680	13.76	968.2	52.17	13,952	13.28
	Memory	K	2,029.3	17.07	2,128	79.22	2,159.6	19.19	2,320	80.10	3,491.5	15.66	2,896	80.37
		E	1,749.6	23.35	2,104	76.45	1,984.7	23.66	2,296	76.82	2,489.5	24.40	2,872	77.52
		D	935.6	27.95	2,336	79.04	1,084.7	28.71	2,536	79.52	1,448.3	28.45	3,200	80.11
M3	Ref. [36]	K	1,317.2		11,564		1,478.8		12,884		2,253.2		16,844	
		E	1,238.6		9,876		1,447.5		10,980		1,850.1		14,292	
		D	828.0		15,572		973.1		17,388		1,265.5		22,820	
	Opt. [36]	K	1,179.0		10,028		1,284.5		11,420		1,922.7		14,532	
		E	1,148.3		8,724		1,316.3		9,684		1,636.4		12,564	
		D	653.4		10,956		767.9		12,196		968.4		15,876	
	Speed	K	929.7	21.15	9,696	3.31	982.4	23.52	11,412	0.07	1,576.3	18.02	14,412	0.83
		E	842.5	26.63	6,992	19.85	943.2	28.34	7,816	19.29	1,192.9	27.10	10,256	18.37
		D	417.5	36.10	9,208	15.95	477.9	37.76	10,304	15.51	636.6	34.26	13,560	14.59
	Memory	K	965.8	18.08	1,828	81.77	1,003.2	21.90	1,996	82.52	1,648.7	14.25	2,588	82.19
		E	856.9	25.37	1,740	80.06	952.1	27.66	1,896	80.42	1,213.7	25.83	2,508	80.04
		D	475.2	27.27	2,008	81.67	528.6	31.16	2,152	82.35	724.0	25.24	2,800	82.36
M4	Ref. [36]	K	1,214.9		11,568		1,356.1		12,888		2,100.2		16,848	
		E	1,169.2		9,880		1,360.1		10,984		1,735.5		14,296	
		D	751.0		15,560		908.1		17,384		1,197.2		22,808	
	Opt. [36]	K	1,125.0		10,032		1,213.5		11,424		1,833.6		14,536	
		E	1,095.4		8,728		1,248.8		9,688		1,553.4		12,568	
		D	612.6		10,944		714.1		12,192		909.0		15,888	
	Speed	K	836.4	25.66	9,700	3.31	889.0	26.74	11,416	0.07	1,362.3	25.70	14,412	0.85
		E	757.2	30.87	6,992	19.89	857.8	31.31	7,816	19.32	1,067.5	31.28	10,256	18.40
		D	312.9	48.92	9,192	16.01	367.2	48.59	10,296	15.55	477.0	47.53	13,560	14.65
	Memory	K	868.0	22.85	1,824	81.82	912.9	24.77	2,000	82.49	1,452.9	20.76	2,592	82.17
		E	782.1	28.60	1,744	80.02	884.0	29.21	1,908	80.31	1,104.5	28.90	2,512	80.01
		D	379.4	38.07	1,992	81.80	427.2	40.18	2,136	82.48	578.2	36.40	2,796	82.40
M7	Ref. [36]	K	732.8		11,556		924.1		12,876		1,384.8		16,836	
		E	618.5		9,868		779.0		10,980		985.5		14,284	
		D	482.8		15,540		634.5		17,364		819.9		22,796	
	Opt. [36]	K	595.9		10,020		666.5		11,424		961.5		14,524	
		E	549.2		8,716		642.0		9,684		771.3		12,556	
		D	338.9		10,924		413.0		12,164		490.3		15,868	
	Speed	K	453.1	23.97	9,700	3.19	500.2	24.96	11,416	0.07	761.7	20.78	14,420	0.72
		E	406.9	25.92	6,992	19.78	479.4	25.33	7,816	19.29	592.7	23.16	10,256	18.32
		D	191.3	43.55	9,192	15.85	246.0	40.44	10,288	15.42	310.8	36.61	13,568	14.49
	Memory	K	472.5	20.72	1,824	81.80	525.4	21.17	1,988	82.60	813.5	15.40	2,592	82.15
		E	422.3	23.11	1,744	79.99	494.3	23.01	1,908	80.30	613.2	20.49	2,512	79.99
		D	245.6	27.52	1,992	81.76	292.8	29.11	2,136	82.44	384.3	21.62	2,792	82.40

