HiFA: A High-Performance and Flexible Acceleration Framework for Large-Size Number Theoretic Transform

QILIN HU, HAOTIAN WANG, and CHUBO LIU, College of Computer Science and Electronic Engineering, Hunan University, Changsha, China and the Ministry of Education Key Laboratory of "Fusion Computing of Supercomputing and Artificial Intelligence", Changsha, China
KEQIN LI, College of Computer Science and Electronic Engineering, Hunan University, Changsha, China and the Ministry of Education Key Laboratory of "Fusion Computing of Supercomputing and Artificial Intelligence", Changsha, China and State University of New York, New Paltz, New York, USA
KENLI LI, College of Computer Science and Electronic Engineering, Hunan University, Changsha, China and the Ministry of Education Key Laboratory of "Fusion Computing of Supercomputing and Artificial Intelligence", Changsha, China

Zero-Knowledge Proofs (ZKP) and Homomorphic Encryption (HE) are crucial for data privacy in applications like cloud, blockchain, and analytics. However, the real-world adoption often faces performance challenges, particularly in the execution of the Number Theoretic Transform (NTT) required for polynomial multiplication involving sizes beyond 2^{20} and large integer widths (e.g., 256 bits). FPGAs offer a promising platform for acceleration, but efficiently implementing large-size NTTs remains difficult due to the limited on-chip resources. The widely adopted four-step NTT method, used to relieve the need for large on-chip memory, introduces performance bottlenecks. Initially, the traditional dataflow NTT architecture may not fully exploit available compute capability, which hinders achieving peak performance. Furthermore, during the matrix transpose phase, the non-sequential access to external High-Bandwidth Memory (HBM) causes inefficiency.

To address these challenges, we introduce HiFA, an FPGA-based automatic accelerator framework designed for high-performance and flexible large-size NTT computations. HiFA utilizes a stacked NTT architecture for high parallelism, maximizing HBM throughput. It supports various decomposed polynomial sizes via a novel reordering module. Additionally, a specialized cyclic shuffle module is integrated to optimize data movement during the matrix transpose step, alleviating random memory access delay. HiFA also provides an automatic

This work was supported by the National Natural Science Foundation of China (Grant Nos. 62441234 and 62502151), the China Postdoctoral Science Foundation (Grant Nos. 2024M750879 and BX20240111), and the Hunan Provincial Innovation Foundation for Postgraduate (Grant No. CX20240420).
Authors' Contact Information: Qilin Hu, College of Computer Science and Electronic Engineering, Hunan University, Changsha, China and the Ministry of Education Key Laboratory of "Fusion Computing of Supercomputing and Artificial Intelligence", Changsha, China; e-mail: hql@hnu.edu.cn; Haotian Wang (corresponding author), College of Computer Science and Electronic Engineering, Hunan University, Changsha, China and the Ministry of Education Key Laboratory of "Fusion Computing of Supercomputing and Artificial Intelligence", Changsha, China; e-mail: wanghaotian@hnu.edu.cn; Chubo Liu, College of Computer Science and Electronic Engineering, Hunan University, Changsha, China and the Ministry of Education Key Laboratory of "Fusion Computing of Supercomputing and Artificial Intelligence", Changsha, China; e-mail: liuchubo@hnu.edu.cn; Keqin Li, College of Computer Science and Electronic Engineering, Hunan University, Changsha, China and the Ministry of Education Key Laboratory of "Fusion Computing of Supercomputing and Artificial Intelligence", Changsha, China and State University of New York, New Paltz, New York, USA; e-mail: lik@newpaltz.edu; Kenli Li, College of Computer Science and Electronic Engineering, Hunan University, Changsha, China and the Ministry of Education Key Laboratory of "Fusion Computing of Supercomputing and Artificial Intelligence", Changsha, China; e-mail: lkl@hnu.edu.cn.

ACM 1936-7414/2025/12-ART54
https://doi.org/10.1145/3771769

Design Space Exploration (DSE) framework that identifies optimal four-step decomposition parameters and generates corresponding hardware configurations. Our experiments show that the FPGA implementation of HiFA achieves an average speedup of 2.97× and up to 7.25× improvement in latency over prior state-of-the-art FPGA solutions. Compared to prior GPU-based methods, HiFA achieves an average energy efficiency gain of 2.24×.

CCS Concepts: • **Hardware** → **Hardware accelerators**; • **Security and privacy** → **Public key encryption**; • **Computer systems organization** → **Data flow architectures**; **High-level language architectures**;

Additional Key Words and Phrases: Design space exploration, FPGA acceleration, homomorphic encryption (HE), number theoretic transform (NTT), zero-knowledge proofs (ZKP)

ACM Reference format:
Qilin Hu, Haotian Wang, Chubo Liu, Keqin Li, and Kenli Li. 2025. HiFA: A High-Performance and Flexible Acceleration Framework for Large-Size Number Theoretic Transform. *ACM Trans. Reconfig. Technol. Syst.* 18, 4, Article 54 (December 2025), 32 pages.
https://doi.org/10.1145/3771769

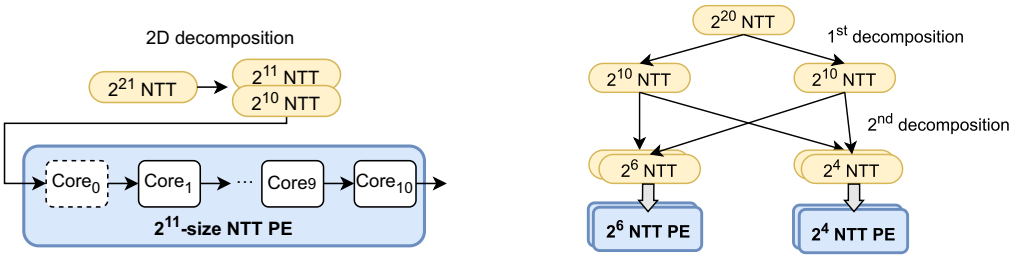
1 Introduction

With the rapid development of the Big Data era, the necessity of protecting data privacy has become increasingly critical. **Zero-Knowledge Proofs (ZKP)** and **Fully Homomorphic Encryption (FHE)** schemes have emerged as essential cryptographic techniques to address this demand [48]. These approaches enable secure computation on encrypted data, ensuring confidentiality while maintaining functionality, thus attracting significant attention across industries such as finance, healthcare, and cloud computing [15, 21]. ZKP protocols allow a prover to convince a verifier of the validity of a statement without revealing the secret information [16]. **Homomorphic Encryption (HE)**, particularly FHE [14], allows arbitrary computations to be performed directly on encrypted data (ciphertext), yielding the result of a decrypted calculation equivalent to the one computed on the original plaintext.

Large-size polynomial multiplication is a major performance bottleneck and a core operation in modern cryptography [25, 45]. Hence, the practical deployment of ZKP and HE schemes is also often constrained by polynomial multiplication, necessitating hardware acceleration. The standard **Number Theoretic Transform (NTT)** algorithm, based on the **Cooley–Tukey (CT) Fast Fourier Transform (FFT)** adapted for finite fields, converts polynomial coefficients into a point-value representation, enabling efficient multiplication, reducing the computation complexity from $O(N^2)$ to $O(N \log N)$. While various hardware platforms can implement NTT, FPGA provides promising NTT acceleration by exploiting fine-grained parallelism and enabling customized, energy-efficient architectures for the inherent structure and various parameters of NTT computations over CPU/GPU implementations [35, 39]. Furthermore, compared to ASICs, the reconfigurability of FPGAs provides design flexibility for different standards or parameters of cryptographic schemes [8].

The ZKP scheme can involve polynomial size exceeding 2^{20}, and particularly, it requires computations over wide integers with bit-widths of several hundred bits, such as 256 bits. Also, a larger polynomial size will highly improve the security and support a deeper multiplicative depth in HE schemes. Such large-size parameters are driven by the need to support complex cryptographic constructs, including secure multi-party computation, verifiable computation in blockchain, and privacy-preserving machine learning in cloud environments. However, implementing large-size NTTs on hardware platforms like FPGA presents significant challenges due to the necessity of using available on-chip computing and storage resources efficiently.

For a polynomial of size N, the standard NTT requires $O(N \log N)$ arithmetic operations and storage for N coefficients and $\frac{N}{2}$ twiddle factors. Different stages in the transform may share the



(a) 2-D decomposition with dataflow NTT PE in PipeZK [46]

(b) Multi-dimensional decomposition with dataflow NTT PE in SAM [40]

Fig. 1. Decomposition structures in prior large-size NTT accelerators.

same twiddle factors. While effective for smaller polynomials (e.g., $N \leq 2^{14}$), the standard NTT becomes impractical for large N (e.g., $N \geq 2^{20}$) on FPGAs, as the on-chip memory (e.g., block RAM) is insufficient to store all coefficients and twiddle factors simultaneously. Moreover, the wide integer widths amplify the computational complexity, requiring extensive modular arithmetic operations that strain FPGA logic resources.

To address these limitations, the four-step NTT algorithm has emerged as a widely adopted approach for large-size NTTs in FPGA and ASIC designs [10, 40, 46]. Unlike the standard NTT, which processes the entire polynomial in a **Butterfly Unit (BU)**-based kernel, the four-step NTT decomposes the transform into four phases: (1) a column-wise NTT on smaller sub-polynomials, (2) a coefficient multiplication and matrix transposition to reorder data, (3) a row-wise NTT, and (4) another matrix transpose. This decomposition significantly reduces the on-chip memory demand. By storing only a subset of coefficients and twiddle factors on-chip, the four-step NTT enables large-size NTTs to be computed on resource-constrained platforms, with the remaining data residing in **High-Bandwidth Memory (HBM)** or DDR off-chip memory. However, the four-step NTT introduces extra overheads, including doubled NTT computations (row- and column-wise), extra modular multiplications, and non-sequential HBM accesses during the transpose phase. Therefore, we need to carefully architect the design and off-chip memory access strategy.

As shown in Figure 1(a), prior accelerators PipeZK leverage the existing NTT dataflow architecture to implement the **Processing Element (PE)** in four-step NTT and use 2D decomposition to decompose the large-size NTT ($N = n_1 \times n_2$) into two equal or close numbers (e.g., $N = 2^{21}$ into $n_1 = 2^{10}$ and $n_2 = 2^{11}$) [46]. However, based on our observation, equal decomposition may not always provide a better performance compared to an imbalanced decomposition (e.g., $N = 2^{21}$ into $n_1 = 2^9$ and $n_2 = 2^{12}$) under specific hardware deployment, thus necessitating a flexible automation framework to explore and generate the design based on the best decomposition. Moreover, because each NTT PE in the dataflow architecture consumes a substantial amount of resources, it is difficult to deploy multiple PEs in parallel, which in turn reduces the overall compute density. To improve parallelism, SAM [40] employs a multi-dimensional decomposition that lowers each PE's resource utilization and enables more PEs to be instantiated concurrently, as illustrated in Figure 1(b). However, it introduces additional element-wise multiplications and off-chip memory accesses, and its dataflow architecture leads to inefficient bandwidth utilization. Therefore, we need a new NTT computation architecture that fully exploits HBM while avoiding the latency penalties of non-sequential off-chip accesses during the matrix-transpose phase.

To overcome the above limitations, we propose HiFA, a high-performance and flexible acceleration framework for large-size NTT on HBM-equipped FPGAs. HiFA employs a stacked NTT architecture with a novel reordering module to support diverse polynomial sizes to maximize parallelism and

explore HBM throughput. It also designs a cyclic shuffle module to streamline data movement during the transpose phase, mitigating random memory access delays. Additionally, HiFA integrates an automated DSE framework to optimize four-step decomposition parameters and generate tailored hardware configurations. In summary, the contributions of HiFA are listed as follows:

(1) HiFA supports an optimized NTT PE that provides four-step NTT execution of imbalanced decomposition while fully utilizing BUs. It also supports indivisible polynomial size in four-step NTT and is equipped with a specific coefficient reorder module, which enables different polynomial size computation.
(2) HiFA provides a cyclic shuffler to avoid the extra latency caused by the random off-chip memory access before matrix transposition based on the evaluation of the impact of random memory access for HBM-FPGAs.
(3) HiFA provides a DSE framework for four-step NTT, which gives insight that the close equally decomposition may not always give the best performance and thus systematically explores the design space and generates the best NTT implementation.
(4) Compared to state-of-the-art FPGA implementations, HiFA achieves an average 2.97× improvement in end-to-end latency and up to 7.25×. In addition, our architecture enhances FPGA routability and enables higher operating frequency. Compared to GPU-based methods, HiFA achieves an average energy efficiency gain of 2.24×.

The remainder of this article is organized as follows: Section 2 reviews the standard and four-step NTT algorithms and outlines the key challenges and opportunities in accelerating large-size NTTs. Section 3 presents HiFA, describing its overall computation flow, the customized NTT PE, the cyclic shuffle module that mitigates random off-chip memory accesses, and the on-chip matrix transpose engine. Section 4 explains our end-to-end DSE workflow. Section 5 reports and analyzes the experimental results. Section 6 surveys related work on large-size NTT acceleration across different hardware platforms: CPUs, GPUs, FPGAs, and ASICs. Finally, Section 7 concludes the paper and discusses future research.

2 Background

NTT can be regarded as a variant of the classical Discrete Fourier Transform within a finite field, such as the residue class ring of integers modulus p, denoted as $\mathbb{Z}_p$. It uses modular addition and multiplication to evaluate the polynomial at Nth roots of unity in $\mathbb{Z}_p$, avoiding complex arithmetic. As a result, it eliminates floating-point errors during calculations, making it particularly suitable for cryptographic applications that require high precision [38, 40]. Despite the algorithmic advantage, NTT computations remain a performance bottleneck in many ZKP and HE workloads. PipeZK shows that the polynomial arithmetic portion of proof generation accounts for ~30% of total proving time, whereas all other polynomial operations contribute less than 2% [46]. In HE schemes, encryption/decryption and homomorphic operations on ciphertexts inherently involve polynomial multiplications, making NTT performance critical, especially during procedures like bootstrapping [2].

2.1 Standard NTT vs. Four-Step NTT

NTT algorithm can be realized into two ways: **Decimation-in-Time (DIT)** and **Decimation-in-Frequency (DIF)**. The DIT formulation is built based on CT butterflies, whereas the DIF formulation relies on **Gentleman–Sande (GS)** butterflies [19, 36]. For a given input coefficient pair (a, b) and a twiddle factor ω^i, the CT and GS butterflies compute the coefficient pairs $\{a + b \cdot \omega^i, a - b \cdot \omega^i\}$ and $\{a + b, (a - b) \cdot \omega^i\}$, respectively.

