Pushing the Limit of Vectorized Polynomial Multiplications for NTRU Prime

Vincent Hwang(✉)

Max Planck Institute for Security and Privacy, Bochum, Germany
vincentvbh7@gmail.com

Abstract. We conduct a systematic examination of vector arithmetic for polynomial multiplications in software. Vector instruction sets and extensions typically specify a fixed number of registers, each holding a power-of-two number of bits, and support a wide variety of vector arithmetic on registers. Programmers then try to align mathematical computations with the vector arithmetic supported by the designated instruction set or extension. We delve into the intricacies of this process for polynomial multiplications. In particular, we introduce "vectorization-friendliness" and "permutation-friendliness", and review "Toeplitz matrix-vector product" to systematically identify suitable mappings from homomorphisms to vectorized implementations.

To illustrate how the formalization works, we detail the vectorization of polynomial multiplication in $\mathbb{Z}_{4591}[x]/\left\langle x^{761} - x - 1\right\rangle$ used in the parameter set `sntrup761` of the NTRU Prime key encapsulation mechanism.

For practical evaluation, we implement vectorized polynomial multipliers for the ring $\mathbb{Z}_{4591}[x]/\left\langle \Phi_{17}\left(x^{96}\right)\right\rangle$ with AVX2 and Neon. We benchmark our AVX2 implementation on Haswell and Skylake and our Neon implementation on Cortex-A72 and the "Firestorm" core of Apple M1 Pro. Our AVX2-optimized implementation is $1.99-2.16$ times faster than the state-of-the-art AVX2-optimized implementation by [Bernstein, Brumley, Chen, and Tuveri, USENIX Security 2022] on Haswell and Skylake, and our Neon-optimized implementation is $1.29-1.36$ times faster than the state-of-the-art Neon-optimized implementation by [Hwang, Liu, and Yang, ACNS 2024] on Cortex-A72 and Apple M1 Pro.

For the overall scheme with AVX2, we reduce the batch key generation cycles (amortized with batch size 32) by 7.9%–12.0%, encapsulation cycles by 7.1%–10.3%, and decapsulation cycles by 10.7%–13.3% on Haswell and Skylake. For the overall performance with Neon, we reduce the encapsulation cycles by 3.0%–6.6% and decapsulation cycles by 12.8%–15.1% on Cortex-A72 and Apple M1 Pro.

Keywords: Vectorization · Polynomial Multiplication · Fast Fourier Transform · NTRU Prime

1 Introduction

At PQCrypto 2016, the National Institute of Standards and Technology called for post-quantum cryptography to replace existing public-key cryptography due to the discovery of polynomial time quantum algorithms solving integer factorization and discrete logarithm [21,23]. Among the candidates, lattice-based cryptosystems are the most popular due to their balanced key sizes, ciphertext size, signature size, and performance cycles. In many lattice-based cryptosystems, polynomial multiplication is one of the dominating operations for the performance cycles. While constructing cryptosystems, cryptographers choose between various polynomial rings to balance between performance cycles and various security notions. This paper aims to improve public understanding of the interactions between the uses of vector arithmetic and the algebraic aspects of the polynomial rings.

Vector instruction sets and extensions typically specify a fixed number of vector registers, each holding power-of-two number of bits, and support a variety of vector arithmetic operating on these registers. Programmers then try to map the mathematical computations to strings of vector arithmetic supported by the target instruction set or extension. We thoroughly investigate this process for polynomial multiplications in lattice-based cryptosystems. There are two questions we wish to answer in this paper:

1. Why homomorphisms defined on polynomial rings with power-of-two-multiple number of coefficients are frequently assumed to admit efficient vectorization processes?
2. Which homomorphisms are suitable for vectorization?

We answer the first question as follows. In a vector instruction set or extension, there are usually component-wise addition, subtraction, multiplication and variants. We formalize the notion **vectorization-friendliness** and explain why homomorphisms resulting in small-dimensional power-of-two size polynomial multiplications can be suitably mapped to component-wise arithmetic. After decomposing a large problem into several small problems, we divide vector instruction sets and extensions into two groups by the presence of vector-by-scalar multiplication instructions. An instruction is called vector-by-scalar multiplication instruction if it multiplies all the components of a vector by a scalar and returns a vector of elements. If there are vector-by-scalar multiplication instructions, we explain that if the remaining polynomial multiplications are **Toeplitz matrix-vector products**, then vectorization-friendliness suffices to justify suitable vectorization of the overall transformation. On the other hand, if there are no vector-by-scalar multiplication instructions, we formalize the notion **permutation-friendliness** and relate it to the power-of-two nature of the number of subproblems.

For the second question, an evident example is the radix-2 Cooley–Tukey fast Fourier transformation (FFT). Recent work [5] showed that radix-2 Schönhage's and Nussbaumer's FFTs, built upon the power-of-two cyclotomic polynomial moduli, are convenient ones when radix-2 Cooley–Tukey FFT cannot be defined over the native coefficient ring. However, Schönhage's and Nussbaumer's FFTs double the number of coefficients for each application, eventually leading to more small-dimensional polynomial multiplications than the traditional Cooley–Tukey FFT. More recently, [14] proposed Rader's FFT for large Fermat-prime-size transformation and radix-2 Bruun's FFT for the small-dimensional transformation and removed the growth of the number of coefficients under the vectorization context. The downside is that the Fermat-prime-size transformation from Rader's FFT does not nicely align with the power-of-two nature of vectorization. We identify that truncated Rader's FFT over Fermat-prime-size cyclotomic polynomial moduli, previously used for computing the norm of an abelian extension with prime conductor [3, Sect. 4.8], is a suitable one for vectorization due to the power-of-two nature of the transformation size.

Contributions. We summarize our contributions as follows.

- We formalize vectorization-friendliness capturing the nature of component-wise arithmetic supported by a vector instruction set or extension.
- If there are vector-by-scalar multiplication instructions, we explain that vectorization-friendly transformations resulting in small-dimensional Toeplitz matrix-vector products are suitable for vectorization.
- On the other hand, if there are no vector-by-scalar multiplication instructions, we formalize permutation-friendliness capturing the power-of-two nature of the number of subproblems.
- We implement our polynomial multipliers in AVX2 and Armv8.0-A Neon for the ring $\mathbb{Z}_{4591}[x]/\left\langle \Phi_{17}\left(x^{96}\right)\right\rangle$ implementing polynomial multiplications in the NTRU Prime parameter set `sntrup761`.
- For the polynomial multiplication, our AVX2 implementation outperforms the state-of-the-art AVX2-optimized implementation from [5] by 1.99× on Haswell and 2.16× on Skylake, and our Neon implementation outperforms the state-of-the-art Neon-optimized implementation from [14] by 1.29× on Cortex-A72 and 1.36× on Apple M1 Pro.
- For the overall scheme, we integrate our AVX2 implementation into the package `libsntrup761` provided by [5] for the batch key generation and the package `supercop` for encapsulation and decapsulation. We reduce the amortized cycles of batch key generation (with batch size 32) by 7.9%–12.0%, encapsulation cycles by 7.1%–10.3%, and decapsulation cycles by 10.7%–13.3% on Haswell and Skylake. As for our Neon implementation, we integrate our Neon code into the artifact provided by [14]. Our Neon implementation reduces encapsulation cycles by 3.0%–6.6% and decapsulation cycles by 12.8%–15.1% on Cortex-A72 and Apple M1 Pro.

Related Works. There is a long list of works related to vectorization and its challenges. The most relevant one is SPIRAL by [13]. They had attempted to formalize the vectorization of FFTs for code generation. However, SPIRAL falls short to cover transformations used in this paper and we believe this paper will give more insights on extending SPIRAL. Regarding polynomial multiplications for NTRU Prime, [1,2,4] discussed polynomial multiplications when one of the operands has coefficients drawn from $\{0, \pm 1\}$, [1] discussed the generic polynomial multiplication with fairly limited support of vectorization, [5,14] discussed the generic polynomial multiplication with high-dimensional vectorization support.

Code. Our source code is publicly available at
https://github.com/vector-polymul-ntru-ntrup/NTRU_Prime_truncation
under CC0 license.

