Optimized Software Implementations of CRYSTALS-Kyber, NTRU, and Saber Using NEON-Based Special Instructions of ARMv8

Duc Tri Nguyen and Kris Gaj

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

Abstract. This paper focuses on optimized constant-time software implementations of three NIST PQC KEM Finalists, CRYSTALS-Kyber, NTRU, and Saber, targeting ARMv8 microprocessor cores. All optimized implementations include explicit calls to Advanced Single-Instruction Multiple-Data instructions (a.k.a. NEON instructions). Benchmarking is performed using two platforms: 1) MacBook Air, based on an Apple M1 System on Chip (SoC), including four high-performance 'Firestorm' ARMv8 cores, running with the frequency of around 3.2 GHz, and 2) Raspberry Pi 4, single-board computer, based on the Broadcom SoC, BCM2711, with four 1.5 GHz 64-bit Cortex-A72 ARMv8 cores. In each case, only one core of the respective SoC is being used for benchmarking. The obtained results demonstrate substantial speed-ups vs. the best available implementations written in pure C. For the 'Firestorm' core of Apple M1, NEON implementations outperform pure C implementations in the case of decapsulation by factors varying in the following ranges: 1.55-1.74 for Saber, 2.96-3.04 for Kyber, and 7.24-8.49 for NTRU, depending on an algorithm's variant and security level. For encapsulation, the corresponding ranges are 1.37-1.60 for Saber, 2.33-2.45 for Kyber, and 3.05-6.68 for NTRU. These uneven speed-ups of the three lattice-based KEM finalists affect their rankings for optimized software implementations targeting ARMv8.

Keywords: ARMv8 · NEON · Karatsuba · Toom-Cook · Number Theoretic Transform · NTRU · Saber · Kyber · Latice · Post-Quantum Cryptography

1 Introduction

In July 2020, NIST announced the Round 3 finalists of the Post-Quantum Cryptography Standardization process. The main selection criteria were security, key/ciphertext sizes, and performance in software. CRYSTALS-Kyber, NTRU, and Saber are three lattice-based finalists in the category of encryption/Key Encapsulation Mechanism (KEM).

There exist constant-time software implementations of all these algorithms on various platforms, including Cortex-M4 (representing the ARMv7E-M instruction set architecture), RISC-V, as well as various Intel and AMD processor cores. However, there is still a lack of optimized software implementations for the ARMv8 architecture.

The popularity of ARM is undeniable, with billions of devices connected to the Internet[1]. As a result, there is clearly a need to maintain secure communication among these devices in the age of quantum computers. Without high-speed implementations, the deployment and adoption of emerging PQC standards may be slowed down.

One of the most recent and popular versions of ARM is ARMv8 (a.k.a. ARMv8-A). ARMv8 supports two instruction set architectures AArch64 (a.k.a. arm64) and

[1] https://www.tomshardware.com/news/arm-6-7-billion-chips-per-quarter

AArch32 (a.k.a. armeabi). The associated instruction sets are referred to as A64 and A32, respectively. AArch32 and A32 are compatible with an older version of ARM called ARMv7-A. AArch64 and A64 support operations on 64-bit operands. The first core compatible with ARMv8 and used in a consumer product was Apple A7, included in iPhones 5S in 2013.

NEON is an alternative name for Advanced Single Instruction Multiple Data (ASIMD) extension, mandatory 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 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 today's market, there is a wide range of ARMv8 processor supporting NEON. They are developed by manufacturers such as Apple Inc., ARM Holdings, Cavium, Fujitsu, Nvidia, Marvell, Qualcomm, and Samsung. They power the majority of smartphones and tablets available on the market.

Apple M1 is an ARMv8-based system on a chip (SoC) designed by Apple Inc. for multiple Apple devices, such as MacBook Air, MacBook Pro, Mac Mini, iMac, and iPad Pro. The M1 has four high-performance 'Firestorm' and four energy-efficient 'Icestorm' ARMv8 cores. Each of them supports NEON. It also includes an Apple-designed eight-core graphics processing unit (GPU) and a 16-core Neural Engine.

The ARM Cortex-A72 is a core implementing the ARMv8-A 64-bit instruction set. It was designed by ARM Holdings' Austin design center. This core is used in Raspberry Pi 4 - a small single-board computer developed in the United Kingdom by the Raspberry Pi Foundation in association with Broadcom. The corresponding SoC is called BCM2711. This SoC contains four Cortex-A72 cores.

In this work, we benchmark the performance of three Round 3 PQC finalists 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).

In the case of benchmarking using Intel and AMD processors, it is commonly agreed that only the results for optimized implementations should be taken into account. For cryptographic algorithms, an optimized implementation targeting these processors almost always takes advantage of the Advanced Vector Extensions 2 (a.k.a. Haswell New Instructions). AVX2 was first supported by Intel with the Haswell processor, which shipped in 2013. Most of the candidates in the NIST PQC standardization process contain such optimized implementations as a part of their submission packages. In particular, CRYSTALS-Kyber, NTRU, and Saber contain AVX2 implementations.

The same is not true for implementations targeting ARMv8 cores. Any benchmarking results available before our work have been based on compiling implementations written in pure C. Thus, the NEON instructions were not called explicitly by the programmers, making these implementations quite far from optimal.

Our goal is to fill the gap between proper benchmarking on low-power embedded processors, such as Cortex-M4 and power-hungry x86-64 platforms (a.k.a. AMD64 platforms). To do that, we have developed constant-time ARMv8 implementations of three lattice-based KEM finalists: CRYSTALS-Kyber, NTRU, and Saber.

We have assumed the use of parameter sets supported by all candidates at the beginning of Round 3. The differences among the implemented algorithms in terms of security, decryption failure rate, and resistance to side-channel attacks are out of scope for this paper.

Contributions. Our work includes the first ARMv8 implementations of three lattice-based PQC KEM finalists optimized using NEON instructions. Our source code is publicly available at: https://github.com/GMUCERG/PQC_NEON

2 Previous Work

Initial work that applied NEON to cryptography was published in 2012 by Bernstein et al. [1]. Subsequent work by Azarderakhsh et al. [2] showed tremendous speed-up in case of RSA and ECC public key encryption.

The paper by Streit et al. [3] was the first work about the NEON-based ARMv8 implementation of New Hope Simple. This work, published prior to the PQC Competition, proposed a "merged NTT levels" structure.

Scott [4] and Westerbaan [5] proposed lazy reduction as a part of their NTT implementation. Seiler [6] proposed an FFT trick, which has been widely adopted in the following work. Lyubashevsky et al. [7] and Zhou et al. [8] proposed fast implementations of NTT, and applied them to NTTRU and Kyber, respectively.

In the area of low-power implementations, most previous work targeted Cortex-M4 [9]. In particular, Botros et al. [10] and Alkim et al. [11] developed ARM Cortex-M4 implementations of Kyber. Karmakar et al. [12] reported results for Saber. Chung et al. [13] proposed an NTT-based implementation for an NTT-unfriendly ring, targeting Cortex-M4 and AVX2. We adapted this method to our NTT-based implementation of Saber.

From the high-performance perspective, Gupta et al. [14] proposed the GPU implementation of Kyber; Roy et al. [15] developed $4 \times$ *Saber* by utilizing 256-bit AVX2 vectors. Danba et al. [16] developed a high-speed implementation of NTRU using AVX2. Finally, Hoang et al. [17] implemented the fast NTT-function using ARMv8 Scalable Vector Extension (SVE).

3 Background

In this section, we introduce cryptographic algorithms implemented in this paper: NTRU (Section 3.1), Saber (Section 3.2), and Kyber (Section 3.3), followed by the Polynomial Multiplication (Section 3.4), used in NTRU and Saber, and Kyber.

	`Polynomial`	n			q $[,p]$		
		1	3	5	1	3	5
Kyber	$x^n + 1$	256			3329		
Saber	$x^n + 1$	256			$2^{13}, 2^{10}$		
NTRU-HPS	$\Phi_1 = x - 1$	677	821	–	2^{11}	2^{12}	–
NTRU-HRSS	$\Phi_n = \frac{x^n-1}{x-1}$	701	-		2^{13}	-	

Table 1: Parameters of Kyber, Saber, and NTRU

	$\|pk\|$ (B)			$\|ct\| - \|pk\|$ (B)		
	1	3	5	1	3	5
Saber	672	992	1312	$\uparrow$ 64	$\uparrow$ 96	$\uparrow$ 160
Kyber	800	1184	1568	$\downarrow$ 32	$\downarrow$ 96	0
NTRU-HPS	931	1230	–	0	0	–
NTRU-HRSS	1138	-				

Table 2: Public key and Ciphertext size of Kyber, Saber, and NTRU

NTRU, Saber, and Kyber use variants of the Fujisaki-Okamoto (FO) transform [18] to define the Chosen Ciphertext Attack (CCA)-secure KEMs based on the underlying public-key encryption (PKE) schemes. Therefore, speeding up the implementation of PKE also significantly speeds up the implementation of the entire KEM scheme.

The parameters of Kyber, Saber, and NTRU are summarized in Table 1. Saber uses two power of two moduli, p and q, across all security levels. NTRU has different moduli for each security level. NTRU-HRSS701 shares similar attributes with NTRU-HPS and has parameters listed in the second line for security level 1.

The symbols $\uparrow$ and $\downarrow$ indicate the increase or decrease of the CCA-secure KEM ciphertext ($|ct|$) size, as compared with the public-key size ($|pk|$) (both in bytes (B)).

3.1 NTRU

```
Algorithm 1: NTRU CPA Encryption
   Input: $m \in R_2, pk = (\mathbf{h}), r$
   Output: $c$
 1 $m' \leftarrow \texttt{Lift}(m)$
 2 $c \leftarrow (r \cdot \mathbf{h} + m') \bmod (q, \Phi_1\Phi_n)$
```

```
Algorithm 2: NTRU CPA Decryption
   Input: $c, sk = (\mathbf{f}, \mathbf{f}_p, \mathbf{h}_q)$
   Output: $r, m$ or fail
 1 if $c \neq 0$ mod $(q, \Phi_1)$ then
 2     return fail
 3 $a \leftarrow (c \cdot \mathbf{f}) \bmod (q, \Phi_1\Phi_n)$
 4 $m \leftarrow (a \cdot \mathbf{f}_p) \bmod (3, \Phi_n)$
 5 $m' \leftarrow \texttt{Lift}(m)$
 6 $r \leftarrow ((c - m') \cdot \mathbf{h}_q) \bmod (q, \Phi_n)$
```

The Round 3 submission of NTRU [19] is a merger of the specifications for NTRU-HPS and NTRU-HRSS. The detailed algorithms for encryption and decryption are shown in Algorithms 1 and 2. The NTRU KEM uses polynomial $\Phi_1 = x - 1$ for *implicit rejection.* It rejects an invalid ciphertext and returns a pseudorandom key, avoiding the need for re-encryption, which is required in Saber and Kyber.

The advantage of NTRU is fast Encapsulation (only 1 multiplication) but the downside is the use of time-consuming inversions in key generation.

