Isomorphic Synthesis of Fast Multiplication Algorithms for Post Quantum Cryptography

WILLIAM SCARBRO, SANJAY RAJOPADHYE

ACM Reference Format:
William Scarbro, Sanjay Rajopadhye. 2023. Isomorphic Synthesis of Fast Multiplication Algorithms for Post Quantum Cryptography. 1, 1 (September 2023), 25 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

ABSTRACT

Multiplication over polynomial rings is a time consuming operation in many Post-Quantum Cryptosystems. State of the art implementations of multiplication for these cryptosystems have been developed by hand using an algebraic framework. A similar class of algorithms, based on the Discrete Fourier Transform (DFT), have been optimized across a variety of platforms using program synthesis. We demonstrate how the algebraic framework used to describe fast multiplication algorithms can be used in program synthesis. Specifically, we extend and then abstract this framework for use in program synthesis allowing AI search techniques to find novel, high performance implementations of polynomial ring multiplication on multiple platforms.

INTRODUCTION

Post-Quantum Cryptosystems which are variants of the Ring Learning With Errors (RLWE) problem [1] rely heavily on operations in polynomial rings. Previous work [11] has shown that polynomial ring multiplication comprises roughly 30% of the execution time of these cryptographic algorithms.

The goal of this work is to use program synthesis tools to optimize polynomial ring multiplication for a variety of computational contexts (Software, FPGAs, ASICs). When developing algorithms using program synthesis, the choice of a Domain Specific Language (DSL) determines the space of possible implementations as well as the reasoning used to search for equivalent programs. We judge that a DSL which is most similar to the algebraic framework used to study polynomial ring structures will be the most effective. This idea is supported by a state of the art implementation of polynomial ring multiplication [17] which was developed using an algebraic framework for describing fast multiplication algorithms [4].

Our choice of DSL varies significantly from DSLs used to synthesize similar algorithms. Namely, in the digital signal processing library, Spiral [9], the authors describe rewrites of the Discrete Fourier Transform (DFT) using a language based on the matrix vector product interpretation of the DFT. The results produced by Spiral justify program synthesis as an appropriate approach, demonstrating that program synthesis methods can feasibly find implementations which outperform human designs within this problem domain. Specifically, polynomial ring multiplication and the Discrete Fourier Transform admit the same unique recursive structure which makes them amenable to program synthesis.

Author’s address: William Scarbro, Sanjay Rajopadhye.

XXXX-XXXX/2023/9-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

By incorporating the recursive structure used by Spiral for DFT synthesis into the algebraic framework describing fast multiplication, we produce a system which can describe a variety of novel fast multiplication operations. Finally, we combine this system with AI search techniques to allow the synthesis of high performance implementations.

Novel programs are described using a set of predefined computational kernels. This high level description is independent of low level optimizations made to the kernels themselves. This approach is amenable to intermediate stages of compilation to increase the efficiency of the resulting implementation, similar to the strategy used by Spiral [9].

1 BACKGROUND

1.1 Definitions

We restrict the definition of a ring to commutative rings with unity. Examples of such rings are $\mathbb{Z}$ and $\mathbb{Z}/n\mathbb{Z}$. A finite field further restricts this class by requiring the multiplication operator to be invertible. An example of a finite field is $\mathbb{F}_P \equiv \mathbb{Z}/p\mathbb{Z}$ where p is prime. Within a finite field, an element which is a primitive Nth root of unity will be called ω_N. The modulus operator (%) is used to map an element of an equivalence class in $\mathbb{Z}/n\mathbb{Z}$ to its canonical representative, e.g. $7\%5 = 2$.

Polynomial rings are formed by extending a ring R by a variable X to form the set $R[X]$. This set consists of elements of the form $a_0 + a_1X + a_2X^2 + \ldots$ with $a_i \in R$. A quotient ring can be constructed by partitioning the elements of the set $R[X]$ into equivalence classes based on their residues modulo a characteristic polynomial Y, producing the set $R[X]/Y$.

A ring isomorphism f, is a structure preserving (homomorphic) map between rings that is also a bijection. Together, these two properties allow ring isomorphisms to rewrite the multiplication operator. Specifically, $a * b$ is equivalent to $f^{-1}(f(a) * f(b))$.

1.1.1 *Notation.*

A type symbol, such as $\mathbb{F}_P$, can be extended by a superscript to describe the structure of an array formed from elements of the type. Examples are $\mathbb{F}_P^n$ and $\mathbb{F}_P^{n\times n}$. When a superscript is attached to a scalar value, it should be interpreted as exponentiation over the relevant multiplication operator. An array structure is indexed to define a scalar value using a list of indices as a subscript, the length of which corresponds to the number of dimensions of the array. Examples are a_j, $\mathcal{F}_n(a)_i$, and $(\mathcal{T}_n)_{i,j}$. The use of superscripts and subscripts in other cases are implicit to a particular definition; e.g. they describe the structure of a transformation's domain, the size of a finite field, or the order of a root of unity.

1.2 Multiplication in RLWE

Many RLWE cryptosystems use rings of the form $\mathbb{F}_P[X]/(X^n + 1)$ where n is a power of two, P is prime, and $2n$ divides $P - 1$. This ring structure is selected to be amenable to fast multiplication. Specifically, the characteristic polynomial $X^n + 1$ has factors of the form $X - \omega_{2n}^{1+2i}$. Factors in the characteristic polynomial of a quotient ring support fast multiplication because these factors determine internal direct products which can be exploited to reduce the number of operations required to perform multiplication.

Our framework generalizes the structure of the characteristic polynomial by allowing polynomial rings of the form $\mathbb{F}_P[X]/(X^n - \omega_N^d)$. In the case of RLWE multiplication, there are many choices of d and N which could represent $(X^n + 1)$ e.g. $(X^n - \omega_{2z}^z)$. However, $2z$ must divide $P - 1$ for ω_{2z} to exist in $\mathbb{F}_P$. To insure the existence of roots of unity, and avoid modification of the variable N, we require the following divisibility constraints on n, d, N and P.

$$n|d$$
$$d|N$$
$$N|P-1$$

The choice of N within the constrained set is inconsequential. This is because all legal choices of N will lead to the same values of the precomputed constants.

1.3 Discrete Fourier Transformation

The Discrete Fourier Transformation (DFT) is a family of algorithms used in a variety of domains including digital signal processing [9], financial models [2], and quantum computing [18].

1.3.1 Number Theoretic Transform. One way to classify Discrete Fourier Transformations describes the algebraic structure of the inputs. The Number Theoretic Transform (NTT) is one such class, and refers to Fourier Transformations performed over vectors of finite fields. A definition of the Number Theoretic Transformation $\mathcal{F}_n$ is given below.

$$\mathcal{F}_n : \mathbb{F}_P^n \longrightarrow \mathbb{F}_P^n \tag{1}$$

$$\mathcal{F}_n(a)_i = \sum_{j=0}^{n-1} a_j \omega_n^{ij} \tag{2}$$

A direct evaluation of the above equation leads to a naive algorithm with computational complexity $O(n^2)$.

1.3.2 Fast Fourier Transform. Another way to classify the Fourier Transformation describes the structure and complexity of the algorithm. The Fast Fourier Transformation (FFT) refers to algorithms with $O(n\log(n))$ complexity. One such algorithm is described by the equation below.

$$\mathcal{F}_n(a)_i = \mathcal{F}_{n/2}(a_{2j})_i + \omega_n^i \mathcal{F}_{n/2}(a_{2j+1})_i \tag{3}$$

The above structure can be further described as the radix-2, decimation-in-time, FFT. Equation (3) is radix-2 because it divides the domain into two subsets ($\{a_{2j}|0 \leq j < n/2\}$ and $\{a_{2j+1}|0 \leq j < n/2\}$). Equation (3) uses decimation-in-frequency because the subsets are selected by striding along the domain using the radix as the step size (the alternative, decimation-in-time, uses contiguous subsets).

1.3.3 DFT as a Linear Operator. One can interpret the Discrete Fourier Transform as a matrix vector product. This interpretation leads to an alternative definition of equation (2) using a transformation matrix $\mathcal{T}$.

$$\mathcal{T}_n \in \mathbb{F}_P^{n\times n} \tag{4}$$

$$(\mathcal{T}_n)_{i,j} = \omega_n^{ij} \tag{5}$$

$$\mathcal{F}_n(a) = \mathcal{T}_n a \tag{6}$$

Using this interpretation, fast Fourier algorithms can be described as sparse decompositions of the transformation matrix. This is the view taken by [9] and will be further explained in section 2.2.

1.3.4 DFT as Polynomial Evaluation. Another interpretation of the Fourier transformation looks at evaluating a polynomial of degree (at most) $n-1$ over the set $\{\omega_n^i | 0 \leq i < n\}$. This interpretation arises from the conversion of a vector of length n into a polynomial of degree (at most) $n-1$ by assigning the jth vector element to the polynomial coefficient corresponding to the X^j term.

$$S = \{\omega_n^i | 0 \leq i < n\} \tag{7}$$

$$a(X) \in \mathbb{F}_P[X] \tag{8}$$

$$\mathcal{F}_n(a(X)) = a(S) \tag{9}$$

Using this interpretation, a fast Fourier algorithm can be described as a chain of ring isomorphisms starting from the input domain $\mathbb{F}_P[X]/(X^N - 1)$ and ending with the codomain $\prod_{i=0}^{n-1} \mathbb{F}_P[X]/(X - \omega_N^i)$. This interpretation was proposed by [10] and was further explored in the context of fast multiplication by [4].

