Fast Falcon Signature Generation and Verification Using ARMv8 NEON Instructions

Duc Tri Nguyen[0009−0001−3739−7991] ✉ and Kris Gaj[0000−0002−5050−8748]

George Mason University, Fairfax, VA, 22030, USA
{dnguye69,kgaj}@gmu.edu

Abstract. We present our speed records for Falcon signature generation and verification on ARMv8-A architecture. Our implementations are benchmarked on Apple M1 'Firestorm', Raspberry Pi 4 Cortex-A72, and Jetson AGX Xavier. Our optimized signature generation is 2× slower, but signature verification is 3–3.9× faster than the state-of-the-art CRYSTALS-Dilithium implementation on the same platforms. Faster signature verification may be particularly useful for the client side on constrained devices. Our Falcon implementation outperforms the previous work targeting Jetson AGX Xavier by the factors 1.48× for signing in `falcon512` and `falcon1024`, 1.52× for verifying in `falcon512`, and 1.70× for verifying in `falcon1024`. We achieve improvement in Falcon signature generation by supporting a larger subset of possible parameter values for FFT-related functions and applying our compressed twiddle-factor table to reduce memory usage. We also demonstrate that the recently proposed signature scheme Hawk, sharing optimized functionality with Falcon, has 3.3× faster signature generation and 1.6–1.9× slower signature verification when implemented on the same ARMv8 processors as Falcon.

Keywords: Number Theoretic Transform · Fast Fourier Transform · Post-Quantum Cryptography · Falcon · Hawk · ARMv8 · NEON

1 Introduction

When large quantum computers arrive, Shor's algorithm [39] will break almost all currently deployed public-key cryptography in polynomial time [16] due to its capability to obliterate two cryptographic bastions: the integer factorization and discrete logarithm problems. While there is no known quantum computer capable of running Shor's algorithm with parameters required to break current public-key standards, selecting, standardizing, and deploying their replacements have already started.

In 2016, NIST announced the Post-Quantum Cryptography (PQC) standardization process aimed at developing new public-key standards resistant to quantum computers. In July 2022, NIST announced the choice of three digital signature algorithms [2]: CRYSTALS-Dilithium [5], Falcon [23], and SPHINCS$^+$ [11] for a likely standardization within the next two years. Additionally, NIST has already standardized two stateful signature schemes, XMSS [26] and LMS [34].

Compared to Elliptic Curve Cryptography and RSA, PQC digital signatures have imposed additional implementation constraints, such as bigger key and signature sizes, higher memory usage, support for floating-point operations, etc. In many common applications, such as the distribution of software updates and the use of digital certificates, a signature is generated once by the server but verified over and over again by clients forming the network.

In this paper, we examine the PQC digital signatures' speed on ARMv8-A platforms. NEON is an alternative name for Advanced Single Instruction Multiple Data (ASIMD) extension, available since ARMv7. NEON includes additional instructions that can perform arithmetic operations in parallel on multiple data streams. It also provides a developer with 32 128-bit vector registers. Each register can store two 64-bit, four 32-bit, eight 16-bit, or sixteen 8-bit integer data elements. NEON instructions can perform the same arithmetic operation simultaneously on the corresponding elements of two 128-bit registers and store the results in the respective fields of a third register. Thus, an ideal speed-up vs. traditional single-instruction single-data (SISD) ARM instructions varies between 2 (for 64-bit operands) and 16 (for 8-bit operands).

In this work, we developed an optimized implementation of Falcon targeting ARMv8 cores. We then reused a significant portion of our Falcon code to implement Hawk – a new lattice-based signature scheme proposed in Sep. 2022 [21]. Although this scheme is not a candidate in the NIST PQC standardization process yet, it may be potentially still submitted for consideration in response to the new NIST call, with the deadline in June 2023.

We then benchmarked our implementation and existing implementations of Falcon, Hawk, CRYSTALS-Dilithium, SPHINCS+, and XMSS using the 'Firestorm' core of Apple M1 (being a part of MacBook Air) and the Cortex-A72 core (being a part of Raspberry Pi 4), as these platforms are widely available for benchmarking. However, we expect that similar rankings of candidates can be achieved using other ARMv8 cores (a.k.a. microarchitectures of ARMv8).

Contributions. In this paper, we overcome the high complexity of the Falcon implementation and present a speed record for its **S**ignature generation and **V**erification on two different ARMv8 processors.

In a signature generation, we constructed vectorized scalable FFT implementation that can be applied to any FFT level greater than five. We compressed the twiddle factor table in our FFT implementation using a method inspired by the complex *conjugate* root of FFT. In particular, we reduced the size of this table from *16 Kilobytes* in the reference implementation down to *4 Kilobytes* in our new `ref` and `neon` implementations. The modified FFT implementation with 4× smaller twiddle factor table is not specific to any processor. Thus, it can be used on any platform, including constrained devices with limited storage or memory.

In the **V**erify operation, we applied the best-known Number Theoretic Transform (NTT) implementation techniques to speed up its operation for Falcon-specific parameters. Additionally, we present the exhaustive search bound anal-

ysis applied to twiddle factors per NTT level aimed at minimizing the number of Barrett reductions in Forward and Inverse NTT.

We also optimized the performance of hash-based signatures, and comprehensively compare three stateless and one stateful digital signature schemes selected by NIST for standardization – Falcon, CRYSTALS-Dilithium, SPHINCS$^+$, and XMSS – and one recently-proposed lattice-based scheme Hawk. We rank them according to signature size, public-key size, and Sign and Verify operations' performance using the best implementations available to date.

Our code is publicly available at https://github.com/GMUCERG/Falcon_NEON

2 Previous Work

The paper by Streit et al. [40] was the first work about a NEON-based ARMv8 implementation of the lattice-based public-key encryption scheme New Hope Simple. Other works implement Kyber [41], SIKE [28], and FrodoKEM [32] on ARMv8. The most recent works on the lattice-based finalists NTRU, Saber, and CRYSTALS-Kyber are reported by Nguyen et al. [35, 36]. The paper improved polynomial multiplication and compared the performance of vectorized Toom-Cook and NTT implementations. Notably, the work by Becker et al. [6] showed a vectorized NEON NTT implementation superior to Toom-Cook, introduced fast Barrett multiplication, and special use of multiply-return-high-only `sq[r]dmulh` instructions. The SIMD implementation of Falcon was reported by Pornin [37] and Kim et al. [31]. On the application side, Falcon is the only viable option in hybrid, partial, and pure PQC V2V design described in the work of Bindel et al. [12]

In the area of low-power implementations, most previous works targeted the ARM Cortex-M4 [29]. In particular, Botros et al. [14], and Alkim et al. [3] developed Cortex-M4 implementations of Kyber. Karmakar et al. [30] reported results for Saber. Chung et al. [17] on Saber and NTRU, and later work by Becker et al. [8] improved Saber by a large margin on Cortex-M4/M55. The latest work by Abdulrahman et al. [1] improved Kyber and Dilithium performance on Cortex-M4.

The most comprehensive Fast Fourier Transform (FFT) work is by Becoulet et al. [9][1]. The publications by Frigo et al. [24] and Blake et al. [13] describe the SIMD FFT implementations.

3 Background

Table 1 summarizes values of parameters n and q for various signature schemes and NIST security levels. n is a parameter in the cyclotomic polynomial $\phi = (x^n + 1)$, and q is a prime defining a ring $\mathbb{Z}_q[x]/(\phi)$. The sizes of the public key and signature in bytes (**B**) are denoted with $|pk|$ and $|sig|$. The signature ratio

[1] https://github.com/diaxen/fft-garden

Table 1. Parameter sets, key sizes, and signature sizes for FALCON, HAWK, DILITHIUM, XMSS, and SPHINCS$^+$. The last column shows the signature size ratio in comparison to FALCON512, DILITHIUM3, or FALCON1024, depending on the security level.

	NIST level	n	q	$\|pk\|$	$\|sig\|$	$\|pk + sig\|$	$\|sig\|$ ratio
FALCON512	I	512	12,289	897	652	1,549	1.00
HAWK512			65,537	1,006	542	1,548	0.83
DILITHIUM2	II	256	8,380,417	1,312	2,420	3,732	3.71
XMSS^{16}-SHA256		-	-	64	2,692	2,756	4.12
SPHINCS$^+$128s	I	-	-	32	7,856	7,888	12.05
SPHINCS$^+$128f		-	-	32	17,088	17,120	26.21
DILITHIUM3	III	256	8,380,417	1,952	3,293	5,245	1.00
SPHINCS$^+$192s		-	-	48	16,224	16,272	4.93
SPHINCS$^+$192f		-	-	48	35,664	35,712	10.83
FALCON1024	V	1024	12,289	1,793	1,261	3,054	1.00
HAWK1024			65,537	2,329	1,195	3,524	0.95
DILITHIUM5		256	8,380,417	2,592	4,595	7,187	3.64
SPHINCS$^+$256s		-	-	64	29,792	29,856	23.62
SPHINCS$^+$256f		-	-	64	49,856	49,920	39.53

$|sig|$ ratio is the result of dividing the signature size of other schemes by the signature size of FALCON512, DILITHIUM3, and FALCON1024.

