High-Throughput FPGA Acceleration of the Number Theoretic Transform for FV Homomorphic Encryption

Benjamin Steeg
CES
KIT
Karlsruhe, Germany
benjamin.steeg@student.kit.edu

Zahra Mojtahedin
Sharif University
Tehran, Iran
zahra.mojtahedin@sharif.edu

Hassan Nassar
CES
KIT
Karlsruhe, Germany
nassar@kit.edu

Joerg Henkel
CES
KIT
Karlsruhe, Germany
henkel@informatik.uni-karlsruhe.de

Abstract

Employing cloud computing resources generally requires trusting the provider to ensure data security. For applications that handle strictly confidential information, such as medical records, this trust may not always be feasible. Homomorphic Encryption (HE) offers a solution by allowing computations to be performed directly on ciphertext rather than plaintext. This way, cloud services can be used without exposing sensitive data, since the servers never decrypt the information they process. However, HE remains highly computationally intensive. While recent schemes have improved performance, the overhead is still substantial.

In this paper, we present an FPGA-based accelerator for polynomial multiplication in the FV HE scheme using the NTT. Our primary objective is to maximize performance. Building on existing research, we design a parallel NTT algorithm and realize it in a novel hardware architecture. Several configurations of our design are implemented and tested on an AMD Xilinx ZCU102 evaluation board. Results are evaluated against both a software implementation on CPU and related work. Our fastest configuration outperformed the fastest CPU, an Apple M1, by 6.9x, while even the slowest configuration achieved a 3.2x speedup. Compared to related work, our accelerator demonstrated performance improvements in several cases, though often at the cost of higher resource utilization.

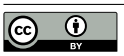
CCS Concepts

• **Security and privacy → Cryptography**; **Privacy-preserving protocols**; **Hardware security implementation**.

Keywords

FPGA, Security, Homomorphic Encryption

ACM Reference Format:
Benjamin Steeg, Zahra Mojtahedin, Hassan Nassar, and Joerg Henkel. 2026. High-Throughput FPGA Acceleration of the Number Theoretic Transform for FV Homomorphic Encryption. In *Proceedings of the 16th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies (HEART 2026), June 17–19, 2026, Heidelberg, Germany*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3814576.3814577



HEART 2026, Heidelberg, Germany

ACM ISBN 979-8-4007-2452-7/26/06
https://doi.org/10.1145/3814576.3814577

1 Introduction

Imagining modern technology without cloud computing is nearly impossible. The platforms that shape today's internet, such as social media and entertainment streaming services, are after all simply applications running on vast cloud infrastructures. In general, cloud computing allows its users to access massive computational power from low-power devices, like smartphones or laptops, while also offering convenient storage solutions for data that must be available across multiple machines. This, however, introduces the challenge of data security. Once data leaves a device and is transmitted to a cloud server for processing, it becomes vulnerable to various attacks aimed at compromising its integrity and confidentiality [2, 13]. Securing the transmission path from the device to the cloud is relatively straightforward, as cryptography is a highly active area of research and numerous well-established encryption algorithms are available [11, 26]. However, even when the communication path is secured, the data must still be decrypted at the server in order for services to be executed [3, 14]. This requires the user to place trust in the cloud provider, which may not always be feasible if highly confidential data is processed. Given the frequency at which data breaches occur, a certain level of distrust is certainly justified. For instance, in 2021 Facebook exposed the data of more than 500 million users, including full names, phone numbers, and location data [15]. In 2024, a breach at UnitedHealth compromised highly sensitive information such as medical and financial records of over 190 million people [1]. Homomorphic encryption offers a solution by enabling computations to be performed on encrypted data directly, without requiring decryption. Thus, clients can use cloud-based services without exposing their data, as the server processes only ciphertext and never knows the plaintext information.

The range of possible applications for this technology is extensive. Notable examples include: (i) Performing machine learning

This work was partially funded by the German Federal Ministry of Research Technology and Space (BMFTR) through grant 01IS23066 as part of the Software Campus Projects "HE-Trust"

tasks, including both training and inference, on encrypted data [12]. (ii) Predictive analysis on medical data, like computing patients risk of suffering from cardiovascular disease, while preserving patient privacy [8]. (iii) Predicting energy consumption through smart meters while preserving the privacy of consumer data [7, 9]. (iv) Electronic voting systems, which reduce time and cost effort over traditional polling, as well as fraud and corruption, and increase voter participation [16].

Unfortunately, HE is highly computationally intensive. While modern schemes have drastically improved in performance, they remain slow [21, 22, 29] in comparison to performing decryption, the operation and subsequent encryption. To address this, researchers have explored software optimizations [27] and leveraged off-the-shelf hardware such as GPUs [5]. In addition, a considerable amount of work has focused on building custom hardware accelerators [25, 28, 29]. FPGAs are a very good candidate as they offer reconfigurability and flexibility [23] to build such accelerators. Closer inspection reveals that polynomial multiplication is largely responsible for this performance deficit. To speed up this operation, most hardware projects implement the Number Theoretic Transform (NTT), as it provides a significant increase in performance over the trivial approach. *Our contributions* are as follows

- We design and implement an FPGA accelerator for *negacyclic polynomial multiplication* in FV/BFV by accelerating the NTT/INTT pipeline (two forward transforms, pointwise multiplication, and inverse transform).
- We introduce a *packed-coefficient* NTT formulation tailored to hardware, reducing the per-butterfly memory traffic from two reads/two writes to one read/one write and enabling higher arithmetic utilization.
- We propose a *multi-core parallelization and scheduling scheme* that maps packed coefficients across C NTT cores and avoids memory-write conflicts as the stride decreases across stages.

This paper is organized as follows. Section 2 introduces homomorphic encryption formally and provides an overview of the encryption scheme we target, as well as the mathematical background for the NTT. Section 3 presents the algorithm used in our accelerator and the reasoning behind it. Section 4 describes our hardware architecture. Section 5 compares our accelerator against a software implementation and related research. Finally, section 6 concludes the paper.

