



HMNTT: A Highly Efficient MDC-NTT Architecture for Privacy-preserving Applications

Changxu Liu 
Fudan University
Shanghai, China

Danqing Tang 
Ant Group
Hangzhou, China

Jie Song 
Ant Group
Hangzhou, China

Hao Zhou 
Fudan University
Shanghai, China

Shoumeng Yan 
Ant Group
Hangzhou, China

Fan Yang 
Fudan University
Shanghai, China






ABSTRACT

In privacy-preserving applications like Post-Quantum Cryptography (PQC) and Fully Homomorphic Encryption (FHE), polynomial multiplication is common, and the Number Theoretic Transform (NTT) is a key algorithm for reducing its complexity. In this paper, we present HMNTT, a highly efficient MDC-NTT architecture. Utilizing the four-step NTT algorithm and a pipelined transpose module, HMNTT offers a highly efficient and scalable architecture for handling NTT with large degrees. We optimize the processing element (PE) to alleviate backpressure and data conflicts in data flow. Leveraging FPGA characteristics, we construct a modular multiplication module to reduce resource usage and improve operating frequency. Evaluation results indicate that HMNTT achieves an average of 2.34× and 1.26× reduction in Area-Time Product compared to the latest pipelined NTT architectures.

KEYWORDS

Number theoretic transform, FPGA, Digital circuit design, Privacy-preserving computing.

ACM Reference Format:

Changxu Liu , Danqing Tang , Jie Song , Hao Zhou , Shoumeng Yan , and Fan Yang . 2024. HMNTT: A Highly Efficient MDC-NTT Architecture for Privacy-preserving Applications. In *Great Lakes Symposium on VLSI 2024 (GLSVLSI '24)*, June 12–14, 2024, Clearwater, FL, USA. ACM, New York, NY, USA, 6 pages. <https://doi.org/10.1145/3649476.3658734>

1 INTRODUCTION

The Number Theoretic Transform (NTT) is an efficient optimization algorithm for polynomial multiplication, a variant of the Fast Fourier Transform (FFT). NTT is widely employed in privacy-preserving applications. For instance, polynomial multiplications account for a significant portion of the computation time in Kyber [3], a representative Post-Quantum Cryptography (PQC) algorithm.

This work is conducted by Changxu Liu during his research internship at Ant Group. Corresponding authors: shoumeng.ysm@antgroup.com, yangfan@fudan.edu.cn.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

GLSVLSI '24, June 12–14, 2024, Clearwater, FL, USA

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0605-9/24/06

<https://doi.org/10.1145/3649476.3658734>

In Fully Homomorphic Encryption (FHE) [5], data must be transformed between the limb-wise representation and the coefficient-wise representation for various computation operations, necessitating the use of NTT and inverse NTT (INTT).

Implementing NTT on commonly used processors such as CPUs and GPUs is a widely adopted approach. However, due to the inherent characteristics of these hardware platforms, it becomes challenging to balance performance and overhead. Much effort has been dedicated to developing custom hardware-accelerated NTT implementations. For instance, HEAX [17] leverages multiple processing elements (PEs) to perform the NTT using a parallel computing approach. While such a parallel computing design indeed delivers high performance, it comes with the trade-off of increased hardware overhead and a more intricate circuit layout. Mert *et al.*'s work [14] offers a parameterized tool for specifying the number of PEs and generating the corresponding NTT accelerator. However, as the number of PEs and the polynomial degree increase, the corresponding bandwidth and control logic demands become more intricate. Furthermore, many works focus on optimizing the data access patterns of NTT, which further complicates the issue. Meanwhile, the physical implementation of this style of NTT is challenging.

The NTT based on Single-path Delay Feedback (SDF) [6, 20] and Multi-path Delay Commutator (MDC) [6, 19] exhibits excellent pipeline characteristics. In addition, it is a bandwidth-efficient circuit structure, devoid of overly complex control logic. While SDF-NTT or MDC-NTT offers advantages, backpressure from later PEs to earlier ones can introduce many pipeline bubbles, leading to larger latency—especially evident in SDF-NTT.

We propose a highly efficient MDC-NTT architecture, HMNTT, intended for deployment in FPGA for privacy-preserving applications. HMNTT is designed for practical use, specifically for handling high-degree polynomials. This capability is increasingly in demand due to the growing use of applications focused on privacy-preserving. By employing the four-step NTT algorithm and a pipelined transpose module, we construct a scalable NTT architecture, avoiding the need for deep FIFO/MEM modules and effectively mitigating area consumption. We optimize the PE design for the MDC-NTT, mitigating data collisions and backpressure between each stage. Regarding the design of the modular multiplication module, a critical component of the PE in HMNTT, we carefully consider its adaptability to FPGA and the structure of DSP resources (refer to AMD-Xilinx's FPGA platform). This strategy minimizes resource consumption, enabling our design to operate at a higher frequency.