time. Decaps repeatedly uses the transform kernels during message recovery, $\hat{r}_2$ computation, and regeneration, and therefore showed the largest reductions. KeyGen and Encaps also include the random-number generation of the harness, together with base inversion and retries in KeyGen and hashing and encoding in Encaps, so the transform improvement did not appear at the same rate in every KEM operation. The official optimized C implementation uses the same 3-way NTT structure. The largest reduction in Decaps therefore suggests that implementing the same transforms with core-specific kernels contributes substantially. Section 4.3 examines this contribution further. Hashing also contributes. Replacing only the hash functions of Opt. with ours reduced its execution time by 6.23% to 25.58%, which corresponds to 19.47% to 92.50% of the reduction of the speed-optimized implementation and to 21.25% to 74.65% of its Decaps reduction. In every KEM operation, this share was largest on M3 and smallest on M0+. Comparing the cores, the Decaps reduction relative to Ref. was smallest on M3, while M0+, M4, and M7 showed similar levels. This is because M3 lacks packed-halfword arithmetic, which processes two coefficients in one register, and computes butterflies with scalar arithmetic, as the controlled experiment in Section 4.3 also shows. M4 and M7 process two coefficients together with packed arithmetic. M0+ also uses scalar arithmetic, but its reduction includes the effect of executing frequently used code from SRAM to avoid Flash wait states, whereas Ref. and Opt. execute entirely from Flash.

Memory-optimized implementation. The memory-optimized implementation reduces the workspace that is live at the same time through KEM-level buffer reuse, restoration of base blocks, and piece regeneration. On the four cores, it reduced execution time by 22.81% to 53.86% and peak stack usage by 79.14% to 87.75% relative to Ref. Relative to Opt., it was 14.25% to 40.18% faster in every KEM operation while using 76.45% to 82.60% less peak stack. For example, in M7 NTRU+864 Decaps, the memory-optimized implementation took 19.02% more time than the speed-optimized implementation but used 79.24% less peak stack. This time increase was largest in Decaps, which recomputes operands piece by piece, for every core and parameter set, and among the cores it was largest on M0+. This is because the memory-optimized implementation on M0+ not only recomputes operands but also executes the transform kernels from Flash rather than SRAM. Radix-3 block piece regeneration reduces part of this recomputation cost, as examined in Section 4.3.

Code placement on M0+. On M0+, the memory-optimized implementation executes only the used Keccak routines from SRAM, whereas the speed-optimized implementation also executes the transforms, base arithmetic, and serialization from SRAM. As a result, the speed-optimized implementation reduced execution time by 32.63% to 61.53% relative to Ref., at the cost of resident SRAM that Section 4.4 analyzes.

4.3 Controlled NTT Replacement and 3-Way NTT Techniques

Table 5 combines two kinds of comparison. The left part is a controlled experiment that replaces only the NTT kernels. The right part compares configurations with and without the ω-folded front layers and radix-3 block piece regeneration, which are specific to the 3-way NTT. The controlled experiment uses the official NTRU+ reference implementation [36], whose NTT kernels can be exchanged in isolation. In this implementation, only the forward and inverse NTT kernels are replaced with the general NTT kernels of the speed-optimized implementation, with one C Montgomery pass after the inverse NTT to match the output format of the reference implementation, while sampling, base operations, hashing, packing, polynomial buffers, function placement, and the KeyGen retry path remain unchanged. The reductions on the left therefore represent the complete NTT kernels, including layer connections, modular arithmetic, reductions, and instruction scheduling. Each final implementation is measured in pairs with a configuration in which only the technique is disabled, using the same 100 inputs. The front layers are measured in the speed-optimized implementation, and piece regeneration in the memory-optimized implementation.

Table 5. Reductions in %. NTT kernel replacement is relative to the controlled reference implementation, and the other columns are relative to the same implementation without the technique. K, E, and D denote KeyGen, Encaps, and Decaps, and Forward, Inverse, and NTT denote kernel time. − means not applied, and † means not measurable on the board. Piece regeneration includes the ω-folded front layers used inside each piece. Kernel columns are medians of repeated runs on one input. With the ω-folded front layers, Encaps time decreased by at most 1.26%.

