Revisiting Keccak and Dilithium Implementations on ARMv7-M

in TCHES 2024, Issue 2

Authors: **Junhao Huang**, Alexandre Adomnicăi, Jipeng Zhang, Wangchen Dai, Yao Liu, Ray C. C. Cheung, Çetin Kaya Koç, Donglong Chen*

September 5, 2024

Outline

01 Introduction

02 Keccak Optimizations on ARMv7-M

03 Dilithium Optimizations on ARMv7-M

04 Results and Conclusions

01

Introduction

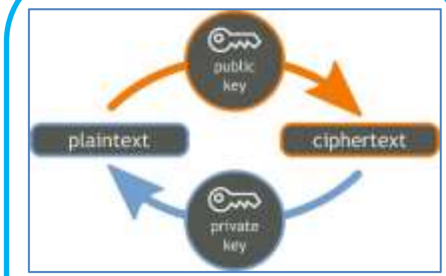
- 1.1 Background
- 1.2 Target Platforms

1.1.1 Quantum Computers

Quantum computers are being developed rapidly. **Shor's algorithm** in quantum computers would break the existing **public-key cryptosystem (PKC)** in **polynomial time**.

2019: Google's **54b** machine;
2021: **66b** Zuchongzhi;
2021: IBM's **126b** Eagle;
2022: IBM's **433b** Osprey.

Shor



RSA

ElGamal

ECC

Threat

Protocols

DNS, TLS, SSH...

Browsers

Chrome, Safari, Edge…

Messaging

WhatsApp, Facebook, Wechat…

This prompted the cryptographic community to search for **suitable alternatives** to traditional PKC.

1.1.2 NIST PQC Project

NIST initiated a standardization project in 2016 to solicit, evaluate, and standardize the **post-quantum cryptographic algorithms (PQC).**

Table 1: Round 3 and Round 4 NIST PQC finalists

Round	Round 3		Round 4	
Types	KEM	DSA	KEM	DSA
Schemes	**Kyber**	**Dilithium**	**Kyber (ML-KEM)**	**Dilithium (ML-DSA)**
	Saber	**Falcon**	-	**Falcon**
	NTRU	Rainbow	-	Sphincs+ (SLH-DSA)
	Classic McEliece	-	-	-

Lattice-Based Cryptography (LBC) is the most promising alternative in terms of security and efficiency:

- **Round 3:** 5 out of 7 candidates belong to LBC;
- **Round 4:** 3 out of 4 finalists belong to LBC.

1.1.3 LBC Core Operations

LBC core operations

- **Symmetric cryptographic primitives:** **SHA-3;**
- **Polynomial multiplication:** **NTT/INTT, pointwise multiplication;**

1. **Symmetric cryptographic primitives SHA-3** accounts for over **70% running-time** according to pqm4. The state-of-the-art Keccak implementations on ARMv7-M is based on the **XKCP library [BDH+]** by Keccak team. The most related work [BK22] studied Keccak optimizations on AArch64. However, these techniques have not been applied to ARMv7-M yet.
2. **(Inverse) Number Theoretic Transform (NTT) :** It is a generalization of the **classic discrete Fourier transform (DFT)** in finite fields. In brief, NTT can reduce the time complexity of multiplying two n-degree polynomial $a = \sum a_i x^i, b = \sum b_i x^i$ from $\boldsymbol{O}(\boldsymbol{n}^2)$ down to $\boldsymbol{O}(\boldsymbol{nlogn})$. The polynomial multiplication with NTT is performed as: **c=a*b=INTT(NTT(a)○NTT(b))** where ○ is **cheap pointwise multiplication.**

This work will revisit both **Keccak and polynomial multiplication of Dilithium** for further optimization potential.

1.2 Target Platforms: ARMv7-M

❑ ARM Cortex-M4: Relative high power, resource and memory IoT platform

- NIST's reference 32-bit platform for evaluating PQC in IoT scenarios (a popular **pqm4** repository: https://github.com/mupq/pqm4);
- **1MB flash, 192KB RAM;**
- **14** 32-bit usable general-purpose registers, **32** 32-bit floating-point registers;
- **Inline barrel shifter operation:** e.g., add rd, rn, rm, asr #16, which can merge the addition and shifting operations in 1 instruction.
- SIMD (DSP) extensions: **uadd16, usub16** instructions perform addition and subtraction for two packed 16-bit vectors;
- **1-cycle** multiplication instructions: **smulw{b,t}, smul{b,t}{b,t};**
- Relative expensive **load/store** instructions: **ldr, ldrd, vldm.**

