

FINAL bootstrap acceleration on FPGA using DSP-free constant-multiplier NTTs

Jonas Bertels¹, Hilder V. L. Pereira² and Ingrid Verbauwhede¹

¹ COSIC, KU Leuven, Leuven, Belgium `{firstname}.{lastname}@esat.kuleuven.be`

² University of Campinas (UNICAMP), Campinas, Brazil `hilder@unicamp.br`

Abstract. This work showcases **Quatorze-bis**, a state-of-the-art Number Theoretic Transform circuit for TFHE-like cryptosystems on FPGAs. It contains a novel modular multiplication design for modular multiplication with a constant for a constant modulus. This modular multiplication design does not require **any** DSP units or any dedicated multiplier unit, nor does it require extra logic when compared to the state-of-the-art modular multipliers. Furthermore, we present an implementation of a constant multiplier Number Theoretic Transform design for TFHE-like schemes. Lastly, we use this Number Theoretic Transform design to implement a FINAL hardware accelerator for the AMD Alveo U55c which improves the Throughput metric of TFHE-like cryptosystems on FPGAs by a factor 9.28x over Li et al.’s NFP CHES 2024 accelerator and by 10-25% over the absolute state-of-the-art design FPT [VDTV23] while using one third of FPTs DSPs.

Keywords: NTT · FHE · FINAL · Hardware Design · FPGA

1 Introduction

Fully Homomorphic Encryption (FHE) is an advanced type of cryptography that enables computation on encrypted data [Gen09], thus allowing protection of data both in transit, and during processing. Typical usages of FHE involve a client, who is the data owner, and a server, which offers some service, such as a trained machine learning model. The client can then encrypt the data and send it to the server, which processes the encrypted data and sends back to the client a new ciphertext encrypting the result of the computation.

Despite its wide range of applications, FHE has encountered some barriers for its adoption. For instance, FHE is often considered very technical [GMT24], meaning that it is not easy to be correctly instantiated and used by non-specialists. Another barrier, which is more critical, is the computational overhead that FHE brings to the applications. Basically, compared to the unprotected scenario where the server runs the application in clear, FHE can easily be 10000 times slower [LKL⁺22, AKT⁺24].

Given this limitation, a lot of effort has been put into making FHE more efficient. And indeed FHE schemes have been steadily improving over the last decade, with many new algorithms that reduce the asymptotic costs of FHE [LMK⁺23, GPV23, WWL⁺24]. However, despite the algorithmic improvements, the performance of existing FHE schemes is still not satisfactory when implemented on CPUs. Thus, studying how to efficiently implement FHE on hardware has become a growing line of research, with works spanning from GPU-assisted implementations [WLH⁺23, YSD⁺24] to ASICs (application-specific integrated circuit) implementing whole schemes [SFK⁺22, GBP⁺23, JLJ22a, PPC⁺23].

Although ASICs have the potential of making FHE practical, they are very expensive to produce, manufacturing them can take years, and they are typically very inflexible. For example, in [AMK⁺23], some ASICs for accelerating FHE have an estimated fabrication

cost of about 25 million USD. Hence, FPGA (field-programmable gate array) is a viable alternative trying to conciliate flexibility, production costs and speedups. Firstly, FPGA can accommodate small changes in the schemes. For example, if a new algorithm for one specific operation of the scheme is proposed or a new set of parameters, one can change their FPGA implementation, while an ASIC could not be updated in such situation. Secondly, FPGAs can still reduce the running time of systems using FHE by orders of magnitude [VDTV23]. However, implementing certain types of FHE on FPGA is challenging, because FHE is typically memory-bound, due to the large ciphertexts and keys. In practice, an FPGA can end up being idle waiting for data transfers from the computer memory to its internal memory. For instance, well-established FHE schemes, such as BGV [BGV12] and CKKS [CKKS17], require several gigabytes just to store the keys used for *bootstrapping*, which is the most important operation in FHE. Basically, in FHE, each ciphertext has an internal noise that is small when the message has just been encrypted, and increases with each homomorphic operation. At some point, the noise is too large and if any additional operation is performed, it is no longer possible to decrypt the ciphertext. Then, the bootstrapping is run, reducing the inherent noise of the ciphertext and allowing for more computation.

Taking this into account, FHE schemes that require less memory seem better suited for FPGA implementations. Such schemes, known as third-generation schemes, started with FHEW [DM15] and were then improved in TFHE [CGGI16]. They have very small ciphertexts and they require bootstrapping after every homomorphic operation. However, their bootstrapping can be evaluated in just a few milliseconds on a CPU and using just a few megabytes of key material, in contrast to a few minutes required to evaluate BGV's or CKKS' bootstrapping and using gigabytes of memory. Hence, some works have implemented TFHE on FPGA [VDTV23, ZCJ24, KL24]. In spite of TFHE being arguably the most famous 3rd-generation FHE scheme, there is a newer one, called FINAL [BIP⁺22], which was around 28% faster on CPU than the original TFHE and used 45% smaller keys. Specially because of its reduced memory requirements, it seems even more suitable for FPGA implementations.

TFHE-like schemes are generally compute-bound, as they require many external products between ciphertexts to be completed. As ciphertexts are polynomials, these external products consist of polynomial multiplications. To achieve the best efficiency either the Fast Fourier Transform or the Number Theoretic Transform is utilized to complete the polynomial multiplication in $\mathcal{O}(n \log n)$. In other words, a single operation (in our case, the Number Theoretic Transform) needs to be completed multiple times to finish one bootstrap. Thus the bottleneck of the bootstrapping operation is the Number Theoretic Transform (NTT), and any improvement in the Number Theoretic Transform translates to a similar improvement in performance of the bootstrapping step.

Thus, in this work, we design and implement a state-of-the-art FPGA hardware accelerator for FINAL, utilizing a constant-multiplier Number Theoretic Transform and extremely efficient modular multiplication. The combination of these two techniques allows us to fit an extremely large quantity of butterfly units (also known as Processing Elements) onto the FPGA while minimizing the logic required to feed these butterfly units. This accelerator reaches a bootstrapping throughput of 31250 Bootstraps Per Second.

Contributions

Our main contributions are as follows:

1. A novel modular reduction design for modular multiplication with a constant for a constant modulus. This design does not require **any** DSP units nor does it require extra logic when compared to state-of-the-art techniques such as K-reduction, LUT-reduction or constant-modulus Barrett reduction. Moreover, unlike many state-

of-the-art modular multipliers, it functions equally well regardless of the hamming weight of the modulus. The core idea of this modular multiplier is that multiplication and modular reduction are both linear functions, and can thus be replaced by the sum of a lookup table, analogous to in-memory computation. This butterfly design requires fewer than 140 LUTs per butterfly.

2. An implementation of a constant multiplier Number Theoretic Transform design, analogous to the FFT version presented in [GM21]. The constant multiplier architecture is equivalent to performing a two-dimensional NTT, where one dimension is time and the other space on the FPGA. Earlier designs dedicate significant logic to preventing memory stalls of the NTT units. Our constant multiplier DSP-free NTT design eliminates much of this logic, and exploits the unique structure of the constant multipliers to implement the butterfly units without using DSPs and with less LUT usage than state-of-the-art butterfly units with the same bit width.
3. An implementation of a FINAL hardware accelerator for the AMD Alveo U55c. This state-of-the-art figure is achieved fully placing one iteration of the FINAL blind rotation on the hardware, and matching the streaming width of the circuit to the streaming width of the NTTs. Thanks to the small size of the butterflies, a large streaming width S of 64 can be achieved, which in turns allows the hardware accelerator to achieve a throughput of 31250 Bootstraps Per Second (BPS) utilizing only 1920 DSPs.

2 Preliminaries

We denote vectors by bold lowercase letters and matrices by bold uppercase letters. For a vector \mathbf{u} , we denote the infinity norm by $\|\mathbf{u}\|$. We define $\mathcal{R} = \mathbb{Z}[X]/\langle X^N + 1 \rangle$, that is, the (cyclotomic) ring of polynomials modulo $X^N + 1$, where N is a power of two. For an integer $q > 1$, we define $\mathcal{R}_q = \mathbb{Z}_q[X]/\langle X^N + 1 \rangle$, i.e., the ring of polynomials from \mathcal{R} but with coefficients reduced modulo a constant q . For any $c \in \mathcal{R}_q$ and integer $B \geq 2$, the gadget decomposition of c , denoted by $\text{SignedDecomp}_B(c)$, is a generalization of the base- B integer signed decomposition for polynomials, that is, $\text{SignedDecomp}_B(c) = (d_0, \dots, d_{\ell-1})$ where $\ell = \lfloor \log_B(q) \rfloor + 1$, each $d_i \in \mathcal{R}$, the coefficients of each d_i belong to $\{-B/2, \dots, -1, 0, 1, \dots, B/2\}$, and $c = \sum_{i=0}^{\ell-1} d_i \cdot B^i$.

2.1 (Ring) Learning With Errors and NTRU

In the learning with errors problem (LWE) [Reg05] with parameters n , q , and σ , an attacker has to find a secret vector $\mathbf{s} \in \mathbb{Z}^n$ given many samples of the form (\mathbf{a}_i, b_i) , where \mathbf{a}_i is uniformly sampled from \mathbb{Z}_q^n and $b_i := \mathbf{a}_i \cdot \mathbf{s} + e_i \bmod q$, with e_i following a discrete Gaussian distribution with parameter σ .

