Generating High-Performance Number Theoretic Transform Implementations for Vector Architectures

Naifeng Zhang* Austin Ebel† Negar Neda† Patrick Brinich‡ Benedict Reynwar§ Andrew G. Schmidt§
Mike Franusich¶ Jeremy Johnson‡ Brandon Reagen† and Franz Franchetti*
*Carnegie Mellon University {naifengz, franzf}@cmu.edu
†New York University {abe5240, nn2231, bjr5}@nyu.edu
‡Drexel University {pjb338, johnsojr}@drexel.edu
§USC Information Sciences Institute {breynwar, aschmidt}@isi.edu
‖SpiralGen, Inc. mike.franusich@spiralgen.com

Abstract—**Fully homomorphic encryption (FHE) offers the ability to perform computations directly on encrypted data by encoding numerical vectors onto mathematical structures. However, the adoption of FHE is hindered by substantial overheads that make it impractical for many applications. Number theoretic transforms (NTTs) are a key optimization technique for FHE by accelerating vector convolutions. Towards practical usage of FHE, we propose to use SPIRAL, a code generator renowned for generating efficient linear transform implementations, to generate high-performance NTT on vector architectures. We identify suitable NTT algorithms and translate the dataflow graphs of those algorithms into SPIRAL's internal mathematical representations. We then implement the entire workflow required for generating efficient vectorized NTT code. In this work, we target the Ring Processing Unit (RPU), a multi-tile long vector accelerator designed for FHE computations. On average, the SPIRAL-generated NTT kernel achieves a $1.7\times$ speedup over naive implementations on RPU, showcasing the effectiveness of our approach towards maximizing performance for NTT computations on vector architectures.**

Index Terms—**Fully homomorphic encryption, number theoretic transform, SPIRAL, code generation, vectorization**

I. Introduction

Fully homomorphic encryption (FHE) [1] enables direct computation on sensitive data by applying mathematical operations on encrypted information. FHE utilizes lattice-based cryptography to encode vectors of numerical data onto mathematical structures, such as lattices and rings. This encryption scheme allows for the execution of basic arithmetic operations while the data remains encrypted. Various applications, including pattern matching, linear algebra, basic statistics, and machine learning, can be achieved by combining basic homomorphic operations.

To handle different types of data, several schemes have been proposed for FHE, such as BGV [2], BFV [3], CKKS [4], TFHE [5] and FHEW [6]. These schemes are based on lattice cryptography and require a fundamental set of mathematical operations in the form of *integer modulo vector* arithmetic. Although these schemes offer ideal privacy protection targeting different data types, their real-world adoption is limited due to prohibitive overheads. On CPU, FHE computations are remarkably slower, ranging from 10,000× to 100,000× compared to equivalent unencrypted computations, even when utilizing highly optimized FHE libraries [7].

This work is supported by the Defense Advanced Research Projects Agency (DARPA) under contract HR0011-20-S0032. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

Achieving efficient implementation of FHE schemes is crucial for their adoption. A major optimization technique is utilizing number theoretic transforms (NTTs) to accelerate the computation of vector convolutions that represent the product of two polynomials whose coefficients are stored in the vectors. This process is similar to the fast Fourier transform (FFT)-based convolution in the signal processing domain. Consequently, numerous previous research efforts have been dedicated to accelerating NTT on multiple platforms [8]–[10], especially for specialized accelerators with a vector architecture [7], [11], [12].

The need for high-performance vectorized NTT with different algorithmic settings targeting different platforms is therefore imminent. This requires automatic code generation and autotuning of various NTT implementations, similar to prior work in the FFT domain [13], [14]. We propose to use SPIRAL [15]–[17], a code generation system that excels in generating high-performance code in the realm of linear transforms such as the discrete Fourier transform (DFT), to automatic generate high-performance NTT code targeting various architectures.

Contributions. Our key contributions are:

- Introducing the utilization of SPIRAL to generate high-performance NTT code on vector architectures, thereby improving the practicality of FHE.
- Identifying and translating suitable NTT algorithms into SPIRAL's internal mathematical representations.
- Implementing an end-to-end workflow that begins with C APIs and progresses through SPIRAL scripts and breakdown rules, ultimately resulting in the generation of optimized vectorized NTT code.
- Demonstrating the effectiveness of the SPIRAL-based approach by achieving an average speedup of $1.7\times$ than naive implementations on a vector architecture designed for FHE computations.