		NTT kernel replacement					ω-folded front			Piece regen.
Core	N	Forward	Inverse	K	E	D	NTT	K	D	D
M0+	768	45.16	40.24	6.09	6.69	16.59	13.85	1.98	2.91	5.63
	864	55.81	38.45	10.37	10.83	21.80	12.74	2.07	2.86	13.15
	1152	56.17	39.04	12.94	11.09	21.58	†	2.68	2.83	11.08
M3	768	42.20	35.90	5.85	6.22	13.94	1.39	1.66	0.45	5.07
	864	52.70	34.68	9.18	9.38	18.61	1.68	1.47	−0.16	11.61
	1152	52.52	38.11	11.71	9.51	18.89	−	−	−	8.31
M4	768	65.11	42.79	8.85	9.19	19.37	5.28	0.96	0.74	11.41
	864	67.25	46.18	11.24	11.21	22.16	4.58	1.00	0.71	17.91
	1152	70.74	46.34	15.07	12.16	22.80	5.24	1.33	0.73	15.06
M7	768	79.41	67.15	14.44	17.23	33.69	0.64	0.83	0.22	6.19
	864	76.65	59.45	17.88	21.29	35.41	0.99	0.82	0.00	12.78
	1152	80.60	63.36	25.33	23.62	37.92	0.47	0.89	0.03	8.96

Under the controlled conditions, the NTT-kernel reductions differed clearly between cores. M4 and M7, which use packed arithmetic, showed larger reductions than M0+ and M3, which rely on scalar arithmetic. The forward NTT reductions were 42.20% to 56.17% on M0+ and M3 and 65.11% to 80.60% on M4 and M7. M4 and M7 use the same packed Plantard arithmetic, but M7, which is scheduled with SLOTHY for dual issue, showed larger reductions for all three parameter sets in both directions. On all cores and parameter sets, the inverse NTT reductions were smaller than the forward ones.

Across parameter sets, NTRU+864 showed the smallest reductions in both directions on M7. Because its base blocks have three coefficients, the coefficient distance of its last CT layer is odd, and this part is processed with scalar rather than packed operations. The packed arithmetic and dual-issue scheduling that increase the reductions on M7 therefore do not reach this part. On M4, by contrast, the reductions increased with the parameter length in both directions. Parameter structure such as the degree of the base blocks thus determines where core-specific optimization applies, and the forward and inverse paths must be configured according to the radix-3 connections and base-block structure of each parameter set.

On M7, Decaps decreased by 33.69% to 37.92% in the controlled experiment and by 60.38% to 62.09% in the speed-optimized implementation. Since the remaining reduction comes from optimizations outside the NTT, such as base operations, hashing, and the KEM dataflow, the NTT-kernel improvement alone cannot explain the performance of complete KEM operations. The peak stack usage with only the NTT kernels replaced was identical to that of the reference implementation. The NTT kernels thus reduce only execution time, and the large stack reductions of the memory-optimized implementation in Table 4 come from changes in the KEM dataflow.

ω-folded front layers. The ω-folded front layers reduced the forward NTT kernel time for ternary inputs by 12.74% to 13.85% on M0+. The reduction was smaller on M4, and M3 and M7 showed no clear effect, with reductions below 2%. This difference arises because the cost of the removed reductions differs between cores. On M0+, which uses only Thumb-1 instructions, one reduction requires several instructions, so removing them roughly halved the instruction count of the first two layers. On M4 and M7, DSP instructions already make reductions cheap, and the instruction count decreased by only 19.1% to 19.2%. In particular, the first layers on M7 are already scheduled with SLOTHY and bound by the multiply-accumulate pipeline, so fewer instructions barely reduced the time. On M3, the front layers require undoing the fusion of the radix-3 and CT layers, which offsets the gain. For NTRU+1152 the kernel even became 2.78% slower, so the front layers are not applied there. In KEM operations, the ternary NTT accounts for only part of the Decaps time. Decaps therefore decreased by 2.83% to 2.91% on M0+, and the difference was below 1% on the other cores. The front layers are thus most effective on low-end cores with costly reductions.

