Efficient Design-Time Flexible Hardware Architecture for Accelerating Homomorphic Encryption

Can Ayduman*[1], Emre Koçer*[1], Selim Kırbıyık*[1], Ahmet Can Mert[2] and Erkay Savaş[1]

[1] Sabancı University, Istanbul, Turkey,

{canayduman,kocer,selimkirbiyik,erkays}@sabanciuniv.edu

[2] Graz University of Technology, Graz, Austria,

ahmet.mert@iaik.tugraz.at

This manuscript has been accepted for publication in an IEEE conference.

The final published version is available in IEEE Xplore:

DOI: 10.1109/VLSI-SoC57769.2023.10321943

© IEEE.

Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, including reprinting, republishing for advertising or promotional purposes, creating new collective works for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

This version is made publicly available in the institutional repository of Sabancı University in accordance with IEEE self-archiving policies and to comply with open access requirements of the European Union-funded project.

Funding Acknowledgement:

This work has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement Number: 101079319.

Note:

This is not the final published version. Please refer to the published version via the DOI link above.

Efficient Design-Time Flexible Hardware Architecture for Accelerating Homomorphic Encryption

Can Ayduman*[1], Emre Koçer*[1], Selim Kırbıyık*[1], Ahmet Can Mert[2] and Erkay Savaş[1]

[1] Sabancı University, Istanbul, Turkey,
{canayduman,kocer,selimkirbiyik,erkays}@sabanciuniv.edu
[2] Graz University of Technology, Graz, Austria,
ahmet.mert@iaik.tugraz.at

Abstract. This paper presents a design-time configurable hardware generator for hardware acceleration of the CKKS Fully Homomorphic Encryption (FHE) scheme. Our design aims to accelerate the multiplication and relinearization operations of the CKKS. It includes a design-time configurable Number Theoretic Transform (NTT) multiplication hardware for polynomial sizes between 2^{10} and 2^{15}. The NTT-based multiplication realizes modular multiplication using an efficient word-level Montgomery reduction algorithm.
Polynomial multiplication is a bottleneck for the FHE operations. The NTT enables very fast polynomial multiplication by reducing its complexity to $\mathcal{O}(n \log_2 n)$ from $\mathcal{O}(n^2)$. The fundamental arithmetic block of the NTT operation is the butterfly, which implements four different operations, namely, modular multiplication and modular addition/subtraction.
The memory access pattern (MAP) of the NTT operation is complex, and it is crucial to design an efficient MAP for NTT for implementing a high-throughput NTT architecture. We designed and implemented an efficient algorithm for the MAP of NTT and generalized this approach for polynomial sizes, 2^{10} to 2^{15}.

Keywords: FPGA · FHE · NTT · acceleration · CKKS

1 Introduction

Fully Homomorphic Encryption (FHE) is a type of encryption that enables calculations on the encrypted data without decrypting it. It was shown to be possible with the work of Gentry et al. presented in [Gen09]. With FHE, computations on plaintext data yield the same result as the computations on the encrypted data once decrypted. The goal of FHE is to allow computations to be done in the encrypted domain and preserve the privacy of the data during the computations. The importance of FHE comes from its ability to keep sensitive data private and secure, especially in situations involving cloud computing and multi-party computations. It enables a secure method to process private data without the need to access unencrypted data.

The Cheon-Kim-Kim-Song (CKKS) [CKKS17] scheme plays a significant role in executing nonlinear operations over real numbers. Initially, CKKS, like several other schemes, began as a Somewhat Homomorphic Encryption (SHE) system before evolving into a Fully Homomorphic system. SHE schemes allow a limited number of arithmetic operations on encrypted data before the 'noise' exceeds a certain threshold. When this noise budget is exhausted, additional operations can lead to decryption errors. In contrast, an FHE

* Co-first authors.

system ideally allows for unlimited computation. The transition from SHE to FHE usually relies on a process called *bootstrapping*, first proposed by Gentry [Gen09]. This method, although beneficial, is computationally intensive and resets the noise level in ciphertexts to ensure continuous computations.

Microsoft SEAL [SEA23] is a software library that optimizes FHE operations and provides an abstraction layer for building applications. One of the implemented schemes is CKKS, which is suitable for hardware acceleration due to its computation-heavy operations.

