Fast NEON-based multiplication for lattice-based NIST Post-Quantum Cryptography finalists

Duc Tri Nguyen and Kris Gaj

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

Abstract. This paper focuses on high-speed NEON-based constant-time implementations of multiplication of large polynomials in the NIST PQC KEM Finalists: NTRU, Saber, and CRYSTALS-Kyber. We use the Number Theoretic Transform (NTT)-based multiplication in Kyber, the Toom-Cook algorithm in NTRU, and both types of multiplication in Saber. Following these algorithms and using Apple M1, we improve the decapsulation performance of the NTRU, Kyber, and Saber-based KEMs at the security level 3 by the factors of 8.4, 3.0, and 1.6, respectively, compared to the reference implementations. On Cortex-A72, we achieve the speed-ups by factors varying between 5.7 and 7.5× for the Forward/Inverse NTT in Kyber, and between 6.0 and 7.8× for Toom-Cook in NTRU, over the best existing implementations in pure C. For Saber, when using NEON instructions on Cortex-A72, the implementation based on NTT outperforms the implementation based on the Toom-Cook algorithm by 14% in the case of the `MatrixVectorMul` function but is slower by 21% in the case of the `InnerProduct` function. Taking into account that in Saber, keys are not available in the NTT domain, the overall performance of the NTT-based version is very close to the performance of the Toom-Cook version. The differences for the entire decapsulation at the three major security levels (1, 3, and 5) are −4, −2, and +2%, respectively. Our benchmarking results demonstrate that our NEON-based implementations run on an Apple M1 ARM processor are comparable to those obtained using the best AVX2-based implementations run on an AMD EPYC 7742 processor. Our work is the first NEON-based ARMv8 implementation of each of the three NIST PQC KEM finalists.

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 and 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, RISC-V, and Intel and AMD processors supporting Advanced Vector Extensions 2 (AVX2, also known as Haswell New Instructions). However, there is still a lack of high-performance software implementations for mobile devices, which is an area dominated by ARM.

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 the 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. Our goal is to fill the gap between low-power embedded processors and power-hungry x86-64 platforms. To do that, we have developed the first optimized constant-time ARMv8 implementations of three lattice-based KEM finalists: Kyber, NTRU, and Saber. We assumed the parameter sets supported by all schemes 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 is out of scope for this paper.

In short, we have implemented the Toom-Cook multiplication for NTRU, NTT-based multiplication for Kyber, and both Toom-Cook and NTT-based multiplications for Saber. We achieved significant speed-ups as compared to the corresponding reference implementations. We have benchmarked our implementations on the best ARMv8 CPU on the market and compared them against the best implementations targeting a high-performance x86-64 CPU.

Contributions Our work is the first optimized ARMv8 `NEON` implementation of the PQC KEM finalists. Our results obtained using Apple M1 are slightly worse than the results achieved using the corresponding AVX2-based implementations, but overall, the speeds of encapsulation and decapsulation are comparable. The lack of an instruction equivalent to the AVX2 instruction `vmulhw` is responsible for a larger number of clock cycles required to implement NTT using ARMv8. We improve the performance of NTRU-HPS677 by proposing a new Toom-Cook implementation setting.

Source code is publicly available at: https://github.com/GMUCERG/PQC_NEON

2 Previous Work

The paper by Streit et al. [23] was the first work about the NEON-based ARMv8 implementation of New Hope Simple. This work, published before the NIST PQC Competition, proposed a "merged NTT layers" structure. Scott [20] and Westerbaan [26] proposed lazy reduction as a part of their NTT implementation.

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

Seiler [21] proposed an FFT trick, which has been widely adopted in the following work. Zhou et al. [27] proposed fast implementations of NTT and applied them to Kyber.

In the area of low-power implementations, most previous works targeted Cortex-M4 [16]. In particular, Botros et al. [7] and Alkim et al. [2] developed ARM Cortex-M4 implementations of Kyber. Karmakar et al. [18] reported results for Saber. Chung et al. [8] 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 GPU implementation of Kyber; Roy et al. [22] developed $4 \times Saber$ by utilizing 256-bit AVX2 vectors; Danba et al. [11] developed a high-speed implementation of NTRU using AVX2. Finally, Hoang et al. [15] implemented a fast NTT-function using ARMv8 Scalable Vector Extension (SVE).

3 Background

NTRU, Saber, and Kyber use variants of the Fujisaki-Okamoto (FO) transform [12] 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

The Round 3 submission of NTRU [1] is a merger of the specifications for NTRU-HPS and NTRU-HRSS. 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.

