# Faster Kyber and Dilithium on the Cortex-M4

Amin Abdulrahman[1,2], Vincent Hwang[3,4], Matthias J. Kannwischer[3], and Daan Sprenkels[5]

[1] Ruhr University Bochum, Germany
amin.abdulrahman@mpi-sp.org
[2] Max Planck Institute for Security and Privacy, Bochum, Germany
[3] Academia Sinica, Taipei, Taiwan
vincentvbh7@gmail.com, matthias@kannwischer.eu
[4] National Taiwan University, Taipei, Taiwan
[5] Digital Security Group, Radboud University, Nijmegen, The Netherlands
daan@dsprenkels.com

**Abstract.** This paper presents faster implementations of the lattice-based schemes Dilithium and Kyber on the Cortex-M4. Dilithium is one of three signature finalists in the NIST post-quantum project (NIST PQC), while Kyber is one of four key-encapsulation mechanism (KEM) finalists. Our optimizations affect the core polynomial arithmetic involving number-theoretic transforms in both schemes. Our main contributions are three-fold: We present a faster signed Barrett reduction for Kyber, propose to switch to a smaller prime modulus for the polynomial multiplications $c\mathbf{s}_1$ and $c\mathbf{s}_2$ in the signing procedure of Dilithium, and apply various known optimizations to the polynomial arithmetic in both schemes. Using a smaller prime modulus is particularly interesting as it allows using the Fermat number transform resulting in especially fast code.
We outperform the state-of-the-art for both Dilithium and Kyber. For Dilithium, our NTT and iNTT are faster by 5.2% and 5.7%. Switching to a smaller modulus results in speed-up of 33.1%–37.6% for the relevant operations (sum of the base multiplication and iNTT) in the signing procedure. For Kyber, the optimizations results in 15.9%–17.8% faster matrix-vector product which is a core arithmetic operation in Kyber.

**Keywords:** Dilithium · Kyber · NIST PQC · Fermat Number Transform · Number-Theoretic Transform · Arm Cortex-M4

## 1 Introduction

Lattice-based cryptography appears to be the most promising family of post-quantum replacements needed for public-key cryptography broken by Shor's algorithm [Sho94]. As lattice-based key encapsulation schemes and digital signatures provide reasonable key, ciphertext, and signature sizes and have particularly good performance on a variety of platforms, they are expected to be standardized soon. One of such standardization efforts is the NIST PQC [Nat] project aiming to find replacements for NIST's standards for key establishment and digital signatures as early as 2024. NIST PQC is nearing the end of its third round

with announcements due in early 2022. Among the third round finalists in the competitions are 5 lattice-based schemes including the three key-encapsulation mechanisms (KEMs) Kyber, NTRU, and Saber as well as the digital signature schemes Dilithium, and Falcon. As there are only two other finalists (Classic McEliece and Rainbow) that are not lattice-based, which both have excessively large keys, it appears very likely that some of the lattice-based schemes are going to be selected for standardization unless there are cryptanalytic breakthroughs.

Lattice-based cryptography is particularly suitable for microcontrollers as the key material is still of manageable size and computational performance is particularly fast with encapsulation and decapsulation in a few milliseconds while signing and verification times in the tens to hundreds of milliseconds. NIST has designated the Arm Cortex-M4 as the primary microcontroller optimization target for NIST PQC, and, hence, it has received the most attention so far.

It appears that the number-theoretic transforms are cores of all high-speed implementations of lattice-based crypto for the Cortex-M4. It is either prescribed in the specification of Dilithium, Falcon, and Kyber, or maintains to be the fastest polynomial multiplication methods in Saber, NTRU [CHK+21], and NTRU Prime [ACC+20].

In this work, we focus on Kyber and Dilithium on the Cortex-M4. They are both part of the "Cryptographic Suite for Algebraic Lattices (CRYSTALS)" and are both designed to benefit from the NTT. We show that even though implementations have been improving for many years, we can still significantly improve the involved arithmetic.

**Contributions.** The contribution of this work is threefold. Firstly, we apply various known techniques from work on the Cortex-M4 optimizing Saber, NTRU, and NTRU Prime. While the techniques are already known, they have so far not been applied to Kyber and Dilithium. This includes (1) the use of Cooley–Tukey butterflies for the inverse NTT of both Kyber and Dilithium previously proposed for Saber in [ACC+21]; (2) the use of floating point registers for caching values in the NTT of Dilithium and Kyber which was first proposed in the context of NTTs for NTRU Prime in [ACC+20]. This allows to merge more layers of the NTT and reduce memory access time for loading twiddle factors; (3) we make use of the "asymmetric multiplication" proposed in [BHK+21] which eliminates some duplicate computation in the base multiplication of Kyber at the cost of extra stack usage; and (4) we use an idea from [CHK+21] to improve the accumulation in the matrix-vector product of Kyber by using a 32-bit accumulator allowing to eliminate some modular reductions at the cost of more stack usage.

Secondly, we present a faster Cortex-M4 instruction sequence to implement a signed Barrett reduction on packed 16-bit values applicable to the Kyber NTT. This immediately improves the Barrett reduction code proposed in [BKS19] from 8 cycles to 6 cycles per packed reduction.

Thirdly, we propose to use a different implementation for computing the product $c\mathbf{s}_1$ as well as $c\mathbf{s}_2$ in Dilithium. Since both $c$ and $\mathbf{s}_1/\mathbf{s}_2$ have very small absolute values, we can switch to a much smaller modulus $q'$ that allows effi-

cient computation of the product. For Dilithium2 and Dilithium5, we make use of the Fermat prime $q' = 257$, which allows using a particularly fast variant of the NTT called the Fermat number transform (FNT), similar to [LMPR08] for SWIFFT. Furthermore, [LMPR08] implements FNT on an Intel processor while we implement FNT on the Cortex-M4 and make use of its barrel shifter. For Dilithium3 the FNT does not work as $\mathbf{s}_1$ and $\mathbf{s}_2$ have larger values. We instead use an incomplete NTT with $q' = 769$ which is still much faster than computing it modulo the original Dilithium prime. To best of our knowledge, we are the first to propose using a smaller modulus for these multiplications within Dilithium.

**Code.** Our code is open-source and available at https://github.com/FasterKyberDilithiumM4/FasterKyberDilithiumM4. We will publish the code alongside the paper under a CC0 copyright waiver.

**Structure.** Section 2 recalls the preliminaries regarding Kyber, Dilithium, and the Cortex-M4. In Section 3 and Section 4, we describe the optimizations applied to Kyber and Dilithium, respectively. Lastly, in Section 5, we present the performance results and compare them to previous work.

## 2 Preliminaries

This section introduces the cryptographic schemes Kyber and Dilithium, which are both part of the Cryptographic Suite for Algebraic Lattices (CRYSTALS). Furthermore we give a brief introduction into the polynomial multiplication using the `NTT`, revisit the Barrett reduction and present relevant details considering our target platform, the Arm Cortex-M4.

### 2.1 Notation

For a prime $q$ and a power of two $n$, we denote the polynomial ring $\mathbb{Z}_q[X]/(X^n + 1)$ by $\mathcal{R}_q$. An element $a \in \mathcal{R}_q$ is represented by a coefficient vector $a_i \in \mathbb{Z}_q$, such that $a = \sum_{i=0}^{n-1} a_i X^i$. We denote polynomials using lower-case letters (e.g., $a$), vectors of polynomials using lower-case boldfaced letters (e.g., $\mathbf{a}$), and matrices of polynomials using upper-case boldfaced letters (e.g., $\mathbf{A}$). We symbolize polynomials, vectors, and matrices inside `NTT`-domain using $\hat{a}$, $\hat{\mathbf{a}}$, and $\hat{\mathbf{A}}$, respectively.

Following the definitions from [BDK$^+$20,ABD$^+$20], for an odd $q$ we define the result of the central reduction $r' = r \bmod {}^{\pm}q$ as the unique element in $[-\frac{q-1}{2}, \frac{q-1}{2}]$ satisfying $r' \equiv r \bmod q$. Similarly, we define the result of $r' = r \bmod {}^{+}q$ as the unique element in $[0, q)$ satisfying $r' \equiv r \bmod q$. For scenarios in which the range of the reduction result does not matter, we write $r' = r \bmod q$.

The function `sampleUniform`$(\cdot)$ samples coefficients for polynomials, vectors of polynomials, or matrices of polynomials from a uniformly random distribution. In case a seed is given as the argument, the output is pseudorandomly generated from the seed.

### 2.2 Polynomial Multiplications using the NTT

The NTT is a variant of the discrete Fourier transform (DFT) defined over finite fields and is commonly used for efficient polynomial multiplications. The efficiency of this strategy is based on the fact that a polynomial multiplication inside NTT domain amounts to the coefficient-wise multiplication of the two polynomials. Specifically, the negacyclic NTT is used for multiplying polynomials in $\mathbb{Z}_q[X]/(X^n+1)$.