In its ring version, known as RLWE [LPR10], the challenge is to find the secret polynomial $s \in \mathcal{R}$ given samples of the form (a_i, b_i) , where a_i is uniformly sampled from \mathcal{R}_q and $b_i := a_i \cdot s + e_i \bmod q$, for some small polynomial $e_i \in \mathcal{R}$ called the noise term.

The NTRU problem with parameters N, q , and σ consists in finding the secret polynomial $f \in \mathcal{R}$ given polynomials $h_i = g_i \cdot f^{-1} \in \mathcal{R}_q$, where the coefficients of f and g_i 's are sampled from a discrete Gaussian distribution with width σ [Dv21].

The three problems are believed to be quantum hard and all known algorithms to solve them run in exponential time [APS15, ACD⁺18].

2.2 Fully homomorphic encryption

Fully homomorphic encryption (FHE) allows computation to be performed on encrypted data. In a typical scenario, a client generates a ciphertext c that encrypts some data m under a public key pk . Then, a server holding c and pk , but not the corresponding secret (decryption) key sk , can choose an arbitrary function f and generate a new ciphertext \tilde{c} encrypting $f(m)$. This step is called the “homomorphic computation” and one says that “ f was evaluated homomorphically”. Finally, the client can download \tilde{c} , decrypt it using sk , and obtain $f(m)$. Therefore, FHE allows one to outsource computation to an untrusted party while preserving the confidentiality of both the input and output data.

Among the FHE schemes, TFHE gained a lot of attention because of its fast bootstrapping. Basically, to encrypt a message m , TFHE adds it to an LWE sample, obtaining thus a ciphertext $\mathbf{c} = (\mathbf{a}, b) \in \mathbb{Z}_q^{n+1}$, where $b = \mathbf{a} \cdot \mathbf{s} + e + m \bmod q$. After one homomorphic operation is performed on \mathbf{c} , one has to run the bootstrapping to refresh the ciphertext and prepare it to new operations. Hence, most of the time spent in any homomorphic evaluation is used with the many executions of the bootstrapping.

For the bootstrapping, two new types of ciphertexts are defined, both constructed on top of the RLWE problem. One is called simply a RLWE ciphertext and it encrypts a polynomial $m_0 \in \mathcal{R}$ as $\mathbf{c} = (a, b) \in \mathcal{R}_q^2$, where $b = a \cdot s + e + m_0 \bmod q$ (for some secret $s \in \mathcal{R}$). The second one is a (ring) GSW ciphertext [DM15], which also encrypts a polynomial $m_1 \in \mathcal{R}$, but into a matrix $\mathbf{C} \in \mathcal{R}_q^{2 \times 2\ell}$ where, basically, each row is a scalar ciphertext and $\ell = \lceil \log q \rceil + 1$. The main operation used during the bootstrapping is the *external product*, which decomposes \mathbf{c} into a vector of 2ℓ small polynomials, then multiplies it by \mathbf{C} , generating a new RLWE ciphertext $\hat{\mathbf{c}}$ encrypting the product $m_0 \cdot m_1$. That is, $\text{RLWE}(m_0) \boxtimes \text{GSW}(m_1) = \text{RLWE}(m_0 \cdot m_1)$. Notice that performing this operation efficiently relies crucially in having an efficient way of multiplying polynomials, which is achieved by using Fast Fourier Transform (FFT) or algorithms alike, such as the Number Theoretic Transform (NTT). Moreover, the bootstrapping is essentially a sequence of n external products, thus, the NTT is the main piece to be optimized in order to obtain faster bootstrapping (and hence, faster homomorphic evaluation of any function, since bootstrapping dominates the running time of any homomorphic computation).

2.3 FINAL

FINAL [BIP⁺22] is an FHE scheme similar to TFHE but constructed using the LWE and NTRU problems instead of the RLWE. As TFHE, it encrypts an integer m into an LWE sample $\hat{\mathbf{c}} = (\mathbf{a}, b) \in \mathbb{Z}_q^{n+1}$, where $b = \mathbf{a} \cdot \mathbf{s} + e + m \bmod q$ and $\mathbf{s} \in \mathbb{Z}_q^{n+1}$ is the secret key. But to bootstrap $\hat{\mathbf{c}}$ (that is, to perform the blind rotation, followed by a key switching), it relies on the NTRU problem to define two types of ciphertexts, a scalar one, which is of the form $c = g_0 \cdot f^{-1} + m_0 \in \mathcal{R}_q$ and a vector ciphertext, which is of the form $\mathbf{c} = \mathbf{g} \cdot f^{-1} + m_0 \in \mathcal{R}_q^\ell$, where each $\mathbf{g}[i] \cdot f^{-1} \in \mathcal{R}_q$ is an NTRU sample. Then, its external product, $c \boxtimes \mathbf{c}$, is defined as $\text{SignedDecomp}_B(c) \cdot \mathbf{c} \in \mathcal{R}_q$, that is, we decompose c into a vector of ℓ small polynomials and compute the dot product with \mathbf{c} . This gadget decomposition can also be performed approximately, in which case it is called approximate gadget decomposition.

Notice that instead of having a matrix of dimension $2 \times 2\ell$, as in TFHE, we now have a vector of dimension ℓ . Hence, the external product is simplified to a dot product instead of a matrix-vector product, which tends to reduce its costs (depending on the size of the parameter ℓ). Moreover, for FINAL, the bootstrapping key, bsk , is composed by a list of n vector ciphertexts, therefore, it has $n\ell$ elements of \mathcal{R}_q , while in TFHE, bsk is a list of n GSW ciphertexts, which are $2 \times 2\ell$ matrices, therefore it has $4n\ell$ elements of \mathcal{R}_q in total. Thus, FINAL tends to have smaller keys. Finally the blind rotation is followed by a key switching step, analogous to TFHE.

2.4 The Number Theoretic Transform

Multiplying two polynomials with coefficients modulo some integer q can be done efficiently using the Number Theoretic Transform (NTT). The “standard” NTT consists of the following function, which takes an input polynomial/vector x of N elements and returns an output vector/polynomial X of N elements where the j -entry is

$$X[j] = \sum_{i=0}^{N-1} x[i] \times \omega^{ij}$$

with $\omega^N \equiv 1 \pmod{q}$ and $\omega^{N/2} \equiv -1 \pmod{q}$.

When we wish to multiply two polynomials modulo $X^N + 1$, so that all polynomials consists of N coefficients, we can choose between three possible methods. A naive method uses two polynomials of ring size $2N$, with the most significant half of the coefficients being zeros. A coefficient-wise multiplication follows after the NTT, which is followed by a reduction step of the polynomial modulo $X^N + 1$ [POG15], followed by an INTT.

A second method, to avoid padding inputs with zeros, *negative wrapped convolution* [POG15] is utilized, taking ψ so that $\psi^2 = \omega \pmod{q}$, and precomputing:

$$\hat{a} = (a[0], a[1]\psi, a[2]\psi^2, \dots, a[n-1]\psi^{n-1})$$

and

$$\hat{b} = (b[0], b[1]\psi, b[2]\psi^2, \dots, b[n-1]\psi^{n-1})$$

With these precomputed values, we can now perform the NTT and pointwise multiplication, followed by an inverse NTT:

$$\hat{c} = \text{INTT}(\text{NTT}(\hat{a}) \circ \text{NTT}(\hat{b}))$$

then “remove” the extra powers of ψ by computing

$$c = (\hat{c}[0], \hat{c}[1]\psi^{-1}, \hat{c}[2]\psi^{-2}, \dots, \hat{c}[n-1]\psi^{-(n-1)})$$

and it turns out that $c = a \times b \pmod{X^N + 1}$.

A third method involves integrating this multiplication with ψ or ψ^{-1} in the NTT butterflies [POG15]. We have:

$$\text{NTT}(\hat{a}) = \sum_{i=0}^{N-1} \hat{a}[i] \times \omega^{ij} = \sum_{i=0}^{N-1} a[i] \times \psi^i \times \omega^{ij} = \sum_{i=0}^{N-1} a[i] \times \psi^{2ij+i}$$

To ensure we perform the NTT efficiently, an algorithm similar to the Fast Fourier Transform is employed (for the basics of the FFT, see [BM67]). The crucial difference between the Fast Fourier Transform and the NTT lies in the arithmetic, rather than the control flow (for software) or architecture (for hardware). The arithmetic of the NTT differs from the Fast Fourier Transform in the sense that the NTT handles integers modulo q . As such, rather than a complex multiplication, a modular multiplication is performed in each butterfly unit. Beyond this change in arithmetic, the structure of the NTT is identical to the Fast Fourier Transform, and all techniques used to optimize the latter can be applied to the former.

The key Processing Element of the FFT/NTT architecture is the Butterfly Unit. This Processing Element consists of an addition, a subtraction and a multiplication, connected to each other in a way that suggests the shape of a butterfly [BNV24]. In the Number Theoretic Transform, as we work over integer modulo q , the butterfly unit consists of a modular addition, a modular subtraction and a modular multiplication respectively.