Table 1. Parameters of Kyber, Saber, and NTRU

	Polynomial	n			p [, q]			$\|pk\|$ (B)			$\|ct\| - \|pk\|$ (B)		
		1	3	5	1	3	5	1	3	5	1	3	5
Kyber	$x^n + 1$	256			3329			800	1184	1568	$\downarrow$ 32	$\downarrow$ 96	0
Saber	$x^n + 1$	256			$2^{13}, 2^{10}$			672	992	1312	$\uparrow$ 64	$\uparrow$ 96	$\uparrow$ 160
NTRU-HPS	$\Phi_1 = x - 1$	677	821	–	2^{11}	2^{12}	–	931	1230	–	0	0	–
NTRU-HRSS	$\Phi_n = \frac{x^n-1}{x-1}$	701	-		2^{13}	-		1138	-				

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

Saber [1] 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 (NTT) to be applied *directly*. Among the three investigated algorithms, Saber has the smallest public keys and ciphertext sizes, $|pk|$ and $|ct|$, as shown in Table 1.

3.3 Kyber

The security of Kyber [3] 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 [12]. 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 of $O(n \log n)$.

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.

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

$$\begin{array}{ccccc} Schoolbook & \longrightarrow & Toom-Cook & \longrightarrow & NTT \\ \uparrow & & \uparrow & & \uparrow \\ O(n^2) & & O(n^{\frac{\log(2k-1)}{\log k}}) & & O(n \log n) \end{array}$$

Schoolbook Multiplication 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.

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

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 [17] is a special case of Toom-Cook (Toom-k) [9, 25]. 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 1. 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.

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

Toom-k evaluates $\mathcal{A}(\mathcal{X})$ and $\mathcal{B}(\mathcal{X})$ in at least $2k-1$ points $[p_0, p_1, \ldots p_{2k-2}]$, starting with two trivial points $\{0, \infty\}$, and extending them with $\{\pm 1, \pm\frac{1}{2}, \pm 2, \ldots\}$ 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.

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

$$\begin{bmatrix} \mathcal{C}(0) \\ \mathcal{C}(1) \\ \mathcal{C}(\text{-}1) \\ \mathcal{C}(\frac{1}{2}) \\ \mathcal{C}(\text{-}\frac{1}{2}) \\ \mathcal{C}(2) \\ \mathcal{C}(\infty) \end{bmatrix} = \begin{bmatrix} \mathcal{A}(0) \\ \mathcal{A}(1) \\ \mathcal{A}(\text{-}1) \\ \mathcal{A}(\frac{1}{2}) \\ \mathcal{A}(\text{-}\frac{1}{2}) \\ \mathcal{A}(2) \\ \mathcal{A}(\infty) \end{bmatrix} \cdot \begin{bmatrix} \mathcal{B}(0) \\ \mathcal{B}(1) \\ \mathcal{B}(\text{-}1) \\ \mathcal{B}(\frac{1}{2}) \\ \mathcal{B}(\text{-}\frac{1}{2}) \\ \mathcal{B}(2) \\ \mathcal{B}(\infty) \end{bmatrix} \tag{5}$$

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 $i \in [0, \ldots 2k-2]$, the optimal interpolation presented by Borato et al. [6] 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 [19], with slight modifications:

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

In summary, the overview of a polynomial multiplication using Toom-k is shown in Algorithm 1, where splitting and recomposition are merged into evaluation and interpolation.

Algorithm 1: Toom-k: Product of two polynomials $A(x)$ and $B(x)$

Input: Two polynomials $A(x)$ and $B(x)$
Output: $C(x) = A(x) \times B(x)$
$[\mathcal{A}_0(\mathcal{X}), \dots \mathcal{A}_{2k-2}(\mathcal{X})] \leftarrow$ **Evaluation** of $A(x)$
$[\mathcal{B}_0(\mathcal{X}), \dots \mathcal{B}_{2k-2}(\mathcal{X})] \leftarrow$ **Evaluation** of $B(x)$
for $i \leftarrow 0$ **to** $2k-2$ **do**
  $\mathcal{C}_i(x) = \mathcal{A}_i(\mathcal{X}) * \mathcal{B}_i(\mathcal{X})$
$C(x) \leftarrow$ **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 $O(n^{\frac{\log 7}{\log 4}}) = O(n^{\log_4 7}) \approx O(n^{1.40})$.

Number Theoretic Transform (NTT) is a transformation used as a basis for a polynomial multiplication algorithm with the time complexity of $O(n \log n)$ [10]. 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 $i \in [0, \dots n-1]$. 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 $q \equiv 1 \bmod n$ [27]. 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 2.