Computing the negacyclic NTT can be viewed as the evaluation of a polynomial at powers of a primitive $n$-th root of unity $\zeta_n$ for the polynomial ring $\mathcal{R}_q$ with $q$ prime. Additionally, multiplying all coefficients $a_i$ of $a \in \mathcal{R}_q$ by powers of a $2n$-th root of unity $\zeta_{2n} = \sqrt{\zeta_n}$ is called "twisting" [Ber01].

This comes down to computing

$$\texttt{NTT}(a) = \hat{a} = \sum_{i=0}^{n-1} \hat{a}_i X^i \text{ with } \hat{a}_i = \sum_{j=0}^{n-1} a_j \zeta_{2n}^{j} \zeta_n^{ij}$$

for the forward transform (NTT) and

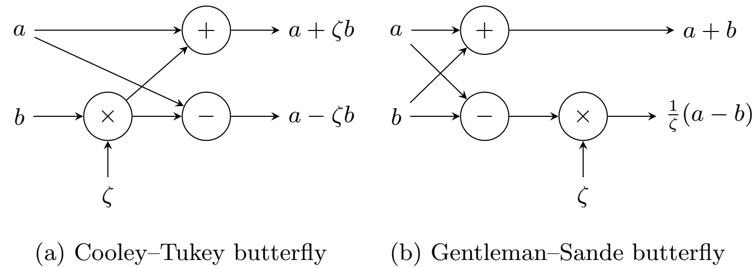
$$\texttt{iNTT}(\hat{a}) = a = \sum_{i=0}^{n-1} a_i X^i \text{ with } a_i = n^{-1} \zeta_{2n}^{-i} \sum_{j=0}^{n-1} \hat{a}_j \zeta_n^{-ij}$$

for the inverse transform (iNTT) [AB74]. The powers of the roots of unity used during the computation of the NTT are also frequently called "twiddle factors".

For computing the NTT itself efficiently, fast Fourier transform (FFT) algorithms, which only require $\Theta(n \log n)$ operations, are commonly used. This algorithm was first described by Gauss in 1805 [Gau66] but it is also oftentimes credited to Cooley and Tukey who published the same algorithm in 1965 [CT65]. The basic idea of the algorithm is to split the computation of a length $n$ NTT into, most commonly, two separate number-theoretic transforms (NTTs) with an input size of $n/2$ each. Formally, we compute the isomorphism $\mathcal{R}_q \to \prod_i \mathbb{Z}_q[X]/(X - \zeta_{2n}^i)$ for $i = 1, 3, 5, \ldots, n-1$ as given by the Chinese Remainder Theorem (CRT), as also explained in [BDK+20, Section 2.2]. For example, in the first instance we map $\mathbb{Z}_q[X]/(X^n+1)$ to $\mathbb{Z}_q[X]/(X^{n/2} - \zeta_{2n}^{n/2}) \times \mathbb{Z}_q[X]/(X^{n/2} + \zeta_{2n}^{n/2})$. This splitting is usually repeated for $\log_2 n$ iterations, called "NTT layers", where the results of the $i$-th layer are the remainders of polynomials $a \bmod (X^{2^{i-1}} \pm \zeta_{2n}^j)$ for some $j$. Computing these remainders involves $n/2$ so-called butterfly operations per layer. The Cooley–Tukey (CT) butterfly, consisting of one addition, one subtraction, and one multiplication in $\mathbb{Z}_q$, is depicted in Figure 1a.

While the CT algorithm is frequently used for computing the NTT, the Gentleman–Sande (GS) FFT algorithm is commonly deployed for computing the iNTT. In contrast to this, we use the CT algorithm for the computation of the NTT and its inverse. A depiction of the GS butterfly is Figure 1b.

Using this method, the product of $f, g \in \mathcal{R}_q$ can be efficiently computed as $\texttt{iNTT}(\texttt{NTT}(f) \circ \texttt{NTT}(g))$, where $\circ$ indicates the base multiplication of two polynomials. In case the NTT is computed on $\log n$ layers, base multiplication is equal to




(a) Cooley–Tukey butterfly (b) Gentleman–Sande butterfly

Fig. 1: NTT butterfly operations

coefficient-wise multiplication requiring only $n$ multiplications. In case the NTT is computed on $l < \log n$ layers, yielding $2^l$ polynomials $\bmod x^m - \omega$ for $m = \frac{n}{2^l}$ and $\omega$ a power of a root of unity, it is called an "incomplete" NTT. For this scenario, the base multiplication corresponds to pairwise $m \times m$ schoolbook multiplications. This idea was initially introduced in [LS19] for the case of the modulus not supporting an NTT on $\log n$ layers, but is also applied for performance reasons in several other implementations, for example, [ABCG20,CHK+21,ACC+21].

### 2.3 Fermat Number Transform

The Fermat number transform (FNT) is a special case of NTT in that the modulus is a Fermat number $F_t := 2^{2^t} + 1$. It was introduced in [SS71] for large integer multiplications and in [AB74,AB75] for digital convolutions. In this paper, we implement FNT for negacyclic convolution. For arbitrary $F_t$ as the modulus, cyclic transformations of sizes dividing $2^{t+2}$ are supported [AB74,AB75]. For computing a negacyclic transformation of size $n = 2^{t+1}$ and $\zeta_{2n} = \sqrt{2}$, the first split becomes

$$\begin{aligned}\mathbb{Z}_{F_t}[X]/(X^n - 2^{2^t}) \quad &\cong \mathbb{Z}_{F_t}[X]/(X^{\frac{n}{2}} - 2^{2^{t-1}}) \times \mathbb{Z}_{F_t}[X]/(X^{\frac{n}{2}} + 2^{2^{t-1}}) \\ &= \mathbb{Z}_{F_t}[X]/(X^{\frac{n}{2}} - 2^{2^{t-1}}) \times \mathbb{Z}_{F_t}[X]/(X^{\frac{n}{2}} - 2^{2^{t-1}(1+2)}).\end{aligned}$$

After applying $t$ layers, all of the polynomial rings are of the form $\mathbb{Z}_{F_t}[x]/(X^{\frac{n}{2^t}} - 2^j)$ where $j$ is an odd number. Since $\zeta_{2n}^2 = 2$, we can apply one more split. Furthermore, if $F_t$ is a prime, then we can compute cyclic transformations of sizes up to $2^{2^t} = F_t - 1$ and negacyclic transformations up to $2^{2^t - 1}$. Since the twiddles in initial $t$ layers are powers of two, we can multiply with the twiddles using shift operations which is much cheaper than explicit multiplications on many platforms. Note that the only known prime Fermat numbers are $F_0 = 3$, $F_1 = 5$, $F_2 = 17$, $F_3 = 257$, $F_4 = 65\,537$. Out of those, only $F_3$ and $F_4$ appear promising for the use in Dilithium. They allow to compute 3 or 4 layers using only shifts.

### 2.4 Kyber

Kyber [ABD+20] is an IND-CCA2-secure lattice-based key-encapsulation mechanism (KEM) constructed from an IND-CPA secure public-key encryption scheme Kyber.CPAPKE using a variant of the Fujisaki–Okamoto (FO) transform [FO99]. The security of the scheme is based on the hardness of the module-learning with errors (MLWE) problem, a trade-off between the ring-learning with errors (RLWE) problem and learning with errors (LWE) problem [ABD+20, Section 1.5]. Kyber is one of four round-three KEM-finalists in the NIST PQC [Nat] next to Saber [DKRV20], NTRU [CDH+20], and Classic McEliece [ABC+20].

**Parameters.** Kyber uses $q = 3329$ as its prime and $n$ is chosen to be 256. Thus, it operates on $\mathcal{R}_q = \mathbb{Z}_{3329}[X]/(X^{256}+1)$ [ABD+20, Section 1.4]. The specification defines three different security levels of Kyber, namely Kyber-512 ($k = 2, \eta_1 = 3$), Kyber-768 ($k = 3, \eta_1 = 2$), and Kyber-1024 ($k = 4, \eta_1 = 2$) [ABD+20, Section 1.4]. Due to the fact that $q$ and $n$ remain constant across the three parameter sets, almost all possible optimizations apply to all variants.

**Notation and Supporting Functions.** We largely follow the notation from the Kyber specification [ABD+20] with only minor deviations. The function $\texttt{sampleCBD}^{\eta}(s)$ samples from a centered binomial distribution in $[-\eta, \eta]$ based on a seed $s$. The symbols $\rho$, $\mu$, and $\sigma$ stand for random bit vectors. The functions `Compress` and `Decompress` handle bit packing, compression, and the serialization of polynomials into byte arrays and vice versa.

