Pushing the Limit of Vectorized Polynomial Multiplication for NTRU Prime

Vincent Hwang

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

Abstract. This paper implements a vectorized polynomial multiplication for the NTRU Prime parameter sets `ntrulpr761/sntrup761` with AVX2. We explore various fast Fourier transformations (FFTs) for multiplying polynomials in $\mathbb{Z}_{4591}[x]/\left\langle x^{761}-x-1\right\rangle$. The polynomial ring $\mathbb{Z}_{4591}[x]/\left\langle x^{761}-x-1\right\rangle$ is a finite field and does *not* enjoy common beliefs on friendliness measure for implementations. Commonly, people believe that radix-2 Cooley–Tukey FFT and other FFTs with the same definibality ($2^k|(q-1)$ for the coefficient ring $\mathbb{Z}_q$ with prime q) are fast and easy to vectorization. We show that this belief should be extended to include the following: (i) Bruun's FFT exploiting the power-of-two factor of $q+1$ if $q \equiv 3 \bmod 4$; (ii) truncated Rader's FFT exploiting the prime factor of $q-1$. We qualify the prime 4591 as FFT and vectorization-friendly and find that most NTRU Prime parameter sets enjoy friendliness measures.
Compared to the state-of-the-art AVX2-optimized implementation by [Bernstein, Brumley, Chen, and Tuveri, USENIX Security 2022], our big-by-big polynomial multiplication is 1.99× and 2.16× faster on Haswell and Skylake, respectively. We port our implementation to the scheme `sntrup761`. For the batch key generation with batch size 32, we reduce the amortized cost by 12% on Haswell and 8% on Skylake. For encapsulation, we reduce the performance cycles by 7% on Haswell and 10% on Skylake. Finally, for the decapsulation, we reduce the performance cycles by 10% on Haswell and 13% on Skylake.

Keywords: NTRU Prime · AVX2 · Good–Thomas FFT · Rader's FFT · Bruun's FFT · Truncation

1 Introduction

OpenSSH 9.0 currently uses the hybrid `sntrup761x25519-sha512` key exchange by default[1]. This paper explores various insights on designing transformations that are suitable for vectorization. We implement AVX2-optimized polynomial multiplication for NTRU Prime parameter sets `ntrulpr761/sntrup761` as a proof of concept on identifying algebraic structures that are friendly for implementations. Our target is the polynomial multiplication in $\mathbb{Z}_{4591}[x]/\left\langle x^{761}-x-1\right\rangle$ used by `ntrulpr761/sntrup761`. We refer to [BBC+20] for the specification of NTRU Prime. For `ntrulpr761/sntrup761`, maintaining the vectorization–friendliness while working over $\mathbb{Z}_{4591}$ was challenging. While computing the product of two polynomials, if one of the polynomials has coefficients within a small range, we call the computing task a big-by-small polynomial multiplication. Otherwise, we call it a big-by-big polynomial multiplication. In NTRU Prime, all the polynomial multiplications in the reference implementation are big by small. Nevertheless, big-by-big polynomial multiplications are used for improving the key generation of `sntrup` [BY19, BBCT22] and can replace big-by-small polynomial multiplications if the performance is improved.

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

In the context of FFT-based polynomial multiplication over coefficient ring $\mathbb{Z}_q$ with prime factorization $q = \prod_h p_h^{d_h}$, one can show that a size-n cyclic discrete Fourier transform (DFT) $\mathbb{Z}_q[x]/\langle x^n - 1\rangle \cong \prod_i \mathbb{Z}_q[x]/\langle x - \omega_n^i\rangle$ is definable if and only if n divides $\gcd(p_h - 1)$ [Pol71, AB74]. See [Pol76, DV78, Für09] for conditions in the ring setting. For `ntrulpr761/sntrup761`, since $4591 = 2 \cdot 3^3 \cdot 5 \cdot 17 + 1$ is a prime, we can only define a cyclic DFT of size n for $n|(2 \cdot 3^3 \cdot 5 \cdot 17)$. [ACC+21] computed size-1530 and size-1620 cyclic convolutions by applying size-153 and size-270 transformations while working over $\mathbb{Z}_{4591}$ for big-by-big polynomial multiplication on Cortex-M4. Both implementations are obviously not suitable for vectorization – the largest power-of-two factors are 2 for 1530 and 4 for 1620. They also proposed to compute size-1536 cyclic convolution for big-by-small polynomial multiplication by switching the coefficient ring to a large ring $\mathbb{Z}_{q'}$ bounding the product's maximum value in $\mathbb{Z}$ for a prime q' with $512|(q' - 1)$. [ACC+21] showed that big-by-big polynomial multiplication is slightly faster than big-by-small one on Cortex-M4 implementing limited SIMD support.

When moving to an architecture implementing a powerful vector instruction set, big-by-small polynomial multiplication can be vectorized in an obvious way. [BBCT22] implemented a AVX2-optimized big-by-small polynomial multiplication by operating over $\mathbb{Z}_{7681}[x]/\langle x^{1536} - 1\rangle$ and $\mathbb{Z}_{10753}[x]/\langle x^{1536} - 1\rangle$. However, big-by-small polynomial multiplications require the smallness of coefficients.

For big-by-big polynomial multiplications (no smallness assumption), [BBCT22, Section 3.3, Paragraph "Problem description and related multiplication."] considered three possible options: (i) In addition to the moduli 7681 and 10753, we also compute the radix-2 FFT over 12289. (ii) Applying a radix-17 algorithm with Rader's FFT as did in [ACC+21]. (iii) Applying radix-2 Schönhage and Nussbaumer.

For (i), this amounts to at least a factor of 1.5 blow up. The actual overhead might be larger since 12289, the smallest 16-bit prime candidate supporting a size-512 cyclic FFT, requires more modular reductions [BBCT22].

For (ii), [BBCT22] argued that it is unfriendly for radix-2 NTT. We quote the corresponding paragraphs below.

> Secondly, q from the NTRU Prime parameter set is not a radix-2 NTT friendly prime. For example, $q = 4591$ in `sntrup761`, and since $4591 - 1 = 2 \cdot 3^3 \cdot 5 \cdot 17$, no simple root of unity is available for recursive radix-2 FFT tricks. Alkim, Cheng, Chung, Evkan, Huang, Hwang, Li, Niederhagen, Shih, Wälde, and Yang [ACC+21] presented a non-radix-2 NTT implementation on $(\mathbb{Z}/4591)[x]/(x^{1530} - 1)$ for embedded systems. They performed radix-3, radix-5, and radix-17 NTT stages in their NTT. We instead use a radix-2 algorithm that efficiently utilizes the full `ymm` registers in the Haswell architecture.

[BBCT22] eventually implemented (iii) – applying radix-2 Schönhage and Nussbaumer and computing in $\mathbb{Z}_{4591}[x]/\langle (x^{1024} + 1)(x^{512} - 1)\rangle$. Since Schönhage and Nussbaumer double the number of coefficients and interprate the movement of coefficients as multiplications by roots of unity, [BBCT22] eventually resulted in $\frac{4 \cdot 1536}{8} = 768$ size-8 base multiplications (small-dimensional polynomial multiplications). Their big-by-big polynomial multiplication is roughly 1.5× slower than their big-by-small one.

Recently, [CCH+23] explored various vectorization ideas for NTRU and NTRU Prime on a Cortex-A72 with Neon. Among the various vectorized transformations, we are interested in their `Good-Rader-Bruun`. `Good-Rader-Bruun` multiplies polynomials in $\mathbb{Z}_{4591}[x]/\langle x^{1632} - 1\rangle$, and introduces no doubling of coefficients while maintaining the vectorization-friendliness. This results in 6 size-16 base multiplications and 192 size-8 base multiplications.

Contributions. We summarize our contributions as follows.

- Let $\Phi_{17}(x^{96}) = 1 + x^{96} + \cdots x^{96\cdot 16}$. We improve [CCH+23]'s transformation choice, and propose an AVX2 implementation multiplying polynomials in $\mathbb{Z}_{4591}[x]/\langle\Phi_{17}(x^{96})\rangle$ with truncated Rader's, Good–Thomas, and Bruun's FFT. Compared to the state-of-the-art AVX2-optimized implementation by [BBCT22], our big-by-big polynomial multiplication is 1.99× and 2.16× faster on Haswell and Skylake, respectively.
- We formalize two friendliness measures — vectorization–friendliness and permutation–friendliness — to rigorously argue about the architectural aspect of algebra monomorphisms with vector arithmetic.
- We provide the benchmark of the full Streamlined NTRU Prime with batch key generation and our improved polynomial multiplication. Comparing to the state-of-the-art [BBCT22], the performance cycles of batch key generation with batch size 32 are reduced by 12% on Haswell and 8% on Skylake. For the encapsulation, the performance cycles are reduced by 7% on Haswell and 10% on Skylake. As for the decapsulation, the performance cycles are reduced by 10% on Haswell and 13% on Skylake.

