FFT Program Generation for Ring LWE-based Cryptography

Masahiro Masuda ✉ and Yukiyoshi Kameyama

University of Tsukuba, Tsukuba, Japan
masa@logic.cs.tsukuba.ac.jp, kameyama@acm.org

Abstract. Fast Fourier Transform (FFT) enables an efficient implementation of polynomial multiplication, which is at the core of any cryptographic constructions based on the hardness of the Ring learning with errors (RLWE) problem. Existing implementations of FFT for RLWE-based cryptography rely on hand-written assembly code for performance, making it difficult to understand, maintain, and extend for new architectures.
We present a novel framework to implement FFT for RLWE-based cryptography, based on a principled program-generation approach. We start with a high-level, abstract definition of an FFT program, and generate low-level code by interpreting high-level primitives and delegating low-level details to an architecture-specific module. Since low-level details concerning modular arithmetic and vectorization are separated from high-level logic, we can easily generate both AVX2- and AVX512-optimized low-level code from the same high-level description of the FFT program. Our generated code is highly competitive compared to expert-written assembly code: For AVX2 (and AVX512, resp) it runs 1.13x (and 1.39x, resp) faster than the AVX2-optimized assembly implementation in the NewHope key-exchange protocol.

1 Introduction

Lattice cryptography has been receiving increasing attention due to its widely believed resistance against quantum attacks while still allowing efficient implementations of important cryptographic protocols. A construction based on the hardness of Ring learning with errors (RLWE) problem [15] is particularly efficient, thanks to its algebraic structure. At the heart of all RLWE-based protocols is the multiplication of polynomials, whose coefficients are taken from integers modulo a certain prime. It is well known that the polynomial multiplication can be computed in $O(n \log n)$ time via Fast Fourier transform (FFT)[1][6]. Since the computational cost for polynomial multiplication is dominated by FFT, there have been many work on optimized FFT implementations for RLWE-based cryptography[7,2,14,19].

However, we believe that existing implementations have some shortcomings, in terms of ease of understanding, maintainability, and reusability:

[1] FFT in which coefficients are taken from a finite field is often called NTT (Number Theoretic Transform), but we use the term FFT throughout this paper.

– They support either only one set of security parameters, or multiple sets of parameters by duplicating code. Duplication makes implementation and maintenance of code tedious and error prone.
– Precomputed constants are hardcoded in the source code. In the context of RLWE, the size and modulus parameters are always fixed, making it possible to precompute all the twiddle factors in an FFT implementation. In addition, it is common to pre-multiply the twiddle factors by other factors arising from Montgomery multiplication or negative wrapped convolution [2,14]. Since each implementation does precomputation in slightly different ways and often comes without explanation of how those constants are computed, it is difficult to precisely understand what each constant represents.
– Most likely, an optimized implementation is written in assembly. This makes it extremely difficult to understand the code and be confident in its correctness. Moreover, since an assembly program is hardcoded using a particular SIMD instruction set (e.g. AVX2), porting the implementation to new architectures requires writing another assembly program from scratch. We are not aware of any FFT implementation that leverages AVX512 instructions in the context of RLWE-based cryptography.

We address these problems by a principled program-generation approach. By adopting program-generation techniques developed in programming language research, it is possible to give an abstract description of an FFT algorithm, from which we can generate highly optimized low-level code for various instruction sets, including AVX2 and AVX512. Moreover, the program-generation approach makes it easy to combine optimization techniques developed in different studies, which allows us to generate more efficient code than the assembly program in NewHope[2], one of the recent FFT implementations.

Our framework is written in a functional programming language OCaml [12]. A user would write an FFT program in a specialized domain-specific language (DSL) embedded in OCaml. The DSL program can be evaluated in multiple ways. For example, we can generate an equivalent C program, including those optimized with SIMD intrinsics. Since our vectorized DSL programs are written in a way that is generic with respect to the vector length and not tied to a particular SIMD instruction set, we can easily generate both AVX2 and AVX512 code from the same high-level DSL program. Supporting a new ISA, such as ARM Neon, should be straightforward: We only need to provide the mapping between vectorized primitives in our DSL, such as `add`, `mullo` and `mulhi`, and corresponding SIMD instructions in the target ISA.

We demonstrate our framework by implementing vectorized FFT code for RLWE-based cryptography. We start with the reference implementation of NewHope [2], and incorporate an optimization technique introduced in Kyber FFT [19]. We demonstrate that our high-level framework allows expressing the low-level optimization in Kyber FFT that was the key to its performance. On a recent laptop with Intel Ice Lake CPU, our AVX2- and AVX512-optimized FFT are 1.13x and 1.39x faster than the NewHope AVX2 implementation respectively.

[2] https://github.com/newhopecrypto/newhope-usenix

We would like to stress that obtaining one FFT implementation that outperforms an existing one is not our primary goal. Our goal is to build a program-generation framework which can be applied to *any* existing FFT algorithm, to improve code maintainability, reusability and modularity. To show the effectiveness of our framework, we chose to start with the NewHope reference implementation because of its relative simplicity compared the to the state of the art implementation of Kyber[19], and compare and report our results with respect to the optimized counterpart.

2 Related Work

There is already a large body of work on optimizing FFT or polynomial multiplication as a whole for RLWE-based cryptography [7,18,13,1,2,14,19]. Here, we focus on the most relevant work whose optimized AVX2 implementations are publically available [2,14,19].

The NewHope key-exchange protocol [2] introduced two implementations of FFT for RLWE-based key exchange: One is a reference C implementation, and the other is an AVX2 assembly implementation. The optimized one uses double-precision floating-point instructions to compute modular reduction by $a \bmod q = a - \left\lfloor a\frac{1}{q} \right\rfloor q$, since they found that the floating-point implementation is faster than their integer one when vectorization is applied.

