A Flexible and Scalable NTT Hardware:

Applications from Homomorphically Encrypted Deep Learning to Post-Quantum Cryptography

Ahmet Can Mert[†‡], Emre Karabulut[‡], Erdinç Öztürk[†*], Erkay Savaş[†*], Michela Becchi[‡], Aydin Aysu[‡*]
[†]Faculty of Engineering and Natural Sciences, Sabanci University, Istanbul, Turkey
[‡]Department of Electrical and Computer Engineering, North Carolina State University, NC, USA

***Abstract*—The Number Theoretic Transform (NTT) enables faster polynomial multiplication and is becoming a fundamental component of next-generation cryptographic systems. NTT hardware designs have two prevalent problems related to *design-time* flexibility. First, algorithms have different arithmetic structures causing the hardware designs to be manually tuned for each setting. Second, applications have diverse throughput/area needs but the hardware have been designed for a fixed, pre-defined number of processing elements.**

This paper proposes a parametric NTT hardware generator that takes arithmetic configurations and the number of processing elements as inputs to produce an efficient hardware with the desired parameters and throughput. We illustrate the employment of the proposed design in two applications with different needs: A homomorphically encrypted deep neural network inference (`CryptoNets`) and a post-quantum digital signature scheme (`qTESLA`). We propose the *first NTT hardware acceleration* for both applications on FPGAs. Compared to prior software and high-level synthesis solutions, the results show that our hardware can accelerate NTT up to 28× and 48×, respectively. Therefore, our work paves the way for high-level, automated, and modular design of next-generation cryptographic hardware solutions.

***Index Terms*—NTT, Flexible, Hardware, CryptoNets, qTESLA**

I. Introduction

Lattice-based cryptography offers alternative schemes enabling interesting applications such as quantum-resistant key-exchange and digital signature protocols [1], provably-secure hash functions [2], and homomorphic encryption [3]. Polynomial multiplication is a computational bottleneck of lattice-based cryptosystems. The Number Theoretic Transform (NTT) reduces the $\mathcal{O}(n^2)$ complexity of the schoolbook polynomial multiplication to $\mathcal{O}(n \cdot \log n)$. NTT is therefore a major building block of lattice cryptography implementations.

There are two design-time flexibility requirements for specialized NTT hardware accelerators. The first one is due to varying algorithm parameters. For example, while `NewHope` algorithm [1] uses polynomials of degree 1024 with 14-bit coefficients, `CryptoNets` [4] operates with polynomials of degree 4096 with up to 60-bit coefficients. The second flexibility need is a result of application requirements. Even for a fixed algorithm, while a cloud computing infrastructure demands a high-throughput hardware, an IoT/embedded device would favor a low area/energy design.

The prior NTT hardware designs have so far been fixed in both aspects. Indeed, proposing an efficient multiplier architecture for fixed polynomial sizes with a fixed number of processing elements (PEs) still merits a publication [5]. Extending the hardware from a specific setting is non-trivial due to memory access and control flow challenges. A recent work offers run-time reconfigurability with a flexible NTT hardware, but only supports a few algorithm parameters and is fixed for low area implementations [6].

In this paper, we propose the *first design-time* configurable NTT—a parametric NTT hardware generator. Our solution has both dimensions of flexibility and supports a wide range of algorithms. First, it can cater different arithmetic structures of polynomial size and modular width. Second, it can provide a trade-off in area vs. performance by incorporating a different number of PEs. The user of our generator simply enters the arithmetic needs of an algorithm and a desired number of PEs, and our tool automatically produces a corresponding efficient hardware. Prior works, by contrast, are either ad-hoc efforts fixed for a specific setting [7], [8], [9], [10], [6], [11], [12], [13] or uses a fixed number of PEs [6].

The most challenging parts of the flexible NTT hardware is the memory management and the modular multiplier design. Memory management is difficult because the access pattern, size, and block RAM (BRAM) count change with the polynomial size and the number of PEs. To achieve a flexible yet efficient memory organization, we adapted the decimation-in-frequency Iterative NTT [14]. Modular multiplier design is difficult because it has to support different modulus values and still has to be efficient. To achieve this modular multiplier hardware, we propose the generalized version of the word-level Montgomery modular reduction algorithm for NTT-friendly primes [13] and implement its fully-pipelined version.

We demonstrate the genericness of our work by tuning it to the NTT of two disparate applications: A homomorphically encrypted deep neural network inference (`CryptoNets`) [4] and a post-quantum digital signature scheme (`qTESLA`) [15]. While the former application enables privacy-friendly neural network classification on encrypted data, the latter secures critical cyberinfrastructure against quantum computer attacks. Our generator allowed proposing the *first NTT hardware acceleration* for both applications on FPGAs. The results quantify that our design can accelerate these applications up to 48×. Furthermore, our tool automates the design space exploration of hardware—the results show a coverage of 95× in area and 21× in latency. Therefore, this work takes a critical step towards high-level, automated, and modular design of next-generation cryptographic hardware solutions.

