An Extensive Study of Flexible Design Methods for the Number Theoretic Transform

Ahmet Can Mert, *Member, IEEE,* Emre Karabulut, Erdinç Öztürk, *Member, IEEE,* Erkay Savaş, *Member, IEEE,* and Aydin Aysu, *Senior Member, IEEE*

Abstract—Efficient lattice-based cryptosystems operate with polynomial rings with the Number Theoretic Transform (NTT) to reduce the computational complexity of polynomial multiplication. NTT has therefore become a major arithmetic component (thus computational bottleneck) in various cryptographic constructions like hash functions, key-encapsulation mechanisms, digital signatures, and homomorphic encryption. Although there exist several hardware designs in prior work for NTT, they all are isolated design instances fixed for specific NTT parameters or parallelization level.

This paper provides an extensive study of flexible design methods for NTT implementation. To that end, we evaluate three cases: (1) parametric hardware design, (2) high-level synthesis (HLS) design approach, (3) and design for software implementation compiled on soft-core processors, where all are targeted on reconfigurable hardware devices. We evaluate the designs that implement multiple NTT parameters and/or processing elements, demonstrate the design details for each case, and provide a fair comparison with each other and prior work. On a Xilinx Virtex-7 FPGA, compared to HLS and processor-based methods, the results show that the parametric hardware design is on average $4.4\times$ and $73.9\times$ smaller and $22.5\times$ and $19.3\times$ faster, respectively. Surprisingly, HLS tools can yield less efficient solutions than processor-based approaches in some cases.

Index Terms—NTT, Flexible, Hardware, HLS, RISC-V.

1 Introduction

Flexibility is a key requirement for digital systems to reduce design costs [1], [2] and consolidate changing standards, performance requirements, target platforms and algorithmic alternatives [3]. There are three primary design methods to build *design-time* flexibility: a parametric hardware generator with customized parameters, a software compiled on a processor, or a software directly transformed into hardware through a high-level synthesis (HLS) tool. Among these options, software tends to be the most flexible solution with the worst performance and the parametric hardware is often the most efficient one with the worst flexibility. HLS, by comparison, forms the middle ground, resulting in medium performance and flexible solutions.

Flexibility is especially important for next-generation cryptographic systems. The conventional cryptosystems such as the AES or RSA are established, widely-used solutions with many hardware/software IPs built specifically for their efficient implementations. Therefore, it is possible to find highly-optimized realizations of such standardized algorithms regardless of the target platform. But this is definitely not the case for developing cryptosystems, such as the lattice-based cryptography proposals.

Lattice-based cryptography offers interesting applications such as provably-secure hash functions [4], quantum-resistant digital signature protocols [5], key-encapsulation mechanisms [6] and homomorphic encryption [7]. Efficient lattice-based cryptography operates with polynomial rings and polynomial multiplication is a well-known computational bottleneck of lattice-based cryptosystems—indeed, schoolbook multiplication corresponds to 95.7% and 98.8% percent of the overall computation time for the encryption and decryption processes in a lattice-based public-key encryption and decryption scheme, respectively [8]. The Number Theoretic Transform (NTT) reduces the $\mathcal{O}(n^2)$ complexity of the schoolbook polynomial multiplication to the quasi-linear complexity of $\mathcal{O}(n \cdot \log n)$. NTT is thus a major building block of lattice cryptography implementations.

The impact of NTT for cryptography can be similar to that of the Fast Fourier Transform for signal processing. Lattice-based cryptography has already gained significant attention over a decade and its popularity will increase even further in the age of practical homomorphic encryption schemes and quantum supremacy [9]. Indeed, the problems in lattice-based cryptography are secure against quantum computing attacks and enable fully homomorphic encryption, which allows computation on encrypted data. NTT will therefore be a critical component since it accelerates the core arithmetic operation of lattice cryptosystems.

There are two design-time flexibility requirements for specialized NTT designs. The first one is due to varying algorithmic parameters which are the degree of the polynomial, n, and coefficient size, $K = \lceil \log_2 q \rceil$, where q is the coefficient modulus. For example, while `NewHope` algorithm [6] uses polynomials of degree 1023 with 14-bit coefficients, `CryptoNets` [7] operates with polynomials of degree 4095 with up to 60-bit coefficients. The second flexibility need is due to a consequence of performance requirements of the applications, even for a fixed algorithm. For instance, while a cloud computing infrastructure demands a high-

A. C. Mert and E. Öztürk are with the Electronics Engineering Program of Sabanci University, Istanbul, Turkey. E. Savaş is with the Computer Science and Engineering Program of Sabanci University, Istanbul, Turkey. E-mails: {ahmetcanmert, erdinco, erkays}@sabanciuniv.edu
E. Karabulut and A. Aysu are with the Department of Electrical and Computer Engineering, North Carolina State University, Raleigh, NC, 27606, USA. E-mails: ekarabu@ncsu.edu and aaysu@ncsu.edu

TABLE 1: Previous NTT Implementations

	Method	Work	n	K	PE	Target Platform
HLS		[12]^{c}	1024	14	10	Virtex-7
		[13]^{c}	1024	10	–	Virtex-7
		[14]^{c}	512	17	–	Zynq US+
Software		[15]^{a,b}	256	12	2	ARM Cortex-M4
		[16]	256 1024	13 14	4	Intel Core i7-4770K
		[17]^{a,b}	1024	14	1	ARM Cortex-M0 ARM Cortex-M4
		[18],[19]	1024	27	–	Intel Core i9-7900X
Hardware		[20]^{a}	256	17	1	Spartan-6
		[21]^{a,b}	256	13	1	Virtex-6
		[22]^{b}	4096	30	2	Zynq US
		[23]^{b}	32768	32	256	Virtex-7
		[24]^{b}	1024	32	1 64	Spartan-6 Virtex-7
		[18]^{b}	1024	32	32 64	Virtex7
		[25]^{c}	256	13	1	Virtex-6
		[11]^{c}	256 512 1024	13 14 14	1	40nm CMOS
		[26]^{c}	256	13	16	40nm CMOS
		[27]^{c}	256	13	1	UMC 65nm
		[28]^{a,b}	1024	14	4	Artix-7
		[29]^{b}	16384	32	1	Virtex-7
		[30]^{b}	65536	30	16	Virtex-6

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

throughput hardware, an IoT/embedded device would favor a low area/energy design. Therefore, throughput is the second flexibility parameter, mainly determined by the number of processing elements (PEs), which carry out the fundamental arithmetic operations in the design.

Table 1 reports a summary of the previous works in the literature, which lists the specific setting for parameters, the design method, and the target platform thereof. The prior NTT designs have so far been fixed in both aspects of algorithm parameters and throughput. Naturally, proposing an efficient multiplier architecture for a fixed polynomial degree and a coefficient size with a fixed number of PEs still merits a publication [10]. Notwithstanding, extending the hardware from a specific setting to a more generic and flexible design is non-trivial due to memory access and control flow challenges. A recent work offers run-time configurability with a flexible NTT hardware, but only supports a few algorithm parameters and is fixed for low area implementations [11]. Thus, as various settings in Table 1 indicates, the quest for a flexible and efficient design that supports a wide range of parameters is still outstanding.

This paper provides the first evaluation of *design-time* flexible NTT designs and uses the FPGA devices as the common demonstrator platform. To that end, we investigate three different design approaches. In our first approach, we propose an optimized, parametric hardware generator for the NTT operation. This design offers flexibility for both algorithm parameters and throughput, and supports a wide range of cryptographic algorithms. First, it can cater different arithmetic structures for varying polynomial degrees and coefficient sizes. 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 polynomial degree and coefficient size and a desired number of PEs, and our tool automatically produces a corresponding efficient hardware [1]. Prior works, by contrast, are either ad-hoc efforts fixed for a specific setting [20], [21], [22], [23], [24], [18], [28], [29], [30] or employ a fixed number of PEs [11], [25], [26], [27]. We furthermore investigate two other flexible design approaches and provide implementation results: software implementation compiled on a soft-core processors and HLS. We finally analyze the resulting implementations of all three design approaches and compare them with each other and to prior fixed NTT designs in the literature.

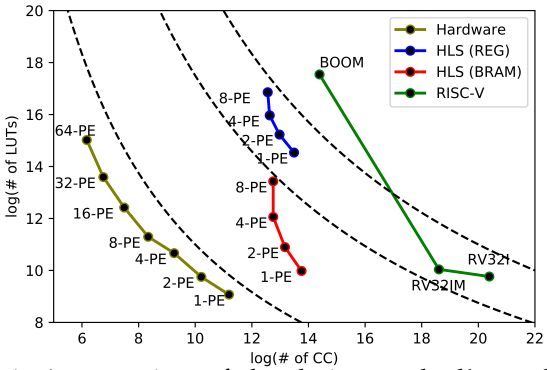


Fig. 1: An overview of the design method's results and comparison for the NTT of `NewHope-512`. The hand-tuned hardware designs lead to most efficient results while HLS and processor-based methods yield similar solutions.

This paper builds on our prior work, which is accepted to DATE 2020 (see https://www.date-conference.com/). While our prior work reports on a preliminary design of the parametric hardware generator (Section 3), the contributions of this paper are as follows:

- We implement the NTT designs with parameters suitable for post quantum key-encapsulation mechanisms (`CRYSTALS-Kyber` [31], `NewHope` [6]), key signature schemes (`CRYSTALS-Dilithium` [32], `Falcon` [33], `qTESLA` [5]) as well as homomorphic encryption applications (`SEAL` [19], `CryptoNets` [7]) using the parametric hardware generator and quantify the area-cost and the latency of the resulting designs.
- We develop software-based NTT running on soft-core RISC-V architectures and realize it on reconfigurable hardware. To conduct a thorough analysis, we instantiate three such architectures and investigate trade-offs between area-cost and throughput.
- We build HLS-based NTT implementations through a commercial C/C++-to-FPGA framework. To explore the design space of possible solutions, we evaluate the impact of different pragmas and HLS parameters on the latency and area-cost metrics.
- We conduct a comprehensive analysis of all the resulting designs and compare different design methods for implementing NTT architectures.

