Pushing the Limit of Vectorized Polynomial Multiplications for NTRU Prime (SoK)

Vincent Hwang

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

Abstract. In this survey paper, 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 an array of vector arithmetic on vector 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 modules homomorphisms to vectorized implementations.

To illustrate how the formalization works, we first review the vectorization of the well-studied polynomial multiplication in the polynomial ring $\mathbb{Z}_{3329}[x]/\left\langle x^{256}+1\right\rangle$ used in the key encapsulation mechanism (KEM) Kyber as a warmup. We then go through, arguably, the most challenging task for vectorization – polynomial multiplication for $\mathbb{Z}_{4591}[x]/\left\langle x^{761}-x-1\right\rangle$ used in the parameter set `sntrup761` of the NTRU Prime KEM.

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, and 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

In this survey paper, we conduct a systematic examination of vector arithmetic for polynomial multiplications in software. 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 vector 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. There are two questions we wish to answer in this paper:

1. Why algebra homomorphisms defined on polynomial rings with power-of-two-multiple number of elements are frequently assumed to admit efficient vectorization processes?

2. Which algebra 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 algebra homomorphisms resulting 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 remaing 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 [BBCT22] 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, and [HLY24] proposed to use radix-2 Bruun's FFT as an alternative. In this paper, 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 [Ber22, Section 4.8], is a suitable one for vectorization.

Contributions. We summarize our contributions as follows.

- We formalize vectrozation-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 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 $\mathbb{Z}_{4591}[x]\Big/\left\langle \sum_{i=0,\ldots,16} x^{96i} \right\rangle$ 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 [BBCT22] by 1.99× on Haswell and 2.16× on Skylake, and our Neon implementation outperforms the state-of-the-art Neon-optimized implementation from [HLY24] 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 [BBCT22] 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 [HLY24]. 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.

Code. Our source code is attached as an artifact and will be publicly available.

Structure of this paper. This paper is structured as follows. Section 2 goes through the preliminaries. Section 3 formalizes the vectorization process, and Section 4 reviews various algebra homomorphisms. We then go through two walkthroughs. Section 5 shows how radix-2 Cooley–Tukey FFT is mapped to vector arithmetic, and Section 6 illustrates the applications of vector arithmetic to truncated Rader's, Good–Thomas, and Bruun's FFTs. Finally, Section 7 shows the performance cycles with AVX2 on Haswell and Skylake, and Neon on Cortex-A72 and Apple M1 Pro.

2 Preliminaries

We go through some preliminaries for this paper. Section 2.1 describes the target polynomial arithmetic of NTRU Prime, Section 2.2 reviews some basics from algebra with emphasis on tensor of module homomorphisms, and Section 2.3 describes our interested vector instruction set architectures and extensions.

2.1 Streamlined NTRU Prime

NTRU Prime [BBC+20] 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 [BBC+20] 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 [BBC+20] for more details of the scheme. We demonstrate the performance of `sntrup` to avoid superflous context blurring the contribution of this paper. 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.

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

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 $\mathbb{Z}_3 \coloneqq \{0, \pm 1\}$ and "big-by-big" otherwise. Our polynomial multipliers target big-by-big ones and also covers the big-by-small ones 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 [BY19] as a future work.

To see why big-by-big polynomial multiplications are actually important, we briefly review the uses of Montgomery's trick for batch inversion [BBCT22]. Let's say we want to invert a finite series of polynomials $(\boldsymbol{a}_0, \ldots, \boldsymbol{a}_{n-1})$ in $\mathbb{Z}_q[x]/\langle x^p - x - 1\rangle$. Instead of inverting each of them one at a time, we first compute the products

$$\boldsymbol{a}_0, \boldsymbol{a}_0\boldsymbol{a}_1, \ldots, \prod_{i=0,\ldots,n-1} \boldsymbol{a}_i$$

with $n-1$ polynomial multiplications, and invert $\prod_{i=0,\ldots,n-1} \boldsymbol{a}_i$. We now compute the following

$$\left(\prod_{i=0,\ldots,n-1} \boldsymbol{a}_i\right)^{-1}, \left(\prod_{i=0,\ldots,n-2} \boldsymbol{a}_i\right)^{-1}, \ldots, \boldsymbol{a}_0^{-1}$$

with $n-1$ polynomial multiplications. Finally, we recover the inverses $\boldsymbol{a}_j^{-1}$ by computing $\boldsymbol{a}_j^{-1} = \left(\prod_{i=0,\ldots,j} \boldsymbol{a}_i\right)^{-1} \left(\prod_{i=0,\ldots,j-1} \boldsymbol{a}_i\right)$ for $j = 1, \ldots, n-1$. In `sntrup`, since all polynomials to be inverted have coefficients in $\mathbb{Z}_3$, 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 for notations whenever needed. We assume that readers are all familiar with monoids, groups, rings, and modules and refer to standard algebra books [Jac12a, Jac12b, Bou89] for references. 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 of this paper revolves around free-module homomorphisms and their tensor products. We call a module free if there is a basis. For a ring R, the n-fold product R^n is a free module since an element (r_i) can be written as the linear combination $\sum_i r_i e_i$ 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 the following equivalences:

- $\forall \boldsymbol{a}_0, \boldsymbol{a}_1 \in R^n, \forall \boldsymbol{b} \in R^m, (\boldsymbol{a}_0 + \boldsymbol{a}_1) \otimes \boldsymbol{b} \sim \boldsymbol{a}_0 \otimes \boldsymbol{b} + \boldsymbol{a}_1 \otimes \boldsymbol{b}$.
- $\forall \boldsymbol{a} \in R^n, \forall \boldsymbol{b}_0, \boldsymbol{b}_1 \in R^m, \boldsymbol{a} \otimes (\boldsymbol{b}_0 + \boldsymbol{b}_1) \sim \boldsymbol{a} \otimes \boldsymbol{b}_0 + \boldsymbol{a} \otimes \boldsymbol{b}_1$.
- $\forall r \in R, \boldsymbol{a} \in R^n, \boldsymbol{b} \in R^m, (r\boldsymbol{a}) \otimes \boldsymbol{b} \sim \boldsymbol{a} \otimes (r\boldsymbol{b})$.

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 tensor product, one can find

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

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 the following isomorphisms:

$$\frac{R[x]}{\langle \boldsymbol{g}(x^v) \rangle} \cong \frac{R[x,y]}{\langle x^v - y, \boldsymbol{g}(y) \rangle} \cong \frac{\frac{R[x]}{\langle x^v - y \rangle}[y]}{\langle \boldsymbol{g}(y) \rangle}.$$

Let's abbreviate with $\mathcal{R} \coloneqq R[x]/\langle x^v - y \rangle$. 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 and give a small example as follows. Suppose $\boldsymbol{g}(x) = x^4 - 1$, $v = 2$, and $\mathcal{R}[y]/\langle y^2 - 1 \rangle \cong \mathcal{R}[y]/\langle y - 1 \rangle \times \mathcal{R}[y]/\langle y + 1 \rangle$ via $\boldsymbol{a}_0 + \boldsymbol{a}_1 y \mapsto (\boldsymbol{a}_0 + \boldsymbol{a}_1, \boldsymbol{a}_0 - \boldsymbol{a}_1)$. Obviously, the transformation matrix over $\mathcal{R}$ is

$$\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}.$$

If we replace y with x^2, define $\boldsymbol{a}_0 = \boldsymbol{a}_{0,0} + \boldsymbol{a}_{0,1}x, \boldsymbol{a}_1 = \boldsymbol{a}_{1,0} + \boldsymbol{a}_{1,1}x$, and rewrite the homomorphism as a transformation matrix over R, we find that $\boldsymbol{a}_{0,0} + \boldsymbol{a}_{0,1}x + \boldsymbol{a}_{1,0}x^2 + \boldsymbol{a}_{1,1}x^3 \mapsto (\boldsymbol{a}_{0,0} + \boldsymbol{a}_{1,0} + (\boldsymbol{a}_{0,1} + \boldsymbol{a}_{1,1})\, x, \boldsymbol{a}_{0,0} - \boldsymbol{a}_{1,0} + (\boldsymbol{a}_{0,1} - \boldsymbol{a}_{1,1})\, x)$ can be written as

$$\begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}.$$

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 $R[x]/\langle \boldsymbol{g}(x^v) \rangle$.

As a side note, there is an analogue of tensor product of algebras and algebra homomorphisms (and they are helpful for understanding techniques in Section 4.3), but we feel that they are too abstract for the audience and decide to stay at tensor product of module homomorphisms with matrix view.

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 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. In other words, each registers can be regarded as an element in the set $\mathbb{Z}_{65536}^{16}$. Along with the component-wise addition `vpaddw` and subtraction `vpsubw`, we have the abelian group structure with the vector of all zeros as the zero element. We also have the component-wise multiplication `vpmullw` with a 16-tuple of all ones as the multiplicative identity. This gives us a $\mathbb{Z}_{65536}$-module structure by implementing $r\,(a_i)$ as the component-wise product of a tuple of r's and (a_i) for $r \in \mathbb{Z}_{65536}^{16}$ and $(a_i) \in \mathbb{Z}_{65536}^{16}$. Furthermore, we also have several permutation instructions with two data operands – `vpunpck{l, h}{wd, dq, qdq}` interleaves packed halfwords/words/double words from the lower/upper 64-bit of each 128-bit of the operands, and `vperm2i128` selects arbitrary pair of 128-bit data from the multi-set formed by unioning all the 128-bit data from the operands and the zero element. Frequently, a series of permutation instructions are used for implementing a certain kind of permutation matrices. We'll give more insights on permutations in Section 3.2.

