Metadata of the chapter that will be visualized in SpringerLink

Book Title	Information Security and Privacy	
Series Title		
Chapter Title	Jumping for Bernstein-Yang Inversion	
Copyright Year	2024	
Copyright HolderName	The Author(s), under exclusive license to Springer Nature Singapore Pte Ltd.	
Author	Family Name	**Jian**
	Particle	
	Given Name	**Li-Jie**
	Prefix	
	Suffix	
	Role	
	Division	
	Organization	Academia Sinica
	Address	Taipei, Taiwan
	Email	jcuyo613@gmail.com
Author	Family Name	**Wang**
	Particle	
	Given Name	**Ting-Yuan**
	Prefix	
	Suffix	
	Role	
	Division	
	Organization	Academia Sinica
	Address	Taipei, Taiwan
	Email	deanwang88528@gmail.com
Corresponding Author	Family Name	**Yang**
	Particle	
	Given Name	**Bo-Yin**
	Prefix	
	Suffix	
	Role	
	Division	
	Organization	Academia Sinica
	Address	Taipei, Taiwan
	Email	by@crypto.tw
Author	Family Name	**Chen**
	Particle	
	Given Name	**Ming-Shing**
	Prefix	
	Suffix	

Role	
Division	
Organization	Academia Sinica
Address	Taipei, Taiwan
Email	mschen@crypto.tw

Abstract	This paper achieves fast polynomial inverse operations specifically tailored for the NTRU Prime KEM on ARMv8 NEON instruction set benchmarking on four processor architectures: Cortex-A53, Cortex-A72, Cortex-A76 and Apple M1. We utilize the *jumping* divison steps of the constant-time GCD algorithm from Bernstein and Yang (TCHES'19) and optimize underlying polynomial multiplication of various lengths to improve the efficiency for computing polynomial inverse operations in NTRU Prime.

Keywords (separated by '-')	NTRU Prime - Multiplication - Extended GCD - Inversion

Jumping for Bernstein-Yang Inversion

Li-Jie Jian, Ting-Yuan Wang, Bo-Yin Yang(✉), and Ming-Shing Chen

Academia Sinica, Taipei, Taiwan
{by,mschen}@crypto.tw

AQ1

Abstract. This paper achieves fast polynomial inverse operations specifically tailored for the NTRU Prime KEM on ARMv8 NEON instruction set benchmarking on four processor architectures: Cortex-A53, Cortex-A72, Cortex-A76 and Apple M1. We utilize the *jumping* divison steps of the constant-time GCD algorithm from Bernstein and Yang (TCHES'19) and optimize underlying polynomial multiplication of various lengths to improve the efficiency for computing polynomial inverse operations in NTRU Prime.

Keywords: NTRU Prime · Multiplication · Extended GCD · Inversion

1 Introduction

Bernstein and Yang [9] proposed a fast constant-time GCD algorithms for preventing leakage of timing information in cryptographic applications. Since then, many have utilized the algorithm on various cryptographic applications, e.g., NTRU [12], NTRU Prime [25], BIKE [27], and even ElGamal cryptosystem [15].

NTRU Prime [5] is one of third-round candidates of key-encapsulation mechanism (KEM) in NIST's Post-Quantum Cryptography standardization process. It has been integrated into OpenSSL [6] and OpenSSH [24], the latter as its default key-exchange method. In the current implementation of NTRU Prime [5], computing polynomial inversion takes almost all its key generation time. Since the standard TLS protocol uses ephemeral key for the forward secrecy property, it performs key generation for every TLS connection. Accelerating the performance of key generation becomes a severe issue.

In this paper, we focus on development for ARMv8. ARM has demonstrated the popularity of its computing platforms, ranging from the tiniest sensors to smartphones and data centers. It is clear that ensuring secure communication among these devices in the age of quantum computers becomes increasingly critical as time goes on. In order to demonstrate substantial advancement, we conduct a comparison of the performance of our NTRU Prime key generation utilizing jumpdivstep with previous approaches employing divstep across various platforms including Cortex-A53, Cortex-A72, Cortex-A76 and Apple M1.

The Problem. In Bernstein–Yang's GCD algorithm [9], they decompose a GCD procedure into numerous constant-time "division steps" (divstep). Depending on the degree of input polynomials, the algorithm iterates a constant number

Y. Li and T. Zhu (Eds.): ACISP 2024, LNCS 14896, pp. 1–21, 2024.
https://doi.org/10.1007/978-981-97-5028-3_6

of divstep. [9] also proposed "jumping division steps" (jumpdivstep) which split a number of divstepx into the combination of several smaller batches of divstepx and "jump" through these smaller steps via matrix multiplication where elements of the matrices are polynomials. They showed jumpdivstep performs the same procedures with a better asymptotic complexity than divstep. However, to the best of our knowledge, there is no efficient implementation of jumpdivstep showing its computational supremacy in the literature. We have identified several primary challenges in realizing jumpdivstep:

– Polynomial multiplication poses challenges in optimization, particularly due to the need to explore various techniques tailored to specific length requirements.
– The implementation of jumpdivstep requires managing extra objects during execution. It is therefore unclear how long the polynomials need to be for jumpdivstep to showcase its complexity advantage.
– Previous research primarily concentrates on optimizing longer polynomials, as the advantages of Number Theoretic Transforms (NTTs) are more evident in such cases compared to shorter ones.
– More recursive layers in jumpdivstep theoretically reduce complexity but also require shorter polynomial multiplications. However, shorter polynomial multiplications often prefer algorithms with higher asymptotic complexity. Thus, determining the optimal number of layers for jumpdivstep is nontrivial.

Our Contributions

1. Low complexity algorithm does not always lead to a faster implementation. Here, we are the first to achieve faster polynomial inversion with jumpdivstep instead of divstep in a practical application (NTRU Prime).
2. We showcase fast vectorized polynomial matrix multiplications across a range of polynomial lengths.
3. Across platforms including Cortex-A53, Cortex-A72, Cortex-A76 and Apple M1, we illustrate that for polynomials of degree as low as 653, jumpdivstep beats divstepx.
4. We noticeably accelerate **sntrup761** keygen (the most common instance and the default in OpenSSH), saving up to 64% time compared to the fastest divstep version so far [20].

The code of this work is publicly available at https://github.com/Jumpdivstepsx/Jumping4Bernstein-YangInversion.git.

Related Work. Most prior art in NTRU Prime [1,20,27,29,30] focused on optimizing the encap and decap (i.e., the polynomial multiplication). Bernstein-Brumley-Chen-Tuveri [6] had applied Montgemery's trick to speed up keygen on average when generating a batch of keys simultaneously in a server. No other research shows significant speedup in polynomial inversion through the jumping strategy of [9], which we do even for the smallest NTRU Prime instance.

Table 1. Parameter sets of sntrup

Scheme	**sntrup653**	**sntrup761**	**sntrup857**	**sntrup953**	**sntrup1013**	**sntrup1277**
p	653	761	857	953	1013	1277
q	4621	4591	5167	6346	7177	7879

2 Preliminaries

2.1 Streamlined NTRU Prime

Streamlined NTRU Prime (sntrup) [7] is a KEM using the polynomial rings $\mathbb{F}_q[x]/(x^p - x - 1)$ and $\mathbb{F}_3[x]/(x^p - x - 1)$, where $\mathbb{F}_q$ is a finite field constructed using a prime number q. The other parameter p is also a prime and $x^p - x - 1$ is irreducible in $\mathbb{F}_q[x]$. The parameter sets of sntrup are listed in Table 1. During key generation, sntrup computes two inversions: **invsntrup** in $\mathbb{F}_q[x]/(x^p - x - 1)$ and **inv3ntrup** in $\mathbb{F}_3[x]/(x^p - x - 1)$. The two inversions consume almost all execution time of the key generation.