In particular, we demonstrate the genericness of our work by tuning it to the NTT of two dissimilar applications: A homomorphically encrypted deep neural network inference tool (`CryptoNets`) [7] and post-quantum key-encapsulation mechanism (KEM) and digital signature

1. **Code is available at https://github.com/acmert/parametric-ntt**

schemes (`CRYSTALS-Kyber` [31], `CRYSTALS-Dilithium` [32], `NewHope` [6], `qTESLA` [5], `Falcon` [33]). While the former application enables privacy-friendly neural network classification on encrypted data, the latter secures critical cyber-infrastructure against quantum computer attacks. Our generator is profitably used to obtain the *first NTT hardware accelerator* for the parameter sets of both applications on FPGAs.

Fig. 1 shows the summary of our results for a particular NTT with n=512 and K=14 used in `NewHope-512`. The results, in $\log_2$ scale, quantifies the superiority of hand-tuned hardware over HLS- and processor-based implementations. For a similar area (LUT) cost, the hardware achieves a 11.6× and 336.6× faster design than HLS- and processor-based design, respectively. Surprisingly, the implementation results of HLS-based method is not clearly better than the processor-based approach. This is significant because while we developed an HLS-friendly algorithm and rigorously explore the HLS directives for improving results, processor-based implementation is a push-button compilation of the reference software—note that up to a factor of 8 improvements are further possible for the NTT over the reference C code with assembly optimizations on embedded processors [34]. Furthermore, the results show that our hardware generator automates the design space exploration of hardware—the results show a coverage of 61.9× in area and 32.5× in latency. Such a coverage is not yet possible with HLS tools because they fail to generate a hardware with more than 8 PEs.

The rest of the paper is as follows. Section 2 gives the background on NTT and the related prior work. Section 3 introduces the parametric hardware design with novel optimizations. Section 4 discusses the HLS-based design method and tuning the HLS framework for efficient exploration of the design space. Section 5 presents the software-based design method and the target RISC-V architectures. Section 6 compares the resulting implementations among themselves and with prior work, and Section 7 concludes the paper.

2 Background and Prior Work

In this section, we give a brief definition of arithmetic operations extensively utilized in the lattice-based cryptography. For the rest of the paper, n, q, $K = \lceil \log_2 q \rceil$ and B will be used to denote the degree of the polynomial ring, the coefficient modulus, the coefficient modulus size and the number of PEs or the butterfly units used, respectively, where the PEs or butterfly units are the fundamental building blocks of NTT hardware.

2.1 The Number Theoretic Transform

The multiplication of two polynomials with large degrees defined over the ring of polynomials $Z_q[x]/\phi(x)$ is the fundamental and the most time-consuming operation utilized in lattice-based cryptography applications, where $Z_q[x]$ and $\phi(x)$ denote the polynomials with coefficients in Z_q and the reduction polynomial, respectively. The multiplication operation in the ring basically takes the polynomials $A(x)$ and $B(x)$, namely $A(x)=\sum_{i=0}^{n-1} a_i x^i$ and $B(x)=\sum_{i=0}^{n-1} b_i x^i$, as inputs, where coefficients a_i and b_i are in Z_q which denotes the numbers $\{0, 1, ..., q-1\}$. The multiplication operation returns the output polynomial $C(x)=\sum_{i=0}^{n-1} c_i x^i$.

Algorithm 1 NTT-based Polynomial Multiplication [35]

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

The classical schoolbook polynomial multiplication technique is far from providing a fast and practical polynomial multiplication operation as it has a complexity of $\mathcal{O}(n^2)$. Besides, the resulting polynomial still needs to be reduced with reduction polynomial, $\phi(x)$. Consequently, the NTT-based polynomial multiplication technique, which reduces the complexity to $\mathcal{O}(n \log n)$, is generally utilized in efficient lattice-based cryptography implementations. The NTT-based polynomial multiplication technique employs NTT/INTT (inverse-NTT) operations to convert costly polynomial multiplication operation into much cheaper coefficient-wise multiplication operations.

When the reduction polynomial, $\phi(x)$, is selected as (x^n+1), the reduction operation after the polynomial multiplication can be avoided using a special technique, called $negative\ wrapped\ convolution$, that directly renders the resulting polynomial $C(x)$ of degree $n-1$. The technique multiplies the coefficients of the input and output polynomials with the powers of Ψ and Ψ^{-1}, respectively, where Ψ is a primitive $2n$-th root of unity in Z_q satisfying $\Psi^{2n} \equiv 1 \pmod q$ and $\forall i < 2n$, $\Psi^i \neq 1 \pmod q$. It also requires $\phi(x) = (x^n+1)$ and $q \equiv 1 \pmod{2n}$. The NTT-based polynomial multiplication with $negative\ wrapped\ convolution$ is described in Algorithm 1, where $\odot$ denotes coefficient-wise modular multiplication and NTT and INTT stands for number theoretic transform and inverse number theoretic transform, respectively.

NTT is a variant of Discrete Fourier Transform defined over the ring $Z[x]_q/\phi(x)$. The n-point (pt) NTT operation takes an $(n-1)$ degree polynomial, $A(x) = \sum_{i=0}^{n-1} a_i x^i$, in the polynomial domain as input and transforms it into an $(n-1)$ degree polynomial, $\bar{A}(x) = \sum_{i=0}^{n-1} \mathcal{A}_i x^i$, in the NTT domain. Each coefficient in the NTT domain, $\mathcal{A}_i$, is defined as $\mathcal{A}_i = \sum_{j=0}^{n-1} a_j \omega^{ij}$ in Z_q. Similarly, n-pt INTT operation produces coefficients in the polynomial domain with $a_i = n^{-1} \sum_{j=0}^{n-1} \mathcal{A}_j \omega^{-ij}$ in Z_q.

The NTT operation uses the constant called twiddle factor, $\omega \in Z_q$, which is a primitive n-th root of unity in Z_q. The twiddle factor should satisfy the conditions $\omega^n \equiv 1 \pmod q$ and $\forall i < n$, $\omega^i \neq 1 \pmod q$, where $q \equiv 1 \pmod n$. Similarly, the INTT operation uses the modular inverse of twiddle factor, ω^{-1}, in Z_q.

There is a plethora of NTT algorithms proposed in the literature [36], [37], [38], [18], which are implemented in different works as shown in Table 1. In this work, we utilize the iterative NTT algorithm, given in Algorithm 2 [36]. The iterative NTT algorithm uses Gentleman-Sande butterfly configuration, whereby input and output of the butterfly unit are in normal and bit-reversed order, respectively. The

Algorithm 2 Iterative NTT Algorithm [36]

Input: $A(x) \in Z_q[x]/(x^n+1)$
Input: primitive n-th root of unity $\omega \in Z_q$, $n = 2^l$
Output: $\overline{A}(x) = \mathbf{NTT}(A) \in Z_q[x]/(x^n+1)$
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: $i_e \leftarrow 2 \cdot j \cdot m + k$ ▷ *index calculation*
6: $i_o \leftarrow 2 \cdot j \cdot m + k + m$
7: $i_w \leftarrow (2^{i-1} \cdot k)$
8: $U \leftarrow A[i_e]$ ▷ *read data*
9: $V \leftarrow A[i_o]$
10: $W \leftarrow \omega^{i_w} \pmod q$
11: $E \leftarrow (U+V) \pmod q$ ▷ *execute butterfly*
12: $O \leftarrow (U-V) \cdot W \pmod q$
13: $A[i_e] \leftarrow E$ ▷ *write data*
14: $A[i_o] \leftarrow O$
15: **end for**
16: **end for**
17: **end for**
18: return A

iterative INTT operation can be implemented with Algorithm 2 by simply using ω^{-1} instead of ω and dividing the coefficients of the output polynomial with n in Z_q.

An n-pt NTT operation consists of $\log_2 n$ stages, in each of which $(n/2)$ butterfly operations are performed (see Steps 5-14 of the Algorithm 2). Thus, an n-pt NTT performs $(\log_2 n) \cdot (n/2)$ butterfly operations in total, whereby one butterfly operation takes a, b, ω^i as inputs and calculates $(a+b) \pmod q$ and $(a-b) \cdot \omega^i \pmod q$. A butterfly operation in Steps 5-14 of the Algorithm 2 can be performed mainly in four steps: i) *index calculation*, ii) *load data*, iii) *execute butterfly* and iv) *store data*. In the *index calculation* and *load data* steps, the read addresses for the input data of the butterfly operation are generated and the input data are read from the memory. In the *butterfly* and *store data* steps, the butterfly operation is performed and the output data is written into the memory using the read addresses generated during the *index calculation* step.

An NTT operation can be parallelized by performing multiple butterfly operations concurrently. The input of an NTT stage is the output of the previous NTT stage. Thus, an NTT operation has limited parallelism for a given input, confined to the number of butterfly operations in one stage, namely at most $(n/2)$ butterfly operations can be performed in parallel. On the other hand, memory operations become more complicated due to concurrent accesses to inputs of the multiple butterfly operations.

2.2 Prior Implementations of NTT

Different algorithms and architectures targeting various platforms for NTT are proposed in the literature in order to facilitate practical implementations of lattice-based cryptography. There are many NTT architectures proposed in the literature using different methods and targeting different platforms: low-level hardware implementations ([20], [21], [22], [23], [24], [18], [25], [11], [26], [27], [28], [29], [30]), HLS implementations ([12], [13], [14]) and software implementations ([7], [19], [39], [40], [41], [15], [16]).

The works [23], [24] use the iterative NTT algorithm [36] and propose balanced NTT implementations considering I/O requirements of the CPU-FPGA system. In [18], two different NTT architectures utilizing Iterative [36] and Four-Step [41] NTT algorithms are presented. The NTT architectures are utilized to accelerate the encryption and decryption operations implemented in SEAL library [19] on the FPGA. In [22] and [30], the authors implement Iterative NTT algorithm on FPGA; while the work in [22] implements modular reduction operation using a *sliding window* method, [30] uses Barrett modular reduction. The work in [29], which targets both FPGA and ASIC platforms, proposes a method eliminating the first stage of the NTT operation.

