TP-NTT: Batch NTT Hardware with Application to Relinearization

Emre Koçer[1,2], Tolun Tosun[1], Beren Aydoğan[1], Erkay Savaş[1], Furkan Turan[2] and Ingrid Verbauwhede[2]

[1] Sabancı University, Istanbul, Türkiye
[2] COSIC, KU Leuven, Leuven, Belgium

Abstract. Fully Homomorphic Encryption (FHE) enables arbitrary computation on encrypted data without decryption, providing strong privacy guarantees for secure cloud computing, encrypted analytics, and privacy-preserving machine learning. However, practical deployment of FHE remains limited by the high computational cost of polynomial arithmetic over large modular rings. In particular, Number Theoretic Transform (NTT)–based polynomial multiplication dominates the execution time of modern lattice-based FHE schemes. In this work, we present TP-NTT, a scalable, throughput-optimized NTT architecture supporting a wide range of ring dimensions used in FHE, from 2^{10} to 2^{16}. Our design applies optimizations at multiple levels, from modular arithmetic to the NTT algorithm itself, including multi-dimensional decomposition without requiring additional multiplication blocks. The decomposition dimensionality is configurable at design time, supporting 2-D, 3-D, and 4-D decompositions, each advantageous in specific scenarios. Furthermore, TP-NTT provides design-time-configurable throughput. Combined with its scalable architecture, this enables significant advantages for batch NTT operations compared to other works in the literature. At $n = 2^{16}$, it outperforms the best-performing prior design by 1.33× in average latency while achieving 1.24× better area–time-product (ATP). To demonstrate its efficiency, we present a case study on FHE relinearization, focusing on the BFV scheme. We propose a relinearization accelerator that leverages the fast batch NTT capability of TP-NTT, achieving 67.34× speedup over state-of-the-art software implementations and highlighting TP-NTT's effectiveness in real-world FHE applications.

Keywords: FPGA · NTT · DSP · FHE · BFV

1 Introduction

Fully Homomorphic Encryption (FHE) enables arbitrary computations to be performed directly on encrypted data without requiring decryption, thereby preserving confidentiality throughout the entire computation process. Since Gentry's seminal construction [Gen09], lattice-based FHE schemes such as BFV, BGV, and CKKS [Bra12,FV12,BGV12,CKKS17] have evolved into practical cryptographic tools supporting privacy-preserving cloud computing, secure outsourced analytics, and encrypted machine learning. These schemes rely heavily on arithmetic over large polynomial rings of the form $R_{q,n} = \mathbb{Z}_q[x]/(x^n + 1)$, where polynomial degrees typically range from 2^{10} to 2^{16}. In most FHE settings, the Number Theoretic Transform (NTT) dominates the computation and becomes the primary bottleneck. Designing a high-performance NTT unit is therefore critical for building efficient and scalable FHE architectures. In certain higher-level operations, such as the so-called relinearization operation, batch NTTs must be performed efficiently.

While significant progress has been made in optimizing FHE software libraries, CPU [BKS+21] and GPU [DÖS15] platforms remain limited by instruction-level parallelism, memory bottlenecks, and power inefficiency when executing large-degree polynomial multiplication and heavy modular arithmetic operations. In contrast, FPGAs offer fine-grained spatial parallelism, customizable datapaths, and deterministic low-latency execution, which makes them particularly attractive for FHE workloads. Prior FPGA-based accelerators span a wide design spectrum, ranging from low-latency architectures [KLP19] to deeply pipelined high-throughput designs [WYW20]. Hierarchical NTT methods, such as four-step algorithm [Bai89] and its recursive application, the seven-step algorithm, decompose a large NTT into smaller sub-NTTs with multi-dimensional decomposition such as 2-D or 3-D [KKT+25]. Architectures implementing this algorithm, such as [KKT+25, GVBLP+23, ZWZ+21, GZH+24], improve memory locality but introduce additional intermediate modular multiplication blocks, increasing arithmetic complexity. The iterative NTT algorithm [RVM+14], on the other hand, avoids the additional multiplication stages needed by the four-step. Although many studies build upon multi-dimensional decomposition within the four-step NTT algorithm [WG23, GZH+24, KKK+22], the multi-dimensional decomposition of the iterative NTT has not been extensively explored.

Moreover, the number of dimensions in the decomposition (e.g., whether 2-D, 3-D, or higher) is a critical factor in the overall hardware complexity of the NTT architecture. Depending on the target ring dimension n and the desired level of parallelism, different numbers of dimensions may be optimal. Therefore, to support diverse scenarios and ensure portability, a parametric design with respect to the decomposition dimensionality is desirable.

To enable efficient batch operations, a natural solution in hardware is pipelining. In the context of NTT architectures, although fully pipelined designs exist [YKKP22, GZH+24], they are limited to small ring sizes, e.g., 2^{12}. There remains significant room for improvement in FPGA-based NTT designs to achieve fully pipelined architectures for large ring dimensions, e.g., 2^{16}. Targeting larger ring sizes is particularly important, as it allows for greater multiplicative depth in FHE ciphertexts, thereby enabling more operations to be performed in a privacy-preserving manner. Moreover, bootstrapping is typically feasible at dimensions around 2^{16} [CCS19]; this operation resets the so-called noise budget, enabling an effectively unbounded number of multiplications. Batch NTTs thus constitute a primary performance bottleneck in many FHE primitives, including operations such as relinearization.

Another critical performance factor in FHE hardware is modular multiplication and reduction. While special-form primes and Goldilocks moduli can simplify reduction, they limit parameter flexibility and hardware portability [TCW+24]. Recent work on modular reduction in FPGAs has shown that further DSP savings can be achieved by adapting the reduction algorithm to the available DSP sizes [TKKA26]. On the other hand, conventional uniform-radix multipliers under-utilize FPGA DSP resources when operand sizes do not match native word widths [RMIT14]. Recent non-standard integer multiplication techniques demonstrate that carefully tailored DSP multipliers can significantly improve efficiency [DP09, TKKA26]. Leveraging the most efficient modular arithmetic modules in the butterfly units of an NTT design directly scales to the overall hardware complexity of the entire architecture. As a result, by improving the area efficiency of the NTT, the level of parallelism can be increased while reserving resources for additional blocks that implement other FHE primitives. Consequently, a larger portion of the computation can be kept within the pipeline, reducing costly intermediate accesses to global memory. In this work, we present TP-NTT, a throughput-oriented hierarchical NTT architecture that combines configurable multi-dimensional decomposition, DSP-efficient modular arithmetic, and fully pipelined execution for batch NTTs. Unlike existing hierarchical FPGA NTT designs, TP-NTT performs the decomposition directly on the iterative NTT algorithm, eliminating

the intermediate twiddle multiplication stages required by four-step approaches while supporting scalable throughput and ring dimensions up to 2^{16}. We further demonstrate the effectiveness of TP-NTT by integrating it into a complete FPGA accelerator for the BFV relinearization operation, achieving substantial end-to-end speedups over the state-of-the-art `OpenFHE` software implementation.

Related Work. Table 1 presents the qualitative comparison of our study with the existing literature. Recall that FHE applications typically target ring dimensions n from 2^{10} to 2^{16}, with the latter being particularly important as it enables bootstrapping, and coefficient moduli q with bit lengths of approximately 32 or 64 bits [BCC+24]. Many studies do not target the large ring dimension 2^{16} [SFK+21, DMOS22, MKO+22, YCH22, YKKP22, MSL25, GZH+24, AKK+23]. We note that the challenges evolve substantially as the problem size (i.e., n) scales. Among the studies supporting the ring dimension 2^{16}, only a limited number provide high-speed solutions [KKT+25, GVBLP+23, KDNL23, GZX26, KHF25, MSDL24, ZYT+26], where the maximum reported average latency (i.e., the average NTT completion time for a large batch of NTT operations [KKT+25]) for $n = 2^{16}$ within this group is 66μs. Notably, the performance of [GVBLP+23, KKT+25, ZYT+26] further stands out, with average latencies of less than 11μs, whereas the next smallest reported latency is 39.14μs. However, we note that [GVBLP+23] targets ASIC platforms, and it supports a 32-bit coefficient modulus. The advantage of these works mainly comes from the fact that they provide pipelined NTT architectures, which enable very efficient batch NTT computations and thereby minimize the average latencies. Note that other solutions offering pipelined architectures are limited in the supported n [GZH+24, YKKP22]. Similar to [GVBLP+23], [SFK+22] presents a pipelined architecture, but is limited to 28-bit, although it supports $n = 2^{16}$. On the other hand, while many studies perform multi-dimensional decomposition, most of them use a fixed dimensionality (such as 2-D) [KKT+25, ZWZ+21, SFK+21, SFK+22, KKK+22, GVBLP+23, GZH+24], while only a few allow decomposition with a parametric dimensionality [WG23, YKKP22, ZYT+26]. Again, note that [WG23, YKKP22] do not provide high-performance solutions for $n = 2^{16}$. Moreover, most of them employ the four-step NTT approach (labeled as 4S), thereby increasing the required number of modular multiplications [KKT+25, WG23, ZWZ+21, SFK+21, SFK+22, KKK+22, GVBLP+23, GZH+24, ZYT+26] compared to the iterative NTT approach (labeled as Iter.). Compared to the recent multi-dimensional decomposition architectures [KKT+25, WG23, YKKP22, ZYT+26], TP-NTT differs in several key aspects. While [KKT+25, ZYT+26] employ recursive four-step NTT decompositions, TP-NTT performs multi-dimensional decomposition directly on the iterative NTT algorithm, eliminating the intermediate twiddle multiplication stages required by the four-step approach. Although [WG23, YKKP22, ZYT+26] support configurable decomposition dimensionality, TP-NTT additionally formulates the memory and hardware complexity with respect to both the decomposition parameters and throughput, thereby enabling systematic architecture optimization. Finally, unlike [WG23, YKKP22], TP-NTT simultaneously supports large ring dimensions up to 2^{16} and a fully pipelined architecture for high-throughput batch NTT execution. Last but not least, the source code is publicly available for only a few studies [KKT+25, KHF25, DMOS22, MKO+22, HMR24, SFK+22].

Our Contributions. In this work, we address these challenges and bridge the gaps in existing NTT literature. In summary:

- We propose a scalable, throughput-optimized NTT architecture, TP-NTT, which supports 2-D, 3-D, and 4-D decompositions optimized for high performance in the parameter range used in FHE applications. The decomposition is configurable at design time, including the radix size along each dimension. Leveraging this flexibility, we formulate the hardware complexity with respect to the desired throughput and

target ring dimension, enabling systematic optimization. Furthermore, the design-time-configurable throughput allows tuning to match the input/output bandwidth of the target platform.

- We propose optimizations at multiple levels, ranging from DSP-optimized modular arithmetic to the NTT algorithm level. Moreover, our design avoids the additional modular multiplication blocks typically required in the four-step NTT, which is commonly used in the literature with multi-dimensional decomposition, by performing the decomposition on the iterative NTT approach.

- Our design methodology reveals how to scale a pipelined NTT to high levels of parallelism, thereby increasing throughput. Furthermore, our design supports large ring degrees such as 2^{16} in a fully pipelined manner.

- We propose a hardware accelerator for FHE relinearization that leverages the fast batch NTTs performed by TP-NTT. The design is throughput-oriented and can easily scale to different throughput settings, tuned according to available FPGA resources and global memory input/output bandwidth.

- We evaluate TP-NTT and the relinearization accelerator across a wide range of ring sizes and moduli, demonstrating up to an order-of-magnitude improvement in average latency and substantial area–time-product gains. The designs are fully open source, enabling reproducibility and further research.

