OpenNTT: An Automated Toolchain for Compiling High-Performance NTT Accelerators in FHE

Florian Krieger
Graz University of Technology
Graz, Austria

Florian Hirner
Graz University of Technology
Graz, Austria

Ahmet Can Mert
Graz University of Technology
Graz, Austria

Sujoy Sinha Roy
Graz University of Technology
Graz, Austria

Abstract

Modern cryptographic techniques such as fully homomorphic encryption (FHE) have recently gained broad attention. Most of these cryptosystems rely on lattice problems wherein polynomial multiplication forms the computational bottleneck. A popular method to accelerate these polynomial multiplications is the Number-Theoretic Transformation (NTT). Recent works aim to improve the practical deployability of NTT and propose toolchains supporting the NTT hardware accelerator design processes. However, existing design tools do not provide on-the-fly twiddle factor generation (TFG) which leads to high memory demands. Inspired by this situation, we present OpenNTT, a fully automated, open-source framework to compile NTT hardware accelerators with TFG for various NTT types and parameter sets. We address the challenge of combining conflict-free memory accesses and efficient, linear twiddle factor generation through a dedicated NTT processing order. Following this order, we develop a flexible twiddle factor generation method with minimal memory usage. These core concepts together with a frequency-optimized hardware architecture form our OpenNTT framework. We use OpenNTT to compile and test NTT hardware designs with various parameter sets on FPGAs. The obtained results show a clear memory reduction due to TFG and a speedup by 2.7× in latency and 2.2× in area-time-product, compared to prior arts.

Keywords

Hardware Design Tool, Number Theoretic Transformation (NTT), Twiddle Factor Generation, Homomorphic Encryption (FHE)

1 Introduction

The advent of new cryptographic techniques such as post-quantum cryptography (PQC) [8, 3], fully homomorphic encryption (FHE) [6, 5], or zero-knowledge proofs (ZKP) [29] demands a performant and efficient deployment. One widely used method to meet these demands is the Number-Theoretic Transformation (NTT). The NTT accelerates the computationally expensive polynomial multiplications required in the arising FHE, PQC, and ZKP schemes. Polynomial multiplication without NTT shows a runtime of $\mathcal{O}(N^2)$, where N is the polynomial degree. In contrast, NTT reduces the runtime to $\mathcal{O}(N \log N)$ [17]. The lower algorithmic complexity already fosters degree $N = 2^8$ polynomial multiplication as used in PQC, but becomes highly essential for $N = 2^{14}$ to 2^{24}-degree polynomials involved in FHE and ZKP.

Although NTT reduces the algorithmic complexity of polynomial multiplication, this computation step remains the clear bottleneck in FHE schemes. Hence, it is required to further accelerate NTT which poses a highly active field in research. Different acceleration approaches have been proposed including optimized software solutions [7, 26] as well as FPGA and AISC designs [15, 18, 22, 28]. Many of the existing works target one specific use case of NTT by only supporting a limited parameter set or a single NTT type. Extending these designs to other parameter sets requires significant engineering effort and know-how, which leads to long design times for NTT hardware accelerators. Hence, latest research focuses on providing frameworks to compile NTT hardware designs, thereby supporting a broader applicability of NTT accelerators.

However, we observe that prior NTT hardware compilation works rely on stored twiddle factors. The twiddle factors are kept in memory and fetched on-demand during NTT transformations. Yet, the memory required to store twiddle factors increases linearly with the polynomial degree N and the modulus size $\log(q)$. This leads to a significant memory overhead in applications with large N and q such as in FHE or ZKP. Furthermore, twiddle factors also depend on the prime q and need to be replaced each time q changes, which is a frequent scenario in FHE applications. Hence, supporting multiple primes introduces additional overheads in on-chip memory or off-chip bandwidth.

In contrast to storing twiddle factors, they can also be generated on the fly. This on-the-fly twiddle factor generation (TFG) is a highly relevant alternative as it significantly reduces the on-chip memory at the cost of additional logic resources. The logic overhead however does not increase with the polynomial degree or the number of supported primes. This makes TFG an attractive choice for various large-degree NTTs as used in FHE. Yet, TFG introduces additional constraints to the transformation flow as twiddle factors cannot be generated in an arbitrary order. These constraints pose a noteworthy challenge in the context of parameter flexibility. This challenge has not been sufficiently addressed in prior work leading to a lack of NTT compilation tools with twiddle factor generation.

In this paper, we address this limitation and present OpenNTT, a fully automated framework to compile efficient NTT architectures for various parameter sets. OpenNTT relies on on-the-fly twiddle factor generation and hence significantly reduces memory consumption. We combine the TFG and the flexibility in parameters through a generic algorithm that unifies efficient TFG and

This project was supported in part by the State Government of Styria, Austria – Department Zukunftsfonds Steiermark.

collision-free NTT transformations. Furthermore, our framework is designed to be user-friendly. Users just provide the desired parameter set while the tool overtakes all instantiation, configuration, and testbench creation steps. Hence, OpenNTT fully automates the hardware generation flow leading to a faster design process.

Compared to other NTT accelerators, the hardware assembled with OpenNTT reaches higher clock frequencies and, hence, up to 2.7× lower latency. Furthermore, OpenNTT shows a 2.2× improvement in area-time product which makes our framework an attractive alternative for rapid NTT hardware accelerator design. Our main contributions are listed as follows:

- We give a comprehensive analysis of the required processing order and twiddle factor order in NTT transformations. Based on this analysis, we explain the challenges of efficient TFG in fully pipelined and conflict-free NTT designs.
- We propose a generic algorithm that compiles twiddle factor generation modules for conflict-free NTT designs. Our algorithm is flexible in various design parameters such as NTT type, polynomial degree, number of processing elements, multiplier latency, and number of supported primes.
- We present OpenNTT, a user-friendly framework to rapidly create efficient and performant NTT designs for hardware acceleration. OpenNTT incorporates our generic TFG algorithm and delivers optimized NTT architectures. The compiled architectures satisfy a wide range of applications covering FHE, PQC, and ZKP demands.
- We make our tool, OpenNTT, and all its source code publically available on GitHub [21]. This aims to support the NTT deployment and further research in the field.

The remainder of this paper is structured as follows. Sec. 2 gives the background of the Number-Theoretic Transformation. Sec. 3 explains our algorithm to generically compile on-the-fly generation of twiddle factors for various NTT types. Sec. 4 highlights our overall hardware design flow. Finally, Sec. 5 presents implementation results and comparisons. Sec. 6 concludes the paper.

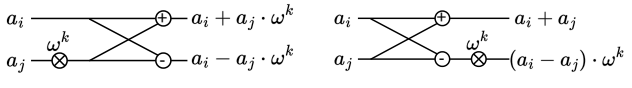

Figure 1: CT butterfly (left) and GS butterfly (right).

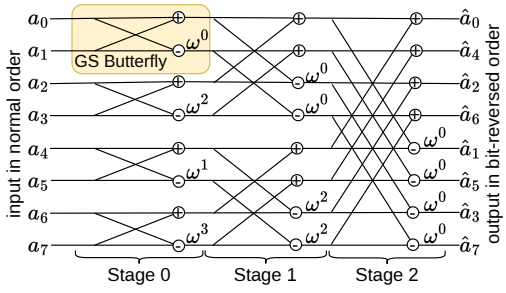

Figure 2: Data flow for a $N = 8$ point NR DIF NTT.