In [27], [11] and [26], the works target ASIC platform. In [11], the authors propose a reconfigurable crypto-processor supporting multiple lattice-based cryptosystems. The work in [11] utilizes the constant-geometry NTT algorithm [38], which features constant read/write pattern in each stage of NTT operation and uses single-port RAM in order to reduce hardware cost. Also, it proposes a configurable Barrett modular reduction implementation. In [27], the authors focus on low-power NTT design for battery-powered IoT devices with a discussion on countermeasures for side-channel attacks. The method in [26] proposes an accelerator for R-LWE based cryptosystems for multiple parameter sets, which uses 16 butterfly units in its NTT core and divides large NTT operations into smaller 64-pt NTTs.

The works in [12], [13], [14] use HLS to generate NTT hardware targeting FPGA. These works use HLS-friendly NTT algorithms and HLS directives to generate efficient NTT hardware. In [12], the authors propose an NTT-based polynomial multiplier implementing the memory efficient NTT algorithm introduced in [20]. In [13], for loop structure of NTT algorithm is modified for efficiently applying Vivado HLS directives. In [14], the authors propose an FFT-based polynomial multiplier which uses an FFT algorithm with *ping-pong* memory buffer utilizing 2D arrays in HLS.

Different software implementations for NTT were also proposed in the literature. The software libraries CryptoNets [7], SEAL [19], HElib [39] and NFLlib [40] provide efficient software implementations of arithmetic blocks for the lattice-based homomorphic encryption schemes, which include efficient NTT implementations. The work in [15] proposes a high-speed implementation of `Crystals-KYBER` [31] on ARM Cortex-M4 microcontroller, where an NTT implementation for n=256 and q=3329 was also introduced. The work in [16] proposes an AVX2 optimized NTT implementation which utilizes a modified Montgomery modular reduction algorithm. In [17], `NewHope-1024` [6] is implemented on ARM Cortex-M0 and Cortex-M4 micro-controllers with NTT/INTT implementation optimized for the parameter set of `NewHope-1024`. Also, there are GPU accelerators for NTT and homomorphic encryption applications such as cuHe [41].

3 Design Method I: Parametric Hardware Generator Design

NTT computations have three parts: loading related data (Steps 5 and 6 in Algorithm 2), performing arithmetic computations (Steps 11 and 12 in Algorithm 2) and storing the

Algorithm 3 Word-Level Montgomery Reduction Algorithm for NTT-friendly primes [18]

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 $R = 2^{w \cdot L}$
1: $L \leftarrow \lceil \frac{K}{w} \rceil$
2: $T \leftarrow C$
3: **for** i from 0 to L **do**
4: $\quad T1_H \leftarrow T >> w$
5: $\quad T1_L \leftarrow T \pmod{2^w}$
6: $\quad T2 \leftarrow two's\ complement\ of\ T1_L$
7: $\quad C_{in} \leftarrow T2[w-1] \ \vee \ T1_L[w-1]$
8: $\quad T \leftarrow T1_H + (q_H \cdot T2[w-1:0]) + C_{in}$
9: **end for**
10: $T4 \leftarrow T - q$
11: **if** $(T4 < 0)$ **then** $Res = T$ **else** $Res = T4$

result (Steps 13 and 14 in Algorithm 2). Within each PE, the so-called butterfly units execute arithmetic computations, which are composed of modular addition, subtraction, and multiplication. To achieve a higher throughput, it is possible to unroll NTT loops and parallelize the butterfly operations by using multiple PEs.

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

The NTT hardware generator takes the polynomial degree, the coefficient size and the number of PEs as input parameters, and generates an NTT hardware unit optimized for the given parameters. It is notable that the design provides flexibility not only for polynomial degree and coefficient size, but also the number of PEs, which improves the throughput of the hardware. The same unit can also execute the INTT.

3.1 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 parameterized version of both parts.

There are mainly two alternative methods for efficient modular reduction: Montgomery [24] and Barrett algorithms [11]. Their selection for a parametric hardware is a non-trivial design decision. Banerjee *et. al.*, for instance, uses a Barrett modular reduction hardware [11], 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 reduction can offer a more efficient solution than the Barrett algorithm. Nevertheless, the baseline Montgomery has to be optimized for NTT primes and extended to support various polynomial degrees and coefficient sizes. To that end, we propose a Montgomery reduction unit that generalizes the word-level Montgomery modular reduction algorithm for NTT-friendly primes [24], whereby the word size is derived from the parameters. Compared to the configurable Barrett reduction [11], our solution either reduces the number of multipliers or results in smaller multiplier units. The implementation results show an advantage in cycle count when the hardware uses the same number of PEs as shown in Table 6.

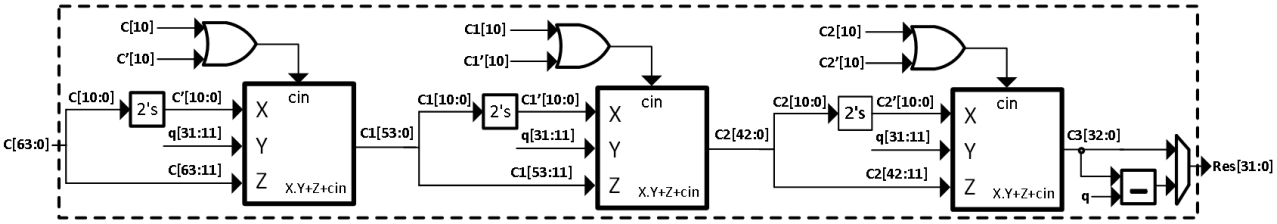
Algorithm 3 provides the details of our generalized algorithm. The algorithm requires different number of integer multipliers of varying bit lengths depending on q and n. The algorithm utilizes the property of $q \equiv 1 \pmod{2n}$, which should be satisfied by any NTT-friendly prime using *negative wrapped convolution* 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 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 $(A \cdot B \pmod{2^w}) \cdot \mu$ in the Montgomery modular reduction to the two's complement (see Step 6 of Algorithm 3).

The Montgomery reduction algorithm takes $C = A \cdot B$ as input and calculates the output $A \cdot B \cdot R^{-1} \pmod q$, where $R = 2^{w \cdot L}$ 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.

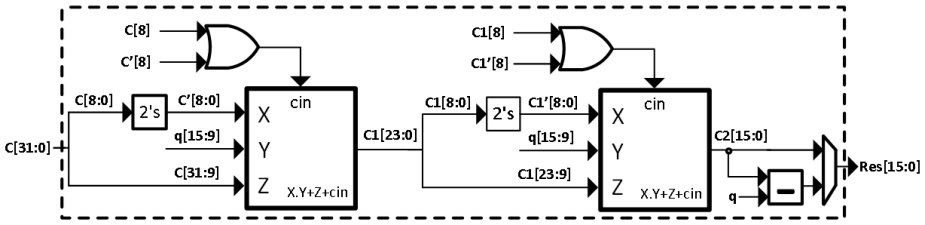
The proposed word-level Montgomery modular reduction algorithm divides reduction operation into a set of multiply and accumulate (MAC) operations. Namely, it performs $X \cdot Y + Z + C_{in}$ operation (see Step 8 of Algorithm 3), which can be implemented using DSP blocks in FPGAs, for different number of times for different arithmetic configurations. Therefore, the algorithm itself is amenable to generating flexible designs. Our hardware generator makes use of this to automatically generate the reduction hardware for the given polynomial degree and coefficient size.

Fig. 2 illustrates two examples of modular reduction hardware units for (a) polynomial degree of 1024 with 32-bit modulus and (b) polynomial degree of 256 with 16-bit modulus, where the rectangular boxes represent the functional units to perform $X \cdot Y + Z + C_{in}$. Both designs offer advantage over the Barret method [11]—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 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 coefficient size as a parameter. Each input of the multiplier is divided into 16-bit pieces and



(a) For n=1024 and K=32



(b) For n=256 and K=16

Fig. 2: Word-Level Montgomery Modular Reduction Hardware

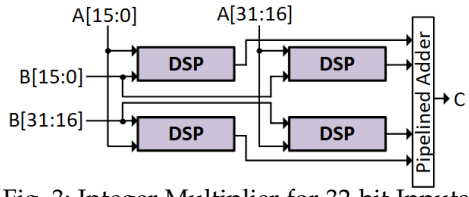


Fig. 3: Integer Multiplier for 32-bit Inputs

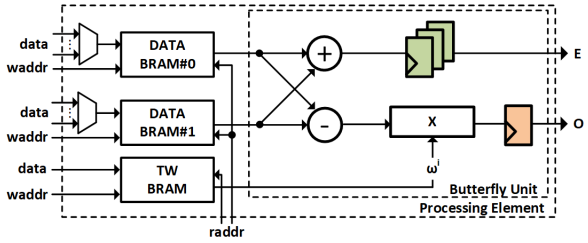


Fig. 4: PE and the Butterfly Unit

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 carry save adders. The proposed integer multiplier is also fully pipelined and it can produce one multiplication result per clock cycle. Fig. 3 shows the hardware of 32-bit multiplication as an example, 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.

3.2 PEs and Butterfly Units

As we already obtain the efficient hardware for the core modular arithmetic, it is now time to 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 [36] corresponding to Steps 8–12 of Algorithm 2. Fig. 4 illustrates one PE. PEs take two coefficients and one twiddle factor as inputs, perform the butterfly operation, and 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 TW BRAM, stores the twiddle factors (with Montgomery correction), which are the design-time constants pre-computed based on input parameters.

Fig. 4 depicts that the even coefficient is the output of the modular addition operation (Step 11 in Algorithm 2) and odd output coefficient is the output of the modular subtraction and multiplication (Step 12 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 (namely, the polynomial degree and the coefficient size). The number of PEs can only be a power-of-2 and their maximum is $(n/2)$ for an n-pt NTT.

