AutoNTT: Automatic Architecture Design and Exploration for Number Theoretic Transform Acceleration on FPGAs

Dilshan Kumarathunga[1], Qilin Hu[1,2,*], Zhenman Fang[1]
[1]Simon Fraser University, Burnaby, BC, Canada; [2]Hunan University, Changsha, Hunan, China
{dilshan_kumarathunga, zhenman}@sfu.ca, hql@hnu.edu.cn
[*]Work done while a visiting PhD student at Simon Fraser University

***Abstract*—Fully Homomorphic Encryption (FHE), which enables homomorphic computing on encrypted data, has emerged as a promising privacy-aware computing method. However, FHE is orders-of-magnitude slower than the same computation on plain data, making it far from practical use. One of the major computation bottlenecks in FHE is the Number Theoretic Transform (NTT). While prior studies have accelerated NTT using specific architectures and FHE parameters, there still lacks a design automation tool to systematically design and explore various NTT architectures to support a diverse range of FHE parameters, such as various polynomial sizes, modulo sizes, and reduction methods.**

In this paper, we present AutoNTT, an open-source automatic architecture design and exploration tool to generate highly scalable NTT accelerators on FPGAs. Unlike prior studies, AutoNTT can automatically generate several optimized NTT acceleration architectures in HLS (i.e., iterative, dataflow, and hybrid architectures) with multiple common reduction methods, and support a large range of polynomial sizes ($2^{10} - 2^{17}$) and modulo sizes ($\log q$: $28 - 64$). In our auto-generated NTT architectures, we have applied many optimizations, such as polynomial and twiddle factor buffer reduction, and simplifying interconnections between different butterfly unit groups. Compared to prior studies, AutoNTT can generate NTT accelerators with 2.48× better latency and 3.61× better throughput on average, while maintaining a similar FPGA resource utilization. AutoNTT will be released soon at https://github.com/SFU-HiAccel/AutoNTT.

I. Introduction

Along with rapid developments in the fields of machine learning and quantum computing, the requirement for efficient and reliable cryptosystems to enable post-quantum privacy-aware computing has gained utter importance [1]–[3]. Consequently, significant research has focused on developing practical and efficient deployment methods for FHE [4] and Post-Quantum Cryptography (PQC) [5] schemes. Data encrypted by these lattice-based schemes are in the form of polynomials, and polynomial multiplication is one of the most expensive operations when computing with encrypted data. To reduce the polynomial multiplication time complexity from $O(N^2)$ to $O(N \log N)$, where N denotes the polynomial size, NTT has been widely used in both FHE and PQC schemes. As a key component of FHE, NTT occupies about 54% of the total computation time in the widely used Microsoft SEAL FHE library [6], [7], which is critical to accelerate. However, it presents multiple challenges to accelerate NTT on hardware.

First of all, FHE schemes span across a diverse range of parameters, including different polynomial sizes ranging from $N = 2^{10} - 2^{17}$, different prime modulo sizes ($\log q = 28 - 64$ bits), and different number of primes (also known as limbs) when the residue number system is used. These parameters directly affect the performance, accuracy, and security of the cryptosystem [8], [9]. Unfortunately, prior NTT and FHE accelerators [7]–[15] only optimize their FPGA designs for a limited set of (sometimes relatively small) FHE parameters. It is nontrivial to adapt their designs to support the diverse FHE parameters (especially scaling to larger designs) while providing optimal performance, not to mention that most of those studies did not open source their designs.

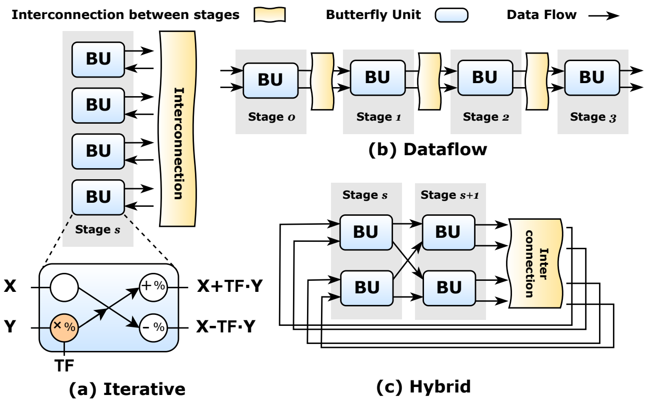

Fig. 1: Different NTT architectures with their butterfly unit (BU) configurations: (a) Iterative, (b) Dataflow, (c) Hybrid

Given the polynomial-size N, NTT contains $\log N$ stages with varying interconnection patterns. In addition to FHE parameters, there are various accelerator architectures and design parameters for NTT acceleration on FPGAs [12], [16] to support these different stages. Fig. 1 illustrates three NTT design architectures, including (a) the iterative architecture that supports only one stage at a time, (b) the dataflow architecture that supports all the stages in a pipeline, and (c) the hybrid architecture that combines the iterative (vertically) and dataflow (horizontally) architectures. In all of these architectures, the basic component is the butterfly unit (BU), which takes two polynomial values (X and Y) and one twiddle factor (TF) value as inputs and produces two outputs ($X + TF * Y$, and $X - TF * Y$). Note that these operations are modular multiplication, modular addition, and modular subtraction.

On a datacenter FPGA, scaling up NTT designs to improve latency and throughput poses significant challenges, including: 1) designing the optimal NTT architecture and configuration to utilize the maximum available resources, 2) supporting parallel memory access required by polynomials and TFs while minimizing on-chip memory utilization, 3) supporting complex interconnection patterns for both NTT and inverse NTT operations in the same hardware architecture, and 4) achieving a good frequency for large FPGA designs. It is es-

TABLE I: Comparison with state-of-the-art NTT automation solutions

Automation Solution	Crypt. Scheme	Supported Architecutre			Supported Reduction Algo	Poly Size (N)	log(q)	Dependency on #Limbs	Max Vector Size	DSE	Open Source
		I	D	H							
Proteus [17]	FHE/PQC/ZKP	N	Y	N	WLM	$2^{10} - 2^{16}$	$28 - 256$	N	2	Y	Y
OpenNTT [18]	FHE	Y	N	N	WLM	$2^{10} - 2^{16}$	$24 - 60$	N	64	N	Y
Mu et al. [12]	PQC	Y	Y	Y	Montgomery	$2^7 - 2^{12}$	$13 - 24$	N/A	64	N	N
NTTGen [16]	HE	Y	Y	Y	Barrett/ Prime Specific	$2^{10} - 2^{14}$	$28 - 52$	Y	32	Y	N
AutoNTT (Ours)	FHE/PQC	Y	Y	Y	Barrett/ Montgomery/WLM/ Custom algorithms	$2^{10} - 2^{17}$	$28 - 64$	N	512	Y	Y

I = Iterative, **D** = Dataflow, **H** = Hybrid, WLM = Word Level Montgomery, Y = Yes, N = No, N/A = Not Available.

sential to develop an automation tool to automatically generate highly scalable NTT designs and explore the best design for the given FHE parameters.

