FN-DSA on Arm Cortex-M55: Revisiting a Floating-Point Lattice Signature with Native FP64 and Helium

Seung-Won Lee[1 [0009−0009−9583−1387]], Min-Joo Sim[1 [0000−0001−5242−214X]], Min-Seo Kim[1 [0009−0008−1979−5778]], and Hwa-Jeong Seo[1 [0000−0003−0069−9061]]

Department of Information Computer Engineering,
Hansung University, Seoul, 02876 Korea
dkajdfhd1@gmail.com, minjoos9797@gmail.com, msqoo0212@gmail.com,
hwajeong84@gmail.com

Abstract. FN-DSA has the shortest signatures among NIST's post-quantum signature standards, but on microcontrollers its signing is slow because the binary64 arithmetic it specifies is emulated with integer instructions. Previous speedups either change the arithmetic or the order of random sampling, so their signatures no longer match the known-answer tests (KATs) of the reference implementation, or rely on double-precision SIMD instructions that microcontrollers lack. The Arm Cortex-M55 has both a double-precision FPU and the M-Profile Vector Extension (MVE). On this core, we present the first FN-DSA implementation that, unlike these approaches, reproduces the KATs of the reference and runs in constant time, yet is faster than the reference. On MVE, we vectorize the NTTs with layer merging and the fixed-point FFT of key generation with split-word Q32 arithmetic on 32-bit lanes. On the FPU, bit-exact FP64 computes the FFT, LDL decomposition, split, merge and sampler of signing without changing any result, and exponent-normalized division turns the bit-by-bit divisions of key generation into a single `VDIV.F64` with operand-independent latency. For Keccak, vectorized bit interleaving speeds up every permutation, and request-level batching processes four independent verification requests at once; SLOTHY then reschedules our MVE and Keccak assembly. Key generation, signing and verification become up to 1.23×, 1.93× and 1.30× faster than the reference on the same core, and batch verification of four signatures up to 1.75× faster. Overall, the FPU pays off where the reference emulates an operation with many integer instructions, and MVE where it already computes with integers or fixed point.

Keywords: FN-DSA · Cortex-M55 · Helium · MVE · Floating-point arithmetic · NTT · FFT · Sampler · Keccak

1 Introduction

In preparation for quantum computers, the migration to post-quantum cryptography (PQC) is well under way. Two years after NIST published ML-KEM [29],

ML-DSA [28] and SLH-DSA [30], post-quantum key encapsulation is widely deployed, whereas post-quantum signatures lag behind, although NIST, the EU and the NSA plan to complete the migration between 2030 and 2035 [26,33,31], with firmware signing among the first use cases to move [31]. However, embedded devices can hardly meet this deadline: they stay in use for a long time after deployment and the way they verify firmware is hard to change later, so devices deployed today must verify firmware and authenticate themselves beyond 2035. Such devices also have tight bandwidth, memory and energy budgets, and from this point of view FN-DSA, derived from Falcon [12], is the most suitable: FN-DSA-512 has 897-byte public keys and 666-byte signatures, smaller than the 1312 and 2420 bytes of ML-DSA-44 [28], which reduces transmission and storage, and it verifies fast.

However, on the microcontrollers widely used in embedded devices, signing with FN-DSA is slow. Signing is specified in binary64 floating point, but the Cortex-M4, the usual platform for embedded PQC, has neither a double-precision floating-point unit (FPU) nor a vector extension. The reference implementation therefore emulates binary64 with integer instructions, and signing with FN-DSA-512 still costs more than ten million cycles [34,36,39]. Previous approaches that make signing faster have so far given up the outputs of the reference. Replacing binary64 with triple-word single precision [14] or with fixed point [9] changes the results, and the vectorized signer of Zhang and Zhang, which batches samples and SHAKE streams, breaks the known-answer tests (KATs) [44]. Vectorized implementations that compute in binary64 exist [22,32], but they rely on double-precision single-instruction multiple-data (SIMD) instructions, which the vector extensions of microcontrollers lack.

Outputs that differ from the reference cause two problems. In the Falcon sampler, tiny deviations between mathematically equivalent computations can, under certain conditions, have security consequences [24]. Moreover, a change in the last bits of the inputs to the sampler occasionally changes the outcome of its rejection sampling, so an implementation that changes the floating-point operations or their order can no longer be guaranteed to reproduce the KATs [39]. Making it faster also easily breaks constant time: the execution time of hardware floating-point instructions can depend on their operands [2,16], so replacing the emulation with hardware instructions can make the signing time depend on the secret key. Speeding up FN-DSA on microcontrollers while keeping the outputs of the reference and constant time is therefore still an open problem.

The Cortex-M55 is a microcontroller that can address this problem, because it pairs a double-precision FPU with the M-Profile Vector Extension (MVE), also known as Helium, so floating point can be computed without emulation and integer arithmetic can be vectorized [3,43]. However, no work has yet used both features for FN-DSA. Applying the vectorization techniques above to the Cortex-M55 may seem enough, but they cannot be carried over as they are: MVE lacks the double-precision SIMD instructions they rely on, and batching samples or SHAKE streams changes the outputs.

This paper therefore divides the work between the two features: we vectorize the arithmetic specified on integers with MVE, and compute binary64 arithmetic on the FPU of the Cortex-M55 (FP64), reproducing every result of the reference in signing while improving performance. Beyond this, we speed up each Keccak computation and, so that no output changes, compute only independent verification requests together, four at a time, with MVE. Finally, to reduce the time that instructions spend waiting for earlier results, we reschedule them with SLOTHY.

Contributions. The contributions of this paper are as follows.

- We present the first optimized implementation of FN-DSA on the Cortex-M55. Unlike previous speedups, it reproduces the KATs of the reference and runs in constant time, yet it makes key generation, signing and verification up to 1.23, 1.93 and 1.30 times faster.
- We optimize the number-theoretic transforms (NTTs), the fast Fourier transform (FFT), the sampler and Keccak/SHAKE, which dominate key generation, signing and verification of FN-DSA-512 and FN-DSA-1024, and propose split-word Q32 arithmetic, bit-exact FP64, exponent-normalized division and request-level batching for them.
- We compute integer arithmetic with MVE and the binary64 arithmetic that the reference emulates on the FPU, choosing the faster of the two for each kernel.

The rest of this paper is organized as follows. Section 2 introduces FN-DSA, its reference implementation and the Cortex-M55, and reviews related work. Section 3 describes our measurement setup and profiles the share of each operation in the reference implementation on the Cortex-M55 to choose the operations to optimize, the feature to use for each, and the conditions that a new kernel must meet. Section 4 presents the NTTs and the key-generation FFT vectorized with MVE, Section 5 the divisions of key generation and the FFT and sampler of signing computed on the FPU, Section 6 Keccak and batch verification, and Section 7 the use of SLOTHY. Section 8 evaluates the overall and kernel performance, the effect of SLOTHY, correctness and constant time, memory, and where the time goes after optimization, and Section 9 concludes with limitations and future work.

2 Background and Related Work

This section provides the background needed to follow the rest of the paper. It introduces how the reference implementation computes the three operations of FN-DSA and the vector and floating-point features of the Cortex-M55, and then reviews related work.

2.1 FN-DSA

FN-DSA is an NTRU-lattice signature derived from Falcon that works in the polynomial ring $\mathbb{Z}_q[x]/(x^n + 1)$. The modulus q is 12289, and the degree n is 512

for FN-DSA-512 and 1024 for FN-DSA-1024 [12]. The public key h is $g/f \bmod q$, and the private key consists of small polynomials f, g, F, G that satisfy the NTRU equation $fG - gF = q$. Figure 1 shows how the reference implementation [38] computes the three operations.