3.1 Falcon

Falcon is a lattice-based signature scheme utilizing the 'hash-and-sign' paradigm. The security of Falcon is based on the hardness of the Short Integer Solution problem over NTRU lattices, and the security proofs are given in the random oracle model with tight reduction. Falcon is difficult to implement, requiring tree data structures, extensive floating-point operations, and random sampling from several discrete Gaussian distributions. The upsides of Falcon are its small public keys and signatures as compared to Dilithium. As shown in Table 1, the signature size of Falcon at the highest NIST security level is still smaller than that of the lowest security level of Dilithium, XMSS, and SPHINCS$^+$. Key generation in Falcon is expensive. However, a key can be generated once and reused later.

The signature generation (Algorithm 1) of Falcon first computes hash value c from message m and salt r. Then, it uses (f, g, F, G) from the secret key components to compute two short values s_1, s_2 such that $s_1 + s_2 h = c \bmod (\phi, q)$. Falcon relies extensively on floating-point computations during signature generation, used in Fast Fourier Transform (FFT) over the ring $\mathbb{Q}[x]/(\phi)$, and Gaussian and Fast Fourier Sampling (`ffSampling`) for Falcon tree T.

Algorithm 1: Falcon Sign

Input: A message m, a secret key sk, a bound $[\beta^2]$
Output: $sig = (r, s)$

1 $r \leftarrow \{0,1\}^{320}$ uniformly $\quad c \leftarrow \texttt{HashToPoint}(r||m, q, n)$
2 $t \leftarrow \left(-\frac{1}{q}\,\mathrm{FFT}(c) \odot \mathrm{FFT}(F), \frac{1}{q}\,\mathrm{FFT}(c) \odot \mathrm{FFT}(f)\right)$ ▷ $t \in \mathbb{Q}[x]/(\phi)$
3 **do**
4 **do**
5 $z \leftarrow \texttt{ffSampling}_n(t, T)$
6 $s = (t - z)\hat{B}$ ▷ $s \in$ Gaussian distribution: $s \sim D_{(c,0)+\Lambda(B),\sigma,0}$
7 **while** $||s^2|| > \lfloor \beta^2 \rfloor$;
8 $(s_1, s_2) \leftarrow \mathrm{invFFT}(s)$ ▷ $s_1 + s_2 h = c \bmod (\phi, q)$
9 $s \leftarrow \texttt{Compress}(s_2, 8 \cdot sbytelen - 328)$
10 **while** $(s = \perp)$;
11 return $sig = (r, s)$

Algorithm 2: Falcon Verify

Input: A message $m, sig = (r, s)$, $pk = h \in \mathbb{Z}_q[x]/(\phi)$, a bound $\lfloor \beta^2 \rfloor$
Output: Accept or reject

1 $c \leftarrow \texttt{HashToPoint}(r \| \mathrm{m}, q, n) \quad s_2 \leftarrow \texttt{Decompress}(\mathrm{s}, 8 \cdot \mathrm{sbytelen} - 328)$
2 **if** $(s_2 = \perp)$ **then**
3 reject
4 $s_1 \leftarrow c - s_2 h \bmod (\phi, q)$ ▷ $|s_1| < \frac{q}{2}$ and $s_1 \in \mathbb{Z}_q[x]/(\phi)$
5 **if** $\|(s_1, s_2)\|^2 \leq \lfloor \beta^2 \rfloor$ **then**
6 accept ;
7 **else**
8 reject

The signature verification (Algorithm 2) checks if two short values (s_1, s_2) are in acceptance bound $\lfloor \beta^2 \rfloor$ using the knowledge from public key pk, and signature (r, s). If the condition at line 5 is satisfied, then the signature is valid; otherwise, it is rejected. As opposed to signature generation, Falcon Verify operates only over integers.

Falcon supports only NIST security levels 1 and 5. A more detailed description of the underlying operations can be found in the Falcon specification [23].

3.2 Dilithium

Dilithium is a member of the Cryptographic Suite for Algebraic Lattices (CRYSTALS) along with the key encapsulation mechanism (KEM) Kyber. The core operations of Dilithium are the arithmetic of polynomial matrices and vectors. Unlike 'hash-and-sign' used in Falcon, Dilithium applies the Fiat-Shamir with Aborts [20, 33] style signature scheme and bases its security upon the Mod-

ule Learning with Errors (M-LWE) and Module Short Integer Solution (M-SIS) problems.

Compared with Falcon, Dilithium only operates over the integer ring $\mathbb{Z}_q[x]/(\phi)$ with $\phi = (x^n + 1)$. Thus, it is easier to deploy in environments lacking floating-point units. Dilithium supports three NIST security levels: 2, 3, and 5, and its parameters are shown in Table 1. More details can be found in the Dilithium specification [5].

3.3 XMSS

XMSS [26] (eXtended Merkle Signature Scheme) is a stateful hash-based signature scheme based on Winternitz One-Time Signature Plus (WOTS+) [27]. XMSS requires state tracking because the private key is updated every time a signature is generated. Hence, the key management of XMSS is considered difficult. Consequently, XMSS should only be used in highly controlled environments [19]. The advantages of XMSS over SPHINCS$^+$ are smaller signature sizes and better performance. XMSS is a single-tree scheme, with a multi-tree variant XMSS^{MT} also included in the specification. The security of XMSS relies on the complexity of the collision search of an underlying hashing algorithm.

Single-tree XMSS has faster signature generation and verification than the multi-tree XMSS^{MT} and comes with three tree heights: $h = [10, 16, 20]$, which can produce up to 2^h signatures. We select a single-tree variant of XMSS with a reasonable number of signatures, 2^{16}, and choose optimized SHA256 as underlying hash functions. This variant is denoted as XMSS^{16}-SHA256 in Table 1.

3.4 SPHINCS$^+$

SPHINCS$^+$ [11] is a stateless hash-based signature scheme that avoids the complexities of state management associated with using stateful hash-based signatures. SPHINCS$^+$ security also relies on hash algorithms. The algorithm is considered a conservative choice, preventing any future attacks on lattice-based signatures. SPHINCS$^+$ provides *'simple'* and *'robust'* construction. The *'robust'* construction affects the security proof and runtime. In addition, small ('*s*') and fast parameters ('*f*') influence execution time. These parameter set variants are over 128, 192, and 256 quantum security bits.

Based on the performance provided in the specification of SPHINCS$^+$, we select the *'simple'* construction, and both '*s*' and '*f*' parameters for NIST security levels 1, 3, and 5, as shown in Table 1. Unlike in XMSS, we select optimized SHAKE as the underlying hash function.

3.5 Hawk

Hawk is a recent signature algorithm proposed by Ducas et al. [21] based on the Lattice Isomorphism Problem (LIP). Hawk avoids the complexities of the floating-point discrete Gaussian sampling, which is a bottleneck in our optimized

Falcon implementation. Hawk chooses to sample in a simple lattice $\mathbb{Z}^n$ [10, 22] with a hidden rotation.

An AVX2 implementation of HAWK1024 is faster than the equivalent implementation of FALCON1024 by 3.9× and 2.2×. With our optimized `neon` implementation of Falcon, we decided to port our optimized Falcon code to Hawk and investigate if such performance gaps between Hawk and Falcon still hold. In this work, we select HAWK512 and HAWK1024 at NIST security levels 1 and 5.

4 Number Theoretic Transform Implementation

The Number Theoretic Transform (NTT) is a transformation used as a basis for a polynomial multiplication algorithm with the time complexity of $O(n \log n)$ [18]. In Falcon, the NTT algorithm is used for polynomial multiplication over the ring $\mathbf{R}_q = \mathbb{Z}_q[x]/(x^n+1)$, where degree $n = [512, 1024]$ and $q = 12289 = 2^{13} + 2^{12} + 1$ with $q = 1 \bmod 2n$.

Complete NTT is similar to traditional FFT (Fast Fourier Transform) but uses the root of unity in the discrete field rather than in a set of real numbers. NTT and NTT^{-1} are forward and inverse operations, where $NTT^{-1}(NTT(f)) = f$ for all $f \in \mathbf{R}_q$.

$NTT(A) * NTT(B)$ denotes pointwise multiplication. Polynomial multiplication using NTT is shown in Equation 1.

$$C(x) = A(x) \times B(x) = NTT^{-1}(NTT(A) * NTT(B)) \tag{1}$$

4.1 Barrett multiplication

In our Falcon implementation, Barrett multiplication is used extensively when one factor is known [6]. As shown in Algorithm 3, b must be a known constant, and $b' = [(b \cdot R/q)/2]$, where $[\ ]$ represent truncation. In fact, b and b' in NTT are from the precomputed table ω_i and ω_i'.