While there are some RTL-based automation solutions [12], [16]–[18], they suffer from one or more of the following limitations: 1) only support limited NTT architectures and reduction methods, 2) only support limited FHE parameters, 3) do not provide automatic design space exploration, and 4) are not open source. A detailed comparison is provided in Table I.

In light of this, we present AutoNTT, an open-source automation framework, which can automatically generate highly scalable NTT architectures in HLS (high-level synthesis) for a diverse set of FHE parameters. In summary, AutoNTT features the following novel contributions:

1. **Support for wider ranges of FHE parameters:** Compared to prior studies, AutoNTT supports wider ranges of FHE parameters, including a large range of polynomial sizes ($N = 2^{10} - 2^{17}$) and modulo sizes ($\log q : 28 - 64$ bits).
2. **Support for diverse NTT architectures:** AutoNTT supports highly optimized iterative, dataflow, and hybrid architectures, with common reduction methods (Barrett [19], Montgomery [20], word level Montgomery [21], and user-customized reductions). Moreover, AutoNTT optimizes NTT interconnection patterns, as well as polynomial and twiddle factor buffer usage.
3. **Automatic design and exploration:** For a user given FHE parameter set and an FPGA resource budget, AutoNTT automatically explores the design space, identifies the optimal architecture and its configuration, and generates the final FPGA design in HLS.
4. **Competitive performance:** On average, AutoNTT provides 2.48× latency improvement and 3.61× better throughput, compared to state-of-the-art RTL designs, while maintaining a similar resource budget.

II. Background and Motivation

A. NTT Implementation for FHE

When discrete Fourier transform (DFT) is performed over a specific finite field of integers modulo a prime q ($\mathbb{Z}_q$), we denote this operation as NTT. Therefore, algorithms for fast DFT such as Cooley-Tukey [22] and Gentleman-Sande [23] for input size N can be applied to NTT with the time complexity of $O(N \log N)$. When converted to the NTT domain, the multiplication of two polynomials becomes element-wise multiplication. We refer interested readers to some of the previous work [7], [9], [16] for more details about NTT.

The ciphertexts of FHE schemes are N-size polynomials with coefficients in $\mathbb{Z}_Q$, where Q ranges from a few hundred to a few thousand bits [24]. The residue number system (RNS) breaks these coefficients into smaller sizes, representing Q as a product of l primes $q_1, ..., q_l$, each roughly the size of a machine word. Each polynomial for a prime q_i is called a limb. NTT can leverage primes in RNS for parallelism, but in FHE schemes like Cheon-Kim-Kim-Song (CKKS), each multiplication reduces one limb [14], requiring careful exploration of limb parallelism.

All the operations performed in NTT are modulo operations. Even though modulo addition and subtraction can be simplified with comparisons with the modulus [25], modulo multiplication is expensive and consumes many DSPs. To avoid the computing overhead of modular multiplication, Barrett [19] and Montgomery [20] reductions are generally employed to support a generic q. Taking advantage of NTT-friendly primes, derived versions of these algorithms are also used in literature to optimize modulo multiplication [16], [21], [26] in terms of latency and resources.

B. Motivation for AutoNTT

Table II provides some key parameters of recent datacenter FPGA-based FHE accelerators. Vector size denotes the number of polynomial values consumed by the NTT unit each cycle. These accelerators optimize their designs for specific FHE parameters. Supporting diverse FHE parameters in such designs and beyond is challenging as it requires parameter-specific optimizations to maintain the performance and handle higher resource utilization required by larger parameters.

TABLE II: Key NTT parameters in recent FHE accelerators

Design	Poly Size (N)	log(q)	Vector Size	Architecture	Reduction
FAB [14]	2^{16}	54	512	Iterative	WK
Poseidon [25]	2^{16}	32	512	Hybrid	B
HEAP [15]	2^{13}	36	512	Iterative	B

B: Barrett reduction. WK: Will-Ko reduction.

Table III summarizes a qualitative comparison of key attributes of different architectures, and it shows a tradeoff between scalability and performance using different architectures. However, to fully explore the advantages of this design space requires scalable architectures to enable a larger design space and have a systematic way to identify the best solution.

TABLE III: Comparison of different architectures

Attribute	**I**	**D**	**H**
Non power of two BUs	☹	🙂	🙂
Independence of #BUs from poly size (N)	🙂	☹	🙂
Interconnection complexity	🙂	😐	☹
Latency in general	🙂	☹	😐
Throughput in general	😐	🙂	😐

I = Iterative, **D** = Dataflow, **H** = Hybrid

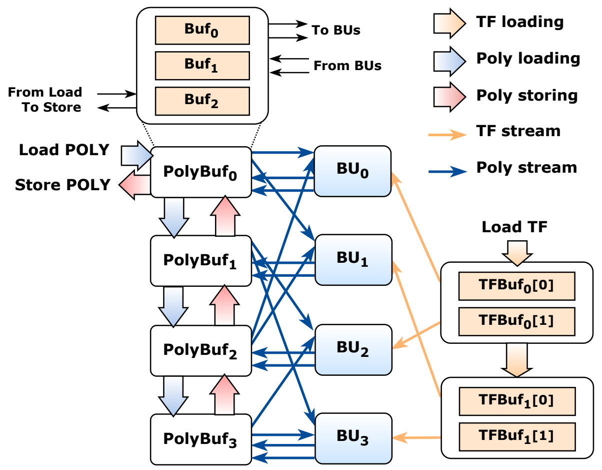

Fig. 2: Iterative NTT architecture

Comparing Tables I and II shows that no previous solutions enable design automation with all the FHE parameters, architecture options, scalability, and support for reduction methods used by recent FHE accelerators. Parameters required for the majority of PQC schemes are smaller than those of FHE [12]. Therefore, AutoNTT provides a single automation solution that can perform design space explorations including algorithmic parameters, architecture options, and design scalability required by real FHE and PQC accelerators.

III. AutoNTT Architecture Designs

Before presenting our automation tool, we first describe our iterative, dataflow, and hybrid architectures and modulo-reduction implementations, as well as our optimizations for each architecture to improve their versatility and scalability.

A. Iterative Architecture

*1) **Overview**:* Fig. 2 illustrates the overview of our iterative architecture. We use tensor-product-based iterative NTT architecture [7] to support a constant polynomial data access pattern during multiple stages of NTT.

Our optimized architecture has three core modules: BUs, PolyBuf, and TFBuf. BUs perform computations required for NTT, while PolyBuf and TFBuf provide the required polynomial data and TF data, respectively. In a design with n_{BU} BUs, each BU reads two input data from the PolyBuf indices $\lfloor BU\_idx/2 \rfloor$ and $\lfloor BU\_idx/2 + n_{BU}/2 \rfloor$, where $BU\_idx \in [0, n_{BU}-1]$. The two outputs of each BU are passed to the PolyBuf index BU_idx. Each BU accesses the TF data from the TFBuf index $(BU\_idx \bmod (n_{BU}/2))$ to optimize the TF storage as described later. All these connections are unique and do not depend on the NTT stage; thus, it simplifies the interconnection between BUs.

