Twiddle Factor Generation for a Vectorized Number Theoretic Transform

Patrick Brinich* Naifeng Zhang† Austin Ebel‡ Franz Franchetti† and Jeremy Johnson*
*Drexel University {pjb338, johnsojr}@drexel.edu
†Carnegie Mellon University {naifengz, franzf}@cmu.edu
‡New York University abe5240@nyu.edu

I. Introduction

Implementations of Fast Fourier Transforms often precompute some or all of the twiddle factors, trading space for efficient reuse and stability. Similar trade-offs arise when implementing Number Theoretic Transforms. We present an approach for generating twiddle factors for vectorized Number Theoretic Transforms using a small number of precomputed twiddle factors.

Contributions. Our key contributions are:

- A matrix factorization representing a Korn-Lambiotte [1] NTT with a formal description of its twiddle factors alongside a mathematical formulation for generating said twiddle factors from precomputed tables using $\mathcal{O}(\sqrt{n})$ space
- An implementation of the algorithm with twiddle factor generation using the SPIRAL [2] system for specialized hardware [3].

II. Approach

The Number Theoretic Transform (NTT) generalizes the Discrete Fourier Transform by working over a ring with roots of unity, such as the integers modulo a prime. Existing Fast Fourier Transforms can be extended to accommodate this new setting. In particular, the Korn-Lambiotte FFT is an efficient algorithm for large vector machines [1]. This algorithm can be generalized for a radix-r NTT of size $n = r^t$ parameterized on an n^{th} root of unity ω. Described as a matrix factorization in the style of Van Loan [4],

$$\mathrm{NTT}_{r^t} = \mathrm{R}_{r^t} \cdot \prod_{c=1}^{t} \mathrm{L}^{r^t}_{r^{t-1}} \left(\mathrm{NTT}_r \otimes \mathrm{I}_{r^{t-1}}\right) \mathrm{D}_{r^t,c}, \qquad (1)$$

where $\mathrm{L}^{r^t}_{r^{t-1}}$ denotes the perfect shuffle matrix and R_{r^t} denotes the r^t-sized base-r digit-reversal permutation. The diagonal matrix $\mathrm{D}_{r^t,c}$ holds the twiddle factors, which consist of repeated blocks of digit-reversed powers of ω. More formally,

$$\mathrm{D}_{r^t,c} = \bigoplus_{i=0}^{r-1} (\mathrm{I}_{r^{c-1}} \otimes \mathrm{R}_{r^{t-c}} \, \mathrm{W}^{ir^{c-1}}_{r^{t-c}} \, \mathrm{R}_{r^{t-c}}), \qquad (2)$$

with $\mathrm{W}_s = \mathrm{diag}(1, \omega, \ldots, \omega^{s-1})$.

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

The digit-reversed structure of the twiddle factors has a nice property for twiddle factor generation. For $1 \le c \le t$ and $0 \le d < c$,

$$\begin{gathered}\mathrm{R}_{r^{t-c+d}} \, \mathrm{W}^{r^{c-d-1}}_{r^{t-c+d}} \, \mathrm{R}_{r^{t-c+d}} = \\ \mathrm{R}_{r^{t-c}} \, \mathrm{W}^{r^{c-1}}_{r^{t-c}} \, \mathrm{R}_{r^{t-c}} \otimes \mathrm{R}_{r^d} \, \mathrm{W}^{r^{c-1-d}}_{r^d} \, \mathrm{R}_{r^d}\end{gathered} \qquad (3)$$

Consequently, twiddle factors from later stages (lower values of c) can be generated from earlier stages, and twiddle factors from earlier stages are contained within later stages.