Algorithm 3: Signed Barrett multiplication with a constant [6]

Input: Any $|a| < R = 2^w$, constant $|b| < q$ and $b' = [(b \cdot R/q)/2]$

Output: $c = \texttt{barrett\_mul}(a, b, b') = a \cdot b \bmod q$, and $|c| < \frac{3q}{2} < \frac{R}{2}$

1 $t \leftarrow \texttt{sqrdmulh}(a, b')$ ▷ $\texttt{hi}(\texttt{round}((2 \cdot a \cdot b'))$

2 $c \leftarrow \texttt{mul}(a, b)$ ▷ $\texttt{lo}(a \cdot b)$

3 $c \leftarrow \texttt{mls}(c, t, q)$ ▷ $\texttt{lo}(c - t \cdot q)$

4.2 Montgomery multiplication

First, Falcon Verify computes only one polynomial multiplication (as in line 4 of Algorithm 2). Two polynomials (s_2, h) are converted to NTT domain. Then, we perform pointwise multiplication between two polynomials in NTT domain. To efficiently compute modular reduction for two unknown factors during pointwise multiplication, the conventional way is to convert one polynomial to the Montgomery domain and perform Montgomery multiplication. Eventually, the multiplication with a constant factor n^{-1} is applied at the end of Inverse NTT. We apply a small tweak by embedding n^{-1} into Montgomery conversion during pointwise multiplication ($a_i b_i n^{-1}$ instead of $a_i b_i$ for $i \in [0, \ldots, n-1]$) to avoid multiplications with n^{-1} in Inverse NTT.

The Montgomery n^{-1} conversion uses Barrett multiplication with a known factor $a_{mont} = \texttt{barrett\_mul}(a, b, b')$, with $b = R \cdot n^{-1} \bmod q$. Furthermore, it is beneficial at the instruction level to embed n^{-1} when n is a power of 2 in Montgomery conversion. In particular, when $(R, n) = (2^{16}, 2^{10})$ then $b = R \cdot n^{-1} = 2^{16} \cdot 2^{-10} = 2^6 \bmod q$. Hence, multiply instruction at line 2 of Algorithm 3 can be replaced by a cheaper shift left (`shl`) instruction.

Secondly, we apply Montgomery multiplication *with rounding* for pointwise multiplication (Section 3 in Becker et al. [6]).

4.3 Minimizing the number of Barrett reductions

In Barrett multiplication (Algorithm 3), the theoretical bound of output c is in $-\frac{3q}{2} \leq c < \frac{3q}{2}$. Details of the proof can be found in Becker et al. [6]. Given that $q = 12289, R = 2^{16}$, the maximum bound of signed arithmetic centered around 0 is $2.6q \approx \frac{R}{2}$ instead of $5.3q \approx R$ in unsigned arithmetic.

During Forward and Inverse NTT, we carefully control the bound of each coefficient by applying our strict Barrett multiplication bound analysis. The naive $2.6q$ bound assumption will lead to performing Barrett reduction after every NTT level. To minimize the number of Barrett reductions, we validate the range of $c = \texttt{barrett\_mul}(b, \omega, \omega')$ for all unknown values of $|b| < \frac{R}{2}$ and $\omega \in \omega_i$ and $\omega' \in \omega'_i$ table according to each NTT level by exhaustive search (aka. brute-force all possible values in space $R = 2^{16}$). The bound output c of `barrett_mul` is increasing if $|b| < q$ and decreasing if $|b| \geq q$. For example, if $\frac{b}{q} \approx (0.5, 1.0, 2.0, 2.5)$, then after Barrett multiplication, the obtained bounds are $\frac{c}{q} \approx (0.69, 0.87, 1.25, 1.44)$.

As a result, we were able to minimize the number of reduction points in the Forward and Inverse NTT from after every one NTT level to every two NTT levels. In our case, an exhaustive search works in an acceptable time for the 16-bit space. A formal, strict bound analysis instead of an exhaustive search approach is considered as future work.

Algorithm 4: Signed Barrett reduction [6] for prime $q = 12289$

Input: Any $|a| < R = 2^w$, constants $(q, w, v, i) = (12289, 16, 5461, 11)$

Output: $c = \texttt{barrett\_mod}(a, q) \equiv a \bmod q$, and $-\frac{q}{2} \leq c < \frac{q}{2}$

1 $t \leftarrow \texttt{sqdmulh}(a, v)$ ▷ $\texttt{hi}(2 \cdot a \cdot v)$

2 $t \leftarrow \texttt{srshr}(t, i)$ ▷ $\texttt{round}(t \gg i)$

3 $c \leftarrow \texttt{mls}(a, t, q)$ ▷ $\texttt{lo}(a - t \cdot q)$

4.4 Forward and Inverse NTT Implementation

Falcon uses NTT to compute polynomial multiplication in the Verify operation. To avoid a bit-reversal overhead, Cooley-Tukey (CT) and Gentleman-Sande (GS) butterflies are used for Forward and Inverse NTT, respectively.

Instead of vectorizing the original reference Falcon NTT implementation, we rewrite the NTT implementation to exploit cache temporal and spatial locality. Our NTT implementation is centered around 0 to use signed arithmetic instructions instead of the unsigned arithmetic approach used by default in Falcon. This choice of implementation significantly improved our work compared to Kim et al. [31] due to special `sq[r]dmulh` instructions, which only work in signed arithmetic. We recommend utilizing multiply-return-high-only instruction for NTT implementation on any platform that supports it.

In Forward and Inverse NTT operations, `barrett_mul` is used extensively due to its compactness, thus yielding optimal performance, eliminating dependency chains by using only 3 instructions [6] rather than 9 instructions from Nguyen et al. [35, 36]. At the instruction level, based on the Becker et al. [6] micro-architecture pipeline trick, we gather the addition and subtraction from multiple butterflies in a group and arrange multiple `barret_mul` together. Note that this behavior also appeared in modern compiler optimization. Since our implementation uses intrinsic instructions instead of assembly instructions, we confirmed that the output assembly code showed similar order of instructions as in intrinsic implementation. On the low-end Cortex-A72 ARMv8 processor, we achieved 10% performance improvement by grouping instructions compared with ungrouping multiply instructions. However, this improvement is negligible in the high-end Apple M1 processor.

In terms of storage, the Barrett multiplication requires twiddle factor table ω and additional storage for the precomputed table ω': $\omega'_i = [(\omega_i \cdot R/q)/2]$, where $\omega_i = \omega^i \bmod q$. We prepared the twiddle factor tables ω_i and ω'_i so that every read from such tables is in the forward direction, and each entry is loaded only once during the entire operation.

Our Forward and Inverse NTT consist of two loops with the constant-stride (cache-friendly) load and store into memory and the permutation following steps in Nguyen et al. [35, 36]. Our Forward and Inverse NTT implementations are constructed by two loops, seven NTT levels are combined into the first loop, and the remaining two (resp. three) NTT levels are in the second loop for $n = 512$

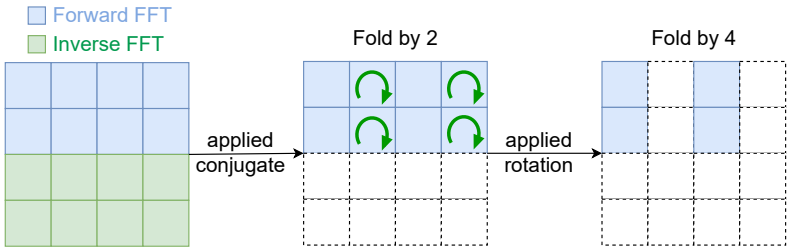

Fig. 1. Deriving full twiddle factor table by applying complex conjugate and rotation.

(resp. 1024). Each coefficient is loaded and stored once in each loop. With two loops, our implementation can reach up to $n = 2048, R = 2^{16}$ or $n = 1024, R = 2^{32}$ with a minimal number of load and store instructions.

5 Fast Fourier Transform Implementation

The Fast Fourier Transform (FFT) is a fast algorithm that computes a Discrete Fourier Transform from the time domain to the frequency domain and vice versa.

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi kn/N} \text{ with } k \in [0, N-1] \tag{2}$$

The discrete Fourier transform in Equation 2 has time complexity of $O(n^2)$. FFT improves the transformation with the time complexity of $O(n \log n)$ [18].

The advantage of FFT over NTT for polynomial multiplication is that the root of unity $e^{i2\pi/N}$ always exists for arbitrary N. Additionally, FFT suffers precision loss caused by rounding in the floating-point-number computations, while polynomial multiplication using NTT guarantees correctness due to NTT operating in the integer domain.