Table 1: Parameters for 128-bit secure FINAL

n	N	q	l
610	1024	$786433 < 2^{20}$	7

Table 2: AMD Alveo U55c resources

LUTs	DSPs	FlipFlops	BRAM+URAM memory
1.304M	9024	2.607M	43 MB

2.5 A computational cost comparison between TFHE and FINAL

The difference between TFHE and FINAL from a hardware perspective lies primarily in the number of NTTs performed. Whereas for common parameters TFHE requires 6300 NTTs for each bootstrapping operation, FINAL requires only 4880 NTTs for similar parameter sets. For a detailed comparison between the algorithms, and how these numbers are derived, see Appendix A.

2.6 Parameters

The FINAL parameters used in this work are provided in Table 1. We note that independently of the hardware optimisations made in this paper, FINALs default 20-bit modulus already provides a nice hardware saving compared to the 32-bit moduli often favored by software-oriented parameter sets.

2.7 FPGA preliminaries

An FPGA is a reconfigurable electronic circuit. On an FPGA, there are various “primitives” such as LUTs, DSPs, Flipflops and BRAMs which can be connected to each other according to the wishes of the hardware designer. LUTs or Look-Up Tables are primitives which takes as input a number of bits and provide a programmable output for any possible combination of these inputs. In the case of the AMD Alveo U55c, 1 output bit when utilizing 6 bits as input, or 2 output bits for 5 input bits. In other words, any 6:1 Lookup Table on the U55c can be seen as a memory element which stores 64 bits on 64 different addresses, and when provided a 6 bit address (hence 64 possible addresses) will give the data bit stored at that address. DSPs or Digital Signal Processors are elements which primarily function as multiplication units. While multipliers can be built with Look-Up Tables only, multiplication is such a common operation that DSPs are provided to optimize this specific operation. On the U55c, DSPs implement an 18 by 27 bit multiplier, along with a few additions. Flipflops are synchronous one bit storage elements. This means that they will store one bit every clock cycle, which is used to reduce the path signals travel through between clock cycles, and thereby increase the frequency of the clock, which in turn results in higher computation speeds. Finally BRAMs or Block Random Access Memories and URAMs or Ultra Random Access Memories are the memories available on the FPGA. They provide large-scale memory storage.

For the purposes of this paper, we will only consider the AMD Alveo U55c, but the techniques are applicable to any FPGA or device with synthesizable Look-Up Tables. Moreover, the Verilog Code implementing this design will synthesize for any 7-series or Ultrascale+-series device. The resources available on the U55c are listed in Table 2

3 Related Work

In this section, we discuss two state-of-the-art designs for TFHE-like bootstrapping. The approach and resulting performance is discussed. This will provide a basis on which the optimizations of our design (**Quatorze-bis**) can be discussed.

3.1 FPT

FPT is a streaming processor by Van Beirendocnk et al. [VDTV23] which accelerates both the blind rotation and the key switching step. To perform the many polynomial multiplications that make up the bulk of the bootstrapping step, it utilizes the FFT.

- As FPT has a streaming-like architecture, it's design is pipelined and thus multiple ciphertexts (12 ciphertexts) are handled simultaneously
- FPT utilizes multiple FFT techniques such as optimized real to complex fourier transforms and folded coefficients to gain a factor 4 improvement over regular FFTs
- FPT utilizes what it calls a "continuous-flow pipelined FFTs", an architecture containing a single radix-64 (for their IFFT) or radix-128 (for their FFT) processing element. These two huge processing elements guarantee a better throughput per area than multiple smaller radix FFTs in parallel or a single FFT with multiple smaller processing elements
- FPT considers the inherent noise of the FFT and the inherent noise of TFHE and optimizes the FFTs fixed point arithmetic to prevent the calculation of data that will be lost in TFHE's inherent noise

Overall then, this combination of techniques results in a state of the art bootstrapping throughput of 28400 Bootstraps per second (BPS) for 128 bit security.

3.2 NFP

Through the hardware accelerator NFP, Li et al. [LLW⁺24] presented a novel TFHE algorithm, utilizing both the NTRU-based bootstrapping from FINAL and approximate gadget decomposition. Consequently, their algorithm requires only half the amount of Number Theoretic Transforms utilized in FINAL.

- NFP has a CPU-like implementation, with an I(NTT) module, a multiply-accumulate module and other modules being utilized sequentially. Thus no pipelining of ciphertexts is done, and the throughput is inversely proportional to the latency.
- NFP places a large quantity of radix-2 butterflies rather than a single large-radix processing element. This means that the ratio of butterflies to control and crossbar logic is much less favorable than in FPT. NFP places only 256 butterfly units on the FPGA, as opposed to the $8 \times 128/2 + 8 \times 64/2 = 768$ butterfly units utilized by FPT.
- NFP utilizes Montgomery Multiplication for its modular multiplication modules. Montgomery Multiplication can be an efficient [BAE⁺24] method for modular multiplication where the modulus changes during runtime. However, its naive implementation generally results in unnecessary area usage when used in hardware for a fixed modulus. This in turn results in larger butterfly units, and thus less butterfly units that can be placed on the FPGA.

The combination of a lower quantity of butterflies as well as the fact that the butterfly units of NFP are not permanently in use means that an overall throughput of only 3448

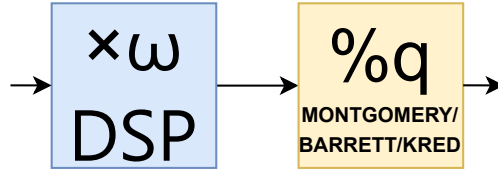


Figure 1: The basic layout of a normal modular multiplier

Bootstraps Per Second (BPS) [LLW⁺24] is reached. While the modified blind rotation algorithm presented in NFP is thus excellent and offers a factor two improvement over FINAL in Number Theoretic Transform calculations, the hardware design implemented is decidedly low throughput. We note that the authors of NFP mistakenly assume that FPT does not pipeline, and assume that since FPTs latency is 0.58ms, its throughput must be 1724 Bootstraps Per Second (BPS) (Table 12 of [LLW⁺24]). In reality, FPT handles 12 ciphertexts per batch, and as such its throughput is more than 12 times higher than the number cited by the NFP paper. In short, they fail to see the difference between throughput and latency.

Having considered the two designs of this section, we looked towards creating a design with both a well-optimized algorithm, such as FINAL, and a well-optimized implementation.

4 Design

A well optimized streaming architecture-style implementation of FINAL requires three design considerations: Firstly, efficient Processing Elements for the Number Theoretic Transform, which is the bottleneck operation. Secondly, an efficient architecture for the NTT that minimizes the area taken up by elements which are not Processing Elements. Lastly, an architecture for our FINAL design **Quatorze-bis** that guarantees the other elements of our design can feed sufficient data to our large NTT accelerators every cycle.

4.1 NTT butterflies

The critical part of any butterfly unit is usually its modular multiplication unit. In this section we will first consider the methods of the designs, then describe the multiple approaches our design takes to optimally utilize the FPGA resources.

Most NTT designs take the approach shown in Figure 1. The multiplication is usually performed using one or more DSPs. For the modular reduction, the usual design chooses Montgomery Multiplication [MKO⁺20]. By using Montgomery Multiplication, most designs for FINAL-like accelerators fail to leverage an important feature of FINAL-like schemes: the constant modulus q . Unlike BGV or CKKS, the constant modulus used in the blind rotation operation offers a great optimization opportunity. Firstly, even in well-known algorithms such as Montgomery or Barrett, exploiting the constant modulus can give significant improvement [XL21]. Secondly, there are only two requirements for the FINAL modulus: its size must be around a certain amount of bits, and if we wish to be efficient, it must support the Number Theoretic Transform. To support a Number Theoretic Transform, the modulus should be a prime or product of primes of the form $1 \bmod 2N$. This allows us to choose a modulus q as long as we satisfy both requirements and gain a large area improvement over other designs, which in turn will lead to a speed up by greatly increasing the number of butterflies which can be placed on the FPGA.

As discussed in the preliminaries in table 1, for the FINAL parameters $2^{19} < q < 2^{20}$ and $q \equiv 1 \pmod{2N} \equiv 1 \pmod{2^{11}}$. Given that q is a constant which can be chosen to have a very low hamming weight (for instance, $q = 786433 = 3 \times 2^{18} + 1$), all operations with q can be implemented efficiently.

In our design we exploit q 's structure in two different modular multiplication techniques: K-reduction, which is a technique more often seen in Post-Quantum Cryptography [BNV24] and LUT-based modular multiplication, a new technique based on modular reduction. K-reduction will be applied in scenarios where both operands of our multiplier change dynamically, whereas LUT-based modular multiplication will be applied wherever only one operand varies, as our novel LUT-based modular multiplication requires no DSPs for either the multiplication or the reduction step.

