

# @NTT: Algorithm-Targeted NTT hardware acceleration via Design-Time Constant Optimization

Mohammed Nabeel, Mahmoud Hafez, Michail Maniatakos  
*New York University Abu Dhabi (NYUAD), Abu Dhabi, UAE*

arXiv:2601.17806v1 [cs.CR] 25 Jan 2026

**Abstract**—The Number Theoretic Transform (NTT) is a critical computational bottleneck in many lattice-based post-quantum cryptographic (PQC) algorithms. By leveraging the Fast Fourier Transform (FFT) algorithm, the NTT of a polynomial of degree  $N - 1$  can be computed with a time complexity of  $\mathcal{O}(N \log N)$ . Hardware implementation of NTT is generally preferred over software ones, as the latter are significantly slower due to complex memory access patterns and modular arithmetic operations. Achieving maximum throughput in hardware, however, typically demands a prohibitively large number of butterfly unit instantiations. In this work, we propose @NTT, which exploits the fact that the ring parameters in these algorithms are fixed, enabling design-time constant optimization and achieving the maximum throughput of  $N$ -point NTT per clock cycle with a compact hardware footprint. Our case study on the Dilithium NTT, implemented using the TSMC 28 nm library, operates at a clock frequency of 1.0 GHz with an area of 1.45 mm<sup>2</sup>. On FPGA, the design achieves a throughput-per-LUT that is 5.2× higher than the state-of-the-art implementation.

## I. INTRODUCTION

As advancements in quantum computers gain momentum, widely used public-key cryptography schemes such as RSA and ECC are at risk [1]. To protect the public key infrastructure from such quantum-computer attacks, the National Institute of Standards and Technology (NIST) has started standardizing replacements for RSA and ECC. These replacement algorithms are commonly referred to as Post-Quantum Cryptography (PQC) algorithms.

Most of the currently standardized PQC algorithms selected by NIST for key exchange and digital signatures are lattice-based, relying on the Ring-LWE or Module-LWE problem. In these schemes, elements are represented as polynomials of degree  $N - 1$  over the ring  $R_Q = \mathbb{Z}_Q[X]/(X^N + 1)$ , where  $Q$  is the coefficient modulus. The most complex arithmetic operation in  $R_Q$  is polynomial multiplication. Since polynomial multiplication in the point-value representation can be performed using simple Hadamard multiplication, it is common to transform the coefficient representation of a polynomial to its point-value representation using NTT.

Table I lists the standardized algorithms from NIST that require NTT during computation. The NTT contributes to the majority of the execution time in Kyber and Dilithium. Falcon employs both Discrete Fourier Transform (DFT) and NTT implementations, and together they account for a significant portion of the execution time dominated by transform operations. Accelerating NTT is critical for these algorithms' performance. Hardware acceleration is preferred over software due to the modular arithmetic involved and the complex

**TABLE I**  
 Standardized PQC algorithms that use NTT and their Parameters (NIST 2025).

| Algorithm          | Usage          | Ring Parameters (Q, N) |
|--------------------|----------------|------------------------|
| ML-KEM (Kyber)     | KEM/Encryption | (3329, 256)            |
| ML-DSA (Dilithium) | Signature      | (8380417, 256)         |
| FN-DSA (FALCON)    | Signature      | (12289, 512/1024)      |

memory access patterns during NTT, which make software implementations significantly slower.

To this end, researchers have developed hardware accelerators for these PQC algorithms, consistently identifying the NTT as their most computationally expensive operation [2]–[7]. For example, [4] reports that a 256-point NTT in Dilithium requires 552 clock cycles, while [4] achieves the same operation in 606 clock cycles.