Analogous to the FFT, NTT is typically implemented as a radix-2 iterative algorithm. The input of length N (usually a power of 2 number) is processed in $\log N$ stages of BUs. The coefficients

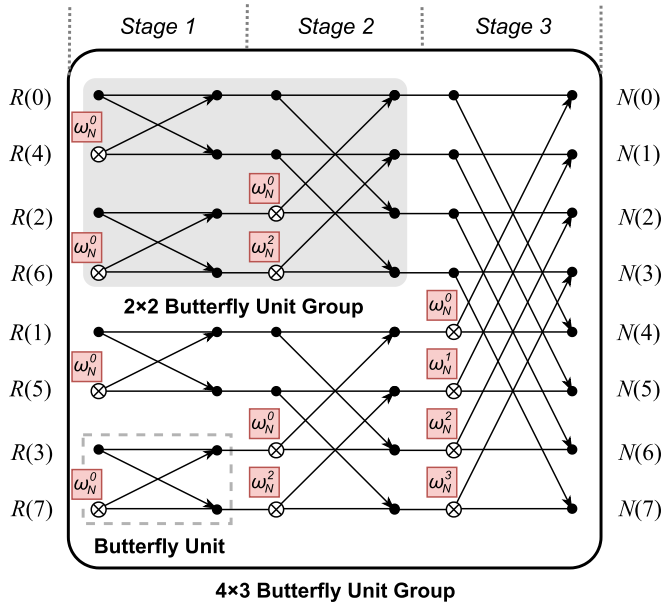


Fig. 2. 8-point NTT dataflow based on Cooley–Tukey butterfly unit.

to be computed in the same BU are decided by a stride that is doubled in each stage, ensuring that after $\log N$ stages, every coefficient has interacted with every other coefficient, producing the transformed output. Figure 2 illustrates a complete 8-point NTT butterfly flow with a bit-reverse input and normal output, where 2 BUs are put in vertical and 2 BUs are put in horizontal for a 2×2 **Butterfly Unit Group (BUG)**. By reusing the 2×2 BUG twice with a specific coefficient shuffler in between, we can compute an 8-point NTT. In this article, we refer to this reuse of the BUG as the *stacked architecture* for convenience. Similarly, 4 BUs are put in vertical and 3 BUs are put in horizontal for a 4×3 BUG. Each BU involves a modular addition, a modular subtraction, and a twiddle factor modular multiplication. With a bit-reversed polynomial as input if the dataflow goes from left to right, NTT is done. The **Inverse NTT (INTT)** is computed in an inverse dataflow: it accepts coefficients in normal order and produces outputs in bit-reversed order. We use $NTT^{DIT}_{R\to N}$ and $INTT^{DIT}_{N\to R}$ to denote these two different dataflows as what [19] does.

The algorithm requires $O(N \log N)$ arithmetic operations and storage for at least N coefficients and $\frac{N}{2}$ twiddle factors, which are constants used in the transform. This makes it efficient for smaller polynomials (e.g., $N \leq 2^{14}$), where on-chip memory can accommodate all data. However, for large polynomials (e.g., $N \geq 2^{20}$) on FPGAs, the standard NTT becomes impractical because (1) the available on-chip memory cannot accommodate all coefficients and twiddle factors simultaneously, and (2) operating on wide (e.g., $\log_2(q) = 256$) integers substantially increases the cost of modular arithmetic, thereby straining FPGA resources.

To address the memory and computation constraints of standard NTT for large-size polynomials, the four-step NTT algorithm leverages the mathematical structure of NTT by decomposing an N-point transform into four phases. As shown in Figure 3, suppose a 1D polynomial N can be decomposed into a 2D matrix $N = n_1 \times n_2$, the steps are as follows:

(1) *Column-Wise NTTs*. Compute n_1-size NTT on all the n_2 columns.
(2) *Twiddle Factor Multiplication*. Multiply the matrix with the corresponding twiddle factors.
(3) *Row-Wise NTTs*. Compute n_2-size NTT on all the n_1 rows.
(4) *Final Transposition*. Transpose the matrix to restore the original 1D polynomial.

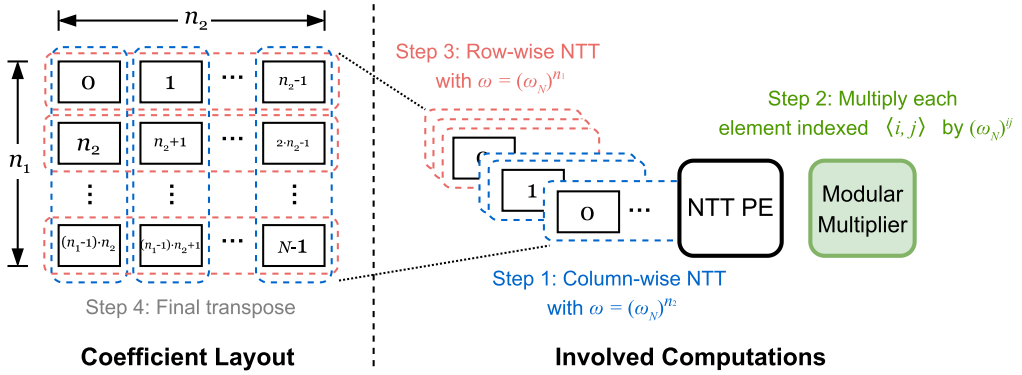


Fig. 3. Four-step NTT algorithm.

Table 1. Comparison with Existing Approaches for Large-Size NTT

Approach	NTT PE Arch.	Decomposition Flexibility	Parallelism	Max Vector Size	Data Movement Overhead
PipeZK [46]	Dataflow	N	N	8	Middle
SAM [40]	Dataflow	Y	Y	32	High
CycloneNTT [1]	Stacked	N/A	N	64	High
HiFA (this article)	Stacked	Y	Y	64	Middle

Y = Yes, N = No, N/A = Not Applicable.

2.2 Motivation of HiFA

Four-step NTT reduces data dependency complexity compared to the standard NTT algorithm. It also enables the parallelism among those sub-polynomial and reduces the number or complexity of twiddle factors needed within the small NTT PEs. While four-step NTT supports the large-size NTTs and reduces on-chip memory demands, it also brings extra challenges. (1) It doubles the number of NTT computations (row-wise and column-wise), amplifying computation overhead. (2) It introduces extra data movement that requires non-sequential access to off-chip HBM during the matrix transpose phase, leading to inefficient memory bandwidth utilization and thus increased delays.

Table 1 lists several NTT accelerators that have been proposed to address the computational demands of large-size NTT. PipeZK [46] is a pipelined architecture for ZKP acceleration, with a subsystem for large-size polynomial computations which uses dataflow architecture to implement four-step NTT. However, PipeZK lacks decomposition flexibility and parallelism because of the fixed and costly implementation of NTT PE. Therefore, the maximum vector size, defined as the amount of data processed per cycle, is highly limited. SAM [40] proposes a multi-dimensional decomposition to support arbitrary NTT sizes. With a finer-grained decomposition, it is able to explore greater parallelism, but introduces double/triple NTT computations and extra data movement depending on the number of decomposition dimensions. CycloneNTT [1] achieves a larger vector size by simply increasing the size of stacked NTT without using four-step NTT. However, it lacks support for arbitrarily large-size NTT, and as the scale of the kernel grows, the interconnection within the architecture will increase design complexity. Because of the usage of standard NTT in CycloneNTT, the off-chip memory needs to be accessed frequently to fetch the necessary inputs and twiddle factors in each stage.

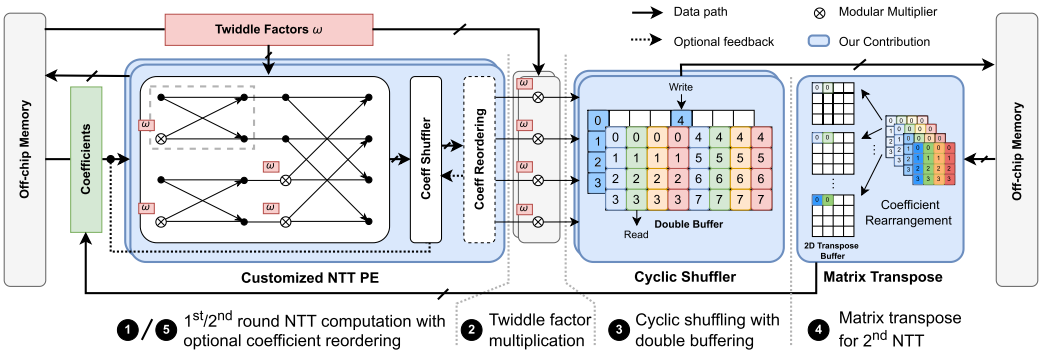


Fig. 4. Overview of HiFA computation architecture. Customized NTT PE is supported by the DSE and the automatic code generator.

To enable the state-of-the-art configuration for large polynomials in the four-step NTT and explore the high-throughput of HBM-FPGAs while avoiding the additional data movement, we propose HiFA, a high-performance and flexible acceleration framework. HiFA provides a fully configurable NTT PE, performs automated DSE, and provides optimal implementations for large-size NTTs.

3 HiFA Design

In this section, we first describe the overview of the hardware implementation of HiFA. Second, we introduce the architecture of the stacked NTT PE utilized in HiFA and the corresponding reordering module to support the correctness of an indivisible NTT. Third, based on the observation of the relationship between HBM bandwidth and random memory access, we elaborate the cyclic shuffler to mitigate additional latency caused by non-sequential off-chip memory access. Fourth, we introduce the matrix transpose designed for stack NTT. Finally, we present several additional optimizations and modular reduction implementation incorporated into HiFA.

3.1 Architecture Overview

Figure 4 illustrates the overview of HiFA's hardware implementation, highlighting three primary modules (shaded in blue):

Customized NTT PE. To support two different small polynomial sizes (n_1, n_2) in the four-step NTT algorithm, it should be able to compute two different sizes of NTT in one hardware according to the optimal decomposition. The coefficient reordering module is designed to expand the flexibility of the NTT PE to support arbitrary polynomial sizes by the same hardware architecture.

Cyclic Shuffler. It shuffles the coefficients that are output from each PE in a cyclic way to ensure that partial coefficients will be stored in sequence according to the length of the double buffer. Therefore, the delay caused by non-sequential access to off-chip memory (i.e., HBM) can be greatly alleviated.

Matrix Transposition. It loads the packed coefficients (e.g., the four numbers in the same color and column in Figure 4, Stage ❹) from the same HBM in sequence, unpacks the coefficients, and rearranges them into the 2D transpose buffer, then sends them to different NTT PEs.

Overall, as shown in Figure 4, HiFA consists of five stages (❶–❺) based on the four phases of four-step NTT: In Stage ❶, coefficients and twiddle factors in the 1st round NTT are loaded from off-chip memory with a bit-reverse order and executed in a series of customized optimal NTT PEs generated by the automation tool in a forward way. In Stage ❷, the coefficients are multiplied by

the corresponding twiddle factor in respective modular multipliers. In Stage ❸, the coefficients are circularly written in a double buffer based on the calculated index and read in sequence, and then, the coefficients are stored in the off-chip memory based on the given non-sequential addresses. In Stage ❹, the coefficients packed in the same memory blocks are loaded in sequence and distributed to the 2D transpose buffer for different NTT PEs. After the matrix transpose, the coefficients are sent back to the customized NTT PEs. In Stage ❺, the 2nd round NTT is computed in an inverse way. Once all of the rows are finished, the matrix is transposed one more time to restore the original NTT sequence. Notice that the bit-reverse operations in between are eliminated by a proper NTT processing chain, as the previous paper mentioned [46], and the matrix transpositions before and after are done in the host if possible to reduce the unnecessary on-chip overhead.

3.2 Customized NTT PE

By leveraging the BUG shown in Figure 2, NTT PEs can be implemented in two primary ways, as described in [23]: a dataflow architecture or a hybrid architecture. In the dataflow architecture, supporting all $\log N$ stages requires $\log N$ layers of BUs. For example, if $\log N = 12$, four 4×3 BUGs will be placed side by side and connected in a deep pipeline. In contrast, the hybrid architecture arranges BUGs vertically and reuses the same 4×3 BUGs across multiple stages, which enables configurable parallelism and resource utilization. Unlike the dataflow or hybrid architecture, our stacked architecture explores a higher external parallelism of the NTT computation by assigning exactly one BUG to each sub-polynomial. By minimizing resource usage per NTT PE, we can instantiate far more PEs in parallel. Furthermore, we enhance this design by configuring each BUG to support two distinct polynomial sizes rather than a single fixed size within the same hardware instance.

NTT Processing Chain. To avoid the unnecessary overhead of on-chip computation, we need to orchestrate the data in a specific way during the host stage. First, the elements in the original polynomial are read in row order and represented as a matrix ($N = n_1 \times n_2$). Then, the matrix is transposed to $N = n_2 \times n_1$, and each row in the matrix is sent to the off-chip memory as the sub-polynomial in a bit-reversed form. Therefore, the 1st round NTT is computed in the $NTT^{DIT}_{R \to N}$ flow, and the output of the 1st round NTT is in normal order. After Stages ❷–❹, the matrix is transposed back to $N = n_1 \times n_2$. Each row in the matrix is sent to the NTT PEs as the sub-polynomial in normal order and computed in the $INTT^{DIT}_{N \to R}$ flow. The output after 2nd round NTT is in bit-reversed form and will be sent back to the off-chip memory. To restore the final result, the rows in the matrix are bit-reversed one more time, and then, it is transposed back to $N = n_2 \times n_1$ and read in row order.

Coefficient Shuffler. Since BUG can only process a certain number of values per cycle among the entire polynomial, the coefficient shuffler is designed to match the correct coefficient pair across different cycles. For example, if $n_1 = 8$ as shown in Figure 2, a 2×2 BUG is able to finish the first two stages within every 4 continuous coefficients. When it comes to Stage 3, the BU needs to wait till it gets the next 4 coefficients. Therefore, it needs a shuffler between two different BUG computation rounds to rearrange the matched coefficients.

Coefficient Reordering. When the $\log N$ stages are not divisible by the layers that can be supported by the BUG, the coefficient reordering is needed to support the correctness of the arbitrary polynomial size. Specifically, for a BUG which has V BUs in vertical and H BUs in horizontal, the front $\left\lfloor \frac{\log N}{H} \right\rfloor \cdot H$ stages are computed by the NTT PE in normal. The last few remaining $\lfloor \frac{\log N}{H} \rfloor \cdot H$ stages are computed normally, but the rest stage(s) in the NTT PE is(are) skipped. At last, the outputs are reordered to satisfy the expected sequence and sent to Stage ❷.

Inspired by the hybrid architecture in AutoNTT [23], we implemented an NTT PE that consists of a single BUG. Using the same hardware structure, we further extended support for different NTT

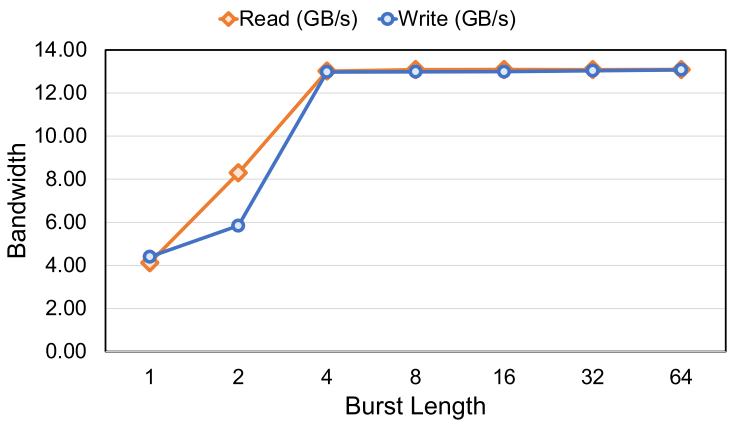


Fig. 5. Effective HBM bandwidth versus burst length (number of consecutive 512-bit beats before a large address change).

computations. Our approach folds multiple stages into a single BUG, which optimizes resource utilization and allows us to instantiate multiple kernels on a single FPGA device. In addition, our design supports INTT without any modifications to the hardware; we simply adjust the parameters in the host code to invoke INTT. This flexibility enables our architecture to accommodate arbitrary polynomial sizes and decomposition strategies, while reusing the BU across all stages to maximize resource efficiency.