1.4 Fast Multiplication

Fast multiplication algorithms rely on finding a function f which is isomorphic to the relevant ring structure. Using the isomorphic property of f, one may replace multiplication in one ring with multiplication in another as follows.

$$a * b = f^{-1}(f(a) \times f(b)) \tag{10}$$

In equation (10) the multiplication operator $*$ (over the domain of f) has been replaced by the multiplication operator $\times$ (over the codomain of f).

1.4.1 FFT Multiplication. The polynomial evaluation interpretation of the Discrete Fourier Transformation leads to a family of fast multiplication algorithms. Specifically, a fast Fourier algorithm can be used to map the inputs to the codomain $\prod_{i=0}^{n-1} \mathbb{F}_P[X]/(X - \omega_N^i)$, where multiplication can be performed pointwise. Then, the inverse Fourier transformation can be used to recover the product in the input domain as shown in equation (11).

$$a * b = \mathcal{F}_n^{-1}(\mathcal{F}_n(a) * \mathcal{F}_n(b)) \tag{11}$$

This method has complexity $O(n \log(n))$ and is usually attributed to [16].

2 RELATED WORK

2.1 State of the Art Polynomial Ring Multiplication

The utility of ring isomorphisms for developing fast multiplication algorithms in cryptography can be shown by a review of the state of the art. Seiler [17] developed a novel NTT implementation for ring multiplication using the ring isomorphism framework from [4]. Following the inclusion of Seiler's NTT implementation in the Kyber Cryptosystem [6] several other projects have studied Seiler's implementation in both software and hardware. Nguyen et al. [13], compare fast multiplication using the Toom-Cook algorithm to Seiler's NTT method. Seiler's implementation was also used by Huang et al. [11] to develop an FPGA implementation of the Kyber Cryptosystem.

Seiler builds upon previous optimizations of NTT for use in Ring-LWE. For example, when using the Fast Fourier Transformation for multiplication in the ring $\mathbb{F}_P[X]/(X+1)$ special adjustment is necessary to insure that a negative wrapped convolution is performed. This is because the standard definition of the FFT performs multiplication in the ring $\mathbb{F}_P[X]/(X-1)$. One strategy applies scaling factors of ω_{2n}^i to each a_i and b_i input, and then applies scaling factors ω_{2n}^{-i} to each c_i result. This can be made more efficient by including these factors in the precomputed constants of the FFT to produce a Negative Wrapped Fourier Transformation. This strategy was developed by [15] and

[14] using analysis of the equational view of the Fourier Transformation. Seiler instead derives this strategy by viewing the FFT as a series of ring isomorphisms. Our framework includes this optimization by using an initial ring structure of the form $\mathbb{F}_P[X]/(X^n - \omega_{2n}^n)$.

Another optimization made to the fast Fourier transformation when used for fast multiplication removes needless permutations of the codomain. This is possible because the missing permutations will be accounted for in the inverse transformation. Seiler implements this optimization through a manual derivation which leads to a precise permutation of elements in the domain, specifically in bit reversal order. We extend this optimization by allowing a family of permutations in the codomain, derived automatically through our synthesis tool.

Finally, efficient modular reductions based on Montgomery's trick [12] are a common optimization made to NTT algorithms. Algorithms in our framework are specified at the kernel level, and are thus independent of the reduction method used.

2.2 FFT in Program Synthesis

Program synthesis has proved to be a feasible and effective method for deriving fast implementations of the Discrete Fourier Transformation. Spiral [9] is a program synthesis engine that uses rewrite rules to describe a space of possible FFT implementations and a combination of experimentation and AI tools to find and analyze elements within this space. While program synthesis methods are not generally feasible when scaled to large problem sizes, the use of rewrite rules which eventually terminate in base cases makes program synthesis feasible within the FFT problem domain.

The efficacy of program synthesis for the FFT relies on a unique property of the Fourier Transformation. In particular, the Fourier Transformation has two recursive structures, decimation-in-time and decimation-in-frequency, which can be exploited at the same time. These structures are demonstrated by a rewrite rule used by Spiral [9]. This rule can be represented as a decomposition of the Fourier transformation matrix (below), where $n = km$.

$$\mathcal{T}_n = (\mathcal{T}_k \otimes I_m) T_m^n (I_k \otimes \mathcal{T}_m) L_k^n \tag{12}$$

See [9] for a precise definition of the matrices T_m^n and L_k^n. This decomposition corresponds to the following formula, which exhibits both decimation in time and decimation in frequency structure.

$$\mathcal{F}_n(a)_{i+mi'} = \sum_{j=0}^{k-1} \omega_k^{ij} \omega_N^{ij} \sum_{j'=0}^{m-1} a_{j+kj'} \omega_m^{ij'} \tag{13}$$

To derive the decomposition in eqn. (12) one can replace the appropriate expressions in eqn. (13) with the definition of the DFT ($\mathcal{F}$) from eqn. (2) to produce the following doubly-recursive algorithm.

$$\mathcal{F}_n(a)_{i+mi'} = \mathcal{F}_k(\omega_n^{ij} \mathcal{F}_m(a_{j+kj'})_i)_{i'} \tag{14}$$

While many divide and conquer algorithms pay a constant factor penalty when the domain is divided into more than two pieces, this doubly-recursive structure allows the number of divisions to be increased without increasing the asymptotic computational complexity. This leads to many implementations of the FFT with different recursive structures but similar computational complexity. However, these implementations may differ on several important characteristics, such as memory access patterns or arithmetic intensity.

3 APPROACH

3.1 Contributions

(1) **Abstract Synthesis** We raise the level of abstraction from previous FFT synthesis methods by viewing each step of an FFT as a ring isomorphism rather than a linear operator. This produces a DSL which more closely represents the underlying polynomial ring structures.
(2) **Generalize Ring Structure** Raising the level of abstraction facilitates synthesizing multiplication algorithms for a more general class of ring structures. This includes the ring structures used in the cryptosystems Crystals Kyber [6] and New Hope [3].
(3) **Remove Unnecessary Permutations in Codomain** Our analysis identifies the ring isomorphism (*SwapJoinProd*) which produces unnecessary permutations in the Codomain. Removing this isomorphism effectively removes these permutations.
(4) **Automated Search** Abstracting each ring isomorphism as a morphism over types allows automated search of the algorithm design space. Extracting performance features from a compiled algorithm allows AI algorithms to effectively search this space.

3.2 Framework

We first give a framework describing ring isomorphisms which can be used as fundamental operations to automatically synthesize fast multiplication algorithms. An imperative definition of each fundamental operation (and it's inverse) is given, allowing programs expressed as ring isomorphisms to be compiled to imperative languages.

There are several goals motivating the design of this framework. First, the fundamental operations are sufficiently expressive to allow synthesis of programs exhibiting the doubly-recursive structure from section 2.2. Second, these fundamental operations adhere to closure properties, which insure the application of any sequence of fundamental operations will eventually terminate.

3.3 Implementation

We use a simplified representation of the ring isomorphism framework to build a program synthesis engine for deriving fast multiplication algorithms. This is achieved by translating the algebraic structures described in by our framework to types in Haskell. In this representation the fundamental operations of the framework correspond to morphisms over these types. We borrow the idea of a functor from category theory to allow manipulation of these types through a set of morphisms and lifting functions (functions over morphisms).

In the end, this produces a method for describing fast multiplication algorithms as a series of morphisms. Each series represents a fast multiplication implementation which can be compiled to an imperative language.

3.4 Design Space Exploration

A simulated annealing algorithm is used to explore a set of fast multiplication algorithms and find elements which correspond to fast implementations. Algorithms in this space are expanded into a set of neighbors which share similar characteristics using a novel suggestion based search algorithm. This expansion algorithm guides brute force search using a set of precomputed symbolic identities.

4 FRAMEWORK

The framework presented by Bernstein [4] is designed to compare and contrast the structure in various fast multiplication algorithms. To use this framework in program synthesis it is necessary to make some refinements. To this end, we require that the fundamental operations must be computationaly precise to allow translation to imperative languages. We define a series of fundamental ring isomorphisms, provide imperative implementations for each isomorphism (and its inverse), and finally describe how they can be used to synthesize a variety of imperative programs. We follow the convention that a capital letter represents the imperative implementation of the ring isomorphism represented by the corresponding letter.

4.1 Fundamental Ring Isomorphisms

4.1.1 Factor.

$$\begin{aligned} \phi_k : \frac{\mathbb{F}_P[X]}{X^n - \omega_N^d} &\longrightarrow \prod_{z=0}^{k-1} \frac{\mathbb{F}_P[X]}{X^{n/k} - \omega_N^{(d+zN)/k}} \\ \phi_k(X^{n/k})_z &= \omega_k^{(d+zN)/k} \end{aligned} \tag{15}$$

The *Factor* ring isomorphism (ϕ_k) exposes internal direct products in the underlying ring structure. Viewed another way, the *Factor* isomorphism is a generalization of the Discrete Fourier Transformation. Specifically, the DFT decomposes the ideal $(X^n - 1)$ into a set of substitution maps which send $X \rightsquigarrow \omega_N^z$ for $z \in [0, n-1]$. This precisely matches the *Factor* isomorphism when $k = n$ and $d = 0$. The *Factor* isomorphism generalizes the DFT by parameterizing the domain through the variable d, and parameterizing the codomain through the variable k. The application of the *Factor* isomorphism (together with *SwapJoinProd*) corresponds to the decimation-in-frequency version of the FFT.