2 Background

2.1 Number-Theoretic Transformation

The Number-Theoretic Transformation (NTT) is a variant of the Discrete Fourier Transformation over the polynomial ring $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^N + 1)$. This polynomial ring is the set of polynomials $\mathbf{a}(x) = a_{N-1}x^{N-1} + \ldots + a_1 x + a_0$ with degree less than N. The coefficients $a_i \in \mathbb{Z}_q$ are integers modulo a prime q, which satisfies $q \equiv 1 \bmod N$. The NTT takes a polynomial $\mathbf{a}(x)$ in *coefficient* representation as input and maps it to its *evaluation* representation $\hat{\mathbf{a}}(x) = \text{NTT}(\mathbf{a}(x))$ following Eq. (1). The variable ω is a N-th primitive root of unity in $\mathbb{Z}_q$, i. e. $\omega^i \neq 1 \bmod q$ for $0 < i < N$ and $\omega^N = 1 \bmod q$. The powers of ω are called twiddle factors and are essential for NTT. The inverse NTT transformation of $\mathbf{a}(x) = \text{INTT}(\hat{\mathbf{a}}(x))$ is defined as in Eq. (2).

$$\hat{a}_k = \sum_{i=0}^{N-1} a_i \omega^{ik} \mod q. \tag{1}$$

$$a_i = N^{-1} \sum_{k=0}^{N-1} \hat{a}_k \omega^{-ik} \mod q. \tag{2}$$

Two approaches can be used to efficiently compute the NTT, namely decimation in time (DIT) and decimation in frequency (DIF) [13]. DIT internally uses the Cooley-Tukey (CT) butterfly operation while DIF uses the Gentleman-Sande (GS) butterfly. Fig. 1 shows the two butterfly configurations. Each butterfly operation takes two coefficients of one polynomial (a_i and a_j) and one twiddle factor (ω^k) as input and returns two output coefficients, as shown in Fig. 1. These butterfly operations form an NTT transformation as shown in Fig. 2, wherein $N/2$ butterfly operations are performed in each of the $S = \log_2(N)$ stages. In NTT hardware designs, one or multiple butterfly operations can performed concurrently by instantiating multiple processing elements (PE). We refer to the number of processing elements as n_{PE}.

There exist two commonly used coefficient orders for DIT and DIF NTT. First, the input polynomial $\mathbf{a}(x)$ is in normal order leading to a bit-reversed output polynomial $\hat{\mathbf{a}}(x)$, as shown in Fig. 2. Conversely, in the second case, the input polynomial is in bit-reversed order leading to normal-ordered output. We refer to these options as normal-to-reversed (NR) transformation and reversed-to-normal (RN) transformation, respectively. For a thorough study of different NTT algorithms, we refer to [17] and [13].

2.2 Polynomial Multiplication using NTT

The NTT is commonly used to accelerate polynomial multiplication denoted as $\mathbf{c}(x) = \mathbf{a}(x) \times \mathbf{b}(x)$. For that, three steps are required. The first step is to NTT-transform the two input polynomials to get $\hat{\mathbf{a}}(x)$ and $\hat{\mathbf{b}}(x)$. In the second step, a coefficient-wise multiplication $\hat{\mathbf{c}}(x) = \hat{\mathbf{a}}(x) \odot \hat{\mathbf{b}}(x)$ is performed ($\odot$ denotes the coefficient-wise multiplication). Finally, $\hat{\mathbf{c}}(x)$ is transformed back to the coefficient domain to yield $\mathbf{c}(x)$. During polynomial multiplication, the degree of the resulting polynomial grows larger than $N - 1$, which needs to be considered in NTT computations. Again, there exist two approaches to cope with this property. The first approach is to perform $2N$-point NTT on zero-padded input polynomials $\mathbf{a}(x)$ and $\mathbf{b}(x)$. The following coefficient-wise multiplication is done

Algorithm 1 NWC-based Polynomial Multiplication [18].

Input: $\mathbf{a}(x), \mathbf{b}(x) \in \mathcal{R}_q$; ψ: primitive $2N$-th root of unity in $\mathbb{Z}_q$
Output: $\mathbf{c}(x) \leftarrow \mathbf{a}(x) \times \mathbf{b}(x) \in \mathcal{R}_q$

1: $\hat{\mathbf{a}}(x) \leftarrow \textsc{Ntt}\big(\mathbf{a}(x) \odot (1, \psi^1, \ldots, \psi^{N-1})\big) = \textsc{Mntt}\big(\mathbf{a}(x)\big)$
2: $\hat{\mathbf{b}}(x) \leftarrow \textsc{Ntt}\big(\mathbf{b}(x) \odot (1, \psi^1, \ldots, \psi^{N-1})\big) = \textsc{Mntt}\big(\mathbf{b}(x)\big)$
3: $\hat{\mathbf{c}}(x) \leftarrow \hat{\mathbf{a}}(x) \odot \hat{\mathbf{b}}(x)$
4: $\mathbf{c}(x) \leftarrow \textsc{Intt}\big(\hat{\mathbf{c}}(x)\big) \odot \big(1, \psi^{-1}, \ldots, \psi^{-(N-1)}\big) = \textsc{Mintt}\big(\hat{\mathbf{c}}(x)\big)$
5: **return** $\mathbf{c}(x)$

on $2N$ coefficients and finally, $2N$-point INTT is performed. The resulting polynomial is then reduced by the irreducible polynomial $(x^N + 1)$. An alternative approach is the negative-wrapped convolution technique [23]. Here, the $2N$-point NTT is reduced to an N-point NTT, and zero-padding is avoided. Instead, preprocessing and postprocessing steps are required. These additional steps multiply powers of $2N$-th roots of unity $\psi^{2N} = 1 \bmod q$ to the input and output polynomial, as Alg. 1 shows. The pre- and postprocessing multiplications can be merged to the NTT computation which reduces the computational overhead [23]. We refer to this merged NTT variant as MNTT and MINTT for forward and inverse transformation, respectively.

2.3 Twiddle Factors in NTT

The twiddle factor management is an important design aspect in NTT accelerators. This design aspect strongly influences the resource utilization of hardware designs. Hence, we discuss common twiddle factor management techniques in this section.

2.3.1 Stored Twiddle Factors. During one forward NTT transformation, twiddle factors $\omega^0, \omega^1, \ldots, \omega^{N/2-1}$ are needed (see Sec. 2.1). The twiddle factors for INTT are $\omega^0, \omega^{-1}, \ldots, \omega^{-(N/2-1)}$. Therefore, the straightforward approach is to precompute the $2 \cdot N/2$ twiddle factors for both transformations and store them in memory. This results in a total of N stored twiddle factors per modulus q. In the case of MNTT, the number of twiddle factors doubles due to the $2N$-th roots of unity. This leads to a memory consumption of $2N$ twiddle factors per prime q in MNTT.

The main issue of stored twiddle factors is the limited scalability of this approach. The number of memory elements needed scales linear with N which becomes critical in large-degree polynomials such as in FHE. In addition, FHE commonly uses the residue-number system [4, 12] to lower computational complexity. This means that NTT must be performed multiple times for different moduli $q_0 \ldots q_{L-1}$ and, hence, L different sets of twiddle factors are needed. Therefore, twiddle factors must (1) either be streamed from off-chip memory or (2) multiple sets of twiddle factors must be stored on-chip. The former approach (1) introduces high data transfer penalties [11] whereas the latter approach (2) increases the on-chip memory consumption. Hence, both approaches limit the efficiency of the NTT design.

