2025 Cryptography Engineering Final Project Report

B12902036 David

December 29, 2025

1 Introduction

Public-key cryptography is a core building block of modern network security, enabling key establishment and authentication over public channels. The concept was introduced by Diffie and Hellman, and shortly after, Rivest–Shamir–Adleman proposed RSA as a practical public-key cryptosystem. Elliptic-curve cryptography (ECC) enables comparable security with substantially smaller key sizes and is therefore widely deployed in performance and bandwidth-constrained settings.

The security of RSA and ECC relies on classical hardness assumptions such as integer factorization and the discrete logarithm problem. Shor's quantum algorithm shows that these problems can be solved in polynomial time on a sufficiently large fault-tolerant quantum computer, motivating the ongoing migration to post-quantum cryptography. In this context, NIST standardized the Module-Lattice-Based Key-Encapsulation Mechanism (ML-KEM) as FIPS 203, whose security is related to the Module Learning With Errors (Module-LWE) problem.

This project targets Arm Cortex-M4 microcontrollers, where implementations must balance performance, memory footprint, and side-channel resistance. Following the constant-time design principle to mitigate timing side channels, I optimize ECDH over Curve25519 (X25519) and ML-KEM with assembly-level and micro-architectural optimizations while preserving functional equivalence. For ML-KEM, the focus is on accelerating the NTT-based polynomial arithmetic under a strict code-size budget; for ECDH25519, the focus is on accelerating finite-field arithmetic and scalar multiplication. Performance is measured using a Cortex-M4 benchmarking workflow consistent with prior embedded PQC benchmarking practice.

2 Preliminaries

2.1 ECDH25519

ECDH25519 refers to Elliptic-Curve Diffie–Hellman key agreement instantiated with X25519 on Curve25519. Each party samples a 32-byte secret scalar, derives a public key by multiplying the fixed base point G, and then computes a shared secret by multiplying the peer public key. Concretely, if Alice has secret scalar k_A and Bob publishes $B = k_B \cdot G$, then Alice computes $S = k_A \cdot B = k_A \cdot (k_B \cdot G)$, while Bob computes $S' = k_B \cdot (k_A \cdot G)$; by commutativity of scalar multiplication, both obtain the same shared point. X25519 outputs

the u-coordinate encoding of this shared point as a 32-byte value, which is then fed into a KDF or hash-based key schedule to derive session keys for symmetric cryptography.

2.1.1 Window-based Fixed Scalar Multiplication

At the core of ECDH25519 is scalar multiplication on Curve25519. For ECDH shared-secret computation, the input point is the peer public key, which is a variable point, and double-and-add always or the Montgomery ladder is commonly used due to its regular, constant-time control flow. In contrast, when the base point is fixed, window methods can reduce the number of additions by trading memory for precomputation. The scalar is recoded into w-bit digits, where w is also called the window size, and the computation reuses a table of precomputed multiples of a fixed point.

As an example with $w = 4$, we interpret the scalar in base $2^w = 16$. Let $k = \texttt{0xD63}$. Splitting into 4-bit windows from least significant to most significant gives $d_0 = \texttt{0x3}$, $d_1 = \texttt{0x6}$, $d_2 = \texttt{0xD}$, so that $k = d_0 + 16 \cdot d_1 + 16^2 \cdot d_2$. A straightforward fixed-base approach precomputes a table $T[d] = d \cdot P$ for $d \in \{0, \ldots, 15\}$ (with $T[0]$ as the neutral element), and evaluates $Q \leftarrow T[d_2], \quad Q \leftarrow 16 \cdot Q + T[d_1], \quad Q \leftarrow 16 \cdot Q + T[d_0]$, where multiplication by 16 is implemented as four consecutive doublings. Importantly, any table lookup must be implemented in constant time by scanning all entries and conditionally moving; otherwise the method can leak information via timing or microarchitectural side channels.

2.1.2 Operand Scanning and Product Scanning and Squaring

Efficient field arithmetic modulo $p = 2^{255} - 19$ dominates the runtime of scalar multiplication. Limb multiplication is typically a convolution followed by reduction. Two classical loop organizations are operand scanning and product scanning.