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 and instruction encodings are determined by the instructions along with specifiers following register names – we append `.8H` to the name of a vector register if the 128-bit data is regarded as packed halfwords. Similarly to AVX2, we have componenet-wise addition and subtraction instructions giving an abelian group. Since there are vector-by-vector and vector-by-scalar multiplication instructions, the module structure is determined by implementing $r(a_i)$ as a component-wise product of a vector-by-vector multiplication instruction or as a straightforward result of applying a vector-by-scalar multiplication instructions. In Neon, a vector-by-scalar multiplication instruction multiplies a vector of elements by a scalar and returns a vector. While dealing with halfwords, the scalar value must come from a lane of a low registers (`v0` to `v7`). Similar to AVX2, there is a wide variaty 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. Suppose we have two input vector registers holding single words $\texttt{v2} = (a_0, \ldots, a_3)$ and $\texttt{v3} = (b_0, \ldots, b_3)$ and one output register `v0`, the assembly code `ext v0.16B, v2.16B, v3.16B, #12` assigns (a_3, b_0, b_1, b_2) (notice that 12 bytes are 3 words) to `v0`. In the matrix view, we have

$$\texttt{v0} = \begin{pmatrix} a_3 \\ b_0 \\ b_1 \\ b_2 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} a_0 \\ \vdots \\ a_3 \\ b_0 \\ \vdots \\ b_3 \end{pmatrix} = \begin{pmatrix} \texttt{v2} \\ \texttt{v3} \end{pmatrix}.$$

Interestingly, if we further issue `ext v1.16B, v3.16B, v2.16B, #12`, then $(\texttt{v0}, \texttt{v1})$ is the cyclic shift $(a_3, b_0, \ldots, b_3, a_0, a_1, a_2)$ of $(\texttt{v2}, \texttt{v3})$.

3 Formalization of Vectorization

Let v be the number of elements contained in a vector register. In this section, we distinguish four basic algebraic constructs: (i) a certain class of block diagonal matrices capturing component-wise multiplication and cyclic/negacyclic shifts, (ii) tensor products of module homomorphisms, (iii) a certain class of permutation matrices interleaving strings of elements, and (iv) Toeplitz matrices. We formalize vectorization-friendliness based on (i) and (ii), permutation-friendliness based on (i), (ii), and (iv), and discuss an alternative of permutation-friendliness based on (iv).

Generally, while computing with vector instructions, we choose algebra monomorphisms 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. Recent work SPIRAL [FLP+18] 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 section will give more insights on extending SPIRAL.

Section 3.1 formalizes vectorization-friendliness, Section 3.2 formalizes permutation-friendliness, and Section 3.3 reviews small-dimensional Toeplitz matrix-vector products.

3.1 Vectorization–Friendliness

Conceptually, we call an algebra monomorphism vectorization-friendly if we can factor it into module homomorphisms with matrix forms 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 $(a_i)_{0 \le i < v'} \mapsto \left(a_{(i+c) \bmod v'}\right)_{0 \le i < v'}$ (cyclic) or $(a_i)_{0 \le i < v'} \mapsto \left((-1)^{[\![i+c \ge v']\!]} a_{(i+c) \bmod v'}\right)_{0 \le i < v'}$ (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 [HLY24].
- 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 [BBCT22].

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 $S_{f_i} \in$ `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. We illustrate the case $v' = 2$ with Algorithm 1 for Neon and Algorithm 2 for AVX2. 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

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

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 $S_{g_i} \in$ `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.

Algorithm 1 `trn{1, 2}` permuting double words in Armv8.0-A Neon registers.

Inputs: $(\texttt{v0}, \texttt{v1}) = ((a_0, a_1), (b_0, b_1))$.
Outputs: $(\texttt{v2}, \texttt{v3}) = ((a_0, b_0), (a_1, b_1))$.

```
1: trn1 v2.2D, v0.2D, v1.2D
2: trn2 v3.2D, v0.2D, v1.2D
```

Algorithm 2 `vperm2i128` permuting double words in AVX2 `%ymm` registers.

Inputs: $(\texttt{\%ymm0}, \texttt{\%ymm1}) = ((a_0, a_1), (b_0, b_1))$.
Outputs: $(\texttt{\%ymm2}, \texttt{\%ymm3}) = ((a_0, b_0), (a_1, b_1))$.

```
1: vperm2i128 %ymm2, %ymm0, %ymm1, 0x20
2: vperm2i128 %ymm3, %ymm0, %ymm1, 0x31
```

3.3 Toeplitz Matrix–Vector Products (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 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. Below we illustrate with $(a_0b_0 - a_1b_1) + (a_0b_1 + a_1b_0) = (a_0 + a_1x)(b_0 + b_1x) \bmod (x^2 + 1)$:

$$\begin{pmatrix} a_0b_0 - a_1b_1 \\ a_0b_1 + a_1b_0 \end{pmatrix} = \begin{pmatrix} a_0 & -a_1 \\ a_1 & a_0 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \end{pmatrix}.$$

Generally, one can write a polynomial multiplication modulo $x^{v'} - \zeta$ as an application of a Toeplitz matrix constructed from one of the operands [BDL+12, Hwa22, IKPC22]. Recently, [CCHY23] 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. Algorithm 3 illustrates the idea for $v' = 8$ in Neon.

Algorithm 3 Toeplitz matrix-vector product approach implementing polynomial multiplication in $\mathbb{Z}_{4951}[x]/\langle x^8+1\rangle$ with Neon, adapted from [CCHY23].

Input(s): $\texttt{v16} = (a_0, \ldots, a_7), \texttt{v0} = (b_0, \ldots, b_7)$.

Output(s): $\texttt{v0} = (c_0, \ldots, c_7)$ where c_i is the i-th coefficient of $\left(\sum_j a_j x^j\right)\left(\sum_k b_k x^k\right) \bmod \left(x^8+1\right)$.

```
neg v1.16B, v16.16B
ext v17.16B, v1.16B, v16.16B, #14
⋮
ext v23.16B, v1.16B, v16.16B,  #2
```
▷ $(\texttt{v16} \cdots \texttt{v23})$ is the transformation matrix implementing negacyclic convolution by $(a_0, \ldots, a_7)$.
```
smull  v8.4S, v16.4H, v0.H[0]
smull2 v9.4S, v16.4H, v0.H[0]
smlal  v8.4S, v17.4H, v0.H[1]
smlal2 v9.4S, v17.4H, v0.H[1]
⋮
smlal  v8.4S, v23.4H, v0.H[7]
smlal2 v9.4S, v23.4H, v0.H[7]
```
▷ $(\texttt{v8}, \texttt{v9})$ now contains the result of negacyclic convolution where elements are unreduced.
```
v0 = reduce(v8, v9)
```
▷ We apply custom reduction subroutine. Typically Montgomery reduction [Mon85]. Since this step is independent from our work, we simply refer to [BHK+22, Algorithm 14] to avoid bombarding our audience.

4 Transformations

This section reviews various algebraic techniques, including Chinese remainder theorem in Section 4.1, Cooley–Tukey FFT in Section 4.2, Good–Thomas FFT in Section 4.3, Rader's FFT in Section 4.4, Bruun's FFT in Section 4.5, twisting in Section 4.7, and Karatsuba in Section 4.8.

4.1 Chinese Remainder Theorem

For finitely many coprime polynomials $\boldsymbol{g}_{i_0,\ldots,i_{h-1}} \in R[x]$, the Chinese remainder theorem implies the following series of isomorphisms

$$\frac{R[x]}{\left\langle \prod_{i_0,\ldots,i_{h-1}} \boldsymbol{g}_{i_0,\ldots,i_{h-1}} \right\rangle} \cong \prod_{i_0} \frac{R[x]}{\left\langle \prod_{i_1,\ldots,i_{h-1}} \boldsymbol{g}_{i_0,\ldots,i_{h-1}} \right\rangle} \cong \cdots \cong \prod_{i_0,\ldots,i_{h-1}} \frac{R[x]}{\left\langle \boldsymbol{g}_{i_0,\ldots,i_{h-1}} \right\rangle}$$

by moving the indices from the ideal parts to the product-ring parts. If all the isomorphisms are fast in terms of computational complexity, the overall isomorphism $R[x]\Big/\left\langle \prod_{i_0,\ldots,i_{h-1}} \boldsymbol{g}_{i_0,\ldots,i_{h-1}} \right\rangle \cong \prod_{i_0,\ldots,i_{h-1}} R[x]\Big/\left\langle \boldsymbol{g}_{i_0,\ldots,i_{h-1}} \right\rangle$ will be fast.

4.2 Cooley–Tukey FFT

Let $n = \prod_j n_j$, and i_j runs over $0, \ldots, n_j - 1$ for each j. The Cooley–Tukey FFT [CT65] computes the following isomorphism:

$$\frac{R[x]}{\langle x^n - \zeta^n \rangle} \cong \prod_{i_0,\ldots,i_{h-1}} \frac{R[x]}{\left\langle x - \zeta\omega_n^{\sum_l i_l \prod_{j<l} n_j} \right\rangle}$$

in a layer-by-layer fashion where ω_n is a principal n-th root of unity[2]. The simplest case is the isomorphism $R[x]\Big/\left\langle x^{2^h} - 1 \right\rangle \cong \prod_{i_0,\ldots,i_{h-1}} R[x]\Big/\left\langle x - \omega_{2^h}^{\sum_l i_l 2^l} \right\rangle$. However, we will encounter various transformations built upon non-power-of-two Cooley–Tukey FFTs.

4.3 Good–Thomas FFT

In the previous section we see how Cooley–Tukey FFT factors the transformation matrix into matrix multiplication of easier ones up to a permutation. Good–Thomas FFT [Goo58] employs a different permutation strategy driven by factorizations of cyclic groups into smaller ones and rewrite transformation matrices as tensor products of transformation matrices of cyclic ones. Suppose we have a coprime factorization $n = \prod_j q_j$, Good–Thomas FFT turns a size-n cyclic transformation into a tensor product of size-q_i cyclic transformations. We explain the idea briefly with the smallest case $n = 6$. Consider the cyclic transformation $R[x]/\langle x^6 - 1 \rangle \cong \prod_i R[x]/\langle x - \omega_6^i \rangle$, 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

$$P_{(14)} \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega_6 & \omega_6^2 & \omega_6^3 & \omega_6^4 & \omega_6^5 \\ 1 & \omega_6^2 & \omega_6^4 & 1 & \omega_6^2 & \omega_6^4 \\ 1 & \omega_6^3 & 1 & \omega_6^3 & 1 & \omega_6^3 \\ 1 & \omega_6^4 & \omega_6^2 & 1 & \omega_6^4 & \omega_6^2 \\ 1 & \omega_6^5 & \omega_6^4 & \omega_6^3 & \omega_6^2 & \omega_6 \end{pmatrix} P_{(14)} = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \otimes \begin{pmatrix} 1 & 1 & 1 \\ 1 & \omega_3 & \omega_3^2 \\ 1 & \omega_3^2 & \omega_3 \end{pmatrix}$$
$$= \left(\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \right) \left(\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} 1 & 1 & 1 \\ 1 & \omega_3 & \omega_3^2 \\ 1 & \omega_3^2 & \omega_3 \end{pmatrix} \right).$$

Comparing to Cooley–Tukey FFT, we save two multiplications by ω_6 and ω_6^2.