Table 1: Qualitative comparison with the state-of-the-art NTT architectures

Work	**Max.** n	**Max.** $\log q$	**Pipe-lined**	**Algo-rithm**	**Decom-position**	**High-Speed** ($n = 2^{16}$)	**Target**	**Open-Source**
TP-NTT	2^{16}	60	✓	Iter.	*-D	✓✓	XCU280	✓
[KKT+25]	2^{16}	64	✓	4S	4-D	✓✓	XCU280	✓
[WG23]	2^{28}	256	✗	4S	*-D	✗	XCU250	✗
[ZWZ+21]	2^{20}	256	✗	4S	2-D	✗	ASIC	✗
[SFK+21]	2^{14}	32	✓	4S	2-D	NR	ASIC	✗
[SFK+22]	2^{16}	28	✓	4S	2-D	NR	ASIC	✓
[KKK+22]	2^{17}	64	✗	4S	3-D	NR	ASIC	✗
[HMR24]	2^{16}	256	✗	Iter.		✗	V7	✓
[GVBLP+23]	2^{16}	32	✓	4S	2-D	✓✓	ASIC	✗
[DMOS22]	2^{10}	30	✗	Iter.		✗	V7/A7	✓
[MKO+22]	2^{12}	60	✗	Iter.		✗	V7	✓
[YCH22]	2^{12}	60	✗	Iter.		✗	V7	✗
[YKKP22]	2^{13}	52	✓	Iter.	*-D	✗	XCU200/V7	✗
[MSL25]	2^{12}	30	✗	Iter.		✗	XCU250	✗
[KDNL23]	2^{16}	60	✗	Iter.		✓	VU37P	✗
[GZH+24]	2^{12}	64	✓	Iter.	3-D	✗	AU280	✗
[GZX26]	2^{16}	64	✗	Iter.		✓	Ultrascale	✗
[AKK+23]	2^{14}	32	✗	Iter.		✗	AU280	✗
[KHF25]	2^{17}	64	✗	Iter.		✓	AU200	✓
[MSDL24]	2^{17}	60	✗	Iter.		✓	XCU250	✗
[ZYT+26]	2^{16}	64	✓	4S	*-D	✓✓	XCU280	✗

NR: Not Reported. *-D: Configurable decomposition.

2 Background

2.1 Notation

Lowercase italic letters denote integers, such as a. Bold lowercase letters denote vectors and matrices, such as $\mathbf{a}$. Elements of vectors and matrices are accessed using subscripts, e.g., $\mathbf{a}_i$. Sometimes, we use ":" along an axis to denote all elements on that axis; for example, $\mathbf{a}_{:,i}$ denotes the i-th column of $\mathbf{a}$. Element-wise multiplication (Hadamard product) is denoted by $\odot$. $R_{q,n} = \mathbb{Z}_q[x]/(x^n + 1)$ denotes the cyclotomic ring of polynomials of degree n, where n is a power of two and q is the coefficient modulus. Most of the time, we omit the indeterminate (x) when representing polynomials to simplify the notation. The logarithm is base 2 ($\log = \log_2$) unless otherwise stated.

2.2 Number Theoretic Transform (NTT)

Polynomial multiplication is the dominant operation in lattice-based FHE schemes. Given two polynomials $\mathbf{a}(x), \mathbf{b}(x) \in R_{q,n}$, their product can be computed efficiently using the Number Theoretic Transform (NTT) algorithm:

$$\mathbf{a}(x) \cdot \mathbf{b}(x) = \mathsf{INTT}\Big(\mathsf{NTT}(\mathbf{a}(x)) \odot \mathsf{NTT}(\mathbf{b}(x))\Big). \tag{1}$$

This multiplication corresponds to a *negacyclic convolution*. NTT requires $q \equiv 1 \pmod{2n}$, so a primitive $2n$-th root of unity $\psi \in \mathbb{Z}_q$ exists such that $\psi^n \equiv -1 \pmod{q}$. The forward NTT evaluates the polynomial at odd powers of ψ, i.e.,

$$\hat{\mathbf{a}}[i] = \mathbf{a}(\psi^{2i+1}), \quad i = 0, \ldots, n-1. \tag{2}$$

Both forward and inverse NTT can be implemented efficiently using so-called butterfly circuits. In general, Cooley–Tukey [CT65] (CT) butterflies are used for the forward transform and Gentleman–Sande [GS66] (GS) butterflies for the inverse transform. The butterfly operation is the fundamental computational primitive of the NTT and FFT.

A radix-2 CT butterfly computes

$$(u, v) \mapsto (u + \zeta \cdot v, u - \zeta \cdot v) \mod q \tag{3}$$

where ζ denotes the twiddle factor, i.e., an appropriate power of the primitive root of unity ψ. A radix-2 GS butterfly computes

$$(u, v) \mapsto (u + v, (u - v) \cdot \zeta^{-1}) \mod q \tag{4}$$

Leveraging the butterfly circuits, the forward and inverse NTT (FNTT and INTT) can be computed in $\log n$ stages, each containing $n/2$ butterfly operations, resulting in $\mathcal{O}(n \log n)$ complexity. This represents a significant advantage over the naive schoolbook approach, which has $\mathcal{O}(n^2)$ complexity. The INTT has the same $\mathcal{O}(n \log n)$ complexity as the NTT, and its final scaling by n^{-1} can be merged into the last GS butterfly stage [LN16]. Thus, polynomial multiplication consists of two NTTs, one Hadamard product ($\mathcal{O}(n)$), and one INTT, yielding an overall complexity of $\mathcal{O}(n \log n)$.

2.3 BFV Homomorphic Encryption Scheme

The Brakerski–Fan–Vercauteren [Bra12,FV12] (BFV) scheme is a lattice-based FHE scheme designed for exact arithmetic over integers. It is based on the Ring Learning With Errors (RLWE) problem and operates over the ring $R_{q,n}$. Here, $R_{q,n} = \mathbb{Z}_q[x]/(x^n + 1)$, where n is the polynomial degree and q is the ciphertext coefficient modulus. In BFV, a plaintext polynomial $\mathbf{m}(x) \in R_{t,n}$ is encrypted into a ciphertext consisting of two polynomials

$(\mathbf{c}_0(x), \mathbf{c}_1(x)) \in R^2_{Q,n}$, where t is the plaintext modulus and Q is the ciphertext modulus. Typically, $Q \gg t$ to provide sufficient noise budget for homomorphic evaluation.

Modern BFV implementations employ a Residue Number System (RNS) representation, where the large modulus Q is decomposed into a product of smaller primes $Q = \prod_{i=0}^{L-1} q_i$. Here, L denotes the number of RNS moduli and each q_i is a machine-word-sized prime selected to support efficient modular arithmetic and NTT operations. Each polynomial coefficient is represented by its residues modulo the q_i, enabling parallel and machine-word-size modular arithmetic.

BFV enables efficient homomorphic addition and multiplication over ciphertexts. While the former is relatively inexpensive, the latter introduces additional noise. Therefore, the complexity of a homomorphic computation is typically determined by its multiplicative depth. Moreover, after homomorphic multiplication between ciphertexts $(\mathbf{a}_0, \mathbf{a}_1)$ and $(\mathbf{b}_0, \mathbf{b}_1)$, the resulting ciphertext consists of three polynomial components $(\mathbf{c}_0, \mathbf{c}_1, \mathbf{c}_2)$. These components are obtained by polynomial multiplication of the input ciphertexts, namely

$$\mathbf{c}_0 = \mathbf{a}_0\mathbf{b}_0, \qquad \mathbf{c}_1 = \mathbf{a}_0\mathbf{b}_1 + \mathbf{a}_1\mathbf{b}_0, \qquad \mathbf{c}_2 = \mathbf{a}_1\mathbf{b}_1,$$

where all polynomial operations are performed in the ring $R_{Q,n}$. The operation known as relinearization reduces this back to the standard two-component form $(\mathbf{c}'_0, \mathbf{c}'_1)$ using public relinearization keys. In the linear form, the decryption formula is linear in the secret polynomial s:

$$t = \mathbf{c}'_0 + \mathbf{c}'_1 \cdot s \tag{5}$$

On the other hand, in the three-component form, it is quadratic:

$$t = \mathbf{c}_0 + \mathbf{c}_1 \cdot s + \mathbf{c}_2 \cdot s^2. \tag{6}$$

Therefore, for relinearization, $\mathbf{c}_2 \cdot s^2$ is computed homomorphically and aggregated into $(\mathbf{c}_0, \mathbf{c}_1)$. The encryption of s^2 is known as the relinearization key, $\mathbf{rk}$. The relinearization key enables key switching from the quadratic decryption form back to the standard linear form without revealing the secret key. In RNS form, to improve noise management, $\mathbf{rk}$ is typically stored in a special matrix of dimension $L \times (L+1)$, and $\mathbf{c}_2 \cdot s^2$ is computed using a specialized technique referred to as the external product.

3 Proposed NTT Architecture: TP-NTT

In this section, we present the hardware architecture of our proposed NTT design, TP-NTT (**T**hrough**P**ut-optimized**NTT**). In a nutshell, the architecture decomposes large-size negacyclic NTTs into pipelined, throughput-oriented processing units. The design supports flexible ring sizes n and throughput parameters (denoted by τ), enabling adaptation to diverse resource budgets and application requirements. n, τ, and $\log q$ are design-time parameters, while q and the direction of the NTT (forward or inverse) can be configured at runtime. It is composed of two main building blocks: NTT Cores (NCs) and Automorphism Units (AUs). NCs are responsible for performing butterfly operations, while AUs handle the permutation of data arrays between NCs. For a 3-D decomposition, the main hardware architecture of TP-NTT is illustrated in Figure 1. The design is fully pipelined, enabling highly efficient batch NTT operations. Accordingly, both NCs and AUs are implemented as fully pipelined modules.

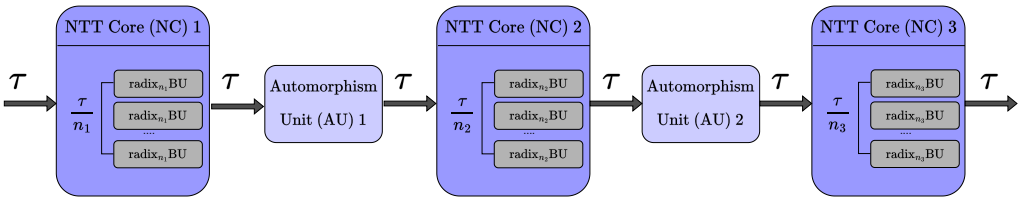


Figure 1: TP-NTT architecture for 3-D decomposition

3.1 Multi-Dimensional Decomposition

TP-NTT decomposes the NTT size n into a product of smaller radices across d dimensions,

$$n = \left(\prod_{i=1}^{d} n_i\right) = n_1 \cdot n_2 \cdot \ldots \cdot n_d, \quad d \in \{2, 3, 4\}, \tag{7}$$

where each n_i denotes the radix of an NC execution stage. As shown, our design supports 2-D, 3-D, and 4-D decompositions, which we show to be sufficient for the range of n used in FHE, as discussed in Section 5. Along each dimension, TP-NTT computes radix-n_i butterfly circuits in a fully pipelined manner, processing τ input and output elements per clock cycle. In this paper, we consider radix-n_i butterfly circuits as compositions of $n_i/2$ radix-2 butterfly circuits across $\log n_i$ stages, without writing intermediate data to memory structures such as BRAM.

The desired decomposition is configured at design time. However, there exist certain constraints and design guidelines on the choice of the dimension d and the radices n_i. The dimension d is selected based on the target n, τ and architectural constraints such as available FPGA resources, allowing the same hardware to support multiple parameter sets by adjusting the decomposition. The choice of n_i is non-uniform and architecture-driven. A constraint in our design is that each dimension must satisfy $n_i \leq \tau$. Otherwise, it is not possible to perform a radix-n_i butterfly circuit in a fully pipelined manner, as the required input/output bandwidth of size n_i cannot be sustained with the given throughput parameter τ. In such cases, one must increase d to reduce each n_i. TP-NTT includes d NCs and $d-1$ AUs for a d-dimensional decomposition. Therefore, as a rule of thumb, the smallest d satisfying this constraint should be chosen, as it minimizes the aggregated cost of AUs. In contrast, the aggregated cost of the NCs is independent of d (in terms of butterfly circuits). The selection of the decomposition dimension d and the radices n_i follows several architecture-oriented design principles. First, the smallest decomposition dimension satisfying $n_i \leq \tau$ is selected, i.e., $d = \lceil \log_\tau n \rceil$. For a 2-D decomposition, a symmetric partition ($n_1 = n_2$) is preferred whenever possible to simplify the implementation. For a 3-D decomposition, we choose $n_1 = n_3 = \tau$ whenever feasible and assign the remaining factor to n_2. Since the RAM requirement of AU_1 is proportional to $2n_1n_2$, maximizing n_3 minimizes n_2 and therefore reduces the memory requirement of AU_1. For a 4-D decomposition, we similarly set $n_1 = n_4 = \tau$ whenever possible and partition the remaining factor as evenly as possible between n_2 and n_3. This minimizes the RAM requirements of both AU_1 ($2n_1n_2$) and AU_3 ($2n_3n_4$), while a near-symmetric partition of n_2 and n_3 balances the memory requirements of the two AUs. In Table 2, we present the optimal decompositions of n with respect to different τ used in this work. In many cases, multiple decompositions may be equally optimal; here, we show only one representative option for each scenario. The reasoning behind these selections is provided in Subsection 3.2 and Subsection 3.4.