Code. Our source code can be found at https://github.com/vincentvbh/NTRU_Prime_polymul_AVX2 under CC0 license.

Structure of this paper. This paper is structured as follows: Section 2 describes modular arithmetic, Section 3 describes various algebraic techniques, Section 4 formalizes the need of vectorization, and Section 5 describes our choice of transformation and compares the design choice to existing vectorized polynomial multiplications for NTRU Prime. Section 6 shows the performance of our polynomial multiplication, and Section 7 drafts several possible future works, including the deployment of FFT-based constant-time GCD [BY19] and applications to other NTRU Prime parameter sets.

2 Modular Multiplication and Reduction

2.1 Definitions and Notations

Let q be a positive integer. We call two integers a, b equivalent modulo q if their difference is a multiple of q, and denote with $a \equiv b \pmod q$. For the set $\mathbb{Z}$ of integers, we can partition $\mathbb{Z}$ into q sets such that integers in the same partition are equivalent modulo q. The set formed by collecting these q sets is denoted by $\mathbb{Z}_q$. In each partition P, we call integers in P representatives of P. A common way for enumerating the members of $\mathbb{Z}_q$ is to select one representative from each partition. If we define $\mathbb{Z}_q \coloneqq [0, q) \cap \mathbb{Z}$, we call the corresponding arithmetic unsigned, and define $\bmod^+ q : \mathbb{Z} \to \mathbb{Z}_q$ as the function satisfying $\forall z \in \mathbb{Z}, z \equiv z \bmod^+ q \pmod q$. On the other hand, if we define $\mathbb{Z}_q \coloneqq \left[-\frac{q}{2}, \frac{q}{2}\right) \cap \mathbb{Z}$, we call the corresponding arithmetic signed, and define $\bmod^\pm q : \mathbb{Z} \to \mathbb{Z}_q$ as the function satisfying $\forall z \in \mathbb{Z}, z \equiv z \bmod^\pm q \pmod q$. Additionally, we define floor function $\lfloor\rfloor : \mathbb{R} \to \mathbb{Z}$ as the function mapping a real number r to the largest integer lower-bounding r, and rounding function $\lfloor\rceil : \mathbb{R} \to \mathbb{Z}$ as $r \mapsto \lfloor r + 0.5 \rfloor$.

2.2 AVX2 Implementations

In AVX2, each vector register holds 256-bit of data. Since we are only dealing with 16-bit coefficients, each vector register holds 16 coefficients. We recall the vectorized Montgomery multiplication [Mon85] and Barrett reduction [Bar86] from [Sei18]. `vpmullw` multiplies corresponding signed 16-bit values and places the lower 16-bit values of the products to

the destination register. `vpmulhw` places the upper 16-bit values to the destination instead. `vpmulhrsw` effectively computes $\left\lfloor \frac{ab}{2^{15}} \right\rceil$ from the signed 16-bit values a and b.

For signed 16-bit values a and b, Montgomery multiplication [Mon85, Sei18] computes a representative of $ab2^{-16} \bmod^{\pm} q$ with

$$\left\lfloor \frac{ab}{2^{16}} \right\rfloor - \left\lfloor \frac{\left(abq' \bmod^{\pm} 2^{16}\right) q}{2^{16}} \right\rfloor \equiv ab2^{-16} \pmod{q}$$

where $q' = q^{-1} \bmod^{\pm} 2^{16}$. Algorithm 1 is an illustration. If b is known in prior, we replace $(b, bq' \bmod^{\pm} 2^{16})$ with $\left(b2^{16} \bmod^{\pm} q, \left(b2^{16} \bmod^{\pm} q\right) q' \bmod^{\pm} 2^{16}\right)$ to save one multiplication and mitigate the scaling by 2^{-16}. Algorithm 2 is an illustration.

Barrett reduction [Bar86, Sei18] reduces a signed 16-bit value a by computing

$$a - \left\lfloor \frac{a \left\lfloor \frac{2^{15}}{q} \right\rceil}{2^{15}} \right\rceil q \equiv a \pmod{q}.$$

Algorithm 3 is an illustration. In the case of $q = 4591$, one can show (by brute-force testing) that for $a \in [-32768, 32767]$, the results lies in $[-2881, 2881]$.

Algorithm 1 Montgomery multiplication [Sei18].

Inputs: $\mathtt{a} = a, \mathtt{b} = b$.
Constants: $q = 4591$, $q' = q^{-1} \bmod^{\pm} 2^{16} = 15631$.
Output: $\mathtt{c} = \left\lfloor \frac{ab}{2^{16}} \right\rfloor - \left\lfloor \frac{\left(abq' \bmod^{\pm} 2^{16}\right) q}{2^{16}} \right\rfloor \equiv ab2^{-16} \bmod^{\pm} q \pmod{q}$.

```
1: vpmullw  b, q', lo
2: vpmullw lo,  a, lo
3: vpmulhw  b,  a, hi
4: vpmulhw lo,  q, lo
5: vpsubw  lo, hi,  c
```

Algorithm 2 Montgomery multiplication with precomputation [Sei18].

Input: $\mathtt{a} = a$.
Constants: $q = 4591$, $\mathtt{b} = b2^{16} \bmod^{\pm} q$, $\mathtt{b'} = \left(b2^{16} \bmod^{\pm} q\right) q^{-1} \bmod^{\pm} 2^{16}$.
Output: $\mathtt{c} = \left\lfloor \frac{a\left(b2^{16} \bmod^{\pm} q\right)}{2^{16}} \right\rfloor - \left\lfloor \frac{\left(a\left(\left(b2^{16} \bmod^{\pm} q\right)q^{-1} \bmod^{\pm} 2^{16}\right)\right)q}{2^{16}} \right\rfloor \equiv ab \bmod^{\pm} q \pmod{q}$.

```
1: vpmullw b',  a, lo
2: vpmulhw  b,  a, hi
3: vpmulhw lo,  q, lo
4: vpsubw  lo, hi,  c
```

Algorithm 3 Barrett reduction [Sei18].

Input: $\mathtt{a} = a$.
Constants: $q = 4591$, $\bar{q} = \left\lfloor \frac{2^{15}}{q} \right\rceil = 7$.
Output: $\mathtt{a} = a - \left\lfloor \frac{a\bar{q}}{2^{15}} \right\rceil q$, $-2881 \le a - \left\lfloor \frac{a\bar{q}}{2^{15}} \right\rceil q \le 2881$.

```
1: vpmulhrsw  a, q̄, hi
2: vpmullw   hi, q, hi
3: vpsubw    hi, a,  a
```

3 Transformations

This section reviews various algebraic techniques, including Chinese remainder theorem in Section 3.2, truncation in Section 3.3, Cooley–Tukey FFT in Section 3.4, Good–Thomas FFT in Section 3.5, Rader's FFT in Section 3.6, Bruun's FFT in Section 3.7, twisting and composed multiplication in Section 3.8, and Karatsuba in Section 3.9.

3.1 Definitions and Notations

In this paper, we assume all readers are familiar with some basic algebraic structures, including, monoids, groups, rings, modulues, and associative algebras. We go through an informal introduction of them in this section.