An imperative definition of the *Factor* ring isomorphism is given below, where $n = mk$.

$$\begin{aligned} \Phi_{n,k,d} &: \mathbb{F}_P^n \longrightarrow \mathbb{F}_P^{k\times m} \\ \Phi_{n,k,d}(a)_{z,j} &= \sum_{i=0}^{k-1} a_{im+j}\,\omega_N^{(d+zN)i/k} \end{aligned} \tag{16}$$

This algorithm is easily derived by inspecting the operations performed by applying the *Factor* isomorphism to a polynomial in $R[X]/(X^n - \omega_N^d)$. The inverse transformation is more difficult to derive. A proof that equation (17) correctly inverts the function defined in equation (16) is given in Appendix A.

$$\begin{aligned} \Phi_{n,k,d}^{-1} &: \mathbb{F}_P^{k\times m} \longrightarrow \mathbb{F}_P^n \\ \Phi_{n,k,d}^{-1}(A)_{im+j} &= \frac{1}{k}\sum_{z=0}^{k-1} A_{z,j}\,\omega_N^{-(d+zN)i/k} \end{aligned} \tag{17}$$

4.1.2 Label.

$$\begin{aligned} \xi_k : \frac{\mathbb{F}_P[X]}{X^{km} - \omega_N^d} &\longrightarrow \left(\frac{\mathbb{F}_P[Y]}{Y^m - \omega_N^d}\right)\frac{[X]}{X^k - Y} \\ \xi_k(X^k) &= Y \end{aligned} \tag{18}$$

The isomorphism *Label* (ξ_k) replaces the expression X^k with the symbol Y. On its own, this transformation only allows reordering of the coefficients, shown below.

$$\begin{aligned} a = a_0x^0 + a_1x^1 &+ a_2x^2 + a_3x^3 + \\ &a_4x^4 + a_5x^5 + a_6x^6 + a_7x^7 \\ &\in \mathbb{F}_P[X]/(X^8 - 1) \\ \xi_2(a) = (a_0y^0 + a_4y^1)x^0 &+ (a_1y^0 + a_5y^1)x + \\ &(a_2y^0 + a_6y^1)x^2 + (a_3y^0 + a_7y^1)x^3 \\ &\in (\mathbb{F}_P[Y]/(Y^2 - \omega_N^d))[X]/(X^4 - Y) \end{aligned}$$

When used in conjunction with *Factor* and *Define*, the *Label* isomorphism allows the synthesis of decimation-in-frequency FFT algorithms, as discussed in [4]. An imperative definition of ξ_k is given below. To insure termination, and to avoid producing identity operations, we require that $m, k > 1$.

$$\begin{aligned} \Xi_{k,m} &: \mathbb{F}_P^{m\times k} \longrightarrow \mathbb{F}_P^{k\times m} \\ \Xi_{k,m}(a)_{i,j} &= a_{j,i} \\ \Xi_{k,m}^{-1}(A)_{j,i} &= A_{i,j} \end{aligned} \tag{19}$$

While this transformation does not involve computation, it represents a derangement of the data in memory. Specifically, the arrays are stored in row major order. This corresponds to the L_m^n permutation in Spiral [9].

4.1.3 Normalize.

$$\begin{aligned} \psi &: \frac{\mathbb{F}_P[X]}{X^n - \omega_N^d} \longrightarrow \left(\frac{\mathbb{F}_P[Y]}{Y^n - 1}\right) \frac{[X]}{X - \omega_N^{d/n}Y} \\ \psi(X) &= \omega_N^{d/n}Y \end{aligned} \tag{20}$$

The *Normalize* (ψ) isomorphism returns a characteristic polynomial to the canonical form $Y^n - 1$. To insure termination, and avoid producing identity operations, we require $d \neq 0$ and $n \neq 1$. *Normalize* is a linear time operation which applies scaling factors to each element of the input, as shown below.

$$\begin{aligned} \Psi_{n,d} &: \mathbb{F}_P^n \longrightarrow \mathbb{F}_P^n \\ \Psi_{n,d}(a)_i &= \omega_N^{id/n}a_i \\ \Psi_{n,d}^{-1}(a)_i &= \omega_N^{-id/n}a_i \end{aligned} \tag{21}$$

This transformation appears in several algorithms. One example are the scaling factors used to allow a standard FFT algorithm to compute a negative wrapped convolution, as discussed in Section 2.1. Another example is the twiddle matrix T_m^n used by Spiral [9], discussed in Section 2.2.

4.1.4 Define.

The two versions of *Define* given below represent two cases for replacing a variable with its definition.

$$\delta^E : \left(\frac{\mathbb{F}_P[Y]}{Y - \omega_N^d}\right) \frac{[X]}{X^m - Y} \longrightarrow \frac{\mathbb{F}_P[X]}{X^m - \omega_N^d} \tag{22}$$

The *DefineExp* (δ^E) isomorphism provides an indirect inverse to the *Label* isomorphism by defining the innermost variable, in this case Y, with an identity produced by *Label*.

$$\delta^R : \left(\frac{\mathbb{F}_P[Y]}{Y - \omega_N^d}\right) \frac{[X]}{X - \omega_N^z Y} \longrightarrow \frac{\mathbb{F}_P[X]}{X - \omega_N^{d+z}} \tag{23}$$

The *DefineRot* (δ^R) isomorphism provides an indirect inverse to the *Normalize* isomorphism by defining the innermost variable, in this case Y, with an identity produced by *Normalize*.

When viewed imperatively, the *Define* isomorphisms are identity operations. This is because the variable Y does not appear in the canonical representation of $\mathbb{F}_P[Y]/(Y - \omega_N^d)$.

4.1.5 Swap.
The four versions of *Swap* given below represent four cases for reversing the order of two surrounding ring structures. Refer to eqns. (35) through (37) for definitions of surrounding ring structures.

$$\zeta^{EP} : \left(\prod_{i=0}^{n_2-1} S[X](i)\right) \frac{[Y]}{Y^{n_1} - X} \longrightarrow \prod_{i=0}^{n_2-1} \left(S[X](i) \frac{[Y]}{Y^{n_1} - X}\right) \tag{24}$$

SwapEP (ζ^{EP}) reverses the order of an *Expansion* surrounding a *Product*. The innermost term, $S[X](i)$, is a polynomial ring structure whose outermost variable extension is X and is parameterized by the variable i.

SwapEP can be implemented as a permutation (Z); shown below. This permutation is similar to the implementation of *Label* (eqn. (19)), except it moves chunks of memory at a time. The length of each chunk, m, corresponds to the length of the data used to represent the innermost term in eqn. (24).

$$\begin{aligned} Z_{n_1,n_2,m} &: \mathbb{F}_P^{n_1 \times n_2 \times m} \longrightarrow \mathbb{F}_P^{n_2 \times n_1 \times m} \\ Z_{n_1,n_2,m}(a)_{i,j,k} &= a_{j,i,k} \\ Z_{n_1,n_2,m}^{-1}(a)_{j,i,k} &= A_{i,j,k} \end{aligned} \tag{25}$$

SwapRP (η^{RP}) reverses the order of a *Rotation* surrounding a *Product*. When viewed imperatively, *SwapRP* is an identity operation.

$$\eta^{RP} : \left(\prod_{i=0}^{n-1} S[X](i)\right) \frac{[Y]}{Y - \omega_N^d X} \longrightarrow \prod_{i=0}^{n-1} \left(S[X](i) \frac{[Y]}{Y - \omega_N^d X}\right) \tag{26}$$

SwapER (θ^{ER}) reverses the order of an *Expansion* surrounding a *Rotation*. The innermost term, $S[W]$ represents a polynomial ring structure whose outermost variable extension is W.

$$\theta^{ER} : \left(S[W]\frac{[Y]}{Y-\omega_N^d W}\right)\frac{[X]}{X^n - Y} \to \left(S[W]\frac{[V]}{V^n - W}\right)\frac{[X]}{X-\omega_N^{d/n}V} \tag{27}$$
$$\theta^{ER}(X) = \omega_N^{d/n}V$$

Similar to *Normalize*, the implementation of *SwapER* only applies scaling factors to elements of the input. The precise method is described by the function Θ.

$$\begin{aligned} \Theta_{n,m,d} &: \mathbb{F}_P^{n\times m} \to \mathbb{F}_P^{n\times m} \\ \Theta_{n,m,d}(a)_{i,j} &= \omega_N^{id/n} a_{i,j} \\ \Theta_{n,m,d}^{-1}(a)_{i,j} &= \omega_N^{-id/n} a_{i,j} \end{aligned} \tag{28}$$

As before, m corresponds to the length of the data used to represent the innermost term. Notice that Θ extends Ψ in the same way that Z extends Ξ.