2.3.2 On-the-fly Generated Twiddle Factors. An alternative to stored twiddle factors is the on-the-fly generation of twiddle factors (TFG in short). TFG relies on computing the twiddle factors needed in NTT instead of storing them. This computation is done by repeatedly multiplying twiddle factors by each other to yield the desired twiddle factor sequence. The generation circuit hence instantiates a dedicated modular multiplier and a small memory containing a few, initial twiddle factors needed to fill the multiplier pipeline. The reduction in memory consumption makes TFG an attractive choice when memory is scarce or N is large. This especially applies to FHE applications with $N \approx 2^{16}$, leading to a memory reduction from several MB to a few kB. Furthermore, switching primes q as in FHE is more efficient in TFG since just a few initial twiddle factors need to be replaced.

3 The proposed Twiddle Factor Generation Algorithm

OpenNTT provides flexible NTT hardware designs with on-the-fly twiddle factor generation. Therein, it is challenging to find generic rules for combining conflict-free memory accesses and linear twiddle factor generation. This section presents our solution to this challenge. First, the conflict-free memory access pattern is discussed. Based on this access pattern, we present our processing orders and our efficient twiddle factor generation for various parameter sets.

3.1 Conflict-free Memory Access Pattern

Computing an NTT transformation requires performing a certain amount of butterfly operations. These butterfly operations can be executed simultaneously in hardware, which leads to a lower latency. For that, multiple processing elements (PE) are instantiated, where each PE performs one butterfly operation. In addition, the processing elements are fully pipelined to perform one butterfly operation in each clock cycle. Hence, two input coefficients and one twiddle factor are consumed by each PE per clock cycle. To load the involved input coefficients simultaneously, they must reside in distinct RAM banks with one read port and one write port per bank. Similarly, the outputs of the PEs need to be stored to memory, which again requires two stores per cycle, per PE. Therefore, two memory banks are needed per PE leading to a total of $2n_{PE}$ banks.

Several works address this problem and propose different NTT memory access patterns [2, 10]. In OpenNTT, we use the approach of [24], which is explained in the following. For this explanation, consider a $N = 16$ point NR NTT with $n_{PE} = 2$ processing elements (denoted as PE0 and PE1). This configuration needs $2n_{PE} = 4$ banks (bank 0 to bank 3), where the two inputs of PEi are hardwired to the read ports of bank $2i$ and bank $2i + 1$. For example, coefficients read from bank 0 and bank 1 enter PE0, and coefficients from bank 2 and bank 3 enter PE1. The memory bank content before and after each of the $\log_2(N) = 4$ stages is shown in Fig. 3.

The NTT transformation starts in stage 0 (top of Fig. 3). In stage 0, the coefficients (a_0, a_8) in red and (a_4, a_{12}) in blue are simultaneously loaded from memory and fed to the two PEs. After several cycles, the result of this operation appears in the output of the PEs and needs to be stored to memory. Yet, before being stored, the resulting coefficients are reordered (RO). This reordering makes pairs of coefficients residing at the correct locations to be fetched together in the next stage. Hence, considering our example, (a_0, a_4) and (a_8, a_{12}) are stored to address 0 (see after stage 0 in Fig. 3). This repeats until all coefficients of stage 0 are processed.

Next, NTT advances to stage 1. Unlike stage 0, stage 1 requires reordering of PE outputs across memory addresses. For example, a_0

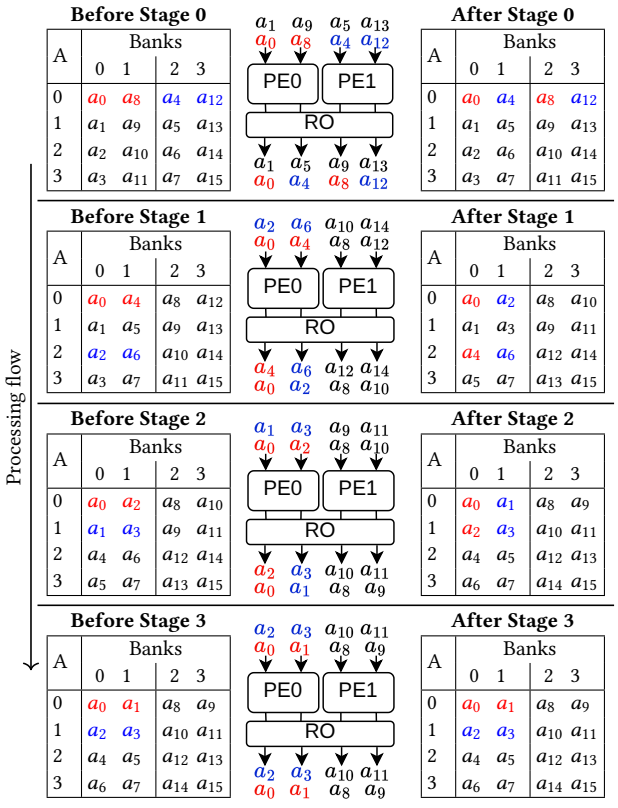

Figure 3: Conflict-free memory management for $N = 16$, $n_{PE} = 2$ NR NTT.

and a_4 (red) are processed together but must be stored to addresses A0 and A2 in Bank 0 (see after stage 1 in Fig. 3). Therefore, the load pattern needs to be adapted: In the first step, (a_0, a_4) and (a_8, a_{12}) are loaded and enter the PE pipeline. One cycle after, (a_2, a_6) and (a_{10}, a_{14}) are loaded to the PE pipeline. Now, all coefficients required for one reorder operation are within consecutive pipeline stages. The reorder logic combines and swaps the outputs of two consecutive cycles leading to store (a_0, a_2) and (a_8, a_{10}) to address A0 in the first cycle. In the following cycle, (a_4, a_6) and (a_{12}, a_{14}) are stored to address A2. This repeats for the remaining coefficients of stage 1 and also the following stages maintain this approach. Yet, the final stage does not need to perform any reordering since no further butterfly operations need to be done after this stage. Therefore, the output coefficients of the PEs are stored to the same address where the input coefficients were loaded from.

Based on the discussed NTT example with parameters $N = 16$, $n_{PE} = 2$, we derive generic requirements for proper NR NTT execution flows: In the first $\log_2(n_{PE})$ stages, only reordering within one PE output happens. Therefore, arbitrary processing orders are allowed in this case. Similarly, in the last stage, no reordering is done. This also allows any processing order. Yet, in stages $s = \log_2(n_{PE}) \dots S-2$, reordering across two consecutive PE outputs must happen. Hence, the coefficients to be reordered together must be loaded right after each other. We denote the load address dependency between even and odd load cycles as $laddr_O = laddr_E + x$, where $laddr_E$ is the address in even cycles, $laddr_O$ is the address in odd cycles, and x is the load address offset (I.e.: the load address sequence A0, A2, A1, A3 is denoted as $laddr_O = laddr_E + 2$). This leads to the formal requirement on the processing order for NR NTTs as follows:

$$\textbf{if } 0 \le s < \log_2(n_{PE}) \textbf{ or } s = S-1 \textbf{ then} : \text{Any order}$$

$$\textbf{if } \log_2(n_{PE}) \le s < S-1 \textbf{ then} :$$

$$laddr_O = laddr_E + \left(N/4/n_{PE} \gg (s - \log_2(n_{PE}))\right)$$

Similar to the NR case, we derive the memory access constraints for the RN NTT transformation. The derivation of the constraints follows the same concept as in the NR case from above and yields:

$$\textbf{if } s < S - \log_2(n_{pe}) \textbf{ then} :$$

$$laddr_O = laddr_E + \left(N/n_{PE} \gg (S - \log_2(n_{PE}) - s)\right)$$

$$\textbf{if } s \ge S - 1 - \log_2(n_{PE}) \textbf{ then} : \text{Any order}$$

These derived constraints for NR and RN NTTs must be obeyed to ensure conflict-free and stall-free NTT transformations. In addition to that, TFG introduces further requirements on the processing order, which is elaborated in the next section.