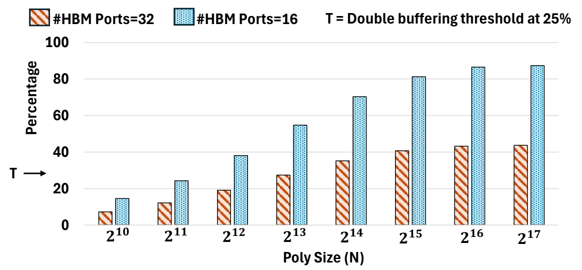

Fig. 3: Off-chip memory access vs. compute time for polynomial sizes at two HBM bandwidths ($n_{BU} = 256$, $\log q = 54$)

*2) **Polynomial Buffer Optimization**:* Storing all the limbs in on-chip memory is not a scalable option, especially when targeting larger parameters. Hence, access to off-chip memory for different limbs is required. However, as shown in Fig. 3, scheduling load, compute, and store sequentially can cause significant overhead in overall computation after a certain polynomial size. To reduce this overhead during the processing of multiple limbs, we support double buffering to hide the off-chip memory access latency after this overhead reaches 25% of the compute time. Double buffering threshold is determined considering performance_gain/resource_increase $\geq 50\%$.

The naïve double buffering would simply include 2 more buffers to load a new limb and store the previous limb's results, requiring 4 total buffers (compute uses 2 buffers). As this limits the scalability, especially for larger parameters, we optimize the design to use a single extra buffer with two ports to support double buffering. During the process, this extra buffer carries the final result of the previous limb computation and uses two ports to load the new limb while sending the result to the off-chip memory. We include synchronization logic for loading and storing data, and for switching between 3 buffers after processing each limb. We connect polyBuf modules as a chain to load data using the same off-chip memory port sequentially. The data store is also done similarly. This helps obtain a highly scalable place-and-route friendly design for a given number of ports by minimizing the number of direct connections to load and store modules.

*3) **TF Buffer Optimization**:* Fig. 4 provides the overall idea of the TF access pattern of this architecture for $N = 2^{10}$ and $n_{BU} = 32$ when performing NTT. In this example, at each NTT stage s, a single BU processes $c = N/(2 * n_{BU}) = 16$ coefficient pairs with 16 TF values. We define a TF group (G_i) with $c = 16$ TFs in each (as shown in the TF index axis). During the first $\log c + 1 = 5$ stages, both BU_3 and BU_{19} only access values in TF group G_0. As the NTT stage s increases, TFs accessed by BUs within the G_0 are doubled. After 5 stages, both BUs switch to G_1. At stage $s = 6$ both BUs move to G_3 and during the last stage only BU_{19} moves to G_{19} while BU_3 remains on G_3. Therefore, two observations can be summarized. 1) Both BU_3 and BU_{19} only access specific TF groups out of all the groups. 2) The TF groups accessed by each BU depend on the BU_{idx} and s (NTT stage).

Based on these observations, the TF index equation for i^{th} TF value access at stage s can be formulated as $TF_{idx} = \{BU\_idx,\ c_i\}\ \&\ mask$, where $0 \leq c_i < c$ and $mask =$

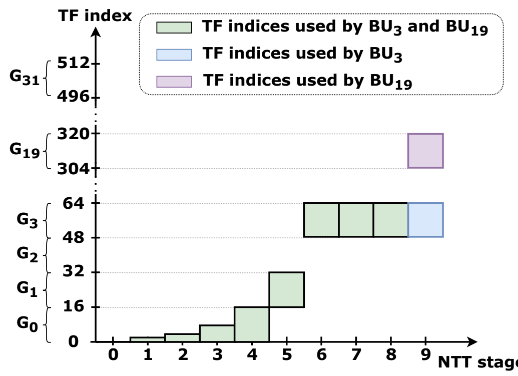

Fig. 4: TF indices used by BU_i and $BU_{(n_{BU}/2+i)}$ in each NTT stage, when $i = 3$, $n_{BU} = 32$, and $N = 2^{10}$

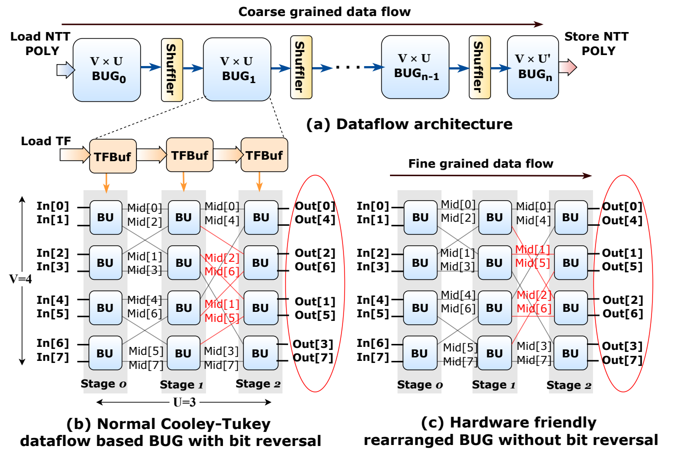

Fig. 5: Dataflow NTT architecture

$(1 << s)-1$. As s increases during the first $(\log c + 1)$ stages, the mask only unveils bits in c_i (only G_0 values). After $(\log c + 1)$ NTT stages, as the mask starts to unveil bits in BU_idx, TF groups (each with c values) start to depend on BU_idx. By storing groups for BU_idx and $(BU\_idx + n_{BU}/2)$ in the same buffer, we can maximize the sharing and support 2 BUs. For the INTT, the mask becomes $\sim ((1 << (\log N - s - 1)) - 1)$, and we can determine the TF groups and values accessed during INTT similarly. It is important to note that the TF group indexes accessed by each BU only depend on the BU_idx and do not depend on the input polynomial size.

TFBuf modules contain the logic to generate TF addresses based on this optimized storage for both NTT and INTT. Each TFBuf module contains 2 buffers to support double buffering.