Structure of this Paper. This paper is structured as follows. Section 2 goes through the preliminaries. Section 3 formalizes the vectorization process, Sect. 4 gives a walkthough on vectorizing polynomial multiplications for `sntrup`. Finally, Sect. 5 shows the performance with AVX2 on Haswell and Skylake, and with Neon on Cortex-A72 and Apple M1 Pro.

2 Preliminaries

2.1 Streamlined NTRU Prime

NTRU Prime [4] is an alternate candidate of key encapsulation mechanism (KEM) in the 3rd round of NIST Post-Quantum Cryptography (PQC) Standardization and is currently used in OpenSSH 9.0 hybrid `sntrup761x25519-sha512` key exchange by default[1]. NTRU prime KEM [4] operates over the polynomial rings $\mathbb{Z}_q[x]/\langle x^p - x - 1\rangle$ and $\mathbb{Z}_3[x]/\langle x^p - x - 1\rangle$ for primes p and q such that $\mathbb{Z}_q[x]/\langle x^p - x - 1\rangle \cong \mathbb{F}_{q^p}$. There are two cryptosystems built upon $\mathbb{Z}_q[x]/\langle x^p - x - 1\rangle$ and $\mathbb{Z}_3[x]/\langle x^p - x - 1\rangle$ – Streamlined NTRU Prime (`sntrup`) and NTRU LPRime (`ntrulpr`). This paper focuses on polynomial multiplications in `sntrup761` with $(p, q) = (761, 4591)$ and the implementations can be straightforwardly ported into `ntrulpr761`. See [4] for more details of the scheme. In the following, we list all the polynomial multiplications and inversions required for `sntrup`.

Key generation: We need one inversion in $\mathbb{Z}_q[x]/\langle x^p - x - 1\rangle$ and one inversion with invertibility check in $\mathbb{Z}_3[x]/\langle x^p - x - 1\rangle$ for the secret key, and one polynomial multiplication in $\mathbb{Z}_q[x]/\langle x^p - x - 1\rangle$ for the public key.

Encapsulation: We need one polynomial multiplication in $\mathbb{Z}_q[x]/\langle x^p - x - 1\rangle$ for encryption.

Decapsulation: We need one polynomial multiplication in $\mathbb{Z}_q[x]/\langle x^p - x - 1\rangle$ for encryption, and one polynomial multiplication in $\mathbb{Z}_q[x]/\langle x^p - x - 1\rangle$ and one polynomial multiplication in $\mathbb{Z}_3[x]/\langle x^p - x - 1\rangle$ for decryption.

We focuses on polynomial multiplications and inversions in $\mathbb{Z}_q[x]/\langle x^p - x - 1\rangle$. We call a polynomial multiplication “big-by-small” if one of the operands is drawn from $\{0, \pm 1\}$ and “big-by-big” otherwise. Our polynomial multipliers target big-by-big ones and also covers the big-by-small ones

[1] https://marc.info/?l=openssh-unix-dev&m=164939371201404&w=2.

by definition. For encapsulation and decapsulation, we only need big-by-small polynomial multiplications. For the key generation, we only need big-by-small polynomial multiplication outside the inversion. As for the inversion, the requirement of polynomial multiplications heavily depends on the choice of approach. We simply focus on the divstep approach avoiding any polynomial multiplications and leave the incorporation of jumpdivstep [7] as future work.

To see why big-by-big polynomial multiplications are important, we review Montgomery's trick for batch inversion used in batch key generation [5]. Let's say we want to invert polynomials $\boldsymbol{a}_0, \dots, \boldsymbol{a}_{n-1}$ in $\mathbb{Z}_q[x]/\langle x^p - x - 1 \rangle$. Instead of inverting each of them one at a time, we first compute $\boldsymbol{a}_0, \boldsymbol{a}_0\boldsymbol{a}_1, \dots, \prod_{i=0,\dots,n-1} \boldsymbol{a}_i$ with $n-1$ polynomial multiplications, and invert $\prod_{i=0,\dots,n-1} \boldsymbol{a}_i$. We now compute $\left(\prod_{i=0,\dots,n-1} \boldsymbol{a}_i\right)^{-1}, \left(\prod_{i=0,\dots,n-2} \boldsymbol{a}_i\right)^{-1}, \dots, \boldsymbol{a}_0^{-1}$ with $n-1$ polynomial multiplications. Finally, we iterate over $j = 1, \dots, n-1$ and compute the inverses as $\boldsymbol{a}_j^{-1} = \left(\prod_{i=0,\dots,j} \boldsymbol{a}_i\right)^{-1} \left(\prod_{i=0,\dots,j-1} \boldsymbol{a}_i\right)$. In `sntrup`, since all polynomials to be inverted have coefficients in $\{0, \pm 1\}$, we need $2n-2$ big-by-small polynomial multiplications, $n-1$ big-by-big polynomial multiplications, and one inversion.

2.2 Basics of Algebra

We first go through some basic notations and definitions of algebraic structures for this paper. Readers familiar with modules and associative algebras are free to skip this section and treat this section as a reference. We assume that readers are all familiar with monoids, groups, rings, and modules and refer to standard algebra books [9,16,17] for reference. In this paper, all rings are commutative and unital. Below we go through a short introduction of free modules and associative algebras over a commutative ring R.

Modules. The central idea of this paper revolves around free-module homomorphisms and their tensor products. A module is a generalization of a vector space where the underlying ground field is relaxed to a ring. We only consider a special kind of modulues – free modulues of finite ranks. In other words, all modulues in this paper are of the form R^n for a ring R and a positive integer n, and all elements can be written as a finite sum of the form $\sum_i r_i e_i$ where e_i is the element with ith element one and zero elsewhere for all i. Given two free modules R^n and R^m, we define **the tensor product of R^n and R^m** as the free module consisting of all the elements of the form $\sum_i \boldsymbol{a}_i \otimes \boldsymbol{b}_i$. where $\boldsymbol{a}_i \in R^n$ and $\boldsymbol{b}_i \in R^m$ up to certain equivalences.

Suppose we have module homomorphisms $f : R^n \to R^n$ and $g : R^m \to R^m$. We define the **tensor product** $f \otimes g : R^n \otimes R^m \to R^n \otimes R^m$ **of** f **and** g as

$$x \otimes y \mapsto f(x) \otimes g(y).$$

Recall that module homomorphism between modules of finite ranks can be written as matrix multiplications if we specify the bases. Suppose we have bases

$\{e_i\} \subset R^n$ and $\{\tilde{e}_j\} \subset R^m$. Then, $\{e_i \otimes \tilde{e}_j\}$ is a basis of $R^n \otimes R^m$. One can show that the matrix form of $f \otimes g$ with the basis $\{e_i \otimes \tilde{e}_j\}$ is the same as the tensor product of the matrix forms of f with $\{e_i\}$ and g with $\{\tilde{e}_j\}$.

By unfolding the definition of a tensor product, we have

$$\forall f_0, f_1 : R^n \to R^n, \forall g_0, g_1 : R^m \to R^m, (f_0 \circ f_1) \otimes (g_0 \circ g_1) = (f_0 \otimes g_0) \circ (f_1 \otimes g_1)$$

where $\circ$ is the function composition. An example that we will frequently encounter in this paper is the case $g_0 = g_1 = \mathrm{id}_m$, the identity map of R^m. Suppose we have a factorization for $f : R^n \to R^n$ with $f = f_0 \circ f_1$, then we also have

$$f \otimes \mathrm{id}_m = (f_0 \circ f_1) \otimes (\mathrm{id}_m \circ \mathrm{id}_m) = (f_0 \otimes \mathrm{id}_m) \circ (f_1 \otimes \mathrm{id}_m).$$

In general, if f factors into $f_0 \circ \cdots \circ f_{d-1}$, then $f \otimes \mathrm{id}_m = (f_0 \otimes \mathrm{id}_m) \circ \cdots \circ (f_{d-1} \otimes \mathrm{id}_m)$.