3.3 Cyclic Shuffler

HBM partitions its wide external interface into multiple narrow banks (or pseudo-channels) that can be activated and precharged independently. Each activation of a new row incurs a fixed latency overhead; consequently, effective bandwidth improves only when a transaction transfers sufficient data to amortize this cost. The AMD Vitis *HBM Bandwidth Explorations* tutorial [5] characterizes this effect by sweeping transaction sizes from 64 B (one 512-bit beat) to 1,024 B (16 beats) under both sequential and random access patterns. Sequential bursts achieve the device's peak bandwidth of approximately 13 GB/s across all transaction sizes, whereas random bursts begin at roughly one-third of the peak for 64 B requests and increase nearly linearly with transaction size up to around 256 B, where bandwidth approaches saturation. A small additional gain occurs at 512 B, and beyond that, no further improvement is observed.

We reproduced a similar experiment on an HBM2-equipped FPGA platform U280, and the results are summarized in Figure 5. With a single sequential access ($burst\_length = 1$), representing random access, both read and write bandwidths drop below 4.4 GB/s due to the high penalty of random row activations. However, increasing the number of consecutive accesses before a large address change raises the bandwidth almost linearly, reaching approximately 13 GB/s for both reads and writes with four consecutive accesses ($burst\_length = 4$), beyond which fewer gains are observed. The experimental results align with the performance figures reported in the AMD Vitis tutorial [5]. This measurement directly informs our reordering strategy: by grouping every four coefficients into a contiguous block before issuing the transaction, we transform random HBM accesses into minimum four-value bursts, eliminating the per-access penalty of random row activations and sustaining near maximum available bandwidth throughout the workflow.

Cyclic Shuffle Pattern for Partial Sequential Access. The cyclic shuffler is designed to ensure burst-aligned transactions during the transpose phase of four-step NTT, addressing a critical

bottleneck in NTT accelerators by minimizing HBM bandwidth underutilization and reducing latency. It employs a double buffer to manage coefficient reordering between multiple NTTs. The shuffle pattern leverages a cyclic sequence to reorder NTT coefficients, grouping them into contiguous blocks that align with the optimal burst length, as motivated by the bandwidth analysis in Figure 5. We define the coefficient-reordering function $f(i)$ in Equation (1), which maps the ith input coefficient to its new buffer position:

$$f(i) = ((i \times burst\_length) \bmod cyclic\_shuffler\_len) + \frac{i \bmod cyclic\_shuffler\_len}{n_1/(V \times 2)}, \quad (1)$$

where i is the index of the coefficient in its original stream, $burst\_length$ is the number of values we expect to group into each sequential access, $cyclic\_shuffler\_len$ determines the depth of the cyclic shuffler buffer, n_1 denotes the sub-polynomial size, and V is the number of BUs in vertical. This formula maps each coefficient to a new position, ensuring that every $burst\_length$ coefficient is grouped contiguously.

During the transpose phase, our goal is to place coefficients occupying the same relative position across different NTT instances contiguously in HBM. Writing them directly to non-sequential addresses would fragment the traffic into numerous 64-byte transactions, drastically lowering effective bandwidth. To avoid this penalty, we first stage the coefficients from several NTTs in on-chip buffers. We then apply the cyclic shuffle strategy that reorders the buffered data so that coefficients destined for adjacent HBM locations are emitted back-to-back. This re-mapping transforms the inherently irregular access pattern into long, burst-aligned writes, eliminates bank conflicts, and sustains near-peak HBM throughput.

Figure 6 illustrates the cyclic shuffle strategy with an example for $N = n_1 \times n_2 = 16 \times 16$. In Figure 6(a), suppose one NTT PE computes 4 NTTs. Each cycle, there are 4 coefficients input by NTT PE, and the depth for computing one NTT is 4; in total, it needs 16 cycles, where each PE accesses coefficients in the same order. The color indicates that the coefficients belong to four different NTTs, while the numerical values represent their consumption index by the NTT PEs (NTT 1 to NTT 4). After applying the cyclic shuffle pattern, as depicted in Figure 6(b), the coefficients are reordered into a cyclic sequence with a stride of 4, ensuring that each group of four consecutive coefficients is accessed sequentially.

To facilitate efficient memory transactions, the cyclic shuffler sequentially outputs the data from its double buffer and places it into the off-chip HBM memory at addresses computed using the following mapping formula. Specifically, for the ith coefficient, the corresponding HBM address $h(i)$ is determined by:

$$\begin{aligned} h(i) = (i \bmod burst\_length) &+ \frac{i}{cyclic\_shuffler\_len} \times burst\_length \\ &+ \frac{i \bmod cyclic\_shuffler\_len}{burst\_length} \times \frac{n_1}{V \times 2}, \end{aligned} \quad (2)$$

ensuring that groups of $burst\_length$ coefficients are written to contiguous memory locations that align with the burst length, while subsequent groups are separated by a larger stride due to the cyclic nature of the reordering. This approach minimizes random memory accesses, leveraging the sequential burst capabilities of HBM to sustain high throughput during the transpose phase.

3.4 Matrix Transpose

To enable efficient data access in subsequent stages, the reordering of coefficients between column-wise and row-wise NTT computations is a critical step in optimizing performance. Directly performing a matrix transposition on-chip can be resource-intensive, as it demands significant memory

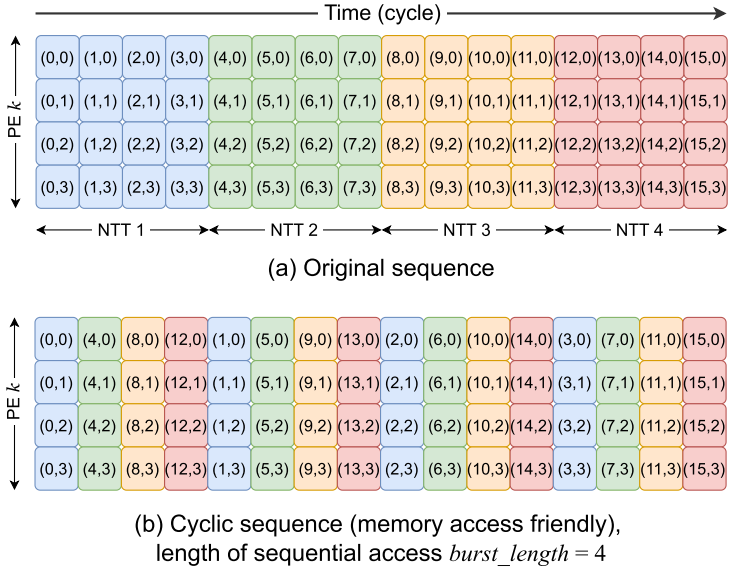


Fig. 6. Cyclic shuffle strategy with 4 buffered NTT for 2×2 BUG NTT PE ($N = 256$, $n_1 = n_2 = 16$). (x, y) represents the processing order of each coefficient in an NTT PE, where x denotes the reception/transmission cycle of the coefficient, and y denotes its consumption index by each NTT PE. The color distinguishes the coefficients between different NTTs.

bandwidth and storage capacity. Conversely, executing the transpose off-chip introduces substantial latency due to the random memory access patterns typically involved in such operations. To address these challenges, HiFA employs an innovative strategy that maximizes the utilization of each read/write operation by leveraging coefficients packed within the same 512-bit HBM burst. This approach, combined with a cyclic shuffle strategy, effectively eliminates the need for a direct matrix transpose, thereby reducing both on-chip memory overhead and off-chip access latency.

Figure 7 provides a detailed illustration of the matrix transpose pattern for an NTT with $N = 256$, utilizing a $\sqrt{N}$-based decomposition for the NTT PE, where $n_1 = n_2 = 16$, and implementing four NTT PEs in parallel. Each entry (x, y) in the figure corresponds to a coefficient indexed consistently within the cyclic shuffler. As shown in Figure 7(a), four coefficients are packed together even though they belong to four different polynomials after the matrix transposition. The read sequence in Figure 7(a) depicts the initial coefficient layout distributed across multiple HBM channels. During each cycle, these four packed coefficients are unpacked and directed to the respective 2D transpose buffers of the four PEs. In Figure 7(b), it enables all four coefficients to be consumed in the same cycle. For example, the four coefficients $(0, 0)$, $(0, 1)$, $(0, 2)$, and $(0, 3)$ are read from a single HBM burst and then written into the four separate NTT buffers. Once every buffer is filled, each NTT PE begins processing its polynomial. By adopting this localized micro-transpose strategy, HiFA alleviates the memory bottleneck of conventional matrix transposition and boosts overall NTT throughput.

3.5 Additional Hardware Optimizations

Optimizing HBM Utilization. In HBM-FPGAs, the availability of HBM ports provides significant memory bandwidth potential. However, to fully harness this capability, careful optimization of HBM utilization is necessary, particularly when considering the interplay between load/store operations

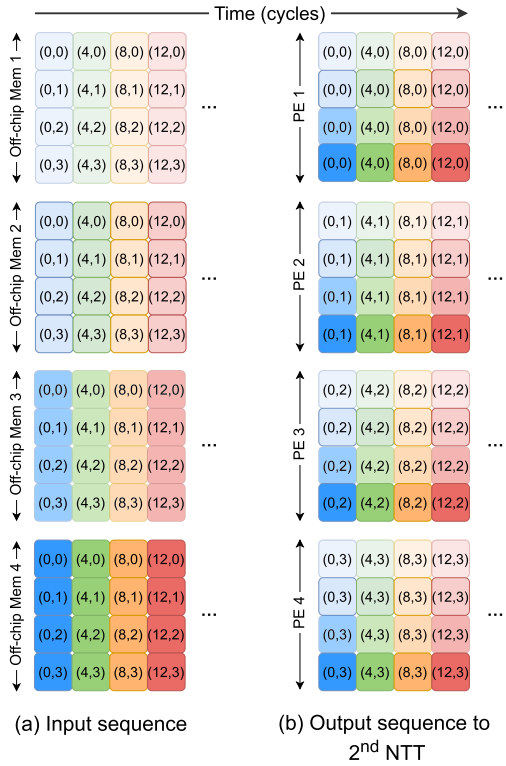


Fig. 7. Matrix transpose pattern among four 2×2 NTT PEs ($N = 256$, $n_1 = n_2 = 16$).

and computation within various NTT PEs. Through detailed profiling, it was observed that the time required for computation is over twice that of the data load/store operations from/to HBM. This imbalance indicates that the system is compute-bound rather than memory-bound, presenting an opportunity to optimize HBM usage without sacrificing overall performance. Therefore, HiFA reduces the HBM ports that are allocated for loading and storing coefficients by half or more, effectively halving the number of HBM ports resource usage.

Enhanced Routing Efficiency. Another critical optimization focuses on the physical design and routing challenges inherent in FPGA implementations. The NTT computation relies heavily on on-chip memory and multipliers, which can introduce significant routing complexity due to the size and interconnect requirements. To mitigate this, the buffer and multiplier are split into smaller, manageable tasks.

The combined optimizations of HBM utilization reduction and buffer/multiplier splitting enhance the efficiency of resource utilization and routing in the HBM-FPGAs, enabling a more efficient and scalable NTT implementation for high-performance cryptographic applications.

3.6 Modular Reduction Implementation

Modular reduction plays a crucial role in NTT implementation as all the underlying computations are performed in modular arithmetic. Barrett reduction [9] and Montgomery reduction [33] are widely employed in research to optimize modular multiplication in hardware in terms of resource and latency. However, we noticed that using the general Barrett and Montgomery reduction for

Algorithm 1: Word-Level Montgomery Reduction Algorithm [19, 31]

1: **Input:** variables a, b, modulus q, where $q = q_H \cdot 2^w + 1$, word size in bits w, number of iterations $I = \left\lceil \frac{\log_2(q)}{w} \right\rceil$
2: **Output:** $c = a \cdot b \cdot R^{-1} \pmod{q}$, where $R = 2^{w \cdot I}$
3: $T \leftarrow a \cdot b$ ▷ $\log_2(q)$-bit × $\log_2(q)$-bit multiplication
4: **for** $i = 0$ to $I - 1$ **do**
5: $T_H \leftarrow T \gg w$
6: $T_L \leftarrow T \pmod{2^w}$
7: $T_2 \leftarrow (2^w - T_L) \pmod{2^w}$
8: $carry \leftarrow \text{bit}_w(T_2) \vee \text{bit}_w(T_L)$
9: $T \leftarrow T_H + (q_H \cdot T_2) + carry$ ▷ $(\log_2(q) - w)$-bit × w-bit multiplication
10: **end for**
11: **if** $T \geq q$ **then**
12: $c \leftarrow T - q$
13: **else**
14: $c \leftarrow T$
15: **end if**
16: **return** c

larger modulus sizes, such as 256 bits, takes a prohibitively expensive amount of resources and leads to higher latency.

To optimize modular arithmetic, we adopt the **Word-Level Montgomery (WLM)** reduction algorithm, as described in [19, 31] and detailed in Algorithm 1. This algorithm avoids direct modular operations by iteratively performing multiplications, additions, and bit-shift operations within the reduction loop (lines 4–10), leveraging the properties of NTT-friendly primes. By utilizing a user-defined word size w, the WLM algorithm decomposes the reduction computation into a series of multiplications, where the number of iterations $I = \left\lceil \frac{\log_2(q)}{w} \right\rceil$ depends on the modulus size $\log_2(q)$ and the word size w. This approach provides fine-grained control over latency and resource utilization, achieving significantly lower resource consumption compared to traditional Barrett or Montgomery reduction methods for the same modulus size.

4 Design Space Exploration

In this section, we provide a comprehensive overview of the design space exploration process for HiFA. We begin by reviewing prior work on DSE methodologies for NTT accelerator. Subsequently, we identify limitations of these approaches that do not adequately address the requirements of our framework. Next, we introduce our proposed DSE framework, accompanied by an explanation of the associated workflow figure and an algorithm that clarifies the implementation details. Finally, we present a detailed analysis of resource and performance estimation, along with an overview of the optional parameters employed in the actual DSE implementation.