Radix-3 block piece regeneration. The configuration without piece regeneration computes all layers inside each piece in C and, for NTRU+768, uses 192-coefficient pieces that are not aligned with the block boundaries. Compared with this configuration, the memory-optimized implementation reduced Decaps time by 5.07% to 17.91%, with almost no effect on KeyGen, Encaps, or peak stack usage. Since NTRU+864 and NTRU+1152 use the same piece size in both configurations, the reduction for these parameter sets reflects computing each piece with the ω-folded front layers and core-specific kernels. For NTRU+768, it additionally includes removing the repeated computation of the same front layers in two pieces, and the two effects are not separated. This gain is possible because aligned pieces are computed independently of each other, so the computation inside each piece can be replaced with faster kernels without increasing the stack.

4.4 Memory Footprint within the Same Executable

A reduction in peak stack usage does not necessarily imply a reduction in the SRAM reservation. This is especially relevant on M0+, where executable code is placed in SRAM. Core-specific kernels can also increase code size while reducing execution time. We therefore measure resident sections, heap usage, and the usage of the main stack pointer (MSP) region within one separate memory-measurement ELF for each implementation and separately report the `.text` size from the same ELF. SRAM Total in Table 6 combines D+B, the SRAM-executed code R, other resident sections and alignment O+A, the additional heap peak H, and the whole-MSP maximum M. The `.text` size indicates the code size placed in Flash and is therefore not included in SRAM Total. Since H and M are independently observed maxima, Total must be interpreted as a conservative SRAM reservation rather than a directly observed simultaneous peak. D+B and H are identical for all implementations on the same core and come almost entirely from the measurement harness, so the part attributable to the KEM is R + M. In Table 6, the % column next to Total gives its reduction relative to Opt.

On M3, M4, and M7, the memory-optimized implementation reduced the SRAM reservation of the same ELF by 27.95% to 29.43% relative to Ref. and by 20.37% to 21.43% relative to Opt. Counting only R + M, the reduction relative to Opt. was 80.50% to 81.58%. KEM-level lifetime control reduced the polynomial workspace that is live at the same time and thus the whole-MSP usage. Resident sections such as data and bss remain, however, so the SRAM reduction was smaller than the reduction in peak stack usage. For example, on M7 NTRU+768, the SRAM reservation decreased by 27.95%, while the Decaps peak stack usage decreased by 87.18%. The speed-optimized implementation, in contrast, keeps operands as full polynomial arrays, so its SRAM reservation on the same three cores was only 1.52% to 2.87% smaller than that of Opt.

The kernels specialized for execution time also increased code size. Relative to Ref., the `.text` size of the speed-optimized implementation increased by 14.91% to 22.43% on M3, M4, and M7. The memory-optimized implementation adds the restructured KEM routines and was 3.75% to 6.05% larger than the speed-optimized implementation.

Table 6. Same-ELF `.text` size and SRAM reservation in B for the NTRU+ reference implementation, the official optimized C implementation, and the Speed and Memory implementations of this work. `.text` denotes the text section and is not included in Total. D+B denotes data and bss, R denotes SRAM-resident code, O+A denotes other resident sections and alignment, H denotes the peak additional heap, and M denotes the observed whole-MSP maximum over 100 inputs. The column marked % gives the reduction of Total relative to Opt.

