A Flexible Hardware Design Tool for Fast Fourier and Number-Theoretic Transformation Architectures

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

Fully Homomorphic Encryption (FHE) and Post-Quantum Cryptography (PQC) involve polynomial multiplications, which are a common performance bottleneck. To resolve this bottleneck, polynomial multiplications are often accelerated in hardware using the Number-Theoretic Transformation (NTT) or the Fast Fourier Transformation (FFT). In particular, NTT operates over modular rings while FFT operates over complex numbers. NTT and FFT are widely deployed in applications with diverse parameter sets, leading to long design times for hardware accelerators. Existing hardware generation tools have limited functionality since they do not support generic on-the-fly twiddle factor generation or different memory-related optimizations. This paper improves the hardware design process and presents a generic and flexible tool to generate FFT and NTT architectures. In contrast to prior work, we combine on-the-fly twiddle factor generation and stall-free memory accesses. Moreover, we enhance hardware design flexibility through memory-optimized or routing-optimized design strategies. While our memory-optimized strategy minimizes twiddle factors in ROM, our routing-optimized strategy allows significantly higher clock frequencies on FPGAs. These optimization strategies allow effective customization of NTT/FFT architectures, spanning from low-end PQC to high-end FHE accelerators. Compared to existing works, we reach up to 15.9× lower latency and up to 7.4× improved ATP for FFT applications such as the Falcon signature scheme. Considering other NTT tools, we decrease latency by up to 1.8× and 2× for PQC and FHE parameter sets, respectively.

Keywords

Hardware Design Tool, NTT, FFT, Twiddle Factor Generation, Fully Homomorphic Encryption (FHE)

1 Introduction

Contemporary research dedicates substantial effort to the implementation of modern cryptographic schemes. Many of those schemes, including fully homomorphic encryption (FHE) [9, 13] and post-quantum cryptography (PQC) [5, 10, 12], rely on lattice problems, making the involved large-scale polynomial multiplications a considerable performance bottleneck. A highly relevant method to accelerate polynomial multiplication is the Number-Theoretic transformation (NTT), which reduces the computational complexity from $\mathcal{O}(N^2)$ to $\mathcal{O}(N \log N)$, where N denotes the polynomial size [22].

The NTT typically operates over finite fields. However, some cryptographic applications such as the PQ-signature scheme Falcon [12] or the FHE scheme CKKS [9] rely on the Fast Fourier transformation (FFT), which operates over the complex numbers $\mathbb{C}$. Hence, the FFT requires a floating-point unit, whereas the NTT needs efficient modular reduction. Although FFT and NTT differ in their underlying arithmetic, they are closely related in the computation flow and algorithmic complexity. While NTT and FFT are widely used in cryptographic schemes, their hardware implementation poses a significant challenge. The wide range of FFT and NTT application scenarios – from low-end PQC designs to high-end FHE accelerators – requires versatile configurations. These configuration possibilities in algorithmic aspects (decimation method [22], negative-wrapped convolution [23], etc.) and in hardware aspects (area consumption, performance, etc.) often lead to dedicated hardware architectures tailored to only one application scenario [18, 29, 37]. Extending these dedicated designs to other application scenarios is challenging and involves long design times and engineering effort.

To reduce the hardware design effort, research has proposed different NTT hardware design tools. One example of prior research is the work [26], where a flexible hardware generator is presented. The generator produces NTT architectures depending on polynomial degree N, modulus size $\log(q)$, and number of processing elements n_{PE}. The main contribution of [26] is the scalable and conflict-free memory access pattern for in-place NTTs, which avoids pipeline stalls. Similarly, the work [21] supports different parameter sets at synthesis time, and even allows to change the polynomial degree N during runtime. The very recent work [20] supports different NTT architectures and an HLS-based design space exploration. There also exist works proposing two-dimensional arrays of processing elements [27]. In contrast to one-dimensional processing element layouts, 2D arrays reduce the overall number of memory accesses and, hence, lower the energy consumed in NTT transformations. Other works [15] reuse the stored twiddle factors. This measure lowers the ROM consumption since fewer twiddle factors need to be kept in on-chip memory.

However, we note that all these prior tools for hardware generation rely on stored twiddle factors, thus requiring large on-chip memory. This especially applies to high-degree polynomials as used in FHE. In contrast to storing, twiddle factors can also be generated on the fly to lower memory consumption. Yet, supporting twiddle

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

factor generation (TFG) in hardware design tools is challenging since conflict-free memory access patterns and a linear TFG order must be combined generically.

Our conference paper presented in [19] solves this issue and offers a hardware generation tool called OpenNTT. The OpenNTT tool from [19] allows for efficiently generating NTT hardware accelerators with TFG in a fully automated procedure. Yet, in this paper, we extend the conference paper [19] and propose two design-time optimizations. The first optimization – called MemOpt – minimizes the twiddle factors needed in ROM. Second, we propose RoutOpt to achieve higher clock frequencies in hardware at the cost of slightly increased ROM consumption. Thereby, we allow design-time flexibility between performance and resource consumption.

Moreover, the focus of the conference paper was on NTT acceleration in FHE applications. In this paper, we extend to PQC applications and introduce support for the floating-point-based FFT. We present detailed benchmarks of various configurations and compare the results to related work in the field. Our contributions can be summarized as follows:

- We present a conflict-free and generic processing order for efficient twiddle factor generation. This processing order is applied to NTT and FFT transformations, which are both supported by our hardware generation tool.
- We propose two design-time optimizations. First, our RoutOpt strategy elevates the frequency of FFT/NTT designs by improving routability at the cost of slightly higher twiddle memory consumption. Second, our MemOpt option minimizes the twiddle memory consumption but has a lower frequency. Thus, the generated hardware designs easily allow an area-performance tradeoff.
- We give a detailed discussion of benchmarks for FFT and NTT architectures in the field of PQC and FHE parameter sets. Moreover, we open-source our OpenNTT tool in [1] to support research in the field.

Outline: Sec. 2 introduces notation and background. Sec. 3 presents our execution flow, RoutOpt, and MemOpt strategies. Sec. 4 presents the flexible OpenNTT hardware architecture and the tooling workflow. Sec. 5 and Sec. 6 give implementation results and comparisons, respectively. Finally, Sec. 7 presents a discussion and Sec. 8 concludes the paper.

2 Background and Notation

We denote the ring of integers modulo a prime q as $\mathbb{Z}_q$. The polynomial ring $\mathcal{R}_q$ is defined as $\mathbb{Z}_q[x]/(x^N+1)$, consisting of polynomials with coefficients in $\mathbb{Z}_q$, reduced modulo the irreducible polynomial x^N+1. Here, N is restricted to powers of 2. We use lowercase letters in normal font ($a \in \mathbb{Z}_q$ or $b \in \mathbb{C}$) and bold font ($\mathbf{a} \in \mathcal{R}_q$) to distinguish between scalars and polynomials, respectively. The i-th coefficient of a polynomial $\mathbf{a}$ is denoted as a_i or $\mathbf{a}[i]$. A polynomial $\mathbf{a}$ in coefficient representation can therefore be expressed as $\mathbf{a} = \sum_{i=0}^{N-1} x^i a_i$. A polynomial $\mathbf{a} \in \mathcal{R}_q$ in its *evaluation* representation is denoted as $\hat{\mathbf{a}}$. We use $\cdot$, $\times$, and $\odot$ to distinguish between integer, polynomial, and coefficient-wise multiplication, respectively.

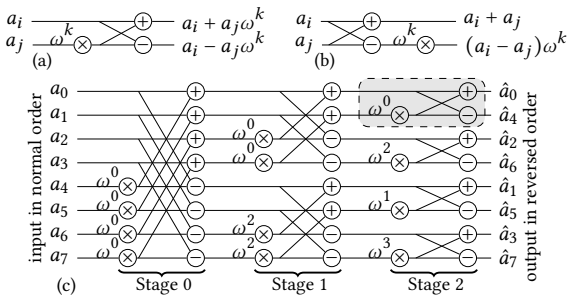

Figure 1: (a) CT butterfly and (b) GS butterfly [19]. (c) Data flow for $N = 8$ point NR DIT transformation with CT butterfly highlighted.

2.1 Number-Theoretic Transformation

The Number-Theoretic Transformation (NTT) is the Discrete Fourier Transformation over the polynomial ring $\mathcal{R}_q$. The NTT maps a polynomial $\mathbf{a}$ in its coefficient representation to its evaluation representation, denoted as $\hat{\mathbf{a}} = \textsc{Ntt}(\mathbf{a})$. The NTT is defined as $\hat{a}_k = \sum_{i=0}^{N-1} a_i \omega^{ik} \mod q$, where ω is an N-th primitive root of unity in $\mathbb{Z}_q$. Thus, ω satisfies the properties $\omega^N = 1 \mod q$ and $\omega^i \neq 1 \mod q$ for $0 < i < N$. The powers of ω are often referred to as *twiddle factors*. The inverse NTT transformation maps $\hat{\mathbf{a}}$ to the coefficient domain $\mathbf{a} = \textsc{Intt}(\hat{\mathbf{a}})$ and is defined as $a_i = N^{-1} \sum_{k=0}^{N-1} \hat{a}_k \omega^{-ik} \mod q$, where N^{-1} is the multiplicative inverse of N modulo q. The naïve computation of NTT has a runtime of $\mathcal{O}(N^2)$. Yet, optimized algorithms such as *Decimation-in-Time (DIT)* and *Decimation-in-Frequency (DIF)* reduce the computational complexity to $\mathcal{O}(N \log N)$ by leveraging symmetries in the twiddle factors in a divide-and-conquer approach [14]. The DIT algorithm employs the Cooley-Tukey (CT) butterfly operation, while the DIF algorithm uses the Gentleman-Sande (GS) butterfly operation. The two butterfly configurations are illustrated in Fig. 1. Each butterfly takes two coefficients (a_i and a_j) and one twiddle factor (ω^k). The butterfly operation is the fundamental operation in the NTT transformation, as shown in Fig. 1 (c) for an $N = 8$ point DIT NTT. In this example, the CT butterfly is applied iteratively across $\log_2 N = 3$ stages, with $N/2 = 4$ butterfly operations per stage.

In hardware designs, parallelism can be exploited by instantiating multiple processing elements (PEs), where each PE performs a butterfly operation. The number of such processing elements is denoted as n_{PE}. By increasing n_{PE}, multiple butterfly operations can be performed concurrently, lowering the latency of the transformation.