3.2 Processing Order in NTT

We build upon the defined requirements from the conflict-free memory layout and derive the actual processing order of NTT. Thereby, we consider the constraints from the memory layout and combine them with an efficient twiddle factor generation order.

Twiddle factor generation relies on a small set of stored twiddle factors, which are multiplied repeatedly by each other. This multiplication leads to a linear generation order, (i.e. $\omega^{0k}, \omega^{1k}, \omega^{2k}, \dots$), whereas an arbitrary generation order is not possible without significant logic overhead. Hence, we adapt the NTT processing flow to cope with this linear generation order. Furthermore, it is desirable to unify the DIT and DIF processing flows to allow a more efficient hardware compilation in OpenNTT. Therefore, we aim for one NTT processing flow that suits both, DIF and DIT decimation methods.

We first discuss the NTT processing flow for DIT NR and DIF NR transformations. For that, we consider $N = 32$ and $n_{PE} = 2$. Tab. 1 shows relevant stages of the NTT transformation and the coefficients involved in each butterfly operation. Furthermore, the twiddle factor is given for each butterfly for the DIT and DIF cases.

We first consider stage 0, where DIT only uses the twiddle factor ω^0. Contrarily, DIF uses $\omega^0, \omega^1, \dots$ in butterfly operation $B0, B1, \dots$ of PE0. Hence, we use the processing order $B0, B1, \dots, B7$ as it allows linear twiddle factor generation and complies with the memory access constraints from Sec. 3.1.

In stage 1, DIT still uses constant twiddle factors per PE and hence does not need any specific order. Yet, memory constraints require to process $B0$ and $B4$ directly after each other (see Sec. 3.1). This means that DIF needs ω^0 and ω^8 as the first two twiddle factors. To still allow a somewhat linear generation, we choose the processing order $B0, B4, B1, B5, \dots$ for stage 1, as shown in Tab. 1. Using this approach, we can perform two concurrent, group-wise linear twiddle factor generations of $\underline{\omega}^0, \boldsymbol{\omega}^8, \underline{\omega}^2, \boldsymbol{\omega}^{10}, \dots$ Thereby, the underlined and bold twiddle factors show a linear generation order in their group, and both groups have the same multiplicative offset

Table 1: Twiddle factor order in $N = 32$, $n_{PE} = 2$ NR NTTs.

Butterfly operation		Proc. order	PE0 Coeffs[1]		PE0 ω DIF[2]	PE0 ω DIT[2]	PE1 Coeffs[1]		PE1 ω DIF[2]	PE1 ω DIT[2]
Stage 0 (any order)	$B0$	0	0	16	0	0	8	24	8	0
	$B1$	1	1	17	1	0	9	25	9	0
	$B2$	2	2	18	2	0	10	26	10	0
	$B3$	3	3	19	3	0	11	27	11	0
	$B4$	4	4	20	4	0	12	28	12	0
	$B5$	5	5	21	5	0	13	29	13	0
	$B6$	6	6	22	6	0	14	30	14	0
	$B7$	7	7	23	7	0	15	31	15	0
Stage 1	$B0$	0	0	8	0	0	16	24	0	8
	$B1$	2	1	9	2	0	17	25	2	8
	$B2$	4	2	10	4	0	18	26	4	8
	$B3$	6	3	11	6	0	19	27	6	8
	$B4$	1	4	12	8	0	20	28	8	8
	$B5$	3	5	13	10	0	21	29	10	8
	$B6$	5	6	14	12	0	22	30	12	8
	$B7$	7	7	15	14	0	23	31	14	8
Stage 2	$B0$	0	0	4	0	0	16	20	0	4
	$B1$	2	1	5	4	0	17	21	4	4
	$B2$	1	2	6	8	0	18	22	8	4
	$B3$	3	3	7	12	0	19	23	12	4
	$B4$	4	8	12	0	8	24	28	0	12
	$B5$	6	9	13	4	8	25	29	4	12
	$B6$	5	10	14	8	8	26	30	8	12
	$B7$	7	11	15	12	8	27	31	12	12
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
Stage 4 (any order)	$B0$	0	0	1	0	0	16	18	0	1
	$B1$	4	2	3	0	8	17	19	0	9
	$B2$	2	4	5	0	4	20	22	0	5
	$B3$	6	6	7	0	12	21	23	0	13
	$B4$	1	8	9	0	2	24	26	0	3
	$B5$	5	10	11	0	10	25	27	0	11
	$B6$	3	12	13	0	6	28	30	0	7
	$B7$	7	14	15	0	14	29	31	0	15

Notation: [1]: Coefficient indices, i.e. a_i; [2]: Twiddle factor powers, i.e. ω^i

of ω^2 between two consecutive twiddle factors. This allows an efficient generation with low logic overhead.

In stage 2, the memory access constraints require butterfly operations $B0$ and $B2$ to be processed directly after each other. Hence, stage 2 follows a similar approach as stage 1 and uses the processing order of $B0, B2, B1, B3, \ldots$. This leads to a proper NTT execution flow that complies with the memory constraints. Thereby, the DIF twiddle factors are generated somewhat linearly (as in stage 1) while DIT twiddle factors are generated linearly. The same applies to stage 3 leading to an order of $B0, B1, B4, B5, \ldots$

Finally, in stage 4, DIF only uses ω^0 and hence does not pose requirements on the processing order. Therefore, we choose a bit-reversed order of processing. This allows a linear generation order of the DIT twiddle factors, namely $\omega^0, \omega^2, \omega^4, \ldots$ for PE0.

With the observations from this specific instance of NTT ($N = 32$, $n_{PE} = 2$), we can derive the general processing order of NR NTTs for arbitrary N and n_{PE}. Alg. 2 computes this sequence unifying efficient twiddle factor generation and a conflict-free memory access pattern. The first $\log_2(n_{PE})$ stages follow a linear processing sequence. Then, from stage $\log_2(n_{PE})$ onwards, a non-linear but TFG-friendly sequence is used. We further see from Alg. 2 that the address generation logic only consists of hardware-friendly operations and yields a correct processing order for all combinations of N and n_{PE}. A similar derivation can be done for RN NTTs as well.

Algorithm 2 Processing order of coefficients during NR NTT.

Input: N, n_{PE}
Output: $addr[N/2/n_{PE} \cdot \log_2(N)]$: array with the address sequence

```
addr ← []
for s = 0 to log2(N) − 1 do
    for i = 0 to N/2/n_PE do
        if s < log2(n_PE) then
            addr.Append(i)                          ▷ linear order
        else
            b ← log2(N/2/n_PE)                      ▷ bit-width of address
            l_bits ← b − s + log2(n_PE)             ▷ bit-width of low part
            h ← i[b − 1 : l_bits]
            l ← i[l_bits − 1 : 0]
            h ← BitReverse(h)                       ▷ bit-reverse high part
            l ← l ⋙ 1                               ▷ rotate low part
            addr.Append({h, l})
        end if
    end for
end for
return addr
```

3.3 Optimized Twiddle Factor Generation

After deriving proper NTT processing orders, we focus on the twiddle factor generation itself. The TFG in OpenNTT is optimized for low memory usage and must support different NTT types, polynomial degrees, and number of processing elements. Each PE is equipped with an exclusive multiplier unit for twiddle factor generation. This multiplier unit is fully pipelined with d_{Mul} stages, where d_{Mul} depends on the prime size and the reduction method. Given these flexibility requirements on N, n_{PE}, d_{Mul}, and NTT type, we detail our optimized TFG architecture throughout this section.