Algorithm 2: 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: $C(x) = A(x) \times B(x)$

$[\mathcal{A}_{odd}, \mathcal{A}_{even}] \leftarrow \mathcal{NTT}(A(x))$
$[\mathcal{B}_{odd}, \mathcal{B}_{even}, \overrightarrow{\mathcal{B}_{odd}}] \leftarrow \mathcal{NTT}(B(x))$
for $i \leftarrow 0$ **to** $n-1$ **do**
  $\mathcal{C}^i_{odd} = \mathcal{A}^i_{even} * \mathcal{B}^i_{odd} + \mathcal{A}^i_{odd} * \mathcal{B}^i_{even}$
  $\mathcal{C}^i_{even} = \mathcal{A}^i_{odd} * \overrightarrow{\mathcal{B}^i_{odd}} + \mathcal{A}^i_{even} * \mathcal{B}^i_{even}$
$C(x) \leftarrow [\mathcal{NTT}^{-1}(\mathcal{C}_{even}), \mathcal{NTT}^{-1}(\mathcal{C}_{odd})]$

4 Toom-Cook in NTRU and Saber Implementations

Batch schoolbook multiplication. To compute multiplication in batch, using vector registers, we allocate 3 memory blocks for 2 inputs and 1 output for 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 requires at least 27 vector registers, thus, memory spills occur when the transpose matrix is of the size 16×16. In our experiment, utilizing batch schoolbook with a matrix of the size 16×16 yields the best throughput. Schoolbook 16×16 has 1 spill, 17×17 and 18×18 cause 5 and 14 spills and waste additional registers to store a few coefficients.

Karatsuba (K2) is implemented in two versions, original Karatsuba [17], and combined two layers of Karatsuba ($K2 \times K2$), as shown in Algorithms 3 and 4. One-layer Karatsuba converts one polynomial of the length n to 3 polynomials of the length $n/2$ and introduces 0 bit-loss due to the addition and subtraction performed only in the interpolation step.

Algorithm 3: 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)

$w_0 = \alpha_0$; $w_2 = \alpha_1$; $w_1 = \alpha_0 + \alpha_1$;
$w_6 = \alpha_2$; $w_8 = \alpha_3$; $w_7 = \alpha_2 + \alpha_3$;
$w_3 = \alpha_0 + \alpha_2$; $w_5 = \alpha_1 + \alpha_3$; $w_4 = w_3 + w_5$;
$[\mathcal{A}_0(x), \ldots \mathcal{A}_8(x)] \leftarrow [w_0, \ldots, w_8]$

Algorithm 4: 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**$(\mathcal{A})$

$[\alpha_0, \ldots \alpha_8] \leftarrow [\mathcal{A}_0(x), \ldots \mathcal{A}_8(x)]$
$w_0 = \alpha_0; \quad w_6 = \alpha_8;$
$w_1 = \alpha_1 - \alpha_0 - \alpha_2; \quad w_3 = \alpha_4 - \alpha_3 - \alpha_5; \quad w_5 = \alpha_7 - \alpha_6 - \alpha_8;$
$w_3 = w_3 - w_1 - w_5; \quad w_2 = \alpha_3 - \alpha_0 + (\alpha_2 - \alpha_6); \quad w_4 = \alpha_5 - \alpha_8 - (\alpha_2 - \alpha_6);$
$A(x) \leftarrow$ **Recomposition** of $[w_0, \ldots w_6]$

Toom-3 (TC3) evaluation and interpolation adopts the optimal sequence from Bodrato et al. [6] 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 5. Toom-3 converts 1 polynomial of the length n to 5 polynomials of the length $n/3$ and introduces 1 bit-loss due to a 1-bit shift operation in interpolation.

Algorithm 5: 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)