2 Preliminaries

This section briefly summarizes the HE setting targeted in this work and the polynomial arithmetic primitive that dominates its runtime. We focus on the FV/BFV family [10] and the negacyclic Number Theoretic Transform (NTT), which enables fast polynomial multiplication and is the main computation accelerated by our hardware.

2.1 Homomorphic Encryption and FV/BFV

Homomorphic Encryption (HE) enables computations to be carried out on ciphertexts such that, after decryption, the result matches the corresponding plaintext computation [35]. Modern HE schemes support at least homomorphic addition and multiplication, allowing the evaluation of rich arithmetic circuits. In lattice-based HE, including FV/BFV [10], ciphertext operations are defined over polynomial rings, and the dominant cost of multiplication-heavy workloads is polynomial multiplication (and closely related routines such as relinearization and key switching), which internally require multiple polynomial products [25, 35]. Consequently, accelerating polynomial multiplication is key to improving end-to-end HE performance.

2.2 Negacyclic Polynomial Multiplication via NTT

FV/BFV commonly operates in the negacyclic ring $\mathbb{Z}_q[X]/(X^n+1)$ with n a power of two. For efficient multiplication, implementations use an NTT defined over $\mathbb{Z}_q$ (a DFT analogue in a finite field) [17, 24]. Let q be a prime such that $q \equiv 1 \pmod{2n}$, so a primitive $2n$-th root of unity $\psi_{2n} \in \mathbb{Z}_q$ exists. Define $\omega = \psi_{2n}^2$ (a primitive n-th root) and the vectors $\psi = (1, \psi_{2n}, \ldots, \psi_{2n}^{n-1})$ and $\psi^{-1} = (1, \psi_{2n}^{-1}, \ldots, \psi_{2n}^{-(n-1)})$. The negacyclic NTT of $a \in \mathbb{Z}_q[X]/(X^n+1)$ is computed by a pointwise pre-/post-processing around a standard n-point NTT [17]:

$$\widehat{a} = \mathrm{NTT}^{\psi}(a) = \mathrm{NTT}(\psi \circ a), \qquad a = \mathrm{INTT}^{\psi^{-1}}(\widehat{a}) = \psi^{-1} \circ \mathrm{INTT}(\widehat{a}), \tag{1}$$

where $\circ$ denotes coefficient-wise multiplication. This transform provides the negative wrapped convolution property [4], enabling negacyclic multiplication $c = a \cdot b \bmod (X^n+1)$ via:

$$c = \mathrm{INTT}^{\psi^{-1}}\left(\mathrm{NTT}^{\psi}(a) \circ \mathrm{NTT}^{\psi}(b)\right). \tag{2}$$

Thus, a polynomial product reduces to two forward transforms, one coefficient-wise multiplication, and one inverse transform. The remainder of the paper focuses on mapping these steps efficiently to FPGA hardware.

Algorithm 1 NTT^{ψ} with CT-Butterflies

Input: $a = (a[0], \ldots, a[n-1])$ coefficients of polynomial $\mathbf{a} \in \mathbb{Z}_q[X]/(X^n+1)$ in normal order with n a power of two and q a prime with $q \equiv 1 \mod 2n$, table Ψ_{rev} of the powers of ψ_{2n} in bitreversed order, where ψ_{2n} is a primitive $2n$-th root of unity in $\mathbb{Z}_q$.
Output: $(\hat{a}[0], \ldots, \hat{a}[n-1]) = NTT^{\psi}(\mathbf{a})$ in bitreversed order.

```
t = n
for (m = 1; m < n; m = 2 · m) do
    t = t/2
    for (i = 0; i < m; i++) do
        j1 = 2 · i · t
        j2 = j1 + t − 1
        w = Ψ_rev[m + i]
        for (j = j1; j ≤ j2; j++) do
            U = a[j]
            V = w · a[j + t]
            a[j] = U + V  mod q
            a[j + t] = U − V  mod q
        end for
    end for
end for
return a
```

3 Algorithmics

This section summarizes the algorithms implemented by our accelerator. We first address a key limitation of textbook in-place NTT/INTT loops for hardware, namely excessive memory traffic per butterfly. We then describe how stage-level independence enables parallel execution across multiple NTT cores.

3.1 Memory Access Scheme

Roy *et al.* [30] observe that standard iterative NTT/INTT implementations are memory-inefficient when mapped to hardware. In the baseline algorithm (Algorithm 1), each butterfly in the innermost loop requires two reads and two writes. Even with dual-port RAM, this creates a structural imbalance: arithmetic is exercised once per butterfly, while memory must be accessed twice, leading to underutilized compute units. Prior work mitigates this by duplicating/banking memories (e.g., two BRAMs per butterfly operand pair) [25] or by applying FFT-style addressing and banking schemes [18].

Algorithm 2 INTT^{ψ} with GS-Butterflies

Input: $a = (a[0], \ldots, a[n-1]) \in \mathbb{Z}_q^n$ in bitreversed order with n being a power of two and prime q with $q \equiv 1 \mod 2n$, a table Ψ_{rev}^{-1} with the powers of ψ_{2n}^{-1} in bitreversed order, where ψ_{2n} is a primitive $2n$-th root of unity in $\mathbb{Z}_q$.
Output: $\mathbf{a} = {}_{INTT^{\psi}}(a)$ in normal order.

```
t = 1
for (m = n; m > 1; m = m/2) do
    j1 = 0
    h = m/2
    for (i = 0; i < h; i++) do
        j2 = j1 + t − 1
        w = Ψ^{-1}_{rev}[h + i]
        for (j = j1; j ≤ j2; j++) do
            U = a[j]
            V = a[j + t]
            a[j] = U + V  mod q
            a[j + t] = w · (U − V)  mod q
        end for
        j1 = j1 + 2 · t
    end for
    t = 2 · t
end for
for (j = 0; j < n; j++) do
    a[j] = a[j] · n^{-1}  mod q
end for
return a
```

We adopt the approach of Roy *et al.* [30] by *packing* two coefficients into one wider memory word. With 30-bit coefficients, packing yields a 60-bit word and reduces each butterfly to one memory read and one memory write (assuming the packed pair is colocated). Initially, coefficient i is packed with $i + n/2$ for $i \in [0, n/2 - 1]$. However, because the pairing between coefficients changes across stages, the packed layout that is optimal for one stage is generally not valid for the next, which introduces a repacking problem.