The current fastest FFT implementation is the one used in Kyber KEM [4], described in detail in [19]. For the first time, it outperformed the floating-point implementation in NewHope using only integer SIMD instructions. It also incorporates the optimization used in [13,14] to remove the bit reversal step.

All of the existing work above implement FFT in assembly using AVX2 instructions. To the best of our knowledge, there is no AVX512 implementation, even though an AVX512-capable CPU is becoming widely available. This work presents an AVX512 implementation of FFT for RLWE-based cryptography. In particular, we generate both AVX2 and AVX512 code from the same FFT program.

3 Background

3.1 FFT in the RLWE context

FFT is an $O(n \log n)$ time algorithm to compute Discrete Fourier Transform (DFT) for an input of size n. Given an input $a = (a_0, a_1, ..., a_{n-1}), a_i \in \mathbb{Z}_q$, its DFT $y = (y_0, y_1, ..., y_{n-1}), y_i \in \mathbb{Z}_q$ is defined by the following equation [6]:

$$y_k = \sum_{j=0}^{n-1} a_j \omega_n^{kj}$$

ω_n is the nth primitive root of unity modulo q, satisfying $\omega_n^n \equiv 1 \pmod q$. All addition and multiplication are done in $\bmod\, q$. In the context of RLWE-based cryptography, n is a power of two, and q must satisfy $q \equiv 1 \pmod{2n}$ for FFT to be valid[3]. For example, NewHope uses $n = 1024$ and $q = 12289$ [2] , while Kyber uses $n = 256$ and $q = 7681$ [4].

All existing implementations of FFT in the context of RLWE-based cryptography compute FFT in an iterative, bottom-up manner [2,14,19]. Moreover, the output is computed in-place. Although in-place FFT generally requires a bit reverse step to make an output in the standard order, recent work showed a way to eliminate it entirely for the two transforms (forward and inverse FFT) used in polynomial multiplication [18,13]. However, for simplicity we do not implement the full polynomial multiplication and focus on a standard, self-contained forward FFT which uses the bit reverse step at the beginning.

Algorithm 1 shows the pseudocode of our FFT implementation. It uses the standard Cooley-Tukey algorithm [6] and all powers of ω_n, called twiddle factors, are precomputed and stored in an array. Each iteration of the outermost loop is often called *stage*, and the number of stages is $\log_2 n$. The innermost loop performs the Cooly-Tukey butterfly with modular arithmetic.

Algorithm 1 The pseudocode for the bottom-up, in-place FFT

```
1: procedure FFT
   Input: $a = (a_0, a_1, ..., a_{n-1}) \in \mathbb{Z}_q^n$, precomputed constants table $\Omega \in \mathbb{Z}_q^n$
2: Output: $y = \mathtt{DFT}(a)$, in standard order
3:     bit-reverse($A$)
4:     for ($s = 1$; $s <= \log_2(n)$; $s = s + 1$) do
5:         $m = 2^s$
6:         $o = 2^{s-1} - 1$
7:         for ($k = 0$; $k < m$; $k = k + m$) do
8:             for ($j = 0$; $j < m/2$; $j = j + 1$) do
9:                 $u = a[k + j]$
10:                $t = (a[k + j + m/2] \cdot \Omega[o + j]) \bmod q$
11:                $a[k + j] = (u + t) \bmod q$
12:                $a[k + j + m/2] = (u - t) \bmod q$
13:            end for
14:        end for
15:    end for
16: end procedure
```

An efficient implementation of modular reduction is the most important component in FFT for RLWE-based cryptography. We follow existing work for the choice of algorithms [2,19]: We use Barrett reduction [3] to reduce the results of addition and subtraction, and Montgomery reduction [16] for multiplication.

3.2 Tagless-final style

The tagless-final style is an approach to embed a domain-specific language (DSL) in a general-purpose host language in a type-safe way. The DSL, embedded in our

[3] The requirement on q comes from the fact that in general multiplying two polynomials of degree n requires a transform of size $2n$. However, thanks to the property of negative wrapped convolution, it is enough to do a transform of size n in practice.

host language OCaml, allows high-level descriptions of algorithms independent of security parameters and target architectures. Here, we give its brief introduction to understand the rest of the paper. For more details, see [5,9].

The following OCaml program `vector_add` adds two vectors of integers:

```
let vector_add arr1 arr2 =
  for i = 0 to (n - 1) do
    arr1.(i) = arr1.(i) + arr2.(i)
  done
```

The variable `n` is the length of arrays and assumed to be a compile-time constant. The operator `.(i)` indexes array elements in OCaml.

In the tagless-final style, a DSL program is implemented using language primitives offered by a *signature*. In particular, the signature declares language primitives as abstract functions, whose implementations are yet to be defined.

It is easy to rewrite the above program into an abstract one; we only have to replace all constants and language syntactic constructs by new, abstract functions, for instance, `for` by `for_`, the integer constant `0` by `zero` or `int_ 0`, and `arr1.(i)` by `arr_get arr1 i`. We can write an abstract program equivalent to `vector_add` above in our DSL as follows:

```
func2 "vector_add" arg_ty arg_ty (fun arr1 arr2 ->
    (for_ zero (int_ n) (int_ 1) (fun i ->
         arr_set arr1 i (D.add (arr_get arr1 i) (arr_get arr2 i)))))
```

This is the signature of our C-like language:[4].

```
module type C_lang = sig
  type 'a expr type 'a stmt ...
  val zero : int expr
  val int_ : int -> int expr
  val for_ : int expr -> int expr -> int expr -> (int expr -> unit stmt)
           -> unit stmt
  val arr_set : int array expr -> int expr -> int expr -> unit stmt
  ...
end
```

The tagless-final style lets us implement the signature in various ways, and different implementations give different meanings to the same program.