In this work, we present @NTT, a framework designed to achieve high throughput of  $N$ -point NTT computations per clock cycle while maintaining lower power consumption and higher efficiency per unit area for the overall computation. To achieve this, we exploit the fact that the ring parameters of these algorithms are fixed in most cases, except for some schemes like Falcon, which use different values of  $N$  for 128-bit and 256-bit security. Instead of storing them as constants in registers and memories, we define them as synthesis-time constants (*parameter* in Verilog, *generic* in VHDL). This not only enables packing more computation units into a smaller area but also reduces storage requirements and memory access complexity that would otherwise be incurred. The main contributions of the work are as follows:

- We propose @NTT, a framework that pushes the limits of NTT throughput to achieve  $N$ -point NTT computation per clock cycle while maintaining low power consumption and high area efficiency.
- We evaluate @NTT both for ASIC and FPGA targets.
- We optimally decompose each constant multiplier required for the NTT into a minimal set of shifts and adders/subtractors, achieving significantly better optimization than industry-standard synthesis tools.
- We will open-source the artifacts.

## II. BACKGROUND

### A. NTT

The Number Theoretic Transform (NTT) [8] is a generalized form of the Discrete Fourier Transform (DFT). NTT can be seen as the DFT on a ring over a finite field, where

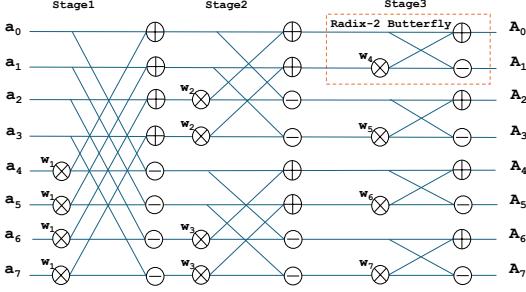


Fig. 1: **NTT Flow.** 8-point NTT flow. Each of the  $\log_2 8$  stages involves  $\frac{8}{2}$  radix-2 butterfly operations. For the fastest execution, one needs to implement all these radix-2 butterflies.

the ring is an integer modulo a prime number  $Q$ . As with the complex root of unity in DFT, for NTT, it is the primitive  $N^{th}$  root of unity  $w$ , such that  $w^N = 1 \pmod{Q}$ , where  $N$  is the polynomial degree. The primitive  $N^{th}$  root of unity is also known as the *twiddle factor*. The NTT enables efficient conversion between the coefficient and point-value representations of polynomials using the FFT algorithm, reducing the complexity of polynomial multiplication from  $\mathcal{O}(N^2)$  to  $\mathcal{O}(N \log N)$ . An 8-point NTT is depicted in Fig. 1. The basic computation unit for NTT is a radix-2 NTT operation, also called the butterfly operation.

For an  $N$ -point NTT operation there will be a total of  $\frac{N}{2} \log_2 N$  butterfly operations. Inverse NTT (iNTT) operation, to convert the point-representation back to the coefficient also can be implemented using forward-NTT flow shown in Fig. 1, but the input elements has to be loaded in bit-reversed<sup>1</sup> order and the twiddle factor for a given butterfly will be the modular inverse ( $w^{-1} \pmod{Q}$ ) of the twiddle that is loaded for forward-NTT. Also, in iNTT, either all input elements or all output elements must be multiplied by  $N^{-1} \pmod{Q}$ .

### B. Butterfly operation

Figure 2 illustrates a typical implementation of the butterfly unit. The butterfly unit performs the radix-2 NTT operation. The bulk of the logic is dedicated to the modular multiplication of  $B \cdot W \pmod{Q}$ . Efficient reduction modulo  $Q$  after the multiplication of  $B \cdot W$  is critical for performance. Two common methods are *Barrett* and *Montgomery reduction* [9], both avoiding costly division but with different trade-offs. For our analysis, we assume *Barrett reduction* for modular reduction after multiplication, since Montgomery reduction requires expensive domain conversion of at least one operand. As shown in Fig. 2, following the regular multiplication (Mult1), *Barrett reduction* requires two additional multiplication operations (Mult2 and Mult3) to perform the modular reduction. Among the various arithmetic circuits, these multipliers constitute the primary contributors to area, power consumption, and critical path delay.

<sup>1</sup>Bit-reversed order refers to reindexing an array such that each index is replaced by the value obtained by reversing the binary representation of that index.