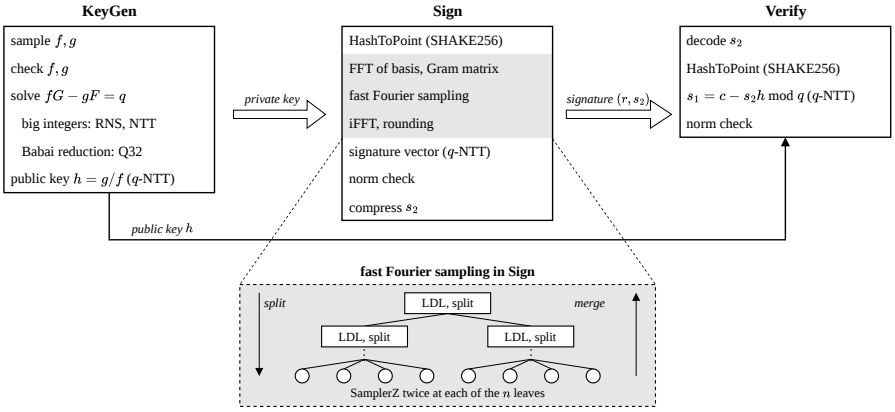

Fig. 1. Key generation, signing and verification of FN-DSA in the reference implementation. Indented steps run inside the step above them, and shaded steps are specified in binary64. The lower part expands fast Fourier sampling in signing.

Key generation samples f and g, checks that they are invertible and that their norm is small enough, and solves the NTRU equation for F and G [35,37]. The coefficients grow to thousands of bits during the solution, so the big integers are kept in a residue number system (RNS) modulo 31-bit primes, and polynomials are multiplied with an NTT modulo each prime (RNS NTT). The Babai reduction [4], which keeps the solution short, needs only an approximation, so it is computed in the Fourier domain with 64-bit fixed-point numbers that have 32 fractional bits (Q32).

Signing first goes through HashToPoint, which hashes the message to a polynomial c with SHAKE256 [27], and then finds a lattice point close to c with the private basis by fast Fourier sampling [11]. As the reference implementation stores only the small private key, each signature first moves the basis to the Fourier domain with an FFT and computes its Gram matrix. Fast Fourier sampling then descends a tree, computing an LDL decomposition and a split at each node, calls the Gaussian sampler SamplerZ at each leaf, and performs a merge on the way up that combines the results. SamplerZ draws a candidate from a fixed half-Gaussian with the base sampler and then performs a Bernoulli trial that accepts the candidate with probability $\exp(-x)$. The steps from the FFT of the basis to the inverse FFT after the sampling, and the floating-point part of SamplerZ, are specified in binary64 [20].

Verification recomputes c, computes $s_1 = c - s_2 h \bmod q$ with the NTT modulo q (q-NTT) and checks the norm of (s_1, s_2); it uses integer arithmetic only.

2.2 The Cortex-M55

The Cortex-M55 implements Armv8.1-M and has both the M-Profile Vector Extension (MVE), also known as Helium, and a double-precision FPU [3]. MVE can split its eight 128-bit vector registers $Q_0, \ldots, Q_7$ into 8-, 16- or 32-bit integer lanes or 16- or 32-bit floating-point lanes, but it has no binary64 lanes [6]. The FPU can compute binary64 arithmetic only in scalar code, and its sixteen registers $D_0, \ldots, D_{15}$ share their storage with the vector registers [3, Sect. B4.3, B5.2]. The core also has small memories attached directly to it: code can be placed in the instruction tightly coupled memory (ITCM) and data in the data tightly coupled memory (DTCM), and since these bypass the caches, every access takes the same time [6].

2.3 Related Work

We first examine how previous implementations of Falcon or FN-DSA compute their floating-point arithmetic: by integer emulation, on an FPU, with other arithmetic, or with vector instructions. We then cover the modular multiplication techniques and Helium implementations that we draw on to implement the NTTs, FFT and Keccak with MVE.

Falcon on microcontrollers. Among Pornin's integer-emulation implementations for the Cortex-M4, the 2025 version is about twice as fast as the 2019 one [34,36]. A recent variant computes some steps with modular integer arithmetic, removing more than half of the floating-point operations; it signs 21% faster on the Cortex-M4 and needs about a third of the RAM, but no longer reproduces the test vectors of the reference [39]. Howe and Westerbaan found that native binary64 on the Cortex-M7, which has a double-precision FPU but no vector extension, makes signing more than six times faster than emulation, but also that on the boards they tested the latency of every double-precision instruction depends on its operands [16]. TWFalcon replaces binary64 with triple-word single precision on the Cortex-M4F to meet the error bound of the security proof, but it is about 1.8 times slower than integer emulation [14]. Hwang formally verified the emulated arithmetic and showed that the intermediate values of the signing FFT never come close to the subnormal range [19].

Fixed point and vectorized signing. Fixed-point signing has also been proposed [9]. Vectorized implementations hold two binary64 values in each Neon register to process the Fourier arithmetic [22,32]. Zhang and Zhang vectorize signing for SSE2, AVX2, AVX-512F, Neon and RVV [44].

Modular multiplication for NTTs. Neon NTT introduced Barrett multiplication, which computes a modular product with a known constant in three instructions using a rounding multiply-high, on the Cortex-A72 and Apple M1, and noted that it also applies to MVE and is close to Shoup's multiplication [7,15]. It also observed that computing two Keccak permutations at once helps only when the SHAKE calls can be parallelized [7]. Plantard multiplication by a constant needs one multiplication fewer than Montgomery or Barrett multiplication [17]. A follow-up work widened its input range and made the Kyber NTT faster on the Cortex-M3 and RISC-V, but reported that a 32-bit modulus needs a 32×64-bit product, which removes the gain on 32-bit platforms [18].

Post-quantum cryptography on Helium. Becker et al. introduced MVE to post-quantum cryptography with NTTs and Toom–Cook multiplication for Saber, measured on an FPGA model of the Cortex-M55 [6]. SLOTHY automates instruction scheduling and register allocation for the NTTs of ML-KEM and ML-DSA and for fixed-point and single-precision FFTs on the Cortex-M55 and Cortex-M85 [1]. On the same STM32N657 as ours, Lee et al. accelerated the NTT, fixed-point FFT, Gaussian CDT and Toom–Cook multiplication of HAETAE and SMAUG-T with MVE and SLOTHY, and computed four independent SHAKE calls that recur within one operation together, with each MVE lane holding a 32-bit half of one state [23].

3 Profiling and Approach

Our objective is to minimize the cycle counts of key generation, signing and verification of FN-DSA-512 and FN-DSA-1024 on the Cortex-M55 by using the two features that the Cortex-M4 lacks, MVE and the double-precision FPU. Our baseline, M55 Reference, is the reference implementation c-fn-dsa [38] at commit `a5f1589`, built for the Cortex-M55 with its Cortex-M4 assembly enabled. We use this build as our baseline because it shows clearly how much these two features improve performance.

Setup. We measure on a NUCLEO-N657X0-Q board with an STM32N657 microcontroller [43], reusing the benchmarking environment of mlkem-native [40]: code runs from ITCM, data and stacks are in DTCM, the core runs at 800 MHz, and the caches are disabled so that timings do not depend on cache state. We compile with GCC 15.2.1 at `-O3`. Each input is timed three times after one warm-up run, including the internal retries of key generation and the rejection loop of signing, which vary with the input [36]. We take the median of the three calls, then the median over three separate loads of the firmware onto the board, and report the mean over inputs.

Where the time goes. Figure 2 breaks down the cycles of M55 Reference into five families: the NTT, the FFT, the Gaussian sampler, Keccak, and Other. Below, numbers are for FN-DSA-512, with FN-DSA-1024 in parentheses. Signing

spends 65.6% (68.3%) of its time in the FFT, 16.2% (15.0%) in the sampler, 13.9% (12.4%) in Keccak and only 2.9% (2.9%) in the NTT. Verification spends 68.2% (64.2%) in Keccak and 20.7% (24.3%) in the NTT, and uses no FFT. Key generation is different: the NTT and the FFT take 38.9% (31.5%), while Other takes 54.6% (61.9%). Other consists of the big-integer arithmetic with which the NTRU solver finds F and G in key generation, and of decoding, format conversion and norm checks in signing and verification. We leave Other unchanged and limit this work to the four families above. The big-integer arithmetic is also integer code that MVE could process across coefficients; we leave this to future work.