Our motivation lies in addressing the computational challenges that hinder the large-scale applicability of Homomorphic Encryption (HE). In this context, our study focuses on a specially optimized hardware accelerator designed to perform high-degree polynomial multiplications efficiently. This accelerator, which offers a unique advantage for the critical component of the CKKS scheme, the NTT operations, boasts design flexibility, allowing it to adapt to various parameter sets. Consequently, it can meet diverse performance and security requirements, potentially expanding the practical applications of HE from a data privacy and security perspective. Ultimately, this paper presents a hardware strategy to overcome the computational barriers of Homomorphic Encryption.

In this paper, we first present a design-time flexible hardware for the Number Theoretic Transform (NTT) operation, one of the most fundamental CKKS operations, enabling fast polynomial multiplication. Then, we design and implement a hardware accelerator for the main homomorphic operations of the CKKS scheme, homomorphic multiplication, and key switching. Compared to the CPU implementation in Microsoft SEAL, our implementation shows 15× and 4× speedups for homomorphic multiplication and key switching operations, respectively.

The rest of the paper is organized as follows. Section 2 presents the background. Section 3 introduces the CKKS scheme. Section 4 presents details of low-level arithmetic units. Section 5 presents the proposed NTT and CKKS architectures. In Section 6, the implementation results are presented and Section 9 concludes the paper.

2 Preliminaries

This section will explain the mathematical background of the CKKS FHE scheme and its arithmetic operations.

2.1 Notation

Throughout the paper, n represents the polynomial size and PE represents the number of processing elements (butterfly units), q is an NTT-friendly prime number (i.e., in the form of $q \equiv 1 \mod 2^m$ where $m \in [13, 19]$). The constant ω is the $n-th$ root of unity and ψ is the $2n-th$ root of unity in polynomial ring $\mathbb{Z}_q[x]/x^n+1$. R is the Montgomery constant. `BitReverse(value, bit length)` is a function returning the binary value in reverse order (i.e., `BitReverse(`11_2`,4)` returns 1100_2). For the CKKS scheme, q_{sp} represents a special modulus that is used for modulus switching operation and l represents the number of word-size primes in q.

2.2 Polynomial Multiplication

Polynomial multiplication is the most fundamental and time-consuming block of FHE schemes. Thus, it is crucial to have efficient hardware for polynomial multiplication. Multiplication of two large-degree polynomials is defined over the ring $\mathbb{Z}_q[x]/\phi(x)$. In this ring, polynomials have a degree at most $n-1$, and coefficients are in modulo q. Also note that $\phi(x) = x^n + 1$. The polynomial multiplication under this ring for two

Algorithm 1 Homomorphic Multiplication for CKKS

Input: $ct_a = (ct_{a,0}, ct_{a,1}), ct_b = (ct_{b,0}, ct_{b,1})$
$ct_a, ct_b \in R_Q^{lx2}, Q = \prod_{i=1}^{l} q_i$
Output: $res = (res_0, res_1, res_2),\ res \in R_Q^{lx3}$
1: **for** $i = 1 \rightarrow l$ **do**
2: $res_{0,i} \leftarrow ct_{a,0,i} \odot ct_{b,0,i}$
3: $res_{1,i} \leftarrow ct_{a,0,i} \odot ct_{b,1,i} + ct_{a,1,i} \odot ct_{b,0,i}$
4: $res_{2,i} \leftarrow ct_{a,1,i} \odot ct_{b,1,i}$
5: **return** res

polynomials $A(x) = \sum_{i=0}^{n-1} \alpha_i x^i$ and $B(x) = \sum_{i=0}^{n-1} \beta_i x^i$ are defined as $C(x) = A(x) \cdot B(x)$ where $C(x) = \sum_{i=0}^{n-1} \gamma_i x^i$.

The naive method to implement polynomial multiplication is schoolbook multiplication. The complexity of this method is $\mathcal{O}(n^2)$ (since every coefficient needs to be multiplied with coefficients of another polynomial). On the other hand, NTT has $\mathcal{O}(n \log_2 n)$ complexity. There are $\log_2 n$ stages for an NTT, and for every stage $\frac{n}{2}$ butterfly operations need to be computed.