1.2 Target Platforms: ARMv7-M

❑ ARM Cortex-M3: Low resource IoT platform

- **512KB flash, 96KB RAM;**
- **14** 32-bit usable general-purpose registers, **no** floating-point registers;
- **Inline barrel shifter operation**, e.g., **add rd, rn, rm, asr #16,** which can **merge the addition and shifting operations in 1 instruction.**
- Relative expensive **load/store** instructions: ldr, ldrd.
- No **SIMD extensions** and limited multiplication instructions: **mul, mla (1, 2 cycles).**
- **Non-constant time** full multiplication instructions: **umull, smull, umlal** and **small**; So the **constant-time 32-bit modular multiplication** is very expensive on Cortex-M3, which also leads to the **slow 32-bit NTT**.

02

Keccak Optimizations on ARMv7-M

- 2.1 Keccak
- 2.2 Existing Optimizations on ARMv7-M
- 2.3 Keccak Optimizations on ARMv7-M

2.1 Keccak

❑ Keccak permutation

- Keccak-$p[b, n_r]$, where $b = 1600, n_r = 24$ in NIST standards.
- Each state A is represented as an array of 5 × 5 lanes, each lane is $\boldsymbol{w}$ **= 64 bits**. $A[x, y]$ refers to the lane at position (x, y) and $A[x, y, z]$ refers to the z-th bit of the lane.
- Keccak-p is an iterated permutation where each round consists of five consecutive operations $\boldsymbol{\theta, \rho, \pi, \chi}$ **and** $\boldsymbol{\iota}$, where $\boldsymbol{\chi}$ **is the only non-linear operation**.

```
# b refers to the permutation width while nr refers to the number of rounds
keccak-p[b,nr](A):
  A = roundperm(A,RC[i])                                         for i in 0..nr-1
  return A

# r[x,y] refer to rotation offsets while RC refers to the round constant
roundperm(A,RC):
  # theta step
  C[x]   = A[x,0] xor A[x,1] xor A[x,2] xor A[x,3] xor A[x,4]   for x in 0..4
  D[x]   = C[x-1] xor rot(C[x+1],1)                              for x in 0..4
  A[x,y] = A[x,y] xor D[x]                               for (x,y) in (0..4,0..4)
  # rho and pi step
  B[y,2*x+3*y] = rot(A[x,y], r[x,y])                     for (x,y) in (0..4,0..4)
  # chi step
  A[x,y] = B[x,y] xor ((not B[x+1,y]) and B[x+2,y]) for (x,y) in (0..4,0..4)
  # iota step
  A[0,0] = A[0,0] xor RC
  return A
```

Listing 1: Pseudo-code of the Keccak-p cryptographic permutation.

2.2 Existing Optimizations on ARMv7-M

❑ Bit interleaving

- To store 1600-bit Keccak state on 32-bit ARMv7-M, we need **50 32-bit registers**, which is not enough on ARMv7-M and requires expensive **memory accesses** to load the state.
- Bit interleaving technique consists of storing bits **at odd positions in one 32-bit register, and bits at even positions in another**. In this way, **the 64-bit rotations** can be easily handled by **two separate 32-bit rotations**.

❑ In-place processing

- The in-place processing means that it is possible to store all processed data back into the same memory location it was loaded from.
- The Keccak designers proposed a method that will return to its initial memory location after **4 rounds**.

❑ Performance analysis

	XOR	AND/BIC	NOT	Rotations
32-bit platforms	152 XORs	50 ANDs	50 NOTs	58 ROTs
32-bit ARMv7-M	**152 EORs**	**50 BICs**	**-**	**48 RORs**

These instructions theoretically takes $\mathbf{250 \times 24 = 6000}$ **cycles** on ARMv7-M. However, the state-of-the-art Keccak-$p[1600, \cdot\ ]$ from XKCP requires **12969 cycles**, meaning that around **54% of cycles are spent in memory accesses.**

2.3 Keccak Optimizations on ARMv7-M

❑ Pipelining memory access

- The original **xor5** macro (listing 2) from XKCP [CDH+] **suffers memory access pipeline stalls.** We manage to relax **the register pressure and group 5 ldr instructions** together (listing 3), which saves **3 cycles** per macro call.
- We also reordered some other instructions throughout the code. Notably, we moved **str instructions after multiple ldr instructions** as much as possible.

```
.macro xor5   result,b,g,k,m,s
    ldr       \result, [r0, #\b]
    ldr       r1, [r0, #\g]
    eors      \result, \result, r1
    ldr       r1, [r0, #\k]
    eors      \result, \result, r1
    ldr       r1, [r0, #\m]
    eors      \result, \result, r1
    ldr       r1, [r0, #\s]
    eors      \result, \result, r1
.endm
```