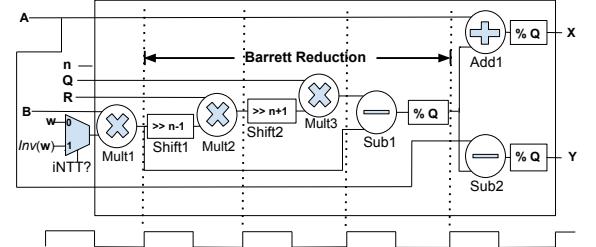


Fig. 2: **Radix-2 Butterfly.** Computes  $A + B \cdot w \pmod{Q}$  and  $A - B \cdot w \pmod{Q}$ .  $A, B \in \mathbb{Z}_Q$ ;  $n = \lceil \log_2 Q \rceil$ ;  $R = \lfloor 4^n / Q \rfloor$

**TABLE II**  
Fixed values in PQC algorithms and their corresponding compute units optimized.

| Fixed Value | Description                 | Compute Unit Optimized |
|-------------|-----------------------------|------------------------|
| $Q$         | Coefficient modulus         | Mult3, % Q             |
| $n$         | $\log_2 Q$                  | Shift1, Shift2         |
| $R$         | $\lfloor 4^n / Q \rfloor$   | Mult2                  |
| $w$         | $w^N = 1 \pmod{Q}$          | Mult1                  |
| $N^{-1}$    | Normalization factor (iNTT) | Multiplier per element |

### III. @NTT FRAMEWORK

This section details the process undertaken to optimize the basic processing element (PE) used for NTT computation, the radix-2 butterfly circuit shown in Fig. 2. The focus is on leveraging operands, which are fixed for the targeted PQC algorithm as determined by the ring parameters (Table I), to enhance hardware accelerators' efficiency, with the goal of fitting more PEs per unit area and thereby improving the throughput of NTT.

In Fig. 2, except for the operands  $A$  and  $B$ , all other inputs can be precomputed since they depend solely on the ring parameters, the polynomial degree ( $N - 1$ ) and the coefficient modulus  $Q$ . Traditionally, these constant values have been stored in registers or memory. However, treating these precomputed values as synthesis-time constants allows for further optimization of the arithmetic units, particularly the multipliers (Mult1, Mult2 and Mult3), which are the dominant contributors to area, power consumption, and critical path delay.

These multipliers become constant multipliers. One option is to rely on the synthesis tool for constant-multiplication optimization. However, our analysis shows that, this introduces significantly higher overhead (10% for ASIC and 20% for FPGA) compared to explicitly decomposing constant multipliers into the minimal number of shifts and add/sub operations. The multiplication of a constant by a variable is generally implemented using a shift-and-add architecture, employing only shift and addition/subtraction operations [10] (Ex:  $13 \cdot w = w \ll 3 + w \ll 2 + w$ , 2 adders and 2 shifts instead of regular multiplication). Note that shifts can be realized using only wires, which incur no hardware cost.

Thus, the optimization problem is to find the minimum number of adders/subtractors required to implement the constant multiplication. Although this optimization problem has been investigated in prior work [11]–[14], @NTT builds upon these foundations and incorporates key ideas from [11] within a new

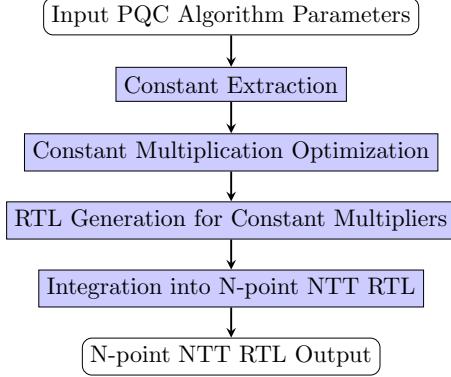


Fig. 3: @NTT flow.

end-to-end framework tailored for the NTT setting used in the target PQC algorithm. The core of the framework is to identify all constants used in the multipliers of the target PQC algorithm, optimize these constant multiplications, generate the corresponding RTL code, and integrate them to form an  $N$ -point NTT RTL design. The high-level steps involved in @NTT are summarized in Fig. 3. The high-level steps involved in @NTT are summarized in Fig. 3.