**Algorithmic Specification.** Algorithms A.1 to A.3 illustrate the key generation, encryption, and decryption of Kyber.CPAPKE [ABD+20, Algorithms 4–6]. As the optimizations in this paper do not concern the FO transform, we omit the IND-CCA2 scheme and refer to [ABD+20, Algorithms 7–9].

**Number Theoretic Transform.** Since polynomial multiplication is among the most costly operations for Kyber, the polynomial ring has been chosen, such that Kyber can profit from efficient polynomial multiplication using the `NTT`.

For $q = 3329$, as deployed in Kyber, no primitive 512-th but only primitive 256-th roots of unity exist for $\mathcal{R}_q$ with the first one being $\zeta_n = 17$ [ABD+20]. This means that the defining polynomial of $\mathcal{R}_q$ ($X^{256}+1$) factors into 128 polynomials of degree one and not into 256 polynomials of degree zero. Therefore, the result of the NTT of $f \in \mathcal{R}_q$ is a vector of 128 polynomials of degree one. Thus, in contrast to Section 2.2, the coefficients $\hat{a}_i$ inside `NTT` domain are given by

$$\hat{a}_{2i} = \sum_{j=0}^{127} a_{2j} \zeta_n^{(2\mathsf{br}_7(i)+1)j}, \text{ and } \hat{a}_{2i+1} = \sum_{j=0}^{127} a_{2j+1} \zeta_n^{(2\mathsf{br}_7(i)+1)j}$$

as defined in [ABD+20]. The function $\mathsf{br}_7$ computes the bit reversal of a 7-bit integer on its argument.

The absence of a primitive 512-th root of unity also has an impact on the base multiplication of two polynomials inside NTT domain: Instead of coefficient-wise multiplication, we need to perform schoolbook multiplications of size $2 \times 2$, i.e., we need to compute 128 products $\bmod(X^2 - \zeta_n^{2\mathsf{br}_7(i)+1})$ [ABD+20].

### 2.5 Dilithium

Dilithium [DKL+18,BDK+20] is a lattice-based digital signature scheme based on the "Fiat-Shamir with Aborts" approach [Lyu09]. Its security is based on the hardness of the modular short integer solution (MSIS) and MLWE problems and it is currently among the three signature-finalists in the NIST PQC project [Nat], next to Falcon [FHK+20] and Rainbow [CDK+20].

**Parameters.** Dilithium deploys the prime $q = 8380417 = 2^{23} - 2^{13} + 1$ and operates on the polynomial ring $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n + 1)$ with $n = 256$. The two parameters $q$ and $n$ are the same across all parameter sets.

Dilithium offers three different parameter sets, namely Dilithium2, Dilithium3, and Dilithium5, which target the three NIST security levels 2, 3, and 5. More details on the differences between the three parameter sets can be obtained from Table 1. The matrix dimension is given by $(k, l)$, the bounds for sampling the secret key by $\eta$, the number of $\pm 1$ in the challenge polynomial $c$ is $\tau$, and #reps refers to the expected number of repetitions during the rejection sampling in the signature generation process [BDK+20]. The parameters $\gamma_1$ and $\gamma_2$ define the range for the coefficient $\mathbf{y}$ and the low-order rounding range [BDK+20].

Table 1: Overview of Dilithium's parameter sets [BDK+20]

| Scheme | NIST level | $(k, l)$ | $\eta$ | $\tau$ | $\gamma_1$ | $\gamma_2$ | #reps | $\mid pk \mid$ | $\mid sig \mid$ |
|---|---|---|---|---|---|---|---|---|---|
| Dilithium2 | 2 | $(4, 4)$ | 2 | 39 | $2^{17}$ | $(q-1)/88$ | 4.25 | 1312 B | 2420 B |
| Dilithium3 | 3 | $(6, 5)$ | 4 | 49 | $2^{19}$ | $(q-1)/32$ | 5.1 | 1952 B | 3293 B |
| Dilithium5 | 5 | $(8, 7)$ | 2 | 60 | $2^{19}$ | $(q-1)/32$ | 3.85 | 2592 B | 4595 B |

**Notation and Supporting Functions.** We largely follow the notation from the Dilithium specification [BDK+20] with only minor deviations. For symbolizing the concatenation of two inputs as byte strings, the operator $\|$ is used. For $w \in \mathcal{R}_q$, $\|w\|_\infty$ refers to the absolute maximum coefficient $\max_i |w_i \bmod^{\pm} q|$ in $w$. Dilithium internally deploys two different kinds of hash functions: CRH is a collision resistant hash function with an output length of 384 bits, while H is a cryptographic hash function which outputs a polynomial with $\tau$ random coefficients being $\pm 1$ and the rest set to 0 [BDK+20, Section 5.3]. Internally both functions deploy SHAKE256 as their extendable-output function (XOF).

Additionally, Dilithium relies on the supporting functions `ExpandA`, `ExpandMask`, `Power2Round`, `SampleInBall`, `MakeHint`, `UseHint`, `Decompose`, `HighBits`, and `LowBits`. We omit a detailed description for brevity at this point, interested readers may refer to [BDK+20, Section 2, 5].

**Algorithmic Specification.** A simplified description of the key generation, signing, and verification procedures based on the description in [BDK+20, Figure 4] can be found in Algorithms B.1 to B.3.

**Number Theoretic Transform.** Since the main algebraic operations used by Dilithium are polynomial multiplications, Dilithium's ring was chosen in such a way that the `NTT` can be applied [BDK+20]. In contrast to Kyber, for the Dilithium ring, a $2n$-th primitive root of unity $r = 1753$ exists [BDK+20] and thus it is possible to compute a complete `NTT` with eight layers as described in Section 2.2. This allows for base multiplication by coefficient-wise multiplication.

### 2.6 Barrett Reduction

The Barrett reduction [Bar87] is an efficient algorithm for reductions in $\mathbb{Z}_q$. Besides its performance, one advantage is that it can be easily implemented in constant-time. A variant of the Barrett reduction that operates on signed integers has been presented in [Sei18, Algorithm 5] which has also been deployed in a previous implementation of Kyber [ABCG20]. Algorithm 2.1 is an illustration.

**Algorithm 2.1:** Signed Barrett Reduction [ABCG20]

**Input** : $q$ with $0 < q < \frac{\beta}{2}, 2 \nmid q$ and $a$ with $-\frac{\beta}{2} \leq a < \frac{\beta}{2}$
**Output:** $r$ with $r = a \pmod{q}, 0 \leq r \leq q$

1. $v \leftarrow \lfloor \frac{2^{\log(q)-1} \cdot \beta}{q} \rceil$ ▷ precomputed
2. $t \leftarrow \lfloor \frac{av}{2^{\log(q)-1} \cdot \beta} \rfloor$ ▷ signed high product and arithmetic right shift
3. $t \leftarrow tq \bmod \beta$ ▷ signed low product
4. **return** $r \leftarrow a - t$

### 2.7 Arm Cortex-M4

The target platform for our implementation is the Arm Cortex-M4(F), which is a NIST-recommended evaluation platform for the candidates of the NIST PQC project. The Arm Cortex-M4 is based on the Armv7E-M instruction set architecture with 14 usable 32-bit general purpose registers. Additionally, on the Cortex-M4F, there are 32 single-precision floating-point registers [ARM11].

The instruction set also provides a number of powerful digital signal processing (DSP) instructions which allow to perform arithmetic operation on two half words or four bytes at the same time and have proven themselves to be beneficial in numerous implementations [BKS19,ABCG20,KMSRV18] of Kyber [ABD+20], and Saber [DKRV20]. In particular, the instructions `smul{b,t}{b,t}` multiply

specific halfwords and `smla{b,t}{b,t}` multiply specific halfwords and accumulate the product to the specified accumulator. Additionally, the instructions `smuad{,x}` perform two halfword-multiplications and add up their products, while `smlad{,x}` perform two halfword-multiplications and add up their products which is then added to an accumulator. All of these instructions take one one cycle to execute. Moreover, the Cortex-M4 can compute the 64-bit product of two 32-bit values (optionally, with accumulation) in a single cycle. Furthermore, the Cortex-M4 provides a barrel shifter for shifting or rotating the second operand for certain instructions with no additional cost.

On the Cortex-M4, store instructions always take a single cycle, while a sequence of independent loads takes $n+1$ cycles. Using the `vldm` instruction, it is possible to directly load data from the memory into the floating point registers. This also consumes $n+1$ cycles for $n$ data words.

## 3 Improvements to Kyber Implementations

For Kyber, we propose several optimizations for implementing `NTT` and `iNTT` and some speed optimizations to the matrix-vector product at the cost of a higher stack usage. We provide one implementation with all optimizations and one with only the optimizations that do not impact the stack usage.

We base our implementations on [ABCG20] and the implementation in the pqm4 [KRSS19] project. In the following we focus on our contributions and omit details of the numerous optimizations present in previous implementations.