Listing 2: Original ARMv7-M assembly code from [BDH+] to compute half a parity lane. Loads from memory are not fully grouped and thus not optimally pipelined on M3 and M4 processors.

```
.macro xor5   result,b,g,k,m,s
    ldr       \result, [r0, #\b]
    ldr       r1,  [r0, #\g]
    ldr       r5,  [r0, #\k]
    ldr       r11, [r0, #\m]
    ldr       r12, [r0, #\s]
    eors      \result, \result, r1
    eors      \result, \result, r5
    eors      \result, \result, r11
    eors      \result, \result, r12
.endm
```

Listing 3: ARMv7-M assembly code after optimization to compute half a parity lane. Loads from memory are now fully grouped and thus optimally pipelined on M3 and M4 processors.

2.3 Keccak Optimizations on ARMv7-M

❑ Lazy rotations

- The original XKCP implementation makes use of **explicit rotations for the ρ step** through **ror** instructions, which requires **47** such instructions per round.
- Recently, Becker and Kannwischer [BK22] proposed that one can omit **these explicit rotations using lazy rotations and defer the explicit rotations until the *θ* step in the next round** (i.e. rotating the second operands using the **inline barrel shifter**) on AArch64.
- Inspired by [BK22], we first utilize the **inline barrel shifter instruction on ARMv7-M** to **merge the xor and ror instructions,** which also helps to reduce some cycles.
- We proposed **two variants of Keccak implementation** considering the code size effect.
  - One has better performance but requiring larger code size: **lazy rotations for all rounds.**
  - One has smaller code size and an acceptable performance: **lazy rotations for three-quarters of the rounds.**

03 Dilithium Optimizations on ARMv7-M

- **3.1 CRYSTAL-Dilithium**
- **3.2 Efficient Multi-moduli NTT for $c\boldsymbol{t}_0$**
- **3.3 Efficient 16-bit for $c\boldsymbol{s}_i$ and $c\boldsymbol{t}_i$**

3.1.1 CRYSTAL-Dilithium

❑ CRYSTAL-Dilithium

- **One out of three DSAs standardized by NIST (FIPS-204).**
- Its hardness is based on MLWE and MSIS problems.
- Parameters: $\boldsymbol{n = 256, q = 8380417 < 2^{23}}, Z_{8380417}[X]\,/(X^{256}+1)$.

Algorithm 2 Dilithium signature generation (sign) [DKL+18]

Input: Secret key sk and message M

Output: $\sigma = (\tilde{c}, \mathbf{z}, \mathbf{h})$

$\mathbf{A} \in R_q^{k\times\ell} := \mathrm{ExpandA}(\rho)$ ▷ $\mathbf{A}$ is generated and stored in NTT representation as $\hat{\mathbf{A}}$

$\mu \in \{0,1\}^{512} := \mathrm{H}(tr\|M)$

$\kappa := 0, (\mathbf{z},\mathbf{h}) := \perp$

$\rho' \in \{0,1\}^{512} := \mathrm{H}(K\|\mu)$ (or $\rho' \leftarrow \{0,1\}^{512}$ for randomized signing)

while $(\mathbf{z},\mathbf{h}) = \perp$ **do** ▷ Pre-compute $\hat{\mathbf{s}}_1 := \mathrm{NTT}(\mathbf{s}_1), \hat{\mathbf{s}}_2 := \mathrm{NTT}(\mathbf{s}_2)$, and $\hat{\mathbf{t}}_0 := \mathrm{NTT}(\mathbf{t}_0)$

  $\mathbf{y} \in \tilde{S}_{\gamma_1}^{\ell} := \mathrm{ExpandMask}(\rho', \kappa)$

  $\mathbf{w} := \mathbf{A}\mathbf{y}$ ▷ $\mathbf{w} := \mathrm{INTT}(\hat{\mathbf{A}} \cdot \mathrm{NTT}(\mathbf{y}))$

  $\mathbf{w}_1 := \mathrm{HighBits}_q(\mathbf{w}, 2\gamma_2)$

  $\tilde{c} \in \{0,1\}^{256} := \mathrm{H}(\mu\|\mathbf{w}_1)$

  $c \in B_\tau := \mathrm{SampleInBall}(\tilde{c})$ ▷ Store c in NTT representation as $\hat{c} = \mathrm{NTT}(c)$