3.3 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. 5 (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. 5 (b) demonstrates the one PE case, where coefficients in 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

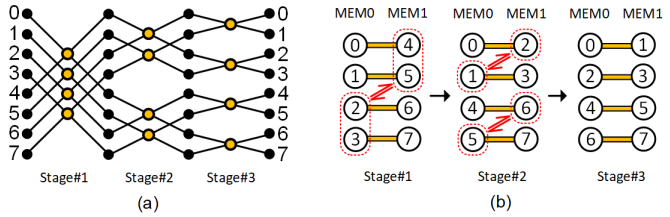


Fig. 5: (a) Coefficient Access Pattern; (b) Memory Access Pattern for n=8

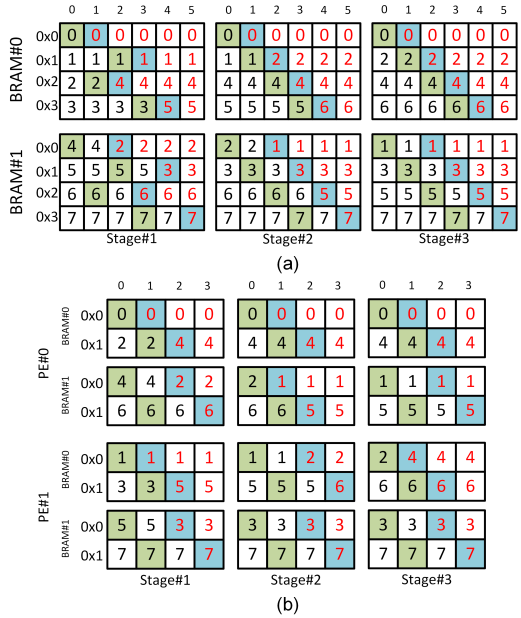


Fig. 6: Memory Access for the 8-pt NTT with (a) One PE, (b) Two PEs

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 into the memory blocks based on the pairing of the next stage. Fig. 5 (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. 4. 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

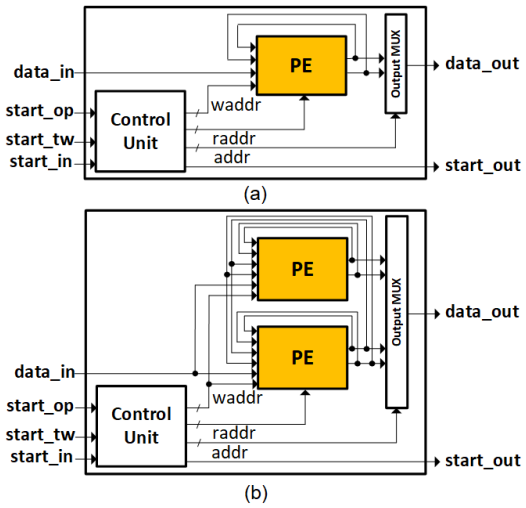


Fig. 7: NTT Hardware (a) with one PE; (b) with two PEs

pairs should be written into the first and second memories, respectively. For example, as shown in Fig. 5, 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 gives the memory access examples for 8-pt NTT with one and two PEs. Green and blue boxes represent the read and write operations, respectively, 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 (see Fig. 6 (b)), the PEs perform the butterfly operation for the first half of the coefficient pairs first. In this case, the first and second PEs can read coefficients (0,4) and (1,5) at the same time because coefficients 0, 1 and 4, 5 will be written into different memory blocks. Since we have two PEs instead of one, latency of each NTT stage is reduced.

Fig. 7 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 the 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 are read via output multiplexers.

4 Design Method II: HLS-Based Design

We aim synthesizing our design-time flexible NTT hardware using the popular Xilinx Vivado HLS tool. Vivado HLS takes C or C++ codes as input and generates synthesizable Verilog or VHDL codes. It also provides specific directives (*pragmas*) for users to intervene the C/C++-to-RTL synthesis process and optimize the generated RTL design. The Vivado HLS tool offers several optimization parameters such as

Algorithm 4 HLS-Friendly NTT Algorithm

Input: $A[n]$ (n element input array)
Input: $\omega[n/2]$ where $\omega[i] = \omega^i \pmod q$
Input: B (number of butterfly operations in parallel)
Output: $\overline{A}[n] = \mathbf{NTT}(A[n])$

1: $l = \log_2 n$
2: $v = n/2$
3: `STAGE_LOOP:`
4: **for** i from 0 by 1 to $(l-1)$ **do** ▷ *outer loop*
5: `BUTTERFLY_LOOP:`
6: **for** s from 0 by B to $(v-1)$ **do** ▷ *middle loop*
7: `IDX_CALC_LOOP:` ▷ *index calculation*
8: **for** b from 0 by 1 to $(B-1)$ **do** ▷ *inner loop#1*
9: $j[b] \leftarrow (s+b) >> (l-1-i)$
10: $k[b] \leftarrow (s+b)\ \&\ ((v >> 1) - 1)$
11: $i_e[b] \leftarrow j[b] \cdot (1 << (l-i)) + k[b]$
12: $i_o[b] \leftarrow i_e[b] \cdot (1 << (l-i-1))$
13: $i_w[b] \leftarrow (1 << i) \cdot k[b]$
14: **end for**
15: `MEM_READ_LOOP:` ▷ *read data*
16: **for** b from 0 by 1 to $(B-1)$ **do** ▷ *inner loop#2*
17: $U[b] \leftarrow A[i_e[b]]$
18: $V[b] \leftarrow A[i_o[b]]$
19: $W[b] \leftarrow \omega[i_w[b]]$
20: **end for**
21: `OP_LOOP:` ▷ *butterfly operation*
22: **for** b from 0 by 1 to $(B-1)$ **do** ▷ *inner loop#3*
23: $E[b] \leftarrow (U[b] + V[b]) \pmod q$
24: $O[b] \leftarrow (U[b] - V[b]) \cdot W[b] \pmod q$
25: **end for**
26: `MEM_WRITE_LOOP:` ▷ *write data*
27: **for** b from 0 by 1 to $(B-1)$ **do** ▷ *inner loop#4*
28: $A[i_e[b]] \leftarrow E[b]$
29: $A[i_o[b]] \leftarrow O[b]$
30: **end for**
31: **end for**
32: **end for**
33: return A

`loop unrolling`, `loop merging`, `pipeline` and `array partitioning`, among others [42].

In order to generate an efficient hardware from C/C++ code, the code should include appropriate HLS directives. This may require re-writing or changing the structure of the code [13], in addition to the exploration of optimization parameters. Indeed, we first observe that the straightforward transition of the software leads to inefficient results or may not even produce a synthesizable code. Specifically, in the original algorithm (Algorithm 2), the NTT operation has 3 loops where indices of middle and inner loops depend on the index of the outer loop. This loop structure makes applying HLS directives complicated and causes the HLS tool to improperly handle the synthesis process.

Algorithm 4 shows our proposed HLS-friendly Iterative NTT algorithm. In order to overcome HLS problems and make the C++ code more HLS-friendly, we modified the loop structure of the NTT algorithm such that dependencies between indices of the loops are removed and the trip count of each loop structure is fixed. In Algorithm 4, the trip count of the outer loop is the number of stages in the n-pt NTT operation, $\log_2 n$, as in Algorithm 2. In the middle loop, we modified trip count to $(n/2)$, which is the number of butterfly operations in one stage of an n-pt NTT. Finally, we set the trip count of the innermost loop as the number of butterfly operations to be performed in parallel, which can be at most $(n/2)$ for an n-pt NTT operation. Then, we divided butterfly operation into 4 separate loops with the same trip count, whereby four loops perform four steps of a butterfly operation as explained in Section 2.1.

TABLE 2: Vivado HLS *pragmas* used in our work

Code	*pragma*
`array A[n]`	`#pragma HLS ARRAY_PARTITION`
`array ω[n/2]`	`#pragma HLS ARRAY_PARTITION`
`STAGE_LOOP`	–
`BUTTERFLY_LOOP`	`#pragma HLS PIPELINE`
`IDX_CALC_LOOP`	`#pragma HLS UNROLL`
`MEM_READ_LOOP`	
`OP_LOOP`	
`MEM_WRITE_LOOP`	

Based on the implemented algorithm and the code section where the directives are used, different combinations of directives can have different impacts on the architecture. Therefore, we applied a set of different directives to the C++ implementation of NTT operation and selected `loop unrolling`, `pipeline` and `array partitioning` directives at different code parts as shown in Table 2.

The `loop unrolling` directive (UNROLL) converts a single operation performed by a loop into multiple independent copies of the operation performed in the loop body and runs these operations concurrently. It can unroll a loop fully or partially, and it increases the parallelism and the performance of the operation. This directive is applied to the innermost four loops in our design where it generates B butterfly units running concurrently. The `pipeline` directive (PIPELINE) allows concurrent execution of operations and reduces the interval required to start processing a new input, e.g., it allows an operation to take a new input every clock cycle. We use PIPELINE directive at the middle loop to pipeline the four steps of butterfly operations.

Memory control is one of the most challenging part of the NTT design as it requires many read/write operations with an irregular pattern. The addressing becomes even more complex with the increasing number of parallel butterfly operations. For instance, an NTT operation with $n = 256$ and $B = 8$ requires to read 16 coefficients and 8 coefficients from the arrays $A[n]$ and $\omega[n]$, respectively, at the same time. This enforces a structure with data stored on multiple small memory blocks or a large memory with multiple read/write ports (i.e., a register file). The straightforward software implementation cannot realize this structure—it generates a single block RAM with insufficient bandwidth. But the `array partitioning` (ARRAY_PARTITION) directive can automate the partition of an array into smaller memories or individual registers instead of instantiating a single large memory block. Therefore, it can effectively increase the read/write ports of a memory, and hence, the throughput, at the expense of more hardware resources.