A detailed step-by-step computation of the DIT NTT using the Cooley-Tukey butterfly is shown in Alg. 1. The coefficients of the input polynomial $\mathbf{a}$ are assumed to be in normal order, and the output coefficients of $\hat{\mathbf{a}}$ are in bit-reversed order. In contrast, when the input polynomial is given in bit-reversed order, the output is in normal order. These two configurations are referred to as *normal-to-reversed (NR)* and *reversed-to-normal (RN)* transformations, respectively.

Algorithm 1 DIT NTT with Cooley-Tukey butterfly [36].

Input: $\mathbf{a} \in \mathcal{R}_q$ (in normal order), $\omega[N/2]$ (powers of ω)
Output: $\hat{\mathbf{a}} \leftarrow \textsc{Ntt}(\mathbf{a}) \in \mathcal{R}_q$ (in bit-reversed order)

1: **for** $(s = \log_2(N); s > 0; s = s - 1)$ **do**
2: **for** $(k = 0; k < N/2^s; k = k + 1)$ **do**
3: $m \leftarrow 2^s$; $tw \leftarrow \omega[\textsc{BitReverse}(k) \cdot m/2]$
4: **for** $(j = 0; j < m/2; j = j + 1)$ **do**
5: $u \leftarrow \mathbf{a}[km + j]$
6: $v \leftarrow \mathbf{a}[km + j + m/2] \cdot tw \mod q$
7: $\mathbf{a}[km + j] \leftarrow u + v \mod q$
8: $\mathbf{a}[km + j + m/2] \leftarrow u - v \mod q$
9: **return** $\mathbf{a}$

We also distinguish between out-of-place NTTs and in-place NTTs [21]. Out-of-place NTT designs store two polynomials on-chip. This increased memory consumption, however, allows a simpler execution flow within the transformation. In contrast, in-place NTT designs only instantiate memory for one polynomial. The resulting memory reduction has the disadvantage of a convoluted and non-uniform execution flow to avoid access conflicts to the single memory.

2.2 Fast Fourier Transformation

The Fast Fourier Transformation (FFT) is another important building block in cryptographic designs. For example, the CKKS FHE scheme [9] relies on FFT in the client-side operations. FFT is also part of Falcon [12], which is one post-quantum signature scheme standardized by NIST [28]. The FFT is very similar to the NTT and uses the same computation rules. However, instead of performing modular ring arithmetic, FFT relies on complex number arithmetic in $\mathbb{C}$ where each $x = x_r + \mathrm{i}x_i \in \mathbb{C}$ is expressed as two real values $x_r, x_i \in \mathbb{R}$ with i being the imaginary unit. Hence, a hardware implementation of NTT can be extended to FFT by replacing the modular ring data path with a floating-point-based complex number datapath. Depending on the desired accuracy of the FFT, single-precision or double-precision floating-point numbers with 32-bit and 64-bit word sizes, respectively, can be deployed. Hence, for storing one complex number $x \in \mathbb{C}$, 64 / 128 bits are required for single / double precision, respectively.

The twiddle factors ω^k in FFT are powers of the N-th primitive roots of unity in $\mathbb{C}$, i.e. $\omega^k = \exp(2\mathrm{i}\pi k/N)$. Unlike in NTT, where twiddle factors depend on the polynomial degree N and the prime q, twiddle factors in FFT only depend on the polynomial degree N. In addition, twiddle factors in FFT can be stored in ROM memory or generated on the fly.

2.3 Polynomial Multiplication using NTT and FFT

NTT and FFT are commonly employed to transform data between the coefficient domain (or time domain) and the evaluation domain (or frequency domain). This is especially useful for cryptography in accelerating polynomial multiplication.

Consider two polynomials $\mathbf{a}$ and $\mathbf{b}$. Their product $\mathbf{c} = \mathbf{a} \times \mathbf{b}$ can be computed efficiently using the NTT or FFT. The process begins by applying the forward NTT/FFT to the input polynomials $\mathbf{a}$ and $\mathbf{b}$, yielding their respective evaluation representations $\hat{\mathbf{a}}$ and $\hat{\mathbf{b}}$. In the evaluation domain, the multiplication is simplified to a coefficient-wise operation, expressed as $\hat{\mathbf{c}} = \hat{\mathbf{a}} \odot \hat{\mathbf{b}}$. This operation is followed by an inverse NTT/FFT to transform $\hat{\mathbf{c}}$ back to the coefficient domain $\mathbf{c}$.

Algorithm 2 NWC-based polynomial multiplication [26].

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

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

Note that using NTT or FFT for polynomial multiplication inherently increases the degree of the resulting polynomial $\mathbf{c}$, which must be managed efficiently. Two common techniques are used to address this challenge. The first is zero-padding, which extends the input polynomials with zeros to double their length, necessitating a $2N$-point transformation. While effective, this approach increases computational and memory requirements. The second technique, known as negative-wrapped convolution (NWC) [23], eliminates the need for zero-padding by leveraging $2N$-th roots of unity denoted as ψ. Alg. 2 shows the NWC-based polynomial multiplication involving pre- and postprocessing by coefficient-wise multiplications with ψ^k. The pre- and postprocessing steps can be merged into the NTT/INTT transformation [30]. We refer to this NTT type as *merged* NTT, denoted as MNTT/MINTT. Merged NTT significantly reduces computational and memory overhead but introduces higher complexity [14]. In addition, the twiddle factor generation (explained in Sec. 2.4) is more complicated in MNTT due to the convoluted twiddle factor sequence. While Alg. 2 shows NWC-based NTT, NWC can be directly applied to FFT as well.

2.4 Twiddle Factors in NTT and FFT

The twiddle factors (TF) ω^k, which are powers of the root of unity in their domain, are critical in NTT and FFT computations. Their management directly influences hardware scalability and performance. Two approaches for TF handling are commonly employed: (1) precomputing and storing TFs, or (2) twiddle factor generation (TFG). In TFG, the TF are dynamically computed during execution without storing them.

2.4.1 Stored Twiddle Factors. The stored approach precomputes and saves all required twiddle factors in memory, making it simple and efficient for small N. However, this method scales poorly for large polynomials since memory requirements grow linearly with N. Moreover, FHE schemes like CKKS use multiple moduli $q_0 \ldots q_{L-1}$, thereby storing multiple sets of twiddle factors – one for each modulus.

2.4.2 On-the-Fly Generated Twiddle Factors. Dynamically computing twiddle factors during execution reduces memory usage. This approach is well-suited for systems with large N or frequent modulus switching, as it removes the need to store multiple sets of precomputed factors. Although this approach introduces computational overhead, it lowers memory consumption. In hardware

designs, only a few base twiddle factors must be stored to fill the generation pipeline initially. Yet, this requires a dedicated processing sequence as twiddle factors cannot be generated efficiently in arbitrary order. This especially applies to merged NTT designs.

3 OpenNTT's Twiddle Factor Generation

It is challenging to ensure an efficient twiddle factor generation for generic parameter sets. This particularly applies to both algorithmic and hardware design perspectives. First, from the algorithmic perspective, conflict-free memory accesses must be combined with linear twiddle factor generation to ensure a fully pipelined design. Second, from the hardware design perspective, different synthesis-time options are desirable to allow customization of the hardware accelerator for different key benchmarks (low area, high performance, etc.).

This section presents both our proposed processing order supporting efficient TFG and our synthesis-time optimization strategies. Our optimizations allow for low memory consumption (MemOpt strategy) or higher design frequencies through improved routing capabilities (RoutOpt strategy).

3.1 Processing Order during Transformation

A fully pipelined hardware design for NTT or FFT must ensure conflict-free access to on-chip memory, as inefficiencies would arise due to stalls in the computational logic. Yet, the requirement of conflict-free memory accesses during in-place transformations introduces certain limitations to the processing order of the transformation: The required address sequence for reading and writing to on-chip memory is dependent on the coefficient ordering (NR and RN) and even changes across consecutive stages [31]. In addition, the twiddle factors can not be generated efficiently in an arbitrary order. Therefore, additional limitations need to be considered for the possible execution flow in NTT and FFT transformations.

In [19], a generic address generation algorithm is derived for NR NTT. This address generation is presented in Alg. 3. We extend Alg. 3 for NR transformations and derive the TFG-friendly address generation for RN transformations. Thereby, we follow a similar approach as in [19]. The resulting algorithm for RN transformations is shown in Alg. 4. We observe from Alg. 4 that in stages where $s < \log_2(N) - 1 - \log_2(n_{PE})$, the RN transformation cannot use a linear address sequence. Instead, the address sequence is computed as presented in lines 7 to 12 in Alg. 4. This computation first splits the index i into a high part h and a low part l, depending on the current stage s. Thereafter, the low part is rotated left and bit-reversed. Finally, the h is appended to l (l is in the most significant bits), yielding the desired address. Yet, in the last $\log_2(n_{PE})$ stages, any address sequence in RN transformation leads to non-conflicting memory accesses. Hence, we apply a linear address sequence in this case for RN transformations. This contrasts with the NR transformation in Alg. 3 where the *first* $\log_2(n_{PE})$ stages follow a linear processing order.

The presented Alg. 3 and 4 compute the desired address sequence generically for all N and n_{PE}. Following this sequence, we present our two optimization strategies for TFG in the next sections.

Algorithm 3 Processing order for NR transformation [19].

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

```
1:  addr ← []
2:  for s = 0 to log2(N) − 1 do                ▷ stage iterator s
3:    for i = 0 to N/2/n_PE do
4:      if s < log2(n_PE) then
5:        addr.Append(i)                       ▷ linear order
6:      else
7:        b ← log2(N/2/n_PE)                   ▷ bit-width of address
8:        l_bits ← b − s + log2(n_PE)          ▷ bit-width of low part
9:        h ← i[b − 1 : l_bits]
10:       l ← i[l_bits − 1 : 0]
11:       h ← BitReverse(h)                    ▷ bit-reverse high part
12:       l ← l ⋙ 1                            ▷ rotate low part
13:       addr.Append({h, l})                  ▷ bit-wise append h and l
14: return addr
```

Algorithm 4 Processing order for RN transformation.

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

```
1:  addr ← []
2:  for s = 0 to log2(N) − 1 do                ▷ stage iterator s
3:    for i = 0 to N/2/n_PE do
4:      if s ≥ log2(N) − 1 − log2(n_PE) then
5:        addr.Append(i)                       ▷ linear order
6:      else
7:        b ← log2(N/2/n_PE)                   ▷ bit-width of address
8:        h ← i[b − 1 : b − s]
9:        l ← i[b − s − 1 : 0]
10:       l ← l ⋘ 1                            ▷ rotate low part
11:       l ← BitReverse(l)                    ▷ bit-reverse low part
12:       addr.Append({l, h})                  ▷ bit-wise append l and h
13: return addr
```

3.2 Routing Optimized TFG Configuration

The twiddle factor generation in OpenNTT should support a wide range of parameters, including polynomial degree N or transformation type (NR or RN). Moreover, OpenNTT is flexible in the number of PEs (n_{PE}) instantiated, allowing a trade-off between performance and area consumption.