Monoids, groups, rings. A monoid M is a set equipped with a binary associative operator $\cdot_M$ admitting an identity element 1_M. If $\cdot_M$ is commutative, we write it $+_M$, denote 0_M for the identity element, and call M a commutative monoid. If each elements in M admits an inverse element, we call M a group. Let G be a group. If there is an element $g \in G$ such that $G = \left\{g^i\right\}$, we call G a cyclic group. Furthermore, if the binary associative operator is commutative, we call G an abelian group. Obviously, cyclic groups are abelian. For an abelian group G, if we additionally find a binary associative operator $\cdot_G$ with 1_G satisfying the left and right distributivity over the commutative operator $+_G$, we call $(G, \cdot_G, 1_G)$ a ring and G its underlying additive group. For a ring R, if $\cdot_R$ is commutative, we call R a commutative ring. In this paper, all rings are commutative. For a subgroup I of the underlying additive group of a ring R, we call I an ideal if $\forall r \in R, \forall a \in I, ra \in I$. If $I = rR \coloneqq \{ra | a \in R\}$ for an $r \in R$, we denote $I = \langle r \rangle$. For an ideal I of a ring R, the set $R/I \coloneqq \{\{r + a | a \in I\} | r \in R\}$ is a ring called quotient ring. A straightforward example of rings is $\mathbb{Z}$ with the usual addition and multiplication as the binary associative operators. For a positive integer q, the set of q-multiples $q\mathbb{Z} = \{qz | z \in \mathbb{Z}\}$ is an ideal. We denote the quotient ring $\mathbb{Z}/q\mathbb{Z}$ as $\mathbb{Z}_q$ in this paper. Notice that $\mathbb{Z}_q$ is in fact a cyclic group of q elements. Since cyclic groups containing the same number of elements are isomorphic in an obvious way, $\mathbb{Z}_q$ is also used for denoting a cyclic group of q elements.

Modules. For an abelian group M, a ring R, and a map $\cdot_{R\times M} : R \times M \to M$, we call $(M, \cdot_{R\times M})$ a left R-module if $\forall r, s \in R, \forall a, b \in M$, (i) $r \cdot_{R\times M} (a +_M b) = r \cdot_{R\times M} a +_M r \cdot_{R\times M} b$ (distributivity over addition in M), (ii) $(r +_R s) \cdot_{R\times M} a = r \cdot_{R\times M} a +_M r \cdot_{R\times M} b$ (distributivity over addition in R), (iii) $(r \cdot_R s) \cdot_{R\times M} a = r \cdot_{R\times M} (s \cdot_{R\times M} a)$ (associativity of $\cdot_R$ and $\cdot_{R\times M}$), and (iv) $1_R \cdot_{R\times M} a = a$ (compatibility of the multiplicative identity 1_R). For a ring R, an immediate example of an R-module is the set R^n: for an $r \in R$ and an n-tuple $(a_i) \in R^n$, we define $r(a_i)$ as (ra_i).

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: (i) $R[x]/\langle \boldsymbol{g} \rangle = R^n$ as R-modules, and (ii) $R[x]/\langle \boldsymbol{g} \rangle$ is a ring. We give more examples that are relevant to this paper. For a group G of n elements, we define $R[G]$ as the set of elements of the form

$$\sum_i r_i g_i$$

for $r_i \in R$ and $g_i \in G$. Obviously, we have $R[G] = R^n$ as R-modules. It remains to verify the ring structure of $R[G]$. We turn $R[G]$ into a ring by defining

$$\left(\sum_i a_i g_i\right)\left(\sum_i b_i g_i\right) \coloneqq \sum_i \left(\sum_{g_j g_k = g_i} a_j b_k\right) g_i.$$

Immediately, if $G = \mathbb{Z}_n$ is a cyclic group of n elements, we have $R[\mathbb{Z}_n] \cong R[x]/\langle x^n - 1\rangle$ as R-algebras. In this paper, $R[\mathbb{Z}_n]$ and $R[x]/\langle x^n - 1\rangle$ are used interchangeably. For two R-algebras $\mathcal{A}, \mathcal{B}$, we can form an R-algebra $\mathcal{A} \otimes \mathcal{B}$ called tensor product of $\mathcal{A}$ and $\mathcal{B}$. We refer to [Bou89, Section 3, Chapter II] and [Jac12, Section 3.9] for definition. If we have a group isomorphism $G \cong G_0 \times G_1$, then $R[G] \cong R[G_0] \otimes R[G_1]$ as R-algebras and isomorphisms defined on $R[G]$ might be much faster in terms of computational complexity if we first move to $R[G_0] \otimes R[G_1]$. For example, for coprime positive integers q_0, q_1, the group isomorphism $\mathbb{Z}_{q_0 q_1} \cong \mathbb{Z}_{q_0} \times \mathbb{Z}_{q_1}$ implies the algebra isomorphism $R[\mathbb{Z}_{q_0 q_1}] \cong R[\mathbb{Z}_{q_0}] \otimes R[\mathbb{Z}_{q_1}]$, and for an arbitrary isomorphism defined on $R[\mathbb{Z}_{q_0 q_1}]$, there is a corresponding tensor product of isomorphisms defined on $R[\mathbb{Z}_{q_0}] \otimes R[\mathbb{Z}_{q_1}]$ by tensor-hom adjunction [Jac12, Exercise 3, Section 3.8]. Such a tensor product of isomorphisms usually admits faster computation. We will give more details in Section 3.5.

3.2 Chinese Remainder Theorem

Let R be a ring. For elements $e_0, e_1 \in R$, we call them orthogonal if $e_0 e_1 = 0$. An element $e \in R$ is called idempotent if $e^2 = e$. For orthogonal idempotent elements e_0 and e_1 in R satisfying $e_0 + e_1 = 1$, we have the ring isomorphism $R \cong R/(1-e_0)R \times R/(1-e_1)R$. This easily generalizes to finitely many orthogonal idempotent elements $(e_0, \ldots, e_{d-1})$ with $\sum_i e_i = 1$ realizing $R \cong \prod_i R/(1-e_i)R$. Explicitly, we have the isomorphism $R \to \prod_i R/(1-e_i)R$ mapping a to the n-tuple $(a \bmod (1-e_i)R)$ with the inverse $(\hat{a}_i) \mapsto \sum_i \hat{a}_i e_i$ [Bou89]. This is the idempotent-element-based approach to the Chinese remainder theorem for rings.

We are interested in two cases: $R[x]\Big/\left\langle \prod_{i_0,\ldots,i_{h-1}} \boldsymbol{g}_{i_0,\ldots,i_{h-1}} \right\rangle$ for coprime polynomials $\boldsymbol{g}_{i_0,\ldots,i_{h-1}}$'s in $R[x]$ and $\mathbb{Z}_{q_0 \cdots q_{d-1}}$ for coprime integers $q_0, \ldots, q_{d-1}$. For the former, since $\boldsymbol{g}_{i_0,\ldots,i_{h-1}}$'s are coprime, we can split in a "layer-by-layer" fashion by moving indices i_j's from the ideal part to the product-ring part as follows:

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

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. For the later case $\mathbb{Z}_{q_0 \cdots q_{d-1}} \cong \prod_i \mathbb{Z}_{q_i}$, a similar complexity argument holds via tensor product. See Section 3.5 for details.

3.3 Truncation

Let $\mathcal{I}$ be a finite index set. Suppose we have an isomorphism $\eta_{\mathcal{I}} : R[x]/\langle \prod_{i \in \mathcal{I}} \boldsymbol{g}_i \rangle \to \prod_{i \in \mathcal{I}} R[x]/\langle \boldsymbol{g}_i \rangle$ for coprime ideals $(\boldsymbol{g}_i)_{i \in \mathcal{I}}$. Truncation [CF94, vdH04, Ber08] is a systematic approach for computing a product of size $n < \deg\left(\prod_{i \mathcal{I}} \boldsymbol{g}_i\right)$ while requiring the defining conditions of $\eta_{\mathcal{I}}$.

For an index set $\mathcal{J} \subset \mathcal{I}$ with $n + 1 = \deg\left(\prod_{j \in \mathcal{J}} \boldsymbol{g}_j\right)$, we have the isomorphism $\eta_{\mathcal{J}} : R[x]\Big/\left\langle \prod_{j \in \mathcal{J}} \boldsymbol{g}_j \right\rangle \to \prod_{j \in \mathcal{J}} R[x]/\langle \boldsymbol{g}_j \rangle$ for computing a size-n product. Obviously, $\eta_{\mathcal{J}}$ is defined whenever $\eta_{\mathcal{I}}$ is defined, and any algorithm implementing $\eta_{\mathcal{I}}$ can be converted into

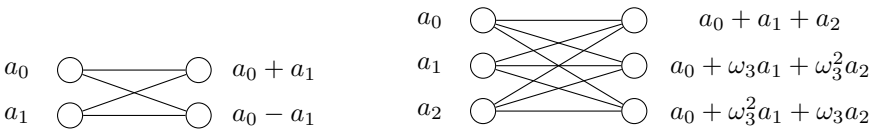
an algorithm implementing $\eta_{\mathcal{J}}$. In the simplest case $\prod_{i\in\mathcal{I}} \boldsymbol{g}_i = x^{2^k} - 1$, the computational complexity of $\eta_{\mathcal{J}}$ is roughly $\frac{n}{2^k}$ of $\eta_{\mathcal{I}}$.