As discussed in Section II-A, to support the iNTT using the same butterfly hardware, the modular inverse of the twiddle factor must be supplied as the input when operating in iNTT mode, as shown in Fig. 2. In other words, Mult1 performs a multiple-constant multiplication (MCM). Finding an optimal MCM solution, i.e., one that minimizes the number of additions and subtractions, is known to be NP-complete. Motivated by this, @NTT adopts the near-optimal strategy of [11] as a core component within our overall optimization flow. Another constant multiplication required for iNTT is the multiplication of all the input or output elements with  $N^{-1} \pmod{Q}$ .

In addition to the improvements in power, performance, and area, constant multiplication also reduces the critical-path delay, which in turn allows for fewer pipeline stages, further lowering area and power and reducing the initiation interval (II). Typically, pipeline registers are inserted after every multiplication (Fig. 2), since the multiplier constitutes the critical path.

Moreover, twiddle factors are typically generated on the fly using dedicated hardware or stored in internal memories or register files. With constant multiplication optimization applied, however, the twiddle factors are directly merged into the design logic, eliminating the need for their generation or storage and the associated data movement logic, thereby contributing to area and power savings.

Although this work performs the analysis using a modular multiplier based on Barrett reduction to identify constant multipliers and convert them to an optimized form, the @NTT framework can be easily adapted for other modular multipliers.

#### IV. EVALUATION

In this section, we evaluate @NTT for Kyber and Dilithium NTTs. The NTT parameters for Kyber and Dilithium are given in Table I. Both require a 256-point NTT; however, Kyber

**TABLE III**  
Radix-2 butterfly without any constant optimization Vs.  
when implemented using @NTT

| NTT-Design         | ASIC                    |             | FPGA            |             |
|--------------------|-------------------------|-------------|-----------------|-------------|
|                    | Area<br>$\mu\text{m}^2$ | Freq<br>Ghz | LUT/REG/<br>DSP | Freq<br>Mhz |
| Kyber (no opt)     | 1872                    | 1.0         | 253 / 94 / 3    | 356         |
| Kyber (@NTT)       | 754                     | 1.0         | 159 / 109 / 0   | 451         |
| Dilithium (no opt) | 4995                    | 1.0         | 480 / 229 / 5   | 229         |
| Dilithium (@NTT)   | 1532                    | 1.0         | 305 / 182 / 0   | 305         |

**TABLE IV**  
256-point NTT with synthesis-time constant optimization Vs.  
when implemented using @NTT

| NTT-Design           | ASIC                  |             | FPGA                |             |
|----------------------|-----------------------|-------------|---------------------|-------------|
|                      | Area<br>$\text{mm}^2$ | Freq<br>Ghz | LUT/REG/<br>DSP     | Freq<br>Mhz |
| Kyber (Synth-opt)    | 0.70                  | 1.0         | 198295 / 97816 / 0  | 433         |
| Kyber (@NTT)         | 0.62                  | 1.0         | 142719 / 97815 / 0  | 451         |
| Dilithium(Synth-opt) | 1.53                  | 1.0         | 385693 / 187567 / 0 | 325         |
| Dilithium (@NTT)     | 1.45                  | 1.0         | 311500 / 187178 / 0 | 305         |

performs the FFT operation over only seven stages instead of eight, and the twiddle factors used for Kyber are odd powers of the  $N$ th root of unity rather than all powers of the  $N$ th root of unity [16]. Evaluation is done for both ASIC and FPGA targets. For ASIC, we use TSMC 28nm, and FPGA evaluation is performed on AMD Xilinx's XCU50 UltraScale FPGA. For ASIC synthesis, we use Synopsys DC and AMD Xilinx's Vivado for FPGA synthesis. Target frequency is set to 1  $\text{Ghz}$  for ASIC and 500  $\text{Mhz}$  for FPGA. All the designs are verified using Synopsys VCS functional simulation. The baseline for the butterfly RTL is taken from the open-source, silicon-proven design presented in [17], [18].