In the proposed design, we applied the ARRAY_PARTITION directive to the array, $A[n]$, storing the input polynomial, and the array, $\omega[n]$, storing twiddle factors. Here, we used two different approaches: i) we partitioned each array into a register file using the `complete` option of the ARRAY_PARTITION directive, where any data can

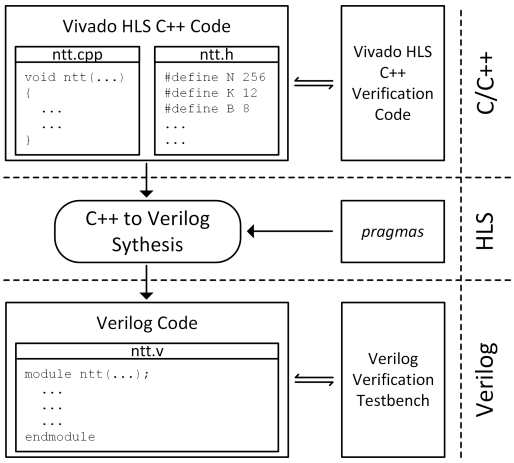


Fig. 8: Vivado HLS Flow

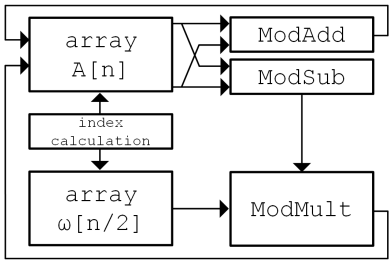


Fig. 9: NTT hardware generated by Vivado HLS tool

be read/written at any time, ii) we partitioned each array into a number of BRAMs using the `block` option of the ARRAY_PARTITION directive so that the algorithm can access multiple data at the same time. Although the former approach generates faster design, it uses high hardware resources due to large multiplexers generated for reading/writing multiple data. We also used the `ap_int.h` library provided by Vivado HLS tool that contains bit accurate models for the C++ code. This helps us to set K and bit-lengths of intermediate calculation steps.

Fig. 8 shows the design flow of the proposed NTT architecture in Vivado HLS. The flow starts with the C++ implementation of the NTT operation and its verification. Then, Vivado HLS directives are applied to the code, the synthesis is performed and the Verilog code is generated. Finally, the generated Verilog code is verified with a Verilog testbench. **The proposed HLS-based design uses conventional Montgomery algorithm for the modular reduction operation as specified in Algorithm 5. Although our parametric hardware generator uses word-level Montgomery algorithm, our HLS-based design with word-level Montgomery algorithm leads to inefficient architectures due to loop-based structure of the word-level Montgomery algorithm. Also, as the number of PEs is increased, the HLS tool shows significant increase in hardware resources and synthesis time because the HLS tool cannot resolve unrolling efficiently although it shows similar performance results with the conventional Montgomery algorithm. Therefore, we utilize conventional Montgomery algorithm for modular reduction operation in our HLS-based design.** The proposed work implements constant-time modular addition and subtraction operations [24]. Fig. 9 illustrates the architecture of the NTT hardware generated by the Vivado HLS tool.

Algorithm 5 Montgomery Modular Reduction Algorithm

Input: $C = A \cdot B$ (a $2K$-bit positive integer)
Input: q (a K-bit modulus)
Input: $\mu = -(q^{-1}) \pmod R$ where $R = 2^K$
Output: $Res = C \cdot R^{-1} \pmod q$
1: $T = C \cdot \mu$
2: $X = T \pmod R$
3: $Y = C + X \cdot q$
4: $U = Y >> K$
5: $V = U - q$
6: **if** $(V < 0)$ **then** $Res = U$ **else** $Res = V$
7: return Res

The NTT operation in [12] is implemented using a loop structure with 3 nested loops, where trip counts of the inner loops are not fixed. This complicates applying HLS directives efficiently and setting the amount of parallelism in the generated hardware architecture. Our work is superior to the work in [12] because our modified loop structure uses fixed trip counts which ease applying HLS directives and we can change the parallelism easily by tuning a single parameter. In [14] and [13], the authors modify the NTT loop structure such that it uses two nested loops with fixed trip counts. However, both works lack flexibility for setting the level of parallelism. The method in [14] manually unrolls its inner loop with a factor of 2 which limits the parallelism and throughput flexibility. Although it shows better performance for a single parameter set, hardware resource usage details are not provided (as shown in Section 6). Similarly, the method in [13] uses manual unrolling and provides only partial flexibility by changing the structure of the code. In our work, the code structure is fixed and only parameter sets are changed for yielding different architectures.

A key takeaway of our HLS experience on NTT was that, in order to generate an efficient hardware, the designer needs an in-depth understanding of the resulting hardware designs and the effect of tool's parameters on the synthesized hardware, which arguably contradicts the goal of enabling easy solutions for software developers. Section 6 discusses the design complexity of the HLS method with respect to the others and compares the efficiency of resulting hardware.

5 Design Method III: Processor-Based Design

Processor-based designs typically have the best flexibility but the worst performance. We choose the RISC-V based ISA to carry out this design method because RISC-V is open source and is especially well suited for embedded domains, providing energy-efficient solutions with a reasonably high performance. The recent study claims that RISC-V is the best and safest choice for a free, open RISC ISA [43].

We build our software environment on two RISC-V cores to quantify the trade-offs between performance and area: `PicoRV32` [44] and the `Berkeley Out-of-Order Machine (BOOM)` [45].

`PicoRV32`. `PicoRV32` core targets embedded applications with limited resources. It therefore has a simple, area-optimized architecture with a single-issue, in-order

core. We configure the `PicoRV32` core for both the `RV32I`, which supports 32-bit base integer instructions, and the `RV32IM` [46], which further supports 32-bit integer multiplication-division instructions.

`BOOM`. `BOOM` core targets applications with high-performance needs. It therefore has a complex architecture employing out-of-order processing, hardware branch prediction, and speculative execution, among other performance-optimization methods. `BOOM` works with `RV64GC` ISA (also known as the 64-bit IMAFDC), which supports 64-bit base integer instructions, integer multiplication and division instructions, atomic instructions, single- and double-precision floating-point instructions and compressed instructions.

We isolate the NTT algorithms from NIST Round 2 submissions' reference implementation of `NewHope`, `CRYSTALS-Dilithium`, `Falcon`, `qTESLA` and NIST Round 1 submission's reference implementation of `CRYSTALS-Kyber`. We define the pre-computed twiddle factors as global arrays in our isolated NTT source C files. This procedure has resulted in 9 different NTT C files before starting the compilation.

We compile our source files with `riscv-gnu tool-chains` which are referenced in the corresponding git repositories [44], [45]. Fig. 10 outlines the source code compiling flow. The RISC-V gcc compiler compiles the reference software from C source language code to object file. Based on target architecture, either the `PicoRV32` or `BOOM` linker links the object files and generates the executable output. The `PicoRV32` design requires dumping the resulting hex files to memory. Therefore, we use `objcopy` command and a Python script to format the output of `.tmp` file into a `.hex` file, which is then dumped to the memory of `PicoRV32`. **For the `BOOM` design, the executable directly drives the core. For both designs, the final step of the compiling flow is using `objdump` command to get the assembly output and program counter values needed to time the NTT execution.**

To provide a fair comparison with respect to hardware and HLS-based methods, we only record the cycle count of the NTT operation, isolated from other instructions such as creating NTT inputs, organizing twiddle factor values, and other instructions under the labels: *putchar, setStats, tohost-exit, callexitprocs, impuredata, memcpy, globalimpureptr, impureptr, bssstart, abort, printstr, printhex, memset, -init, strcmp, GMb, register-exitproc, do-global-dtors-aux, atol*. We employ the assembly dump file to identify the program counter value and use those to achieve this isolation.

Fig. 11 illustrates the implementation flow. The input to this flow is the `.hex` file obtained from the software flow. The correctness of the behavioral simulation is verified with `ModelSim` and `Verilator` simulators. While ModelSim simulates the `PicoRV32`, Verilator simulates for the `BOOM`. The simulator records provide the count cycles for the different NTT algorithms' executions. To obtain pure cores' hardware utilization result, we followed the Synthesis and Place & Route flow only with the RISC-V cores. Finally, we combine the core utilization results and the algorithms' memory footprint in area calculation.

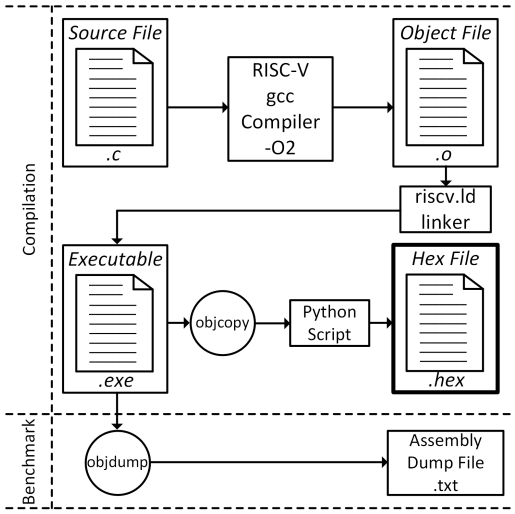


Fig. 10: NTT source code compiling flow for RISC-V

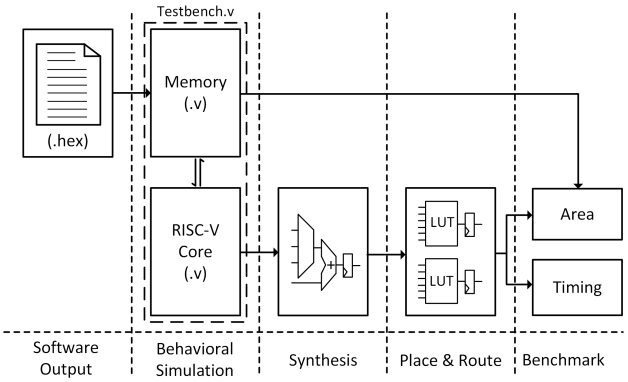


Fig. 11: RISC-V Implementation Flow

6 Implementation Results and Comparisons

To provide a fair comparison of the design methods, we use a common set of EDA tools and we target the same FPGA. We also compare our work with previous results in the literature and comment on the design complexity.