4.1.6 Join.

The three versions of $Join$ given below represent three cases for joining identical surrounding ring structures.

$$\rho :: \prod_{i=0}^{n_1-1}\left(\prod_{j=0}^{n_2-1} S(i,j)\right) \to \prod_{i=0}^{n_1n_2-1} S(\lfloor i/n_2\rfloor, i\%n_2) \tag{29}$$

$$\mu :: \left(S[Z]\frac{[X]}{X^n - Z}\right)\frac{[Y]}{Y^{n_2}-X} \to S[Z]\frac{[Y]}{Y^{n_1n_2}-Z} \tag{30}$$

$$\sigma :: \left(S[Z]\frac{[X]}{X-\omega_N^{d_1}Z}\right)\frac{[Y]}{Y-\omega_N^{d_2}X} \to S[Z]\frac{[Y]}{Y-\omega_N^{d_1+d_2}Z} \tag{31}$$

JoinProd (ρ), *JoinExp* (μ), and *JoinRot* (σ) can all be used to reduce the number of surrounding expressions in a ring structure. When viewed imperatively each of these functions is an identity operation.

4.1.7 SwapJoin.

$$\tau :: \prod_{i=0}^{n_1-1}\left(\prod_{j=0}^{n_2-1} S(i,j)\right) \to \prod_{i=0}^{n_1n_2-1} S(i\%n_1, \lfloor i/n_1\rfloor) \tag{32}$$

SwapJoinProd (τ) reverses the order of two *Products* and then combines the two. *SwapJoinProd* is necessary to describe many fast Fourier algorithms (See Appendix C), where the order of elements in the codomain ($\prod_{i=0}^{n-1}\mathbb{F}_P[X]/(X-\omega_N^i)$) is important. However, this isomorphism tends to obstruct fast polynomial multiplication, where the order of elements within a *Product* does not matter, and the permutation will have to be undone by the inverse operation. For this reason, we avoid using *SwapJoinProd* when possible.

The imperative implementation of *SwapJoinProd* matches the implementation Z, the imperative definition of *SwapEP*, presented by eqn. (25).

5 IMPLEMENTATION

5.1 Simplified Representation

5.1.1 Type System. The polynomial ring structures described in the previous section can be simplified by recognizing common structures in surrounding expressions. This simplified representation produces a type system which can be used to describe the domain and codomain of each fundamental ring isomorphism. The structure of this system is given below.

$$Ring := Base\ \mathbb{N}\ \mathbb{N} \mid Product\ \mathbb{N}\ (\mathbb{N} \longrightarrow Ring) \mid Expansion\ \mathbb{N}\ Ring \mid Rotation\ \mathbb{N}\ Ring \tag{33}$$

The correspondence of this system with the surrounding ring structures manipulated by the previous section is given by eqns. (34) through (37). The variables P and N are omitted from the simplified representation because they are not modified by any fundamental ring isomorphism.

$$Base\ n\ d := \frac{\mathbb{F}_P[X]}{X^n - \omega_N^d} \tag{34}$$

$$Product\ n\ f := \prod_{i=0}^{n-1} f(i) \tag{35}$$

$$Expansion\ n\ (S[Y]) := (S[Y])\frac{[X]}{X^n - Y} \tag{36}$$

$$Rotation\ d\ (S[Y]) := (S[Y])\frac{[X]}{X - \omega_N^d Y} \tag{37}$$

5.1.2 Morphisms. We can now translate the type signature of each fundamental ring isomorphism into the simplified representation as a morphism. In doing so, we change the level of abstraction so what was once a type signature is now a definition. At this abstraction level, eqns. (38) through (49) represent morphisms with signature $(Ring \longrightarrow Ring)$. In these equations, Church's lambda notation is used to describe anonymous functions.

$$\begin{aligned}
Factor\ k &: \phi_k(Base\ n\ d) = Product\ k\ (\lambda z.\ Base\ (n/k)\ ((d+zN)/k)) && (38)\\
Label\ k &: \xi_k(Base\ n\ d) = Expansion\ k\ (Base\ (n/k)\ d) && (39)\\
Normalize &: \psi(Base\ n\ d) = Rotation\ (d/n)\ (Base\ n\ 0) && (40)\\
DefineExp &: \delta^E(Expansion\ n\ (Base\ 1\ d)) = Base\ n\ d && (41)\\
DefineRot &: \delta^R(Rotation\ d_1\ (Base\ 1\ d_2)) = Base\ n\ (d_1+d_2) && (42)\\
SwapEP &: \zeta^{EP}(Expansion\ n_1\ (Product\ n_2\ f) = Product\ n_2\ (\lambda i.\ Expansion\ n_1\ f(i)) && (43)\\
SwapRP &: \eta^{RP}(Rotation\ d\ (Product\ n\ f)) = Product\ n\ (\lambda i.\ Rotation\ d\ f(i)) && (44)\\
SwapER &: \theta^{ER}(Expansion\ n\ (Rotation\ d\ S)) = Rotation\ (d/n)\ (Expansion\ n\ S) && (45)\\
JoinExp &: \mu(Expansion\ n_1\ (Expansion\ n_2\ S)) = Expansion\ (n_1n_2)\ S && (46)\\
JoinRot &: \sigma(Rotation\ d_1\ (Rotation\ d_2\ S)) = Rotation\ (d_1+d_2)\ S && (47)\\
JoinProd &: \rho(Product\ n_1\ (\lambda i.\ Product\ n_2\ (\lambda j.\ f(i,j))))\\
&= Product\ (n_1n_2)\ (\lambda z.\ f(\lfloor z/n_2\rfloor, z\%n_2) && (48)\\
SwapJoinProd &: \tau(Product\ n_1\ (\lambda i.\ Product\ n_2\ (\lambda j.\ f(i,j)))\\
&= Product\ (n_1n_2)\ (\lambda z.\ f(z\%n_1, \lfloor z/n_1\rfloor) && (49)
\end{aligned}$$

It is advantageous to represent morphisms symbolically as a type *Morphism* defined as follows.

$$\begin{aligned}
Morphism\ &:= Factor\ Int \mid Label\ Int \mid Normalize \mid DefineExp \mid && (50)\\
&DefineRot \mid SwapEP \mid SwapRP \mid SwapER \mid JoinExp \mid && (51)\\
&JoinRot \mid JoinProd \mid SwapJoinProd \mid && (52)
\end{aligned}$$

To unsymbolize a *Morphism* we use an *applyMorphism* function which returns the original *Ring* map.

$$applyMorphism :: Morphism \rightarrow Ring \rightarrow Ring \tag{53}$$

5.1.3 Algorithms in the Simplified Representation. A fast multiplication algorithm can be described as an initial *Ring* structure, a series of *Morphism*, and a final *Ring* structure. This is summarized by the type *Path*.

$$Path := Path\ \{start :: Ring,\ series :: [Morphism],\ end :: Ring\} \tag{54}$$

A *Path* is legal when when the composition of functions formed by the unsymbolized *series* maps *start* to *end*.

5.1.4 Compiling the Simplified Representation. The *Path* representation of a multiplication algorithm can be translated to an imperative equation by first applying a *compile* function to each *Morphism* in *series*. This step simply substitutes the current *Morphism* with the corresponding forward imperative definition found in Section 4.1. However, in the simplified representation, each *Morphism* on its own does not contain enough information to define the imperative definition, so one must also know the *Ring* structure the *Morphism* is applied to. This results in a compile function with the following signature.

$$compile :: Morphism \rightarrow Ring \rightarrow LinearOperator \tag{55}$$

To simplify notation, many of the imperative definitions in section 4.1 are defined using multidimensional vectors (e.g. $\mathbb{F}_p^{k\times m}$). For compilation, we implicitly convert to one dimensional vectors (e.g. $\mathbb{F}_p^{km}$). The type *LinearOperator* represents symbolized functions of type ($\mathbb{F}_p^n \to \mathbb{F}_p^n$).

Second, one must also define the inverse operation of a *Path*. This is achieved by applying a corresponding *compile_inverse* function to each *Morphism* in *series*. The *compile_inverse* function recovers the inverse, imperative definition found in Section 4.1. The *compile_inverse* function has the same signature as *compile*. Notice that in Section 4.1 the inverse, imperative definitions are given using variables defined in the type of the domain of the forward function. As a result, and somewhat counter intuitively, the inverse of a *Morphism* is defined using the *Ring* of its codomain rather than its domain.

Lastly, one must determine how to perform multiplication in the *end Ring* structure of the *Path*. We simplify this step by requiring that all *end Ring* structures may only contain the *Product* surrounding structure. This insures that multiplication may always be performed pointwise using the same implementation.

These three steps determine the functions f and f^{-1} as well as the operator $\times$ from equation (10).

5.2 Functors

Fundamental ring isomorphisms alone are not enough to describe fast Fourier algorithms. This is because the domain of many isomorphisms cannot be matched against their codomain. For example, a single application of the *Factor* isomorphism (ϕ_k) produces a polynomial ring structure of the form $Product(Base)$ which cannot be matched against the domain of *Factor*; simply $Base$. Therefore, using only the ring isomorphisms presented, even the most simple algorithms remain out of reach.

Defining functors for each of $Rotation$, $Expansion$, and $Product$ solves this problem. These functors extend the signature of a function defined over *Ring* by wrapping additional structure around the domain and codomain.