In each NTT transformation, the generation multiplier's pipeline must be initially filled with twiddle factors obtained from a ROM memory. After the initial filling, output twiddle factors are fed back to the multiplier to generate the remaining sequence of twiddle factors. The corresponding architecture is shown in Fig. 4.

A design goal of OpenNTT is to reduce the number of twiddle factors in ROM memory (SharedTwROM in Fig. 4). To illustrate our corresponding optimizations, we consider a multiplier with latency $d_{Mul} = 4$. Moreover, Tab. 2 shows the first two stages of a DIF NR transformation with $N = 128$ and $n_{PE} = 4$.

The NTT transformation starts with the initialization for stage 0. During initialization, the registers c and $t[0 \ldots n_{PE} - 1]$ in Fig. 4 are initialized with a total of $1 + n_{PE} = 5$ twiddle factors from the shared TwROM memory. The datapath used during initialization is shown in red in Fig. 4. After this initialization, the transformation starts, and the first $d_{Mul} = 4$ twiddle factors ($\omega^0 \ldots \omega^3$, indicated in red in Tab. 2) are loaded from ROM. These four twiddle factors are broadcast to the four PEs, and each PEi multiplies $\omega^0 \ldots \omega^3$ with $t[i]$ to yield the first four twiddle factors for step 0 to step 3.

After that, from step 4 onwards, the four generation modules (TwGen) start to independently multiply the produced twiddle factors with the constant ω^c, which is stored in registers, to yield the remaining twiddle factor sequence for stage 0 (shown in Tab. 2). For example, as soon as PE3 outputs ω^{48}, this twiddle factor is fed back to the multiplier and gets multiplied by $\omega^c = \omega^4$ to give ω^{52} after $d_{Mul} = 4$ cycles.

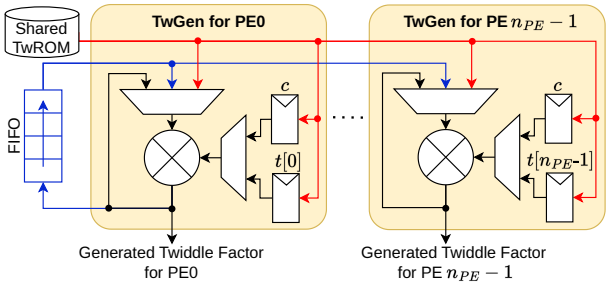

Figure 4: Architecture of our generic twiddle factor generation. Red arrows indicate data flow during initialization. Blue indicates data flow for stage setup.

During the twiddle factor generation of stage 0, we prepare the values for c and $t[0\ldots3]$ registers needed in stage 1. For that, a systematic and scalable strategy is used to collect c and $t[0\ldots3]$ for stage 1 during stage 0, which avoids ROM usage. This strategy is detailed as follows: The constant factor ω^c for stage 1 is obtained from the twiddle factor output in step $N/4/n_{PE} = 8$ of PE0's generation module (indicated with ③ in Tab. 2). Further, the elements $t[i]$ are obtained from outputs marked with ① in Tab. 2. Finally, $d_{Mul} = 4$ many twiddle factors are needed to fill the multiplier pipeline at the beginning of stage 1. These twiddle factors are obtained from stage 0 and fed into a FIFO with d_{Mul} elements (blue datapath in Fig. 4). The FIFO is filled with the twiddle factors produced by PE0 in even steps, which are marked with ② in Tab. 2.

In the transition from stage 0 to stage 1, the generation multipliers are provided with the FIFO content during the last d_{Mul} steps of stage 0 (indicated in blue in Tab. 2). The multipliers then output the desired twiddle factors after d_{Mul} clock cycles, which is just in time for the beginning of stage 1, thereby avoiding pipeline stalls. Note that this approach does not access the twiddle factor ROM after the initialization of stage 0, allowing a reduced ROM size.

This procedure is repeated for the remaining stages, and can easily be scaled to arbitrary parameter sets and NTT types. Note that our execution sequence from Sec. 3.2 must be maintained. The discussed stages from Tab. 2 allow a linear processing order, which is not the case in further stages. However, the presented approach also applies to the somewhat linear processing sequence from Sec. 3.2, with a small amount of additional control logic needed.

4 Overall Design of OpenNTT

After presenting our optimized and generalized TFG approach, this section explains our whole OpenNTT framework. The section first explains the parameter sets of OpenNTT and the flexible hardware architecture that uses our generalized TFG. Then, the OpenNTT design flow is discussed.

4.1 Supported Parameters in OpenNTT

Our OpenNTT framework compiles hardware designs for a wide range of NTT parameter sets. OpenNTT can compile unified NTT modules (UNTT) as well as standalone forward (FNTT) and inverse (INTT) transformation modules. UNTT modules support forward and inverse NTT in a unified hardware architecture, e.g. needed in PQC [3]. Unlike UNTT, the standalone NTT modules only support one direction of transformation and hence save hardware resources. Thus, standalone modules are relevant in constrained applications, such as in client-side FHE [20]. In addition, OpenNTT supports multiple primes $q_0 \ldots q_{L-1}$ efficiently in one architecture. A seamless transition between primes can be performed without off-chip communication due to the on-the-fly TFG. This is a clear benefit compared to stored twiddle factors which introduce overheads in frequent prime changes.

Finally, OpenNTT provides the choice of incorporating coefficient arithmetic into the NTT design. Thereby, users select whether they want hardware support for coefficient-wise arithmetic such as addition, subtraction, and multiplication along with the NTT transformation. To efficiently compute these coefficient-wise operations, a user-defined number of polynomials n_{Poly} can be stored in OpenNTT's memory subsystem. Two of these stored polynomials are involved in each arithmetic operation. The selection of involved polynomials is done by control signals to allow a high flexibility in polynomial arithmetic computation.

Table 2: Memory-optimized TFG for DIF NR.

Stage 0

Initialization: $c \leftarrow 4$ from ROM, $t \leftarrow [0, 16, 32, 48]$ from ROM

Step	PE0	PE1	PE2	PE3
0	$\mathbf{0} = 0 + t[0]^{①②}$	$\mathbf{16} = 0 + t[1]$	$\mathbf{32} = 0 + t[2]^{①}$	$\mathbf{48} = 0 + t[3]$
1	$\mathbf{1} = 1 + t[0]$	$\mathbf{17} = 1 + t[1]$	$\mathbf{33} = 1 + t[2]$	$\mathbf{49} = 1 + t[3]$
2	$\mathbf{2} = 2 + t[0]^{②}$	$\mathbf{18} = 2 + t[1]$	$\mathbf{34} = 2 + t[2]$	$\mathbf{50} = 2 + t[3]$
3	$\mathbf{3} = 3 + t[0]$	$\mathbf{19} = 3 + t[1]$	$\mathbf{35} = 3 + t[2]$	$\mathbf{51} = 3 + t[3]$
4	$\mathbf{4} = 0 + c^{②}$	$\mathbf{20} = 16 + c$	$\mathbf{36} = 32 + c$	$\mathbf{52} = 48 + c$
5	$\mathbf{5} = 1 + c$	$\mathbf{21} = 17 + c$	$\mathbf{37} = 33 + c$	$\mathbf{53} = 49 + c$
6	$\mathbf{6} = 2 + c^{②}$	$\mathbf{22} = 18 + c$	$\mathbf{38} = 34 + c$	$\mathbf{54} = 50 + c$
7	$\mathbf{7} = 3 + c$	$\mathbf{23} = 19 + c$	$\mathbf{39} = 35 + c$	$\mathbf{55} = 51 + c$
8	$\mathbf{8} = 4 + c^{③}$	$\mathbf{24} = 20 + c$	$\mathbf{38} = 36 + c$	$\mathbf{56} = 52 + c$
⋮	⋮	⋮	⋮	⋮
15	$\mathbf{15} = 11 + c$	$\mathbf{31} = 27 + c$	$\mathbf{47} = 43 + c$	$\mathbf{63} = 59 + c$