II. Background

Number Theoretic Transform. The DFT is defined by

$$y(k) = \sum_{j=0}^{n-1} x(j)\omega_n^{jk}, \quad 0 \le k \le n-1, \tag{1}$$

where $\omega_n = e^{-2\pi i/n}$ and $i = \sqrt{-1}$. NTT is the special case of DFT which operates on integers over a finite field $\mathbb{F}_p = \mathbb{Z}/p\mathbb{Z}$, where p is a prime number. NTT is defined as

$$y(k) = \sum_{j=0}^{n-1} x(j)\omega_n^{jk} \bmod p, \quad 0 \le k \le n-1, \tag{2}$$

where ω_n is the n-th primitive root of unity. As the Fourier transform converts a signal from the time domain to its representation in the frequency domain, NTT can be seen as a transform of a polynomial from the coefficient form (e.g., $p(x) = 3x^3 + 4x^2 + 4x + 1 \bmod 5$) to the evaluation form over the finite field ($\{p(0), p(1), p(2), p(3), p(4)\}$), thereby reducing the time complexity of polynomial multiplication to $O(n \log n)$. Given the close mathematical nature of NTT and DFT, FFT algorithms can be easily applied to NTT.

SPIRAL. SPIRAL is a program generation/synthesis system that takes in high-level mathematical specifications and selected architectural and microarchitectural parameters and produces highly optimized implementations. The system uses domain-specific language based on mathematics, which is declarative and platform-independent, to represent *algorithm knowledge* in the form of breakdown rules. The breakdown rules are divide-and-conquer algorithms that enable the mapping to various forms of parallelism, and the recursion step closure helps derive the library structure for general input size implementations. *Platform knowledge* is organized into paradigms, which are features of a platform that require structural optimization and possibly source code extensions. Each paradigm consists of a set of parameterized rewrite rules and base cases that interact with the breakdown rules to produce optimized algorithms for the considered paradigm. SPIRAL also uses *empirical search* to automatically explore choices in a feedback loop, generating candidate implementations and evaluating their performance. This approach enables further optimization for intricate microarchitectural details that may be unknown or not well understood.

SPIRAL has demonstrated across a wide range of hardware architectures that it is able to produce software that outperforms the best human programmers, especially for linear transforms such as DFT. Early work in SPIRAL has already provided support for modular FFT for the Maple computer algebra system [18], which paves the way for us to expand SPIRAL to NTT and associated helper functions.

III. Related Work

As FHE gains popularity, there has been a large body of work on accelerating NTTs over the past few years, with a focus on hand-optimized implementations on CPU, GPU, FPGA, and ASIC. We also discuss past work on auto-generating FFT-based implementations using SPIRAL.

NTT Acceleration. Takahashi [8] implements parallelized NTT using Intel Advanced Vector Extensions 512 (AVX-512) on the CPU. The author uses AVX-512 instructions to vectorize NTT kernels and OpenMP to parallelize NTT using the six-step FFT algorithm. Ye et al. [10] designed the first FPGA architecture specifically for TFHE primitives. In order to facilitate the effective utilization of multi-level parallelism, the authors tailor the data arrangement of TFHE ciphertext for on-chip SRAM in FPGA. Özerk et al. [9] develop an efficient and fast implementation of NTT for GPU. To demonstrate the practical application of the GPU implementation, they conduct experiments on the key generation, encryption, and decryption operations in FHE using Microsoft's SEAL homomorphic encryption library on GPU. Samardzic et al. [7] introduce CraterLake, the first FHE wide-vector uniprocessor with specialized functional units, supporting FHE computations of unbounded depth. Soni et al. [12] introduce B512, a novel vector instruction set architecture (ISA) tailored to the needs of ring processing in homomorphic encryption. B512 supports a vector length of 512 for highly parallel execution.

Code Generation. To the best of our knowledge, limited work has been done for NTT code generation targeting different platforms. Yang et al. [19] propose NTTGen, a framework to automatically generate low latency NTT designs targeting homomorphic encryption-based applications, which takes in application parameters, latency, and hardware resource constraints and outputs synthesizable Verilog code based on the hardware templates.

SPIRAL is known for its success in generating high-performance FFT code. Powered by SPIRAL, FFTX [13] is a novel framework designed to facilitate the development of high-performance applications that utilize FFT on exascale machines. The complex architectures of these machines introduce multiple levels of parallelism, requiring efficient methods for data communication. In the graph domain, GBTLX [20] transforms graph processing programs written using the GraphBLAS Template Library (GBTL) into high-performance C programs. This code generator is capable of producing C programs that achieve performance comparable to manually optimized implementations. NTTX, the SPIRAL-based code generator for NTT and its applications, has been briefly discussed in terms of how it functions in an end-to-end FHE accelerator [21] and its significant speedup over expert implementations of certain NTT sizes on GPU [22]. In this work, we will discuss in detail the code generation process in NTTX and how to target vector architectures via generating single instruction, multiple data (SIMD) instructions, using the Ring Processing Unit (RPU) [12] as an example.