5.1 Compressed twiddle factor table

In Falcon, each complex point utilizes 128 bits of storage. Reducing the required storage amount improves cache locality and minimizes memory requirements. Both improvements are especially important in constrained devices. When analyzing the twiddle factor table, we realized that the real and imaginary parts of complex points are repeated multiple times because of the complex number negation, conjugation, and rotation. For example, complex roots of $x^8 + 1$ can be derived from a single complex root $a = (a_{re}, a_{im})$ to $[a, -a, -ia, ia]$ as the first half of the roots, and $[\hat{a}, -\hat{a}, -i\hat{a}, i\hat{a}]$ as the second half of the roots. It is notable that the second half is the *complex conjugate* of the first half, where $\hat{a} = conjugate(a) = (a_{re}, -a_{im})$. As a result, we only need to store $a = (a_{re}, a_{im})$

and use the add and subtract instructions to perform negation, conjugation, and rotation.

In summary, we fold the twiddle factor table by a factor of 2 by applying the *complex conjugate* to derive the second half from the first half. Furthermore, we use addition, subtraction, and rotation to derive the variants of complex roots within the first half, thus saving another factor of 2, as shown in Fig. 1. However, the FFT implementation in the reference implementation of Falcon no longer works with our new twiddle factor table (`tw`). As a result, we rewrite our FFT implementation in `C` to adopt our newly compressed twiddle factor table.

In general, we can compress complex roots from n down to $\frac{n}{4}$. In particular, when $n = 512, 1024$, the default twiddle factor table size is $16n$ bytes. With compressed twiddle factors, we only need to store $128, 256$ complex roots, and the table size becomes $4n$ bytes. A special case in $x^4 + 1$, when $a = (a_{re}, a_{im}) = (\sqrt{2}, \sqrt{2})$, thus we exploit the fact that $a_{re} = a_{im}$ to save multiply instructions by writing a separate loop at the beginning and end of Forward and Inverse FFT, respectively.

In Forward FFT, only the first half of the roots is used, while in Inverse FFT, only the second half is used.

Our Iterative SIMD FFT. Many FFT implementations prefer a recursive approach for high degree $N \geq 2^{13}$ [13, 24], as it is more memory cache-friendly than the iterative approach. First, we decided to avoid using a vendor-specific library to maintain high portability. Secondly, we gave preference to an iterative approach to avoid function call overheads (since Falcon's $N \leq 1024$) and scheduling overheads for irregular butterfly patterns. Thirdly, we must support our compressed twiddle factor since the cost of deriving complex roots is minimal. Lastly, we focused on simplicity, so our code could be deployed and implemented on constrained devices and used as a starting point for hardware accelerator development.

In our literature search, we could not find either an FFT implementation or detailed algorithms fitting our needs. Hence, we wrote our own iterative FFT in `C`, then we vectorized our `C` FFT implementation. We are not aware of any published FFT implementation similar to our work.

5.2 Improved Forward FFT implementation

Similar to NTT, we use Cooley-Tukey butterflies in Algorithms 5 and 6 to exploit the first half of the roots in Forward FFT. We rewrote the Forward FFT implementation, so each twiddle factor is always loaded once, and the program can take advantage of the cache spatial and temporal locality with constant access patterns when load and store instructions are executed. All the butterflies in Algorithm 9 are computed in place. Note that the two for loops from line 10 to 17 can be executed in parallel, which may be of interest when developing a hardware accelerator.

In a signature generation, at line 5 in Algorithm 1, Fast Fourier sampling (`ffSampling`) builds an FFT tree by traveling from the top level, $l = \log_2(N)$,

Algorithm 5: `CT_BF`

Input: a, b, ω
Output: $(a, b) = (a+\omega b, a-\omega b)$

1 $t \leftarrow b * \omega$
2 $b \leftarrow a - t$
3 $a \leftarrow a + t$

Algorithm 6: `CT_BF_90`

Input: a, b, ω
Output: $(a, b) = (a+i\omega b, a-i\omega b)$

1 $t \leftarrow b * (i\omega)$
2 $b \leftarrow a - t$
3 $a \leftarrow a + t$

Algorithm 7: `GS_BF`

Input: a, b, ω
Output: $(a, b) = (a+b, (a-b)\hat{\omega})$

1 $t \leftarrow a - b$
2 $a \leftarrow a + b$
3 $b \leftarrow t * conjugate(\omega)$

Algorithm 8: `GS_BF_270`

Input: a, b, ω
Output: $(a, b) = (a+b, (a-b)i\hat{\omega})$

1 $t \leftarrow b - a$ ▷ avoid negation
2 $a \leftarrow a + b$
3 $b \leftarrow t * conjugate(-i\omega)$

to the lower level $l-1, l-2, \ldots 1$, where N is a total number of real and imaginary points. Hence, FFT implementation must support all FFT levels from $\log_2(N)$ to 1.

Our `C` FFT implementation supports all levels of Forward and Inverse FFT trivially. However, our vectorized FFT must be tailored to support all FFT levels. Instead of vectorizing $l-1$ FFT implementations, first, we determined the maximum number of coefficients that can be computed using 32 vector registers. Then, we select $l = 5$ as the baseline to compute FFT that only uses one load and store per coefficient. To scale up to $l - 1$ FFT levels, we apply `#FFT levels` $= 5 + 2 \cdot x + 1 \cdot y$ where $2 \cdot x$ supports multiple of 2 FFT levels with x load and store per coefficient, and $1 \cdot y$ supports a single FFT level, we aim to minimize y to minimize load and store, thus $y \in \{0, 1\}$. In case $l < 5$, we unroll the FFT loop completely, and save instructions overhead. When $l \geq 5$, we use the base loop with $l = 5$ with 32 64-bit coefficients and implement two additional FFT loops. The second loop computes two FFT levels per iteration to save load and store instructions. The third loop is an unrolled version of a single FFT level per iteration. For example, when $N = 2^l = 2^{10}$, applied the formula above, $l = 5 \cdot \underline{1} + 2 \cdot \underline{2} + 1 \cdot \underline{1}$, in total, each coefficient is loaded and stored $4 = \underline{1} + \underline{2} + \underline{1}$ times.

In short, using three FFT loops, we can construct arbitrary FFT level $l \geq 5$ by using the base loop with 5 levels, then a multiple of two FFT levels by the second loop. Finally, the remaining FFT levels are handled by the third loop.

5.3 Improved Inverse FFT implementation

The butterflies in Inverse FFT in Algorithm 7 and Algorithm 8 exploit complex *conjugate* by using add and subtract instructions. A tweak at no cost in line 1 of Algorithm 8 to avoid floating-point negation instruction `fneg` in line 3. Similar to Forward FFT, two loops from line 6 to 13 in Algorithm 10 can be

Algorithm 9: In-place cache-friendly Forward FFT (`split` storage)

Input: Polynomial $f \in \mathbb{Q}[x]/(x^{N/2}+1)$, twiddle factor table `tw`
Output: $f = \texttt{FFT}(f)$

$\omega \leftarrow \texttt{tw}[0][0]$
for $j = 0$ **to** $N/4 - 1$ **do**
  `CT_BF`$(f[j], f[j + N/4], \omega)$ ▷ exploit $\omega_{re} = \omega_{im}$
  $j \leftarrow j + 1$
$level \leftarrow 1$
for $len = N/8$ **to** 1 **do**
  $k \leftarrow 0$ ▷ reset k at new $level$
  for $s = 0$ **to** $N/2 - 1$ **do**
    $\omega \leftarrow \texttt{tw}[level][k]$ ▷ ω is shared between two loops
    for $j = s$ **to** $s + len - 1$ **do**
      `CT_BF`$(f[j], f[j + len], \omega)$
      $j \leftarrow j + 1$
    $s \leftarrow s + (len \ll 1)$ **for** $j = s$ **to** $s + len - 1$ **do**
      `CT_BF_90`$(f[j], f[j + len], \omega)$
      $j \leftarrow j + 1$
    $s \leftarrow s + (len \ll 1)$
    $k \leftarrow k + 1$ ▷ increase by *one* point
  $level \leftarrow level + 1$ ▷ increase $level$
  $len \leftarrow len \gg 1$ ▷ half distance

Algorithm 10: In-place cache-friendly Inverse FFT (`split` storage)

Input: Polynomial $f \in \mathbb{Q}[x]/(x^{N/2}+1)$, twiddle factor table `tw`
Output: $f = \texttt{invFFT}(f)$

$level \leftarrow \log_2(N) - 2$ ▷ `tw` index starts at 0, and N/2 *re, im* points
for $len = 1$ **to** $N/8$ **do**
  $k \leftarrow 0$ ▷ reset k at new $level$
  for $s = 0$ **to** $N/2 - 1$ **do**
    $\omega \leftarrow \texttt{tw}[level][k]$ ▷ ω is shared between two loops
    for $j = s$ **to** $s + len - 1$ **do**
      `GS_BF`$(f[j], f[j + len], \omega)$
      $j \leftarrow j + 1$
    $s \leftarrow s + (len \ll 1)$
    for $j = s$ **to** $s + len - 1$ **do**
      `GS_BF_270`$(f[j], f[j + len], \omega)$
      $j \leftarrow j + 1$
    $s \leftarrow s + (len \ll 1)$
    $k \leftarrow k + 1$ ▷ increase by *one* point
  $level \leftarrow level - 1$ ▷ decrease $level$
  $len \leftarrow len \ll 1$ ▷ double distance
$\omega \leftarrow \texttt{tw}[0][0] \cdot \frac{2}{N}$
for $j = 0$ **to** $N/4 - 1$ **do**
  `GS_BF`$(f[j], f[j + N/4], \omega)$ ▷ exploit $\omega_{re} = \omega_{im}$
  $f[j] \leftarrow f[j] \cdot \frac{2}{N}$
  $j \leftarrow j + 1$

executed in parallel and share the same twiddle factor ω. The last loop from line 17 to 21 multiplies $\frac{2}{N}$ by all coefficients of FFT, and exploits the special case of twiddle factor $\omega_{re} = \omega_{im}$. We also employ three FFT loop settings as described in Forward FFT to implement vectorized multi-level l of Inverse FFT to maximize vector registers usage, hence improving execution time. Note that butterflies in Algorithm 10 are computed in place.