Table 2: Optimal decompositions

	τ			
n	2^4	2^5	2^6	2^7
2^{10}	2^4, 2^2, 2^4	2^5, 2^5	2^5,2^5	2^5,2^5
2^{11}	2^4, 2^3, 2^4	2^5, 2^1, 2^5	2^6,2^5	2^6,2^5
2^{12}	2^4, 2^4, 2^4	2^5, 2^2, 2^5	2^6,2^6	2^6,2^6
2^{13}	2^4, 2^2, 2^3, 2^4	2^5, 2^3, 2^5	2^6,2^1,2^6	2^7,2^6
2^{14}	2^4, 2^3, 2^3, 2^4	2^5, 2^4, 2^5	2^6,2^2,2^6	2^7,2^7
2^{15}	2^4, 2^3, 2^4, 2^4	2^5, 2^5, 2^5	2^6,2^3,2^6	2^7,2^1,2^7
2^{16}	2^4, 2^4, 2^4, 2^4	2^5, 2^3, 2^3, 2^5	2^6,2^4,2^6	2^7,2^2,2^7

3.2 NTT Core (NC)

The NTT Cores (NCs) implement the pipelined radix-n_i butterfly stages of the negacyclic NTT. Each NC includes τ/n_i independent and identical pipelined radix-n_i Butterfly Units (BUs). Each radix-n_i BU contains $\log n_i \cdot n_i/2$ radix-2 BUs. In our terminology, BUs (when the radix is not specified) refer to units that implement radix-2 butterfly circuits, presented in Subsection 3.3.

Because each radix-n_i stage operates on relatively small sub-transforms, it requires fewer computational resources while still contributing to the overall pipeline throughput. Each NC requires

$$\frac{\tau}{n_i} \cdot \frac{n_i}{2} \cdot \log n_i = \frac{\tau}{2} \log n_i \tag{8}$$

BUs. Summing across the NCs in all stages, the total number of BUs required for the overall NTT design is

$$\frac{\tau}{2} \cdot \log n. \tag{9}$$

Therefore, the total resource requirement is independent of the specific decomposition parameters n_i and d and depends only on τ. As a result, the decomposition strategy does not affect the overall resource usage in terms of BUs.

Twiddle Factor Storage. The NCs utilize on-chip memory resources such as Block RAM (BRAM) or UltraRAM (URAM) to store the twiddle factors required for the corresponding sub-transforms. When the coefficient modulus q is specified at runtime, the corresponding n unique twiddle factors are loaded into on-chip memory and streamed to the NC BRAMs/URAMs using decomposition-specific placement strategies during NTT execution. This approach eliminates the need to store the complete set of twiddle factors for all NTT stages in a central manner, thereby providing more flexible on-chip memory requirements. For a radix-n_i butterfly structure, the number of required twiddle factors is $n_i - 1$. However, each radix-n_i butterfly is executed with a different set of twiddle factors. Therefore, the number of twiddle factors that must be stored in the memory of an NC depends on the dimension index i of the NC. For instance, the first dimension always uses the same set of twiddle factors, whereas the last dimension uses n/n_{d-1} different sets of twiddle factors, resulting in a total of $(n_d - 1) \cdot n/n_d$ twiddle factors to be stored. In general, the number of distinct twiddle factors used by NC i for the forward NTT is given by

$$\Big(\prod_{k=1}^{i-1} n_k\Big) \cdot \Big(n_i - 1\Big) \tag{10}$$

The above analysis considers a forward NTT. On the other hand, in the inverse NTT, the storage requirements are reversed. The last dimension uses the least distinct twiddles while the number of distinct twiddle factors increases towards the first dimension. The number of required distinct twiddle factors for the i-th NC in the inverse NTT is given by

$$\Big(\prod_{k=i+1}^{d} n_k\Big) \cdot \Big(n_i - 1\Big) \tag{11}$$

Since our proposed architecture is unified for both forward and inverse NTTs, we take the maximum of the two cases:

$$\mathsf{R}_d(i) = \mathsf{max}\Big(\Big(\prod_{k=1}^{i-1} n_k\Big), \Big(\prod_{k=i+1}^{d} n_k\Big)\Big) \cdot \Big(n_i - 1\Big) \tag{12}$$

The second term, $(n_i - 1)$, specifies the number of independent RAMs, while the first term determines the depth of each RAM. τ does not affect the storage requirements of the twiddle factors, as it does not impact the stage indices of the NTT handled by an NC. For the 2-D case, it is straightforward to see that $\mathsf{R}_2(i)$ is independent of the radices n_i. The same observation holds for the first and last dimensions in 3-D and 4-D decompositions, namely $\mathsf{R}_3(1)$ and $\mathsf{R}_3(3)$ for 3-D, and $\mathsf{R}_4(1)$ and $\mathsf{R}_4(4)$ for 4-D. On the other hand, $\mathsf{R}_d(i)$ for intermediate dimension indices i is minimized when the radices of the first and last dimensions are maximized. In particular, $\mathsf{R}_3(2)$ is minimized when n_1 and n_3 are maximized. The same observation applies to $\mathsf{R}_4(2)$ and $\mathsf{R}_4(3)$ in the 4-D case. This optimization reduces RAM usage in practice by minimizing the twiddle factor storage requirements.

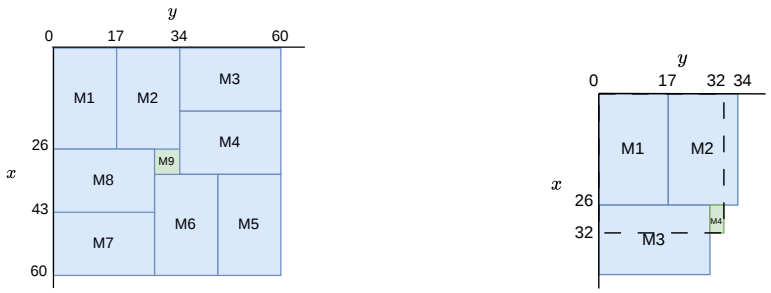
3.3 Butterfly Unit (BU)

Recall from Section 2 that every NTT stage computes $n/2$ butterfly circuits. The butterfly circuit is the core operation in any NTT design, and consequently, its complexity directly affects the overall NTT implementation. In our architecture, we refer to the units responsible for implementing these circuits as Butterfly Units (BUs). These implement both CT butterflies (Equation 3) and GS butterflies (Equation 4) used in forward and inverse NTTs, respectively, in a unified architecture.

From a hardware perspective, the performance of a butterfly is dominated by modular multiplication, which itself consists of two main steps: integer multiplication and modular reduction. Optimizing these operations has a direct impact on the overall NTT design. In our work, we optimize the integer multiplication and modular reduction strategies with respect to DSP operand sizes in FPGA architectures to minimize the DSP usage of BUs. The optimizations discussed in the rest of this section are adopted from the literature.

Integer Multiplication. Standard integer multiplication techniques often assume fixed word sizes aligned with processor architectures, such as partitioning a 64×64-bit integer multiplication into 16 core multiplications of 16×16-bit. In contrast, DSPs in FPGA architecture offer asymmetric operand sizes, such as 27×18 in DSP48E2[1]. A trivial strategy to partition a 64×64 -bit multiplication using 27×18 -bit DSP multipliers would lead to $12 = \lceil 64/27 \rceil \times \lceil 64/18 \rceil$ core multiplications executed by DSPs, which is known as standard tiling. However, more efficient approaches exist. [DP09] demonstrates that a non-standard tiling approach can further reduce the number of DSP multiplications required. For operand sizes 27×18, a 60×60-bit multiplication can be realized using 8 DSPs along with a small LUT-based multiplier. This strategy is illustrated in Figure 2a.

[1] https://docs.amd.com/r/en-US/ug958-vivado-sysgen-ref/DSP48E2



(a) 60×60-bit [DP09] (b) 32×32-bit

Figure 2: Non-standard tiling for integer multiplication. The x- and y-axes represent the binary representations of the multiplication operands x and y, respectively. Each rectangle (M_i) denotes a partial multiplication corresponding to the indicated bit ranges of x and y, implemented using either a 27×18-bit DSP multiplier (blue) or LUT-based distributed logic (green).

Using a similar approach, a 32×32-bit multiplication can be implemented with 3 DSPs instead of 4 DSPs required by standard tiling, as shown in Figure 2b.

Modular Reduction. After computing the double-width integer product during modular multiplication, the result must be reduced modulo q. Optimized reduction techniques such as Montgomery [Mon85] or Barrett [Bar86] reduction are often employed. To further reduce DSP utilization, we utilize the mixed-radix Montgomery reduction algorithm presented in [TKKA26], which is a modified variant of the word-level Montgomery reduction approach [MÖS20]. In the employed variant, the word-level reduction iterations are arranged according to DSP operand sizes, at the cost of slightly reducing the number of primes used given the ring dimension n and modulus q. It is shown by [TKKA26] that this decrease in the number of primes does not affect the correctness of the FHE construction, as there are still significantly more primes than required for the RNS representation. For example, in the case of a 60-bit modulus and 27×18 DSP operand sizes, the reduction iteration lengths are 34-bit and 26-bit, requiring 3 DSPs in total for modular reduction. In this configuration, q is formed as $q_h \cdot 2^{43} + 1$, with the lower 43 bits fixed. For the 32-bit case, 2 DSPs are sufficient for modular reduction. With different DSP operand sizes, the configuration may vary slightly but can be adapted with similar advantages. We refer the reader to [TKKA26] for a detailed breakdown of DSP usage in both scenarios. Notably, this reduction approach allows the modulus to be defined efficiently at runtime, which is essential for batch NTTs and FHE schemes, whereas some NTT implementations hardcode the modulus [ACM+23].

3.4 Automorphism Unit (AU)

Data rearrangement is essential in hierarchical NTTs to avoid pipeline stalls and maintain data locality. Automorphism Units (AUs) are used in hierarchical execution to permute input vectors according to the processing pattern of the next stage. Specifically, they adjust the stride of input data vectors in a fully pipelined manner. To illustrate the functionality of AUs, we present a toy example in Figure 3. The example considers an NTT with $n = 16$ and $\tau = n_1 = n_2 = 4$. The input data stride is 4, corresponding to entries at indices $k, k+4, k+8, k+12$ for $k \in \{0, 1, 2, 3\}$. Before the next phase of computation, however, the coefficients must be realigned into contiguous blocks with a stride of 1, i.e., $k', k'+1, k'+2, k'+3$ for $k' \in \{0, 4, 8, 12\}$. Without this realignment, the pipeline would

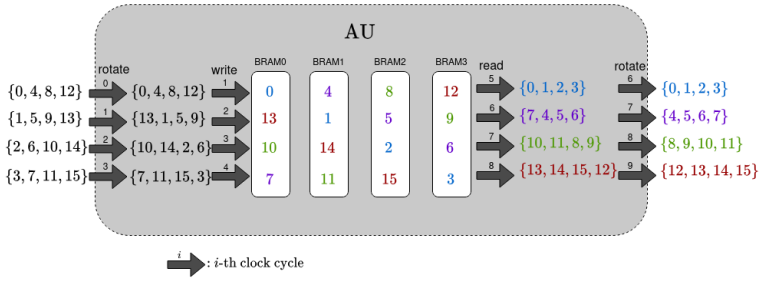


Figure 3: Illustration of AU for $n = 16$, $n_1 = n_2 = 4$ and $\tau = 4$.

incur idle cycles while waiting for data in the required order. The AU performs this task, permuting vector elements to satisfy interstage access patterns.

AUs employ on-chip memory (BRAMs or URAMs) along with two rotation circuits. The rotations are performed both before writing to RAM and after reading from RAM. To meet the throughput constraint, τ independent RAMs are used. The depth of each RAM depends on the decomposition parameters as well as the AU index i. For a 2-D decomposition, the depth of each RAM in an AU is $2 \cdot n/\tau$, resulting in $2n$ total entries. The factor of 2 arises from using a ping-pong memory scheme to support pipelining: one half of the memory stores the coefficients for the current NTT operation, while the other half is used by the subsequent operation.