2 PRELIMINARY

NTT is defined over the polynomial ring $\mathbb{Z}_q[x]/\phi(x)$, where $\mathbb{Z}_q[x]$ represents the ring of integers modulo q , and $\phi(x)$ represents the polynomial modulus. The NTT can convert polynomials in the coefficient domain to the NTT (evaluation) domain. Taking a simple example: NTT can convert two polynomials $a(x) = \sum_{i=0}^{n-1} a_i x_i$ and $b(x) = \sum_{i=0}^{n-1} b_i x_i$ from the coefficient domain into $A(x)$ and $B(x)$ within the NTT domain. After point-wise multiplication of $A(x)$ and $B(x)$, we obtain $C(x)$. The result of $C(x)$ undergoes the INTT, resulting in $c(x)$, which represents the polynomial multiplication result of $a(x)$ and $b(x)$. The algorithmic complexities of NTT/INTT are both $O(n \log n)$, while the algorithmic complexity of point-wise multiplication is $O(n)$. Compared to the schoolbook polynomial multiplication complexity of $O(n^2)$, the utilization of NTT and INTT enhances the efficiency of polynomial multiplication.

A_i can be obtained using the following formula:

$$A_i = \sum_{j=0}^{n-1} a_j g_i^j \text{ mod } q. \quad (1)$$

When g_i is one of n -th primitive roots of modulo q , satisfying $g_i^n \equiv 1$. By exploiting the properties of modular arithmetic and primitive roots, we can unfold the NTT into a butterfly structure, enabling parallel computation. The degree of the NTT is typically padded to a power of 2. Fig. 1 depicts a butterfly structure of 8-pt NTT. There are two commonly used butterfly unit structures: Gentleman-Sanderson (GS) type and Cooley-Tukey (CT) type. In GS-type, modular addition/subtraction is done first, followed by modular multiplication with the twiddle factors. CT-type, on the other hand, performs modular multiplication first, followed by modular addition/subtraction. In terms of hardware implementation, there is a significant latency difference between modular addition/subtraction and modular multiplication. By utilizing the results of modular addition in GS-type NTT in advance, we can enhance overall efficiency. The proposed optimization in our work significantly improves pipelined GS-type NTT. Both types of NTT have the same computational effectiveness.

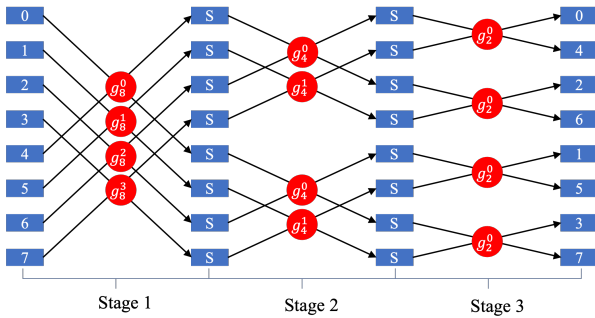


Figure 1: Data flow of 8-pt NTT.

In NTT, polynomial coefficients are extended from n -pt to $2n$ -pt through zero-padding to enable polynomial multiplication over the ring $x^n + 1$. However, this increases computational overhead, reducing NTT efficiency. The Negative Wrapped Convolution (NWC) [13] is introduced to avoid zero-padding, addressing this efficiency concern. Polynomials $a(x)$ and $b(x)$ can be modified before applying NTT by defining $\hat{a}_i = a_i \omega^i$ and $\hat{b}_i = b_i \omega^i$, where ω represents

one of $2n$ -th primitive roots of modulo q . After INTT, result $c(x)$ can be adjusted using $\hat{c}_i = c_i \omega^{-i}$. This allows for coefficient corrections under $x^n + 1$ without zero-padding; However, it comes at the expense of introducing three additional n -pt point-wise multiplications.

3 PROPOSED DESIGN

In this section, we present the details of HMNTT. Firstly, we discuss the modular multiplication module's design. Here, we construct the modular multiplication module based on the resources and characteristics of the FPGA to achieve maximum resource utilization efficiency and operating frequency. Next, we introduce our improvements to the Processing Element (PE) in the MDC-NTT, effectively mitigating data collisions and backpressure between each stage of PEs. Finally, we present the overall architecture of HMNTT, which applies a four-step NTT algorithm. With the pipelined transpose module, HMNTT allows flexible configuration of the number of PEs, enhancing the trade-off between performance and cost across diverse scenarios.

3.1 Modular Multiplication Module

Modular multiplication is a fundamental computational module in NTT, comprising two key components: a large-number multiplier and a modular reduction module.

In a large-number multiplier, the large numbers are decomposed into sizes that closely match the bit-width of DSPs on the FPGA, and multiple DSPs are utilized to perform the multiplication. We employ the multiple layers Karatsuba method [9], which uses a small number of additions to reduce the required multiplications. Considering the structure of DSPs [8], which includes multiple configurable adders, we incorporate some of the additions and subtractions from the Karatsuba method into the DSPs. This approach allows us to utilize the internal registers of the DSPs, saving a significant amount of LUTs and FFs. Furthermore, the large-number multiplier can achieve higher operating frequency with dedicated routing available in the DSPs.