4.1.1 K-reduction

K-reduction of a number c (for FINAL parameters, a $2 \times \log_2 q = 40$ -bit number) exploits the fact that our low Hamming weight modulus q can be written as a power of two multiplied with a small constant k plus one (in the case of $q = 3 \times 2^{18} + 1$, we have $k = 3$). A modular reduction of numbers that are to be multiplied by k is then a low-complexity operation because any number multiplied by $k \times 2^{18}$ is equivalent to the negative of that number. In other words, the upper bits, starting from position 18 (the notation for these upper bits is written here as $c_{[39:18]}$), can be subtracted from the lower bits ($c_{[17:0]}$), once the lower bits have been multiplied by k (See Algorithm 1). Thus, rather than calculating the desired result:

$$c \bmod q$$

The following is calculated (which is off by a constant factor $k = 3$):

$$\mathbf{K-red}(c) = c \times k \bmod q$$

with $q = 786433 = 3 \times 2^{18} + 1 = k \times 2^x + 1$ with $x = 18$ and $k = 3$. By splitting $k \times c$ into $k \times c_{[39:18]} \times 2^{18} + k \times c_{[17:0]}$, and replacing the upper part $k \times 2^{18}$ with -1 , the end result is a single subtraction $-c_{[39:18]} + k \times c_{[17:0]}$ to reduce the size of the product by almost 18 bits. The fact that the calculation is off by a constant factor k is rectified by multiplying one of our inputs (either our twiddle factor or the bootstrapping key) by a constant factor $k^{-1} \pmod{q}$.

Algorithm 1: K-red Algorithm

Data: a and b , 20 bit inputs, $q = k \times 2^x + 1$ with $x = 18$ and $k = 3$ so that $q = 786433$ and $k \times 2^{18} \equiv -1 \pmod{q}$

Result: $r = a \times b \times k \bmod q$

```

1 begin
2    $c = a \times b = c_{[39:18]} \times 2^{18} + c_{[17:0]}$ ;
3    $r = (c_{[39:18]} \times (-1) + k \times c_{[17:0]}) \bmod q$    ( $= k \times c \bmod q =$ 
       $(c_{[39:18]} \times k \times 2^{18} + k \times c_{[17:0]}) \bmod q$ );
4 end

```

K-reduction does not fully reduce a 40-bit value to a 20-bit value since $c_{[39:18]}$ is a 22-bit value, but the final steps of the modular reduction can be handled by a branching conditional statement, or more efficiently by exploiting the fact that $2^{20} \equiv 2^{18} - 1 \pmod{q}$ to reduce the final result by the required two bits, followed by a single modular subtraction. The exact implementation of this K-reduction algorithm can be seen in Figure 2.

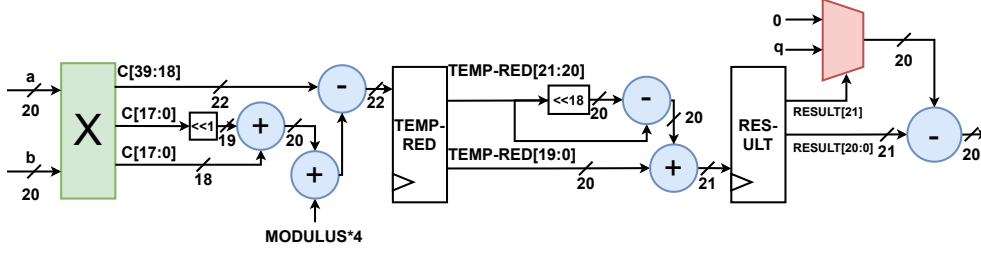


Figure 2: K-Reduction as implemented in our design

Full utilization figures will be provided in section 5 but we note that the K-reduction design (without the multiplier) requires 89 LUTs and 105 Flipflops. The 20-bit multiplier element can be synthesized by commercial tools as requiring either 2 DSPs or 1 DSP and 47 LUTs. We will choose the former as the next section will demonstrate a multiplier which requires no DSPs, leaving most unused.

4.1.2 LUT-based modular multiplication

We now present a novel modular multiplication method for modular multiplication by a constant with reduction by a constant modulus. This method is similar to in-memory computation, but utilizes the LUT primitives available on the FPGA and takes advantage of the inherent linearity of both the multiplication and modular reduction operation. This method is not to be confused with LUT-based reduction [BNV24], as LUT-based reduction only performs the reduction operation, not the multiplication operation.

Consider the following function:

$$f(a) = a \times b \bmod q$$

where b and q are 20-bit constants.

We see that f is a linear function by looking at the evaluation of an input $C_0 \times a_0 + C_1 \times a_1$:

$$f(C_0 \times a_0 + C_1 \times a_1) = (a_0 \times C_0 \times b \bmod q + a_1 \times C_1 \times b \bmod q) \bmod q$$

Moreover any $\log_2(q) = 20$ -bit integer a can be split up in 4 sections of 5 bits.

$$a = a_{[19:15]} \times 2^{15} + a_{[14:10]} \times 2^{10} + a_{[9:5]} \times 2^5 + a_{[4:0]} \times 2^0 \quad (1)$$

Thus our modular multiplication reduces to $\log_2(q)/5 = 4$ functions with 5-bit inputs.

$$f(a) = (f(a_{[19:15]} \times 2^{15}) + f(a_{[14:10]} \times 2^{10}) + \dots) \bmod q$$

When we consider each term individually, we see that these functions take 5 bit inputs and return 20 bit outputs.

$$f(a_{[19:15]} \times 2^{15}) = g_3(a_{[19:15]}) = a_{[19:15]} \times b \times 2^{15} \bmod q$$

$$f(a_{[14:10]} \times 2^{10}) = g_2(a_{[14:10]}) = a_{[14:10]} \times b \times 2^{10} \bmod q$$

$$f(a_{[9:5]} \times 2^5) = g_1(a_{[9:5]}) = a_{[9:5]} \times b \times 2^5 \bmod q$$

$$f(a_{[4:0]} \times 2^0) = g_0(a_{[4:0]}) = a_{[4:0]} \times b \times 2^0 \bmod q$$

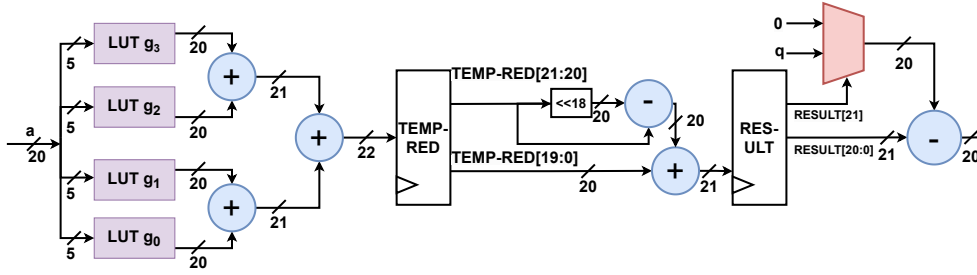


Figure 3: LUT Modular Multiplication

It is now important that the primitive logic element on most AMD FPGAs is a Look-up table containing 5 bits of input and 2 bits of independent output. From 10 Look up Tables, each taking 5 bit inputs and outputting 2 bits, we can create the required g_i function. From four of these functions, we can create the four terms of Equation (1).

The only step which remains is to then sum these four terms (as an extra optimization, we used an efficient 3 to 1 adder identical to FloPoCo [dDK24]), and to reduce the resulting 22-bit sum in the same fashion as reduced the 22-bit sum in the K-reduction case. The full process is shown in Figure 3.¹ In our design, we employed lazy reduction wherever possible to remove the 22-bit reduction step after the multiplication and the reduction steps after addition and subtraction. This reduces the total LUT usage of our butterfly circuit to somewhere between 100 and 140 LUTs.

There are some important advantages and disadvantages to this method. A great advantage is that this method removes the need for DSPs or other types of multipliers for any modulus, as long as that modulus is constant. Whereas K-reduction requires the modulus to have the structure $k \times 2^x + 1$ and to have a low k , preferably 3, this LUT-based modular reduction offers better performance with fewer requirements for the modulus structure. Namely, the modulus is not required to have a low Hamming weight or a structure of the form $k \times 2^x + 1$. The main disadvantage is that one of the multiplicands must be a constant, which limits its application. A point-wise multiplication is generally a part of any application where the Number Theoretic Transform is performed, and FINAL is no exception. Since a pointwise multiplication in FINAL is done between our accumulated ciphertext and the bootstrapping key, neither of which is a constant, this method is not applicable for this pointwise multiplication. Moreover, the Number Theoretic Transform architecture must be designed in such a way that the twiddle factors fed in to the butterflies are constants.

We will see in the next section that we can design an architecture which primarily employs constant twiddle factors, and requires only as many non-constant twiddle factor multipliers as the number of elements that are fed into the circuit each clock cycle. We will call this number the Streaming Width S . K-reduction will then provide S multipliers and LUT-based reduction will provide the remaining $S \log_2 N$ multiplier units for each NTT unit handling input polynomials with ring size N .

¹It might at first sight seem advantageous to employ the function g_0 for all four cases, and shift afterwards, but before addition. We note that since the function g must still take 4 different inputs, this doesn't improve the area delay product. Moreover, one of the terms of the addition will contain 35 bits, so extensive reduction steps would still have to be taken to achieve a 20-bit end result.

4.2 NTT architecture

In this section, we will consider the NTT architecture in light of the previous section, and discuss various possibilities which satisfy our requirements.