In operand scanning, we fix one limb index i of $a = \sum_i a_i B^i$ in the outer loop and accumulate products $a_i b_j$ into output positions c_{i+j} in the inner loop. This layout is simple and often yields good locality for the fixed limb a_i, allowing a tight multiply-accumulate pipeline with explicit carry propagation.

In product scanning, we fix the output index k in the outer loop and sum all products on the diagonal $i + j = k$ in the inner loop: $c_k = \sum_{i+j=k} a_i b_j$. This can reduce the amount of live carry state per iteration and can be attractive under tight register constraints .

Squaring is the special case $a = b$. A naive multiplication repeats work because $a_i a_j$ and $a_j a_i$ are identical. An optimized squaring exploits symmetry – cross terms with $i < j$ are computed once and doubled, while diagonal terms a_i^2 are computed once. This reduces the number of limb multiplications and is a common optimization target in Cortex-M4-oriented implementations [Len25].

2.2 ML-KEM

2.2.1 Cooley-Tukey Butterfly

ML-KEM heavily relies on polynomial arithmetic modulo $q = 3329$ with degree $n = 256$. Its arithmetic works in the ring $\mathbb{Z}_{3329}[X]/(X^{256} + 1)$. The forward NTT is commonly implemented using Cooley–Tukey (CT) butterflies. For a pair (a, b) and a twiddle factor ζ, a CT butterfly computes $t \leftarrow \zeta \cdot b \bmod q, \quad a' \leftarrow a + t \bmod q, \quad b' \leftarrow a - t \bmod q$. The cost center

is the modular multiplication and reduction in $t = \zeta b \bmod q$. Practical implementations use Montgomery or Barrett reduction to avoid division, and optimize memory traffic by arranging loops so that intermediate values stay within controlled ranges, thereby reducing the number of explicit reductions [Abd+22].

2.2.2 Gentleman-Sande Butterfly

The inverse NTT is often implemented using Gentleman–Sande (GS) butterflies [Abd+22]. For a pair (a, b) and twiddle factor ζ, a GS butterfly computes $t \leftarrow a + b \bmod q, \quad u \leftarrow a - b \bmod q, \quad a' \leftarrow t, \quad b' \leftarrow \zeta \cdot u \bmod q$. Compared to CT, GS applies the twiddle multiplication after forming the difference u. Hence, under lazy reduction, coefficient growth in GS can be more aggressive than CT, and many Cortex-M4-oriented implementations place reductions very frequently (often per butterfly) to keep values within bounded ranges.

2.3 ARM Cortex-M4

ARM Cortex-M4 is a 32-bit microcontroller core widely used in embedded systems, and it is a representative target for constant-time cryptographic implementations under tight resource constraints. From an implementation perspective, the key performance bottlenecks are typically multi-precision integer arithmetic, modular reduction, and memory traffic. On Cortex-M4, the working register file is limited, which makes register allocation and loop structure critical – excessive spilling can dominate runtime even when the arithmetic itself is efficient.

Cortex-M4 provides a fast multiply instruction and a richer set of DSP-style instructions that can be leveraged to accelerate limb arithmetic when data are arranged appropriately; instruction timing depends on the microcontroller system configuration. Consequently, many optimizations in this report focus on reducing the number of loads/stores. At the same time, cryptographic software on Cortex-M4 must be written with side-channel resistance in mind: branches and memory accesses that depend on secrets can leak information through timing or microarchitectural effects. Therefore, the optimizations considered here preserve constant-time control flow by construction, and evaluate trade-offs among cycle count, code size, and memory usage (stack/RAM) under the constraints of a microcontroller environment.

3 Optimization Strategies

3.1 ECDH25519

3.1.1 Side Channel Security

In the original C reference, scalar multiplication contains a secret-dependent branch, which can violate constant-time requirements. To eliminate this risk, two standard constant-time control-flow patterns are the Montgomery ladder and double-and-add always. In our setting, I adopt double-and-add always because it achieves lower cycle counts than the ladder in my Cortex-M4 benchmarks, while still maintaining regular control flow. The implementation relies on two constant-time helpers, conditional move and conditional swap, to avoid secret-dependent branches and memory-access patterns .