To maintain a packed layout across stages without overwriting values that are still needed, we compute *two* butterflies together and then repack their four outputs into two new packed words that match the next stage's pairing (Fig. 1). Consider two consecutive stages m_1 and m_2 with strides t_1 and $t_2 = t_1/2$. In m_1, a butterfly processes $(j,\ j + t_1)$. In m_2, the coefficient at j is paired with $j + t_2$, and the coefficient at $j + t_1$ is paired with $j + t_1 + t_2$. These two next-stage pairs are produced by evaluating the two butterflies operating on $(j,\ j + t_1)$ and $(j + t_2,\ j + t_2 + t_1)$, enabling conflict-free write-back into the packed format for the next stage. In the final stage, repacking is unnecessary because results are directly produced and no subsequent stage imposes a new pairing.

3.2 Parallelization

Within a fixed stage, butterfly computations are independent and can be parallelized. We therefore instantiate *C NTT cores*, each evaluating two butterflies in lockstep to support the packed-coefficient scheme. At the start of an n-point transform, there are $n/2$ packed words; we distribute them evenly across the C cores (with C chosen as a power of two) such that the two packed words required for the paired-butterfly step reside on the same core.

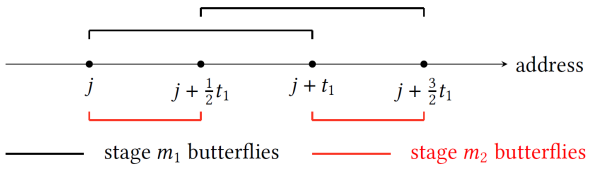


Figure 1: Address gap between two subsequent computation stages m_1 and m_2.

As stages progress, the effective stride decreases and the computation naturally decomposes into independent sub-transforms. Our controller adapts the mapping of packed words to cores to preserve locality as long as possible (*Mode 1*). When the stride becomes small enough that naive in-place updates would require multiple writes to the same local memory within one cycle, we avoid conflicts by pairing neighboring cores and scheduling them in a complementary order over their local index ranges (*Mode 2*). In the final stage, coefficients are emitted (in the expected bit-reversed order of this iterative NTT) rather than written back, so no further inter-core arrangement is required (*Mode 3*). Figure 2 illustrates the evolution of packed-coefficient distribution across stages for an example configuration.

The inverse transform is handled analogously by reversing the stage schedule and using the corresponding inverse butterflies and twiddle factors. Aside from an additional scaling multiplication at the end of the inverse transform, the same packing, distribution, and scheduling principles apply.

4 Architecture

In this section, we present the hardware architecture that implements the previously described algorithms. The design follows a hierarchical approach, where modules providing partial functionality are combined step by step until the complete system is realized. We begin with the simplest building blocks, modules for modular arithmetic, and progress through increasingly complex components, ending with the full NWC processor.

4.1 Parameters

We use the same parameters as [29]. The polynomials have a length of $n = 4096$, and the prime numbers q_i, and thus also the coefficients, are 30 bits wide.

4.2 Modular Arithmetic

4.2.1 Modular Addition & Subtraction. Performing addition on two operands $a, b \in \mathbb{Z}_q$, each 30 bits wide, results in a 31-bit number

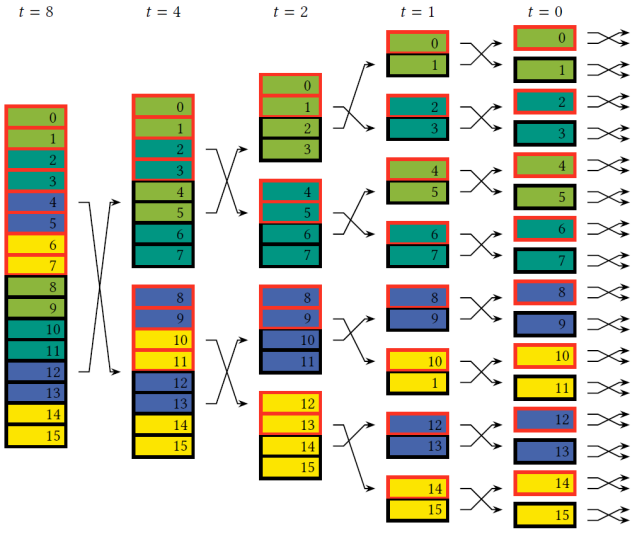
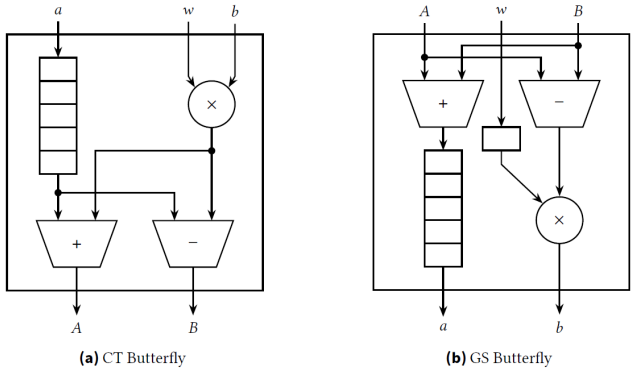


Figure 2: Distribution of packed coefficients among NTT cores (example with $n = 32$ and $C = 4$). Each color represents one core. Coefficients outlined in red belong to the upper butterfly of a core, while coefficients outlined in black belong to its lower butterfly. The numbers are the global addresses used in our explanation. The coefficients are output after the last stage.

within $[0, 2q - 2]$. Therefore, we can simply perform the addition and, if the result exceeds $q - 1$, subtract q. Both the initial addition and the conditional subtraction are completed within one clock cycle. Subtraction is handled similarly, with results in the range $[-q + 1, q - 1]$ and a conditional addition of q if $a - b < 0$.