*Dr. Öztürk and Dr. Savaş were supported by TUBITAK under Grant Number 118E725. The project is supported by NSF under Grant #1850373.

Algorithm 1 NTT-based Polynomial Multiplication [16]

```
Input: $A(x), B(x) \in \mathbb{Z}_q[x]/(x^n+1)$
Input: primitive $2n$-th root of unity $\psi \in \mathbb{Z}_q$
Output: $C(x) = A(x) \times B(x)$, $C(x) \in \mathbb{Z}_q[x]/(x^n+1)$
1: $\overline{A}(x) \leftarrow \mathbf{NTT}(A(x) \odot (\psi^0, \psi^1, \ \dots \ , \psi^{n-1}))$
2: $\overline{B}(x) \leftarrow \mathbf{NTT}(B(x) \odot (\psi^0, \psi^1, \ \dots \ , \psi^{n-1}))$
3: $\overline{C}(x) \leftarrow \overline{A}(x) \odot \overline{B}(x)$
4: $C(x) \leftarrow \mathbf{INTT}(\overline{C}(x)) \odot (\psi^0, \psi^{-1}, \ \dots \ , \psi^{-(n-1)}))$
5: return $C(x)$
```

Algorithm 2 Iterative NTT Algorithm [14]

```
Input: $A(x) \in \mathbb{Z}_q[x]/(x^n+1)$
Input: primitive $n$-th root of unity $\omega \in \mathbb{Z}_q$, $n = 2^l$
Output: $\overline{A}(x) = \mathbf{NTT}(A) \in \mathbb{Z}_q[x]/(x^n+1)$
1: for $i$ from 1 by 1 to $l$ do
2:     $m = 2^{l-i}$
3:     for $j$ from 0 by 1 to $2^{i-1}-1$ do
4:         for $k$ from 0 by 1 to $m-1$ do
5:             $U \leftarrow A[2 \cdot j \cdot m + k]$
6:             $V \leftarrow A[2 \cdot j \cdot m + k + m]$
7:             $A[2 \cdot j \cdot m + k] \leftarrow U + V$
8:             $A[2 \cdot j \cdot m + k + m] \leftarrow \omega^{(2^{i-1}.k)} \cdot (U - V)$
9:         end for
10:    end for
11: end for
12: return $A$
```

II. Polynomial Multiplication and the NTT

A fundamental arithmetic and time-consuming operation in lattice-based cryptography is the multiplication of polynomials. This operation is defined over the ring of polynomials $\mathbb{Z}_q[x]/\phi(x)$, i.e., polynomials of degree $n-1$ with coefficients modulo q. Therefore, the multiplication takes two polynomials $A(x)=\sum_{i=0}^{n-1} a_i x^i$ and $B(x)=\sum_{i=0}^{n-1} b_i x^i$ and returns the output $C(x)=\sum_{i=0}^{n-1} c_i x^i$. Choosing the reduction polynomial $\phi(x)$ as (x^n+1) enables an efficient reduction of multiplication output $C(x)$ to a degree $n-1$ polynomial.

Algorithm 1 presents the polynomial multiplication with NTT and Algorithm 2 describes the Iterative NTT algorithm [14]. NTT is a Discrete Fourier Transform defined over the ring $\mathbb{Z}_q/\phi(x)$. NTT converts a polynomial into the NTT domain where polynomial multiplication becomes a coefficient-wise multiplication, which is denoted by $\odot$. NTT operation takes $A(x) = \sum_{i=0}^{n-1} a_i x^i$ as input and generates $\mathbf{A}(x) = \sum_{i=0}^{n-1} \mathcal{A}_i x^i$ as output, where each NTT coefficient $\mathcal{A}_i$ is defined as $\mathcal{A}_i = \sum_{j=0}^{n-1} a_j \omega^{ij}$ over $\mathbb{Z}_q$.

The constant used in the NTT operation, $\omega \in \mathbb{Z}_q$, is called the twiddle factor which is a primitive n-th root of unity in $\mathbb{Z}_q$ satisfying $\omega^n \equiv 1 \pmod q$ and $\forall i < n$, $\omega^i \neq 1 \pmod q$, where $q \equiv 1 \pmod n$. Similarly, the inverse NTT (INTT) can be performed as $a_i = n^{-1} \sum_{j=0}^{n-1} \mathcal{A}_j \omega^{-ij}$ in $\mathbb{Z}_q$. The reduction is computed by multiplying the coefficients before and after the NTT/INTT with powers of ψ/ψ^{-1} where $\psi = \sqrt{\omega} \pmod q$. The parameters we consider for flexibility are n and q, which respectively define the ring size and bit-width (or coefficient size) of the modulus.