2.2 Fast Constant-Time GCD

The Bernstein-Yang GCD algorithm [9] decomposed the GCD procedure into a constant number of divstep. In contrast to the traditional GCD that eliminates the head coefficients at any degree, divstep always eliminates the head coefficients at the degree-0 position. This leads to extra coefficient reversal processes for reversing input polynomials such that the degree-0 coefficient in the reversed polynomial represents the head coefficient in the original polynomial. The algorithm describes the two input polynomials (f, g) as a column vector $\begin{bmatrix} f & g \end{bmatrix}^{\mathbf{T}}$. One divstep first determines a transition matrix $\mathcal{T}$ from the degree-0 coefficients of inputs $(f(0), g(0))$ and the their degree difference δ as

$$\mathcal{T}(\delta, f, g) = \begin{cases} \begin{bmatrix} 0 & 1 \\ \frac{g(0)}{x} & \frac{-f(0)}{x} \end{bmatrix} & \text{if } \delta > 0 \text{ and } g(0) \neq 0, \\ \\ \begin{bmatrix} 1 & 0 \\ \frac{-g(0)}{x} & \frac{f(0)}{x} \end{bmatrix} & \text{otherwise,} \end{cases}$$

and then performs a matrix-vector multiplication $\begin{bmatrix} f_{\text{output}} \\ g_{\text{output}} \end{bmatrix} = \mathcal{T} \cdot \begin{bmatrix} f \\ g \end{bmatrix}$ to eliminate the degree-0 term of the polynomial of higher degree.

For performing a number of consecutive divstep, Algorithm 1 shows divstepx which iterates n divstep in one function. The two input polynomials f and g are in reverse order. It outputs the degree difference δ, two modified polynomials f and g, and the transition matrix transforming the input polynomials to outputs. In Algorithm 1, f_0 and g_0 are shorthand of the constant terms $f(0)$ and $g(0)$. The loop from line 2 to 11 effectively performs n divstep. Line 8 and 9 contain the most heavy computations. They require multiplication of 6 polynomials (f, g, u, v, q, r) by constants f_0 and g_0. Since the loop has n iterations, the two steps resemble the Schoolbook multiplication for $O(n^2)$ operations.

In contrast to the iterative divstepx, jumpdivstep in Algorithm 2 applies a recursive divide-and-conquer approach to achieve the same functionality as divstepx. It partitions the n steps into two $n/2$ steps and calls two jumpdivstep for the two smaller computations. The whole recursive process can be described as splitting a tree into balanced subtrees as Fig. 5. In each jumpdivstep, it requires one matrix-matrix multiplication for the output transition matrix at line 10 and two matrix-vector multiplication to update polynomials (f, g) at line 7 and 9. It thus needs a multiplication library to compute polynomial multiplication of different lengths in each recursive layer. On the other hand, jumpdivstep can utilize optimized algorithms, e.g. NTT-based multiplications, to update transition matrices and polynomials. This way its complexity is reduced from $O(n^2)$ operations to $O(n \log n)$ operations. We present more details of these optimization in Sect. 4.

Algorithm 1. divstepx $(n,\ \delta,\ f,\ g)$

Input: $n \geq 0, \delta \in \mathbb{Z}$
Output: $\delta, f, g, M \in \mathbf{R}_q[x]^{2\times 2}$
1: $\begin{bmatrix} u & v \\ q & r \end{bmatrix} \in \mathbf{R}_q[x]^{2\times 2} \leftarrow \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$
2: **for** $i \leftarrow 1$ to n **do**
3:     **if** $\delta > 0$ and $g_0 \neq 0$ **then** ▷ swap
4:         $\delta \leftarrow -\delta$
5:         $f, g, u, v, q, r \leftarrow g, f, q, r, u, v$
6:     **end if**
7:     $\delta \leftarrow \delta + 1$
8:     $g \leftarrow (g \cdot f_0 - f \cdot g_0)/x$
9:     $q, r \leftarrow (q \cdot f_0 - u \cdot g_0), (r \cdot f_0 - v \cdot g_0)$
10:     $u, v \leftarrow u \cdot x, v \cdot x$ ▷ Raise degree
11: **end for**
12: **return** $\delta, f, g, \begin{bmatrix} u & v \\ q & r \end{bmatrix}$

Algorithm 2. jumpdivstep $(n,\ \delta,\ f,\ g)$

Input: $n \geq 0, \delta \in \mathbb{Z}$
Output: $\delta, f, g, M \in \mathbf{R}_q[x]^{2\times 2}$
1: **if** $n < n_{threshold}$ **then**
2:     **return** divstepx(n, δ, f, g)
3: **end if**
4: $j \leftarrow \lfloor n/2 \rfloor$
5: $k \leftarrow n - j$
6: $\delta, f', g', M_1 \leftarrow$ jumpdivstep(j, δ, f, g)
7: $\begin{bmatrix} f \\ g \end{bmatrix} \leftarrow x^{-j} \cdot M_1 \cdot \begin{bmatrix} f \\ g \end{bmatrix} + \begin{bmatrix} f' \\ g' \end{bmatrix}$
8: $\delta, f', g', M_2 \leftarrow$ jumpdivstep(k, δ, f, g)
9: $\begin{bmatrix} f \\ g \end{bmatrix} \leftarrow x^{-k} \cdot M_2 \cdot \begin{bmatrix} f \\ g \end{bmatrix} + \begin{bmatrix} f' \\ g' \end{bmatrix}$
10: $M \leftarrow M_2 \cdot M_1$
11: **return** δ, f, g, M

Computing Polynomial Inversion. We compute reciprocal of polynomial g in $\mathbb{F}_q[x]/(x^p - x - 1)$ by performing GCD on $(x^p - x - 1,\ g)$ as multiplying a transition matrix by the input vector

$$\begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} u & v \\ q & r \end{bmatrix} \cdot \begin{bmatrix} x^p - x - 1 \\ g \end{bmatrix} .$$

Here the GCD polynomial becomes 1 because $x^p - x - 1$ is irreducible. Since the GCD can be written as

$$1 = u \cdot (x^p - x - 1) + v \cdot g \implies 1 \equiv v \cdot g \mod (x^p - x - 1) , \tag{1}$$

we get $g^{-1} = v$ in $\mathbb{F}_q[x]/(x^p - x - 1)$.

In divstepx, g and $x^p - x - 1$ together comprise a total of $2p + 1$ coefficients and a degree difference $\delta = 1$. After performing $2p - 1$ steps of divstepx, both g and $x^p - x - 1$ are eliminated to only one coefficient.

2.3 Chinese Reminder Theorem

In a polynomial ring, the Chinese Remainder Theorem (CRT) presents that

$$\frac{\mathbf{R}_q[x]}{\langle \prod_{i=0}^{n} g_i(x) \rangle} \cong \frac{\mathbf{R}_q[x]}{\langle g_0(x) \rangle} \times \frac{\mathbf{R}_q[x]}{\langle g_1(x) \rangle} \times \cdots \times \frac{\mathbf{R}_q[x]}{\langle g_n(x) \rangle} \cong \prod_{i=0}^{n} \frac{\mathbf{R}_q[x]}{\langle g_i(x) \rangle} \quad ,$$

where $\mathbf{R}_q$ is a polynomial ring and $g_i(x)$ are coprime polynomials. This implies that a significant improvement in polynomial multiplications can be achieved by efficiently mapping $\frac{\mathbf{R}_q[x]}{\langle \prod_{i=0}^{n} g_i(x) \rangle}$ to $\prod_{i=0}^{n} \frac{\mathbf{R}_q[x]}{\langle g_i(x) \rangle}$ and computing multiplication in $\prod_{i=0}^{n} \frac{\mathbf{R}_q[x]}{\langle g_i(x) \rangle}$. CRT is extensively utilized in Sect. 3.3 for polynomial ring transformation in NTT.

2.4 The ARMv8 Architecture

In this study, we conduct implementations on the ARMv8 architecture [2]. Besides usual 32- and 64-bit operations, ARMv8 offers a set of instructions for 32 128-bit Single Instruction Multiple Data(SIMD) registers, known as NEON. Specifically, ARMv8 NEON instructions can be used on two 64-bit, four 32-bit, eight 16-bit, or sixteen 8-bit integers in each register.

2.5 Modular Arithmetic

We introduce two widely employed reduction algorithms for modular arithmetic, which are Barrett [3] and Montgomery [22] reductions. In the context of NTRU Prime, modular reduction after multiplication are critical operations for efficiency. Therefore, we leverage the implementations of these reductions on ARMv8, as suggested in [4,20], to accelerate our efforts.