4.2.2 Multiplication. Multiplication is implemented using the DSP slices of our device. Since a single DSP can only perform a 27-bit × 19-bit multiplication, four DSP slices are required for each 30-bit × 30-bit multiplier. Each multiplication is completed in a single clock cycle, although input buffering is employed. The result of multiplying $a, b \in \mathbb{Z}_q$ lies in $[0, q^2 - 2q + 1]$, making modular reduction more complex than addition or subtraction. Several reduction methods exist, the most common being Barrett reduction [6], used in [28], and Montgomery reduction [20], used in [19]. Both of these approaches necessitate additional multiplications. In [29], Roy *et al.* employ a sliding-window method combined

with lookup tables. They reduce 6 bits of the operand at a time until they obtain a 31-bit number and subtract q or $2q$ if necessary. This approach has the drawback of being restricted to the prime numbers for which lookup tables are provided in the design. Their approach is unrolled and thus supports pipelined evaluation. Since our parameters match theirs exactly, their code is open source and we want to avoid multiple multiplications in one modular multiplication, we adopt their method. As a result, modular multiplication is achieved in five clock cycles, three of which are spent on modular reduction.



(a) CT Butterfly **(b)** GS Butterfly

Figure 3: Architecture of our Butterfly Modules

4.3 NTT Core

Each NTT core contains two independent CT butterflies shown in Figure 3 ("upper" and "lower"), each backed by its own local RAM. Twiddle factors are stored in ROM; each butterfly uses a dedicated table since the final stage requires different twiddles per butterfly and ROM is not dual-port. This ROM-based approach is common [28, 29, 31, 34], while alternative designs compute twiddles on-the-fly to save resources [30]. The INTT core is identical except that it uses GS butterflies and ROM tables for powers of ψ_{2n}^{-1}.

Core-level control is kept minimal: global stage parameters (e.g., stride and stage index) are generated once at the processor level and broadcast to all cores. At the core level, only the twiddle selection differs across cores, so each core primarily computes the appropriate twiddle index and drives its butterflies accordingly.

A naïve single-memory implementation would suffer write-before-read hazards when neighboring cores exchange packed coefficients during *Mode 2*: one core may write a next-stage value into an address the other core has not read yet. Pipelining could hide this only with an impractically deep pipeline for small C (e.g., C=2). We therefore implement a double-buffered local memory: two internal memories are alternated between read and write roles across stages, eliminating write-before-read conflicts at the cost of doubling the local memory per core.

4.4 NTT Processor

With the NTT cores available, we can construct the NTT processor. This module is capable of receiving a polynomial as input and outputting its NTT as result. It contains C NTT cores and implements the logic from section 3.2. The data movement described in section 3.2 is implemented in a module we call *Router*, which can be thought of as a part of the NTT processor. We developed two designs, out of which one was implemented, while the other remains a theoretical model.

The first design, which is the one we implemented, is called *Vector* and shown in Figure 4. Its name comes from the arrangement of the NTT cores, which are aligned in a single column. In this configuration, all cores work on the same NTT computation. They are connected through a router that transfers results from the NTT cores to the correct recipients. Once the computation is complete, the NTT coefficients are also output through the router. When

active, the output delivers $2C$ packed coefficients (equivalent to $4C$ normal coefficients) per clock cycle. In theory, the input could also take $2C$ packed coefficients per clock cycle. However, as explained in Section 5.2, our implementation inputs one packed coefficient per clock cycle.

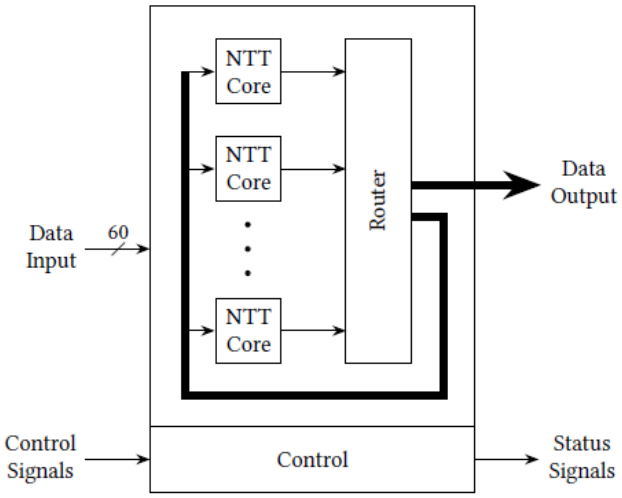


Figure 4: Architectural design for the NTT processor.

It is in this module that we implement the state machine of needed for parallelization and distribute all required values to the NTT cores. This module also contains the logic for accepting input and routing it to the appropriate cores. Most variables, such as m and t, are powers of two, which allows us to save resources by storing their logarithms instead of their actual values. Although parallelization from section 3.2 contains many multiplications and divisions, closer inspection reveals that all of them involve powers of two. This makes them trivial to implement as bit shifts, or even simple additions and subtractions on the logarithmic values. Through control signals, the NTT processor is instructed to load data into the cores and start computation. The computation status, such as whether the output is currently active and must be read to avoid losing the computed NTT, is reported through status signals.

The INTT processor is constructed in the same way, with the obvious difference that it uses INTT cores instead of NTT cores and its control logic implements the INTT explained in section 3.2. However, the INTT requires an additional multiplication after the main computation. Therefore, $4C$ modular multipliers must be added to the output, as each INTT core can output four coefficients per clock cycle.

4.5 NWC Processor

The NWC processor forms the top level of our hierarchical architecture and is capable of computing an entire NWC. Its structure is shown in Figure 5. Two NTT processors compute the respective NTT of the input polynomials in parallel, an array of modular multipliers performs the coefficient-wise multiplication, and an INTT processor generates the final result by evaluating the INTT on the multiplier's output. Since the INTT processor outputs $2C$ packed coefficients per clock cycle, while our outer testing setup can accept only one packed coefficient per clock cycle, a memory module with sufficient bandwidth is placed after the INTT core to serve as a buffer.