2.3 NTT-based Polynomial Multiplication

NTT-based polynomial multiplication has three stages: Forward NTT (FNTT), Inner Product, and Inverse NTT (INTT). Twiddle factors ($\psi \equiv 1 \mod 2n$) are utilized for computing FNTT and INTT. The inner product ($\odot$) is a simple element-wise multiplication between coefficients of two polynomials in the NTT domain. In summary, NTT-based polynomial multiplication is defined as shown in Eqn. 1.

$$A(x) \cdot B(x) = INTT(FNTT(A) \odot FNTT(B)) \quad (1)$$

3 CKKS Scheme Operations in Microsoft SEAL

This section briefly explains the fundamental operations of the CKKS homomorphic encryption scheme, namely multiplication and key switching, as implemented in Microsoft SEAL library [SEA23].

3.1 Homomorphic Multiplication

In the CKKS scheme, the homomorphic multiplication operation is carried out using a sequence of smaller moduli (q_i) referred to as an RNS base rather than using a single large coefficient modulus (q). This allows the required operations to be executed parallel across the RNS moduli. The homomorphic multiplication operation in the CKKS scheme is relatively cheaper compared to the BFV homomorphic multiplication, which uses auxiliary RNS bases and performs expensive base extensions. The CKKS homomorphic multiplication, as shown in Alg. 1, does not use any auxiliary RNS base and only consists of coefficient-wise ciphertext multiplication operations. The homomorphic multiplication operation for the CKKS scheme takes two ciphertexts, each with two components in the RNS base. Then, it computes a ciphertext with three components as output.

3.2 Key Switching

After the homomorphic multiplication, the resulting ciphertext has three components. The key switching operation is used to reduce these three components to two components.

Key switching is a more computationally intensive operation compared to homomorphic multiplication. The NTT is the bottleneck for key switching operations because the total number of NTTs and INTTs used in key switching operations is approximately equal to the square of the number of RNS bases.

The key switching procedure for the CKKS scheme consists of two parts. The first part is the relinearization operation, which begins by applying the INTT and NTT operations in sequence to the third component of the ciphertext produced by the homomorphic multiplication. Subsequently, the first part is completed with the multiplication of the relinearization key. In the second part, the resulting component corresponding to the last prime modulus of the RNS base is converted back to the polynomial domain using INTT. Then, the modulus switching operations are applied [SEA23].

4 Low-level Arithmetic Units

This section explains the low-level arithmetic units of the proposed hardware architecture, modular addition/subtraction, integer multiplication, modular reduction, and butterfly modules (Cooley-Tukey (CT) and Gentleman-Sande (GS)).

4.1 Modular Addition and Subtraction Units

The modular addition/subtraction module is one of the main building blocks of this architecture. Hence, having an efficient and fast addition-subtraction module in the design is crucial. We designed and implemented constant-time modular addition and subtraction units.

4.2 Modular Multiplication Unit

Modular multiplication is a crucial block. While there are several approaches for modular multiplication (Barrett [Bar87], K2-RED [BNAMK21], Plantard [HZZ$^+$22]), Montgomery modular multiplication [M19] is often preferred for FPGA implementations due to its suitability for pipelining. The module takes two 32-bit integers (A, B) and produces a 64-bit output C, where $C = A \cdot B$. DSP blocks in the FPGA are utilized to improve performance.

We adopted the word-level Montgomery reduction algorithm for modular reduction with NTT-friendly primes [M19]. The word-level Montgomery reduction unit is implemented with pipelining techniques to improve its performance.

4.3 Unified Butterfly Unit

Our butterfly module can perform multiple operations by changing its control inputs, including CT configuration for forward NTT, GS configuration for inverse NTT, modular multiplication, and modular addition/subtraction.

5 The Proposed NTT and CKKS Architectures

5.1 Merged NTT/INTT Algorithm

Merged-NTT algorithm (Alg. 2) performs the NTT operation while avoiding the costly coefficient reorganization steps. Unlike the traditional NTT algorithms, merged-NTT/INTT deals with n elements instead of $2n$ while performing polynomial multiplication in polynomial ring $\mathbb{Z}_q[x]/x^n + 1$. Roy et al. [RVM$^+$14] merged pre-processing and NTT algorithm