3.1.2 Mutliplications and Squarings under $2^{255} - 19$

As discussed above, finite-field multiplication and squaring modulo $p = 2^{255}-19$ dominate the runtime of ECDH25519. Therefore, we prioritize optimizing these routines and translate the critical paths into assembly.

First, the reference implementation represents field elements with a conservative limb size to minimize overflow risks. On Cortex-M4, however, general-purpose registers are 32-bit wide, so overly small limbs can waste arithmetic capacity. I therefore switch to a larger radix close to $2^{25.5}$, implemented by alternating 25-bit and 26-bit limbs. Under a schoolbook multiplication, this reduces the number of limb products from the byte-radix baseline (up to $32\times32 = 1024$ products) to a $10\times10 = 100$-product convolution, i.e., a 90% reduction in the number of multiplications.

Second, I replace the nested-loop operand-scanning layout with a product-scanning layout. Although product scanning makes register allocation more challenging on Cortex-M4, it reduces intermediate stores and improves dataflow for carry handling. In particular, the design better matches multiply-accumulate instructions such as **smlal**, which combine multiplication and accumulation efficiently.

Third, the reference squaring routine reuses the generic multiplication path with identical inputs. Since many cross terms are duplicated in squaring, I implement a dedicated squaring routine that exploits symmetry: cross terms are computed once and doubled, and diagonal terms are computed once. This reduces the total number of limb multiplications and improves performance [Len25].

Finally, under the larger-radix design, we integrate the required carry propagation and modular reduction into the mul/sqr output contraction step (i.e., the routine that converts the internal 25/26-bit limbs back to the 32-byte representation). Therefore, we avoid calling a separate fe25519_freeze-style canonicalization in the middle of the computation and only invoke canonicalization when it is semantically required. This reduction fusion removes redundant normalization calls from the critical path and yields additional cycle savings.

3.1.3 Scalar Multiplication

Besides ensuring constant-time behavior, we also optimize the fixed-base scalar multiplication path. In the baseline, fixed-base scalar multiplication reuses the variable-point routine, which misses a significant time–space trade-off opportunity. With a fixed base point, we precompute the multiples required by a window method.

We use window size $w = 4$ and precompute a table of multiples so that $table[i][j] = j \cdot 2^{iw}G$. This reduces the main loop from a sequence dominated by doublings and additions (e.g., 255 doublings and about 128 additions) to roughly 64 additions after recoding, at the cost of increased storage. A key side-channel risk is indexed table loading; therefore, we implement table selection in constant time using conditional moves . The precomputed table occupies about $512KB$ of flash, which fits within the $1MB$ flash budget of our target Cortex-M4 board.

3.2 ML-KEM

3.2.1 Open Source Implementation

A common intuition is that NTT-based polynomial arithmetic dominates ML-KEM. However, in many ML-KEM implementations on Cortex-M4, a large fraction of cycles is spent in Keccak-based hashing (SHAKE and related permutations) during key generation, encapsulation, and decapsulation. Therefore, adopting a high-performance Keccak implementation is important for end-to-end speed.

In this project, I integrate the ARMv7-M Keccak implementation by Alexandre Adomnicai [Ado23]. It provides two variants, *balanced* and *fast*, using techniques such as memory grouping for pipelining, bit interleaving, and lazy rotations. The fast variant achieves marginal speedups at the cost of substantially larger code size, which makes it harder to fit within my integration and code-size constraints. Hence, I use the balanced variant in our codebase.

3.2.2 Polynomial Operations

First, there are many small function calls across different files in the baseline, which introduces overhead. I therefore replace the original base multiplication call chain with a single assembly function that implements the full base multiplication path directly.

Although Cortex-M4 does not provide wide SIMD (e.g., NEON), it offers DSP instructions that enable a SIMD-like style by packing two 16-bit values into one 32-bit register. Consequently, **poly_add** and **poly_sub** use **sadd16** and **ssub16** to achieve about 2× speedup.