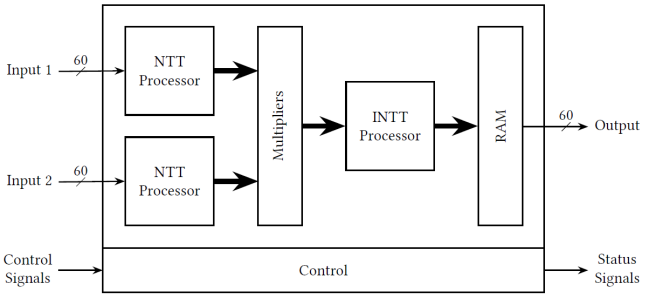


Figure 5: Architecture of the NWC processor.

The module's control logic orchestrates cooperation between the processors by setting their control signals and managing the output buffer. Because the processors are separate from each other, evaluating only a single NWC at a time would leave either the NTT processors or the INTT processor idle, resulting in significant inefficiency. Instead, the NWC processor advances two NWC computations simultaneously, where, while the INTT processor completes one operation, the NTT processors already begin processing the next. Thus, the NWC processor features a two-stage pipeline.

The core count C of our NWC processor specifies how many cores each of the NTT processors and the INTT processor contain. In principle, it would be possible for these processors to have different core counts, but in most cases, this is impractical. If the NTT processors differ in core count, one would have to wait for the other to finish before the coefficient-wise multiplication could proceed, wasting the extra resources invested in the faster processor. A mismatch between the NTT and INTT processors also causes complications. If the NTT processors have more cores, they will produce more coefficients per clock cycle than the INTT processor can consume, requiring additional buffering memory. This could still be a viable performance optimization when memory is abundant but resources do not allow for increasing the INTT processor's core count. Conversely, if the INTT processor has more cores than the NTT processors, its start must be delayed to avoid running out of input coefficients, again necessitating extra buffering. This configuration may also offer performance benefits when memory is plentiful but the NTT processors' core count cannot be increased.

5 Results

In this section, we present the resources used to benchmark the NWC in software and the tools employed to implement our accelerator in hardware. We compare the performance of the accelerator against the software implementation and the different accelerator configurations against each other. Finally, we compare our accelerator against related work in the literature.

5.1 Implementation & Comparison Basis

5.2 Software Comparison Basis

To serve as a baseline for comparison with our accelerator, we implemented NWC in the C programming language using Algorithms 1 and 2. For modular reduction, we used the Barrett method, implemented based on an algorithm by Wu *et al.* [33].

The program was executed on four machines, each equipped with a different CPU: AMD Ryzen 7 2700X, Intel i5 12500, Intel i9 12900, and Apple M1. The x86_64 systems (AMD and Intel) were running Ubuntu 22.04, while the ARM-based Apple system was running macOS Sequoia 15.5. The source code was compiled using GCC 11.4.0 for the x86_64 CPUs and Clang 19.1.7 for the ARM CPU. Both compilers were invoked with the flags `-O3` and `-ffast-math` to ensure that our code runs as performant as possible.

The execution time of a single NWC operation was measured and averaged over one million iterations. The best performance was achieved by the Apple M1 CPU, with an average execution time of 95 μs per operation, followed by the Intel i9 at 119 μs, and the Intel i5 at 135 μs. The AMD Ryzen 7 performed the slowest, with an average of 258 μs. An overview of the system configurations and results is provided in Table 1.

Table 1: Configuration and timing results of used machines.

CPU	ISA	OS	NWC time
Apple M1	ARM64	Sequoia 15.5	95μs
Intel i9-12900	x86_64	Ubuntu 22.04	119μs
Intel i5-12500	x86_64	Ubuntu 22.04	135μs
AMD Ryzen 7 2700X	x86_64	Ubuntu 22.04	258μs

5.3 Hardware Implementation

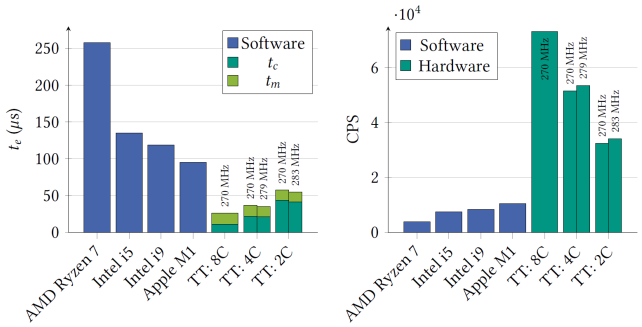
We implemented the architecture described in Section 4 using Verilog and SystemVerilog in the Vivado 2022.2 development environment. Our target platform is the AMD Xilinx UltraScale+ MPSoC on the ZCU102 development board. The software running on the MPSoC's ARM Cortex-A53 cores, which were used exclusively for testing and performance measurements, was developed using Vitis 2022.2[1]. The number of cores C in the NTT and INTT processors, as well as the index of the modulus used, can be set parametrically at synthesis time. Due to the resource limitations of the FPGA device and the requirement that C be a power of two, the highest core count we were able to implement and test on hardware was $C = 8$. We also evaluated configurations with $C = 4$ and $C = 2$.

In our experimental setup, we use an AXI GPIO interface to enable the Processing System (PS), which contains the ARM cores, to set control signals and read status signals from the NWC processor. Said processor is also connected to six true dual-port BRAMs through dedicated AXI BRAM controllers, enabeling memory-mapped access. Four of these BRAMs are used for storing the two input polynomials before they are loaded into the NTT processors, while the remaining two store the output polynomial after processing. Each polynomial is distributed across two BRAMs, as the NWC processor requires an input of at least two coefficients per clock cycle due to the coefficient packing described in Section 3.1. This setup facilitates convenient data transfer of inputs and outputs between the PS and the NWC processor.