4.1 Prior Work on DSE for NTT Acceleration

Both NTTGen [43] and AutoNTT [23] provide design space exploration for standard NTT FPGA accelerations. NTTGen integrates application parameters such as latency, polynomial degree, and a list of prime moduli, along with hardware resource constraints including DSP, BRAM, I/O bandwidth, and platform-specific metadata. To generate design candidates, the DSE systematically enumerates possible configurations of the NTT core. Any candidate meeting all resource constraints is included in the list of valid designs, which are subsequently subjected to synthesis and place-and-route processes. In contrast, AutoNTT begins by employing a DSP resource model to determine the BU organization of NTT PE based on the available DSP resources. It then leverages latency

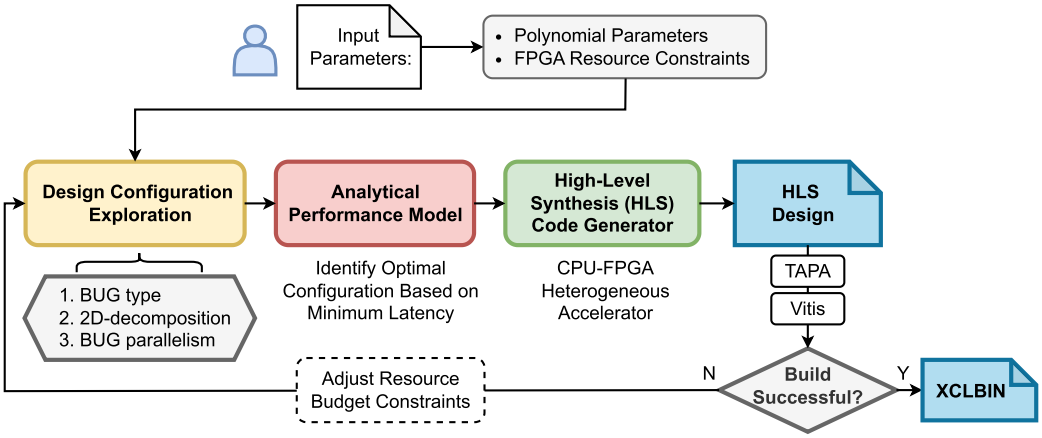


Fig. 8. Overview of DSE workflow.

and resource analytical models to identify the design with the lowest latency. Finally, the selected architecture and configuration are passed to the code generator to produce the HLS code.

SAM [40] derives three primary design parameters through design space exploration in four-step NTT acceleration: the size of each NTT PE, the number of parallel lanes, and the buffer capacity. The DSE seeks to balance compute and memory latencies, thereby adjusting the tradeoffs between resource utilization and parallelism to set the final design parameters. SZKP [12] performs an extensive design space exploration for the ZKP accelerator, encompassing both individual modules and the entire chip architecture. Specifically, it investigates the dimensions of each NTT PE and the number of parallel lanes as critical components of the four-step NTT module.

Inspired by the previous methods, we take polynomial-related parameters and platform resource constraints as inputs and explore the BUG type and parallelism within our design space. Furthermore, based on our observations that imbalanced 2D decomposition outperforms balanced decomposition in certain cases, we extend the design space to include 2D-decomposition parameters, addressing the limits of the existing DSE schemes for four-step NTT approach.

4.2 Proposed Workflow and Detailed Algorithm

Figure 8 illustrates the automated workflow of HiFA. The process begins with two user-specified inputs: polynomial parameters and platform resource limits, which define the search space. These inputs are fed into a DSE engine that evaluates candidate implementations across three dimensions: BUG type, 2D decomposition, and BUG parallelism. Guided by an analytical performance model, the DSE continues iterating until it identifies the configuration that minimizes predicted latency within the given resources. This configuration is then passed to an HLS code generator, which outputs a CPU–FPGA heterogeneous accelerator. If the build is successful, the framework adopts the current configuration. Otherwise, it adjusts the resource constraints and iteratively searches for a new configuration until a successful design is found.

Algorithm 2 outlines the detailed workflow for the proposed design space exploration process. It takes the polynomial size N and bit width W as inputs and produces optimized HLS code as output. Lines 3–5 initialize the necessary values based on these input parameters, setting up the resource budget and computing key design constraints, including the extraction of the DSP limit NUM_{DSP_budget} from R and the DSP cost per modular multiplier $NUM_{multiplier}$ derived from W. This enables the calculation of the maximum number of modular multipliers $M_{multiplier}$. Lines 9–13

Algorithm 2: Design Space Exploration Algorithm for HiFA

1: **Input:** Polynomial parameters (polynomial size N, bit width W), FPGA resource constraints $R_{initial}$ (LUT, FF, DSP, BRAM, URAM)
2: **Output:** Optimized NTT accelerator configuration C_{opt} and HLS code
3: Initialize resource budget $R \leftarrow R_{initial}$ ▷ Set initial budget from user-specified constraints
4: Define design parameters: BUG type T, parallelism P, decomposition dimensions (n_1, n_2)
5: Compute $NUM_{multiplier}$ based on W ▷ DSP cost per modular multiplier, derived from practical implementation
6: **repeat**
7: Select NUM_{DSP_budget} from R ▷ DSP limit specified by the user within resource constraints R
8: $M_{multiplier} \leftarrow \lfloor NUM_{DSP\_budget}/NUM_{multiplier} \rfloor$ ▷ Maximum number of modular multipliers
9: **for** each candidate BUG type T_i **do**
10: Define V and H ▷ Vertical and horizontal dimensions of corresponding BUG
11: $M_{PE} \leftarrow \lfloor M_{multiplier}/(V \times H + 2V) \rfloor$ ▷ Maximum PEs fitting DSP constraints
12: $P_i \leftarrow 2^{\lfloor \log_2(M_{PE}) \rfloor}$ ▷ Practical maximum parallelism
13: **end for**
14: **for** each candidate configuration (T_i, P_i) **do**
15: Compute k decomposition pairs (n_1, n_2) with $n_1 \times n_2 = N$ ▷ k is the number of candidate pairs
16: **for** each candidate decomposition pairs (n_1, n_2) **do**
17: Compute resource consumption R_i (LUT, FF, DSP, BRAM, URAM) for each (T_i, P_i, n_1, n_2)
18: **if** $R_i \leq R$ **then**
19: Compute estimated latency L_i using analytical model
20: Add L_i to valid designs
21: **end if**
22: **end for**
23: **end for**
24: Select $C_{opt} \leftarrow \arg\min_{C \in \text{valid designs}} L(C)$ ▷ Select the lowest latency design
25: Generate HLS code for NTT accelerator using C_{opt}
26: Perform synthesis and build using TAPA/Vitis framework
27: **while** build fails and valid designs remain **do**
28: Remove C_{opt} from valid designs
29: Select next $C_{next} \leftarrow \arg\min_{C \in \text{valid designs}} L(C)$ ▷ Choose the best remaining configuration
30: Set $C_{opt} \leftarrow C_{next}$ ▷ Update the optimal configuration
31: Generate HLS code for NTT accelerator using C_{opt}
32: Perform synthesis and build using TAPA/Vitis framework
33: **end while**
34: **if** all valid designs fail **then**
35: Adjust resource budget $R \leftarrow R'$ (e.g., decrease DSP limits)
36: **end if**
37: **until** build succeeds
38: **return** C_{opt} and HLS code

calculate the candidate BUG types T and their corresponding maximum parallelism P, computing the maximum processing elements M_{PE} as $\lfloor M_{multiplier}/(V \times H + 2V) \rfloor$, and setting P_i to $2^{\lfloor \log_2(M_{PE}) \rfloor}$ for power-of-2 efficiency. Lines 14–23 further refine the exploration by selecting top k decomposition pairs (n_1, n_2) for each candidate pair (T, P), where $\log_2(n_1)$ is set to $\{\lfloor \log_2(N)/2 \rfloor, \lfloor \log_2(N)/2 \rfloor + 1, \lfloor \log_2(N)/2 \rfloor + 2\}$, and $\log_2(n_2)$ is defined as $\log_2(N) - \log_2(n_1)$, followed by resource consumption estimation and validation against R. Lines 24–33 evaluate the optimal design by comparing the estimated latency of all candidate configurations and selecting C_{opt} via arg min. Finally, Lines 34–36 address build failures by adjusting the resource budget (e.g., decreasing DSP limits) and iterating the process from Lines 7–33 until a successful build is achieved.

4.3 Resource and Performance Estimation

We further elaborate on the details of our resource and performance analytical models, as utilized in Algorithm 2. These models offer a lightweight approach for estimating the computational resources and performance.

LUT and FF Resources. Existing research primarily employs machine learning to estimate LUT and FF consumption in general-application RTL/HLS designs [13, 24, 29]. These approaches typically depend on a large database of C-to-FPGA results from diverse benchmarks to support model training. To save the extensive computational resources required for such training, we utilize a lightweight resource analytic model to predict logic utilization within our DSE algorithm. For every parameterized module (e.g., twiddle factor allocator, BUG, coefficient shuffler, cyclic shuffler, and matrix transposition), we first generate a small test set by synthesizing a few instances with different configurations. We then fit a simple linear regression to the resulting (data size, LUT/FF) pairs to capture how resource demand scales with different configurations. At exploration time, the cost of a candidate design is computed by summing the predicted LUT and FF counts of its modules. A ±5% fluctuation is added in the results because the regressions are empirical. These fast and tolerable predictions allow the search engine to prune infeasible points early and keep the exploration runtime within seconds.

DSP Resources. DSPs are consumed by two kinds of modular multiplications: those performed inside the BUs of each NTT PE and the additional twiddle-factor multiplications required by the four-step algorithm. For an FPGA DSP block with a fixed multiplier size of $dsp\_width_1 \times dsp\_width_2$ bits, the DSP slice consumption for a modular multiplier is based on the full multiplication before the WLM reduction loop and partial multiplications in the loop (Algorithm 1). For I multiplications, where the ith multiplication has operand bit widths a_i and b_i, the total DSP slices required is:

$$NUM_{multiplier} = \sum_{i=1}^{I} \left\lceil \frac{a_i}{dsp\_width_1} \right\rceil \times \left\lceil \frac{b_i}{dsp\_width_2} \right\rceil. \tag{3}$$

The overall DSP demand is calculated as:

$$NUM_{DSP} = NUM_{multiplier} \times (V \times H + 2V) \times P. \tag{4}$$

BRAM Resources. BRAM is mainly consumed by three parts: the input/output FIFO, the coefficient shuffler NUM_{Coeff_BRAM}, and the 2D transpose buffer $NUM_{Transpose\_BRAM}$. The input/output FIFO is needed when the number of HBM ports is reduced, determined by both sub-polynomial sizes n_1, n_2, and the H in BUG. The BRAM for the FIFOs and coefficient shuffler is configured with dual-port, enabling read operations on one port and write operations on the other port to ensure continuous dataflow. Similarly, the transpose BRAM is implemented with a dual-port configuration that allows both read and write operations on both ports to avoid bank conflicts. These two components are represented as follows:

$$NUM_{Coeff\_BRAM} = P \times 2V \times \left\lceil \frac{\log_2(q)}{bram\_width} \right\rceil \times \left\lceil \frac{\frac{2n_1}{2V}}{bram\_depth} \right\rceil, \tag{5}$$

$$NUM_{Transpose\_BRAM} = \max\{2V, P\} \times 2V \times \left\lceil \frac{\log_2(q)}{bram\_width} \right\rceil \times \left\lceil \frac{\frac{n_2}{2V}}{bram\_depth} \right\rceil. \tag{6}$$

URAM Resources. To balance on-chip memory utilization, the twiddle factors for the NTT PE and the cyclic shuffler are allocated to URAM blocks. Each twiddle factor URAM block within the TF buffer supports 2 BUs. In the stacked architecture, each NTT stage s requires 2^s twiddle factors. However, depending on the number of vertical BUs in the BUG, the twiddle factors for each stage can be replicated or distributed across multiple buffers. Consequently, based on the polynomial

size, the supported NTT stages, and the number of parallel TF buffers, we denote the number of twiddle factors allocated to each buffer in layer i as TF_BUF_i. The URAM is implemented with a dual-port configuration that supports both read and write on both ports. During data loading to the URAM, both ports are utilized for writing operations. When reading twiddle factors from the URAM, both ports are employed to read data for the BU. The URAM consumption for twiddle factors is calculated as follows:

$$NUM_{TF\_URAM} = P \times \frac{V}{2} \times \left\lceil \frac{\log_2(q)}{uram\_width} \right\rceil \times \sum_{i=0}^{H-1} \left\lceil \frac{TF\_BUF_i}{uram\_depth} \right\rceil. \tag{7}$$

Assuming the number of NTTs shuffled by the cyclic shuffler is c. Based on a dual-port configuration that supports read operations on one port and write operations on the other port, to support continuous data loading across different iterations of the cyclic shuffler, the URAM consumption for the cyclic shuffler is calculated as follows:

$$NUM_{Cyclic\_URAM} = 2 \times\ P \times 2V \times \left\lceil \frac{\log_2(q)}{uram\_width} \right\rceil \times \left\lceil \frac{c \times \frac{n_1}{2V}}{uram\_depth} \right\rceil \tag{8}$$

Latency. In our performance analytical model, we first consider the one-time sub-polynomial processing latency, referred to as the customized NTT PE in Figure 4. Let BUG_PD denote the pipeline depth of a BUG, and let $MULTI\_PD$ denote the pipeline depth of a single modular multiplier. Furthermore, let $CoeffShuffler\_DL_i$ be the delay introduced by the ith shuffler. For a sub-polynomial of size n_1 (with n_2 treated analogously), the latency in terms of clock cycles can be written as:

$$Latency_{n_1} = \sum_{i=1}^{\frac{log\ n_1}{H}-1} \max\left\{\frac{n_1}{2V}, BUG\_PD + CoeffShuffler\_DL_i\right\} + BUG\_PD + \frac{n_1}{2V}. \tag{9}$$

To demonstrate the latency contributions as defined by Equation (9), Figure 9 illustrates the input-to-output timing path. This example is configured with $BUG = V \times H = 4 \times 3$, a sub-polynomial size $n_1 = 2^{12} = 4096$, and a BUG pipeline depth $BUG\_DP = 45$ clock cycles. The delays for $CoeffShuffler\_DL_i$ are defined as $2V = 8$, $(2V)^2 = 64$, and $\frac{n_1}{2V} = 512$ for $i = 1, 2, 3$, respectively. In Stage (a), coefficients are loaded from off-chip memory and transmitted to the NTT PE. The coefficients are first processed by the BUG and subsequently passed to the coefficient shuffler. Stage (b) mirrors the cycle count of Stage (a), as the data loading time $\frac{n_1}{2V}$ exceeds the one-time PE computing duration $BUG\_PD + CoeffShuffler\_DL_1$ and $BUG\_PD + CoeffShuffler\_DL_2$. The overlap occurs when the last cycle of coefficients from Round 1 is shuffled, allowing Round 2 to start as the BUG becomes idle. In Stage (c), the total cycles are determined by the one-time PE computing time, as the processing duration of the PE in Round 3 exceeds the data loading time during this phase. No overlap occurs in this shuffler, as the coefficient shuffler must fully process all data before Round 4 starts. In Stage (d), the BUG processes the final round (Round 4) without additional shuffling, after which the coefficients are written to the subsequent task. The latency for this stage is $BUG\_PD + \frac{n_1}{2V}$.