Associative Algebras. For an R-module M, if we adjoin a ring structure to M by introducing a binary associative operator with an identity compatible with 1_R to the underlying additive group M, we call M an **associative** R**-algebra**. For simplicity, we call an associative R-algebra an R**-algebra** or an **algebra** when the context is clear. For a degree-n polynomial $\boldsymbol{g} \in R[x]$, the quotient ring $R[x]/\langle \boldsymbol{g} \rangle$ is an R-algebra since (i) $R[x]/\langle \boldsymbol{g} \rangle$ is a ring and (ii) $R[x]/\langle \boldsymbol{g} \rangle = R^n$ as R-modules by specifying $x^i = \left(\underbrace{0, \ldots, 0}_{i}, 1, \underbrace{0, \ldots, 0}_{n-1-i} \right)$. Suppose $\boldsymbol{g} = \boldsymbol{g}(x^v)$ for a positive integer v, we have $R[x]/\langle \boldsymbol{g}(x^v) \rangle \cong \mathcal{R}[y]/\langle \boldsymbol{g}(y) \rangle$ where $\mathcal{R} \coloneqq R[x]/\langle x^v - y \rangle$. The crucial point is to interpret an $\mathcal{R}$-algebra homomorphism $f_{\mathcal{R}}$ for $\mathcal{R}[y]/\langle \boldsymbol{g}(y) \rangle$ as an R-algebra homomorphism for $R[x]/\langle \boldsymbol{g}(x^v) \rangle$. We claim that $f_{\mathcal{R}} \otimes \mathrm{id}_v$ is the desired R-algebra homomorphism. Similarly, if we have a factorization of an $\mathcal{R}$-algebra homomorphism $f = f_0 \circ f_1$ for $\mathcal{R}[y]/\langle \boldsymbol{g}(y) \rangle$, we have a composition of R-algebra homomorphisms $f_0 \otimes \mathrm{id}_v$ and $f_1 \otimes \mathrm{id}_v$ for $R[x]/\langle \boldsymbol{g}(x^v) \rangle$.

2.3 Vector Arithmetic

We go through the vector instruction set/extension covered in this paper.

AVX2. Advanced vector extension 2 (AVX2) is a vector extension to the x86 instruction architecture. In AVX2, there are 16 `ymm` registers each holding 256 bits of data. In this paper, we only consider 16-bit arithmetic and regard each vectors as packed 16-bit elements. Furthermore, we also have several permutation instructions with two data operands. Frequently, a series of permutation instructions are used for implementing a certain kind of permutation matrices.

Armv8.0-A Neon. The instruction set architecture Armv8.0-A comes with the vector extension Neon. In Neon, there are 32 vector registers (`v0` to `v31`) each holding 128 bits of data. In addition to vector-by-vector multiplication instructions, we have vector-by-scalar multiplications multiplying a vector of elements by a scalar. Similar to AVX2, there is a wide variety of permutation instructions. One of the convenient ones is `ext`: we concatenate two 128-bit vector registers and extract a certain contiguous 16-byte data from the 32-byte data. This allows us to implement cyclic shifts of tuples in a convenient way.

2.4 Cooley–Tukey FFT

Let $n = \prod_j n_j$, and i_j runs over $0, \dots, n_j - 1$ for each j. The Cooley–Tukey FFT [12] computes $R[x]/\langle x^n - \zeta^n \rangle \cong \prod_{i_0,\dots,i_{h-1}} R[x]\Big/\Big\langle x - \zeta\omega_n^{\sum_l i_l \prod_{j<l} n_j} \Big\rangle$ in a layer-by-layer fashion where ω_n is a principal n-th root of unity[2]. The simplest case is the isomorphism $R[x]\Big/\Big\langle x^{2^h} - 1 \Big\rangle \cong \prod_{i_0,\dots,i_{h-1}} R[x]\Big/\Big\langle x - \omega_{2^h}^{\sum_l i_l 2^l} \Big\rangle$. However, we will encounter various transformations built upon non-power-of-two Cooley–Tukey FFTs.

2.5 Good–Thomas FFT

Good–Thomas FFT is an alternative FFT built upon the coprime factors of the transformation size n. We explain the idea briefly with the smallest case $n = 6$. Consider the cyclic transformation $\mathbf{F}_6 : R[x]/\langle x^6 - 1 \rangle \to \prod_i R[x]/\langle x - \omega_6^i \rangle$, If we perform pre- and post-permutation for the 1st and the 4th element (we start with 0), and define $\omega_3 \coloneqq \omega_6^4, \omega_2 \coloneqq \omega_6^3$, we have $P_{(14)}\mathbf{F}_6 P_{(14)} = \big(\mathbf{F}_2 \otimes I_3\big)\big(I_2 \otimes \mathbf{F}_3\big)$ where $\mathbf{F}_2$ and $\mathbf{F}_3$ are cyclic transformation of sizes 2 and 3, respectively. Comparing to Cooley–Tukey FFT, we save two multiplications by ω_6 and ω_6^2.

2.6 Truncated Rader's FFT and Its Inverse

Let p be an odd prime, and $\mathcal{I} = \{0, \dots, p-1\}, \mathcal{I}^* = \{1, \dots, p-1\}$ be index sets. Rader's FFT [22] computes the map $R[x]/\langle x^p - 1 \rangle \cong \prod_{i \in \mathcal{I}} R[x]/\langle x - \omega_p^i \rangle$ with a size-$\lambda(p)$ cyclic convolution where λ is the Carmichael's lambda function. Due to the page limit, we refer to [22] for the original version and jump into the truncated version introduced by [3].

Let Φ_p be the p-th cyclotomic polynomial. Since p is a prime, we have $\Phi_p(x) = \sum_{i \in \mathcal{I}} x^i$ and $\Phi_p(x) | (x^p - 1)$. A natural question is to build an efficient transformation for $R[x]/\langle \Phi_p(x) \rangle$ from the Rader's FFT for $R[x]/\langle x^p - 1 \rangle$. We start with isomorphism $\sum_{i \in \mathcal{I}^*} a_{i-1} x^{i-1} \mapsto \Big(\hat{a}_j = \sum_{i \in \mathcal{I}^*} a_{i-1} \omega_p^{(i-1)j}\Big)_{j \in \mathcal{I}^*}$: $R[x]/\langle \Phi_p(x) \rangle \to \prod_{j \in \mathcal{I}^*} R[x]/\langle x - \omega_p^j \rangle$, and reindex with $i \mapsto -\log_g i$ and $j \mapsto \log_g j$. We have

$$\hat{a}_{g^{\log_g j}} = \sum_{i \in \mathcal{I}^*} a_{i-1} \omega_p^{(i-1)j} = \omega_p^{-j} \sum_{-\log_g i \in \mathbb{Z}_{\lambda(p)}} a_{g^{\log_g i} - 1} \omega_p^{g^{\log_g i + \log_g j}}$$

[2] $\forall j = 1, \dots, n-1, \sum_{i=0}^{n-1} \omega_n^{ij} = 0$.

and find that $\left(\omega_p^k \hat{a}_{g^k}\right)_{k\in\mathbb{Z}_{\lambda(p)}}$ is the convolution of $\left(a_{g^{-k}-1}\right)_{k\in\mathbb{Z}_{\lambda(p)}}$ and $\left(\omega_p^{g^k}\right)_{k\in\mathbb{Z}_{\lambda(p)}}$. This is called the **truncated Rader's FFT**. Below is an illustration for $p = 5$ and $g = 2$: $P_{(23)} \begin{pmatrix} \omega_5 & \omega_5^2 & \omega_5^3 & \omega_5^4 \\ \omega_5^2 & \omega_5^4 & \omega_5 & \omega_5^3 \\ \omega_5^3 & \omega_5 & \omega_5^4 & \omega_5^2 \\ \omega_5^4 & \omega_5^3 & \omega_5^2 & \omega_5 \end{pmatrix} P_{(312)} = \begin{pmatrix} \omega_5^{2^0} & \omega_5^{2^3} & \omega_5^{2^2} & \omega_5^{2^1} \\ \omega_5^{2^1} & \omega_5^{2^0} & \omega_5^{2^3} & \omega_5^{2^2} \\ \omega_5^{2^2} & \omega_5^{2^1} & \omega_5^{2^0} & \omega_5^{2^3} \\ \omega_5^{2^3} & \omega_5^{2^2} & \omega_5^{2^1} & \omega_5^{2^0} \end{pmatrix}$.