$w_0 = \alpha_0; \quad w_1 = (\alpha_0 + \alpha_2) + \alpha_1; \quad w_2 = (\alpha_0 + \alpha_2) - \alpha_1;$
$w_3 = ((w_2 + \alpha_2) \ll 1) - \alpha_0; \quad w_4 = \alpha_2;$
$e_0 = (\alpha_0 + \alpha_6) + \alpha_3; \quad e_1 = (\alpha_1 + \alpha_7) + \alpha_4; \quad e_2 = (\alpha_2 + \alpha_8) + \alpha_5;$
$w_{05} = e_0; \quad w_{06} = (e_0 + e_2) + e_1; \quad w_{07} = (e_0 + e_2) - e_1;$
$w_{08} = ((w_{07} + e_2) \ll 1) - e_0; \quad w_{09} = e_2;$
$e_0 = (\alpha_0 + \alpha_6) - \alpha_3; \quad e_1 = (\alpha_1 + \alpha_7) - \alpha_4; \quad e_2 = (\alpha_2 + \alpha_8) - \alpha_5;$
$w_{10} = e_0; \quad w_{11} = (e_2 + e_0) + e_1; \quad w_{12} = (e_2 + e_0) - e_1;$
$w_{13} = ((w_{12} + e_2) \ll 1) - e_0; \quad w_{14} = e_2;$
$e_0 = ((2 \cdot \alpha_6 - \alpha_3) \ll 1) + \alpha_0; \quad e_1 = ((2 \cdot \alpha_7 - \alpha_4) \ll 1) + \alpha_1;$
$e_2 = ((2 \cdot \alpha_8 - \alpha_5) \ll 1) + \alpha_2; \quad w_{15} = e_0; \quad w_{16} = (e_2 + e_0) + e_1;$
$w_{17} = (e_2 + e_0) - e_1; \quad w_{18} = ((w_{17} + e_2) \ll 1) - e_0; \quad w_{19} = e_2;$
$w_{20} = \alpha_6; \quad w_{21} = (\alpha_6 + \alpha_8) + \alpha_7; \quad w_{22} = (\alpha_6 + \alpha_8) - \alpha_7;$
$w_{23} = ((w_{22} + \alpha_8) \ll 1) - \alpha_6; \quad w_{24} = \alpha_8;$
$[\mathcal{A}_0(x), \ldots \mathcal{A}_{24}(x)] \leftarrow [w_0, \ldots w_{24}]$

Toom-4 (TC4) evaluation and interpolation over points $\{0, \pm 1, \pm \frac{1}{2}, 2, \infty\}$. The interpolation adopts the optimal inverse-sequence from [6], with the slight modification, as shown in Algorithms 6 and 7. Toom-4 introduces a 3 bit-loss, thus a combined Toom-4 implementation was considered but not implemented due to a 6 bit-loss and high complexity.

Algorithm 6: 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 $w_0 = \alpha_0; \quad e_0 = \alpha_0 + \alpha_2; \quad e_1 = \alpha_1 + \alpha_3; \; w_1 = e_0 + e_1; \quad w_2 = e_0 - e_1;$
2 $e_0 = (4 \cdot \alpha_0 + \alpha_2) \ll 1; \quad e_1 = 4 \cdot \alpha_1 + \alpha_3;$
$w_3 = e_0 + e_1; \quad w_4 = e_0 - e_1;$
3 $w_5 = (\alpha_3 \ll 3) + (\alpha_2 \ll 2) + (\alpha_1 \ll 1) + \alpha_0; \quad w_6 = \alpha_3;$
4 $[\mathcal{A}_0(x), \dots \mathcal{A}_6(x)] \leftarrow [w_0, \dots w_6]$

Algorithm 7: 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 $w_5 \mathrel{+}= w_3; \quad w_4 \mathrel{+}= w_3; \quad w_4 \gg\!= 1; \quad w_2 \mathrel{+}= w_1; \quad w_2 \gg\!= 1;$
3 $w_3 \mathrel{-}= w_4; \quad w_1 \mathrel{-}= w_2; \quad w_5 \mathrel{-}= w_2 \cdot 65; \quad w_2 \mathrel{-}= w_6; \quad w_2 \mathrel{-}= w_0;$
4 $w_5 \mathrel{+}= w_2 \cdot 45; \quad w_4 \mathrel{-}= w_6; \quad w_4 \gg\!= 2; \quad w_3 \gg\!= 1;$
5 $w_5 \mathrel{-}= w_3 \ll 2; \quad w_3 \mathrel{-}= w_1; \quad w_3 \mathrel{/}= 3; \quad w_4 \mathrel{-}= w_0 \ll 4;$
6 $w_4 \mathrel{-}= w_2 \ll 2; \quad w_4 \mathrel{/}= 3; \quad w_2 \mathrel{+}= w_4; \quad w_5 \gg\!= 1; \quad w_5 \mathrel{/}= 15;$
7 $w_1 \mathrel{-}= w_5; \quad w_1 \mathrel{-}= w_3; \quad w_1 \mathrel{/}= 3; \quad w_3 \mathrel{+}= 5 \cdot w_1 \quad w_5 \mathrel{-}= w_1;$
8 $A(x) \leftarrow$ **Recomposition** of $[w_0, -w_1, w_2, w_3, -w_4, w_5, w_6]$