### 3.1 NTT

**Caching in FPU registers.** For Kyber, on the layers 7–4, 15 twiddle factors are required and re-used multiple times throughout the iterations. By using the floating-point registers for caching the twiddle factors, the number of cycles for memory loads are reduced. This technique has been proven to be beneficial in past work [ACC+20,CHK+21,ACC+21]. In our implementations, we load the 15 twiddle factors (packed into eight registers) into the floating-point registers once with `vldm` instruction in nine cycles. Then, in each iteration the twiddle factors are fetched from the floating-point registers with `vmov` in a single cycle each.

On the three remaining layers, it is not beneficial to make use of the floating point registers because in each of the 16 iterations at least one unique twiddle factor per layer is required, meaning none of the twiddle factors are re-used.

**Better Layer Merging.** In our implementations we make use of the common optimization strategy of merging layers of the `NTT` computation [GOPS13]. The idea behind this strategy is to load multiple coefficients at once such that more than one layer of `NTT` can be computed at a time. This reduces the number of memory operations required at the cost of taking up more registers. The state-of-the-art implementation of Kyber [ABCG20] also deploys this strategy merging layers 7–5 and 4–2 while computing layer 1 separately.

By making use of the floating point registers, we instead implement the NTT by merging layers 7–4 and 3–1. Layers 7–4 can be merged by first computing three layers of NTT on each $(a_1, a_3, a_5, a_7, a_9, a_{11}, a_{13}, a_{15})$ and $(a_0, a_2, a_4, a_6, a_8, a_{10}, a_{12}, a_{14})$ and then combining their results. First, the NTT on $(a_1, a_3, \ldots, a_{15})$ is computed and each of the layer 5 outputs is multiplied by the corresponding twiddle factors of the fourth layer. Then, $(a_1, a_3, \ldots, a_{15})$ are moved to the floating point registers for later use. After that, the polynomials $(a_0, a_2, \ldots, a_{14})$ are loaded and the NTT is computed on them. Finally, we vmov $(a_0, a_2, \ldots, a_{14})$ one at a time and compute the final add-sub. In summary, this requires 128 additional vmovs, whereas a separate layer requires 128 loads and 128 stores.

### 3.2 Inverse NTT

The most significant change we apply to the inverse NTT is the switch from Gentleman–Sande butterflies to Cooley–Tukey butterflies. Therefore, all of the optimizations mentioned in the context of the NTT also apply to the inverse NTT.

**Switch to CT-Butterflies.** In previous implementations of Kyber for the Arm Cortex-M4, the NTT was always implemented using CT butterflies, while the inverse NTT was implemented using GS butterflies, which is a commonly seen pattern for implementations using the NTT in general. Opposed to that, we implement the inverse NTT using CT butterflies in order to avoid the necessity of intermediate modular reductions by limiting the coefficients' growths, as for example suggested in [Sei18, Section 2.1] or implemented for Saber in [ACC+21].

Using CT butterflies for the inverse NTT requires to do additional twisting during the computation of the last layer but the total number of multiplications does generally not increase because multiplications in the same amount can be omitted during the butterfly operations ("light butterflies"). One side effect of this approach is that some coefficients will grow larger than in the forward NTT because the multiplications in the butterflies always include reductions and now the operands of the addition and subtraction in the butterfly are not always limited by this. To counteract, we insert two modular multiplications on the fourth layer to limit the growth of the coefficients to be in $(-9q, 9q)$, at most after the fourth layer. By detailed range analysis, we found that on the last three layers we need 20 additional reductions on packed arguments in total.

Moreover, the Montgomery multiplication during the twisting removes the need of a separate Barrett reduction of every coefficient at the end of the last layer. This saves 256 Barrett reductions.

Note that due to the new structure of the iNTT the input coefficients' absolute values need to be smaller than $q$.

### 3.3 Faster Barrett Reduction

Similar to previous implementations, we deploy the Barrett reduction to reduce the coefficients.The Barrett reduction of two 16-bit integers packed in one 32-bit

**Algorithm 3.1:** Packed Barrett Reduction [BKS19]

**Input :** $a = (a_t \,||\, a_b)$
**Output:** $c = (c_t \,||\, c_b) \bmod^{\pm} q$

```
1 smulbb t0, a, ⌊2^26/q⌉
2 smultb t1, a, ⌊2^26/q⌉
3 asr t0, t0, #26
4 asr t1, t1, #26
5 smulbb t0, t0, q
6 smulbb t1, t1, q
7 pkhbt t0, t0, t1, lsl #16
8 usub16 r, a, t0
```

**Algorithm 3.2:** Improved Packed Barrett Reduction

**Input :** $a = (a_t \,||\, a_b)$
**Output:** $c = (c_t \,||\, c_b) \bmod^{\pm} q$

```
1 smlawb t0, −⌊2^32/q⌉, a, 2^15
2 smlabt t0, q, t0, a
3 smlawt t1, −⌊2^32/q⌉, a, 2^15
4 smulbt t1, q, t1
5 add t1, a, t1, lsl #16
6 pkhbt c, t0, t1, lsl #16
```

register has been previously implemented [BKS19] as shown in Algorithm 3.1. Using the `smlaw{b,t}` instructions as in Algorithm 3.2, the cycle count of one Barrett reduction is reduced by one. This means for reducing a packed argument, two cycles are saved. In contrast to the implementation from Algorithm 3.1, the technique presented in Algorithm 3.2 requires two Barrett constants which are both different from the previous one. Moreover, using this optimization removes the guarantee of the reduction's result being in $[0, q)$, instead it will result in $[-\frac{q-1}{2}, \frac{q-1}{2}]$ for an odd $q$. Therefore, its output must not be passed to one of the packing or compression functions because they assume the input to be in $[0, q)$. This means, it may not be used in the `poly_reduce` function but it can be used inside the `NTT` and `iNTT`.

### 3.4 Matrix-Vector Product

For speed optimization of the matrix-vector product, we implement two techniques. Both of them require additional stack space and therefore, if a low memory footprint is a concern, the applicability needs to be checked. Further, we re-implement the C function for the computation of the matrix-vector product in assembly which allows us to significantly lower the number of function calls required by efficiently using the registers and making use of macros. We proceed similarly for the inner product in the decryption.

**Asymmetric Multiplication.** For the computation of the matrix-vector product $\mathbf{As}$ in Kyber, we compute $\texttt{iNTT}(\hat{\mathbf{A}} \circ \texttt{NTT}(\mathbf{s}))$. During this computation, every row of $\hat{\mathbf{A}}$ needs to be multiplied by $\hat{\mathbf{s}}$. Therefore it is a common strategy to cache the result of $\hat{\mathbf{s}}$ instead of recomputing it for every row of $\hat{\mathbf{A}}$ [BKS19]. Using a trick for integer multiplication presented in [BDL+11], [BHK+21] extended the aforementioned concept for which incomplete NTTs are deployed.

Recall that the Kyber `NTT` is incomplete, i.e., 7 instead of 8 layers are computed, and therefore the product of two polynomials inside `NTT`-domain $\hat{a} \circ \hat{s} = \hat{c}$

consists of 128 $2 \times 2$ schoolbook multiplications. For computing $\hat{c}_{2i} + \hat{c}_{2i+1}X = (\hat{a}_{2i} + \hat{a}_{2i+1}X)(\hat{s}_{2i} + \hat{s}_{2i+1}X) \bmod (X^2 - \zeta^{2\mathsf{br}_7(i)+1})$, we have $\hat{c}_{2i} = \hat{a}_{2i}\hat{s}_{2i} + \hat{a}_{2i+1}\hat{s}_{2i+1}\zeta^{2\mathsf{br}_7(i)+1}$ and $\hat{c}_{2i+1} = \hat{a}_{2i}\hat{s}_{2i+1} + \hat{s}_{2i}\hat{a}_{2i+1}$.

The idea behind the proposal from [BHK+21, Section 4.2] is that during the computation of $\hat{\mathbf{A}} \circ \hat{\mathbf{s}}$, each polynomial of $\hat{\mathbf{s}}$ is used $k$ times which means that the computation of $\hat{s}_{2i+1}\zeta^{2\mathsf{br}_7(i)+1}$ is repeated $k$ times. This can be avoided by caching the intermediate results of $\hat{s}_{2i+1}\zeta^{2\mathsf{br}_7(i)+1}$ in a separate vector $\hat{\mathbf{s}}'$.

We implement two separate variants for the base multiplication, one of which is only used for the first row of the matrix in the matrix-vector product, while the other one is used for all of the following ones. The first variant computes the same base multiplication as before except that it stores the result of $\hat{s}_{2i+1}\zeta^{2\mathsf{br}_7(i)+1}$ separately. This comes at the cost of two additional stores and one additional load from the stack for the argument containing the address of $\hat{\mathbf{s}}'$ per two polynomial multiplications. The second variant saves two `smultb` instructions, two montgomery reductions, and the load of one twiddle factor per two polynomials by loading the cached values instead. The precomputed vector can also be re-used in the inner product following the matrix-vector multiplication in encryption.