In addition, the compression and decompression routines for single polynomials are rewritten in assembly. The main techniques are: (1) performing bit-packing within registers to reduce redundant memory traffic, (2) using the barrel shifter to fuse shifts into logical operations, and (3) merging scaling factors into multiplications to avoid extra instructions.

3.2.3 Number Theoretic Transform(NTT)

Apart from Keccak, the NTT is one of the most important kernels in ML-KEM. The baseline implements the forward NTT using Cooley–Tukey (CT) butterflies and the inverse NTT using Gentleman–Sande (GS) butterflies, both with triple-nested loops. I optimize this kernel along two dimensions: memory traffic and arithmetic throughput.

For memory traffic, loads and stores are a major bottleneck on Cortex-M4. ML-KEM needs to do 7 layers incomplete NTT, so I apply layer fusing together with loop unrolling, organizing the 7 layers as 2+2+2+1. When fusing two layers, we can eliminate at least one full pair of load/store operations for intermediate values. In addition, I implement the inverse transform using a CT-style structure with appropriate post-processing constants so that the final output matches the baseline. This organization allows tighter control of coefficient ranges and reduces the number of explicit reductions on the critical path. The required post-processing twist factors are constants, and the final scaling can be merged into these constants to avoid extra work.

For arithmetic throughput, ML-KEM stores coefficients as 16-bit integers while registers are 32-bit. Therefore, I use **ldr**/**str** to load/store two adjacent 16-bit values at once, and perform the add/sub steps of the butterfly with **uadd16** and **usub16**. The remaining challenge

is modular multiplication and Montgomery reduction. I adopt the paired reduction strategy (double Montgomery) from [Abd+22] to reduce overhead and improve throughput.

4 Results

4.1 ECDH25519

Compared to the original C reference, which costs 54552132/51772998 cycles for fixed-base scalar multiplication and variable-base scalar multiplication, respectively, my implementation achieves 3141710/12454872 cycles. This corresponds to a 94.24% and 75.94% reduction in cycle counts, respectively. Looking into the hotspots, the multiplication routine achieves an 81.31% reduction from 9695045 → 1810245 cycles, and the squaring routine improves by 85.24% from 9695078 → 1431045 cycles. The fixed-base scalar multiplication benefits from both optimized field-arithmetic kernels and the precomputed table used by the window method.

Also, among individual finite-field routines in our codebase, **fe25519_invsqrt** is the most expensive. To preserve constant-time behavior and functional correctness, it is implemented as a fixed sequence of field operations (i.e., an addition-chain style exponentiation), leaving limited headroom for structural simplification. Hence, even though we optimize the multiplication and squaring routines, its cycle count only improves from 46332245 → 39010210, which corresponds to a 15.80% reduction.