4.4 Rader's FFT

Let p be an odd prime, and $\mathcal{I} = \{0, \ldots, p-1\}, \mathcal{I}^* = \{1, \ldots, p-1\}$ be index sets. Rader's FFT [Rad68] computes the map $R[x]/\langle x^p - 1 \rangle \cong \prod_i R[x]/\langle x - \omega_p^i \rangle$ with a size-$\lambda(p)$ cyclic convolution where λ is the Carmichael's lambda function. See [Win78] for the odd-prime-power case.

Since p is a prime, there is a $g \in \mathcal{I}$ with $\mathcal{I}^* = \left\{1, g, \ldots, g^{\lambda(p)-1}\right\}$. We define $\log_g : \mathcal{I}^* \to \mathbb{Z}_{\lambda(p)}$ as the discrete logarithm. This allows us to introduce the following reindexing for $(\hat{a}_j)_{j\in\mathcal{I}} = \left(\sum_{i\in\mathcal{I}} a_i \omega_p^{ij}\right)_{j\in\mathcal{I}}$: (i) $i \in \mathcal{I}^* \mapsto -\log_g i \in \mathbb{Z}_{\lambda(p)}$ and (ii) $j \in \mathcal{I}^* \mapsto \log_g j \in \mathbb{Z}_{\lambda(p)}$. For $j \in \mathcal{I}^*$, this gives us

$$\hat{a}_{g^{\log_g j}} - a_0 = \sum_{i\in\mathcal{I}^*} a_i \omega_p^{ij} = \sum_{-\log_g i \in \mathbb{Z}_{\lambda(p)}} a_{g^{\log_g i}} \omega_p^{g^{\log_g i + \log_g j}}.$$

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

Therefore, we find that $\left(\hat{a}_{g^k} - a_0\right)_{k\in\mathbb{Z}_{\lambda(p)}}$ is the convolution of $\left(a_{g^{-k}}\right)_{k\in\mathbb{Z}_{\lambda(p)}}$ and $\left(\omega_p^{g^k}\right)_{k\in\mathbb{Z}_{\lambda(p)}}$. The reindexing transforming $R[x]/\langle x^p-1\rangle \cong \prod_i R[x]/\langle x-\omega_p^i\rangle$ into a size-$(p-1)$ cyclic convolution along with the post-processing is called the **Rader's FFT**.

4.4.1 Truncated Rader's FFT and its Inverse

Let Φ_p be the p-th cyclotomic polynomial. Since p is a prime, we have $\Phi_p(x) = \sum_{i=0,\dots,p-1} x^i$ and $\Phi_p(x)|(x^p-1)$. A natural question is how to build an efficient transformation for $R[x]/\langle\Phi_p(x)\rangle \cong \prod_{i\in\mathcal{I}^*} R[x]/\langle x-\omega_p^i\rangle$ based on the Rader's FFT for $R[x]/\langle x^p-1\rangle$. We first find the following isomorphism

$$\begin{cases} R[x]/\langle\Phi_p(x)\rangle & \cong \prod_{j\in\mathcal{I}^*} R[x]/\langle x-\omega_p^j\rangle\,, \\ \sum_{i\in\mathcal{I}^*} a_{i-1}x^{i-1} & \mapsto \left(\hat{a}_j = \sum_{i\in\mathcal{I}^*} a_{i-1}\omega_p^{(i-1)j}\right)_{j\in\mathcal{I}^*}. \end{cases}$$

With the same reindexing $i \mapsto -\log_g i$ and $j \mapsto \log_g j$, we have

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

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, [Ber22, Section 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 can 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 [Ber22, Sections 4.12.3 and 4.12.4] for the generalization to arbitrary p.

4.5 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 $\mathbb{Z}_q[x]/\langle x^{2^w}+1\rangle$ as follows:

$$\frac{\mathbb{Z}_q[x]}{\langle x^{2^w}+1\rangle} \cong \prod_i \frac{\mathbb{Z}_q[x]}{\langle x^2 \pm \alpha_i x - 1\rangle}.$$

See [BGM93] for a proof. For $q = 4591$, we can split $\mathbb{Z}_q[x]/\langle x^{16}+1\rangle$ 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 $\mathbb{Z}_q[x]/\langle x^{16}+1\rangle \cong \prod \mathbb{Z}_q[x]/\langle x^8 \pm \sqrt{2}x^4 + 1\rangle$. For simplicity, we illustrate with the case $\mathbb{Z}_q[x]/\langle x^4+1\rangle \cong \prod \mathbb{Z}_q[x]/\langle x^2 \pm \sqrt{2}x + 1\rangle$: We compute $(a_0 - a_2, a_1 + a_3, \sqrt{2}a_2, \sqrt{2}a_3)$, swap the last two values, and apply add-sub pairs [HLY24]. See below for the corresponding transformation matrices:

$$\begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & -1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & \sqrt{2} & 0 \\ 0 & 0 & 0 & \sqrt{2} \end{pmatrix}.$$

Bruun's FFT was originally proposed with $\mathbb{C}$ as the coefficient ring. See [Bru78] for the power-of-two case and [Mur96] for the even case.

4.6 Schönhage's and Nussbaumer's FFTs

Schönhage's [Sch77] and Nussbaumer's [Nus80] FFTs convert polynomial reduction modulo cyclotomic polynomials into multiplications by roots of unity at the cost of doubling the number of coefficients. We go through a brief introduction for the power-of-two cyclotomic cases.

For a ring $R[x]\Big/\left\langle x^{2^{k_0+k_1}} - 1\right\rangle$, Schönhage's FFT introduces $x^{2^{k_0}} \sim y$ rewriting the ring as $\left(R[x]\Big/\left\langle x^{2^{k_0}} - y\right\rangle\right)[y]\Big/\left\langle y^{2^{k_1}} - 1\right\rangle$. If we perform an injection $R[x]\Big/\left\langle x^{2^{k_0}} - y\right\rangle \hookrightarrow \mathcal{R} \coloneqq R[x]\Big/\left\langle x^{2^{k_0+1}} + 1\right\rangle$ by padding 2^{k_0} zeros, we have x a principal 2^{k_0+2}-th root supporting a size-2^{k_0+2} cyclic FFT. In other words, $\mathcal{R}[y]\Big/\left\langle y^{2^{k_1}} - 1\right\rangle$ splits into linear factors in y via additions, subtractions, and multiplications by x in $\mathcal{R}$ if $k_1 \leq k_0 + 2$. Since x is a formal variable, multiplications by powers of x in $\mathcal{R}$ amount to negacyclic shifts. Let's take $k_0 = 1, k_1 = 2$ as an example. We have

$$\frac{R[x]}{\langle x^8 - 1\rangle} \cong \frac{\frac{R[x]}{\langle x^2 - y\rangle}[y]}{\langle y^4 - 1\rangle} \hookrightarrow \frac{\frac{R[x]}{\langle x^4+1\rangle}[y]}{\langle y^4 - 1\rangle} \cong \prod_{i=0,\ldots,3} \frac{\frac{R[x]}{\langle x^4+1\rangle}[y]}{\langle y - x^{2i}\rangle}.$$

For the matrix view, we first observe that $R[x]/\langle x^2 - y\rangle \hookrightarrow R[x]/\langle x^4 + 1\rangle$ can be written as the application of $\begin{pmatrix} 1 \\ 0 \end{pmatrix} \otimes I_2$. Since we have a size-4 cyclic convolution in y, the overall injection map can be written as

$$I_4 \otimes \begin{pmatrix} 1 \\ 0 \end{pmatrix} \otimes I_2,$$

which is vectorization-friendly if each vectors contain two elements. Similar justification holds when 2^{k_0} is a multiple of the number of elements contained in a vector register. Now, let's write down the matrix view of $\mathcal{R}[y]/\langle y^4 - 1\rangle \cong \prod_{i=0,\ldots,3} \mathcal{R}[y]/\langle y - x^{2i}\rangle$ via Cooley–Tukey FFT over $\mathcal{R}$ as:

$$\left(I_2 \otimes \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}\right) \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & x^2 \end{pmatrix} \left(\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \otimes I_2\right).$$

Since $\mathcal{R}$ is a rank-4 module over R, the transformation matrix over R can be written as:

$$\left(I_2 \otimes \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \otimes I_4\right) \begin{pmatrix} I_4 & \mathbf{0}_4 & \mathbf{0}_4 & \mathbf{0}_4 \\ \mathbf{0}_4 & I_4 & \mathbf{0}_4 & \mathbf{0}_4 \\ \mathbf{0}_4 & \mathbf{0}_4 & I_4 & \mathbf{0}_4 \\ \mathbf{0}_4 & \mathbf{0}_4 & \mathbf{0}_4 & \texttt{shift2} \bmod (x^4+1) \end{pmatrix} \left(\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \otimes I_8\right)$$

where

$$\mathbf{0}_4 = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}, \texttt{shift2} \bmod (x^4+1) = \begin{pmatrix} 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}.$$

Obviously, the transformation matrix is vectorization-friendly if each vector registers contains four elements since all the matrices are either right-tensored by I_4 or a block diagonal matrix with each blocks a 4×4 diagonal or cyclic/negacyclic shift matrix.

On the other hand, Nussbaumer starts with

$$\frac{R[x]}{\left\langle x^{2^{k_0+k_1}}+1 \right\rangle} \cong \frac{\mathcal{R}'[x]}{\left\langle x^{2^{k_1-1}}-y \right\rangle} \hookrightarrow \frac{\mathcal{R}'[x]}{\left\langle x^{2^{k_1}}-1 \right\rangle}$$

for $\mathcal{R}' \coloneqq \frac{R[y]}{\left\langle y^{2^{k_0+1}}+1 \right\rangle}$, and regards y as a principal 2^{k_0+2}-th root of unity defining an FFT for $\mathcal{R}'[x] \Big/ \left\langle x^{2^{k_1}}-1 \right\rangle$. It can be shown that the matrix form of Nussbaumer only differs from Schönhage by an interleaving matrix. For the case $k_0 = 1, k_1 = 2$, its matrix form can be obtained by replacing $I_4 \otimes \begin{pmatrix} 1 \\ 0 \end{pmatrix} \otimes I_2$ in Schönhage by the following:

$$T_{16} \left(I_4 \otimes \begin{pmatrix} 1 \\ 0 \end{pmatrix} \otimes I_2 \right).$$

Therefore, Nussbaumer's FFT is permutation-friendly.

4.7 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.