**Better Accumulation.** We also make use of an improved accumulation strategy in the matrix-vector product as presented in [CHK+21]. For the computation of one element of the output vector in a matrix-vector product, a total number of $k$ base multiplications as well as $k-1$ accumulating additions are required. Instead of reducing each coefficient directly after the base multiplication before accumulating, we delay this step until all three base multiplication results have been accumulated. We also implement this technique for the computation of the inner product. For the implementation, we define three variants of the caching and non-caching base multiplication functions each: One that takes 16-bit input values and writes to a 32-bit output array, one that takes unreduced 32-bit input values and writes to a 32-bit output array, as well as one function that also takes unreduced 32-bit input values but outputs reduced and packed coefficients in a 16-bit integer array. For the second type of the function, the operation on 32-bit values also allows for usage of `smla{b,t}` instead of `smul{b,t}` such that no extra addition is required for the accumulation, compared to the case when computing on packed 16-bit coefficients.

Due to the small size of the Kyber prime, the sum will never overflow a signed 32-bit integer: For the matrix-vector products in Kyber using asymmetric multiplication, possible vector-inputs are the output of an `NTT` which is in $[-\frac{q-1}{2}, \frac{q-1}{2}]$ or the cached Montgomery multiplication result from the asymmetric multiplication which is in $(-q, q)$. The coefficients of the matrix generated using the on-the-fly approach from [BKS19] are smaller than $q$. Therefore, the maximum result for one of the multiplications is $\in (-q^2, q^2)$. For $k$ accumulations with $k \in \{2, 3, 4\}$, we get a maximum absolute intermediate value of $kq^2 = 4q^2 < 2^{31}$.

# 4 Improvements to Dilithium Implementations

For Dilithium we deploy similar strategies for optimizing the NTT and iNTT as for Kyber and optimize the multiplication of $c$ and $\mathbf{s}_1$, as well as $c$ and $\mathbf{s}_2$.

## 4.1 NTT and Inverse NTT

For the NTT, we merge the layers as 7–5, 4–2, 1–0 to reduce the number of memory operations. This differs from the previous implementation [GKS20,GKOS18] where layers 7–6, 5–4, 3–2, and 1–0 are merged. For the iNTT, we similarly switch to CT-butterflies and merge as in the NTT.

**Switch to CT-Butterflies.** Just as for Kyber, we switch to CT butterflies for the computation of the iNTT. Further, we make use of a technique introduced in [ACC+21, Appendix D] which computes light butterflies with one less reduction. As opposed to the Kyber, the coefficients' growth due to the light butterflies is not of concern for the Dilithium since values up to $256q$ fit in a 32-bit register.

## 4.2 Small NTTs for Dilithium

In the signature generation of Dilithium, we recall that the polynomial $c$ consists of $\tau$ $\pm 1$'s and $256 - \tau$ 0's, and all polynomials in $\mathbf{s}_1$ and $\mathbf{s}_2$ consist of elements in $[-\eta, \eta]$. The absolute values of the coefficients in $c\mathbf{s}_1$ and $c\mathbf{s}_2$ are bounded by $\tau\eta$, and the computation can be regarded as in $\mathbb{Z}_{q'}$ for $q' > 2\tau\eta$ [CHK+21, Section 2.4.6]. As far as we know, all implementations choose $q' = 8380417$ and employ the NTT defined for Dilithium. However, since only the correct $c\mathbf{s}_1$ and $c\mathbf{s}_2$ are required, there is some freedom for choosing $q'$. The parameters $\tau \cdot \eta$ are $39 \cdot 2 = 78$ for, $49 \cdot 4 = 196$ for Dilithium3, and $60 \cdot 2 = 120$ for Dilithium5. Consequently, we choose the Fermat number $q' = F_3 = 257$ for Dilithium2 and Dilithium5, and $q' = 769$ for Dilithium3. Alternatively, one can also re-use the Kyber prime $q' = 3329$ for any of the parameters in case re-using the code is of interest. We have also experimented with the Fermat number $q' = F_4 = 65537$ for Dilithium3. However, this did not result in in a speed-up compared to $q' = 769$.

**FNT for Dilithium2 and Dilithium5.** For $q' = 257 = 2^8 + 1$, we have FNT defined over $\mathbb{Z}_{257}[X]/(X^{256} + 1)$. We implement the forward transformation with 7 layers of CT butterflies. Since the input coefficients for $c$, $\mathbf{s}_1$, and $\mathbf{s}_2$ are at most in $[-\eta, \eta]$, we only need very few reductions. Recall that a CT butterfly maps $(a, b)$ to $(a + \omega b, a - \omega b)$, we can implement it with mla and mls. Furthermore, we can also take a closer look at the initial layers. Since $-1 \equiv 2^8 \pmod{257}$, the first layer can be written as $\mathbb{Z}_{257}[X]/(X^{256} + 1) \cong \mathbb{Z}_{257}[X]/(X^{128} - 2^4) \times \mathbb{Z}_{257}[X]/(X^{128} + 2^4)$ and the corresponding CT butterfly maps $(a, b)$ to $(a + 2^4 b, a - 2^4 b)$. We denote such computation as CT_FNT$(a, b, 4)$. Notice that without loading twiddle factors, we can implement CT_FNT$(a, b, \texttt{logW})$ efficiently with barrel shifter as illustrated in Algorithm 4.1.

**Algorithm 4.1:** CT_FNT($a, b$, logW).

**Input :** $(\mathtt{a}, \mathtt{b}) = (a, b)$

**Output:** $(\mathtt{a}, \mathtt{b}) = (a + 2^{\mathtt{logW}} b, a - 2^{\mathtt{logW}} b)$

```
1 add a, a, b, lsl #logW
2 sub b, a, b, lsl #(logW+1)
```

**Algorithm 4.2:** CT_iFNT($a, b$, logW).

**Input :** $(\mathtt{a}, \mathtt{b}) = (a, b)$

**Output:** $(\mathtt{a}, \mathtt{b}) = (a - 2^{\mathtt{logW}} b, a + 2^{\mathtt{logW}} b)$

```
1 sub a, a, b, lsl #logW
2 add b, a, b, lsl #(logW+1)
```

Let iFNT be the inverse of FNT. We first observe that the inverse of $2^k$ can be written as $2^{-k} \equiv 2^{16-k} \equiv -2^{8-k} \pmod{2^8 + 1}$. There are two places where we need to multiply by an inverse of a power of two: (i) the inverses corresponded to the butterflies with $\omega = 2^{\mathtt{logW}}$ in CT_FNT, and (ii) the scaling by $128^{-1}$ at the end of iFNT. We denote CT_iFNT($a, b$, logW) as the function mapping $(a, b)$ to $(a - 2^{\mathtt{logW}} b, a + 2^{\mathtt{logW}} b) = (a + 2^{8+\mathtt{logW}} b, a - 2^{8+\mathtt{logW}} b)$ and implement it with barrel shifter as shown in Algorithm 4.2. Clearly, if CT_FNT($a, b, k$) computes $(a + 2^k b, a - 2^k b)$, then CT_iFNT($a, b, 8 - k$) computes $(a + 2^{-k} b, a - 2^{-k} b)$ which can be used in iFNT. We compute iFNT with four layers of GS butterflies followed by three layers of CT butterflies. During the GS butterflies, since the twiddle factors are also very small, we can replace some of the `mul`, `add`, and `sub` with `mla` and `mls`. For CT butterflies, since the twiddle factors are powers of two, we implement them with Algorithm 4.2. Lastly, at the end of CT butterflies, we merge the twisting by powers of two with the multiplication by $128^{-1}$.

**NTT over 769 for Dilithium3.** For Dilithium3, since the maximum absolute value of $c\mathbf{s}_1$ and $c\mathbf{s}_2$ is bounded by $\tau\eta = 4 \cdot 49 = 196$, we cannot use $q' = 257 < 2 \cdot 196$. We therefore choose $q' = 769$ and modify the NTT and iNTT from Kyber. Except for discarding most of the Barrett reductions, the code is the same.

Recall that for the NTT in Kyber, we require the output to be in $[-\frac{q'}{2}, \frac{q'}{2}]$ for the secret key. However, for Dilithium3, since we are only using 16-bit NTT for computing $c\mathbf{s}_1$ and $c\mathbf{s}_2$, we can remove the Barrett reductions at the end and allow elements growing up to $7q'$ in absolute value.