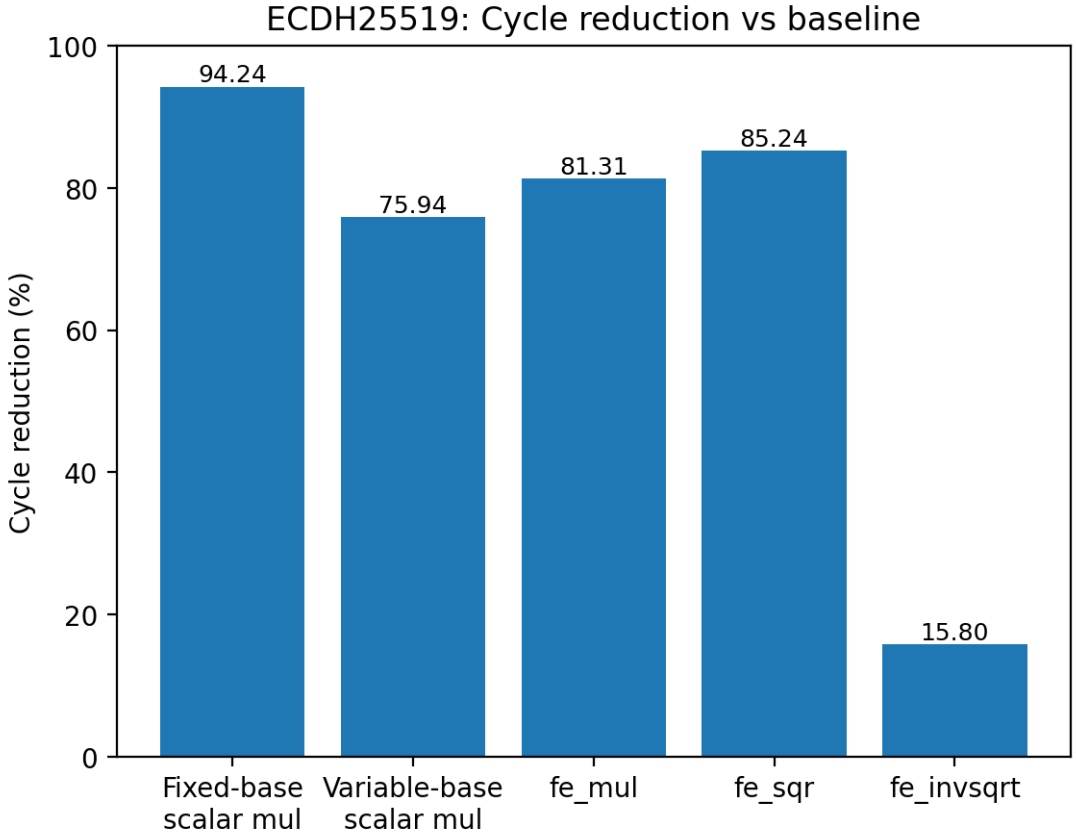

Figure 1: ECDH25519 cycle reduction relative to the original C reference

4.2 ML-KEM

In the original reference, we evaluate four parts: poly_ntt, keypair generation, encapsulation, and decapsulation. They cost 35380/1325464/1481614/1741181 cycles, respectively. With the proposed optimizations, we achieve 13399 cycles for poly_ntt (62.13%), 697377 cycles for keypair generation (47.39%), 728243 cycles for encapsulation (50.85%), and 826535 cycles for decapsulation (52.53%).

For individual functions, our forward NTT and inverse NTT improve from 23424 → 8224 (64.89%) and 32871 → 8681 (73.59%), respectively. The **poly_add** and **poly_sub** functions also improve as expected (2056 → 1045 and 2313 → 1045). On the other hand, the project requires the code size of each NTT routine to be less than 1024 bytes (excluding twiddle-factor constants). Our **ntt.S** file contains three functions (base multiplication(268 bytes), forward NTT(876 bytes), and inverse NTT(860 bytes)) with a total text size of 2004 bytes. Hence, we can observe that the code size for either NTT function may be close to the constraint but not exceed it yet. However, this restriction may induce further limitations that will be mentioned below.