For example, if we use MetaOCaml [10] for the implementation, the meaning of the above program would become "OCaml code that adds two arrays". The brackets `.<>.` surrounds the value representation of the generated OCaml code.

[4] It is similar to "interface" in other languages.

```
.<let vector_add arg0 arg1 =
  let num_iter_3 = (1024 - 0) / 1 in
  for i_4 = 0 to num_iter_3 - 1 do
    let index_5 = 0 + (1 * i_4) in
    let t_7 = Array.get arg0 index_5 in
    let t_6 = Array.get arg1 index_5 in
    Array.set arg0 index_5 ((t_7 + t_6) mod 12289)
  done in vector_add>.
```

Similarly, we can also obtain equivalent C code, using a different interpretation of the same program. Under this interpretation, a string representation of the C program is generated.

```
void vector_add(uint16_t *arg0, uint16_t *arg1) {
  for (int v_8 = 0; v_8 < ((1024 - 0) / 16); v_8 += 1) {
    arg0[v_8] = arg0[v_8] + arg1[v_8];
  }
}
```

Program transformation can be done by redefining the meaning of language primitives. For example, in this program the meaning of language primitives are overwritten to be a "vectorization mode" by `Vectorize` module (see Appendix A for more details). Under this new interpretation, integer addition, array access and assignment are reinterpreted as vector addition, vector load and store respectively. The loop bound and index are also recalculated accordingly.

```
func2 "vector_add" (fun arr1 arr2->
    let open Vectorize(AVX2_UInt16(D)) in
    for_ (int_ 0) (int_ 1024) (int_ 1) (fun i ->
        arr_set arr1 i (D.add (arr_get arr1 i) (arr_get arr2 i))))
```

Here is the generated code using the AVX2 instruction set. By changing one line, we can also generate AVX512 code, without modifying the DSL program.

```
void vector_add(uint16_t *arg0, uint16_t *arg1) {
  for (int v_8 = 0; v_8 < ((1024 - 0) / 16); v_8 += 1) {
    _mm256_storeu_si256(
        (__m256i *)(arg0 + (0 + (v_8 * 16))),
        _mm256_add_epi16(
            _mm256_loadu_si256((__m256i *)(arg0 + (0 + (v_8 * 16)))),
            _mm256_loadu_si256((__m256i *)(arg1 + (0 + (v_8 * 16))))));
  }
}
```

This shows the strength of the tagless-final style: From a high-level, abstract description of a program, we can generate a variety of low-level code. Although the example above is trivial, the same technique can be applied to vectorize the innermost loop of FFT, as we will see next.

4 The proposed approach

We propose to apply program-generation techniques to an FFT implementation using the tagless-final style, to obtain highly optimized FFT implementations tailored to various security parameters and target architectures. The tagless-final style allows us to give different interpretations of the same abstract program, which makes it possible to generate both AVX2 and AVX512 vectorized FFT code from a single, abstract FFT program.

While the approach and the language used are both high-level, that does not mean we would lose low-level control necessary for the optimal performance. We show that it is possible to reason and program at very low-level, involving delicate arithmetic or vector shuffling, for example. A clean separation between high- and low-level layers is the key to the generality and reusability of our framework.

4.1 Abstract definition of the FFT innermost loop

We begin by translating the pseudocode of bottom up, in-place FFT in Algorithm 1 into our DSL. This is a description of the innermost loop using primitives defined in our DSL.

```
for_ (int_ 0) m_half (int_ 1) (fun j ->
    let index = k %+ j in
    let omega = arr_get prim_root_powers (coeff_begin %+ j) in
    let2
      (D.mul (arr_get input (index %+ m_half)) omega)
      (arr_get input index)
      (fun t u ->
         seq
           (arr_set input index (D.add u t))
           (arr_set input (index %+ m_half) (D.sub u t))))
```

`let2 V1 V2 (fun t u -> V3)` is an syntax sugar for a doubly-nested let binding: `let t = V1 in let u = V2 in V3`. The variable `prim_root_powers` stores precomputed twiddle factors in an array. An array and operations on it are also abstracted using the following signature:

```
module type Array_lang = sig
  include C_lang
  type 'a arr = 'a array

  val arr_init: int -> (int -> 'a expr) -> 'a arr expr
  val arr_get: 'a arr expr -> int expr -> 'a expr
  val arr_set: 'a arr expr -> int expr -> 'a expr -> unit stmt
end
```

Unlike previous implementations where constants are precomputed offline and embedded in the source code without further information, we compute constants at code generation time, in OCaml (not shown). Therefore, our OCaml source code tells exactly how all constants are precomputed.

The domain our FFT will operate on, which in our case is always integers modulo q, is abstracted with the following signature. This is not strictly necessary in the context of this work, but this abstraction increases the reusability of our FFT program: For example, by implementing this signature for complex numbers, we would obtain a standard complex valued FFT implementation from the same FFT program.

```
module type Domain = sig
  type 'a expr   type t
  val lift: t -> t expr
  val add: t expr -> t expr -> t expr
  val sub: t expr -> t expr -> t expr
  val mul: t expr -> t expr -> t expr
end
```

4.2 Vectorizing modular reductions

Our goal is to vectorize the innermost loop. The first challenge we need to address is the vectorization of Barrett and Montgomery reductions [3,16]. Even though the FFT program itself is generic with respect to the choice of parameters and the data type of inputs and outputs, instantiating the implementation of modular reductions requires choosing them ahead of time. We follow the setting of NewHope reference implementation: The input size n is 1024, the modulus parameter q is 12289, and inputs and outputs are arrays of unsigned 16 bit integers whose values fit in 14 bits.