During the transformation, each of the n_{PE} PEs requires a unique twiddle factor sequence that depends on the parameters. Hence, we instantiate one dedicated multiplier unit (TwGen module) for each PE to compute its twiddle factor sequence. The multiplier is fully pipelined and has a latency of d_{Mul} stages, where d_{Mul} depends on the prime size or the floating point precision, respectively. Before each stage of a transformation, the multiplier pipeline must be filled with proper twiddle factors. These twiddle factors are obtained from a small ROM memory (TwROM) containing the corresponding subset of twiddle factors. Based on this initial filling, the remaining twiddle factors are computed by repeated multiplications.

In our routing-optimized TFG (RoutOpt), each PE has an exclusive TwROM in its TwGen module, as shown in Fig. 2. This allows the placement of these hardware resources near the PE, leading to improved routability and shorter critical paths. Thus, higher frequencies can be supported. The twiddle factor ROMs –

Table 1: Routing Optimized Twiddle Factor Generation for DIF NR.

Stage 0

Initialize registers: $c \leftarrow [3, 3, 3, 3]$ from TwROM

Step	TF for PE_0	TF for PE_1	TF for PE_2	TF for PE_3
0	**0** = 0	**16** = 16	**32** = 32	**48** = 48
1	**1** = 1	**17** = 17	**33** = 33	**49** = 49
2	**2** = 2	**18** = 18	**34** = 34	**50** = 50
3	**3** = 0 + $c[0]$	**19** = 16 + $c[1]$	**35** = 32 + $c[2]$	**51** = 48 + $c[3]$
4	**4** = 1 + $c[0]$	**20** = 17 + $c[1]$	**36** = 33 + $c[2]$	**52** = 49 + $c[3]$
5	**5** = 2 + $c[0]$	**21** = 18 + $c[1]$	**37** = 34 + $c[2]$	**53** = 50 + $c[3]$
⋮	⋮	⋮	⋮	⋮
15	**15** = 12 + $c[0]$	**31** = 28 + $c[1]$	**47** = 44 + $c[2]$	**63** = 60 + $c[3]$

Stage 1

Initialize registers: $c \leftarrow [6, 6, 6, 6]$ from TwROM

Step	TF for PE_0	TF for PE_1	TF for PE_2	TF for PE_3
0	**0** = 0	**32** = 32	**0** = 0	**32** = 32
1	**2** = 2	**34** = 34	**2** = 2	**34** = 34
2	**4** = 4	**36** = 36	**4** = 4	**36** = 36
3	**6** = 0 + $c[0]$	**38** = 32 + $c[1]$	**6** = 0 + $c[2]$	**38** = 32 + $c[3]$
4	**8** = 2 + $c[0]$	**40** = 34 + $c[1]$	**8** = 2 + $c[2]$	**40** = 34 + $c[3]$
5	**10** = 4 + $c[0]$	**42** = 36 + $c[1]$	**10** = 4 + $c[2]$	**42** = 36 + $c[3]$
⋮	⋮	⋮	⋮	⋮
15	**30** = 24 + $c[0]$	**62** = 56 + $c[1]$	**30** = 24 + $c[2]$	**62** = 56 + $c[3]$

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

instantiated once per PE – contain the minimal set of initial twiddle factors needed by the supplied PE. This initial set is used to fill the multiplier pipeline before each transformation stage. Note that some twiddle factors are stored redundantly in multiple TwROM instances due to missing interconnects between different PEs.

Tab. 1 shows an example of two stages within a DIF NR transformation for $N = 128$, $n_{PE} = 4$. The table indicates the power of twiddle factors (ω^i) needed by each PE and each transformation step. The twiddle factors forwarded to the PE are in bold characters, while the computation rule for these twiddle factors is in normal font. In addition, Alg. 5 details computation steps for the twiddle factors (TF) in RoutOpt.

Before the transformation starts, each TwGen_i loads a twiddle factor $c[i]$ from TwROM into a local register, as shown in line 2 in Alg. 5. Then, considering a multiplier latency of $d_{Mul} = 3$ clock cycles (cc), the first three TF of each stage are obtained from the TwROM (marked in red in Tab. 1 and shown in lines 4-7 in Alg. 5). In TwGen_0, stage 0, this would be $\omega^0, \omega^1, \omega^2$. The three TF from TwROM are directly forwarded to the PE performing the butterfly operation. Simultaneously, they are provided as input to the TwGen's multiplier to fill the multiplier's pipeline. During these multiplications, the pre-loaded TF $c[i]$ (e.g. $c[i] = \omega^3$ in stage 0) serves as multiplicand for generating the TF sequence from step 3 onward, which avoids TwROM access for the remaining stage. Thereby, only $c[i]$ is kept in a register. After the multiplier pipeline was filled with d_{Mul} TF from TwROM (lines 4-7 in Alg. 5), the multiplier outputs one computed TF per cc (starting in line 8) for the rest of the stage. The output TF is directly fed back and serves as input to another multiplication by $c[i]$ utilizing the TwGen

Algorithm 5 TF generation for PE_i in DIF NR RoutOpt.

Input: TwRom$[i]$: TwROM memory content for PE_i ($0 \le i < n_{PE}$),
Input: n_{PE}: number of PEs, d_{Mul}: multiplier latency
Output: TF_i: Twiddle factor sequence for PE_i

1: $TF_i \leftarrow []$
2: $c[i] \leftarrow$ TwRom$[i]$.GetData() ▷ Initialization for stage 0
3: **for** $s = 0$ to $\log_2(N) - 1$ **do** ▷ Stage iterator
4: **repeat** d_{Mul} **times** ▷ Steps 0 to 2 in Tab. 1
5: $\omega \leftarrow$ TwRom$[i]$.GetData()
6: TF_i.Append(ω)
7: Start multiplication of $\omega \cdot c[i]$ ▷ Result ready after d_{Mul} cc
/* *From now on, the multiplier outputs one result in each cc* */
8: **repeat** $\frac{N}{2n_{PE}} - 2 \cdot d_{Mul}$ **times** ▷ Step 3 onward in Tab. 1
9: $\omega \leftarrow$ multiplier output
10: TF_i.Append(ω)
11: Start multiplication of $\omega \cdot c[i]$ ▷ Result ready after d_{Mul} cc
12: **if** $s < \log_2(N) - 1$ **then** ▷ Initialize for next stage
13: $c[i] \leftarrow$ TwRom$[i]$.GetData()
14: **repeat** d_{Mul} **times** ▷ No multiplications issued
15: $\omega \leftarrow$ multiplier output
16: TF_i.Append(ω)
17: **return** TF_i

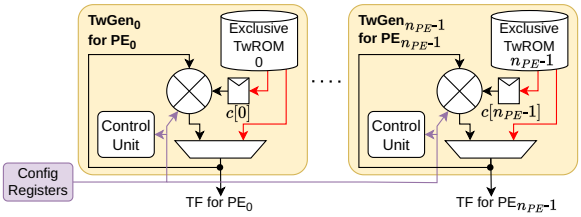

Figure 2: Architecture of our RoutOpt TwGen module.

multiplier (lines 9-11). This allows a streamlined computation of the TF sequence.

After a certain stage s is completed, we use a very similar approach for TFG in stage $s + 1$. The registers $c[i]$ are initialized with new values from TwROM, and the multiplier pipeline is again filled with twiddle factors obtained from TwROM as indicated in Tab. 1. To avoid pipeline stalls during stage transition, loading $c[i]$ for stage $s + 1$ is performed during the last steps of stage s (lines 12-13 in Alg. 5). This strategy allows a fully pipelined TFG without stall cycles.

Fig. 2 shows the hardware architecture of our TwGen module following the RoutOpt strategy. The red arrows show data flow during the stage initialization to fill the multiplier pipeline from TwROM. In addition, the constant multiplicands $c[i]$ are also obtained from TwROM. Once the multiplier pipeline is filled, the generated twiddle factors are fed back along the black arrows until the stage is done. Thereafter, the TwGen module is prepared for the next stage by again retrieving twiddle factors from TwROM. This methodology directly applies to the remaining stages of the DIF NR transformation. Moreover, similar techniques can easily be extended to other transformation types such as DIT and RN transformations.

Table 2: Memory Optimized Twiddle Factor Generation for DIF NR [19].

Stage 0

Initialize registers: $c \leftarrow 4$ from TwROM, $t \leftarrow [0, 16, 32, 48]$ from TwROM

Step	TF for PE_0	TF for PE_1	TF for PE_2	TF for PE_3
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 registers: $c \leftarrow 8$ 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	TF for PE_0	TF for PE_1	TF for PE_2	TF for PE_3
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 $\omega^3 = \omega^1 \cdot \omega^2$.

Algorithm 6 TF generation for PE_i in DIF NR MemOpt.