4.8 Karatsuba

Karatsuba [KO62] computes the product $(a_0 + a_1 x)(b_0 + b_1 x)$ by evaluating at the point set $\{0, 1, \infty\}$. We compute $(a_0 + a_1 x)(b_0 + b_1 x) = a_0 b_0 + (a_0 b_1 + a_1 b_0)x + a_1 b_1 x^2$ with three multiplications $a_0 b_0$, $a_1 b_1$, and $(a_0 + a_1)(b_0 + b_1)$ by observing $a_0 b_1 + a_1 b_0 = (a_0 + a_1)(b_0 + b_1) - a_0 b_0 - a_1 b_1$.

5 A Walkthrough for $\mathbb{Z}_{3329}[x]/\langle x^{256}+1 \rangle$

This section goes through a walkthrough demonstrating a suitable mapping from the Cooley–Tukey FFT for

$$\frac{\mathbb{Z}_{3329}[x]}{\langle x^{256}+1 \rangle} \cong \prod_i \frac{\mathbb{Z}_{3329}[x]}{\langle x^2 - \omega_{256}^{2i+1} \rangle}$$

to vector arithmetic as a warmup. We choose the ring $\mathbb{Z}_{3329}[x]/\langle x^{256}+1 \rangle$ since it is used in the KEM Kyber recently standardized by NIST, and it is well-studied to some extent. We go through a detailed justification of vectorization- and permutation-friendliness of the Neon-optimized implementation by [BHK+22]. Since each vector registers in Neon holds $\frac{128}{16} = 8$ cofficients, our justification for vectorization-friendliness boils down to right-tensoring by I_8. For simplicity, we denote $R = \mathbb{Z}_{3329}$ in this section. Recall that

instead of transforming $R[x]/\langle x^{256}+1\rangle$ into $R[x]/\langle x^2-\omega_{256}^{2i+1}\rangle$ directly, Cooley–Tukey FFT applies a series of algebra isomorphisms as follows:

$$\frac{R[x]}{\langle x^{256}+1\rangle} \cong \prod_{i_0=0,1} \frac{R[x]}{\langle x^{128}-\omega_4^{1+2i_0}\rangle} \cong \cdots \cong \prod_{i_0,\ldots,i_6=0,1} \frac{R[x]}{\left\langle x^2-\omega_{256}^{1+2\sum_l i_l 2^l}\right\rangle}.$$

We cut this series into two parts. The first part implements

$$\frac{R[x]}{\langle x^{256}+1\rangle} \cong \prod_{i_0,\ldots,i_4=0,1} \frac{R[x]}{\left\langle x^8-\omega_{64}^{1+2\sum_{l<5} i_l 2^l}\right\rangle}$$

in a vectorization-friendly way and the second part implements

$$\prod_{i_0,\ldots,i_4=0,1} \frac{R[x]}{\left\langle x^8-\omega_{64}^{1+2\sum_{l<5} i_l 2^l}\right\rangle} \cong \prod_{i_0,\ldots,i_6=0,1} \frac{R[x]}{\left\langle x^2-\omega_{256}^{1+2\sum_{l<7} i_l 2^l}\right\rangle}$$

in a permutation-friendly way.

To obtain a vectorization-friendly transformation, we first find the isomorphism $R[x]/\langle x^{256}+1\rangle \cong \left(R[x]/\langle x^8-y\rangle\right)[y]/\langle y^{32}+1\rangle$ reducing the computation to right-tensoring an isomorphism defined on $R[x]/\langle x^{32}+1\rangle$ by I_8. Suppose we have an isomorphism η implementing

$$\frac{R[x]}{\langle x^{32}+1\rangle} \cong \prod_{i_0,\ldots,i_4=0,1} \frac{R[x]}{\left\langle x-\omega_{64}^{1+2\sum_{l<5} i_l 2^l}\right\rangle},$$

we can craft the isomorphism $\eta \otimes I_8$ computing

$$\frac{\left(\frac{R[x]}{\langle x^8-y\rangle}\right)[y]}{\langle y^{32}+1\rangle} \cong \prod_{i_0,\ldots,i_4=0,1} \frac{\left(\frac{R[x]}{\langle x^8-y\rangle}\right)[y]}{\left\langle y-\omega_{64}^{1+2\sum_{l<5} i_l 2^l}\right\rangle}.$$

Replacing y with x^8 gives the desired computation. See Figure 1 for an illustration of the resulting computations and their mapping to 128-bit Neon registers denoted by rectangles.

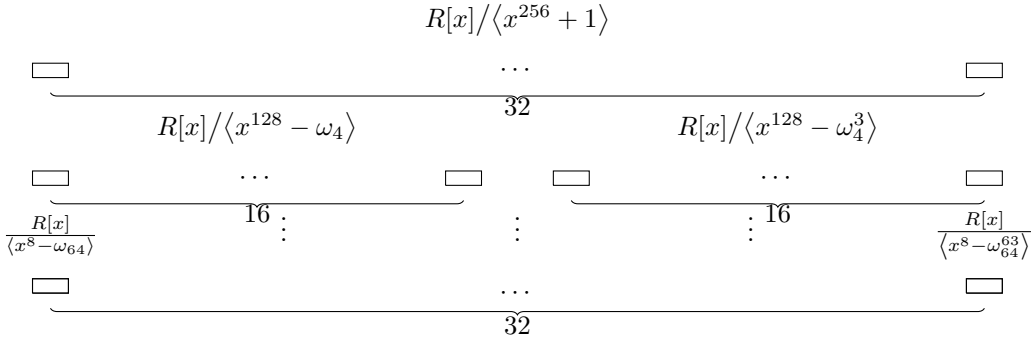

Figure 1: Neon register view of the isomorphism $R[x]/\langle x^{256}+1\rangle \cong \prod_{i_0,\ldots,i_4=0,1} R[x]\Big/\left\langle x^8-\omega_{64}^{1+2\sum_{l<5} i_l 2^l}\right\rangle$ with Cooley–Tukey FFT where $R=\mathbb{Z}_{3329}$ and each rectangles corresponds to a 128-bit Neon register.

The second part illustrates a permutation-friendly way implementing

$$\prod_{i_0,\ldots,i_4=0,1} \frac{R[x]}{\left\langle x^8 - \omega_{64}^{1+2\sum_{l<5} i_l 2^l} \right\rangle} \cong \prod_{i_0,\ldots,i_6=0,1} \frac{R[x]}{\left\langle x^2 - \omega_{256}^{1+2\sum_{l<7} i_l 2^l} \right\rangle}.$$

The basic idea is to group four polynomial rings of the form $R[x]\Big/\left\langle x^8 - \omega_{64}^{1+2\sum_{l<5} i_l 2^l} \right\rangle$ together, permutate with $T_4 \otimes I_2$, and finally apply Cooley–Tukey FFT. Since $256 = 8 \cdot 4 \cdot 8$, we can group all the polynomials into eight groups with no leftovers, implying permutation-friendliness. Readers might wonder why applying $T_4 \otimes I_2$ instead of T_8. The reason is that although algebraically we are working over $\mathbb{Z}_{3329}$, we can in fact introduce $x^2 \sim y$ throughout the FFT. This reduces the complexity of permutation and was implemented by [BHK+22]. See Figure 2 for a concrete layout of permutation where each rectangles represents a 128-bit vector register.

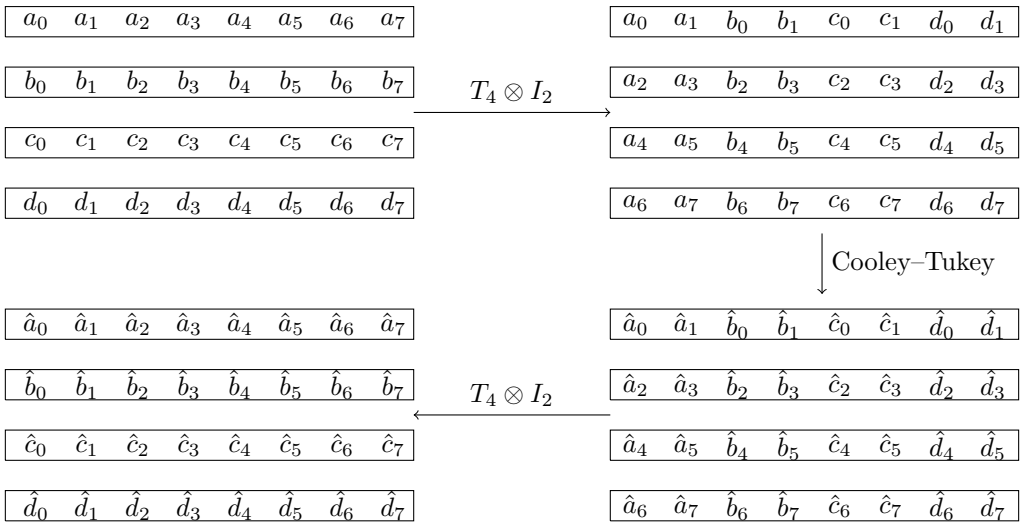

Figure 2: An implementation for $\prod_{i\in\mathcal{I}} \mathbb{Z}_{3329}[x]/\langle x^8 - \omega_{64}^i \rangle \cong \prod_{i\in\mathcal{J}} \mathbb{Z}_{3329}[x]/\langle x^2 - \omega_{256}^i \rangle$ where $\mathcal{I} = \{1, 33, 17, 49\}$, $\mathcal{J} = \{1, 129, 65, 193, 33, 161, 97, 225\}$. Each rectangles represents a 128-bit vector register in Neon.

6 A Walkthrough for $\mathbb{Z}_{4591}[x]/\langle x^{761} - x - 1 \rangle$

This section goes through the implementation with truncated Rader's FFT, Good–Thomas FFT, and Bruun's FFT. We multiply polynomials in

$$\frac{\mathbb{Z}_{4591}[x]}{\langle \Phi_{17}(x^{96}) \rangle}.$$

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 6.1 discusses how to implement the following isomorphisms resulting several size-16 subproblems:

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

The second step is to decide isomorphisms solving the subproblems. Generally, there are two lines of approaches distinguished by the existence of vector-by-scalar multiplication instructions (instructions multiplying vectors by elements of other vectors). We single out the corresponding identifying algebraic structures as follows:

Permutation-friendly isomorphisms: The first approach is to find a permutation-friendly isomorphism. Section 6.2.1 discusses our implementation for the following isomorphism

$$\left(\prod \frac{R[x]}{\langle x^{16} \pm 1\rangle}\right)^{48} \cong \left(\prod \frac{R[x]}{\langle x^{8} \pm 1\rangle} \times \prod \frac{R[x]}{\langle x^{8} \pm \sqrt{2}x^4 + 1\rangle}\right)^{48}$$

via Cooley–Tukey and Bruun's FFT, and specifies the corrsponding interleaving matrices justifying permutation-friendliness.