Below is the implementation of Barrett and Montgomery reductions from the NewHope reference implementation, modified slightly for our exposition. The code does not fully reduce modulo q: the output is correct as long as the output from reduction fits in 14 bits [2].

```
static const uint32_t rlog = 18;
static const uint32_t R = 1 << rlog;

uint16_t barrett_reduce(uint16_t a) {
  uint32_t u = ((uint32_t) a * 5) >> 16;
  return a - (uint16_t)(u * PARAM_Q);
}
```

```
uint16_t montgomery_multiply_reduce(uint16_t x, uint16_t twiddle) {
  uint32_t a = ((uint32_t)x * (uint32_t)twiddle) & (R - 1);
  uint32_t u = (a * PARAM_Q_INV) & (R - 1);
  return (a + u * PARAM_Q) >> rlog;
}
```

Suppose that the size of a vector register is 256 bit, so that we can pack 16 unsigned 16 bit integer into one vector register. Since the scalar code uses 32 bit arithmetic, a direct vectorization of these routines requires extracting 8 32 bit values from a 16 element vector, do 8-way 32 bit integer arithmetic, and packing reduced 16 bit values back into an output 16 element register. This is highly inefficient, and not surprisingly the optimized assembly implementation of NewHope uses floating-point SIMD instructions to compute reduction by $a \bmod q = a - \left\lfloor a\frac{1}{q} \right\rfloor q$.

An idea for efficient vectorization of modular reduction using 16 bit integer SIMD instructions was introduced in [19]. The key observation is that 32 bit value is introduced as a result of multiplying two 16 bit integers, and the multiplication is always followed by division or modulo by a power of two. For example, the first multiplication in Barrett reduction is immediately followed by a right shift of 16. Since we immediately discard the lower 16 bits half of the product, we only have to compute the upper 16 bits half of it. AVX has the `mulhi` instruction for this purpose. Similarly, the second 32 bit multiplication can be replaced by `mullo` instruction, since the cast to 16 bit discards the upper 16 bits half of the product.

The case for Montgomery reduction is not as straightforward, since NewHope uses the constant $R = 2^{18}$ as the divisor. Naively replacing 18 by 16 leads to an incorrect result, because the result of reduction is not guaranteed to fit in 14 bits if we right shift a 32 bit value by 16. Fortunately, one conditional subtraction by q is enough to make the output of the reduction fit in 14 bits. Using $R = 2^{16}$, we can replace the two occurrences of the multiplication followed by modulo R (`& (R - 1)` in the code) by 16 bit `mullo` instruction.

The last multiplication by q is not followed by either division or modulo by R, so it seems we cannot replace this 32 bit multiplication by either 16 bit `mulhi` or `mullo` instructions. Here, we can exploit a property of Montgomery reduction: The result of the last addition is guaranteed to be divisible by R. Since we can make R to be 2^{16}, this means that low 16 bits half of the addition result is 0. This suggests the possibility to multiply only the high 16 bits half just before the addition, and do the addition of the high 16 bits. To realize this, we need to take care of a carry bit from the lower 16 bits half addition that we are going to omit. We can determine if a carry is necessary by examining the lower 16 bits half of the left hand side of the addition (`a` in the code). If it is zero, there is no carry. Otherwise, there must be a carry into the 17-th bit because of the requirement that lower 16 bits half is zero after addition. In the latter case, we need to explicitly add a carry bit after the last addition.

Based on the above reasoning, we declare a DSL signature necessary to implement vectorized reductions as follows:

```
module type SIMD_Instr = sig
  val broadcast: t -> (t, n) vec expr
  val add: (t, n) vec expr -> (t, n) vec expr -> (t, n) vec expr
  val sub: (t, n) vec expr -> (t, n) vec expr -> (t, n) vec expr
  val mullo: (t, n) vec expr -> (t, n) vec expr -> (t, n) vec expr
  val mulhi: (t, n) vec expr -> (t, n) vec expr -> (t, n) vec expr
  val bitwise_and: (t, n) vec expr -> (t, n) vec expr -> (t, n) vec expr
  val shift_right_a: (t, n) vec expr -> int -> (t, n) vec expr
  val not_zero: (t, n) vec expr -> (t, n) vec expr
  module Infix: sig
    val (%+): (t, n) vec expr -> (t, n) vec expr -> (t, n) vec expr
    val (%-): (t, n) vec expr -> (t, n) vec expr -> (t, n) vec expr
  end
end
```

`(t, n) vec` is a type of a length `n` vector whose element type is `t`, but the details are not important. `not_zero` takes a vector and return `0x0000` or `0x0001` for each element depending on whether or not the input element is not zero. `shift_right_a` does an arithmetic right shift and it is used to implement the constant time conditional subtraction `csub` (not shown) [19].

Using the primitives above, we can implement vectorized reductions as follows. The final line in `vmul` (for Montgomery modular multiplication) does the conditional subtraction, to ensure that the result of modular multiplication fits in 14 bits as required by our specification.

```
func "barrett_reduce" in_ty (fun v ->
    let vec_5 = broadcast 5 in
    let v_1 = mulhi v vec_5 in
    let vec_q = broadcast Param.q in
    return_ (v %- (mullo v_1 vec_q)))

func2 "vmul" in_ty in_ty (fun v1 v2 ->
   let_ (mullo v1 v2) (fun mlo ->
   let_ (mulhi v1 v2) (fun mhi ->
   let_ (broadcast Param.q) (fun vec_q ->
   let_ (broadcast Param.qinv) (fun vec_qinv ->
   let_ (mullo mlo vec_qinv) (fun mlo_qinv ->
   let_ (mulhi mlo_qinv vec_q) (fun t ->
   let_ (not_zero mlo) (fun carry ->
   let_ (mhi %+ t %+ carry) (fun res ->
   return_ (app csub res)))))))))
```