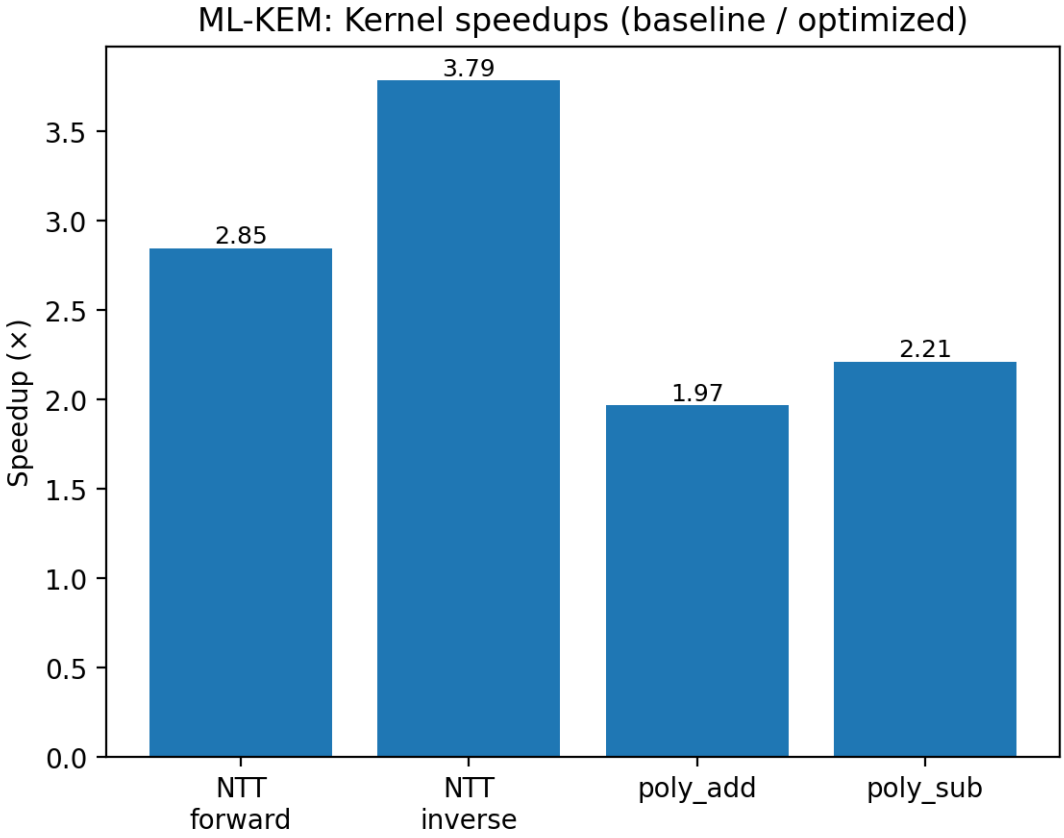

Figure 2: ML-KEM kernel speedups (baseline / optimized) on Cortex-M4

5 SOTA

5.1 ECDH25519

For ECDH25519, Lenngren presents a state-of-the-art constant-time X25519 implementation for Cortex-M4, reporting about 441k cycles on Cortex-M4 for variable-base scalar

multiplication. The main performance gains come from an all-assembly design with custom ABIs, where intermediate values are kept in registers and function-call overhead under the standard C ABI is avoided. In addition, their field arithmetic uses an 8-word representation modulo $2p = 2^{256} - 38$ and performs final reduction only at the end, which reduces redundant canonicalization steps.

In contrast, our implementation focuses on a minimally invasive optimization strategy: we hand-optimize the dominant finite-field multiplication/squaring kernels in assembly while keeping the high-level control flow in C for maintainability and easier integration. We also adopt a 25/26-bit limb representation, which differs from the 8-word representation used in the SOTA. Finally, while the SOTA work evaluates only the variable-base scalar multiplication, we additionally explore a fixed-base acceleration path (window method) to speed up basepoint multiplication in key generation, with constant-time table selection. [Len25]

5.2 ML-KEM

The state-of-the-art Kyber/ML-KEM implementation on Cortex-M4 by Huang et al. [Hua+22] is built around an improved Plantard arithmetic that can replace the Montgomery/Barrett-based modular multiplication and reduction kernels in the core polynomial arithmetic (NTT/INTT and base multiplication) on this platform. Their design is not merely an arithmetic swap, it is integrated with Cortex-M4-specific constant-time butterfly structures, packed-coefficient processing, and layer-merging strategies.

A key advantage of Plantard arithmetic is that when one operand is a constant, the modular multiplication can be implemented with two instructions on Cortex-M4. The improved arithmetic also supports signed inputs and extends the valid input range to $[-q2^{\alpha}, q2^{\alpha}]$, which simplifies handling negative coefficients and enables more aggressive lazy reduction.

The implementation further exploits the SIMD-like DSP extensions of ARMv7E-M by packing two 16-bit coefficients into one 32-bit register and applying the Plantard constant multiplication to both halfwords in parallel via `smulw{b,t}`. As a result, Huang et al. obtain compact double CT/GS butterflies and reduce the instruction count compared with Montgomery-based butterflies.

A practical trade-off is that Plantard-domain twiddle factors become $2l$-bit values (32-bit when $l = 16$), increasing the load cost. Huang et al. mitigate this overhead using more aggressive layer-merging techniques than mine, including a strategy that caches frequently reused twiddles in FP registers and uses `vmov` to reduce memory traffic.