Toeplitz matrices: The second approach splits all $R[x]/\langle x^{16}-1\rangle$ into $\prod R[x]/\langle x^8 \pm 1\rangle$, and implements all cyclic and negacyclic convolutions with Toeplitz matrix-vector products using vector-by-scalar multiplication instructions. See Section 6.2.2 for more details.

Finally, we go through a detailed comparisons to existing works with emphases on vectorization-friendliness and permutation-friendliness in Section 6.3.

6.1 A Vectorzation-Friendly Approach

We first go through the implementation of

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

via truncated Rader's FFT and twisting. 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. In other words, η_0 maps a tuple $(a_i)_{i=0,\ldots,15}$ to $\left(\omega_{17}^{i+1}\sum_{j=0}^{15} a_j\omega_{17}^{j(i+1)}\right)_{i=0,\ldots,15}$. See Algorithm 4 for an illustration. The isomorphism $R[x]/\langle\Phi_{17}(x)\rangle \cong \prod_{i=0}^{15} R[x]/\langle x-\omega_{17}^{i+1}\rangle$ is then implemented as $\texttt{mul}_0 \circ \eta_0$ where $\texttt{mul}_0 \coloneqq (a_i)_{i=0,\ldots,15} \mapsto \left(\omega_{17}^{-(i+1)}a_i\right)_{i=0,\ldots,15}$. Recall that $R[x]/\langle\Phi_{17}\left(x^{96}\right)\rangle \cong \left(R[x]/\langle x^{96}-y\rangle\right)[y]/\langle\Phi_{17}(y)\rangle$ implies one can always build a transformation defined on $R[x]/\langle\Phi_{17}\left(x^{96}\right)\rangle$ by tensoring an isomorphism defined on $R[x]/\langle\Phi_{17}(x)\rangle$ by I_{96}, we tensor the composition $\texttt{mul}_0 \circ \eta_0$ by I_{96} for implementing $R[x]/\langle\Phi_{17}\left(x^{96}\right)\rangle \cong \prod_{i=0}^{15} R[x]/\langle x^{96}-\omega_{17}^{i+1}\rangle$. We then twist all the rings to the cyclic ones via the product map $\texttt{twist}_0 \coloneqq \prod_{i=0}^{15}\left(x \mapsto \omega_{17}^{14(i+1)}x\right)$ (notice that $\omega_{17} = \omega_{17}^{1344} = \left(\omega_{17}^{14}\right)^{96}$). To sum up, we implement the isomorphism $R[x]/\langle\Phi_{17}\left(x^{96}\right)\rangle \cong \left(R[x]/\langle x^{96}-1\rangle\right)^{16}$ as

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

which is obviously vectorization friendly.

Algorithm 4 Implementation of $\eta_0 = (a_i)_{i=0,\dots,15} \mapsto \left(\omega_{17}^{i+1} \sum_{j=0}^{15} a_j \omega_{17}^{j(i+1)}\right)_{i=0,\dots,15}$.

Input(s): $(a_i)_{i=0,\dots,15}$.

Output(s): $(c_i)_{i=0,\dots,15} = \left(\omega_{17}^{i+1} \sum_{j=0}^{15} a_j \omega_{17}^{j(i+1)}\right)_{i=0,\dots,15}$.

1: **for** $i = 0, \dots, 15$ **do**
2: $\quad$ `src[`$(16 - \log_3(i+1)) \bmod 16$`]` $= a_i$.
3: $\quad$ `twiddle[`$\log_3(i+1)$`]` $= \omega_{17}^{i+1}$.
4: **end for**
5: `buff[0-15]` $=$ `src[0-15]` $\cdot$ `twiddle[0-15]` $\bmod \left(x^{16} - 1\right)$.
6: **for** $i = 0, \dots, 15$ **do**
7: $\quad c_i =$ `buff[`$\log_3(i+1)$`]` $\triangleright$ $c_i = \omega_{17}^{i+1} \sum_{j=0}^{15} a_j \omega_{17}^{j(i+1)}$
8: **end for**

Next, we turn all the rings $R[x]/\langle x^{96} - 1\rangle$ 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 Good–Thomas FFT, and $\texttt{twist}_1 \coloneqq \left(\prod_{i=0,1,2} \left(x \mapsto \omega_3^{2i \bmod 3} x\right)^2\right)^{16}$ twisting $\prod_{i=0,\dots,5} R[x]/\langle x^{16} - \omega_6^i\rangle$ into $\left(\prod R[x]/\langle x^{16} \pm 1\rangle\right)^3$. Then, $\texttt{twist}_1 \circ (\eta_1 \otimes I_{16})$ implements the isomorphism $R[x]/\langle x^{96} - 1\rangle \cong \left(\prod R[x]/\langle x^{16} \pm 1\rangle\right)^3$. See Algorithm 5 for an implementation of η_1. Since there are 16 copies of $R[x]/\langle x^{96} - 1\rangle$, the overall map is

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

Obviously, this is vectorization friendly since $I_{16} \otimes \texttt{twist}_1$ is a diagonal matrix and $I_{16} \otimes \eta_1 \otimes I_{16}$ is right-tensored by I_{16}.

Algorithm 5 Implementation of $\eta_1 \coloneqq (a_i)_{i=0,\dots,5} \mapsto \left(\sum_{j=0}^{5} a_j \omega_6^{ij}\right)_{i=0,\dots,5}$ with Good–Thomas FFT.

Input(s): $(a_i)_{i=0,\dots,5}$.

Output(s): $\left(\sum_{j=0}^{5} a_j \omega_6^{ij}\right)_{i=0,\dots,5}$.

1: $(\texttt{a0}, \dots, \texttt{a5}) = (a_0, a_4, a_2, a_3, a_1, a_5)$.
2: $(\texttt{a0}, \texttt{a1}, \texttt{a2}) = \left(\texttt{a0} + \texttt{a1} \cdot \omega_3^i + \texttt{a2} \cdot \omega_3^{2i}\right)_{i=0,1,2}$.
3: $(\texttt{a3}, \texttt{a4}, \texttt{a5}) = \left(\texttt{a3} + \texttt{a4} \cdot \omega_3^i + \texttt{a5} \cdot \omega_3^{2i}\right)_{i=0,1,2}$.
4: $(\texttt{a0}, \texttt{a3}) = (\texttt{a0} + \texttt{a3}, \texttt{a0} - \texttt{a3})$.
5: $(\texttt{a1}, \texttt{a4}) = (\texttt{a1} + \texttt{a4}, \texttt{a1} - \texttt{a4})$.
6: $(\texttt{a2}, \texttt{a5}) = (\texttt{a2} + \texttt{a5}, \texttt{a2} - \texttt{a5})$.
$\triangleright$ $(\texttt{a0}, \texttt{a4}, \texttt{a2}, \texttt{a3}, \texttt{a1}, \texttt{a5}) = \left(\sum_{j=0}^{5} a_j \omega_6^{ij}\right)_{i=0,\dots,5}$.

For a more illustrative explanation of how polynomials are mapped to 128-bit registers, we outline the workflow in Figure 3 where each rectangles represent a 128-bit register. Note that similar justification holds for 256-bit registers since we are right-tensoring by I_{16}.

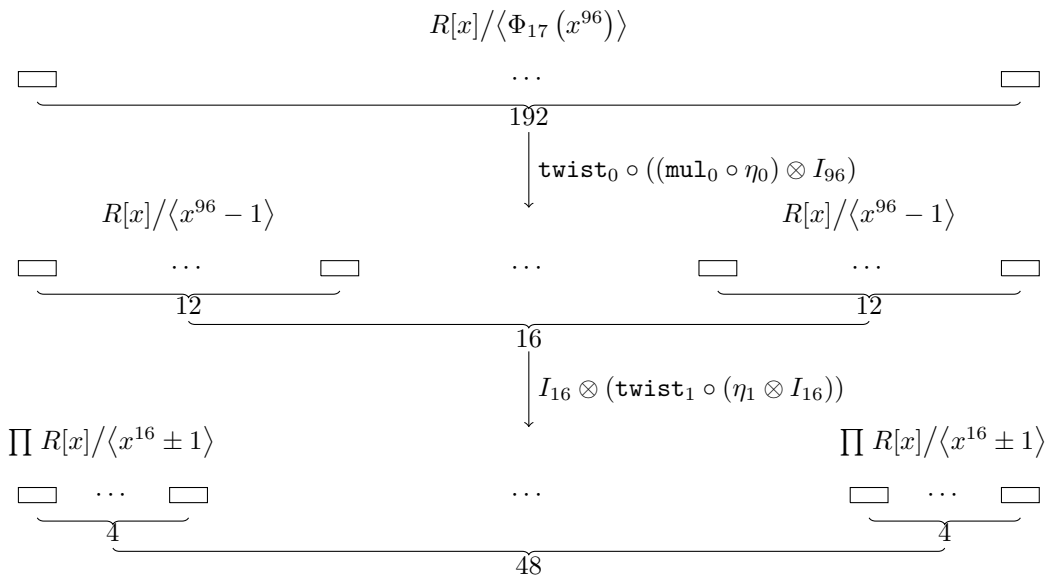

Figure 3: Overview of the correspondence between algebraic maps and 128-bit vector register view in Neon. Each rectangles holds $\frac{128}{16} = 8$ coefficients and is loaded to a vector register. Similar justification of vectorization-friendliness holds if we move two 256-bit vector registers in AVX2.

6.2 Small-Dimensional Cyclic/Negacyclic Convolutions

This section goes through our approaches multiplying in

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

We propose two approaches: a permutation-friendly approach for AVX2 and a Toeplitz matrix-vector product approach for Neon.

6.2.1 A Permutation-Friendly Approach

We first go through a permutation-friendly approach used in our AVX2 implementation. Since the goal is to interleave 16 polynomial rings with the same shape of computation, we explain 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 Figure 4 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.