3.4 Cooley–Tukey FFT

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

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

by choosing $\boldsymbol{g}_{i_0,\dots,i_{h-1}} = x - \zeta\omega_n^{\sum_l i_l \prod_{j<l} n_j}$ where ω_n is a principal n-th root of unity[2]. The Cooley–Tukey FFT is invertible if ζ and n are invertible in R. Since $\prod_{i_0,\dots,i_{h-1}} \boldsymbol{g}_{i_0,\dots,i_{h-1}} = x^n - \zeta^n$, we can multiply polynomials in $R[x]/\langle x^n - \zeta^n \rangle$ via $\prod_{i_0,\dots,i_{h-1}} R[x]\Big/\left\langle \boldsymbol{g}_{i_0,\dots,i_{h-1}} \right\rangle$. See Figure 1a for the radix-2 cyclic case $R[x]/\langle x^2 - 1 \rangle$ and Figure 1b for the radix-3 cyclic case $R[x]/\langle x^3 - 1 \rangle$.



(a) Radix-2 CT butterfly for $R[x]/\langle x^2 - 1 \rangle$. (b) Radix-3 CT butterfly for $R[x]/\langle x^3 - 1 \rangle$.

Figure 1: Radix-2 and Radix-3 Cooley–Tukey butterflies (CT butterflies).

3.5 Good–Thomas FFT

Let $n = \prod_j q_j$ for coprime integers $q_0, \dots, q_{d-1}$. There are two ways for stating Good–Thomas FFT [Goo58]: (i) as an isomorphism from a group algebra to a tensor product of associative algebras; and (ii) as a correspondence between one-dimensional FFT and multi-dimensional FFT. (ii) was stated in [Goo58], and (i) is a more general statement in the modern algebra language and is apparent from [Goo58].

Recall that we have a group isomorphism $\mathbb{Z}_n \cong \prod_j \mathbb{Z}_{q_j}$. This implies an isomorphism between the group algebras $R[\mathbb{Z}_n]$ and $\bigotimes_j R[\mathbb{Z}_{q_j}]$. Suppose n is invertible in R, and there is a principal n-th root of unity $\omega_n \in R$ realizing the isomorphism $\eta : R[x]/\langle x^n - 1 \rangle \cong \prod_i R[x]/\langle x - \omega_n^i \rangle$. By definition, we also have a principal n_j-th root of unity ω_{n_j} for each j. Let e_j be the unique tuple of orthogonal idempotents implementing $\forall a \in \mathbb{Z}_n, a \equiv \sum_j (a \bmod q_j) e_j \pmod{n}$. We choose $\omega_{n_j} \coloneqq \omega_n^{e_j}$ so $\prod_j \omega_{n_j} = \omega_n^{\sum_j e_j} = \omega_n$. We define η_j as the isomorphism $R[x_j]/\langle x_j^{n_j} - 1 \rangle \cong \prod_{i_j} R[x_j]\Big/\left\langle x_j - \omega_{n_j}^{ij} \right\rangle$ for each j, and relate the tensor product $\bigotimes_j \eta_j$ to η via the equivalence $x \sim \prod_j x_j$. Figure 2 is an illustration. This is how Good–Thomas FFT converts the computation η into $\bigotimes_j \eta_j$.

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

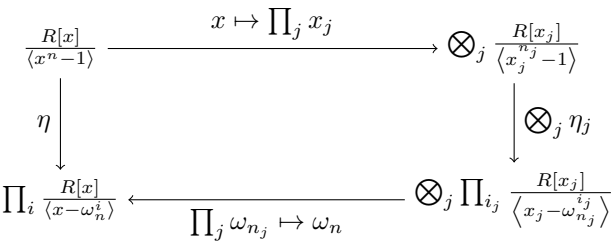


Figure 2: Commutative diagram of Good–Thomas FFT.

We give a small example for $n = 6$. By defining $\omega_3 \coloneqq \omega_6^4$ and $\omega_2 \coloneqq \omega_6^3$, we rewrite the transformation matrix implementing $R[x]/\langle x^6 - 1\rangle \cong \prod_i R[x]/\langle x - \omega_6^i\rangle$ as:

$$
\begin{aligned}
& P_{(14)} \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega_6 & \omega_6^2 & \omega_6^3 & \omega_6^4 & \omega_6^5 \\ 1 & \omega_6^2 & \omega_6^4 & 1 & \omega_6^2 & \omega_6^4 \\ 1 & \omega_6^3 & 1 & \omega_6^3 & 1 & \omega_6^3 \\ 1 & \omega_6^4 & \omega_6^2 & 1 & \omega_6^4 & \omega_6^2 \\ 1 & \omega_6^5 & \omega_6^4 & \omega_6^3 & \omega_6^2 & \omega_6 \end{pmatrix} P_{(14)} \\
= & \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega_6^4 & \omega_6^2 & 1 & \omega_6^4 & \omega_6^2 \\ 1 & \omega_6^2 & \omega_6^4 & 1 & \omega_6^2 & \omega_6^4 \\ 1 & 1 & 1 & \omega_6^3 & \omega_6^3 & \omega_6^3 \\ 1 & \omega_6^4 & \omega_6^2 & \omega_6^3 & \omega_6 & \omega_6^5 \\ 1 & \omega_6^2 & \omega_6^4 & \omega_6^3 & \omega_6^5 & \omega_6 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & \omega_2 \end{pmatrix} \otimes \begin{pmatrix} 1 & 1 & 1 \\ 1 & \omega_3 & \omega_3^2 \\ 1 & \omega_3^2 & \omega_3 \end{pmatrix}
\end{aligned}
$$

for $P_{(14)}$ the permutation matrix swapping the 1st and 4th elements.

3.6 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) $i \in \mathcal{I}^* \mapsto -\log_g i \in \mathbb{Z}_{\lambda(n)}$ and (ii) $j \in \mathcal{I}^* \mapsto \log_g j \in \mathbb{Z}_{\lambda(n)}$. For $j \in \mathcal{I}^*$, this gives us

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

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 $\left(\omega_p^{g^k}\right)_{k\in\mathbb{Z}_{\lambda(p)}}$.
We give an example for $p = 5$ and $g = 2$:

$$
\left(\hat{a}_{2^k} - a_0\right)_{k\in\mathbb{Z}_4} = \begin{pmatrix} a_1\omega_5^1 + a_2\omega_5^2 + a_3\omega_5^3 + a_4\omega_5^4 \\ a_1\omega_5^2 + a_2\omega_5^4 + a_3\omega_5^1 + a_4\omega_5^3 \\ a_1\omega_5^4 + a_2\omega_5^3 + a_3\omega_5^2 + a_4\omega_5^1 \\ a_1\omega_5^3 + a_2\omega_5^1 + a_3\omega_5^4 + a_4\omega_5^2 \end{pmatrix} = \begin{pmatrix} a_{2^0}\omega_5^{2^0} + a_{2^3}\omega_5^{2^3} + a_{2^2}\omega_5^{2^2} + a_{2^1}\omega_5^{2^1} \\ a_{2^0}\omega_5^{2^1} + a_{2^3}\omega_5^{2^0} + a_{2^2}\omega_5^{2^3} + a_{2^1}\omega_5^{2^2} \\ a_{2^0}\omega_5^{2^2} + a_{2^3}\omega_5^{2^1} + a_{2^2}\omega_5^{2^0} + a_{2^1}\omega_5^{2^3} \\ a_{2^0}\omega_5^{2^3} + a_{2^3}\omega_5^{2^2} + a_{2^2}\omega_5^{2^1} + a_{2^1}\omega_5^{2^0} \end{pmatrix}.
$$

Obviously, for the inversion, we apply Rader's FFT with inverted roots.