NTT computations have three parts: loading related data (Steps 5 and 6 in Algorithm 2), performing arithmetic computations (right hand side of Steps 7 and 8 in Algorithm 2) and storing the result (left hand side assignment of Steps 7 and 8 in Algorithm 2). PEs perform these steps. Within each PE, the so-called butterfly units execute arithmetic computations, which are composed of modular addition, subtraction, and multiplication. The algorithm loops over $log_2(n)$ stages and performs $n/2$ butterfly at each stage. To achieve a higher throughput, it is possible to unroll and parallelize the NTT loops by using multiple PEs.

We will discuss the design of the flexible hardware in a bottom-up fashion, starting from the design of efficient modular multiplication. We will then describe the construction of the PE and finally explain its parallelization along with an optimized memory access and organization.

III. Hardware Design

The proposed parametric NTT hardware generator takes the ring size, bit-width of the modulus and number of PEs as input parameters, and it generates an optimized NTT hardware that performs NTT operation for the given parameters. The proposed work provides flexibility not only for ring size and bit size of modulus but also the number of PEs which determines the throughput of the hardware. The same design can also execute the INTT.

A. A Novel Word-Level Montgomery Modular Multiplier

The modular multiplier is the key component of the NTT arithmetic. This unit consists of two parts: an integer multiplier followed with a modular reduction. We developed the parametrized version of both parts.

There are two options to design an efficient modular reduction: Montgomery [13] and Barrett [6]. Choosing the correct one for a parametric hardware is a non-trivial design decision. Banerjee *et. al.*, for instance, uses a Barrett modular reduction hardware [6], which includes two different reduction units. The first one is a configurable Barrett reduction hardware which enables run-time flexibility. The second one employs a separate, specialized reduction hardware which is only compatible with a small set of pre-determined special moduli.

We argue that for the design-time flexibility, Montgomery can offer a more efficient solution than Barrett. But the baseline Montgomery has to be optimized for NTT primes and extended to support the parametric ring sizes and bit widths. To that end, we propose a novel Montgomery based reduction unit that generalizes the word-level Montgomery modular reduction algorithm for NTT-friendly primes [13], with a word-size derived from the parameters. Compared to the configurable Barrett reduction [6], our solution either saves the number of multipliers or results in smaller multiplier units. The results show our cycle count advantage when the hardware uses the same number of PEs.

Algorithm 3 provides the details of our generalized algorithm. The algorithm will require a different number and bit-width of multipliers for different parameters of q and n. The proposed algorithm utilizes the property of $q \equiv 1$

Algorithm 3 Word-Level Montgomery Reduction Algorithm for NTT-friendly primes

```
Input: $C = A \cdot B$ (a $2K$-bit positive integer)
Input: $q$ (a $K$-bit modulus), $q = q_H \cdot 2^w + 1$
Input: $w = log_2(2n)$ (word size)
Output: $Res = C \cdot R^{-1} \pmod{q}$ where $R = 2^{w \times L} \pmod{q}$
1: $L = \lceil \frac{K}{w} \rceil$
2: $T1 = C$
3: for $i$ from 0 to $L$ do
4:     $T1_H = T1 >> w$
5:     $T1_L = T1 \pmod{2^w}$
6:     $T2 = two's\ complement\ of\ T1_L$
7:     $cin = T2[w-1] \vee T1_L[w-1]$
8:     $T1 = T1_H + (q_H \cdot T2[w-1:0]) + cin$
9: end for
10: $T4 = T1 - q$
11: if ($T4 < 0$) then $Res = T1$ else $Res = T4$
```

$\pmod{2n}$, which should be satisfied by any NTT-friendly prime using $negative\ wrapped\ convoluti$ technique. Thus, an NTT prime q can be written as $q = q_H \cdot 2^{log_2(2n)} + 1$. In order to exploit that property, we can perform a word-level Montgomery reduction with the word size $w = log_2(2n)$ and divide the reduction operation into smaller parts instead of performing it all at once. This will result in a flexible modular reduction structure. The modular reduction operation executes $L = \lceil \frac{K}{w} \rceil$ times for a K-bit modulus. The Montgomery modular reduction variable $\mu = -q^{-1} \pmod{2^w}$ becomes -1, simplifying the multiplication of $(AB \pmod{2^w}) \cdot \mu$ in the Montgomery modular reduction to the two's complement.

Montgomery reduction algorithm takes $A \cdot B$ as input and calculates the output $A \cdot B \cdot R^{-1} \pmod{q}$, where $R = 2^{wL}$ is defined as Montgomery reduction residual. The residual has to be corrected using an extra multiplication with R to obtain $A \cdot B \pmod{q}$. This extra multiplication can be moved to the input by multiplying one of the inputs by R. Since one of the inputs in NTT is the constant twiddle factor, ω, we can fuse the multiplication by pre-computing it ($\omega \cdot R \pmod{q}$) and loading it into the related memory at design-time to save one multiplication at run-time.