Core	N	Implementation	`.text`	D+B	R	O+A	H	M	Total	%
M0+	768	Ref. [36]	155,632	52,312	0	0	0	15,768	68,080	
		Opt. [36]	172,692	52,312	0	0	0	11,144	63,456	
		Speed	161,992	52,312	10,136	0	0	9,980	72,428	−14.14
		Memory	178,816	52,312	3,692	4	0	2,336	58,344	8.06
	864	Ref. [36]	164,096	55,480	0	0	0	17,584	73,064	
		Opt. [36]	182,596	55,480	0	0	0	12,384	67,864	
		Speed	172,296	55,480	9,128	0	0	11,692	76,300	−12.43
		Memory	188,472	55,480	3,692	4	0	2,536	61,712	9.07
	1152	Ref. [36]	202,160	64,984	0	0	0	23,016	88,000	
		Opt. [36]	224,660	64,984	0	0	0	16,088	81,072	
		Speed	213,312	64,984	10,136	0	0	14,644	89,764	−10.72
		Memory	229,968	64,984	3,692	4	0	3,200	71,880	11.34
M3	768	Ref. [36]	158,172	30,100	0	4	2,664	15,732	48,500	
		Opt. [36]	173,396	30,100	0	4	2,664	11,116	43,884	
		Speed	186,108	30,100	0	4	2,664	9,856	42,624	2.87
		Memory	195,988	30,100	0	4	2,664	2,168	34,936	20.39
	864	Ref. [36]	166,972	33,268	0	4	3,592	17,548	54,412	
		Opt. [36]	183,484	33,268	0	4	3,592	12,356	49,220	
		Speed	196,500	33,268	0	4	3,592	11,572	48,436	1.59
		Memory	203,876	33,268	0	4	3,592	2,312	39,176	20.41
	1152	Ref. [36]	204,932	42,772	0	4	2,280	22,980	68,036	
		Opt. [36]	224,396	42,772	0	4	2,280	16,036	61,092	
		Speed	235,996	42,772	0	4	2,280	14,572	59,628	2.40
		Memory	248,684	42,772	0	4	2,280	2,960	48,016	21.40
M4	768	Ref. [36]	157,268	30,100	0	4	2,664	15,720	48,488	
		Opt. [36]	172,628	30,100	0	4	2,664	11,104	43,872	
		Speed	180,716	30,100	0	4	2,664	9,860	42,628	2.84
		Memory	191,652	30,100	0	4	2,664	2,152	34,920	20.40
	864	Ref. [36]	166,012	33,268	0	4	3,592	17,544	54,408	
		Opt. [36]	182,700	33,268	0	4	3,592	12,352	49,216	
		Speed	193,388	33,268	0	4	3,592	11,576	48,440	1.58
		Memory	202,180	33,268	0	4	3,592	2,296	39,160	20.43
	1152	Ref. [36]	204,028	42,772	0	4	2,280	22,968	68,024	
		Opt. [36]	223,692	42,772	0	4	2,280	16,048	61,104	
		Speed	238,564	42,772	0	4	2,280	14,572	59,628	2.42
		Memory	249,092	42,772	0	4	2,280	2,956	48,012	21.43
M7	768	Ref. [36]	157,516	30,092	0	4	2,672	15,700	48,468	
		Opt. [36]	172,876	30,092	0	4	2,672	11,084	43,852	
		Speed	188,348	30,092	0	4	2,672	9,860	42,628	2.79
		Memory	199,412	30,092	0	4	2,672	2,152	34,920	20.37
	864	Ref. [36]	166,212	33,260	0	4	3,600	17,524	54,388	
		Opt. [36]	183,116	33,260	0	4	3,600	12,324	49,188	
		Speed	203,500	33,260	0	4	3,600	11,576	48,440	1.52
		Memory	212,508	33,260	0	4	3,600	2,296	39,160	20.39
	1152	Ref. [36]	204,236	42,764	0	4	2,288	22,956	68,012	
		Opt. [36]	224,004	42,764	0	4	2,288	16,028	61,084	
		Speed	245,492	42,764	0	4	2,288	14,580	59,636	2.37
		Memory	257,436	42,764	0	4	2,288	2,952	48,008	21.41

M0+ showed a different memory trade-off. The speed-optimized implementation keeps 9,128 B to 10,136 B of frequently executed code resident in SRAM.

Because this resident code is larger than the reduction in peak stack usage, its SRAM reservation exceeded those of both Ref. and Opt. The memory-optimized implementation, which keeps only 3,692 B of Keccak routines in SRAM, reduced the reservation by 8.06% to 11.34% and R + M by 45.91% to 57.16% relative to Opt. Conversely, the `.text` of the memory-optimized implementation was 7.81% to 10.39% larger than that of the speed-optimized implementation, because the speed-optimized implementation moves its main execution path into the SRAM-executed region whereas the memory-optimized implementation keeps it in Flash. The M0+ implementations therefore offer a choice between the speed-optimized implementation, which spends SRAM on executable code for the shortest execution time, and the memory-optimized implementation, which reduces both peak stack usage and the SRAM reservation.