3.6.1 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 truncate Rader's FFT defined over $x^p - 1$ to $\Phi_p(x)$. We first find the following isomorphism

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

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

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

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 $\left(\omega_p^{g^k}\right)_{k \in \mathbb{Z}_{\lambda(p)}}$. Below is an illustration for $p = 5$ and $g = 2$:

$$\left(\omega_p^k \hat{a}_{2^k}\right)_{k \in \mathbb{Z}_4} = \begin{pmatrix} a_0\omega_5^1 + a_1\omega_5^2 + a_2\omega_5^3 + a_3\omega_5^4 \\ a_0\omega_5^2 + a_1\omega_5^4 + a_2\omega_5^1 + a_3\omega_5^3 \\ a_0\omega_5^4 + a_1\omega_5^3 + a_2\omega_5^2 + a_3\omega_5^1 \\ a_0\omega_5^3 + a_1\omega_5^1 + a_2\omega_5^4 + a_3\omega_5^2 \end{pmatrix} = \begin{pmatrix} a_0\omega_5^{2^0} + a_1\omega_5^{2^1} + a_2\omega_5^{2^3} + a_3\omega_5^{2^2} \\ a_0\omega_5^{2^1} + a_1\omega_5^{2^2} + a_2\omega_5^{2^0} + a_3\omega_5^{2^3} \\ a_0\omega_5^{2^2} + a_1\omega_5^{2^3} + a_2\omega_5^{2^1} + a_3\omega_5^{2^0} \\ a_0\omega_5^{2^3} + a_1\omega_5^{2^0} + a_2\omega_5^{2^2} + a_3\omega_5^{2^1} \end{pmatrix}$$

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

For the inverse η^{-1}, [Ber22, Section 4.8.2] showed how to implement it with a size-$\lambda(p)$ cyclic convolution. Since convoluting with $\left(\omega_p^{g^k}\right)_{k \in \mathbb{Z}_{\lambda(p)}}$ is exactly the same as multiplying by $\left(\omega_p^{g^k}\right)_{k \in \mathbb{Z}_{\lambda(p)}}$ in the group algebra $R[\mathbb{Z}_{\lambda(p)}]$ by definition, it suffices to identify the multiplicative inverse of $\left(\omega_p^{g^k}\right)_{k \in \mathbb{Z}_{\lambda(p)}}$ in $R[\mathbb{Z}_{\lambda(p)}]$. [Ber22] found $\left(\omega_p^{g^k}\right)_{k \in \mathbb{Z}_{\lambda(p)}}^{-1} = \frac{1}{p}\left(\omega_p^{-g^{-k}} - 1\right)_{k \in \mathbb{Z}_{\lambda(p)}}$. We illustrate below for $p = 5$ and $g = 2$:

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

$$= \begin{pmatrix} \omega_5^1 & \omega_5^3 & \omega_5^4 & \omega_5^2 \\ \omega_5^2 & \omega_5^1 & \omega_5^3 & \omega_5^4 \\ \omega_5^4 & \omega_5^2 & \omega_5^1 & \omega_5^3 \\ \omega_5^3 & \omega_5^4 & \omega_5^2 & \omega_5^1 \end{pmatrix} \begin{pmatrix} \omega_5^{-1} - 1 \\ \omega_5^{-3} - 1 \\ \omega_5^{-4} - 1 \\ \omega_5^{-2} - 1 \end{pmatrix} = \begin{pmatrix} 5 \\ 0 \\ 0 \\ 0 \end{pmatrix}.$$

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.

3.7 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 $\mathbb{Z}_q[x]/\langle x^{2^w}+1\rangle$ 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. The benefit is that the number of coefficients is the same after the transformation. Earlier work [BBCT22] with Nussbaumer for the same scenario resulted in $2\times$ many coefficients after transforming. 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 implicitly, and apply add-sub pairs [CCH+23]. See Figure 3 for illustration.

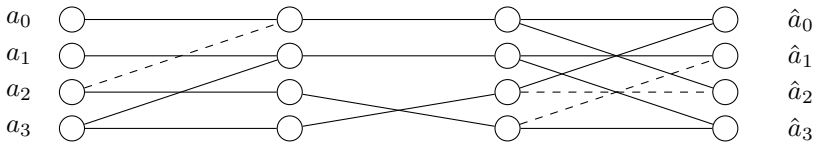


Figure 3: Bruun's butterfly computing $\hat{a}_0 + \hat{a}_1 x = \boldsymbol{a}(x) \bmod \left(x^2 + \sqrt{2}x + 1\right)$, $\hat{a}_2 + \hat{a}_3 x = \boldsymbol{a}(x) \bmod \left(x^2 - \sqrt{2}x + 1\right)$ for $\boldsymbol{a}(x) = \sum_{i=0,\ldots,3} a_i x^i$ [CCH+23].

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. The finite field case is closely related to [BGM93, Mey96, TW13, BMGVdO15, WYF18, WY21].

3.8 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. In the literature, twising is commonly specialized to $\xi = 1$. In this paper, we need the cases $\xi = \pm 1$.

3.9 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 $a_0 b_1 + a_1 b_0 = (a_0 + a_1)(b_0 + b_1) - a_0 b_0 - a_1 b_1$.

4 The Need of Vectorization

Let v be the number of elements contained in a vector register. In this section, we formalize vectorization and permutation–friendliness of an algebra monomorphism. Since algebra monomorphisms can be characterized as matrix multiplications, our formulation is based on manipulations of matrices with standard bases unless specified otherwise. Recent work SPIRAL [FLP+18] had attempted to formalize the vectorization of FFTs for code generation. However, it seems to fall short to cover the transformations considered in this paper. We believe this section will give more insights on extending SPIRAL.

4.1 Vectorization–Friendliness

We first identify a set SD of matrices that can be implemented efficiently with vector instructions. Although SD is definitely platform-dependent, we fix SD to be a union of certain matrices and explain why they are usually suitable for vectorization. We define SD as a set of all possible block diagonal matrices with each block a $v' \times v'$ matrix of the following form for a v-multiple v':

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 [CCH+23].
- 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 $S_{f_i} \in S$. We first observe that vector instruction sets usually provide instructions loading/storing consecutive v coefficients from/to memory. 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.

4.2 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 4 for Neon and Algorithm 5 for AVX2. Now we are ready to specify the set SI 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 SI consists of interleaving matrices of all possible steps and is closed under inversion.

We now define permutation–friendliness formally as follows. We call an algebra monomorphism g permutation–friendly if we can factor its matrix form M_g as

$$M_g = M'_{SI} M_f M_{SI}$$

for a vectorization–friendly M_f and $M_{SI}, M'_{SI} \in SI$ an interleaving matrix. Immediately, we know that g is permutation–friendly if and only if g^{-1} is permutation–friendly.

Algorithm 4 `trn{1, 2}` permuting double words in Armv8.0-A Neon registers.

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

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

Algorithm 5 `vperm2i128` permuting double words in AVX2 `%ymm` registers.

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

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

Generally, while computing with vector instructions, we choose algebra monomorphisms f and g such that f is vectorization–friendly and g is permutation–friendly. Their composition $g \circ f$ then admits a suitable mapping to our target vector instruction set. Concretely, we vectorize f, transpose the coefficients, and vectorize the vectorization–friendly part of g.

5 Implementation

This section goes through the implementation with truncated Rader's FFT, Good–Thomas FFT, and Bruun's FFT. For simplicity, we assume $R = \mathbb{F}_{4591}$ and multiply polynomials in

$$\frac{R[x]}{\langle \Phi_{17}(x^{96}) \rangle}.$$

5.1 Large-Dimensional Transformations

We first split $R[x]/\langle \Phi_{17}(x^{96}) \rangle$ into $\prod_{i=1,\ldots,16} R[x]/\langle x^{96} - \omega_{17}^i \rangle$ with truncated size-17 Rader's FFT. We then twist all $R[x]/\langle x^{96} - \omega_{17}^i \rangle$ into $R[x]/\langle x^{96} - 1 \rangle$ by observing $\omega_{17} = \omega_{17}^{1344} = \left(\omega_{17}^{14}\right)^{96}$. See Algorithms 6 and 7 for illustrations. In practice, twisting is merged with the scaling by ω_{17}^{-i} at the end of truncated Rader's FFT. For each $R[x]/\langle x^{96} - 1 \rangle$, we split it into $\prod_j R[x]\Big/\left\langle x^{16} - \omega_6^j \right\rangle$ with Good–Thomas FFT. This amounts to applying the tensor product of size-2 and size-3 cyclic FFT. Algorithm 8 is an illustration. Since everything so far is defined over chunks of 16-tuples, all of the above are vectorization-friendly.

Algorithm 6 Big picture of $R[x]/\langle\Phi_{17}(x^{96})\rangle \cong \left(R[x]/\langle x^{96}-1\rangle\right)^{16}$.