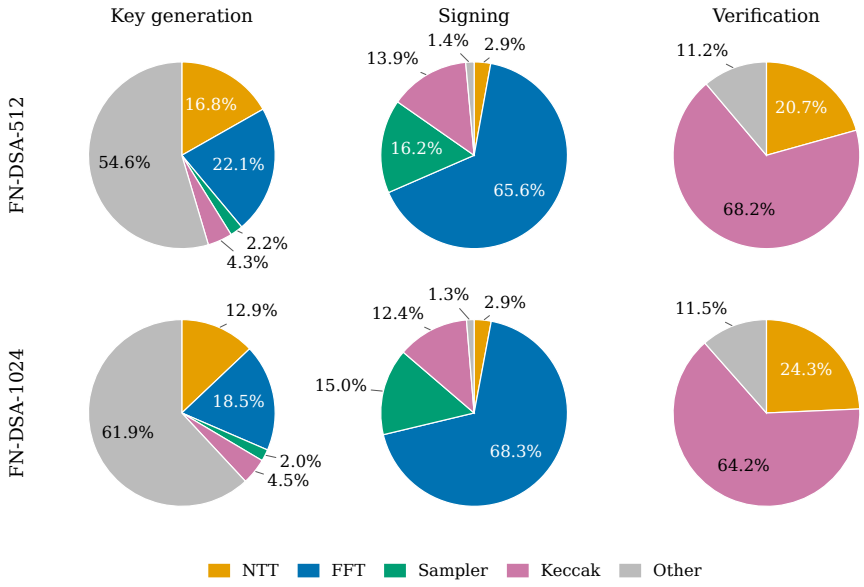

Fig. 2. Profile of the cycle shares of key generation, signing and verification in M55 Reference on the Cortex-M55.

So, we choose the arithmetic of each kernel from the specification. Where FN-DSA is specified on integers, as in the NTT and Keccak, we vectorize the code with MVE instructions. In contrast, where it is specified in floating point but the reference computes with integer instructions on the Cortex-M4, as in the FFT and the sampler of signing, we restore floating point on the double-precision FPU; this code stays scalar because MVE has no binary64 lanes. The one exception is the Fourier arithmetic of key generation: it is specified in floating point, but we keep the fixed point of the reference for reasons given in Section 4.1. We write each kernel by hand in assembly and C and evaluate its performance before and after rescheduling it with SLOTHY, where SLOTHY supports it. However, we

adopt a kernel only if it passes five checks: it makes the whole operation faster; its intermediate values may differ from the reference, but never enough to change the outcome of a rounding, a comparison or a rejection; all KATs still match; valid signatures are accepted and tampered ones are rejected; and it runs in constant time. For constant time, we inspect the machine code of every kernel and test every kernel that handles secret data on the board with dudect [41]. Like the reference, we do not address power, electromagnetic or fault attacks.

What we optimize. Tables 1 to 3 split each family of Figure 2 further and show the share of every operation in it. Next to each operation we give the method we use to optimize it, MVE or FP64; a dash marks an operation we leave unchanged. We chose the operations to optimize in order of their share.

Table 1. Key generation in M55 Reference: share of each operation within its family and the method we use.

Operation	512 / 1024	Method
NTT (16.8% / 12.9% of key generation)		
RNS forward NTT	43.0% / 40.9%	MVE
RNS inverse NTT	38.7% / 38.7%	MVE
RNS twiddle tables	13.7% / 13.5%	—
q-NTT	4.4% / 6.7%	MVE
Division in NTT form	0.2% / 0.1%	—
FFT (22.1% / 18.5% of key generation)		
Self-adjoint division, inverse norm	27.6% / 28.8%	FP64
Q32 inverse FFT	26.3% / 28.1%	MVE
Q32 FFT	24.6% / 25.5%	MVE
Big integer to Q32	10.4% / 9.7%	MVE
Q32 reciprocal	6.7% / 3.9%	MVE
Point-wise multiplication	4.4% / 4.0%	—
Sampler (2.2% / 2.0% of key generation)		
Sampling f and g	100% / 100%	—

Excluding Other, the NTT and the FFT take 38.9% (31.5%) of key generation, against 4.3% (4.5%) for Keccak and 2.2% (2.0%) for the sampler, so we optimize them first. In the NTT, the forward and inverse transforms over the RNS primes take 81.6% (79.7%) of the family. The q-NTT takes a large share of verification, so optimizing it benefits key generation, signing and verification alike. The generation of the RNS twiddle tables takes 13.7% (13.5%) of the family, comparatively little next to the RNS transforms, and we leave it unchanged. In the FFT, the largest operation is not a transform but the self-adjoint division and the inverse norm, at 27.6% (28.8%); together with the transforms, the conversion of big integers to Q32 and the Q32 reciprocal, the operations we optimize take 95.6% (96.0%) of the family. However, the FFT is the exception: it is specified in floating point,

but keeping the fixed point of the reference and vectorizing it with MVE was faster than binary64, so we optimize it with integer MVE. Only the self-adjoint division and the inverse norm ran faster with FP64 and use the FPU. Sections 4 and 5 explain why. We leave the sampling of f and g, which is integer code and takes 2.2% (2.0%) of key generation, because its share is comparatively small. Keccak/SHAKE takes 4.3% (4.5%) of key generation and becomes faster too, because key generation calls the same permutation that we optimize for verification.

Table 2. Signing in M55 Reference: share of each operation within its family and the method we use.

Operation	512 / 1024	Method
FFT (65.6% / 68.3% of signing)		
FFT	26.4% / 26.6%	FP64
LDL decomposition	16.5% / 16.6%	FP64
Split	11.7% / 11.8%	FP64
Merge	11.1% / 11.1%	FP64
Inverse FFT	11.0% / 11.0%	FP64
Point-wise multiplication	7.8% / 7.7%	—
Split of self-adjoint polynomials	4.5% / 4.6%	—
Gram matrix	3.7% / 3.4%	—
Subtraction	3.2% / 3.2%	—
Addition	3.0% / 3.0%	—
Other helpers	1.0% / 0.9%	—
Sampler (16.2% / 15.0% of signing)		
Next sample (centre, scaling)	31.7% / 31.5%	FP64
Deepest level of the tree	27.9% / 28.2%	FP64
Bernoulli trial on $\exp(-x)$	15.3% / 15.2%	FP64
Exponential polynomial	11.2% / 11.1%	—
Base sampler	8.4% / 8.3%	—
Recursion control	5.5% / 5.7%	—

In signing, the FFT is the largest family at 65.6% (68.3%). Within it, the FFT, the LDL decomposition, split, merge and the inverse FFT take 76.7% (77.2%). The remaining point-wise helpers take 23.3% (22.8%) of the family, but each of them takes at most 7.8%, and we leave them unchanged. In the sampler, 16.2% (15.0%), the three routines that compute in binary64 take 75.0% (74.9%) and are computed in FP64 as well, while the integer part stays as it is. Keccak/SHAKE, 13.9% (12.4%) of signing, and the NTT, 2.9% (2.9%), become faster too, because signing calls the same permutation and q-NTT that we optimize for verification.

Verification contains only two families, Keccak at 68.2% (64.2%) and the NTT at 20.7% (24.3%), and both are large, so we optimize both.

Table 3. Verification in M55 Reference: share of each operation within its family and the method we use.