The commonly used modular reduction methods include Barrett reduction [2] and Montgomery reduction [15]. Inspired by work [6], the NTT-friendly Montgomery reduction method is highly suitable for FPGA platforms, as it consumes significantly fewer DSP resources than Barrett reduction while remaining DSP-friendly. We adopt the Montgomery reduction method proposed in [6], which results in a modular reduction module consisting primarily of DSPs. This approach saves logic resources and improves operating frequency.

3.2 PE in HMNTT

In HMNTT, the Processing Element (PE) comprises a butterfly computation logic and a data permutation logic, as illustrated in Fig. 2. The butterfly computation logic, constructed with GS-type NTT, is represented by the blue section, while the red FIFOs and green MUXs represent the data permutation logic. Stride in NTT refers to the interval step between elements within each stage. For instance, in Fig. 1, the stride value is 4 for Stage 1, 2 for Stage 2, and 1 for Stage 3.

In the butterfly computation logic, the computational delay is smaller in the modular addition/subtraction module compared to

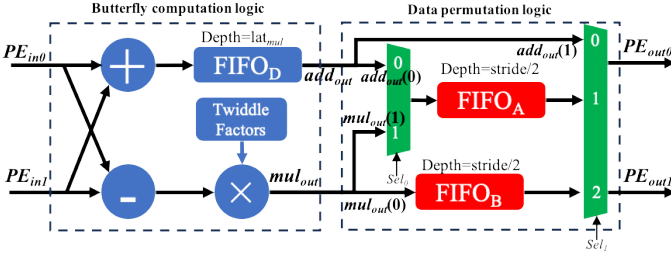


Figure 2: Microarchitecture of PE in HMNTT. The blue circle with a ‘+’ symbol represents the modular addition module, the blue circle with a ‘-’ symbol represents the modular subtraction module, and another one with an ‘x’ symbol represents the modular multiplication module. The lat_{mul} represents the latency of the modular multiplication module. We divide the add_{out} and mul_{out} into two groups, splitting them in half in terms of time. The first half is denoted with the suffix “(0)”, while the second half is denoted with the suffix “(1)”.

the modular multiplication module, as the former has a relatively simpler nature. To maintain the pipeline characteristics of the PE and synchronize them with the output of the modular multiplication module, it is essential to introduce a delay unit before feeding them into the subsequent data permutation logic. In previous work, conventional delay units were employed [6, 20], or certain details were omitted [16, 19]. In our approach, we utilize a FIFO, $FIFO_D$ in Fig. 2, to implement data delay logic. This enhances the data access pattern and optimizes the data flow for NTT.

Specifically, when the data (PE_{in0} and PE_{in1}) enters the PE, it is initially aligned for entry into the butterfly computation logic. In the previous handling, it was essential to align the outputs to the data permutation logic. We refer to the outputs as add_{out} and mul_{out} , corresponding respectively to the delayed output of the modular addition module and the output of the modular multiplication module. The first half of add_{out} and mul_{out} , denoted as $add_{out}(0)$ and $mul_{out}(0)$, will be directly written into $FIFO_A$ and $FIFO_B$, in the data permutation logic, with Sel_0 set to 0. Subsequently, the latter half of add_{out} , $add_{out}(1)$, bypasses to the PE’s output, denoted as PE_{out1} , while the $FIFO_A$ storing $add_{out}(0)$ is simultaneously read, labeled as PE_{out0} . The latter half of mul_{out} , $mul_{out}(1)$, is concurrently written into $FIFO_A$. At this point, the Sel_0 is set to 1, while the Sel_1 is set to 0 and 1.

Once $add_{out}(0)$ and $add_{out}(1)$ are fully passed through into the PE in the next stage, $FIFO_A$ and $FIFO_B$ are read, i.e., $mul_{out}(1)$ and $mul_{out}(0)$ are routed to the subsequent PE, referred to as PE_{out1} and PE_{out0} , respectively. Simultaneously, the new add_{out} and mul_{out} can be stored in $FIFO_A$ and $FIFO_B$. At this point, the Sel_1 is set to 1 and 2, while Sel_0 is set to 0, to select the new $add_{out}(0)$ for writing into $FIFO_A$.

When using $FIFO_D$ instead of conventional delay units, add_{out} can enter the data permutation logic before mul_{out} , allowing it to be processed and output to the next stage ahead of time. As a result, when mul_{out} is pipelined, add_{out} has already been processed by the data permutation logic, requiring a shorter wait time or potentially entering the data permutation logic without waiting. This leads to faster processing speeds. Meanwhile, FIFOs have a queue depth. When the subsequent pipeline stalls, add_{out} can be stored in $FIFO_D$

instead of causing backpressure on the preceding PE, preventing the generation of numerous bubbles in the pipeline for the modular multiplication module. The depth of $FIFO_D$ is consistent with the depth of the previous delay units, as shown in Fig. 2. The difference is that it utilizes a few additional control signals to indicate the empty and full state of the $FIFO_D$. Additionally, at the interface, PE_{out0} is augmented with a pair of handshake signals to ensure compactness and correctness of the timing. By adding a minimal amount of control logic, we achieve significant optimization benefits as demonstrated in the data flow shown in Fig. 3.