AU 2 in a 4-D decomposition operates in the same manner, with identical RAM requirements and operating principles. Note that AU 2 is the middle unit in the design over all AUs. Conceptually, the 4-D decomposition can be viewed as a recursive application of 2-D decompositions. Consequently, the RAM requirements for AU 1 and AU 3 follow immediately. The total RAM size in AU 1 is $2 \cdot n_1 \cdot n_2$, while for AU 3 it is $2 \cdot n_3 \cdot n_4$.

For a 3-D decomposition, the situation is similar but asymmetric. We consider AU 2 as the middle AU, analogous to AU 2 in the 4-D case. Consequently, the total RAM size of AU 2 is $2 \cdot n$. For AU 1, we treat it as a recursive application of a 2-D decomposition. The total RAM size for AU 1 is therefore $2 \cdot n_1 \cdot n_2$. Notice that the size of the sub-transforms prior to AU 2 is $n_1 \cdot n_2$.

The RAM sizes for AU 1 in the 2-D case, and for the middle AU (namely AU 2) in 3-D and 4-D, are independent of the selected n_i. In contrast, the RAM size for AU 1 in the 3-D case is minimized when n_3 is maximized. For the 4-D case, the total RAM size of AU 1 and AU 3 is minimized when $n_1 \cdot n_2$ and $n_3 \cdot n_4$ are close, i.e., when the radices are symmetric or nearly symmetric.

The critical path of the AU increases with the degree of parallelism τ, since larger values require wider rotation and multiplexing networks. Beyond a certain τ, the delay introduced by the rotation logic becomes the dominant contributor to the critical path, reducing the achievable operating frequency.

3.5 Extended Architecture

In this section, we present an extended architecture as a wrapper layer for the TP-NTT architecture, which can be useful in certain scenarios. The extended architecture is illustrated in Figure 4. In the figure, NTT Core Subsystem refers to the TP-NTT architecture presented in Figure 1. One of the units in the extended architecture is the BitReverse AU. Recall that TP-NTT is a unified architecture for both forward and inverse NTTs, with the direction determined at runtime. Therefore, a bit-reversal mechanism is required to ensure a consistent data access pattern for both transformations. The operating principle of the BitReverse AU is the same as that of the AUs described in Subsection 3.4.

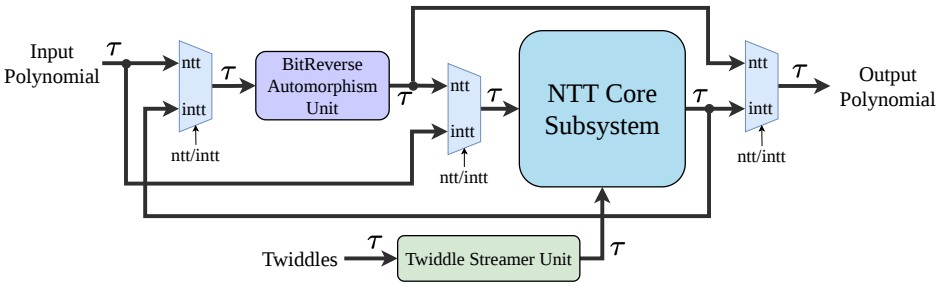


Figure 4: TP-NTT extended architecture.

However, it represents a special case, as it permutes the input vectors by bit-reversing their indices. Specifically, for the forward NTT, the input coefficients are first sent to the BitReverse AU for pre-processing, while the output of the NTT Core Subsystem is used directly. In contrast, for the inverse NTT, the input is left unchanged, and the output of the NTT Core Subsystem is forwarded to the BitReverse AU. The RAM cost of the BitReverse AU is $2n$, implemented using τ independent memory banks. The BitReverse AU adds only lightweight control logic and $2n$ memory, requiring no additional butterfly or modular arithmetic units. It is fully pipelined, introducing only a one-time latency of n/τ clock cycles while preserving the steady-state throughput. The BitReverse AU ensures consistency between the input and output orders of NTT operations with respect to the normal order. Without it, the input of the forward NTT and the output of the inverse NTT would be in bit-reverse order, which is acceptable for some applications. Therefore, we retain the BitReverse AU in the extended architecture.

Another unit in the extended architecture is the Twiddle Streamer Unit. Recall from Subsection 3.2 that the number of independent RAMs used in each NC for twiddle storage is less than or equal to the data throughput, i.e., $n_i \leq \tau$. On the other hand, τ is conceptually the data input/output bandwidth. Therefore, it is reasonable to assume that twiddle factors are received in batches of τ elements per clock cycle. Since the number of independent RAMs also constrains the maximum number of write operations that can be performed in a single cycle, a mechanism is needed to systematically store the incoming τ twiddle factors and distribute them to the NCs according to their respective n_i. This is the role of the Twiddle Streamer Unit. We retain this unit in the extended architecture because many applications can tolerate receiving twiddle factors at a slightly reduced rate, avoiding the need for additional hardware. Figure 5 exemplifies the operation of the Twiddle Streamer Unit for $n = 16$, $n_1 = n_2 = 4$ and $\tau = 4$. In this setting, the overall design inputs τ twiddles per cycle while only $\tau - 1$ twiddles are required for both NC 1 and NC 2. Therefore, the final twiddles of the stream, *i.e.* $\{12, 13, 14\}$, are appended to the set of twiddles sent at each clock cycle -one per cycle- to form complete τ-word transfers. These appended twiddles are subsequently restored to their correct functional positions during subsequent memory row accesses. For instance, when $\{0, 1, 2, 12\}$ are read, twiddle-12 is written back to BRAM 0 for future read. This approach maintains a regular sequential memory access pattern while avoiding wasted memory bandwidth. This strategy is formulated for each supported n, d, and the related decomposition.

3.6 Relation to Hierarchical NTTs

Our formulation is similar to the classical four-step FFT algorithm introduced by Bailey [Bai89], which decomposes a large NTT into smaller sub-NTTs to improve data locality and reduce global data movement. Hierarchical NTT architectures implementing the four-step algorithm, such as in [KKT+25, GVBLP+23, WG23, SFK+22], demonstrate improved

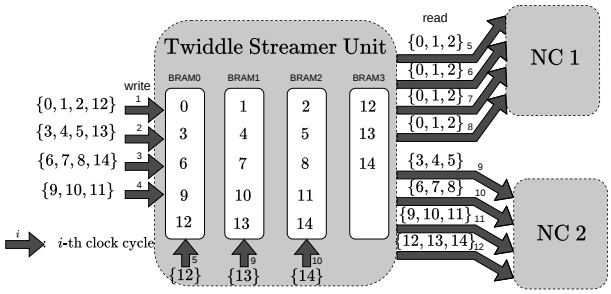


Figure 5: Illustration of Twiddle Streamer Unit for $n = 16$, $n_1 = n_2 = 4$ and $\tau = 4$.

Table 3: Quantitative analysis of storage requirements and modular multiplier counts.

Algorithm	**Decomposition**	**Storage**	**# MM**
Iterative	2-D	$4n$	$\frac{\tau}{2}\log_2 n$
Iterative	3-D	$2(n_1n_2 + n) + 3n$	$\frac{\tau}{2}\log_2 n$
Iterative	4-D	$2(n_1n_2 + n + n_3n_4) + 4n$	$\frac{\tau}{2}\log_2 n$
Seven-step	4-D	$2(n_1n_2 + n + n_3n_4) + n$	$\frac{\tau}{2}\log_2 n + 3\tau$
Seven-step (OTF)	4-D	$2(n_1n_2 + n + n_3n_4) + \sqrt{n}$	$\frac{\tau}{2}\log_2 n + 4\tau$

performance over purely iterative designs by exploiting data locality and reducing global memory traffic. However, these approaches require an additional n modular multiplications with twiddle factors between sub-NTTs. In contrast, the architecture presented in this work preserves the advantages of hierarchical execution while eliminating intermediate twiddle multiplications between blocks. Specifically, we implement the iterative NTT algorithm [RVM+14] by decomposing it into radix-n_i butterfly circuits, which are analogous to the smaller sub-NTTs in the four-step algorithm. The data access pattern of our approach is identical to that of the four-step algorithm.

For example, with a 4-D decomposition, $n = 2^{16}$, and $\tau = 2^4$, a recursive application of the four-step NTT would be required, as in [KKT+25]. In that case, three intermediate multiplication blocks would be needed, and the modular multiplication cost for each block is equivalent to that of two NTT stages, as each NTT stage performs $\tau/2$ modular multiplications, whereas the intermediate multiplication blocks perform τ. This results in a 6/16 increase in total modular-multiplication-related hardware cost. A drawback of our approach is that it requires more storage, since the twiddle factors must be stored explicitly, whereas in the four-step algorithm some twiddle factors can be computed on-the-fly. Nevertheless, as we show in Section 5, trading memory for modular multiplication units leads to better area outcomes.

Storage and Multiplier Comparison. Table 3 compares the storage and modular multiplier requirements of the considered NTT architectures. For iterative NTT, the multiplier count is fixed at $\frac{\tau}{2}\log_2 n$, while the storage depends on the decomposition. For $n = 2^{16}$ and $\tau = 2^8$, the iterative 2-D design requires 262,144 stored elements and 2,048 modular multipliers, compared to 197,632 (132,352 with OTF) stored elements and 2,816 (3,072 with OTF) modular multipliers for the seven-step counterpart. Similarly, for the 4-D decomposition 2^4–2^4–2^4–2^4 with $\tau = 2^4$, 4-D design requires 394,240 stored elements and 128 modular multipliers, whereas seven-step requires 197,632 (132,352 with OTF) stored elements and 176 (192 with OTF) modular multipliers. Thus, TP-NTT intentionally trades additional on-chip memory for substantially fewer modular multipliers, improving DSP efficiency.

Algorithm 1 NTT-Based Relinearization

Input: Ciphertext components $(\mathbf{c}_0, \mathbf{c}_1, \mathbf{c}_2)$ in RNS form over base $Q = \{q_j\}_{j=0}^{L-1}$, relinearization keys $(\mathbf{rk}_0, \mathbf{rk}_1)$, extended base $\tilde{Q} = \{q_i\}_{i=0}^{L}$
Output: Relinearized ciphertext $(\mathbf{c}'_0, \mathbf{c}'_1)$ over base Q

1: **for** $i' = 0$ **to** L **do**
2: $\quad i \leftarrow (i' == 0)\ ?\ L-1\ :\ i'-1$
3: $\quad \hat{\mathbf{a}}_0 \leftarrow \mathbf{0}$, $\hat{\mathbf{a}}_1 \leftarrow \mathbf{0}$
4: $\quad$ **for** $j = 0$ **to** $L-1$ **do**
5: $\quad\quad \hat{\mathbf{c}}_{2,j} \leftarrow \mathsf{NTT}_{q_i}(\mathbf{c}_{2,j})$ ▷ NTT Unit
6: $\quad\quad \hat{\mathbf{d}}_{0,j} \leftarrow \hat{\mathbf{c}}_{2,j} \odot \mathbf{rk}_{0,i,j} \bmod q_i$ ▷ Hadamard Unit
7: $\quad\quad \hat{\mathbf{d}}_{1,j} \leftarrow \hat{\mathbf{c}}_{2,j} \odot \mathbf{rk}_{1,i,j} \bmod q_i$ ▷ Hadamard Unit
8: $\quad\quad \hat{\mathbf{a}}_0 \leftarrow \hat{\mathbf{a}}_0 + \hat{\mathbf{d}}_{0,j} \bmod q_i$ ▷ Accum. Unit
9: $\quad\quad \hat{\mathbf{a}}_1 \leftarrow \hat{\mathbf{a}}_1 + \hat{\mathbf{d}}_{1,j} \bmod q_i$ ▷ Accum. Unit
10: $\quad \mathbf{a}_0 \leftarrow \mathsf{INTT}_{q_i}(\hat{\mathbf{a}}_0)$ ▷ NTT Unit
11: $\quad \mathbf{a}_1 \leftarrow \mathsf{INTT}_{q_i}(\hat{\mathbf{a}}_1)$ ▷ NTT Unit
12: $\quad$ **if** $(i' == 0)$ **then**
13: $\quad\quad \mathbf{t}_0 \leftarrow \mathbf{a}_0 + q_L/2 \bmod q_L$ ▷ Modulus-Down Unit
14: $\quad\quad \mathbf{t}_1 \leftarrow \mathbf{a}_1 + q_L/2 \bmod q_L$ ▷ Modulus-Down Unit
15: $\quad$ **else**
16: $\quad\quad \mathbf{c}'_{0,i} \leftarrow ((\mathbf{a}_0 - (\mathbf{t}_0 - q_L/2)) \cdot q_L^{-1}) \bmod q_i$ ▷ Modulus-Down Unit
17: $\quad\quad \mathbf{c}'_{1,i} \leftarrow ((\mathbf{a}_1 - (\mathbf{t}_1 - q_L/2)) \cdot q_L^{-1}) \bmod q_i$ ▷ Modulus-Down Unit
18: $\quad\quad \mathbf{c}'_{0,i} \leftarrow \mathbf{c}'_{0,i} + \mathbf{c}_{0,i} \bmod q_i$ ▷ Modulus-Down Unit
19: $\quad\quad \mathbf{c}'_{1,i} \leftarrow \mathbf{c}'_{1,i} + \mathbf{c}_{1,i} \bmod q_i$ ▷ Modulus-Down Unit
20: **return** $(\mathbf{c}'_0, \mathbf{c}'_1)$

4 Relinearization Accelerator

As a case study of the proposed batch NTT architecture, we consider the relinearization operation in BFV. Recall from Section 2 that this operation is typically performed after each homomorphic multiplication to restore the ciphertext to linear form. The relinearization is carried out via an external product with the relinearization keys [FV12], which is highly NTT-dominant, making it a suitable use case for the proposed batch NTT hardware.