Operation	512 / 1024	Method
Keccak (68.2% / 64.2% of verification)		
Permutation	85.5% / 85.8%	scalar; MVE in batches
Bit interleaving	12.7% / 12.8%	MVE
Sponge control	1.7% / 1.5%	—
NTT (20.7% / 24.3% of verification)		
Inverse q-NTT	49.0% / 49.3%	MVE
q-NTT	43.0% / 43.3%	MVE
Point-wise multiplication in NTT form	8.0% / 7.3%	—

4 Integer on MVE

The NTT and the FFT both use the Cooley–Tukey algorithm [8], which halves the array repeatedly. Each layer therefore computes butterflies on pairs of values at distance d, and d halves from one layer to the next; the inverse transforms run the layers backwards, doubling d. Following earlier work on Helium [6,1], we map this structure onto MVE as shown in Figure 3. Table 4 defines the symbols of the figure and the text and gives their values for the three transforms.

Table 4. Notation of Figure 3 and its values for the three transforms.

Symbol	Meaning	q-NTT	RNS NTT	Q32 FFT
w	bits per lane	16	32	32
$L = 128/w$	lanes per vector	8	4	4
d	distance between the inputs of a butterfly	$n/2, \ldots, 1$	$n/2, \ldots, 1$	$n/4, \ldots, 1$
$d < L$	layers handled as in (b)	$d = 4, 2, 1$	$d = 2, 1$	$d = 2, 1$
k	layers merged in registers	≤ 3	2	1
ω	twiddle factor	mod q	mod p	complex Q32

An MVE instruction combines only corresponding lanes of two vectors, so the two inputs of a butterfly must sit in the same lane of different vectors. If $d \geq L$, as in panel (a) of Figure 3, loading the coefficients into two vectors in memory order already aligns the pairs, such as 0 and 4 or 1 and 5, and since every instruction acts on all lanes, L butterflies are computed at once. If $d < L$, as in panel (b), loading in order puts a pair, such as 0 and 1, into one vector. Helium has no instruction that permutes the lanes of a vector, so loads that spread the coefficients, such as `VLD4`, align the pairs; the butterflies are then computed as in panel (a), and `VST4` stores the results in memory order. As in panel (c), the two NTTs also compute several layers in registers before storing, which reduces memory traffic; the eight vector registers bound the number k of merged layers.

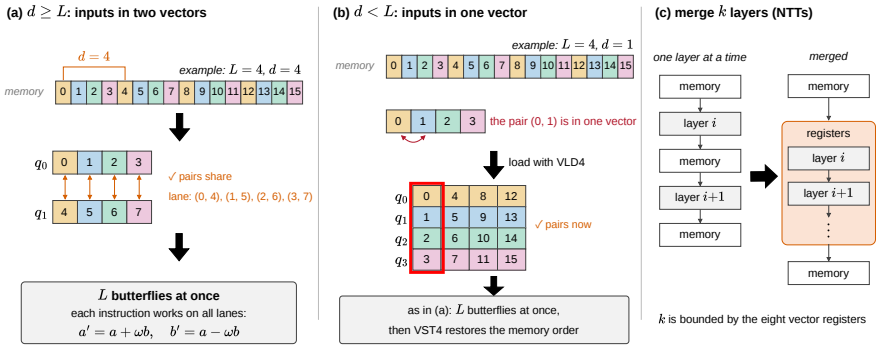

Fig. 3. Mapping the butterflies of the NTTs and the Q32 FFT onto MVE for $L = 4$: **(a)** $d \geq L$, **(b)** $d < L$ with `VLD4`/`VST4`, **(c)** k layers merged in registers.

Inside each lane, products are assembled from their high and low halves, and value-dependent fix-ups use predicated additions instead of branches. The lane counts, multiplications and layer groupings of the three transforms are given in Table 4 and in the subsections.

4.1 Key Generation

Layer merging for the RNS NTT. Key generation multiplies big integers in RNS form and computes an NTT modulo each prime below 2^{31}. M55 Reference computes these transforms in scalar C; we apply the mapping of Figure 3 to the forward and inverse transforms for $\log n \geq 4$. Since a vector holds four 32-bit coefficients, the layers with $d \geq 4$ follow panel (a), the last two layers use `VLD4`/`VST4` as in panel (b), and we merge layers in pairs. For $\log n < 4$, there are fewer than 16 coefficients, which cannot fill the 4×4 tiles that `VLD4`/`VST4` handle, so these small transforms keep the C code. The multiplication by a twiddle factor is a rounding Montgomery multiplication [25,7], computed with one `VMUL`, two `VQRDMULH`, one `VHADD` and a predicated fix-up. The inverse transform does not halve in the intermediate layers and instead folds the factor $1/n$ into the last layer, so it is applied only once. We also tried Plantard multiplication [17], which saves one multiplication but needs a 32×64-bit product that MVE lanes do not provide; it made key generation 0.34% (0.22%) slower, so we did not adopt it.

Split-word Q32 FFT. M55 Reference computes the FFTs of the Babai reduction of key generation in Q32 fixed point with scalar C code; we apply the mapping of Figure 3 to the transforms for $\log n \geq 4$. Unlike the coefficients of the NTTs, a Q32 value has 64 bits and does not fit into a 32-bit lane. We therefore put the low and high 32-bit halves of each value into different vectors with `VLD2` and build the 64-bit arithmetic from 32-bit instructions: a predicated addition propagates the

carry, and products are assembled from 32-bit partial products computed with `VMUL` and `VMULH`. As a result, every value takes two vectors, and the inputs of four butterflies already fill the eight vector registers, so we do not merge layers. The arithmetic is that of M55 Reference, so the results are identical bit for bit and the KATs still match. Since the specification defines these operations in floating point, we also tried computing them in floating point instead of Q32. Binary64 was slower, because its additions and multiplications are slow. Representing values as sums of binary32 numbers on floating-point MVE was slower as well: triple-word arithmetic with three binary32 values [14] was 8.1–9.7 times slower than Q32, and double-single arithmetic with two [10] made key generation slower because of the conversions to and from Q32. We therefore use floating point only at the top level of the NTRU solver, where these conversions do not occur.

4.2 Verification

Layer merging for the q-NTT. All three operations use the NTT modulo $q = 12289$, and verification spends the largest share of its time in it. M55 Reference packs two 16-bit coefficients into a 32-bit register and computes two butterflies at a time; we apply the mapping of Figure 3 to the forward and inverse transforms, with eight 16-bit coefficients per vector. The multiplication by a twiddle factor is a Barrett multiplication [5,7] of three instructions, `VQRDMULH`, `VMUL` and `VMLA`; the twiddle factor and its Barrett constant share one general-purpose register, low and high half, and enter scalar-vector instructions, so they occupy no vector register. M55 Reference uses Montgomery multiplication. Applied with MVE in the representation of the Cortex-M4 code, it takes 17 instructions per product, six of them multiplications, whereas Barrett multiplication takes six instructions including one correction, three of them multiplications, and made verification 1.4% (2.8%) faster, so we switched to Barrett. We merge layers 2+3+2 for FN-DSA-512 and 2+2+2+2 for FN-DSA-1024. For FN-DSA-512, 2+3+2 made verification 2.1% faster, against 0.4% for 2+2+2+1 and a 0.6% slowdown for 2+2+3; for FN-DSA-1024, merging three layers was 0.5% slower, so we merge pairs. With eight lanes, three layers have $d < L$: for $d = 4$, each vector is loaded as two 64-bit halves from two different groups, with the twiddle factors differing between the halves, and $d = 2, 1$ use `VLD4`/`VST4` as in panel (b).

5 Binary64 on the FPU

M55 Reference emulates every 64-bit floating-point operation of signing with integer instructions. The emulation rounds as the IEEE 754 binary64 rules [20] prescribe, and signing never produces subnormals [36,19]. The double-precision FPU of the Cortex-M55 computes the same operations in single instructions, and its results are bit-identical if no multiplication is fused with an addition, rounding is to nearest-even, and the order of operations and the rules for integer conversion follow M55 Reference. We therefore implement the time-consuming kernels in assembly with binary64 instructions; the remaining point-wise helpers keep the emulation of M55 Reference.