Fig. 3 demonstrates the optimization effect using a demo. The ‘Before Improvement’ represents the data access pattern of the PE before optimization, while the ‘After Improvement’ illustrates the optimized data access pattern. The demo showcases that our optimization accelerates data processing and alleviates the backpressure from the subsequent PE to the preceding PE. Considering an n -pt NTT, our optimization allows each stage to reduce latency by half of the stride, with the stride starting at $\frac{n}{2}$ and decreasing exponentially at each stage, leading to a total optimization of $\lfloor \frac{1}{4} \sum_{i=0}^{\log_2 n} \frac{n}{2^i} \rfloor = \frac{n}{2} - 1$ cc. This is achieved by leveraging the delay in the modular multiplication module to advance the add_{out} , thus mitigating data conflicts. Additionally, during the 9 cc to 11 cc in Fig. 3, there are pipeline bubbles in the output of the optimized PE. This is because our demo starts with initialization, which introduces bubbles in the pipeline. However, once the pipeline is fully operational, it operates efficiently without any bubbles, as shown in the timeframe from 19 cc to 20 cc in Fig. 3. Furthermore, by replacing the delay units with $FIFO_D$, we have gained significant flexibility in the timing of the add_{out} . This allows the add_{out} to be output based on the occupancy of the $FIFO_D$ in data permutation logic, as demonstrated in the 9 cc to 11 cc timeframe in Fig. 3.

3.3 Flexible Architecture of HMNTT

We first introduce the four-step NTT employed by HMNTT, as illustrated in Alg. 1. We make slight adjustments to the four-step NTT algorithm by using bit-reversal of subsize NTT, achieving NTT mapping from natural order to natural order. BR in Alg. 1 denotes the bit-reversal permutation, and the subscripts indicate the dimensions of the corresponding operations. In the preprocessing step of the four-step NTT algorithm, the coefficients in natural order undergo a reshaping process from a one-dimensional layout to a two-dimensional array of size $I \times J$ (when $I * J = N$), as illustrated in Fig. 4 (Coeff Reshape). I -size NTT is performed separately for each column, a total of J times as shown in Loop 1 in Alg. 1. Then, the results of Loop 1 are multiplied by the twiddle factors. The power of the twiddle factor is determined by the bit reverse of the row index, i , and column index, j , of the corresponding element in the two-dimensional array, as illustrated in Loop 2. The output results also need to undergo bit reversal. Next, a J -size NTT is performed for each row of the original two-dimensional array for I times, as shown in Loop 3. The output results are once again subjected to bit reversal in column style and unfolded into a one-dimensional array to obtain the evaluation representations in the natural order, as shown in Fig. 4 (Eval Reshape).

Through Alg. 1, we establish the flexible architecture of HMNTT, as depicted in Fig. 5. We cascade the PEs to create TinyNTT capable of handling the I -size NTT/ J -size NTT. Each level of PE is

Time/cc	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27										
PE_{in0}	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	...																					
PE_{in1}	8	9	10	11	12	13	14	15	8	9	10	11	12	13	14	15	...																					
Before Improvement																																						
add_{out}	...							0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	...														
mul_{out}	...							8	9	10	11	12	13	14	15	8	9	10	11	12	13	14	15	...														
PE_{out0}	...							0	1	2	3	8	9	10	11	0	1	2	3	8	9	10	11	8	9	10	11	...										
PE_{out1}	The flexibility provided by FIFO _n							4	5	6	7	12	13	14	15	4	5	6	7	12	13	14	15	12	13	14	15	...										
After Improvement																																						
add_{out}	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	... Stride/2																					
mul_{out}	...							8	9	10	11	12	13	14	15	8	9	10	11	12	13	14	15	...														
PE_{out0}	...							0	1	2	3	8	9	10	11	0	1	2	3	8	9	10	11	...														
PE_{out1}	...							4	5	6	7	12	13	14	15	4	5	6	7	12	13	14	15	...														

Figure 3: Optimized Data Access Pattern for HMNTT: A 16-pt NTT Demo. The figure shows the first stage of a 16-pt NTT with a stride of 8. Assuming a modular multiplication module latency of 8 clock cycles (cc) and modular addition module latency of 1 cc to emphasize their delay disparity, the optimized data access pattern allows for data output to be advanced by 4 cc (half the stride).