5.4 Floating-point complex instructions and data storage

ARMv8.3 supports two floating-point complex instructions: `fcadd` and `fcmla`. The floating-point complex `fcadd` instruction offers addition and counterclockwise rotation by 90 and 270 degrees: $(a+ib)$ and $(a-ib)$. The combination of `fmul` and `fcmla` instructions can perform complex point multiplication $(a * b), (a * \hat{b})$, as shown in lines 1 and 3 of Algorithms 5 and 7 by applying counterclockwise rotation by 90 and 270 degrees, respectively. Note that `fcmla` has the same cycle count as `fmla, fmls`.

The floating-point complex instructions offer a convenient way to compute *single pair* complex multiplications. Conversely, the `fmla, fmls` instructions require at least *two pairs* for complex multiplication. The only difference between floating-point complex instructions and traditional multiplication instructions is the data storage of real and imaginary values during the multiplication.

Our first approach is to use floating-point complex `fmul, fcmla, fcadd` instructions, where real and imaginary values are stored adjacent in memory (`adj` storage). This data storage setting is also seen in other FFT libraries, such as FFTW [24], and FFTS [13]. Complex multiplications using such instructions are demonstrated in Fig. 2. The second approach uses default data storage in Falcon: real and imaginary values are split into two locations (`split` storage). The complex multiplication using `fmul`, `fmla`, and `fmls` instructions is shown at Fig. 3. in Appendix A.

To find the highest performance gain, we implemented vectorized versions of the first and second approaches mentioned above. The former approach offers better cache locality for small $l \leq 4$. However, it introduces a vector permutation overhead to compute complex multiplication in lines 1 and 3 of Algorithms 6 and 8 using aforementioned floating-point complex instructions. Another disadvantage of the first approach is preventing the deployment of Falcon to devices that do not support ARMv8.3, such as Cortex-A53/A72 on Raspberry Pi 3/4, respectively. The latter approach can run on any ARMv8 platform. However, the second approach computes at least two complex point multiplications instead of one, and the pure C implementation is slightly faster compared to the original reference FFT implementation of Falcon. On the other hand, the pure C implementation of the first approach is slightly better than `split` storage in our experiment. We recommend using the first approach (`adj` storage) for non-vectorized implementation.

Eventually, we chose the second approach as our best-vectorized implementation and kept the reference implementation of the first approach in our code base for community interest. For $l \leq 2$, we unroll the loop, so the multiplication

is done by scalar multiplication. By rearranging vector registers appropriately using `LD2, LD4, ST2, ST4` instructions, when $l \geq 3$, the latter approach is slightly faster than the first approach when benchmarked on the high-end Apple M1 CPU.

5.5 Floating-point to integer conversion

Notably, both GCC and Clang can generate native floating-point to integer conversion instructions during compilation. These instructions include `fpr_floor, fpr_trunc` using rounding toward zero instruction `fcvtzs` except for `fpr_rint` function. As described in the 64-bit floating-point to 64-bit signed integer `fpr_rint` implementation (a constant time conversion written in `C`), the obtained assembly language code generated by Clang and GCC, respectively, is not consistent and does not address constant-time concerns described in Howe et al. [25]. In our implementation on an `aarch64` ARMv8, we use the rounding to the nearest with ties to even instruction `fcvtns` to convert a 64-bit floating-point to a 64-bit signed integer[2]. This single instruction, used to replace the whole `fpr_rint` implementation, costs 3 cycles on Cortex-A72/A78.

5.6 Rounding concern in Floating-point Fused Multiply-Add

Another concern while implementing vectorized code is floating-point rounding [4]. In `ref` implementation, when using the independent multiply (`fmul`) and add (`fadd, fsub`) instructions, the floating-point rounding occurs after multiplication and after addition. In the `neon` implementation, when we use Fused Multiply-Add instruction `(fmla, fmls)`, the rounding is applied only after addition.

When we repeat our experiment with `fpr_expm_p63` function used in Gaussian sampling to observe the rounding of `fmla` and `(fmul, fadd)`, the differences between them grow. In our experiment, we sample random values as input to `fpr_expm_p63` function on both Apple M1 and Cortex-A72, the differences are consistent in both CPUs[3], about 7,000 out of 100,000 output values of `fpr_expm_63` function with `fmla` are different from `(fmul, fadd)`.

We have carefully tested our implementation according to the test vectors and KATs (Known-Answer-Tests) provided by Falcon submitters. Although all tests passed, the security of the floating-point rounding differences in ARMv8 is unknown. Therefore, by default, our code uses the independent multiply (`fmul`) and add (`fadd, fsub`) instructions. We chose to optionally enable `fmla` instructions for benchmarking purposes only and observed negligible differences $3 \rightarrow 4\%$ between the two approaches in terms of the total execution time for the Sign operation, as shown in Table 2.

[2] https://godbolt.org/z/esP78P33b
[3] https://godbolt.org/z/613vvzh3Y

Table 2. Performance of Signature generation with Fused Multiply-Add instructions *enabled* (`fmla`), and *disabled* (`fmul, fadd`). Results are in *kc - kilocycles.*

CPU	`neon` Sign	`(fmul, fadd)`	`fmla`	`fmla/(fmul, fadd)`
Cortex-A72	`falcon512`	1,038.14	1,000.31	0.964
	`falcon1024`	2,132.08	2,046.53	0.960
`Apple M1`	`falcon512`	459.19	445.91	0.971
	`falcon1024`	914.91	885.63	0.968

6 Results

ARMv8 Intrinsics are used for ease of implementation and to take advantage of the compiler optimizers. The optimizers know how intrinsics behave and tune performance toward the processor features such as aligning buffers, scheduling pipeline operations, and instruction ordering[4]. In our implementation, we always keep vector register usage under 32 and examine assembly language code obtained during our development process. We acknowledge that the compiler occasionally spills data from registers to memory and hides load/store latency through the instructions supporting pipelining.

Falcon, Hawk, and Dilithium. The reference implementation of Hawk[5] uses a fixed-point data type by default. Since our choice of processors supports floating-point arithmetic, we convert all fixed-point arithmetic to floating-point arithmetic and achieve a significant performance boost in reference implementation as compared to the default setting. Notably, this choice disables NTT implementation in Hawk Sign and Verify, while Falcon Verify explicitly uses integer arithmetic for NTT implementation by default. Although it is possible to vectorize NTT implementation in Hawk Verify, we consider this as future work. In both implementations, we measure the Sign operations in the dynamic signing - the secret key is expanded before signing, and we do not use floating-point emulation options. For Dilithium, we select the state-of-the-art ARMv8 implementation from Becker et al. [6].

XMSS, SPHINCS$^+$. To construct a comprehensive digital signature comparison, we select the XMSS implementation[6] with the forward security by Buchmann et al. [15], which limited the way one-time signature keys are computed to enhance security. We accelerate SHA-256 by applying the OpenSSL SHA2-NI instruction set extensions in our `neon` implementation. For SPHINCS$^+$, we select *s, f*-variant and 'simple' instantiation to compare with lattice-based signatures. Recent work by Becker et al. [7] proposed multiple settings to accelerate Keccak-`f1600`, combine with scalar, `neon` and SHA3 instructions. To make sure

[4] https://godbolt.org/z/zPr94YjYr
[5] https://github.com/ludopulles/hawk-sign/
[6] https://github.com/GMUCERG/xmssfs

our hash-based signature is up-to-date, we measure and apply the best settings on Apple M1 to yield our best result for a hash-based signature.

The optimized implementation of SPHINCS$^+$[7] already included high-speed Keccak-F1600×2 `neon` implementation. For both, the `ref` implementation is a pure `C` hash implementation.