B. Dataflow Architecture

*1) **Overview**:* Dataflow architecture unrolls all the NTT stages by employing dedicated BUs for each stage and processes the data in a streaming fashion. We use a butterfly unit group (BUG) based streaming architecture [9] for our dataflow architecture as it uses a fixed connection between a couple of NTT stages without complex data rearrangement and buffering. Fig. 5(a) shows our overall dataflow architecture. In the coarse-grained dataflow, multiple BUGs and shuffler modules are connected in a pipeline to support all NTT stages. BUG contains a fine-grained dataflow with $V \times U$ BUs connected in a specific pattern to sample $(2 * V)$ data and

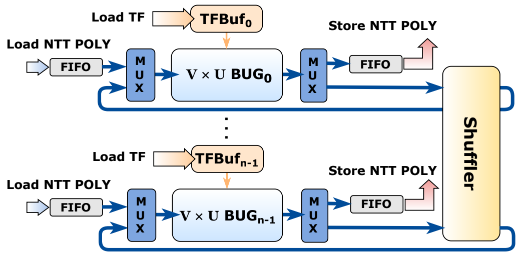

Fig. 6: Hybrid NTT architecture

support $U = \log(2 * V)$ NTT stages inside it. We include a partial BUG as the last BUG if $\log N$ is not perfectly divisible by U. Shuffler modules are used between BUGs to rearrange data for the next BUG. In this architecture, the number of TFs required for each NTT stage s can be identified as 2^s. Similar to the iterative architecture, we analyze the TF access pattern based on the BUG size and store only the required TFs for each stage in their respective TFBuf modules.

*2) **Remove Bit Reversal in BUG**:* Fig. 5(b) illustrates the naïve Cooley-Tukey based BUG, which includes a bit reversed connection. This would cause logic in the shufflers and TFBuf modules to be complex. Hence, we rearrange BUs in each layer as shown in Fig. 5(c) to eliminate bit reversal, thus reducing additional complexities.

C. Hybrid Architecture

*1) **Overview**:* The hybrid architecture has a couple of unrolled NTT stages in a pipeline, and these unrolled stages are iteratively used to complete all the $\log N$ stages of NTT. Fig. 6 illustrates our overall hybrid architecture, which also incorporates BUGs to keep a fixed interconnection across the unrolled stages. We combine n BUGs vertically to support larger input vector sizes $W = (2 * V * n)$. Multiplexers located before and after the BUGs help control polynomial loading and storing, and different rounds of the polynomial processing. Similar to the dataflow architecture, the shuffler is responsible for rearranging polynomial data for the next round. FIFOs at loading and storing data paths help load and store a new limb and the previously processed limb while BUGs are working on the current limb.

*2) **Frequency Optimized Interconnection**:* In the hybrid architecture, increasing input vector size complicates the shuffler logic and lowers the frequency. To fix this, we analyze unique access patterns across rounds to optimize interconnections.

Fig. 7(a) shows the naïve monolithic shuffler design with two $(2 * n * V)^2$ crossbars and $(2 * n * V)$ buffers. The C_0 crossbar reorders incoming data from all the BUGs before storing them in B_0 buffers. Buffers hold data coming during different clock cycles, and once they receive enough data to support the output pattern, they start sending data to the C_1 crossbar. The C_1 crossbar routes the data back to their respective output data paths.

As shown in Fig. 7(b), in round 1, local shufflers receive data from BUGs in a block-cyclic distribution after supporting polynomial pair distances of 1 and 2 in the BUG. Since the data needed for the next round follows a cyclic distribution, inter-BUG communication occurs after the local shufflers. This

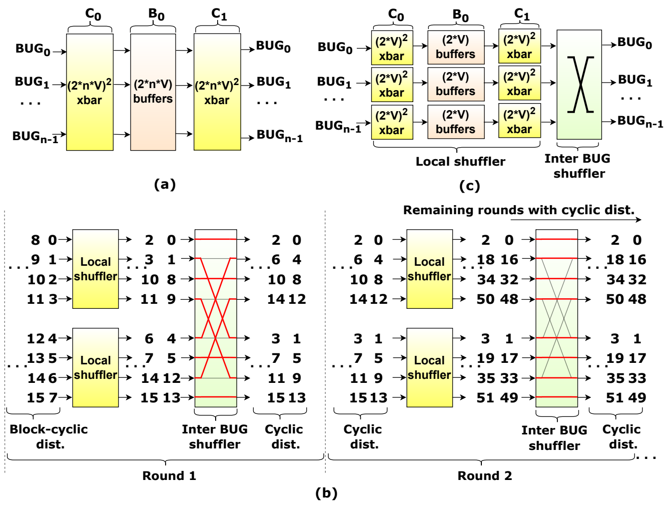

Fig. 7: (a) Monolithic shuffler design with two large crossbars, (b) Inter-BUG and intra-BUG data shuffle patterns in different rounds, (c) Hardware-friendly split shuffler design with smaller local crossbars and simplified inter-BUG shuffler

output will be used to process polynomial pair distances of 4 and 8. During round 2, the shuffler needs to rearrange data to support pairs with distances 16 and 32, as shown in the figure. However, in this round, as the data is already in cyclic distribution, the output data for each BUG is located in the local shuffler. Therefore, inter BUG shuffler just forwards the data. Based on this access pattern, the remaining stages can also be supported only using the local shuffler with cyclic data distribution, and no extra inter-BUG connection is needed.

Based on this access pattern, we design a hardware-friendly split shuffler as shown in Fig. 7(c) by breaking the initial monolithic modules of C_0, B_0, and C_1 into smaller modules to work as local shufflers. An inter-BUG shuffler is added after local shufflers to support fixed inter-BUG interconnection required in round 1 and forwarding connections required in the remaining rounds. Compared to very large two $(2*n*V)^2$ monolithic crossbars, the split shuffler contains fewer connections, helping the routing and frequency. It is also noteworthy to mention that the inter-BUG connection required in the inter-BUG shuffler does not depend on the input polynomial size, but only on the architectural parameter, input vector size.

D. Different Modular Reductions & Multi Word Multiplication

We implement optimized Barrett reduction [27], general Montgomery reduction [20], and customizable word level Montgomery reduction for NTT-friendly primes [21] in our automation framework. During our testing, we noticed that Will-Ko reduction used in [14] consumes more LUTs, thus limiting the scalability; indeed, the same authors confirmed that Barrett reduction is better [15]. Hence, our default framework does not implement Will-Ko reduction. We also provide an interface in our framework for users to integrate custom reduction methods written in a code template. We use arbitrary precision data types [28] available in Vitis HLS to reduce resources. However, we noticed that naïve HLS coding does not efficiently use the DSPs available in FPGA. Hence, we implement multi-word multiplication in HLS by dividing the multiplication into smaller parts to fit into individual DSPs, reducing the DSP usage compared to naïve HLS coding.