5.1 Key Generation

Exponent-normalized division. In the Fourier arithmetic of key generation, the self-adjoint division and the inverse norm run in FP64. The inverse norm $1/(|f|^2+|g|^2)$ is used to check the orthogonalized norm of a candidate (f, g), and the self-adjoint division reduces (F, G) at the top level of the NTRU solver. Since the Cortex-M4 has neither binary64 instructions nor a 64-bit division instruction, M55 Reference computes the divisions of both operations with integer instructions, in a loop of 64 iterations that each produce one bit of the quotient; we compute each with a single `VDIV.F64`. The loop executes about 1,580 instructions per division, while ours takes at most 148, `VDIV.F64` included. However, `VDIV.F64` alone could break constant time, because the execution time of floating-point instructions can depend on their operands [2,16]; indeed, we found that `VDIV.F64` on the Cortex-M55 finishes earlier when an operand is zero. To avoid this, we set the exponents of both operands to zero, so that `VDIV.F64` only sees values in $[1, 2)$, and rebuild the exponent of the result exactly from the difference of the exponents. No branch is then needed for zeros, subnormals or other exceptional inputs, and the execution time of `VDIV.F64` no longer depends on the input. We also computed the FFT and inverse FFT of key generation in FP64 in the same way, but they were 1.22–1.30 times slower than M55 Reference: the FPU of the Cortex-M55 does not pipeline binary64 additions and multiplications, so a binary64 butterfly takes longer than a Q32 butterfly built from integer multiplications. In key generation, we therefore use FP64 only for the inverse norm and the self-adjoint division.

5.2 Signing

Bit-exact FP64 for the FFT. For every addition and multiplication in the FFT-family routines, M55 Reference calls a function that emulates it with integer instructions, using 67 instructions per addition and 33 per multiplication. M55 Reference keeps each value in a 64-bit integer variable, but its bits are already in binary64 format. We therefore load these bits from memory straight into the double-precision registers with `VLDR`, without any conversion, and compute additions and multiplications with `VADD.F64` and `VMUL.F64`, which reduces the number of instructions. The complex multiplications by twiddle factors in the FFT, the inverse FFT, split and merge, and the multiply-and-add steps of the LDL decomposition, use `VMLA.F64` and `VMLS.F64`; like M55 Reference, these instructions round the product first and then round the sum or difference again, so they produce the same results as M55 Reference. In addition, our assembly merges layers of the FFT in the same way as Section 4.2, to reduce memory traffic. Computed one layer at a time, every layer loads all values from memory and stores them back. To avoid this, we load the four complex values that two consecutive layers combine into the registers, compute the two butterflies of the first layer and then the two of the second, and store the values only once. The number of layers we can merge is set by the number of registers, because every value used while the layers are computed must stay in a register. Two layers use eight complex values, the four values being transformed, three twiddle factors and

one intermediate product, and each complex value needs two registers, sixteen in total; the Cortex-M55 has exactly sixteen double-precision registers, so we can merge two layers. This only eliminates the stores and loads between the two layers; the additions and multiplications stay the same, so the results match M55 Reference. The inverse FFT is computed in the same way.

Bit-exact FP64 for the sampler. The sampler is called in every signing attempt, and each call mixes 64-bit floating-point and integer operations. The floating-point operations happen in three routines: the step that splits the centre of each sample into integer and fractional parts, the Bernoulli trial that decides whether to keep a sample, and the deepest level of the recursion, which computes the standard deviations and draws four samples. As described above, M55 Reference calls an emulation function for each of these operations. We compute the multiplications, subtractions, divisions and square roots in the three routines each with a single binary64 instruction, `VMUL.F64`, `VSUB.F64`, `VDIV.F64` or `VSQRT.F64`. This replaces the 33, 439 and 597 instructions that M55 Reference spends on a multiplication, a division and a square root with one instruction each. The sampler draws, sample by sample, as many pseudorandom bytes as each sample needs. Drawing the randomness of several samples at once for speed changes the bytes each sample receives, and with it the signature [44]. Unlike Zhang and Zhang, we therefore draw the randomness one sample at a time, as M55 Reference does, which keeps the signatures and the KATs unchanged.

6 Keccak/SHAKE

In FN-DSA, each use of SHAKE256 is one sequential stream. Its permutations depend on each other, so batching, as in four-way Keccak, cannot by itself speed up a single stream. Splitting the stream into several independent streams and computing them together changes the outputs; this is one reason why the signer of Zhang and Zhang does not reproduce the KATs [44]. We therefore make each permutation faster within one operation, and use four-way Keccak only across independent requests.

6.1 Single-State Keccak

The Cortex-M4 assembly of the reference stores each 64-bit lane as two 32-bit words, one with the even-indexed bits and one with the odd-indexed bits [36]. If a lane were simply cut in half, every 64-bit rotation would move bits between the two words and cost many instructions; with this split, it is just two 32-bit rotations. We start from this form and change two things.

Vectorized bit interleaving. First, this form requires the bits of each input block to be split into even and odd bits before absorbing, and merged back when the output is extracted. We do this repetitive work with the MVE instructions of the Cortex-M55, four 32-bit words at a time, which cuts the instructions that convert four lanes from 120 to 47.

Load scheduling. Second, the 50 words of the state do not fit into the integer registers, so M55 Reference keeps 32 of them in FP registers and 18 on the stack and reads and writes them in every round. This has a drawback: an instruction that uses a word right after reading it must wait until the read completes. We therefore change four places to reduce instructions and waiting. In θ and π, instead of reading one word and computing with it at once, we read two words first and then compute, so that the computation on the first word proceeds while the second word is read. In χ, we read the two words of each lane on the stack with one doubleword load instead of two word loads. In ι, M55 Reference recomputes the address of the round constant from the round counter in every round, whereas we keep a pointer to the constants in a register and advance it with each read. Since every SHAKE256 computation calls this permutation, these techniques speed up key generation, signing and verification alike.

6.2 Request-Level Batching

A device often verifies several signatures at once, for example a certificate chain. We therefore bundle four signature-verification requests and compute their Keccak permutations at once with MVE. The requests are independent of each other, so bundling them does not change any output. Each of the four 32-bit lanes of an MVE vector holds the state of one request. We keep each 64-bit Keccak lane as its low and high 32-bit halves, so, unlike the single-state form above and the MVE Keccak of mlkem-native [40], the input and output need no bit conversion. When fewer than four requests remain, however, we use single-state Keccak. We also tried bundling four key-generation or signing requests, but it was mostly slower: rejection sampling makes each request consume a different amount of randomness, so the four soon fall out of step, and managing their separate random streams costs more than the bundled permutations save. We therefore bundle requests only for verification.

7 SLOTHY

We apply SLOTHY [1,42], with its public Cortex-M55 model and without modification, to those of our assembly files that contain only instructions the model supports: the q-NTT, the RNS NTT, the key-generation FFT, and the single-state and four-way Keccak. The four binary64 files of signing are outside its reach, because the model has no scalar floating-point registers and no binary64 instructions [42]. SLOTHY treats unsupported instructions, calls and conditional execution as boundaries and reorders instructions and reallocates registers in the straight-line code between them. It software-pipelines whole loops only in the four-way Keccak, its round and squeeze loops; the loops of the NTTs and the FFT contain unsupported instructions, so only their straight-line parts are rescheduled. Because SLOTHY only reorders instructions and renames registers and keeps the order of memory accesses, it adds no branches and preserves constant time [1]. Since the rescheduled code is still new code, we applied the five checks of Section 3 to it again, and it passed all of them.

8 Evaluation

All measurements follow the setup of Section 3.