Constant-time treatment. For operations that use floating-point arithmetic extensively, we use vectorized division instruction as in the reference implementation [37]. In Falcon **V**erify, there is only integer arithmetic. Thus, the division is replaced by modular multiplication. In both operations, secret data is not used in branch conditions or memory access patterns.

Benchmarking setup Our benchmarking setup for ARMv8 implementations included MacBook Air with Apple M1 @ 3.2GHz, Jetson AGX Xavier @ 1.9 Ghz, and Raspberry Pi 4 with Cortex-A72 @ 1.8GHz. For AVX2 implementations, we used a PC based on Intel 11th gen i7-1165G7 @ 2.8GHz with Spectre and Meltdown mitigations disabled via a kernel parameter[8].

For cycle count on Cortex-A72, we used the `pqax`[9] framework . In Apple M1, we rewrote the work from Dougall Johnson[10] to perform cycle count[11]. On both platforms, we use Clang 13 with `-O3`, we let the compiler to do its best to vectorize pure C implementations, denoted as `ref` to fairly compare them with our `neon` implementations. Thus, we did not employ `-fno-tree-vectorize` option. We acknowledge that compiler automatically enables Fuse Multiply-Add to improve performance. Explicitly on Jetson AGX Xavier, we use Clang 6.0 and count cycle using similar method in the work of Seo et al. [38].

We report the average cycle count of 1,000 and 10,000 executions for hash-based and lattice-based signatures, respectively. Benchmarking is conducted using a single core and a single thread to fairly compare results with those obtained for lattice-based signatures, even though hash-based signatures can execute multiple hash operations in parallel.

Results for FFT and NTT. are summarized in Table 3 with FMA *enabled*. When $N \leq 4$, we realized that our vectorized code runs slower than the C implementation due to the FFT vectorized code being too short. Therefore, we use unroll `ref` implementation when $N \leq 4$, and `neon` for $N \geq 8$. In Table 3, when $N \geq 128$, our vectorized FFT achieved speed-up $1.9\times \rightarrow 2.3\times$ and $2.1\times \rightarrow 2.4\times$ compared to the `ref` implementation of the Forward and Inverse FFT on Apple M1, and $2.2\times \rightarrow 1.9\times$ in Forward and $2.3\times \rightarrow 1.9\times$ in Inverse FFT on

[7] https://github.com/sphincs/sphincsplus
[8] `mitigations=off` https://make-linux-fast-again.com/
[9] https://github.com/mupq/pqax/tree/main/enable_ccr
[10] https://github.com/dougallj
[11] https://github.com/GMUCERG/PQC_NEON/blob/main/neon/kyber/m1cycles.c

Table 3. Cycle counts for the implementation of FFT (with FMA *enabled*) and NTT with the size of N coefficients on Apple M1 and Cortex-A72 - `neon` vs. `ref`.

Apple M1	Forward FFT(*cycles*)			Inverse FFT(*cycles*)		
N	ref	neon	ref/neon	ref	neon	ref/neon
128	759	404	1.88	847	401	2.11
256	1,633	789	2.07	1,810	794	2.28
512	3,640	1,577	2.31	3,930	1,609	2.44
1024	7,998	3,489	2.29	8,541	3,547	2.41
	Forward NTT(*cycles*)			Inverse NTT(*cycles*)		
512	6,607	840	7.87	6,449	811^n	7.95
1024	13,783	1,693	8.14	13,335	$1,702^n$	7.83

Cortex-A72	Forward FFT(*cycles*)			Inverse FFT(*cycles*)		
N	ref	neon	ref/neon	ref	neon	ref/neon
128	2,529	1,155	2.19	2,799	1,216	2.30
256	5,474	2,770	1.98	6,037	2,913	2.07
512	11,807	5,951	1.98	13,136	6,135	2.14
1024	27,366	14,060	1.95	28,151	14,705	1.91
	Forward NTT(*cycles*)			Inverse NTT(*cycles*)		
512	22,582	3,561	6.34	22,251	$3,563^n$	6.25
1024	48,097	7,688	6.26	47,196	$7,872^n$	6.00

Intel i7-1165G7	Forward FFT(*cycles*)			Inverse FFT(*cycles*)		
N	REF	AVX2	REF/AVX2	REF	AVX2	REF/AVX2
128	787	481	1.64	873	499	1.75
256	1,640	966	1.70	1,798	1,024	1.76
512	3,486	2,040	1.71	3,790	2,138	1.77
1024	7,341	4,370	1.68	7,961	4,572	1.74

^{n} no multiplication with n^{-1} at the end of `Inverse NTT`

Cortex-A72. These speed-ups are due to our unrolled vectorized implementation of FFT, which supports multiple FFT levels and twiddle-factor sharing.

In Falcon Verify, our benchmark of Inverse NTT is without multiplication by n^{-1} because we already embed this multiplication during Montgomery conversion. We achieve speed-up by 6.0× and 7.8× for NTT operations on Cortex-A72 and Apple M1, respectively. For both FFT and NTT, our `neon` implementation in both Cortex-A72 and Apple M1 achieve better speed-up ratio than the AVX2 implementation, as shown in Table 3. There is no AVX2 optimized implementation of Falcon Verify, it is the same as pure REF implementation. Overall, our

Table 4. Comparison with previous work by Kim et al. [31] on Jetson AGX Xavier. Additional results for Apple M1 and Raspberry Pi 4. Notation: *kc*-kilocycles.

	Jetson AGX Xavier					
	`ref`(kc)		`neon`(kc)		`ref/neon`	
	S	**V**	**S**	**V**	**S**	**V**
`falcon512` [31]	580.7	48.0	498.6	29.0	1.16	1.65
`falcon512` (Ours)	582.3	44.1	336.6	19.1	1.73	2.31
`falcon1024` [31]	1,159.6	106.0	990.5	62.5	1.17	1.69
`falcon1024` (Ours)	1,151.2	93.2	671.2	36.7	1.72	2.54
	Apple M1					
`falcon512` (Ours)	654.0	43.5	442.0	22.7	1.48	1.92
`falcon1024` (Ours)	1,310.8	89.3	882.1	42.9	1.49	2.08
	Raspberry Pi 4					
`falcon512` (Ours)	1,490.7	126.3	1,001.9	58.8	1.49	2.15
`falcon1024` (Ours)	3,084.8	274.3	2,048.9	130.4	1.51	2.10

`neon` NTT implementation is greatly improved compared to `ref` implementation, which determines the overall speed-up in Falcon Verify.

Comparison with previous work. As shown in Table 4, on the same platform, Jetson AGX Xavier, our NEON-based implementation of Falcon is consistently faster than the the implementation by Kim et al. [31]. The achieved speed-up vs. [31] is 1.48× for signing in `falcon512` and `falcon1024`, 1.52× for verifying in `falcon512`, and 1.70× for verifying in `falcon1024`.

In a signature generation, as compared to the Kim et al. [31] approach, we decided against replacing macro functions `FPC_MUL, FPC_DIV`, etc. Our manual work of unrolled versions of the Forward and Inverse FFT, as well as `splitfft, mergefft, mulfft`, etc. contribute to greatly improving the performance of Falcon. We also modify the code logic to reduce calling `memcpy` functions during operation and sharing twiddle factors, which greatly reduces memory load and store overhead. In signature verification, our NTT speed-up is 6.2× and 6.0× with respect to `ref` implementation for Forward and Inverse NTT, as shown in Table 3, while Kim et al. [31] only achieve less than 3× speed-up. The significant speed-up is accomplished due to our signed-integer implementation of NTT, with values centered around 0, while previous work used unsigned integer NTT.

Falcon and Dilithium. In Table 5, we rank our implementations with respect to the state-of-the-art CRYSTALS-Dilithium implementation from Becker et al. [6] across all security levels. Please note that in the top rows, only DILITHIUM2, XMSS^{16}-SHA256 have security level 2, while the rest algorithms have security level 1. For all security levels of Falcon and Dilithium, when executed over messages

Table 5. **S**ignature generation and **V**erification speed comparison (with FMA *disabled*) over three security levels, for a 59-byte message. `ref` and `neon` results for Apple M1. *kc*-kilocycles.