IV. NTT Algorithms in Operator Language

FFT/NTT Algorithms. Given the similarities between definitions of DFT and NTT, FFT algorithms can be directly applied to NTT computations. We started with the classic Cooley-Tukey FFT/NTT algorithm [23] and added the Korn-Lambiotte FFT/NTT algorithm [24] and its inverse, the Pease FFT/NTT algorithm [25] to SPIRAL for vector architectures.

Both algorithms' dataflow graph is shown in Fig. 1 and Fig. 2, respectively. In the following texts, FFT/NTT algorithms will be referred to as NTT algorithms for simplicity.

We chose the Korn-Lambiotte and Pease NTT algorithms due to their constant geometry characteristics. That is, the butterfly (i.e., the cross in the dataflow graphs) accessing pattern and communication pattern are all the same across stages [26]. This is due to the fact that most vector architectures have expensive shuffle instructions while having relatively limited shuffle capability. RPU, for example, has a reduced ISA working with long vectors (vector length of 1,024).

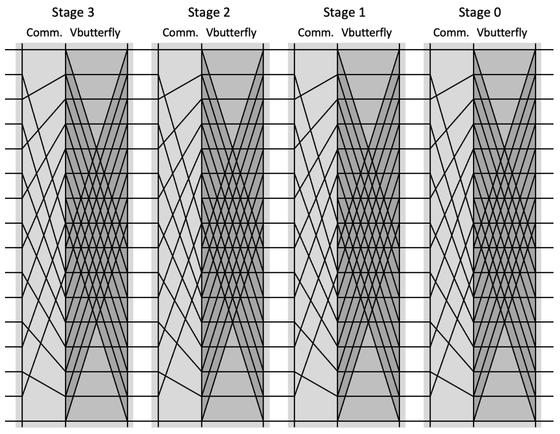

Fig. 1. Dataflow of the Korn-Lambiotte FFT/NTT Algorithm.

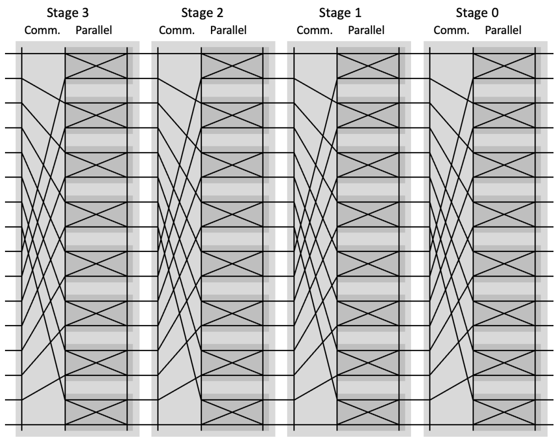

Fig. 2. Dataflow of the Pease FFT/NTT Algorithm.

Operator Language. SPIRAL represents linear transform algorithms and beyond using the Operator Language (OL) [27]. OL is a mathematical domain-specific language to describe structured divide-and-conquer algorithms for data-independent kernels, based on the Kronecker product formalism summarized in [28]–[30]. Here we provide a brief overview.

In SPIRAL, linear transforms are treated as matrix-vector multiplications. For example, the NTT definition (2) is viewed as the matrix-vector product, defined as

$$y = \mathrm{NTT}_n\, x, \quad \mathrm{NTT}_n = \left[\omega_n^{k\ell} \bmod p\right]_{0\le k,\ell<n}, \tag{3}$$

where ω is defined the same as in (2).

Using this *point-free* notation, we drop the explicit representation of x and y and consider NTT_n as the transform matrix that is implicitly multiplied with x. NTT algorithms can be expressed as factorizations of NTT_n. We denote the $n \times n$ *identity* matrix as I_n and the *butterfly* matrix as

$$\mathrm{NTT}_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}. \tag{4}$$

The *Kronecker product* of matrices A and B is defined as

$$A \otimes B = [a_{k,\ell} B], \qquad \text{for } A = [a_{k,\ell}], \tag{5}$$

which essentially replaces every entry of matrix A by the matrix $a_{k,l}B$. The *stride permutation* matrix L_m^{mn} permutes the elements of an input vector according to the following pattern:

$$in + j \mapsto jm + i, \quad 0 \le i < m, \quad 0 \le j < n. \tag{6}$$

Using OL, NTTs of size r^k have different representations according to different breakdown strategies (i.e., algorithms):

$$\mathrm{NTT}_{r^k} = \left(\prod_{i=0}^{k-1} (\mathrm{I}_{r^i} \otimes \mathrm{NTT}_r \otimes \mathrm{I}_{r^{k-i-1}})\, \mathrm{D}_i^{r^k}\right) \mathrm{R}_r^{r^k}, \tag{7}$$

$$\mathrm{NTT}_{r^k} = \mathrm{R}_r^{r^k} \left(\prod_{i=0}^{k-1} \mathrm{L}_{r^{k-1}}^{r^k}\, \mathrm{D}_i^{r^k} (\mathrm{NTT}_r \otimes \mathrm{I}_{r^{k-1}})\right), \tag{8}$$

$$\mathrm{NTT}_{r^k} = \left(\prod_{i=0}^{k-1} \mathrm{L}_r^{r^k}\, (\mathrm{I}_{r^{k-1}} \otimes \mathrm{NTT}_r)\, \mathrm{D}_i^{r^k}\right) \mathrm{R}_r^{r^k}, \tag{9}$$

which correspond to the Cooley-Tukey algorithm, the Korn-Lambiotte algorithm, and the Pease algorithm, respectively. Here, D is the twiddle factor diagonal matrix and R is the bit reversal permutation matrix.

V. NTTX System Walkthrough

NTTX C API. We developed an initial C API for NTTX that closely follows the FFTW [14] coding style and pattern, using a plan/execute paradigm. Listing 1 showcases single NTT/inverse NTT (iNTT) invocation while batch NTT/iNTT invocation is supported as well.

```c
// NTTX C API example: compute a single NTT
#include "nttx.h"

nttx_int n = NTT_SIZE;
nttx_uint modulus = NTT_MODULUS,
    in[NTT_SIZE],
    out[NTT_SIZE];

nttx_plan *p;

// initialize NTTX and plan
nttx_initialize(MY_NTTX_MODE);
p = nttx_plan_ntt(in, out, n, modulus,
    NTTX_FORWARD);
if (!p) exit(NTTX_ERROR);

// execute the plan
nttx_execute(p);

// cleanup
nttx_free(p);
nttx_shutdown();
```

Listing 1: NTTX C API for single NTT invocation.

Generator Script. When the user uses the NTTX frontend C API to plan an NTT, the code generation backend of NTTX starts with a script file written in the GAP programming language [31]. We wrote a SPIRAL generator script for high-performance NTT code for vector architectures, as shown in Listing 2 (simplified), which breaks down as follows:

Lines 2-5: Loading and importing the necessary SPIRAL domain-specific libraries, namely FFTX and NTTX. The NTTX package can be seen as an expansion of the FFTX package.

Lines 8-9: Setting up NTT size to be 4,096 and the target backend to be a vector ISA (discussed in Section VI).

Lines 12-18: Choosing whether forward or inverse NTT, setting the breakdown algorithms, switching on or off certain algorithmic optimizations.

Lines 21-29: Configuring the NTT given the above settings.

Lines 33-45: Declaring multiple variables that will be referenced in the code generation, depending on the NTT configuration. For example, forward NTT does not need the *cyclo* parameter.

Lines 49-59: Attaching the above settings and configurations to NTT.

Lines 62-66: Loading implementation options for NTT and tagging the NTT with hardware features and constraints.

Lines 69-72: Code generation for NTT. SPIRAL will take in all the information from the algorithm and the target hardware to find the best path to break the NTT down into smaller pieces using the breakdown rules (discussed next). There are multiple stages in the SPIRAL code generation backend that continuously apply optimizations and use backtrack search in the end to find the best NTT implementation.