3.2 Saber

Algorithm 3: Saber CPA Encryption

```
  Input: $m \in R_2, pk = (seed_{\mathbf{A}}, \mathbf{b}), r$
  Output: $ct = (c_m, \mathbf{b}')$
1 $\mathbf{A} \in R_q^{l \times l} \leftarrow \texttt{GenMatrix}(seed_{\mathbf{A}})$
2 $\mathbf{s}' \in R_q^{l} \leftarrow \texttt{Sample}_B(r)$
3 $\mathbf{b}' \in R_p^{l} \leftarrow ((\mathbf{A} \circ \mathbf{s}' + \mathbf{h}) \bmod \text{q}\ ) \gg (\epsilon_q - \epsilon_p)$
4 $v' \in R_p \leftarrow \mathbf{b}^T \cdot (\mathbf{s}' \bmod \text{p})$
5 $c_m \in R_T \leftarrow (v' + h_1 - 2^{\epsilon_p - 1} m \bmod \text{p}) \gg (\epsilon_p - \epsilon_T)$
```

Algorithm 4: Saber CPA Decryption

```
  Input: $ct = (c_m, \mathbf{b}'), sk = (\mathbf{s})$
  Output: $m'$
1 $v \in R_p \leftarrow \mathbf{b}'^T \cdot (\mathbf{s} \bmod \text{p})$
2 $m' \in R_2 \leftarrow ((v - 2^{\epsilon_p - \epsilon_T} c_m + h_2) \bmod \text{p}) \gg (\epsilon_p - 1)$
```

Saber [19] relies on the hardness of the Module Learning With Rounding problem (M-LWR). Similarly to NTRU, the Saber parameter p is a power of two. This feature supports inexpensive reduction mod p. However, such parameter p prevents the best time complexity multiplication algorithm $O(n \log n)$ to be applied *directly*. The same parameter n is used across three security levels. Saber encryption and decryption are shown in Algorithms 3 and 4. The detailed parameters of Saber are summarized in Table 1. Among the three investigated algorithms, Saber has the smallest public keys and ciphertext sizes, $|pk|$ and $|ct|$, as shown in Table 2.

$\texttt{Sample}_B$ are samples from the binomial distribution. `GenMatrix` constructs matrix $\mathbf{A}$ of the dimensions $l \times l$ from the 32-byte $seed_{\mathbf{A}}$, where $l \in \{2, 3, 4\}$ for security levels 1, 3, and 5 respectively. This is a common technique in M-LWR and M-LWE to save bandwidth. The algorithms for `GenMatrix` and $\texttt{Sample}_B$ can be found in the Saber specification [19].

3.3 Kyber

Algorithm 5: Kyber CPA Encryption

```
  Input: $m \in R_2, pk = (seed_{\mathbf{A}}, \hat{\mathbf{t}}), r$
  Output: $c = (\mathbf{u}, v)$
1 nonce $\leftarrow 0$;
2 $\hat{\mathbf{A}}^T \in R_q^{k \times k} \leftarrow \texttt{GenMatrix}(seed_{\mathbf{A}}, i, j)$ for $i, j \in [0, \dots k-1]$
3 $\mathbf{r} \in R_q^{k} \leftarrow \texttt{Sample}_B(r, \texttt{nonce++})$ for $i \in [0, \dots k-1]$
4 $\mathbf{e}_1 \in R_q^{k} \leftarrow \texttt{Sample}_B(r, \texttt{nonce++})$ for $i \in [0, \dots k-1]$
5 $e_2 \in R_q \leftarrow \texttt{Sample}_B(r, \texttt{nonce})$
6 $\hat{\mathbf{r}} \leftarrow \mathcal{NTT}(\mathbf{r})$
7 $\mathbf{u} \leftarrow \mathcal{NTT}^{-1}(\hat{\mathbf{A}}^T \circ \hat{\mathbf{r}}) + \mathbf{e}_1$
8 $v \leftarrow \mathcal{NTT}^{-1}(\hat{\mathbf{t}}^T * \hat{\mathbf{r}}) + e_2 + m$
```

Algorithm 6: Kyber CPA Decryption

```
  Input: $sk = (\hat{\mathbf{s}}), c = (\mathbf{u}, v)$
  Output: $m$
1 $m \leftarrow v - \mathcal{NTT}^{-1}(\hat{\mathbf{s}}^T * \mathcal{NTT}(\mathbf{u}))$
```

The security of Kyber is based on the hardness of the learning with errors problem in module lattices, so-called M-LWE. Similar to Saber and NTRU, the KEM construction is based on CPA public-key encryption scheme with a slightly tweaked FO transform [18]. Improving performance of public-key encryption helps speed up KEM as well. Kyber public and private keys are assumed to be already in NTT domain. This feature clearly differentiates Kyber from Saber and NTRU. The multiplication in the NTT domain has the best time complexity $O(n \log n)$.

The algorithms for encryption and decryption are shown in Algorithms 5 and 6. The `GenMatrix` operation generates the matrix $\hat{\mathbf{A}}$ of the dimensions $k \times k$, where $k \in \{2, 3, 4\}$. Additionally, `GenMatrix` enables parallel SHA-3 sampling for multiple rows and columns, with the parameters $seed_{\mathbf{A}}$, i, and j. $\texttt{Sample}_B$ are samples from a binomial distribution, similar to `GenMatrix`. Unlike Saber, $\texttt{Sample}_B$ and `GenMatrix` in Kyber enable parallel computation. More details can be found in the Kyber specification [19].

3.4 Polynomial Multiplication

In this section, we introduce polynomial multiplication algorithms, arranged from the worst to the best in terms of time complexity. The goal is to compute the product of two polynomials in Equation 1 as fast as possible.

$$C(x) = A(x) \times B(x) = \sum_{i=0}^{n-1} a_i x^i \times \sum_{i=0}^{n-1} b_i x^i \tag{1}$$

$$\begin{tikzcd}
Schoolbook \arrow[r] & Toom - Cook \arrow[r] & NTT \\
O(n^2) \arrow[u] & O(n^{\frac{\log(2k-1)}{\log k}}) \arrow[u] & O(n \log n) \arrow[u]
\end{tikzcd}$$

3.4.1 Schoolbook Multiplication

Schoolbook is the simplest form of multiplication. The algorithm consists of two loops with the $O(n)$ space and $O(n^2)$ time complexity, as shown in Equation 2.

$$C(x) = \sum_{k=0}^{2n-2} c_k x^k = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} a_i b_j x^{(i+j)} \tag{2}$$

3.4.2 Toom-Cook and Karatsuba

Toom-Cook and Karatsuba are multiplication algorithms that differ greatly in terms of computational cost versus the most straightforward schoolbook method when the degree n is large. Karatsuba [20] is a special case of Toom-Cook [21, 22] (Toom-k). Generally, both algorithms consist of five steps: splitting, evaluation, point-wise multiplication, interpolation, and recomposition. An overview of polynomial multiplication using Toom-k is shown in Algorithm 7. Splitting and recomposition are often merged into evaluation and interpolation, respectively.

Examples of these steps in Toom-4 are shown in Equations 3, 4, 5, and 6, respectively. In the splitting step, Toom-k splits the polynomial A(x) of the degree $n-1$ (containing n coefficients) into k polynomials with the degree $n/k - 1$ and n/k coefficients each. These polynomials become coefficients of another polynomial denoted as $\mathcal{A}(\mathcal{X})$. Then, $\mathcal{A}(\mathcal{X})$ is evaluated for $2k-1$ different values of $\mathcal{X} = x^{n/k}$. Below, we split $A(x)$ and evaluate $\mathcal{A}(\mathcal{X})$ as an example.

$$\begin{bmatrix}\mathcal{A}(0)\\ \mathcal{A}(1)\\ \mathcal{A}(\text{-}1)\\ \mathcal{A}(\frac{1}{2})\\ \mathcal{A}(\text{-}\frac{1}{2})\\ \mathcal{A}(2)\\ \mathcal{A}(\infty)\end{bmatrix} = \begin{bmatrix}0 & 0 & 0 & 1\\ 1 & 1 & 1 & 1\\ \text{-}1 & 1 & \text{-}1 & 1\\ \frac{1}{8} & \frac{1}{4} & \frac{1}{2} & 1\\ \text{-}\frac{1}{8} & \frac{1}{4} & \text{-}\frac{1}{2} & 1\\ 8 & 4 & 2 & 1\\ 1 & 0 & 0 & 0\end{bmatrix} \cdot \begin{bmatrix}\alpha_3\\ \alpha_2\\ \alpha_1\\ \alpha_0\end{bmatrix} \quad (4) \qquad \begin{bmatrix}\mathcal{C}(0)\\ \mathcal{C}(1)\\ \mathcal{C}(\text{-}1)\\ \mathcal{C}(\frac{1}{2})\\ \mathcal{C}(\text{-}\frac{1}{2})\\ \mathcal{C}(2)\\ \mathcal{C}(\infty)\end{bmatrix} = \begin{bmatrix}\mathcal{A}(0)\\ \mathcal{A}(1)\\ \mathcal{A}(\text{-}1)\\ \mathcal{A}(\frac{1}{2})\\ \mathcal{A}(\text{-}\frac{1}{2})\\ \mathcal{A}(2)\\ \mathcal{A}(\infty)\end{bmatrix} \cdot \begin{bmatrix}\mathcal{B}(0)\\ \mathcal{B}(1)\\ \mathcal{B}(\text{-}1)\\ \mathcal{B}(\frac{1}{2})\\ \mathcal{B}(\text{-}\frac{1}{2})\\ \mathcal{B}(2)\\ \mathcal{B}(\infty)\end{bmatrix} \quad (5)$$

$$\begin{aligned} A(x) &= x^{\frac{3n}{4}} \sum_{i=\frac{3n}{4}}^{n-1} a_i x^{(i-\frac{3n}{4})} + \cdots + x^{\frac{n}{4}} \sum_{i=\frac{n}{4}}^{\frac{2n}{4}-1} a_i x^{(i-\frac{n}{4})} + \sum_{i=0}^{\frac{n}{4}-1} a_i x^i \\ &= \alpha_3 \cdot x^{\frac{3n}{4}} + \alpha_2 \cdot x^{\frac{2n}{4}} + \alpha_1 \cdot x^{\frac{n}{4}} + \alpha_0 \\ \implies \mathcal{A}(\mathcal{X}) &= \alpha_3 \cdot \mathcal{X}^3 + \alpha_2 \cdot \mathcal{X}^2 + \alpha_1 \cdot \mathcal{X} + \alpha_0, \quad \text{where} \quad \mathcal{X} = x^{\frac{n}{4}}. \end{aligned} \quad (3)$$

Toom-k evaluates $\mathcal{A}(\mathcal{X})$ and $\mathcal{B}(\mathcal{X})$ in at least $2k-1$ points $[p_0, p_1, \dots p_{2k-2}]$, starting with two trivial points $\{0, \infty\}$, and extending them with $\{\pm 1, \pm\frac{1}{2}, \pm 2, \dots\}$ for the ease of computations. Karatsuba, Toom-3, and Toom-4 evaluate in $\{0, 1, \infty\}$, $\{0, \pm 1, -2, \infty\}$ and $\{0, \pm 1, \pm\frac{1}{2}, 2, \infty\}$, respectively.