For the inverse, [3, Sect. 4.8.2] showed how to implement it with a size-$\lambda(p)$ cyclic convolution. They found that convoluting with $\frac{1}{p}\left(\omega_p^{-g^{-k}} - 1\right)_{k\in\mathbb{Z}_{\lambda(p)}}$ results in the desired inversion. We illustrate below for $p = 5$ and $g = 2$:

$$\begin{pmatrix} \omega_5^{2^0} & \omega_5^{2^3} & \omega_5^{2^2} & \omega_5^{2^1} \\ \omega_5^{2^1} & \omega_5^{2^0} & \omega_5^{2^3} & \omega_5^{2^2} \\ \omega_5^{2^2} & \omega_5^{2^1} & \omega_5^{2^0} & \omega_5^{2^3} \\ \omega_5^{2^3} & \omega_5^{2^2} & \omega_5^{2^1} & \omega_5^{2^0} \end{pmatrix} \begin{pmatrix} \omega_5^{-2^{-0}} - 1 \\ \omega_5^{-2^{-1}} - 1 \\ \omega_5^{-2^{-2}} - 1 \\ \omega_5^{-2^{-3}} - 1 \end{pmatrix} = \begin{pmatrix} 5 \\ 0 \\ 0 \\ 0 \end{pmatrix}.$$

In summary, we implement η^{-1} by mapping $(\hat{a}_{g^k})_{k\in\mathbb{Z}_{\lambda(p)}}$ to $\left(\omega_p^k \hat{a}_{g^k}\right)_{k\in\mathbb{Z}_{\lambda(p)}}$ and convoluting with $\left(\omega_p^{-g^{-k}} - 1\right)_{k\in\mathbb{Z}_{\lambda(p)}}$. Scaling by $\frac{1}{p}$ is postponed to the end. See [3, Sects. 4.12.3 and 4.12.4] for a generalization to arbitrary p.

2.7 Bruun's FFT

Let q be a prime with $q \equiv 3 \bmod 4$ and $q + 1 = r2^w$ for an odd r. Bruun's FFT allows us to split $\mathbb{Z}_q[x]/\left\langle x^{2^w} + 1\right\rangle$ into $\prod_i \frac{\mathbb{Z}_q[x]}{\langle x^2 \pm \alpha_i x - 1\rangle}$. See [8] for a proof. For $q = 4591$, we can split $\mathbb{Z}_q[x]/\left\langle x^{16} + 1\right\rangle$ into size-2 polynomial rings with moduli of the form $x^2 \pm \alpha_i x - 1$ since $4591 + 1 = 287 \cdot 2^4$. In this paper, we are interested in the case $\mathbb{Z}_q[x]/\left\langle x^{16} + 1\right\rangle \cong \prod \mathbb{Z}_q[x]/\left\langle x^8 \pm \sqrt{2}x^4 + 1\right\rangle$. See [14, Sect. 3.3] for its implementation.

Bruun's FFT was originally proposed with $\mathbb{C}$ as the coefficient ring. See [10] for the power-of-two case and [20] for the even case.

2.8 Twisting

Let R be a ring, $\zeta \in R$ be an invertible element, n be an integer, and $\xi \in R$ be an element. We have the isomorphism $R[x]/\langle x^n - \xi\zeta^n\rangle \cong R[y]/\langle y^n - \xi\rangle$ by sending x to ζy. This is called twisting. Obviously, twisting amounts to multiplying all the coefficients by certain constants and its transformation matrix is a diagonal matrix. In the literature, twising is commonly specialized to $\xi = 1$, but we need the cases $\xi = \pm 1$ in this paper.

2.9 Karatsuba

Karatsuba [18] computes the product $(a_0 + a_1 x)(b_0 + b_1 x)$ by evaluating at the point set $\{0, 1, \infty\}$. We compute $(a_0 + a_1 x)(b_0 + b_1 x) = a_0 b_0 + (a_0 b_1 + a_1 b_0)x +$

$a_1b_1x^2$ with three multiplications a_0b_0, a_1b_1, and $(a_0+a_1)(b_0+b_1)$ by observing $a_0b_1+a_1b_0=(a_0+a_1)(b_0+b_1)-a_0b_0-a_1b_1$.

3 Formalization of Vectorization

We formalize "vectorization-friendliness" and "permutation-friendliness", and review the role of Toeplitz matrix-vector products in vectorization. While computing with vector instructions, we choose algebra homomorphisms f and g such that f is vectorization-friendly and g is permutation-friendly or amounts to computing Toeplitz matrices when there are vector-by-scalar multiplication instructions. Their composition $g \circ f$ then admits suitable mapping to vector arithmetic.

Section 3.1 formalizes vectorization-friendliness capturing the power-of-two nature of vector registers, Sect. 3.2 formalizes permutation-friendliness capturing the permutation nature, and Sect. 3.3 reviews small-dimensional Toeplitz matrix-vector products.

3.1 Vectorization–Friendliness

Conceptually, we call an algebra homomorphism vectorization-friendly if we can factor it into module homomorphisms with matrix forms of certain kinds of block diagonal matrices or tensor products with I_v as the right operand. We first identify a set of matrices that can be implemented efficiently with vector instructions straightforwardly. Let v' be a multiple of v. We define `BlockDiag` as the set of all block diagonal matrices with each block a $v' \times v'$ matrix of the following form:

1. Diagonal matrix: a matrix with non-diagonal entries all zeros.
2. Cyclic/negacyclic shift matrix: a matrix implementing $(a_i) \mapsto \left(a_{(i+c) \bmod v'}\right)$ (cyclic) or $(a_i) \mapsto \left((-1)^{[\![i+c \geq v']\!]} a_{(i+c) \bmod v'}\right)$ (negacyclic) for a non-negative integer c.

Diagonal matrices are suitable for vectorization since we can load v coefficients, multiply them by v constants, and store them back to memory with vector instructions. For cyclic/negacyclic shift matrices, we discuss how to implement them for the following vector instruction sets:

– Armv7/8-A Neon: For cyclic shifts, we use the instruction `ext` extracting consecutive elements from a pair of vector registers. We negate one of the registers before applying `ext` for negacyclic shifts [14].
– AVX2: For cyclic shifts, we perform unaligned loads, shuffle the last vector register, and store the vectors to memory. Again, the last vector register is negated for negacyclic shifts [5].

Let f be an algebra monomorphism, and M_f be the matrix form of f. We call f **vectorization-friendly** if

$$M_f = \prod_i \left(M_{f_i} \otimes I_v\right) S_{f_i}$$

for some M_{f_i} and $S_{f_i} \in \texttt{BlockDiag}$. The tensor product $M_{f_i} \otimes I_v$ ensures that each v-chunk is regarded as a whole while applying $M_{f_i} \otimes I_v$. Additionally, f is vectorization-friendly if and only if f^{-1} is vectorization-friendly, so we only need to discuss the vectorization-friendliness of a monomorphism and its inverse follows immediately.

3.2 Permutation–Friendliness

We introduce the notion "permutation-friendliness". Conceptually, permutation-friendliness stands for vectorization-friendliness after applying a special type of permutation—interleaving. Again, let v' be a multiple of v. We define the transposition matrix $T_{v'^2}$ as the $v'^2 \times v'^2$ matrix permuting the elements as if transposing a $v' \times v'$ matrix. Now we are ready to specify the set `Interleave` of interleaving matrices. We call a matrix M interleaving matrix with step v' if it takes the form

$$M = \left(\pi' \otimes I_{v'}\right)\left(I_m \otimes T_{v'^2}\right)\left(\pi \otimes I_{v'}\right)$$

for a positive integer m and permutation matrices π, π' permuting mv' elements. The set `Interleave` consists of interleaving matrices of all possible steps and is closed under inversion.