Input: TwRom: TwROM memory content (shared among all PEs)
Input: d_{Mul}: multiplier latency, n_{PE}: number of PEs
Output: TF_i: Twiddle factor sequence for PE_i

1: $TF_i \leftarrow []$
   /* *Initialization for stage* $s = 0$ */
2: $c \leftarrow$ TwRom.GetData()
3: $t[i] \leftarrow$ TwRom.GetData() ▷ Done serially for all PE
4: **repeat** d_{Mul} **times** ▷ Issue first d_{Mul} multiplics.
5: $\omega \leftarrow$ TwRom.GetData() ▷ All PE get the same ω
6: Start multiplication of $\omega \cdot t[i]$ ▷ Result ready after d_{Mul} cc
   /* *From now on, the multiplier outputs one result each cc and the actual transform starts.* TwRom *no longer needed* */
7: **for** $s = 0$ to $\log_2(N) - 1$ **do** ▷ Stage iterator
8: **for** $j = 0$ to $\frac{N}{2n_{PE}} - d_{Mul} - 1$ **do**
9: $\omega \leftarrow$ multiplier output
10: TF_i.Append(ω)
11: Start multiplication of $\omega \cdot c$ ▷ Result ready after d_{Mul} cc
12: **if** $i = 0$ and $j = N/4/n_{PE}$ **then**
13: $c_n \leftarrow \omega$ ▷ c_n is c in next stage
14: **if** $j = 0$ and i is even **then** ▷ Setup $t[i]$ for next stage
15: $t[i/2] \leftarrow \omega$, $t[(i + n_{PE})/2] \leftarrow \omega$
16: **if** $i = 0$ and $j < 2d_{Mul}$ and j is even **then**
17: Fifo.PutData(ω) ▷ Fill FIFO for next stage
   /* *Last steps of stage* s *and transition to stage* $s + 1$ */
18: $c \leftarrow c_n$ ▷ Setup c for next stage
19: **for** $j = \frac{N}{2n_{PE}} - d_{Mul}$ to $\frac{N}{2n_{PE}} - 1$ **do**
20: $\omega \leftarrow$ multiplier output
21: TF_i.Append(ω)
22: **if** $s < \log_2(N) - 1$ **then** ▷ Fill pipeline for next stage
23: $\omega_n \leftarrow$ Fifo.GetData()
24: Start multiplication of $\omega_n \cdot t[i]$
25: **return** TF_i

3.3 Memory Optimized TFG Configuration

In contrast to RoutOpt, which optimizes for high frequency, our memory-optimized configuration for TFG (MemOpt) aims to minimize the number of twiddle factors stored in ROM. This contrasts with RoutOpt, where a higher number of twiddle factors are needed in memory. Tab. 2 illustrates our corresponding MemOpt optimizations and shows the twiddle factor computation within the first two stages of a DIF NR transformation. The chosen parameters for this example are $N = 128$, $n_{PE} = 4$, and $d_{Mul} = 4$ cycles. Yet, our MemOpt methodology is not restricted to these parameters. In addition, Alg. 6 gives detailed steps for our MemOpt twiddle factor generation, and Fig. 3 shows the hardware architecture.

The execution in our MemOpt design starts with an initialization phase (lines 2-6 in Alg. 6). First, the initialization sequentially loads $1 + n_{PE}$ TFs from the shared TwROM and stores them in the registers c and $t[i]$ within the $TwGen_i$ modules ($0 \le i < n_{PE}$). The corresponding data path is shown in red in Fig. 3. In addition, the initialization is indicated in the top line of Tab. 2 with concrete values for the demonstration parameters. The second part of initialization fills the multiplier pipelines (lines 4-6 in Alg. 6) with d_{Mul} TFs obtained from TwROM. Note that all TwGen modules receive the same TF in this step, and $TwGen_i$ issues a multiplication of the obtained TF by $t[i]$. In the example of Tab. 2, the $d_{Mul} = 4$ TF from TwROM (ω^0 to ω^3) are highlighted in red. In total, the initialization loads $1 + n_{PE} + d_{Mul}$ many TFs from TwROM.

Upon completion of the initialization, the multiplier pipelines in TwGen modules are fully filled, and the actual transformation starts. The first set of produced TFs (step 0 in stage 0 in Tab. 2) is ω^0, ω^{16}, ω^{32}, and ω^{48}. Each TwGen module forwards its generated TF to the PE, which performs the butterfly operation. Simultaneously, the TwGen module feeds the TF back and starts a multiplication with c stored in a register (lines 9-11 in Alg. 6). This computes the remaining TF sequence for stage 0, starting from step 4 in Tab. 2. The corresponding datapath is colored in black in Fig. 3

While the values for the registers c and $t[i]$ in stage $s = 0$ stem from the TwROM, their values for the following stages are obtained from the produced TFs in the prior stage to minimize TwROM consumption. In particular, we collect the values for c and $t[i]$ for stage $s > 0$ from stage $s - 1$ using a generic and scalable method presented in lines 12-15 in Alg. 6. In Tab. 2, the collected TFs for c and $t[i]$ are indicated with ③ and ①, respectively. In addition to c and $t[i]$, d_{Mul} many TFs are needed to fill the multiplier pipeline at the beginning of stage s. These TFs are collected during stage $s - 1$ and fed into a FIFO with d_{Mul} elements (blue datapath in Fig. 3). The FIFO is filled with the TF produced by $TwGen_0$ in even steps, as shown in lines 16-17 in Alg. 6 and marked with ② in Tab. 2. In the

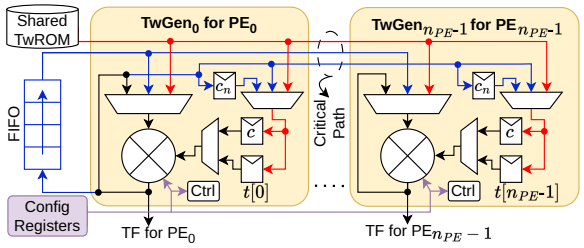

Figure 3: Architecture of MemOpt twiddle factor generation.

transition from stage $s-1$ to stage s, the multipliers in TwGen are provided with the FIFO content during the last d_{Mul} steps of stage $s-1$ (indicated in line 22-24 in Alg. 6 and colored blue in Tab. 2). The multipliers then output the desired TFs after d_{Mul} clock cycles, which is just in time for the beginning of stage s, thereby avoiding pipeline stalls.

The remaining stages follow this procedure to generate the TFs on the fly. We also extend this strategy to other transformation types and parameter sets in our open-source repository. Note that the shared TwROM in our MemOpt architecture is only required during the initialization phase (i.e., only the red TFs in Tab. 2 are from TwROM). Hence, the TwROM only contains a small set of TFs, minimizing the memory requirements. While this effectively lowers the memory consumption, the critical path indicated in Fig. 3 reduces the achievable clock frequency compared to RoutOpt. Thus, our OpenNTT tool provides design-time tradeoffs between latency and ROM consumption of the produced hardware architectures. Sec. 5 presents a detailed comparison of MemOpt and RoutOpt.

3.4 Multiplier Architecture and Parameter Flexibility of TwGen

Our RoutOpt and MemOpt designs allow a high degree of parameter flexibility through synthesis-time and run-time configuration. We refer to Sec. 4.1 for an extensive explanation.

The synthesis-time configuration fixes the polynomial degree N, n_{PE}, and the transformation type. It also instantiates and configures the TwGen and PE modules. The run-time configurability of OpenNTT allows selecting between forward or inverse transformations and the NTT modulus used. Therefore, the host system writes to configuration registers shown in Fig. 2 and 3, which select the transformation direction and the NTT modulus. To change the NTT modulus at runtime, a word-level Montgomery modular reduction for generic NTT-friendly primes is used as in [26]. Moreover, our design allows variable-width moduli by instantiating integer multiplier and Montgomery reduction according to the *largest* modulus. This readily supports smaller moduli, which is required in RNS-based FHE applications [32]. Next to the multipliers in TwGen, the TwROM content also reflects the selected parameter set. Specifically, the TwROM contains one dedicated set of initial TFs, either following MemOpt or RoutOpt, for each modulus. For example, consider the MemOpt DIF NR case in Tab. 2. Therein, a set of 9 TFs is stored in TwROM per modulus. Thus, the TFs for the first modulus are stored in TwROM locations 0 to 8 (base address: 0). The TFs for the second modulus are in locations 9 to 17 (base address: 9), etc. The base address is provided by configuration registers at runtime to select the TF set. The control unit in TwGen adds the offset to select a concrete TF. We use the same strategy in our RoutOpt designs.

4 Overall Design of OpenNTT

This section details the parameter flexibility of OpenNTT and shows our fully parametrized hardware architecture. Finally, we explain the OpenNTT hardware design flow.

4.1 Supported Parameters in OpenNTT

Our OpenNTT tool generates efficient FFT and NTT hardware accelerators for various scenarios. Therefore, OpenNTT shows flexibility in scheme-specific parameters such as polynomial degree N, coefficient types (modular ring and floating-point), and hardware implementation-specific considerations. The latter includes our MemOpt and RoutOpt design strategies to balance performance and memory consumption. Moreover, the number of PEs is adaptable to different resource constraints.

Tab. 3 gives an overview of the design parameters and their supported values. OpenNTT can compile standalone forward and inverse FFT/NTT architectures, wherein only one transformation direction is supported. This saves hardware resources compared to unified architectures (Unif.), which can perform both forward and inverse FFT/NTT transformations. To select between FFT and NTT hardware architectures, users provide a flag to the toolchain, which subsequently instantiates either the modular ring datapath or the complex number datapath. The complex number datapath complies with the IEEE754 floating-point standard, which allows a straightforward adoption of OpenNTT's FFT designs.

Other important configurations supported by OpenNTT include the polynomial degree N and the number of processing elements n_{PE}, as listed in Tab. 3. For NTT designs, the desired primes q and the roots of unity ω can be specified. Alternatively, the OpeNTT tool can overtake the prime and twiddle factor generation according to the specified prime-size $\log_2(q)$. Furthermore, OpenNTT supports multiple primes $q_0 \dots q_{L-1}$ efficiently in one architecture. This capability is especially useful for FHE accelerators where frequent prime changes occur due to residue-number system splitting [8]. A seamless transition between primes can be performed without off-chip communication due to the on-the-fly TFG.

OpenNTT also provides optional coefficient arithmetic support. Thereby, the tool integrates the functionality of performing coefficient-wise addition, subtraction, and multiplication into the hardware unit. The coefficient-wise operations re-use the existing data path within the PEs to perform arithmetic operations. Moreover, a configurable number of polynomials n_{Poly} can be stored in the hardware accelerator. These are addressed via instruction operands and serve as input operands for coefficient-wise operations and transformations.

4.2 Overall Hardware Architecture

OpenNTT incorporates a fully parametrizable hardware architecture shown in Fig. 4. The architecture is carefully designed to reach high clock frequencies, thereby allowing performant FFT and NTT hardware accelerators. Moreover, it supports a wide range of parameter sets relevant to PQC and FHE applications, as shown in Tab. 3.

Table 3: Parameter Space of OpenNTT.

Design Parameter	Supported Values
Standalone forward types	DIT_NR, DIF_NR, M_DIT_NR
Standalone inverse types	DIT_RN, DIF_RN, M_DIF_RN
Unified types (Unif.)	DIT_NR-DIT_RN, M_DIT_NR-M_DIF_RN
Transformation type	FFT, NTT
Optimization strategy	MemOpt, RoutOpt
N	Powers of two, at least 128
n_{PE}	Powers of two (Limited by FPGA resources)
Fixed q	True, False
List of q or $\log_2(q)$	at least 1 element
List of ω	Twiddle factors for each q or auto-select ω
Multiplier latency d_{Mul}	≤ 32 (Easily extendable to higher latencies)
Coefficient arithmetic	No, Mul_Add_Sub
n_{Poly}: Nr polys in ROM	Any (Limited by FPGA resources)

OpenNTT's hardware architecture instantiates n_{PE} many processing elements (PE_0 to $PE_{n_{PE}-1}$), as shown in Fig. 4. Each PE contains one butterfly circuit and either an exclusive or external twiddle factor generation module (TwGen), depending on the used strategy. The TwGen internally uses a dedicated multiplier to compute the twiddle factor sequence for the PE as described in Sec. 3. Furthermore, the TwGen modules either follow our routing-optimized (RoutOpt) or our memory-optimized (MemOpt) option to store precomputed twiddle factors. In RoutOpt, each TwGen module has an exclusive twiddle factor ROM, colored green in Fig. 4. This avoids data exchange across PEs, leading to better routability in dense designs. Contrarily, in the MemOpt strategy, the shared twiddle factor ROM is used (indicated in blue). The single shared ROM distributes twiddle factors to all PEs along the blue datapath in Fig. 4 to initially fill the pipeline. This reduces the memory demand compared to RoutOpt since twiddle factors are not redundantly stored (see Sec. 3.3).