The pointwise multiplication computes $\mathcal{C}(p_i) = \mathcal{A}(p_i) * \mathcal{B}(p_i)$ for all values of p_i in $2k-1$ evaluation points. If the sizes of polynomials are small, then these multiplications can be performed directly using the Schoolbook algorithm. Otherwise, additional layers of Toom-k should be applied to further reduce the cost of multiplication.

The inverse operation for evaluation is interpolation. Given evaluation points $\mathcal{C}(p_i)$ for $i \in [0, \dots 2k-2]$, the optimal interpolation presented by Borato et al. [23] yields the shortest inversion-sequence for up to Toom-5.

We adopt the following formulas for the Toom-4 interpolation, based on the thesis of F. Mansouri [24], with slight modifications:

$$\begin{bmatrix}\theta_0\\ \theta_1\\ \theta_2\\ \theta_3\\ \theta_4\\ \theta_5\\ \theta_6\end{bmatrix} = \begin{bmatrix}0 & 0 & 0 & 0 & 0 & 0 & 1\\ 1 & 1 & 1 & 1 & 1 & 1 & 1\\ 1 & \text{-}1 & 1 & \text{-}1 & 1 & \text{-}1 & 1\\ \frac{1}{64} & \frac{1}{32} & \frac{1}{16} & \frac{1}{8} & \frac{1}{4} & \frac{1}{2} & 1\\ \frac{1}{64} & \text{-}\frac{1}{32} & \frac{1}{16} & \text{-}\frac{1}{8} & \frac{1}{4} & \text{-}\frac{1}{2} & 1\\ 64 & 32 & 16 & 8 & 4 & 2 & 1\\ 1 & 0 & 0 & 0 & 0 & 0 & 0\end{bmatrix}^{-1} \cdot \begin{bmatrix}\mathcal{C}(0)\\ \mathcal{C}(1)\\ \mathcal{C}(\text{-}1)\\ \mathcal{C}(\frac{1}{2})\\ \mathcal{C}(\text{-}\frac{1}{2})\\ \mathcal{C}(2)\\ \mathcal{C}(\infty)\end{bmatrix} \quad \text{where} \quad \mathcal{C}(\mathcal{X}) = \sum_{i=0}^{6} \theta_i \mathcal{X}^i \quad (6)$$

In summary, the overview of a polynomial multiplication using Toom-k is shown in Algorithm 7, where splitting and recomposition are merged into evaluation and interpolation.

Algorithm 7: Toom-k: Product of two polynomials $A(x)$ and $B(x)$

```
  Input: Two polynomials $A(x)$ and $B(x)$
  Output: $C(x) = A(x) \times B(x)$
1 $[\mathcal{A}_0(\mathcal{X}), \dots \mathcal{A}_{2k-2}(\mathcal{X})] \leftarrow$ Evaluation of $A(x)$
2 $[\mathcal{B}_0(\mathcal{X}), \dots \mathcal{B}_{2k-2}(\mathcal{X})] \leftarrow$ Evaluation of $B(x)$
3 for $i \leftarrow 0$ to $2k-2$ do
4     $\mathcal{C}_i(\mathcal{X}) = \mathcal{A}_i(\mathcal{X}) * \mathcal{B}_i(\mathcal{X})$
5 $C(x) \leftarrow$ Interpolation of $[\mathcal{C}_0(\mathcal{X}), \dots \mathcal{C}_{2k-2}(\mathcal{X})]$
```

Toom-k has a complexity $O(n^{\frac{\log(2k-1)}{\log k}})$. As a result, Toom-3 has a complexity of $O(n^{\frac{\log 5}{\log 3}})= O(n^{\log_3 5}) \approx O(n^{1.46})$, and Toom-4 has a complexity of $O(n^{\frac{\log 7}{\log 4}})= O(n^{\log_4 7}) \approx O(n^{1.40})$.

3.4.3 Number Theoretic Transform

The Number Theoretic Transform (NTT) is a transformation used as a basis for a polynomial multiplication algorithm with the time complexity of $O(n \log n)$ [25]. This algorithm performs multiplication in the ring $\mathcal{R}_q = \mathbf{Z}_q[X]/(X^n + 1)$, where degree n is a power of 2. The modulus $q \equiv 1 \bmod 2n$ for complete NTT, and $q \equiv 1 \bmod n$ for incomplete NTT, respectively. Multiplication algorithms based on NTT compute pointwise multiplication of vectors with elements of degree 0 in the case of Saber, and of degree 1 in the case of Kyber.

Complete $\mathcal{NTT}$ is similar to traditional $\mathcal{FFT}$ but uses the root of unity in the discrete field rather than in real numbers. $\mathcal{NTT}$ and $\mathcal{NTT}^{-1}$ are forward and inverse operations, where $\mathcal{NTT}^{-1}(\mathcal{NTT}(f)) = f$ for all $f \in \mathcal{R}_q$. $\mathcal{C}_i = \mathcal{A}_i * \mathcal{B}_i$ denote pointwise multiplication for all $i \in [0, \ldots n-1]$. The algorithm used to multiply two polynomials is shown in Equation 7.

$$\begin{aligned} C(x) &= A(x) \times B(x) \\ &= \mathcal{NTT}^{-1}(\mathcal{C}) = \mathcal{NTT}^{-1}(\mathcal{A} * \mathcal{B}) = \mathcal{NTT}^{-1}(\mathcal{NTT}(A) * \mathcal{NTT}(B)) \end{aligned} \tag{7}$$

In Incomplete $\mathcal{NTT}$, the idea is to pre-process polynomial before converting it to the NTT domain. In Kyber, the Incomplete NTT has $q \equiv 1 \bmod n$ [8]. The two polynomials $A(x), B(x)$, and the result $C(x)$ are split to polynomials with odd and even indices, as shown in Equation 8. $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ and A, B, C indicate polynomials in the NTT domain and time domain, respectively. An example shown in this section is Incomplete $\mathcal{NTT}$ used in Kyber.

$$\begin{aligned} C(x) = A(x) \times B(x) &= (A_{even}(x^2) + x \cdot A_{odd}(x^2)) \times (B_{even}(x^2) + x \cdot B_{odd}(x^2)) \\ &= (A_{odd} \times (x^2 \cdot B_{odd}) + A_{even} \times B_{even}) + x \cdot (A_{even} \times B_{odd} + A_{odd} \times B_{even}) \\ &= C_{even}(x^2) + x \cdot C_{odd}(x^2) \in \mathbf{Z}_q[x]/(x^n + 1). \end{aligned} \tag{8}$$

The pre-processed polynomials are converted to the NTT domain in Equation 9. In Equation 8, we combine $\beta(x^2) = x^2 \cdot B_{odd}(x^2)$, because $\beta(x^2) \in \mathbf{Z}_q[x]/(x^n + 1)$, so $\beta(x^2) = (-B_{odd}[n-1], B_{odd}[0], B_{odd}[1], \ldots B_{odd}[n-2])$. From Equation 8, we derive Equations 10 and 11.

$$\begin{aligned} A(x) &= A_{even}(x^2) + x \cdot A_{odd}(x^2) \\ \implies \mathcal{A} &= \mathcal{NTT}(A) \\ \Leftrightarrow [\mathcal{A}_{even}, \mathcal{A}_{odd}] &= [\mathcal{NTT}(A_{even}), \mathcal{NTT}(A_{odd})] \qquad (9) \\ (8) \implies \mathcal{C} &= [\mathcal{C}_{even}, \mathcal{C}_{odd}] \\ \text{where} \quad \mathcal{C}_{even} &= \mathcal{A}_{odd} * \mathcal{NTT}(x^2 \cdot B_{odd}) + \mathcal{A}_{even} * \mathcal{B}_{even} \\ &= \mathcal{A}_{odd} * \overrightarrow{\mathcal{B}_{odd}} + \mathcal{A}_{even} * \mathcal{B}_{even} \quad \text{with} \quad \overrightarrow{\mathcal{B}_{odd}} = \mathcal{NTT}(\beta) \qquad (10) \\ \text{and} \quad \mathcal{C}_{odd} &= \mathcal{A}_{even} * \mathcal{B}_{odd} + \mathcal{A}_{odd} * \mathcal{B}_{even} \qquad (11) \end{aligned}$$

After $\mathcal{C}_{odd}$ and $\mathcal{C}_{even}$ are calculated, the inverse $\mathcal{NTT}$ of $\mathcal{C}$ is calculated as follows:

$$\begin{aligned} C(x) &= \mathcal{NTT}^{-1}(\mathcal{C}) = [\mathcal{NTT}^{-1}(\mathcal{C}_{odd}), \mathcal{NTT}^{-1}(\mathcal{C}_{even})] \\ &= C_{even}(x^2) + x \cdot C_{odd}(x^2) \end{aligned} \tag{12}$$

To some extent, Toom-Cook evaluates a certain number of points, while $\mathcal{NTT}$ evaluates all available points and then computes the pointwise multiplication. The inverse $\mathcal{NTT}$ operation has similar meaning to the interpolation in Toom-k. $\mathcal{NTT}$ suffers overhead in pre-processing and post-processing for all-point evaluations. However, when polynomial degree n is large enough, the computational cost of $\mathcal{NTT}$ is smaller than the cost of Toom-k. The downside of $\mathcal{NTT}$ is the NTT friendly ring $\mathcal{R}_q$.

The summary of polynomial multiplication using the incomplete $\mathcal{NTT}$, a.k.a. **1Pt**$\mathcal{NTT}$, is shown in Algorithm 8.

Algorithm 8: 1Pt$\mathcal{NTT}$: Product of $A(x)$ and $B(x) \in \mathbf{Z}_q[x]/(x^n+1)$

```
   Input: Two polynomials $A(x)$ and $B(x)$ in $\mathbf{Z}_q[x]/(x^n+1)$
   Output: $C(x) = A(x) \times B(x)$
1  $[\mathcal{A}_{odd}, \mathcal{A}_{even}] \leftarrow \mathcal{NTT}(A(x))$
2  $[\mathcal{B}_{odd}, \mathcal{B}_{even}, \overrightarrow{\mathcal{B}_{odd}}] \leftarrow \mathcal{NTT}(B(x))$
3  for $i \leftarrow 0$ to $n-1$ do
4      $\mathcal{C}^i_{odd} = \mathcal{A}^i_{even} * \mathcal{B}^i_{odd} + \mathcal{A}^i_{odd} * \mathcal{B}^i_{even}$
5      $\mathcal{C}^i_{even} = \mathcal{A}^i_{odd} * \overrightarrow{\mathcal{B}^i_{odd}} + \mathcal{A}^i_{even} * \mathcal{B}^i_{even}$
6  $C(x) \leftarrow [\mathcal{NTT}^{-1}(\mathcal{C}_{even}), \mathcal{NTT}^{-1}(\mathcal{C}_{odd})]$
```

4 Toom-Cook in NTRU and Saber Implementations

Batch schoolbook multiplication. To compute multiplication in batch, using vector registers of the NEON technology, we allocate 3 memory blocks for 2 inputs and 1 output of each multiplication. Inputs and the corresponding output are transposed before and after batch schoolbook, respectively. To make the transposition efficient, we only transpose matrices of the size 8×8 and remember the location of each 8×8 block in batch-schoolbook. A single 8×8 transpose operation requires at least 27 vector registers. Thus, memory spills occur when the transpose matrix is of the size 16×16. In our experiments, utilizing batch schoolbook with a matrix of the size 16×16 yields the smallest latency. Schoolbook 16×16 has one spill, 17×17 and 18×18 cause 5 and 14 spills, respectively, and waste additional registers to store a few coefficients.