We present the relinearization process in Algorithm 1. Given an extended RNS base $\tilde{Q} = q_0, \ldots, q_L$, the external product proceeds independently for each modulus q_i, as reflected in the main loop of the algorithm. For each modulus, L consecutive forward NTTs are performed in a batched manner, taking $\mathbf{c}_{2,j}$, the j-th polynomial of the quadratic term of the ciphertext, as input. In total, $L \cdot (L+1)$ forward NTTs are required.

In addition to forward NTTs, the procedure involves Hadamard products, accumulations, and inverse NTTs. Furthermore, a modulus switching operation, referred to as Modulus-Down, is performed at the end (Lines 12–17). The extended base $\tilde{Q}$, which augments the original RNS basis with an additional modulus q_L, is introduced during homomorphic multiplication. The Modulus-Down operation removes this extra modulus, enabling improved noise management. Finally, in Lines 18–19, the relinearization output is aggregated into the constant and linear components $(\mathbf{c}_0, \mathbf{c}_1)$ of the input ciphertext. Note that after the forward NTT stage, all subsequent operations are duplicated for both components.

Hardware Architecture. We design a hardware architecture that executes all steps of Algorithm 1 under a single command. A simplified view of the architecture is shown

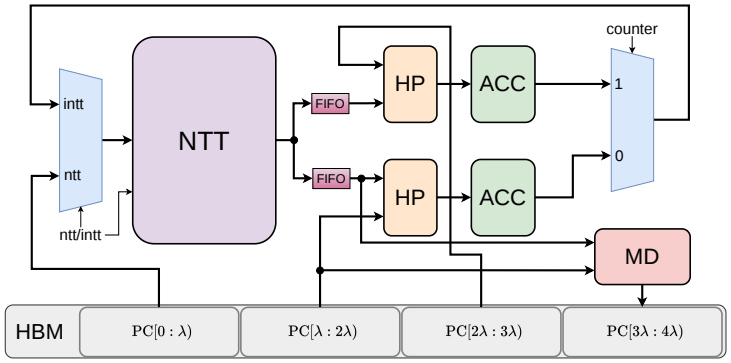


Figure 6: Simplified relinearization architecture.

in Figure 6. Note that the control path is largely omitted for clarity. The proposed relinearization accelerator is organized as a collection of streaming hardware modules, each specialized for a distinct computational phase of the relinearization procedure. Similar to our NTT design, the remaining units in the architecture follow a throughput-oriented datapath, where all modules operate in a pipelined manner and exchange data through on-chip buffers, enabling continuous processing of ciphertext polynomials. In Figure 6, HP, ACC, and MD denote the Hadamard Product, Accumulation, and Modulus-Down units, respectively. In Algorithm 1, each line is annotated with the corresponding command to indicate which hardware unit is responsible for the associated operation.

4.1 HBM and Global Data Access

Since our design targets large-scale FPGAs, we assume that both input and output data reside in the High Bandwidth Memory (HBM). HBM is a high-capacity, high-throughput stacked memory technology integrated with modern FPGA devices, providing substantially higher memory bandwidth than conventional off-chip DDR memory. As shown in Figure 6, the aggregate HBM bandwidth is partitioned into four segments to support continuous streaming of input and output data. Each segment comprises λ pseudo-channels (PCs), which are mapped to dedicated banks in HBM to enable parallel data access. To efficiently utilize the pipelined NTT units, we generalize the throughput parameter τ from the NTT architecture to the relinearization design. We set τ as $\tau = \lfloor \lambda \cdot \Omega / \beta \rfloor$, where Ω denotes the bit-width of each PC at the desired frequency and β denotes the bit-length of the polynomial coefficients, which is either 32 or 60 in our study. With this formulation, we maximize the achievable degree of parallelism under the constraint of the global data access bandwidth.

The first PC segment PC$[0 : \lambda)$ is used to read the input polynomials $\mathbf{c}_{2,j}$ and the twiddle factors. PC$[\lambda, 2\lambda)$ and PC$[2\lambda, 3\lambda)$ are used for reading the relinearization keys $\mathbf{rk}_{0,i,j}$ and $\mathbf{rk}_{1,i,j}$, respectively (see Algorithm 1). Note that $\mathbf{c}_{2,j}$ and $\mathbf{rk}_{:,i,j}$ are accessed in parallel, necessitating the allocation of separate PC segments to them. After the accumulation is done, PC$[\lambda, 2\lambda)$ is repurposed to fetch $\mathbf{c}_{0,i}$ and $\mathbf{c}_{1,i}$ which are used in Modulus-Down operation. At this stage, $\mathbf{rk}_{:,i,j}$ is no longer needed. The last PC segment PC$[3\lambda, 4\lambda)$ is dedicated to writing the output data back to the HBM.

For synchronizing the data between different modules, we employ on-chip RAM-based FIFOs (BRAMs or URAMs). Specifically, consider the output of the NTT, $\hat{\mathbf{c}}_{2,j}$, which is forwarded to the HP unit, where it is multiplied by $\mathbf{rk}_{0,i,j}$ (see Lines 5-6 of Algorithm 1). We use the FIFO buffer to temporarily store $\hat{\mathbf{c}}_{2,j}$. Once $\hat{\mathbf{c}}_{2,j}$ is produced, the data is written into the FIFO. A read request to the HBM is issued to fetch $\mathbf{rk}_{0,i,j}$ with the first valid-output clock cycle for $\hat{\mathbf{c}}_{2,j}$. Recall that $\hat{\mathbf{c}}_{2,j}$ is output from NTT in n/τ cycles. When

the HBM starts sending valid data in response to read request, $\hat{\mathbf{c}}_{2,j}$ is subsequently read from the FIFO and used in the multiplication stage. The same synchronization mechanism is used with an independent FIFO for the linear term, $\mathbf{rk}_{1,i,j}$ (Line 7). In addition, this mechanism is reused to synchronize $\mathsf{INTT}(\hat{\mathbf{a}}_0)$ and $\mathsf{INTT}(\hat{\mathbf{a}}_1)$ with $\mathbf{c}_{0,i}$ and $\mathbf{c}_{1,i}$, respectively, during the Modulus-Down operation and the final accumulation stage. Specifically, the FIFO previously used for $\mathbf{rk}_{0,i,j}$ is repurposed for this synchronization.

4.2 Main Operation Flow

We separate the steps of Algorithm 1 into two parts. The first part consists of Lines 4–9, where the input polynomials $\hat{\mathbf{c}}_{2,j}$ are continuously read from the HBM and sent to the NTT unit. Similarly, the relinearization keys $\mathbf{rk}_{:,i,j}$ are continuously read from the HBM and forwarded to the HP units, synchronized with the NTT outputs as described in the previous section. The ACC units then continuously accumulate the valid inputs. The second part consists of Lines 10–19. Based on a 1-bit counter, the accumulated polynomials $\hat{\mathbf{a}}_0$ and $\hat{\mathbf{a}}_1$ are sent to the NTT unit to perform inverse NTT. The outputs of the inverse NTT are then sequentially sent to the MD Unit, which finalizes the computation and produces the result.

We note that the twiddle factors for forward NTT in $R_{q_i,n}$ are fetched prior to the loading of $\hat{\mathbf{c}}_{2,j}$, while the twiddle factors for inverse NTT are fetched afterwards. Furthermore, the first part reads data from HBM and produces intermediate results in on-chip RAMs (such as BRAMs or URAMs, within the ACC units), whereas the second part reads from both on-chip RAMs and HBM and writes the final results back to HBM.

4.3 Hadamard Product (HP) Unit

As previously described, the Hadamard Product (HP) units perform coefficient-wise modular multiplications between $\hat{\mathbf{c}}_{2,j}$ and the relinearization keys $\mathbf{rk}_{0,i,j}$ and $\mathbf{rk}_{1,i,j}$ (see Lines 6–7 of Algorithm 1). Each HP unit consists of an array of parallel modular multipliers, identical to those employed in the BUs of the proposed NTT design. Our architecture incorporates two HP units, each equipped with τ modular multipliers, resulting in a total of 2τ modular multipliers. This matches the modular multiplier utilization across the four layers of the NTT.

4.4 Accumulation (ACC) Unit

The Accumulation (ACC) units are responsible for combining partial products across the modulus q_i (see Lines 6–7 of Algorithm 1). Each unit is implemented using an array of parallel modular adders that perform accumulation over $\tilde{q}_i$. These modular adders are identical to those employed in the BUs of the proposed NTT design. Our architecture employs two ACC units to process the constant and linear terms in parallel. Each ACC unit contains τ modular adders and τ independent BRAMs, each with a depth of $2n/\tau$ entries, for storing intermediate accumulation results. The factor of 2 arises from a ping-pong memory organization, allowing one half of the memory to be written while the other half is simultaneously read. The row address is determined by an internal counter during both read and write operations, as the input and output data are organized sequentially. The use of parallel BRAMs enables simultaneous read and write operations at the target throughput of τ.

4.5 Modulus-Down (MD) Unit

The Modulus-Down (MD) Unit is responsible for Lines 12–19 of Algorithm 1, performing the modular arithmetic for modulus switching as well as the final addition. It contains τ

modular multiplication units, 2τ modular subtraction units, and τ modular addition units. The MD unit also stores intermediate results $\mathbf{t}_0$ and $\mathbf{t}_1$, which are computed over the auxiliary modulus q_L. These are held in a set of BRAMs. As in the ACC units, we employ τ independent BRAMs, each with a depth of $2n/\tau$ entries, to meet the throughput constraint. The factor of 2 arises because each BRAM stores the two intermediate polynomials $\mathbf{t}_0$ and $\mathbf{t}_1$. Since these polynomials are processed sequentially rather than in parallel, increasing the BRAM depth is sufficient without requiring additional memory banks.

Additionally, the modular multipliers in the MD unit could be shared with one of the HP units, potentially reducing the overall number of modular multipliers by τ. We leave this optimization for future work.

5 Evaluation

In this section, we first evaluate the performance of TP-NTT in a standalone setting, and then assess the proposed relinearization architecture built upon it. The TP-NTT hardware and the overall relinearization architecture are implemented using Verilog/SystemVerilog HDL. The design-time parameters n, $\log q$, τ, n_i, and L are implemented as Verilog parameters. The project is fully open-source and made available at https://github.com/cisec-su/TP-NTT and https://github.com/cisec-su/Relin-FPGA.