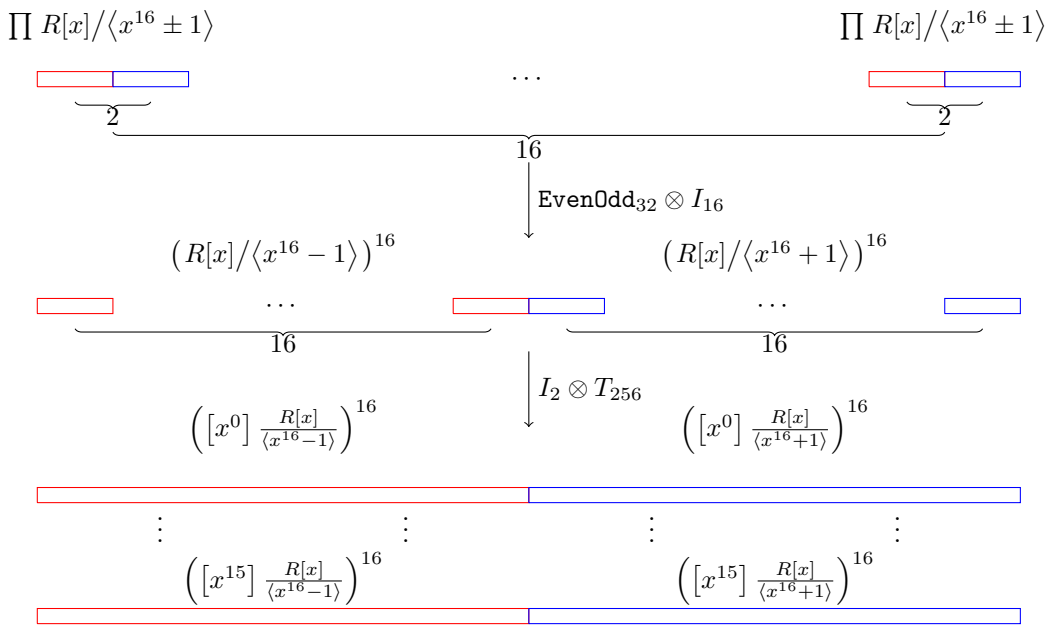

Figure 4: Overview of permutations implementing permutation-friendliness for our AVX2 implementation defined on $\left(R[x]/\langle x^{16} \pm 1\rangle\right)^{16}$. Same idea applies to $\left(R[x]/\langle x^{16} \pm 1\rangle\right)^{48}$ since $48 = 3 \cdot 16$. Each rectangles represents a 16-tuple mapped to a 256-bit vector register in AVX2.

6.2.2 Toeplitz Matrix-Vector Products

Recall that one can phrases polynomial multiplications in $R[x]\Big/\left\langle x^{v'} \pm 1\right\rangle$ as Toeplitz matrix-vector products for v' a multiple of v (cf. Section 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 the example in Section 3.3.

6.3 Comparisons to Prior Implementations

We briefly compare our vectorized implementation to prior FFT works working 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.

Comparison(s) to $R[x]/\langle x^{1530} - 1\rangle$ from [ACC+21]. The earliest FFT work over R was implemented by [ACC+21]. 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 Rader's and Cooley–Tukey FFTs. For the size-10 polynomial multiplications in $\prod_i R[x]/\langle x^{10} - \omega_{102}^i\rangle$, they implemented the naïve approach. Since 2 is the only power-of-two factor of 1530, the isomorphisms are not vectorization-friendly if there are more than two elements in a vector register. Therefore, their approach is unsuitable for vectorization

while moving to Neon and AVX2 where eight and sixteen halfwords are contained in each vector registers, respectively. On the other hand, our transformation is suitable for vectorization in Neon and AVX2.

Comparison(s) to $R[x]\Big/\left\langle \frac{x^{2048}-1}{x^{512}+1} \right\rangle$ from [BBCT22]. We compare our AVX2 implementation to the state-of-the-art AVX2 work by [BBCT22]. In [BBCT22], they made a first attempt to deliver a power-of-two-sized FFT polynomial multiplier in AVX2 based on Schönhage's and Nussbaumer's FFTs. They performed Schönhage's FFT with the following injection:

$$\frac{R[x]}{\left\langle \frac{x^{2048}-1}{x^{512}+1} \right\rangle} \cong \frac{\left(\frac{R[x]}{\langle x^{32}-y \rangle}\right)[y]}{\left\langle \frac{y^{64}-1}{y^{16}+1} \right\rangle} \hookrightarrow \frac{\left(\frac{R[x]}{\langle x^{64}+1 \rangle}\right)[y]}{\left\langle \frac{y^{64}-1}{y^{16}+1} \right\rangle}.$$

Since x^2 is now a principal 32-nd root of unity, we can split the resulting polynomial ring into linear ones in y by appling Cooley–Tukey FFT with twiddle factors x^{2i}'s. After splitting into linear ones in y, they applied Nussbaumer's FFT with the following injection to all the 48 copies of $R[x]/\langle x^{64}+1 \rangle$:

$$\frac{R[x]}{\langle x^{64}+1 \rangle} \cong \frac{\left(\frac{R[z]}{\langle z^{8}+1 \rangle}\right)[x]}{\langle x^{8}-z \rangle} \hookrightarrow \frac{\left(\frac{R[z]}{\langle z^{8}+1 \rangle}\right)[x]}{\langle x^{16}-1 \rangle}.$$

Again, since z is now a principal 16-th root of unity, we can split the ring into size-1 polynomial rings in x. Since Schönhage's FFT is vectorization-friendly and Nussbaumer's FFT is permutation-friendly (cf. Section 4.6), the overall computation is suitable for vectorization. As for polynomial multiplications in $R[z]/\langle z^8+1 \rangle$, they applied recursive Karatsuba. The downsize of their choice is the number of subproblems. Recall that 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 multiplications in $R[x]/\langle x^8-1 \rangle$, 48 multiplications in $R[x]/\langle x^8+1 \rangle$, 48 multiplications in $R[x]/\langle x^8+\sqrt{2}x^4+1 \rangle$, and 48 multiplications in $R[x]/\langle x^8-\sqrt{2}x^4+1 \rangle$. To sum up, only $48 \cdot 4 = 192$ size-8 polynomial multiplications are required in our implementation. This is the main reason why our AVX2 implementation outperform [BBCT22] implementation.

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.

	[ACC+21]	[BBCT22]	[HLY24]	This work
ISA/extension	Armv7E-M	AVX2	Neon	Neon/AVX2
# halfwords 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)

Table 2: Summary of vectorization-friendly approaches.

	[BBCT22]	[HLY24]	This work
ISA/extension	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	$\left(\frac{R[x]}{\langle x^{64}+1\rangle}\right)^{48}$	$\prod_i \frac{R[x]}{\left\langle x^{16}-\omega_{102}^i \right\rangle}$	$\left(\prod \frac{R[x]}{\langle x^{16}\pm 1\rangle}\right)^{48}$

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

	[BBCT22]	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	$\left(\frac{R[z]}{\langle z^{8}+1\rangle}\right)^{768}$	$\left(\prod \frac{R[x]}{\langle x^{8}\pm 1\rangle} \times \prod \frac{R[x]}{\left\langle x^{8}\pm\sqrt{2}x^4+1\right\rangle}\right)^{48}$
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.

	[HLY24]	This work
Domain	$\prod_i \frac{R[x]}{\left\langle x^{16}-\omega_{102}^i \right\rangle}$	$\left(\prod \frac{R[x]}{\langle x^{16}\pm 1\rangle}\right)^{48}$
FFT	`CT` + `Bruun`	`CT`
Image	$\prod_{i<48}\left(\prod \frac{R[x]}{\left\langle x^8\pm\omega_{51}^i\right\rangle}\right) \times \prod_{i<48}\left(\prod \frac{R[x]}{\left\langle x^8\pm\sqrt{2}\omega_{51}^{64i}x^4+\omega_{51}^{128i}\right\rangle}\right) \times \prod_{i>=96} \frac{R[x]}{\left\langle x^{16}-\omega_{102}^i\right\rangle}$	$\left(\prod \frac{R[x]}{\langle x^{8}\pm 1\rangle} \times \frac{R[x]}{\langle x^{16}+1\rangle}\right)^{48}$
Follow up polymul.	Naïve (size-8) + `K` (size-16)	Toeplitz
Multiplication instruction	Vector-by-vector	Vector-by-scalar

Comparison(s) to $R[x]/\langle x^{1632}-1\rangle$ from [HLY24]. Finally, we compare our Neon implementation to the state-of-the-art Neon work by [HLY24]. Let $\mathcal{R} \coloneqq R[x]/\langle x^{16}-y\rangle$. They started by applying a 3-dimensional Good–Thomas FFT to $R[x]/\langle x^{1632}-1\rangle$ as follows:

$$\frac{R[x]}{\langle x^{1632}-1\rangle} \cong \frac{\mathcal{R}[u,w,z]}{\langle u^{17}-1, w^3-1, z^2-1\rangle} \cong \prod_{i_0,i_1,i_2} \frac{\mathcal{R}[u,w,z]}{\left\langle u-\omega_{17}^{i_0}, w-\omega_3^{i_1}, z-\omega_2^{i_2}\right\rangle}.$$

The size-2 and size-3 cyclic transformations are obvious. For the size-17 cyclic transformation, they applied size-17 Rader's FFT. After moving back to the coefficient ring R, we have

$$\frac{R[x]}{\langle x^{1632}-1\rangle} \cong \prod_{i_0,i_1,i_2} \frac{R[x]}{\left\langle x^{16}-\omega_{17}^{i_0}\omega_3^{i_1}\omega_2^{i_2}\right\rangle} \cong \prod_i \frac{R[x]}{\left\langle x^{16}-\omega_{102}^i\right\rangle}$$

up to a suitable permutation, resulting in 102 instantces of polynomial multiplications of the form $R[x]/\langle x^{16} - \omega_{102}^i \rangle$. 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 separated the computing tasks into two parts: $\prod_{i<96} R[x]/\langle x^{16} - \omega_{102}^i \rangle$ and $\prod_{i\geq 96} R[x]/\langle x^{16} - \omega_{102}^i \rangle$. For $\prod_{i<96} R[x]/\langle x^{16} - \omega_{102}^i \rangle$, they splitted it into

$$\prod_{i<96,2|i} \prod \frac{R[x]}{\langle x^8 \pm \omega_{51}^i \rangle} \times \prod_{i<96,2\perp i} \prod \frac{R[x]}{\langle x^8 \pm \sqrt{2}\omega_{51}^{64i} x^4 + \omega_{51}^{128i} \rangle}$$ [3]

amounting to 96 polynomial multiplications of the form $R[x]/\langle x^8 - \gamma \rangle$ and 96 polynomial multiplications of the form $R[x]/\langle x^8 + \alpha x^4 + \beta \rangle$ for $\alpha, \beta, \gamma \in R$. For the remaining part $\prod_{i\geq 96} R[x]/\langle x^{16} - \omega_{102}^i \rangle$, since there are only six polynomial multiplications, they interleaved the polynomials with don't-cares and applied naïve computation. Our transformation removes this part.