Karatsuba (K2) is implemented in two versions, as original Karatsuba [20], and combined two layers of Karatsuba ($K2 \times K2$), as shown in Algorithms 9 and 10. One-layer Karatsuba converts one polynomial of the length of n coefficients to 3 polynomials of the length of

$n/2$ coefficients. It introduces a zero bit-loss.

```
Algorithm 9: 2×Karasuba: Evaluate4(A) over points: $\{0,1,\infty\}$
Input: $A \in \mathbf{Z}[X] : A(X) = \sum_{i=0}^{3} \alpha_i \cdot X^i$
Output: $[\mathcal{A}_0(x), \ldots \mathcal{A}_8(x)] \leftarrow$ Evaluate4(A)
1 $w_0 = \alpha_0; \quad w_2 = \alpha_1; \quad w_1 = \alpha_0 + \alpha_1;$
2 $w_6 = \alpha_2; \quad w_8 = \alpha_3; \quad w_7 = \alpha_2 + \alpha_3;$
3 $w_3 = \alpha_0 + \alpha_2; \quad w_5 = \alpha_1 + \alpha_3; \quad w_4 = w_3 + w_5;$
4 $[\mathcal{A}_0(x), \ldots \mathcal{A}_8(x)] \leftarrow [w_0, \ldots, w_8]$
```

```
Algorithm 10: 2×Karatsuba: Interpolate4(A) over points: $\{0,1,\infty\}$
Input: $[\mathcal{A}_0(x), \ldots \mathcal{A}_8(x)] \in \mathbf{Z}[X]$
Output: $A(x) \leftarrow$ Interpolate4($\mathcal{A}$)
1 $[\alpha_0, \ldots \alpha_8] \leftarrow [\mathcal{A}_0(x), \ldots \mathcal{A}_8(x)]$
2 $w_0 = \alpha_0; \quad w_6 = \alpha_8;$
3 $w_1 = \alpha_1 - \alpha_0 - \alpha_2; \quad w_3 = \alpha_4 - \alpha_3 - \alpha_5; \quad w_5 = \alpha_7 - \alpha_6 - \alpha_8;$
4 $w_3 = w_3 - w_1 - w_5; \quad w_2 = \alpha_3 - \alpha_0 + (\alpha_2 - \alpha_6); \quad w_4 = \alpha_5 - \alpha_8 - (\alpha_2 - \alpha_6);$
5 $A(x) \leftarrow$ Recomposition of $[w_0, \ldots w_6]$
```

The **Toom-3** (TC3) evaluation and interpolation adopts the optimal sequence from Bodrato et al. [23,24] over points $\{0, \pm 1, -2, \infty\}$. To utilize 32 registers in ARM and reduce memory load and store, the two evaluation layers of Toom-3 are combined ($TC3 \times TC3$), as shown in Algorithm 11. Toom-3 converts one polynomial of the length n to five polynomials of the length $n/3$ and introduces a 1-bit loss due to a 1-bit shift operation in interpolation.

```
Algorithm 11: 2×Toom-3: Evaluate9(A) over points: $\{0, \pm 1, -2, \infty\}$
Input: $A \in \mathbf{Z}[X]$: $A(X) = \sum_{i=0}^{8} \alpha_i \cdot X^i$
Output: $[\mathcal{A}_0(x), \ldots \mathcal{A}_{24}(x)] \leftarrow$ Evaluate9($A$)
1 $w_0 = \alpha_0; \quad w_1 = (\alpha_0 + \alpha_2) + \alpha_1; \quad w_2 = (\alpha_0 + \alpha_2) - \alpha_1;$
    $w_3 = ((w_2 + \alpha_2) \ll 1) - \alpha_0; \quad w_4 = \alpha_2;$
2 $e_0 = (\alpha_0 + \alpha_6) + \alpha_3; \quad e_1 = (\alpha_1 + \alpha_7) + \alpha_4; \quad e_2 = (\alpha_2 + \alpha_8) + \alpha_5;$
    $w_{05} = e_0; \quad w_{06} = (e_0 + e_2) + e_1; \quad w_{07} = (e_0 + e_2) - e_1;$
    $w_{08} = ((w_{07} + e_2) \ll 1) - e_0; \quad w_{09} = e_2;$
3 $e_0 = (\alpha_0 + \alpha_6) - \alpha_3; \quad e_1 = (\alpha_1 + \alpha_7) - \alpha_4; \quad e_2 = (\alpha_2 + \alpha_8) - \alpha_5;$
    $w_{10} = e_0; \quad w_{11} = (e_2 + e_0) + e_1; \quad w_{12} = (e_2 + e_0) - e_1;$
    $w_{13} = ((w_{12} + e_2) \ll 1) - e_0; \quad w_{14} = e_2;$
4 $e_0 = ((2 \cdot \alpha_6 - \alpha_3) \ll 1) + \alpha_0; \quad e_1 = ((2 \cdot \alpha_7 - \alpha_4) \ll 1) + \alpha_1;$
    $e_2 = ((2 \cdot \alpha_8 - \alpha_5) \ll 1) + \alpha_2; \quad w_{15} = e_0; \quad w_{16} = (e_2 + e_0) + e_1;$
    $w_{17} = (e_2 + e_0) - e_1; \quad w_{18} = ((w_{17} + e_2) \ll 1) - e_0; \quad w_{19} = e_2;$
5 $w_{20} = \alpha_6; \quad w_{21} = (\alpha_6 + \alpha_8) + \alpha_7; \quad w_{22} = (\alpha_6 + \alpha_8) - \alpha_7;$
    $w_{23} = ((w_{22} + \alpha_8) \ll 1) - \alpha_6; \quad w_{24} = \alpha_8;$
6 $[\mathcal{A}_0(x), \ldots \mathcal{A}_{24}(x)] \leftarrow [w_0, \ldots w_{24}]$
```

Toom-4 (TC4) does evaluation and interpolation over points $\{0, \pm 1, \pm \frac{1}{2}, 2, \infty\}$. The interpolation adopts the optimal inverse-sequence from [23,24], with the slight modification, as shown in Algorithms 16 and 17. Toom-4 introduces a 3 bit-loss. A combined Toom-4 implementation was considered but not implemented due to a 6-bit loss and high complexity.

4.1 Implementation Strategy.

Each NEON vector register holds 128 bits. Each coefficient is a 16-bit integer. Hence, we can pack at most 8 coefficients into 1 vector register. The base case of Toom-Cook is a schoolbook multiplication, as shown in Algorithm 7, line 4. The pointwise multiplication is implemented using either the Schoolbook method or additional Toom-k. We use the

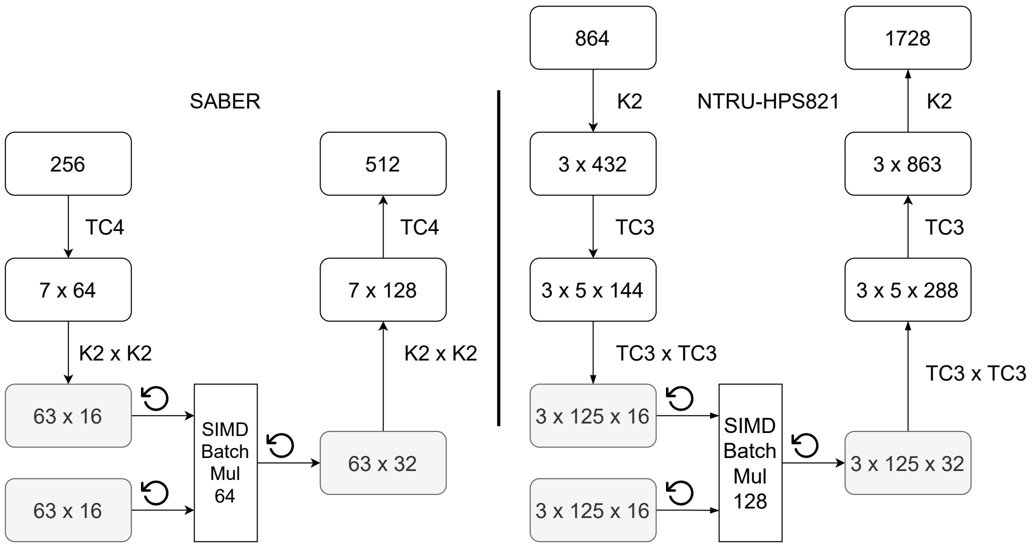

Figure 1: The Toom-Cook implementation strategy for Saber and NTRU-HPS821

notion $(k_1, k_2, \dots)$ for the Toom-k strategy at each layer. The Toom-k strategy for the polynomial of length n follows four simple rules:

1. Utilize available registers by processing as many coefficients as possible.
2. Schoolbook size should be close to 16.
3. The number of polynomials in the batch Schoolbook should be close to a multiple of 8.
4. The Toom-k strategy must generate a minimum number of polynomials.

4.2 Saber

We follow the optimization strategy from Mera et al. [26]. We precompute evaluation and lazy interpolation, which helps to reduce the number of evaluations and interpolations in `MatrixVectorMul` from $(2l^2, l^2)$ to (l^2+l, l), where l is $(2, 3, 4)$ for the security levels $(1, 3, 5)$, respectively. We also employ the Toom-k settings $(k_1, k_2) = (4, 4)$ and $(k_1, k_2, k_3) = (4, 2, 2)$ for both `InnerProd` and `MatrixVectorMul`. A graphical representation of a polynomial multiplication in Saber is shown in Fig. 1. The $\uparrow$ and $\downarrow$ are evaluation and interpolation, respectively.

4.3 NTRU

In NTRU, `poly_Rq_mul` and `poly_S3_mul` are polynomial multiplications in $(q, \Phi_1\Phi_n)$ and $(3, \Phi_n)$ respectively, where Φ_1 and Φ_n are defined in Table 2. Our `poly_Rq_mul` multiplication supports $(q, \Phi_1\Phi_n)$. In addition, we implement `poly_mod_3_Phi_n` on top of `poly_Rq_mul` to convert to $(3, \Phi_n)$. Thus, only the multiplication in $(q, \Phi_1\Phi_n)$ is implemented.