The proposed word-level Montgomery modular reduction algorithm divides reduction operation into a set of multiply and accumulate (MAC) operations. Namely, it performs $X \cdot Y + Z + cin$ operation, which can be implemented using DSP blocks in FPGAs, for different number of times for different arithmetic configurations. Therefore, the algorithm itself provides easiness for generating flexible designs. Our tool makes use of this to automatically generate the reduction hardware for the given ring size and bit-size of the modulus.

Fig. 1 illustrates two example modular reduction hardware generated by the proposed work for (a) ring size of 1024 with 32-bit modulus and (b) ring size of 256 with 16-bit modulus. Both designs offer advantage over the Barret method [6]—while the first design uses smaller multiplier units, the second one saves one multiplication. The first implementation requires 3 MAC operations while the second implementation uses only

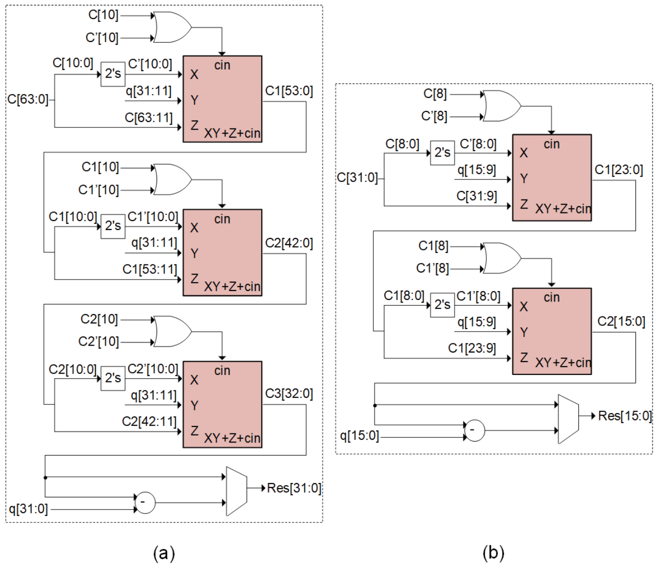

Fig. 1. Reduction Hardware for (a) n=1024, q=32-bit; (b) n=256, q=16-bit

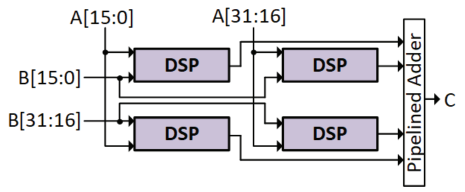

Fig. 2. Integer Multiplier for 32-bit Inputs

2 MAC operations. The proposed reduction hardware is fully pipelined, and it produces one output each clock cycle after filling the pipeline. The generated modular reduction hardware runs in constant time for a given arithmetic configuration.

The other part of the modular multiplication, integer multiplier hardware, uses the bit size of modulus as a parameter. Each input of the multiplier is divided into 16-bit pieces and one DSP block is used for each 16-bit×16-bit multiplication operation. The resulting intermediate values are then added up to calculate multiplication result using a pipelined adder tree. Therefore, the proposed integer multiplier is fully pipelined and it can produce one multiplication result per clock cycle. Fig. 2 shows the hardware of 32-bit multiplication, which requires 4 DSP blocks. The number and configuration of the DSP blocks along with the adder tree are automatically synthesized based on the input parameters. Although it may be possible to partition input integers more efficiently, we divide all inputs into 16-bit (power-of-2) pieces to preserve regularity and reduce the complexity of the control unit.

B. PEs and Butterfly Units

Since we established the efficient hardware for the core modular arithmetic, next we discuss the design of the butterfly units that use modular operations and the PEs that contain butterfly units.

Each PE implements the Gentleman-Sande butterfly configuration [14] corresponding to the Steps 5–8 of the Algorithm 2. Fig. 3 illustrates one PE. PEs take two coefficients and one twiddle factor as inputs, perform the butterfly operation, and

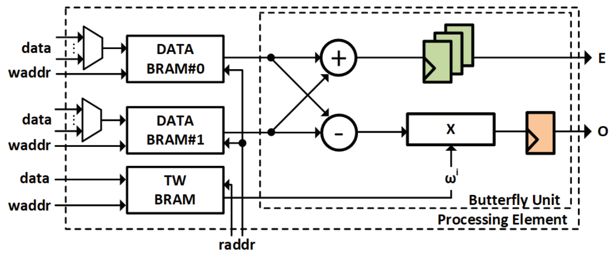

Fig. 3. PE and the Butterfly Unit

output two resulting coefficients, namely even (E) and odd (O) coefficients. A PE consists of one modular adder, one modular subtractor, and one modular multiplier for implementing the butterfly operation. Each PE also uses three dual-port BRAMs, where two data BRAMs store input and intermediate coefficients while the other, twiddle factor BRAM, stores the twiddle factors (with Montgomery correction), which are design-time constants pre-computed based on input parameters.

