NTT Mini: Exploring Winograd's Heuristic for Faster NTT

Carol Danvers
carol@ingonyama.com

Abstract

We report on the Winograd-based implementation for the Number Theoretical Transform. It uses less multiplications than the better-known Cooley-Tuckey alternative. This optimization is important for very high order finite-fields. Unfortunately, the Winograd scheme is difficult to generalize for arbitrary sizes and is only known for small-size transforms. We open-source our hardware implementation for size 32 based on [1].

1 Motivation

Zero Knowledge Proofs (ZKP) rely on a small number of computationally intensive primitives such as Multi Scalar Multiplication (MSM) and Number Theoretic Transform (NTT). The acceleration of these primitives is a necessary enabler for global adoption of these technologies. In [2], we discussed MSM. The focus of this note is NTT.

NTT is a generalization of the Discrete Fourier Transform (DFT) for finite-fields. Being a linear transform, it can be written in a matrix form

$$\vec{y} = \mathbf{F}\,\vec{x} \tag{1}$$

The transform matrix $\mathbf{F}$ has a special form:

$$\mathbf{F} = \begin{pmatrix} 1 & 1 & 1 & \dots \\ 1 & \omega_N & \omega_N^2 & \dots \\ 1 & \omega_N^2 & \omega_N^4 & \dots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix} \tag{2}$$

where ω_N is the root-of-unity of order N and N is the transform length.

Multiplication is computationally expensive, so our goal is to minimize the number of multiplications, (this comes at the expense of more additions). For an arbitrary full-rank matrix of size N, computing (1) costs N^2 multiplications. The special form of the NTT matrix $\mathbf{F}$ allows factorization to a product of sparse matrices where many of the non-zero elements are ± 1.

The Cooley-Tuckey (CT) factorization can be applied recursively for any N depending on its prime factorization. For N that is a power of a small prime, CT achieves a transform cost of $N \log N$ multiplications. Of particular interest is N that is a power of 2.

The Winograd factorization, discussed here, is a more efficient factorization, reducing $\mathbf{F}$ to a product of sparse matrices with only ± 1's, and a single diagonal matrix with non-trivial values. The rank r of the diagonal matrix is always $N \leq r < N \log N$. For Winograd r is actually the number of required multiplications. By definition, the Winograd factorization is not limited by N, though it is not recursive and only known for some small values of N. Below is a table comparing CT to Winograd for $N = 16, 32$.

Size	CT	Winograd
16	17	13
32	49	40

Table 1: Number of multiplications for different transform sizes

2 Theoretical Background

Winograd, much like CT, can be presented both as a series of recursive steps and as a factorization of the DFT matrix. The symmetries in the DFT matrix allow elegant factorization.

2.1 Notation

Define the following notations used hereafter:

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \tag{3}$$

$$A \otimes B = \begin{pmatrix} a_{11}B & a_{12}B \\ a_{21}B & a_{22}B \end{pmatrix} \tag{4}$$

$$A \oplus B = \begin{pmatrix} A & 0 \\ 0 & B \end{pmatrix} \tag{5}$$

Additionally, let us denote by $\mathbf{M}_n$ the matrix of dimension 2^n.

2.2 Cooley-Tuckey Factorization

We follow the presentation of the factorization from [3].

Theorem 2.1 (Cooley-Tuckey Factorization) *The $2^n \times 2^n$ DFT matrix F_n can be factored as:*

$$\mathbf{F}_n = \mathbf{P}_n \mathbf{A}_n^{(0)} ... \mathbf{A}_n^{(n-1)} \tag{6}$$

where, for $k = 0, ..., n-1$:

$$\mathbf{A}_n^{(k)} = \mathbf{I}_{n-k-1} \otimes \mathbf{B}_{k+1} \tag{7}$$

$$\mathbf{B} = (\mathbf{I}_k \oplus \mathbf{\Omega}_k)(H \otimes \mathbf{I}_k) \tag{8}$$

$$\mathbf{\Omega}_k = \mathrm{diag}(w_{2^{k+1}}^0, ..., w_{2^{k+1}}^{2^k-1}) \tag{9}$$

and $\mathbf{P}_n$ is a bit reversal permutation that satisfies:

$$\mathbf{P}_n(v_1 \otimes ... \otimes v_n) = v_n \otimes ... \otimes v_1 \tag{10}$$

Notice that $\mathbf{A}_n^{(k)}$ is a sparse matrix, containing only 2 non-zero entries in each row. Thus, multiplying a vector by this matrix can be done in $O(2^n)$ time, which is linear in the dimension 2^n. The number of $\mathbf{A}_n^{(k))}$ matrices in the factorization is n, which is logarithmic in the dimension 2^n. Thus, this leads to a total of $O(n \cdot 2^n)$ time (or $N \log N$ for $N = 2^n$).