[1]Code is available at: https://github.com/B-Ste/ntt
https://github.com/B-Ste/fpga_ntt_accelerator

5.4 Experimental Results

5.4.1 Performance. In the following, we denote the eight-core, four-core, and two-core versions of our accelerator as 8C, 4C, and 2C, respectively. 8C achieves a maximum operating frequency of 270 MHz. This frequency is also used for 4C and 2C to provide a common basis for comparison. Unless stated otherwise, all performance figures refer to measurements taken at that frequency.



(a) t_e of single NWC operation **(b)** Computations Per Second (CPS)

Figure 6: Performance of this paper (TT) in comparison to the software implementation.

The maximum achievable frequencies for 4C and 2C are 279 MHz and 283 MHz, respectively. Three performance metrics were measured: the execution time t_e, which captures the duration from the beginning of data input to the end of data output; the computation time t_c, which measures the period during which the processor actively performs computations; and the maximum number of computations the processor can complete in one second, expressed as Computations Per Second (CPS).

By subtracting t_c from t_e, we obtain the time spent on memory I/O, denoted as t_m. As described in Section 5.3, during the transfer of polynomials into and out of the processors, two coefficients per polynomial and processor can be transferred per clock cycle. Therefore, the input transfer requires 2048 clock cycles, and the output—slightly overlapping with the final computation iteration—requires a similar amount. As shown in our architecture in section 4, this setup is not optimal. In theory, up to 4C coefficients per clock cycle and processor can be transferred. Nevertheless, the above setup was intentionally chosen to simplify the testing process. Thus, when discussing performance figures, it is important to note that t_m could be significantly reduced.

Table 2 summarizes the measured performance. At 270 MHz, 8C achieves a 3.6× speedup over the fastest CPU baseline (Apple M1) in end-to-end time t_e, and 1.4× over 4C. Since t_e includes the largely constant I/O component t_m, the compute-only time t_c shows the expected scaling: 8C provides 2× speedup over 4C (and 1.9× at maximum frequency). Similarly, 4C improves over the Apple M1 by 2.6× in t_e and scales by 2× over 2C in t_c (reduced to 1.9× when

Table 2: Performance measurements of tested configurations for common frequency 270 MHz and the respective maximum frequencies.

C	f (MHz)	t_e (μs)	t_c (μs)	CPS
8	270	26.13	11.08	73087
4	270	36.61	21.9	51560
4	279	35.34	21.25	53479
2	270	57.4	43.68	32435
2	283	54.71	41.65	34099

comparing 4C@270 MHz to 2C@283 MHz). Even 2C remains faster than the Apple M1 ($\approx 1.7\times$ in t_e), and the overall scaling from 2C to 8C approaches 4× (3.8× when 2C runs at 283 MHz). Figure 6(a) visualizes the single-operation speedups.

Sustained operation requires sufficient external memory bandwidth. For sequential execution, the minimum bandwidth is 13.2/8.6/4.9 Gbit/s for 8C/4C/2C, respectively, under the constraint that writes avoid unread addresses. Our design additionally supports two-operation pipelining, which is captured by computations-per-second (CPS). Relative to the Apple M1, CPS improves by 6.9× for 8C, 4.9× (5.1× at 279 MHz) for 4C, and 3.1× (3.2× at 283 MHz) for 2C; these results are shown in Fig. 6(b). For uninterrupted pipelined operation, the corresponding minimum bandwidth increases to 18.8/13.4/8.6 Gbit/s for 8C/4C/2C, respectively.

Table 3: Resource utilization and power consumption of tested configurations for common frequency $f = 270$ MHz and the respective maximum frequencies.

C	f (MHz)	LUT	FF	BRAM	DSP	Power (W)
8	270	172407	31938	112	448	9.012
4	270	85443	17403	56	224	4.133
4	279	85411	19045	56	224	4.378
2	270	45437	9810	24	112	2.275
2	283	45531	10249	24	112	2.413

5.4.2 Resource Usage & Power. The primary objective of this paper is to develop an accelerator with the highest possible performance. Consequently, resource utilization and power consumption were not primary concerns beyond ensuring that the design fits on the target device. This prioritization of performance is common in FPGA-accelerated cloud systems. Table 3 lists resource and power measurements for our accelerators. The values exclude additional components from our testing setup, such as the memory-mapped BRAMs and their controllers.

8C is the most resource-intensive configuration, consuming roughly twice the resources of 4C, which corresponds well to 8C theoretically delivering twice the performance of 4C. However, when considering the effects of memory I/O or pipelining, a twofold increase in resources for only a 1.4× performance gain appears suboptimal. In contrast, 4C is only 88% larger than 2C while providing a 1.6× performance improvement, making the upgrade from 2C to 4C a more compelling decision than the upgrade from 4C to 8C. The same observations apply to power consumption. While 4C consumes only 81% more power than 2C, 8C increases its power consumption by 2.2× compared to 4C.

Between the base-frequency and maximum-frequency configurations, our resource usage changes only slightly: the design uses slightly fewer LUTs and slightly more flip-flops. Both 4C and 2C consume roughly 6% more power at their respective maximum frequencies. However, 2C achieves an additional 4.7% performance increase, whereas 4C's performance improves by only 3.5%.

5.5 Comparison to Related Work

Our design is most closely related to Roy *et al.* [29], from which we also adopt the same parameters and primes. Their accelerator targets the full FV scheme and processes multiple RNS components (total modulus size 180/372 bits with 30-bit residues), whereas our design accelerates a single residue component and is limited to NWC. Using the per-instruction timings reported in [29] (NTT 73 μs, INTT 85 μs, mul 13.1 μs) and normalizing by the number of simultaneously processed RNS components, we estimate 92.1 μs per NWC for one residue. Under this normalization, our 8C/4C/2C configurations achieve $3.5\times/2.5\times/1.6\times$ speedup, respectively. This comparison is not resource-fair: their full processor is reported at roughly 50k LUTs, substantially below our configurations.