Input(s): $\texttt{poly[0-15][0-95]} = \boldsymbol{a}(x) = \sum_{i=0,\ldots,1535} a_i x^i$.

Output(s): $\left(\boldsymbol{a}\left(\omega_{17}^{14(i_0+1)}y\right) \bmod \left(y^{96}-1\right)\right)_{i_0=0,\ldots,15}$.

1: **for** $i_1 \in \{0,\ldots,95\}$ **do**
2: $\quad\texttt{poly\_NTT[0-15][}i_1\texttt{]} = \texttt{trunc\_Rader(poly[0-15][}i_1\texttt{])}$.
3: **end for**
4: $\quad\triangleright\ \forall i_0 \in \{0,\ldots,15\}, \texttt{poly\_NTT[}i_0\texttt{][0-95]} = \omega_{17}^{i_0+1}\boldsymbol{a}(x) \bmod \left(x^{96}-\omega_{17}^{i_0+1}\right)$.
5: **for** $i_0 \in \{0,\ldots,15\}$ **do**
6: $\quad\texttt{poly\_NTT[}i_0\texttt{][0-95]} = \texttt{twist96}\left(\texttt{poly\_NTT[}i_0\texttt{][0-95]}, \omega_{17}^{14(i_0+1)}\right)$.
7: **end for**
8: $\quad\triangleright\ \forall i_0 \in \{0,\ldots,15\}, \texttt{poly\_NTT[}i_0\texttt{][0-95]} = \boldsymbol{a}\left(\omega_{17}^{14(i_0+1)}y\right) \bmod \left(y^{96}-1\right)$.

Algorithm 7 Implementation of `trunc_Rader`.

Input(s): $\boldsymbol{a}(x) = \sum_{i=0,\ldots,15} a_i x^i$.

Output(s): $(c_i)_{i=0,\ldots,15} = \left(\omega_{17}^{i+1}\boldsymbol{a}(\omega_{17}^{i+1})\right)_{i=0,\ldots,15}$.

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

Algorithm 8 $\mathsf{NTT}_{R[x]/\langle x^6-1\rangle} = \pi \circ \left(\mathsf{NTT}_{R[y]/\langle y^2-1\rangle} \otimes \mathsf{NTT}_{R[z]/\langle z^3-1\rangle}\right) \circ \pi$ via Good–Thomas FFT where $\pi = P_{(14)}$ is the permutation matrix swapping the 1st and 4th element.

Input(s): $\boldsymbol{a}(x) = \sum_{i=0,\ldots,5} a_i x^i$.

Output(s): $(\mathsf{a0},\mathsf{a4},\mathsf{a2},\mathsf{a3},\mathsf{a1},\mathsf{a5}) = \left(\boldsymbol{a}(\omega_6^i)\right)_{i=0,\ldots,5}$.

1: $(\mathsf{a0},\ldots,\mathsf{a5}) = (a_0,a_4,a_2,a_3,a_1,a_5)$.
2: $(\mathsf{a0},\mathsf{a1},\mathsf{a2}) = \mathsf{NTT}_{R[z]/\langle z^3-1\rangle}(\mathsf{a0},\mathsf{a1},\mathsf{a2})$.
3: $(\mathsf{a3},\mathsf{a4},\mathsf{a5}) = \mathsf{NTT}_{R[z]/\langle z^3-1\rangle}(\mathsf{a3},\mathsf{a4},\mathsf{a5})$.
4: $(\mathsf{a0},\mathsf{a3}) = \mathsf{NTT}_{R[y]/\langle y^2-1\rangle}(\mathsf{a0},\mathsf{a3})$.
5: $(\mathsf{a1},\mathsf{a4}) = \mathsf{NTT}_{R[y]/\langle y^2-1\rangle}(\mathsf{a1},\mathsf{a4})$.
6: $(\mathsf{a2},\mathsf{a5}) = \mathsf{NTT}_{R[y]/\langle y^2-1\rangle}(\mathsf{a2},\mathsf{a5})$.
7: $\quad\triangleright\ (\mathsf{a0},\mathsf{a4},\mathsf{a2},\mathsf{a3},\mathsf{a1},\mathsf{a5}) = \left(\boldsymbol{a}(\omega_6^i)\right)_{i=0,\ldots,5}$ for $\omega_6 = \omega_2\omega_3$.

The remaining problems are multiplying in $R[x]\Big/\left\langle x^{16}-\omega_6^j\right\rangle$ – we have 16 instances for each of $R[x]\Big/\left\langle x^{16}-\omega_6^j\right\rangle$. Obviously, this implies permutation-friendliness – we simply partition $6\cdot 16$ size-16 polynomial multiplications into 6 partitions where each partition contains 16 instances, and apply transposition matrices to all the partitions. By definition, we have $\omega_6^j = \omega_2^{j \bmod {}^+2}\omega_3^{j \bmod {}^+3}$. Let's rewrite $(j \bmod {}^+2, j \bmod {}^+3)$ as (j_0,j_1) for all j.

After interleaving, we twist as follows:

$$\frac{R[x]}{\left\langle x^{16}-\omega_6^j\right\rangle}=\frac{R[x]}{\left\langle x^{16}-\omega_2^{j_0}\omega_3^{j_1}\right\rangle}=\frac{R[x]}{\left\langle x^{16}-\omega_2^{j_0}\left(\omega_3^{j_1}\right)^{16}\right\rangle}\cong\frac{R[x]}{\left\langle x^{16}-\omega_2^{j_0}\right\rangle}.$$

In practice, we merge the twisting $R[x]\Big/\left\langle x^{16}-\omega_6^j\right\rangle\cong R[x]\Big/\left\langle x^{16}-\omega_2^{j_0}\right\rangle$ and interleaving.

5.2 Small-Dimensional Polynomial Multiplications

The remaining problems are multiplying in $R[x]/\langle x^{16}\pm 1\rangle$.

$R[x]/\langle x^{16}-1\rangle$. For $R[x]/\langle x^{16}-1\rangle$, if one of multiplicands is known in prior (as in the case of truncated Rader's FFT), we split $R[x]/\langle x^{16}-1\rangle$ with Cooley–Tukey butterflies whenever the ring takes the form $R[x]\Big/\left\langle x^{2^k}-1\right\rangle$ for a $k>0$:

$$\frac{R[x]}{\langle x^{16}-1\rangle}\cong\frac{R[x]}{\langle x-1\rangle}\times\frac{R[x]}{\langle x+1\rangle}\times\frac{R[x]}{\langle x^2+1\rangle}\times\frac{R[x]}{\langle x^4+1\rangle}\times\frac{R[x]}{\langle x^8+1\rangle}.$$

For a size-16 polynomial $\boldsymbol{c}(x)$ with coefficients known in prior, we precompute its image with proper scaling. In other words, we compute $8\boldsymbol{c}(x=1), 8\boldsymbol{c}(x=-1), 4\boldsymbol{c}(x^2=-1), 2\boldsymbol{c}(x^4=-1), \boldsymbol{c}(x^8=-1)$ where $\boldsymbol{c}(x=1)$ is obtained by replacing x with 1 and similarly for $\boldsymbol{c}(x=-1), \boldsymbol{c}(x^2=-1), \boldsymbol{c}(x^4=-1), \boldsymbol{c}(x^8=-1)$. The scaling of the coefficients is then the same while inverting Cooley–Tukey butterflies. On the other hand, if both of the multiplicands are unknown, we split $R[x]/\langle x^{16}-1\rangle$ into

$$\frac{R[x]}{\langle x^{16}-1\rangle}\cong\frac{R[x]}{\langle x^4-1\rangle}\times\frac{R[x]}{\langle x^4+1\rangle}\times\frac{R[x]}{\langle x^8-1\rangle}$$

and perform additional additions doubling half of the coefficients in order to have consistent scaling for all the coefficients.

$R[x]/\langle x^{16}+1\rangle$. For $R[x]/\langle x^{16}+1\rangle$, we split with Bruun's butterfly as follows

$$\frac{R[x]}{\langle x^{16}+1\rangle}\cong\frac{R[x]}{\left\langle x^8+\sqrt{2}x^4+1\right\rangle}\times\frac{R[x]}{\left\langle x^8-\sqrt{2}x^4+1\right\rangle}.$$

Finally, we perform Karatsuba for size-8 polynomial multiplications and schoolbook otherwise. The criteria for choosing between Cooley–Tukey butterflies, Bruun's butterflies, Karatsuba, and schoolbook is based on our experiments.