5.2.1 *Rotation.*

The functor for $Rotation$ is named $\mathcal{R}$. It's signature and definition are given below.

$$\mathcal{R} : (Ring \to Ring) \to Ring \to Ring \tag{56}$$

$$\mathcal{R}(f_L,\ Rotation\ d\ S) = Rotation\ d\ f_L(S) \tag{57}$$

Let F_L be the imperative version of the lifted function f_L acquired through the expression $compile(f_L, S)$. The imperative definition of $\mathcal{R}$, which we will call $\hat{\mathcal{R}}$, simply applies the function F_L to the domain.

$$\hat{\mathcal{R}}(F_L) : \mathbb{F}_p^n \to \mathbb{F}_p^n \tag{58}$$

$$\hat{\mathcal{R}}(F_L, a)_i = F_L(a)_i \tag{59}$$

5.2.2 *Expansion.*

The functor for $Expansion$ is named $\mathcal{E}$. It's signature and definition are given below.

$$\mathcal{E} : (Ring \rightarrow Ring) \rightarrow Ring \rightarrow Ring \tag{60}$$

$$\mathcal{E}(f_L,\ Expansion\ n\ S) = Expansion\ n\ f_L(S) \tag{61}$$

Let F_L be the imperative version of the lifted function f_L. The imperative definition of $\mathcal{E}$, which we will call $\hat{\mathcal{E}}$, repeats the function F_L over n subsets of the domain. This corresponds to the construction $I_n \otimes F_L$, when F_L is viewed as a linear operator. The signature of the function F_L is $\mathbb{F}_P^{n'} \rightarrow \mathbb{F}_P^{n'}$ in the definition of $\hat{\mathcal{E}}_n$ below.

$$\hat{\mathcal{E}}_n(F_L) : \mathbb{F}_P^{n\times n'} \rightarrow \mathbb{F}_P^{n\times n'} \tag{62}$$

$$\hat{\mathcal{E}}_n(F_L, a)_{i,j} = F_L(a_i)_j \tag{63}$$

5.2.3 *Product.*

The functor for *Product* is named $\mathcal{P}$, and is described below.

$$\mathcal{P} : (Ring \rightarrow Ring) \rightarrow Ring \rightarrow Ring \tag{64}$$

$$\mathcal{P}(f_L,\ Product\ n\ f_P) = Product\ n\ (f_L \circ f_P) \tag{65}$$

The imperative definition of $\mathcal{P}$ is more subtle. In the definition above the function f_P is composed with the function f_L to form the function in the result ($f_C : \mathbb{N} \rightarrow Ring$). Let F_C be the imperative version of this function with with signature $\mathbb{N} \rightarrow \mathbb{F}_P^{n'} \rightarrow \mathbb{F}_P^{n'}$. F_C is obtained by composing f_P with the imperative version of f_L through the expression $(compile\ f_L) \circ f_P$. The imperative definition of the *Product* functor, $\hat{\mathcal{P}}_n$, is given below.

$$\hat{\mathcal{P}}_n(F_C) : \mathbb{F}_P^{n\times n'} \rightarrow \mathbb{F}_P^{n\times n'} \tag{66}$$

$$\hat{\mathcal{P}}_n(F_C, a)_{i,j} = F_C(i, a_i)_j \tag{67}$$

5.3 Algorithm Synthesis

The framework developed in the previous sections leads to a novel domain of fast multiplication algorithms. This domain is represented by the type *Path* from equation (54). We will first show how to construct a legal *Path* and then how to find similar legal *Path*s.

To find a legal *Path* we use a *findMorphisms* function to identify a set of *Morphism*s which can be applied to a given *Ring* structure. The signature of the *findMorphisms* function is given below.

$$findMorphisms :: Ring \rightarrow [Morphism] \tag{68}$$

A function *findLegalPath* may be constructed from the functions previously described (*applyMorphism* and *findMorphisms*) as well as some selection function of type (*[Morphism] -> Morphism*). The implementation of the function *findLegalPath* is given in Listing 1. The variables *current_ring* and *series* are initialized to *start* and an empty list respectively.

Listing 1. findLegalPath function

```
findLegalPath ::
    ([Morphism] -> Morphism) ->
    Ring ->
    Ring ->
```

```
        [Morphism] ->
        Path
    findLegalPath selection_func start current_ring series =
        let
            possible_morphs = findMorphisms current
            new_morph = selection_func morphs
            new_ring = applyMorphism new_morph current
        in
            if null possible_morphs then
                Path start series current
                else
                findLegalPath
                    selection_func
                    start
                    new_ring
                    (new_morph:series)
```

Before applying an optimization algorithm we need a method of expanding one *Path* into a neighborhood of similar *Paths*. In the following sections (5.3.1, 5.3.2) we present two possible expansion algorithms.

5.3.1 Bounded Search Algorithm. The Bounded Search Algorithm (BSA) expands a single *Path* into a neighborhood of similar *Paths*. Given an initial *Path*, the Bounded Search Algorithm (BSA) first identifies the sequence of *Ring* structures traversed by this *Path*. After selecting two *Ring* structures from this sequence (*sub_start* and *sub_end*), BSA attempts to find a series of *Morphism* which will lead from *sub_start* to *sub_end*. If such a series exists, it can be spliced it into the initial *Path* to form a new *Path* with similar characteristics. Fig. 1 depicts an example.

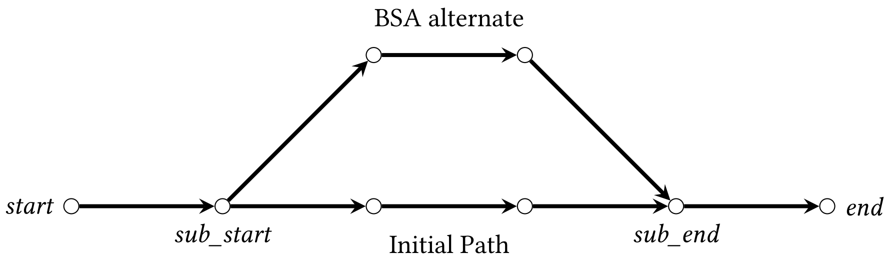

Fig. 1. A new Path found by BSA

BSA is bounded in two ways. First, the distance between *sub_start* and *sub_end* in the initial *Morphism* sequence is bounded. This insures the new *Path* will share characteristics with the initial *Path*. Second, the length of possible new subseries between *sub_start* and *sub_end* is bounded. This limits the depth of the search and the time it will take BSA to expand a *Path*.

The neighborhood of expanded *Paths* found by BSA is a result of the selection of *sub_start* and *sub_end* pairs as well as the possible replacement subseries between any selected pair.

BSA works well on small problem sizes but does not scale well to larger values of n. This is because, as the value of n increases the number of possible *Morphisms* produced by *findMorphisms* increases, which in turn increases the branching factor of the search algorithm.

5.3.2 Suggestion Search Algorithm. The Suggestion Search Algorithm (SSA) is inspired by the rewrite rules used in Spiral. In Spiral, rewrite rules are used to recursively redefine the decomposition of a linear operator and are derived by hand and then supplied to the synthesis engine. In our system these rewrite rules take the form of suggested replacements of *Morphism* series and are derived automatically.

Suggestions are derived from equivalence classes over *Morphism* series. These equivalence classes are discovered by examining a *Ring* with structure (*Base 6 0*). We start by enumerating all reachable *Ring* structures from (*Base 6 0*), and all possible *Morphism* series (of bounded length) between these *Ring* structures. Then, each equivalence class is constructed as a set of *Morphism* series which all lead from the same start *Ring* to the same end *Ring*.

Notice this is not an exact notion of equivalence. We would like to use equivalence to replace one *Morphism* series with another from the same equivalence class. To do this blindly, we would need know that for any two *Morphism* series from the same equivalence class, given any starting *Ring* structure, both *Morphism* series would lead to the same end *Ring* structure. This is not the notion of equivalence produced by the above algorithm, which only assures that two "equivalent" *Morphism* series will lead to the same end *Ring* when given a particular start *Ring* structure. Perhaps one could formulate conditions on the initial *Ring* which ensure strict equivalence, but we take a different approach. Rather than producing identities, which are strictly equivalent, we use our relaxed notion of equivalence to produce suggestions, which must be verified.

(give example)

Suggestions are discovered from the (*Base* 6 0) structure. As a result they are initially limited to trivial sizes. We can extend the equivalence classes discovered from (*Base* 6 0) to larger sizes by representing *Morphisms* in these equivalence classes symbolically. We do this by replacing integer factors with symbolic variables. The size 6 is chosen so we can differentiate between the two factors: 3 and 2. This method produces 27 equivalence classes, some of which correspond to the rewrite rules used by Spiral. For example, the *Morphism* series used in Appendix C - eqn. (85) - and its equivalence with (*Factor mk*), can be discovered by this method.

SSA follows the same design as BSA but uses suggestions to limit the branching factor of the search step. First, SSA considers a slice of *Morphisms* taken from a *Path*. SSA then matches the slice against symbolic representations of *Morphism* series in the library of equivalence classes. This match is defined by a map between symbolic variables and integers. The other symbolic *Morphism* series in a matched equivalence class represent suggested replacements of the initial slice. Importantly, some of the suggested replacements may contain variables which have not been mapped because they do not appear in the original matched symbolic representation. SSA uses the search step of BSA to discover values for these missing variables. Viewed another way, the symbolic match contexts defined by the matched equivalence class are used to filter candidates from the result of the *findMorphisms* function which are more likely to lead to the correct end *Ring* structure. SSA terminates a search sequence when the correct *sub_end* structure is reached or there are no *Morphisms* discovered which correspond to a symbolic match context.