6.1 Experimental Setup

The parametric hardware generator code is written in Verilog RTL and the generated NTT hardware are synthesized, placed, and routed using Xilinx Vivado tools on the Xilinx Virtex-7 FPGA xc7vx690tffg1761-2. The proposed HLS-friendly software code is written in C++ and is enhanced with Xilinx HLS pragmas. The NTT software for the processor-based designs are obtained from the reference C source code of NIST submissions [31], [6], [32], [33], [5] and CryptoNets [7]. The details of processor implementation flows are discussed in Section 5.

6.2 Implementation Results of the Design Methods and Comparison to Prior Work

We first provide the results and comparison of the three design method we propose in this work.

- For our parametric hardware generator, the area-cost, number of clock cycles (CC) needed to finish one NTT, and LUT/latency improvements for 7 different (n, K) sets with 1, 8 and 32 PEs are shown in Table 3. As the number of PEs increases from 1 to 32, the

latency improves up to $25.4\times$ at the expense of more hardware resources. The increase in the number of PEs (which can be as much as $(n/2)$) further improves the performance for larger n values. As expected, the parametric hardware generator yields (at least an order-of-magnitude) better results than other methods in latency and/or area.

- For HLS implementations, the area-cost, number of CC to finish one NTT and latency improvements for 7 different (n, K) sets with 1 and 8 PEs are shown in Table 4. We obtained two different synthesis results, register memory and BRAM memory, for each parameter set as explained in Section 4. In Table 4, the first and the second rows reflect the results for the implementation with register and BRAM memories, respectively, for a given parameter set. The implementations with register memory show slightly better performance at the expense of more hardware resources. The first 4 columns of Table 4 highlight the impact of Montgomery modular reduction algorithm and the proposed HLS-friendly NTT algorithm. The Montgomery modular reduction algorithm improves the performance up to $5\times$ and the HLS-friendly NTT algorithm further improves the performance up to $767\times$ compared to the baseline design (iterative NTT algorithm without Montgomery modular reduction algorithm) with similar hardware resources. Compared to the parametric hardware generator, the increase in the number of PEs has a weaker effect—the performance increases only by $2\times$ as the number of PEs is increased by $8\times$, which reveals the inefficiencies in the HLS tools. Vivado HLS cannot synthesize the hardware with register memory for $(2048, 30)$ and $(4096, 60)$ because the `array partitioning` pragma is limited to arrays with length up to 1024. The number of PEs is limited to 8 because no significant performance improvement is observed with 16 or more PEs.
- For RISC-V implementations, the area-cost, number of CC to finish one NTT, and latency improvements for 6 different (n, K) sets and 3 different RISC-V configurations are shown in Table 5. The second implementation , `RV32IM`, improves the latency up to $40\times$ over `RV32I` at the expense of 4 extra DSPs. The third implementation, `BOOM-RV64GC`, improves the latency up to $400\times$ at the expense of $220\times$ increase in LUT count. HLS does not clearly outperform RISC-V but instead provides a trade-off: e.g., for (n, K)=(512,14), it reduces the CC by $57\times$ at the expense of $11\times$ increase in area-cost. The aim of this work is to evaluate flexible solutions for the NTT implementation. Therefore, we pick the NIST reference submission NTT source codes written in C and compile them to the RISC-V ISA. Some NIST submissions or related follow-up works in the literature include assembly optimizations, which may achieve a performance improvement on NTT by $8.3\times$ [34]. RISC-V designs thus will arguably be more competitive compared to HLS solutions.

We then compare our results with the prior work. The key takeaways are:

- 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. Also, note that only the NTT implementations of the same polynomial degree (n) and coefficient size (K) make a meaningful comparison. In Table 6, a subset of our implementations for selected parameter sets are compared with the prior works.

TABLE 3: Our Hardware Implementation Results

PE	*(n, K)*	LUTs/DSPs/BRAMs	# of CC	LUT-Lat. Impr.
1	(256, 13)	489 / 3 / 2.5	1056	–
8		2371 / 24 / 12	160	$\times 0.20, \times 6.60$
32		15888 / 96 / 48	64	$\times 0.03, \times 16.5$
1	(256, 23)	888 / 7 / 5	1096	–
8		5071 / 56 / 12	200	$\times 0.15, \times 5.48$
32		30847 / 224 / 48	104	$\times 0.02, \times 10.5$
1	(512, 14)	537 / 3 / 5.5	2340	–
8		2514 / 24 / 12	324	$\times 0.21, \times 7.20$
32		16983 / 96 / 48	108	$\times 0.03, \times 21.6$
1	(1024, 14)	575 / 3 / 11	5160	–
8		2584 / 24 / 16	680	$\times 0.22, \times 7.58$
32		17188 / 96 / 48	200	$\times 0.03, \times 25.8$
1	(1024, 29)	966 / 7 / 21.5	5210	–
8		6788 / 56 / 24	730	$\times 0.14, \times 7.13$
32		38093 / 224 / 48	250	$\times 0.02, \times 20.8$
1	(2048, 30)	991 / 7 / 45	11363	–
8		6821 / 56 / 44	1507	$\times 0.15, \times 7.54$
32		38598 / 224 / 64	451	$\times 0.02, \times 25.2$
1	(4096, 60)	2720 / 31 / 180	24708	–
8		23215 / 248 / 176	3276	$\times 0.11, \times 7.54$
32		99384 / 992 / 176	972	$\times 0.02, \times 25.4$

TABLE 4: Our HLS-Based Implementation Results

PE	*(n, K)*	LUTs/FFs/DSPs/BRAMs	# of CC	Lat. Impr.
$1^{(1)}$	(256, 13)	12336 / 6639 / 1 / –	3934226	–
		1045 / – / 1 / 2	4065298	–
$1^{(2)}$		12185 / 6390 / 3 / –	788498	$\times 5.0$
		893 / – / 3 / 2	919570	$\times 4.4$
$1^{(3)}$		12255 / 10197 / 3 / –	5124	$\times 767.8$
		979 / – / 3 / 2	6147	$\times 661.3$
$8^{(4)}$		61297 / 11187 / 24 / –	2436	$\times 1615$
		10487 / – / 24 / 16	3075	$\times 1322$
1	(256, 23)	12553 / 18763 / 6 / –	4100	–
		1213 / – / 6 / 6	5123	–
8		63241 / 19387 / 48 / –	2308	$\times 1.8$
		12565 / – / 48 / 32	3075	$\times 1.7$
1	(512, 14)	23757 / 21727 / 3 / –	11524	–
		1010 / – / 3 / 4	13827	–
8		118804 / 22841 / 24 / –	6050	$\times 1.9$
		11020 / – / 24 / 16	6915	$\times 2.0$
1	(1024, 14)	36061 / 43239 / 3 / –	25604	–
		1045 / – / 3 / 4	30723	–
8		167018 / 44373 / 24 / –	13442	$\times 1.9$
		11305 / – / 24 / 16	15363	$\times 2.0$
1	(1024, 29)	36405 / 89454 / 12 / –	25604	–
		1445 / – / 12 / 6	30723	–
8		169560 / 91578 / 96 / –	13442	$\times 1.9$
		13975 / – / 96 / 32	15363	$\times 2.0$
1	(2048, 30)	–	–	–
		1479 / – / 12 / 6	67587	–
8		–	–	–
		13886 / – / 96 / 64	33795	$\times 2.0$
1	(4096, 60)	–	–	–
		2145 / – / 45 / 22	147459	–
8		–	–	–
		17768 / – /360 / 128	73731	$\times 2.0$

$^{(1)}$: Iterative NTT algorithm (no Mont. Mod. Red.). $^{(2)}$: Iterative NTT algorithm (with Mont. Mod. Red.). $^{(3)}$: HLS-friendly NTT algorithm (with Mont. Mod. Red.). $^{(4)}$: HLS-friendly NTT algorithm (with Mont. Mod. Red. and 8 PE).

TABLE 5: Our RISC-V ISA Based Implementation Results

Design	*(n, K)*	Slice/LUTs/DSPs	# of CC	Lat. Impr.
(1)	(256, 13)	349 / 870 / –	1338795	–
	(256, 23)		1617001	–
	(512, 14)^a		1372930	–
	(1024, 14)^a		3101102	–
	(512, 14)^b		1338783	–
	(1024, 14)^b		30170940	–
	(1024, 29)		10738094	–
	(2048, 30)		23236893	–
(2)	(256, 13)	448 / 1053 / 4	481096	$\times$2.7
	(256, 23)		122600	$\times$13.2
	(512, 14)^a		399931	$\times$3.4
	(1024, 14)^a		885530	$\times$3.5
	(512, 14)^b		481096	$\times$2.8
	(1024, 14)^b		1063310	$\times$28.3
	(1024, 29)		274549	$\times$39.1
	(2048, 30)		575117	$\times$40.4
(3)	(256, 13)	– / 191K / 36	41725	$\times$32
	(256, 23)		17419	$\times$92.8
	(512, 14)^a		21456	$\times$63.9
	(1024, 14)^a		49786	$\times$62.3
	(512, 14)^b		68456	$\times$19.6
	(1024, 14)^b		97563	$\times$309.2
	(1024, 29)		27337	$\times$392.8
	(2048, 30)		58082	$\times$400

(1): `PicoRV32-RV32I`, (2): `PicoRV32-RV32IM`, (3): `BOOM-RV64GC`
^{a}: `NewHope`, ^{b}: `Falcon`

- **Albeit target device and technology differences, the results in Table 6 show that our parametric NTT hardware generator can outperform most of the existing hardware, software and high-level synthesis designs respectively by up to 30.7$\times$, 200$\times$ and 47.5$\times$ in terms of number of cycle count.** Our parametric generator can produce a hardware that is comparable to fixed setting hardware units and can even be better in some cases. Our designs can achieve either a lower area or a faster design (in cycle) compared to prior FPGA solutions. For example, our one PE design outperforms the work in [11] in terms of latency (cycle count) due to our proposed word-level Montgomery reduction unit. **Some implementations, by contrast, show better performance results than our parametric generator since these implementations are optimized for fixed parameters. For example, the works in [18] and [24] show better performance than our work since they utilize 64 PEs. However, our parametric hardware generator can outperform these implementations by simply increasing the number of PEs.** Our HLS implementations with 1 and 8 PEs show similar area$\times$latency performance with the works in [12] and [13], respectively. Although RISC-V `BOOM` implementation uses the non-optimized code of `NewHope-1024`, it shows better performance than the implementation in [17].
- Using our parametric hardware generator, we instantiate the hardware implementations of NTT for the parameters of `CryptoNets` [7] and `qTESLA` [5], which outperform the reference software by up to $25.3\times$ and $5.5\times$ on Intel Xeon processor, and $95.7\times$ and $76.5\times$ compared to HLS-based design, respectively.