4.2.1 The impossibility of a fully implemented NTT

The key specification to designing the NTT architectures, given our butterflies, thus becomes ensuring that each butterfly is assigned to one twiddle factor. The aforementioned NTT butterflies must also be fed with the input data in a way which minimizes non-butterfly logic. Interestingly, this is one of the few examples of hardware problems that become easier as the throughput increases. Each individual input of the Number Theoretic Transform circuit has fewer pieces of data that need to be fed in as the circuit size increases. As such, the overhead of multiplexing logic decreases relative to the NTT. In the ideal scenario, the streaming width is equal to the ring size, which would require no overhead for feeding in data outputs. Unfortunately, such a design would exhaust the resources of the FPGA by itself. Consider a 240 LUT butterfly unit: with a ring size of $N = 1024$, our $N \times \log(N) = 1024 \times 10 = 10240$ butterflies require 2457600 LUTs. Placing even a single NTT is beyond the capabilities of a U55c FPGA, and even if it were possible to place a single NTT and a single smaller INTT similar to FPT, balancing the NTTs and INTTs in a manner similar to FPT is extremely difficult for the given parameter set in which $l = 7$ as l does not divide $N = 1024$ [VDTV23].

4.2.2 Our NTT-1024 architecture

As we cannot implement the full NTT, we must split our task into smaller sub-NTTs. One way of achieving this is the multi-dimensional NTT, where our input vector is split into two dimensions [CG19]. One of the two dimensions will be S , our streaming width. The other dimension is $\frac{N}{S}$, with N as always our ring dimension. From a data perspective this looks as follows:

$$[x_0 \ x_1 \ x_2 \ x_3 \ x_4 \ x_5 \ x_6 \ x_7 \ \dots] \equiv \begin{bmatrix} x_0 & x_1 & \dots \\ x_S & x_{S+1} & \dots \\ \vdots & \vdots & \ddots \end{bmatrix}$$

S is chosen to divide N , and must therefore also be a power of two. Our NTT-operation then (roughly) consists of performing $\frac{N}{S}$ NTTs of size S , followed by performing S NTTs of size $\frac{N}{S}$. This is consistent with how the multidimensional NTT is performed. From a mathematical perspective, take the definition of the NTT transformation of a vector x to X :

$$X[j] = \sum_{i=0}^{N-1} x[i] \times \omega^{ij}$$

If we now set $i = a \times N/S + b$ and $j = c \times S + d$:

$$X[c \times S + d] = \sum_{a=0}^{S-1} \sum_{b=0}^{N/S-1} x[a \times N/S + b] \times \omega^{(a \times N/S + b) \times (c \times S + d)}$$

Before proceeding, we note that as in the preliminaries, $\omega^N \equiv 1 \pmod{q}$, thus $(\omega^N)^{a \times c} \equiv \omega^{a \times c \times N} \equiv 1 \pmod{q}$. Thus:

$$X[c \times S + d] = \sum_{a=0}^{S-1} \sum_{b=0}^{N/S-1} x[a \times N/S + b] \times \omega^{ac \times N + bc \times S + ad \times N/S + bd}$$

$$\begin{aligned}
&= \sum_{a=0}^{S-1} \sum_{b=0}^{N/S-1} x[a \times N/S + b] \times \omega^{bc \times S + ad \times N/S + bd} \\
&= \sum_{b=0}^{N/S-1} \omega^{bc \times S + bd} \sum_{a=0}^{S-1} x[a \times N/S + b] \times \omega^{ad \times N/S} \\
&= \sum_{b=0}^{N/S-1} \omega^{bc \times S} \omega^{bd} \sum_{a=0}^{S-1} x[a \times N/S + b] \times \omega^{ad \times N/S} \tag{2}
\end{aligned}$$

We can now see the inner sum is equivalent to performing an S -sized for every possible value of b (there being N/S possible values of b , so this will be N/S NTT of size S). There is then a pointwise multiplication with ω^{bd} , followed by an N/S -sized NTT, which must be done for every value of d , of which there are S different possibilities.

As we cannot implement a streaming width S of 1024, we select the highest possible power of two streaming width S which results in a FINAL design that can be implemented at a high frequency. For the U55c, this design has a streaming width of 64 coefficients.

4.2.3 The matrix tranpose with BRAMs

Our streaming width S is called the “streaming width” because it corresponds to the exact number of elements which will flow through our Number Theoretic Transform at any given moment, namely $S = 64$ values of $\log_2(q) = 20$ bits each. If our NTT is pipelined, it then follows logically that every $N/S = 16$ cycles a new vector may be loaded in at the inputs, and a new vector is streamed out at the output.

Our initial NTT is of the form given in Equation (2), namely $\sum_{a=0}^{S-1} x[a \times N/S + b] \times \omega^{ad \times N/S}$. This means that the values fed into our NTT-64 module during the first clock cycle should be of the form:

$$\text{CLK \#0: } x[0], x[16], x[32], x[48], \dots$$

However, the standard representation of data outside the NTT requires data to be of the form:

$$\text{CLK \#0: } x[0], x[1], x[2], x[3], \dots$$

We represent this data as a matrix where one dimension is time (i.e. the index of the clock cycle) and the other dimension is space (i.e. the index of the parallel track, one of the S paths in our design). It is then clear that we must perform the equivalent of a matrix transpose to output data to the first NTT correctly. Following the logic of Equation (2), a backward matrix transpose must follow the pointwise multiplication, and after all NTT-16s have been performed, a matrix transpose identical to the first must once again be performed. The process for the matrix transpose is similar to the one described in [RV08]², but in this design is performed for non-square transposes.³

4.2.4 The full design

Conceptually, the design then consists of a single NTT-64 (A number Theoretic Transform with a streaming width of 64 coefficients), followed by a multiplication with twiddle factors (this is the pointwise multiplication with ω^{bd}), followed by 4 parallel NTT-16’s. Before, in-between, and after the NTT’s, a transpose must be performed to ensure coefficients are fed in the right order. The design can be seen in Figure 4.

²Out of personal preference, we mapped addresses sequentially rather than employing xor, but the effect is identical

³To create a rectangular transpose module in dimensions S by N/S , one can simply remap the inputs to $\frac{S}{N/S} = \frac{S^2}{N} = 4$ different instances of a N/S by N/S matrix transpose.

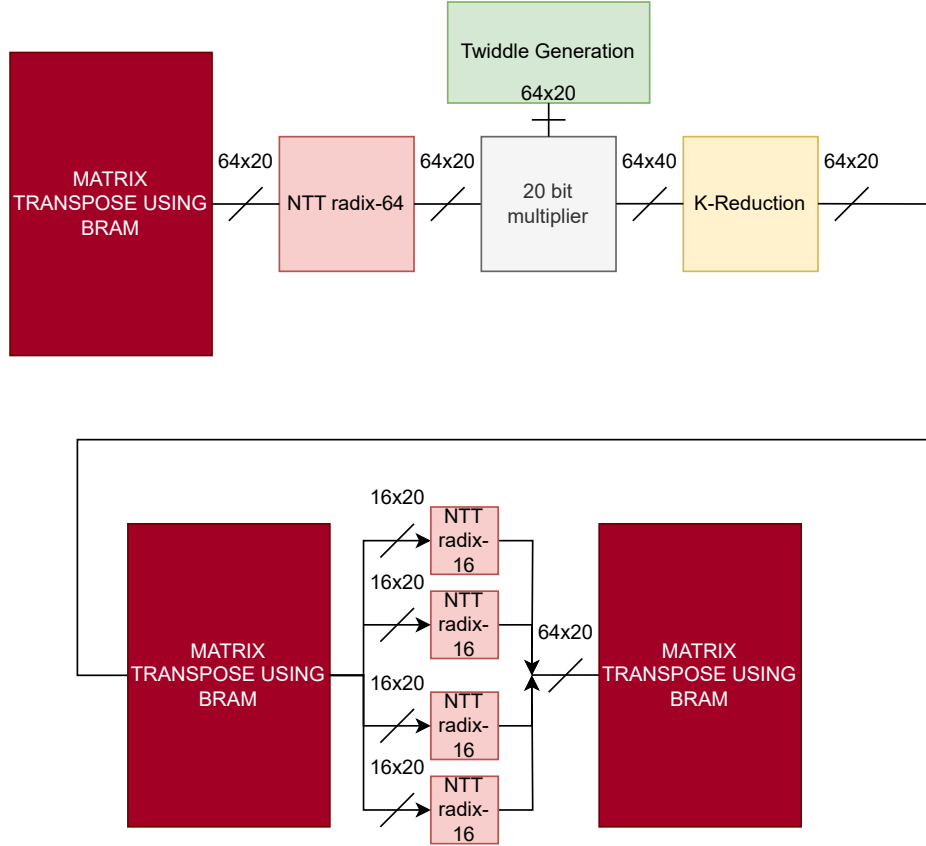


Figure 4: Datapath of our NTT design

We note that the final NTT design has an additional complication: the integration of the *negacyclic convolution*. Nonetheless, the multiplication with ψ can still be integrated into the NTT, by performing the calculation above with \hat{x} rather than x and swapping out multiplication with ω for multiplication with some power of ψ as required [POG15]. To maximize the number of ω^0 terms (which require no multiplier), we integrated multiplication with ψ into the NTT-64 butterflies and the pointwise multiplication (light red and grey boxes in Figure 4), but not in the NTT-16 elements.