For a polynomial of size $N = n_1 \times n_2$, the total computational latency comprises two NTT rounds, one extra modular multiplication, and the pipelined matrix transposition. Consequently, the overall latency can be expressed as:

$$Latency_N = \frac{Latency_{n_1} \times n_2 + Latency_{n_2} \times n_1}{P} + MULTI\_PD + \frac{N}{2V \times P}. \tag{10}$$

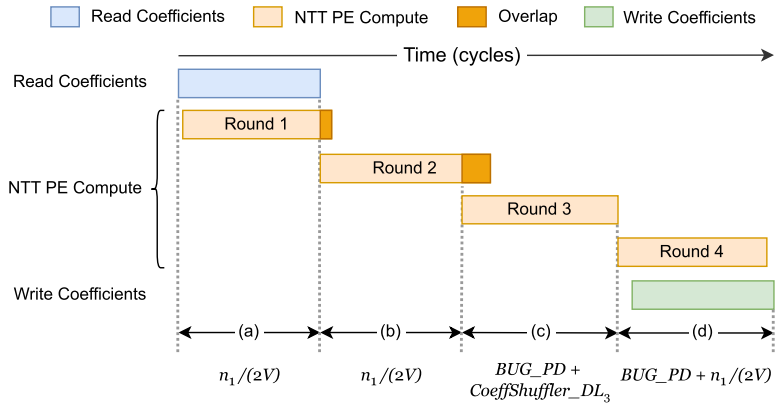
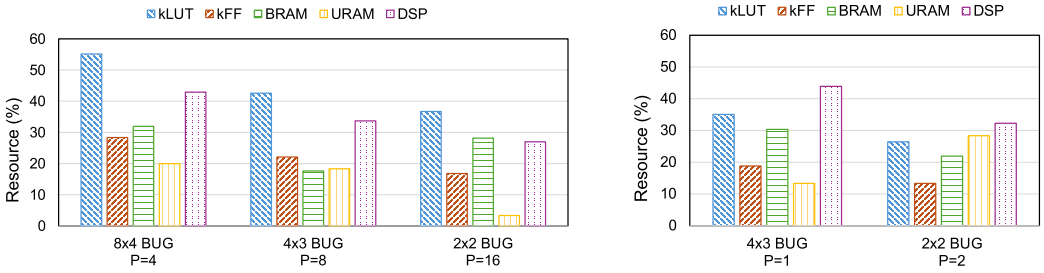


Fig. 9. Pipeline timing diagram illustrating the input-to-output timing path for the NTT Processing Element (PE), with an example configuration of $n_1 = 2^{12}$, $BUG = V \times H = 4 \times 3$, and $BUG\_DP = 45$.



(a) $\log_2(q) = 64$ (b) $\log_2(q) = 256$

Fig. 10. Resource utilization of different BUGs and parallelism for polynomial size $N = 2^{24}$ on U280.

4.4 Parameter Space Exploration

The theoretical performance model is characterized by four tunable parameters: the polynomial size N, the BUG configuration, the NTT decomposition, and the degree of parallelism P. These parameters interact in non-trivial ways.

BUG Configuration and Parallelism. Choosing a larger BUG instantiates more BUs, which shortens the execution time of each sub-polynomial. Raising P launches additional NTT PEs in parallel and therefore boosts throughput. In HiFA, the theoretical performance model is fully determined by three parameters: the BUG type, the parallelism P, and the selected NTT decomposition. Once these parameters are specified, the framework analytically estimates the corresponding performance, thereby giving an implementation with the best performance. Figure 10 shows that, for a 64-bit modulus, the U280 can host at most $P = 4$, 8, and 16 concurrent NTT PEs under the 8×4, 4×3, and 2×2 BUG configurations, respectively. Because the matrix transpose engine requires P to be a power of two, these values set the effective upper bound on parallelism, even though some device resources remain available. On the U50 with a 64-bit modulus, the 8×4, 4×3, and 2×2 BUG configurations can host at most $P = 2$, 8, and 16 concurrent NTT PEs, respectively. For 256-bit designs, the U280 accommodates up to $P = 1$ and $P = 2$ PEs under the 4×3 and 2×2 BUGs, while the U50, limited by the number of DSPs, can only support $P = 2$ PEs with the 2×2 BUG configuration.

Polynomial Decomposition. The number of BUs mainly decides the maximum degree of parallelism on different available resources. Consequently, Figure 11 plots kernel cycles versus the number of BUs for 64-bit NTTs with $N = 2^{21}$, 2^{24}, and 2^{26} on U280. Each colored cluster represents one of four 2D decompositions obtained by factoring N into sub-polynomials that range from a near-balanced split around $\sqrt{N}$ to increasingly skewed pairs. Within a decomposition, individual points correspond to a larger BUG or a higher parallelism. For $N = 2^{21}$ (Figure 11(a)), the cycles decrease steadily as the BUs budget grows, reflecting the benefit of instantiating more BUs. The best tradeoff occurs at 160 BUs, where an 8 parallel 4×3 BUG operating on a $2^9 \times 2^{12}$ decomposition achieves the lowest cycles while using fewer resources. The $N = 2^{24}$ case (Figure 11(b)) illustrates that an evenly balanced decomposition is optimal, while allocating a larger number of BUs delivers the best performance. When the polynomial size increases to $N = 2^{26}$ (Figure 11(c)), a similar trend appears; however, an asymmetric decomposition of $2^{12} \times 2^{14}$ paired with an 8×4 BUG using four PEs produces the lowest cycle count.

In summary, the DSE framework automatically adjusts the decomposition, BUG configuration, and PE parallelism to match the available on-chip resources. These results confirm the importance of DSE and show that the default decomposition does not always yield the best performance.

5 Results and Analysis

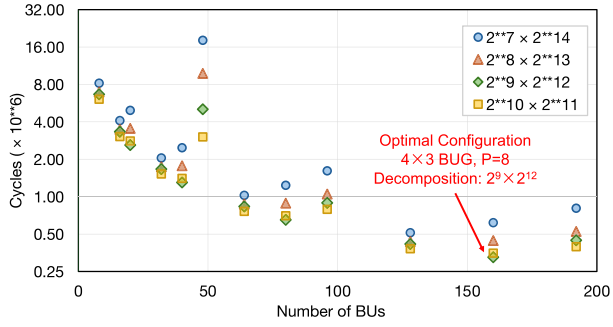
5.1 Experimental Setup

All experiments are conducted on AMD/Xilinx U280 [6] and U50 [7] datacenter FPGA boards. Designs are written in TAPA framework [18] and compiled with Vitis HLS 2023.2. The resource utilization is collected after the place-and-route. The bitstreams are executed on the physical boards using XRT 2022.2. We measure the kernel execution time and the off-chip memory read/write time. Both the polynomial coefficients and pre-computed twiddle factors reside in off-chip memory at kernel launch. The design supports both NTT and inverse NTT without re-synthesis; all outputs are verified against a CPU reference implementation.

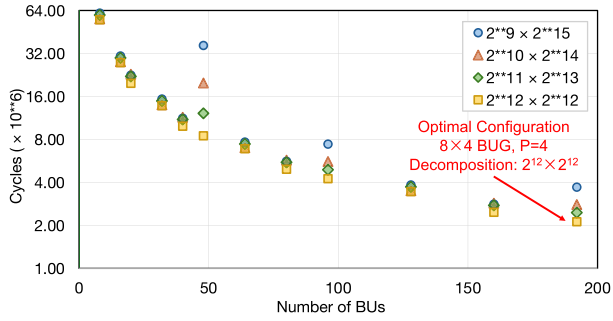
5.2 Overall Performance and Validation

This section summarizes the latency, operating frequency, and FPGA resource utilization of the NTT accelerators HiFA. We evaluate 64-bit coefficient designs for polynomial sizes N from 2^{20} to 2^{26}, and 256-bit coefficient designs for N from 2^{20} to 2^{24}, reflecting typical parameter requirements for HE and ZKP workloads; each row of tables also records the BUG type, the degree of parallelism P, and the decomposition parameter of (n_1, n_2). We also report the estimated latency and resource utilization calculated based on the analytical model in Section 4. The latency results for each of the optimized designs were derived from a combination of synthesis reports and theoretical calculations. Specifically, we calculated the average latency per module from multiple synthesis results and then applied Equations (9) and (10) to determine the total latency in clock cycles. For practical comparison with observed performance, the estimated runtime was derived by dividing the cycle count by the corresponding clock frequency. The resource assumptions for this analysis are based on BRAM, configured with a data width of 36 bits and a depth of 512 entries, while each URAM is similarly configured with a data width of 72 bits and a depth of 4,096 entries. Finally, the DSP is assumed to be a standard 27×18 multiplier configuration, as per the specifications of the targeted devices (e.g., U280 and U50).

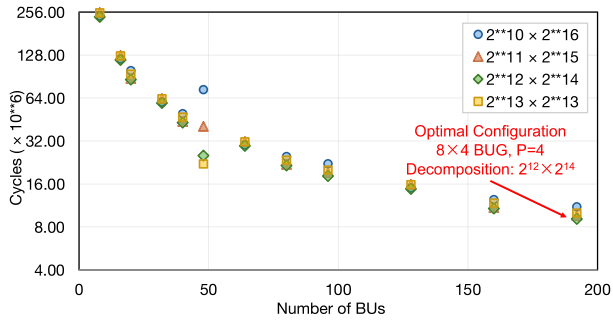
For 64-bit polynomials on the U280 FPGA (Table 2), our DSE selects a 4×3 BUG with $P = 8$ for $N = 2^{20}$ and $N = 2^{21}$, as the associated decompositions (n_1, n_2) minimize overall latency compared to other options by enabling higher parallelism in smaller sub-polynomials. For polynomial sizes ranging from 2^{22} to 2^{26}, an 8×4 BUG with $P = 4$ proves more effective, balancing resource



(a) $N = 2^{21}$, $\log_2(q) = 64$



(b) $N = 2^{24}$, $\log_2(q) = 64$



(c) $N = 2^{26}$, $\log_2(q) = 64$

Fig. 11. Estimated cycles–BU tradeoffs of $\log_2(q) = 64$ polynomial coefficient for two polynomial sizes under different decompositions.

utilization with computational demands for larger N values, where the increased BUG size supports efficient scaling without excessive latency penalties. On the U50 (Table 4), resource constraints make an 8×4 BUG configuration with $P = 4$ infeasible, as it would push LUT utilization close to the device's limit. Consequently, our DSE flow consistently selects the 4×3 BUG with $P = 8$. Compared to the estimated results, the latency derived from the proposed analytical model demonstrates close alignment with the measured values, with an average relative difference of approximately 4.38% for Table 3 when compared to Table 2, and 6.5% for Table 5 when compared to Table 4. This consistency

Table 2. The Best Performing Design on U280 for $\log_2(q) = 64$

N	BUG	P	Decomp. (n_1, n_2)	Freq. (MHz)	Latency (ms)	kLUT	kFF	BRAM	URAM	DSP
2^{20}	4×3	8	$(2^{11}, 2^9)$	217	0.8	431 (37.6%)	451 (18.95%)	128 (7.04%)	176 (18.33%)	3,064 (33.97%)
2^{21}	4×3	8	$(2^{12}, 2^9)$	212	1.51	404 (35.31%)	402 (16.91%)	192 (10.57%)	176 (18.33%)	3,064 (33.97%)
2^{22}	8×4	4	$(2^{11}, 2^{11})$	250	2.17	548 (48.01%)	551 (23.22%)	592 (32.58%)	192 (20%)	3,648 (40.44%)
2^{23}	8×4	4	$(2^{12}, 2^{11})$	211	5.02	565 (49.48%)	574 (24.16%)	592 (32.58%)	192 (20%)	3,648 (40.44%)
2^{24}	8×4	4	$(2^{12}, 2^{12})$	243	8.15	536 (46.96%)	546 (22.98%)	592 (32.58%)	192 (20%)	3,648 (40.44%)
2^{25}	8×4	4	$(2^{13}, 2^{12})$	220	20.56	549 (48.08%)	589 (24.8%)	656 (36.1%)	192 (20%)	3,712 (41.15%)
2^{26}	8×4	4	$(2^{14}, 2^{12})$	201	45.81	555 (48.66%)	598 (25.19%)	788 (43.37%)	320 (33.33%)	3,712 (41.15%)

Table 3. Estimated Latency and Resource Utilization Based on the Configuration in Table 2

N	c	BUG	P	Decomp. (n_1, n_2)	Est. Lat. (ms)	FB	CB	TB	BRAM	TF	CU	URAM	M	DSP
2^{20}	16	4×3	8	$(2^{11}, 2^9)$	0.76	0	128	128	128	48	128	176	21	3,360
2^{21}	8	4×3	8	$(2^{12}, 2^9)$	1.52	0	256	128	192	48	128	176	21	3,360
2^{22}	16	8×4	4	$(2^{11}, 2^{11})$	2.38	512	128	512	576	64	128	192	21	4,032
2^{23}	8	8×4	4	$(2^{12}, 2^{11})$	5.31	512	128	512	576	64	128	192	21	4,032
2^{24}	8	8×4	4	$(2^{12}, 2^{12})$	8.65	512	128	512	576	64	128	192	21	4,032
2^{25}	8	8×4	4	$(2^{13}, 2^{12})$	20.96	512	256	512	640	64	128	192	21	4,032
2^{26}	8	8×4	4	$(2^{14}, 2^{12})$	45.16	512	512	512	768	64	256	320	21	4,032

Decomp.: Decomposition. **Est. Lat.**: Estimated Latency.
M: $NUM_{multiplier}$. **DSP:** NUM_{DSP}.
FB: NUM_{FIFO_BRAM}. **CB:** NUM_{Coeff_BRAM}. **TB:** $NUM_{Transpose\_BRAM}$. **BRAM:** NUM_{total_BRAM}.
TF: NUM_{TF_URAM}. **CU:** NUM_{Cyclic_URAM}. **URAM:** NUM_{total_URAM}.

underscores the efficacy of the estimation equation in accurately capturing latency trends. It is observed that for $N = 2^{26}$ in Table 4, the estimated latency shows a 16.02% underestimation relative to the measured value. This is because of the reduced bandwidth caused by a configuration with $c = 4$, where the burst size is insufficient to sustain optimal data transfer rates.