Fig. 3 depicts that the even coefficient is the output of the modular addition operation (Step 7 in Algorithm 2) and odd output coefficient is the output of the modular subtraction and multiplication (Step 8 in Algorithm 2). To synchronize the output generation of even and odd coefficients, the proposed design inserts a parametric number of registers, shown in green, on the even path based on the input parameters (ring size and modulus bit-width). The input parameter of the number of PE can only be a power-of-2, and it could be a maximum of $n/2$ for an n-pt NTT.

C. Flexible Memory Access and Overall Design

A significant challenge for the NTT hardware design is managing the complex memory access schedule. This problem becomes more challenging for us because our hardware aims to provide flexibility in the number of core PEs. We need our parametric hardware generator to synthesize the address generation logic that will control the 2 BRAMs in each PE without adding any stalls to the NTT pipeline.

The proposed design uses the Iterative NTT scheme of Algorithm 2, which consists of $log_2(n)$ stages and performs $n/2$ butterfly operations at each stage. Fig. 4 (a) demonstrates an example of the memory read access pattern of coefficients for n=8. Each yellow dot represents a butterfly operation, which consumes and produces two coefficients mapping to the same degree. For example, 0^{th} and 4^{th} coefficients in the first stage will correspond to the 0^{th} and 4^{th} coefficients of the second stage.

The irregular access pattern of the NTT enforces storing each coefficient to a unique address. Fig. 4 (b) demonstrates the one PE case where coefficients going to the same butterfly operation are stored in two distinct memory blocks. For example, at the first stage of the 8-pt NTT, four coefficient pairs (0, 4), (1, 5), (2, 6), (3, 7) go into butterfly operation. These pairs need to be read at the same clock cycle. Therefore, coefficients 0, 1, 2, 3 and 4, 5, 6, 7 should be stored in separate memory blocks and accessed in parallel.

Unfortunately, the pairings of the coefficients change at each stage. The output of the stage, therefore, has to be stored back

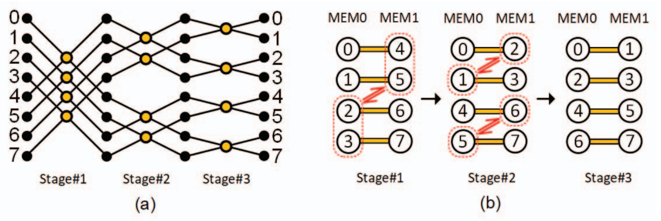

Fig. 4. (a) Coefficient Access Pattern; (b) Memory Access Pattern for n=8

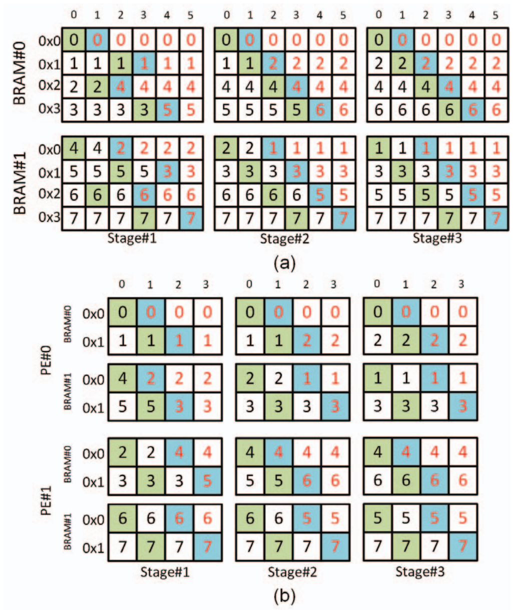

Fig. 5. Memory Access for the 8-pt NTT with (a) One PE, (b) Two PEs

into the memory blocks based on the pairing of the next stage. Fig. 4 (b) shows the example for n=8 where the coefficient pairings for the first and second stages are respectively (0, 4), (1, 5), (2, 6), (3, 7) and (0, 2), (1, 3), (4, 6), (5, 7). Hence, both outputs of the pairs (0, 4) and (1, 5) should be written into the first memory block. Likewise, (2, 6) and (3, 7) output will be placed in the second memory. This guarantees that all coefficient pairs at the second stage can be read in a cycle.

Our parametric hardware design automatically generates the required access pattern to handle memory access operations for different number of PEs. This pattern, however, requires coefficient pairs to be written into the same memory block. For example, for n=8, the coefficient pair (0, 4) should be written into the first memory block after the first stage to improve coalescing. This is enabled by adding one extra register to the output of the modular multiplier unit in the PE, shown as orange register in Fig. 3. This extra latency allows storing coefficients into the same memory in 2 cycles. Since the PEs are pipelined, this extra register does not affect the throughput.