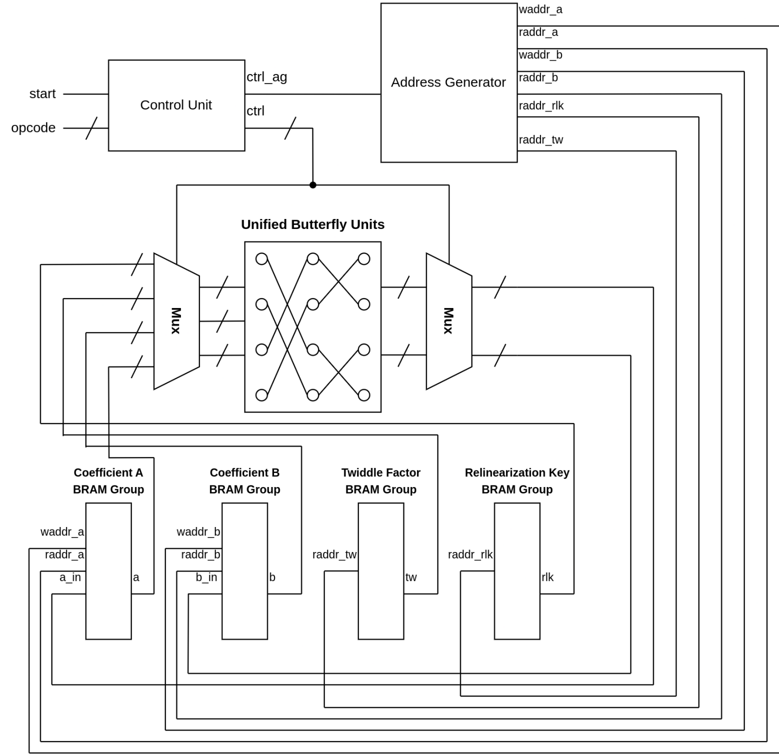

Figure 1: Overview of the Accelerator Architecture

(merged NTT), and Pöppelmann et al. [POG15] merged INTT and post-processing operations for INTT algorithm (merged INTT). Merged-NTT utilizes CT butterfly operation, and Merged-INTT utilizes GS butterfly operation. These two algorithms are the current state-of-the-art NTT-INTT techniques to implement NTT-based polynomial multiplication. In this work, we used these operations. ψ_{table} are precomputed powers of ψ given in natural order. Also, note that ψ_{table} is stored in Block RAMs in Montgomery form (i.e., $\psi \times R \bmod q$).

5.2 The Parametric NTT Hardware and Memory Optimizations

Implementing an NTT architecture on an FPGA is challenging due to the need for complex Memory Access Patterns (MAP) to achieve high throughput. Supporting various butterfly units provides greater flexibility in resource allocation, which is particularly important in FPGA systems where resources may be limited. Mert et al. [Mer21] provide an open-source parametric hardware design for merged-NTT, and its MAP can be generalized to the larger ring sizes and different numbers of butterfly units. The NTT has $\log_2 n$ stages, and the MAP for each NTT stage is different. Therefore, the locations of coefficients stored in the memory must be determined carefully for every stage to eliminate any data conflicts. We utilized a generalized MAP for every ring size ranging from 2^{10} to 2^{15}, and we have designed and implemented parametric hardware that can be configured at design-time to perform Merged NTT/INTT for a given polynomial size and the number of butterfly units. Fig. 1 shows the architecture of the overall design. The unified butterfly unit implements butterfly operations (CT and GS), modular multiplication, and modular addition/subtraction for 32-bit numbers. An address generator is included to produce BRAM read/write addresses for read/write operations. The control unit is responsible

Algorithm 2 Merged-NTT Algorithm with CT Butterfly

Input: $A(x) \in R_{q,n}$ in natural order
Input: $\Psi_{table} \in \mathbb{Z}_q$
Output: $B(x) \in R_{q,n}$ in bit-reversed order

1: $N \leftarrow \text{length}(A)$
2: $B \leftarrow A$
3: $l \leftarrow \log_2(N)$
4: **for** m_{power} from 0 to $l-1$ **do**
5: $\quad m \leftarrow 2^{m_{power}}$
6: $\quad t \leftarrow \frac{N}{2m}$
7: $\quad$ **for** i from 0 to $m-1$ **do**
8: $\quad\quad z_1 \leftarrow 2 \times i \times t$
9: $\quad\quad z_2 \leftarrow z_1 + t$
10: $\quad\quad \Psi_{pow} \leftarrow \text{BitReverse}(m+i, l)$
11: $\quad\quad S \leftarrow \Psi_{table}[\Psi_{pow}]$
12: $\quad\quad$ **for** z from z_1 to $z_2 - 1$ **do**
13: $\quad\quad\quad U \leftarrow B[z]$
14: $\quad\quad\quad V \leftarrow (B[z+t] \times S) \mod q$
15: $\quad\quad\quad B[z] \leftarrow (U+V) \mod q$
16: $\quad\quad\quad B[z+t] \leftarrow (U-V) \mod q$
17: **return** B