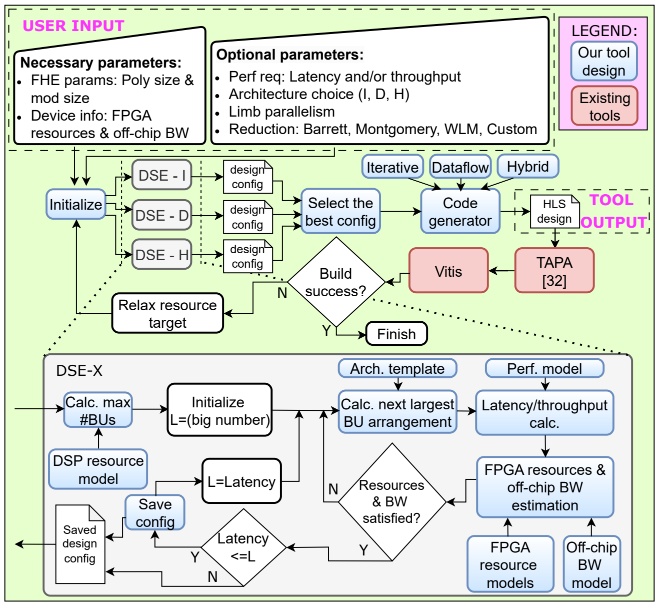

Fig. 8: AutoNTT design automation flow

IV. Design Automation and Exploration

With three highly scalable and optimized NTT architectures and support for common reduction methods, AutoNTT enables systematic design space exploration to find the optimal design choice for user given parameters.

As shown in Fig. 8, AutoNTT takes FHE parameters, FPGA resources, off-chip memory bandwidth, and other optional parameters as inputs. Then it determines the maximum number of BUs based on available DSPs using the DSP resource model. For each architecture, starting from the largest design configuration that fits into the available BU budget, AutoNTT searches for the design with the lowest latency and resources using the latency and resource analytical models. Since each architecture scales differently, the tool selects the optimal configuration for each and then chooses the best overall. Finally, the selected architecture option and configuration are passed to the code generator to generate the HLS code.

We use N to denote polynomial size, P for the total pipeline depth of the hardware modules, and X for the total number of unrolled NTT stages. X is 1 for iterative architecture, $logN$ for dataflow architecture, and U for hybrid architecture. W denotes the input vector size and $Y = \left\lceil \frac{logN}{U} \right\rceil - 1$ denotes the number of shufflers. Our analytical models are as follows.

*1) **Latency** (L):* Latency is the total number of cycles required to process a single polynomial. Similar to prior studies [16], we use 250MHz as the target clock frequency when converting it to time.

$$L = \begin{cases} \left(\frac{N}{W} + P\right) \times \log N & \text{if iterative} \\ \frac{N}{W} + P + \sum_{i=1}^{Y} R_i & \text{if dataflow} \\ \frac{N}{W} + P + \sum_{i=1}^{Y} max\{\frac{N}{W}, P + R_i\} & \text{if hybrid} \end{cases} \quad (1)$$

where R_i represents the delay in the i^{th} shuffler in the dataflow architecture, and i^{th} round in the hybrid architecture.

*2) **DSP Resources (D)**:* Based on the data width, reduction method, and our multi-word multiplication (Section III-D), we calculate how many DSPs are required for modular multiplication. If the number of DSPs required for a BU is d_{BU}, the number of DSPs for the entire design can be calculated as:

$$D = d_{BU} \times W/2 \times X \tag{2}$$

*3) **BRAM Resources (B)**:* Using URAM causes II violations during polyBuf optimizations, as Vitis HLS does not exploit read-first mode for URAMs. So, we prioritize assigning polynomial data into BRAMs. Without considering double buffering for simplicity, the number of buffers required by iterative and hybrid architectures is equal to W, and the depth of each buffer $d = \frac{2N}{W}$. So, the total number of BRAMs is:

$$B = W \times \left\lceil \frac{\log q}{bram\_width} \right\rceil \times \left\lceil \frac{d}{bram\_depth} \right\rceil \tag{3}$$

In the dataflow architecture, each shuffler gets W buffers, and the depth d_i depends on the shuffler index i. Hence, we can calculate the total BRAMs by accumulating the BRAMs required for each shuffler, which follows the above equation.

*4) **URAM Resources (U)**:* We prioritize assigning TFs to URAMs to maintain a balance between resources. Since we do TF storage optimization for each architecture, URAM storage varies for each architecture. For each architecture, we get the total URAM consumption by calculating the number of URAMs required for each TFBuf module based on the TFs stored in the buffer, $uram\_width$, and $uram\_depth$.

*5) **LUT & FF Resources**:* We use a linear interpolation-based method to estimate LUTs and FFs in a design. For each task, we obtain a linear model of how LUT and FF usages scale up based on the data width. For modules like crossbars, we use normalized utilization based on the number of inputs for the model. Since the linear models only output an estimation, we add $\pm 5\%$ tolerance for the output.

*6) **Off-Chip Memory Bandwidth (BW)**:* To ease the memory coalescing, we define the $DRAM\_WORD\_SIZE$ variable as either 4 or 8 to denote the number of bytes required for each value based on the data width $\log q$. Considering both loading and storing, the total data T_d for the polynomial is $T_d = 2N$. With TF storage optimization, the total TFs T_{tf} loaded from off-chip memory depend on architecture-specific optimizations, so we calculate $T_d = T_{tf}$ separately for TF loading. The total bandwidth required for each is:

$$BW = T_d \times DRAM\_WORD\_SIZE/L \tag{4}$$

V. Evaluation

A. Experimental Setup

AutoNTT is evaluated on the AMD/Xilinx Alveo U200 [29], U280 [30] and U50 [31] datacenter FPGAs. We use the TAPA [32] framework to develop our designs as task-parallel HLS designs and help improve the floorplanning. We use Vitis HLS 2023.2 to synthesize the HLS design to RTL and Vivado 2023.2 to place and route the designs. We run all designs on the actual FPGA boards with XRT 2022.2 and measure the FPGA kernel execution time (excluding CPU-to-FPGA transfer time). Unless otherwise specified, we assume both polynomial and TF data are available on off-chip memory at the start of the process. We report post place-and-route resource utilization. All designs shown in this section are built to support both NTT and INTT on the same hardware.

B. Comparison with State-of-the-art Efforts

Table IV compares our automatically generated NTT designs with state-of-the-art efforts. Based on our architectures, we generate these designs in a way similar to the original design configuration or similar to the resource utilization. As different architectures utilize different resources effectively, for a fair comparison, we define an Area Time Product (ATP) metric (lower is better), including all the resources.

1. Compared to the RTL-based exploration tool NTTGen [16], our dataflow and hybrid architectures deliver an average 1.15x improvement in latency over their simulation results while using 1.65x fewer BRAMs on average due to the integration of BUGs in our designs.
2. Proteus [17] generates a smaller design with 1 BU per NTT stage. Designs generated by AutoNTT for similar configurations consume more resources but achieve higher frequency and provide better average latency.
3. OpenNTT [18] uses on-the-fly TF generation to reduce on-chip memory. However, this consumes more DSPs and LUTs. Instead, as our optimizations reduce on-chip memory, we utilize other resources for BU computation and achieve similar performance with 1.64x average ATP gain.
4. By operating at 300MHz, ESC-NTT [10] achieves a lower latency, but if AutoNTT is also operated at the same frequency, it can achieve a similar latency. AutoNTT achieves 1.36x better throughput compared to them.
5. SAM's [33] lower frequency and reliance on accessing off-chip memory to support high polynomial sizes slow them down at FHE parameters. By using similar resources AutoNTT achieves 8.48x average latency improvement.
6. Compared to recent FHE accelerators FAB [14], Poseidon [25], and HEAP [15], AutoNTT generates designs with 4.35x average throughput gain.
7. Compared to [9], our optimized shuffler reduces on-chip buffer usage by 62.35%. While their custom Barrett reduction uses fewer DSPs than AutoNTT, we achieve lower latency with 1.33x ATP improvement.