5 Conclusion

This work optimizes the 3-way NTT and the complete KEM dataflow of NTRU+768, NTRU+864, and NTRU+1152 for Cortex-M0+, M3, M4, and M7. We integrate core-specific NTT kernels tailored to the transform structure of each parameter set into the official optimized C implementation [36] and present two implementations that target execution time and memory usage, respectively, through buffer reuse based on the last use of intermediate data. In particular, the layer-fusion boundary selection, the applicability condition of the ω-folded front layers, radix-3 block piece regeneration, and workspace reuse are independent of the instruction set, and the ω-folded front layers remain within the 16-bit range for CTRU, which uses the same $q = 3457$. These techniques therefore have room to be carried over to other 3-way NTT-based schemes and other embedded platforms.

In our experiments, the speed-optimized implementation reduced execution time by up to 52.43% relative to the official optimized C implementation, and the memory-optimized implementation reduced peak stack usage by up to 82.60% while remaining faster than the official optimized C implementation in every KEM operation. These results show three points. First, the same technique, such as the four-multiplication radix-3 butterfly or the ω-folded front layers, yielded different gains on different cores. Second, SRAM-resident code can outweigh a reduction in peak stack usage, as on M0+, where the SRAM reservation of the speed-optimized implementation even increased because of the executable code placed in SRAM. Third, aligning the pieces with the radix-3 block boundaries allowed the core-specific kernels to be used inside each piece, which reduced the recomputation cost of the memory-optimized implementation without increasing the stack. The optimization strategy should therefore be chosen by considering both the execution characteristics and the overall memory budget of the target core.

However, the SRAM reservation measured in this work is the sum of independently observed memory maxima and may differ from the peak concurrent memory usage of an actual application, and the power and electromagnetic side-

channel characteristics were not evaluated. In addition, the proposed techniques were measured only on NTRU+, so their performance on other 3-way NTT-based schemes such as CTRU has not yet been determined. The implementation and evaluation are also limited to Arm Cortex-M cores and have not been extended to other embedded instruction sets such as RISC-V. Future work will therefore apply the techniques identified as independent of the instruction set to other 3-way NTT-based algorithms and other embedded platforms to verify their effect, and will analyze power and electromagnetic leakage.

We used Claude AI (Anthropic) to translate and edit the text and to assist with writing code, and the authors reviewed and verified all content and results and take full responsibility for them.

References

1. Abdulrahman, A., Becker, H., Kannwischer, M.J., Klein, F.: Fast and Clean: Auditable high-performance assembly via constraint solving. IACR Transactions on Cryptographic Hardware and Embedded Systems **2024**(1), 87–132 (2024). https://doi.org/10.46586/tches.v2024.i1.87-132
2. Abdulrahman, A., Chen, J.P., Chen, Y.J., Hwang, V., Kannwischer, M.J., Yang, B.Y.: Multi-moduli NTTs for Saber on Cortex-M3 and Cortex-M4. IACR Transactions on Cryptographic Hardware and Embedded Systems **2022**(1), 127–151 (2022). https://doi.org/10.46586/tches.v2022.i1.127-151
3. Abdulrahman, A., Kannwischer, M.J., Lim, T.H.: Enabling microarchitectural agility: Taking ML-KEM & ML-DSA from Cortex-M4 to M7 with SLOTHY. In: Proceedings of the 20th ACM Asia Conference on Computer and Communications Security. pp. 1756–1771. ACM (2025). https://doi.org/10.1145/3708821.3736210
4. Adomnicai, A.: An update on Keccak performance on ARMv7-M. Cryptology ePrint Archive, Paper 2023/773 (2023), revised May 15, 2024. https://eprint.iacr.org/2023/773. Source code: https://github.com/aadomn/keccak_armv7m
5. Alkim, E., Bilgin, Y.A., Cenk, M., Gérard, F.: Cortex-M4 optimizations for {R,M}LWE schemes. IACR Transactions on Cryptographic Hardware and Embedded Systems **2020**(3), 336–357 (2020). https://doi.org/10.13154/tches.v2020.i3.336-357
6. Alkim, E., Cheng, D.Y.L., Chung, C.M.M., Evkan, H., Huang, L.W.L., Hwang, V., Li, C.L.T., Niederhagen, R., Shih, C.J., Wälde, J., Yang, B.Y.: Polynomial multiplication in NTRU Prime: Comparison of optimization strategies on Cortex-M4. IACR Transactions on Cryptographic Hardware and Embedded Systems **2021**(1), 217–238 (2021). https://doi.org/10.46586/tches.v2021.i1.217-238
7. Alkim, E., Hwang, V., Yang, B.Y.: Multi-parameter support with NTTs for NTRU and NTRU Prime on Cortex-M4. IACR Transactions on Cryptographic Hardware and Embedded Systems **2022**(4), 349–371 (2022). https://doi.org/10.46586/tches.v2022.i4.349-371
8. Arm: Cortex-M3 Technical Reference Manual, Revision r2p0. Arm (2008), ARM DDI 0337G
9. Arm: Cortex-M0+ Devices Generic User Guide. Arm (2012), ARM DUI 0662B. https://developer.arm.com/documentation/dui0662/b
10. Barrett, P.: Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In: Advances in Cryptology, CRYPTO '86. Lecture Notes in Computer Science, vol. 263, pp. 311–323. Springer (1987). https://doi.org/10.1007/3-540-47721-7_24