Barrett Reduction. Let q be an odd number such that $q < R = 2^k$, and $a, b \in \mathbb{Z}$. Algorithm 3 and Algorithm 4 effectively calculate $(a \mod q)$ and $(ab \mod q)$ in $\mathbb{Z}_q$, respectively. Considering $x \equiv a \mod q = a - (\text{round}\left(\frac{a}{q}\right) \cdot q)$, we implement reduction by substituting one multiplication and one shift for the division, as shown in Algorithm 3. Algorithm 4 computes $(ab \mod q)$ in the same way. Furthermore, according to [4], when computing $a \pm bc$ and one of b or c is known, the second step of Algorithm 4 can be replaced by `mla` or `mls`, saving one instruction per computation.

Algorithm 3. Barrett reduction

Input: $x, 2^e < q < 2^{e+1}, R = 2^k$

Output: $x \equiv a \mod q$, $|x| \leq \frac{(q-1)}{2}$

$d \leftarrow \texttt{qrdmulh}(a, \lfloor \frac{2^{e-1}R}{q} \rceil)$

$d \leftarrow \texttt{srsra}(d, e)$

return $\texttt{mlsq}(n, d, q)$

Algorithm 4. Barrett multiplication

Input: $a, b, q, R = 2^k$

Output: $x \equiv ab \mod q$, $|x| \leq \frac{(q-1)}{2}$

$d \leftarrow \texttt{qrdmulh}(a, \lfloor \frac{\frac{bR}{q}}{2} \rceil)$

$x \leftarrow \texttt{mul}(a, b)$

return $\texttt{mls}(x, d, q)$

Montgomery Reduction. Let q be an odd number, $0 < a, b < q$, and $R = 2^k$. Montgomery multiplication accelerates modulo operation by mapping the multiplication result into "Montgomery space". Specifically, from line 1 to 5, Algorithm 5 calculates $c_R = abR^{-1} \mod q$ instead of $c = ab \mod q$. In this Montgomery space, one controls the range of value by subtracting some multiple of q (line 4) to enable the operation of modulo R (line 5). The inverse mapping process (line 6–9) transforms the numbers in Montgomery space into its original form as $abR^{-1} \cdot R \mod q$. In general, Montgomery multiplication turns the modulo q operation to modulo R operation to increase the efficiency in the case of massive consecutive multiplications.

Moreover, when b is known, we integrate the inversion mapping by preparing $bR \mod q$ and $bR \mod q \cdot (q^{-1} \mod R)$ beforehand to control the range of numbers. Thus, we save one instruction and inversion mapping.

Algorithm 5. Montgomery multiplication for Neon

Input: a, b, q, R

Output: $c = ab \mod q$

1: $low \leftarrow \texttt{mul}(a, (q^{-1} \mod R))$

2: $high \leftarrow \texttt{qdmulh}(a, b)$

3: $d \leftarrow \texttt{mul}(low, b)$

4: $e \leftarrow \texttt{qdmulh}(d, q)$

5: $c_R \leftarrow \texttt{hsub}(high, e)$

6: ---

7: $low \leftarrow \texttt{mul}(c_R, (xq^{-1} \mod R))$ ▷ after operating n-times inner products

8: $high \leftarrow \texttt{qdmulh}(c_R, x)$ ▷ where x is a $R^{n+1} \mod q$

9: $e \leftarrow \texttt{qdmulh}(low, q)$

10: $c \leftarrow \texttt{hsub}(high, e)$

According to prior research [4,11,18], we conclude that Montgomery modular multiplication is well-suited for divstepx (Algorithm 1), where numerous multiplications are performed continuously. In contrast, Barrett reduction can operate directly and takes advantages of fewer instructions when known values, e.g., constants in NTT-based multiplication, are provided for multiplication.

3 Polynomial Multiplication

We present algorithms for polynomial multiplications over $\mathbb{F}_q[x]$ where q is a prime in this section.

3.1 Karatsuba

Karatsuba multiplications [21] multiplies two polynomials with three half-length polynomial multiplications and a series of additions and subtractions. Let $f(y) = f_0 + f_1 y$ and $g(y) = g_0 + g_1 y$, where f_0, f_1, g_0, g_1 are polynomials in $\mathbb{F}_q[x]$, their product is $f(y)g(y) = [f_0 g_0] + [(f_0 + f_1)(g_0 + g_1) - (f_0 g_0 + f_1 g_1)]y + [f_1 g_1]y^2$.

It splits polynomials into two parts, as $f_0 + tf_1$, and then evaluates them at the points set $t = \{0, 1, \infty\}$, which explains why we need three multiplications to compute the product of two polynomials. The asymptotic complexity is well understood to be $n^{\log_2 3}$ where n is the length of the polynomials.

We utilize Karatsuba in polynomials of relatively shorter lengths where NTTs has not yet demonstrated an advantage.

3.2 Toom-Cook

Our polynomials can be subdivided into length-N polynomials in a variable t and evaluated at $2N - 1$ different t's analogous to Karatsuba. This multiplication, named Toom-N, has asymptotic complexity $O(n^{\log_N (2N-1)})$ [13]. While the complexity of Toom is lower than Karatsuba, it entails more preliminary computations and so tends to demonstrate advantages only in longer polynomials. Our test results in Sects. 5.1 and 5.2 indicate that Toom is more efficient in lengths ranging between 32 and 64, with no suitable utilization when inverting in $\mathbb{F}_{4591^{761}}$. However, due to the absence of the need to find roots of unity in $\mathbb{F}_q$, Toom exhibits greater flexibility compared to NTT. For example, in our implementation of polynomial multiplication in $\mathbb{F}_{653}[x]$ in Sect. 5.4, we adapted Toom implementation for polynomials of lengths 64 and 128 and achieved satisfactory performance.

3.3 NTT

In this work, FFT/NTT-based polynomial multiplications not only offer a divided-and-conquer strategy for multiplication with lower complexity than Karatsuba and Toom but also leverage the structure of jumpdivstep within the sequence of updates through trade-off analysis (see Sect. 4.1). It turns out we employ FFT/NTT-based algorithms in jumpdivstep even for lengths of polynomials that Karatsuba is faster due to their algorithmic structures.

The NTT-based polynomial multiplication algorithms in $\mathbb{F}_q[x]$ consist of three main phases, as Fig. 1. First, depending on the factorization of $q - 1$, the input polynomials are mapped to some NTT representation, known as **Input Transformation**. Subsequently, there is **Pointwise Multiplication**, where the products of corresponding points in the NTT representation are computed. Finally, the temporary result of these multiplications in NTT representation is converted back to normal representation, called **Output Transformation**.

We now introduce five common FFT/NTT-based algorithms and provide a detailed analysis of when to employ each of them.

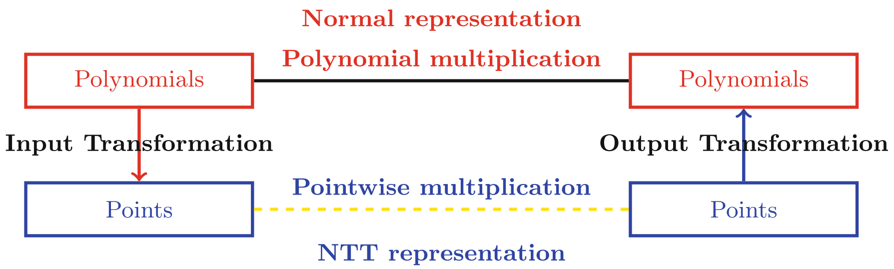

Fig. 1. The overview of FFT/NTT-based polynomial multiplication.

Cooley-Tukey. Based on CRT, we can transform a multiplication into smaller multiplications in $\mathbf{R}[x]$, e.g., $\frac{\mathbf{R}[x]}{\langle x^{2n}-c^2\rangle} \rightarrow \frac{\mathbf{R}[x]}{\langle x^n - c\rangle} \times \frac{\mathbf{R}[x]}{\langle x^n + c\rangle}$. Specifically, [14,16] mentioned that $\frac{\mathbf{R}[x]}{\langle x^n - \zeta^m\rangle}$ can be split and computed with $\prod_{i=0}^{m-1} \frac{\mathbf{R}[x]}{\langle x^{\frac{n}{m}} - \zeta\omega_m^i\rangle}$ to simplify the overall multiplication cost. Particularly, if there is an element $\zeta \in \mathbf{R}$ such that $\zeta^{2^{n-1}} = -1$, we prefer Cooley-Tukey FFT because applying it to $x^{2^n} - 1$ results in splitting down to linear polynomials.