For the iNTT, replacing with $q' = 769$ allows us to postpone the Barrett reductions by one layer and reduce the number of Barrett reductions by half. At the end of iNTT, we replace the 16-bit Montgomery multiplication with straight multiplication and 32-bit Barrett reduction. By using 32-bit Barrett reduction, the result is within $[-384, 384]$ if the product is in $[-113025697, 113025697]$. Since $\log_2(\frac{113025697}{384}) \approx 18.17$, we derive values in $[-384, 384]$ by applying 32-bit Barrett reduction to the product of any signed 16-bit value and any constant from $[-384, 384]$. The downside for using 32-bit Barrett reduction is a slightly higher register pressure, but overall it is more favorable because we don't need to reduce them again. This is different from the 16-bit NTT in [ACC+21]. They implemented the twist with Montgomery multiplication and then reduced the result to $[-384, 384]$ with an additional 32-bit Barrett reduction.

## 5 Results

In this section, we present the implementations results of Kyber and Dilithium.

### 5.1 Benchmarking setup

Our concrete hardware target is the STM32F4DISCOVERY with the STM32-F407VG MCU, which also is the target of previous publications concerning implementations of post-quantum schemes on microcontrollers. It comes with 1 MiB of flash memory, and 192 KiB of RAM.

Our benchmarking setup is based on pqm4 [KRSS19]. During the benchmarks, we clock the microcontroller at 24 MHz in order to avoid wait states during memory operations. We compile the code using `arm-none-eabi-gcc` version 10.2.1 with the `-O3` option. Regarding the Keccak implementation, we make use of the code provided in pqm4. For the randomness generation we rely on the microcontroller's hardware random number generator (RNG).

We compare our Kyber implementations to the code currently present in pqm4 which is based on the work in [ABCG20] and [BKS19]. Similarly, we compare our implementations of Dilithium (2 and 3) to the code in pqm4 which is based on [GKS20]. For Dilithium5, pqm4 does not currently have an implementation due to a lack of stack space. We apply some of the stack optimizations of [GKS20] to our implementations, especially to make Dilithium5 work as well. It is important to note that the parameters of Kyber and Dilithium were changed at the start of the third round of the NISTPQC competition. The numbers presented here reflect the round 3 versions contained in pqm4. Those are optimizations from the original papers ported to the third round parameters. The performance results for the full schemes do not match the original publications.

### 5.2 Performance of NTT-Related Functions

In Table 2, we present the cycle counts for the transformations we deploy in our implementations of Kyber and Dilithium. For the Kyber `NTT`, we achieve a speedup of 12.6%. Regarding the Kyber `iNTT`, we obtain a speedup of up-to 21.3%. Note that for the stack-optimized variant an additional reduction is required before the `iNTT` because of the absence of asymmetric multiplication.

We achieve a speedup of 5.2% for the Dilithium `NTT`, and 5.7% for the `iNTT`. For the small NTTs the metric we are optimizing is $(k + l) \cdot \texttt{NTT} + \#\text{reps} \cdot (\texttt{NTT} + (k + l) \cdot (\text{basemul} + \texttt{iNTT}))$. As most of the small `NTT` are computed outside of the loop, we moved some of the reductions into the `NTT` resulting in a faster basemul. Note that for $q = 257$ and $q = 769$ the `NTT` and `iNTT` have very close performance, but the basemul differs. This results in the `FNT` being advantageous for Dilithium2 and Dilithium5. For (basemul + `iNTT`), we achieve a speedup of 37.6% for $q = 257$, and 33.1% for $q = 769$ compared to $q = 8380417$ from [GKS20]. We also compare our $q = 769$ implementation to an existing one by [ACC+21], because theoretically, their 6-layer approach could also be used as well. Since the computation is dominated by (basemul + `iNTT`), we find that

our 7-layer approach is faster. We also carefully examine the code by [ACC+21], and find that the last 32-bit Barrett reduction is performed outside the reported `iNTT`, so the speedup is more.

Table 2: Cycle counts for transformation operations of Kyber and Dilithium. `NTT` and `iNTT` correspond to the schemes default transformations, i.e., $q = 3329$ for Kyber and $q = 8380417$ for Dilithium. The `NTT` with $q = 257$ is deployed for Dilithium2 and Dilithium5, and the `NTT` with $q = 769$ is used used for Dilithium3.

| | Prime | Implementation | `NTT` | `iNTT` | basemul |
|---|---|---|---|---|---|
| Kyber | $q = 3329$ | [ABCG20] | 6 852 | 6 979 | 2 317 |
| | | This work | 5 992 | 5 491/6 282[a] | 1 613[b] |
| Dilithium | $q = 8380417$ | [GKS20] | 8 540 | 8 923 | 1 955 |
| | | This work | 8 093 | 8 415 | 1 955 |
| | $q = 257$ | This work | 5 524 | 5 563 | 1 225 |
| | $q = 769$ | [ACC+21] (6-layer) | 4 852 | 4 817 | 2 966 |
| | | This work | 5 200 | 5 537 | 1 740 |

[a] First value is for speed-optimization, second for stack-optimization.
[b] Asymmetric basemul as used in the IP (enc). As the basemul in the MVP and IP consists of individual function calls, the cycle count is not straight forward to measure.

Table 3 contains the result for our benchmarks of the MVP and inner product (IP) functions as deployed in Kyber. For the MVP, we consider the MVP as it is computed in the key generation. The MVP in the encryption is similar but contains $k$ NTTs less. Note that in the actual implementation of Kyber, the MVP is interleaved with the on-the-fly generation of the matrix. For ease of comparison, we additionally provide benchmarks for a stripped down variant of the MVP excluding the hashing. Regarding our benchmarks, we count the caching for the asymmetric multiplication towards the MVP although the IP for the encryption also benefits of this pre-computation. For the same reasons as for the MVP, the benchmarks of our IP functions only include the NTTs, the base multiplications, and deserialization, if applicable. For the speed optimized MVP implementation, we get speedups between 15.9% and 17.8% (excl. hashing). The stack optimized variant, achieves speedups between 12.1% and 12.5%. We achieve speedups of 26.9%–31.7% (enc) and 21.6%–23.3% (dec) for the speed optimized inner product, while for the stack variant we obtain speedups of 4%–6.3% and 17.3%–18.9%, respectively. We observe that for larger $k$, the speed optimization strategy gives increasingly lower cycle counts due to asym. multiplication.

### 5.3 Performance of Schemes

Per Table 4, we achieve speedups of 3.3%–4.2%, 3.1%–3.6%, and 5.1%–5.2% for the key generation, encapsulation, and decapsulation our speed optimized

Table 3: Cycle counts for matrix-vector and inner products used in Kyber.

| implementation | variant | operation | Kyber-512 | Kyber-768 | Kyber-1024 |
|---|---|---|---|---|---|
| pqm4 | | Matrix-Vector Product[a] | 66 291 | 127 634 | 209 517 |
| | | Matrix-Vector Product[b] | 226 580 | 484 077 | 840 498 |
| | | Inner Product (enc) | 11 978 | 14 696 | 17 429 |
| | | Inner Product (dec) | 29 888 | 41 910 | 53 792 |
| This work | speed | Matrix-Vector Product[a] | 55 746 | 106 380 | 172 152 |
| | | Matrix-Vector Product[b] | 211 606 | 457 213 | 796 349 |
| | | Inner Product (enc) | 8 762 | 10 331 | 11 898 |
| | | Inner Product (dec) | 23 425 | 32 354 | 41 275 |
| | stack | Matrix-Vector Product[a] | 58 028 | 112 503 | 184 149 |
| | | Matrix-Vector Product[b] | 214 053 | 463 590 | 808 206 |
| | | Inner Product (enc) | 11 218 | 13 877 | 16 733 |
| | | Inner Product (dec) | 24 722 | 34 167 | 43 619 |

[a] Measurement excluding the hashing.
[b] Measurement including the hashing.

variant. As to be expected due to the caching of intermediate values for speed optimizations, our speed implementation has a higher stack usage. Our stack implementations use essentially the same stack as previous work.

Table 4: Cycle counts and stack usage for Kyber for the key generation, encapsulation, and decapsulation. Cycle counts are averaged over 100 executions.

| implementation | variant | | Kyber-512 | | Kyber-768 | | Kyber-1024 | |
|---|---|---|---|---|---|---|---|---|
| | | | cc | stack [B] | cc | stack [B] | cc | stack [B] |
| pqm4, [ABCG20] | | K | 458k | 2 220 | 745k | 3 100 | 1 188k | 3 612 |
| | | E | 553k | 2 308 | 899k | 2 780 | 1 373k | 3 292 |
| | | D | 513k | 2 324 | 839k | 2 804 | 1 294k | 3 324 |
| This work | speed | K | 443k | 4 272 | 718k | 5 312 | 1 138k | 6 336 |
| | | E | 536k | 5 376 | 870k | 6 416 | 1 324k | 7 432 |
| | | D | 487k | 5 384 | 796k | 6 432 | 1 227k | 7 448 |
| | stack | K | 444k | 2 220 | 724k | 2 736 | 1 149k | 3 256 |
| | | E | 540k | 2 308 | 879k | 2 808 | 1 341k | 3 328 |
| | | D | 492k | 2 324 | 807k | 2 824 | 1 246k | 3 352 |

Table 5 contains the results for Dilithium. We achieve consistent speedups for all parameter sets. The absolute savings due to our optimizations are clearly seen, particularly in signing. The speedup for signing ranges from 1.5% to 5.6%. In relative terms, the impact of our optimizations on the full Kyber and Dilithium seem relatively small compared to the speedups we gain for the polynomial

arithmetic. This is due to dominance of the hashing operations as thoroughly analyzed in previous work [KRSS19].