Breakdown Rules. As discussed in Section II, SPIRAL explores the code implementation space through recursive expansions of the transform of size n by applying breakdown rules (e.g., (7)-(9)). The choice of the specific breakdown strategy can be guided by heuristics or performance feedback

```
1  // load SPIRAL FFTX and NTTX package
2  Load(fftx);
3  ImportAll(fftx);
4  Load(nttx);
5  ImportAll(nttx);
6
7  // NTT size and target ISA
8  n := 4096;
9  isa := B1024x128i;
10
11 // algorithmic settings
12 fwd := true;
13 useBarrettMult := false;
14 useIter := false;
15 useCT := false;
16 usePease := true;
17 useShuffle := true;
18 useTwiddleGen := true;
19
20 // NTT configuration
21 conf := LocalConfig.nttx.simdBigIntConf(
22     rec(useBarrettMult := useBarrettMult,
23         useIter := useIter,
24         usePease := usePease,
25         useCT := useCT,
26         useShuffle := useShuffle,
27         useTwiddleGen := useTwiddleGen,
28         isa := isa,
29         fwd := fwd));
30
31 // declare variables based on
32 // the NTT configuration
33 vlen := isa.v;
34 name := When(fwd, "ntt", "intt")::StringInt(n)
35     ::"x"::StringInt(vlen)::When(useBarrettMult,
36     "bmul", "")::"_b1024";
37 p := var("modulus", conf.type());
38 if not fwd then cyclo :=
39     var("cyclo", conf.type()); fi;
40 if useBarrettMult then mu :=
41     var("mu", conf.type()); fi;
42 twiddles := var("twiddles", TPtr(conf.type()));
43
44 // declare the transform as NTT
45 ntt := When(fwd, NTT, iNTT);
46
47 // the transform carries a record of
48 // settings and configurations
49 twrec := CopyFields(
50     rec(n := n, modulus := p,
51         twiddles := twiddles),
52         When(fwd, rec(), rec(cyclo := cyclo)),
53         When(useBarrettMult,
54             rec(mu := mu), rec()));
55 funcrec := CopyFields(
56     rec(abstractType := conf.type(),
57     fname := name, params := [p, twiddles]
58     ::When(fwd, [],[cyclo])
59     ::When(useBarrettMult, [mu], [])), twrec);
60
61 // load NTT options
62 opts := conf.getOpts(t);
63
64 // tag the tranform with hardware
65 // features and constraints
66 tt := opts.tagIt(t);
67
68 // multi-stage code generation for NTT
69 c := opts.genNttx(tt);
70
71 // output final code
72 opts.prettyPrint(c);
```

Listing 2: SPIRAL script for generating high-performance NTT code for vector architecture.

loops. Listing 3 is the core of the breakdown rule implementation of the Pease NTT algorithm in SPIRAL.

```
children := (self, nt) >> let(
    ...
    i := Ind(N/vlen),
    l := Lambda(i, cond(eq(i, V(0)), V(1),
        vbcast(v, N/2^(j+1), N/2^(j+1)))),
    l.setDomain(N),
    List(ap, rdx -> [
        TICompose(j, LogInt(N, rdx),
            TCompose([
                TTensorI(NTT(2, nt.params[2]),
                    N/rdx, APar, AVec),
                VDiag(l, N)
            ])
        ).withTags(nt.getTags())
    ])
)
```

Listing 3: Core implementation of the Pease NTT algorithm breakdown rule in SPIRAL.

Generated Code. Listing 4 shows the SPIRAL-generated NTT code targeting RPU that corresponds to the SPIRAL script shown in Listing 2. As the generated code serves as an abstract layer to be converted into RPU ISA, we designed an API that facilitates communication between the host processor, kernel launcher, and the kernel itself, drawing inspiration from the CUDA framework. The host code is written in standard C. The launch code employs abstract low-level system libraries and built-in constructs to convert host-based C data structures into scratchpad-based data structures on RPU. Every generated C function can be mapped to the instruction provided by the ISA. While Listing 4 demonstrates 4,096-point NTT code, SPIRAL can generate NTT of any two-power sizes.

VI. Evaluation

Vector Architecture Setup. We evaluate SPIRAL's effectiveness at targeting custom vector architectures by generating NTT instructions for RPU [12] and its associated ISA, B1K (successor of B512). RPU is a multi-tile vector architecture that includes 64 128-bit vector registers, 64 128-bit scalar registers, a 64 MiB vector data memory (VDM), and a 4 MiB scalar data memory (SDM). It operates on fixed vector lengths of 1,024 elements to reduce the overhead of programmable computing and to allow for scalability in the architecture. The tile microarchitecture is designed for simplicity and efficiency by avoiding the complexity of caches, dynamic scheduling logic, and branch prediction. Instead, the microarchitecture relies on the compiler to handle scheduling and data movement at compile time. This makes RPU an ideal platform to demonstrate the capabilities of SPIRAL.