4.3 FINAL and Quatorze-bis

The basic FINAL datapath is given by Figure 5, which is the hardware datapath of Algorithm 3. The algorithm consists of a CMUXing step, which depends on the input vector \mathbf{a} , followed by the external product with the bootstrapping key. The external product itself consists of a decomposition, followed by a polynomial multiplication. To perform the polynomial multiplication efficiently, we perform an NTT for each decomposed value, followed by a pointwise multiplication with the bootstrapping key, which is followed by an INTT. A common optimisation to all TFHE-like schemes is to take advantages of the linearity of the Inverse NTT, and to perform the INTT after all polynomial products have

been summed together. The decomposition to a smaller base is efficiently implementable in hardware and requires a negligible amount of logic⁴. The CMUX has a similarly negligible area, but its design is less straightforward and will thus be discussed briefly.

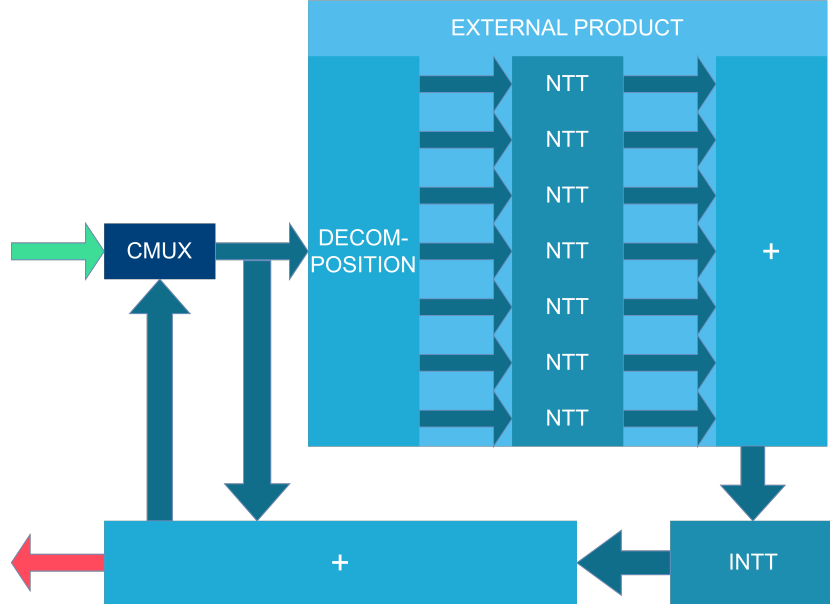


Figure 5: Datapath of **Quatorze-bis**, our FINAL hardware accelerator

4.3.1 CMUX

The CMUX circuit must take the inputs and provide the correct coefficient a_i at any given iteration (See Algorithm 3). This can be implemented simply by a BRAM and a S -to-1 multiplexer. Having obtained the a_i value, it must utilize this value to calculate $\mathbf{ACC} \times (X^{a_i} - 1)$. We wish to perform the CMUX operation in a pipelined fashion with a streaming width of S so our CMUX circuit matches our external product circuit. Therefore, we combine a barrel rotator with a BRAM to ensure that S coefficients indexed at a_i of \mathbf{ACC} are matched with S coefficients from the same \mathbf{ACC} indexed at 0 in an adder/subtractions unit.

4.3.2 Pipelining

Our circuit is fully pipelined. As a single iteration (1 CMUX + NTT + Multiply + INTT + Sum) takes slightly less than 512 clock cycles, and a new ciphertext can be fed into the circuit after $N/S = 16$ cycles, we logically chose for a pipeline depth of 32 ciphertexts. For $n = 610$ iterations, it thus takes $512 \times 610 \approx 305000$ clock cycles to complete bootstrapping for 32 ciphertexts. The circuit thus has a "FINAL" latency of 1 ms when clocked at 305 Mhz, for a throughput of 32000 Bootstraps per Second.

5 Results

In this section, we will consider the various design choices made in this design and other state-of-the-art designs and see how each choice impacts the final performance. We begin

⁴78 LUTs for our parameter set per decomposition element, for a total of $S \times 78 = 64 \times 78 < 5000$ LUTs, which is less than 0.5 % of the total FPGA area

Table 3: Our design **Quatorze-bis** versus state-of-the-art Hardware Accelerators

Name	Platform	LUTs	DSPs	FFs	BRAMs	f (MHz)	Lat. (ms)	TP (BPS)
FPT I	FPGA	526 K	5494	916 K	17.5 Mb	200	0.66	28400
FPT II	FPGA	595 K	5980	1024 K	14.5 Mb	200	0.74	25000
YKP [YKP22]	FPGA	442 K	6910	342 K	409Mb	180	1.88	2700
NFP II [LLW ⁺ 24]	FPGA	891 K	4508	217 K	33 Mb	-	0.29	3448
Quatorze-bis	FPGA	850 K	1924	1.4M	160Mb	305	1.02	31250
Xiao et al. [XLK ⁺ 24]	GPU	NVIDIA RTX4090				-	0.23	4350
MATCHA [JLJ22b]	ASIC	36.96 mm ² 16 nm PTM				2000	0.2	10000
FINAL [BIP ⁺ 22]	CPU	3.1 GHz processor				3100	48	20.8

Table 4: Utilization of various modules of **Quatorze-bis**

Name	LUTs	DSPs	FFs	BRAMs
Available on U55c	1.3M	9024	2.6M	41MB
Full Design	850K	1924	1.4M	20MB
FINAL Fig. 5	650K	1920	1.1M	18MB
CMUX	12K	0	15K	72KB
NTT Fig. 4	72.5K	128	82K	0
NTT-64	29K	0	21K	0
NTT-16	3.8K	0	3.6K	0

by considering a blind rotate as a number of butterfly units that must be computed, we discuss how different schemes and FFT/NTT choices affect this number and finally we consider how many butterfly units each state-of-the-art accelerator places on chip. This method of analysis removes many complications introduced by the varying techniques of different designs, and allows the impact of each technique to be considered individually.

In the subsequent sections, we will consider the reason behind each of the given factors. As main comparison design, we will look at FPT, as it is the only design that comes anywhere close to the throughput achieved in our design **Quatorze-bis** (see Table 3)

5.1 A computational comparison with FPT

One straightforward way of considering the throughput of a pipelined design is to look at how many operations must be performed, look at how many operators are available and divide the latter by the former and multiply by the frequency to get the throughput. In our case, we have $S/2 \times \log N \times (l+1) = 32 \times 10 \times 8 = 2560$ butterflies (See Figure 4 and Figure 5). In total we must perform $n \times (l+1) \times N/2 \times \log N = 610 \times 8 \times 512 \times 10 = 24985600 = 25$ million butterflies. Our operating frequency is $f = 305$ MHz, so we are using our butterflies 305 million times per second. As our all our butterflies are used all the time, we can thus perform 31250 Bootstraps Per Second.

As previously mentioned in Section 3, FPT [VDTV23] places $8 \times 128/2 + 8 \times 64/2 = 768$ butterflies on the FPGA. They present two different TFHE parameter sets: a benchmarking parameter set of 110-bit security common to many hardware accelerators (FPT II) and their own optimized parameter set providing 128 bits of security (FPT I). We focus on their optimized parameter set, as it achieves the highest performance **and** the highest security. For this parameter set, their initial vector size N is equal to 512 as opposed to the more common N utilized in this paper of 1024. They use both the negacyclic convolution trick and a trick unique to FFTs to reduce the FFT size down to $N/2$ by mapping real coefficients to complex coefficients. Moreover they have an $l = 2$ and have $(k+1) \times (k+1) = 3 \times 3$ polynomials. This in turn means that they must perform

6 FFTs and 3 IFFTs per iteration, for a total of 586 iterations, therefore, a total of $n \times (l \times (k + 1) + (k + 1)) \times N/2 \times \log N = 586 \times (6 + 3) \times 128 \times 8 = 5400576 = 5.4$ million butterflies must be performed. At a frequency of 200 Mhz, they can perform 28441 Bootstraps Per Second.

From the above analysis, it is immediately clear that FPT needs to perform less than a fourth of the butterflies our design must perform (namely, 5.4 M butterflies versus our 25M butterflies). It is also immediately apparent that FPT has severely fewer Butterfly Processing Elements on the chip.

As FPTs FFT works on the complex domain, their butterflies require complex additions, subtractions and multiplications. The additional area required is offset by the fact that an FFT can fold real coefficients into the imaginary coefficients, and thus requires only half the number of butterflies [VDTV23]. It is therefore reasonable to state that an FFT butterfly is worth two NTT butterflies.

Because the NTT butterfly requires modular multiplication, commonly implemented with a technique such as Montgomery Multiplication, the NTT butterfly usually requires more logic to implement than an FFT butterfly despite being half as useful. As the NTT in our design is implemented without using DSPs and using a minimum of LUTs (an average of about 120 LUTs), we can afford to fit almost four butterflies per FPT butterfly (fitting 2560 butterflies versus 768 butterflies).

We now list the three factors affecting our and FPTs final throughput, and go over how each affects performance.

1. FPT must perform 5.4M FFT-style butterflies, while our design must perform 25M NTT-style butterflies. If we consider a theoretical FFT butterfly to be worth 2 NTT butterflies, our design still has more than twice the computational load as FPT.
2. FPT fits only 768 butterflies, while our design fits 2560 butterflies. If we again consider an FFT butterfly to be worth 2 NTT butterflies, our design still fits twice the amount of butterflies as FPT, thanks to our innovative modular multiplication design
3. Our operating frequency is 305 MHz, while FPTs operating frequency is 200 MHz. This pushes the throughput of our design 10% above that achieved by FPT I. This comes at the cost of latency: while FPTs design batches only 12 ciphertexts, we batch 32 ciphertexts. As a single bootstrap consists of only a single bit operation, throughput is the more salient question when comparing TFHE/FINAL-like designs.