Table 5: Cycle counts and stack usage for Dilithium. K, S, and V correspond to the key generation, signature generation, and signature verification. Cycle counts are averaged over 10000 executions.

| implementation | variant | | Dilithium2 | | Dilithium3 | | Dilithium5 | |
|---|---|---|---|---|---|---|---|---|
| | | | cc | stack [B] | cc | stack [B] | cc | stack [B] |
| pqm4, [GKS20] | | K | 1 602k | 38k | 2 835k | 61k | 4 836k | 98k |
| | | S | 4 336k | 49k | 6 721k | 74k | 9 037k | 115k |
| | | V | 1 579k | 36k | 2 700k | 58k | 4 718k | 93k |
| This work | speed | K | 1 596k | 8 508 | 2 827k | 9 540 | 4 829k | 11 696 |
| | | S | 4 093k | 49k | 6 623k | 69k | 8 803k | 116k |
| | | V | 1 572k | 36k | 2 692k | 58k | 4 707k | 93k |

**Acknowledgments** This work has been supported by the European Commission through the ERC Starting Grant 805031 (EPOQUE), the Sinica Investigator Award AS-IA-109-M01, and the Taiwan Ministry of Science and Technology Grant 109-2221-E-001-009-MY3. We thank Bo-Yin Yang for sharing the idea of 16-bit Barrett reductions.

## References

AB74. Ramesh C. Agarwal and C. Sidney Burrus. Fast convolution using Fermat number transforms with applications to digital filtering. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 22(2):87–97, 1974.

AB75. Ramesh C. Agarwal and C. Sidney Burrus. Number theoretic transforms to implement fast digital convolution. *Proceedings of the IEEE*, 63(4):550–560, 1975.

ABC+20. Martin R. Albrecht, Daniel J. Bernstein, Tung Chou, Carlos Cid, Jan Gilcher, Tanja Lange, Varun Maram, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Kenneth G. Paterson, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, Jakub Szefer, Cen Jung Tjhai, Martin Tomlinson, and Wen Wang. Classic McEliece. Submission to the NIST Post-Quantum Cryptography Standardization Project [Nat], 2020. https://classic.mceliece.org/.

ABCG20. Erdem Alkim, Yusuf Alper Bilgin, Murat Cenk, and François Gérard. Cortex-M4 Optimizations for {R,M}LWE Schemes. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(3):336–357, Jun. 2020.

ABD+20. Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler,

and Stehlé Damien. *CRYSTALS-Kyber: Algorithm Specifications And Supporting Documentation (version 3.0)*. Submission to round 3 of the NIST post-quantum project [Nat], October 2020.

ACC+20. Erdem Alkim, Dean Yun-Li Cheng, Chi-Ming Marvin Chung, Hülya Evkan, Leo Wei-Lun Huang, Vincent Hwang, Ching-Lin Trista Li, Ruben Niederhagen, Cheng-Jhih Shih, Julian Wälde, and Bo-Yin Yang. Polynomial multiplication in NTRU Prime: Comparison of optimization strategies on Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(1):217–238, Dec. 2020.

ACC+21. Amin Abdulrahman, Jiun-Peng Chen, Yu-Jia Chen, Vincent Hwang, Matthias J. Kannwischer, and Bo-Yin Yang. Multi-moduli NTTs for Saber on Cortex-M3 and Cortex-M4. Cryptology ePrint Archive, Report 2021/995, 2021. https://ia.cr/2021/995.

ARM11. ARM. *Cortex-M4 Devices Generic User Guide*. ARM, August 2011.

Bar87. Paul Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In Andrew M. Odlyzko, editor, *Advances in Cryptology — CRYPTO' 86*, pages 311–323, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.

BDK+20. Shi Bai, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. *CRYSTALS-Dilithium: Algorithm Specifications And Supporting Documentation (version 3.0)*. Submission to round 3 of the NIST post-quantum project [Nat], October 2020.

BDL+11. Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2011*, pages 124–142, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

Ber01. Daniel J. Bernstein. Multidigit multiplication for mathematicians, 2001.

BHK+21. Hanno Becker, Vincent Hwang, Matthias J. Kannwischer, Bo-Yin Yang, and Shang-Yi Yang. Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1. Cryptology ePrint Archive, Report 2021/986, 2021. https://ia.cr/2021/986.

BKS19. Leon Botros, Matthias J. Kannwischer, and Peter Schwabe. Memory-efficient high-speed implementation of Kyber on Cortex-M4. In Johannes Buchmann, Abderrahmane Nitaj, and Tajjeeddine Rachidi, editors, *Progress in Cryptology – AFRICACRYPT 2019*, pages 209–228, Cham, 2019. Springer International Publishing.

CDH+20. Cong Chen, Oussama Danba, Jeffrey Hoffstein, Andreas Hulsing, Joost Rijneveld, John M. Schanck, Peter Schwabe, William Whyte, Zhenfei Zhang, Tsunekazu Saito, Takashi Yamakawa, and Keita Xagawa. NTRU. Submission to the NIST Post-Quantum Cryptography Standardization Project [Nat], 2020. https://ntru.org/.

CDK+20. Ming-shing Chen, Jintai Ding, Matthias Kannwischer, Jacques Patarin, Albrecht Petzoldt, Dieter Schmidt, and Bo-Yin Yang. Rainbow. Submission to round 3 of the NIST post-quantum project [Nat], 2020.

CHK+21. Chi-Ming Marvin Chung, Vincent Hwang, Matthias J. Kannwischer, Gregor Seiler, Cheng-Jhih Shih, and Bo-Yin Yang. 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.

CT65. James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

DKL+18. Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Dilithium: A lattice-based digital signature scheme. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(1):238–268, Feb. 2018.

DKRV20. Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. SABER. Submission to round 3 of the NIST post-quantum project [Nat], 2020.

FHK+20. Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Prest, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. FALCON. Submission to round 3 of the NIST post-quantum project [Nat], 2020. https://falcon-sign.info/.

FO99. Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 537–554. Springer, 1999.

Gau66. Carl Friedrich Gauss. Theoria Interpolationis Methodo Nova Tractata. *Nachlass*, (3):265–330, 1866.

GKOS18. Tim Güneysu, Markus Krausz, Tobias Oder, and Julian Speith. Evaluation of Lattice-Based Signature Schemes in Embedded Systems. *2018 25th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pages 385–388, 2018.

GKS20. Denisa O. C. Greconici, Matthias J. Kannwischer, and Daan Sprenkels. Compact Dilithium implementations on Cortex-M3 and Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(1):1–24, Dec. 2020.

GOPS13. Tim Güneysu, Tobias Oder, Thomas Pöppelmann, and Peter Schwabe. Software speed records for lattice-based signatures. In *PQCrypto*, 2013.

KMSRV18. Angshuman Karmakar, Jose Mera, Sujoy Sinha Roy, and Ingrid Verbauwhede. Saber on ARM: CCA-secure module lattice-based key encapsulation on ARM. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 243–266, 08 2018.

KRSS19. Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. pqm4: Testing and Benchmarking NIST PQC on ARM Cortex-M4. Second NIST PQC Standardization Conference, 2019.

LMPR08. Vadim Lyubashevsky, Daniele Micciancio, Chris Peikert, and Alon Rosen. SWIFFT: A Modest Proposal for FFT Hashing. In Kaisa Nyberg, editor, *Fast Software Encryption*, pages 54–72, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

LS19. Vadim Lyubashevsky and Gregor Seiler. NTTRU: Truly Fast NTRU Using NTT. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(3):180–201, May 2019.

Lyu09. Vadim Lyubashevsky. Fiat-Shamir with Aborts: Applications to lattice and factoring-based signatures. In Mitsuru Matsui, editor, *Advances in Cryptology – ASIACRYPT 2009*, pages 598–616, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

Nat. National Institute of Standards and Technology. Post-Quantum Cryptography Standardization Project. Accessed: 04/04/2021.

Sei18. Gregor Seiler. Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography, 2018. Report 2018/039.