2.2.1 Winograd Factorization

Winograd utilizes different symmetries in the DFT matrix.

Theorem 2.2 (Winograd's Heuristic Factorization) *The $2^n \times 2^n$ DFT matrix F_n can be factored as:*

$$\mathbf{F}_n = (H_2 \cdot \mathbf{I}_{n-1})(\mathbf{F}_n \oplus \mathbf{Q}_{n-1}\mathbf{P}_n^{\pi_n}) \tag{11}$$

where:

$$\mathbf{P}_n^{\pi_n} = \begin{pmatrix} \mathbf{I}_{n-1} \otimes \mathbf{\Psi}_{2\otimes 4} \\ \mathbf{I}_{n-1} \otimes \mathbf{\Psi}_{2\otimes 4}\mathbf{I}_{n-1}^{1\rightarrow} \end{pmatrix} \tag{12}$$

$$\mathbf{\Psi}_{2\otimes 4} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{13}$$

$\mathbf{Q}_{n-1}$ is a prefix matrix, and $\mathbf{I}_{n-1}^{1\rightarrow}$ is is the matrix obtained from the identity matrix by shifting its columns by one position to the right.

Intuitively, the main goal of this factorization is to decompose the DFT matrix as follows:

$$\mathbf{F}_n = \mathbf{M}_1 \cdot ... \cdot \mathbf{M}_k \cdot \mathbf{D} \cdot \mathbf{M}_{k+1} ... \mathbf{M}_n \tag{14}$$

where $\mathbf{M}_i$ is a sparse matrix consisting of ± 1 entries, and $\mathbf{D}$ is a diagonal matrix (typically of dimension greater than 2^n).

The strategy of [1] is to gradually decompose the matrices $\mathbf{F}_n, \mathbf{Q}_n$ by utilizing the above theorem, as well as several permutations and the following decomposition rules that capitalize on inherent symmetries in each of these matrices.

Claim 2.3 *For $n \times n$ matrices A, B, C, the following holds.*

$$\begin{pmatrix} A & B \\ A & -B \end{pmatrix} = (H_2 \otimes I_n)(A \oplus B) \tag{15}$$

$$\begin{pmatrix} A & A \\ B & -B \end{pmatrix} = (A \oplus B)(H_2 \otimes I_n) \tag{16}$$

$$\begin{pmatrix} A & B \\ B & A \end{pmatrix} = \frac{1}{2}(H_2 \otimes I_n)((A + B) \oplus (A - B))(H_2 \otimes I_n) \tag{17}$$

$$\begin{pmatrix} A & B \\ B & -A \end{pmatrix} = \frac{1}{2}(T \otimes I_n) \begin{pmatrix} A - B & 0 & 0 \\ 0 & -(A + B) & 0 \\ 0 & 0 & B \end{pmatrix} (T \otimes I_n) \tag{18}$$

$$\begin{pmatrix} A & B \\ C & A \end{pmatrix} = (Q \otimes H \otimes I_{n-1}) \begin{pmatrix} C - A & 0 & 0 \\ 0 & B - A & 0 \\ 0 & 0 & A \end{pmatrix} (T \otimes H \otimes I_{n-1}) \tag{19}$$

where

$$T = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{pmatrix} \tag{20}$$

$$Q = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix} \tag{21}$$

Using these rules, the authors of [1] managed to decrease the dimension of the factorization by 2. The caveat of this scheme is the non-uniformity of the factorization of $\mathbf{Q}_n$ requiring a well-designed permutation to exploit its symmetries.

3 Results

This note summarizes our initial experience with Winograd factorizations of size $N = 2^n$, based on the derivations of [1]. We used Symbolic Algebra System (SAS) tools to automate the ad-hoc factorizations for arbitrary finite-fields. We outline our workflow as follows:

1. SAS code generates C++ template. The template captures the structure of size 2^n transform only, and does not depend on specific finite field selection. It implements
   (a) sparse matrix multiplication (14), and
   (b) diagonal matrix $\mathbf{D}$ computation
2. C++ template is instantiated for a specific finite field as C++ code
3. C++ code is compatible with High Level Synthesis EDA tools, that eventually produce RTL (Verilog, VHDL), and, finally, FPGA bitstreams or GL1 for ASICs.