The proposed design uses an alternating memory read pattern because the first and second half of coefficient pairs should be written into the first and second memories, respectively. For example, as shown in Fig. 4, only the coefficients (0, 1), (4, 5) are written into the first memory while (2, 3), (6, 7) are written into the second memory. Therefore, the memory read pattern for 8-pt NTT should be in the order (0, 4), (2, 6), (1, 5), (3, 7) instead of (0, 4), (1, 5), (2, 6), (3, 7).

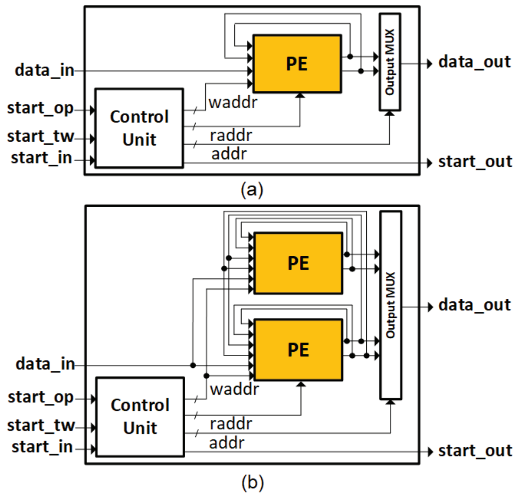

Fig. 6. NTT Hardware (a) with one PE; (b) with two PEs

Fig. 5 gives the memory access examples for 8-pt NTT with one and two PEs. Green and blue boxes respectively represent the read and write operations, and the letters in red represent coefficients written into the memory. For the 8-pt NTT with one PE, coefficients (0, 4) and (1, 5) need to be stored in the same memory block. The proposed addressing scheme thus first reads coefficients (0, 4) and (2, 6), which should be written into the first and second memories, respectively. Therefore, operation can continue without any stall. For the 8-pt NTT with two PEs, the first PE performs the butterfly operation for the first half of the coefficient pairs while the second PE performs it for the other half. In this case, the first PE can read coefficients (0, 4) and (1, 5) in two consecutive cycles because coefficients 4 and 5 will be written into the memory of the second PE. Since we have two PEs instead of one, latency of each NTT stage is reduced.

Fig. 6 outlines the high-level block diagram of the generated NTT hardware with (a) one and (b) two PEs. The outputs of one PE are connected to all PEs in the design to broadcast the coefficients needed due to the memory dependency of NTT. Before the NTT starts, the hardware first takes twiddle factors followed with the input coefficient as inputs and writes them to their related BRAMs within each PE. The data BRAMs also keep the resulting output coefficients, which can be read through the output multiplexers.

IV. Results

The parametric hardware generator code is written in Verilog RTL and the generated NTT hardware are synthesized, placed, and routed using Xilinx Vivado 2018.1 tool on the Xilinx VIRTEX-7 FPGA xc7vx690tffg1761-2. The reference software is compiled with the GCC version `7.3.` in Ubuntu OS `18.04.3` LTS. Timing performance of the software implementations are obtained on a high-end Intel Xeon CE5-1650 @ 3.50 GHz $\times$ 12 CPU with 16GB RAM.

We first compare our results on a new implementation with the reference optimized software and high-level synthesis (HLS) generated code. These two approaches are the alternatives of our design methodology. To that end, we use

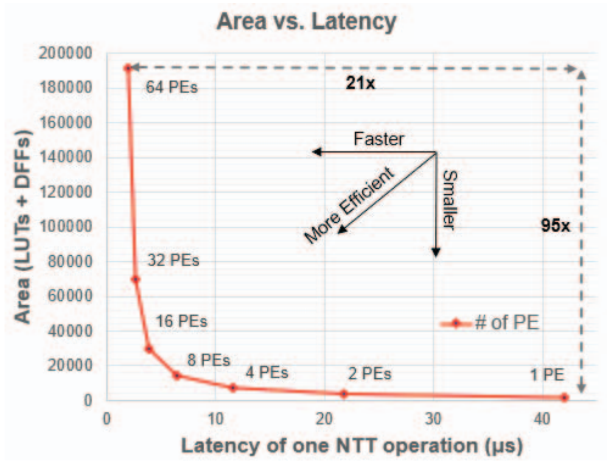

Fig. 7. Area vs. Latency for qTESLA with different PEs

`qTESLA` and `CryptoNets` as the driving applications. This work proposes the *first NTT hardware acceleration* for both applications on FPGAs. Both applications represent next generation cryptosystems—while `qTESLA` is a running candidate for the post-quantum standard, `CryptoNets` enables privacy-friendly neural network inference on encrypted data.