Apple-M1 3.2 GHz	NIST level	ref(kc) S	ref(kc) V	neon(kc) S	neon(kc) V	ref/neon S	ref/neon V
FALCON512	I, II	654.0	43.5	459.2	22.7	1.42	1.92
HAWK512		138.6	34.6	117.7	27.1	1.18	1.27
DILITHIUM2[b]		741.1	199.6	224.1	69.8	3.31	2.86
XMSS^{16}-SHA256[x]		26,044.3	2,879.4	4,804.2	531.0	5.42	5.42
SPHINCS$^+$128s[s]		1,950,265.0	1,982.4	549,130.7	658.6	3.55	3.01
SPHINCS$^+$128f[s]		93,853.9	5,483.8	26,505.3	1,731.2	3.54	3.16
DILITHIUM3[b]	III	1,218.0	329.2	365.2	104.8	3.33	3.14
SPHINCS$^+$192s[s]		3,367,415.5	2,753.1	950,869.9	893.2	3.54	3.08
SPHINCS$^+$192f[s]		151,245.2	8,191.5	42,815.1	2,515.8	3.53	3.25
FALCON1024	V	1,310.8	89.3	915.0	42.9	1.43	2.08
HAWK1024		279.7	73.7	236.9	58.5	1.18	1.26
DILITHIUM5[b]		1,531.1	557.7	426.6	167.5	3.59	3.33
SPHINCS$^+$256s[s]		2,938,702.4	3,929.3	840,259.4	1,317.5	3.50	2.98
SPHINCS$^+$256f[s]		311,034.3	8,242.5	88,498.9	2,593.8	3.51	3.17

[b] the work from Becker et al. [6]
[s] our benchmark SPHINCS$^+$ *'simple'* variants using Keccak-`f1600`
[x] our benchmark XMSS^{16}-SHA256 using SHA2 Crypto instruction

of the size of 59 bytes, for **S**ignature generation, Falcon is comparable with Dilithium in `ref` implementations. However, the landscape drastically changes in the optimized `neon` implementations. Dilithium has an execution time 2× smaller than Falcon at the lowest and highest security levels.

The speed-up ratio of Dilithium in ARMv8 is 3.3× as compared to `ref` implementation. This result is due to the size of operands in the vectorized implementation. Dilithium uses 32-bit integers and parallel hashing SHAKE128/256. This leads to a higher speed-up ratio as compared to 64-bit floating-point operations, serial hashing using SHAKE256, and serial sampling in Falcon. Additionally, the computation cost of floating-point operations is higher than for integers. Hence, we only achieve 1.42× speed-up (with FMA *disable*) compared to the `ref` implementation for the signature generation operation in Falcon. Although there are no floating-point computations in Falcon **V**erify, our speed-up is smaller than for Dilithium due to the serial hashing using SHAKE256. We believe that if Falcon

adopts parallel hashing and parallel sampling algorithms, its performance could be further improved.

In **V**erification across all security levels, Falcon is consistently faster than Dilithium by 3.0× to 3.9× in both `ref` and `neon` implementations.

Hawk vs. Falcon and Dilithium. At security levels 1 and 5, Hawk outperforms both Falcon and Dilithium in `ref` and `neon` implementations. The exception is the `neon` implementation of Falcon Verify due to two polynomial multiplications in Hawk instead of one in Falcon. An optimized `neon` implementation of Hawk Verify is unlikely to be faster than Falcon Verify, even if Hawk has optimized `neon` NTT implementation in its signature verification. The performance of Hawk versus Falcon and Dilithium in Sign are 3.9× and 1.9× faster, and in Verify are 0.8× and 2.5× faster for the `neon` implementation at security level 1. Our `neon` implementation of Hawk achieves a similar speed-up ratio as in the case of AVX2 implementations reported in Ducas et al. [21] (Table 1).

Lattice-based signatures vs. Hash-based signatures. Notably, in Table 5, the execution times of Falcon, Dilithium, and Hawk lattice-based signatures are shorter than for XMSS and SPHINCS$^+$ hash-based signatures by orders of magnitude in both `ref` and `neon` implementations. In **V**erification alone, Falcon is faster than XMSS and SPHINCS$^+$ by 23.4 to 28.8×. Similarly, the Dilithium verification is faster than for XMSS and SPHINCS$^+$ by 7.6 to 9.4×. The speed-up of hash-based signatures is higher than for lattice-based signatures, and it can be even greater if parallelism is fully exploited, e.g., through multithreading in CPUs or GPUs. These improvements are left for future work.

Table 6. Signature generation and **V**erification speed comparison (with FMA *disabled*) over three security levels, signing a 59-byte message. Ranking over the **V**erification speed ratio. `ref` and `neon` results for Cortex-A72. *kc*-kilocycles.

Cortex-A72	NIST	`ref`(kc)		`neon`(kc)		`ref/neon`		
1.8 GHz	Level	**S**	**V**	**S**	**V**	**S**	**V**	**V** ratio
FALCON512		1,553.4	127.8	1,044.6	59.9	1.49	2.09	1.0
HAWK512	I, II	400.3	127.1	315.9	94.8	1.26	1.34	1.6
DILITHIUM2[b]		1,353.8	449.6	649.2	272.8	2.09	1.65	4.5
DILITHIUM3[b]	III	2,308.6	728.9	1,089.4	447.5	2.12	1.63	-
FALCON1024		3,193.0	272.1	2,137.0	125.2	1.49	2.17	1.0
HAWK1024	V	822.1	300.0	655.2	236.9	1.25	1.27	1.9
DILITHIUM5[b]		2,903.6	1,198.7	1,437.0	764.9	2.02	1.57	6.1

[b] the work from Becker et al. [6]

Lattice-based signature in constrained devices. In Table 6, we rank the **V**erfication performance of Falcon, Dilithium, and Hawk. Notably, our Hawk Verify uses floating-point arithmetic, while the signature **V**erifications of Falcon and Dilithium only require integer operations. We exclude hash-based signatures from this comparison due to their low performance already shown for high-speed processors in Table 5. Falcon Verify at security level 5 is faster than Dilithium at security levels 1 and 5 by 2.2× and 6.1×. Hawk outperforms Dilithium and is only slower than Falcon in Verify operation by 1.6× and 1.9× at the same security level. In combination with Table 1, it is obvious that Falcon is more efficient than Dilithium in terms of both bandwidth and workload.

7 Conclusions

Falcon is the only PQC digital signature scheme selected by NIST for standardization using floating-point operations. Unless significant changes are introduced to Falcon, floating-point instructions required in Key generation and Sign operations will continue to be a key limitation of Falcon deployment. Additionally, the complexity of serial sampling and serial hashing significantly reduces the performance of Falcon Key and Signature generation. We demonstrate that Hawk outperforms Dilithium and has a faster Signature generation than Falcon. Its performance and bandwidth may be interesting to the community.

In summary, we report the new speed record for Falcon Sign and Verify operations using NEON-based instruction on Cortex-A72 and Apple M1 ARMv8 devices. We present a comprehensive comparison in terms of performance and bandwidth for Falcon, CRYSTALS-Dilithium, XMSS, and SPHINCS$^+$ on both aforementioned devices. We believe that in some constrained protocol scenarios, where bandwidth and verification performance matter, Falcon is the better option than Dilithium, and lattice-based signatures are a far better choice than hash-based signatures in terms of key size and efficiency.

Lastly, we present a 7% mismatch between the Fuse Multiply-Add instructions on ARMv8 platforms. We recommend disabling the Fuse Multiply-Add instruction to guarantee implementation correctness. Further security analysis of this behavior is needed.

Acknowledgments. This work has been partially supported by the National Science Foundation under Grant No.: CNS-1801512 and by the US Department of Commerce (NIST) under Grant No.: 70NANB18H218.

References

1. Abdulrahman, A., Hwang, V., Kannwischer, M.J., Sprenkels, D.: Faster Kyber and Dilithium on the Cortex-M4. In: ACNS 22. vol. 13269, pp. 853–871 (2022)
2. Alagic, G., Apon, D., Cooper, D., Dang, Q., Dang, T., Kelsey, J., Lichtinger, J., Liu, Y.K., Miller, C., Moody, D., et al.: Status report on the third round of the nist post-quantum cryptography standardization process (2022)