We open-source a C++ template for NTT of size $2^5 = 32$ together with a C++ instance for the scalar finite field of the Elliptic Curve *bn254*. Template ntt32_winograd uses arbitrary precision unsigned integers to represent the elements of the finite field. Specifically, we use type template ap_uint<W> from Xilinx' Vitis HLS toolchain [4]. The template takes the vector $\vec{x} = (x0, \ldots, x31)$ as input and returns the output $\vec{y} = (y0, \ldots, y31)$ by reference:

```
template<int W> void ntt32_winograd(
    const ap_uint<W> x0, ... , const ap_uint<W> x31,
    ap_uint<W>* y0, ..., ap_uint<W>* y31
) {...}
```

All finite field matrix operations are unrolled and call scalar functions basic_add_mod(), basic_sub_mod() and mult_red(), which are specific for the selected finite field.

The template is instantiated in function ntt32_winograd_bn254_scalar

```
void ntt32_winograd_bn254_scalar(
    const ap_uint<254> x0,..., const ap_uint<254> x31,
    ap_uint<254>* y0,..., ap_uint<254>* y31,
    )
    {
        ntt32_winograd(x0,..., x31, y0,...,y31);
        return;
    }
```

The header file ntt32_winograd_bn254_scalar.hpp declares the above instance and optimized finite field operations for scalar bn254 field.

When using our example, this header is the only file to include.

```
#include "ntt32_winograd_bn254_scalar.hpp"
...
// define input vector x
...
ntt32_winograd_bn254_scalar(x0,..., x31, &y0,...,&y31);
// use output vector y
...
```

4 Usage

1. Make sure you have g++ toolchain installed.

2. Make sure you have Xilinx Vitis installed. Environment variable XILINX_HLS should be defined and point to the distribution. This will allow the toolchain to find Xilinx's arbitrary precision headers.

3. Make sure you have C++ Boost library [5]. Environment variable BOOST should point to the installation. We need this library for tests only.

4. Download our code from here [6]
5. Run make test

5 Future Directions

In this note, we demonstrated the Winograd factorization only for small degree polynomials. Real-world instances of Zero Knowledge Proofs and Fully Homomorphic Encryption require higher degree polynomial arithmetic, with common sizes of N reaching 2^{15} and often higher. There are various interesting directions to proceed. One is to extend the work of [1], finding the Winograd factorization for specific power-of-two N's larger than 32, potentially discovering a closed-form extendable expression. A second, more immediate, is to utilize the recursive structure of NTT, together with the small-size Winograd building-blocks, to extend the construction to higher N's similarly to what was done by CT.

Our HLS code correctness was verified in C++ and Verilog simulations. We will soon integrate it as part of our Cloud-ZK dev-kit [7], enabling developers to integrate with a fast NTT implementation running on AWS F1 FPGA instances.

We hope that our implementation will lead to a better intuition on the complexity of Winograd. The number of multiplications, dominating the computation time is behaving as $\mathcal{O}(N)$. We think it will be interesting to compare the theoretical complexity to concrete measurements. In general, Winograd takes more additions than CT, which might become a non-negligible factor in total running time.

References

[1] Mateusz Raciborski and Aleksandr Cariow. On the derivation of winograd-type dft algorithms for input sequences whose length is a power of two. *Electronics*, 11:1342, 04 2022.

[2] Charles F Xavier. Pipemsm: Hardware acceleration for multi-scalar multiplication. *Cryptology ePrint Archive*, 2022.

[3] Daan Camps, Roel Van Beeumen, and Chao Yang. Quantum fourier transform revisited. *Numerical Linear Algebra with Applications*, 28(1), sep 2020.

[4] Vitis high-level synthesis user guide (ug1399). `https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Overview-of-Arbitrary-Precision-Integer-Data-Types`.

[5] C++ boost library. `https://www.boost.org/`.

[6] Winograd ntt32 code. `https://github.com/ingonyama-zk/ntt_winograd.git`.

[7] Cloud zk. `https://github.com/ingonyama-zk/cloud-ZK`.