Stage 1

Setup: $c \leftarrow 8$ from ③ in Stage 0, $t \leftarrow [0, 32, 0, 32]$ from ① in Stage 0, $FIFO \leftarrow [0, 2, 4, 6]$ from ② in Stage 0

Step	PE0	PE1	PE2	PE3
0	$\mathbf{0} = 0 + t[0]^{①②}$	$\mathbf{32} = 0 + t[1]$	$\mathbf{0} = 0 + t[2]^{①}$	$\mathbf{32} = 0 + t[3]$
1	$\mathbf{2} = 2 + t[0]$	$\mathbf{34} = 2 + t[1]$	$\mathbf{2} = 2 + t[2]$	$\mathbf{34} = 2 + t[3]$
2	$\mathbf{4} = 4 + t[0]^{②}$	$\mathbf{36} = 4 + t[1]$	$\mathbf{4} = 4 + t[2]$	$\mathbf{36} = 4 + t[3]$
3	$\mathbf{6} = 6 + t[0]$	$\mathbf{38} = 6 + t[1]$	$\mathbf{6} = 6 + t[2]$	$\mathbf{38} = 6 + t[3]$
4	$\mathbf{8} = 0 + c^{②}$	$\mathbf{40} = 32 + c$	$\mathbf{8} = 0 + c$	$\mathbf{40} = 32 + c$
5	$\mathbf{10} = 2 + c$	$\mathbf{42} = 34 + c$	$\mathbf{10} = 2 + c$	$\mathbf{42} = 34 + c$
6	$\mathbf{12} = 4 + c^{②}$	$\mathbf{44} = 36 + c$	$\mathbf{12} = 4 + c$	$\mathbf{44} = 36 + c$
⋮	⋮	⋮	⋮	⋮
15	$\mathbf{30} = 22 + c$	$\mathbf{62} = 54 + c$	$\mathbf{30} = 22 + c$	$\mathbf{62} = 54 + c$

Notation is in $\log_\omega$: $3 = 1 + 2$ refers to $\omega^3 = \omega^1 \cdot \omega^2 \bmod q$.

4.2 Flexible Hardware Architecture

OpenNTT uses a parametric and adaptive hardware architecture to meet the desired flexibility in hardware generation. This adaptive architecture is shown in Fig. 5.

OpenNTT instantiates n_{PE} many processing elements (PE), illustrated in the center of Fig. 5. Each PE has one TwGen module which

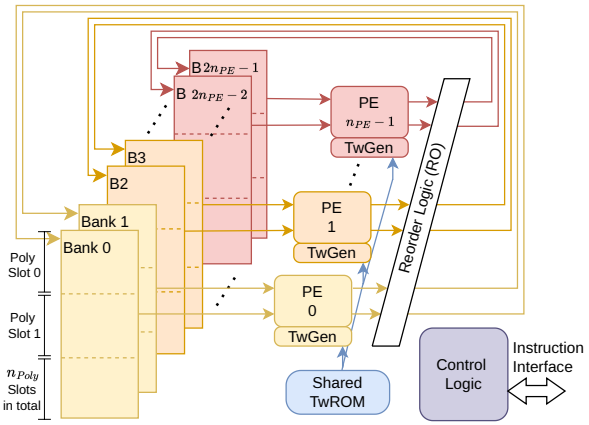

Figure 5: Overall Hardware Architecture of OpenNTT.

is responsible for delivering twiddle factors to the PE. Thereby, TwGen uses our generic TFG algorithm detailed in Sec. 3. The shared twiddle factor ROM (TwROM) serves all PEs along the blue datapath. This ROM contains a minimal amount of precomputed twiddle factors and is used to fill the multiplier pipeline initially.

Next to the twiddle factor, each PE takes two coefficients from memory as input. Hence, two memory banks with one read and one write port each are required per PE. The resulting $2n_{PE}$ memory banks are shown on the left in Fig. 5. The memory banks store up to n_{Poly} polynomials. Each polynomial resides in one PolySlot spread across all memory banks. The polynomial slots can be selected as input operands for NTT and coefficient-wise operations.

Finally, the control logic (bottom right in Fig. 5) is responsible for orchestrating the operations in OpenNTT. The control logic takes various input signals, which partially depend on the chosen NTT configuration. Based on these signals, it outputs proper read and write addresses and control signals for the datapath and the ReorderLogic. The ReorderLogic is required to store the butterfly results to proper memory locations, as detailed in Sec. 3.1.

The presented hardware architecture is efficiently adaptable to different parameter sets. This allows an automated and user-friendly design flow which is discussed in the next section.

4.3 Design Flow in OpenNTT

The design flow in OpenNTT overtakes all main steps of hardware generation. Based on the provided parameter set, OpenNTT uses Python to produce parametrized SystemVerilog code. The resulting SystemVerilog code can easily be integrated into larger designs.

In addition to that, OpenNTT provides testbenches and testvectors to verify the correctness of the hardware design in simulation and on real FPGAs. For that, the generated NTT accelerator is instantiated in a prepared RTL wrapper to interface with the FPGA. Subsequently, a ready-to-use C program executes prepared tests on the FPGA and verifies the correct functionality of the overall design. It is noteworthy that all these steps are fully automated and require no user interaction except specifying parameters.

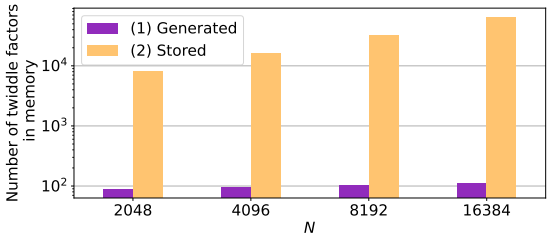

Figure 6: Comparison of twiddle factors needed in memory.

5 Results and Evaluation

We use our OpenNTT framework and the presented concepts to generate various NTT hardware designs. For that, we use Xilinx Vivado and Vitis 2022.2 with default settings to synthesize, place, and route our designs for different FPGAs. Furthermore, the correct functionality of OpenNTT was verified by testing the designs with different parameter sets on the ZYNQ-Z2 and Alveo U250 FPGAs.

The remainder of this section presents the gathered implementation results of OpenNTT. We first discuss the impact of TFG on resource utilization before comparing our results to related work.

5.1 Resource Utilization

The selection between stored and generated twiddle factors in NTT strongly influences resource utilization. We discuss this influence by considering an FHE application with typical parameters [25, 1]. The selected parameter set uses 8 different primes q_0 to q_7 each having $\lceil \log_2(q_i) \rceil = 54$ bits.