5.3 Comparisons to Prior Vectorized Implementation

We briefly compare our vectorized implementation to prior AVX2 work [BBCT22] and Neon work [CCH+23].

5.3.1 Comparison to [BBCT22]

[BBCT22] reduced the computing task to

$$\frac{R[x]}{\langle(x^{1024}+1)(x^{512}-1)\rangle}.$$

They first introduced $x^{32}-y$ and rewrite $R[x]/\langle(x^{1024}+1)(x^{512}-1)\rangle$ as

$$\frac{\frac{R[x]}{\langle x^{32}-y\rangle}[y]}{\langle(y^{32}+1)(y^{16}-1)\rangle}.$$

They then applied Schönhage [Sch77] by replacing $x^{32}-y$ with $x^{64}+1$[3]. Since $x^{64}\sim -1$ in $R[x]/\langle x^{64}+1\rangle$, x is a principal 128-th root of unity for radix-2 Cooley–Tukey FFT in y. We argue that Schönhage is vectorzation-friendly: the replacement of $x^{32}-y$ by $x^{64}+1$ can be described as $\begin{pmatrix}1\\0\end{pmatrix}\otimes I_{32}$ and multiplication by powers of x over $R[x]/\langle x^{64}+1\rangle$ are negacyclic shifts. Therefore, the overall transformation can be written as:

$$\left(\prod(\texttt{AddSub}_i\otimes I_{16})\texttt{BlockNegShift}_i\right)\left(I_{48}\otimes\begin{pmatrix}1\\0\end{pmatrix}\otimes I_{32}\right)$$

for $\texttt{BlockNegShift}_i\in SD$ and $\texttt{AddSub}_i$'s implementing add-sub pairs of various sizes.

For $R[x]/\langle x^{64}+1\rangle$, they applied Nussbaumer [Nus80] as follows:

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

Again, z is a principal 16-th root of unity for radix-2 Cooley–Tukey FFT in x. We claim that Nussbaumer is permutation friendly with an informal justification: Observe that Nussbaumer can be seen as a composition of Schönhage and interleaving, it remains to specify clearly the interleaving step. We leave the formal justification to readers.

The remaining problem is to compute $48\cdot 16=768$ polynomial multiplications of the form $R[z]/\langle z^8+1\rangle$. [BBCT22] applied recursive Karatsuba to $R[z]/\langle z^8+1\rangle$.

On the other hand, after the FFT computation, our transformation results in only 48 size-4 cyclic convolution, 48 size-4 negacyclic convolution, 48 size-8 negacyclic convolution, 48 polynomial multiplications in $R[x]/\langle x^8+\sqrt{2}x^4+1\rangle$, and 48 polynomial multiplications in $R[x]/\langle x^8-\sqrt{2}x^4+1\rangle$. A rough measure is to count the number of coefficients involved. $768\cdot 8=6144$ coefficients are involved in [BBCT22]'s implementation while only $48\cdot 4+48\cdot 4+48\cdot 8+48\cdot 8+48\cdot 8=1536$, *a quater of* 6144, are involved in our implementation. This explains the significant overal performance improvement.

5.3.2 Comparison to [CCH+23]

In this section, we compare our transformation to the `Good-Rader-Bruun` approach used in [CCH+23]. We will explain the benefit of truncated Rader's FFT over Rader's FFT in terms of permutation-friendliness.

Although we rely on several techniques used in [CCH+23], our implementation is actually quite different from them in terms of transformation choices. They computed in $R[x]/\langle x^{1632}-1\rangle$. [CCH+23] first permuted with the equivalences $x^{16}\sim uvw, u^{17}\sim v^3\sim w^2\sim 1$. This gives them $\bar{R}[u]/\langle u^{17}-1\rangle\otimes\bar{R}[v]/\langle v^3-1\rangle\otimes\bar{R}[w]/\langle w^2-1\rangle$ where $\bar{R}=R[x]/\langle x^{16}-uvw\rangle$. They then applied size-17 Rader's FFT without truncation to $\bar{R}[u]/\langle u^{17}-1\rangle$, size-3 FFT to $\bar{R}[v]/\langle v^3-1\rangle$, and size-2 FFT to $\bar{R}[w]/\langle w^2-1\rangle$. Obviously, since all transformations operate over $\bar{R}$, they are vectorization-friendly.

Substituting uvw with various ω_{102}^i, [CCH+23] splitted $R[x]/\langle x^{16}-\omega_{102}^i\rangle$ with Cooley–Tukey and Bruun's FFTs without twisting. This step is not permutation-friendly: Observe that permutation-friendliness is defined only for monomorphisms with rank over R a

[3]This map is a monomorphism in the sense that products in $R[x]/\langle x^{32}-y\rangle$ can be uniquely identified while computing in $R[x]/\langle x^{64}+1\rangle$

multiple of $v^2 = 16^2 = 256$, we can't find a suitable interleaving matrix for permutation-friendliness. They applied Cooley–Tukey to 48 instances of $R[x]/\langle x^{16} - \omega_{102}^i \rangle$, $i \equiv 0 \pmod 2$ and Bruun to 48 instances of $R[x]/\langle x^{16} - \omega_{102}^i \rangle$, $i \equiv 1 \pmod 2$. For the remaining 6 instances, they interleaved with don't-cares and applied Karatsuba.

In our implementation, we compute in $R[x]/\langle \Phi_{17}(x^{96}) \rangle$ by first applying truncated size-17 Rader's FFT which results in $\frac{16}{17}$ number of subproblems of size-17 Rader's FFT. Since 16|96, the transformation can be written as a tensor product of a matrix and I_{16}. After twisting and Good–Thomas FFT, we have 48 instances of $R[x]/\langle x^{16} - 1 \rangle$ and 48 instances of $R[x]/\langle x^{16} + 1 \rangle$ with no leftovers – hence permutation-friendly.

For the size-16 polynomial multiplications of the form $R[x]/\langle x^{16} - \zeta \rangle$, [CCH+23] applied the Cooley–Tukey and Bruun's FFTs defined on $R[x]/\langle x^{16} - \zeta \rangle$. We apply twisting instead as implemented in common AVX2-optimized implementations. The benefit of twisting is twofold: (i) twisting effectively reduces the range of the coefficients (this is widely known, but somehow [CCH+23] didn't implement it), and (ii) Bruun's FFT is more favorable when the modulus takes the form $x^8 \pm \sqrt{2}x^4 + 1$ in terms of arithmetic and register scheduling since $\sqrt{2}^2 - 1 = 1$ and the dependency chain is shorter than $x^8 \pm \alpha x^4 + \beta$ in general.

6 Results

6.1 Benchmarking Environment

We benchmark 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.

6.2 Performance of Polynomial Multiplication

We provide the performance cycles of functions `mulcore` and `polymul` in Table 1. `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. Compared to [BBCT22], our `mulcore` is 1.90× and 2.05× faster on Haswell and Skylake. For `polymul`, our implementation is 1.99× and 2.16× faster on Haswell and Skylake. In Appendix A, we provide the detailed performance numbers. Additionally, Table 2 summarizes the Haswell and Skylake cycles of existing x86 implementations.

Table 1: Performance cycles of big-by-big polynomial multiplications for `ntrulprs761/sntrup761` on Haswell and Skylake with AVX2.

	[BBCT22]*	This work
Haswell		
`mulcore` ($\mathbb{Z}_{4591}[x]$)	23 460	**12 336**
`polymul` $\left(\frac{\mathbb{Z}_{4591}[x]}{\langle x^{761}-x-1\rangle}\right)$	25 356	**12 760**
Skylake		
`mulcore` ($\mathbb{Z}_{4591}[x]$)	20 070	**9 778**
`polymul` $\left(\frac{\mathbb{Z}_{4591}[x]}{\langle x^{761}-x-1\rangle}\right)$	21 364	**9 876**

* Our own benchmarks.

Table 2: Performance cycles of existing polynomial multiplications for `ntrulpr761/sntrup761`.

	Haswell cycles	Skylake cycles	Applicability
`round1`***	29 013	26 401	Big-by-small
`round2`***	17 546	14 448	Big-by-small
`avx800`***	16 514	13 222	Big-by-small
[BBCT22]**	25 356	21 364	Big-by-big
This work	**12 760**	**9 876**	Big-by-big

** Our own benchmarks.
*** Implementations in SUPERCOP (version `supercop-20230530`). See https://bench.cr.yp.to/impl-core/multsntrup761.html for numbers on other processors.