HEPCloud [28] accelerates FV at a larger polynomial degree ($n = 2^{16}$) and supports generic cyclotomic reduction (Newton method [32]). Since [28] reports timings for NTT/mul/INTT (excluding the polynomial reduction step), we compare only these kernels by scaling their reported runtime at 100 MHz to our $n = 4096$ (expected 21.3× reduction). This yields an estimated runtime of 236.37 μs for 8C under the same kernel-only basis, corresponding to a 6.4× speedup (and $3.2\times/1.6\times$ for 4C/2C). Again, functionality and resource budgets differ substantially: their full processor fits in about 72k LUTs and 63k FFs.

Mert *et al.* [19] present a parametric NTT-only generator and evaluate, among others, $n = 4096$ with 60-bit coefficients. Since our measurements report NWC time, we conservatively bound one NTT by $t_c/2$. Under the assumption that a 60-bit-capable variant achieves similar NTT latency with additional resources, 8C would exceed their fastest (32-core) configuration by 1.4×, while 4C and 2C achieve 0.7× and 0.4×, respectively; these differences are accompanied by substantial resource trade-offs.

6 Conclusion

In this paper, we introduced homomorphic encryption from both a conceptual and theoretical perspective. We analyzed the FV scheme and saw that polynomial multiplication, specifically NWC, is a key performance bottleneck. To address this, we presented the NTT, which reduces the computational complexity of NWCs from $O(n^2)$ to $O(n \log n)$. Furthermore, we detailed the algebraic foundations of the NTT and provided algorithms suitable for software implementation. Then, we refined our algorithms to better suit hardware implementation by resolving a memory conflict that would have led to underutilization of the arithmetic circuitry. We also parallelized the algorithms across multiple NTT cores. We propose and implement the vector architecture. This architecture was evaluated under different configurations on an AMD ZCU102 evaluation board. Our accelerators outperformed the software implementation

by a considerable amount. The fastest configuration achieved a 6.9× speedup over the fastest CPU, while the slowest configuration was still 3.2× faster. We also expect our accelerator to surpass the performance of several related works. However, this improvement generally comes at the cost of significantly higher resource consumption compared to those implementations.

References

[1] Lawrence Abrams. 2024. UnitedHealth now says 190 million impacted by 2024 data breach. BleepingComputer. https://www.bleepingcomputer.com/news/security/unitedhealth-nowsays-190-million-impacted-by-2024-data-breach/

[2] Mohamed Aboelenien Ahmed, Mohamed Alsharkawy, Simon Bothe, Hassan Nassar, Lars Bauer, and Jorg Henkel. 2026. FPGA Acceleration of Homomorphic Encryption Assisted Federated Learning. *Journal of Circuits, Systems and Computers* Early Access (2026). doi:10.1142/S0218126626450040

[3] Mohamed Aboelenien Ahmed, Mohamed Alsharkawy, Hassan Nassar, Heba Khdr, Jeferson Gonzalez, and Jörg Henkel. 2026. Efficient Federated Learning with Low-Rank Updates under Homomorphic Encryption. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE)*. Verona, Italy. Apr. 20–22, 2026.

[4] A. V. Aho and J. E. Hopcroft. 1974. *The Design and Analysis of Computer Algorithms*. Pearson Education India.

[5] A. A. Badawi, B. Veeravalli, C. F. Mun, and K. M. M. Aung. 2018. High-Performance FV Somewhat Homomorphic Encryption on GPUs: An Implementation using CUDA. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (May 2018), 70–95. doi:10.13154/tches.v2018.i2.70-95

[6] Paul Barrett. 1987. Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor. In *Advances in Cryptology — CRYPTO' 86*, Andrew M. Odlyzko (Ed.). Springer, Berlin, Heidelberg, 311–323. doi:10.1007/3-540-47721-7_24

[7] J. W. Bos, W. Castryck, I. Iliashenko, and Frederik Vercauteren. 2017. Privacy-Friendly Forecasting for the Smart Grid Using Homomorphic Encryption and the Group Method of Data Handling. In *Progress in Cryptology - AFRICACRYPT 2017*, Marc Joye and Abderrahmane Nitaj (Eds.). Springer International Publishing, Cham, 184–201. doi:10.1007/978-3-319-57339-7_11

[8] J. W. Bos, Kristin Lauter, and Michael Naehrig. 2014. Private Predictive Analysis on Encrypted Medical Data. *Journal of Biomedical Informatics* 50 (Aug. 2014), 234–243. doi:10.1016/j.jbi.2014.04.003 Special Issue on Informatics Methods in Medical Privacy.

[9] Simon Bothe, Hassan Nassar, Lars Bauer, and Jörg Henkel. 2024. Client-Server Framework for FPGA Acceleration of Fan-Vercauteren-Based Homomorphic Encryption. In *2024 International Conference on Microelectronics (ICM)*. 1–5. doi:10.1109/ICM63406.2024.10815906

[10] Zvika Brakerski. 2012. Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP. In *Advances in Cryptology – CRYPTO 2012*, Reihaneh Safavi-Naini and Ran Canetti (Eds.). Springer, Berlin, Heidelberg, 868–886. doi:10.1007/978-3-642-32009-5_50

[11] Joan Daemen and Vincent Rijmen. 2000. The Block Cipher Rijndael. In *Smart Card Research and Applications*, Jean-Jacques Quisquater and Bruce Schneier (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 277–284.