The B1K ISA is tailored to support both FHE operations and general-purpose programming through 28 instructions. We make use of its *Butterfly*, *Shuffle*, and strided *Load* instructions when generating NTT kernels, however, more general instructions exist for modular arithmetic, non-strided memory accesses, control logic, and inter-tile communication. RPU's frontend has three independent queues for compute, memory, and shuffle instructions. Once an instruction is

```
#include "b1024.h"

// NTT kernel
void _ntt4096x1024_b1024() {
    enter(OP_DEFAULT);
    _vload_1024x128i(REG_V64, REG_A3, 0);
    _vbroadcast_1024x128i(REG_V1, REG_A3, 1, 1);
    _vload_1024x128i(REG_V2, REG_A1, 32768);
    _vload_1024x128i(REG_V3, REG_A1, 0);
    _vbutterfly_1024x128i(REG_V4, REG_V5, REG_V1,
        REG_V2, REG_V3, REG_M1);
    _vunpacklo_1024x128i(REG_V6, REG_V4, REG_V5);
    _vunpackhi_1024x128i(REG_V7, REG_V4, REG_V5);
    _vbroadcast_1024x128i(REG_V8, REG_A3, 1, 1);
    _vload_1024x128i(REG_V9, REG_A1, 49152);
    ...
    _sload_128i(REG_S3, REG_A3, 16400);
    _vsmulmod_1024x128i(REG_V8, REG_V64,
        REG_S3, REG_M1);
    _vload_1024x128i(REG_V9, REG_A1, 49152);
    _vload_1024x128i(REG_V10, REG_A1, 32768);
    _vbutterfly_1024x128i(REG_V12, REG_V11,
        REG_V8, REG_V10, REG_V9, REG_M1);
    _vunpacklo_1024x128i(REG_V13, REG_V12,
        REG_V11);
    _vunpackhi_1024x128i(REG_V14, REG_V12,
        REG_V11);
    _sload_128i(REG_S3, REG_A3, 16416);
    _vsmulmod_1024x128i(REG_V15, REG_V64,
        REG_S3, REG_M1);
    _vbutterfly_1024x128i(REG_V17, REG_V16,
        REG_V15, REG_V13, REG_V6, REG_M1);
    _vstores_1024x128i(REG_A2, 0, REG_V17, 2);
    _vstores_1024x128i(REG_A2, 16, REG_V16, 2);
    _sload_128i(REG_S3, REG_A3, 16432);
    _vsmulmod_1024x128i(REG_V18, REG_V64,
        REG_S3, REG_M1);
    _vbutterfly_1024x128i(REG_V20, REG_V19,
        REG_V18, REG_V14, REG_V7, REG_M1);
    _vstores_1024x128i(REG_A2, 32768,
        REG_V20, 2);
    _vstores_1024x128i(REG_A2, 32784,
        REG_V19, 2);
    leave(OP_DEFAULT);
}

// host code
int ntt4096x1024_b1024(unsigned __int128 *Y,
    unsigned __int128 *X,
    unsigned __int128 modulus,
    unsigned __int128 *twiddles) {
    int i305;
    required_kernel(ntt4096x1024_b1024);
    load_once_from_dram(SCRATCH0, 0,
        twiddles,
        (4096*sizeof(unsigned __int128 )));
    load_from_dram(SCRATCH0,
        (4096*sizeof(unsigned __int128 )),
        X, (4*sizeof(__uint1024x128 )));
    set_desc(SCRATCH0, REG_A1,
        (4096*sizeof(unsigned __int128 )));
    ...
    // launch NTT kernel
    i305 = execute(PROC0, SCRATCH0,
        _ntt4096x1024_b1024);
    store_to_dram(SCRATCH0,
        ((4096*sizeof(unsigned __int128 ))
        + (4*sizeof(__uint1024x128 ))),
        Y, (4*sizeof(__uint1024x128 )));
    swappable_kernel(ntt4096x1024_b1024);
    return i305;
}
```

Listing 4: SPIRAL-generated 4,096-point vectorized NTT code for RPU.

queued, it can be executed in parallel with other instruction types without data hazards. This parallel execution through decoupled pipelines is crucial for achieving high performance with general-purpose processing by hiding much of the data movement latency. Using SPIRAL, we can target RPU through the B1K ISA without concerning ourselves with underlying microarchitectural details.

Correctness. We generated test inputs and outputs for various sizes of NTTs using OpenFHE, a popular open-source software library that provides implementations of FHE schemes [32]. We then built a C functional simulator that simulates all 28 RPU instructions by implementing its corresponding functionality in C. Listing 5 shows the C implementation of the unit-stride vector load in B1K.

```
void _vload_1024x128i(
    reg_v &register_to,
    reg_a register_from,
    unsigned int offset){
    int index = (offset / ESIZE);
    for (int i = 0; i < VLEN; i++){
        register_to.elements[i] =
            register_from.address[index + i];
    }
}
```

Listing 5: Implementaion of B1K's unit-stride vector load in C functional simulator.