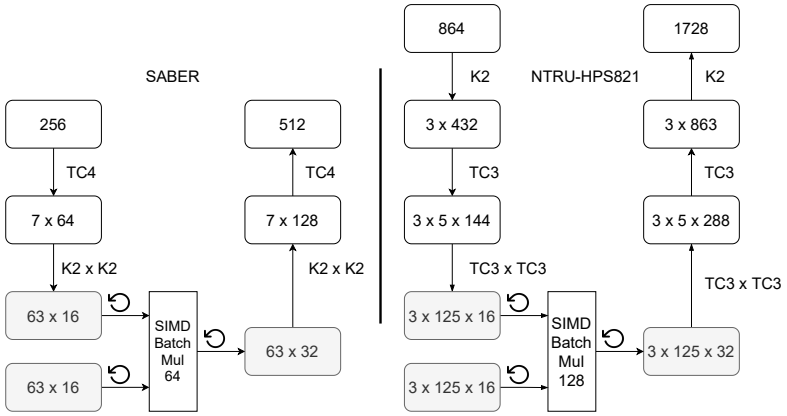

Fig. 1. The Toom-Cook implementation strategy for SABER and NTRU-HPS821

Implementation Strategy. Each vector register is 128-bit, 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 1, line 4. The point-wise multiplication is either schoolbook or additional Toom-k. We use notion $(k_1, k_2, \dots)$ as Toom-k strategy for each layer. The Toom-k strategy for the polynomial length n follows 4 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 batch schoolbook close to a multiple of 8;
4. The Toom-k strategy must generate a minimum number of polynomials.

4.1 Saber

We follow the optimization strategy from Mera et al. [4]. 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 setting $(k_1, k_2, k_3) = (4, 2, 2)$ for both `InnerProd` and `MatrixVectorMul`. An example of a polynomial multiplication in Saber is shown in Fig. 1. The $\uparrow$ and $\downarrow$ are evaluation and interpolation, respectively.

4.2 NTRU

In NTRU, `poly_Rq_mul` and `poly_S3_mul` are polynomial multiplications in $(q, \Phi_1\Phi_n)$ and $(3, \Phi_n)$ respectively. 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, 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 5, NTT implementation consists of two functions. `Forward` NTT uses the Cooley-Tukey [10] and `Inverse` NTT uses the Gentleman-Sande [13] 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

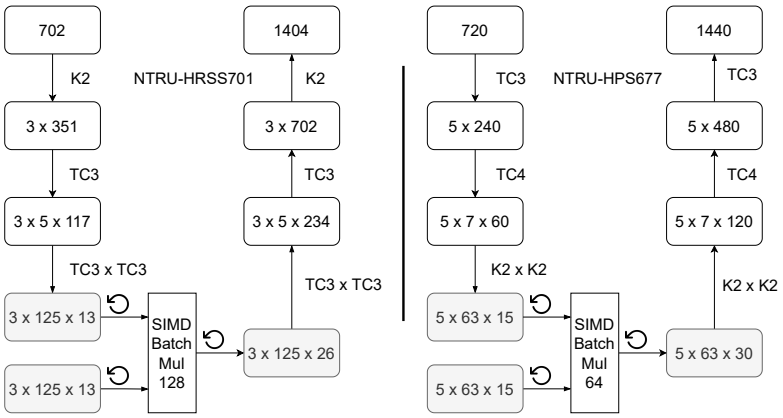

Fig. 2. Toom-Cook implementation strategy for NTRU-HPS677 and NTRU-HRSS701

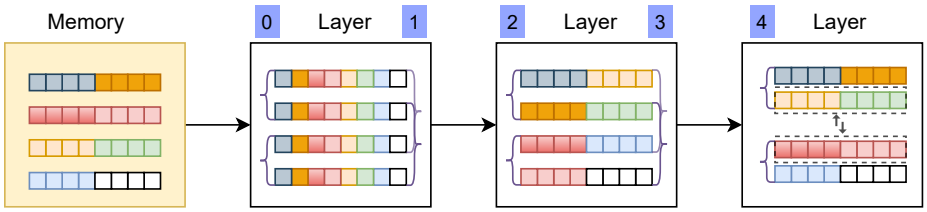

Fig. 3. Index traversals up to the NTT level FOURTH

SECOND level, the distances are 2^1 and 2^2, respectively. For simplicity, we consider 32 consecutive coefficients, with indices starting at $32i$ for $i \in [0, \ldots 7]$ 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 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 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 layer 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 $-2^{15} \leq \beta \cdot q < 2^{15}$ before 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 [26] and applied to the reference implementation. After Barrett reduction has been changed from targeting $0 \leq r < q$ to $-\frac{q-1}{2} \leq r < \frac{q-1}{2}$, 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 2.