In summary, on average, AutoNTT achieves 2.48x and 3.61x latency and throughput improvements compared to all these RLT-based implementations, providing competitive performance. AutoNTT achieves 3.53x ATP improvement on average, indicating it provides a better utilization of overall resources. It is also important to note that our solution offers significantly more flexibility than all these works.

C. Design Space Exploration Results

We performed design space exploration for the polynomial range of $2^{14}-2^{17}$, our supported data width range with a stride of 4, and using Barrett reduction. We provided U280 FPGA

TABLE IV: Comparison of AutoNTT with previous work

Method	N	q	**Red.**	**Device**	**Freq. (MHz)**	**Latency** (μs)	**Throughput** (NTT/s)	k**LUT / kFF / DSP / BRAM / URAM**	**ATP**
NTTGen [16]	2^{10}	28	Mer	Virtex7	210	1.10	-	206 / 159 / 640 / 80 / 0	238
AutoNTT-D[3]			Mer	U50	250	0.89	4,747,100	203 / 165 / 640 / 48 / 0	187
NTTGen [16]	2^{12}	30	B	U200	250	24.70	-	54 / 56 / 288 / 84 / 0	2,381
AutoNTT-H[3]			B	U200	250	24.78	40,352	58 / 42 / 288 / 72 / 0	2,279
Proteus [17]		32	WLM	Virtex7	150	13.80	-	8 / 4 / 44 / 8 / 0	176
AutoNTT-D			WLM	U50	250	9.94	100,506	27 / 25 / 72 / 19 / 0	284
OpenNTT [18]		60	WLM	Virtex7	240	13.10	-	35 / 44 / 352 / 18 / 0	1,176
AutoNTT-I			WLM	U50	250	13.21	37,826	26 / 24 / 152 / 64 / 0	702
Proteus [17]		64	WLM	Virtex7	150	13.80	-	23 / 18 / 220 / 16 / 0	764
AutoNTT-D			WLM	U50	250	9.94	100,506	51 / 52 / 228 / 38 / 0	733
ESC-NTT [10]			WLM	U280	300	3.81	1,171,875	523 / 1,478 / 6,518 / 275[2]/ 0	6,706
AutoNTT-D			WLM	U280	234	4.58	1,598,961	469 / 608 / 4,800 / 192 / 0	5,559
HEAP [15]	2^{13}	36	B	U280	300	-	210,000	-	/
AutoNTT-I			B	U280	231	1.80	556,534	699 / 504 / 5,888 / 1,024 / 384	4,027
NTTGen [16][4]		52	Mer	U200	220	4.50	-	276 / 250 / 480 / 400 / 0	1,265
AutoNTT-H[3]			Mer	U200	233	3.65	273,416	228 / 202 / 480 / 188 / 0	802
FAB [14]	2^{14}	54	WK	U280	300	-	167,000	-	/
AutoNTT-I			B	U280	231	3.11	321,046	663 / 608 / 5,632 / 1,152 / 256	6,284
Poseidon [25]	2^{16}	32	B	U280	450	-	12,474	358 / 344 / 4,032 / 1,024 / 0	/
AutoNTT-H			B	U280	249	9.43	105,986	491 / 435 / 2,816 / 896 / 0	8,747
OpenNTT [18]		52	WLM	ZCU102	210	78.40	-	157 / 117 / 896 / 98 / 0	19,882
AutoNTT-I			WLM	U50	250	66.54	15,029	80 / 60 / 288 / 498 / 0	12,323
SAM [33]		64	-	U250	165	380.00	-	267 / 328 / 2,736 / 2,126[2]/ 0	414,732
AutoNTT-I			B	U200	249	39.14	25,550	206 / 160 / 2,048 / 718 / 0	24,517
Kim et al. [9]	2^{17}	62	B	xcvu190	200	3760.00[1]	-	365 / 335 / 1,332 / 2,258[2] / 0	3,226,080
AutoNTT-D			B	U280	250	3457.00[1]	12,461	243 / 231 / 2,176 / 850 / 0	2,419,900
SAM [33]		64	-	U250	165	560.00	-	267 / 328 / 2,736 / 2,126[2]/ 0	611,184
AutoNTT-I			B	U200	246	77.11	12,969	218 / 166 / 2,048 / 1,343 / 0	58,218

Red.: Reduction. Freq.: Frequency. -: Not available. ATP: Area Time Product = $Latency \times (kLUT + kFF + DSP + BRAM + 8 * URAM)/5$. Mer: Mersenne prime reduction. B: Barrett reduction. WLM: Word Level Montgomery reduction. WK: Will-Ko reduction. [1]: 42 limbs. [2]: Converted KB to 36K BRAM. [3]: As in the original design, TFs are assumed to be in on-chip memory. [4]: Considered one limb resources for actual on board evaluation.

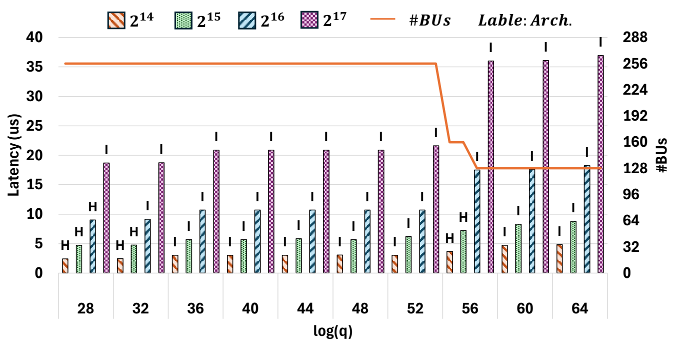

Fig. 9: Latency of different design points on U280. The results of the best architecture configurations are shown here.

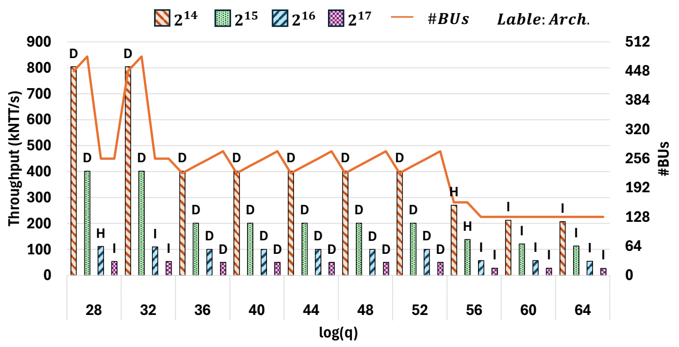

Fig. 10: Throughput of different design points on U280. The results of the best architecture configurations are shown here.

resources as the target resources, and limb parallelism is set to 1 to test the scalability of our designs. Fig. 9 and Fig. 10 show the best latency and the best throughput design points, respectively. In general, iterative architecture provides a better latency while dataflow architecture provides better throughput.