Via the function simulator, all of the SPIRAL-generated forward and inverse vectorized NTTs with sizes ranging from 1,024 to 131,072 are verified against OpenFHE data.

Performance. While SPIRAL has already produced the correct vectorized NTT code that will naturally take advantage of RPU's vector processing power, we further tested SPIRAL's optimization capabilities by comparing the naive implementation with the SPIRAL-optimized code.

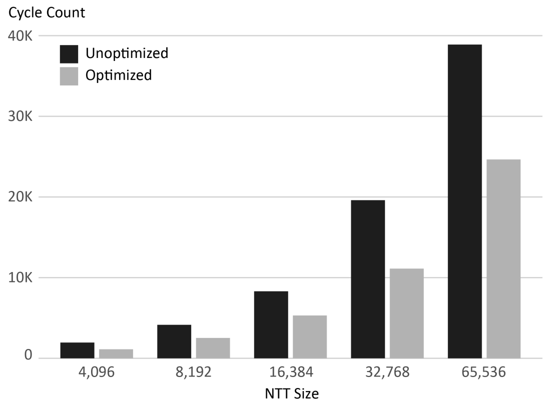

Fig. 3. Cycle count comparison of unoptimized and optimized NTT code.

We implemented a clock-cycle analyzer within our functional simulator to count the cycles of each NTT kernel and benchmarked NTT code from size 4,096 to 65,536. As shown in Fig. 3, SPIRAL-optimized NTT code is on average 1.7× faster than the unoptimized NTT implementation. The results demonstrate that SPIRAL can effectively take advantage of hardware-specific knowledge to schedule instructions and perform optimizations.

VII. Conclusion

The usage of FHE has been limited by significant overheads, rendering it impractical for many real-world applications. To address this challenge, we propose to use SPIRAL to generate high-performance NTT code on vector architectures. Throughout this paper, we identify suitable NTT algorithms namely the Korn-Lambiotte and the Pease algorithm, and translate their dataflow graphs into OL. We implement an end-to-end workflow that starts with user-friendly library C APIs, goes through SPIRAL scripts and breakdown rules, and produces optimized vectorized NTT code. We choose to target the vector accelerator designed for FHE computations, RPU, and its associated ISA, B1K. The results of our experiments demonstrate the effectiveness of our approach. This highlights the potential of leveraging SPIRAL to significantly improve the performance of NTT-based applications on different platforms, thereby overcoming the limitations imposed by the overheads for FHE applications.

References

[1] C. Gentry, *A fully homomorphic encryption scheme*. Stanford university, 2009.

[2] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," *ACM Transactions on Computation Theory (TOCT)*, vol. 6, no. 3, pp. 1–36, 2014.

[3] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," *Cryptology ePrint Archive*, 2012.

[4] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *International conference on the theory and application of cryptology and information security*, pp. 409–437, Springer, 2017.

[5] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Tfhe: fast fully homomorphic encryption over the torus," *Journal of Cryptology*, vol. 33, no. 1, pp. 34–91, 2020.

[6] L. Ducas and D. Micciancio, "Fhew: bootstrapping homomorphic encryption in less than a second," in *Annual international conference on the theory and applications of cryptographic techniques*, pp. 617–640, Springer, 2015.

[7] N. Samardzic, A. Feldmann, A. Krastev, N. Manohar, N. Genise, S. Devadas, K. Eldefrawy, C. Peikert, and D. Sanchez, "Craterlake: a hardware accelerator for efficient unbounded computation on encrypted data," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pp. 173–187, 2022.

[8] D. Takahashi, "An implementation of parallel number-theoretic transform using intel avx-512 instructions," in *Computer Algebra in Scientific Computing: 24th International Workshop, CASC 2022, Gebze, Turkey, August 22–26, 2022, Proceedings*, pp. 318–332, Springer, 2022.

[9] Ö. Özerk, C. Elgezen, A. C. Mert, E. Öztürk, and E. Savaş, "Efficient number theoretic transform implementation on gpu for homomorphic encryption," *The Journal of Supercomputing*, vol. 78, no. 2, pp. 2840–2872, 2022.

[10] T. Ye, R. Kannan, and V. K. Prasanna, "Fpga acceleration of fully homomorphic encryption over the torus," in *2022 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–7, IEEE, 2022.

[11] N. Samardzic, A. Feldmann, A. Krastev, S. Devadas, R. Dreslinski, C. Peikert, and D. Sanchez, "F1: A fast and programmable accelerator for fully homomorphic encryption," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 238–252, 2021.