Here is the generated vectorized Barrett reduction and Montgomery modular multiplication, using the AVX2 instruction set.

```
__m256i barrett_reduce(__m256i arg0) {
  return _mm256_sub_epi16(
    arg0, _mm256_mullo_epi16(_mm256_mulhi_epu16(arg0,
                                                 _mm256_set1_epi16(5)),
                             _mm256_set1_epi16(12289)));
}

__m256i vmul(__m256i arg0, __m256i arg1) {
  __m256i v_19 = _mm256_mullo_epi16(arg0, arg1);
  __m256i v_20 = _mm256_mulhi_epu16(arg0, arg1);
  __m256i v_21 = _mm256_set1_epi16(12289);
  __m256i v_22 = _mm256_set1_epi16(12287);
  __m256i v_23 = _mm256_mullo_epi16(v_19, v_22);
  __m256i v_24 = _mm256_mulhi_epu16(v_23, v_21);
  __m256i v_25 = _mm256_add_epi16(
      _mm256_cmpeq_epi16(v_19, _mm256_set1_epi16(0)),
      _mm256_set1_epi16(1));
  __m256i v_26 = _mm256_add_epi16(_mm256_add_epi16(v_20, v_24), v_25);
  return csub(v_26);
}
```

Note that the optimizations in this section is generic with respect to the choice of instruction sets. Indeed, we can generate AVX512 code by changing only the last part of code generation (see Section 4.6).

4.3 Subtraction

Since we use unsigned arithmetic following the reference implementation of NewHope, we need to be careful with the subtraction in the butterfly operation. To prevent an unsigned underflow, we need to add a sufficiently large multiple of q, which we call the bias, to the left hand side of the subtraction. The reference implementation of NewHope first casts the left-hand side to 32 bit and adds $3q$. However, we would like to avoid the cast to 32 bit, because we want to vectorize with 16 bit arithmetic. Since $3q = 36867 > 2^{15}$, and the left hand side of the subtraction can be 15 bit because of the lazy reduction explained later, naively adding $3q$, using 16 bit arithmetic, could lead to an overflow. Therefore, a more careful analysis is needed to allow 16 bit vectorized arithmetic without a concern for overflow or unsigned underflow.

The right hand side of the subtraction is always the result of a modular multiplication, which is guaranteed to fit in 14 bit. Therefore, the bias needs to be bigger than the maximum of a 14 bit unsigned integer, $2^{14} - 1$. Since the left hand side can be 15 bit, the bias needs to fit in 15 bits to avoid overflow in the addition using 16 bit arithmetic. The above considerations leads to the choice of $2q = 24578$ as the bias, since $2^{14} - 1 < 24578 < 2^{15} - 1$.

The result of the bias addition followed by subtraction in general requires 16 bits. However, the result of a modular subtraction needs to be either 14 or 15 bit, following the specification we adopted from NewHope. We chose to reduce the result of the subtraction to 14 bits via one Barrett reduction. This means that the lazy reduction will not be concerned with the result of the subtraction in the butterfly operation.

This is the implementation of the vectorized modular subtraction in our DSL. Generated AVX2 code is shown below.

```
func2 "vsub" in_ty in_ty (fun v1 v2 ->
    let bias = broadcast (Param.q * 2) in
    return_ (app barrett_reduce ((v1 %+ bias) %- v2)))
```

```
__m256i vsub(__m256i arg0, __m256i arg1) {
  return barrett_reduce(
      _mm256_sub_epi16(_mm256_add_epi16(arg0, _mm256_set1_epi16(24578)),
                       arg1));
}
```

We also define a vectorized addition `vadd`, whose implementation is omitted because it is simply a wrapper around `add` (`%+` in the infix notation).

4.4 Vectorizing the innermost loop

Having decided all the ingredients necessary to vectorize the butterfly operation, we can now discuss how we vectorize the innermost loop.

Since the number of iterations in the innermost loop is different for each stage, vectorization needs to be done carefully. Suppose that the width of a vector is 16. When the input size is 1024, as in NewHope, the number of iterations becomes greater or equal to 16 after the fifth stage. Therefore, for stages after the fifth stage, vectorization can be done by just adding one line, thanks to `Vectorize` module introduced in the section 3.2.

```
let open Vectorize(V_lang) in
for_ (int_ 0) m_half (int_ 1) (fun j ->
   ...
```

Here is an example of the generated AVX2 code, for the fifth stage. Note the use of `vadd`, `vsub`, and `vmul` introduced earlier.

```
for (int v_90 = 0; v_90 < ((16 - 0) / 16); v_90 += 1) {
  __m256i v_99 =
      vmul(_mm256_loadu_si256(
               (__m256i *)(arg0 + ((v_89 + (0 + (v_90 * 16))) + 16))),
           _mm256_loadu_si256((__m256i *)(v_9 + (64 + (0 + (v_90 * 16))))));
  __m256i v_100 =
      _mm256_loadu_si256((__m256i *)(arg0 + (v_89 + (0 + (v_90 * 16)))));
  _mm256_storeu_si256((__m256i *)(arg0 + (v_89 + (0 + (v_90 * 16)))),
                    vadd(v_100, v_99));
  _mm256_storeu_si256((__m256i *)(arg0 + ((v_89 + (0 + (v_90 * 16))) + 16)),
                    vsub(v_100, v_99));
}
```

Vectorizing the earlier stages is more difficult, since the number of iterations of the loop we want to vectorize is less than the vector width. The situation and the idea for the solution are illustrated in Fig.1, for the case where the vector width is 4. Two butterfly operations on four neighboring elements on the left corresponds to one innermost loop. Since the top two and bottom two elements undergo different operations, we cannot apply vectorization to the four-element group. The key to enable vectorization is to operate on two neighboring four-element groups at the same time. By shuffling elements between two groups, we can group elements that undergo the same operations into one vector. For example, the blue elements are first multiplied by twiddle factors, and added and subtracted with the red elements. By applying the same shuffling to the result of vectorized butterfly operations, we recover the expected results.