The memory consumption for (1) our optimized on-the-fly generated twiddle factors and (2) stored twiddle factors for different N is shown in Fig. 6. We observe a significantly higher (between 93× and 585×) memory consumption for stored twiddle factors compared to our optimized on-the-fly TFG. Moreover, the difference in memory usage increases with larger polynomial degrees. This is caused by the linear memory growth w.r.t. N in stored twiddle factors, whereas TFG shows logarithmic growth w.r.t. $\log_2(N)$. The advantage of lower memory consumption in TFG comes with the drawback of higher logic utilization. This increased logic utilization in terms of DSPs and LUTs is due to the additional multipliers in the generation logic. However, the DSP and LUT overhead only scales with the number of processing elements, but not with the polynomial degree or the number of primes. This makes TFG an attractive choice for large-degree NTTs as used in FHE.

5.2 Comparison to Related Works

Throughout this section, we present OpenNTTs implementation results and compare them with related works. The related works report NTT results for various parameter sets and different FPGA platforms. Hence, we use our OpenNTT framework to compile NTT designs for matching parameter sets and identical FPGAs, which enhances fair comparisons.

OpenNTT is a generic hardware compilation tool with on-the-fly twiddle factor generation. Yet, related works mostly use stored twiddle factors, which is an orthogonal design approach to TFG. This fact makes a direct comparison in terms of area consumption hard. To improve the comparability, we use the area-time product (ATP) as defined in [27]: $ATP = (LUT + 100 \cdot DSP + 300 \cdot BRAM) \cdot Lat$.

Table 3: Performance Comparison with Related Works targeting FHE Parameters.

Work	Platform	N	$\log(q)$	n_{PE}	LUT / FF / BRAM / DSP	Freq. MHz	Latency NTT cc	Latency NTT μs (impr.)	ATP (impr.)
[19]	Virtex-7 xc7vx690tffg1761-2	4096	24	1	802 / 525 / 7 / 4	185	24,583	132.9	439
		4096	24	8	5,665 / 3,188 / 8.5 / 33	157	3,079	19.7	227
Our		4096	24	1	1,254 / 1,927 / 4 / 8	333	24,610	73.9 (1.8)	240 (1.82)
		4096	24	8	8,630 / 9,963 / 5 / 64	320	3,106	9.7 (2.0)	160 (1.41)
[14]	Virtex-7 xc7vx485tffg1761-2	4096	32	4	6,300 / 5,200 / 14 / 24	224	6,158	27.46	354
		4096	60	1	2,600 / 2,500 / 21 / 26	144	24,590	171.6	1,973
		4096	60	8	22,100 / 19,500 / 48 / 208	141	3,086	21.91	1,255
Our		4096	32	4	5,973 / 8,030 / 9 / 56	290	6,194	21.4 (1.3)	305 (1.16)
		4096	60	1	5,996 / 9,845 / 16 / 44	250	24,650	98.6 (1.7)	1,498 (1.32)
		4096	60	8	34,886 / 45,316 / 18 / 352	225	3,146	14.0 (1.6)	1,055 (1.19)
[16]	Virtex-7 xc7vx690tffg1761-3	4096	24	16	10,800 / 9,500 / 40 / 112	220	1,545	7.0	238
		4096	60	1	1,900 / 1,800 / 17 / 42	154	24,585	159.6	1,788
		4096	60	8	14,100 / 12,500 / 41 / 336	150	3,081	20.5	1,230
Our		4096	24	16	19,912 / 19,583 / 17 / 128	290	1,570	5.4 (1.3)	205 (1.16)
		4096	60	1	6,003 / 9,600 / 16 / 44	250	24,650	98.6 (1.6)	1,499 (1.19)
		4096	60	8	35,262 / 44,107 / 18 / 352	240	3,146	13.1 (1.6)	994 (1.24)
[9]	ZCU102	65536	52	32	149k / 91k / 137 / 564	200	16,776	83.9	20,675
Our	xczu9eg-ffvb1156-2-e	65536	52	32	157k / 117k / 98 / 896	210	16,455	78.4 (1.1)	21,653 (0.95)
[18]	Virtex-7 xc7vx690tffg1761-2	1024	28	1	1,000 / 1,000 / 2 / 7	125	5,290	42.0	97
		1024	28	8	16,000 / 14,000 / 24 / 56	125	490	3.9	112
Our		1024	28	1	1,689 / 2,278 / 3 / 10	333	5,169	15.5 (2.7)	56 (1.73)
		1024	28	8	12,717 / 14,217 / 9 / 80	320	689	2.2 (1.8)	50 (2.23)

Tab. 3 presents the ATP and other benchmarks of related works. The table shows the parameter set, the used FPGA platform, the reported implementation results, and the ATP. In addition, the transformation latency and ATP improvement achieved by OpenNTT are given. Note, that all comparisons are based on identical FPGAs.

We first consider [19], which presents a scalable NTT design approach for different PE structures. Compared to [19], OpenNTT shows close to 50% lower BRAM consumption but needs 2× more DSPs. The cycle count for NTT transformations is similar, but OpenNTT reaches significantly higher clock frequencies due to improved pipelining. This leads to a latency reduction in OpenNTT of up to 2× and an ATP improvement of up to 82%. The work in [14] proposes runtime flexibility in the polynomial degree N and also relies on stored twiddle factors. OpenNTT reaches 1.3× to 1.7× and 16% to 33% lower latency and ATP, respectively across the different NTT configurations. Considering the BRAM consumption of [14], we see a clear benefit of generated twiddle factors over stored ones for large $N, \log(q)$, and n_{PE}. OpenNTT lowers the BRAM consumption by 63% from 48 to 18 BRAMs for $N = 4096, \log(q) = 60, n_{PE} = 8$. The authors of [16] propose a constant geometry NTT design based on stored twiddle factors. Their twiddle factor memory is optimized to reduce the BRAM consumption. Despite this optimization, the TFG in OpenNTT leads to up to 2.35× lower BRAM consumption, while the DSP consumption in OpenNTT is just 5% to 14% higher. However, OpenNTT requires up to 3× more LUTs, limiting the overall ATP improvement to 1.24×.

Next, we consider [9], which is an NTT accelerator for FHE. The accelerator features large polynomials with $N = 65536$ and $n_{PE} = 32$ PEs. OpenNTT produces similar results as [9] since both works use twiddle factor generation. The latency in OpenNTT is 7% lower but the ATP is 5% higher compared to [9]. Finally, the work in [18] presents an NTT design methodology for PQC and FHE applications. Compared to [18], OpenNTT reaches 2.5× higher frequency causing a significantly lower NTT latency and ATP.

Overall, it can be seen that OpenNTT designs outperform other NTT hardware generation tools by up to 2.7× and 2.23× in latency and ATP respectively. Furthermore, OpenNTT reduces the transformation latency compared to application-specific FHE designs such as [9]. The latency improvement of OpenNTT is explained by our design strategy that allows higher clock frequencies on the same FPGAs for a wide range of parameter sets. OpenNTT's TFG is efficient in terms of ATP and trades a higher DSP utilization for reduced memory consumption. This memory reduction supports the NTT deployment in memory-critical applications such as FHE.

6 Conclusion

In this paper, we presented OpenNTT, a flexible framework to compile efficient NTT designs for hardware platforms. In contrast to existing works, OpenNTT relied on on-the-fly twiddle factor generation to reduce the memory consumption of our designs. We proposed a generic design strategy to combine this twiddle factor generation with conflict-free memory accesses for various NTT types and parameter sets. Implementation results show the competitiveness of OpenNTT, which achieves a speedup of up to 2.7× compared to related work. Hence, OpenNTT supports a performant and efficient NTT deployment on hardware platforms.