11. Becker, H., Mera, J.M.B., Karmakar, A., Yiu, J., Verbauwhede, I.: Polynomial multiplication on embedded vector architectures. IACR Transactions on Cryptographic Hardware and Embedded Systems **2022**(1), 482–505 (2022). https://doi.org/10.46586/tches.v2022.i1.482-505
12. Cho, S., Bang, J., Sim, M., Seo, H.: A memory-efficient and assembly-optimized implementation of NTRU+. Cryptology ePrint Archive, Paper 2026/1410 (2026), https://eprint.iacr.org/2026/1410
13. Chung, C.M.M., Hwang, V., Kannwischer, M.J., Seiler, G., Shih, C.J., Yang, B.Y.: NTT multiplication for NTT-unfriendly rings: New speed records for Saber and NTRU on Cortex-M4 and AVX2. IACR Transactions on Cryptographic Hardware and Embedded Systems **2021**(2), 159–188 (2021). https://doi.org/10.46586/tches.v2021.i2.159-188
14. Cooley, J.W., Tukey, J.W.: An algorithm for the machine calculation of complex Fourier series. Mathematics of Computation **19**(90), 297–301 (1965). https://doi.org/10.1090/S0025-5718-1965-0178586-1
15. Gentleman, W.M., Sande, G.: Fast Fourier transforms: For fun and profit. In: Proceedings of the November 7–10, 1966, Fall Joint Computer Conference. pp. 563–578. Association for Computing Machinery (1966). https://doi.org/10.1145/1464291.1464352
16. Huang, J., Adomnicăi, A., Zhang, J., Dai, W., Liu, Y., Cheung, R.C.C., Koç, Ç.K., Chen, D.: Revisiting Keccak and Dilithium implementations on ARMv7-M. IACR Transactions on Cryptographic Hardware and Embedded Systems **2024**(2), 1–24 (2024). https://doi.org/10.46586/tches.v2024.i2.1-24
17. Huang, J., Zhang, J., Zhao, H., Liu, Z., Cheung, R.C.C., Koç, Ç.K., Chen, D.: Improved Plantard arithmetic for lattice-based cryptography. IACR Transactions on Cryptographic Hardware and Embedded Systems **2022**(4), 614–636 (2022). https://doi.org/10.46586/tches.v2022.i4.614-636
18. Huang, J., Zhao, H., Zhang, J., Dai, W., Zhou, L., Cheung, R.C.C., Koç, Ç.K., Chen, D.: Yet another improvement of Plantard arithmetic for faster Kyber on low-end 32-bit IoT devices. IEEE Transactions on Information Forensics and Security **19**, 3800–3813 (2024). https://doi.org/10.1109/TIFS.2024.3371369
19. Ireland, K., Rosen, M.: A Classical Introduction to Modern Number Theory, Graduate Texts in Mathematics, vol. 84. Springer, second edn. (1990). https://doi.org/10.1007/978-1-4757-2103-4
20. Kannwischer, M.J., Rijneveld, J., Schwabe, P., Stoffelen, K.: pqm4: Testing and benchmarking NIST PQC on ARM Cortex-M4. Cryptology ePrint Archive, Paper 2019/844 (2019), https://eprint.iacr.org/2019/844
21. Keccak Team: XKCP: The eXtended Keccak Code Package. Source repository (2026), accessed October 16, 2026. https://github.com/XKCP/XKCP
22. Kim, G., Seo, E., Lee, Y., Kim, Y.S., No, J.S.: Lazy modular reduction for NTT. Electronics **13**(24), 4887 (2024). https://doi.org/10.3390/electronics13244887
23. Kim, J., Cho, H., Park, J.H.: Accelerating NTRU+ key generation via hierarchical batch inversion. Cryptology ePrint Archive, Paper 2026/1191 (2026), https://eprint.iacr.org/2026/1191
24. Kim, J., Cho, H., Park, J.H.: Hierarchical batch inversion for NTRU+. Source repository (2026), accessed October 5, 2026. https://github.com/yoswuk/hierarchical_batch_inversion
25. Kim, J., Park, J.H.: NTRU+: Compact construction of NTRU using simple encoding method. IEEE Transactions on Information Forensics and Security **18**, 4760–4774 (2023). https://doi.org/10.1109/TIFS.2023.3299172