NTRU-HPS821. According to Table 1, we have 4 available bits from a 16-bit type. The optimal design that meets all rules is $(k_1, k_2, k_3, k_4) = (2, 3, 3, 3)$, as shown in Fig. 1. Using this setting, we compute 125 schoolbook multiplications of the size 16×16 in each batch,

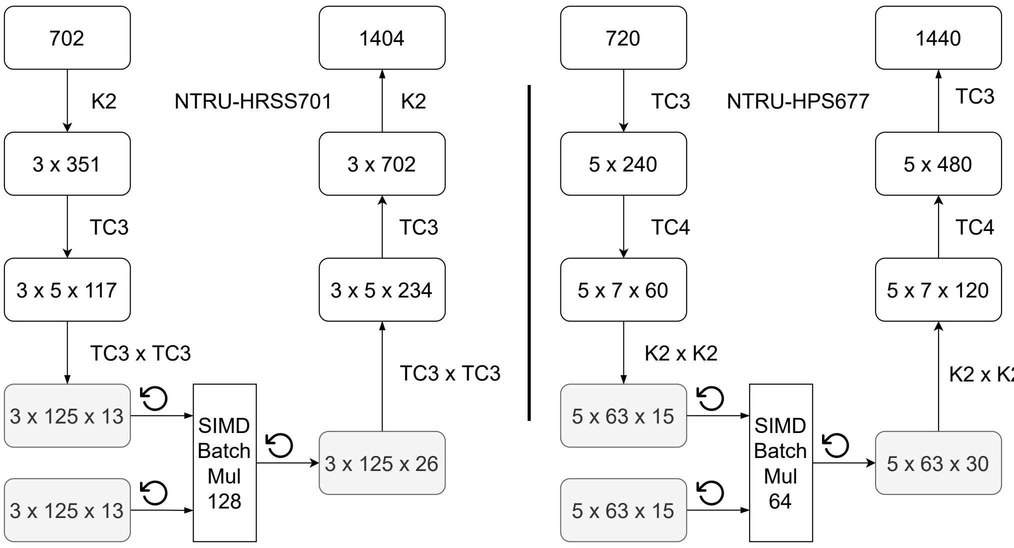

Figure 2: Toom-Cook implementation strategy for NTRU-HRSS701 and NTRU-HPS677

with 3 batches in total.

NTRU-HRSS701. With 3 bits available, there is no option other than $(k_1, k_2, k_3, k_4) = (2, 3, 3, 3)$, similar to NTRU-HPS821. We apply the $TC3 \times TC3$ evaluation to reduce the load and store operations, as shown in Fig. 2.

NTRU-HPS677. With 5 bits available, we could pad the polynomial length to 702 and reuse the NTRU-HRSS701 implementation. However, we improve the performance by 27% on Cortex-A72 by applying the new setting $(k_1, k_2, k_3, k_4) = (3, 4, 2, 2)$, which utilizes 4 available bits. This requires us to pad the polynomial length to 720, as shown in Fig. 2.

5 NTT in Kyber and Saber Implementations

5.1 NTT

As mentioned in Section 3.4.3, the NTT implementation consists of two functions. `Forward` NTT uses the Cooley-Tukey [25] and `Inverse` NTT uses the Gentleman-Sande [27] algorithms. Hence, we define the ZEROTH, FIRST, ... SEVENTH NTT level by the distance of indices in power of 2. For example, in the FIRST and SECOND level, the distances are 2^1 and 2^2, respectively. For simplicity, we consider 32 consecutive coefficients, with indices starting at $32i$, for $i \in [0, \dots 7]$, as a block. The index traversals of the first 5 levels are shown in Fig. 3. Each color defines four consecutive indices.

NTT level 0 to 4. In the ZEROTH and FIRST level, we utilize a single load and interleave instruction `vld4q_s16` to load data to 4 consecutive vector registers $[r_0, r_1, r_2, r_3]$. The computation between registers $[r_0, r_1], [r_2, r_3]$ and $[r_0, r_2], [r_1, r_3]$ satisfy the distances 2^0 and 2^1 in the ZEROTH and FIRST level respectively. This feature is shown using curly brackets on the left and right of the second block in Fig. 3.

In the SECOND and THIRD level, we perform 4×4 matrix transpose on the left-half and right-half of four vector registers, with the pair of registers $[r_0, r_1]$, $[r_2, r_3]$ and $[r_0, r_2]$, $[r_1, r_3]$ satisfying the SECOND and THIRD level respectively. See the color changes in the

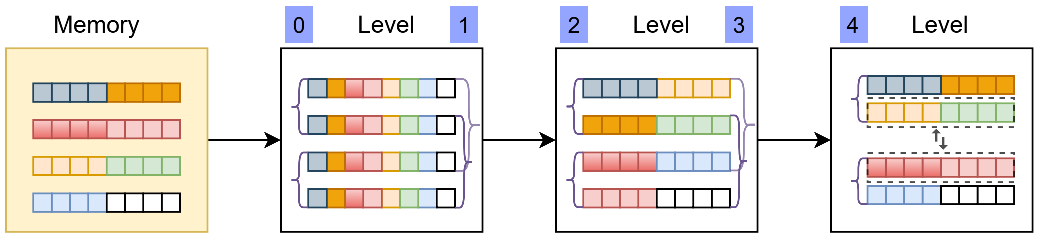

Figure 3: Index traversals up to the FOURTH level of NTT

third block in Fig. 3.

In the FOURTH level, we perform 4 transpose instructions to arrange the left-half and right-half of two vector pairs $[r_0, r_1]$ and $[r_2, r_3]$ to satisfy the distance 2^4. Then, we swap the indices of two registers $[r_1, r_2]$ by twisting the addition and subtraction in butterfly output. Doing it converts the block to its original order, used originally in the memory. See the memory block and the fourth block in Fig. 3.

NTT level 5 to level 6. In the FIFTH level, we create one more block of 32 coefficients and duplicate the steps from previous levels. We process 64 coefficients and utilize 8 vector registers $[r_0, \ldots r_3], [r_4, \ldots r_7]$. It is obvious that the vector pairs $[r_i, r_{i+4}]$ for $i \in [0, \ldots 3]$ satisfy the distance 2^5 in the butterfly. The SIXTH level is similar to the FIFTH level. Two blocks are added and duplicate the process from the NTT levels 0 to 5. Additionally, 128 coefficients are stored in 16 vector registers as 4 blocks, the operations between vector pairs $[r_i, r_{i+8}]$, for $i \in [0, \ldots 7]$, satisfy the distance 2^6.

NTT level 7 and n^{-1}. The SEVENTH level is treated as a separate loop. We unroll the loop to process 128 coefficients with the distance 2^7. Additionally, the multiplication with n^{-1} in `Inverse` NTT is precomputed with a constant multiplier at the last level, which further saves multiplication instructions.

5.2 Range Analysis

The Kyber and Saber NTT use 16-bit signed integers. Thus, there are 15 bits for data and 1 sign bit. With 15 bits, we can store the maximum value of $-2^{15} \le \beta \cdot q < 2^{15}$ before an overflow. In case of Kyber $(\beta, q) = (9, 3329)$. In case of Saber, $q = (7681, 10753)$ and $\beta = (4, 3)$, respectively.

Kyber. The optimal number of Barrett reductions in Inverse NTT is 72 points, as shown in Westerbaan [5] and applied to the reference implementation. After Barrett reduction has been changed from targeting $0 \le r < q$ to $-\frac{q-1}{2} \le r < \frac{q-1}{2}$, coefficients grow by at most q instead of $2q$ in absolute value at the level 1. We can decrease the number of reduction points further, from 72 to 64. The indices of 64 lazy reduction points in Kyber can be seen in Table 3.

Level	Indices	Total
4	$32 \to 35$, $96 \to 99$, $160 \to 163$, $224 \to 227$	16
5	$0 \to 7$, $64 \to 71$, $128 \to 135$, $192 \to 199$	32
6	$8 \to 15$, $136 \to 143$	16

Table 3: Barrett reduction over 64 points in Inverse NTT of Kyber

Saber. In Twisted-NTT [6, 13], we can compute the first 3 levels without additional reductions. We can apply range analysis and use Barrett reduction. Instead, we twist constant multipliers to the ring of the form $Z_q[x]/(x^n - 1)$ in the THIRD level, which not only reduces coefficients to the range $-q \leq r < q$, but also reduces the number of modular multiplications at subsequent levels. This approach is less efficient than regular NTT uses Barrett reduction in `neon`, however the performance different is negligible due to small $\beta = 3$.

5.3 Vectorizing modular reduction

In Algorithms 12 and 13, we show the implementations of the vectorized multiplication modulo a 16-bit q and vectorized central Barrett reduction, respectively. Both implementations are expressed using NEON intrinsics. Neon intrinsics are function calls that the compiler replaces with an appropriate Neon instruction or a sequence of Neon instructions. Intrinsics provide almost as much control as writing assembly language, but leave the allocation of registers to the compiler.

Inspired by *Fast mulmods* from [6,13], in Algorithm 12, we use four `smull_s16` multiply long and one `mul_s16` multiply instructions. We use the unzip instructions to gather 16-bit low and high half-products. Unlike AVX2, ARMv8 does not have an instruction similar to `vpmulhw`. Therefore, dealing with 32-bit products is unavoidable. In Algorithm 12, lines $1 \rightarrow 4$ can be simplified with 2 AVX2 instructions `vpmullw, vpmulhw`. Similarly, lines $6 \rightarrow 8$ can be simplified with a single high-only half-product `vpmulhw`. The multiplication by q^{-1} in line 5 can be incorporated into lines $1 \rightarrow 2$ to further save one multiplication. In total, we use twice as many multiplication instructions as compared to the code for AVX2 [13]. In the vectorized Barrett reduction, used in both Kyber and Saber, we use three multiplication instructions – one additional multiplication as compared to AVX2, as shown in Algorithm 13.

Algorithm 12: Vectorized multiplication modulo a 16-bit q

```
  Input: $B = (B_L, B_H)$, $C = (C_L, C_H), R = 2^{16}$
  Output: $A = B * (CR)$ mod q
1 $T_0 \leftarrow$ smull_s16($B_L, C_L$)
2 $T_1 \leftarrow$ smull_s16($B_H, C_H$)
3 $T_2 \leftarrow$ uzp1_s16($T_0, T_1$)
4 $T_3 \leftarrow$ uzp2_s16($T_0, T_1$)
5 $(A_L, A_H) \leftarrow$ mul_s16($T_2, q^{-1}$)
6 $T_1 \leftarrow$ smull_s16($A_L, q$)
7 $T_2 \leftarrow$ smull_s16($A_H, q$)
8 $T_0 \leftarrow$ uzp2_s16($T_1, T_2$)
9 $A \leftarrow T_3 - T_0$
```

Algorithm 13: Vectorized central Barrett reduction

```
  Input: $B = (B_L, B_H)$, constant $V = (V_L, V_H)$, Kyber: $(i, n) = (9, 10)$
  Output: $A = B$ mod q and $-q/2 \leq A < q/2$
1 $T_0 \leftarrow$ smull_s16($B_L, V_L$)
2 $T_1 \leftarrow$ smull_s16($B_H, V_H$)
3 $T_0 \leftarrow$ uzp2_s16($T_0, T_1$)
4 $T_1 \leftarrow$ vadd_n_s16($T_0, 1 \ll i$)
5 $T_1 \leftarrow$ shr_n_s16($T_1, n$)
6 $A \leftarrow$ mls_s16($B, T_1, q$)
```

6 Methodology

ARMv8 NEON intrinsics are used for ease of implementation and to take advantage of the compiler optimizers. These optimizers have built-in knowledge on how to best translate intrinsics to assembly language instructions. As a result, some optimizations may be available to reduce the number of NEON instructions. The optimizer can expand the intrinsic and align the buffers, schedule pipeline, or make adjustments depending on the platform architecture[2]. In our implementation, we always keep vector register usage under 32 and examine assembly language code obtained during the development process. We acknowledge the compiler spills to memory and hide load/store latency in favor of pipelining multiple multiplication instructions.