Typically, we apply Cooley-Tukey when $\mathbb{F}_q$ has roots of unity of the order of powers of 2, i.e. $2^a | q-1$. Given its concise form, Cooley-Tukey is often favored for NTT implementations, especially when $q-1$ contains an ample number of 2 factors.

Bruun. When applying CRT in $\mathbf{R}_q$ where $q-1$ has few factors of 2, e.g., **sntrup761**, Cooley-Tukey becomes impractical for radix-2 NTTs. In the case, [20] employed Bruun FFT [10] in NTRU Prime.

The Bruun [10] FFT breaks down a polynomial ring into two distinct rings with trinomials. It is important to note that Bruun can be applied n times if $2^{n+1}|(q+1)$ and $q \equiv 3 \mod 4$. The input transformation in Bruun is

$$\mathbf{Bruun_{in}} : \frac{a_0 + a_1 x + a_2 x^2 + a_3 x^3}{\langle x^4 + (2\beta - \alpha^2)x^2 + \beta^2\rangle} \Rightarrow \frac{b_0 + b_1 x}{\langle x^2 + \alpha x + \beta\rangle} \times \frac{b_2 + b_3 x}{\langle x^2 - \alpha x + \beta\rangle}$$

where

$$\begin{aligned}(b_0, b_1) &= (a_0 - \beta a_2 + \alpha\beta a_3, a_1 + (\alpha^2 - \beta)a_3 - \alpha a_2) \quad \text{, and} \\ (b_2, b_3) &= (a_0 - \beta a_2 - \alpha\beta a_3, a_1 + (\alpha^2 - \beta)a_3 + \alpha a_2) \quad .\end{aligned}$$

We compute $(a_0 - \beta a_2, a_1 + (\alpha^2 - \beta)a_3, \alpha a_2, \alpha\beta a_3)$ and assemble them to get (b_0, b_1, b_2, b_3). The format of the output transform is

$$\mathbf{Bruun_{out}} : \frac{b_0 + b_1 x}{\langle x^2 + \alpha x + \beta\rangle} \times \frac{b_2 + b_3 x}{\langle x^2 - \alpha x + \beta\rangle} \Rightarrow \frac{a_0 + a_1 x + a_2 x^2 + a_3 x^3}{\langle x^4 + (2\beta - \alpha^2)x^2 + \beta^2\rangle}$$

where

$$\begin{aligned}2(a_0, a_1) &= (b_0 + b_2 + (b_3 - b_1)\alpha^{-1}\beta, b_1 + b_3 - (b_0 - b_2)\alpha^{-1}\beta^{-1}(\alpha^2 - \beta)) \quad \text{, and} \\ 2(a_2, a_3) &= ((b_3 - b_1)\alpha^{-1}, (b_0 - b_2)\alpha^{-1}\beta^{-1}) \quad .\end{aligned}$$

Here, it suffices to compute $(b_0 + b_2, b_1 + b_3, b_0 - b_2, b_3 - b_1)$ and then multiplies them by specific constants $(\alpha^{-1}, \beta, \alpha^{-1}\beta^{-1}, \alpha^2 - \beta)$.

Good-Thomas. The Good-Thomas FFT [17] decomposes a DFT of size N, where $N = N_1 N_2$ and $\mathsf{gcd}(N_1, N_2) = 1$, into separate DFTs of sizes N_1 (repeated N_2 times) and N_2 (repeated N_1 times).

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{-2\pi i}{N} nk} \Rightarrow X_{k_1,k_2} = \sum_{n_1=0}^{N_1-1} (\sum_{n_2=0}^{N_2-1} x_{n_1 N_2 + n_2 N_1} e^{-\frac{2\pi i}{N_2} n_2 k_2}) e^{-\frac{2\pi i}{N_1} n_1 k_1},$$

where $n = n_1 N_2 + n_2 N_1$, $k = 0, 1, 2 \ldots N-1$, $k_1 = k \bmod N_1$, and $k_2 = k \bmod N_2$.

When $N_1 \gg N_2$, the advantage of the Good-Thomas lies in partitioning a polynomial into N_2 smaller polynomials of size N_1, thereby significantly reducing computational complexity compared to one polynomial of size N. The algorithm can be recursively applied while all N_1 and N_2 remain co-prime throughout the process.

Typically, we employ Good-Thomas when there are relatively small odd roots of unity in $\mathbb{F}_q$. We utilize Good-Thomas at the initial level decomposing polynomials because it allows us to obtain the smaller NTT instance over smaller rings.

Rader. Rader's FFT [20,26] converts a DFT in $\frac{\mathbf{R}[x]}{\langle x^p - 1 \rangle}$, where p is a prime, into computing a cyclic convolution as

$$\mathbf{Rader_{in}} : \frac{\mathbf{R}[x]}{\langle x^p - 1 \rangle} \Rightarrow \prod_{i=0}^{p-1} \frac{\mathbf{R}[x]}{\langle x - \omega_p^i \rangle}$$

Moreover, Hwang [19] presented truncated Rader

$$\mathbf{Truncated\ Rader_{in}} : \frac{\mathbf{R}[x]}{\langle \Phi_p(x) \rangle} \Rightarrow \prod_{1 \le i \le p,\ \gcd(i,p)=1} \frac{\mathbf{R}[x]}{\langle x - \omega_p^i \rangle} \;,$$

where $\Phi_p(x)$ is the p-th cyclotomic polynomial for multiplication in **sntrup761**. In other words, multiplication in $\frac{\mathbf{R}[x]}{\langle \Phi_p(x) \rangle}$ is morphed into pointwise multiplications in $\frac{\mathbf{R}[x]}{\langle x - \omega_p{}^i \rangle}$ through the inherent property of the cyclotomic polynomial.

We employ Rader for composing polynomial ring with larger odd roots of unity in $\mathbb{F}_q$. In the case of **sntrup761**, $r = 17$ is a relatively large root of unity in $\mathbb{F}_{4591}$. Considering the vectorized architecture is suitable for processing polynomial of length $k \cdot (r-1) \cdot 8$, we employ truncated Rader as possible in **sntrup761** as concluded in [19].

Schönhage. Schönhage's trick [23,28] creates the root of -1 by introducing new variables instead of the approach of splitting roots of unity in other other

FFT/NTT algorithms. For instance, let $\mathbb{F}$ be a field and $n = n_1 n_2$ is a positive integer. Schönhage's trick maps $\mathbb{F}[x]/(x^n - 1)$ to $((\mathbb{F}[x][y]/(x^{n_2} - y))/(y^{n_1} - 1)$. With lifting $\mathbb{F}[x]/(x^{n_2} - y)$ to $\mathbf{R} := \mathbb{F}[x]/(x^m - 1)$ where m is a multiple of n_2, we map the original $\mathbb{F}[x]/(x^n - 1)$ to $\mathbf{R}[y]/(y^{n_1} - 1)$ which is capable for further NTT mapping.

Schönhage's algorithm does not rely on the roots of unity in $\mathbb{F}_q$, making it a general-purpose FFT algorithm. However, because it involves lifting operations, the length of polynomials in the lifted ring becomes twice as large. Therefore, we employ Schönhage's algorithm only when there is no other suitable roots of unity for other NTT algorithms.

4 Optimizing jumpdivstep

In Sect. 4.1, we analyze the cost of jumpdivstep with respect to input transform, pointwise multiplication, and output transform in multiplication. In Sect. 4.2, we introduce three different strategies for achieving jumpdivstep and compare their costs. We remove redundant computation for inversion with jumpdivstep in Sect. 4.3.

4.1 Decomposing jumpdivstep

Although the complexity of jumpdivstep is lower than divstepx, divstepx can still outperform jumpdivstep when input step n is small. Bernstein and Yang [9] pointed out the reason lies in that jumpdivstep keeps all four elements of the transition matrix while divstepx keeps only v and r. Hence, we optimize the structure of jumpdivstep and polynomial multiplication algorithms to lower its cost.

Besides the speed of multiplication algorithms, we minimize the number of input and output transforms among the 2 matrix-vector and 1 matrix-matrix multiplication in jumpdivstep. We reuse the input transforms and exploit the additive property of output transforms to remove redundant transforms. Due to heavier input/output transforms, NTT-based algorithms have more advantages than Karatsuba or Toom-based algorithms when applying the techniques. Our experiments in Sect. 5.2 show NTT-based multiplication can result in faster jumpdivstep even when it is not the fastest multiplication algorithm among our comparisons.