for altering the mode of operation for the unified butterfly units, address generator, and input multiplexers for the unified butterfly units. Multiplexers (Mux) are used to order the input and output data to ensure correct read/write operations. BRAM groups serve as the storage for coefficients for input polynomials, twiddle factors, and key switching keys.

For a given polynomial size and number of butterfly units, a new NTT architecture is required, as each configuration has a different MAP and computation order. Our NTT design generates the necessary address and control logic that can handle the complex MAP for a given configuration (polynomial size and number of butterfly units). This provides significant flexibility over an implementation tailored for a fixed configuration. Our NTT design can accommodate any resource constraint and can generate NTT configurations for polynomial sizes ranging from 2^{10} to 2^{15}.

5.3 The Architecture for the CKKS Scheme Operations

In this section, for parameters $n = 8192$ and $\lceil \log_2(q) \rceil = 218$, we present a high-performance and scalable hardware architecture that implements the homomorphic multiplication, relinearization and rescaling operations for the RNS variant of the CKKS scheme, as discussed in Section 3.

5.3.1 Homomorphic Multiplication

In CKKS, as seen in the Alg. 1, homomorphic multiplication consists of point-wise modular multiplication and modular addition operations. The unified butterfly unit presented in Section 4 is used to perform these operations. According to the chosen parameters, each of the four ciphertexts consists of six polynomials, each in a different RNS base. In total, there are 18 polynomials of degree 2^{13}. Each polynomial is divided into 8 chunks, each with 1024 elements, and each is stored in one BRAM group.

Algorithm 3 Homomorphic Rescale for CKKS

Input: $ct'_0, ct'_1 \in R^l_Q$
Output: $\bar{ct}_0, \bar{ct}_1 R^l_Q$
1: $\alpha \leftarrow INTT(ct'_{0,l})$
2: $\beta \leftarrow INTT(ct'_{1,l})$
3: $halfmod \leftarrow (q_l \gg 1)$
4: $\bar{\alpha} \leftarrow \alpha + halfmod \mod q_l$
5: $\bar{\beta} \leftarrow \beta + halfmod \mod q_l$
6: **for** $i = 0 \rightarrow l - 1$ **do**
7: $\quad \alpha' \leftarrow \bar{\alpha} - halfmod \mod q_i$
8: $\quad \beta' \leftarrow \bar{\beta} - halfmod \mod q_i$
9: $\quad \tilde{\alpha} \leftarrow NTT(\alpha')$
10: $\quad \tilde{\beta} \leftarrow NTT(\beta')$
11: $\quad \gamma_0 \leftarrow sum_{0,i} - \tilde{\alpha}$
12: $\quad \gamma_1 \leftarrow sum_{1,i} - \tilde{\beta}$
13: $\quad \gamma'_0 \leftarrow ct_{0,i} + q_l^{-1} \odot \gamma_0 \mod q_i$
14: $\quad \gamma'_1 \leftarrow ct_{1,i} + q_l^{-1} \odot \gamma_1 \mod q_i$
15: $\bar{ct}_0 \leftarrow \gamma'_0$
16: $\bar{ct}_1 \leftarrow \gamma'_1$
17: **return** $\bar{ct}_0, \bar{ct}_1$

5.3.2 Key Switching

This section describes the design for the CKKS relinearization and rescale operations. These operations utilize modular multiplication, addition, and subtraction operations. To enable parallel execution of data-independent operations, 96 unified butterfly units are employed, along with 408 BRAM groups to store inputs and intermediate values. Among these BRAM groups, 144 are set to a depth of 2048, 56 to 2559, and 208 to 1024.