Benchmarking setup.

Our benchmarking setup for ARMv8 implementations included MacBook Air with Apple M1 SoC and Raspberry Pi 4 with `Cortex-A72 @ 1.5 GHz`. For AVX2 implementations, we used a PC based on `Intel Core i7-8750H @ 4.1 GHz`. Additionally, in Tables 4 and 9, we report benchmarking results for the newest x86-64 chip in `supercop-20210125` [28], namely `AMD EPYC 7742 @ 2.25 GHz`. There is no official clock frequency documentation for Apple M1 CPU. However, independent benchmarks strongly indicate that the clock frequency of 3.2 GHz is used[3].

We use PAPI [29] library to count cycles on Cortex-A72. In Apple M1, we rewrite the work from Dougall Johnson[4] to perform cycles count[5].

In terms of compiler, we used `clang 12.0` (default version) for Apple M1 and `clang 11.1` (the most recent stable version) for Cortex-A72 and Core i7-8750H. All benchmarks were conducted with the compiler settings `-O3 -mtune=native -fomit-frame-pointer`. 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.

The number of executions on ARMv8 Cortex-A72 and Intel i7-8750H was $1,000,000$. On Apple M1, it was $10,000,000$ to force the benchmarking process to run on the high-performance 'Firestorm' core. The benchmarking results are in kilocycles *(kc)*.

7 Results

Speed-up vs. pure C implementations.

In Table 4, we summarize benchmarking results, expressed in clock cycles for a) C only implementation (denoted as `ref`), running on Apple M1; b) NEON implementation (denoted as `neon`), running on Apple M1; and c) AVX2 implementation (denoted as AVX2), running on AMD EPYC 7742. The column `ref/neon` represents the speed-up of the NEON implementation over the C only implementation. The column AVX2/`neon` represents the speed-up of the NEON implementation running on Apple M1 vs. the AVX2 implementation running on AMD EPYC 7742. $assuming that both processors operate with the same clock frequency$.

The speed-up of the NEON implementation over the C only implementation is the smallest for Saber and the largest for NTRU. For encapsulation, the speed-ups vary in the ranges 1.37-1.60 for Saber, 2.33-2.45 for Kyber, and 3.05-3.24 for NTRU-HPS. For NTRU-HRSS, it reaches 6.68. For decapsulation, the corresponding speed-ups vary in the ranges 1.55-1.74 for Saber, 2.96-3.04 for Kyber, and 7.89-8.49 for NTRU-HPS. For NTRU-HRSS, it reaches 7.24.

[2] https://godbolt.org/z/5qefG5
[3] https://www.anandtech.com/show/16252/mac-mini-apple-m1-tested
[4] https://github.com/dougallj
[5] https://github.com/GMUCERG/PQC_NEON/blob/main/neon/kyber/m1cycles.c

Algorithm	`ref` (kc)		`neon` (kc)		AVX2 (kc)		`ref/neon`		AVX2/`neon`	
	E	**D**	**E**	**D**	**E**	**D**	**E**	**D**	**E**	**D**
lightsaber	50.9	54.9	37.2	35.3	41.9	42.2	1.37	1.55	1.13	1.19
kyber512	75.7	89.5	32.6	29.4	28.4	22.6	2.33	3.04	0.87	0.77
ntru-hps677	183.1	430.4	60.1	54.6	26.0	45.7	3.05	7.89	0.43	0.84
ntru-hrss701	152.4	439.9	22.8	60.8	20.4	47.7	6.68	7.24	0.90	0.78
saber	90.4	96.2	59.9	58.0	70.9	70.7	1.51	1.66	1.18	1.22
kyber768	119.8	137.8	49.2	45.7	43.4	35.2	2.43	3.02	0.88	0.77
ntru-hps821	245.3	586.5	75.7	69.0	29.9	57.3	3.24	8.49	0.39	0.83
firesaber	140.9	150.8	87.9	86.7	103.3	103.7	1.60	1.74	1.18	1.20
kyber1024	175.4	198.4	71.6	67.1	63.0	53.1	2.45	2.96	0.88	0.79

Table 4: Execution time of **E**ncapsulation and **D**ecapsulation for three security levels. `ref` and `neon`: results for Apple M1. AVX2: results for AMD EPYC 7742. kc-kilocycles.

The only algorithm for which the NEON implementation on Apple M1 takes fewer clock cycles than the AVX2 implementation on AMD EPYC 7742 is Saber. The speed-up of Apple M1 over this AMD processor varies between 1.13 and 1.22, depending on the operation and the security level.

Rankings. Based on Table 5, the ranking of C only implementations on Apple M1 is consistently 1. Saber, 2. Kyber, 3. NTRU (with the exception that NTRU is not represented at level 5). The advantage of Saber over Kyber is relatively small (by a factor of 1.24-1.49 for encapsulation and 1.31-1.63 for decapsulation). This advantage decreases for higher security levels. The advantage of Saber over NTRU is more substantial, 2.71-3.00 for encapsulation and 6.10-8.01 for decapsulation. This advantage decreases between security levels 1 and 3.

For NEON implementations, running on Apple M1, the rankings change substantially, as shown in Table 6. For encapsulation at level 1, the ranking becomes 1. NTRU, 2. Kyber, 3. Saber, i.e., reversed compared to pure C implementations. For all levels and both major operations, Kyber and Saber swap places. At the same time, the differences between Kyber and Saber do not exceed 29% and slightly increase for higher security levels.

The rankings for pure C implementations do not change when a high-speed core of Apple M1 is replaced by the Cortex-A72 core, as shown in Table 7. Additionally, the differences between Saber and Kyber become even smaller.

The ranking of NEON implementations also does not change between Apple M1 and Cortex-A72, as shown in Table 8. However, at level 1, NTRU is almost in tie with Kyber. For all other cases, the advantage of Kyber increases as compared to the rankings for Apple M1.

Finally, in Table 9, the rankings for AVX2 implementations running on the AMD EPYC 7742 core are presented. For encapsulation, at levels 1 and 3, the rankings are 1. NTRU, 2. Kyber, 3. Saber. Compared to the NEON implementations, the primary difference is the no. 1 position of NTRU at level 3. For decapsulation, the rankings are identical at all three security levels. The advantage of Kyber over Saber is higher than for NEON implementations.

NTT implementation. In Table 10, the speed-ups of `neon` vs. `ref` are reported for the forward and inverse NTT of Kyber, running on Cortex-A72. These speed-ups are 5.77 and 7.54 for NTT and inverse NTT, respectively. Both speed-ups are substantially higher than for the entire implementation of Kyber, as reported in Table 4. There is no official NTT-based reference implementation of Saber released yet. Therefore, we analyzed cycle

Rank	C Apple M1					
	E	*kc*	↑	**D**	*kc*	↑
1	`lightsaber`	50.9	1.00	`lightsaber`	54.9	1.00
2	`kyber512`	75.7	1.49	`kyber512`	89.5	1.63
3	`ntru-hrss701`	152.4	3.00	`ntru-hps677`	430.4	7.84
4	`ntru-hps677`	183.1	3.60	`ntru-hrss701`	439.9	8.01
1	`saber`	90.4	1.00	`saber`	96.2	1.00
2	`kyber768`	119.8	1.32	`kyber768`	137.8	1.43
3	`ntru-hps821`	245.3	2.71	`ntru-hps821`	586.5	6.10
1	`firesaber`	140.9	1.00	`firesaber`	150.8	1.00
2	`kyber1024`	175.4	1.24	`kyber1024`	198.4	1.31

Table 5: Ranking of investigated candidates for C only implementations in terms of **E**ncapsulation and **D**ecapsulation on Apple M1 processor. The baseline is the smallest number of cycles for each security level.

Rank	neon Apple M1					
	E	*kc*	↑	**D**	*kc*	↑
1	`ntru-hrss701`	22.7	1.00	`kyber512`	29.4	1.00
2	`kyber512`	32.5	1.43	`lightsaber`	35.3	1.20
3	`lightsaber`	37.2	1.63	`ntru-hps677`	54.5	1.85
4	`ntru-hps677`	60.1	2.64	`ntru-hrss701`	60.7	2.06
1	`kyber768`	49.2	1.00	`kyber768`	45.7	1.00
2	`saber`	59.9	1.22	`saber`	58.0	1.27
3	`ntru-hps821`	75.7	1.54	`ntru-hps821`	69.0	1.51
1	`kyber1024`	71.6	1.00	`kyber1024`	67.1	1.00
2	`firesaber`	87.9	1.23	`firesaber`	86.7	1.29

Table 6: Ranking of investigated candidates for NEON implementations in terms of **E**ncapsulation and **D**ecapsulation on Apple M1 processor. The baseline is the smallest number of cycles for each security level.

Rank	C Cortex-A72					
	E	*kc*	↑	**D**	*kc*	↑
1	`lightsaber`	154.8	1.00	`lightsaber`	165.9	1.00
2	`kyber512`	184.5	1.19	`kyber512`	223.4	1.35
3	`ntru-hrss701`	458.7	2.96	`ntru-hps677`	1,346.7	8.12
4	`ntru-hps677`	570.8	3.69	`ntru-hrss701`	1,353.4	8.16
1	`saber`	273.4	1.00	`saber`	294.5	1.00
2	`kyber768`	298.9	1.09	`kyber768`	349.1	1.19
3	`ntru-hps821`	748.1	2.74	`ntru-hps821`	1,830.0	6.21
1	`firesaber`	427.3	1.00	`firesaber`	460.9	1.00
2	`kyber1024`	440.6	1.03	`kyber1024`	503.8	1.09

Table 7: Ranking of investigated candidates for C only implementations in terms of **E**ncapsulation and **D**ecapsulation on Cortex-A72 core. The baseline is the smallest number of cycles for each security level.

Rank	neon Cortex-A72					
	E	*kc*	↑	**D**	*kc*	↑
1	ntru-hrss701	93.6	1.00	kyber512	94.1	1.00
2	kyber512	95.3	1.02	lightsaber	131.2	1.39
3	lightsaber	130.1	1.39	ntru-hps677	205.8	2.19
4	ntru-hps677	181.7	1.94	ntru-hrss701	262.9	2.79
1	kyber768	151.0	1.00	kyber768	149.8	1.00
2	saber	213.6	1.41	saber	215.4	1.44
3	ntru-hps821	232.6	1.54	ntru-hps821	274.5	1.83
1	kyber1024	223.8	1.00	kyber1024	220.7	1.00
2	firesaber	321.6	1.44	firesaber	329.6	1.49

Table 8: Ranking of investigated candidates for NEON implementations in terms of **E**ncapsulation and **D**ecapsulation on Cortex-A72 core. The baseline is the smallest number of cycles for each security level.