For $r = 2$, the unique twiddle factors needed throughout the algorithm are the first 2^{t-1} powers of ω. At stage $c = 1$, the second half of the twiddle factors are these powers in bit-reversed order. Earlier stages of c requires the first 2^{t-c} of these unique twiddle factors repeated 2^{c-1} times. Table I shows this pattern for $t = 5$. In this example, the twiddle

Twiddle Factor	c=1	c=2	c=3	c=4	c=5
0—15	1	1	1	1	1
16	ω^0	ω^0	ω^0	ω^0	ω^0
17	ω^8	ω^8	ω^8	ω^8	ω^0
18	ω^4	ω^4	ω^4	ω^0	ω^0
19	ω^{12}	ω^{12}	ω^{12}	ω^8	ω^0
20	ω^2	ω^2	ω^0	ω^0	ω^0
21	ω^{10}	ω^{10}	ω^8	ω^8	ω^0
22	ω^6	ω^6	ω^4	ω^0	ω^0
23	ω^{14}	ω^{14}	ω^{12}	ω^8	ω^0
24	ω^1	ω^0	ω^0	ω^0	ω^0
25	ω^9	ω^8	ω^8	ω^8	ω^0
26	ω^5	ω^4	ω^4	ω^0	ω^0
27	ω^{13}	ω^{12}	ω^{12}	ω^8	ω^0
28	ω^3	ω^2	ω^0	ω^0	ω^0
29	ω^{11}	ω^{10}	ω^8	ω^8	ω^0
30	ω^7	ω^6	ω^4	ω^0	ω^0
31	ω^{15}	ω^{14}	ω^{12}	ω^8	ω^0

TABLE I
Twiddle factors for each stage c in (1) for $r = 2$ and $t = 5$.

factors for any stage can be generated from stage $c = 3$. For later stages, this requires some additional factors. In particular, to generate the twiddles for $c = 1$ requires a tensor product:

$$\mathrm{R}_{2^{16}} \, \mathrm{W}_{2^{16}} \, \mathrm{R}_{2^{16}} = \begin{bmatrix} \omega^0 \\ \omega^8 \\ \omega^4 \\ \omega^{12} \end{bmatrix} \otimes \begin{bmatrix} \omega^0 \\ \omega^2 \\ \omega^1 \\ \omega^3 \end{bmatrix}.$$

The twiddle factors for $c = 2$ require fewer additional factors, but more repetition:

$$\mathrm{I}_2 \otimes \mathrm{R}_{2^8}\, \mathrm{W}_{2^8}\, \mathrm{R}_{2^8} = \begin{bmatrix} \omega^0 \\ \omega^8 \\ \omega^4 \\ \omega^{12} \end{bmatrix} \otimes \begin{bmatrix} \omega^0 \\ \omega^2 \\ \omega^0 \\ \omega^2 \end{bmatrix}.$$

For stages 4 and 5, no additional twiddle factors need to be generated, they require only a repeated subsequence of unique twiddle factors from $c = 3$.

To be able to generate any twiddle factors in the $r = 2$ case, implementations of (1) must pick a stage c and precompute two tables. The first table, the seed table, contains the diagonal entries of $\mathrm{R}_{2^{t-c}}\, \mathrm{W}_{2^{t-c}}^{2^{c-1}}\, \mathrm{R}_{2^{t-c}}$, and the second table contains the diagonal entries of $\mathrm{R}_{2^{c-1}}\, \mathrm{W}_{2^{c-1}}\, \mathrm{R}_{2^{c-1}}$. These tables require $2^{t-c} + 2^{c-1}$ precomputed factors. Picking a c close to $\lceil t/2 \rceil$ thus requires only $2\sqrt{n/2}$ factors.

For a large vector machine, generation of twiddle factors occurs in stages after stage c and requires at most three vector operations. First, a vector containing the necessary repeated subsequence is loaded from the the first table. Another vector is loaded similarly from the second table. Multiplying these two vectors produces the required twiddle factor vector. Implementations for machines with shorter vectors may load a base vector from the seed table and perform scalar multiplication with a single entry from the second table.

III. Results

We implemented the twiddle generation optimization technique in SPIRAL's NTTX package, aiming to reduce the data transfer between the main memory and the on-chip memory for a fully homomorphic encryption (FHE) accelerator [3]. The NTTX package leverages SPIRAL's capability to automatically generates high-performance NTT code for various algorithmic settings and hardware specifications. Listing 1 showcases SPIRAL-generated $4,096$-point radix-2 NTT code with twiddle generation. We load the base vector (where the seed table is stored) on line 5 and multiply it with another seed vector to compute all needed twiddle factors on the fly (e.g., line 21-22).