We call an algebra monomorphism g **permutation-friendly** if we can factor its matrix form M'_g as

$$M'_g = \prod_i S_{g_i} M_{g_i}$$

for $S_{g_i} \in \texttt{Interleave}$ and vectorization-friendly M_{g_i}'s. Immediately, we know that g is permutation-friendly if and only if g^{-1} is permutation-friendly.

3.3 Toeplitz Matrix–Vector Product (Small Dimensional)

We go through an alternative for permutation friendliness when there are vector-by-scalar multiplication instructions. Suppose we have a vectorization-friendly monomorphism resulting several small-dimensional power-of-two-size cyclic/negacyclic convolutions. By the definition of vectorization-friendliness, a cyclic/negacyclic convolution can be phrased as applying a $v' \times v'$ Toeplitz matrix to a vector for a v-multiple v'. We call a matrix M Toeplitz if $M_{i,j} = M_{i+1,j+1}$ for all possible i, j. Generally, one can write a polynomial multiplication modulo $x^{v'} - \zeta$ as an application of a Toeplitz matrix constructed from one of the operands [6,15,19]. Recently, [11] decomposed the application of a $v' \times v'$ Toeplitz matrix as a sum of column-to-scalar multiplications and implemented each with a vector-by-scalar multiplication instruction.

4 Vectorized Polynomial Multipliers

This section describes our polynomial multiplications for $\mathbb{Z}_{4591}[x]/\langle x^{761} - x - 1\rangle$. A standard approach is to multiply in $\mathbb{Z}_{4591}[x]/\langle \boldsymbol{g}\rangle$ with $\deg(\boldsymbol{g}) > 2 \cdot 760$ followed by polynomial reduction modulo $x^{761} - x - 1$. We propose to multiply in $\mathbb{Z}_{4591}[x]/\left\langle \Phi_{17}\left(x^{96}\right)\right\rangle$ where $\deg\left(\Phi_{17}\left(x^{96}\right)\right) = 1536 > 2 \cdot 760$. For simplicity, we assume $R = \mathbb{F}_{4591}$ in this section.

There are two steps for deciding isomorphisms admitting suitable mapping to vector arithmetic. The first step is to find an isomorphism honoring our intuition of the memory layout – we choose an isomorphism dividing a large problem into several subproblems of sizes multiples of v (the number of elements contained in a vector register). Section 4.1 describes our isomorphisms resulting several size-16 subproblems. The second step is to decide isomorphisms computing the remaining task. Section 4.2.1 discusses a permutation-friendly approach and Sect. 4.2.2 discusses our Toeplitz matrix-vector product approach. Finally, we go through a detailed comparisons to existing works with emphases on vectorization-friendliness and permutation-friendliness in Sect. 4.3.

4.1 The Vectorization-Friendly Phase

We first go through the implementation of

$$\frac{R[x]}{\left\langle \Phi_{17}\left(x^{96}\right)\right\rangle} \cong \left(\prod \frac{R[x]}{\langle x^{16} \pm 1\rangle}\right)^{48}.$$

Since the resulting polynomial rings have size 16, our transformation is evidently vectorization-friendly. We detail below the construction and its vectorization-friendliness.

4.1.1 Truncated Rader's FFT

Let $\eta_0 : R^{16} \to R^{16}$ be the module map implementing the permutation and cyclic convolution parts of the truncated size-17 Rader's FFT. $R[x]/\langle \Phi_{17}(x)\rangle \cong \prod_{i=0}^{15} R[x]/\left\langle x - \omega_{17}^{i+1}\right\rangle$ is implemented as $\mathtt{mul}_0 \circ \eta_0$ where $\mathtt{mul}_0 \coloneqq (a_i)_{i=0,\ldots,15} \mapsto \left(\omega_{17}^{-(i+1)} a_i\right)_{i=0,\ldots,15}$. We tensor the composition $\mathtt{mul}_0 \circ \eta_0$ by I_{96} to implement $R[x]/\left\langle \Phi_{17}\left(x^{96}\right)\right\rangle \cong \prod_{i=0}^{15} R[x]/\left\langle x^{96} - \omega_{17}^{i+1}\right\rangle$. We then twist all the rings to the cyclic ones via the product map $\mathtt{twist}_0 \coloneqq \prod_{i=0}^{15}\left(x \mapsto \omega_{17}^{14(i+1)} x\right)$[3]. To sum up, we implement $R[x]/\left\langle \Phi_{17}\left(x^{96}\right)\right\rangle \cong \left(R[x]/\left\langle x^{96} - 1\right\rangle\right)^{16}$ as

$$\mathtt{twist}_0 \circ ((\mathtt{mul}_0 \circ \eta_0) \otimes I_{96})$$

which is obviously vectorization friendly.

[3] Notice that $\omega_{17} = \omega_{17}^{1344} = \left(\omega_{17}^{14}\right)^{96}$.

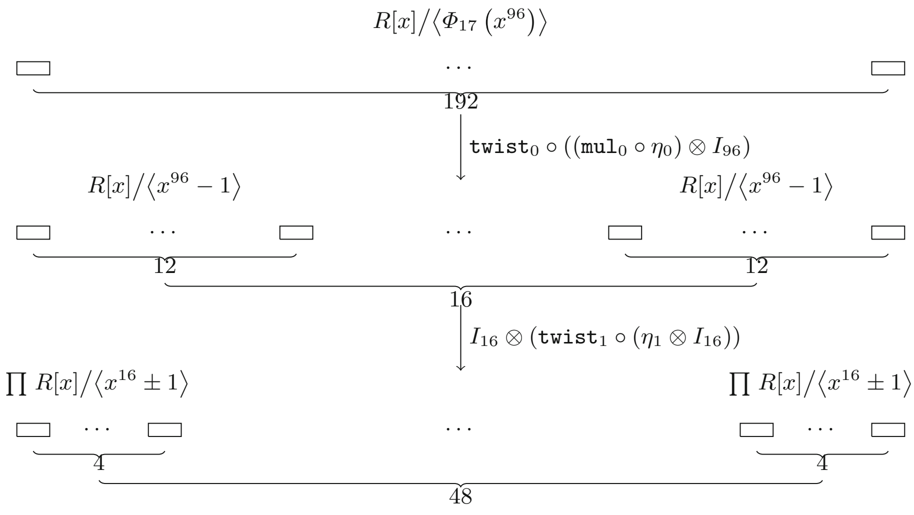

$R[x]/\langle\Phi_{17}(x^{96})\rangle$
192
$\mathtt{twist}_0 \circ ((\mathtt{mul}_0 \circ \eta_0) \otimes I_{96})$
$R[x]/\langle x^{96}-1\rangle$
$R[x]/\langle x^{96}-1\rangle$
12
12
16
$I_{16} \otimes (\mathtt{twist}_1 \circ (\eta_1 \otimes I_{16}))$
$\prod R[x]/\langle x^{16}\pm 1\rangle$
$\prod R[x]/\langle x^{16}\pm 1\rangle$
4
4
48

Fig. 1. Overview of the correspondence between algebraic maps and 128-bit vector register view in Neon. Each rectangles holds $\frac{128}{16} = 8$ coefficients and is loaded to a vector register. Similar justification of vectorization-friendliness holds if we move to 256-bit vector registers in AVX2.