The achievable clock frequency, along with the utilization of LUTs and FFs, is influenced by four internal primary factors. (1) A larger number of multipliers increases the consumption of LUTs and FFs. (2) The larger size between the decomposed sub-polynomials, denoted as n_1 and n_2. The configurable array sizes for both coefficient and twiddle factor computations are configured to accommodate the larger of the two sub-polynomials. This dictates the total register count and the complexity of the data path, thereby directly impacting LUT and FF utilization. (3) The balance of the decomposition, i.e., whether $n_1 = n_2$, significantly impacts the achievable clock frequency. Unbalanced decompositions need additional control logic and a second set of smaller arrays to manage the distinct computational requirements of the different sub-polynomial sizes. This architectural overhead introduces a longer critical path, consequently increasing frequency and resource utilization. (4) The divisibility of the logarithms of the decomposed sub-polynomials. For instance, if the BUG has a horizontal dimension of $H = 3$ (e.g., for a 4×3 BUG) or $H = 4$ (e.g.,

Table 4. The Best Performing Design on U50 for $\log_2(q) = 64$

N	BUG	P	Decomp. (n_1, n_2)	Freq. (MHz)	Latency (ms)	kLUT	kFF	BRAM	URAM	DSP
2^{20}	4×3	8	$(2^{11}, 2^9)$	208	0.86	436 (61.49%)	379 (24.96%)	128 (11.02%)	176 (27.67%)	3,064 (51.51%)
2^{21}	4×3	8	$(2^{12}, 2^9)$	207	1.61	418 (59.05%)	409 (26.92%)	256 (16.54%)	176 (27.67%)	3,064 (51.51%)
2^{22}	4×3	8	$(2^{11}, 2^{11})$	229	3.06	426 (60.06%)	502 (33.07%)	192 (16.54%)	176 (27.67%)	3,040 (51.11%)
2^{23}	4×3	8	$(2^{11}, 2^{12})$	200	6.36	436 (61.46%)	378 (24.87%)	192 (16.54%)	176 (27.67%)	3,040 (51.11%)
2^{24}	4×3	8	$(2^{12}, 2^{12})$	227	11.12	402 (56.71%)	444 (29.2%)	192 (16.54%)	304 (47.8%)	3,040 (51.11%)
2^{25}	4×3	8	$(2^{13}, 2^{12})$	181	31.91	432 (60.98%)	390 (25.65%)	392 (33.76%)	304 (47.8%)	3,064 (51.51%)
2^{26}	4×3	8	$(2^{14}, 2^{12})$	176	72.79	442 (62.27%)	395 (25.96%)	616 (53.06%)	336 (52.83%)	3,064 (51.51%)

Table 5. Estimated Latency and Resource Utilization Based on the Configuration in Table 4

N	c	BUG	P	Decomp. (n_1, n_2)	Est. Lat. (ms)	FB	CB	TB	BRAM	TF	CU	URAM	M	DSP
2^{20}	16	4×3	8	$(2^{11}, 2^9)$	0.79	0	128	128	128	48	128	176	21	3,360
2^{21}	8	4×3	8	$(2^{12}, 2^9)$	1.55	0	256	256	256	48	128	176	21	3,360
2^{22}	16	4×3	8	$(2^{11}, 2^{11})$	2.81	0	128	256	192	48	128	176	21	3,360
2^{23}	16	4×3	8	$(2^{11}, 2^{12})$	6.30	0	128	256	192	48	128	176	21	3,360
2^{24}	16	4×3	8	$(2^{12}, 2^{12})$	10.88	0	256	128	192	48	256	304	21	3,360
2^{25}	8	4×3	8	$(2^{13}, 2^{12})$	29.78	0	512	128	320	48	256	304	21	3,360
2^{26}	4	4×3	8	$(2^{14}, 2^{12})$	61.13	0	1,024	128	576	80	256	336	21	3,360

Table 6. The Best Performing Design on U280 for $\log_2(q) = 256$

N	BUG	P	Decomp. (n_1, n_2)	Freq. (MHz)	Latency (ms)	kLUT	kFF	BRAM	URAM	DSP
2^{20}	2×2	2	$(2^{10}, 2^{10})$	220	6.45	300 (25.72%)	345 (14.28%)	251 (13.8%)	80 (8.33%)	2,912 (32.28%)
2^{21}	4×3	1	$(2^{11}, 2^{10})$	216	12.35	395 (33.99%)	448 (18.64%)	294 (16.18%)	88 (9.17%)	3,640 (40.35%)
2^{22}	4×3	1	$(2^{11}, 2^{11})$	182	28.91	394 (33.95%)	392 (16.26%)	436 (24%)	88 (9.17%)	3,640 (40.35%)
2^{23}	4×3	1	$(2^{12}, 2^{11})$	140	69.24	398 (34.29%)	416 (17.3%)	551 (30.32%)	64 (6.67%)	3,640 (40.35%)
2^{24}	4×3	1	$(2^{12}, 2^{12})$	208	92.32	387 (33.37%)	415 (17.27%)	551 (30.32%)	128 (13.33%)	3,640 (40.35%)

for an 8×4 BUG), and the logarithm is indivisible by H, it will introduce a coefficient reordering task. This additional logic for data realignment increases the consumption of logic resources.

For factor (1), the design employing an 8×4 BUG configuration utilizes more LUTs and FFs compared to the design utilizing a 4×3 BUG configuration. In the 8×4 BUG configuration (detailed in Table 2) for $\log_2(q) = 64$, the achievable clock frequency exhibits a decline, and LUT and FF usage increase as the sub-polynomial size n_1 grows from an input size of $N = 2^{25}$ to 2^{26}, as per factor (2). Similarly, for the 4×3 BUG configuration with $\log_2(q) = 64$ (Tables 2 and 4), a continuous decrease

in frequency and a corresponding increase in LUT and FF usage are observed as polynomial sizes scale from $N = 2^{20}$ to $2^{21}, 2^{23}, 2^{25}$, and 2^{26}. For factor (3), as detailed in Tables 2 and 4, notably, $N = 2^{22}$ and $N = 2^{24}$ on both the U280 and U50 platforms achieve the highest frequencies and relatively low LUT and FF usage within the set. This is attributed to their balanced decompositions ($n_1 = n_2 = 2^{11}$ and $n_1 = n_2 = 2^{12}$, respectively). A balanced decomposition allows the two rounds of NTT operations to share the same PEs with simplified control logic, thereby reducing the critical path delay and overall resource overhead. For factor (4), $N = 2^{24}$ exhibits lower LUT and FF usage compared to $N = 2^{22}$ across both the 4×3 and 8×4 BUG configurations, attributable to the divisibility of the decomposed sub-polynomials n_1 and n_2 by the horizontal dimensions $V = 3$ and $V = 4$.

The amount of BRAM is determined by the FIFOs, coefficient shuffler, and transpose buffer, while the utilization of URAM is governed by the twiddle factors and cyclic shuffler. DSP consumption is directly influenced by the number of multipliers in each configuration. When estimating the BRAM count in each table, the total is calculated by summing the resource requirements of the *FB*, *CB*, and *TB* and dividing the result by two. This approach reflects the assumption of a 36-bit by 512-entry memory configuration, which utilizes half of a single 36 Kb BRAM block. Similarly, the URAM count is determined by adding *TF* and *CU*. It is worth noting that our analytical model slightly underestimates the actual BRAM utilization, particularly in designs with large FIFOs. This is due to the implementation of FIFO-based data streaming within the TAPA framework [18], which uses relay_station modules that consume additional BRAM resources beyond our estimation. However, for $N = 2^{25}$ and 2^{26} in Table 5, we still observe an underestimation, even though no large FIFOs are present. We determined that this extra BRAM is consumed by the tasks that also involve twiddle factors. The number of URAMs estimated by our model perfectly matches the actual resource utilization. For DSP, our model provides a theoretical upper bound, as it does not account for the synthesis-level optimizations such as multiplier sharing and resource packing.

Tables 6 and 8 present the performance of our 256-bit four-step NTT accelerator on the U280 and U50 platforms, respectively. For each polynomial size $N = 2^{20}$ through 2^{24}, we also report the chosen BUG configuration, parallelism factor P, decomposition (n_1, n_2), achieved clock frequency, end-to-end latency, and utilization of key FPGA resources. These data reveal how our framework adapts the BUG configuration and parallelism to each device's resource constraints. On the U280 (Table 6), a 2×2 BUG with $P = 2$ is employed for $N = 2^{20}$. As N increases, we switch to the 4×3 BUG where $P = 1$, because its BUG and corresponding decomposition generate a better performance than the 2×2 BUG configuration. Table 8 summarizes the best-performing 256-bit NTT configurations on the U50 FPGA for polynomial sizes $N = 2^{20}$ through 2^{24}. In all cases, we employ a 2×2 BUG with parallelism $P = 2$. This is because a larger BUG would push LUT utilization beyond a feasible limit. The latencies obtained from the proposed analytical model are close to the measured values when compared to the estimated results, exhibiting an average relative difference of approximately 6.83% between Tables 6 and 7, and 4.77% between Tables 8 and 9.

For $\log_2(q) = 256$, the 4×3 BUG design exhibits a relatively lower frequency compared to the 2×2 BUG design, as indicated in Table 6, primarily due to the increased number of multipliers, aligning with factor (1). However, a notable decline in frequency is observed from $N = 2^{21}$ to $N = 2^{22}, 2^{23}$, and 2^{24} in Table 6. This drop is attributed to the resource placement and routing challenges, as $N = 2^{21}$ utilizes six slots on the U280, whereas $N = 2^{22}, 2^{23}$, and 2^{24} are constrained to five slots, complicating the efficient allocation of logic and memory resources. Consequently, $N = 2^{24}$ achieves the highest frequency among these configurations, benefiting from its balanced decomposition ($n_1 = n_2 = 2^{12}$) and the divisibility of $\log_2(n_1)$ by $H = 3$, as outlined in factors (3) and (4). In contrast, $N = 2^{22}$ ($n_1 = n_2 = 2^{11}$) gets the second-highest frequency, outperforming the unbalanced decomposition of $N = 2^{23}$ ($n_1 = 2^{12}, n_2 = 2^{11}$) due to the advantages conferred by factor

Table 7. Estimated Latency and Resource Utilization Based on the Configuration in Table 6

N	c	BUG	P	Decomp. (n_1, n_2)	Est. Lat. (ms)	FB	CB	TB	BRAM	TF	CU	URAM	M	DSP
2^{20}	16	2×2	2	$(2^{10}, 2^{10})$	7.09	256	64	128	224	16	64	80	198	3,168
2^{21}	16	4×3	1	$(2^{11}, 2^{10})$	13.43	256	64	512	416	24	64	88	198	3,960
2^{22}	16	4×3	1	$(2^{11}, 2^{11})$	29.62	256	64	512	416	24	64	88	198	3,960
2^{23}	8	4×3	1	$(2^{12}, 2^{11})$	74.61	256	128	512	544[a]	24	64	64[b]	198	3,960
2^{24}	16	4×3	1	$(2^{12}, 2^{12})$	97.20	256	128	512	544[a]	24	128	128[b]	198	3,960

[a]The *TF* is implemented using BRAM and is included in the *BRAM*.
[b]The *TF* is implemented using BRAM and is excluded from the *URAM*.

Table 8. The Best Performing Design on U50 for $\log_2(q) = 256$

N	BUG	P	Decomp. (n_1, n_2)	Freq. (MHz)	Latency (ms)	kLUT	kFF	BRAM	URAM	DSP
2^{20}	2×2	2	$(2^{10}, 2^{10})$	238	6.08	303 (41.48%)	361 (23.22%)	251 (21.62%)	80 (12.58%)	2,912 (48.96%)
2^{21}	2×2	2	$(2^{11}, 2^{10})$	206	14.72	309 (42.25%)	334 (21.49%)	299 (25.75%)	80 (12.58%)	2,916 (49.02%)
2^{22}	2×2	2	$(2^{12}, 2^{10})$	208	31.67	307 (41.92%)	339 (21.82%)	355 (30.58%)	144 (22.64%)	2,916 (49.02%)
2^{23}	2×2	2	$(2^{11}, 2^{12})$	199	66.93	308 (42.18%)	326 (20.94%)	497 (42.81%)	64 (10.06%)	2,912 (48.96%)
2^{24}	2×2	2	$(2^{12}, 2^{12})$	228	123.44	308 (42.27%)	362 (23.33%)	370 (31.87%)	144 (22.64%)	2,912 (48.96%)

Table 9. Estimated Latency and Resource Utilization Based on the Configuration in Table 8

N	c	BUG	P	Decomp. (n_1, n_2)	Est. Lat. (ms)	FB	CB	TB	BRAM	TF	CU	URAM	M	DSP
2^{20}	8	2×2	2	$(2^{10}, 2^{10})$	6.56	256	64	128	224	16	64	80	198	3,168
2^{21}	8	2×2	2	$(2^{11}, 2^{10})$	16.15	256	128	128	256	16	64	80	198	3,168
2^{22}	8	2×2	2	$(2^{12}, 2^{10})$	31.68	256	256	128	320	16	128	144	198	3,168
2^{23}	8	2×2	2	$(2^{11}, 2^{12})$	70.15	256	128	256	416[a]	16	64	64[b]	198	3,168
2^{24}	8	2×2	2	$(2^{12}, 2^{12})$	121.66	256	256	256	384	16	128	144	198	3,168

[a]The *TF* is implemented using BRAM and is included in the *BRAM*.
[b]The *TF* is implemented using BRAM and is excluded from the *URAM*.

(4). Similarly, in Table 8, $N = 2^{20}$ and $N = 2^{24}$ exhibit the highest achieved frequencies, attributable to the combined effects of factors (3) and (4). Furthermore, $N = 2^{20}$ outperforms $N = 2^{21}$ and $N = 2^{23}$ in frequency, primarily due to the benefits of factor (4). Regarding resource utilization trends, the consumption of LUTs and FFs exhibits relative stability with minor fluctuations across the configurations detailed in Tables 6 and 8. In our analysis, we observed an overestimation of BRAM utilization for the $N = 2^{21}$ configuration (Table 7) and the $N = 2^{24}$ configuration (Table 9). This difference is due to the synthesis optimizing and reducing the BRAM required by the transpose buffer. Moreover, the estimated URAM utilization continues to align with the actual implementation. For DSP, our analytical model provides a theoretical upper bound as observed and discussed in $\log_2(q) = 64$.

Overall, these results indicate that our framework customizes the BUG configuration and parallelism level based on the resources of each FPGA and is able to provide a validated performance and

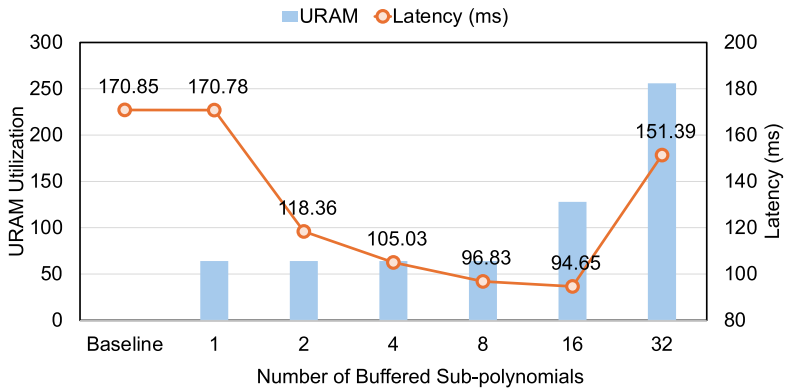


Fig. 12. Latency versus cyclic-shuffler buffer size for a $\log_2(q) = 256$, $N = 2^{24}$ polynomial. All cases run at 200 MHz, except the 32-NTT case, which runs at 126 MHz.

resource estimation. On the U280, abundant on-chip memory and DSP capacity allow deployment of a larger BUG, yielding lower end-to-end latency. In contrast, on the U50, tighter resource constraints limit the BUG size, resulting in higher resource utilization while still maintaining comparable clock frequencies. Consequently, our approach sustains high concurrency and efficiently accelerates large-size NTTs within the on-chip budgets of diverse FPGA platforms.