Algorithm 1 Four-step NTT

- 1: **Input:** $coeff_{1 \times N}$ (natural order).
- 2: **Coeff Reshape:** $coeff_{1 \times N} \rightarrow coeff_{I \times J}$
- 3: **for** j **in range** (J) **do** ▷ Loop 1
- 4: $coeff_{I \times J} = NTT_{I-pt}(coeff[0 \rightarrow I][j])$ ▷ Step 1: I -size NTT.
- 5: **for** j **in range** (J) **do** ▷ Loop 2
- 6: **for** i **in range** (I) **do**
- 7: $i' = \text{Bit reverse}(i)$
- 8: $coeff_{I \times J}[i][j] = coeff_{I \times J}[i][j] * g_N^{i' * j}$ ▷ Step 2:
- 9: Multiplying by twiddle factors.
- 9: **BR_I** ($coeff_{I \times J}$) ▷ Step 3: Bit reversal in the I -size NTT.
- 10: **for** i **in range** (I) **do** ▷ Loop 3
- 11: $eval_{I \times J} = NTT_{J-pt}(coeff[i][0 \rightarrow J])$ ▷ Step 4: J -size NTT.
- 12: **BR_J** ($eval_{I \times J}$) ▷ Step 5: Bit reversal in the J -size NTT.
- 13: **Eval Reshape:** $eval_{I \times J} \rightarrow eval_{1 \times N}$.
- 14: **Output:** $eval_{1 \times N}$ (natural order).

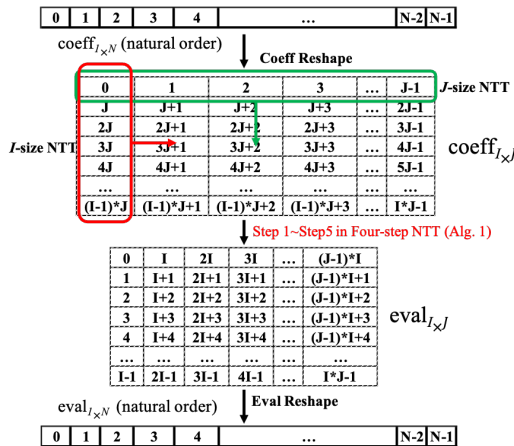


Figure 4: The reshaping process of elements in the four-step NTT algorithm. The letters/numbers in the figure represent the indices of the elements.

responsible for completing a stage outlined in Fig. 1. The final level of PE requires only a modular addition module and a modular subtraction module, as the twiddle factor is 1. The number of TinyNTT modules in HMNTT is configurable, allowing for faster or more area-efficient NTT. This configuration is analogous to partially unrolling the three loops in Alg. 1. Meanwhile, the results of the I -size NTT enter the Twisting Module for modular multiplication with the twiddle factors, as indicated in Loop 2 in Alg. 1. Since HMNTT is pipelined, the Twisting Module is primarily composed of a set of modular multiplication modules equivalent to the number of the TinyNTT modules, along with a memory structure for twiddle factors. Moreover, the Twisting Module does not affect the overall latency. We store twiddle factors in advance due to their relatively constant nature during calculations. One optimization involves the on-the-fly (OF) generation of twiddle factors during computation to reduce on-chip storage. However, introducing new modular multiplication modules for OF calculations can increase the area consumption. In our benchmark, reusing modular multiplication modules in the Twisting module leads to about 33% extra latency with no significant area reduction, which is not cost-effective for ATP. OF should be considered when facing limited on-chip storage or NTT with much larger degrees.

However, some performance-degrading issues may arise once there is more than one TinyNTT module in HMNTT. Assuming the original data access of the NTT is in I banks of SRAM or J banks of SRAM, the distinct input dimensions required by I -size NTT and J -size NTT inevitably lead to bottlenecks in data read/write due to the limitations on the number of SRAM read/write ports. This bottleneck results in stalls and introduces significant bubbles in the pipeline, thus compromising performance. To address this issue, we introduce a pipelined transpose module to switch the data access dimensions between I -size NTT and J -size NTT. We leverage the low bandwidth requirements of the pipelined NTT and implement data transposition using a limited number of registers. As illustrated in Fig. 5, data enters along the green (orange) arrows, propagates to the right (down), and when filled, proceeds downward (to the right) along the orange (green) arrow, serving as input to the TinyNTT. Simultaneously, new data can enter the transpose module along the orange (green) arrows and, once filled, change direction for output.

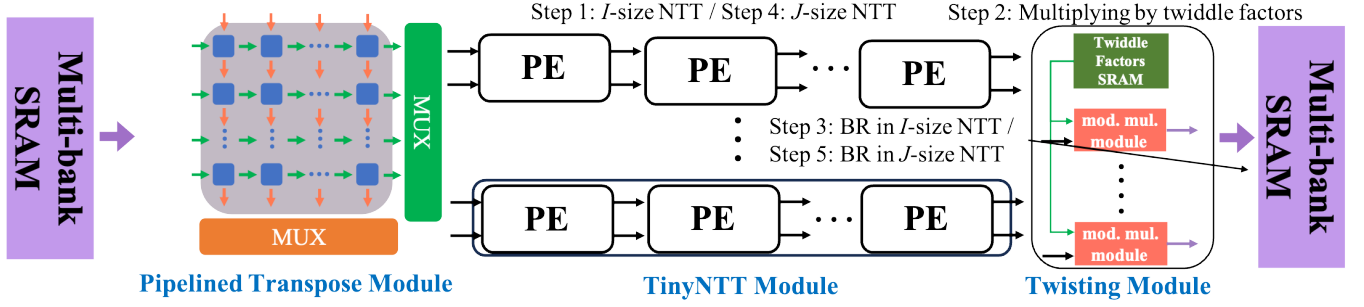


Figure 5: Architecture of HMNTT. Elements are accessed and stored in Multi-bank SRAM in the form of a two-dimensional array, as depicted in Fig. 4. TinyNTT performs I -size NTT and J -size NTT. The number of PEs needed in one TinyNTT is determined by $\log_2(\max(I, J))$. To maintain compatibility for $\min(I, J)$ -size NTT, we can bypass the preceding PEs. The Twisting Module handles Step 3 in Alg. 1, which involves multiplying intermediate results with twiddle factors. HMNTT computes the address for writing data into Multi-bank SRAM based on the bit-reversal rule.