The evaluated configurations span ring sizes from $n = 2^{10}$ to $n = 2^{16}$ and coefficient moduli with $\log q \in \{32, 60\}$. All hardware designs are synthesized and implemented using the AMD-Xilinx Vivado Design Suite 2023.2 [2] and the Vitis 2023.2 toolchain. The primary target platform is the AMD-Xilinx Alveo U280 FPGA accelerator card (XCU280) [3], which provides HBM along with abundant DSP and BRAM resources suitable for high-throughput cryptographic workloads. The device includes 1303K LUTs, 2607K FFs, 9024 DSPs (DSP48E2s), 2016 BRAMs (BRAM36E2s[4]), and 960 URAMs. HBM provides an aggregate bandwidth of 8192 bits operating at 450 MHz. The software implementation was tested on Ubuntu 22.04.5 with Intel(R) Xeon(R) w3-2423 CPU. All reported FPGA results are post-place-and-route and meet timing closure at the reported operating frequencies. The reported BRAM and URAM usage reflects the implementation results produced by the design tools rather than a manually constrained memory allocation.

5.1 NTT Evaluation

We compare our proposed TP-NTT architecture with prior FPGA-based implementations across multiple parameter sets in terms of average latency and area–time–product (ATP), as summarized in Table 4. Batch NTT refers to processing m independent NTT instances as a group by launching them without waiting for the completion of preceding NTTs. In our architecture, this is realized by consecutively streaming NTTs that share the same modulus (q) and twiddle factors into the accelerator, allowing them to be processed in a fully pipelined fashion. To demonstrate the effectiveness of the proposed architecture, we set $m = 100$, i.e., 100 consecutive NTTs are issued before waiting for the batch to complete. Average latency is computed as $T_{\text{avg}} = (t_{\text{end}} - t_{\text{start}})/m$, where t_{start} and t_{end} correspond to the timestamps immediately before sending the input coefficients of the first NTT and after receiving the all output coefficients of the m-th NTT computation, respectively. ATP, a commonly used metric in the literature [KKT+25], is defined as $\text{ATP} = \text{Latency } (\mu s) \times \left(\text{LUT} + \frac{\text{FF}}{2} + 100 \times \text{DSP} + 300 \times \text{BRAM}\right)$. We only consider works targeting large-scale FPGAs capable of high performance, rather than resource-constrained devices, as such architectures are fundamentally limited by the significantly

[2] https://www.xilinx.com/products/design-tools/vivado.html
[3] https://www.xilinx.com/products/boards-and-kits/alveo/u280.html
[4] https://docs.amd.com/r/en-US/ug953-vivado-7series-libraries/RAMB36E2

Table 4: NTT performance comparison with literature

Work	Platform	$\log q$	n	τ	LUT/FF/DSP/RAM[§]	Freq. (MHz)	Avg. Lat. (μs)	ATP ($\cdot 10^{-3}$)
[YKKP22][†]	Virtex-7	28	2^{10}	32	95 / 104 / 640 / 80	210	1.12 (**8.00x**)	0.263 (**8.57x**)
[KKT+25][†]	XCU280	32	2^{10}	16	76 / 94 / 864 / 24	250	0.26 (**1.85x**)	0.056 (**1.83x**)
TP-NTT[†]	XCU280	32	2^{10}	32	94 / 90 / 800 / 0	250	0.14 (**1.00x**)	0.030 (**1.00x**)
[KKT+25][†]	XCU280	64	2^{11}	16	185 / 204 / 2640 / 40	250	0.52 (**2.00x**)	0.293 (**2.05x**)
TP-NTT[†]	XCU280	60	2^{11}	32	233 / 212 / 1936 / 0	250	0.26 (**1.00x**)	0.143 (**1.00x**)
[MSL25]	XCU280	30	2^{12}	-	232.7 / 111.6 / 2112 / 32	175	0.90 (**5.52x**)	0.459 (**3.17x**)
[KKT+25][†]	XCU280	32	2^{12}	32	137 / 160 / 1632 / 40	250	0.52 (**3.19x**)	0.204 (**1.41x**)
[ZYT+26][†]	XCU280	32	2^{12}	-	652 / 814 / 5760 / 228	245	0.16 (**1.00x**)	0.278 (**1.92x**)
TP-NTT[†]	XCU280	32	2^{12}	64	242 / 212 / 1920 / 0	250	0.27 (**1.64x**)	0.145 (**1.00x**)
TP-NTT[†]	XCU280	32	2^{12}	128	414 / 503 / 3840 / 0	192	0.18 (**1.12x**)	0.191 (**1.32x**)
[KDNL23]	V.Ultrascale+	60	2^{12}	-	74 / 61 / 288 / 155	250	3.80 (**11.35x**)	0.686 (**2.24x**)
[GZH+24][†]	XCU280	64	2^{12}	-	523 / 1478 / 6518 / 34	300	1.17 (**3.49x**)	2.25 (**7.35x**)
[GZX26]	XCZU15EG	64	2^{12}	-	119 / 110 / 1536 / 174	300	1.54 (**4.58x**)	0.583 (**1.90x**)
[ZYT+26]	XCU280	64	2^{12}	-	527 / 909 / 4640 / 124	292	0.50 (**1.49x**)	0.741 (**2.42x**)
[KKT+25][†]	XCU280	64	2^{12}	32	356 / 375 / 5040 / 72	250	0.52 (**1.56x**)	0.556 (**1.81x**)
TP-NTT[†]	XCU280	60	2^{12}	32	250 / 226 / 2112 / 32	250	0.52 (**1.56x**)	0.305 (**1.00x**)
TP-NTT[†]	XCU280	60	2^{12}	64	532 / 434 / 4224 / 0	200	0.33 (**1.00x**)	0.390 (**1.28x**)
[AKK+23]	XCU280	32	2^{13}	-	29 / 21 / 224 / 64	182	9.29 (**25.24x**)	0.753 (**2.42x**)
[KHF25]	XCU200	36	2^{13}	-	699 / 504 / 5888 / 4096	231	1.80 (**4.89x**)	4.983 (**16.02x**)
[KKT+25][†]	XCU280	32	2^{13}	16	89 / 107 / 1008 / 32	250	2.07 (**5.62x**)	0.524 (**1.68x**)
TP-NTT[†]	XCU280	32	2^{13}	64	257 / 238 / 2080 / 32	250	0.52 (**1.42x**)	0.311 (**1.00x**)
TP-NTT[†]	XCU280	32	2^{13}	128	521 / 447 / 4160 / 0	182	0.37 (**1.00x**)	0.427 (**1.37x**)
[YKKP22]	XCU200	52	2^{13}	-	1107 / 1002 / 1920 / 1600	220	4.50 (**6.87x**)	10.26 (**15.19x**)
TP-NTT[†]	XCU280	60	2^{13}	32	287 / 250 / 2288 / 37	250	1.03 (**1.58x**)	0.675 (**1.00x**)
TP-NTT[†]	XCU280	60	2^{13}	64	576 / 488 / 4576 / 64	200	0.65 (**1.00x**)	0.849 (**1.25x**)
[AKK+23]	XCU280	32	2^{14}	-	29 / 21 / 224 / 96	182	19.9 (**26.1x**)	1.801 (**2.64x**)
[ZYT+26][†]	XCU280	32	2^{14}	-	430 / 579 / 3264 / 128	280	1.26 (**1.66x**)	1.367 (**2.00x**)
[KKT+25][†]	XCU280	32	2^{14}	16	94 / 112 / 1056 / 48	250	4.14 (**5.45x**)	1.118 (**1.64x**)
TP-NTT[†]	XCU280	32	2^{14}	64	294 / 260 / 2240 / 34.5	250	1.04 (**1.37x**)	0.682 (**1.00x**)
TP-NTT[†]	XCU280	32	2^{14}	128	542 / 484 / 4480 / 64	172	0.76 (**1.00x**)	0.953 (**1.40x**)
[KDNL23]	V.Ultrascale+	64	2^{14}	-	74.5 / 61.4 / 288 / 155	250	17.3 (**13.3x**)	3.133 (**2.14x**)
[ZYT+26][†]	XCU280	64	2^{14}	-	650 / 844 / 5440 / 188	298	2.29 (**1.76x**)	3.827 (**2.61x**)
[KKT+25][†]	XCU280	64	2^{14}	16	234.1 / 255.7 / 3280 / 96	250	4.14 (**3.18x**)	2.979 (**2.03x**)
TP-NTT[†]	XCU280	60	2^{14}	32	311 / 268 / 2464 / 69	250	2.06 (**1.58x**)	1.467 (**1.00x**)
TP-NTT	XCU280	60	2^{14}	64	526 / 632 / 4928 / 69	200	1.30 (**1.00x**)	1.755 (**1.20x**)
TP-NTT[†]	XCU280	32	2^{15}	64	303 / 278 / 2400 / 66.5	250	4.14 (**1.32x**)	2.909 (**1.00x**)
TP-NTT[†]	XCU280	32	2^{15}	128	601 / 548 / 4800 / 66.5	164	3.14 (**1.00x**)	4.318 (**1.48x**)
[KDNL23]	V.Ultrascale+	64	2^{15}	-	74.5 / 61.4 / 288 / 155	250	33.7 (**11.2x**)	6.090 (**1.81x**)
[KKT+25][†]	XCU280	64	2^{15}	-	444.6 / 459.6 / 6160 / 176	250	4.14 (**1.38x**)	5.566 (**1.65x**)
TP-NTT[†]	XCU280	60	2^{15}	32	326 / 286 / 2640 / 268	250	4.14 (**1.38x**)	3.371 (**1.00x**)
TP-NTT[†]	XCU280	60	2^{15}	64	561 / 682 / 5280 / 133	172	2.99 (**1.00x**)	4.401 (**1.31x**)
[ZYT+26][†]	XCU280	32	2^{16}	-	476 / 638 / 3648 / 252	285	3.59 (**1.10x**)	4.439 (**1.40x**)
[KKT+25][†]	XCU280	32	2^{16}	32	169.5 / 191.5 / 2016 / 152	250	8.20 (**2.52x**)	4.202 (**1.32x**)
TP-NTT[†]	XCU280	32	2^{16}	64	324 / 296 / 2560 / 133	250	4.14 (**1.27x**)	3.182 (**1.00x**)
TP-NTT[†]	XCU280	32	2^{16}	128	633 / 582 / 5120 / 130.5	158	3.26 (**1.00x**)	4.808 (**1.51x**)
[KDNL23]	V.Ultrascale+	64	2^{16}	-	74.5 / 61.4 / 288 / 155	250	66.5 (**10.6x**)	12.00 (**1.26x**)
[WG23]	XCU280	64	2^{16}	-	267.1 / 328.4 / 2736 / 2126	165	380 (**60.5x**)	510.2 (**53.5x**)
[MSDL24]	XCU250	60	2^{16}	-	274.4 / 83.1 / 630 / 241	250	66.2 (**10.54x**)	29.87 (**3.13x**)
[GZX26]	XCZU15EG	64	2^{16}	-	82.1 / 76.2 / 678 / 356.5	300	54.9 (**8.74x**)	16.02 (**1.70x**)
[KHF25]	XCU200	64	2^{16}	-	206 / 160 / 2048 / 718	249	39.14 (**6.23x**)	27.641 (**2.90x**)
[ZYT+26][†]	XCU280	64	2^{16}	-	747 / 929 / 6080 / 1648	280	8.48 (**1.35x**)	19.62 (**2.06x**)
[KKT+25][†]	XCU280	64	2^{16}	32	460 / 470 / 6320 / 280	248	8.35 (**1.33x**)	11.777 (**1.24x**)
TP-NTT[†]	XCU280	60	2^{16}	32	361 / 340 / 2816 / 1128	250	8.28 (**1.32x**)	9.530 (**1.00x**)
TP-NTT[†*]	XCU280	60	2^{16}	32	359 / 341 / 2816 / 1200	250	8.28 (**1.00x**)	9.696 (**1.01x**)
TP-NTT[†]	XCU280	60	2^{16}	64	595 / 759 / 5632 / 522	164	6.28 (**1.00x**)	10.641 (**1.12x**)

LUTs and FFs are in thousands. [§] BRAM equivalent calculated as BRAM+8·URAM. [†] fully pipelined.

[*] is implemented as 2^5–2^3–2^3–2^5

lower availability of FPGA resources. To enable a fair comparison with such works, either our design would need to operate at a very low τ, which conflicts with our design goals, or those designs would need to be scaled to the τ values considered in this work. The latter is non-trivial and would require substantial engineering effort, and is therefore beyond the scope of this paper.