  $\mathbf{z} := \mathbf{y} + c\mathbf{s}_1$ ▷ Compute $c\mathbf{s}_1$ as $\mathrm{INTT}(\hat{c} \cdot \hat{\mathbf{s}}_1)$

  $\mathbf{r}_0 := \mathrm{LowBits}_q(\mathbf{w} - c\mathbf{s}_2, 2\gamma_2)$ ▷ Compute $c\mathbf{s}_2$ as $\mathrm{INTT}(\hat{c} \cdot \hat{\mathbf{s}}_2)$

  if $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$ **or** $\|\mathbf{r}_0\|_\infty \geq \gamma_2 - \beta$, then $(\mathbf{z},\mathbf{h}) := \perp$

  else

    $\mathbf{h} := \mathrm{MakeHint}_q(-c\mathbf{t}_0, \mathbf{w} - c\mathbf{s}_2 + c\mathbf{t}_0, 2\gamma_2)$ ▷ Compute $c\mathbf{t}_0$ as $\mathrm{INTT}(\hat{c} \cdot \hat{\mathbf{t}}_0)$

    if $\|c\mathbf{t}_0\|_\infty \geq \gamma_2$ **or** the # of 1's in $\mathbf{h}$ is greater than ω, then $(\mathbf{z},\mathbf{h}) := \perp$

  $\kappa := \kappa + \ell$

return $\sigma = (\tilde{c}, \mathbf{z}, \mathbf{h})$

3.1.2 Polynomial multiplication of Dilithium

❑ Small polynomial multiplications: cs_i, ct_i

- In Dilithium signature generation and verification, there exists a **small polynomial** c with at most $\boldsymbol{\tau}$ **nonzero coefficients ($\pm$1)** and **the rest of coefficients are 0**.
- The coefficient range of $\boldsymbol{s_i}$ is $[-\eta, \eta]$, then the coefficients of the product $c\boldsymbol{s_i}$ are smaller than $\boldsymbol{\beta = \tau \cdot \eta}$ **(smaller than 16-bit)**.
- The coefficient range of $\boldsymbol{t_i}$ is smaller than 2^{12} or 2^{10}, then the coefficients of the product $c\boldsymbol{t_i}$ are smaller than $\boldsymbol{\beta' = \tau \cdot 2^{12}}$ **or** $\boldsymbol{\beta' = \tau \cdot 2^{10}}$ **(bigger than 16-bit).**
- According to [CHK+21, Section 2.4.6], these kinds of polynomial multiplications can be treated as multiplications over $Z_{q'}[X]/(X^n + 1)$ with a large prime modulus $q' > 2\beta$ or $q' > 2\beta'$. In sum, we can use **16-bit NTT for $c\boldsymbol{s_i}$ and 32-bit NTT for $c\boldsymbol{t_i}$.**

Table 1: Dilithium parameters [DKL+18]

NIST security level	2	3	5
q [modulus]	8380417	8380417	8380417
n [the order of polynomial]	256	256	256
d [drop bits from $\mathbf{t}$]	13	13	13
τ [# of $\pm$1's in c]	39	49	60
γ_1 [$\mathbf{y}$ coefficient range]	2^{17}	2^{19}	2^{19}
γ_2 [low-order rounding range]	$(q-1)/88$	$(q-1)/32$	$(q-1)/32$
(k, l) [dimensions of $\mathbf{A}$]	(4,4)	(6,5)	(8,7)
η [secret key range]	2	4	2
$\beta = \tau \cdot \eta$ [$c\mathbf{s}_i$ coefficient range]	78	196	120
$\mathbf{t}_0$ coefficient range	2^{12}	2^{12}	2^{12}
$\mathbf{t}_1$ coefficient range	2^{10}	2^{10}	2^{10}

3.1.3 16-bit NTT vs 32-bit NTT on Cortex-M3

❑ 16-bit NTT vs 32-bit NTT on Cortex-M3

- **Cortex-M3** does not have **constant-time full multiplication,** which may lead to insecure 32-bit modular multiplication implementation (side-channel attack).
- The constant-time 32-bit modular multiplication in [GKS20] takes **6-8 instructions**.
- The constant-time 32-bit CT butterfly takes in [GKS20] **19 instructions, compared to 5 instructions for 16-bit CT butterfly;**
- **The 16-bit NTT with Plantard arithmetic in [HZZ+23] is at least 2~3 × faster than 32-bit NTT in [GKS20] on Cortex-M3.**