However, as parameters change, other options emerge as a better solution, breaking the general trend. For example, at $log\ q = 56$, when $N = 2^{14}$ and 2^{15}, the hybrid architecture supports a design of 160 BUs (i.e., $(16 \times 5\ BUG) \times 2$), whereas other architectures are limited to fewer BUs, resulting in improved latency and throughput. However, for polynomial sizes larger than 2^{15}, the hybrid architecture with (16×5) BUG experiences underutilized unrolled stages in the final round, causing the iterative design to outperform it once again. Thus, the results of the DSE emphasize the significance of leveraging different design options to meet a wide range of latency and throughput targets while adapting to varying resources.

D. Results Breakdown for Architecture Optimizations

1) ***Polynomial Buffer Optimization:*** Table V summarizes the BRAM savings on the U280 with two large data widths and higher n_{BU} counts, when $N = 2^{17}$. By using 25% (relatively)

TABLE V: BRAM saving of polynomial buffer optimization

Data Width (bit)	n_{BU}	#BRAMs 18K(%)	
		w/o Opt	w/ Opt
54	256	3,072 (76.19%)	2,304 (57.14%)
64	128	2,048 (50.79%)	1,536 (38.09%)

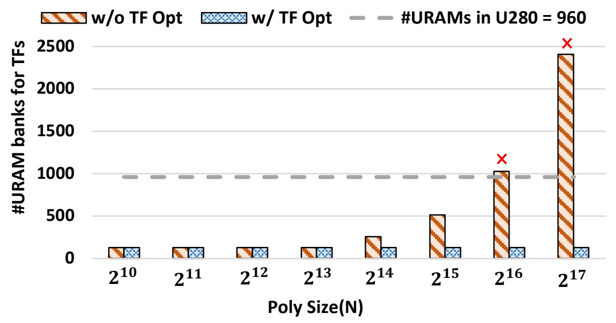

Fig. 11: URAM savings from TF buffer optimization for $n_{BU} = 256, \log q = 54$. ×: required URAM exceeds limit.

less BRAMs, this optimization significantly reduces BRAM usage for larger parameters in the iterative design.

*2) **TF Buffer Optimization**:* Fig. 11 indicates the total URAM savings from our TF buffer optimization for the iterative design. It shows that our optimization enables support for larger polynomial sizes with a URAM savings of $\sim$94%, making previously prohibitively expensive cases feasible.

*3) **Frequency Improvement From Shuffler Optimization**:* Table VI shows the frequency gain achieved by our hardware-friendly shuffler design compared to the monolithic shuffler design. This enables larger vector sizes, such as 64 and 128, with 1.65x average frequency gain in the hybrid architecture.

TABLE VI: Frequency improvement from the split shuffler

Data Width	Arch. Config	Vector Size	Freq w/o Opt(MHz)	Freq w/ Opt(MHz)
32	(8x4 BUG) x 8	128	161	249
36	(4x3 BUG) x 16	128	120	250
52	(16x5 BUG) x 2	64	175	233

*4) **Multi-word Multiplication**:* Table VII shows the resource and latency comparison of our optimized BU with different reduction methods for two data sizes. Other data sizes also follow a similar trend. Using multi-word multiplication, AutoNTT achieves similar resource utilization and better latency compared to those RTL-based designs.

TABLE VII: Comparison of BU resource and latency

Data Width	Reduction	Design	LUT	FF	DSP	Latency (cycles)
30	B	NTTGen [16]	1081	920	12	14
	B	**AutoNTT**	766	403	11	10
	WLM	OpenNTT [18]	380	449	7	14
	WLM	**AutoNTT**	635	429	6	10
	M	**AutoNTT**	813	434	11	10
52	B	NTTGen [16]	1837	1682	17	19
	B	**AutoNTT**	1319	1121	17	12
	WLM	OpenNTT [18]	1476	1787	14	20
	WLM	**AutoNTT**	1168	891	9	11
	M	**AutoNTT**	1491	1075	17	12

B: Barrett. M: Montgomery. WLM: Word Level Montgomery.

VI. Related Work

The NTT studies found in the literature are twofold: 1) architectures supporting a set of fixed parameters, 2) automation frameworks. Zhang et al. [7] proposed a tensor product-based iterative architecture with a unified communication pattern between different NTT stages. Kim et al. [9] proposed an NTT architecture by combining 12 BUs as a group and connecting them in a pipeline fashion to support a large polynomial size of 2^{17} and a higher number of limbs compared to prior work. Duong-Ngoc et al. [8] presented an architecture with 8×4 2D BU array and conflict-free memory access pattern. All these architectures are RTL-based fixed architectures and do not provide automation to support diverse conditions.

Hirner et al. [17] proposed a tool to generate radix-2-based NTT architectures with single-path delay feedback and multi-path delay commutator approaches. However, it only supports dataflow architecture and limits a single BU per NTT stage, limiting the scalability and performance. SAM [33] proposes an NTT architecture to support very large polynomial sizes and data sizes used in zero-knowledge proof (ZKP) applications. Due to larger parameters, they always communicate with off-chip memory, hence limiting the performance when it comes to FHE parameters. OpenNTT [18] provides an automation tool to generate iterative designs, but does not explore other design configurations and has a limited input vector size in its results. Mu et al. [12] proposed an NTT automation solution with a conflict-free memory access pattern; but their glue logic between BUs and buffers is implemented with MUXes and is not scalable.

NTTGen [16] is an automation framework that is closest to ours and supports different architectures. However, they only target HE parameters and generate smaller NTT units to process multiple limbs in parallel, which limits the scalability. They use a streaming permutation network (SPN) after each BU layer to support different data access patterns between stages. SPN is a building block that supports N-to-N connection using two spatial networks and one temporal network [16]. However, using generalized SPN after every layer increases the latency and BRAM resources.

VII. Conclusion

NTT implementations on FPGA span across diverse architecture options and configurations to support diverse input parameters and performance targets. The current approaches have limited design space by supporting limited NTT architectures and FHE parameters, while the scalability of their designs does not meet the expectations of recent FHE accelerators. Therefore, we present AutoNTT that can support a wide range of FHE parameters, highly scalable iterative, dataflow, and hybrid architectures with diverse configurations, and common modular reduction algorithms. AutoNTT performs design space exploration and generates highly efficient NTT designs for a given resource budget. Supporting HLS based design approach, AutoNTT generates NTT accelerator designs with 2.48× better latency and 3.61× better throughput on average, while maintaining a similar resource utilization.

References

[1] M. Zheng, Q. Lou, and L. Jiang, "Primer: Fast private transformer inference on encrypted data," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2023, pp. 1–6.