4.1.2 Good–Thomas FFT

Next, we turn the ring $R[x]/\langle x^{96}-1\rangle$ into $\left(\prod R[x]/\langle x^{16}\pm 1\rangle\right)^3$ by applying Good–Thomas FFT and twisting. Let η_1 be the map implementing the Good–Thomas FFT of dimension 3×2, and $\mathtt{twist}_1$ twisting the product ring into $\left(\prod R[x]/\langle x^{16}\pm 1\rangle\right)^3$. Then, $\mathtt{twist}_1 \circ (\eta_1 \otimes I_{16})$ implements $R[x]/\langle x^{96}-1\rangle \cong \left(\prod R[x]/\langle x^{16}\pm 1\rangle\right)^3$. Since there are 16 copies of $R[x]/\langle x^{96}-1\rangle$, we have

$$I_{16} \otimes (\mathtt{twist}_1 \circ (\eta_1 \otimes I_{16})) = (I_{16} \otimes \mathtt{twist}_1) \circ (I_{16} \otimes \eta_1 \otimes I_{16})$$

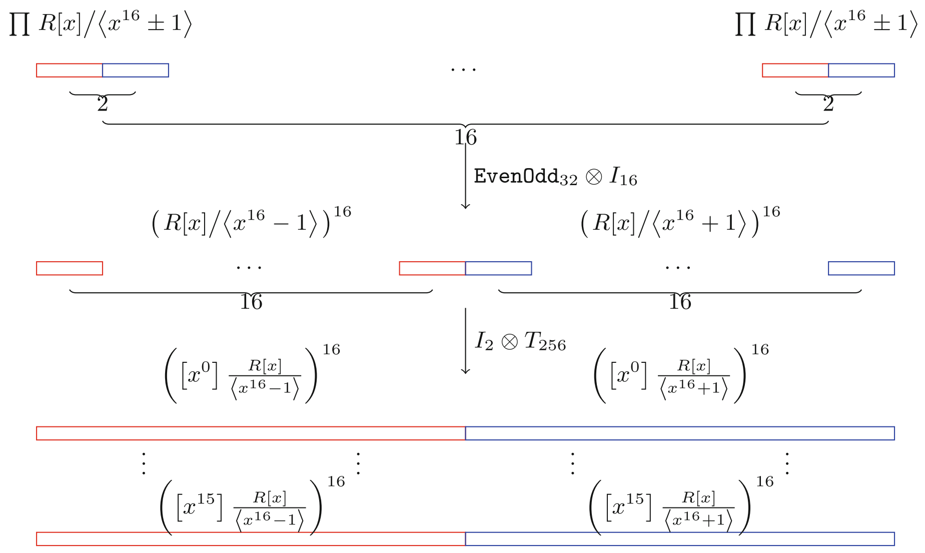
as the overall transformation. Obviously, this is vectorization friendly.

For a more illustrative explanation of how polynomials are mapped to 128-bit registers, we outline the workflow in Fig. 1 where each rectangles represents a 128-bit register. Note that similar justification holds for 256-bit registers since we are right-tensoring by I_{16}.

4.2 Small-Dimensional Cyclic/Negacyclic Convolutions

This section goes through our approaches multiplying in $\left(\prod R[x]/\langle x^{16}\pm 1\rangle\right)^{48}$. We propose two approaches: a permutation-friendly approach for AVX2 and a Toeplitz matrix-vector product approach for Neon.

4.2.1 A Permutation-Friendly Approach



$\prod R[x]/\langle x^{16} \pm 1\rangle$
$\prod R[x]/\langle x^{16} \pm 1\rangle$
2
2
16
$\texttt{EvenOdd}_{32} \otimes I_{16}$
$\left(R[x]/\langle x^{16} - 1\rangle\right)^{16}$
$\left(R[x]/\langle x^{16} + 1\rangle\right)^{16}$
16
16
$I_2 \otimes T_{256}$
$\left([x^0]\,\frac{R[x]}{\langle x^{16}-1\rangle}\right)^{16}$
$\left([x^0]\,\frac{R[x]}{\langle x^{16}+1\rangle}\right)^{16}$
$\left([x^{15}]\,\frac{R[x]}{\langle x^{16}-1\rangle}\right)^{16}$
$\left([x^{15}]\,\frac{R[x]}{\langle x^{16}+1\rangle}\right)^{16}$

Fig. 2. Overview of permutations implementing permutation-friendliness for our AVX2 implementation defined on $\left(R[x]/\langle x^{16} \pm 1\rangle\right)^{16}$. Same idea applies to $\left(R[x]/\langle x^{16} \pm 1\rangle\right)^{48}$ since $48 = 3 \cdot 16$. Each rectangles represents a 16-tuple mapped to a 256-bit vector register in AVX2.

We first go through the permutation-friendly approach used in our AVX2 implementation. Since the goal is to interleave 16 polynomial rings with the same shape of computation, we show how to map the multiplication in $\left(\prod R[x]/\langle x^{16} \pm 1\rangle\right)^{16}$ to vector arithmetic. We perform an even-odd permutation over 16-tuples resulting $\left(R[x]/\langle x^{16} - 1\rangle\right)^{16} \times \left(R[x]/\langle x^{16} + 1\rangle\right)^{16}$ followed by two copies of T_{256}. This gives us the map

$$(I_2 \otimes T_{256})\,(\texttt{EvenOdd}_{32} \otimes I_{16})$$

where $\texttt{EvenOdd}_{32}$ moves the even indices to the first half and the odd indices to the second half. See Fig. 2 for an illustration. The overall interleaving matrix for $\left(\prod \frac{R[x]}{\langle x^{16} \pm 1\rangle}\right)^{48}$ can be written as:

$$(I_6 \otimes T_{256})\,(I_3 \otimes \texttt{EvenOdd}_{32} \otimes I_{16})$$

which is permutation-friendly. Finally, we apply Cooley–Tukey to $R[x]/\langle x^{16} - 1\rangle \cong \prod R[x]/\langle x^8 \pm 1\rangle$ and Bruun to $R[x]/\langle x^{16} + 1\rangle \cong R[x]/\langle x^8 \pm \sqrt{2}x^4 + 1\rangle$ followed by Karatsuba defined over vector registers.

4.2.2 Toeplitz Matrix-Vector Products

Recall that one can phrases polynomial multiplications in $R[x]\Big/\Big\langle x^{v'} \pm 1\Big\rangle$ as Toeplitz matrix-vector products for v' a multiple of v (cf. Sect. 3.3). We describe an alternative approach for multiplying in $\left(\prod R[x]/\langle x^{16} \pm 1\rangle\right)^{48}$ with Neon. Since each vector registers in Neon holds eight coefficients, we first split $R[x]/\langle x^{16} - 1\rangle$ into $\prod R[x]/\langle x^{8} \pm 1\rangle$, and apply Toeplitz matrix-vector multiplications in $R[x]/\langle x^{8} \pm 1\rangle$ and $R[x]/\langle x^{16} - 1\rangle$. The implementations follow analogously from [11].

4.3 Comparisons to Prior Implementations

We compare our vectorized implementations to prior FFT works operating over $R = \mathbb{Z}_{4591}$. Table 1 summarizes the vectorization- and permutation-friendliness of existing polynomial multipliers over R. Table 2 summarizes existing vectorization-friendly approaches with AVX2 and Neon, Table 3 summarizes existing permutation-friendly approaches with AVX2, and Table 4 summarizes existing permutation-friendly and Toeplitz matrix-vector product approaches with Neon.

Table 1. Summary of maximum possible v justifying vectorization- and permutation-friendliness of existing polynomial multipliers over $\mathbb{Z}_{4591}$ for $\mathbb{Z}_{4591}[x]/\langle x^{761} - x - 1\rangle$. `CT` stands for Cooley–Tukey FFT and `GT` stands for Good–Thomas FFT. If the maximum possible v of a transformation is greater or equal to the number of halfwords in a vector register, then the FFT transformation is vectorization-friendly/permutation-friendly for the given ISA/extension.

	[1]	[5]	[14]	This work
ISA/extension	Armv7E-M	AVX2	Neon	Neon/AVX2
# halfword in a vector register	2	16	8	8 / 16
Domain	$\frac{R[x]}{\langle x^{1530}-1\rangle}$	$\frac{R[x]}{\left\langle \frac{x^{2048}-1}{x^{512}+1}\right\rangle}$	$\frac{R[x]}{\langle x^{1632}-1\rangle}$	$\frac{R[x]}{\langle \Phi_{17}(x^{96})\rangle}$
FFT	Rader, `CT`	Schönhage, Nussbaumer	Rader, `GT`	truncated Rader, `GT`
Vectorization-friendly	$v = 2$ (Yes)	$v = 64$ (Yes)	$v = 32$ (Yes)	$v = 32$ (Yes)
Permutation-friendly	$v = 1$ (No)	$v = 32$ (Yes)	$v = 4$ (No)	$v = 16$ (Yes)

Comparison(s) to $R[x]/\langle x^{1530} - 1\rangle$ from [1]. The earliest FFT work over R was implemented by [1]. Since 4591 is a prime, one can only define Cooley–Tukey FFTs of sizes factors of $4591 - 1 = 2 \cdot 3^2 \cdot 5 \cdot 17$. They computed the isomorphsims $R[x]/\langle x^{1530} - 1\rangle \cong \prod_i R[x]/\langle x^{90} - \omega_{17}^i\rangle \cong \prod_i R[x]/\langle x^{10} - \omega_{102}^i\rangle$ with size-17

Table 2. Summary of vectorization-friendly approaches.