8.1 Overall Performance

Our implementation makes all three operations faster, signing the most. Table 5 compares it with M55 Reference. Signing becomes 1.89 and 1.93 times faster for FN-DSA-512 and FN-DSA-1024, because the FFT and the sampler, whose binary64 arithmetic we replaced, take most of its time. Verification becomes 1.28 and 1.30 times faster. Batch verification becomes 1.57 and 1.62 times faster, which is also 1.22 and 1.25 times faster than four of our single verifications. Key generation gains the least, 1.23 and 1.17 times, because more than half of its time goes to the big-integer arithmetic of the NTRU solver, which lies outside the scope of our optimizations.

Table 5. Cycles per call of M55 Reference and of our implementation before SLOTHY. Speedups are given as the mean ± the half-width of the 95% confidence interval over paired inputs (100 inputs; 25 batches for batch verification). For batch verification, M55 Reference verifies the four requests one after another.

	Operation	M55 Reference	Ours	Speedup
FN-DSA-512	Key generation	63,700,895	51,948,889	1.226 ± 0.006
	Signing	17,585,038	9,284,418	1.894 ± 0.001
	Verification	387,364	301,581	1.284 ± 0.002
	Batch verification	1,549,210	985,475	1.572 ± 0.008
FN-DSA-1024	Key generation	265,266,692	226,615,048	1.171 ± 0.007
	Signing	37,693,162	19,548,668	1.928 ± 0.001
	Verification	755,559	579,830	1.303 ± 0.001
	Batch verification	3,021,991	1,862,330	1.623 ± 0.004

8.2 Kernel Performance

Most of the kernels we optimized become three to six times faster. Table 6 compares each of them with M55 Reference. The binary64 kernels of signing become 3.1 to 3.3 times faster, and the LDL decomposition of signing and the self-adjoint division of key generation 5.9 and 4.9 times, because each of their divisions becomes a single `VDIV.F64`. The RNS NTTs become 3.3 to 4.0 times faster and the q-NTTs 2.8 to 3.3 times. The following kernels gain less. The Q32 FFTs gain 1.7 to 1.9 times, because each 64-bit value takes two vectors and the layers cannot be merged. The sampler gains 1.2 to 1.5 times, because its base sampler, its exponential polynomial and the PRNG stay as in M55 Reference. The Keccak permutation gains 1.17 times, because a single state does not fill the

vector lanes and its permutation stays scalar. Since the kernels cover only part of each operation, their gains shrink in Table 5, least in signing, where they take the most time.

Table 6. Cycles per call of each optimized kernel in M55 Reference (Ref.) and in our implementation before SLOTHY.

	FN-DSA-512			FN-DSA-1024		
Kernel	Ref.	Ours	Speedup	Ref.	Ours	Speedup
Key generation						
RNS NTT	56,549	17,089	3.309	123,488	36,950	3.342
RNS inverse NTT	64,576	16,457	3.924	141,332	35,431	3.989
Self-adjoint division	901,650	184,876	4.877	1,803,282	369,708	4.878
Inverse norm	464,159	117,821	3.940	928,287	235,581	3.940
Q32 inverse FFT	63,730	37,795	1.686	143,312	83,599	1.714
Q32 FFT	55,839	29,699	1.880	125,889	64,851	1.941
Big integer to Q32	10,996	3,310	3.322	21,876	6,542	3.344
Q32 reciprocal	7,055	2,133	3.308	7,055	2,133	3.308
Signing						
FFT	602,831	187,163	3.221	1,353,522	421,671	3.210
LDL decomposition	236,060	39,688	5.948	472,092	79,368	5.948
Split	84,198	27,417	3.071	168,358	54,809	3.072
Merge	75,686	23,870	3.171	151,334	47,678	3.174
Inverse FFT	626,020	191,234	3.274	1,403,137	429,606	3.266
Next sample	3,651	2,958	1.234	3,480	2,817	1.235
Deepest level	17,691	12,286	1.440	17,147	11,822	1.450
Verification						
Permutation, bit interleaving	14,437	12,374	1.167	14,437	12,374	1.167
Inverse q-NTT	38,623	12,862	3.003	85,609	31,018	2.760
q-NTT	33,863	10,297	3.289	75,086	25,597	2.933

8.3 Effect of SLOTHY

SLOTHY makes a clear difference only in batch verification. Table 7 compares our implementation before and after SLOTHY. It removes 7.5% and 7.4% of the cycles of batch verification, because it software-pipelines the whole round and squeeze loops of the four-way Keccak. The other operations gain at most 0.35%: in the NTTs and the key-generation FFT, SLOTHY reschedules only the straight-line code between unsupported instructions, and signing runs mostly binary64 arithmetic that it cannot handle.

Table 7. Cycles per call of our implementation before and after SLOTHY. Speedups as in Table 5.

	Operation	Before	After	Speedup
FN-DSA-512	Key generation	51,948,889	51,766,744	1.0035 ± 0.0002
	Signing	9,284,418	9,281,366	1.0003 ± 0.0000
	Verification	301,581	301,159	1.0014 ± 0.0000
	Batch verification	985,475	911,994	1.0806 ± 0.0014
FN-DSA-1024	Key generation	226,615,048	226,104,941	1.0023 ± 0.0002
	Signing	19,548,668	19,543,950	1.0002 ± 0.0000
	Verification	579,830	579,246	1.0010 ± 0.0000
	Batch verification	1,862,330	1,723,652	1.0805 ± 0.0007

8.4 Correctness and Constant Time

Correctness. All API outputs of our implementation are those of the reference. For each degree, key generation matches 200 KAT vectors and signing 10; verification accepts all valid signatures and rejects all 1,609 inputs with a tampered signature, key, message or context and all 352 malformed inputs. Below the API, we compared the output of every kernel call with that of the reference on 2,303 inputs captured from the API, and all kernels except the self-adjoint division of key generation agreed bit for bit. The self-adjoint division computes in binary64 instead of Q32, so its output differs by a relative error of at most $1.5 \cdot 10^{-10}$, but the generated keys were the same.

Constant time. To check constant time, we first inspected the machine code of every new kernel for branches, conditional execution and memory addresses that depend on secret values, and replaced the conditional execution we found in the rounding of key generation with integer bit masks. We then ran each kernel on fixed and random inputs, interleaving the two classes, and compared their timings with Welch's t-test [41]. All kernels pass the usual threshold of $|t| < 4.5$: over 428 conditions of the production kernels, the largest $|t|$ was 3.51 before and 2.95 after SLOTHY, and over 14 conditions of the sampler it was 2.34 after accounting for the number of rejection iterations. A stricter criterion, which records any differing cycle value and any instruction latency that depends on the operand class, yields two findings. First, `VDIV.F64` finishes in 6 cycles when an operand is zero, and `VSQRT.F64` when its operand is zero or negative; otherwise they take 32 and 31 cycles. However, key generation normalizes the exponents so that `VDIV.F64` never sees zero, and signing only passes positive values derived from the Gram matrix, so neither uses this fast path. Second, some conditions occasionally took one more cycle; we did not find the instruction responsible, but it also occurred in a control that executes identical instructions on identical operands, and it was not associated with the input class.

8.5 Memory

Memory was not a target of our optimizations; for reference, Table 8 lists code size, constants and peak stack usage per operation. Our implementation needs 1.5 to 3.3 times the code and 1.4 to 3.0 times the constants of M55 Reference. Its stack usage is similar in signing and verification and grows from 1,160 to 9,408 bytes in key generation. SLOTHY only adds 2,460 bytes of code to batch verification.

Table 8. Code size, constants, and peak stack usage (bytes) of the M55 Reference and our implementation before SLOTHY. The 512 and 1024 signing rows denote FN-DSA-512 and FN-DSA-1024, respectively; all other rows apply to both variants. Const. denotes constants.