5.3.3 High Level Optimization. Before

6 EVALUATION

6.1 Algorithm Compilation

Several compilation pathways were used to generate C code from the algorithm description given by a *Path*. The *compile* function produces a symbolic representation of linear operators from *Morphisms*. These symbolic linear operators correspond to the capital Greek letters used in Section 4.1 and are used as a common intermediate representation for both compilation pathways.

The more direct compilation pathway groups symbolic linear operators into classes based on their sparse structure. These classes are square, tensor-with-identity, diagonal, and permutation. Predefined function kernels implement each one of these classes which are adapted to a particular linear operator using parameters and precomputed constants. We call this pathway *Direct*.

Another compilation pathway unrolls all loop structures. This facilitates the application of pinhole optimizations such as removing multiplication by one, and introducing subtraction. There are two versions of this pathway, one which uses temporary variables, another which stores data in memory. These two versions are called *UnrolledTemp* and *UnrolledInMem* respectively.

All compilation pathways use an implementation of the Montgomery fast reduction algorithm [**?**] to perform operations in finite fields. This method is much less sophisticated than those found in [**?**] and [**?**].

The goal of the compilation pathways used is not to produce state of the art implementations of fast multiplication, but rather to evaluate the efficacy of the proposed program synthesis framework. For this reason, we value generality and diversity of generated code over platform specific optimizations. Specifically, the three compilation techniques used isolate the effect of

6.2 Algorithm Optimization

The overarching goal of the methods proposed is to expose the process of designing fast multiplication algorithms to automated search through autotuning [**?**]. As a result we have focused our efforts on the development of a program synthesis framework and use a relatively simple optimization algorithm, simulated annealing [**?**], to search for optimal implementations. Applying more sophisticated optimization algorithms is left to future work.

The implementation of simulated annealing is relatively straightforward. We use program execution time as a cost measure. Initially this method performed well on programs of smaller size but did not scale to larger problems. This was mitigated by using solutions on smaller sizes to produce initial states for larger sizes. This dynamic programming strategy is similar to the one used by **??**.

6.3 Experimental Results

Experimental results measuring the performance of synthesized algorithms were collected on an Intel® Xeon® Processor E3-1230 v2 machine. Binary code was produced using gcc version *8.5.0* using the *-O3* optimization flag.

6.4 Baseline

For a point of comparison, baseline algorithms are produced containing only the Factor isomorphism. This baseline represent algorithms which might feasibly be derived and implemented by hand. The results of an initial study of baseline algorithms is given by figures **??**.

For baseline algorithms compiled using *UnrolledTemp* and *UnrolledInMem* the optimal Factor size is two (Figures 1,2). For baseline algorithms compiled using *Direct* the optimal Factor size is 4. This discrepancy is likely a result of a trade off between the number of field operations, which increases with Factor size, and the overhead incurred by the control structure of additional function calls, which decreases with Factor size. In the entirely unrolled implementations, there is no overhead in control structure so the smallest Factor size is the most successful. In the looped implementation, a Factor size of 4 appears to strike the optimal balance between the number field operations and the complexity of the control structure.

6.5 Optimization Performance

The success of the optimization algorithm varies greatly depending on compilation method and kernel size. One of the more successful examples of the simulated annealing algorithm converging on a local optimum is given by figure **??**.

* example simulated annealing convergence (speedup relative to baseline)

The execution time of the simulated annealing algorithm on different problem sizes is in given in figure **??**.

* table of execution times on different sizes

6.6 Baseline Comparison

* for each compilation method * graph size vs execution time (maybe use 5nlogn, or whatever the standard is) * baseline and optimized result

6.7 Future Work

Extensions of the current work can be applied to the ring isomorphism framework, the generation of imperative code, and the optimization algorithm used.

The Ring isomorphism framework can be generalized to include other fast Fourier algorithms. These include the split radix FFT [8], multidimensional FFTs [], and Bluestein's algorithm [5]. In addition, the equivalence classes over *Morphism* series produced as part of the suggestion search algorithm deserve better study. It is likely these identities exhibit useful group or categorical structure which may aid in their discovery and application. Finally, the imperative definitions given by 4.1 were developed by hand through a combination of derivation and observation. It would be extremely useful develop a method to discover these implementations automatically given only the signature of the ring isomorphism they implement.

The code generation processes in this work are severely limited. They merely aim to evaluate the program synthesis method presented. It is likely that more sophisticated code generation techniques could produce state of the art implementations for fast multiplication and fast Fourier transformations. A more sophisticated code generation process might include; a representation of parallelism, generalization to any scalar structure, and hardware specific instructions.

Finally, a better search algorithm would likely find more optimal implementations more quickly. A study of related work in autotuning [7] suggests the use of machine learning models combined with learned cost measures might be successful for this application.

REFERENCES

[1] O. Regev. On Lattices, Learning With Errors, Random Linear Codes, and Cryptography. In Symposium on Theory of Computing, STOC, pages 84–93, 2005.

[2] *Fourier Transform Methods in Finance.* John Wiley Sons, Ltd, 2012.

[3] Shi Bai, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. NewHope. pages 1–32, 2020.

[4] Daniel J Bernstein. MULTIDIGIT MULTIPLICATION FOR MATHEMATICIANS. 2001.

[5] L. Bluestein. A linear filtering approach to the computation of discrete fourier transform. *IEEE Transactions on Audio and Electroacoustics*, 18(4):451–455, 1970.

[6] Joppe Bos, Leo Ducas, Eike Kiltz, T. Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehle. CRYSTALS - Kyber: A CCA-Secure Module-Lattice-Based KEM. In *Proceedings - 3rd IEEE European Symposium on Security and Privacy, EURO S and P 2018*, pages 353–367. Institute of Electrical and Electronics Engineers Inc., jul 2018.

[7] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Tvm: An automated end-to-end optimizing compiler for deep learning, 2018.

[8] Pierre Duhamel and Henk D. L. Hollmann. 'split radix' fft algorithm. *Electronics Letters*, 20:14–16, 1984.

[9] F. Franchetti et al. Spiral: Extreme performance portability.
[10] Charles M Fiduccia. Polynomial evaluation via the division algorithm the fast fourier transform revisited. 1972.
[11] Yiming Huang, Miaoqing Huang, Zhongkui Lei, and Jiaxuan Wu. A pure hardware implementation of CRYSTALS-KYBER PQC algorithm through resource reuse. *IEICE Electronics Express*, 17(17), aug 2020.
[12] Peter L. Montgomery. Modular Multiplication Without Trial Division. *Mathematics of Computation*, 44(170):519, 1985.
[13] Duc Tri Nguyen and Kris Gaj. Fast neon-based multiplication for lattice-based nist post-quantum cryptography finalists. In Jung Hee Cheon and Jean-Pierre Tillich, editors, *Post-Quantum Cryptography*, pages 234–254, Cham, 2021. Springer International Publishing.
[14] Thomas Pöppelmann, Tobias Oder, and Tim Güneysu. High-performance ideal lattice-based cryptography on 8-bit atxmega microcontrollers. In Kristin Lauter and Francisco Rodríguez-Henríquez, editors, *Progress in Cryptology – LATINCRYPT 2015*, pages 346–365, Cham, 2015. Springer International Publishing.
[15] Sujoy Sinha Roy, Frederik Vercauteren, Nele Mentens, Donald Donglong Chen, and Ingrid Verbauwhede. Compact Ring-LWE Cryptoprocessor. pages 1–18, 2014.
[16] A. Schonhage and V. Strassen. Schnelle multiplikation großer zahlen. *Computing 7*, pages 281–292, 1971.
[17] Gregor Seiler. Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography. 2018.
[18] P.W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. *Proceedings 35th Annual Symposium on Foundations of Computer Science.*, 1994.