[12] Thore Graepel, Kristin Lauter, and Michael Naehrig. 2013. ML Confidential: Machine Learning on Encrypted Data. In *Information Security and Cryptology – ICISC 2012*, Taekyoung Kwon, Mun-Kyu Lee, and Donghoon Kwon (Eds.). Springer, Berlin, Heidelberg, 1–21. doi:10.1007/978-3-642-37682-5_1

[13] Rawan Hagag, Hassan Nassar, Jörg Henkel, and Mohamed A. Abd El Ghany. 2025. Hardware-Accelerated Mode-Switching Polymorphic Encryption for Privacy Preserving Machine Learning. In *2025 14th International Conference on Modern Circuits and Systems Technologies (MOCAST)*. 1–5. doi:10.1109/MOCAST65744.2025.11083924

[14] Rawan Hagag, Hassan Nassar, Jörg Henkel, and Mohamed A. Abd El Ghany. 2025. A Polymorphic Encryption Framework for Cross-Silo and Cross-Device Distributed Learning. In *2025 37th International Conference on Microelectronics (ICM)*. 1–5. doi:10.1109/ICM66518.2025.11322452

[15] Aaron Holmes. 2021. 533 million Facebook users' phone numbers and personal data have been leaked online. Business Insider. https://www.businessinsider.com/stolen-dataof-533-million-facebook-users-leaked-online-2021-4

[16] Ihsan Jabbar and Saad Najim Alsaad. 2017. Design and Implementation of Secure Remote e-Voting System Using Homomorphic Encryption. *International Journal of Network Security* 19, 5 (Sept. 2017). doi:10.6633/IJNS.201709.19(5).06

[17] Zhen Liang and Yexin Zhao. 2022. Number Theoretic Transform and Its Applications in Lattice-based Cryptosystems: A Survey. arXiv:2211.13546 [cs]. doi:10.48550/arXiv.2211.13546

[18] Y. Ma and L. Wanhammar. 2000. A Hardware Efficient Control of Memory Addressing for High-Performance FFT Processors. *IEEE Transactions on Signal Processing* 48, 3 (March 2000), 917–921. doi:10.1109/78.824693

[19] A. C. Mert, E. Karabulut, E. Öztürk, E. Savaş, and A. Aysu. 2022. An Extensive Study of Flexible Design Methods for the Number Theoretic Transform. *IEEE Trans. Comput.* 71, 11 (Nov. 2022), 2829–2843. doi:10.1109/TC.2020.3017930

[20] Peter L. Montgomery. 1985. Modular Multiplication without Trial Division. *Math. Comp.* 44, 170 (1985), 519–521. doi:10.1090/S0025-5718-1985-0777282-X

[21] Hassan Nassar, Lars Bauer, and Jörg Henkel. 2024. HBMorphic: FHE Acceleration via HBM-Enabled Recursive Karatsuba Multiplier on FPGA. In *2024 IEEE 32nd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 217–217. doi:10.1109/FCCM60383.2024.00038

[22] Hassan Nassar, Lars Bauer, and Jörg Henkel. 2025. Turbo-FHE: Accelerating Fully Homomorphic Encryption With FPGA and HBM Integration. *IEEE Design & Test* 42, 3 (2025), 86–93. doi:10.1109/MDAT.2025.3527368

[23] Hassan Nassar, Martin Rapp, Lars Bauer, Mostafa Elshimy, Zeynep Guelbeyaz Demirdag, and Jörg Henkel. 2026. MIQARA: Mixed-Criticality Queue-based Architecture for Reconfigurable Accelerator Platforms. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE)*. Verona, Italy. Apr. 20–22, 2026.

[24] J. M. Pollard. 1971. The Fast Fourier Transform in a Finite Field. *Math. Comp.* 25, 114 (1971), 365–374. doi:10.1090/S0025-5718-1971-0301966-0

[25] Thomas Pöppelmann and Tim Güneysu. 2012. Towards Efficient Arithmetic for Lattice-Based Cryptography on Reconfigurable Hardware. In *Progress in Cryptology – LATINCRYPT 2012*, Alejandro Hevia and Gregory Neven (Eds.). Springer, Berlin, Heidelberg, 139–158. doi:10.1007/978-3-642-33481-8_8

[26] Ronald L. Rivest, Adi Shamir, and Leonard Adleman. 1978. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Commun. ACM* 21, 2 (Feb. 1978), 120–126. doi:10.1145/359340.359342

[27] Gregor Seiler. 2018. Faster AVX2 Optimized NTT Multiplication for Ring-LWE Lattice Cryptography. Cryptology ePrint Archive, Report 2018/039. https://eprint.iacr.org/2018/039

[28] Sujoy Sinha Roy, Kimmo Järvinen, J. Vliegen, Frederik Vercauteren, and Ingrid Verbauwhede. 2018. HEPCloud: An FPGA-Based Multicore Processor for FV Somewhat Homomorphic Function Evaluation. *IEEE Trans. Comput.* 67, 11 (Nov. 2018), 1637–1650. doi:10.1109/TC.2018.2816640

[29] Sujoy Sinha Roy, Furkan Turan, Kimmo Järvinen, Frederik Vercauteren, and Ingrid Verbauwhede. 2019. FPGA-Based High-Performance Parallel Architecture for Homomorphic Computing on Encrypted Data. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 387–398. doi:10.1109/HPCA.2019.00052

[30] S. S. Sinha Roy, Frederik Vercauteren, N. Mentens, D. D. Chen, and Ingrid Verbauwhede. 2014. Compact Ring-LWE Cryptoprocessor. In *Cryptographic Hardware and Embedded Systems – CHES 2014*, Lejla Batina and Matthew Robshaw (Eds.). Springer, Berlin, Heidelberg, 371–391. doi:10.1007/978-3-662-44709-3_21

[31] Y. Su, B.-L. Yang, C. Yang, Z.-P. Yang, and Y.-W. 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 (Aug. 2022), 993–1006. doi:10.1109/TVLSI.2022.3166355

[32] Joachim von zur Gathen and Jürgen Gerhard. 2003. *Modern Computer Algebra*. Cambridge University Press.

[33] T. Wu, S.-G. Li, and L.-T. Liu. 2012. Modular Multiplier by Folding Barrett Modular Reduction. In *2012 IEEE 11th International Conference on Solid-State and Integrated Circuit Technology*. 1–3. doi:10.1109/ICSICT.2012.6467863

[34] Furkan Yaman, A. C. Mert, Erdem Öztürk, and Erkay Savaş. 2021. A Hardware Accelerator for Polynomial Multiplication Operation of CRYSTALS-KYBER PQC Scheme. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 1020–1025. doi:10.23919/DATE51398.2021.9474139

[35] Xun Yi, Russell Paulet, and Elisa Bertino (Eds.). 2014. *Homomorphic Encryption and Applications*. Springer International Publishing.