The decomposition of n with respect to the given τ is chosen based on Table 2. Accordingly, in the presented results, TP-NTT uses a 2-D decomposition for $n = 2^{10}$, a 3-D decomposition for $n = 2^{11}$ to 2^{15}, and a 4-D decomposition for $n = 2^{16}$. An exception is that, for $n = 2^{16}$ and $\tau = 32$, we use $n_1 = n_4 = 2^5$, $n_2 = 2^1$, and $n_3 = 2^5$, as this configuration is more engineering-friendly, although theoretically slightly less efficient. The alternative optimal decomposition for $n = 2^{16}$, with $n_1 = n_4 = 2^5$ and $n_2 = n_3 = 2^3$, could also be adopted. However, it is expected to achieve nearly identical resource utilization, operating frequency, and latency, as the DSP utilization remains unchanged and the minor differences in on-chip memory are effectively eliminated by Vivado's BRAM/URAM optimizations. This observation is confirmed by the implementation results in Table 4.

Recall that [KKT+25] presents a fully pipelined architecture similar to our design. Compared to this work, TP-NTT achieves 1.24× to 1.85× better ATP results, due to reduced area requirements such as the elimination of additional multiplication steps inherent to the four-step approach (see Subsection 3.6) and more efficient modular arithmetic. Moreover, the reduction in area allows increased parallelism through larger τ, further reducing average latency. For instance, for $n = 2^{12}$, the more efficient TP-NTT design allows τ to reach 128 and 64 for $\log q = 32$ and $\log q = 60$, respectively, resulting in 2.84× and 1.56× improvements in average latency. The same trend is observed when comparing with [ZYT+26], where TP-NTT achieves 2.06× better ATP and 1.35× lower average latency at $n = 2^{16}$ and $\log q = 64$. In cases with ($\tau = 128$, $\log q = 32$) and ($\tau = 64$, $\log q = 60$), the design could not be synthesized at 250 MHz as it could for lower values of τ, slightly degrading the ATP of our design. The critical path lies in the rotation circuits of the AUs. We believe there is room for further optimization of these critical paths, which is left as future work. When τ and the clock frequency are the same, both designs achieve the same speed, as they are fully pipelined.

[GZH+24] and [YKKP22] also provide fully pipelined architectures similar to our design. However, their implementations are limited to $n = 2^{12}$ and $n = 2^{13}$, respectively. This highlights the unique capability of TP-NTT in scaling pipelined architectures to large ring degrees. Moreover, TP-NTT achieves 3.49× and 6.87× better average latency and 7.35× and 15.19× better ATP results compared to these works, respectively.

Designs that are not pipelined fail to provide competitive average latency. For instance, compared to [GZX26], TP-NTT achieves 4.58× and 8.74× lower average latency for $n = 2^{12}$ and $n = 2^{16}$, respectively, at $\tau = 64$ and $\log q = 60$. At $\tau = 32$, TP-NTT further provides 1.7× to 1.9× improvements in ATP (and up to 1.48× for $\tau = 64$). Similarly, compared to [KDNL23], TP-NTT achieves 10.6× to 13.3× lower latency at $\tau = 64$, along with 1.26× to 2.24× ATP improvements at $\tau = 32$ (and up to 1.75× for $\tau = 64$). When compared to [KHF25], TP-NTT achieves 4.89× to 6.23× lower average latency and 2.90× to 16.02× improvements in ATP. On the other hand, [WG23] employs a design similar to ours in supporting decomposition across multiple dimensions. Nevertheless, TP-NTT achieves 60.6× lower latency and 53.5× lower ATP.

[YKKP22] employs the same τ as our implementation for $n = 2^{10}$ and $\log q = 32$. Under this matched configuration, their design requires more RAM resources and is approximately 8.00× slower than our implementation. Similarly, [KKT+25] uses the same τ for $n = 2^{12}$ and $\log q = 64$; however, it consumes more than 2× the DSP and RAM resources while providing comparable performance. Likewise, for $n = 2^{12}$ and $\log q = 60$, their implementation requires more than 2× the DSP resources of our design, yet achieves similar performance. For all implementations evaluated under the same

throughput parameter τ, our design achieves superior efficiency in terms of both ATP and latency.

Overall, the results demonstrate that our architectures achieve significantly lower latency while maintaining superior ATP across a wide range of parameter sets. This highlights the effectiveness of our design in balancing parallelism, memory bandwidth, and hardware resource utilization, particularly for large-scale NTT computations.

To provide additional insight into our implementation, we consider the largest evaluated TP-NTT configuration ($n = 2^{16}$, $\log q = 60$, and $\tau = 64$) as a representative example. This implementation utilizes 5632 DSPs (62.4% of the Alveo U280) but only 522 RAM-equivalent blocks (5.4% of the available on-chip memory). Hence, DSPs are the dominant resource bottleneck, and the achievable throughput is primarily limited by DSP availability despite TP-NTT's DSP-efficient design that trades modular multipliers for memory (see Subsection 3.6).

5.2 Relinearization Evaluation

Next, we evaluate the relinearization architecture presented in Section 4 that leverages the fast batch NTTs performed by TP-NTT. For evaluation, we integrated the proposed relinearization architecture as a Vitis kernel targeting the Alveo U280 platform, with host-to-device communication handled via XRT. For different ring dimensions, we adopt corresponding $\tilde{L}$ values following standard parameter selection in homomorphic encryption schemes for 128-bit security level, where $\tilde{L} = L + 1$. Specifically, the proposed scheme utilizes two primary parameter configurations. For $\log q = 32$, the parameters are set to $n \in \{2^{13}, 2^{14}, 2^{15}, 2^{16}\}$ with corresponding $\tilde{L} \in \{10, 16, 30, 60\}$. In the case where $\log q = 60$, the scheme employs $n \in \{2^{12}, 2^{13}, 2^{14}, 2^{15}, 2^{16}\}$ with $\tilde{L} \in \{3, 5, 8, 15, 30\}$. These values of $\tilde{L}$ represent practical NTT batch sizes in FHE applications. Table 5 presents the FPGA resource utilization of the relinearization design. Notice that, for all n, the additional modules beyond TP-NTT nearly double the total area cost of TP-NTT for the given n and τ, highlighting the importance of optimizing the NTT unit to reduce overall design congestion. We note that the increased critical path introduced by the AUs for large values of τ does not limit the overall relinearization design. For such large configurations, for example when $\tau = 2^7$ and $n = 2^{16}$, the target FPGA lacks sufficient resources to implement the design, making the critical path bottleneck a moot point. As a reference, we provide the detailed breakdown of the FPGA resource utilization in Table 6 for $n = 2^{16}$ and $\log q = 60$. Note that, as the NTT unit, we employ the extended TP-NTT architecture presented in Subsection 3.5 to reduce engineering effort during testing. However, this is not strictly necessary, as long as the normal-domain polynomials (before the forward NTT) can be maintained in bit-reverse order throughout execution. The total utilization of the two instances of both the ACC and HP is reported, as required in our design. The HBM Adapter corresponds to the resources required for HBM communication, including FIFO buffers for transferring data between clock domains and control circuits. The *Others* category refers to all additional components that Vitis includes in its platform module to enable communication with the host via XRT.

The purpose of this section is to evaluate the practical impact of the proposed accelerator within an end-to-end BFV relinearization workflow. While the previous subsection presented a hardware-to-hardware comparison of the underlying NTT architecture, the evaluation here focuses on the overall performance improvement achieved when integrating the accelerator into a complete software stack. Therefore, we compare against `OpenFHE`, a widely adopted state-of-the-art FHE software library, to demonstrate the acceleration obtained for the relinearization operation under representative parameter sets.

We evaluate the proposed design against the state-of-the-art FHE library, `OpenFHE`, under varying parameter configurations. For 128-bit security level, the maximum supported values of $\log Q$ are approximately 109, 218, 438, 881, 1730 bits for $n = 2^{12}$, 2^{13}, 2^{14}, 2^{15},

Table 5: FPGA resource utilization for relinearization

d	$\log q$	n	τ	LUT (K)	FF (K)	DSP	BRAM	URAM	Freq. (MHz)
3	32	2^{13}	64	625	712	3044	901.5	0	165
3	32	2^{14}	64	624	732	3204	1020	0	200
3	32	2^{15}	64	655	753	3367	1173.5	0	200
3	32	2^{16}	64	673	773	3527	1314.5	87	200
3	60	2^{12}	32	626	701	3172	893.5	0	200
3	60	2^{13}	32	638	732	3348	1014.5	0	200
3	60	2^{14}	32	641	741	3524	1160	0	200
3	60	2^{15}	32	666	769	3700	1049.5	145	200
4	60	2^{16}	32	698	798	3879	695.5	339	200

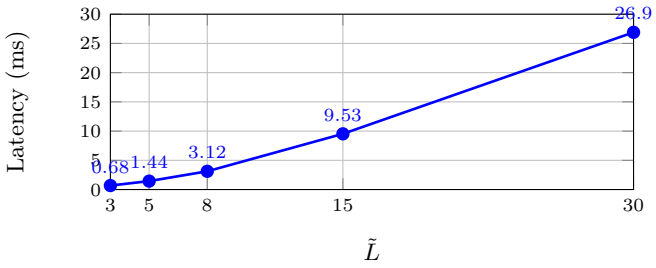


Figure 7: FPGA relinearization latency for different $\tilde{L}$ values ($\log q = 60$, $n = 2^{16}$, $\tau = 32$).

and 2^{16}, respectively. Table 7 presents a comparison between the proposed FPGA-based relinearization accelerator and the `OpenFHE` software implementation across different parameter sets. The results clearly demonstrate the efficiency and scalability of the proposed architecture. For smaller parameter sizes, such as $n = 2^{12}$ and $\log q = 60$ with $\tilde{L} = 3$, the `OpenFHE` implementation requires 0.57 ms, whereas our hardware achieves the same operation in 0.13 ms, corresponding to approximately a 4.4× speedup. Furthermore, for $n = 2^{13}$ and $\log q = 32$ with $\tilde{L} = 10$, our implementation requires only 0.48 ms to compute, whereas `OpenFHE` requires 10.3 ms, demonstrating a speedup of approximately 21.46×. While the performance gain is moderate at this scale due to relatively lower computational intensity, the advantage of hardware acceleration is already evident through reduced latency and efficient parallel operation of NTT, HP, and ACC units.

As the parameters n and $\tilde{L}$ increase, the performance gap becomes more pronounced. Note that the advantage of TP-NTT becomes more significant as the batch size increases, which is $\tilde{L}$ in this application. In particular, for $n = 2^{16}$ and $\tilde{L} = 30$, the `OpenFHE` implementation requires approximately 916 ms, while our FPGA-based design completes the operation in 26.9 ms, yielding a speedup of approximately 34.05×. Moreover, for the parameter set $\log q = 32$ and $\tilde{L} = 60$, our implementation finishes in 48.8 ms, while `OpenFHE` requires 3.28 s to compute. This result corresponds to a speedup of approximately **67.34**×. This substantial improvement highlights the limitations of general-purpose processors in handling large-scale homomorphic encryption workloads, where repeated modular arithmetic operations, memory accesses, and sequential execution lead to significant overhead.

As shown in Figure 7, the runtime of the proposed relinearization accelerator scales approximately quadratically with $\tilde{L}$. This is because the relinearization algorithm requires a number of forward and inverse NTT operations proportional to $\tilde{L}(\tilde{L}+1)$, which constitute

Table 6: Breakdown of FPGA resource utilization for relinearization, $n = 2^{16}$, $\log q = 60$, and $\tau = 32$

Module	**LUT** (K)	**FF** (K)	**DSP**	**BRAM**	**URAM**
Kernel	502	518	3875	496	339
HBM Adapter	2.4	8.6	3	464	0
Relinearization	485	506	3872	32	339
TP-NTT Ext.	386	396	2816	31	204
BitReverse AU	9	6	0	0	32
Twiddle Streamer Unit	16	8	31	31	32
HP x2	44	4	754	0	0
ACC x2	11	11	0	0	54
MD	36	42	352	0	27
Others	196	280	4	199.5	0

the dominant portion of the computation. Consequently, for fixed $\log n$ and $\log q$, the overall latency increases approximately quadratically as $\tilde{L}$ grows.