Table 2. Improved 64 points Barrett reduction in Inverse NTT of Kyber

Layer	Indexes				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

Saber. In Twisted-NTT [8, 21], 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 Vectorized modular reduction

Inspired by *Fast mulmods* in [8, 21], we implemented 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`, thus dealing with 32-bit products is unavoidable. In Algorithm 8, 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 two more multiplication instructions, as compared to AVX2 [8]. 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 9.

Algorithm 8: 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 9: 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 Results

ARMv8 intrinsics are used for ease of implementation and to take advantage of the compiler optimizers. The optimizers know how intrinsics behave. As a result, some optimizations may be available to reduce the number of intrinsic instructions. The optimizer can expand the intrinsic and align the buffers, schedule pipeline, or make adjustments depending on the platform architecture[2].

[2] https://godbolt.org/z/5qefG5

In our implementation, we always keep vector register usage under 32 and examine assembly language code obtained during our 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 6 and 8, we report benchmarking results for the newest x86-64 chip in `supercop-20210125` [5], 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 [24] 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)*.

Table 3. Cycle counts of the NEON-based NTT implementation on `Cortex-A72` and Apple M1

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

NTT implementation. In Table 3, the speed-ups of `neon` vs. `ref` are 5.8 and 7.5 for forward and inverse NTT on Cortex-A72. On Apple M1, the corre-

[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

Table 4. Fast NTT-based and Toom-Cook implementations of multiplication in NTRU, Saber and Kyber measured in kilocycles – `neon` vs. `ref`

Cortex-A72	Level 1 *(kilocycles)*			Level 3 *(kilocycles)*		
1500 MHz	`ref`	`neon`	`ref/neon`	`ref`	`neon`	`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 5. Fast NTT-based and Toom-Cook implementations of multiplication in NTRU, Saber and Kyber measured in kilocycles – `neon` vs. AVX2

Apple M1	Level 1 *(kilocycles)*			Level 3 *(kilocycles)*		
INTEL I7-8750H	AVX2	`neon`	AVX2/`neon`	AVX2	`neon`	AVX2/`neon`
Level 1: NTRU-HRSS701 | NTRU-HPS677, Level 3: NTRU-HPS821						
`poly_Rq_mul`	6.0| 6.0	15.7| 11.6	0.38| 0.52	8.7	17.2	0.51
`poly_S3_mul`	6.2| 6.3	15.7| 11.9	0.40| 0.53	9.2	17.4	0.53
Saber: Toom-Cook | NTT						
`InnerProd`	2.2| 1.8	3.2| 6.1	0.69| 0.29	3.5| 2.4	4.3| 8.5	0.80| 0.29
`MatrixVectorMul`	3.9| 2.9	6.6| 10.1	0.59| 0.28	8.2| 5.6	14.0| 18.9	0.59| 0.30
Kyber						
`VectorVectorMul`	0.5	1.9	0.27	0.7	2.5	0.26
`MatrixVectorMul`	0.7	2.8	0.26	1.2	4.9	0.25

sponding speed-ups are 7.8 and 12.0. There is no official NTT-based reference implementation of Saber released yet. We analyzed cycle counts in the forward and inverse NTT transform for our NEON-based implementation without comparing it with any reference implementation.

NTT and Toom-Cook multiplication. In Table 4, NTRU-HRSS701 and NTRU-HPS677 share polynomial multiplication implementation 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-

Table 6. Encapsulation and **D**ecapsulation speed comparison over three security levels. `ref` and `neon` results for Apple M1. AVX2 results for AMD EPYC 7742. *kc*-kilocycles.

Apple M1	`ref` (*kc*)		`neon` (*kc*)		AVX2 (*kc*)		`ref/neon`		AVX2/`neon`	
AMD EPYC 7742	**E**	**D**	**E**	**D**	**E**	**D**	**E**	**D**	**E**	**D**
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
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-HPS821	245.3	586.5	75.7	69.0	29.9	57.3	3.24	8.49	0.39	0.83
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
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 7. SHAKE128 performance with dual-lane 2×`KeccakF1600 neon` vs. 2×`ref`, benchmark on Apple M1.

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

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. Our benchmarks show that NTT-based implementations performs better in AVX2 than in the `neon` implementation, as shown in the Saber section of Table 5. 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 6.0 in `VectorVectorMul` and `MatrixVectorMul`, as shown in Table 4.