Rank	avx2 AMD EPYC 7742					
	E	*kc*	↑	**D**	*kc*	↑
1	ntru-hrss701	20.4	1.00	kyber512	22.5	1.00
2	ntru-hps677	25.9	1.27	lightsaber	42.1	1.87
3	kyber512	28.3	1.39	ntru-hps677	45.7	2.03
4	lightsaber	41.9	2.05	ntru-hrss701	47.6	2.11
1	ntru-hps821	29.9	1.00	kyber768	35.2	1.00
2	kyber768	43.4	1.45	saber	57.3	1.63
3	saber	70.9	2.37	ntru-hps821	70.7	2.01
1	kyber1024	63.0	1.00	kyber1024	53.1	1.00
2	firesaber	103.3	1.64	firesaber	103.7	1.95

Table 9: Ranking of investigated candidates for AVX2 implementations in terms of **E**ncapsulation and **D**ecapsulation on AMD EPYC 7742 processor. The baseline is the smallest number of cycles for each security level.

`Cortex-A72` Apple M1	`ref` NTT	`ref` NTT^{-1}	`neon` NTT	`neon` NTT^{-1}	`ref/neon` NTT	`ref/neon` NTT^{-1}	Levels
				`Cortex-A72`			
`saber`	-	-	1,991	1,893	-	-	$0 \to 7$
`kyber`	8,500	12,533	1,473	1,661	5.8	7.5	$1 \to 7$
				Apple M1			
saber	-	-	539	531	-	-	$0 \to 7$
kyber	3,211	5,118	413	428	7.8	12.0	$1 \to 7$

Table 10: Cycle counts of the NEON-based NTT implementation on `Cortex-A72` and Apple M1

`Cortex-A72` 1500 MHz	Level 1 *(kilocycles)* `ref`	Level 1 `neon`	Level 1 `ref/neon`	Level 3 *(kilocycles)* `ref`	Level 3 `neon`	Level 3 `ref/neon`
	Level 1: NTRU-HRSS701 | NTRU-HPS677, Level 3: NTRU-HPS821					
`poly_Rq_mul`	426.8	70.1| 55.0	6.09 | 7.78	583.9	83.5	6.99
`poly_S3_mul`	432.8	72.2| 56.1	5.99 | 7.76	588.7	83.1	7.08
	Saber: Toom-Cook | NTT					
`InnerProd`	27.7	18.1| 22.5	1.53 | 1.23	41.4	25.0| 31.5	1.64 | 1.31
`MatrixVectorMul`	55.2	40.2| 37.0	1.37 | 1.49	125.7	81.0| 71.3	1.55 | 1.76
	Kyber					
`VectorVectorMul`	44.4	7.1	6.3	59.7	9.9	6.1
`MatrixVectorMul`	68.1	10.7	6.4	117.5	19.3	6.1

Table 11: Cycle counts for the implementations of polynomial multiplication in NTRU, Saber, and Kyber measured in kilocycles – `neon` vs. `ref`

counts in the forward and inverse NTT transform for our NEON-based implementation only without comparing it with any reference implementation.

NTT and Toom-Cook multiplication. In Table 11, cycle counts for the implementations of polynomial multiplication in NTRU, Saber, and Kyber are reported. NTRU-HRSS701 and NTRU-HPS677 share the implementation of the polynomial multiplication in the `ref` implementation. In the `neon` implementation of `poly_Rq_mul`, the NTRU-HPS677 takes 55.0 *kilocycles*, which corresponds to the speed-up of 7.78 over `ref`, as compared to 6.09 for NTRU-HRSS701. In the case of Saber, the two numbers for `neon` and `ref/neon` represent Toom-Cook and NTT-based implementations, respectively. The Toom-Cook implementation of `InnerProd` shows better speed across security levels 1, 3, and 5. In contrast, for `MatrixVectorMul`, the NTT-based implementation outperforms Toom-k implementation for all security levels. When Saber uses NTT as a replacement for the Toom-Cook implementation on Cortex-A72 and Apple M1, performance gains in encapsulation are $(-1\%, +2\%, +5\%)$ and $(-15\%, -13\%, -14\%)$. For decapsulation, they are $(-4\%, -2\%, +2\%)$ and $(-21\%, -18\%, -19\%)$, respectively. In most cases, these gains are negative. Overall, they do not warrant replacing the use of the Toom-Cook algorithm by an algorithm based on NTT.

In Table 12, we summarize results for key generation executed on Cortex-A72 and Apple M1. The NTRU key generation was not implemented as it requires inversion. As a result, it is both time-consuming to implement and has a much longer execution time.

By using `neon` instructions, the key generation for Kyber is sped up, as compared to the reference implementation, by a factor in the range 2.03–2.15 for Cortex-A72 and 2.58–2.91 for Apple M1. For Saber, the speed-ups are more moderate in the ranges 1.13–1.29 and 1.41–1.55, respectively.

Keygen	`Cortex-A72`*(kc)*			Apple M1*(kc)*		
	`ref`	`neon`	`ref/neon`	`ref`	`neon`	`ref/neon`
LIGHTSABER	134.9	119.5	1.13	44.0	31.2	1.41
KYBER512	136.7	67.4	2.03	59.3	23.0	2.58
SABER	237.3	192.9	1.23	74.4	51.3	1.45
KYBER768	237.7	110.7	2.15	104.9	36.3	2.89
FIRESABER	370.5	286.6	1.29	119.2	77.0	1.55
KYBER1024	371.9	176.2	2.11	162.9	55.9	2.91

Table 12: Key generation time for Saber and Kyber over three security levels measured in kilocycles (*kc*) - `Cortex-A72` vs. `Apple M1`

8 Conclusions

The rankings of lattice-based PQC KEM finalists in terms of speed in software are substantially different for C only implementations and implementations sped-up using SIMD instructions of ARMv8 and Intel processors. At the same time, these rankings are similar for NEON implementations and AVX2 implementations. The biggest changes are the lower positions of `ntru-hps677` and `ntru-hps821` for NEON implementations. Based on the results obtained to date, the optimal algorithms for implementing polynomial multiplication in ARMv8 using NEON instructions are NTT for CRYSTALS-Kyber and Toom-Cook for NTRU and Saber.

References

[1] D. J. Bernstein and P. Schwabe, "NEON Crypto," in *Cryptographic Hardware and Embedded Systems - CHES 2012*, vol. 7428 of *LNCS*, (Leuven, Belgium), pp. 320–339, Sept. 2012.

[2] R. Azarderakhsh, Z. Liu, H. Seo, and H. Kim, "NEON PQCryto: Fast and Parallel Ring-LWE Encryption on ARM NEON Architecture," p. 8.

[3] S. Streit and F. De Santis, "Post-Quantum Key Exchange on ARMv8-A: A New Hope for NEON Made Simple," *IEEE Transactions on Computers*, vol. 67, pp. 1651–1662, Nov. 2018.

[4] M. Scott, "A Note on the Implementation of the Number Theoretic Transform," in *Cryptography and Coding. IMACC 2017.* (M. O'Neill, ed.), vol. 10655 of *Lecture Notes in Computer Science*, (Cham), pp. 247–258, Springer International Publishing, 2017.

[5] B. Westerbaan, "When to Barrett reduce in the inverse NTT," Cryptology ePrint Archive 2020/1377, Nov. 2020.

[6] G. Seiler, "Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography," Cryptology ePrint Archive 2018/039, Jan. 2018.

[7] V. Lyubashevsky and G. Seiler, "NTTRU: Truly Fast NTRU Using NTT," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2019, May 2019.

[8] S. Zhou, H. Xue, D. Zhang, K. Wang, X. Lu, B. Li, and J. He, "Preprocess-then-NTT Technique and Its Applications to Kyber and NewHope," in *International Conference on Information Security and Cryptology, Inscrypt 2018*, vol. 11449 of *LNCS*, (Cham), pp. 117–137, Springer, 2019.

[9] M. J. Kannwischer, J. Rijneveld, P. Schwabe, and K. Stoffelen, "Pqm4 - Post-quantum crypto library for the {ARM} {Cortex-M4}." https://github.com/mupq/pqm4, 2019.

[10] L. Botros, M. J. Kannwischer, and P. Schwabe, "Memory-Efficient High-Speed Implementation of Kyber on Cortex-M4," Tech. Rep. 489, 2019.

[11] E. Alkim, Y. Alper Bilgin, M. Cenk, and F. Gérard, "Cortex-M4 optimizations for {R,M} LWE schemes," *TCHES*, vol. 2020, pp. 336–357, June 2020.

[12] A. Karmakar, J. M. Bermudo Mera, S. Sinha Roy, and I. Verbauwhede, "Saber on ARM," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2018, pp. 243–266, Aug. 2018.

[13] C.-M. M. Chung, V. Hwang, M. J. Kannwischer, G. Seiler, C.-J. Shih, and B.-Y. Yang, "NTT Multiplication for NTT-unfriendly Rings: New Speed Records for Saber and NTRU on Cortex-M4 and AVX2," *TCHES*, pp. 159–188, Feb. 2021.

[14] N. Gupta, A. Jati, A. K. Chauhan, and A. Chattopadhyay, "PQC Acceleration Using GPUs: FrodoKEM, NewHope, and Kyber," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, pp. 575–586, Mar. 2021.

[15] S. Sinha Roy, "SaberX4: High-Throughput Software Implementation of Saber Key Encapsulation Mechanism," in *2019 IEEE 37th International Conference on Computer Design (ICCD)*, (Abu Dhabi, United Arab Emirates), pp. 321–324, IEEE, Nov. 2019.

[16] O. Danba, *Optimizing NTRU Using AVX2.* Master's Thesis, Radboud University, Nijmegen, Netherlands, July 2019.

[17] G. L. Hoang, *Optimization of the NTT Function on ARMv8-A SVE.* Bachelor's Thesis, Radboud University, The Netherlands, June 2018.

[18] E. Fujisaki and T. Okamoto, "Secure Integration of Asymmetric and Symmetric Encryption Schemes," *Journal of Cryptology*, vol. 26, pp. 80–101, Jan. 2013.

[19] "Post-Quantum Cryptography: Round 3 Submissions." https://csrc.nist.gov/Projects/post-quantum-cryptography/round-3-submissions, 2021.

[20] A. Karatsuba and Y. Ofman, "Multiplication of many-digital numbers by automatic computers," *Doklady Akademii Nauk SSSR*, vol. 145, no. 2, pp. 293–294, 1962.

[21] A. Toom, "The complexity of a scheme of functional elements realizing the multiplication of integers," *Soviet Mathematics Doklady*, vol. 3, pp. 714–716, 1963.

[22] S. A. Cook and S. O. Aanderaao, "On the Minimum Computation Time of Functions," *Transactions of the American Mathematical Society*, vol. 142, pp. 291–314, 1969.