7 Results

7.1 Benchmarking Environment

Intel processors with AVX2. We benchmark our AVX2 implementation on 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 [BBCT22]. 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.

7.2 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 [BBCT22] 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 [HLY24] by factors of 1.25× and 1.29× on Cortex-A72, and 1.25× and 1.36× on Appl1 M1 Pro. In Appendix A, we provide the detailed performance numbers.

[3] The notation $n \perp m$ means that n and m are coprime.

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

AVX2				
	[BBCT22]*	This work	[BBCT22]*	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				
	[HLY24]	This work	[HLY24]*	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.

7.3 Performance of Scheme

Finally, we compare the overall performance of `sntrup761`, and summarize them in Tables 6 and 7.

AVX2 code package(s). For the AVX2-optimized implementation, we integrate our code into the package `libsntrup761` with version `20210608` provided by [BBCT22], and report the 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 [BBCT22] 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 [HLY24].

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. We skip the comparison of key generation on Cortex-A72 and Apple M1 Pro since it is dominated by inversions, which are out of the scope of this paper. 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.

A Profiling of Polynomial Multiplication

This section provides detailed performance profiling. For AVX2 implementations, we report the median cycles of 100,000 iterations on Haswell and Skylake. For Neon implementations, we run all the components with 1,000 iterations and report the median cycles on Cortex-A72 and average cycles on Firestorm.

Table 6: Performance cycles of `sntrup761` with batch key generation using Montgomery's trick on Haswell and Skylake. For the batch key generation, we benchmark with batch size 32.

	[BBCT22]**	SUPERCOP	This work
Haswell			
Batch key generation (amortized)	154 552	-	**136 003**
Encapsulation	-	47 464	**44 108**
Decapsulation	-	56 064	**50 080**
Skylake			
Batch key generation (amortized)	129 159	-	**118 939**
Encapsulation	-	40 653	**36 486**
Decapsulation	-	47 387	**41 070**

** Our own benchmark.

Table 7: Performance cycles of `sntrup761` on Cortex-A72 and Apple M1 Pro.

	[HLY24]***	This work	[HLY24]***	This work
	Cortex-A72		Apple M1 Pro	
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 benchmark.

A.1 AVX2 Performance

Table 8 gives an overview of the number of corresponding function calls in our AVX2 implementation, Table 9 summarizes the performance of truncated size-17 Rader's FFT and radix-$(3, 2)$ butterflies with Good–Thomas FFT, and Table 10 summarizes the transpositions and polynomial multiplications in $R[x]/\langle x^{16} - \zeta \rangle$. For the radix-$(3, 2)$ butterflies, we provide two implementations – one twists the polynomials prior to Good–Thomas FFT and one twists the polynomials after Good–Thomas FFT. For the transpositions, we provide two implementations – one twists the polynomials prior to transposing and one twists the polynomials after transposing.

Table 8: Overview of function calls in our AVX2 implementation.

Function	Count
Truncated Rader-17 for $R[x]/\langle \Phi_{17}(x) \rangle$ $(128\times)$	2
Inverse of truncated Rader-17 for $R[x]/\langle \Phi_{17}(x) \rangle$ $(128\times)$	1
Radix-$(3, 2)$ with pre-twist (256)	2
Rader-$(3, 2)$ with post-twist (256)	1
Twist with pre-transpose	12
Twist with post-transpose	6
Polynomial multiplication in $R[x]/\langle x^{16} - 1 \rangle$ $(16\times)$	3
Polynomial multiplication in $R[x]/\langle x^{16} + 1 \rangle$ $(16\times)$	3

Table 9: AVX2 Performance cycles of butterfly operations. Numbers are medians of 100,000 iterations where each iteration computes with the indicated repetions of algebraic operations.

	Haswell	Skylake
Radix-17		
Truncated Rader-17 for $R[x]/\langle\Phi_{17}(x)\rangle$ (128×)	2 504	1 776
Inverse of truncated Rader-17 for $R[x]/\langle\Phi_{17}(x)\rangle$ (128×)	2 528	1 752
Radix-$(3,2)$		
Radix-$(3,2)$ with pre-twist (256×8)	5 112	3 794
Radix-$(3,2)$ with post-twist (256×8)	4 904	3 550

Table 10: AVX2 Performance of permutations with twisting. Numbers are medians of 100,000 iterations where each iteration computes with the indicated repetions of algebraic operations.

	Haswell	Skylake
Twist with pre-transpose (8×, 256 coeff. each)	768	632
Twist wit post-transpose (8×, 256 coeff. each)	720	618

A.2 Neon Performance

Table 12 gives an overview of the number of corresponding function calls in our Neon implementation, Table 13 summarizes the performance of truncated size-17 Rader's FFT and radix-$(3,2)$ butterflies with Good–Thomas FFT, and Table 14 summarizes the performance of polynomial multiplications in $R[x]/\langle x^{16} \pm 1\rangle$. For the radix-$(3,2)$ butterflies, we perform two layers of twisting – we twist the polynomials, apply Good–Thomas FFT, and twist the results again.

B Future Works

This work shows that polynomial multiplication in a ring lacking common beliefs of friendliness measures for implementations, truncated Rader's, Good–Thomas, and Bruun's FFT are more favorable than Schönhage's and Nussbaumer's FFTs.

There are several future works for parameter set `sntrup761`. An immediate one is to generate several multipliers of sizes $2^{i_0}3^{i_1}5^{i_2}$ based on this work for FFT-based fast constant-time GCD computation [BY19]. Additionally, an ambitious goal is to explore various possible vectorized multipliers for other NTRU Prime parameter sets. We first draft vectorized polynomial multipliers for `ntrulpr857/sntrup857` based on various ideas presented in this paper and give theoretical estimations for the resulting performance. Additionally, we also briefly draft some ideas for `ntrulpr1013/sntrup1013` and `ntrulpr1277/sntrup1277`.

B.1 Theoretical Estimations for `ntrulpr857/sntrup857`

Definition of a meaningful theoretical estimation. Before jumping to the technical details, we first define the terminology "meaningful theoretical esitmation" for clarity since from submission experience, reviewers had different interpretations of this term. We call

Table 11: AVX2 Performance cycles of power-of-two base multiplications. Numbers are medians of 100,000 iterations where each iteration computes with the indicated repetions of algebraic operations. Karatsuba for $R[x]/\langle x^{16}-\zeta\rangle$ is only involved in our development and not used in our implementation.

	Approach	Haswell	Skylake
$R[x]/\langle x^{16}-1\rangle$ (128×)	Cooley–Tukey	3 168	2 576
$R[x]/\langle x^{16}+1\rangle$ (128×)	Bruun	4 720	3 616
$R[x]/\langle x^{16}-\zeta\rangle$ (128×)	Karatsuba	5 588	4 428

Table 12: Overview of function calls in our Neon implementation.

Function	Count
Truncated Rader-17 for $R[x]/\langle\Phi_{17}(x)\rangle$ (128×)	2
Inverse of truncated Rader-17 for $R[x]/\langle\Phi_{17}(x)\rangle$ (128×)	1
Radix-$(3,2)$ with twists (256)	3
Polynomial multiplication in $R[x]/\langle x^{16}-1\rangle$ (16×)	3
Polynomial multiplication in $R[x]/\langle x^{16}+1\rangle$ (16×)	3

a performance number **a theoretical estimation** of a computation if someone claims that target computation runs in the estimated cycles. Not all theoretical estimations are meaningful in the common sense. For example, claiming that multiplying two 1024-bit integers takes $10^{10^{10^{10^{10}}}}$ Firestorm cycles is not meaningful even when Apple M1 Pro is broken and repairing time is counted! We call a theoretical estimation of a computation **meaningful** if the computation can be constructed from already implemented computations, up to replacements of tables of constants. In the following, we propose our vectorized polynomial multipliers and deliver meaningful theoretical estimations of Haswell and Cortex-A72 cycles.

Overview of our proposed vectorized polynomial multipliers. For the parameter sets `ntrulpr857/sntrup857`, we want to multiply polynomials in $\mathbb{Z}_{\mathbb{Z}_{5167}[x]}/\langle x^{857}-x-1\rangle$. We propose to multiply in $\mathbb{Z}_{5167}[x]/\langle\Phi_7(x^{288})\rangle$ with truncated Rader's, Good–Thomas, and Bruun's FFTs. Since $5167-1=2\cdot 3^2\cdot 7\cdot 41$ and $5167+1=2^4\cdot 17\cdot 19$, we can define principal roots ω_7, ω_2, ω_9, and $\mathbb{Z}_{5167}[x]/\langle x^{16}+1\rangle$ splits into eight trinomials. We first compute the isomorphism

$$\frac{\mathbb{Z}_{5167}[x]}{\langle\Phi_7(x^{288})\rangle}\cong\prod_{i=1,\ldots,6}\frac{\mathbb{Z}_{5167}[x]}{\langle x^{288}-\omega_7^i\rangle}\cong\left(\frac{\mathbb{Z}_{5167}[x]}{\langle x^{288}-1\rangle}\right)^6$$

with truncated size-7 Rader's FFT and twisting. We then apply Good–Thomas FFT turning a size-18 cyclic DFT into a tensor product of a size-2 cyclic DFT and a size-9 cyclic DFT. The size-9 cyclic DFT is then implemented with Cooley–Tukey FFT using radix-3 butterflies. After applying the size-18 cyclic DFT, we twist all the polynomial rings into cyclic and negacyclic ones. Below is an illustration:

$$\frac{\mathbb{Z}_{5167}[x]}{\langle x^{288}-1\rangle}\cong\prod_{i_0,i_1}\frac{\mathbb{Z}_{5167}[x]}{\langle x^{16}-\omega_2^{i_0}\omega_9^{i_1}\rangle}\cong\prod_{i_0}\left(\frac{\mathbb{Z}_{5167}[x]}{\langle x^{16}-\omega_2^{i_0}\rangle}\right)^9.$$

Table 13: Neon Performance cycles of butterfly operations. Numbers are medians on Cortex-A72 and averages on Firestorm of 1,000 iterations where each iteration computes with the indicated repetions of algebraic operations.