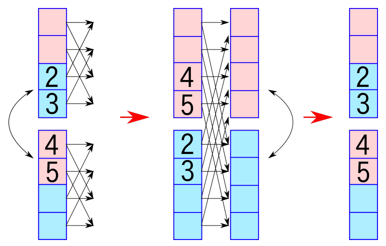

Fig. 1. Shuffling elements between neighboring two vectors enables vectorization

Based on the observation above, we introduce the `shuffle` primitive in our language. Since each stage requires different shuffling, this primitive takes an integer representing the stage as its arguments. Given two vectors, it shuffles elements between them and returns a new pair of vectors.

```
val shuffle: int -> t vec expr -> t vec expr -> (t vec expr * t vec expr)
```

Using the `shuffle` primitive, we can write the vectorized inner loop for earlier stages as follows. `vadd`, `vsub`, and `vmul` are vectorized modular arithmetic introduced earlier. `vload`, `vstore` are vector load and store respectively.

```
for_ zero (int_ n) (int_ (vec_len * 2)) (fun k ->
   let_ (vload input k) (fun v0 ->
   let_ (vload input (k %+ (int_ vec_len))) (fun v1 ->
   let2_ (shuffle s v0 v1) (fun v_lo v_hi ->
  let_ (vmul v_hi coeff) (fun v_mul ->
  let_ (vadd v_lo v_mul) (fun tmp_add ->
  let_ (vsub v_lo v_mul) (fun tmp_sub ->
  let2_ (shuffle s tmp_add tmp_sub) (fun v0_res v1_res ->
  seq
     (vstore input k v0_res)
     (vstore input (k %+ (int_ vec_len)) v1_res))))))))))
```

By abstracting the details of different shuffling into the `shuffle` primitive, we are able to write a generic vectorized inner loop that can be specialized to each stage. Unlike the vectorization of the innermost loop for later stages where we do not have to change the original sequential FFT program, the vectorization for earlier stages required rewriting the innermost loop with explicit vectorized primitives. Although it is not ideal in terms of program reuse, we believe that the rewriting was necessary to enable low-level optimization involving shuffling. Our framework is flexible enough to accomodate both the trivial vectorization via `Vectorize` module and the explicit vectorization using vectorized primitives.

4.5 Lazy reduction

The reference implementation of NewHope does not always reduce the result of addition to 14 bits. Since the result of adding two 14 bit values fits in 15 bits, in the next stage we can do another addition of 15 bit values without causing 16 bit overflow. Therefore, Barrett reduction is applied every other stage. This is called lazy reduction [2].

In our implementation, the addition follows the same lazy approach as NewHope, applying Barrett reduction every other stage. For subtraction, we always reduce the result to 14 bits as explained earlier. The implementation detail of our lazy reduction is in Appendix B.

4.6 SIMD Backend implementation

We have shown "vectorized" programs without specifying which SIMD instruction sets we use. At the lowest layer of abstraction in our framework, we need a mapping between our primitives and concrete SIMD instructions.

For the AVX2 backend, we specify the mapping in the following way. The AVX512 backend is entirely similar, modulo the names of instructions and the vector length.

```
module AVX2_v16_Instr = struct
  let vec_len = 16
  let add = "_mm256_add_epi16"
  let mullo = "_mm256_mullo_epi16"
  let mulhi = "_mm256_mulhi_epi16"
  ...
end
```

We also need to specify the implementation of shuffle operations for earlier stages. For the AVX2 backend, for example, we need to implement different shuffle operations for the stages between 1 and 4. See Appendix C for more details.

5 Experiments

We benchmarked our generated code against the optimized AVX2 implementation of NewHope. We used Clang version 11 with `-O3` to compile our code. Each implementation was run 100000 times on an input of size 1024 and we recorded average cycles spent using `perf_event` feature in the Linux kernel. Furthermore, we take the median of 100 average cycles measurements, since we observed some variations in the average cycles count during our experiment.

The result on a desktop machine with Intel Coffee Lake CPU is shown in Table 1. The efficient vectorization using only 16 bit integer instructions is the only reason we were able to outperform NewHope: While we are able to pack four times more elements into a one vector register[5], NewHope uses more optimizations at the assembly level, such as merging multiple stages to compute as much as possible inside registers [7,19]. In contrast, we compute intermediate outputs stage by stage following the pseudocode in Algorithm 1 and overall our code is much simpler and more readable than the NewHope assembly implementation.

Table 1. Cycle counts on Core i7-8700K (Coffee Lake, AVX2)

	Cycle counts	Speedup over NewHope
NewHope	6903	
Our AVX2 result	6099	1.132

AVX512 is becoming widely available in consumer laptops. Since our framework can easily target AVX512 instructions from the same abstract definition of FFT, we were able to generate an optimized AVX512 FFT implementation without significant effort after we completed the AVX2 one[6]. Table 2 shows the benchmark result on a laptop with AVX512-capable Intel Ice Lake CPU. AVX512 gave speedup of 23% over our AVX2 result.

Table 2. Cycle counts on Core i7-1065G7 (Ice Lake, AVX2 and AVX512)

	Cycle counts	Speedup over NewHope
NewHope	6082	
Our AVX2 result	5398	1.127
Our AVX512 result	4381	1.388

[5] Recall that NewHope uses double precision floating-point instructions to compute reductions.

[6] Both of our AVX2 and AVX512 support code are less than 90 lines of OCaml.

The work on Kyber showed that their AVX2 forward NTT implementation, tailored for the NewHope parameters ($n = 1024, q = 12289$), achieved 3.5x speedup against the NewHope AVX2 implementation [19]. Based on results from Table 1 and 2, our AVX2 implementation are expected to be significantly slower than Kyber. A direct comparison is not possible at the moment since the Kyber implementation that works with the NewHope parameters is not publically available and both we and Kyber implement optimizations that are specific to respective choice of parameters (for example, Kyber does not apply any Barrett reduction during the forward transform). It would be interesting to apply our program-generation framework to a Kyber-based implementation: To do so, we need to analyze the low-level optimizations of Kyber further and provide high-level descriptions for them, which is left for future work.