Real-world FHE applications require NTTs of size up to 2^{17}. For a 2^{17}-point NTT, using on-the-fly twiddle generation reduces the the number of twiddle factors loaded from the main memory from $131,072$ to $1,152$, which is less than 1% of the previous data I/O.

IV. Conclusion

Using a mathematical treatment of the Korn-Lambiotte n-point NTT algorithm and its twiddle factors, we provide an approach to generating the twiddle factors troughout the algorithm from a pair of tables requiring $\mathcal{O}(\sqrt{n})$ space. Making use of this approach, we discuss an implementation using this approach for 4096-point and 2^{17}-point radix-2 NTTs on an hardware accelerator, achieving reduced data I/O from a previous implementation. We consider adapting this approach to higher radix NTT implementations and implementations on other hardware avenues for future work.

```
1  #include "b1.h"
2
3  void _ntt4096x1024_b1() {
4      enter(OP_DEFAULT);
5      _vload_1024x128i(REG_V64, REG_A3, 0);
6      _vbroadcast_1024x128i(REG_V1, REG_A3, 1, 1);
7      _vload_1024x128i(REG_V2, REG_A1, 32768);
8      _vload_1024x128i(REG_V3, REG_A1, 0);
9      _vbutterfly_1024x128i(REG_V4, REG_V5, REG_V1,
10         REG_V2, REG_V3, REG_M1);
11     _vunpacklo_1024x128i(REG_V6, REG_V4, REG_V5);
12     _vunpackhi_1024x128i(REG_V7, REG_V4, REG_V5);
13     _vbroadcast_1024x128i(REG_V8, REG_A3, 1, 1);
14     _vload_1024x128i(REG_V9, REG_A1, 49152);
15     _vload_1024x128i(REG_V10, REG_A1, 16384);
16     _vbutterfly_1024x128i(REG_V11, REG_V12, REG_V8,
17         REG_V9, REG_V10, REG_M1);
18     _vunpacklo_1024x128i(REG_V13, REG_V11, REG_V12);
19     _vunpackhi_1024x128i(REG_V14, REG_V11, REG_V12);
20     ...
21     _sload_128i(REG_S3, REG_A3, 16400);
22     _vsmulmod_1024x128i(REG_V8, REG_V64,
23         REG_S3, REG_M1);
24     _vload_1024x128i(REG_V9, REG_A1, 49152);
25     _vload_1024x128i(REG_V10, REG_A1, 32768);
26     _vbutterfly_1024x128i(REG_V12, REG_V11, REG_V8,
27         REG_V10, REG_V9, REG_M1);
28     _vunpacklo_1024x128i(REG_V13, REG_V12, REG_V11);
29     _vunpackhi_1024x128i(REG_V14, REG_V12, REG_V11);
30     _sload_128i(REG_S3, REG_A3, 16416);
31     _vsmulmod_1024x128i(REG_V15, REG_V64,
32         REG_S3, REG_M1);
33     _vbutterfly_1024x128i(REG_V17, REG_V16, REG_V15,
34         REG_V13, REG_V6, REG_M1);
35     _vstores_1024x128i(REG_A2, 0, REG_V17, 2);
36     _vstores_1024x128i(REG_A2, 16, REG_V16, 2);
37     _sload_128i(REG_S3, REG_A3, 16432);
38     _vsmulmod_1024x128i(REG_V18, REG_V64,
39         REG_S3, REG_M1);
40     _vbutterfly_1024x128i(REG_V20, REG_V19, REG_V18,
41         REG_V14, REG_V7, REG_M1);
42     _vstores_1024x128i(REG_A2, 32768, REG_V20, 2);
43     _vstores_1024x128i(REG_A2, 32784, REG_V19, 2);
44     leave(OP_DEFAULT);
45 }
```

Listing 1: SPIRAL-generated radix-2 4,096-point NTT code with twiddle generation using intrinsics for a FHE accelerator.

References

[1] 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.

[2] 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.

[3] 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.

[4] C. Van Loan, *Computational Frameworks for the Fast Fourier Transform*. USA: Society for Industrial and Applied Mathematics, 1992.