[12] D. Soni, N. Neda, N. Zhang, B. Reynwar, H. Gamil, B. Heyman, M. Nabeel, A. Al Badawi, Y. Polyakov, K. Canida, *et al.*, "Rpu: The ring processing unit," in *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 272–282, IEEE, 2023.

[13] F. Franchetti, D. G. Spampinato, A. Kulkarni, D. T. Popovici, T. M. Low, M. Franusich, A. Canning, P. McCorquodale, B. Van Straalen, and P. Colella, "Fftx and spectralpack: A first look," in *2018 IEEE 25th International Conference on High Performance Computing Workshops (HiPCW)*, pp. 18–27, IEEE, 2018.

[14] M. Frigo and S. G. Johnson, "Fftw: An adaptive software architecture for the fft," in *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP'98 (Cat. No. 98CH36181)*, vol. 3, pp. 1381–1384, IEEE, 1998.

[15] F. Franchetti, T. M. Low, D. T. Popovici, R. M. Veras, D. G. Spampinato, J. R. Johnson, M. Püschel, J. C. Hoe, and J. M. Moura, "Spiral: Extreme performance portability," *Proceedings of the IEEE*, vol. 106, no. 11, pp. 1935–1968, 2018.

[16] M. Puschel, J. M. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, *et al.*, "Spiral: Code generation for dsp transforms," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 232–275, 2005.

[17] M. Püschel, F. Franchetti, and Y. Voronenko, "Spiral.," *Encyclopedia of parallel computing*, vol. 4, pp. 1920–1933, 2011.

[18] L. Meng, Y. Voronenko, J. R. Johnson, M. Moreno Maza, F. Franchetti, and Y. Xie, "Spiral-generated modular fft algorithms," in *Proceedings of the 4th International Workshop on Parallel and Symbolic Computation*, pp. 169–170, 2010.

[19] Y. Yang, S. R. Kuppannagari, R. Kannan, and V. K. Prasanna, "Nttgen: a framework for generating low latency ntt implementations on fpga," in *Proceedings of the 19th ACM International Conference on Computing Frontiers*, pp. 30–39, 2022.

[20] S. Rao, A. Kutuluru, P. Brouwer, S. McMillan, and F. Franchetti, "Gbtlx: A first look," in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–7, IEEE, 2020.

[21] N. Zhang, H. Gamil, P. Brinich, B. Reynwar, A. Al Badawi, N. Neda, D. Soni, K. Canida, Y. Polyakov, P. Broderick, *et al.*, "Towards full-stack acceleration for fully homomorphic encryption," *IEEE HPEC*, 2022.

[22] N. Zhang and F. Franchetti, "Generating number theoretic transforms for multi-word integer data types," *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2023.

[23] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," *Mathematics of computation*, vol. 19, no. 90, pp. 297–301, 1965.

[24] D. G. Korn and J. J. Lambiotte, "Computing the fast fourier transform on a vector computer," *Mathematics of computation*, vol. 33, no. 147, pp. 977–992, 1979.

[25] M. C. Pease, "An adaptation of the fast fourier transform for parallel processing," *Journal of the ACM (JACM)*, vol. 15, no. 2, pp. 252–264, 1968.

[26] F. Franchetti and M. Püschel, "Fast fourier transform," *Encyclopedia of Parallel Computing. Springer*, p. 51, 2011.

[27] F. Franchetti, F. d. Mesmay, D. McFarlin, and M. Püschel, "Operator language: A program generation framework for fast kernels," in *IFIP Working Conference on Domain-Specific Languages*, pp. 385–409, Springer, 2009.

[28] J. R. Johnson, R. W. Johnson, D. Rodriguez, and R. Tolimieri, "A methodology for designing, modifying, and implementing fourier transform algorithms on various architectures," *Circuits, Systems and Signal Processing*, vol. 9, pp. 449–500, 1990.

[29] R. Lu, *Algorithms for discrete Fourier transform and convolution*. Springer, 1989.

[30] C. Van Loan, *Computational frameworks for the fast Fourier transform*. SIAM, 1992.

[31] M. Schonert, "Gap-groups, algorithms and programming," *Lehrstul D Fur Matematik*, 1992.

[32] A. Al Badawi, J. Bates, F. Bergamaschi, D. B. Cousins, S. Erabelli, N. Genise, S. Halevi, H. Hunt, A. Kim, Y. Lee, *et al.*, "Openfhe: Open-source fully homomorphic encryption library," in *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, pp. 53–63, 2022.