Let the transition matrices be $M_1 = \begin{bmatrix} u_1 & v_1 \\ q_1 & r_1 \end{bmatrix}$ and $M_2 = \begin{bmatrix} u_2 & v_2 \\ q_2 & r_2 \end{bmatrix}$ in Algorithm 2. It performs two matrix-vector multiplication (MxV) as

$$\begin{bmatrix} f' \\ g' \end{bmatrix} = x^{-n} \times \begin{bmatrix} u_1 & v_1 \\ q_1 & r_1 \end{bmatrix} \times \begin{bmatrix} f \\ g \end{bmatrix} \tag{2}$$

to update the 2 input polynomials. One MxV requires 6 input transforms, 4 pointwise multiplications, and 2 output transforms. At line 10, the matrix-matrix multiplication (MxM)

$$M_2 \cdot M_1 = \begin{bmatrix} u_2 & v_2 \\ q_2 & r_2 \end{bmatrix} \times \begin{bmatrix} u_1 & v_1 \\ q_1 & r_1 \end{bmatrix} \tag{3}$$

computes the output transition matrix. At this step, we reuse the NTT representation of M_1 and M_2 and thus MxM requires only 8 pointwise multiplications and 4 output transforms. In a nutshell, we keep the NTT representations of transition matrices from MxVs and reuse them while doing MxM. Figure 2 depicts the details of the computation.

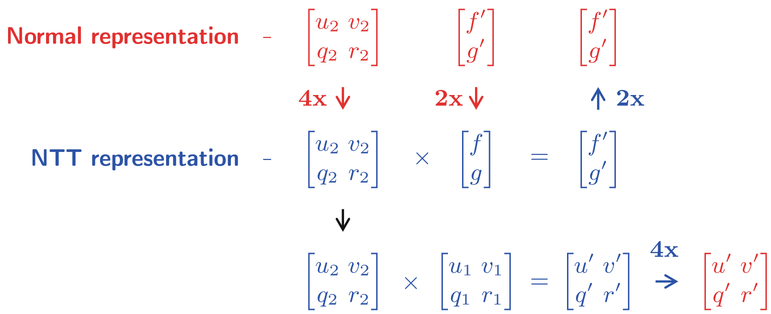

Fig. 2. MxV and MxM in NTT representation. Red arrows represent input transforms. Blue arrows represent output transforms. (Color figure online)

When the total division steps are a multiple of 3, we save more input and output transforms by decomposing one jumpdivstep into 3 smaller jumpdivstep. Figure 3 and Fig. 4 show the structure of radix-2 and radix-3 jumpdivstep. There are 2 consecutive MxM operations in the radix-3 jumpdivstep. Each MxM saves 8 input transforms by reusing them from previous MxV operations. When performing two consecutive MxM operations, we keep the results of the first MxM in NTT representation and then multiply them by the second matrix immediately. It only requires 4 output transforms for the output transition matrix.

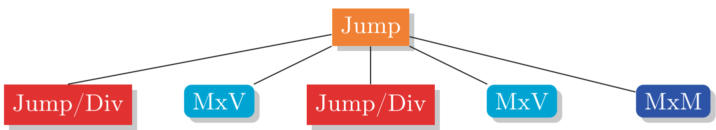

Fig. 3. radix-2 jumpdivstep

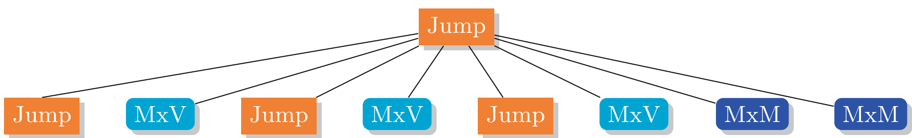

Fig. 4. radix-3 jumpdivstep

4.2 Polynomial Representations in jumpdivstep

Since the maximum polynomial degrees of row vectors (u, v) and (q, r) differ by one in the transition matrix, we present three different strategies for storing polynomials. These strategies affect the efficiency for performing MxV and MxM operations. At line 10 in Algorithm 1, divstepx raises the degree of polynomials (u, v) by one for each iteration and brings an inconsistent of degrees between the polynomials (u, v) and (q, r). Since raising the degree does not produce new coefficients for (u, v), we can still store the (u, v) in its original storage without increasing the storage size for coefficients. However, different polynomial representations affects the efficiency of jumpdivstep.

Saturated divstepx. We denote Saturated divstepx as the strategy that sufficiently utilizes all storage of vector registers while keeping coefficients aligned as possible. Assuming a vector register stores m coefficients, the strategy performs n steps of divstep where $m|n$ and stores n coefficients for each polynomial. However, during these steps, the pairs (f, g), (u, q), and (v, r) may not swap or may have only been swapped once in the first loop during divstep. This leads to (u, v) being multiplied by x a total of n times. If we store intact coefficients ranging from degrees-0 to $n1$ in vector registers, the situation causes overflow. Thus, we assign the lowest position of the registers as degree-n and perform the multiplication by x by rotating the storage space. The highest coefficient of (u, v) would rotate back to the lowest position in the vector register when overflow occurs and the degree-n term is securely saved. Noted that all of the other coefficients would be 0 in the situation because it occurs only when $g_0 = 0$ throughout the n iterations. When computing MxV and MxM, we split the degree-n term from (u, v) and perform polynomial multiplication for the rest of the degree-$(n-1)$ parts of (u, v). Therefore, besides a normal polynomial multiplication of degree-$(n-1)$, we need post-processing for the case that (u, v) are single-term polynomials of degree-n. These multiplications by a single term are processed by conditional addition. Thus MxV in Saturated divstepx includes 6 input transforms, 4 point-wise multiplication, 2 output transforms for degree-$(n-1)$ parts as Eq. 2, and post-processing for conditionally adding input f and g to updated destination with masks of degree-n terms from (u, v).

For performing MxM, there are 2 polynomials (u, v) that we need to adjust. The procedure includes a matrix by matrix multiplication and adjustments on $(u_2 \cdot u_1, v_2 \cdot q_1, u_2 \cdot v_1, v_2 \cdot r_1, q_2 \cdot u_1, q_2 \cdot v_1)$:

1. $\begin{bmatrix} u' & v' \\ q' & r' \end{bmatrix} = \begin{bmatrix} u_2 & v_2 \\ q_2 & r_2 \end{bmatrix} \times \begin{bmatrix} u_1 & v_1 \\ q_1 & r_1 \end{bmatrix}$.
2. If $u_2[0]$ or $v_2[0] = 1$, $u_2 \cdot u_1, v_2 \cdot q_1, u_2 \cdot v_1, v_2 \cdot r_1$ has to multiply by x^n.
3. If $u_1[0]$ or $v_1[0] = 1$, $u_2 \cdot u_1, u_2 \cdot v_1, q_2 \cdot u_1, q_2 \cdot v_1$ has to multiply by x^n.
4. If both conditions are satisfied, $u_2 \cdot u_1, u_2 \cdot v_1$ don't have to do any of the adjustments above.

To make the conditional multiplication in constant time, we use some masks to represent if u or v is degree-n. Then we do a condition swap on the higher and lower half of the product polynomials based on the masks. To make as less adjustments as possible, we modify Saturated divsteps into Sheared divsteps.

Sheared divstepx. In Sheared divsteps, we skip the last degree raising $\begin{bmatrix} u \ v \end{bmatrix} \leftarrow \begin{bmatrix} u \ v \end{bmatrix} \cdot x$ from Algorithm 1. This makes (u, v) multiply by x only $n-1$ times but results in different alignment of coefficients for polynomials between (u, v) and (q, r). In the strategy, the rotation and the mask operations are unnecessary. The output transition matrix becomes $\begin{bmatrix} u/x & v/x \\ q & r \end{bmatrix}$ and MxV in Sheared divsteps is shown as follows:

1. $\begin{bmatrix} f'/x \\ g' \end{bmatrix} = x^{-n} \times \begin{bmatrix} u/x & v/x \\ q & r \end{bmatrix} \times \begin{bmatrix} f \\ g \end{bmatrix}.$
2. Multiply f'/x by x.