The conclusion is that our design **Quatorze-bis** beats FPT by a factor two on *hardware design*, that is, there are twice as many NTT-style Butterfly Processing Elements per given LUT. But FPT beats our design by a factor two on *algorithm design*, thereby approximately evening out the final throughput.

5.2 Calculating the Throughput Per DSP

The utilization of various elements our design is given by Table 4. As can be seen, our design utilizes only 1920 DSPs, all of which are solely used for the pointwise multiplication with the bootstrapping key ($l \times S = 7 \times 64 = 448$ multipliers using 2 DSPs per multiplier) or multiplication with the twiddle factors in between the NTT blocks (see 20-bit multiplier block in Figure 4, which is utilized $(l + 1) \times S = 64 \times 8 = 512$ times given we have $l + 1$ NTTs).

Our LUT utilisation consists primarily of our Number Theoretic Transform modules. Each module implementing Figure 4 takes up ≈ 72500 LUTs (and $2 \times S = 128$ DSPs). Of

these, 44000 LUTs are butterfly units, 22000 LUTs make up the matrix transposes and 6500 LUTs plus 128 DSPs the multiply accumulate units. In total, out of the 850 000 LUTs used for the design, $(l + 1) \times 44000 = 352000$ LUTs are dedicated to butterfly units. This number is comparable to FPT, which utilizes about the same amount of LUTs for their FFT units.

Our throughput per DSP is then $31250/1920 = 16.28$ Bootstraps Per Second per DSP. In comparison, FPT reaches $28400/5494 = 5.16$ BPS/DSP, a third of our performance. Other designs perform much worse, with NFP reaching $3448/4508 = 0.77$ BPS/DSP and a design like YKP [YKP22] reaching only $531/6910 = 0.077$ BPS/DSP. Our throughput per DSP is thus more than 21 times better than the CHES 2024 NFP design [LLW+24].

5.3 Demo

To showcase our design and to prove its functionality, we implemented a demo which takes the open source FINAL code from the original FINAL paper [BIP+22] and accelerates the blind rotation step. The blind rotation is thus accelerated from around 2 seconds to 1 millisecond for each batch of 32 ciphertexts.

5.4 Conclusion

Our design **Quatorze-bis** has a 10% higher throughput than FPT and is multiple factors faster than other designs (see Table 3). Our design's main contribution is its extremely small butterfly, which allows up to 2560 butterflies to be placed on an AMD Alveo U55c, and which, applied to FPT's TFHE algorithm, would have given a throughput twice that of FPT. Applied to FINAL, as it is in this design, it achieves slightly better throughput, and a factor three higher throughput per DSP.

6 Conclusion

This paper introduces a high-throughput FPGA hardware accelerator for TFHE-like bootstrapping. Its first contribution is a novel method for modular multiplication given a constant modulus. This method requires no DSPs and no additional logic compared to most state-of-the-art modular multiplication designs and requires less than 140 LUTs. This makes it significantly smaller than the modular multiplication units of other TFHE Hardware Accelerators, and thus makes it possible to fit many more butterfly processing units on an FPGA.

Its second contribution is a Number Theoretic Transform which maximizes the number of constant multiplier butterflies on the FPGA, by adapting the constant multiplier FFT design from the Signal Processing field [GM21] for the Number Theoretic Transform. This design considers the Number Theoretic Transform in two dimensions, with one dimension multiplexed in time and the other multiplexed in space.

Finally we combine our Number Theoretic Transform modules to form a state-of-the-art hardware accelerator **Quatorze-bis** for TFHE-like bootstrapping with a streaming width of $S = 64$ and a fully pipelined circuit matched to provide a constant stream of coefficients to all NTTs. Hereby, our design **Quatorze-bis** improves on the current state-of-the-art throughput by 10-25% with 31250 Programmable Bootstraps Per Second, and improves on the state-of-the-art throughput per DSP by a factor 3 by reaching a figure of merit of 16.28 Bootstraps Per Second Per DSP.

7 Acknowledgements

The authors would like to thank Wouter Legiest and Pcy Sluys for their valuable advice on all aspects of C++. Without them, the software-hardware interface and the testing of the hardware accelerator using the FINAL source code would have taken significantly longer. Moreover, the authors would like to thank Furkan Turan, whose HBM interface formed the basis on which the interface of this paper was built.

A Appendix

Algorithm 2: TFHE algorithm

Data: $\mathbf{BSK} \in \mathcal{R}_q^{630 \times 2 \times 3 \times 2}$, $\mathbf{ACC} \in \mathcal{R}^2$, $\mathbf{a} = \{a_0, a_1, \dots, a_{n-1}\}$
Result: \mathbf{ACC} , updated with $a_i * s_i$ for $i = 0 \dots n - 1$

```

1 begin
2   for  $i = 0, 1, \dots, 630 - 1$  do
3     if  $a_i > 0$  then
4       CoefACC = 0 ;
5       for  $k = 0, 2 - 1$  do
6         CMUXACC[k] = CMUX $_{a_i}$ (ACC[k]);
7         dcmp[k][2:0] = SignedDigitDecompose(CMUXACC[k]);
8         for  $l = 0, 1, \dots, 3 - 1$  do
9           evalACC[k][l] = NTT(dcmp[k][l]);
10          // 630*2*3=3780 NTTs
11          for  $m = 0, 2 - 1$  do
12            CoefACC[k][m] += evalACC[m][l] * BSK $_i$ [k][l][m];
13            // 630*2*3*2=7560 pointwise mult.
14          end
15        end
16      end
17    end
18  end
19  Return ACC;
20 end
21 end

```

References

- [ACD⁺18] Martin R. Albrecht, Benjamin R. Curtis, Amit Deo, Alex Davidson, Rachel Player, Eamonn W. Postlethwaite, Fernando Virdia, and Thomas Wunderer. Estimate all the LWE, NTRU schemes! In Dario Catalano and Roberto De Prisco, editors, *SCN 18*, volume 11035 of *LNCS*, pages 351–367. Springer, Cham, September 2018.
- [AKT⁺24] Aftab Akram, Fawad Khan, Shahzaib Tahir, Asif Iqbal, Syed Aziz Shah, and Abdullah Baz. Privacy preserving inference for deep neural networks:

Algorithm 3: FINAL algorithm

Data: $\text{BSK} \in \mathcal{R}_q^{610 \times 7}$, $\text{ACC} \in \mathcal{R}$, $\mathbf{a} = \{a_0, a_1, \dots, a_{n-1}\}$
Result: ACC , updated with $a_i * s_i$ for $i = 0 \dots n - 1$

```

1 begin
2   for  $i = 0, 1, \dots, 610 - 1$  do
3     if  $a_i > 0$  then
4       CoefACC = 0 ;
5       // No k loop
6       CMUXACC = CMUX $_{a_i}$ (ACC);
7       dcmp[6:0] = SignedDigitDecompose(CMUXACC);
8       for  $l = 0, 1, \dots, 7 - 1$  do
9         evalACC[l] = NTT(dcmp[l]);
10        // 610*7=4270 NTTs
11        // No m loop
12        CoefACC+ = evalACC[l] * BSK $_i$ [l];
13        // 610*7=4270 pointwise mult.
14      end
15      // No m loop
16      ACC[k] += INTT(CoefACC);
17      // 610=610 INTTs
18    end
19  end
20  Return ACC;
21 end

```

Optimizing homomorphic encryption for efficient and secure classification. *IEEE Access*, 12:15684–15695, 2024.

- [AMK⁺23] Aikata Aikata, Ahmet Can Mert, Sunmin Kwon, Maxim Deryabin, and Sujoy Sinha Roy. REED: Chiplet-based accelerator for fully homomorphic encryption. *Cryptology ePrint Archive*, Report 2023/1190, 2023.
- [APS15] Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptology*, 9(3):169–203, 2015.
- [BAE⁺24] Olivier Bronchain, Melissa Azouaoui, Mohamed ElGhamrawy, Joost Renes, and Tobias Schneider. Exploiting small-norm polynomial multiplication with physical attacks application to CRYSTALS-Dilithium. *IACR TCHES*, 2024(2):359–383, 2024.
- [BGV12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In Shafi Goldwasser, editor, *ITCS 2012*, pages 309–325. ACM, January 2012.
- [BIP⁺22] Charlotte Bonte, Ilia Iliashenko, Jeongeun Park, Hilder V. L. Pereira, and Nigel P. Smart. FINAL: Faster FHE instantiated with NTRU and LWE. In Shweta Agrawal and Dongdai Lin, editors, *ASIACRYPT 2022, Part II*, volume 13792 of *LNCS*, pages 188–215. Springer, Cham, December 2022.
- [BM67] E. O. Brigham and R. E. Morrow. The fast fourier transform. *IEEE Spectrum*, 4(12):63–70, 1967.