The relinearization operation of the CKKS scheme is presented in Alg. 4. During the key switching operation, each polynomial needs to be multiplied with a relinearization key, and for each polynomial $(l-1) \times 2$ relinearization keys need to be stored. A total of 84 relinearization keys are stored for the selected parameters, and each polynomial must be multiplied by the relinearization key. BRAMs are utilized for storing 6 relinearization keys, and we are reusing the same BRAMs for storing other relinearization keys. To store relinearization keys, 48 BRAM groups with a depth of 2048 are fully utilized. The BRAM groups can store relinearization keys for 6 polynomials, and the operations of 6 polynomials can be carried out in parallel. The relinearization keys required for the next polynomials are stored in the same BRAMs in sequence. The primitive root of unity powers required for NTT and INTT operations are stored in 56 BRAMs with a depth of 2559. Due to the use of the word-level Montgomery reduction algorithm, the powers of the primitive root of unity and the relinearization keys are multiplied by the Montgomery constant R before being sent to the FPGA.

In the CKKS scheme, the relinearization process starts with the INTT and NTT operations which are performed in two nested loops. By unrolling these loops, it is possible to parallelize the operations. For each RNS base, eight unified butterfly units are used to parallelize these operations. Each unified butterfly unit is connected to two BRAM groups, which store polynomials, and one BRAM group, which stores the twiddle factors. An address generation control unit produces the necessary reading and writing addresses for the BRAM groups.

Unified butterfly units are used for the modular multiplication operation of ciphertexts with the relinearization keys. The unified butterfly units operate with inputs polynomial

Algorithm 4 Key Switching for CKKS

Input: $ct_a, ct_b, ct_c \in R_Q^l$,
$\tilde{Q} = q_{sp} \cdot Q,\ KRK_0, KRK_1 \in \tilde{Q}$
Output: $ct'_0, ct'_1 \in R_Q^l$
1: $val0, val1 \leftarrow 0$
2: **for** $i = 0 \rightarrow l-1$ **do**
3: $\alpha \leftarrow INTT_{q_i}(ct_{c,i})$
4: **for** $j = 0 \rightarrow l$ **do**
5: $\bar{\beta} \leftarrow NTT_{q_j}(\alpha)$
6: $val0 \leftarrow \bar{\beta} \odot KRK_{0,i,j} \mod q_j$
7: $ct'_{0,j} \leftarrow ct'_{0,j} + val0 \mod q_j$
8: $val1 \leftarrow \bar{\beta} \odot KRK_{1,i,j} \mod q_j$
9: $ct'_{1,j} \leftarrow ct'_{1,j} + val1 \mod q_j$
10: **return** ct'_0, ct'_1

Table 1: Results for the Merged NTT

# of PE	n	LUT	BRAM	DSP	LUTRAM	FF	Period (ns)	Clock Cycle	Latency (μs)
8	2^{12}	7390	24	56	542	5549	5.5	3096	17
	2^{13}	7099	40	56	544	5599	5.5	6682	36.7
	2^{14}	7428	68	56	546	5653	5.5	14364	79
16	2^{12}	15201	32	112	1073	11830	5.5	1560	8.5
	2^{13}	14695	48	112	1075	11839	5.5	3354	18.4
	2^{14}	15227	80	112	1077	11915	5.5	7196	39.5
32	2^{12}	29760	48	224	2088	21406	5.5	792	4.3
	2^{13}	29189	64	224	2100	21443	5.5	1690	9.2
	2^{14}	29100	96	224	2098	21556	5.5	3612	19.8

coefficients (A, B), twiddle factor (PSI), and modulus (Q), being 0, polynomial term, relinearization key, and the RNS base to be used, respectively. When performing modular multiplication operations, reading addresses ranging from 0 to 1023 are generated with the purpose of reading the polynomial terms to be sent as input to the unified butterfly units from the BRAM groups. The outputs of the modular multiplication operation performed by the unified butterfly units are written to all of the previously uninitialized 96 BRAM groups with a depth of 1024. After these operations, for the modular addition of the resulting polynomials, the unified butterfly units operate with inputs A, B, PSI, and Q, being the first polynomial term, the second polynomial term, 0, and the RNS base to be used, respectively. During the modular addition operations, addresses are generated to read the polynomial terms to be sent as input to the unified butterfly units from the BRAM groups, and the outputs are written to all of the previously uninitialized 108 BRAM groups with a depth of 1024.