MxM is also modified to

1. $\begin{bmatrix} u' & v' \\ q' & r' \end{bmatrix} = \begin{bmatrix} (u_2/x) \cdot x & (v_2/x) \cdot x \\ q_2 & r_2 \end{bmatrix} \times \begin{bmatrix} (u_1/x) \cdot x & (v_1/x) \cdot x \\ q_1 & r_1 \end{bmatrix}.$
2. Divide u', v' by x to obtain $u'/x, v'/x, q, r$.

However, some products in MxM such as $(\frac{u_2}{x} \cdot x \cdot \frac{u_1}{x} \cdot x) + (\frac{v_2}{x} \cdot x \cdot q_1)$ become degree-$2n$ polynomials, which is 1 coefficient longer than the storage space and unfriendly for vectorized NTT algorithms. Therefore, we cannot perform pure matrix by matrix multiplication in NTT representation. Although it may appear that there are not many redundant operations in MxV and MxM, this version is slower than our best strategy in practice. We aim to make all the computations of MxV and MxM feasible to compute in NTT representation in vectorized storage space.

Unsaturated divstepx. In Unsaturated divstepx, we execute fewer steps of divstepx than the storage size, e.g., performing $n-1$ steps for storage of size n. This straightforward adjustment effectively eliminates all overhead present in previous versions.

Within a vectorized hardware structure, we aim to maximize the utilization of Unsaturated divstepx. Should the available steps prove insufficient, our approach involves supplementing them by uniformly substituting some Unsaturated divstepx with Sheared divstepx to attain additional steps.

Comparison. Now, we compare the required computations with strategies of storing (u, v). Assume n is the number of elements in a vector register, and m is the number of registers for each polynomial in divstepx. In Saturated divstepx, we first perform $m \times n$ steps of divstepx and then we compute two MxV and one MxM separately. It takes 6 input transforms, 4 pointwise multiplications, and 4 output transforms in one MxV. One MxM takes 8 pointwise multiplications and 8 output transforms. We use a mask vector to check if u or v is the degree of $m \cdot n$. It takes one `ceq`, one `dup`, and one `orr` NEON instruction. Then, we swap

Table 2. Operation counts in different methods of jumpdivstep

	Operation	In	Mul	Out	ceq	dup	and	or	mvn	ext
MxV	**Saturated**	6	4	2	2	2	2m	2m	0	0
	Sheared	6	4	2	0	0	0	0	0	2m
	Unsaturated	6	4	2	0	0	0	0	0	4m
MxM	**Saturated**	0	8	4	0	0	10m	8m	2(m−1)	0
	Sheared	0	8	8	0	0	0	0	0	8m
	Unsaturated	0	8	4	0	0	0	0	0	0

the higher and lower half of the polynomial based on the mask, which takes some `orr`, some `and`, and some `mvn` instructions. We use one more mask in MxM to identify if the situation happens in both matrices.

For Sheared divsteps, we also use 6 input transforms, 4 pointwise multiplications, and 2 output transforms in each MxV. Because both (u, v) are 1 degree short, we compute MxV as matrix-vector multiplication in NTT representation and use `ext` instructions to adjust the degree of the result f'. As for MxM, it still takes 8 pointwise multiplications, and 8 output transforms. Because all (u_2, v_2, u_1, v_1) are 1 degree shorter than the storage space, we can adjust degrees with `ext` instructions without overflowing storage space when summing up polynomials.

Unsaturated divstepx computes MxV and MxM as normal multiplications in NTT representation without processing the overflow situation. It only takes 6

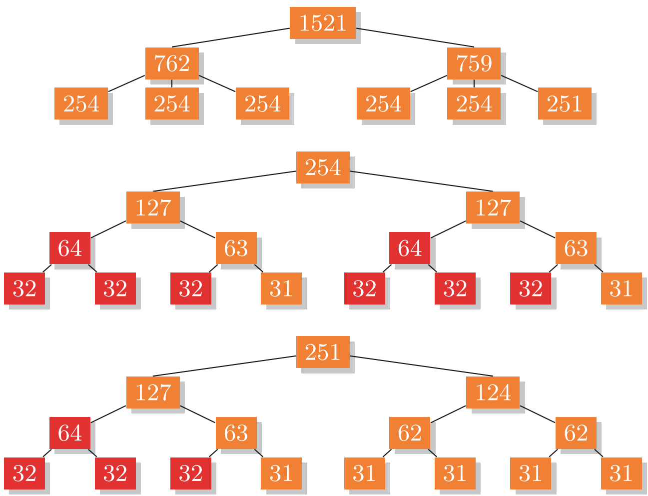

Fig. 5. jumpdivstep structure for **invsntrup761**. Red and orange boxes represent sheared and unsaturated divstepx, respectively.

input transforms, 4 pointwise multiplication, 2 output transforms, and some ext for each MxV. MxM takes 8 pointwise multiplications and 4 output transforms.

We list all operation counts in Table 2. According to the result, unsaturated divstepx is our best strategy for jumpdivstep and sheared divstepx is the second choice. Therefore, if the number of layers and radix of each layer is set, we conclude our approach to jumpdivstep:

1. Use unsaturated divstepx as much as possible.
2. If it still lacks steps, replace unsaturated divstepx with sheared divstepx evenly to gain the extra steps.

We finally implement jumpdivstep for **invsntrup761** as Fig. 5.

4.3 Optimization for Computing Reciprocal Elements

When computing polynomial inversion in a polynomial ring as Eq. 1, we need only the v polynomial in the resulting transition matrix instead of the full matrix. Thus we reduce operations in jumpdivstep structure.

We show these reductions through the example $\mathsf{jumpdivstep}_{1521}$ as Fig. 5. In the last jumpdivstep of each layer, we omit the second MxV and conduct a reduced MxM as

$$\begin{bmatrix} u' \; v' \end{bmatrix} = \begin{bmatrix} u_2 \; v_2 \end{bmatrix} \times \begin{bmatrix} u_1 & v_1 \\ q_1 & r_1 \end{bmatrix} .$$

In the second layer, we reduce the MxM operation in the first jumpdivstep to

$$\begin{bmatrix} v' \\ r' \end{bmatrix} = \begin{bmatrix} u_2 & v_2 \\ q_2 & r_2 \end{bmatrix} \times \begin{bmatrix} v_1 \\ r_1 \end{bmatrix} .$$

In the top layer of $\mathsf{jumpdivstep}_{1521}$, we compute the final v with an inner product

$$\begin{bmatrix} v' \end{bmatrix} = \begin{bmatrix} u_2 \; v_2 \end{bmatrix} \times \begin{bmatrix} v_1 \\ r_1 \end{bmatrix} .$$

Moreover, since we use the same ring structure for polynomial multiplication in $\mathsf{jumpdivstep}_{762}$, $\mathsf{jumpdivstep}_{759}$ and $\mathsf{jumpdivstep}_{1521}$, we omit output transforms after the MxM operation in $\mathsf{jumpdivstep}_{762}$ and $\mathsf{jumpdivstep}_{759}$ and pass the matrices in NTT representation to $\mathsf{jumpdivstep}_{1521}$ to save further operations.

5 Implementations

In this section, we showcase our optimized Neon implementations for **invsntrup761**, **inv3ntrup761**, and **invsntrup653**. Section 5.1 evaluates polynomial multiplications with different algorithms on various lengths. Section 5.2 analyzes the performance of jumpdivstep regarding the number of recursive layers in **invsntrup761**. Section 5.3 and Sect. 5.4 detail the implementations for **inv3ntrup761** and **invsntrup653**, respectively. Section 5.5 compares our jumpdivstep implementation with divstepx and benchmarks the resulting key generation with other **sntrup761** implementations.

5.1 Base Polynomial Multiplication

In this section, we explore approaches to implement multiplications on various lengths of polynomials. Table 3 shows the profiles, including input/output transforms (**In**/**Out**) and point-wise multiplication(**Mul**), of polynomial multiplication(**PxP**) as well as the performance of the resulting matrix (**MxV** and **MxM**) and jump operations measuring in Arm Cortex-A72. All the implementations apply to $q < 2^{16}$ except NTT-based multiplications are tailored to $q = 4591$.