[23] M. Bodrato and A. Zanoni, "Integer and Polynomial Multiplication: Towards Optimal Toom-Cook Matrices," in *International Symposium on Symbolic and Algebraic Computation, ISSAC 2007*, pp. 17–24, July 2007.

[24] F. Mansouri, *On The Parallelization Of Integer Polynomial Multiplication.* Master's Thesis, The University of Western Ontario, 2014.

[25] J. W. Cooley and J. W. Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series," *Mathematics of computation*, vol. 19, no. 90, pp. 297–301, 1965.

[26] J. M. Bermudo Mera, A. Karmakar, and I. Verbauwhede, "Time-memory trade-off in Toom-Cook multiplication: An application to module-lattice based cryptography," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2020, pp. 222–244, Mar. 2020.

[27] W. M. Gentleman and G. Sande, "Fast Fourier Transforms: For fun and profit," in *Fall Joint Computer Conference, AFIPS '66*, (San Francisco, CA), pp. 563–578, ACM Press, Nov. 1966.

[28] D. J. Bernstein and T. Lange, "eBACS: ECRYPT Benchmarking of Cryptographic Systems." https://bench.cr.yp.to, 2021.

[29] D. Terpstra, H. Jagode, H. You, and J. Dongarra, "Collecting Performance Data with PAPI-C," in *Tools for High Performance Computing 2009*, pp. 157–173, Berlin, Heidelberg: Springer Berlin Heidelberg, 2010.

A NEON vs. AVX2

Our benchmarks show that in Saber, NTT performs better in AVX2 than in the `neon` implementation, as shown in Table 14. We believe that the gap is caused by the lack of an instruction of ARMv8 equivalent to `vmulhw`. In the case of Kyber, we consistently achieve `ref/neon` ratio greater than 9.0 in `VectorVectorMul` $NTT^{-1}(\hat{t} * NTT(r))$ and `MatrixVectorMul` $NTT^{-1}(\hat{A} * NTT(r))$, as shown in Table 11.

In Table 6, `neon` and AVX2 implementations of Kyber are the fastest in decapsulation across all security levels. In the `neon` implementation of Kyber, the leftover bottleneck is `SHAKE128/256`. Although we implemented a 2×`KeccakF1600` permutation function that utilizes 128-bit vector registers, the performance gain is only 25%. We expect that the speed-up will be greater when there is hardware support for `SHA-3`. In the AVX2/`neon` comparison, the `neon` implementations of `MatrixVectorMul`, `VectorVectorMul`, and NTT have the performance at the levels of 25% → 27% of the performance of AVX2 (see Table 15). However, for the entire encapsulation and decapsulation, these differences are significantly smaller (see Table 4).

In the case of Saber, we selected the Toom-Cook implementation approach for `ref`, `neon`, and AVX2. The `neon` consistently outperforms AVX2. Please note that the `ref` implementations of Saber and NTRU employ similar Toom-k settings as the `neon` and AVX2 implementations. In addition, the `neon` Toom-k multiplications in `InnerProd`, `MatrixVectorMul` perform better than the NTT implementations, as shown in Table 14.

The performance of `neon` for NTRU-HPS677 and NTRU-HPS821 are very close to the performance of AVX2. Additionally, when compared to the `ref` implementation, the decapsulation speed-up of `neon` is consistently greater than 7.

NTRU (cycles)	`ref`	`neon`	AVX2	`ref/neon`	AVX2/`neon`
677-poly_Rq_mul	134,965	11,571	5,848	11.66	0.51
677-poly_S3_mul	137,657	11,857	6,426	11.61	0.54
701-poly_Rq_mul	133,765	15,580	5,902	8.59	0.38
701-poly_S3_mul	136,514	15,975	6,159	8.55	0.39
821-poly_Rq_mul	186,136	17,148	8,744	10.85	0.51
821-poly_S3_mul	189,189	17,538	8,685	10.79	0.50

Table 13: The NTRU multiplication benchmark on Apple M1 vs. Core i7-8750H

Saber (cycles)	`ref`	`neon`	AVX2	`ref/neon`	AVX2/`neon`
poly_ntt		539	161		0.30
poly_invntt_tomont		519	145		0.28
InnerProd-1	8,006	3,184	4,248	2.51	1.33
InnerProd-3	11,825	4,345	6,253	2.72	1.44
InnerProd-5	16,064	5,318	8,446	3.02	1.59
InnerProd-NTT-1		6,117	1,796		0.29
InnerProd-NTT-3		8,470	2,475		0.29
InnerProd-NTT-5		10,839	3,171		0.29
MatrixVectorMul-1	16,002	6,640	7,556	2.41	1.14
MatrixVectorMul-3	35,499	14,029	15,014	2.53	1.07
MatrixVectorMul-5	64,343	21,591	25,412	2.98	1.18
MatrixVectorMul-NTT-1		10,077	2,976		0.30
MatrixVectorMul-NTT-3		18,931	5,575		0.29
MatrixVectorMul-NTT-5		30,393	9,018		0.30

Table 14: SABER multiplication benchmark on Apple M1 vs Core i7-8750H AVX2.

Kyber (cycles)	`ref`	`neon`	AVX2	`ref/neon`	AVX2/`neon`
poly_ntt	3,211	413	125	7.77	0.30
poly_invntt_tomont	5,118	428	131	11.96	0.31
VectorVectorMul-1	17,319	1,858	493	9.32	0.27
VectorVectorMul-3	23,317	2,545	658	9.16	0.26
VectorVectorMul-5	29,632	3,271	841	9.06	0.26
MatrixVectorMul-1	26,830	2,809	770	9.55	0.27
MatrixVectorMul-3	46,454	4,910	1,247	9.46	0.25
MatrixVectorMul-5	70,084	7,651	1,908	9.16	0.25

Table 15: The Kyber multiplication benchmark on Apple M1 vs. Core i7-8750H AVX2

Input Length	Output Length	2× `ref`	`neon`	2×`ref/neon`
32	1,664	15,079	11,620	1.30
32	3,744	33,249	26,251	1.27
32	6,656	57,504	45,658	1.26

Table 16: SHAKE128 performance with dual-lane 2×`KeccakF1600` `neon` vs. 2×`ref`, benchmark on Apple M1.

B Toom-Cook algorithms

B.1 Toom-3

Algorithm 14: Toom-3: Evaluate3(A) over points: $\{0, \pm 1, -2, \infty\}$

```
**Input:** $A \in \mathbf{Z}[X] : A(X) = \sum_{i=0}^{2} \alpha_i \cdot X^i$
**Output:** $[\mathcal{A}_0(x), \dots \mathcal{A}_4(x)] \leftarrow$ **Evaluate3**(A)
1 $w_0 = \alpha_0$;   $w_1 = (\alpha_0 + \alpha_2) + \alpha_1$;   $w_2 = (\alpha_0 + \alpha_2) - \alpha_1$;
2 $w_3 = ((w_2 + \alpha_2) \ll 1) - \alpha_0$;   $w_4 = \alpha_2$;
3 $[\mathcal{A}_0(x), \dots \mathcal{A}_4(x)] \leftarrow [w_0, \dots w_4]$
```

Algorithm 15: Toom-3: Interpolate3($\mathcal{A}$) over points: $\{0, \pm 1, -2, \infty\}$

```
**Input:** $[\mathcal{A}_0(x), \dots \mathcal{A}_4(x)] \in \mathbf{Z}[X]$
**Output:** $A(x) \leftarrow$ **Interpolate3**($\mathcal{A}$)
1 $[\alpha_0, \dots \alpha_4] \leftarrow [\mathcal{A}_0(x), \dots \mathcal{A}_4(x)]$
2 $e_1 = (\alpha_1 - \alpha_2) \gg 1$;   $e_2 = \alpha_2 - \alpha_0$;   $e_3 = (\alpha_3 - \alpha_1)/3$;
3 $w_0 = \alpha_0$;   $w_4 = \alpha_4$;   $w_2 = e_2 + e_1 - \alpha_4$;
4 $w_3 = (e_2 - e_3) \gg 1 + (\alpha_4 \ll 1)$;   $w_1 = e_1 - w_3$;
5 $A(x) \leftarrow$ **Recomposition** of $[w_0, \dots w_4]$
```

B.2 Toom-4

Algorithm 16: Toom-4: Evaluate4(A) over points: $\{0, \pm 1, \pm \frac{1}{2}, 2, \infty\}$

```
**Input:** $A \in \mathbf{Z}[X] : A(X) = \sum_{i=0}^{3} \alpha_i \cdot X^i$
**Output:** $[\mathcal{A}_0(x), \dots \mathcal{A}_6(x)] \leftarrow$ **Evaluate4**(A)
1 $w_0 = \alpha_0$;   $e_0 = \alpha_0 + \alpha_2$;   $e_1 = \alpha_1 + \alpha_3$; $w_1 = e_0 + e_1$;   $w_2 = e_0 - e_1$;
2 $e_0 = (4 \cdot \alpha_0 + \alpha_2) \ll 1$;   $e_1 = 4 \cdot \alpha_1 + \alpha_3$;   $w_3 = e_0 + e_1$;   $w_4 = e_0 - e_1$;
3 $w_5 = (\alpha_3 \ll 3) + (\alpha_2 \ll 2) + (\alpha_1 \ll 1) + \alpha_0$;   $w_6 = \alpha_3$;
4 $[\mathcal{A}_0(x), \dots \mathcal{A}_6(x)] \leftarrow [w_0, \dots w_6]$
```

Algorithm 17: Toom-4: Interpolate4(A) over points: $\{0, \pm 1, \pm \frac{1}{2}, 2, \infty\}$

```
**Input:** $[\mathcal{A}_0(x), \dots \mathcal{A}_6(x)] \in \mathbf{Z}[X]$
**Output:** $A(x) \leftarrow$ **Interpolate4**($\mathcal{A}$)
1 $[w_0, \dots w_6] \leftarrow [\mathcal{A}_0(x), \dots \mathcal{A}_6(x)]$
2 $w_5$ += $w_3$;   $w_4$ += $w_3$;   $w_4 \gg= 1$;   $w_2$ += $w_1$;   $w_2 \gg= 1$;
3 $w_3$ −= $w_4$;   $w_1$ −= $w_2$;   $w_5$ −= $w_2 \cdot 65$;   $w_2$ −= $w_6$;   $w_2$ −= $w_0$;
4 $w_5$ += $w_2 \cdot 45$;   $w_4$ −= $w_6$;   $w_4 \gg= 2$;   $w_3 \gg= 1$;
5 $w_5$ −= $w_3 \ll 2$;   $w_3$ −= $w_1$;   $w_3$ /= 3;   $w_4$ −= $w_0 \ll 4$;
6 $w_4$ −= $w_2 \ll 2$;   $w_4$ /= 3;   $w_2$ += $w_4$;   $w_5 \gg= 1$;   $w_5$ /= 15;
7 $w_1$ −= $w_5$;   $w_1$ −= $w_3$;   $w_1$ /= 3;   $w_3$ += $5 \cdot w_1$   $w_5$ −= $w_1$;
8 $A(x) \leftarrow$ **Recomposition** of $[w_0, -w_1, w_2, w_3, -w_4, w_5, w_6]$
```