6.3 Performance of Scheme

Finally, we compare the overall performance of `sntrup761` on Haswell and Skylake. For the batch key generation with batch size 32, we reduce the amortized cost by 12% on Haswell and 8% on Skylake. For encapsulation, we reduce the cost by 7% on Haswell and 10% on Skylake. For decapsulation, we reduce the cost by 10% on Haswell and 13% on Skylake. We summarize the performance in Table 3.

Table 3: Performance cycles of `sntrup761` with batch key generation using Montgomery's trick. 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.

7 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 briefly draft below for the parameter sets `ntrulpr857/sntrup857`, `ntrulpr1013/sntrup1013`, and `ntrulpr1277/sntrup1277`.

In `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:

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

The remaining problems are 54 polynomial multiplications in each of $\mathbb{Z}_{5167}[x]/\langle x^{16}-1\rangle$ and $\mathbb{Z}_{5167}[x]/\langle x^{16}+1\rangle$. 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 5.2. 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. As a side note, if the number of elements is smaller than 16, one can possibly choose other approaches. Let's take Neon as an example. Since each vector register in Neon holds 128-bit of data, or equivalently, 8 elements in $\mathbb{Z}_{5167}$, we can instead interleave 6 polynomials in $\mathbb{Z}_{5167}[x]/\langle x^{16}-1\rangle$ with don't-cares and apply Cooley–Tukey FFT. Similar arguement applies to $\mathbb{Z}_{5167}[x]/\langle x^{16}+1\rangle$ with Bruun's FFT.

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 Table 5, 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 $5112 \cdot \frac{1}{256 \cdot 8} \cdot \frac{1728}{6} \cdot 4 = 2875.5$ cycles. For the performance estimation of size-16 cyclic/negacyclic polynomial multiplications, 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 $3168 \cdot \frac{48}{128} + 4720 \cdot \frac{48}{128} + 5588 \cdot \frac{16}{128} = 3656.5$ cycles. Finally, the remaining estimation is about interleaving polynomials. 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 $2875.5 \cdot 3 + 672 \cdot 3 + 3656.5 = 14299$ cycles.

We briefly draft below some promising approaches for larger parameter sets. For `ntrulpr1013/sntrup1013`, we reduce polynomial multiplication in $\mathbb{Z}_{7177}[x]/\langle x^{1013} - x - 1 \rangle$ to $\mathbb{Z}_{7177}[x]\Big/\left\langle \frac{x^{2496}-1}{x^{312}+1} \right\rangle$ with Rader's, truncated Good–Thomas [HVDH22, Section 3.5], and Cooley–Tukey FFTs. For `ntrulpr1277/sntrup1277`, we reduce the computing task in $\mathbb{Z}_{7879}[x]/\langle x^{1277} - x - 1 \rangle$ to $\mathbb{Z}_{7879}[x]/\langle (x^{2496} - 1)(x^{64} + 1) \rangle$ with Rader's, Good–Thomas, and Cooley–Tukey FFTs.

A Profiling of Polynomial Multiplication

Table 4: 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

Table 5: 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-$(3, 2)$		
Radix-$(3, 2)$ with pre-twist (256×8)	5 112	3 794
Radix-$(3, 2)$ with post-twist (256×8)	4 904	3 550
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

Table 6: 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's	4 720	3 616
$R[x]/\langle x^{16} - \zeta \rangle$ (128×)	Karatsuba	5 588	4 428

References

[AB74] Ramesh C. Agarwal and Charles S. Burrus. Fast convolution using Fermat number transforms with applications to digital filtering. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 22(2):87–97, 1974. 2

[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. 2

[Bar86] Paul Barrett. Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor. In *CRYPTO 1986*, LNCS, pages 311–323. SV, 1986. 3, 4

[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/. 1

[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. 1, 2, 3, 10, 11, 14, 15, 16, 17

[Ber08] Daniel J. Bernstein. Fast multiplication and its applications. *Algorithmic number theory*, 44:325–384, 2008. 6

[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. 9

[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. 10

[BMGVdO15] F.E. Brochero Martínez, C. R. Giraldo Vergaraand, and L. Batista de Oliveira. Explicit factorization of $x^n - 1 \in \mathbb{F}_q[x]$. *Designs, Codes and Cryptography*, 77:277–286, 2015. https://link.springer.com/article/10.1007/s10623-014-0005-y. 10

[Bou89] Nicolas Bourbaki. *Algebra I*. Springer, 1989. 6

[Bru78] Georg Bruun. z-transform DFT Filters and FFT's. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 26(1):56–63, 1978. 10

[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. 1, 3, 18

[CCH+23] Han-Ting Chen, Yi-Hua Chung, Vincent Hwang, Chi-Ting Liu, and Bo-Yin Yang. Algorithmic Views of Vectorized Polynomial Multipliers for NTRU and NTRU Prime (Long Paper). Cryptology ePrint Archive, Paper 2023/541, 2023. https://eprint.iacr.org/2023/541. 2, 3, 10, 11, 14, 15, 16

[CF94] Richard Crandall and Barry Fagin. Discrete Weighted Transforms and Large-integer Arithmetic. *Mathematics of computation*, 62(205):305–324, 1994. 6

[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. 7

[DV78] Eric Dubois and Anastasios N. Venetsanopoulos. The discrete Fourier transform over finite rings with application to fast convolution. *IEEE Computer Architecture Letters*, 27(07):586–593, 1978. 2

[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. 10

[Für09] Martin Fürer. Faster Integer Multiplication. *SIAM Journal on Computing*, 39(3):979–1005, 2009. https://doi.org/10.1137/070711761. 2

[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. 7

[HVDH22] David Harvey and Joris Van Der Hoeven. Polynomial Multiplication over Finite Fields in time O (n log n). *Journal of the ACM*, 69(2):1–40, 2022. https://dl.acm.org/doi/full/10.1145/3505584. 19

[Jac12] Nathan Jacobson. *Basic Algebra II*. Courier Corporation, 2012. 6

[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. 10

[Mey96] Helmut Meyn. Factorization of the Cyclotomic Polynomial $x^{2^n}+1$ over Finite Fields. *Finite Fields and Their Applications*, 2(4):439–442, 1996. 10

[Mon85] Peter L. Montgomery. Modular Multiplication Without Trial Division. *Mathematics of computation*, 44(170):519–521, 1985. 3, 4

[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. 10

[NIS] NIST, the US National Institute of Standards and Technology. Post-quantum cryptography standardization project. https://csrc.nist.gov/Projects/post-quantum-cryptography. 20

[Nus80] Henri Nussbaumer. Fast Polynomial Transform Algorithms for Digital Convolution. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 28(2):205–215, 1980. 15

[Pol71] John M. Pollard. The Fast Fourier Transform in a Finite Field. *Mathematics of computation*, 25(114):365–374, 1971. 2

[Pol76] John M Pollard. Implementation of Number-Theoretic Transforms. *Electronics Letters*, 15(12):378–379, 1976. 2

[Rad68] Charles M. Rader. Discrete fourier transforms when the number of data samples is prime. *Proceedings of the IEEE*, 56(6):1107–1108, 1968. 8

[Sch77] Arnold Schönhage. Schnelle multiplikation von polynomen über körpern der charakteristik 2. *Acta Informatica*, 7(4):395–398, 1977. 15

[Sei18] Gregor Seiler. Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography. 2018. https://eprint.iacr.org/2018/039. 3, 4

[TW13] Aleksandr Tuxanidy and Qiang Wang. Composed products and factors of cyclotomic polynomials over finite fields. *Designs, codes and cryptography*, 69(2):203–231, 2013. 10

[vdH04] Joris van der Hoeven. The truncated Fourier transform and applications. In *Proceedings of the 2004 international symposium on Symbolic and algebraic computation*, pages 290–296, 2004. 6

[Win78] Shmuel Winograd. On Computing the Discrete Fourier Transform. *Mathematics of computation*, 32(141):175–199, 1978. 8

[WY21] Yansheng Wu and Qin Yue. Further factorization of $x^n - 1$ over a finite field (II). *Discrete Mathematics, Algorithms and Applications*, 13(06):2150070, 2021. 10

[WYF18] Yansheng Wu, Qin Yue, and Shuqin Fan. Further factorization of $x^n - 1$ over a finite field. *Finite Fields and Their Applications*, 54:197–215, 2018. 10