Table 3. Cycle counts for various operations in $\mathbb{F}_{4591}[x]$

Length	Algorithm	In	Mul	Out	PxP	MxV	MxM	Jump
8 × 8	Schoolbook	0	94	0	94	376	752	1,504
	Karatsuba	0	56	0	56	224	448	896
	Extend	**0**	**50**	**0**	**50**	**200**	**400**	**800**
	Batched(x8)	**0**	**360**	**0**	**360**	-	-	-
16 × 16	Schoolbook	0	231	0	231	924	1,848	3,696
	Karatsuba	**0**	**182**	**0**	**182**	**728**	**1,456**	**2,912**
32 × 32	Schoolbook	0	760	0	760	3,040	6,080	12,160
	Toom	114	374	462	950	2,762	5,296	10,364
	Karatsuba	**0**	**614**	**0**	**614**	**2,456**	**4,912**	**9,824**
64 × 64	Schonhage	367	2,319	521	3207	11419	22,104	43,474
	Karatsuba	0	1,999	0	1,999	7,996	15,992	31,984
	Toom	207	1,295	944	2,446	7,689	14,964	29,514
	Rader	**1,228**	**411**	**570**	**2,209**	**6,468**	**10,480**	**18,504**
128 × 128	Karatsuba	0	6,998	0	6,998	27,992	55,984	111,968
	Schonhage	1,691	4,903	1,521	8,115	27,727	52,072	100,762
	Toom	454	3,096	1,896	5,446	17,538	34,168	67,428
	Bruun	1,982	2,443	1,764	6,189	19,246	34,528	65,092
	Rader	**2,908**	**828**	**1,240**	**4,976**	**14,516**	**23,216**	**40,616**
768	**Good-3**	**11,022**	**2,494**	**5,349**	**18,865**	**53,740**	**85,436**	**222,520**

8×8: The schoolbook multiplication outperforms Karatsuba in our 8×8 implementations. Among our two schoolbook implementations, the Extend version provides the fastest implementation. This method leverages the `SMULL` instruction to multiply two sets of 16-bit integers, yielding two sets of extended 32-bit integers. Subsequently, the addition operation is performed on 32-bit integers without modular operations. After accumulating all 32-bit products, we bring results back to 16-bit with Barrett reductions.

Additionally, we develop a batched 8×8 implementation providing better throughput for performing 8 multiplications in parallel. It based on the same Extend technique. When performing one schoolbook multiplication, we use the `EXT` instruction to align coefficients of different degrees. These data movements are replaced by accessing registers when storing coefficients of the same degree from 8 different batches in one register. However, we need extra transpose (`TRN`) instructions to rearrange the data before and after the batched multiplication. The batched implementation is useful since the 8×8 multiplication serves as the foundation for longer multiplications.

16 × 16 : Karatsuba emerges as the faster option in our 16×16 implementations. We compare only schoolbook and Karatsuba since $\text{Toom}_{16\times16}$ results in a 4×4 sub-multiplication.

32 × 32: Karatsuba remains the fastest choice among our 32×32 implementations. Although $\text{Toom}_{32\text{x}32}$ uses seven 8×8 multiplications while $\text{Karatsuba}_{32\text{x}32}$ uses nine 8×8 multiplications, Toom is slower due to its heavy output transform.

64 × 64: In 64×64 multiplications, truncated Rader emerges as the fastest method for $q = 4591$, while Toom remains the fastest in general cases. For multiplying polynomials of length 64, NTT-based algorithm shows its advantage since it uses 16 sets of 8×8 for multiplying polynomials of length 64 while $\text{Karatsuba}_{64\times64}$ uses 27. Hence we devise an implementation of truncated Rader, utilizing the root 17 of $q = 4591$ to divide a ring of length 128 into 16 rings of length 8. Then, we perform 16 batches of Extend 8×8.

128 × 128: Rader remains the fastest method for 128×128 multiplication. We include a Bruun implementation for comparison. $\text{Bruun}_{128\times128}$ partitions a ring of 256 coefficients into 16 sets of 16×16 using Cooley-Tukey, resulting in 48 sets of 8×8 operations with $\text{Karatsuba}_{16\times16}$. In contrast, $\text{Rader}_{128\times128}$ divides the ring into 32 sets of 8×8 using a layer of Cooley-Tukey.

Table 4. Estimation time for jumpdivstep in different number of layers

invsntrup761		time	count	MxV+MxM	Recip
0 layer	divstepx 1521	11,750,719	1	0	11,750,719
1 layer	divstepx 251	639,990	1	445,040	4,322,970
	divstepx 254	647,588	5		
2 layers	divstepx 124	151,885	1	688,736	2,551,825
	divstepx 127	155,564	11		
3 layers	divstepx 62	33,602	2	910,784	1,733,017
	divstepx 63	34,139	11		
	divstepx 64	34,500	11		
4 layers	divstepx 31	8,267	15	1,146,560	1,553,078
	divstepx 32	8,561	33		
5 layers	divstepx 15	2,291	15	1,286,336	1,515,830
	divstepx 16	2,409	81		
6 layers	divstepx 7	681	15	1,363,136	1,498,844
	divstepx 8	709	177		

5.2 jumpdivstep in $\mathbb{F}_{4591}[x]/(x^{761} - x - 1)$

For the multiplication of long polynomials, we use an NTT of size 768 in $\mathsf{jumpdivstep}_{769}$, $\mathsf{jumpdivstep}_{762}$, and $\mathsf{jumpdivstep}_{1521}$ because they all result in polynomials of length < 761 in **invsntrup761**. Since there's only one additional layer of radix-3 NTT, we directly apply a radix-3 Good-Thomas approach on $\mathrm{Rader}_{128\times128}$.

In Table 4, we extrapolate the execution time of different layers of jumpdivstep through Table 3 and the benchmark of divstepx. To elaborate, we accumulate the total execution time of divstepx in the lowest layer with the cumulative MxV and MxM execution time. This analysis continues until reaching the maximum decomposition level that the size of a polynomial is equal to the register size, i.e., 6 layers in ARMv8. Notably, the expected execution time persistently reduces as the number of layers of jumpdivstep increases.

5.3 jumpdivstep in $\mathbb{F}_3[x]/(x^{761} - x - 1)$

For $q = 3$, we use bit-slice representation for processing 2-bit $\mathbb{F}_3$ coefficients, i.e., place the 2 bits in different registers and perform arithmetic with bit operations simultaneously on 128 coefficients. Therefore, the minimum steps of divstepx becomes 128 as the size of the registers.

We start jumpdivstep for polynomials of length > 128 and develop 128×128 polynomial multiplication over $\mathbb{F}_3$. For performing multiplication with NEON integer instructions, we first rearrange the 2-bit bit-slice data into 8-bit numbers. Then we apply `UMULL` to multiply 32-bit numbers to 64-bit products, which is equivalent to 4×4 polynomial multiplication on 8-bit coefficients. While each register contains 4 32-bit elements, we implement a 16×16 polynomial multiplication with a 4×4 schoolbook multiplication on 32-bit elements. We utilize Karatsuba to build multiplications for longer polynomials based on the 16×16 multiplication.

5.4 jumpdivstep in $\mathbb{F}_{4621}[x]/(x^{653} - x - 1)$

We implement **invsntrup653**, which uses the smallest parameter sets of NTRU Prime, to show that jumpdivstep consistently outperforms divstepx. Figure 6 depicts the structure of our **invsntrup653** implementation.

We choose polynomial multiplication of closed 2-powered lengths from Sect. 5.1 when implementing multiplication of non-2-powered lengths. For example, we utilize $\mathrm{Toom}_{64\times64}$ and $\mathrm{Toom}_{128\times128}$ for 56×56 and 112×112 polynomial multiplications. For 448×224 and 672×672, we employ a layer of Good-Thomas-3 followed by 2 layers of Cooley-Tukey, succeeded by 12 sets of $\mathrm{Toom}_{64\times64}$.