- [BNV24] Jonas Bertels, Quinten Norga, and Ingrid Verbauwhede. A better kyber butterfly for FPGAs. In *34th International Conference on Field-Programmable Logic and Applications, FPL 2024, Torino, Italy, September 2-6, 2024*, pages 171–177. IEEE, 2024.
- [CG19] Eleanor Chu and Alan George. Inside the FFT black box: Serial and parallel fast fourier transform algorithms. In *Inside the FFT Black Box: Serial and Parallel Fast Fourier Transform Algorithms*, 2019.
- [CGGI16] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *ASIACRYPT 2016, Part I*, volume 10031 of *LNCS*, pages 3–33. Springer, Berlin, Heidelberg, December 2016.
- [CKKS17] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. Homomorphic encryption for arithmetic of approximate numbers. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part I*, volume 10624 of *LNCS*, pages 409–437. Springer, Cham, December 2017.
- [dDK24] Florent de Dinechin and Martin Kumm. *Application-Specific Arithmetic*. Springer, 2024.
- [DM15] Léo Ducas and Daniele Micciancio. FHEW: Bootstrapping homomorphic encryption in less than a second. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 617–640. Springer, Berlin, Heidelberg, April 2015.
- [Dv21] Léo Ducas and Wessel P. J. van Woerden. NTRU fatigue: How stretched is overstretched? In Mehdi Tibouchi and Huaxiong Wang, editors, *ASIACRYPT 2021, Part IV*, volume 13093 of *LNCS*, pages 3–32. Springer, Cham, December 2021.
- [GBP+23] Robin Geelen, Michiel Van Beirendonck, Hilder V. L. Pereira, Brian Huffman, Tynan McAuley, Ben Selfridge, Daniel Wagner, Georgios D. Dimou, Ingrid Verbauwhede, Frederik Vercauteren, and David W. Archer. BASALISC: Programmable hardware accelerator for BGV fully homomorphic encryption. *IACR TCHES*, 2023(4):32–57, 2023.
- [Gen09] Craig Gentry. Computing on encrypted data (invited talk). In Juan A. Garay, Atsuko Miyaji, and Akira Otsuka, editors, *CANS 09*, volume 5888 of *LNCS*, page 477. Springer, Berlin, Heidelberg, December 2009.
- [GM21] Mario Garrido and Pedro Malagón. The constant multiplier FFT. *IEEE Trans. Circuits Syst. I Regul. Pap.*, 68(1):322–335, 2021.
- [GMT24] Charles Gouert, Dimitris Mouris, and Nektarios Georgios Tsoutsos. Juliet: A configurable processor for computing on encrypted data. *IEEE Transactions on Computers*, 73(9):2335–2349, 2024.
- [GPV23] Antonio Guimarães, Hilder V. L. Pereira, and Barry Van Leeuwen. Amortized bootstrapping revisited: Simpler, asymptotically-faster, implemented. In Jian Guo and Ron Steinfeld, editors, *ASIACRYPT 2023, Part VI*, volume 14443 of *LNCS*, pages 3–35. Springer, Singapore, December 2023.

- [JLJ22a] Lei Jiang, Qian Lou, and Nrushad Joshi. MATCHA: a fast and energy-efficient accelerator for fully homomorphic encryption over the torus. In Rob Oshana, editor, *DAC '22: 59th ACM/IEEE Design Automation Conference, San Francisco, California, USA, July 10 - 14, 2022*, pages 235–240. ACM, 2022.
- [JLJ22b] Lei Jiang, Qian Lou, and Nrushad Joshi. Matcha: a fast and energy-efficient accelerator for fully homomorphic encryption over the torus. In *Proceedings of the 59th ACM/IEEE Design Automation Conference, DAC '22*, page 235–240, New York, NY, USA, 2022. Association for Computing Machinery.
- [KL24] Tianqi Kong and Shuguo Li. Hardware acceleration and implementation of fully homomorphic encryption over the torus. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 71(3):1116–1129, 2024.
- [LKL⁺22] Joon-Woo Lee, HyungChul Kang, Yongwoo Lee, Woosuk Choi, Jieun Eom, Maxim Deryabin, Eunsang Lee, Junghyun Lee, Donghoon Yoo, Young-Sik Kim, et al. Privacy-preserving machine learning with fully homomorphic encryption for deep neural network. *IEEE Access*, 10:30039–30054, 2022.
- [LLW⁺24] Zhihao Li, Xianhui Lu, Zhiwei Wang, Ruida Wang, Ying Liu, Yinhang Zheng, Lutan Zhao, Kumpeng Wang, and Rui Hou. Faster ntru-based bootstrapping in less than 4 ms. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2024(3):418–451, 2024.
- [LMK⁺23] Yongwoo Lee, Daniele Micciancio, Andrey Kim, Rakyong Choi, Maxim Deryabin, Jieun Eom, and Donghoon Yoo. Efficient FHEW bootstrapping with small evaluation keys, and applications to threshold homomorphic encryption. In Carmit Hazay and Martijn Stam, editors, *EUROCRYPT 2023, Part III*, volume 14006 of *LNCS*, pages 227–256. Springer, Cham, April 2023.
- [LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 1–23. Springer, Berlin, Heidelberg, May / June 2010.
- [MKO⁺20] A. C. Mert, E. Karabulut, E. Ozturk, E. Savas, and A. Aysu. An extensive study of flexible design methods for the number theoretic transform. *IEEE Transactions on Computers*, pages 1–1, 2020.
- [POG15] Thomas Pöppelmann, Tobias Oder, and Tim Güneysu. High-performance ideal lattice-based cryptography on 8-bit atxmega microcontrollers. In Kristin E. Lauter and Francisco Rodríguez-Henríquez, editors, *Progress in Cryptology - LATINCRYPT 2015 - 4th International Conference on Cryptology and Information Security in Latin America, Guadalajara, Mexico, August 23-26, 2015, Proceedings*, volume 9230 of *Lecture Notes in Computer Science*, pages 346–365. Springer, 2015.
- [PPC⁺23] Adiwena Putra, Prasetyo, Yi Chen, John Kim, and Joo-Young Kim. Strix: An end-to-end streaming architecture with two-level ciphertext batching for fully homomorphic encryption with programmable bootstrapping. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2023, Toronto, ON, Canada, 28 October 2023 - 1 November 2023*, pages 1319–1331. ACM, 2023.

- [Reg05] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In Harold N. Gabow and Ronald Fagin, editors, *37th ACM STOC*, pages 84–93. ACM Press, May 2005.
- [RV08] Dionysios Reisis and Nikolaos Vlassopoulos. Conflict-free parallel memory accessing techniques for fft architectures. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 55(11):3438–3447, 2008.
- [SFK⁺22] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Nathan Manohar, Nicholas Genise, Srinivas Devadas, Karim Eldefrawy, Chris Peikert, and Daniel Sanchez. Craterlake: a hardware accelerator for efficient unbounded computation on encrypted data. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 173–187, New York, NY, USA, 2022. Association for Computing Machinery.
- [VDTV23] Michiel Van Beirendonck, Jan-Pieter D’Anvers, Furkan Turan, and Ingrid Verbauwhede. FPT: A fixed-point accelerator for torus fully homomorphic encryption. In Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda, editors, *ACM CCS 2023*, pages 741–755. ACM Press, November 2023.
- [WLH⁺23] Zhiwei Wang, Peinan Li, Rui Hou, Zhihao Li, Jiangfeng Cao, XiaoFeng Wang, and Dan Meng. He-booster: An efficient polynomial arithmetic acceleration on GPUs for fully homomorphic encryption. *IEEE Transactions on Parallel and Distributed Systems*, 34(4):1067–1081, 2023.
- [WWL⁺24] Ruida Wang, Yundi Wen, Zhihao Li, Xianhui Lu, Benqiang Wei, Kun Liu, and Kunpeng Wang. Circuit bootstrapping: Faster and smaller. In Marc Joye and Gregor Leander, editors, *Advances in Cryptology – EUROCRYPT 2024*, pages 342–372, Cham, 2024. Springer Nature Switzerland.
- [XL21] Yufei Xing and Shuguo Li. A compact hardware implementation of CCA-secure key exchange mechanism CRYSTALS-KYBER on FPGA. *IACR TCHES*, 2021(2):328–356, 2021.
- [XLK⁺24] Yu Xiao, Feng-Hao Liu, Yu-Te Ku, Ming-Chien Ho, Chih-Fan Hsu, Ming-Ching Chang, Shih-Hao Hung, and Wei-Chao Chen. Gpu acceleration for fhew/TFHE bootstrapping. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2025(1):314–339, Dec. 2024.
- [YKP22] Tian Ye, Rajgopal Kannan, and Viktor K. Prasanna. FPGA acceleration of fully homomorphic encryption over the torus. In *IEEE High Performance Extreme Computing Conference, HPEC 2022, Waltham, MA, USA, September 19-23, 2022*, pages 1–7. IEEE, 2022.
- [YSD⁺24] Hao Yang, Shiyu Shen, Wangchen Dai, Lu Zhou, Zhe Liu, and Yunlei Zhao. Phantom: A CUDA-accelerated word-wise homomorphic encryption library. *IEEE Transactions on Dependable and Secure Computing*, pages 1–12, 2024.
- [ZCJ24] Jian Zhang, Aijiao Cui, and Yier Jin. Acceleration of the bootstrapping in TFHE by FPGA. *IEEE Transactions on Emerging Topics in Computing*, pages 1–16, 2024.