References

[1] Martin Albrecht et al. 2019. Homomorphic encryption standard. Cryptology ePrint Archive, Paper 2019/939. (2019).

[2] Zahra Azad, Guowei Yang, Rashmi Agrawal, Daniel Petrisko, Michael Taylor, and Ajay Joshi. 2022. Race: risc-v soc for en/decryption acceleration on the edge for homomorphic computation. In *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design* (ISLPED '22) Article 13. Association for Computing Machinery, Boston, MA, USA, 6 pages. ISBN: 9781450393546. DOI: 10.1145/3531437.3539725.

[3] Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. 2018. Crystals-kyber: a cca-secure module-lattice-based kem. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 353–367.

[4] Jung Hee Cheon et al. 2019. A full rns variant of approximate homomorphic encryption. In *Selected Areas in Cryptography–SAC 2018: 25th International Conference, Calgary, AB, Canada, August 15–17, 2018, Revised Selected Papers 25*. Springer, 347–368.

[5] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic encryption for arithmetic of approximate numbers. In *Advances in Cryptology – ASIACRYPT 2017*. Tsuyoshi Takagi and Thomas Peyrin, (Eds.) Springer International Publishing, Cham, 409–437. ISBN: 978-3-319-70694-8.

[6] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2020. Tfhe: fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33, 1, 34–91.

[7] Ruan de Clercq, Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. 2015. Efficient software implementation of ring-lwe encryption. In *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 339–344. DOI: 10.7873/DATE.2015.0378.

[8] Léo Ducas, Eike Kiltz, Tancrede Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. 2018. Crystals-dilithium: a lattice-based digital signature scheme. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 238–268.

[9] Phap Duong-Ngoc, Sunmin Kwon, Donghoon Yoo, and Hanho Lee. 2022. Area-efficient number theoretic transform architecture for homomorphic encryption. *IEEE Transactions on Circuits and Systems I: Regular Papers*, PP, (Jan. 2022), 1–14. DOI: 10.1109/TCSI.2022.3225208.

[10] Yue Geng, Xiao Hu, Minghao Li, and Zhongfeng Wang. 2023. Rethinking parallel memory access pattern in number theoretic transform design. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 70, 5, 1689–1693. DOI: 10.1109/TCSII.2023.3260811.

[11] Zhenyu Guan et al. 2024. Esc-ntt: an elastic, seamless and compact architecture for multi-parameter ntt acceleration. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 1–6.

[12] Shai Halevi, Yuriy Polyakov, and Victor Shoup. 2019. An improved rns variant of the bfv homomorphic encryption scheme. In *Topics in Cryptology–CT-RSA 2019: The Cryptographers' Track at the RSA Conference 2019, San Francisco, CA, USA, March 4–8, 2019, Proceedings*. Springer, 83–105.

[13] Florian Hirner, Ahmet Can Mert, and Sujoy Sinha Roy. 2024. Proteus: a pipelined ntt architecture generator. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 1–11. DOI: 10.1109/TVLSI.2024.3377366.

[14] Xiao Hu, Jing Tian, Minghao Li, and Zhongfeng Wang. 2023. Ac-pm: an area-efficient and configurable polynomial multiplier for lattice based cryptography. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 70, 2, 719–732. DOI: 10.1109/TCSI.2022.3218192.

[15] Florian Krieger, Florian Hirner, Ahmet Can Mert, and Sujoy Sinha Roy. 2024. Aloha-he: a low-area hardware accelerator for client-side operations in homomorphic encryption. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 1–6.

[16] Si-Huang Liu, Chia-Yi Kuo, Yan-Nan Mo, and Tao Su. 2023. An area-efficient, conflict-free, and configurable architecture for accelerating ntt/intt. *IEEE Trans. Very Large Scale Integr. Syst.*, 32, 3, (Dec. 2023), 519–529. DOI: 10.1109/TVLSI.2023.3336951.

[17] Patrick Longa and Michael Naehrig. 2016. Speeding up the number theoretic transform for faster ideal lattice-based cryptography. In *Cryptology and Network Security*. Sara Foresti and Giuseppe Persiano, (Eds.) Springer International Publishing, Cham, 124–139. ISBN: 978-3-319-48965-0.

[18] Ahmet Can Mert, Emre Karabulut, Erdinç Öztürk, Erkay Savaş, Michela Becchi, and Aydin Aysu. 2020. A flexible and scalable ntt hardware: applications from homomorphically encrypted deep learning to post-quantum cryptography. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 346–351.

[19] Jianan Mu et al. 2023. Scalable and conflict-free ntt hardware accelerator design: methodology, proof, and implementation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 42, 5, 1504–1517. DOI: 10.1109/TCAD.2022.3205552.

[20] Deepika Natarajan and Wei Dai. 2021. Seal-embedded: a homomorphic encryption library for the internet of things. *IACR Trans. on Cryptographic Hardware and Embedded Systems*, 2021, 3, (July 2021), 756–779. DOI: 10.46586/tches.v2021.i3.756-779.

[21] 2024. Openntt source core. https://github.com/flokrieger/OpenNTT. (2024).

[22] Rogério Paludo and Leonel Sousa. 2022. Ntt architecture for a linux-ready risc-v fully-homomorphic encryption accelerator. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 69, 7, 2669–2682. DOI: 10.1109/TCSI.2022.3166550.

[23] Thomas Pöppelmann, Tobias Oder, and Tim Güneysu. 2015. High-performance ideal lattice-based cryptography on 8-bit atxmega microcontrollers. Cryptology ePrint Archive, Paper 2015/382. https://eprint.iacr.org/2015/382. (2015). https://eprint.iacr.org/2015/382.

[24] Sujoy Sinha Roy, Frederik Vercauteren, Nele Mentens, Donald Donglong Chen, and Ingrid Verbauwhede. 2014. Compact ring-lwe cryptoprocessor. In *Cryptographic Hardware and Embedded Systems – CHES 2014*. Lejla Batina and Matthew Robshaw, (Eds.) Springer Berlin Heidelberg, Berlin, Heidelberg, 371–391. ISBN: 978-3-662-44709-3.

[25] 2023. Microsoft SEAL (release 4.1). https://github.com/Microsoft/SEAL. Microsoft Research, Redmond, WA. (Jan. 2023).

[26] Kaustubh Shivdikar, Gilbert Jonatan, Evelio Mora, Neal Livesay, Rashmi Agrawal, Ajay Joshi, José L. Abellán, John Kim, and David Kaeli. 2022. Accelerating polynomial multiplication for homomorphic encryption on gpus. In *2022 IEEE International Symposium on Secure and Private Execution Environment Design (SEED)*, 61–72. DOI: 10.1109/SEED55351.2022.00013.

[27] Zewen Ye, Ray C. C. Cheung, and Kejie Huang. 2022. Pipentt: a pipelined number theoretic transform architecture. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 69, 10, 4068–4072. DOI: 10.1109/TCSII.2022.3184703.

[28] Neng Zhang, Bohan Yang, Chen Chen, Shouyi Yin, Shaojun Wei, and Leibo Liu. 2020. Highly efficient architecture of newhope-nist on fpga using low-complexity ntt/intt. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 49–72.

[29] Ye Zhang et al. 2021. Pipezk: accelerating zero-knowledge proof with a pipelined architecture. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 416–428. DOI: 10.1109/ISCA52012.2021.00040.