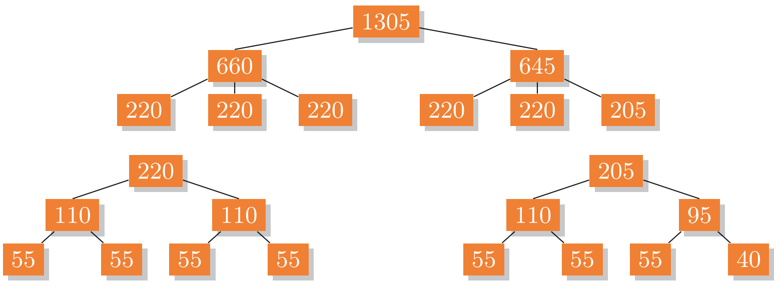

Fig. 6. jumpdivstep structure for **invsntrup653**

5.5 Benchmark

Table 5. Cycle counts for polynomial inversion in **sntrup761** and **sntrup631**

Inversion		Cortex-A53	Cortex-A72	Cortex-A76	M1
invsntrup761	divstepx	5,819,737	4,949,369	2,703,328	851,937
	jumpdivstep	2,031,221	1,457,946	1,150,369	317,285
inv3ntrup761	divstepx	839,852	568,323	344,628	199,455
	jumpdivstep	625,518	531,825	279,444	154,286
invsntrup653	divstepx	4,286,957	3,640,900	1,985,212	645,106
	jumpdivstep	3,342,664	2,351,016	1,819,333	579,342

Table 5 shows our final results of optimizations. As a result, jumpdivstep spends only 29% cycle counts to complete an **invsntrup761** operation compared to divstepx on Cortex-A72. Table 5 also shows the advantage of jumpdivstep over divstepx in **inv3ntrup761**. jumpdivstep outperforms divstepx even in the smallest parameter **invsntrup653** in NTRU Prime.

While integrating the inversion operations to key generation of NTRU Prime in Table 6, we exert significant effort on **sntrup761** particularly due to its

Table 6. Cycle counts for key generation in **sntrup761**

sntrup761	Cortex-A53	Cortex-A72	Cortex-A76	M1
`ref` from supercop [8]	33,504,035	23,837,956	16,958,229	13,449,469
divstepx [20]	6,547,768	5,517,692	3,047,956	1,051,392
jumpdivstep	2,569,555	1,969,656	1,429,813	471,571
jumpdivstep/`ref`	7.66%	8.26%	8.43%	3.5%
jumpdivstep/divstepx	39.24%	**35.69%**	46.91%	**44.85%**

widespread use in OpenSSH. Our implementation spends only 35.69% running time on Cortex-A72 compared to the version using `divstepx` as reported in [20], and 44.85% on M1.

Acknowledgement. We thank Jin-Han Liu and Vincent Hwang for valuable suggestions and discussions. This project was supported by TACC project NSTC-112-2634-F-001-001-MBK and the Academia Sinica Investigator Award AS-IA-109-M01.

References

1. Alkim, E., et al.: Polynomial multiplication in NTRU prime: comparison of optimization strategies on Cortex-M4. Cryptology ePrint Archive, Report 2020/1216 (2020)
2. ARM: Arm architecture reference manual armv8, for a-profile architecture (2021)
3. Barrett, P.: Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In: Odlyzko, A.M. (ed.) CRYPTO 1986. LNCS, vol. 263, pp. 311–323. Springer, Heidelberg (1987). https://doi.org/10.1007/3-540-47721-7_24
4. Becker, H., Hwang, V., Kannwischer, M.J., Yang, B.Y., Yang, S.Y.: Neon NTT: faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1. IACR Trans. Crypt. Hardware Embed. Syst. **2022**(1), 221–244 (2022)
5. Bernstein, D.J., et al.: NTRU Prime. Technical report, National Institute of Standards and Technology (2020). https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions
6. Bernstein, D.J., Brumley, B.B., Chen, M., Tuveri, N.: OpenSSLNTRU: faster post-quantum TLS key exchange. In: Butler, K.R.B., Thomas, K. (eds.) 31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, 10–12 August 2022, pp. 845–862. USENIX Association (2022)
7. Bernstein, D.J., Chuengsatiansup, C., Lange, T., van Vredendaal, C.: NTRU Prime. Technical report, National Institute of Standards and Technology (2019). https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-2-submissions
8. Bernstein, D.J., Lange, T.: eBACS: ecrypt benchmarking of cryptographic systems. https://bench.cr.yp.to. Accessed 19 Feb 2024
9. Bernstein, D.J., Yang, B.Y.: Fast constant-time gcd computation and modular inversion. IACR Trans. Crypt. Hardware Embed. Syst. **2019**(3), 340–398 (2019). https://doi.org/10.13154/tches.v2019.i3.340-398
10. Bruun, G.: z-transform DFT filters and FFT's. IEEE Trans. Acoust. Speech Sig. Process. **26**(1), 56–63 (1978)
11. Cao, Z., Wei, R., Lin, X.: A fast modular reduction method. Cryptology ePrint Archive, Report 2014/040 (2014). https://eprint.iacr.org/2014/040
12. Chen, C., et al.: NTRU. Technical report, National Institute of Standards and Technology (2020). https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions
13. Cook, S.A., Aanderaa, S.O.: On the minimum computation time of functions. Trans. Am. Math. Soc. **142**, 291–314 (1969)
14. Cooley, J.W., Tukey, J.W.: An algorithm for the machine calculation of complex Fourier series. Math. Comput. **19**, 297–301 (1965)

15. Deshpande, S., del Pozo, S.M., Mateu, V., Manzano, M., Aaraj, N., Szefer, J.: Modular inverse for integers using fast constant time GCD algorithm and its applications. In: International Conference on Field-Programmable Logic and Applications (FPL) (2021)
16. Gentleman, W.M., Sande, G.: Fast Fourier transforms: for fun and profit. In: Proceedings of the Fall Joint Computer Conference, AFIPS 1966 (Fall), 7–10 November 1966, pp. 563–578. Association for Computing Machinery, New York, NY, USA (1966). https://doi.org/10.1145/1464291.1464352
17. Good, I.J.: Random motion on a finite Abelian group. Math. Proc. Camb. Philos. Soc. **47**, 756–762 (1951)
18. Huang, J., et al.: Improved plantard arithmetic for lattice-based cryptography. IACR Trans. Crypt. Hardware Embed. Syst. **2022**(4), 614–636 (2022)
19. Hwang, V.: Pushing the limit of vectorized polynomial multiplication for NTRU prime. Cryptology ePrint Archive, Paper 2023/604 (2023)
20. Hwang, V., Liu, C.T., Yang, B.Y.: Algorithmic views of vectorized polynomial multipliers - NTRU prime. Cryptology ePrint Archive, Paper 2023/1580 (2023)
21. Karatsuba, A., Ofman, Y.: Multiplication of multidigit numbers on automata. Sov. Phys. Dokl. **7**, 595 (1962)
22. Montgomery, P.L.: Modular multiplication without trial division. Math. Comput. **44**(170), 519–521 (1985)
23. Nussbaumer, H.: Fast polynomial transform algorithms for digital convolution. IEEE Trans. Acoust. Speech Sig. Process. **28**(2), 205–215 (1980)
24. OpenSSH: OpenSSH 9.0 release notes (2022)
25. Paksoy, I.K., Cenk, M.: Faster NTRU on ARM cortex-M4 with TMVP-based multiplication. Cryptology ePrint Archive, Report 2022/300 (2022)
26. Rader, C.: Discrete Fourier transforms when the number of data samples is prime. Proc. IEEE **56**(6), 1107–1108 (1968)
27. Richter-Brockmann, J., Chen, M.S., Ghosh, S., Güneysu, T.: Racing BIKE: improved polynomial multiplication and inversion in hardware. IACR Trans. Crypt. Hardware Embed. Syst. **2022**(1), 557–588 (2022)
28. Schönhage, A., Strassen, V.: Schnelle multiplikation großer zahlen. Computing **7**, 281–292 (1971). https://api.semanticscholar.org/CorpusID:9738629
29. Yeniaras, E., Cenk, M.: Faster characteristic three polynomial multiplication and its application to NTRU prime decapsulation. Cryptology ePrint Archive, Report 2020/1336 (2020). https://eprint.iacr.org/2020/1336
30. Yeniaras, E., Cenk, M.: Faster characteristic three polynomial multiplication and its application to NTRU Prime decapsulation. J. Cryptogr. Eng. **12**(3), 329–348 (2022). https://doi.org/10.1007/s13389-021-00282-7

Author Queries

Chapter 6

Query Refs.	Details Required	Author's response
AQ1	This is to inform you that corresponding author has been identified as per the information available in the Copyright form.	