We finally highlight the fast design-space exploration that can be achieved by our parametric hardware generator. To test this aspect, we sweep the parameter that controls the number of PEs used in `NewHope-512` hardware and report the implementation results in Fig. 1. Our generator is able to cover a space of 61.9$\times$ in area cost for a tradeoff of 32.5$\times$ in performance by simply tuning a parameter knob. By contrast, 5$\times$ in area and 1.9$\times$ in performance is achievable in HLS since it cannot parallelize the hardware beyond 8 PEs. By contrast, our parametric generator can parallelize up to $(n/2)$ as long as the resulting hardware fits in the FPGA.

6.3 Design Effort and Complexity

Memory control of the NTT operation is complex. It may take many man-hours to design and implement an NTT architecture with a fixed number of PEs. Besides, polynomial degree and the coefficient size (fixed or not) also has impact on the design-time of the NTT architecture. The design process gets more challenging as the number of PEs changes. For example, if the number of PEs is doubled in a design, it requires re-design of the memory structure and the control unit of the NTT architecture. Considering all parameters affecting the design-time, it requires a significant effort to design a flexible NTT hardware generator supporting different polynomial degree, coefficient size and the number of PEs. It takes approximately 450 man-hours to build entire system. Once we have the flexible NTT generator design, it only takes minutes to tune parameters and synthesize the design with desired parameter set.

Design effort for the flexible design in HLS is easier compared to the hardware. It takes about 60 man-hours to explore different optimizations and finalize the design. The most challenging part is the HLS tool. Vivado HLS tool has synthesis limitations. Even though the tool significantly reduces the design-time cost compared to the parametric hardware generator, its capabilities are limited. For example, when the number of PEs is more than 8, the synthesis time of the tool significantly increases and no remarkable improvement is observed. Similarly, when the polynomial degree is larger than 1024, memory partitioning directives may not work properly and manual memory partitioning may be required. We note that we report the HLS experience of hardware design experts, this process will be significantly harder for software developers.

The challenge in the RISC-V based design method is to build the RISC-V tool-chains and its environment. It takes approximately 290 man-hours to setup the RISC-V tool-chains and simulation environment. Although PicoRV32 environment setup process takes reasonable time, the time duration of building the BOOM environment depends on the capabilities of the host Linux machine because executing one script (e.g. ./scripts/build-toolchains.sh) may take more than 4 hours and each setup fatal error causes to start over the setup spec. The BOOM environment building stages require dozens of libraries and the platform occupies more than 40 GB area on solid storage. Once this non-recurring engineering effort is completed, RISC-V flow becomes able to support a flexible environment to execute different algorithms on one platform. After building the tool-chains, there is only one easy task: compiling the target source file, running the resulting executable and generating the benchmark results, all of which can be automated with an end-to-end wall-clock time of a few minutes.

TABLE 6: A Summary of Our Implementation Results and its Comparison to Prior Work

Met.	Work	Platform	n	K	LUT / REG / DSP / BRAM	Clock (MHz)	Latency CC	Latency μs
HLS	[12]	Virtex-7	1024	14	4737 / 3243 / 8 / 2	–	16569	76
	[13]	Virtex-7	1024	10	38984 / 30498 / 19 / 21.5	100	5291	53
			2048		46738 / 38224 / 21 / 24.5		10731	107
			4096		58082 / 44767 / 22 / 41.5		22072	221
	[14]	Zynq US+	512	17	– / – / – / –	250	1202	–
	TW-1 PE	Virtex-7	1024	14	1045 / – / 3 / 4	100	30723	307
	TW-8 PE	Virtex-7	1024	14	11305/ – / 24 / 16	100	15363	153
Software	[15][a,b]	ARM Cortex-M4	256	12	– / – / – / –	–	7725	–
	[16]	Intel Core i7-4770K	256	13	– / – / – / –	–	460	–
			1024	14	– / – / – / –	–	2784	–
	[17]	ARM Cortex-M0	1024	14	– / – / – / –	–	148517	–
		ARM Cortex-M4			– / – / – / –	–	86769	–
	[18],[19]	Intel Core i9-7900X	1024	27	– / – / – / –	–	–	14
	`CryptoNets` [7][a]	Intel Xeon CE5-1650	4096	60	– / – / – / –	–	–	195
	`qTESLA` [5][a,d]	Intel Xeon CE5-1650	1024	28	– / – / – / –	–	–	11
	TW$^{(1)}$-`Falcon`	Virtex-7	1024	14	870 / – / – / –	50	30170940	6.10^6
	TW$^{(3)}$-`Falcon`	Virtex-7	1024	14	191K / 73K / 36 / 180	50	97563	1951
	TW$^{(2)}$-`NewHope`	Virtex-7	1024	14	1053 / – / 4 / –	50	885530	17710
	TW$^{(3)}$-`NewHope`	Virtex-7	1024	14	191K / 73K / 36 / 180	50	49786	995
Hardware	[20][a]	Spartan-6	256	17	250 / – / 3 / 2	–	–	25
			512		240 / – / 3 / 2		–	50
			1024		250 / – / 3 / 2		–	100
	[21][a,b]	Virtex-6	256	13	4549 / 3624 / 1 / 12	262	–	8
	[22][b]	Zynq US	4096	30	64K / – / 200 / 400	225	–	73
	[23][b]	Virtex-7	32768	32	219K / – / 768 / 193	250	7709	51
	[24][b]	Spartan-6	1024	32	1208 / – / 14 / 14	212	–	12
		Virtex-7			34K / 16K / 476 / 228	200	80	0.4
	[18][b]	Virtex-7	1024	32	67K / – / 599 / 129	200	140	0.7
					77K / – / 952 / 325.5		80	0.4
	[25][c]	Virtex-6	256	13	1349 / 860 / 1 / 2	313	1691	5.4
			512	14	1536 / 953 / 1 / 3	278	3443	12.3
	[11][c]	40nm CMOS	256	13	106K / – / – / –	72	1289	17
			512	14			2826	32
			1024	14			6155	81
	[26][c]	40nm CMOS	256	13	– / – / – / –	300	160	0.5
			512	14	– / – / – / –		492	1.6
	[27][c]	UMC 65nm	256	13	14K / – / – / –	25	2056	82
			512	14			4616	184
			1024	14			10248	409
	[28][a,b]	Artix-7	1024	14	4823 / 2901 / 8 / –	153	1280	–
	[29][b]	Virtex-7	16384	32	2.81K / 1.25K / 39 / 80	168	28672	–
			32768	32	2.86K / 1.27K / 39 / 160	166	61440	–
	[30][b]	Virtex-6	65536	30	72K / 63K / 250 / 84	100	47795	–
	TW-1 PE	Virtex-7	1024	14	575 / – / 3 / 11	125	5160	41.2
			4096	60	2720 / – / 31 / 180		24708	197.6
	TW-8 PE	Virtex-7	1024	14	2584 / – / 24 / 16	125	680	5.4
			4096	60	23215 / – / 248 / 176		3276	26.2
	TW-32 PE	Virtex-7	1024	14	17188 / – / 96 / 48	125	200	1.6
			4096	60	99384 / – / 992 / 176		972	7.7

TW:This Work. [a]:Uses fixed q. [b]:Uses fixed n. [c]:Works with multiple n and q.
(1): `PicoRV32-RV32I`. (2): `PicoRV32-RV32IM`. (3): `BOOM-RV64GC`.

7 Conclusions and Future Work

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-based cryptosystems mature and gear towards massive deployment, there will be a heavier emphasis on flexible design methods for faster adoption and design space exploration, which enforce scalable and configurable solutions. This paper conducts an extensive study of flexible design methods for NTT, proposes a flexible yet efficient hardware generator, and compares its efficiency against HLS- and processor-based design approaches. The results show the superiority of hand-tuned, parameterized hardware designs over other techniques (which is expected) and the inefficiencies of HLS tools over conventional, processor-based methods (which is unexpected). Therefore, this work calls for better HLS tools that can close the order(s)-of-magnitude gap compared to RTL-based designs or at least provide significantly better solutions compared to push-button, processor-based approaches. **Although implementation attacks/defenses such as the fault and side-channel analysis on lattice-based cryptography [47], [48], [49], [50] are out of our scope, future extensions of this work can cover those aspects.**

8 Acknowledgements

We thank anonymous reviewers for their thoughtful comments. Dr. Öztürk and Dr. Savaş are supported by TUBITAK under Grant Number 118E725. Dr. Aysu is supported in part by the National Science Foundation under Grant No. 1850373. We acknowledge Xilinx for donating an FPGA.

References

[1] K. Keutzer, A. R. Newton, J. M. Rabaey, and A. Sangiovanni-Vincentelli, "System-level design: orthogonalization of concerns

and platform-based design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, no. 12, pp. 1523–1543, Dec 2000.