Table 2: Comparative Table for Merged NTT Implementations

Work	Platform	n	LUT/BRAM/DSP	Per. (ns)	Lat. (μs)
[SS17]	Virtex-7	2^{14}	219K/193/768	4	24.5
[SRTJ+19]	UltraScale	2^{12}	64K/400/200	4.4	73
[DMGS23]	UltraScale	2^{12}	3320/29.5/42	5.5	136.58
Our	AU280	2^{12}	29760/48/224	5.5	4.3
		2^{13}	29189/64/224	5.5	9.2
		2^{14}	29100/96/224	5.5	19.8

Table 3: Resource Utilization for CKKS Operations (Multiplication+Key Switching)

Platform	n	**LUT/BRAM/DSP**	**Per.** (ns)	**Lat.** (μs)
AU280	2^{13}	372940/768/672	5.5	554.1

After completing the relinearization part, the rescale operation, described in Alg. 3, begins. The outputs of the INTT operation and the addition with half of the final RNS base are saved to the BRAM groups where the inputs were taken from. The unified butterfly units are configured to for the INTT and addition operations. The modular subtraction operations, NTT operations, and modular addition operations are then executed in parallel using 96 unified butterfly units. The configuration of the unified butterfly units for modular subtraction is executed in the same manner as the previously mentioned modular addition configuration. Finally, the outputs generated are written back to the BRAM groups where the ciphertexts used as inputs at the beginning of the relinearization process used to be stored.

6 Implementation Results and Comparison

The architecture presented in this paper is coded using Verilog and synthesized using the Xilinx Vivado 2019.2 tool, targeting the Xilinx Alveo U280 board.

6.1 Implementation Results for the NTT

We synthesized our NTT design and attained a 181MHz operating frequency. Specifically, our NTT is synthesized for $n = 8192$ when used for homomorphic encryption operations within the CKKS scheme, while standalone NTT architectures supporting $n = 4096$ and $n = 16384$ are also synthesized for exploratory purposes, without being integrated into specific operations. These setups operate with a word size defined by $\log_2 q = 32$-bit. Table 1 summarizes the synthesis results of our NTT unit with 32-bit word size.

In the literature, there are several NTT implementations. However, these implementations are tailored specifically for particular ring sizes and numbers of processing elements. Our parametric implementation, on the other hand, not only supports a wide range of polynomial sizes but also can achieve better performance than those in the literature by increasing the number of PEs. Architectures from different works targeting various polynomial sizes are presented to ensure a fair comparison.

6.2 Implementation Results for the CKKS Scheme Operations

The architecture targeting the CKKS scheme supports homomorphic multiplication, relinearization, and rescale operations. The proposed architecture targeting Xilinx Alveo U280 FPGA uses 372940 LUTs, 672 DSPs, and 768 BRAM. The homomorphic multiplication and key switch operations are completed in 3091 and 97668 clock cycles, respectively. Operating at a frequency of 181.1MHz, they take 17 μs and 537.1 μs, respectively, without coefficient load operations taken into account. The synthesis results are presented in Table 3. Compared to the software implementation in Microsoft SEAL library running on an AMD Ryzen 7 3800x CPU, we get 15× speedup on homomorphic multiplication, and 4× speedup on key switch operation.

7 Discussion & Future Work

7.1 Discussion

Compared to other studies, our implementations for the CKKS operations have certain limitations. First, while our performance is worse than some other works, on the other hand, we consume fewer FPGA resources. Another challenge was designing a fully pipelined architecture for different ring sizes to achieve high throughput and low latency. Our approach achieves a low latency with no stall cycles in our NTT computation. However, while our NTT design offers flexibility and improved performance for a range of polynomial sizes, it is not without limitations. As we attempt into higher ring sizes, it's anticipated that the BRAM resources may become insufficient. This potential shortfall could necessitate off-chip memory, introducing latency challenges and potential communication bottlenecks in performance. To support these larger ring sizes, expanding the design would not only require additional resources but might also introduce integration complexities. Additionally, our current design needs further optimization regarding energy consumption since energy efficiency remains a critical factor for cryptographic hardware.