6 Conclusion

We have proposed implementing optimized FFT for RLWE-based cryptography via a program-generation approach. By separating the high-level description of the FFT program from low-level details concerning arithmetic and vectorization, we have achieved a reusable FFT program-generation framework. Generated code is also efficient, outperforming the NewHope assembly implementation by non-trivial factors using AVX2 and AVX512 instruction sets. Our implementation is available at `https://github.com/masahi/iwsec21_ntt`.

This work opens up several avenues for future work. For the code generation aspect, we believe further speedup is possible: For example, while both existing and our work use the simplest formulation of FFT using the radix 2 butterfly exclusively, we are also interested in exploring the radix 4 or the split radix variants which involve fewer multiplications than the dominant radix 2 case [11,8]. Instantiating the implementation of Kyber FFT [19] in our framework or adding new targets, such as ARM or RISC-V would also be interesting.

Finally, since the correctness of the code is of paramount importance in cryptography in general, we would like to offer some notion of correctness assurance. One way is to automatically prove that our implementation does not have the possibility of overflow: As we have shown in Section 4.2 and 4.3, making sure and be confident that our implementation of modular arithmetic is free from overflow required very careful low-level reasoning. A promising direction has been demonstrated in the recent work of Navas et al. [17]. We believe starting from abstract high-level description of the program, as proposed in this work, opens up many possibilities for such verification effort.

Acknowledgement

We thank Tadanori Teruya for helpful discussion. Feedbacks from anonymous reviewers helped improve this paper and are greatly appreciated. The second author is supported in part by JSPS Grant-in-Aid for Scientific Research (B) 18H03218.

Appendix A Vectorize module

Vectorize module is used to generate vectorized code for trivially vectorizable loops. It simply redefines the meaning of language primitives used in a sequential program so that the same program can evaluated to vectorized loop code. It is implemented as a OCaml functor, which is often used in the tagless-final style to extend the meaning of existing DSL.

```
module Vectorize(Base_lang: Vector_lang): Vec
 ...
= struct

  module D = struct
    let add = Vec_D.vadd
    ...
  end

  let arr_get = Base_lang.vload
  let arr_set = Base_lang.vstore

  let vec_len = ...

  let for_ lo hi _ body =
    let num_loop = (hi %- lo) %/ (int_ vec_len) in
    Base_lang.for_ (int_ 0) num_loop (int_ 1) (fun i ->
        body (lo %+ (i %* (int_ vec_len))))
end
```

Appendix B Lazy reduction implementation

We implement lazy reduction again as a OCaml functor, extending the original meanings of vadd and vsub to give semantics of lazy reduction. As explained in Section 4.3, we allow the result of addition to stay in 15 bits and apply Barrett reduction every other stage, while the result of subtraction is reduced to 14 bits in every stage by Barrett reduction. We can implement such specification for lazy reduction as follows:

```
module Lazy_reduction(V: Vector_lang)(Stage: sig val s: int end) : Vector_lang
  (* ... *)
struct
  include V
  module Vector_domain = struct
    let vadd v0 v1 =
      let res = V.Vector_domain.vadd v0 v1 in
      if Stage.s mod 2 == 0 then barrett_reduce res
      else res
    let vsub = V.Vector_domain.vsub
  end
end
```

This is used in our FFT code generator as follows. `Lazy_reduction` is instantiated for each stage `s`, and by simply wrapping the original meanings of vectorized primitives such as `vadd` and `vsub` defined in `V_lang`, the innermost loop now executes with lazy reduction enabled. Note that we do not have to change the code of the innermost loop at all. The tagless-final style allows such extension in a highly modular manner.

```
let fft n =
  ...
  let fft_stage s =
    ...
    let module V_lang_lazy = Lazy_reduction(V_lang)(struct let s = s end) in
    func fname input_ty (fun input ->
         ...
         let open V_lang_lazy in
         let open V_lang_lazy.Vector_domain in
         ...
```

Appendix C Details on SIMD backend implementation

This is the full mapping between language primitives used in vectorized reductions of Section 4.2 and corresponding AVX2 instructions.

```
module AVX2_v16_Instr : SIMD_str = struct
  let add = "_mm256_add_epi16"
  let sub = "_mm256_sub_epi16"
  let mullo = "_mm256_mullo_epi16"
  let mulhi = "_mm256_mulhi_epu16"
  let broadcast = "_mm256_set1_epi16"
  let shift_right_a = "_mm256_srai_epi16"
  let bitwise_and = "_mm256_and_si256"
  let not_zero v =
    sprintf "_mm256_add_epi16(_mm256_cmpeq_epi16(%s, _mm256_set1_epi16(0)),
                              _mm256_set1_epi16(1))" v
end
```

`not_zero` primitive is implemented in a cumbersome way, since AVX instruction returns `0xFFFF` or `0x0000` for the result of comparison instructions, while we need `0x0001` or `0x0000` to represent the presence or absence of the carry bit. `not_zero` primitive hides such details specific to a particular ISA and provides a straightforward interface to a programmer.