The transpose unit is fully pipelined with registers exchanging data with adjacent ones and registers on the output side using 2-to-1 MUXs for selection. It efficiently addresses the previously mentioned performance degradation issues with minimal resource overhead.

4 EVALUATION

We conduct a comprehensive evaluation of HMNTT. Our proposed design is evaluated using Verilog HDL on an AMD-Xilinx XC7VX690T FPGA, and synthesis and implementation are carried out using AMD-Xilinx Vivado 2022.1 tools. For our benchmark, we choose I and J as the powers of 2. TinyNTT is designed to handle $\max(I, J)$ -size NTT. By selecting close values of I and J (Identical is better), we optimize the computational efficiency of the PEs in HMNTT while minimizing the number of PEs required and the twiddle factors usage. The number of twiddle factors used in the Twisting module is unaffected. For instance, with a 1024-pt NTT, selecting $I=J=32$ requires only 5 PEs for TinyNTT, whereas choosing $I=128$ and $J=8$ would require 7 PEs, with the first 4 PEs remaining idle during J -size NTT.

In Tab. 1, we present a performance comparison between HMNTT with existing works [1, 4, 6, 10–12, 14, 18, 20]. Given that HMNTT is designed for privacy-preserving applications such as RNS-FHE and PQC algorithms, where the bit width is typically constrained to 64 bits or less, we benchmark three scenarios: $N = 1024$, $\log_2 q = 28$; $N = 4096$, $\log_2 q = 28$; and $N = 4096$, $\log_2 q = 64$. We provide evaluations for HMNTT_v1 (with one TinyNTT module) and HMNTT_v2 (with two TinyNTT modules) under each benchmark.

We provide a detailed report of the platforms, latency/clock cycles, and resource consumption for all works. Given the variability in the resources utilized by each work, we normalize all resources using the formula referred to in [20], as outlined in Tab. 1 under the ‘Area’ metric. Furthermore, we employ the Area-Time Product (ATP) metric, comprehensively considering latency and resource consumption. A smaller ATP value is preferable. Works [1, 7, 10, 11] utilize AMD-Xilinx’s advanced FPGA platforms, featuring more advanced process technology and enhanced DSP for reduced resource consumption and smaller latency. Compared to works [20] and [6], both of which represent the latest works employing pipelined NTT

architectures, HMNTT achieves an average of $2.34\times$ fewer and $1.26\times$ fewer ATP when considering both configurations. Results demonstrate a significant ATP advantage for HMNTT compared to other works. Work [6] achieves low clock cycles and resource usage. However, it underperforms in terms of ATP compared to our design. This is because this work prioritizes minimizing clock cycles, neglecting FPGA adaptability and large-degree NTT applicability in a pipeline architecture. HMNTT employs the four-step NTT algorithm to eliminate the requirement for deep FIFOs. It optimizes the PE by replacing the conventional delay unit with FIFO and introducing additional handshake signals and registers to enhance its performance. Despite the increased clock cycles introduced by the four-step NTT algorithm for data access, the optimization in PE partially mitigates this impact. These techniques result in better timing and higher FPGA efficiency. Consequently, HMNTT outperforms work [6] in ATP metrics. Optimized PE design and the use of the four-step NTT algorithm, which is well-suited for pipelined NTT architectures, minimizes resource usage and enables higher operating frequency. However, this comes with increased BRAM storage for twiddle factors and intermediate results. Nevertheless, the BRAM consumption in our design remains relatively low, significantly below the BRAM consumption in other works. Across the three benchmarks, our design achieves an average savings of $3.6\times$, $2.8\times$, and $4.7\times$ in terms of BRAM consumption. With lower bandwidth requirements in pipelined NTT architectures, HMNTT can efficiently compute bit-reversal data indices during data output, eliminating the need for reorder operations on BRAM-stored data.

5 CONCLUSION

In this paper, we propose a highly efficient MDC-NTT architecture, HMNTT. Built on the four-step NTT algorithm, HMNTT is well-suited for privacy-preserving applications involving polynomials with large degrees, showcasing remarkable scalability. We optimize PE in HMNTT to reduce data collision and alleviate backpressure in the data flow. In practical FPGA implementation, we consider device characteristics, conserving resources, and improving operating frequency. The evaluation findings indicate that HMNTT achieves an average $2.34\times$ and $1.26\times$ reduction in ATP when compared to the latest pipelined NTT architectures.