	Cortex-A72	Firestorm
Radix-17		
Truncated Rader-17 for $R[x]/\langle\Phi_{17}(x)\rangle$ (96×)	4 369	1 016
Inverse of truncated Rader-17 for $R[x]/\langle\Phi_{17}(x)\rangle$ (96×)	4 379	1 016
Radix-$(3,2)$		
Radix-$(3,2)$ with twistings (256×)	3 184	505

Table 14: Neon Performance cycles of power-of-two base multiplications. Numbers are medians on Cortex-A72 and averages on Firestorm of 1,000 iterations where each iteration computes with the indicated repetions of algebraic operations.

	Approach	Cortex-A72	Firestorm
$R[x]/\langle x^{16}-1\rangle$ (96×)	Cooley–Tukey + Toeplitz	2 590	788
$R[x]/\langle x^{16}+1\rangle$ (96×)	Toeplitz	3 670	1 144

The remaining problems are 54 polynomial multiplications in each of $\mathbb{Z}_{5167}[x]/\langle x^{16}-1\rangle$ and $\mathbb{Z}_{5167}[x]/\langle x^{16}+1\rangle$. In Neon, we simply reuse the implementations for `sntrup761`. As for AVX2, we interleave 48 polynomials in $\mathbb{Z}_{5167}[x]/\langle x^{16}-1\rangle$ and 48 polynomials in $\mathbb{Z}_{5167}[x]/\langle x^{16}+1\rangle$, and apply AVX2-optimized Cooley–Tukey and Bruun's FFT as shown in Section 6.2. For the remaining 12 polynomial multiplications in $\mathbb{Z}_{5167}[x]/\langle x^{16}\pm 1\rangle$, we interleave them with four don't-care polynomials and apply AVX2-optimized Karatsuba.

Meaningful theoretical estimations for the performance. We estimate the performance of $\mathbb{Z}_{5167}[x]/\langle\Phi_7(x^{288})\rangle$ as follows. For the truncated Rader-7 FFT, the performance is the same as a size-6 cyclic convolution. We overestimate the size-6 cyclic convolution with two size-6 cyclic FFTs followed by twisting since one of the operands contains only publicly known constants where precomputation is allowed. We regard this as an overestimation since there could be faster way for the size-6 cyclic convolution. For the follow up size-18 cyclic DFT via Good–Thomas and Cooley–Tukey FFTs, since the performance is no worsee than a size-36 cyclic DFT and one size-36 cyclic DFT can be implemented as two layers of size-6 cyclic FFTs followed by twisting, we again overestimate the performance of size-18 cyclic DFT with two size-6 cyclic FFTs followed by twisting. Based on the performance numbers in Tables 9 and 13, we overestimate the performance cycles of

$$\frac{\mathbb{Z}_{5167}[x]}{\langle\Phi_7(x^{288})\rangle}\cong\left(\prod\frac{\mathbb{Z}_{5167}[x]}{\langle x^{16}\pm 1\rangle}\right)^{54}$$

as $5112\cdot\frac{1}{256\cdot 8}\cdot\frac{1728}{6}\cdot 4=2875.5$ Haswell cycles and $3184\cdot\frac{1}{256}\cdot\frac{1728}{6}\cdot 4=14328.0$ Cortex-A72 cycles. For the performance estimation of size-16 cyclic/negacyclic polynomial multiplications with AVX2, we estimate it as the sum of performance numbers for 48 Cooley–Tukey FFT for $\mathbb{Z}_{5167}[x]/\langle x^{16}-1\rangle$, 48 Bruun's FFT for $\mathbb{Z}_{5167}[x]/\langle x^{16}+1\rangle$, and 16 Karatsuba for $\mathbb{Z}_{5167}[x]/\langle x^{16}-\zeta\rangle$ where ζ could be different among the 16 polynomials. Therefore, the performance of size-16 polynomial multiplication is $3168\cdot\frac{48}{128}+4720\cdot\frac{48}{128}+5588\cdot\frac{16}{128}=3656.5$ Haswell cycles. On the other hand, for the Neon implementation, we

simply reuse the implementations for `sntrup761` and estimate as $(2590 + 3670) \cdot \frac{1}{1536} \cdot 1728 = 7042.5$ Cortex-A72 cycles. Finally, the remaining estimation is about interleaving polynomials for the AVX2 implementation. Since $1728 < 256 \cdot 7$, we overestimate the performance as $768 \cdot \frac{7}{8} = 672$. In summary, the overall overestimation of the performance is $2875.5 \cdot 3 + 672 \cdot 3 + 3656.5 = 14299$ Haswell cycles and $14328.0 \cdot 3 + 7042.5 = 50026.5$ Cortex-A72 cycles.

C References

[ACC+21] Erdem Alkim, Dean Yun-Li Cheng, Chi-Ming Marvin Chung, Hülya Evkan, Leo Wei-Lun Huang, Vincent Hwang, Ching-Lin Trista Li, Ruben Niederhagen, Cheng-Jhih Shih, Julian Wälde, and Bo-Yin Yang. Polynomial Multiplication in NTRU Prime Comparison of Optimization Strategies on Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(1):217–238, 2021. https://tches.iacr.org/index.php/TCHES/article/view/8733. 19, 20

[BBC+20] Daniel J. Bernstein, Billy Bob Brumley, Ming-Shing Chen, Chitchanok Chuengsatiansup, Tanja Lange, Adrian Marotzke, Bo-Yuan Peng, Nicola Tuveri, Christine van Vredendaal, and Bo-Yin Yang. NTRU Prime. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS], 2020. https://ntruprime.cr.yp.to/. 3

[BBCT22] Daniel J. Bernstein, Billy Bob Brumley, Ming-Shing Chen, and Nicola Tuveri. OpenSSLNTRU: Faster post-quantum TLS key exchange. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 845–862, 2022. 2, 3, 4, 7, 20, 21, 22, 23, 24

[BDL+12] Daniel J Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of cryptographic engineering*, 2(2):77–89, 2012. 8

[Ber22] Daniel J. Bernstein. Fast norm computation in smooth-degree abelian number fields. Cryptology ePrint Archive, Paper 2022/980, 2022. https://eprint.iacr.org/2022/980. 2, 11

[BGM93] Ian F. Blake, Shuhong Gao, and Ronald C. Mullin. Explicit Factorization of $x^{2^k} + 1$ over $\mathbb{F}_p$ with Prime $p \equiv 3 \bmod 4$. *Applicable Algebra in Engineering, Communication and Computing*, 4(2):89–94, 1993. 12

[BHK+22] Hanno Becker, Vincent Hwang, Matthias J. Kannwischer, Bo-Yin Yang, and Shang-Yi Yang. Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):221–244, 2022. https://tches.iacr.org/index.php/TCHES/article/view/9295. 9, 13, 15

[Bou89] Nicolas Bourbaki. *Algebra I*. Springer, 1989. 4

[Bru78] Georg Bruun. z-transform DFT Filters and FFT's. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 26(1):56–63, 1978. 12

[BY19] Daniel J. Bernstein and Bo-Yin Yang. Fast constant-time gcd computation and modular inversion. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(3):340–398, 2019. https://tches.iacr.org/index.php/TCHES/article/view/8298. 4, 25

[CCHY23] Han-Ting Chen, Yi-Hua Chung, Vincent Hwang, and Bo-Yin Yang. Algorithmic Views of Vectorized Polynomial Multipliers – NTRU, 2023. To appear at Indocrypt 2023, currently available at https://eprint.iacr.org/2023/1637. 8, 9

[CT65] James W. Cooley and John W. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of Computation*, 19(90):297–301, 1965. 10

[FLP+18] Franz Franchetti, Tze Meng Low, Doru Thom Popovici, Richard M Veras, Daniele G Spampinato, Jeremy R Johnson, Markus Püschel, James C Hoe, and José MF Moura. Spiral: Extreme performance portability. *Proceedings of the IEEE*, 106(11):1935–1968, 2018. https://ieeexplore.ieee.org/document/8510983. 7

[Goo58] I. J. Good. The Interaction Algorithm and Practical Fourier Analysis. *Journal of the Royal Statistical Society: Series B (Methodological)*, 20(2):361–372, 1958. 10

[HLY24] Vincent Hwang, Chi-Ting Liu, and Bo-Yin Yang. Algorithmic Views of Vectorized Polynomial Multipliers – NTRU Prime. 2024. To appear at ACNS 2024, currently available at https://eprint.iacr.org/2023/1580. 2, 3, 7, 12, 20, 21, 22, 23, 24

[Hwa22] Vincent Bert Hwang. Case Studies on Implementing Number–Theoretic Transforms with Armv7-M, Armv7E-M, and Armv8-A. Master's thesis, 2022. https://github.com/vincentvbh/NTTs_with_Armv7-M_Armv7E-M_Armv8-A. 8

[IKPC22] Írem Keskinkurt Paksoy and Murat Cenk. Faster NTRU on ARM Cortex-M4 with TMVP-based multiplication. 2022. https://ia.cr/2022/300. 8

[Jac12a] Nathan Jacobson. *Basic Algebra I.* Courier Corporation, 2012. 4

[Jac12b] Nathan Jacobson. *Basic Algebra II.* Courier Corporation, 2012. 4

[KO62] Anatolii Alekseevich Karatsuba and Yu P Ofman. Multiplication of many-digital numbers by automatic computers. In *Doklady Akademii Nauk*, volume 145(2), pages 293–294, 1962. 13

[Mon85] Peter L. Montgomery. Modular Multiplication Without Trial Division. *Mathematics of computation*, 44(170):519–521, 1985. 9

[Mur96] Hideo Murakami. 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*, volume 3, pages 1311–1314, 1996. 12

[NIS] NIST, the US National Institute of Standards and Technology. Post-quantum cryptography standardization project. https://csrc.nist.gov/Projects/post-quantum-cryptography. 28

[Nus80] Henri Nussbaumer. Fast Polynomial Transform Algorithms for Digital Convolution. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 28(2):205–215, 1980. 12

[Rad68] Charles M. Rader. Discrete fourier transforms when the number of data samples is prime. *Proceedings of the IEEE*, 56(6):1107–1108, 1968. 10

[Sch77] Arnold Schönhage. Schnelle multiplikation von polynomen über körpern der charakteristik 2. *Acta Informatica*, 7(4):395–398, 1977. 12

[Win78] Shmuel Winograd. On Computing the Discrete Fourier Transform. *Mathematics of computation*, 32(141):175–199, 1978. 10