7.2 Future Work

Our current design primarily emphasizes accelerating the CKKS scheme, focusing on low latency. While this has led our NTT implementation to be optimized for swift operations, it simultaneously indicates potential areas of improvement, particularly regarding the throughput of the CKKS key switching algorithm. As we chart our future research trajectory, we are contemplating multiple enhancements. Firstly, we are keen on integrating our parametric merged-NTT hardware with other homomorphic encryption schemes to understand its adaptability and broader applications better. Additionally, we want to support the CKKS scheme across different ring sizes. This flexibility would be instrumental in fine-tuning the balance between encryption security, performance, and overall efficiency. We're considering leveraging algorithms such as the 4-step NTT, known for its efficient memory locality utilization, for higher ring sizes that might exceed the onboard FPGA memory. Another avenue we're exploring is augmenting the number of processing elements, aiming to increase the performance further. This approach could lead the way for more resourceful designs adaptable to various FPGA platforms. Lastly, but crucially, energy efficiency remains on our radar. We recognize its significance in cryptographic hardware and are committed to optimizing our future designs for superior performance and reduced energy consumption.

8 Acknowledgement

This paper is partially supported by the European Union's Horizon Europe research and innovation program under grant agreement No: 101079319 and by TUBITAK under Grant Number 118E72.

9 Conclusion

In this paper, we first proposed a parametric merged-NTT hardware and used it to accelerate the homomorphic operations of the CKKS scheme. Our parametric merged-NTT hardware is capable of accommodating hardware resource constraints. Our design provides a trade-off between the performance and the area consumption. The proposed hardware can perform the NTT operation with no stalls between the NTT stages. Compared to the

CKKS implementation in the Microsoft SEAL library, it is shown that our design shows 15× speed-up for homomorphic multiplication and 4× speed-up for key switching.

References

[Bar87] Paul Barrett. Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In Andrew M. Odlyzko, editor, *Advances in Cryptology — CRYPTO' 86*, pages 311–323, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.

[BNAMK21] Mojtaba Bisheh-Niasar, Reza Azarderakhsh, and Mehran Mozaffari-Kermani. High-speed ntt-based polynomial multiplication accelerator for post-quantum cryptography. In *2021 IEEE 28th Symposium on Computer Arithmetic (ARITH)*, pages 94–101, 2021.

[CKKS17] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *International Conference on the Theory and Application of Cryptology and Information Security*, 2017.

[DMGS23] Stefano Di Matteo, Matteo Lo Gerfo, and Sergio Saponara. Vlsi design and fpga implementation of an ntt hardware accelerator for homomorphic seal-embedded library. *IEEE Access*, 11:72498–72508, 2023.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*, STOC '09, page 169–178, New York, NY, USA, 2009. Association for Computing Machinery.

[HZZ+22] Junhao Huang, Jipeng Zhang, Haosong Zhao, Zhe Liu, Ray C. C. Cheung, Çetin Kaya Koç, and Donglong Chen. Improved plantard arithmetic for lattice-based cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(4):614–636, Aug. 2022.

[Mer21] Ahmet Can Mert. *Efficient hardware implementations for lattice-based cryptography primitives*. Phd dissertation, Sabancı University, 2021.

[M19] Ahmet Can Mert, Erdinç Öztürk, and Erkay Savaş. Design and implementation of a fast and scalable ntt-based polynomial multiplier architecture. In *2019 22nd Euromicro Conference on Digital System Design (DSD)*, pages 253–260, 2019.

[POG15] Thomas Pöppelmann, Tobias Oder, and Tim Güneysu. High-performance ideal lattice-based cryptography on 8-bit atxmega microcontrollers. In *Proceedings of the 4th International Conference on Progress in Cryptology – LATINCRYPT 2015 - Volume 9230*, page 346–365, Berlin, Heidelberg, 2015. Springer-Verlag.

[RVM+14] {Sujoy Sinha} Roy, Frederik Vercauteren, Nele Mentens, {Donald Donglong} Chen, and Ingrid Verbauwhede. *Compact Ring-LWE Cryptoprocessor*, pages 371–391. Lecture Notes in Computer Science. Springer, September 2014. 16th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2014) ; Conference date: 23-09-2014 Through 26-09-2014.

[SEA23] Microsoft SEAL (release 4.1). https://github.com/Microsoft/SEAL, January 2023. Microsoft Research, Redmond, WA.

[SRTJ+19] Sujoy Sinha Roy, Furkan Turan, Kimmo Jarvinen, Frederik Vercauteren, and Ingrid Verbauwhede. Fpga-based high-performance parallel architecture for homomorphic computing on encrypted data. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 387–398, 2019.

[SS17] Erdinç Öztürk, Yarkın Doröz, Erkay Savaş, and Berk Sunar. A custom accelerator for homomorphic encryption applications. *IEEE Transactions on Computers*, 66(1):3–16, 2017.