Work	Plat.	Param. ($N, \log_2 q$)	Freq. (MHz)	Resource				Area ¹	Lat.(μ s) / Clock Cycles	ATP
				LUT	FF	DSP	BRAM			
Ye <i>et al.</i> [20]	Virtex-7	1024, 28-bit	175	3.4k	3.1k	63	6	11.5k	6/2114	69.0k (2.49 \times)
Hirner <i>et al.</i> [6]	Virtex-7	1024, 28-bit	150	6.9k	3.2k	36	2	11.1k	3.46/518	38.4k (1.39 \times)
Mert <i>et al.</i> [14]	Virtex-7	1024, 28-bit	125	16k	14k	56	24	28.8k	3.9/490	112.3k (4.05 \times)
Geng <i>et al.</i> [4]	Virtex-7	1024, 32-bit	244	9.5k	4.7k	64	12	19.5k	2.67/652	52.1k (1.88 \times)
Su <i>et al.</i> [18]	Virtex-7	1024, 32-bit	250	10.3k	6.7k	80	79	42k	2.6/-	109.2k (3.94 \times)
Li <i>et al.</i> [11]	XCKU060	1024, 32-bit	190	25.6k	16.5k	48	28.5	39.0k	1.81/343	70.6k (2.55 \times)
HMNTT_v1 ²	Virtex-7	1024, 28-bit	332	3.2k	4.0k	36	3	7.7k	3.6/1202	27.7k (1.00 \times)
HMNTT_v2 ³	Virtex-7	1024, 28-bit	319	6.7k	8.6k	72	6	15.7k	2.2/690	34.5k (1.25 \times)
Hirner <i>et al.</i> [6]	Virtex-7	4096, 28-bit	150	8.4k	4.0k	44	8	15.2k	13.8/2069	209.8k (1.44 \times)
Hu <i>et al.</i> [7]	UltraScale+	4096, 28-bit	400	5.9k	4.5k	12	8	9.5k	15.4/6158	146.3k (1.01 \times)
Ayduman <i>et al.</i> [1]	AU280	4096, 32-bit	181	29.8k	21.4k	224	48	66.6k	4.3/792	286.4k (2.00 \times)
Geng <i>et al.</i> [4]	Virtex-7	4096, 32-bit	244	5k	2.8k	32	14	12.4k	25.2/6156	312.5k (2.15 \times)
Su <i>et al.</i> [18]	Virtex-7	4096, 32-bit	250	14k	8.7k	80	79	45.7k	12.3/-	562.1k (3.87 \times)
HMNTT_v1 ²	Virtex-7	4096, 28-bit	329	3.9k	4.9k	42	12	11.7k	13.2/4336	154.4k (1.06 \times)
HMNTT_v2 ³	Virtex-7	4096, 28-bit	317	8.2k	10.3k	84	12	20.2k	7.2/2290	145.4k (1.00 \times)
Ye <i>et al.</i> [20]	Virtex-7	4096, 60-bit	150	17k	11k	286	24.5	53.0k	27.5/8284	1457.5k (2.61 \times)
Kurniawan <i>et al.</i> [10]	UltraScale+	4096, 60-bit	250	74.5k	61.4k	288	155	149.8k	3.8/951	569.2k (1.02 \times)
Mert <i>et al.</i> [14]	Virtex-7	4096, 60-bit	125	22k	17k	248	96	75.6k	26/3276	1965.6k (3.52 \times)
Hu <i>et al.</i> [7]	UltraScale+	4096, 60-bit	152	17.2k	17.5k	144	48	46k	20.37/3086	937.0k (1.68 \times)
Liu <i>et al.</i> [12]	Virtex-7	4096, 60-bit	150	14.1k	12.5k	336	41	60k	20.5/3081	1230k (2.20 \times)
Hirner <i>et al.</i> [6]	Virtex-7	4096, 64-bit	150	22.6k	18.0k	220	16	49.4k	13.8/2069	681.7k (1.22 \times)
HMNTT_v1 ²	Virtex-7	4096, 64-bit	241	9.2k	12.6k	133	24	29.7k	18.8/4536	558.4k (1.00 \times)
HMNTT_v2 ³	Virtex-7	4096, 64-bit	211	18.9k	26.7k	266	24	52.7k	11.8/2490	621.9k (1.11 \times)

¹ The Area is measured by #LUTs+100 \times #DSPs+300 \times #BRAMs [20]. ² Use one TinyNTT module. ³ Use two TinyNTT modules.

Table 1: Evaluation Results and Comparison with Previous Works.

ACKNOWLEDGMENTS

This work was supported by National Key R&D Program of China 2023YFB2704600, partly by Ant Group Research Intern Program, partly by National Natural Science Foundation of China (NSFC) research projects 92373207 and 62090025.