In addition to the twiddle factor, each PE consumes two input coefficients and produces two output coefficients per clock cycle. A total of two memory banks ($2n_{PE}$) are required per PE to provide the necessary memory bandwidth. These banks form the memory subsystem in OpenNTT, as shown on the left in Fig. 4. The memory subsystem can hold a user-defined number of polynomials, n_{Poly}, where each polynomial is stored in one PolySlot spanning over all banks. This layout is indicated in Fig. 4 as PolySlot 0, PolySlot 1, and PolySlot 2. It allows the processing of multiple polynomials, e.g., during coefficient-wise multiplication. OpenNTT uses dedicated control signals to select the targeted PolySlot for each operation.

The control logic, shown bottom right in Fig. 4, controls the overall operation. Depending on the configuration (given by the configuration registers), the control logic accepts instructions for forward and inverse transformation as well as coefficient-wise addition, subtraction, and multiplication. The control logic generates the addresses for the memory subsystem according to the conflict-free memory access pattern and the efficient twiddle factor generation order as shown in Tab. 1 and 2. Subsequently, the ReorderLogic is responsible for storing the PE's results in proper memory locations to ensure a stall-free NTT and FFT computation.

The presented hardware architecture is implemented in SystemVerilog and fully parametrized. The parameterization allows for an adaptation to various parameter sets for FFT and NTT with minimal user intervention.

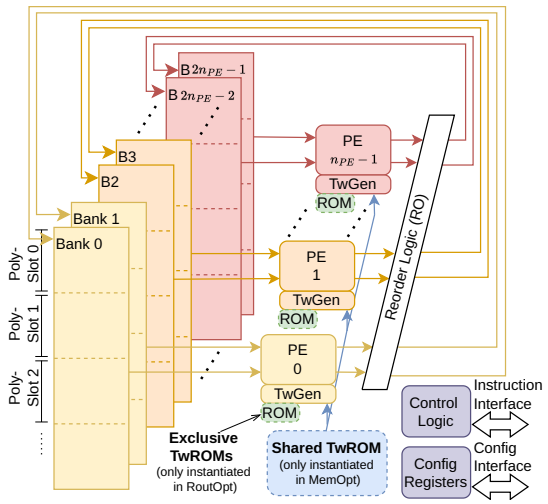

Figure 4: Configurable hardware architecture of OpenNTT.

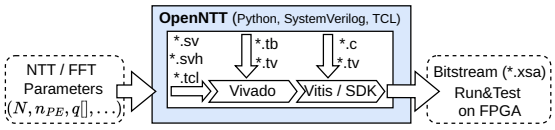

Figure 5: Illustration of OpenNTT's hardware design flow.

4.3 OpenNTT Design Flow

The fully flexible design flow of OpenNTT is illustrated in Fig. 5. The flow covers all stages of NTT and FFT hardware generation based on the input parameter set (left in Fig. 5). Using this parameter set, OpenNTT executes a Python script to generate parameterized SystemVerilog code. In particular, the hardware design (*.sv files in Fig. 5) is fully parametrized, and the Python script generates a configuration file (*.svh) which assigns concrete values to the *.sv parameters in the top module instance. In addition, OpenNTT provides testbenches (*.tb) and test vectors (*.tv) to validate the functionality of the compiled design via Vivado's behavioral simulation.

In the next step, OpenNTT uses a template RTL wrapper with a CPU IP (e.g., Xilinx' Microblaze [4] or ARM core IPs [3]) to interface with the OpenNTT hardware modules. This interface uses memory-mapped IO for the configuration registers and a DMA to stream the large polynomial data. In addition, OpenNTT prepares C code (*.c in Fig. 5), which is executed on the CPU and operates the implemented hardware design on real FPGAs. Finally, the functionality on the FPGA is verified using prepared test vectors (*.tv). OpenNTT automates all these steps, to lower the effort for the user.

Table 4: Comparison of TwROM Consumption for MemOpt and RoutOpt.

N	Nr Twiddle Factors in TwROM			BRAMs for TwROM		
	Stored	RoutOpt	MemOpt	Stored	RoutOpt	MemOpt
2^{11}	8,192	1,728	88	12	4	1
2^{12}	16,384	1,792	96	24	4	1
2^{13}	32,768	1,728	104	48	4	1
2^{14}	65,536	1,792	112	96	4	1

Table 5: Results for RoutOpt / MemOpt NTT Designs on ZCU102.

N, n_{PE}	Freq. MHz	Latency cc	Latency μs	Power W	Energy μJ per NTT
2^{12}, 4	499 / 448	6,189 / 6,218	12.4 / 13.9	4.3 / 3.8	53 / 53
2^{13}, 8	468 / 437	6,698 / 6,727	14.3 / 15.4	7.2 / 6.8	103 / 105
2^{14}, 16	411 / 396	7,210 / 7,239	17.5 / 18.3	13.7 / 12.7	241 / 233
2^{16}, 32	295 / 270	16,426 / 16,455	55.7 / 60.9	23.5 / 20.7	1,309 / 1,259

5 Results of MemOpt and RoutOpt Designs

This section presents results and compares OpenNTT's MemOpt and RoutOpt architectures. All results are obtained from implementation runs on Vivado 2022.2 with default settings. The correctness of each design is validated on real FPGAs.

5.1 Resource Utilization

The memory involved in NTT and FFT designs is a major factor in the overall resource utilization. Hence, reducing the ROM memory for twiddle factors is an important design strategy. To show the effectiveness of our MemOpt and RoutOpt strategies, we consider FHE-typical NTT parameters [2, 32]: We assume 8 different primes q_0 to q_7 each having $\log_2(q_i) = 54$ bits and $n_{PE} = 4$.

Tab. 4 shows the number of twiddle factors (TF) needed in TwROM memory across different N for (1) stored TFs, (2) generated TFs using our RoutOpt, and (3) generated TFs using our MemOpt strategy. For stored TFs, we observe a linear dependency between N and the memory consumption, reaching up to 65.5k TFs in memory for $N = 2^{14}$. In contrast, generated TFs just show a slight memory increase due to the lower number of TFs kept in TwROM. For example, in FHE-typical $N = 2^{14}$-point NTTs, TFG lowers the number of TFs by one to two orders of magnitude compared to stored TFs. Tab. 4 also reports the BRAM consumption. Therein, RoutOpt uses 4 BRAMs to store the required TFs while MemOpt only requires one BRAM. Regarding stored TFs, our MemOpt and RoutOpt reduce the BRAM count by 3× to 96×.

Comparing our RoutOpt and MemOpt strategies, we require up to 20× more twiddle factors in RoutOpt than in MemOpt. For example, for $N = 2^{11}$, RoutOpt stores 1.7k TFs while MemOpt only stores 88 TFs. This is caused by the redundant storing of TFs within TwROM instances of different PEs, as discussed in Sec. 3.2. Nevertheless, the number of TFs in RoutOpt is one order of magnitude less than in stored TFs and does not grow significantly with larger N. The key benefit of our RoutOpt strategy is the improved routability, which allows higher frequencies and hence higher performance.

5.2 Achieved Frequency and Latency in our Designs

As discussed in the previous section, our RoutOpt TFG has a higher memory usage than MemOpt due to the redundant twiddle factors. Yet, the RoutOpt approach increases the achievable clock frequency in dense designs. This is highlighted in Tab. 5, which reports the maximum frequencies for different parameter sets. We observe that our RoutOpt designs achieve up to 51MHz higher clock frequencies than our MemOpt due to the independent TwGen modules. Unlike the frequency, the cycle counts in MemOpt and RoutOpt designs are nearly identical as reported in Tab. 5. MemOpt needs <1% more clock cycles caused by its deeper pipeline taking longer for initial filling. Therefore, the up to 12% lower latency in μs of RoutOpt mainly stems from its higher frequency, whereas the cycle count contributes negligibly to the speedup.

5.3 Power and Energy Consumption in our Designs

Tab. 5 reports the power and energy consumption of RoutOpt and MemOpt designs using different NTT configurations. The power results in Tab. 5 show a 6% to 14% higher power consumption in RoutOpt than in MemOpt. This is mainly caused by up to 11% higher clock frequency of RoutOpt. Considering the energy consumed per NTT, MemOpt requires up to 4% less energy than RoutOpt. Yet, for smaller parameters, the energy consumed per NTT is almost identical.

6 Comparison to Related Works

This section compares the benchmarks of OpenNTT's NTT and FFT designs to related works. We use matching parameter sets and identical FPGAs to allow fair comparisons. OpenNTT is a hardware compilation tool with on-the-fly twiddle factor generation, whereas other NTT design tools mostly use stored twiddle factors. To allow a direct comparison between these orthogonal design strategies, we use the area-time product $ATP = (LUT + 100 \cdot DSP + 300 \cdot BRAM) \cdot Lat$, as defined in [35]. OpenNTT produces either MemOpt or RoutOpt designs. We use MemOpt designs as a default option for comparison. RoutOpt performs best in larger designs occupying a significant portion of FPGA resources. Hence, we provide RoutOpt results for such large and congested designs.

6.1 Comparison to NTT Designs with Stored Twiddle Factors

Tab. 6 presents a comparison to prior works with stored twiddle factors targeting PQC and FHE use cases. The table shows the chosen parameter set, the used FPGA platform, the reported implementation results, and the ATP for each configuration. In addition, the transformation latency and ATP improvement achieved by OpenNTT are given. Note that all comparisons are based on identical FPGAs.

We first compare to [27], which presents a conflict-free memory access pattern for different processing element structures. Moreover, [27] uses stored twiddle factors, causing up to 2× higher BRAM consumption than in OpenNTT's MemOpt configuration.

Table 6: Comparison with NTT Hardware Accelerators using Stored Twiddle Factors.