26. Korean Post-Quantum Cryptography Research Group: KpqC algorithms (2026), final specification documents, January 30, 2026. Accessed October 5, 2026. https://www.kpqc.or.kr/contents/03_exhibit/sub_03.html
27. Liang, Z., Fang, B., Zheng, J., Zhao, Y.: Compact and efficient KEMs over NTRU lattices. Cryptology ePrint Archive, Paper 2022/579 (2022), https://eprint.iacr.org/2022/579
28. Lorenser, T.: The DSP capabilities of ARM Cortex-M4 and Cortex-M7 processors. White paper (2016), https://developer.arm.com/community/cfs-file/__key/communityserver-blogs-components-weblogfiles/00-00-00-21-42/7563.ARM-white-paper-_2D00_-DSP-capabilities-of-Cortex_2D00_M4-and-Cortex_2D00_M7.pdf
29. Lyubashevsky, V., Seiler, G.: NTTRU: Truly fast NTRU using NTT. IACR Transactions on Cryptographic Hardware and Embedded Systems **2019**(3), 180–201 (2019). https://doi.org/10.13154/tches.v2019.i3.180-201
30. Montgomery, P.L.: Modular multiplication without trial division. Mathematics of Computation **44**(170), 519–521 (1985). https://doi.org/10.1090/S0025-5718-1985-0777282-X
31. mupq contributors: pqm3. Embedded benchmarking source repository (2026), accessed October 5, 2026. https://github.com/mupq/pqm3
32. mupq contributors: pqm7. Embedded benchmarking source repository (2026), accessed October 5, 2026. https://github.com/mupq/pqm7
33. National Institute of Standards and Technology: SHA-3 standard: Permutation-based hash and extendable-output functions. Tech. Rep. FIPS PUB 202, National Institute of Standards and Technology (2015), https://doi.org/10.6028/NIST.FIPS.202
34. National Institute of Standards and Technology: Module-lattice-based key-encapsulation mechanism standard. Tech. Rep. FIPS PUB 203, National Institute of Standards and Technology (2024), https://doi.org/10.6028/NIST.FIPS.203
35. NTRU+ developers: NTT for cyclotomic trinomials. Reference notebook and source repository (2026), accessed October 5, 2026. https://github.com/ntruplus/ntt_for_ntruplus
36. NTRU+ developers: Official NTRU+ implementations. Source repository (2026), accessed October 5, 2026. https://github.com/ntruplus/ntruplus
37. Park, J.H., Kim, J.: NTRU+: Compact construction of NTRU using simple encoding method. Final Version of the KpqC Competition (2026), july 10, 2026. https://data.ntruplus.org/NTRU%2B_20260710.pdf
38. Shor, P.W.: Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM Journal on Computing **26**(5), 1484–1509 (1997). https://doi.org/10.1137/S0097539795293172
39. SLOTHY contributors: SLOTHY: Assembly superoptimization via constraint solving. Source repository (2026), accessed October 16, 2026. https://github.com/slothy-optimizer/slothy
40. UIC-ESLAS: Improved Plantard arithmetic for lattice-based cryptography. Source repository (2022), accessed October 5, 2026. https://github.com/UIC-ESLAS/ImprovedPlantardArithmetic