In contrast, our work focuses on a code-size- and integration-constrained setting: we keep the Montgomery-based arithmetic backend (paired reduction) and primarily optimize memory traffic and instruction scheduling (e.g., loop unrolling and layer fusing) without changing the underlying reduction algebra. As a result, our design does not inherit the same reduction-free forward NTT advantage enabled by Plantard arithmetic, but it remains more compatible with the baseline arithmetic interfaces and our project constraints. For Kyber512 on Cortex-M4, Huang et al. report 438k/531k/479k cycles for KeyGen/Encaps/Decaps in their speed-optimized implementation (Speed[AHKS22]). Under the same parameter set, our optimized implementation takes 697,377/728,243/826,535 cycles, i.e., about 1.59×/1.37×/1.73× of the SOTA numbers. At the kernel level, their Plantard-based NTT/INTT achieves 4,474 and 4,684 cycles (Kyber512), while our forward/inverse NTT costs 8,224 and 8,681 cycles,

respectively. The remaining gap is largely explained by two design constraints in our project: (i) we keep the Montgomery-style arithmetic backend (paired reduction) rather than switching to improved Plantard arithmetic, and (ii) we enforce a strict per-routine code-size budget for NTT/INTT, which limits aggressive layer-merging and constant-caching strategies used in SOTA implementations.

6 Limitations and Future Plans

6.1 ECDH25519

First, although we use double-and-add always in the optimized implementation for scalar multiplication, it remains an open question whether a Montgomery-ladder-based approach would outperform it after a full assembly rewrite, given the ladder's highly regular control flow and potentially more favorable instruction scheduling on Cortex-M4. Second, the current hotspots (multiplication and squaring) use a radix close to $2^{25.5}$. Extending this larger-radix representation to other field-arithmetic operations may further improve end-to-end performance, but it requires careful range analysis and consistent carry handling across the full field-arithmetic stack. Finally, the multiplication routine contains long chains of multiply-accumulate operations, which can increase register pressure and cause spills to the stack. A promising direction is to experiment with using floating-point registers (FPRs) as temporary storage for frequently reused intermediates and to evaluate whether the additional **vmov** overhead can be offset by reducing memory traffic and stack spills.

6.2 ML-KEM

Due to the strict code-size constraint on the NTT routines, the overall structure of the current NTT implementation has limited headroom for further optimization. The present design already fuses at most two layers at a time and is close to the per-routine size budget. More aggressive fusion would likely increase both register spills and code size; while it could reduce additional load/store operations, it risks violating the code-size requirement and therefore may be infeasible under the project constraints.

For incremental improvements, we can implement additional loop-heavy polynomial routines in assembly to reduce call overhead and enable controlled unrolling. Moreover, it is worthwhile to audit the final assembly for redundant **mov** instructions and other avoidable data shuffles, as such overhead can be nontrivial on Cortex-M4.

References

[Abd+22] Amin Abdulrahman, Vincent Hwang, Matthias J. Kannwischer, and Amber Sprenkels. *Faster Kyber and Dilithium on the Cortex-M4*. Cryptology ePrint Archive, Paper 2022/112. 2022. URL: https://eprint.iacr.org/2022/112.

[Ado23] Alexandre Adomnicai. *An update on Keccak performance on ARMv7-M*. Cryptology ePrint Archive, Paper 2023/773. 2023. URL: https://eprint.iacr.org/2023/773.

[Hua+22] Junhao Huang, Jipeng Zhang, Haosong Zhao, Zhe Liu, Ray CC Cheung, Çetin Kaya Koç, and Donglong Chen. "Improved Plantard arithmetic for lattice-based cryptography". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems*. Volume 2022, Issue 4. IACR, 2022, pp. 614–636.

[Len25] Emil Lenngren. *Assembly optimised Curve25519 and Curve448 implementations for ARM Cortex-M4 and Cortex-M33*. Cryptology ePrint Archive, Paper 2025/523. https://eprint.iacr.org/2025/523. 2025.