Listing 5 Schoolbook SMULL (SBSMULL)

```
; Input:  a = a0 + a1*2^16
;         b = b0 + b1*2^16
; Output: c = a*b = c0 + c1*2^32
mul  c0, a0, b0
mul  c1, a1, b1
mul  tmp, a1, b0
mla  tmp, a0, b1, tmp
adds c0, c0, tmp, lsl #16
adc  c1, c1, tmp, asr #16
```

Listing 6 Schoolbook SMLAL (SBSMLAL)

```
; Input:  a = a0 + a1*2^16
;         b = b0 + b1*2^16
;         c = c0 + c1*2^32
; Output: c = c + a*b
;           = c0 + c1*2^32
mul  tmp, a0, b0
adds c0, c0, tmp
mul  tmp, a1, b1
adc  c1, c1, tmp
mul  tmp, a1, b0
mla  tmp, a0, b1, tmp
adds c0, c0, tmp, lsl #16
adc  c1, c1, tmp, asr #16
```

Constant-time 32-bit multiplication implementation on Cortex-M3 [GKS20]

3.2 The Proposed cs_i, ct_i Implementations

❑ NTT over 769 for cs_i

- The coefficient range of s_i is $[-\eta, \eta]$, then the coefficients of the product cs_i are smaller than $\beta = \tau \cdot \eta$ **=78, 196 and 120 for three security levels.** [AHKS22] used **FNT over 257** for Dilithium2 and Dilithium5, and used **NTT over 769** for Dilithium3.
- **On Cortex-M4:** We reuse **FNT over 257** for Dilithium2 and Dilithium5, and optimize **NTT over 769 with Plantard arithmetic.**
- **On Cortex-M3:** We reuse **NTT over 769 with Plantard arithmetic** for all Dilithium variants, because we can then combine it with multi-moduli NTT.

❑ Multi-moduli NTT for ct_i

- The coefficient range of t_i is 2^{12} or 2^{10}, then the coefficients of the product ct_i are smaller than $\beta' = \tau \cdot 2^{12} = 245760, q' > 2\beta' = 491520$. We choose a composite modulus $q' = 769 \times 3329 = 2560001$ and perform multiplications over $Z_{q'}[X]/(X^n + 1)$.
- **On Cortex-M4:** The 16-bit NTT and 32-bit NTT has not much differences. So we cannot use multi-moduli NTT for ct_i on Cortex-M4.
- **On Cortex-M3:** We optimize ct_i with the **multi-moduli NTT over the $q' = 769 \times 3329$** for all three Dilithium variants and **separately optimize the 16-bit NTT over 769 and 3329 with Plantard arithmetic.**

3.2.1 Efficient Multi-moduli NTT for ct_i

- **Multi-moduli NTTs for ct_i on Cortex-M3**

$$\mathbb{Z}_{q_0 q_1} \cong \mathbb{Z}_{q_0} \times \mathbb{Z}_{q_1};$$

$$\mathbb{Z}_{q_0}[X]/(X^{256}+1) \cong \mathbb{Z}_{q_0}[X]/(X^2 - \zeta_0^j), j = 1, 3, 5, \ldots, 255;$$

$$\mathbb{Z}_{q_1}[X]/(X^{256}+1) \cong \mathbb{Z}_{q_1}[X]/(X^2 - \zeta_1^j), j = 1, 3, 5, \ldots, 255;$$

3.2.1 Efficient Multi-moduli NTT for ct_i

❑ Multi-moduli NTTs for ct_i on Cortex-M3

Algorithm 4 Multi-moduli NTT for computing 32-bit NTT on Cortex-M3

Input: Declare arrays: `int32_t c_32[256],t_32[256],tmp_32[256],res_32[256]`

Input: Declare pointers:

```
int16_t *cl_16=(int16_t*)c_32;
int16_t *ch_16=(int16_t*)&c_32[128];
int16_t *tl_16=(int16_t*)t_32;
int16_t *th_16=(int16_t*)&t_32[128];
int16_t *tmpl_16=(int16_t*)tmp_32;
int16_t *tmph_16=(int16_t*)&tmp_32[128];
```

`cl_16[256]` $\leftarrow c$, `ch_16[256]` $\leftarrow c$ ▷ Pre-store c in the bottom and top halves of `c_32` as 16-bit arrays