ISA/extension	[5]	[14]	This work
	AVX2	Neon	Neon/AVX2
Domain	$\frac{R[x]}{\left\langle \frac{x^{2048}-1}{x^{512}+1} \right\rangle}$	$\frac{R[x]}{\langle x^{1632}-1 \rangle}$	$\frac{R[x]}{\langle \Phi_{17}(x^{96}) \rangle}$
FFT	Schönhage	Rader-17 + `GT`	truncated Rader-17 + `GT`
Image	$\left(\frac{R[x]}{\langle x^{64}+1 \rangle} \right)^{48}$	$\prod_i \frac{R[x]}{\langle x^{16}-\omega_{102}^i \rangle}$	$\left(\prod \frac{R[x]}{\langle x^{16} \pm 1 \rangle} \right)^{48}$

Rader's and Cooley–Tukey FFTs. Since 2 is the only power-of-two factor of 1530, their isomorphisms are not vectorization-friendly if there are more than two elements in a vector register.

Comparison(s) to $R[x] \Big/ \left\langle \frac{x^{2048}-1}{x^{512}+1} \right\rangle$ from [5]. We compare our AVX2 implementation to the state-of-the-art AVX2 work by [5]. In [5], they made a first attempt to deliver a large-dimensional power-of-two-sized FFT polynomial multiplier in AVX2 based on Schönhage's and Nussbaumer's FFTs. Since $(x^{2048}-1)/(x^{512}+1)$ is a factor of $x^{2048}-1$, they applied the Schönhage's FFT in a similar way for $R[x]/\langle x^{2048}-1 \rangle$, leading to polynomial multiplications in $R[x]/\langle x^{64}+1 \rangle$. They then applied Nussbaumer's FFT to all the 48 copies of $R[x]/\langle x^{64}+1 \rangle$. One can show that power-of-two Schönhage's FFT is vectorization-friendly and Nussbaumer's FFT is permutation-friendly, and the overall computation is suitable for vectorization. As for polynomial multiplications in $R[z]/\langle z^8+1 \rangle$, they applied recursive Karatsuba. The downside of their approach is the number of subproblems. Since each applications of Schönhage's and Nussbaumer's FFTs doubles the number of coefficients, there are eventually $\frac{1536 \cdot 4}{8} = 768$ polynomial multiplications in the ring $R[z]/\langle z^8+1 \rangle$. In our transformation for AVX2, we only need $48 \cdot 4 = 192$ size-8 polynomial multiplications. This is the main reason why our AVX2 implementation outperforms [5]'s implementation.

Comparison(s) to $R[x]/\langle x^{1632}-1 \rangle$ from [14]. Finally, we compare our Neon implementation to the state-of-the-art Neon work by [14]. They applied a 3-dimensional Good–Thomas FFT to $R[x]/\langle x^{1632}-1 \rangle$ built upon the coprime factorization $\frac{1632}{16} = 2 \cdot 3 \cdot 17$ and Rader's FFT for the size-17 transformation, resulting in $R[x]/\langle x^{16}-\omega_{102}^i \rangle$ up to a suitable permutation. Since $102 \cdot 16 = 1632$ is not a multiple of 64 (there are 8 elements in each vector register and $64 = 8^2$), the follow up computation can't be permutation-friendly. They then applied radix-2 Cooley–Tukey and Bruun's FFT to $\prod_{i<96} R[x]/\langle x^{16}-\omega_{102}^i \rangle$. For the remaining part $\prod_{i \geq 96} R[x]/\langle x^{16}-\omega_{102}^i \rangle$, they interleaved the polynomials with don't-cares and applied naïve computation. Our transformation removes this part.

Table 3. Summary of permutation-friendly approaches with AVX2. `K` stands for Karatsuba.

	[5]	This work
Domain	$\left(\frac{R[x]}{\langle x^{64}+1\rangle}\right)^{48}$	$\left(\prod \frac{R[x]}{\langle x^{16}\pm 1\rangle}\right)^{48}$
FFT	Nussbaumer	`CT` + `Bruun`
Image	$\left(\frac{R[z]}{\langle z^{8}+1\rangle}\right)^{768}$	$\left(\prod \frac{R[x]}{\langle x^{8}\pm 1\rangle} \times \prod \frac{R[x]}{\langle x^{8}\pm\sqrt{2}x^4+1\rangle}\right)^{48}$
Follow up polymul.	Recursive `K`	`K`
Multiplication instruction	Vector-by-vector	Vector-by-vector

Table 4. Summary of permutation-friendly and Toeplitz matrix-vector product approaches multiplying small-dimensional polynomials in Neon.

	[14]	This work
Domain	$\prod_i \frac{R[x]}{\langle x^{16}-\omega_{102}^i\rangle}$	$\left(\prod \frac{R[x]}{\langle x^{16}\pm 1\rangle}\right)^{48}$
FFT	`CT` + `Bruun`	`CT`
Image	$\prod_{i<48}\left(\prod \frac{R[x]}{\langle x^{8}\pm\omega_{51}^i\rangle}\right) \times \prod_{i<48}\left(\prod \frac{R[x]}{\langle x^{8}\pm\sqrt{2}\omega_{51}^{64i}x^4+\omega_{51}^{128i}\rangle}\right) \times \prod_{i>=96} \frac{R[x]}{\langle x^{16}-\omega_{102}^i\rangle}$	$\left(\prod \frac{R[x]}{\langle x^{8}\pm 1\rangle} \times \frac{R[x]}{\langle x^{16}+1\rangle}\right)^{48}$
Follow up polymul.	Naïve (size-8) + `K` (size-16)	Toeplitz
Multiplication instruction	Vector-by-vector	Vector-by-scalar

5 Results

5.1 Benchmarking Environment

Intel Processors with AVX2. We benchmark our AVX2 implementation on a single core of an Intel(R) Core(TM) i7-4770K (Haswell) processor with frequency 3.5 GHz, and Intel(R) Xeon(R) CPU E3-1275 v5 (Skylake) with frequency 3.6 GHz. For benchmarking polynomial multiplications, we compile with GCC 10.4.0 on Haswell and GCC 11.3.0 on Skylake using the optimization flag `-O3`. For the batch key generation, we reuse the `libsntrup761-20210608` package from [5]. For the encapsulation and decapsulation, we benchmark with the benchmarking framework SUPERCOP, version `supercop-20230530`. TurboBoost and hyperthreading are disabled throughout the entire benchmarking.

Armv8.0$^+$-A Neon. We benchmark our Neon implementation on a Raspberry Pi 4 Model B and Apple M1 Pro. Raspberry Pi 4 comes with the quad-core (Cortex-A72) Broadcom BCM2711 chipset and runs at 1.5GHz. Apple M1 Pro is a system-on-chip featuring eight high-performance cores "Firestorms" running at 3.2 GHz and two energy-efficient cores "Icestorm" running at 2.0 GHz. We compile our code with GCC version 12.3.0 with -O3 on Cortex-A72, and GCC version 13.2.0 with -O3 on Firestorm.

5.2 Performance of Polynomial Multiplication

We provide the performance cycles of functions mulcore and polymul in Table 5. mulcore computes the product in $\mathbb{Z}_{4591}[x]$ with potential scaling by a predefined constant, and polymul additionally reduces the product modulo $x^{761} - x - 1$ and mitigates the potential scaling. Our AVX2-optimized mulcore outperforms the state-of-the-art AVX2 implementation from [5] by factors of 1.90× and 2.05× on Haswell and Skylake, and polymul outperforms the state-of-the-art AVX2 implementation by factors of 1.99× and 2.16× on Haswell and Skylake. As for our Neon-optimized mulcore and polymul, they outperform the state-of-the-art Neon implementation from [14] by factors of 1.25× and 1.29× on Cortex-A72, and 1.25× and 1.36× on Apple M1 Pro.