REFERENCES

- [1] Can Ayduman, Emre Koçer, Selim Kirbiyık, Ahmet Can Mert, and ErKay Savaş. 2023. Efficient Design-Time Flexible Hardware Architecture for Accelerating Homomorphic Encryption. In *2023 IFIP/IEEE 31st International Conference on Very Large Scale Integration (VLSI-SoC)*. 1–7.
- [2] Paul Barrett. 1986. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In *Conference on the Theory and Application of Cryptographic Techniques*. Springer, 311–323.
- [3] Joppe Bos, Leo Ducas, Eike Kiltz, T Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehle. 2018. CRYSTALS - Kyber: A CCA-Secure Module-Lattice-Based KEM. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. 353–367. <https://doi.org/10.1109/EuroSP.2018.00032>
- [4] Yue Geng, Xiao Hu, Minghao Li, and Zhongfeng Wang. 2023. Rethinking Parallel Memory Access Pattern in Number Theoretic Transform Design. *IEEE Transactions on Circuits and Systems II: Express Briefs* 70, 5 (2023), 1689–1693. <https://doi.org/10.1109/TCSII.2023.3260811>
- [5] Craig Gentry. 2009. *A fully homomorphic encryption scheme*. Stanford university.
- [6] Florian Hirner, Ahmet Can Mert, and Sujoy Sinha Roy. 2023. PROTEUS: A Tool to generate pipelined Number Theoretic Transform Architectures for FHE and ZKP applications. *Cryptology ePrint Archive* (2023).
- [7] Xiao Hu, Jing Tian, Minghao Li, and Zhongfeng Wang. 2023. AC-PM: An Area-Efficient and Configurable Polynomial Multiplier for Lattice Based Cryptography. *IEEE Transactions on Circuits and Systems I: Regular Papers* 70, 2 (2023), 719–732.
- [8] Xilinx Inc. 2018. *7Series DSP48E1 Slice User Guide*. Technical Report UG479 (v1.10). Xilinx Inc., San Jose, CA. https://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf Accessed: February 6, 2024.
- [9] Anatolii Alekseevich Karatsuba and Yu P Ofman. 1962. Multiplication of many-digit numbers by automatic computers. In *Doklady Akademii Nauk*, Vol. 145. Russian Academy of Sciences, 293–294.
- [10] Stefanus Kurniawan, Phap Duong-Ngoc, and Hanho Lee. 2023. Configurable Memory-Based NTT Architecture for Homomorphic Encryption. *IEEE Transactions on Circuits and Systems II: Express Briefs* 70, 10 (2023), 3942–3946. <https://doi.org/10.1109/TCSII.2023.3289489>
- [11] Bin Li, Yunfei Yan, Yuanxin Wei, and Heru Han. 2024. Scalable and Parallel Optimization of the Number Theoretic Transform Based on FPGA. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 32, 2 (2024), 291–304.
- [12] Si-Huang Liu, Chia-Yi Kuo, Yan-Nan Mo, and Tao Su. 2023. An Area-Efficient, Conflict-Free, and Configurable Architecture for Accelerating NTT/INTT. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2023), 1–11.
- [13] Vadim Lyubashevsky, Daniele Micciancio, Chris Peikert, and Alon Rosen. 2008. SWIFFT: A Modest Proposal for FFT Hashing. In *Fast Software Encryption, 15th International Workshop, FSE 2008, Lausanne, Switzerland, February 10–13, 2008, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 5086)*. Springer, 54–72. https://doi.org/10.1007/978-3-540-71039-4_4
- [14] Ahmet Can Mert, Emre Karabulut, Erdiç Öztürk, ErKay Savaş, Michela Becchi, and Aydın Aysu. 2020. 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)*. 346–351. <https://doi.org/10.23919/DATE48585.2020.9116470>
- [15] Peter L Montgomery. 1985. Modular multiplication without trial division. *Mathematics of computation* 44, 170 (1985), 519–521.
- [16] Ziyang Ni, Ayesha Khalid, Weiqiang Liu, and Maire O'Neill. 2023. Towards a Lightweight CRYSTALS-Kyber in FPGAs: an Ultra-lightweight BRAM-free NTT Core. In *2023 IEEE International Symposium on Circuits and Systems (ISCAS)*. 1–5.
- [17] M. Sadegh Riazzi, Kim Laine, Blake Pelton, and Wei Dai. 2020. 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 (Lausanne, Switzerland) (ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 1295–1309.
- [18] Yang Su, Bai-Long Yang, Chen Yang, Ze-Peng Yang, and Yi-Wei Liu. 2022. A Highly Unified Reconfigurable Multicore Architecture to Speed Up NTT/INTT for Homomorphic Polynomial Multiplication. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 30, 8 (2022), 993–1006.
- [19] Weihang Tan, Antian Wang, Yingjie Lao, Xinmiao Zhang, and Keshab K. Parhi. 2021. Pipelined High-Throughput NTT Architecture for Lattice-Based Cryptography. In *2021 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*. 1–4. <https://doi.org/10.1109/AsianHOST53231.2021.9699608>
- [20] Zewen Ye, Ray C. C. Cheung, and Kejie Huang. 2022. PipeNTT: A Pipelined Number Theoretic Transform Architecture. *IEEE Transactions on Circuits and Systems II: Express Briefs* 69, 10 (2022), 4068–4072. <https://doi.org/10.1109/TCSII.2022.3184703>