Table 7: Relinearization time performance comparison

$\log q$	$\log n$	$\tilde{L}$	τ	**OpenFHE (ms)**	**FPGA (ms)**	**Speedup**
32	13	10	64	10.3	0.48	**21.46**×
32	14	16	64	62.07	1.31	**47.38**×
32	15	30	64	428	6.94	**61.67**×
32	16	60	64	3286	48.8	**67.34**×
60	12	3	32	0.57	0.13	**4.38**×
60	13	5	32	2.29	0.26	**8.81**×
60	14	8	32	13.94	0.88	**15.84**×
60	15	15	32	99	4.18	**23.68**×
60	16	30	32	916	26.9	**34.05**×

6 Conclusion

In this work, we presented TP-NTT, a scalable, throughput-optimized NTT architecture. We applied optimizations at both the low level, such as DSP-optimized modular arithmetic, and at the algorithm level, including multi-dimensional decomposition of iterative NTT. These optimizations significantly reduced the hardware complexity of the fully pipelined architecture, enabling higher parallelism. Our highly parametric design supports 2-D, 3-D, and 4-D decompositions configurable at design time. We also showed that the flexibility in the number of dimensions and choice of radices allows further area optimization. We leave a fully parametric decomposition supporting an arbitrary number of dimensions as future work. TP-NTT specifically targets batch NTT operations due to its pipelined design and outperforms prior work from the literature by up to an order of magnitude in average latency, while achieving substantial improvements in the area–time-product.

To assess the effectiveness of TP-NTT in a real-world batch NTT scenario, we developed a case study on the FHE relinearization function. We proposed an efficient hardware architecture for BFV relinearization that leverages the fast batch NTTs performed by TP-NTT. Our relinearization hardware achieved a 67.34× speedup over state-of-the-art

software implementations. Notably, our solution can be extended to other FHE schemes.

7 Acknowledgments

This work is partially supported by the European Union's Horizon Europe research and innovation program under grant agreement No: 101079319, the Horizon 2020 ERC Advanced Grant (101020005 Belfort) and the CyberSecurity Research Flanders with reference number VOEWICS02.

References

[ACM+23] Kaveh Aasaraai, Emanuele Cesena, Rahul Maganti, Nicolas Stalder, Javier Varela, and Kevin Bowers. Cyclone-ntt: An ntt/fft architecture using quasi-streaming of large datasets on ddr- and hbm-based fpga platforms. In *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '23, page 47, New York, NY, USA, 2023. Association for Computing Machinery.

[AKK+23] Can Ayduman, Emre Koçer, Selim Kırbıyık, Ahmet Can Mert, and Erkay Savaş. Efficient design-time flexible hardware architecture for accelerating homomorphic encryption. In *2023 IFIP/IEEE VLSI-SoC*, 2023.

[Bai89] David H. Bailey. Ffts in external or hierarchical memory. In E. Oran Brigham, editor, *Fast Fourier Transform—Algorithms and Applications*. Prentice Hall, 1989.

[Bar86] Paul Barrett. 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*, pages 311–323. Springer, 1986.

[BCC+24] Jean-Philippe Bossuat, Rosario Cammarota, Ilaria Chillotti, Benjamin R. Curtis, Wei Dai, Huijing Gong, Erin Hales, Duhyeong Kim, Bryan Kumara, Changmin Lee, Xianhui Lu, Carsten Maple, Alberto Pedrouzo-Ulloa, Rachel Player, Yuriy Polyakov, Luis Antonio Ruiz Lopez, Yongsoo Song, and Donggeon Yhee. Security guidelines for implementing homomorphic encryption. Cryptology ePrint Archive, Paper 2024/463, 2024.

[BGV12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. Fully homomorphic encryption without bootstrapping. *ITCS*, 2012.

[BKS+21] Fabian Boemer, Sejun Kim, Gelila Seifu, Fillipe D.M. de Souza, and Vinodh Gopal. Intel hexl: Accelerating homomorphic encryption with intel avx512-ifma52. In *Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, WAHC '21, page 57–62, New York, NY, USA, 2021. Association for Computing Machinery.

[Bra12] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 868–886. Springer, Heidelberg, August 2012.

[CCS19] Hao Chen, Ilaria Chillotti, and Yongsoo Song. Improved bootstrapping for approximate homomorphic encryption. In *Advances in Cryptology* –

EUROCRYPT 2019: 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19–23, 2019, Proceedings, Part II, page 34–54, Berlin, Heidelberg, 2019. Springer-Verlag.

[CKKS17] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. Homomorphic encryption for arithmetic of approximate numbers. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part I*, volume 10624 of *LNCS*, pages 409–437. Springer, Heidelberg, December 2017.

[CT65] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

[DMOS22] Kemal Derya, Ahmet Can Mert, Erdinç Öztürk, and Erkay Savaş. Coha-ntt: A configurable hardware accelerator for ntt-based polynomial multiplication. *Microprocess. Microsyst.*, 89(C), March 2022.

[DÖS15] Y. Doröz, Erdinç Öztürk, and B. Sunar. Accelerating fully homomorphic encryption in hardware. In *HOST*, 2015.

[DP09] Florent Dinechin and Bogdan Pasca. Large multipliers with fewer dsp blocks. pages 250 – 255, 10 2009.

[FV12] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012. https://eprint.iacr.org/2012/144.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. *STOC*, 2009.

[GS66] W. M. Gentleman and G. Sande. Fast fourier transforms: for fun and profit. In *Proceedings of the 1966 Fall Joint Computer Conference*, 1966.

[GVBLP+23] Robin Geelen, Michiel Van Beirendonck, Hilder V. L. Pereira, Brian Huffman, Tynan McAuley, Ben Selfridge, Daniel Wagner, Georgios Dimou, Ingrid Verbauwhede, Frederik Vercauteren, and et al. Basalisc: Programmable hardware accelerator for bgv fully homomorphic encryption. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, page 32–57, Aug 2023.

[GZH+24] Zhenyu Guan, Yongqing Zhu, Yicheng Huang, Luchang Lei, Xueyan Wang, Hongyang Jia, Yi Chen, Bo Zhang, Jin Dong, and Song Bian. Esc-ntt: An elastic, seamless and compact architecture for multi-parameter ntt acceleration. In *DATE 2024*, 2024.

[GZX26] Jianbo Guo, Jiaoyang Zhu, and Hao Xiao. A high-performance and configurable ntt accelerator based on scalable 2-d architecture. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 34(2):506–518, 2026.

[HMR24] Florian Hirner, Ahmet Can Mert, and Sujoy Sinha Roy. Proteus: A pipelined ntt architecture generator. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 32(7):1228–1238, 2024.

[KDNL23] Stefanus Kurniawan, Phap Duong-Ngoc, and Hanho Lee. Configurable memory-based ntt architecture for homomorphic encryption. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 2023.

[KHF25] Dilshan Kumarathunga, Qilin Hu, and Zhenman Fang. 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)*, pages 1–9, 2025.

[KKK+22] Sangpyo Kim, Jongmin Kim, Michael Jaemin Kim, Wonkyung Jung, John Kim, Minsoo Rhu, and Jung Ho Ahn. Bts: an accelerator for bootstrappable fully homomorphic encryption. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 711–725, New York, NY, USA, 2022. Association for Computing Machinery.

[KKT+25] Emre Koçer, Selim Kırbıyık, Tolun Tosun, Ersin Alaybeyoğlu, and Erkay Savaş. Io-optimized design-time configurable negacyclic seven-step ntt architecture for fhe applications. In *Proceedings of the Great Lakes Symposium on VLSI 2025*, pages 14–21. Association for Computing Machinery, 2025.

[KLP19] Sungwoo Kim, Yongwoo Lee, and Jongsun Park. A low-latency and scalable number theoretic transform architecture for homomorphic encryption. In *IEEE FCCM*, 2019.

[LN16] Patrick Longa and Michael Naehrig. Speeding up the number theoretic transform for faster ideal lattice-based cryptography. Cryptology ePrint Archive, Paper 2016/504, 2016.

[MKO+22] Ahmet Can Mert, Emre Karabulut, Erdinc Ozturk, Erkay Savas, and Aydin Aysu. An Extensive Study of Flexible Design Methods for the Number Theoretic Transform . *IEEE Transactions on Computers*, 71(11):2829–2843, November 2022.

[Mon85] Peter Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 1985.

[MÖS20] Ahmet Can Mert, Erdinç Öztürk, and Erkay Savaş. Design and implementation of encryption/decryption architectures for bfv homomorphic encryption scheme. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 28(2):353–362, 2020.

[MSDL24] Rella Mareta, Ardianto Satriawan, Phap Ngoc Duong, and Hanho Lee. A bootstrapping-capable configurable ntt architecture for fully homomorphic encryption. *IEEE Access*, 12:52911–52921, 2024.

[MSL25] Rella Mareta, Ardianto Satriawan, and Hanho Lee. High throughput arithmetic computing unit for bfv homomorphic encryption. *IEEE Open Journal of Circuits and Systems*, 6:457–466, 2025.

[RMIT14] Debapriya Basu Roy, Debdeep Mukhopadhyay, Masami Izumi, and Junko Takahashi. Tile before multiplication: An efficient strategy to optimize dsp multiplier for accelerating prime field ecc for nist curves. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2014.

[RVM+14] Sujoy Sinha Roy, Frederik Vercauteren, Nele Mentens, Donald Donglong Chen, and Ingrid Verbauwhede. Compact ring-lwe cryptoprocessor. In Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and Embedded Systems – CHES 2014*, pages 371–391, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[SFK+21] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Srinivas Devadas, Ronald Dreslinski, Christopher Peikert, and Daniel Sanchez. F1: A fast and programmable accelerator for fully homomorphic encryption. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 238–252, New York, NY, USA, 2021. Association for Computing Machinery.

[SFK+22] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Nathan Manohar, Nicholas Genise, Srinivas Devadas, Karim Eldefrawy, Chris Peikert, and Daniel Sanchez. Craterlake: a hardware accelerator for efficient unbounded computation on encrypted data. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 173–187, New York, NY, USA, 2022. Association for Computing Machinery.

[TCW+24] Weihang Tan, Sin-Wei Chiu, Antian Wang, Yingjie Lao, and Keshab K. Parhi. Parentt: Low-latency parallel residue number system and ntt-based long polynomial modular multiplication for homomorphic encryption. *IEEE Transactions on Information Forensics and Security*, 19:1646–1659, 2024.

[TKKA26] Tolun Tosun, Selim Kırbıyık, Emre Koçer, and Ersin Alaybeyoğlu. Optimized fpga architecture for modular reduction in ntt. In Erkay Savaş, Amir Moradi, and Gregor Leander, editors, *Lightweight Cryptography for Security and Privacy*, pages 117–137, Cham, 2026. Springer Nature Switzerland.

[WG23] Cheng Wang and Mingyu Gao. Sam: A scalable accelerator for number theoretic transform using multi-dimensional decomposition. In *2023 IEEE/ACM ICCAD*, 2023.

[WYW20] Shuai Wang, Shouyi Yin, and Shaojun Wei. A high-throughput fpga architecture for number theoretic transform in homomorphic encryption. In *FPL*, 2020.

[YCH22] Zewen Ye, Ray C. C. Cheung, and Kejie Huang. Pipentt: A pipelined number theoretic transform architecture. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 69:4068–4072, 2022.

[YKKP22] Yang Yang, Sanmukh R. Kuppannagari, Rajgopal Kannan, and Viktor K. Prasanna. Nttgen: a framework for generating low latency ntt implementations on fpga. In *19th ACM International Conference on Computing Frontiers*. ACM, 2022.

[ZWZ+21] Ye Zhang, Shuo Wang, Xian Zhang, Jiangbin Dong, Xingzhong Mao, Fan Long, Cong Wang, Dong Zhou, Mingyu Gao, and Guangyu Sun. Pipezk: Accelerating zero-knowledge proof with a pipelined architecture. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 416–428, 2021.

[ZYT+26] Yilan Zhu, Geng Yang, Xingyu Tian, Dilshan Kumarathunga, Liang Kong, Xianglong Deng, Shengyu Fan, Guang Fan, Guiming Shi, Lei Chen, Bo Zhang, Yisong Chang, Shoumeng Yan, Zhenman Fang, and Mingzhe Zhang. An efficient and scalable hardware architecture for number theoretic transform on fpga with design automation. In *2026 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–14, 2026.