`tl_16[256]` $\leftarrow t$, `th_16[256]` $\leftarrow t$ ▷ Pre-store t in the bottom and top halves of `t_32` as 16-bit arrays

`cl_16[256]` $= \text{NTT}_{q_0}$(`cl_16`) ▷ $\hat{c}_0 = \text{NTT}_{q_0}(c)$

`ch_16[256]` $= \text{NTT}_{q_1}$(`ch_16`) ▷ $\hat{c}_1 = \text{NTT}_{q_1}(c)$

`tl_16[256]` $= \text{NTT}_{q_0}$(`tl_16`) ▷ $\hat{t}_0 = \text{NTT}_{q_0}(t)$

`th_16[256]` $= \text{NTT}_{q_1}$(`th_16`) ▷ $\hat{t}_1 = \text{NTT}_{q_1}(t)$

`tmpl_16[256]` $= \text{basemul}_{q_0}$(`cl_16, tl_16`) ▷ $\hat{c}_0 \cdot \hat{t}_0 = \text{basemul}_{q_0}(\hat{c}_0, \hat{t}_0)$

`tmph_16[256]` $= \text{basemul}_{q_1}$(`ch_16, th_16`) ▷ $\hat{c}_1 \cdot \hat{t}_1 = \text{basemul}_{q_1}(\hat{c}_1, \hat{t}_1)$

`tmpl_16[256]` $= \text{INTT}_{q_0}$(`tmpl_16`) ▷ $\text{INTT}_{q_0}(\hat{c}_0 \cdot \hat{t}_0)$

`tmph_16[256]` $= \text{INTT}_{q_1}$(`tmph_16`) ▷ $\text{INTT}_{q_1}(\hat{c}_1 \cdot \hat{t}_1)$

`res_32[256]` $= \text{CRT}$(`tmpl_16, tmph_16`) ▷ $\text{CRT}(\text{INTT}_{q_0}(\hat{c}_0 \cdot \hat{t}_0), \text{INTT}_{q_1}(\hat{c}_1 \cdot \hat{t}_1))$

return `res_32`

3.2.2 Efficient 16-bit NTT for cs_i and ct_i

❑ Efficient 16-bit NTT with Plantard arithmetic on Cortex-M3 [HZZ+23]

- ➢ The 16×32-bit multiplication is implemented with **mul** instruction, and the effective result lies in the **higher 16-bit of r.** We can merge the **addition and shiftting operation** using the inline barrel shifter operation as in Step 3 of Algorithm 4.
- ➢ The Plantard implementation is **1-multiplication faster than the Montgomery's.**
- ➢ **No modular reduction in INTT over 769 and 3329 at all.**

Algorithm 3 Plantard multiplication with enlarged input range

Input: Two signed integers a, b such that $ab \in [q2^l - q2^{l+\alpha}, 2^{2l} - q2^{l+\alpha}), q < 2^{l-\alpha-1}, q' = q^{-1} \bmod^{\pm} 2^{2l}$

Output: $r = ab(-2^{-2l}) \bmod^{\pm} q$ where $r \in [-\frac{q+1}{2}, \frac{q}{2})$

1: $r = \left[\left([[abq']_{2l}]^l + 2^\alpha\right) q\right]^l$

2: **return** r

Algorithm 5 Efficient Plantard multiplication by a constant for 16-bit modulus q_i on Cortex-M3 [HZZ+23]

Input: Two signed integers a, b such that $a \in (q_i 2^{16} - q_i 2^{16+\alpha_i}, 2^{32} - q_i 2^{16+\alpha_i})$, a precomputed 32-bit integer bq'_i where b is a constant and $q'_i = q_i^{-1} \bmod^{\pm} 2^{32}$

Output: $r = ab(-2^{-32}) \bmod^{\pm} q_i$

1: $bq'_i \leftarrow bq_i^{-1} \bmod 2^{32}$ ▷ precomputed

2: **mul** r, a, bq'_i

3: **add** $r, 2^{\alpha_i}, r$, asr#16

4: **mul** r, r, q_i

5: **asr** $r, r, \#16$

6: **return** r

3.2.2 Efficient 16-bit NTT for cs_i and ct_i

❑ The explicit CRT implementation with Plantard arithmetic

- The constant $\boldsymbol{m_1 = q_0^{-1} \, mod^{\pm} \, q_1}$ in CRT computation can be precomputed as ($m_1' = m_1 \cdot (-2^{32} \, mod \, q_1) \cdot (q_1^{-1} \, mod \, 2^{32}) \, mod \, 2^{32}$) and **speeded up with the efficient Plantard multiplication by a constant.**
- The implementation is **1-multiplication faster than the Montgomery's.**