[2] J. Park, S. Lee, and J. Lee, "Ntt-pim: Row-centric architecture and mapping for efficient number-theoretic transform on pim," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2023, pp. 1–6.

[3] J. Casas, Z. Yang, W. Wang, J. Yang, and A. Godbole, "Towards a formally verified fully homomorphic encryption compute engine," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2023, pp. 1–6.

[4] N. Samardzic, A. Feldmann, A. Krastev, N. Manohar, N. Genise, S. Devadas, K. Eldefrawy, C. Peikert, and D. Sanchez, "Craterlake: a hardware accelerator for efficient unbounded computation on encrypted data," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 173–187.

[5] M. Kumar and P. Pattnaik, "Post quantum cryptography (pqc)-an overview," in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2020, pp. 1–9.

[6] "Microsoft SEAL (release 3.2)," https://github.com/Microsoft/SEAL, Feb. 2019, microsoft Research, Redmond, WA.

[7] Y. Zhang, S. R. Sathi, Z. Kou, S. Sinha, and W. Zhang, "Tensor-product-based accelerator for area-efficient and scalable number theoretic transform," in *2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2023, pp. 174–183.

[8] P. Duong-Ngoc, S. Kwon, D. Yoo, and H. Lee, "Area-efficient number theoretic transform architecture for homomorphic encryption," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 70, no. 3, pp. 1270–1283, 2023.

[9] S. Kim, K. Lee, W. Cho, Y. Nam, J. H. Cheon, and R. A. Rutenbar, "Hardware architecture of a number theoretic transform for a bootstrappable rns-based homomorphic encryption scheme," in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2020, pp. 56–64.

[10] Z. Guan, Y. Zhu, Y. Huang, L. Lei, X. Wang, H. Jia, Y. Chen, B. Zhang, J. Dong, and S. Bian, "Esc-ntt: An elastic, seamless and compact architecture for multi-parameter ntt acceleration," in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2024, pp. 1–6.

[11] A. C. Mert, E. Karabulut, E. Öztürk, E. Savaş, M. Becchi, and A. Aysu, "A flexible and scalable ntt hardware : Applications from homomorphically encrypted deep learning to post-quantum cryptography," in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2020, pp. 346–351.

[12] J. Mu, Y. Ren, W. Wang, Y. Hu, S. Chen, C.-H. Chang, J. Fan, J. Ye, Y. Cao, H. Li, and X. Li, "Scalable and conflict-free ntt hardware accelerator design: Methodology, proof, and implementation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 5, pp. 1504–1517, 2023.

[13] M. S. Riazi, K. Laine, B. Pelton, and W. Dai, "Heax: An architecture for computing on encrypted data," in *Proceedings of the twenty-fifth international conference on architectural support for programming languages and operating systems*, 2020, pp. 1295–1309.

[14] R. Agrawal, L. de Castro, G. Yang, C. Juvekar, R. Yazicigil, A. Chandrakasan, V. Vaikuntanathan, and A. Joshi, "Fab: An fpga-based accelerator for bootstrappable fully homomorphic encryption," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 882–895.

[15] R. Agrawal, A. Chandrakasan, and A. Joshi, "Heap: A fully homomorphic encryption accelerator with parallelized bootstrapping," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, 2024, pp. 756–769.

[16] Y. Yang, S. R. Kuppannagari, R. Kannan, and V. K. Prasanna, "Nttgen: a framework for generating low latency ntt implementations on fpga," in *Proceedings of the 19th ACM International Conference on Computing Frontiers*, 2022, pp. 30–39.

[17] F. Hirner, A. C. Mert, and S. S. Roy, "Proteus: A pipelined ntt architecture generator," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2024.

[18] F. Krieger, F. Hirner, A. C. Mert, and S. S. Roy, "Openntt: An automated toolchain for compiling high-performance ntt accelerators in fhe," *Cryptology ePrint Archive*, 2024.

[19] P. 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*. Springer, 1986, pp. 311–323.

[20] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of computation*, vol. 44, no. 170, pp. 519–521, 1985.

[21] A. C. Mert, E. Öztürk, and E. Savaş, "Design and implementation of a fast and scalable ntt-based polynomial multiplier architecture," in *2019 22nd Euromicro Conference on Digital System Design (DSD)*, 2019, pp. 253–260.

[22] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," *Mathematics of Computation*, vol. 19, pp. 297–301, 1965. [Online]. Available: https://api.semanticscholar.org/CorpusID:121744946

[23] W. M. Gentleman and G. Sande, "Fast fourier transforms: for fun and profit," in *Proceedings of the November 7-10, 1966, Fall Joint Computer Conference*, ser. AFIPS '66 (Fall). New York, NY, USA: Association for Computing Machinery, 1966, p. 563–578.

[24] R. Agrawal, L. De Castro, C. Juvekar, A. Chandrakasan, V. Vaikuntanathan, and A. Joshi, "Mad: Memory-aware design techniques for accelerating fully homomorphic encryption," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 685–697.

[25] Y. Yang, H. Zhang, S. Fan, H. Lu, M. Zhang, and X. Li, "Poseidon: Practical homomorphic encryption accelerator," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 870–881.

[26] B. Zhang, Z. Cheng, and M. Pedram, "An iterative montgomery modular multiplication algorithm with low area-time product," *IEEE Transactions on Computers*, vol. 72, no. 1, pp. 236–249, 2023.

[27] S. Kim, K. Lee, W. Cho, J. H. Cheon, and R. A. Rutenbar, "Fpga-based accelerators of fully pipelined modular multipliers for homomorphic encryption," in *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 2019, pp. 1–8.

[28] AMD/Xilinx, "Vitis high-level synthesis user guide (ug1399)," 2023. [Online]. Available: https://docs.amd.com/r/2023.2-English/ug1399-vitis-hls/Arbitrary-Precision-AP-Data-Types

[29] ——, "Alveo u200 and u250 data center accelerator cards data sheet (ds962)," 2023. [Online]. Available: https://docs.amd.com/r/en-US/ds962-u200-u250

[30] ——, "Alveo u280 data center accelerator card data sheet (ds963)," 2023. [Online]. Available: https://docs.xilinx.com/r/en-US/ds963-u280/Summary

[31] ——, "Alveo u50 data center accelerator card data sheet (ds965)," 2023. [Online]. Available: https://docs.amd.com/r/en-US/ds965-u50/Summary

[32] L. Guo, Y. Chi, J. Lau, L. Song, X. Tian, M. Khatti, W. Qiao, J. Wang, E. Ustun, Z. Fang, Z. Zhang, and J. Cong, "TAPA: A scalable task-parallel dataflow programming framework for modern FPGAs with co-optimization of HLS and physical design," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 16, no. 4, dec 2023.

[33] C. Wang and M. Gao, "Sam: A scalable accelerator for number theoretic transform using multi-dimensional decomposition," in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 2023, pp. 1–9.