5.3 Analysis of Individual Optimizations

Cyclic Shuffler. Building on the HBM characteristics discussed in Section 3.3, we evaluated how cyclic buffering influences end-to-end NTT latency. Figure 12 shows the tradeoff between URAM utilization and kernel latency as the number of buffered polynomials varies; each sub-polynomial has size 2^{12}, uses 256-bit coefficients, and is processed by a 4×3 BUG with $P = 1$. All examples achieve 200 MHz, except the 32-NTT case, which runs at only 126 MHz. For clarity, only the cyclic double buffer is stored in URAM, while all other data remain in BRAM. The baseline design (no buffering) exhibits a latency of 170.85 ms and consumes no URAM. Adding a single cyclic buffer (1 NTT) raises URAM usage to 64 blocks but still keeps a similar latency at 170.78 ms. Doubling the buffer to two polynomials still uses only 64 URAM blocks because each block stores $4{,}096 \times 72$ bits, and we stripe four blocks to form one 256-bit row. Consequently, because the sub-polynomial is organized as 8×512 coefficients, four URAM blocks can accommodate up to eight sub-polynomials; the additional buffering therefore further reduces latency without increasing on-chip memory usage. When the buffer size increases to 16 sub-polynomials (16 NTTs), URAM usage doubles to 128 blocks, yet execution latency improves by only a 1.02× speedup, indicating that buffering more than eight NTTs yields only marginal gains. However, buffering more than 16 NTTs (i.e., 32 NTTs) does not improve performance and degrades the frequency.

These results demonstrate that by enforcing as few as four to eight consecutive NTT coefficient streams before issuing each HBM transaction, the shuffler effectively converts a random access pattern into near-optimal 4-burst transfers, thereby recovering almost all of the 13 GB/s bandwidth. However, when the buffer size increases from 8 NTTs to 16 NTTs, extending the cyclic shuffler to rearrange more than 16 NTTs only slightly reduces latency but significantly increases URAM utilization. The exact overhead varies with the BUG template and sub-polynomial size. Consequently, we choose the buffer length on a case by case basis.

Resource Utilization of Modular Reduction Methods. Table 10 highlights how our modular reduction methods affect HiFA's on-chip resources. In the 64-bit case, a specialized Mersenne-prime-based

Table 10. Resource Utilization for Reduction Methods on HiFA

Method	$\log_2(q)$	Reduction	LUT	FF	DSP
SAM [40]	64	N/A	-	-	24[a]
HiFA		WLM	1,106	757	17~19
HiFA		Mersenne	1,128	546	12
SAM [40]	256	N/A	-	-	242[a]
HiFA		WLM	12,252	11,411	182

[a]Estimated from the full-design resource utilization.

reduction method reduces the DSP count compared to the WLM approach. We also observe that DSP usage can vary across different polynomial sizes and HLS mappings. Moreover, against the SAM multiplier, our BUs consume fewer DSPs in both 64- and 256-bit designs, which contributes to achieving higher clock frequencies.

5.4 Comparison with State-of-the-Art Methods

Table 11 presents a detailed comparison of HiFA with two state-of-the-art FPGA NTT accelerators, SAM [40] and CycloneNTT [1], as well as the ASIC NTT accelerator PipeZK [46], across various modular reduction schemes, moduli, and polynomial sizes, evaluated based on performance and resource metrics. Referring to the previous works [23, 30], we adopt the **Area-Delay Product (ADP)** as a metric to evaluate resource utilization across different FPGA platforms, where a lower value indicates better efficiency. To minimize per-PE resource use and maximize parallelism, SAM adopts a complex multi-dimensional NTT decomposition. However, this approach incurs additional off-chip memory traffic and significant interconnect overhead, which in turn depresses the achievable clock frequency and hinders scalability on large FPGA platforms. Thanks to its carefully optimized architecture, HiFA attains higher clock frequencies than both SAM and CycloneNTT. In the 64-bit workloads, HiFA delivers 3.56× and 4.31× lower latency than SAM for $N = 2^{20}$ and 2^{24}, respectively; even when normalized to the same frequency or compared by clock-cycle count, the speedups remain 2.7× and 2.89×. Moreover, despite using only half as many DSPs as SAM, our design still achieves 1.94× and 1.55× improvements for $N = 2^{20}$ and 2^{24} when $\log_2(q) = 256$. On-chip memory utilization is also reduced relative to SAM. Compared to SAM, HiFA achieves a 4.3× lower ADP on average. This improvement reflects the higher computational capability of its PEs, which reduces execution latency. In addition, the reduced BRAM suggests that HiFA's architecture is more routing-friendly on FPGA, enabling us to achieve higher clock frequencies.

By contrast, CycloneNTT implements a similar stacked NTT kernel but is constrained by on-chip memory capacity, forcing off-chip accesses to shuffle coefficients after each BUG computation stage. These repeated off-chip transactions introduce substantial latency that negates the inherent performance advantages of the standard NTT algorithm. In HiFA, we instead use on-chip buffers to locally shuffle sub-polynomial coefficients, so that off-chip memory is accessed only once during the transpose phase. Another limitation of CycloneNTT is that it cannot accommodate polynomial sizes whose number of NTT stages is not an integer multiple of the BUG horizontal size H (e.g., the CycloneNTT-6 variant fails for $N = 2^{20}$), severely limiting its applicability. It also instantiates only a single NTT PE, leading to routing congestion and a degraded clock frequency in the larger BU implementation (e.g., CycloneNTT-6). In the 64-bit cases, HiFA achieves latency speedups of 1.11×, 7.25×, and 1.04× for $N = 2^{20}$, 2^{21}, and 2^{24}, respectively. For $N = 2^{24}$, CycloneNTT implements a 32×6 BUG design. It uses 1.83× more BRAM and runs at a lower frequency. Our framework thus achieves a 1.51× lower ADP. Although HiFA exhibits higher ADP than CycloneNTT at 2^{20} and 2^{21},

Table 11. Performance and Resource Comparison across FPGA/ASIC Platforms

Method	$\log q$	N	Reduction	Platform	Freq. (MHz)	Latency (ms) (Speedup)	kLUT	kFF	BRAM	DSP	ADP (Speedup)
SAM [40]	64	2^{20}	-	U250	165	2.84 (3.56×)	267	328	2,126[a]	2,736	1,546 (4.63×)
HiFA			WLM	U280	217	0.8 (1×)	424	418	1,216	3,064	334 (1×)
CycloneNTT-5 [1]			Mersenne	C1100	176	1.04 (1.11×)	170	121	960	960[b]	259 (0.91×)
HiFA			Mersenne	U280	200	0.93 (1×)	420	320	768	1,944	285 (1×)
CycloneNTT-3 [1]		2^{21}	Mersenne	C1100	300	10.86 (7.25×)	17	17	144	144[b]	386 (0.79×)
HiFA			Mersenne	U280	205	1.5 (1×)	420	300	896	1,944	491 (1×)
SAM [40]		2^{24}	-	U250	165	34.12 (4.31×)	267	328	2,126[a]	2,736	18,572 (4.98×)
HiFA			WLM	U280	246	7.92 (1×)	535	544	1,232	3,648	3,727 (1×)
CycloneNTT-6 [1]			Mersenne	C1100	161	8.08 (1.04×)	564	319	2,304	2,304[b]	5,152 (1.51×)
HiFA			Mersenne	U280	250	7.75 (1×)	507	412	1,256	2,304	3,413 (1×)
SAM [40]	256	2^{20}	-	U250	100	12.61 (1.94×)	593	534	2,269[a]	6,776	8,564 (4.65×)
PipeZK-scaled [46]			Montgomery	-	220	15 (2.31×)	-	-	-	-	-
HiFA			WLM	U280	220	6.48 (1×)	300	351	737	2,912	1,840 (1×)
SAM [40]		2^{24}	-	U250	100	183.56 (1.55×)	593	534	2,269[a]	6,776	124,665 (2.92×)
HiFA			WLM	U280	165	118.18 (1×)	380	399	986	3,640	42,668 (1×)

ADP: Area-Delay Product = Latency × (kLUT + kFF + BRAM + DSP)/5.
[a]Converted KB to 36 K BRAM.
[b]Calculated using the Mersenne-prime reduction method.

it offers greater flexibility by supporting a broader range of prime moduli, which makes it more adaptable to diverse application requirements.

PipeZK implements a dataflow NTT PE with limited flexibility in parallelism. As an ASIC design with a higher frequency, it was scaled down to match the frequency of HiFA. At $N = 2^{20}$, HiFA outperforms the scaled PipeZK by 2.31×. Overall, HiFA achieves a maximum latency improvement of 7.25× and an average speedup of 2.97× compared to prior FPGA implementations, with an average ADP gain of 2.91×.

We also present a performance and energy efficiency comparison in Table 12, evaluating our FPGA-based implementation (HiFA on the U280) against GPU-based methods. These include 4step-ntt [34] on the NVIDIA V100 for NTT operations with 64-bit coefficients across polynomial sizes ranging from $N = 2^{20}$ to 2^{24}, as well as bellperson [11] and GZKP [28] on the NVIDIA V100 and GTX 1080Ti platforms for NTT operations with 256-bit coefficients across polynomial sizes $N = 2^{20}, 2^{22}, 2^{24}$. The table shows latency (ms), throughput (NTT/s), total power consumption (W), dynamic power consumption (W), and energy efficiency ($ms \cdot W$, with the energy gain relative to HiFA). The power consumption of the GPU utilized by 4step-ntt is measured during kernel runtime. For bellperson and GZKP, the power consumption of the corresponding GPU platforms is estimated by scaling the power of the respective GPU by a factor of 0.75. The total power consumption and the dynamic power consumption of the FPGA are measured directly on the actual board.

In a comparative performance analysis against the 4step-ntt GPU approach [34] on the V100 platform, our HiFA implementation exhibits higher latency, resulting in a negative power gain. However, our dynamic power consumption is approximately 12.49× lower than that of the 4step-ntt GPU method. When compared to other GPU-based implementations, HiFA demonstrates superior performance on the GTX 1080Ti, achieving lower latency and higher throughput than bellperson across all tested scenarios. Furthermore, HiFA shows a clear advantage in power efficiency, consuming less power than both bellperson and GZKP on the GTX 1080Ti, as well as bellperson on the V100. On average, our method achieves an energy gain of 2.24× compared to these GPU-based methods.

Table 12. Performance and Energy Efficiency Comparison across GPU Platforms

Method	$\log_2(q)$	N	Platform	Latency (ms)	Throughput (NTT/s)	Total Power (W)	Dynamic Power (W)	Energy Efficiency ($ms \cdot W$) (Energy Gain)
4step-ntt [34]	64	2^{20}	V100	0.118	8,474	183.84	160.27	22 (0.66×)
HiFA			U280	0.8	1,250	41.19	9.64	33 (1×)
4step-ntt [34]		2^{21}	V100	0.216	4,629	236.01	212.44	51 (0.80×)
HiFA			U280	1.51	662	41.99	10.20	63 (1×)
4step-ntt [34]		2^{22}	V100	0.436	2,293	232.37	208.80	101 (0.83×)
HiFA			U280	2.17	460	56.13	20.97	122 (1×)
4step-ntt [34]		2^{23}	V100	0.892	1,121	248.66	225.09	222 (0.69×)
HiFA			U280	5.02	199	63.91	28.61	321 (1×)
4step-ntt [34]		2^{24}	V100	1.873	533	247.88	224.31	464 (0.84×)
HiFA			U280	8.15	122	67.90	31.24	553 (1×)
bellperson [11]	256	2^{20}	V100	5.19	192	187.5[a]	/	973 (3.67×)
GZKP [28]			V100	1.07	934	187.5[a]	/	201 (0.76×)
bellperson [11]			GTX 1080Ti	23.8	42	187.5[a]	/	4,463 (16.81×)
GZKP [28]			GTX 1080Ti	2.87	348	187.5[a]	/	538 (2.03×)
HiFA			U280	6.45	155	41.15	10.56	265 (1×)
bellperson [11]		2^{22}	V100	12.69	78	187.5[a]	/	2,379 (1.26×)
GZKP [28]			V100	4.96	201	187.5[a]	/	930 (0.49×)
bellperson [11]			GTX 1080Ti	70.5	14	187.5[a]	/	13,219 (7.00×)
GZKP [28]			GTX 1080Ti	12.83	77	187.5[a]	/	2,406 (1.27×)
HiFA			U280	28.91	34	65.35	34.72	1,889 (1×)
bellperson [11]		2^{24}	V100	46.74	21	187.5[a]	/	8,764 (1.43×)
GZKP [28]			V100	20.99	47	187.5[a]	/	3,936 (0.64×)
bellperson [11]			GTX 1080Ti	234.59	4	187.5[a]	/	43,986 (7.18×)
GZKP [28]			GTX 1080Ti	56.18	17	187.5[a]	/	10,534 (1.72×)
HiFA		2^{24}	U280	92.32	10	66.31	34.11	6,122 (1×)

[a]Estimated by multiplying the platform's maximum power by a coefficient of 0.75.

6 Related Work

CPU-Based Acceleration. The Microsoft SEAL library [32], a widely adopted framework for FHE, supports BFV and CKKS schemes with a CPU-based NTT implementation optimized for modern processors. OpenFHE [3] is a community-driven FHE library that supports multiple schemes, including BGV, BFV, and CKKS. Its NTT implementation is primarily CPU-based, with optimizations for multi-core processors via OpenMP for parallelism. In the domain of ZKP, DIZK [42] introduces a distributed algorithm that utilizes compute clusters to enable parallel processing for applications involving billions of circuit gates. Despite these efforts, purely CPU-based implementations of cryptographic schemes still fail to meet practical requirements.

GPU-Based Acceleration. Özcan and Savaş [34] propose GPU NTT algorithms that minimize slow global memory accesses and exploit spatial locality, even tuning CUDA parameters (kernel count, thread block size/shape) to maximize throughput for each polynomial size. GME [37] accelerates FHE using AMD CDNA GPUs with custom microarchitectural extensions and optimizes NTT by improving memory access locality. Badawi et al. [4] employ multi-GPU clusters with data parallelism and use Discrete Galois Transform to reduce memory usage for NTT, enhancing FHE scalability. Both approaches significantly boost FHE performance for privacy-preserving computations.

Recent GPU research also focuses on scaling the NTT kernel to ever-larger ZKP circuits. cuZK [27] follows a classical $\log N$ butterfly schedule, streaming intermediate data through global memory so thousands of threads can process $\frac{N}{2}$ butterflies per stage, and it integrates seamlessly with parallel multi-scalar multiplication and matrix-vector multiplication modules for large-size proofs. GZKP [28] redesigns the memory hierarchy by performing all data shuffling inside low-latency shared memory and arranging thread blocks so that global-memory accesses remain contiguous, thereby eliminating costly global shuffles. UniNTT [20] decomposes the transform hierarchically across multiple GPUs and pairs primitive-root reuse with pipelined communication and layout

optimizations, allowing NTT stages at each hierarchy level to execute with minimal memory and interconnect overhead.