Table 5. Performance cycles of polynomial multiplications over $\mathbb{Z}_{4591}$ for sntrup761.

AVX2				
	[5]*	This work	[5]*	This work
	Haswell		Skylake	
mulcore ($\mathbb{Z}_{4591}[x]$)	23 460	**12 336**	20 070	**9 778**
polymul $\left(\frac{\mathbb{Z}_{4591}[x]}{\langle x^{761}-x-1\rangle}\right)$	25 356	**12 760**	21 364	**9 876**
Neon				
	[14]	This work	[14]*	This work
	Cortex-A72		Apple M1 Pro	
mulcore ($\mathbb{Z}_{4591}[x]$)	37 475	**29 909**	8 120	**6 508**
polymul $\left(\frac{\mathbb{Z}_{4591}[x]}{\langle x^{761}-x-1\rangle}\right)$	39 788	**30 912**	9 091	**6 697**

* Our own benchmarks.

5.3 Performance of Scheme

Finally, we compare the overall performance of sntrup761, and summarize them in Table 6.

AVX2 Code Package(s). For the AVX2-optimized implementation, we integrate our code into the package libsntrup761 with version 20210608 provided by [5], and report the amortized cost of batch key generation with batch size 32. Additionally, we also integrate our code into the package supercop with version 20230530, and report the performance of encapsulation and decapsulation after contacting the authors of [5] for reproducing the numbers in their work.

Neon Code Package(s). For the Neon-optimized implementation, We integrate our code into the artifact provided by [14].

Overall Performance with AVX2. For the batch key generation with batch size 32, we reduce the amortized cost by 12.0% on Haswell and 7.9% on Skylake. For encapsulation, we reduce the cost by 7.1% on Haswell and 10.3% on Skylake. For decapsulation, we reduce the cost by 10.7% on Haswell and 13.3% on Skylake.

Overall Performance with Neon. For the encapsulation, we reduce the cycles by 6.6% on Cortex-A72 and 3.0% on Apple M1 Pro, and for the decapsulation, we reduce the cycles by 15.1% on Cortex-A72 and 12.8% on Apple M1 Pro.

Table 6. Overall performance of our AVX2 implementation on Haswell and Skylake and our Neon implementation on Cortex-A72 and Apple M1.

AVX2				
	Haswell		Skylake	
	[5]**	This work	[5]**	This work
Batch key generation	154 552	**136 003**	129 159	**118 939**
	SUPERCOP	This work	SUPERCOP	This work
Encapsulation	47 464	**44 108**	40 653	**36 486**
Decapsulation	56 064	**50 080**	47 387	**41 070**
Neon				
	Cortex-A72		Apple M1 Pro	
	[14]**	This work	[14]**	This work
Key generation	6 574 055	**6 539 849**	1 813 947	**1 806 741**
Encapsulation	150 054	**140 107**	64 924	**62 959**
Decapsulation	159 286	**135 184**	43 778	**38 196**

** Our own benchmarks.

References

1. Alkim, E., et al.: Polynomial multiplication in NTRU prime comparison of optimization strategies on Cortex-M4. IACR Trans. Crypt. Hardw. Embed. Syst. **2021**(1), 217–238 (2021). https://tches.iacr.org/index.php/TCHES/article/view/8733
2. Alkim, E., Hwang, V., Yang, B.Y.: Multi-parameter support with NTTs for NTRU and NTRU prime on Cortex-M4. IACR Trans. Crypt. Hardw. Embed. Syst. **2022**(4), 349–371 (2022)
3. Bernstein, D.J.: Fast norm computation in smooth-degree abelian number fields. Cryptology ePrint Archive, Paper 2022/980 (2022). https://eprint.iacr.org/2022/980

4. Bernstein, D.J., et al.: NTRU Prime. Submission to the NIST Post-Quantum Cryptography Standardization Project [21] (2020). https://ntruprime.cr.yp.to/
5. Bernstein, D.J., Brumley, B.B., Chen, M.S., Tuveri, N.: OpenSSLNTRU: faster post-quantum TLS key exchange. In: 31st USENIX Security Symposium (USENIX Security 22), pp. 845–862 (2022)
6. Bernstein, D.J., Duif, N., Lange, T., Schwabe, P., Yang, B.Y.: High-speed high-security signatures. J. Cryptogr. Eng. **2**(2), 77–89 (2012)
7. Bernstein, D.J., Yang, B.Y.: Fast constant-time gcd computation and modular inversion. IACR Trans. Crypt. Hardw. Embed. Syst. **2019**(3), 340–398 (2019). https://tches.iacr.org/index.php/TCHES/article/view/8298
8. Blake, I.F., Gao, S., Mullin, R.C.: Explicit factorization of $x^{2^k}+1$ over $\mathbb{F}_p$ with prime $p \equiv 3 \bmod 4$. Appl. Algebra Eng. Commun. Comput. **4**(2), 89–94 (1993)
9. Bourbaki, N.: Algebra I. Springer, Heidelberg (1989)
10. Bruun, G.: z-transform DFT filters and FFT's. IEEE Trans. Acoust. Speech Sig. Process. **26**(1), 56–63 (1978)
11. Chen, H.T., Chung, Y.H., Hwang, V., Yang, B.Y.: Algorithmic views of vectorized polynomial multipliers – NTRU. In: Chattopadhyay, A., Bhasin, S., Picek, S., Rebeiro, C. (eds.) Progress in Cryptology, INDOCRYPT 2023. LNCS, vol. 14460, pp. 177–196. Springer, Cham (2024). https://doi.org/10.1007/978-3-031-56235-8_9
12. Cooley, J.W., Tukey, J.W.: An algorithm for the machine calculation of complex Fourier series. Math. Comput. **19**(90), 297–301 (1965)
13. Franchetti, F., et al.: Spiral: extreme performance portability. Proc. IEEE **106**(11), 1935–1968 (2018). https://ieeexplore.ieee.org/document/8510983
14. Hwang, V., Liu, C.T., Yang, B.Y.: Algorithmic views of vectorized polynomial multipliers – NTRU prime. In: Pöpper, C., Batina, L. (eds.) Applied Cryptography and Network Security, ACNS 2024. LNCS, vol. 14584, pp. 24–46. Springer, Cham (2024). https://doi.org/10.1007/978-3-031-54773-7_2
15. Hwang, V.B.: Case studies on implementing number–theoretic transforms with Armv7-M, Armv7E-M, and Armv8-A. Master's thesis, National Taiwan University (2022). https://github.com/vincentvbh/NTTs_with_Armv7-M_Armv7E-M_Armv8-A
16. Jacobson, N.: Basic Algebra I. Courier Corporation (2012)
17. Jacobson, N.: Basic Algebra II. Courier Corporation (2012)
18. Karatsuba, A.A., Ofman, Y.P.: Multiplication of many-digital numbers by automatic computers. Dokl. Akad. Nauk **145**(2), 293–294 (1962)
19. Írem Keskinkurt Paksoy, Cenk, M.: Faster NTRU on ARM Cortex-M4 with TMVP-based multiplication. IEEE Transac. Circ. Syst. I Regul. Pap. **69**(10), 4083–4092 (2022). https://ieeexplore.ieee.org/document/9835023
20. Murakami, H.: Real-valued fast discrete Fourier transform and cyclic convolution algorithms of highly composite even length. In: 1996 IEEE International Conference on Acoustics, Speech, and Signal Processing Conference Proceedings, vol. 3, pp. 1311–1314 (1996)
21. NIST, The US National Institute of Standards and Technology: Post-quantum cryptography standardization project. https://csrc.nist.gov/Projects/post-quantum-cryptography
22. Rader, C.M.: Discrete Fourier transforms when the number of data samples is prime. Proc. IEEE **56**(6), 1107–1108 (1968)
23. Shor, P.W.: Algorithms for quantum computation: discrete logarithms and factoring. In: Proceedings 35th Annual Symposium on Foundations of Computer Science, pp. 124–134. IEEE (1994)