7 APPENDIX

7.1 Appendix A: Proof of Inverse Factor Definition

The following is a proof that the implementation of $\Phi^{-1}_{n,k,d}$ given by eqn. (17) correctly computes the inverse of $\Phi_{n,k,d}$. We do so by showing $\Phi^{-1}_{n,k,d}(A)_{im+j}$ recovers a_{im+j} where A is attained through the forward application of $\Phi_{n,k,d}$ on a.

$$Let\ A_{z,j} = \Phi_{n,k,d}(a)_{z,j} = \sum_{i=0}^{k-1} a_{im+j}\omega_N^{(d+zN)i/k} \tag{69}$$

$$\Phi^{-1}_{n,k,d}(A)_{im+j} = \frac{1}{k}\sum_{z=0}^{k-1} A_{z,j}\omega_N^{-(d/k+zN/k)i} \tag{70}$$

$$\Phi^{-1}_{n,k,d}(A)_{im+j} = \frac{1}{k}\sum_{z=0}^{k-1}\sum_{l=0}^{k-1} a_{lm+j}\omega_N^{(d/k+zN/k)l}\omega_N^{-(d/k+zN/k)i} \tag{71}$$

$$\Phi^{-1}_{n,k,d}(A)_{im+j} = \frac{1}{k}\sum_{l=0}^{k-1}\sum_{z=0}^{k-1} a_{lm+j}\omega_N^{(d/k+zN/k)l-(d/k+zN/k)i} \tag{72}$$

$$\Phi^{-1}_{n,k,d}(A)_{im+j} = \frac{1}{k}\sum_{l=0}^{k-1}\sum_{z=0}^{k-1} a_{lm+j}\omega_N^{(d/k)(l-i)}\omega_N^{(zN/k)(l-i)} \tag{73}$$

$$\Phi^{-1}_{n,k,d}(A)_{im+j} = \frac{1}{k}\sum_{l=0}^{k-1}\left(a_{lm+j}\omega_N^{(d/k)(l-i)}\sum_{z=0}^{k-1}\omega_N^{(zN/k)(l-i)}\right) \tag{74}$$

Consider the following sub-expression from (74) which we will label as the function $\gamma(l,i)$

$$\gamma(l,i) = \sum_{z=0}^{k-1}\omega_N^{(zN/k)(l-i)}$$

Using the identity $\omega_N^{N/k} = \omega_k$ we can transform this sub-expression to be

$$\gamma(l,i) = \sum_{z=0}^{k-1}\omega_k^{z(l-i)}$$

Furthermore, consider the case when $l = i$, under this condition this sub-expression becomes

$$\gamma(l, i) = \sum_{z=0}^{k-1} \omega_k^0 = k$$

In the other case, when $l \neq i$, we can use the following identity (proven in 7.1.1).

$$(\forall \alpha \neq 0 \mod k) : \sum_{z=0}^{k-1} \omega_k^{\alpha z} = 0 \tag{75}$$

by setting $\alpha = (l - i)$ with the assurance that:

- $l - i \neq 0$
- $|l - i| < k$ because $i, l \in [0, k-1]$

$\Rrightarrow \alpha \neq 0 \mod k$

Therefore, when we make the assumption that $l \neq i$ and apply identity (75) we find that $\gamma(l, i) = 0$. Using the two cases considered ($l = i$ and $l \neq i$) we can rewrite $\gamma(l, i)$ as

$$\gamma(l, i) = k\delta_{i,l}$$

(Where δ is the Kronecker Delta function)
Replacing the rewrite of $\gamma(l, i)$ in (74) produces the following

$$\Phi_{n,k,d}^{-1}(A)_{im+j} = \frac{1}{k} \sum_{l=0}^{k-1} a_{lm+j} \omega_N^{(d/k)(l-i)} k\delta_{i,l} \tag{76}$$

Inspecting this summation, we find that when $l \neq i$, the $\delta_{i,l}$ term causes the entire summand to go to zero. Therefore, we only need to consider the case when $l = i$.

$$\Phi_{n,k,d}^{-1}(A)_{im+j} = \frac{1}{k} a_{im+j} \omega_N^{(d/k)(i-i)} k$$
$$\Phi_{n,k,d}^{-1}(A)_{im+j} = a_{im+j}$$

□

7.1.1 Proof of (75). This is a well known property of roots of unity but is reproven here for completeness.

$$\begin{aligned}
S &= \sum_{z=0}^{k-1} \omega_k^{\alpha z} \\
\omega_k^{\alpha} S &= \sum_{z=0}^{k-1} \omega_k^{\alpha(z+1)} \\
\omega_k^{\alpha} S &= \omega_k^{\alpha k} + \sum_{z=1}^{k-1} \omega_k^{\alpha z} \\
\omega_k^{\alpha} S &= \omega_k^{0} + \sum_{z=1}^{k-1} \omega_k^{\alpha z} \\
\omega_k^{\alpha} S &= \sum_{z=0}^{k-1} \omega_k^{\alpha z} \\
\omega_k^{\alpha} S &= S \\
S(\omega_k^{\alpha} - 1) &= 0
\end{aligned}$$

Assume $\alpha \neq 0 \mod k$. Then $\omega_k^{\alpha} - 1 \neq 0$. Under this condition $S = 0$. In conclusion

$$(\forall \alpha \neq 0 \mod k) : \sum_{z=0}^{k-1} \omega_k^{\alpha z} = 0 \tag{77}$$

□

7.2 Appendix B: Proof of θ^{ER}

The following is a rough proof that the function θ^{ER} in eqn. (27) is an isomorphism. The ring structure describing the domain of θ^{ER} implies the following identities.

$$Y = \omega_N^d Z \tag{78}$$

$$X^n = Y \tag{79}$$

We can combine this system into a single identity which describes X using Z.

$$X^n = \omega_N^d Z \tag{80}$$

We would like to define a variable V with the property $V^n = Z$. We can achieve this by rearranging the previous equation into the following form.

$$(\omega_N^{-d/n} X)^n = Z \tag{81}$$

By setting $V = \omega_N^{-d/n} X$, we have the following identities.

$$V^n = Z \tag{82}$$

$$X = \omega_N^{d/n} V \tag{83}$$

Eqns. (82) and (83) are then used to construct the codomain of θ^{ER} in eqn. (27).

7.3 Appendix C: Interpreting the Doubly Recursive FFT as a series of Ring Isomorphisms

The doubly recursive algorithms used by Spiral [9] to synthesize fast Fourier transformations can be interpreted as a series of ring isomorphisms. An imperative definition of one of these algorithms is given in eqn. (13), preceded by a four step array decomposition representing the same algorithm in eqn. (12). For comparison with ring isomorphisms, we use the six step array decomposition, given below.

$$\mathcal{T}_n = L_k^n(I_m \otimes \mathcal{T}_k)T_k^n L_m^n(I_k \otimes \mathcal{T}_m)L_k^n \tag{84}$$

We choose to read the elements of this decomposition from right to left. We start with the ring structure ($Base\ n\ 0$) where $n = mk$ and $N = n$. At each step; a linear operator element is given, then a morphism which produces this linear operator when compiled, and finally the ring structure resulting from applying the current morphism to the previous ring structure. In some cases it is necessary to insert identity steps into the linear operator chain to manipulate intermediate ring structures. The final result, $Prod\ n\ (\lambda i.\ Base\ i)$, is the same structure produced by a traditional Fourier transformation ($Factor\ n$).

LO Element	*Morphism*	*Ring Structure*
–	–	$Base\ n\ 0$
L_k^n	$Label\ k$	$Exp\ k\ (Base\ m\ 0)$
$I_k \otimes \mathcal{T}_m$	$\mathcal{E}\ (Factor\ m)$	$Exp\ k\ (Prod\ m\ (\lambda i.Base\ 1\ ki))$
L_m^n	$SwapEP$	$Prod\ m\ (\lambda i.Exp\ k\ (Base\ 1\ ki))$
(I_n)	$\mathcal{P}\ DefineExp$	$Prod\ m\ (\lambda i.Base\ k\ ki)$
T_k^n	$\mathcal{P}\ Norm$	$Prod\ m\ (\lambda i.Rot\ i\ (Base\ k\ 0))$
$I_m \otimes \mathcal{T}_k$	$\mathcal{P}\ (\mathcal{R}\ (Factor\ k))$	$Prod\ m\ (\lambda i.Rot\ i\ (Prod\ k\ (\lambda j.Base\ 1\ mj)))$
(I_n)	$\mathcal{P}\ SwapRP$	$Prod\ m\ (\lambda i.Prod\ k\ (\lambda j.Rot\ i\ (Base\ 1\ mj)))$
(I_n)	$\mathcal{P}\ (\mathcal{P}\ DefineRot)$	$Prod\ m\ (\lambda i.Prod\ k\ (\lambda j.Base\ 1\ (mj + i)))$
L_k^n	$SwapJoinProd$	$Prod\ n\ (\lambda i.Base\ 1\ i)$

(85)

7.4 Appendix D: Proof that a Composition of Fundamental Ring Isomorphisms must be finite

To show that a composition of Fundamental Ring Isomorphisms (FRIs, given in Section 4.1) must be finite, it suffices to show that the number of occurrences of all isomorphisms in a composition sequence is finite. This is accomplished with the help of a metric function $\mathcal{M}$, defined over the space of possible ring structures (*Ring*) found in the simplified representation (Section 5.1).

$$\mathcal{M} : Ring \longrightarrow (\mathbb{N} \times \mathbb{N}) \tag{86}$$

$$\mathcal{M}(S) = (\mathcal{M}_P(S), \mathcal{M}_E(S)) \tag{87}$$

The functions $\mathcal{M}_P$ and $\mathcal{M}_E$ are defined as follows. The metric function $\mathcal{M}_P$ represents the product over the dimensions of all *Products* in a ring structure S. The metric function $\mathcal{M}_E$ represents the product over the dimensions of all *Expansions* in the ring structure S. Examples of $\mathcal{M}_P$ and $\mathcal{M}_S$ are given below.

$$\mathcal{M}_P(Prod\ m\ (\lambda i.Prod\ k\ (\lambda j.Exp\ t\ (Base\ 1\ mj)))) = m * k \tag{88}$$

$$\mathcal{M}_E(Prod\ m\ (\lambda i.Prod\ k\ (\lambda j.Exp\ t\ (Base\ 1\ mj)))) = t \tag{89}$$

Only three of the fundamental ring isomorphisms have an effect on the metric space of $\mathcal{M}$. These isomorphisms are ϕ_k, ξ_k, and δ^E; their effects are described below.

$$\mathcal{M}(\phi_k(S)) = \mathcal{M}(S) * (k, 1) \tag{90}$$
$$\mathcal{M}(\xi_k(S)) = \mathcal{M}(S) * (1, k) \tag{91}$$
$$\mathcal{M}(\delta^E(Exp\ k\ S)) = \mathcal{M}(Exp\ k\ S) * (1, 1/k) \tag{92}$$

Define the total dimension of a ring structure S to be the product of the dimension of the inner *Base* term together with $\mathcal{M}_P(S)$ and $\mathcal{M}_E(S)$. The total dimension is a universal invariant of all ring structures in an FRI composition sequence.