	M55 Reference			Ours		
Operation	Code	Const.	Stack	Code	Const.	Stack
Key generation	38,588	30,086	1,160	75,736	63,526	9,408
Signing (512)	21,052	20,701	1,264	32,604	28,893	1,232
Signing (1024)	21,052	20,701	1,280	32,604	28,893	1,248
Verification	7,052	4,141	592	18,156	12,333	608
Batch verification	7,052	4,141	592	22,928	12,525	1,688

8.6 Where the Time Goes Now

Figure 4 profiles our final implementation, after SLOTHY, in the same way as Figure 2. Below, numbers are for FN-DSA-512, with FN-DSA-1024 in parentheses, and changes are in percentage points relative to Figure 2. The families we optimized shrink: in key generation, the NTT by 8.0 (6.5) points and the FFT by 5.3 (4.6) points; in verification, the NTT by 10.8 (11.6) points; and in signing, the FFT by 9.2 (9.0) points. The families we did not optimize, or that gained less, grow instead: Other grows by 12.6 (10.8) points to 67.2% (72.7%) of key generation, and Keccak grows by 8.2 (7.8) points in signing and by 7.5 (8.1) points in verification. The FFT of signing shrinks by only 9.2 points because, although the kernels we optimized become 3.4 times faster, the point-wise helpers that we left unchanged now make up half of the family.

9 Conclusion

We present the first optimized implementation of FN-DSA on the Cortex-M55. It uses the two features that the Cortex-M4 lacks, MVE and the double-precision FPU, each where it fits. MVE carries the NTTs and the Q32 FFT of key generation, and the FPU carries the FFT and the sampler of signing and the

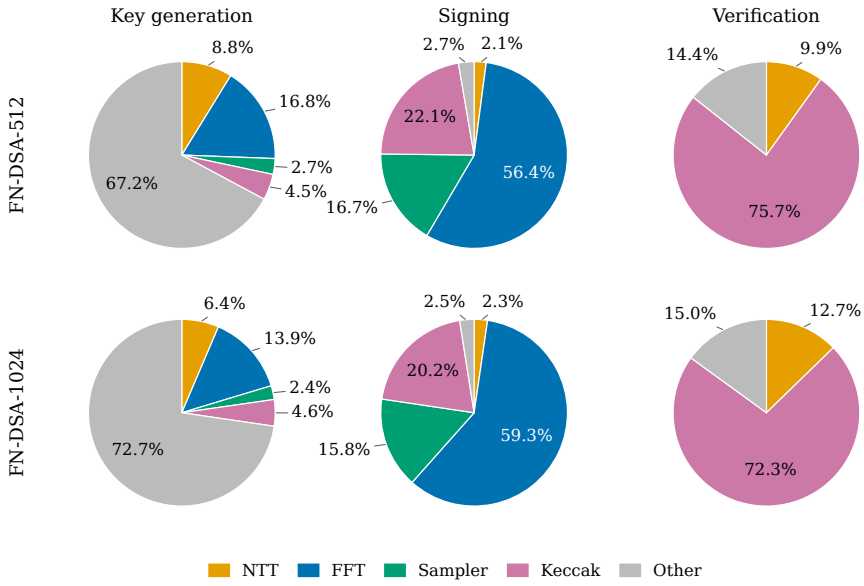

Fig. 4. Profile of the cycle shares of key generation, signing and verification in our final implementation, after SLOTHY, on the Cortex-M55.

divisions of key generation. The FPU is faster where the reference emulates an operation with many integer instructions: binary64 arithmetic in signing and bit-by-bit division in key generation. MVE is faster where the reference already computes with integer or fixed-point instructions, including the FFT of key generation, which the specification defines in floating point. This shows that neither feature fits everywhere. As a result, compared with M55 Reference on the same core, key generation becomes 1.23 and 1.17 times faster for FN-DSA-512 and FN-DSA-1024, signing 1.89 and 1.93 times, verification 1.28 and 1.30 times, and batch verification of four requests 1.57 and 1.62 times; SLOTHY further raises batch verification to 1.70 and 1.75 times. All API outputs are those of the reference, and we observe no case in which the execution time of the code we changed depends on secret data.

Limitations. Our work has the following limitations. First, all results come from one board with the caches disabled, code in ITCM and data in DTCM, compiled with GCC 15.2.1; running from flash with the caches enabled or using another compiler may change the cycle counts. Second, most of the time of key generation goes to the big-integer arithmetic of the NTRU solver, which we do not optimize, so key generation becomes only 1.23 and 1.17 times faster. The point-wise helpers of signing and the integer part of the sampler also remain those of the reference. Third, to produce the same outputs as the reference, we use

neither fused multiply-add, nor a different order of operations, nor drawing the randomness of several samples at once, nor parallel SHAKE streams; four-way Keccak, too, applies only to batch verification with at least four independent requests. Fourth, we evaluate timing leakage only; power and electromagnetic analysis are outside our scope [21,13,45]. Within timing, we do not prove that the Gram values of signing stay positive after rounding for every valid key, we do not test the unchanged helper functions of the reference or Inf and NaN inputs, and we do not yet know the cause of the rare one-cycle variations.

Future work. Future work has four directions. The first is to reduce memory and code size. Our implementation needs 1.5 to 3.3 times the code of M55 Reference and, in key generation, 9,408 bytes of stack instead of 1,160; reducing them to fit devices with small RAM and flash remains to be done. The second is to apply SLOTHY more widely. Adding scalar binary64 instructions to the public Cortex-M55 model would bring the signing kernels within its reach, and adding the unsupported instructions in the loops of the NTTs and the FFT would let it software-pipeline those loops as a whole. The third is to resolve the open points of the constant-time tests by adjusting our techniques. Normalizing the exponents of the operands of `VDIV.F64` and `VSQRT.F64` in signing, as in key generation, would rule out the fast path without assuming that the Gram values are positive, and we look for the cause of the rare one-cycle variations by varying code placement and alignment. The fourth is to vectorize the big-integer arithmetic of the NTRU solver across coefficients with MVE, which now takes 67.2% and 72.7% of key generation for FN-DSA-512 and FN-DSA-1024.

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. Andrysco, M., Kohlbrenner, D., Mowery, K., Jhala, R., Lerner, S., Shacham, H.: On subnormal floating point and abnormal timing. In: 2015 IEEE Symposium on Security and Privacy. pp. 623–639. IEEE Computer Society (2015). https://doi.org/10.1109/SP.2015.44
3. Arm Limited: Armv8-M Architecture Reference Manual (Feb 2026), document number DDI0553, issue B.z
4. Babai, L.: On Lovász' lattice reduction and the nearest lattice point problem. Combinatorica **6**(1), 1–13 (1986). https://doi.org/10.1007/BF02579403
5. 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 (1986). https://doi.org/10.1007/3-540-47721-7_24
6. Becker, H., Bermudo Mera, J.M., 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