Algorithm 6 The explicit CRT with Plantard arithmetic on Cortex-M3

Input: $u_0 = u \bmod q_0, u_1 = u \bmod q_1, m_1 = q_0^{-1} \bmod^{\pm} q_1, m_1' = m_1 \cdot (-2^{32} \bmod q_1) \cdot (q_1^{-1} \bmod 2^{32}) \bmod 2^{32}, q_1 2^{\alpha_1} < 2^{15}$

Output: $u = u_0 + ((u_1 - u_0)m_1 \bmod^{\pm} q_1)q_0$

1: **sub** t, u_1, u_0
2: **mul** t, t, m_1'
3: **add** $t, 2^{\alpha_1}, t, \text{asr}\#16$
4: **mul** t, t, q_1
5: **asr** $t, t, \#16$ ▷ $t \leftarrow (u_1 - u_0)m_1 \bmod^{\pm} q_1$
6: **mla** u, t, q_0, u_0 ▷ $u \leftarrow u_0 + tq_0$
7: **return** u

---

04 Results and Conclusions

- 4.1 Results and Comparisons
- 4.2 Conclusions
- 4.3 References

4.1 Results and Comparisons

❑ Keccak results

- **Setup: Cortex-M3: ATSAM3X8E; Cortex-M4: STM32F407VG.**
- The pipelining memory access optimization results in **17.13% and 12.84%** speedups on Cortex-M3 and M4, respectively.
- When combined with the **lazy rotation** technique, we achieve up to **24.78% and 21.4%** performance boosts on Cortex-M3 and M4, respectively.

Table 2: Keccak-p[1600, 24] benchmark on Cortex-M3 and M4.

Ref.	Implementation characteristics*		Speed (clock cycles)		Code size	RAM
	ldr/str	lazy ror	M3	M4	(bytes)	(bytes)
XKCP	mostly grouped	✗	13 015	11 725	5 576	**264**
This work	grouped	✗	10 785	10 219	5 772	**264**
	grouped	✓ (3/4)	9 981	9 415	6 556	**264**
	grouped	✓ (4/4)	**9 789**	**9 218**	9 536	**264**

*All listed implementations take advantage of the in-place processing and bit-interleaving techniques.

4.1 Results and Comparisons

❑ NTT results on Cortex-M3

- Using the Plantard arithmetic, the **16-bit NTT, INTT, and pointwise multiplication** on Cortex-M3 are **4.22×, 4.29×, and 2.14× faster** than the constant-time 32-bit NTT, INTT, and pointwise multiplication in [GKS20], respectively. Compared to the 32-bit variable-time NTT, INTT, and pointwise multiplication, the speed ups are **2.48×, 2.46×, and 1.24×,** respectively.
- The **proposed multi-moduli NTT, INTT and pointwise multiplication** implementations yield **52.76% ~ 54.76%** performance improvements compared to the constant-time 32-bit NTT in [GKS20]. And over **19.47% and 19.07% speed-ups** compared with the variable-time 32-bit NTT and INTT in [GKS20].

Platform	Prime	Ref.	NTT	INTT	Pointwise	CRT
M3	8380417	[GKS20] constant-time	33 077	36 661	8 528	✗
	8380417	[GKS20] variable-time	19 405	21 051	4 944	✗
	3329 × 7681	[ACC+22]	16 770	19 056	11 927	4 637
	769	This work	7 830	8 543	3 989	✗
	769 × 3329	This work	15 626	17 037	8 061	3 735

4.1 Results and Comparisons

❑ Dilithium results on Cortex-M3

Platform	Operation	Dilithium2		Dilithium3		Dilithium5	
		[GKS20]	This work	[GKS20]	This work	[GKS20]	This work
M3	cs_1	$346k$	$106k$	$424k$	$128k$	$580k$	$172k$
	cs_2	$346k$	$106k$	$502k$	$150k$	$658k$	$194k$
	ct_0	$269k$	$195k$	$328k$	$284k$	$446k$	$372k$
	ct_1	$213k$	$195k$	$311k$	$284k$	$409k$	$372k$

Table 5: Performance of Dilithium on Cortex-M3. Averaged over 1000 executions.