AVX2 and NEON. 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 25% as compared to 2× reference implementation, as shown in Table 7. This speed-up translates to only a fraction of the encapsulation/decapsulation time. We expect that the speed-up will be greater when there is hardware support for `SHA3`. 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 5). However, for the entire encapsulation and decapsulation, these differences are significantly smaller (see Table 6).

Table 8. Encapsulation and **D**ecapsulation ranking benchmarked on Apple M1 and AMD EPYC processor. The baseline is the largest number of cycles for each security level.

Rank	neon				AVX2			
	E	↑	**D**	↑	**E**	↑	**D**	↑
1	ntru-hrss701	1.00	kyber512	1.00	ntru-hrss701	1.00	kyber512	1.00
2	kyber512	1.43	lightsaber	1.20	ntru-hps677	1.27	lightsaber	1.87
3	lightsaber	1.63	ntru-hps677	1.85	kyber512	1.39	ntru-hps677	2.03
4	ntru-hps677	2.64	ntru-hrss701	2.06	lightsaber	2.05	ntru-hrss701	2.11
1	kyber768	1.00	kyber768	1.00	ntru-hps821	1.00	kyber768	1.00
2	saber	1.22	saber	1.27	kyber768	1.45	saber	1.63
3	ntru-hps821	1.54	ntru-hps821	1.51	saber	2.37	ntru-hps821	2.01
1	kyber1024	1.00	kyber1024	1.00	kyber1024	1.00	kyber1024	1.00
2	firesaber	1.23	firesaber	1.29	firesaber	1.64	firesaber	1.95

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 4.

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

In Table 8, the rankings for `neon` implementations running on Apple M1 and AVX2 implementations running on the AMD EPYC 7742 core are presented. For decapsulation, the rankings are identical at all three security levels. The advantage of Kyber over Saber is higher for AVX2 than for `neon`. For encapsulation, at levels 1 and 3, NTRU-HPS is faster than Kyber and Saber only for AVX2. As a result, at level 3, NTRU is ranked no. 3 for `neon` and no. 1 for AVX2.

In Table 9, 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.

7 Conclusions

In conclusion, 1. The NEON-based NTT implementation is slower than the corresponding implementation using AVX2 due to the lack of a NEON instruction equivalent to `vmulhw`. 2. Performance of NTT in Saber is close to the performance of the Toom-Cook algorithm. We advise to continue using Toom-Cook

Table 9. Key generation time for Saber and Kyber over three security levels measured in kilocycles (*kc*) - `Cortex-A72` vs. `Apple M1`

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

on ARMv8. 3. The rankings of lattice-based PQC KEM finalists in terms of speed in software are similar for NEON-based implementations and AVX2-based implementations. The biggest change is the lower position of `ntru-hps677` and `ntru-hps821` in NEON-based implementations.

References

1. Post-Quantum Cryptography: Round 3 Submissions. https://csrc.nist.gov/Projects/post-quantum-cryptography/round-3-submissions (2021)
2. Alkim, E., Alper Bilgin, Y., Cenk, M., Gérard, F.: Cortex-M4 optimizations for {R,M} LWE schemes. TCHES **2020**(3), 336–357 (Jun 2020). https://doi.org/10.46586/tches.v2020.i3.336-357
3. Avanzi, R., Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J.M., Schwabe, P., Seiler, G., Stehlé, D.: CRYSTALS-Kyber: Algorithm Specifications and Supporting Documentation (version 3.01). Tech. rep. (Jan 2021)
4. Bermudo Mera, J.M., Karmakar, A., Verbauwhede, I.: Time-memory trade-off in Toom-Cook multiplication: An application to module-lattice based cryptography. IACR Transactions on Cryptographic Hardware and Embedded Systems **2020**(2), 222–244 (Mar 2020). https://doi.org/10.13154/TCHES.V2020.I2.222-244
5. Bernstein, D.J., Lange, T.: eBACS: ECRYPT Benchmarking of Cryptographic Systems. https://bench.cr.yp.to (2021)
6. Bodrato, M., Zanoni, A.: Integer and Polynomial Multiplication: Towards Optimal Toom-Cook Matrices. In: International Symposium on Symbolic and Algebraic Computation, ISSAC 2007. pp. 17–24 (Jul 2007). https://doi.org/10.1145/1277548.1277552
7. Botros, L., Kannwischer, M.J., Schwabe, P.: Memory-Efficient High-Speed Implementation of Kyber on Cortex-M4. In: Progress in Cryptology – AFRICACRYPT 2019. pp. 209–228. Springer International Publishing, Cham (2019)
8. Chung, C.M.M., Hwang, V., Kannwischer, M.J., Seiler, G., Shih, C.J., Yang, B.Y.: NTT Multiplication for NTT-unfriendly Rings: New Speed Records for Saber and NTRU on Cortex-M4 and AVX2. TCHES pp. 159–188 (Feb 2021). https://doi.org/10.46586/tches.v2021.i2.159-188
9. Cook, S.A., Aanderaao, S.O.: On the Minimum Computation Time of Functions. Transactions of the American Mathematical Society **142**, 291–314 (1969)