Work	Platform	Application	N	$\log q$	n_{PE}	LUT / FF / BRAM / DSP	Freq. MHz	Latency NTT cc	Latency NTT μs (impr.)	ATP[a] (impr.)
[27]	Virtex-7[1]	PQC	256	24	1	691 / 451 / 3 / 5	187	1,031	5.5	12
		FHE	4096	24	1	802 / 525 / 7 / 4	185	24,583	132.9	439
		FHE	4096	24	8	5,665 / 3,188 / 8.5 / 33	157	3,079	19.7	227
Our MemOpt	Virtex-7[1]	PQC	256	24	1	1,169 / 1,890 / 1.5 / 10	350	1,060	3.0 (1.8)	8 (1.45)
		FHE	4096	24	1	1,254 / 1,927 / 4 / 8	333	24,610	73.9 (1.8)	240 (1.82)
		FHE	4096	24	8	8,630 / 9,963 / 5 / 64	320	3,106	9.7 (2.0)	160 (1.41)
[15]	ZCU102[2]	PQC	256	24	2	2700 / 2200 / 5 / 8	333	526	1.58	8
	Artix-7[3]	PQC	256	24	1	1,300 / 1,100 / 2.5 / 4	152	1,038	6.85	17
	Virtex-7[4]	FHE	4096	32	4	6,300 / 5,200 / 14 / 24	224	6,158	27.46	354
		FHE	4096	60	1	2,600 / 2,500 / 21 / 26	144	24,590	171.6	1,973
		FHE	4096	60	8	22,100 / 19,500 / 48 / 208	141	3,086	21.91	1,255
Our MemOpt	ZCU102[2]	PQC	256	24	2	1,989 / 3,202 / 3 / 20	450	548	1.2 (1.3)	6 (1.33)
	Artix-7[3]	PQC	256	24	1	1,128 / 1,579 / 2 / 10	220	1,073	4.9 (1.4)	13 (1.26)
	Virtex-7[4]	FHE	4096	32	4	5,973 / 8,030 / 9 / 56	290	6,194	21.4 (1.3)	305 (1.16)
		FHE	4096	60	1	5,996 / 9,845 / 16 / 44	250	24,650	98.6 (1.7)	1,498 (1.32)
		FHE	4096	60	8	34,886 / 45,316 / 18 / 352	225	3,146	14.0 (1.6)	1,055 (1.19)
Our RoutOpt	Virtex-7[4]	FHE	4096	60	8	33,196 / 41,715 / 32 / 352	259	3,117	12.0 (1.8)	939 (1.34)
[21]	Virtex-7[5]	PQC	256	13	1	632 / 402 / 2.5 / 3	335	1,033	3.1	5.2
		FHE	4096	24	16	10,800 / 9,500 / 40 / 112	220	1,545	7.0	238
		FHE	4096	60	1	1,900 / 1,800 / 17 / 42	154	24,585	159.6	1,788
		FHE	4096	60	8	14,100 / 12,500 / 41 / 336	150	3,081	20.5	1,230
Our MemOpt	Virtex-7[5]	PQC	256	13	1	577 / 723 / 2 / 6	300	1,056	3.5 (0.9)	6.3 (0.83)
		FHE	4096	24	16	19,912 / 19,583 / 17 / 128	290	1,570	5.4 (1.3)	205 (1.16)
		FHE	4096	60	1	6,003 / 9,600 / 16 / 44	250	24,650	98.6 (1.6)	1,499 (1.19)
		FHE	4096	60	8	35,262 / 44,107 / 18 / 352	240	3,146	13.1 (1.6)	994 (1.24)
Our RoutOpt	Virtex-7[5]	FHE	4096	60	8	33,380 / 41,760 / 32 / 352	278	3,117	11.2 (1.8)	877 (1.40)
[20]	U50[8]	FHE	65536	52	32	80,000 / 60,000 / 498 / 288	250	16,635	66.5	17,181
Our MemOpt	U50[8]	FHE	65536	52	32	158,059 / 117,392 / 98 / 896	250	16,455	65.8 (1.0)	18,236 (0.94)
Our RoutOpt	U50[8]	FHE	65536	52	32	156,138 / 116,354 / 160 / 896	295	16,426	55.7 (1.2)	16,356 (1.05)
[6]	Virtex-7[5]	FHE	1024	14	2×2	1,196 / 969 / 3 / 12	270	1,308	4.8	16
		FHE	1024	14	8	6,300 / 2,124 / 10 / 27	227	654	2.9	35
Our MemOpt	Virtex-7[5]	FHE	1024	14	4	2,167 / 2,640 / 5 / 24	290	1,327	4.6 (1.1)	28 (0.57)
		FHE	1024	14	8	4,431 / 4,948 / 9 / 48	290	686	2.4 (1.2)	28 (1.23)
[7]	Virtex-7[7]	FHE	1024	32	2×2	4,655 / 4,327 / 9 / 16	333	1,310	3.9	35
		FHE	4096	32	2×2	4,747 / 4,034 / 24 / 16	297	6,168	20.8	281
Our MemOpt	Virtex-7[7]	FHE	1024	32	4	6,082 / 8,250 / 5 / 56	333	1,330	4.0 (1.0)	53 (0.67)
		FHE	4096	32	4	6,236 / 8,626 / 9 / 56	340	6,194	18.2 (1.1)	265 (1.06)
[33]	Virtex-7[1]	FHE	4096	32	4	13,630 / 14,441 / 18.5 / 0	294	6,195	21.1	404
Our MemOpt	Virtex-7[1]	FHE	4096	32	4	6,209 / 8,496 / 9 / 56	310	6,194	20.0 (1.1)	290 (1.39)
[34]	Virtex-7[1]	FHE	4096	60	12	11,856 / 11,222 / 23.5 / 96	183	2,069	11.3	322
Our MemOpt	Virtex-7[1]	FHE	4096	60	8	35,976 / 44,942 / 18 / 352	230	3,146	13.7 (0.8)	1,047 (0.31)
Our RoutOpt	Virtex-7[1]	FHE	4096	60	8	32,158 / 39,458 / 32 / 352	310	3,117	10.1 (1.1)	774 (0.42)
[14]^{SDF}	Virtex-7[6]	FHE	1024	32	10	5,200 / 2,800 / 2 / 36	150	2.1k/1.0k[b]	14.1/6.9[b]	133/65[b]
[14]^{MDC}	Virtex-7[6]	FHE	1024	32	10	6,900 / 5,200 / 2 / 36	150	1.1k/0.5k[b]	7.3/3.5[b]	81/39[b]
Our MemOpt	Virtex-7[6]	FHE	1024	32	8	11,859 / 14,046 / 9 / 112	270	667	2.5(1.4/5.4)	65 (0.6/2)

[a]: Area-time product; ^{SDF}: pipelined with SDF; ^{MDC}: pipelined with MDC; [b]: single / average NTT; [1]:xc7vx690tffg1761-2; [2]: xczu9eg-ffvb1156-2-e; [3]: xc7a200tfbg676-2; [4]: xc7vx485tffg1761-2; [5]: xc7vx690tffg1761-3; [6]: xc7vx485tffg1157-2; [7]: xc7v585tffg1157-3; [8]: xcu50-fsvh2104-2-e

Yet, OpenNTT's DSP consumption is approximately 2× higher due to the additional multipliers for TFG. Our OpenNTT design reaches higher clock frequencies for PQC and FHE parameters due to optimized pipelining. This allows up to 2× and 1.82× performance improvement and ATP reduction, respectively, compared to [27].

The authors of [15] propose an area-efficient NTT architecture with runtime flexible polynomial degrees N. Compared to [15] using stored twiddle factors, OpenNTT's MemOpt designs reach up to 1.4× and 1.7× lower latency for PQC and FHE parameters, respectively. Considering BRAM consumption, OpenNTT's MemOpt designs instantiate up to 2.6× fewer BRAMs than [15] due to the twiddle factor generation in OpenNTT. When considering OpenNTT's RoutOpt design, a higher clock frequency of 259MHz is achieved. Yet, RoutOpt needs more BRAMs compared to MemOpt due to the independent TFG modules. However, the BRAMs in RoutOpt are just partly filled; just 118 out of 512 memory locations are used. This means that up to four primes can be supported without increasing OpenNTT's BRAM consumption in this case.

The paper [21] presents a constant geometry out-of-place NTT hardware architecture. The out-of-place approach allows a uniform control flow while requiring a larger memory consumption. In contrast, OpenNTT uses in-place NTTs and twiddle factor generation, thereby lowering the memory consumption by up to 2.4× in MemOpt. At the same time, OpenNTT uses just 5% to 14% more DSPs than [21]. Thus, OpenNTT reaches a 1.24× improved ATP. Our RoutOpt design further reduces the NTT latency by 1.8× and 1.2× compared to [21] and our MemOpt design, respectively. The recent paper AutoNTT [20] presents an HLS tool for automatic NTT design space exploration and supports iterative, pipelined, and hybrid architectures using stored TFs. Compared to iterative AutoNTT, our MemOpt design reaches similar latency and ATP, while RoutOpt shows a 20% latency reduction.

The work in [6] proposes a radix-flexible hardware design relying on stored TFs. Using the same n_{PE} as [6], OpenNTT's MemOpt reaches 1.1× to 1.2× lower latency. Yet, [6] has a better ATP for 2×2 PEs due to their low area consumption, whereas OpenNTT has a 1.23× lower ATP than [6] for 8 PEs. The authors of [7] introduce optimizations for the Montgomery multiplication and design a 2×2 PE array for NTT. Their design uses stored TFs, which results in a 2.7× higher memory consumption than in OpenNTT. Nevertheless, [7] reaches a lower ATP for $N = 2^{10}$ while OpenNTT has a lower ATP for $N = 2^{12}$, showing the benefits of our TFG in large-degree NTTs. The latency is similar in OpenNTT and [7]. The work in [33] presents an *in-place* constant-geometry NTT dedicated to radix-4 PEs. The presented technique uses stored TFs and five memory banks to store the polynomial. Compared to [33] using LUT-based multipliers, OpenNTT reaches 1.1× and 1.39× lower latency and ATP, respectively. Next, we consider [34], presenting an optimized memory access scheme within a pipelined NTT architecture. [34] uses stored TFs, which ease the NTT control flow. As reported in Tab. 6, [34] instantiates 12 PEs and reaches a lower logic consumption than OpenNTT with 8 PEs. This is due to our TFG and the Shoup reduction algorithm used in [34]. Nevertheless, when scaled to the same n_{PE}, OpenNTT reaches 1.2× and 1.6× better performance for MemOpt and RoutOpt, respectively, while the ATP is higher than in [34]. As the last NTT work with stored TFs, we consider [14]. Unlike OpenNTT, which compiles iterative NTT designs, [14] proposes pipelined NTT designs using SDF and MDC. Pipelined NTT designs allow higher throughputs when multiple sequential NTTs need to be performed. Therefore, Tab. 6 shows two latency values for single NTT latency / average NTT latency. Compared to [14], OpenNTT reaches a latency reduction between 1.4× and 5.4×, although using fewer PEs. However, the resource utilization of OpenNTT is significantly higher than in [14], leading to a worse ATP when considering an average case.