[2] I. Verbauwhede and P. Schaumont, "Skiing the embedded systems mountain," *ACM Trans. Embed. Comput. Syst.*, vol. 4, no. 3, p. 529–548, Aug. 2005. [Online]. Available: https://doi.org/10.1145/1086519.1086523

[3] P. Milder, F. Franchetti, J. C. Hoe, and M. Püschel, "Computer generation of hardware for linear digital signal processing transforms," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 17, no. 2, Apr. 2012. [Online]. Available: https://doi.org/10.1145/2159542.2159547

[4] 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

[5] 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.

[6] E. Alkim, L. Ducas, T. Pöppelmann, and P. Schwabe, "Post-quantum key exchange—a new hope," in *25th USENIX*, 2016, pp. 327–343.

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

[8] T. Pöppelmann and T. Güneysu, "Area optimization of lightweight lattice-based encryption on reconfigurable hardware," in *2014 IEEE Int. Symp. on Circuits and Systems*, June 2014, pp. 2796–2799.

[9] B. Wang, "IBM predicts Lattice Cryptography will be big within 5 years to stop hackers," https://www.nextbigfuture.com/2018/03/ibm-predicts-lattice-cryptography-will-be-big-within-5-years-to-stop-hackers.html, 2018, accessed: 2010-09-30.

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

[11] 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.

[12] E. Ozcan and A. Aysu, "High-level-synthesis of number-theoretic transform: A case study for future cryptosystems," *IEEE Embedded Systems Letters*, pp. 1–1, 2019.

[13] 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.

[14] K. Millar, "Design of a flexible schoenhage-strassen fft polynomial multiplier with high-level synthesis," 2019.

[15] L. Botros, M. J. Kannwischer, and P. Schwabe, "Memory-efficient high-speed implementation of kyber on cortex-m4," in *Int. Conference on Cryptology in Africa*. Springer, 2019, pp. 209–228.

[16] G. Seiler, "Faster avx2 optimized ntt multiplication for ring-lwe lattice cryptography." *IACR Cryptology ePrint Archive*, vol. 2018, p. 39, 2018.

[17] E. Alkim, P. Jakubeit, and P. Schwabe, "Newhope on arm cortex-m," in *International Conference on Security, Privacy, and Applied Cryptography Engineering*. Springer, 2016, pp. 332–349.

[18] A. C. Mert, E. Öztürk, and E. Savaş, "Design and implementation of encryption/decryption architectures for bfv homomorphic encryption scheme," *IEEE Tran. on VLSI Systems*, pp. 1–10, 2019.

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

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

[21] T. Pöppelmann and T. Güneysu, "Towards practical lattice-based public-key encryption on reconfigurable hardware," in *Int. Conf. on Selected Areas in Cryptography*. Springer, 2013, pp. 68–85.

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

[23] 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.

[24] 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.

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

[26] 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*, April 2018, pp. 1–4.

[27] T. Fritzmann and J. Sepúlveda, "Efficient and flexible low-power ntt for lattice-based cryptography," in *2019 IEEE International Symposium on HOST*, May 2019, pp. 141–150.

[28] Y. Xing and S. Li, "An efficient implementation of the newhope key exchange on fpgas," *IEEE TCAS I: Regular Papers*, pp. 1–13, 2019.

[29] N. Zhang, Q. Qin, H. Yuan, C. Zhou, S. Yin, S. Wei, and L. Liu, "Nttu: an area-efficient low-power ntt-uncoupled architecture for ntt-based multiplication," *IEEE Tran. on Computers*, pp. 1–1, 2019.

[30] S. Sinha Roy, K. Järvinen, J. Vliegen, F. Vercauteren, and I. Verbauwhede, "Hepcloud: An fpga-based multicore processor for fv somewhat homomorphic function evaluation," *IEEE Transactions on Computers*, vol. 67, no. 11, pp. 1637–1650, Nov 2018.

[31] J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé, "Crystals-kyber: a cca-secure module-lattice-based kem," in *2018 IEEE EuroS&P*. IEEE, 2018, pp. 353–367.

[32] L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, P. Schwabe, G. Seiler, and D. Stehlé, "Crystals-dilithium: A lattice-based digital signature scheme," *IACR Tran. on CHES*, pp. 238–268, 2018.

[33] P.-A. Fouque, J. Hoffstein, P. Kirchner, V. Lyubashevsky, T. Pornin, T. Prest, T. Ricosset, G. Seiler, W. Whyte, and Z. Zhang, "Falcon: Fast-fourier lattice-based compact signatures over ntru."

[34] S. Streit and F. De Santis, "Post-quantum key exchange on armv8-a: A new hope for neon made simple," *IEEE Transactions on Computers*, vol. 67, no. 11, pp. 1651–1662, Nov 2018.

[35] 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.

[36] 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.

[37] X. Feng, S. Li, and S. Xu, "Rlwe-oriented high-speed polynomial multiplier utilizing multi-lane stockham ntt algorithm," *IEEE Transactions on Circuits and Systems II: Express Briefs*, pp. 1–1, 2019.

[38] J. M. Pollard, "The fast fourier transform in a finite field," *Mathematics of computation*, vol. 25, no. 114, pp. 365–374, 1971.

[39] S. Halevi and V. Shoup, "Algorithms in helib," in *Advances in Cryptology – CRYPTO 2014*, Santa Barbara, CA, USA, Aug. 2014, pp. 554–571.

[40] C. Aguilar-Melchor, J. Barrier, S. Guelton, A. Guinet, M.-O. Killijian, and T. Lepoint, "Nfllib: Ntt-based fast lattice library," in *Topics in Cryptology - CT-RSA 2016*, San Francisco, CA, USA, pp. 341–356.

[41] W. Dai and B. Sunar, "cuhe: A homomorphic encryption accelerator library," in *Cryptography and Information Security in the Balkans*, Koper, Slovenia, Sep. 2016, pp. 169–186.

[42] V.-H. Xilinx, "Vivado design suite user guide-high-level synthesis," 2014.

[43] K. Asanović and D. A. Patterson, "Instruction sets should be free: The case for risc-v," *EECS Department, UC, Berkeley, Tech. Rep. UCB/EECS-2014-146*, 2014.

[44] C. Wolf, "Picorv32 - a size-optimized risc-v cpu," https://github.com/cliffordwolf/picorv32.

[45] J. Zhao, "Boom: Berkeley out-of-order machine," https://github.com/riscv-boom/riscv-boom.

[46] A. Waterman, Y. Lee, D. Patterson, and K. Asanovic, "The risc-v instruction set manual," *volume I: User-level ISA, version 2.0, EECS Department, UC, Berkeley, Tech. Rep. UCB/EECS-2014-54*, 2014.

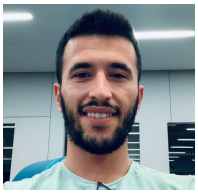
[47] O. Reparaz, S. S. Roy, R. de Clercq, F. Vercauteren, and I. Verbauwhede, "Masking ring-lwe," *Journal of Cryptographic Engineering*, vol. 6, no. 2, pp. 139–153, 2016.

[48] A. Aysu, Y. Tobah, M. Tiwari, A. Gerstlauer, and M. Orshansky, "Horizontal side-channel vulnerabilities of post-quantum key exchange protocols," in *2018 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2018, pp. 81–88.

[49] A. Aysu, M. Orshansky, and M. Tiwari, "Binary ring-lwe hardware with power side-channel countermeasures," in *2018 Design, Automation Test in Europe Conference Exhibition*, 2018, pp. 1253–1258.

[50] A. Sarker, M. Mozaffari-Kermani, and R. Azarderakhsh, "Hardware constructions for error detection of number-theoretic transform utilized in secure cryptographic architectures," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 3, pp. 738–741, 2019.

Ahmet Can Mert received the B.S. and M.S. degrees in Electronics Engineering Program from Sabanci University, Istanbul, Turkey, in 2015 and 2017, respectively. Currently, he is working towards the Ph.D. degree in Electronics Engineering Program at Sabanci University, Istanbul, Turkey. His research interest include homomorphic encryption, lattice-based cryptography and cryptographic hardware design.



Emre Karabulut received the B.S. degree in Control and Automation Engineering Department from Yıldız Technical University, Istanbul, Turkey, in 2016. After receiving his B.S. degree, he has worked in industry as a hardware engineer for three years. Currently, he is working towards his Ph.D. degree in Computer Engineering Department of North Carolina State University, Raleigh, NC, USA. His research interest includes post-quantum cryptography, hardware design, and computer architecture.

Erdinç Öztürk received the B.S. degree in microelectronics from Sabanci University in 2003. He received the M.S. degree in electrical engineering in 2005 and Ph.D. degree in electrical and computer engineering in 2009 from Worcester Polytechnic Institute. After receiving his PhD degree, he was with Intel in Massachusetts for almost 5 years as a hardware engineer, before joining Istanbul Commerce University as an assistant professor. He joined Sabanci University in 2017 as an assistant professor. His research interest include cryptographic hardware design and he focused on efficient identity based encryption implementations.

Erkay Savaş received the B.S. and M.S. degrees in electrical engineering from the Electronics and Communications Engineering Department, Istanbul Technical University in 1990 and 1994, respectively. He received the Ph.D. degree from the Department of Electrical and Computer Engineering, Oregon State University in June 2000. He has been a faculty member at Sabanci University since 2002. His research interests include applied cryptography, data and communication security, security and privacy in data mining applications, embedded systems security, and distributed systems.

Aydin Aysu is an assistant professor in the Electrical and Computer Engineering Department of North Carolina State University. Prior to that, he was a post-doctoral research fellow at the University of Texas at Austin from 2016 to 2018. He received his Ph.D. degree in Computer Engineering from Virginia Tech in 2016 and his M.S. degree in Electronics Engineering from Sabanci University in 2010. Dr. Aysu's research focuses on the development of secure systems that prevent cyberattacks targeting hardware vulnerabilities and lies at the intersection of applied cryptography, digital hardware design, and computer architectures. He received the 2018 NSF CRII award, 2019 NSF CAREER award, and is an IEEE senior member.