Sho94. P.W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *FOCS 1994*, pages 124–134. IEEE, 1994.

SS71. Arnold Schönhage and Volker Strassen. Schnelle Multiplikation großer Zahlen. *Computing*, 7(3-4):281–292, 1971.

# A Kyber

**Algorithm A.1:** Kyber-.CPAPKE key generation

**Output:** public key: $pk = (\hat{\mathbf{t}}, \rho)$
**Output:** secret key: $sk = (\hat{\mathbf{s}})$
1 $\rho, \sigma \in \{0,1\}^{256} \leftarrow \texttt{sampleUniform}()$
2 $\hat{\mathbf{A}} \in \mathcal{R}_q^{k\times k} \leftarrow \texttt{sampleUniform}(\rho)$
3 $\mathbf{s}, \mathbf{e} \in \mathcal{R}_q^{k\times 1} \leftarrow \texttt{sampleCBD}^{\eta_1}(\sigma)$
4 $\hat{\mathbf{t}} \leftarrow \hat{\mathbf{A}} \circ \texttt{NTT}(\mathbf{s}) + \texttt{NTT}(\mathbf{e})$
5 **return** $(pk, sk)$

**Algorithm A.2:** Kyber-.CPAPKE decryption

**Input :** secret key: $sk = (\hat{\mathbf{s}})$
**Input :** compressed ciphertext: $(\mathbf{u}', v')$
**Output:** message $m \in \mathcal{R}_q$
1 $\mathbf{u} \leftarrow \texttt{Decompress}(\mathbf{u}')$
2 $v \leftarrow \texttt{Decompress}(v')$
3 **return** $m \leftarrow v - \texttt{iNTT}(\hat{\mathbf{s}}^T \circ \texttt{NTT}(\mathbf{u}))$

**Algorithm A.3:** Kyber-.CPAPKE encryption

**Input :** public key: $pk = (\hat{\mathbf{t}}, \rho)$
**Input :** message: $m \in \mathcal{R}_q$
**Input :** random coins: $\mu \in \{0,1\}^{256}$
**Output:** ciphertext $(\mathbf{u}', v')$
1 $\hat{\mathbf{A}} \in \mathcal{R}_q^{k\times k} \leftarrow \texttt{sampleUniform}(\rho)$
2 $\mathbf{r} \in \mathcal{R}_q^{k\times 1} \leftarrow \texttt{sampleCBD}^{\eta_1}(\mu)$
3 $\mathbf{e}_1 \in \mathcal{R}_q^{k\times 1}, e_2 \in \mathcal{R}_q \leftarrow \texttt{sampleCBD}^{\eta_2}(\mu)$
4 $\hat{\mathbf{r}} \leftarrow \texttt{NTT}(\mathbf{r})$
5 $\mathbf{u} \leftarrow \texttt{iNTT}(\hat{\mathbf{A}}^T \circ \hat{\mathbf{r}}) + \mathbf{e}_1$
6 $v \leftarrow \texttt{iNTT}(\hat{\mathbf{t}}^T \circ \hat{\mathbf{r}}) + e_2 + m$
7 **return** $(\texttt{Compress}(\mathbf{u}), \texttt{Compress}(v))$

# B Dilithium

**Algorithm B.1:** Dilithium key generation

**Output:** secret key $sk = (\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0)$
**Output:** public key $pk = (\rho, \mathbf{t}_1)$
1 $\rho, \varsigma, K \in \{0,1\}^{256} \leftarrow \texttt{sampleUniform}()$
2 $\mathbf{s}_1 \in [-\eta, \eta]^{l\times 1}, \mathbf{s}_2 \in [-\eta, \eta]^{k\times 1} \leftarrow \texttt{sampleUniform}(\varsigma)$
3 $\hat{\mathbf{A}} \in \mathcal{R}_q^{k\times l} \leftarrow \texttt{ExpandA}(\rho)$
4 $\mathbf{t} \leftarrow \texttt{iNTT}(\hat{\mathbf{A}} \circ \texttt{NTT}(\mathbf{s}_1)) + \mathbf{s}_2$
5 $(\mathbf{t}_1, \mathbf{t}_0) \leftarrow \texttt{Power2Round}(\mathbf{t})$
6 $tr \in \{0,1\}^{384} \leftarrow \texttt{CRH}(\rho \| \mathbf{t}_1)$
7 **return** $(pk, sk)$

**Algorithm B.2:** Dilithium signing

**Input** : secret key $sk = (\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0)$
**Input** : message: $M \in \{0,1\}^*$
**Output:** signature $\sigma = (\mathbf{z}, \mathbf{h}, c)$

1 $\hat{\mathbf{A}} \in \mathcal{R}_q^{k \times l} \leftarrow \texttt{ExpandA}(\rho)$
2 $\mu \in \{0,1\}^{384} \leftarrow \texttt{CRH}(tr \| M)$
3 $\kappa \leftarrow 0, (\mathbf{z}, \mathbf{h}) \leftarrow \perp$
4 $\rho' \in \{0,1\}^{384} \leftarrow \texttt{CRH}(K \| \mu)$
5 $\hat{\mathbf{s}}_1 \leftarrow \texttt{NTT}(\mathbf{s}_1)$, $\hat{\mathbf{s}}_2 \leftarrow \texttt{NTT}(\mathbf{s}_2)$, $\hat{\mathbf{t}}_0 := \texttt{NTT}(\mathbf{t}_0)$
6 **while** $(\mathbf{z}, \mathbf{h}) = \perp$ **do**
7 | $\mathbf{y} \in \mathcal{R}_q^{l \times 1} \leftarrow \texttt{ExpandMask}(\rho', \kappa)$
8 | $\mathbf{w} \leftarrow \texttt{iNTT}(\hat{\mathbf{A}} \circ \texttt{NTT}(\mathbf{y}))$
9 | $\mathbf{w}_1 \leftarrow \texttt{HighBits}(\mathbf{w}, 2\gamma_2)$
10 | $c \leftarrow \texttt{SampleInBall}(\texttt{H}(\mu \| \mathbf{w}_1))$
11 | $\hat{c} \leftarrow \texttt{NTT}(c)$
12 | $\mathbf{z} \leftarrow \mathbf{y} + \texttt{iNTT}(\hat{c} \circ \hat{\mathbf{s}}_1)$
13 | $\mathbf{r}_0 \leftarrow \texttt{LowBits}(\mathbf{w} - \texttt{iNTT}(\hat{c} \circ \hat{\mathbf{s}}_2), 2\gamma_2)$
14 | **if** $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$ or $\|\mathbf{r}\|_\infty \geq \gamma_2 - \beta$ **then**
15 | | $(\mathbf{z}, \mathbf{h}) \leftarrow \perp$
16 | **else**
17 | | $\mathbf{h} \leftarrow \texttt{MakeHint}(-\texttt{iNTT}(\hat{c} \circ \hat{\mathbf{t}}_0), \mathbf{w} - \texttt{iNTT}(\hat{c} \circ \hat{\mathbf{s}}_2 + \texttt{iNTT}(\hat{c} \circ \hat{\mathbf{t}}_0)), 2\gamma_2)$
18 | | **if** $\|\texttt{iNTT}(\hat{c} \circ \hat{\mathbf{t}}_0)\|_\infty \geq \gamma_2$ or # of 1's in $\mathbf{h} > \omega$ **then**
19 | | | $(\mathbf{z}, \mathbf{h}) \leftarrow \perp$
20 | $\kappa \leftarrow \kappa + l$
21 **return** $\sigma = (\mathbf{z}, \mathbf{h}, c)$

**Algorithm B.3:** Dilithium verification

**Input** : public key $pk = (\rho, \mathbf{t}_1)$
**Input** : message: $M \in \{0,1\}^*$
**Input** : signature $\sigma = (\mathbf{z}, \mathbf{h}, c)$
**Output:** signature valid or signature invalid

1 $\hat{\mathbf{A}} \in \mathcal{R}_q^{k \times l} \leftarrow \texttt{ExpandA}(\rho)$
2 $\mu \in \{0,1\}^{384} \leftarrow \texttt{CRH}(\texttt{CRH}(\rho \| \mathbf{t}_1) \| M)$
3 $\mathbf{w}'_1 \leftarrow \texttt{UseHint}(\mathbf{h}, \texttt{iNTT}(\hat{\mathbf{A}} \circ \texttt{NTT}(\mathbf{z}) - \texttt{NTT}(c) \circ \texttt{NTT}(2^d \cdot \mathbf{t}_1)))$
4 **if** $\|\mathbf{z}\|_\infty < \gamma_1 - \beta$ and $c = \texttt{CRH}(\mu \| \mathbf{w}'_1)$ and # of 1's in $\mathbf{h} \leq \omega$ **then**
5 | **return** signature valid
6 **else**
7 | **return** signature invalid