We use the reference software of `qTESLA-p-I` [15], which targets NISTs security category 1. `CryptoNets` software uses Microsoft SEAL library [19]. For HLS comparison, we use the result of a prior work [18]. We report our hardware results in three configurations: an area-optimizde design (1 PE), a throughput optimized design (64 PEs), and a balanced design (8 PEs). Note that only the NTT implementations of the same ring size (n) and coefficient size (q) make a meaningful comparison. The results in Table I shows the superiority of our approach, which outperforms existing software and high-level synthesis designs respectively by up to 28$\times$ and 48$\times$.

We next compare our work with optimized hardware designs. *Poly256* and *Poly512* hardware for n=256 and n=512 are also auto-generated for this experiment. We note that the target devices are implemented under different FPGA technology or even ASIC, hence, the comparison should serve as a first-order estimate rather than an idealized method. Table I provides these detailed results, which quantify that our parametric generator can produce a hardware that is comparable to fixed setting hardware units and can even be better in some cases. Note that the majority of prior hardware are fixed for a single arithmetic and/or performance setting. For instance, as we argue in Section III-A, our one PE design outperforms the work of Banarjee *et al.* [6] in latency (cycle count) due to our proposed word-level Montgomery reduction unit. Likewise, our designs can achieve either a lower area or a faster design (in cycle) compared to prior FPGA solutions.

We finally demonstrate the fast design-space exploration that can be achieved by our parametric hardware generator. To test this, we sweep the parameter that controls the number of PEs used in `qTESLA` hardware and report the implementation results in Fig. 7. Our generator is able to cover a space of 95$\times$ in area cost for a tradeoff of 21$\times$ in performance by simply tuning a parameter knob.

TABLE I
IMPLEMENTATION RESULTS AND COMPARISON TO PRIOR WORK

Work	**Platform**	n	q	**LUT / REG / DSP / BRAM**	**Clock (MHz)**	**Latency**	
						CC	μs
[7]^a	SPARTAN-6	256 512 1024	17-bit	250 / – / 3 / 2 240 / – / 3 / 2 250 / – / 3 / 2	–	– – –	25 50 100
[8]^{a,d}	VIRTEX-6	256	13-bit	4549 / 3624 / 1 / 12	262	–	8
[9]^d	Zynq UltraScale	4096	30-bit	64K / – / 200 / 400	225	–	73
[10]^d	VIRTEX-7	32768	32-bit	219K / – / 768 / 193	250	7709	51
[13]^d	SPARTAN-6 VIRTEX-7	1024	32-bit	1208 / – / 14 / 14 34K / 16K / 476 / 228	212 200	– 80	12 1.25
[17]^b	VIRTEX-6	256 512	13-bit 14-bit	1349 / 860 / 1 / 2 1536 / 953 / 1 / 3	313 278	1691 3443	5.4 12.3
[6]^b	40nm CMOS	256 512 1024	13-bit 14-bit 14-bit	106K / – / – / –	72	1289 2826 6155	17 32 81
[11]^b	40nm CMOS	256 512	13-bit 14-bit	– / – / – / – – / – / – / –	300	160 492	0.5 1.6
[12]^b	UMC 65nm	256 512 1024	13-bit 14-bit 14-bit	14K / – / – / –	25	2056 4616 10248	82 184 409
[18]^c	VIRTEX-7	1024 2048 4096	–	21K / 16K / 10 / 12 25K / 20K / 11 / 19 30K / 23K / 12 / 36	100	7597 15852 33337	76 159 333
qTESLA (Software) [15]^{a,d}	Intel Xeon CE5-1650	1024	28-bit	– / – / – / –	–	–	11
CryptoNets (Software) [4]^a	Intel Xeon CE5-1650	4096	60-bit	– / – / – / –	–	–	195
qTESLA (Hardware: Area-Opt.)	VIRTEX-7	1024	28-bit	1K / 1K / 7 / 2	125	5290	42
qTESLA (Hardware: Balanced)	VIRTEX-7	1024	28-bit	16K / 14K / 56 / 24	125	490	3.9
qTESLA (Hardware: Perf.-Opt.)	VIRTEX-7	1024	28-bit	132K / 59K / 448 / 96	125	250	2
CryptoNets (Hardware: Area-Opt.)	VIRTEX-7	4096	60-bit	2.7K / 2.6K / 31 / 12	142	24780	173
CryptoNets (Hardware: Balanced)	VIRTEX-7	4096	60-bit	22K / 17K / 248 / 96	125	3276	26
CryptoNets (Hardware: Perf.-Opt.)	VIRTEX-7	4096	60-bit	338K / 138K / 1984 / 768	125	972	7
Poly256 (Balanced)	VIRTEX-7	256	16-bit	7.4K / 5K / 24 / 24	147	160	1.1
Poly512 (Balanced)	VIRTEX-7	512	16-bit	8.1K / 5.2K / 24 / 24	141	345	2.4

^{a}:Uses fixed q. ^{b}:Can work with multiple n and q. ^{c}:Uses HLS. ^{d}:Uses fixed n.