Table III presents the results of a non constant optimized butterfly unit and compares them with the average area of butterfly units after synthesizing the  $N$  point NTT design generated by the @NTT flow. Although this comparison is somewhat unfair since the non optimized design includes additional logic external to the butterfly such as data movement logic and storage or generation of twiddle factors while the @NTT butterfly has effectively absorbed and merged this functionality into the RTL, we still present this comparison to demonstrate that constant optimization alone already achieves significant saving even before accounting for any additional reductions at the system level.

On ASIC, @NTT reduces the area significantly, from  $1872 \mu\text{m}^2$  to  $754 \mu\text{m}^2$  for Kyber, and from  $4995 \mu\text{m}^2$  to  $1532 \mu\text{m}^2$  for Dilithium, demonstrating the effectiveness of constant multiplication optimization.

On FPGA, @NTT also reduces resource usage, especially LUTs, and eliminates the need for DSP blocks by absorbing constant multiplications into logic. For example, Kyber's LUT usage decreases from 253 to 159, while DSP usage drops from 3 to 0. Similarly, Dilithium's LUT usage drops from 480 to 305, with DSP blocks eliminated. The achieved operating frequency also remains high, resulting in an overall improvement in performance per area efficiency. These results illustrate that @NTT can generate highly area efficient constant-optimized

**TABLE V**  
NTT throughput Comparison for Dilithium and Kyber with the SoTA

| Scheme    | Work | FPGA    | Freq (MHz) | CC  | LUTs   | Latency (μs) | NTT/ms | (NTT/ms)/LUT |
|-----------|------|---------|------------|-----|--------|--------------|--------|--------------|
| Dilithium | [15] | Artix-7 | 180        | 128 | 7451   | 0.7111       | 1406   | 0.19         |
|           | @NTT | XCU50   | 305        | 1   | 311500 | 0.0033       | 305000 | 0.98         |
| Kyber     | [10] | Artix-7 | 250        | 277 | 2466   | 1.108        | 903    | 0.37         |
|           | @NTT | XCU50   | 451        | 1   | 142719 | 0.002        | 451000 | 3.16         |

butterfly units for NTT designs.

The results in Table IV compare a 256-point NTT generated using @NTT against a baseline design relying only on synthesis-time constant optimization. For Kyber, @NTT reduces LUT count by approximately 28% while improving operating frequency by 4.2%. On ASIC, the @NTT design achieves an 11–13% area reduction while still meeting the target frequency of 1 GHz. For Dilithium, @NTT yields a LUT reduction of 19% on FPGA at the cost of a slight frequency drop, while ASIC area remains comparable to the synthesis-optimized version.

Importantly, the resulting designs fits comfortably within high-end FPGA devices such as the XCU50 (17% utilization for Kyber and 36% for Dilithium), highlighting the practicality of the generated RTL for deployment. Overall, these results demonstrate that @NTT provides tangible logic and area savings even when synthesis-time optimizations are already enabled, validating the benefit of performing constant optimization at RTL rather than relying on synthesis tools.

In Table V, we compare the NTT throughput against the state-of-the-art (SoTA) FPGA implementations for Kyber and Dilithium reported in [15]. Despite the increase in LUT usage due to full pipelining and constant-optimized datapaths, @NTT achieves a higher throughput-per-LUT efficiency, with improvements of 5.2× for Dilithium and 8.5× for Kyber.

@NTT achieves substantially higher throughput, producing up to 305,000 NTT/ms for Dilithium and 451,000 NTT/ms for Kyber on FPGA. In ASIC, the design delivers one N-point NTT every nanosecond. This is enabled by a deeply pipelined architecture that implements every NTT stage in hardware, increasing area but maximizing throughput.

## V. CONCLUSION

In this work, we presented @NTT, an end-to-end framework for optimizing Number Theoretic Transform (NTT) architectures targeted to specific post-quantum cryptography (PQC) algorithms. By applying constant multiplication optimization and fully pipelined architecture generation, @NTT produces high-throughput, resource-efficient N-point RTL tailored to the target PQC algorithm. Compared to prior state-of-the-art designs, @NTT achieves substantial improvements in throughput while maintaining efficient area usage.