FPGA/ASIC-Based Acceleration. Prior FPGA studies explore standard NTT that can be fit in on-chip resources [19, 22, 43] and the four-step NTT that is designed for offloading the storage during the NTT computation [10, 17, 40, 45] to support the wide moduli and million-point polynomials required by FHE and ZKP. ASIC work also pushes the similar ideas [12, 41, 44, 46, 47].

Proteus [19] is a parametric hardware framework that generates pipelined Radix-2 NTT architectures, supporting both single-path delay feedback and multipath delay commutator approaches. OpenNTT [22] is a fully automated, open source framework that compiles NTT hardware accelerators and incorporates on-the-fly twiddle factor generation to support various NTT types and parameter sets. NTTGen [43] is an automated framework for generating low-latency NTT designs tailored for HE applications on FPGAs, taking application parameters, latency, and resource constraints as input to produce synthesizable Verilog code. However, these architectures limited flexibility for diverse NTT sizes and types beyond specific parameters. Additionally, they lack the key points to address four-step NTT bottlenecks like transposition overhead.

CycloneNTT [1] supports a larger polynomial size compared to the previous methods, but it is still limited by the bit-width (i.e., 64 bits). Although it proposes different NTT kernels to support various polynomial sizes, it can only support the corresponding divisible polynomial sizes and thus cannot explore the maximum performance. Moreover, due to the limited on-chip storage, it has to frequently access the off-chip storage to obtain relevant computing parameters.

ESC-NTT [17] is a fully pipelined, flexible NTT architecture designed for FHE and PQC, which provides three customized NTT modules for a maximum of 4,096-point NTT. It supports a transpose and rearrange module by constructing FIFO arrays. However, it implements two different types of NTT Pes, respectively, for the two rounds of NTT computations in the four-step NTT algorithm, which is unnecessary. HMNTT [26] employs the four-step NTT algorithm and a pipelined transpose module by optimizing the NTT PE to alleviate backpressure and data conflicts in the workflow. SHP-FsNTT [10] combines deep pipelining with parallelism and employs a memory-access scheme that eliminates the need for a dedicated matrix transpose engine in its four-step NTT implementation. However, these designs are constrained to 2^{16}-point transforms with 64-bit modulus. When scaling to a larger NTT size, new obstacles arise. In particular, the off-chip memory will be accessed far more frequently, and the irregular access patterns incur higher latency. This increased memory latency can introduce backpressure, degrading overall pipeline throughput.

PipeZK [46] first implements 2D four-step NTT decomposition on the dataflow architecture for polynomial size up to 2^{20}. SAM [40] uses multi-dimensional decomposition to flexibly support various large-size polynomials (up to 2^{28}) with fixed small hardware NTT units (i.e., 64-point NTT). It also optimizes data movement and on-the-fly twiddle generation for this decomposed flow. However, none of the existing work explores the design space among different 2D decompositions, which leads to the inability to fully explore the optimal decomposition configuration in four-step NTT. Moreover, to fully explore the performance of the NTT PE, we need to carefully avoid the non-sequential off-chip memory access.

7 Conclusion

In this work, we present HiFA, a high-performance and flexible acceleration framework that supports large-size NTTs across multiple bit-widths and modular reduction schemes. HiFA integrates a stacked BUG PE, an on-chip cyclic shuffle buffer, a localized micro-transpose engine, and an automated DSE flow to accelerate four-step NTT on HBM-equipped FPGAs. By minimizing per-PE resource consumption and converting random memory accesses into near-optimal bursts, HiFA achieves up to a 7.25× reduction in latency and an average 2.97× speedup over state-of-the-art

FPGA designs, while remaining portable across different platforms. Compared to prior GPU-based implementations, HiFA achieves an average energy efficiency gain of 2.24×. In the future, HiFA can be extended to multi-FPGA clusters, further increasing parallelism and boosting performance to fully leverage the framework's flexibility and DSE capability. We also plan to integrate HiFA into end-to-end FHE and ZKP workflows on FPGA, enabling the entire cryptographic scheme to run in hardware.

References

[1] Kaveh Aasaraai, Emanuele Cesena, Rahul Maganti, Nicolas Stalder, Javier Varela, and Kevin Bowers. 2022. Cyclonentt: An ntt/fft architecture using quasi-streaming of large datasets on ddr-and hbm-based fpga platforms. *Cryptology ePrint Archive*.

[2] Rashmi Agrawal, Anantha Chandrakasan, and Ajay Joshi. 2024. Heap: A fully homomorphic encryption accelerator with parallelized bootstrapping. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 756–769.

[3] Ahmad Al Badawi, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, et al. 2022. OpenFHE: Open-source fully homomorphic encryption library. In *Proceedings of the 10th Workshop on Encrypted Computing and Applied Homomorphic Cryptography (WAHC '22)*. ACM, New York, NY, 53–63. DOI: https://doi.org/10.1145/3560827.3563379

[4] Ahmad Al Badawi, Bharadwaj Veeravalli, Jie Lin, Nan Xiao, Matsumura Kazuaki, and Aung Khin Mi Mi. 2020. Multi-GPU design and performance evaluation of homomorphic encryption on GPU clusters. *IEEE Transactions on Parallel and Distributed Systems* 32, 2 (2020), 379–391.

[5] AMD. 2021. *Random Accesses*. RetrievedJune 10, 2025 from https://docs.amd.com/r/en-US/Vitis-Tutorials-Vitis-Hardware-Acceleration/Random-Accesses

[6] AMD/Xilinx. 2023. Alveo U280 Data Center Accelerator Card Data Sheet (DS963). Retrieved from https://docs.xilinx.com/r/en-US/ds963-u280/Summary

[7] AMD/Xilinx. 2023. Alveo U50 Data Center Accelerator Card Data Sheet (DS965). Retrieved from https://docs.amd.com/r/en-US/ds965-u50/Summary

[8] Thilini Kaushalya Bandara, Dan Wu, Rohan Juneja, Dhananjaya Wijerathne, Tulika Mitra, and Li-Shiuan Peh. 2023. Flex: Introducing flexible execution on cgra with spatio-temporal vector dataflow. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 1–9.

[9] Paul Barrett. 1986. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In *Conference on the Theory and Application of Cryptographic Techniques*. Springer, 311–323.

[10] Xiaojie Chen, Weicong Lu, Tao Su, and Dihu Chen. 2024. SHP-FsNTT: A scalable and high-performance NTT accelerator based on the four-step algorithm. In *2024 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 1–5.

[11] Filecoin Corp. 2022. Bellperson: GPU Parallel Acceleration for zk-SNARK. Retrieved from https://github.com/filecoin-project/bellperson

[12] Alhad Daftardar, Brandon Reagen, and Siddharth Garg. 2024. SZKP: A scalable accelerator architecture for zero-knowledge proofs. In *Proceedings of the 2024 International Conference on Parallel Architectures and Compilation Techniques*, 271–283.

[13] Steve Dai, Yuan Zhou, Hang Zhang, Ecenur Ustun, Evangeline F. Y. Young, and Zhiru Zhang. 2018. Fast and accurate estimation of quality of results in high-level synthesis with machine learning. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 129–132.

[14] Craig Gentry. 2009. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing*, 169–178.

[15] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. 2016. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *International Conference on Machine Learning*. PMLR, 201–210.

[16] Shafi Goldwasser, Silvio Micali, and Chales Rackoff. 2019. The knowledge complexity of interactive proof-systems. In *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*, 203–225.

[17] Zhenyu Guan, Yongqing Zhu, Yicheng Huang, Luchang Lei, Xueyan Wang, Hongyang Jia, Yi Chen, Bo Zhang, Jin Dong, and Song Bian. 2024. Esc-ntt: An elastic, seamless and compact architecture for multi-parameter ntt acceleration. In *2024 Design, Automation and Test in Europe Conference & Exhibition (DATE)*. IEEE, 1–6.

[18] Licheng Guo, Yuze Chi, Jason Lau, Linghao Song, Xingyu Tian, Moazin Khatti, Weikang Qiao, Jie Wang, Ecenur Ustun, Zhenman Fang, et al. 2023. TAPA: A scalable task-parallel dataflow programming framework for modern

FPGAs with co-optimization of HLS and physical design. *ACM Transactions on Reconfigurable Technology and Systems* 16, 4 (2023), 1–31.

[19] Florian Hirner, Ahmet Can Mert, and Sujoy Sinha Roy. 2024. Proteus: A pipelined NTT architecture generator. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 32, 7 (Jul. 2024), 1228–1238.

[20] Zhuoran Ji, Jianyu Zhao, Peimin Gao, Xiangkai Yin, and Lei Ju. 2025. Accelerating number theoretic transform with multi-GPU systems for efficient zero knowledge proof. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 1–14.

[21] Ahmed Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. 2016. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 839–858.

[22] Florian Krieger, Florian Hirner, Ahmet Can Mert, and Sujoy Sinha Roy. 2024. OpenNTT-an automated toolchain for compiling high-performance NTT accelerators in FHE. In *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*, 1–9.

[23] 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 Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 1–9.

[24] Benzheng Li, Xi Zhang, Hailong You, Zhongdong Qi, and Yuming Zhang. 2022. Machine learning based framework for fast resource estimation of RTL designs targeting FPGAs. *ACM Transactions on Design Automation of Electronic Systems* 28, 2 (2022), 1–16.

[25] Mengyuan Li, Haoran Geng, Michael Niemier, and Xiaobo Sharon Hu. 2023. Accelerating polynomial modular multiplication with crossbar-based compute-in-memory. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 1–9.

[26] Changxu Liu, Danqing Tang, Jie Song, Hao Zhou, Shoumeng Yan, and Fan Yang. 2024. HMNTT: A highly efficient MDC-NTT architecture for privacy-preserving applications. In *Proceedings of the Great Lakes Symposium on VLSI 2024*, 7–12.

[27] Tao Lu, Chengkun Wei, Ruijing Yu, Chaochao Chen, Wenjing Fang, Lei Wang, Zeke Wang, and Wenzhi Chen. 2023. Cuzk: Accelerating zero-knowledge proof with a faster parallel multi-scalar multiplication algorithm on gpus. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2023, 3 (2023), 194–220.

[28] Weiliang Ma, Qian Xiong, Xuanhua Shi, Xiaosong Ma, Hai Jin, Haozhao Kuang, Mingyu Gao, Ye Zhang, Haichen Shen, and Weifang Hu. 2023. Gzkp: A gpu accelerated zero-knowledge proof system. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 340–353.

[29] Hosein Mohammadi Makrani, Farnoud Farahmand, Hossein Sayadi, Sara Bondi, Sai Manoj Pudukotai Dinakarrao, Houman Homayoun, and Setareh Rafatirad. 2019. Pyramid: Machine learning framework to estimate the optimal timing and resource usage of a high-level synthesis design. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 397–403.

[30] Suraj Mandal and Debapriya Basu Roy. 2024. Winograd for NTT: A case study on Higher-Radix and Low-Latency implementation of NTT for post quantum cryptography on FPGA. *IEEE Transactions on Circuits and Systems I: Regular Papers* 71, 12 (Dec. 2024), 6396–6409.

[31] Ahmet Can Mert, Erdinç Öztürk, and Erkay Savaş. 2019. Design and implementation of a fast and scalable NTT-based polynomial multiplier architecture. In *2019 22nd Euromicro Conference on Digital System Design (DSD)*. IEEE, 253–260.

[32] Microsoft Research. 2025. Microsoft SEAL: Fast and Easy-to-Use Homomorphic Encryption Library. Retrieved June 24, 2025 from https://github.com/microsoft/SEAL

[33] Peter L. Montgomery. 1985. Modular multiplication without trial division. *Mathematics of Computation* 44, 170 (1985), 519–521.

[34] Ali Şah Özcan and Erkay Savaş. 2023. Two algorithms for fast gpu implementation of ntt. *Cryptology ePrint Archive*.

[35] Sahand Salamat, Behnam Khaleghi, Mohsen Imani, and Tajana Rosing. 2019. Workload-aware opportunistic energy efficiency in multi-FPGA platforms. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–8.

[36] Michael Scott. 2017. A note on the implementation of the number theoretic transform. In *Proceedings of the 16th IMA International Conference on Cryptography and Coding (IMACC '17)*. Springer, 247–258.

[37] Kaustubh Shivdikar, Yuhui Bao, Rashmi Agrawal, Michael Shen, Gilbert Jonatan, Evelio Mora, Alexander Ingare, Neal Livesay, José L. Abellán, John Kim, et al. 2023. Gme: Gpu-based microarchitectural extensions to accelerate homomorphic encryption. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 670–684.

[38] Weihang Tan, Yingjie Lao, and Keshab K. Parhi. 2023. KyberMat: Efficient accelerator for matrix-vector polynomial multiplication in CRYSTALS-Kyber scheme via NTT and polyphase decomposition. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 1–9.

[39] Mario Vestias and Horacio Neto. 2014. Trends of CPU, GPU and FPGA for high-performance computing. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 1–6.

[40] Cheng Wang and Mingyu Gao. 2023. SAM: A scalable accelerator for number theoretic transform using multi-dimensional decomposition. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 1–9.

[41] Cheng Wang and Mingyu Gao. 2025. UniZK: Accelerating Zero-Knowledge proof with unified hardware and flexible kernel mapping. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 1101–1117.

[42] Howard Wu, Wenting Zheng, Alessandro Chiesa, Raluca Ada Popa, and Ion Stoica. 2018. {DIZK}: A distributed zero knowledge proof system. In *27th USENIX Security Symposium (USENIX Security '18)*, 675–692.

[43] Yang Yang, Sanmukh R. Kuppannagari, Rajgopal Kannan, and Viktor K. Prasanna. 2022. Nttgen: A framework for generating low latency ntt implementations on fpga. In *Proceedings of the 19th ACM International Conference on Computing Frontiers*, 30–39.

[44] Zhengbang Yang, Lutan Zhao, Peinan Li, Han Liu, Kai Li, Boyan Zhao, Dan Meng, and Rui Hou. 2025. LegoZK: A dynamically reconfigurable accelerator for zero-knowledge proof. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 113–126.

[45] Chenkai Zeng, Debiao He, Qi Feng, Cong Peng, and Min Luo. 2024. The implementation of polynomial multiplication for lattice-based cryptography: A survey. *Journal of Information Security and Applications* 83 (2024), 103782.

[46] Ye Zhang, Shuo Wang, Xian Zhang, Jiangbin Dong, Xingzhong Mao, Fan Long, Cong Wang, Dong Zhou, Mingyu Gao, and Guangyu Sun. 2021. Pipezk: Accelerating zero-knowledge proof with a pipelined architecture. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 416–428.

[47] Hao Zhou, Changxu Liu, Lan Yang, Li Shang, and Fan Yang. 2025. ReZK: A highly reconfigurable accelerator for zero-knowledge proof. *IEEE Transactions on Circuits and Systems I: Regular Papers* 72, 2 (Feb. 2025), 802–815.

[48] Lu Zhou, Abebe Diro, Akanksha Saini, Shahriar Kaisar, and Pham Cong Hiep. 2024. Leveraging zero knowledge proofs for blockchain-based identity sharing: a survey of advancements, challenges and opportunities. *Journal of Information Security and Applications* 80 (2024), 103678.

Received 26 June 2025; revised 8 September 2025; accepted 6 October 2025