V. CONCLUSION

Designing an efficient hardware accelerator is a delicate process. Indeed, proposing an optimized NTT hardware for one arithmetic setting and for one performance goal has been sufficient to publish in cryptography and hardware design conferences. But as the lattice cryptosystems mature and gear towards massive deployment, there will be a heavier emphasis on HLS for faster adoption and design space exploration, which enforce scalable and configurable hardware. This paper shows how to construct a flexible yet efficient hardware generator for a building block of lattice cryptosystems, pushing towards that end. Our tool can be integrated as a hard macro to the HLS flows and ease the process of hardware design for system developers. The generated hardware executes in constant-time, closing timing side-channels, but future extensions can consider mitigation against, e.g., power side-channels.

REFERENCES

[1] E. Alkim, L. Ducas, T. Pöppelmann, and P. Schwabe, “Post-quantum key exchangea new hope,” in *25th USENIX*, 2016, pp. 327–343.

[2] Y. Arbitman, G. Dogon, V. Lyubashevsky, D. Micciancio, C. Peikert, and A. Rosen, “Swifftx: A proposal for the sha-3 standard,” *Submission to NIST*, 2008. [Online]. Available: https://eprint.iacr.org/2012/343.pdf

[3] C. Gentry, “A fully homomorphic encryption scheme,” Ph.D. dissertation, Stanford, CA, USA, 2009.

[4] A. Brutzkus, O. Elisha, and R. Gilad-Bachrach, “Low latency privacy preserving inference,” *CoRR*, vol. abs/1812.10659, 2018.

[5] W. Liu, S. Fan, A. Khalid, C. Rafferty, and M. O’Neill, “Optimized schoolbook polynomial multiplication for compact lattice-based cryptography on fpga,” *IEEE Transactions on VLSI Systems*, pp. 1–5, 2019.

[6] U. Banerjee, T. S. Ukyab, and A. P. Chandrakasan, “Sapphire: A configurable crypto-processor for post-quantum lattice-based protocols,” *IACR Transactions on CHES*, pp. 17–61, 2019.

[7] A. Aysu, C. Patterson, and P. Schaumont, “Low-cost and area-efficient fpga implementations of lattice-based cryptography,” in *2013 IEEE International Symposium on HOST*. IEEE, 2013, pp. 81–86.

[8] T. Pöppelmann and T. Güneysu, “Towards practical lattice-based public-key encryption on reconfigurable hardware,” in *International Conference on Selected Areas in Cryptography*. Springer, 2013, pp. 68–85.

[9] S.S. Roy et al., “Fpga-based high-performance parallel architecture for homomorphic computing on encrypted data,” Cryptology ePrint Archive, Report 2019/160, 2019.

[10] E. Ozturk, Y. Doroz, E. Savas, and B. Sunar, “A custom accelerator for homomorphic encryption applications,” *IEEE Transactions on Computers*, vol. 66, no. 1, pp. 3–16, Jan 2017.

[11] S. Song, W. Tang, T. Chen, and Z. Zhang, “Leia: A 2.05mm2140mw lattice encryption instruction accelerator in 40nm cmos,” in *2018 IEEE Custom Integrated Circuits Conference (CICC)*, April 2018, pp. 1–4.

[12] T. Fritzmann and J. Seplveda, “Efficient and flexible low-power ntt for lattice-based cryptography,” in *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, May 2019, pp. 141–150.

[13] A. C. Mert, E. Ozturk, and E. Savas, “Design and implementation of a fast and scalable ntt-based polynomial multiplier architecture,” Cryptology ePrint Archive, Report 2019/109, 2019.

[14] P. Longa and M. Naehrig, “Speeding up the number theoretic transform for faster ideal lattice-based cryptography,” in *Cryptology and Network Security*, Milan, Italy, Nov. 2016, pp. 124–139.

[15] E. Alkim, P. S. Barreto, N. Bindel, P. Longa, and J. E. Ricardini, “The lattice-based digital signature scheme qtesla.” *IACR Cryptology ePrint Archive*, vol. 2019, p. 85, 2019.

[16] T. Pöppelmann and T. Güneysu, “Towards efficient arithmetic for lattice-based cryptography on reconfigurable hardware,” in *Progress in Cryptology – LATINCRYPT 2012*, 2012, pp. 139–158.

[17] S. S. Roy, F. Vercauteren, N. Mentens, D. D. Chen, and I. Verbauwhede, “Compact ring-lwe cryptoprocessor,” in *CHES*, 2014, pp. 371–391.

[18] K. Kawamura, M. Yanagisawa, and N. Togawa, “A loop structure optimization targeting high-level synthesis of fast number theoretic transform,” in *2018 19th ISQED*, March 2018, pp. 106–111.

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