3. Alkim, E., Bilgin, Y.A., Cenk, M., Gérard, F.: Cortex-M4 optimizations for {R,M}LWE schemes. IACR TCHES **2020**(3), 336–357 (2020)
4. Andrysco, M., Nötzli, A., Brown, F., Jhala, R., Stefan, D.: Towards Verified, Constant-time Floating Point Operations. In: ACM CCS 2018. pp. 1369–1382 (2018)
5. Bai, S., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schwabe, P., Seiler, G., Stehlé, D.: CRYSTALS-Dilithium: Algorithm Specifications and Supporting Documentation (Version 3.1) (2021)
6. 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 TCHES **2022**(1), 221–244
7. Becker, H., Kannwischer, M.J.: Hybrid scalar/vector implementations of Keccak and SPHINCS+ on AArch64. Cryptology ePrint Archive, Report 2022/1243
8. Becker, H., Mera, J.M.B., Karmakar, A., Yiu, J., Verbauwhede, I.: Polynomial multiplication on embedded vector architectures. IACR TCHES **2022**(1), 482–505 (2022)
9. Becoulet, A., Verguet, A.: A Depth-First Iterative Algorithm for the Conjugate Pair Fast Fourier Transform. IEEE Transactions on Signal Processing **69**, 1537–1547 (2021). https://doi.org/10.1109/TSP.2021.3060279
10. Bennett, H., Ganju, A., Peetathawatchai, P., Stephens-Davidowitz, N.: Just how hard are rotations of $\mathbb{Z}^n$? Algorithms and cryptography with the simplest lattice. Cryptology ePrint Archive, Report 2021/1548 (2021)
11. Bernstein, D.J., Hülsing, A., Kölbl, S., Niederhagen, R., Rijneveld, J., Schwabe, P.: The SPHINCS+ Signature Framework. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (Nov 2019)
12. Bindel, N., McCarthy, S., Twardokus, G., Rahbari, H.: Drive (quantum) safe! – towards post-quantum security for v2v communications. Cryptology ePrint Archive, Paper 2022/483 (2022)
13. Blake, A.M., Witten, I.H., Cree, M.J.: The Fastest Fourier Transform in the South. IEEE Transactions on Signal Processing (2013)
14. Botros, L., Kannwischer, M.J., Schwabe, P.: Memory-Efficient High-Speed Implementation of Kyber on Cortex-M4. In: AFRICACRYPT 19. LNCS, vol. 11627, pp. 209–228 (2019)
15. Buchmann, J., Dahmen, E., Hülsing, A.: XMSS - A Practical Forward Secure Signature Scheme Based on Minimal Security Assumptions. In: Post-Quantum Cryptography, pp. 117–129. Springer Berlin Heidelberg (2011)
16. Chen, L., Jordan, S., Liu, Y.K., Moody, D., Peralta, R., Perlner, R., Smith-Tone, D.: Report on Post-Quantum Cryptography. Tech. Rep. NIST IR 8105, National Institute of Standards and Technology (Apr 2016)
17. 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 (Feb 2021)
18. Cooley, J.W., Tukey, J.W.: An Algorithm for the Machine Calculation of Complex Fourier Series. Mathematics of computation **19**, 297–301 (1965)
19. Cooper, D.A., Apon, D.C., Dang, Q.H., Davidson, M.S., Dworkin, M.J., Miller, C.A., et al.: Recommendation for stateful hash-based signature schemes. NIST Special Publication **SP 800–208** (2020)
20. Dagdelen, Ö., Fischlin, M., Gagliardoni, T.: The Fiat-Shamir Transformation in a Quantum World. In: ASIACRYPT 2013, Part II. LNCS, vol. 8270, pp. 62–81 (2013)

21. Ducas, L., Postlethwaite, E.W., Pulles, L.N., van Woerden, W.: Hawk: Module LIP makes Lattice Signatures Fast, Compact and Simple. Cryptology ePrint Archive, Report 2022/1155 (2022), https://eprint.iacr.org/2022/1155
22. Ducas, L., van Woerden, W.P.J.: On the Lattice Isomorphism Problem, Quadratic Forms, Remarkable Lattices, and Cryptography. In: EUROCRYPT 2022, Part III. LNCS, vol. 13277, pp. 643–673 (2022)
23. 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: Specifications v1.2 (2020)
24. Frigo, M., Johnson, S.G.: FFTW: Fastest Fourier transform in the west. Astrophysics Source Code Library pp. ascl–1201 (2012)
25. Howe, J., Westerbaan, B.: Benchmarking and Analysing the NIST PQC Finalist Lattice-Based Signature Schemes on the ARM Cortex M7. Cryptology ePrint Archive, Paper 2022/405 (2022)
26. Huelsing, A., Butin, D., Gazdag, S.L., Rijneveld, J., Mohaisen, A.: XMSS: eXtended Merkle Signature Scheme. RFC 8391 (May 2018), https://www.rfc-editor.org/info/rfc8391
27. Hülsing, A.: W-OTS+ - Shorter Signatures for Hash-Based Signature Schemes. In: AFRICACRYPT 13. LNCS, vol. 7918, pp. 173–188 (2013)
28. Jalali, A., Azarderakhsh, R., Mozaffari Kermani, M., Campagna, M., Jao, D.: ARMv8 SIKE: Optimized Supersingular Isogeny Key Encapsulation on ARMv8 Processors. IEEE Transactions on Circuits and Systems I: Regular Papers (2019)
29. Kannwischer, M.J., Petri, R., Rijneveld, J., Schwabe, P., Stoffelen, K.: PQM4: Post-quantum crypto library for the ARM Cortex-M4, https://github.com/mupq/pqm4
30. Karmakar, A., Bermudo Mera, J.M., Sinha Roy, S., Verbauwhede, I.: Saber on ARM. IACR Transactions on Cryptographic Hardware and Embedded Systems **2018**(3), 243–266 (Aug 2018)
31. 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
32. Kwon, H., Jang, K., Kim, H., Kim, H., Sim, M., Eum, S., Lee, W.K., Seo, H.: ARMed Frodo. In: Information Security Applications. pp. 206–217. Springer International Publishing, Cham (2021)
33. Lyubashevsky, V.: Fiat-Shamir with Aborts: Applications to Lattice and Factoring-Based Signatures. In: ASIACRYPT 2009. LNCS, vol. 5912, pp. 598–616 (2009)
34. McGrew, D., Curcio, M., Fluhrer, S.: RFC 8554: Leighton-Micali hash-based signatures (2019), https://www.rfc-editor.org/rfc/rfc8554
35. Nguyen, D.T., Gaj, K.: Fast NEON-Based Multiplication for Lattice-Based NIST Post-quantum Cryptography Finalists. In: Post-Quantum Cryptography - 12th International Workshop, PQCrypto 2021. pp. 234–254 (Jul 20–22, 2021)
36. Nguyen, D.T., Gaj, K.: Optimized software implementations of CRYSTALS-Kyber, NTRU, and Saber using NEON-based special instructions of ARMv8. In: Proceedings of the NIST 3rd PQC Standardization Conference (NIST PQC 2021) (2021)
37. Pornin, T.: New Efficient, Constant-Time Implementations of Falcon. Cryptology ePrint Archive, Report 2019/893 (2019), https://eprint.iacr.org/2019/893
38. Seo, H., Sanal, P., Jalali, A., Azarderakhsh, R.: Optimized Implementation of SIKE Round 2 on 64-bit ARM Cortex-A Processors. IEEE Transactions on Circuits and Systems I: Regular Papers **67**(8), 2659–2671 (2020)
39. Shor, P.: Algorithms for quantum computation: Discrete logarithms and factoring. In: Proceedings 35th Annual Symposium on Foundations of Computer Science. pp. 124–134. IEEE Comput. Soc. Press, Santa Fe, NM, USA (1994)

40. Streit, S., De Santis, F.: Post-Quantum Key Exchange on ARMv8-A: A New Hope for NEON Made Simple. IEEE Transactions on Computers (11), 1651–1662 (2018)
41. Zhao, L., Zhang, J., Huang, J., Liu, Z., Hancke, G.: Efficient Implementation of Kyber on Mobile Devices. In: 2021 IEEE 27th International Conference on Parallel and Distributed Systems (ICPADS). pp. 506–513

A Visualizing complex point multiplication

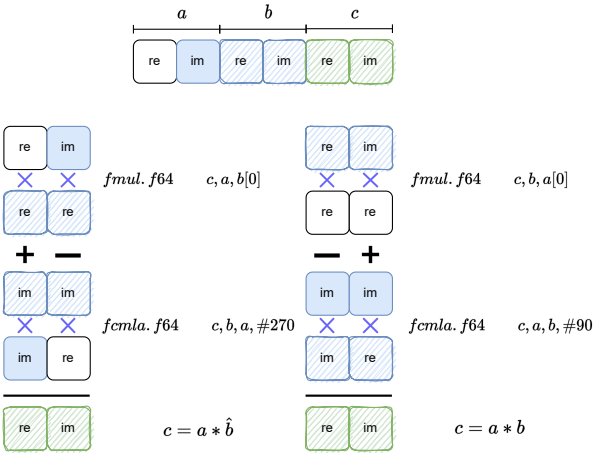

Fig. 2. Single pair complex multiplication using `fmul, fcmla`. Real and imagine points are stored adjacently.

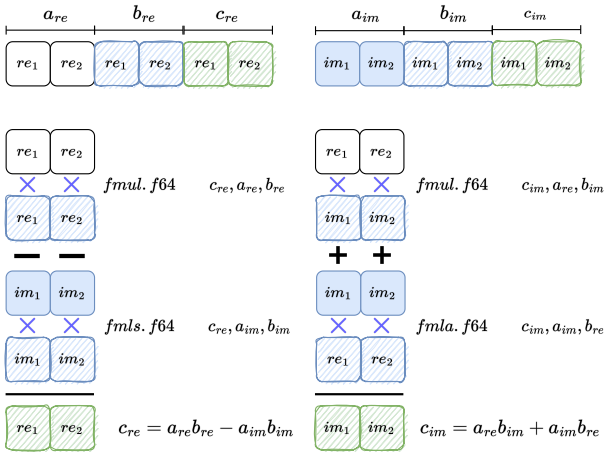

Fig. 3. Two pairs complex multiplication using `fmul, fmls, fmla`. Real and imagine points are stored separately.