7. Becker, H., Hwang, V., Kannwischer, M.J., Yang, B.Y., Yang, S.Y.: Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1. IACR Transactions on Cryptographic Hardware and Embedded Systems **2022**(1), 221–244 (2022). https://doi.org/10.46586/tches.v2022.i1.221-244
8. 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
9. De Almeida Braga, D., Fouque, P.A., Lachguel, B., Prest, T.: Toward a secure fixed-point implementation of the Falcon signature scheme. In: Advances in Cryptology – CRYPTO 2026, Part III. Lecture Notes in Computer Science, vol. 16802, pp. 166–197. Springer (2026). https://doi.org/10.1007/978-3-032-35377-1_6
10. Dekker, T.J.: A floating-point technique for extending the available precision. Numerische Mathematik **18**(3), 224–242 (1971). https://doi.org/10.1007/BF01397083
11. Ducas, L., Prest, T.: Fast Fourier orthogonalization. In: Proceedings of the ACM on International Symposium on Symbolic and Algebraic Computation, ISSAC 2016. pp. 191–198. ACM (2016). https://doi.org/10.1145/2930889.2930923
12. Fouque, P.A., Hoffstein, J., Kirchner, P., Lyubashevsky, V., Pornin, T., Prest, T., Ricosset, T., Seiler, G., Whyte, W., Zhang, Z.: Falcon: Fast-Fourier lattice-based compact signatures over NTRU. Specification v1.2. Submission to the NIST Post-Quantum Cryptography Standardization Process, Round 3 (2020), https://falcon-sign.info/
13. Guerreau, M., Martinelli, A., Ricosset, T., Rossi, M.: The hidden parallelepiped is back again: Power analysis attacks on Falcon. IACR Transactions on Cryptographic Hardware and Embedded Systems **2022**(3), 141–164 (2022). https://doi.org/10.46586/tches.v2022.i3.141-164
14. Halmans, S., van Vredendaal, C., Schneider, T., Custers, F., Güneysu, T.: TWFalcon: Triple-word arithmetic for Falcon: Giving Falcon the precision to fly securely. IACR Transactions on Cryptographic Hardware and Embedded Systems **2026**(2), 347–371 (2026). https://doi.org/10.46586/tches.v2026.i2.347-371
15. Harvey, D.: Faster arithmetic for number-theoretic transforms. Journal of Symbolic Computation **60**, 113–119 (2014). https://doi.org/10.1016/j.jsc.2013.09.002
16. Howe, J., Westerbaan, B.: Benchmarking and analysing the NIST PQC lattice-based signature schemes standards on the ARM Cortex M7. In: Progress in Cryptology – AFRICACRYPT 2023. Lecture Notes in Computer Science, vol. 14064. Springer (2023). https://doi.org/10.1007/978-3-031-37679-5_19
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)
19. Hwang, V.: Formal verification of emulated floating-point arithmetic in Falcon. In: Advances in Information and Computer Security – IWSEC 2024. Lecture Notes in Computer Science, vol. 14977, pp. 125–141. Springer (2024). https://doi.org/10.1007/978-981-97-7737-2_7
20. IEEE: IEEE standard for floating-point arithmetic. IEEE Std 754-2019 (2019). https://doi.org/10.1109/IEEESTD.2019.8766229

21. Karabulut, E., Aysu, A.: FALCON down: Breaking FALCON post-quantum signature scheme through side-channel attacks. In: 58th ACM/IEEE Design Automation Conference, DAC 2021. pp. 691–696. IEEE (2021). https://doi.org/10.1109/DAC18074.2021.9586131
22. Kim, Y., Song, J., Seo, S.C.: Accelerating Falcon on ARMv8. IEEE Access **10**, 44446–44460 (2022). https://doi.org/10.1109/ACCESS.2022.3169784
23. Lee, S.W., Sim, M.J., Seo, H.J.: High-performance implementations of HAETAE and SMAUG-T on Arm Cortex-M55 using MVE and Slothy. Cryptology ePrint Archive, Paper 2026/2168 (2026)
24. Lin, X., Tibouchi, M., Yu, Y., Zhang, S.: Do not disturb a sleeping Falcon: Floating-point error sensitivity of the Falcon sampler and its consequences. In: Advances in Cryptology – EUROCRYPT 2025, Part II. pp. 213–244. Lecture Notes in Computer Science, Springer (2025). https://doi.org/10.1007/978-3-031-91124-8_8
25. 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
26. Moody, D., Perlner, R., Regenscheid, A., Robinson, A., Cooper, D.: Transition to post-quantum cryptography standards. NIST Internal Report 8547 (Initial Public Draft), National Institute of Standards and Technology (Nov 2024). https://doi.org/10.6028/NIST.IR.8547.ipd
27. National Institute of Standards and Technology: SHA-3 standard: Permutation-based hash and extendable-output functions. Federal Information Processing Standards Publication 202, National Institute of Standards and Technology (2015). https://doi.org/10.6028/NIST.FIPS.202
28. National Institute of Standards and Technology: Module-lattice-based digital signature standard. Federal Information Processing Standards Publication FIPS 204, U.S. Department of Commerce (Aug 2024). https://doi.org/10.6028/NIST.FIPS.204
29. National Institute of Standards and Technology: Module-lattice-based key-encapsulation mechanism standard. Federal Information Processing Standards Publication FIPS 203, U.S. Department of Commerce (Aug 2024). https://doi.org/10.6028/NIST.FIPS.203
30. National Institute of Standards and Technology: Stateless hash-based digital signature standard. Federal Information Processing Standards Publication FIPS 205, U.S. Department of Commerce (Aug 2024). https://doi.org/10.6028/NIST.FIPS.205
31. National Security Agency: Announcing the commercial national security algorithm suite 2.0. Cybersecurity Advisory PP-22-1338, Ver. 1.0 (Sep 2022), https://media.defense.gov/2025/May/30/2003728741/-1/-1/0/CSA_CNSA_2.0_ALGORITHMS.PDF, accessed 2026-10-01
32. Nguyen, D.T., Gaj, K.: Fast Falcon signature generation and verification using ARMv8 NEON instructions. In: Progress in Cryptology – AFRICACRYPT 2023. Lecture Notes in Computer Science, vol. 14064, pp. 417–441. Springer (2023). https://doi.org/10.1007/978-3-031-37679-5_18
33. NIS Cooperation Group, EU PQC Workstream: A coordinated implementation roadmap for the transition to post-quantum cryptography, part 1, version 1.1 (Jun 2025), https://digital-strategy.ec.europa.eu/en/library/coordinated-implementation-roadmap-transition-post-quantum-cryptography, accessed 2026-10-01
34. Pornin, T.: New efficient, constant-time implementations of Falcon. Cryptology ePrint Archive, Paper 2019/893 (2019), https://eprint.iacr.org/2019/893
35. Pornin, T.: Improved key pair generation for Falcon, BAT and Hawk. Cryptology ePrint Archive, Paper 2023/290 (2023), https://eprint.iacr.org/2023/290

36. Pornin, T.: Falcon on ARM Cortex-M4: an update. Cryptology ePrint Archive, Paper 2025/123 (2025), https://eprint.iacr.org/2025/123
37. Pornin, T.: Improved (again) key pair generation for Falcon, BAT and Hawk. Cryptology ePrint Archive, Paper 2025/1239 (2025), https://eprint.iacr.org/2025/1239
38. Pornin, T.: c-fn-dsa: FN-DSA implementation in C. https://github.com/pornin/c-fn-dsa (2026), commit a5f15894bf1a68017074650d5298cecf9bb29a79, 2026-08-29
39. Pornin, T.: A RAM-efficient implementation of Falcon. Cryptology ePrint Archive, Paper 2026/1915 (2026), https://eprint.iacr.org/2026/1915
40. PQ Code Package: mlkem-native. https://github.com/pq-code-package/mlkem-native (2026), commit 637d076aa113d8faaec2277ed4a46b657acaf35f
41. Reparaz, O., Balasch, J., Verbauwhede, I.: Dude, is my code constant time? In: Design, Automation & Test in Europe Conference & Exhibition, DATE 2017. pp. 1697–1702. IEEE (2017). https://doi.org/10.23919/DATE.2017.7927267
42. SLOTHY developers: SLOTHY: assembly superoptimization via constraint solving. https://github.com/slothy-optimizer/slothy (2026), commit a5cea114f1d9cf221dc9295f244fbaaf3a9d0d9f
43. STMicroelectronics: STM32N6x5xx and STM32N6x7xx datasheet (Aug 2026), document DS14791, Rev. 10, https://www.st.com/resource/en/datasheet/stm32n645i0.pdf, accessed 2026-10-09
44. Zhang, J., Zhang, J.: Vectorized Falcon-Sign implementations using SSE2, AVX2, AVX-512F, NEON, and RVV. IACR Transactions on Cryptographic Hardware and Embedded Systems **2026**(1), 302–324 (2026). https://doi.org/10.46586/tches.v2026.i1.302-324
45. Zhang, S., Lin, X., Yu, Y., Wang, W.: Improved power analysis attacks on Falcon. In: Advances in Cryptology – EUROCRYPT 2023, Part IV. pp. 565–595. Lecture Notes in Computer Science, Springer (2023). https://doi.org/10.1007/978-3-031-30634-1_19