6.2 Comparison to NTT Designs with TFG

We now compare OpenNTT to related works that use TFG. Tab. 7 shows the according benchmarks. The work in [11] presents an NTT accelerator dedicated to FHE applications. The accelerator supports FHE-typical large polynomials with $N = 2^{16}$ and $n_{PE} = 32$ PEs. Since [11] and OpenNTT use TFG, the results are similar for our MemOpt configuration; MemOpt shows 7% lower latency but 5% higher ATP. OpenNTT's RoutOpt shows a latency reduction by 1.4× and 1.3× compared to [11] and our MemOpt design, respectively. Yet, this speedup demands a higher BRAM consumption, leading to a slightly lower ATP improvement of 1.14× concerning [11]. The authors of [16] target large-scale FHE parameter sets while reducing area consumption. Their constant-geometry NTT methodology requires stall cycles between transformation stages to flush the pipeline. In addition, [16] only instantiates one TwGen module for two PEs, which lowers the area consumption but increases latency. This explains the 36× / 28× performance / ATP improvement of OpenNTT over [16] and underlines the benefit of stall-free transformations. Another work using TFG is [25], which is a full-fledged FHE accelerator with high BRAM consumption for key storage. This causes an up to 2.57× and 1.4× higher ATP and latency, respectively, in [25] than in OpenNTT's MemOpt and RoutOpt designs.

6.3 Comparison to FFT Designs

Finally, we compare OpenNTT's FFT designs to related works; the according benchmarks are shown in Tab. 7 (bottom). The authors of [18] design an FHE accelerator for client-side operations in the CKKS scheme [8]. Therein, an FFT transformation is needed, which takes 267μs. OpenNTT reduces the latency for the same parameter set by 1.2×. A comparison of resource consumption and ATP is hard since [18] is an end-to-end FHE accelerator while OpenNTT just provides FFT designs. Nevertheless, OpenNTT lowers memory consumption by 2.5× and LUT consumption by 40%. The work in [24] presents a hardware-software co-design for accelerating the FFT in the Falcon PQ signature scheme. [24] instantiates one floating point adder and multiplier in hardware, which is iterated several times for one complex multiplication. This explains the low area consumption. Yet, compared to [24], OpenNTT reaches 15.9× and 14× lower FFT latency for $N = 256$ and $N = 512$, respectively. This benefit of OpenNTT is due to the 1.7× higher frequency achieved and due to the pure hardware implementation of FFT, whereas [24] follows a hardware-software co-design approach. OpenNTT also shows a moderate reduction in memory consumption and a slight increase in logic consumption, therefore, yielding an ATP improvement of up to 7.42×.

Table 7: Comparison to NTT Hardware Accelerators with Generated Twiddle Factors and FFT Designs.

Work	Platform	Application	N	$\log q$ / Precision	n_{PE}	LUT / FF / BRAM / DSP	Freq. MHz	Latency NTT / FFT cc	Latency NTT / FFT μs (impr.)	ATP[a] (impr.)
NTT Designs with Generated TF										
[11]	ZCU102[2]	FHE	65536	52	32	149k / 91k / 137 / 564	200	16,776	83.9	20,675
Our MemOpt	ZCU102[2]	FHE	65536	52	32	157k / 117k / 98 / 896	210	16,455	78.4 (1.1)	21,653 (0.95)
Our RoutOpt	ZCU102[2]	FHE	65536	52	32	156k / 116k / 160 / 896	265	16,426	62.0 (1.4)	18,191 (1.14)
[16]	Virtex-7[5]	FHE	32768	32	32	36k / 51k / 88 / 392	240	337,272	1,405.3	142,216
Our MemOpt	Virtex-7[5]	FHE	32768	32	32	73k / 55k / 65 / 384	200	7,728	38.6 (36)	5,073 (28)
[25]	U250[9]	FHE	16384	54	16	76k / 57k / 496 / 360[b]	200	7,200	36.0	9,406
Our MemOpt	U250[9]	FHE	16384	54	16	71k / 71k / 50 / 544	250	7,239	29.0 (1.2)	4,059 (2.32)
Our RoutOpt	U250[9]	FHE	16384	54	16	69k / 69k / 80 / 544	290	7,210	24.9 (1.4)	3,667 (2.57)
FFT Designs										
[18] (TFG)	Kintex-7[7]	FHE	8192	double	1	20,728 / 17,647 / 82.5 / 100	200	53,400	267	14,813
Our MemOpt	Kintex-7[7]	FHE	8192	double	1	15,086 / 14,607 / 33 / 88	235	53,292	226.8 (1.2)	7,662 (1.93)
[24] (stored TFs)	ZCU104[10]	PQC	256	double	1	8,396 / 2,526 / 9.5 / 9	200	10,236	51.2	622
		PQC	512	double	1	8,396 / 2,526 / 9.5 / 9	200	19,800	99	1,202
Our MemOpt	ZCU104[10]	PQC	256	double	1	14,922 / 13,838 / 8 / 88	333	1,068	3.2 (15.9)	84 (7.42)
		PQC	512	double	1	15,027 / 13,819 / 8 / 88	333	2,355	7.1 (14.0)	185 (6.48)

[a]: Area-time product; [b]: One RPAU; [2]: xczu9eg-ffvb1156-2-e; [5]: xc7vx690tffg1761-3; [7]: xc7k325tffg900-2; [9]: xcu250-figd2104-2L-e; [10]: xczu7ev-ffvc1156-2-e

7 Discussion

7.1 Other Twiddle Factor Generation Approaches

Although existing works propose on-the-fly TFG architectures [11, 16, 18, 25], most of them are dedicated to one or a few parameter sets and a single transformation type. Generalizing these designs to any configuration and presenting a unified design tool with TFG is challenging.

For example, the authors of [11] present a TFG method for 8×4 PE arrays, which require 15 twiddle factors (TF) in each cycle. They follow a similar approach to OpenNTT and instantiate 15 TwGen modules, each delivering one TF sequence. Their TwGen modules fill the multiplier pipelines using a twiddle factor ROM, which requires storing 15×25 = 375 TFs. This is 5× more than our MemOpt TFG storing 75 TFs for the same configuration. In addition, [11] packs TFs for all PEs into a single centralized memory, whereas our RoutOpt strategy allows *distributed* placement of the TwROM to improve the frequency by 32.5% over [11] (see Tab. 7). Yet, our RoutOpt needs 4.8k TFs in TwROM, which is more than in [11]. Note that [11] only uses modular multipliers with 21 cc latency and $N = 2^{16}$-point DIT forward and DIF inverse NTTs. [11] does not discuss how the presented methodology can be extended to other parameter sets, while OpenNTT directly supports a wide range of configurations.

The paper [16] proposes an out-of-place constant-geometry NTT with TFG and a variable number of PEs. Constant geometry NTT follows the same memory access pattern in each stage, which simplifies the processing order. Moreover, [16] only instantiates $\lceil \frac{n_{PE}}{2} \rceil$ TFG modules to save area at the cost of pipeline stalls and increased latency. [16] stores $2(\log_2(N)+1)\lceil \frac{n_{PE}}{2} \rceil$ TFs in ROM per modulus, whereas MemOpt, also reducing resource consumption, stores $1 + n_{PE} + d_{Mul}$ TF while allowing a fully pipelined design.

Another interesting work is [17], which proposes a hybrid solution between stored and on-the-fly generated TFs. [17] splits each NTT TF into two chunks and stores them separately in memory. The chunks are combined during runtime, yielding the desired TF sequence. This reduces the ROM demand but still has a linear dependency on the polynomial degree, unlike in OpenNTT. The open-source work in [18] uses TFG but requires pipeline stalls between transformation stages. This increases the latency of NTT and FFT, which is not the case in OpenNTT thanks to our fully pipelined TFG and conflict-free memory accesses. Other works [25] do not detail their TFG architecture, making a comparison of methodology hard.

Overall, our generic methodology of on-the-fly TFG complements existing works. In addition, OpenNTT extends to stall-free transformations for any relevant NTT and FFT configuration within a unified hardware generation tool.

7.2 Other Conflict-Free Memory Access Strategies

There exist different approaches to conflict-free memory access patterns in the literature. One of these efforts is [6] proposing a conflict-free memory access pattern for multi-PE and higher radix NTTs. However, while ensuring conflict-free memory accesses, the processing order in [6, Alg. 4] requires TFs in bit-reversed order. For example, in the last stage of a radix-4, $N = 256$-point DIT NR NTT, [6] needs a non-linear TF sequence of $\omega^1, \omega^{65}, \omega^{33}, \omega^{97}, \ldots$ which is hard to compute on the fly. Thus, [6] precomputes and stores all TFs in random-access ROM, thereby allowing straightforward access to any required TF. Similarly, the work in [7] improves the radix-4 processing flow compared to [6]. Yet, [7] also stores TFs in ROM, which relaxes the demands on processing order and memory access pattern. The work in [27] introduces a generic processing flow for varying PE array structures (including single PE

and radix-4 designs). Also [27] stores TFs in ROM, which avoids special handling of the TF generation. Similarly, [33] uses an in-place constant-geometry NTT which enhances uniformity of memory accesses across stages. [33] also keeps the TFs in ROM, thereby simplifying the control flow constraints. The work [34] targets a pipelined NTT design and presents an optimized memory access pattern. Nevertheless, [34] uses stored TFs, and the control flow for pipelined NTT do not directly apply to our iterative approach.

In contrast to [6, 7, 27, 33, 34], where ROM lookups easily obtain any TF, our design offers on-the-fly TFG. This significantly reduces the memory demand but introduces additional constraints on the processing order, which is not the case when using stored TF with ad-hoc loading from random-access ROM. Our novel processing order (discussed in Sec. 3.1 and detailed in [19]) obeys these constraints and ensures an efficient TFG pattern which contrasts with the TF sequence in [6, 7, 27]. In addition, [6, 7] only target merged DIT NR forward and DIF RN inverse NTTs. Yet, our unified NR and RN flows in Sec. 3 readily support all relevant NR/RN, DIT/DIF, and plain/merged NTT/FFT combinations.