Operation	Dilithium2		Dilithium3		Dilithium5	
	[GKS20]	This work	[GKS20]	This work	[GKS20]	This work
keygen	$2\,059k$	$1\,739k$	$3\,594k$	$3\,011k$	✗	$5\,034k$
sign	$7\,139k$	$5\,582k$	$11\,916k$	$9\,087k$	✗	$20\,193k$
verify	$1\,949k$	$1\,648k$	$3\,283k$	$2\,755k$	✗	$4\,694k$

4.1 Results and Comparisons

❑ Kyber and Dilithium hash profiling on Cortex-M4

Table 6: Performance and hash profiling of Kyber and Dilithium on the Cortex-M4 using the pqm4 framework. Averaged over 1000 executions.

Scheme	Keccak Impl.	keygen		sign/encaps		verify/decaps	
		speed	hashing	speed	hashing	speed	hashing
Dilithium2	XKCP	1 595k	83.47%	4 052k	64.53%	1 576k	80.47%
	This work	1 357k	80.57%	3 487k	60.02%	1 350k	77.2%
Dilithium3	XKCP	2 828k	85.54%	6 523k	62.95%	2 702k	82.62%
	This work	2 394k	82.92%	5 574k	58.97%	2 302k	79.61%
Dilithium5	XKCP	4 817k	86.6%	8 534k	68.08%	4 714k	84.69%
	This work	4 069k	84.14%	7 730k	63.05%	3 998k	81.95%
Kyber512	XKCP	432k	80.12%	527k	82.86%	472k	73.76%
	This work	369k	76.75%	448k	79.85%	409k	69.74%
Kyber768	XKCP	704k	79.04%	860k	82.38%	778k	74.75%
	This work	604k	75.59%	732k	79.32%	674k	70.84%
Kyber1024	XKCP	1 122k	79.58%	1 314k	82.46%	1 208k	76.07%
	This work	962k	76.18%	1 119k	79.41%	1 043k	72.29%

4.2 Conclusions

❑ Optimized Keccak and Dilithium on ARMv7-M

- We significantly improved Keccak's efficiency using two optimized techniques on ARMv7-M.
- We explored efficient multi-moduli NTT and small NTT implementation with Plantard arithmetic for Dilithium on Coretx-M3.
- Open-source (https://github.com/UIC-ESLAS/Dilithium-Multi-Moduli) and merge into pqm4 (PR#254 and PR#338).

4.3 References

[BDH+] Guido Bertoni, Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche,and Ronny Van Keer. XKCP: eXtended Keccak Code Package.https://github.com/XKCP/XKCP. commit 7fa59c0.
[BK22] Hanno Becker and Matthias J. Kannwischer. Hybrid Scalar/Vector Im-plementations of Keccak and SPHINCS+ on AArch64. In Takanori Isobeand Santanu Sarkar, editors,Progress in Cryptology – INDOCRYPT 2022,pages 272–293, Cham, 2022. Springer International Publishing.https://eprint.iacr.org/2022/1243.
[CHK+21] Chi-Ming Marvin Chung, Vincent Hwang, Matthias J. Kannwischer, GregorSeiler, Cheng-Jhih Shih, and Bo-Yin Yang. NTT Multiplication for NTT-unfriendly Rings New Speed Records for Saber and NTRU on Cortex-M4 andAVX2.IACR Trans. Cryptogr. Hardw. Embed. Syst., 2021(2):159–188, 2021.
[HZZ+23] Junhao Huang, Haosong Zhao, Jipeng Zhang, Wangchen Dai, Lu Zhou, RayC. C. Cheung, Çetin Kaya Koç, and Donglong Chen. Yet another Improvementof Plantard Arithmetic for Faster Kyber on Low-end 32-bit IoT Devices. IEEE Transactions on Information Forensics and Security, 2024.
[AHKS22] Amin Abdulrahman, Vincent Hwang, Matthias J. Kannwischer, and AmberSprenkels. Faster Kyber and Dilithium on the Cortex-M4. In Giuseppe Ateniese and Daniele Venturi, editors,Applied Cryptography and Network Security -20th International Conference, ACNS 2022, Rome, Italy, June 20-23, 2022,Proceedings, volume 13269 ofLecture Notes in Computer Science, pages 853–871.Springer, 2022.
[GKS20]Denisa O. C. Greconici, Matthias J. Kannwischer, and Amber Sprenkels.Compact Dilithium Implementations on Cortex-M3 and Cortex-M4.IACRTrans. Cryptogr. Hardw. Embed. Syst., 2021(1):1–24, Dec. 2020

Thanks for listening!