## REFERENCES

- [1] P. W. Shor, “Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer,” *SIAM Review*, vol. 41, no. 2, pp. 303–332, Oct 1990.
- [2] F. Yaman, A. C. Mert, E. Öztürk, and E. Savaş, “A hardware accelerator for polynomial multiplication operation of crystals-kyber pqc scheme,” in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2021, pp. 1020–1025.

- [3] Z. Ni, A. Khalid, W. Liu, and M. O’Neill, “Towards a lightweight crystals-kyber in fpgas: an ultra-lightweight bram-free ntt core,” in *IEEE International Symposium on Circuits and Systems 2023*. Institute of Electrical and Electronics Engineers Inc., 2023.
- [4] C. Zhao, N. Zhang, H. Wang, B. Yang, W. Zhu, Z. Li, M. Zhu, S. Yin, S. Wei, and L. Liu, “A compact and high-performance hardware architecture for crystals-dilithium,” *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 270–295, 2022.
- [5] N. Gupta, A. Jati, A. Chattopadhyay, and G. Jha, “Lightweight hardware accelerator for post-quantum digital signature crystals-dilithium,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 70, no. 8, pp. 3234–3243, 2023.
- [6] D.-T. Dam, T.-H. Tran, T.-H. Nguyen, T.-T. Hoang, and C.-K. Pham, “Compact falcon fft/ntt accelerator for post-quantum cryptography,” in *2025 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2025, pp. 1–5.
- [7] F. Yaman, A. C. Mert, E. Öztürk, and E. Savaş, “A hardware accelerator for polynomial multiplication operation of crystals-kyber pqc scheme,” in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2021, pp. 1020–1025.
- [8] M. Scott, “A note on the implementation of the number theoretic transform,” in *IMA International Conference on Cryptography and Coding*. Springer, 2017, pp. 247–258.
- [9] A. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. CRC Press, 1996. [Online]. Available: <http://cacr.uwaterloo.ca/hac/>
- [10] H. T. Nguyen and A. Chattejee, “Number-splitting with shift-and-add decomposition for power and hardware optimization in linear dsp synthesis,” *IEEE Transactions on very large scale integration (VLSI) systems*, vol. 8, no. 4, pp. 419–424, 2000.
- [11] Y. Voronenko and M. Püschel, “Multiplierless multiple constant multiplication,” *ACM Transactions on Algorithms (TALG)*, vol. 3, no. 2, pp. 11–es, 2007.
- [12] “Multiplierless constant multiplication.” <https://www.spiral.net/hardware/multless.html>, 2025, accessed: 2025-10-24.
- [13] F. De Dinechin, S.-I. Filip, M. Kumm, and L. Forget, “Table-based versus shift-and-add constant multipliers for fpgas,” in *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*. IEEE, 2019, pp. 151–158.
- [14] V. Lagoon and A. Metodi, “Deriving optimal multiplication-by-constant circuits with a sat-based constraint engine,” in *Proc. ModRef 19th Workshop Constraint Modelling Reformulation*, 2020, pp. 1–6.
- [15] T.-H. Nguyen, B. Kieu-Do-Nguyen, C.-K. Pham, and T.-T. Hoang, “High-speed ntt accelerator for crystal-kyber and crystal-dilithium,” *IEEE Access*, vol. 12, pp. 34 918–34 930, 2024.
- [16] PQClean Developers, “ntt.c – ntt implementation for ml-kem-768,” [https://github.com/PQClean/PQClean/blob/master/crypto\\_kem/ml-kem-768/clean/ntt.c](https://github.com/PQClean/PQClean/blob/master/crypto_kem/ml-kem-768/clean/ntt.c), accessed: 2025-10-25.
- [17] M. Nabeel, H. Gamil, D. Soni, M. Ashraf, M. A. Gebremichael, E. Chielle, R. Karri, M. Sanduleanu, and M. Maniatakos, “Silicon-proven asic design for the polynomial operations of fully homomorphic encryption,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 43, no. 6, pp. 1924–1928, 2024.
- [18] MomaLab, “Cofhee: Hardware accelerator for fhe,” <https://github.com/momalab/CoFHEE>, 2025.