Shuffle operations can be implemented by `shift`, `blend`, `unpack`, and `permute` instructions. The implementation using AVX2 is shown below. The AVX512 counterpart is entirely similar but uses different instruction combinations to realize desired permutations.

```
let shuffle1 v0 v1 =
  let v1_left_shift = sprintf "_mm256_slli_epi32(%s, 16)" v1 in
  let v0_right_shift = sprintf "_mm256_srli_epi32(%s, 16)" v0 in
  let v_lo = sprintf "_mm256_blend_epi16(%s, %s, 0xAA)" v0 v1_left_shift in
  let v_hi = sprintf "_mm256_blend_epi16(%s, %s, 0xAA)" v0_right_shift v1 in
  v_lo, v_hi

let shuffle2 v0 v1 =
  let v1_left_shift = sprintf "_mm256_slli_epi64(%s, 32)" v1 in
  let v0_right_shift = sprintf "_mm256_srli_epi64(%s, 32)" v0 in
  let v_lo = sprintf "_mm256_blend_epi32(%s, %s, 0xAA)" v0 v1_left_shift in
  let v_hi = sprintf "_mm256_blend_epi32(%s, %s, 0xAA)" v0_right_shift v1 in
  v_lo, v_hi

let shuffle3 v0 v1 =
  let v_lo = sprintf "_mm256_unpacklo_epi64(%s, %s)" v0 v1 in
  let v_hi = sprintf "_mm256_unpackhi_epi64(%s, %s)" v0 v1 in
  v_lo, v_hi

let shuffle4 v0 v1 =
  let v_lo = sprintf "_mm256_permute2x128_si256(%s, %s, 0x20)" v0 v1 in
  let v_hi = sprintf "_mm256_permute2x128_si256(%s, %s, 0x31)" v0 v1 in
  v_lo, v_hi

let shuffle n v0 v1 = match n with
  | 1 -> shuffle1 v0 v1
  | 2 -> shuffle2 v0 v1
  | 3 -> shuffle3 v0 v1
  | 4 -> shuffle4 v0 v1
  | _ -> assert false
```

References

1. Aguilar-Melchor, C., Barrier, J., Guelton, S., Guinet, A., Killijian, M.O., Lepoint, T.: NFLlib: NTT-based fast lattice library. In: Sako, K. (ed.) Topics in Cryptology - CT-RSA 2016. pp. 341–356. Springer International Publishing, Cham (2016)
2. Alkim, E., Ducas, L., Pöppelmann, T., Schwabe, P.: Post-quantum key exchange: A new hope. In: Proceedings of the 25th USENIX Conference on Security Symposium. p. 327–343. SEC'16, USENIX Association, USA (2016)
3. Barrett, P.: Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In: Proceedings on Advances in Cryptology—CRYPTO '86. p. 311–323. Springer-Verlag, Berlin, Heidelberg (1987)

4. Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J., Schwabe, P., Seiler, G., Stehle, D.: Crystals - kyber: A cca-secure module-lattice-based kem. pp. 353–367 (04 2018). https://doi.org/10.1109/EuroSP.2018.00032
5. Carette, J., Kiselyov, O., Shan, C.c.: Finally tagless, partially evaluated. In: Shao, Z. (ed.) Programming Languages and Systems. pp. 222–238. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
6. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, Third Edition. The MIT Press, 3rd edn. (2009)
7. Güneysu, T., Oder, T., Pöppelmann, T., Schwabe, P.: Software speed records for lattice-based signatures. In: Gaborit, P. (ed.) Post-Quantum Cryptography. pp. 67–82. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
8. Johnson, S.G., Frigo, M.: A modified split-radix FFT with fewer arithmetic operations. Trans. Sig. Proc. **55**(1), 111–119 (Jan 2007). https://doi.org/10.1109/TSP.2006.882087
9. Kiselyov, O.: Typed Tagless Final Interpreters. Generic and Indexed Programming: International Spring School, SSGIP 2010, Oxford, UK, March 22-26, 2010, Revised Lectures, pp. 130–174. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
10. Kiselyov, O.: Reconciling abstraction with high performance: A MetaOCaml approach. Found. Trends Program. Lang. **5**(1), 1–101 (Jun 2018). https://doi.org/10.1561/2500000038
11. Kiselyov, O., Taha, W.: Relating fftw and split-radix. In: Wu, Z., Chen, C., Guo, M., Bu, J. (eds.) Embedded Software and Systems. pp. 488–493. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)
12. Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D., Vouillon, J.: The OCaml system release 4.11. `https://caml.inria.fr/pub/docs/manual-ocaml/` (2020)
13. Liu, Z., Pöppelmann, T., Oder, T., Seo, H., Roy, S.S., Güneysu, T., Großschädl, J., Kim, H., Verbauwhede, I.: High-performance ideal lattice-based cryptography on 8-bit AVR microcontrollers. ACM Trans. Embed. Comput. Syst. **16**(4) (Jul 2017). https://doi.org/10.1145/3092951
14. Longa, P., Naehrig, M.: Speeding up the number theoretic transform for faster ideal lattice-based cryptography. In: Foresti, S., Persiano, G. (eds.) Cryptology and Network Security. pp. 124–139. Springer International Publishing, Cham (2016)
15. Lyubashevsky, V., Peikert, C., Regev, O.: On ideal lattices and learning with errors over rings. In: Gilbert, H. (ed.) Advances in Cryptology – EUROCRYPT 2010. pp. 1–23. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
16. Montgomery, P.L.: Modular multiplication without trial division. Mathematics of Computation **44**, 519–521 (1985)
17. Navas, J.A., Dutertre, B., Mason, I.A.: Verification of an optimized ntt algorithm. In: Christakis, M., Polikarpova, N., Duggirala, P.S., Schrammel, P. (eds.) Software Verification. pp. 144–160. Springer International Publishing, Cham (2020)
18. Roy, S.S., Vercauteren, F., Mentens, N., Chen, D.D., Verbauwhede, I.: Compact ring-lwe cryptoprocessor. In: Batina, L., Robshaw, M. (eds.) Cryptographic Hardware and Embedded Systems – CHES 2014. pp. 371–391. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)
19. Seiler, G.: Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography. IACR Cryptol. ePrint Arch. **2018**, 39 (2018)