7.3 Applicability to Higher Radices

Our current OpenNTT design supports a variable number of radix-2 PEs. Yet, extending our MemOpt and RoutOpt concepts to higher radices is possible. For example, in radix-4, three TFs per PE are required in each cycle. These TFs can be computed by instantiating three TwGen modules, each serving one TF sequence. Extending our conflict-free yet TFG-friendly processing order to high radix designs is another interesting topic, which we leave open for future investigations.

7.4 Side-Channel Security

The NTT and FFT algorithms and twiddle factors are public. Since our RoutOpt and MemOpt strategies compute the public twiddle factor sequence in a secret-independent manner, our methodology does not introduce timing side-channels. However, mitigating power side channels is beyond the scope of this work. Nevertheless, techniques like arithmetic masking may instantiate multiple OpenNTT-generated modules, where each instance processes one independent share. Thus, OpenNTT also supports the design process of masked hardware.

8 Conclusion

In this paper, we proposed an extended version of our OpenNTT tool that generates efficient FFT and NTT hardware accelerators with on-the-fly TFG. OpenNTT combines efficient TFG and conflict-free memory accesses through our generic processing order. Moreover, we presented the RoutOpt and MemOpt optimizations, where MemOpt effectively reduces the ROM memory consumption and RoutOpt trades slightly larger memories for higher clock frequencies. The comparison to other NTT and FFT designs showed competitive results and speedups in ATP and latency, respectively.

References

[1] [n. d.]. OpenNTT source code. https://github.com/flokrieger/OpenNTT.

[2] Martin Albrecht, Melissa Chase, Hao Chen, Jintai Ding, et al. 2019. Homomorphic Encryption Standard. Cryptology ePrint Archive, Paper 2019/939.

[3] AMD Inc. [n. d.]. AMD Zynq 7000 SoCs. https://www.amd.com/en/products/adaptive-socs-and-fpgas/soc/zynq-7000.html. Acessed April 30, 2025.

[4] AMD Inc. 2024. MicroBlaze Processor Reference Guide (UG984). https://docs.amd.com/r/en-US/ug984-vivado-microblaze-ref?tocId=kapC3do5RYcn10xvXtz0Ig.

[5] 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 EuroS&P*. 353–367.

[6] Xiangren Chen, Bohan Yang, Shouyi Yin, Shaojun Wei, and Leibo Liu. 2021. CFNTT: Scalable Radix-2/4 NTT Multiplication Architecture with an Efficient Conflict-free Memory Mapping Scheme. *IACR Trans. Crypt. Hardw. Embed. Syst.* 2022, 1 (Nov. 2021), 94–126. https://doi.org/10.46586/tches.v2022.i1.94-126

[7] Zeming Cheng, Bo Zhang, and Massoud Pedram. 2024. A High-Performance, Conflict-Free Memory-Access Architecture for Modular Polynomial Multiplication. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.* 43, 2 (2024), 492–505. https://doi.org/10.1109/TCAD.2023.3316988

[8] Jung Hee Cheon et al. 2019. A full RNS variant of approximate homomorphic encryption. In *Select. Areas in Cryptogr.* Springer, 347–368.

[9] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic Encryption for Arithmetic of Approximate Numbers. In *Adv. in Crypt. – ASIACRYPT 2017*. Springer International Publishing.

[10] 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 Trans. Crypt. Hardw. Embed. Syst.* (2018), 238–268.

[11] Phap Duong-Ngoc, Sunmin Kwon, Donghoon Yoo, and Hanho Lee. 2023. Area-Efficient Number Theoretic Transform Architecture for Homomorphic Encryption. *IEEE Trans. Circuits Syst. I* 70, 3 (2023). https://doi.org/10.1109/TCSI.2022.3225208

[12] Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Prest, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. 2024. Falcon - Fast-Fourier Lattice-based Compact Signatures over NTRU. https://falcon-sign.info/.

[13] 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*. Springer, 83–105.

[14] Florian Hirner, Ahmet Can Mert, and Sujoy Sinha Roy. 2024. Proteus: A Pipelined NTT Architecture Generator. *IEEE Trans. VLSI Syst.* (2024), 1–11. https://doi.org/10.1109/TVLSI.2024.3377366

[15] Xiao Hu, Jing Tian, Minghao Li, and Zhongfeng Wang. 2023. AC-PM: An Area-Efficient and Configurable Polynomial Multiplier for Lattice Based Cryptography. *IEEE Trans. Circuits Syst. I* 70, 2 (2023). https://doi.org/10.1109/TCSI.2022.3218192

[16] Jingwen Huang, Chiayi Kuo, Sihuang Liu, and Tao Su. 2024. An Area-Efficient and Configurable Number Theoretic Transform Accelerator for Homomorphic Encryption. *Electronics* 13, 17 (2024). https://doi.org/10.3390/electronics13173382

[17] Sangpyo Kim, Jongmin Kim, Michael Jaemin Kim, Wonkyung Jung, John Kim, Minsoo Rhu, and Jung Ho Ahn. 2022. BTS: an accelerator for bootstrappable fully homomorphic encryption. In *49th Ann. Intern. Symp. on Comp. Arch.* ACM. https://doi.org/10.1145/3470496.3527415

[18] 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 *Design, Autom. & Test in Europe Conf. & Exhib.*

[19] Florian Krieger, Florian Hirner, Ahmet Can Mert, and Sujoy Sinha Roy. 2025. OpenNTT - An Automated Toolchain for Compiling High-Performance NTT Accelerators in FHE. In *43rd IEEE/ACM Inter. Conf. on Comput.-Aided Design*. association for computing machinery, Article 13, 9 pages. https://doi.org/10.1145/3676536.3697123

[20] Dilshan Kumarathunga, Qilin Hu, and Zhenman Fang. 2025. AutoNTT: Automatic Architecture Design and Exploration for Number Theoretic Transform Acceleration on FPGAs. In *2025 IEEE 33rd Ann. Intern. Symp. on Field-Programm. Custom Comp. Machines*. https://doi.org/10.1109/FCCM62733.2025.00024

[21] 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. VLSI Syst.* 32, 3 (dec 2023), 519–529. https://doi.org/10.1109/TVLSI.2023.3336951

[22] Patrick Longa and Michael Naehrig. 2016. Speeding up the Number Theoretic Transform for Faster Ideal Lattice-Based Cryptography. In *Cryptology and Network Security*. Springer International Publishing, 124–139.

[23] Vadim Lyubashevsky, Daniele Micciancio, Chris Peikert, and Alon Rosen. 2008. SWIFFT: A Modest Proposal for FFT Hashing. In *Fast Software Encryption*, Kaisa Nyberg (Ed.). Springer Berlin Heidelberg, 54–72.

[24] Suraj Mandal and Debapriya Basu Roy. 2024. Design of a Lightweight Fast Fourier Transformation for FALCON using Hardware-Software Co-Design. In *Great Lakes Symp. on VLSI* (Clearwater, FL, USA). Association for Computing Machinery, 5 pages. https://doi.org/10.1145/3649476.3660370

[25] Ahmet Can Mert, Aikata, Sunmin Kwon, Youngsam Shin, Donghoon Yoo, Yongwoo Lee, and Sujoy Sinha Roy. 2022. Medha: Microcoded Hardware Accelerator for computing on Encrypted Data. *IACR Trans. Crypt. Hardw. Embed. Syst.* 2023, 1 (2022). https://doi.org/10.46586/tches.v2023.i1.463-500

[26] 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 Des., Autom. & Test in Europe Conf. & Exhib.*

[27] Jianan Mu, Yi Ren, Wen Wang, Yizhong Hu, Shuai Chen, Chip-Hong Chang, Junfeng Fan, Jing Ye, Yuan Cao, Huawei Li, and Xiaowei Li. 2023. Scalable and Conflict-Free NTT Hardware Accelerator Design: Methodology, Proof, and Implementation. *IEEE Trans. Circuits Syst. I* 42, 5 (2023), 1504–1517. https://doi.org/10.1109/TCAD.2022.3205552

[28] NIST. 2022. Announcment Of Quantum-Resistant Cryptographic Algorithms. https://www.nist.gov/news-events/news/2022/07/nist-announces-first-four-quantum-resistant-cryptographic-algorithms.

[29] Rogério Paludo and Leonel Sousa. 2022. NTT Architecture for a Linux-Ready RISC-V Fully-Homomorphic Encryption Accelerator. *IEEE Trans. Circuits Syst. I* 69, 7 (2022), 2669–2682. https://doi.org/10.1109/TCSI.2022.3166550

[30] 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.

[31] Sujoy Sinha Roy, Frederik Vercauteren, Nele Mentens, Donald Donglong Chen, and Ingrid Verbauwhede. 2014. Compact Ring-LWE Cryptoprocessor. In *IACR Trans. Crypt. Hardw. Embed. Syst.*

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

[33] Jianfei Wang, Chen Yang, Yishuo Meng, Fahong Zhang, Jia Hou, Siwei Xiang, and Yang Su. 2025. A Reconfigurable and Area-Efficient Polynomial Multiplier Using a Novel In-Place Constant-Geometry NTT/INTT and Conflict-Free Memory Mapping Scheme. *IEEE Trans. Circuits Syst. I* 72, 3 (2025). https://doi.org/10.1109/TCSI.2024.3483229

[34] Zihang Wang, Yushu Yang, Jianfei Wang, Jia Hou, Yang Su, and Chen Yang. 2025. A Scalable and Efficient NTT/INTT Architecture Using Group-Based Pairwise Memory Access and Fast Interstage Reordering. *IEEE Trans. VLSI Syst.* 33, 2 (2025), 588–592. https://doi.org/10.1109/TVLSI.2024.3465010

[35] Zewen Ye, Ray C. C. Cheung, and Kejie Huang. 2022. PipeNTT: A Pipelined Number Theoretic Transform Architecture. *IEEE Trans. Circuits Syst. II* 69, 10 (2022), 4068–4072. https://doi.org/10.1109/TCSII.2022.3184703

[36] Neng Zhang, Qiao Qin, Hang Yuan, Chenggao Zhou, Shouyi Yin, ShaoJun Wei, and Leibo Liu. 2020. NTTU: An Area-Efficient Low-Power NTT-Uncoupled Architecture for NTT-Based Multiplication. *IEEE Trans. Comput.* 69, 4 (2020). https://doi.org/10.1109/TC.2019.2958334

[37] 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 Trans. Crypt. Hardw. Embed. Syst.* (2020), 49–72.