Consider a ring structure S_0 with total dimension $D \in \mathbb{N}$. Define the domain G_{S_0} to be the set of ring structures reachable from S_0 through a composition sequence of FRIs. The image $\mathcal{M}(G_{S_0})$ is bounded by the following inequalities.

$$(i, j) \in \mathcal{M}(G_{S_0})$$
$$i * j \leq D \tag{93}$$
$$\mathcal{M}_P(S_0) \leq i \leq D \tag{94}$$
$$1 \leq j < D/\mathcal{M}_P(S_0) \tag{95}$$

Equation (93) comes from the fact that the total dimension is constant. Equation (94) may be inferred from the following; there is no fundamental ring isomorphism which decreases $\mathcal{M}_P$, equation (93), and a naive lower bound on $\mathcal{M}_E(G_{S_0})$. Finally, equation (95) follows from eqns. (93) and (94), as well as the same naive lower bound on $\mathcal{M}_E(G_{S_0})$.

Consider a composition sequence of fundamental ring operations C, which leads from S_0 to S^*. By definition, $\mathcal{M}(S^*) \in \mathcal{M}(G_{S^0})$. The notation $(f)^C$ represents the set of occurrences of the FRI f in the composition C.

We can conclude the value $|(\phi_k)^C|$ is bounded using the following reasoning. There is no fundamental ring isomorphism which decreases $\mathcal{M}_P$ and the value of $\mathcal{M}_P(S^*)$ is bounded, therefore the value of $\mathcal{M}_P$ is increased a finite number of times in the composition C. Every occurrence of the function ϕ_k in C increases the value of $\mathcal{M}_P$, therefore $|(\phi_k)^C|$ is finite.

We can bound $|(\delta^E)^C|$ by considering the domain $\mathbf{B}^1$: the set of ring structures with metric points in the domain $\{(i, j) \in \mathcal{M}(G_{S_0}) \mid i * j = D\}$. All ring structures in $\mathbf{B}^1$ contain a *Base* of dimension of 1, e.g. *Base* $1\ d$. Notice that ϕ_k is the only FRI which can map ring structures from $\neg\mathbf{B}^1$ to elements in $\mathbf{B}^1$. In addition, δ^E is the only FRI which can map elements in $\mathbf{B}^1$ to elements in $\neg\mathbf{B}^1$ and this is a universal property of δ^E. Therefore, if both S_0 and S^* are in $\neg\mathbf{B}^1$, then for each occurrence of δ^E in C there is a unique, corresponding occurrence of ϕ_k. Using this correspondence we can conclude $|(\delta^E)^C|$ is bounded. Figure 2 displays an example sequence adhering to the conditions on S_0 and S^*. The cases when S_0 and S^* are allowed to be in $\mathbf{B}^1$ lead to the same result.

We can use the bound on $|(\delta^E)^C|$ to bound $|(\xi_k)^C|$ in the following way. With a bound on $|(\delta^E)^C|$, there are a limited number of steps in C which decrease the value of $\mathcal{M}_E$, and the value of $\mathcal{M}_E(S^*)$ is bounded, therefore the value of $\mathcal{M}_E$ is increased only a finite number of times in C. Every occurrence of ξ_k increases the value of $\mathcal{M}_E$, therefore $|(\xi_k)^C|$ is finite.

The value of $|(\psi)^C|$ may be bounded by considering the domain $\mathbf{d}^0$, the set of ring structures with a *Base* in normal form: (*Base* n 0). Notice that ψ is the only FRI that maps elements from $\neg\mathbf{d}^0$ to elements in $\mathbf{d}^0$, and this is a universal property of ψ. In addition, ψ can only be applied to elements in $\neg\mathbf{B}^1$. For this reason, consider the set of longest possible subsequences of C for which the mapped ring structures are in $\neg\mathbf{B}^1$, call this set $C_{\neg B^1}$. There are finitely many such subsequences,

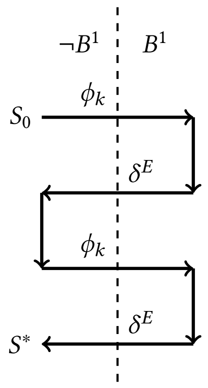

Fig. 2. An example composition sequence demonstrating the bound on $|(\delta^E)^C|$.

because $|(\delta_E)^C|$ has been bounded. Let $c_{\neg B^1}$ be an element of $C_{\neg B^1}$. With the condition that the mapped ring structures are in $\neg\mathbf{B}^1$, ϕ_k is the only FRI which can map elements in $\neg\mathbf{d}^0$ to elements in $\mathbf{d}^0$ (without this condition, we would need to consider δ^R, refer to Figure 3). Therefore, if both the start and end ring structures of $c_{\neg B^1}$ are in $\neg\mathbf{d}^0$, then for every occurrence of ψ in $c_{\neg B^1}$ there is a unique, corresponding occurrence of ϕ_k in $c_{\neg B^1}$. Notice, $|(\phi_k)^{c_{\neg B^1}}|$ is bounded. Together these two properties bound $|(\psi)^{c_{\neg B^1}}|$. The sequence $c_{\neg B^1}$ represents an arbitrary element of $C_{\neg B^1}$ and the set $C_{\neg B^1}$ contains only finitely many elements, therefore the bound on $|(\psi)^{c_{\neg B^1}}|$ implies a bound on $\sum_{c_{\neg B^1}}^{C_{\neg B^1}} |(\psi)^{c_{\neg B^1}}|$. The morphism ψ may only be applied to elements of $\neg B^1$, therefore the bound on $\sum_{c_{\neg B^1}}^{C_{\neg B^1}} |(\psi)^{c_{\neg B^1}}|$ implies a bound on $|(\psi)^C|$. The cases when the start and end ring structures of $c_{\neg B^1}$ are allowed to be in $\mathbf{d}^0$ lead to the same result.

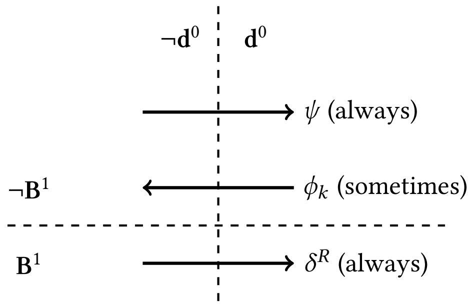

Fig. 3. A summary of the functions ψ, ϕ_k and δ^R with respect to the domains $\mathbf{d}^0$ and $\mathbf{B}^1$.

The value $|(\delta^R)^C|$ is trivially bounded by $|(\psi)^C|$. Each occurrence of ψ adds a *Rotation* surrounding ring structure, and ψ is the only FRI to do so. The function δ^R must remove a *Rotation* surrounding ring structure, so $|(\delta^R)^C| \leq |(\psi)^C|$.

The *Join* functions, ρ, τ, μ, and σ, all combine two structures of the same type. This allows us to bound the occurrences of these isomorphisms by considering the FRI which produces the corresponding structure type.

$$|(\rho)^C| + |(\tau)^C| < |(\phi_k)^C| \tag{96}$$

$$|(\mu)^C| < |(\xi_k)^C| \tag{97}$$

$$|(\sigma)^C| < |(\psi)^C| \tag{98}$$

The *Swap* functions, ζ^{EP}, η^{RP}, and θ^{ER}, can be viewed as a strict total order of *Product*, *Rotation*, and *Expansion* (see Fig. 4). This insures the same two structures cannot be swapped more than once. Knowing this, we can bound the occurrences of these isomorphisms by considering the FRIs which produce the corresponding structure types. These bounds are given by eqns. (99) through (101).

$$\begin{tikzcd}
 & Prod \arrow[dl, "\eta^{RP}"'] \arrow[dr, "\zeta^{EP}"] & \\
Rot \arrow[rr, "\theta^{ER}"'] & & Exp
\end{tikzcd}$$

Fig. 4. The strict total order implied by the swap functions ζ^{EP}, η^{RP}, and θ^{ER}

$$|(\zeta^{EP})^C| \leq \min(\ |(\xi_k)^C|,\ |(\phi_k)^C|\ ) \tag{99}$$

$$|(\eta^{RP})^C| \leq \min(\ |(\psi)^C|,\ |(\phi_k)^C|\ ) \tag{100}$$

$$|(\theta^{ER})^C| \leq \min(\ |(\xi_k)^C|,\ |(\psi)^C|\ ) \tag{101}$$

After bounding the number of occurrences of each FRI type in an unconstrained composition sequence of FRIs, we conclude that any composition sequence of FRIs must be finite.
□