10. Cooley, J.W., Tukey, J.W.: An Algorithm for the Machine Calculation of Complex Fourier Series. Mathematics of computation **19**(90), 297–301 (1965)
11. Danba, O.: Optimizing NTRU Using AVX2. Master's Thesis, Radboud University, Nijmegen, Netherlands (Jul 2019)
12. Fujisaki, E., Okamoto, T.: Secure Integration of Asymmetric and Symmetric Encryption Schemes. Journal of Cryptology **26**(1), 80–101 (Jan 2013). https://doi.org/10/bxwqr4
13. Gentleman, W.M., Sande, G.: Fast Fourier Transforms: For fun and profit. In: Fall Joint Computer Conference, AFIPS '66. pp. 563–578. ACM Press, San Francisco, CA (Nov 1966). https://doi.org/10.1145/1464291.1464352
14. Gupta, N., Jati, A., Chauhan, A.K., Chattopadhyay, A.: PQC Acceleration Using GPUs: FrodoKEM, NewHope, and Kyber. IEEE Trans. Parallel Distrib. Syst. **32**(3), 575–586 (Mar 2021). https://doi.org/10.1109/TPDS.2020.3025691
15. Hoang, G.L.: Optimization of the NTT Function on ARMv8-A SVE. Bachelor's Thesis, Radboud University, The Netherlands (Jun 2018)
16. Kannwischer, M.J., Rijneveld, J., Schwabe, P., Stoffelen, K.: Pqm4 - Post-quantum crypto library for the {ARM} {Cortex-M4}. https://github.com/mupq/pqm4 (2019)
17. Karatsuba, A., Ofman, Y.: Multiplication of many-digital numbers by automatic computers. Doklady Akademii Nauk SSSR **145**(2), 293–294 (1962)
18. Karmakar, A., Bermudo Mera, J.M., Sinha Roy, S., Verbauwhede, I.: Saber on ARM. IACR Transactions on Cryptographic Hardware and Embedded Systems **2018**(3), 243–266 (Aug 2018). https://doi.org/DOI:10.13154/tches.v2018.i3.243-266
19. Mansouri, F.: On The Parallelization Of Integer Polynomial Multiplication. Master's Thesis, The University of Western Ontario (2014)
20. Scott, M.: A Note on the Implementation of the Number Theoretic Transform. In: O'Neill, M. (ed.) Cryptography and Coding. IMACC 2017. Lecture Notes in Computer Science, vol. 10655, pp. 247–258. Springer International Publishing, Cham (2017). https://doi.org/10.1007/978-3-319-71045-7$_1$3
21. Seiler, G.: Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography. Cryptology ePrint Archive 2018/039 (Jan 2018)
22. Sinha Roy, S.: SaberX4: High-Throughput Software Implementation of Saber Key Encapsulation Mechanism. In: 2019 IEEE 37th International Conference on Computer Design (ICCD). pp. 321–324. IEEE, Abu Dhabi, United Arab Emirates (Nov 2019). https://doi.org/10.1109/ICCD46524.2019.00050
23. Streit, S., De Santis, F.: Post-Quantum Key Exchange on ARMv8-A: A New Hope for NEON Made Simple. IEEE Transactions on Computers **67**(11), 1651–1662 (Nov 2018). https://doi.org/10/gff3sc
24. Terpstra, D., Jagode, H., You, H., Dongarra, J.: Collecting Performance Data with PAPI-C. In: Tools for High Performance Computing 2009, pp. 157–173. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
25. Toom, A.: The complexity of a scheme of functional elements realizing the multiplication of integers. Soviet Mathematics Doklady **3**, 714–716 (1963)
26. Westerbaan, B.: When to Barrett reduce in the inverse NTT. Cryptology ePrint Archive 2020/1377 (Nov 2020)
27. Zhou, S., Xue, H., Zhang, D., Wang, K., Lu, X., Li, B., He, J.: Preprocess-then-NTT Technique and Its Applications to Kyber and NewHope. In: International Conference on Information Security and Cryptology, Inscrypt 2018. LNCS, vol. 11449, pp. 117–137. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-14234-6$_7$