NTTGen: A Framework for Generating Low Latency NTT Implementations on FPGA

Yang Yang
yyang172@usc.edu
University of Southern California
Los Angeles, California, USA

Sanmukh R. Kuppannagari
kuppanna@usc.edu
University of Southern California
Los Angeles, California, USA

Rajgopal Kannan
rajgopal.kannan.civ@mail.mil
US Army Research Lab-West
Los Angeles, California, USA

Viktor K. Prasanna
prasanna@usc.edu
University of Southern California
Los Angeles, California, USA

ABSTRACT

Homomorphic encryption (HE) is a promising technique to ensure the security and privacy of applications in the cloud. Number Theoretic Transform (NTT) is a key operation in HE-based applications. HE requires vastly different NTT parameters to meet the performance and security requirements of applications. The increasing compute capabilities and flexibility of FPGAs make them attractive to accelerate NTT. However, programming FPGA still involves hardware design expertise and significant development effort. To close the gap, we propose NTTGen, a framework to automatically generate low latency NTT designs targeting HE-based applications. NTTGen takes application parameters, latency and hardware resource constraints as input, determines the design parameters, and produces synthesizable Verilog code as output. Low latency NTT implementations are obtained by varying the data, pipeline and batch parallelism. NTTGen utilizes streaming permutation network to reduce the interconnect complexity between stages in the NTT computation. The framework supports two types of NTT cores to perform modular arithmetic, the key computation in NTT: a low latency and resource efficient NTT core for a specific class of prime moduli and a general purpose NTT core for other primes. We further develop a design space exploration flow to identify the hardware design parameters of an optimal design. We evaluate NTTGen by generating designs for various NTT parameters. The designs result in up to 2.9× improvement in latency over the state-of-the-art FPGA implementations.

CCS CONCEPTS

• **Computer systems organization** → *Parallel architectures*; • **Security and privacy** → *Security in hardware.*

KEYWORDS

Number Theoretic Transform, Parallel Computing, FPGA Acceleration

CF'22, May 17–19, 2022, Torino, Italy

ACM ISBN 978-1-4503-9338-6/22/05.
https://doi.org/10.1145/3528416.3530225

ACM Reference Format:
Yang Yang, Sanmukh R. Kuppannagari, Rajgopal Kannan, and Viktor K. Prasanna. 2022. NTTGen: A Framework for Generating Low Latency NTT Implementations on FPGA. In *19th ACM International Conference on Computing Frontiers (CF'22), May 17–19, 2022, Torino, Italy.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3528416.3530225

1 INTRODUCTION

Homomorphic Encryption (HE) is a promising technique to address data privacy in cloud computing. It allows computation on encrypted data directly [8, 15]. However, computation on homomorphic encrypted data is orders of magnitude slower than computation on unencrypted data [34]. Number Theoretic Transform (NTT) is one of the most computationally expensive operations in HE [20, 34]. NTT is a specialized form of Discrete Fourier Transform in which the computations are performed in a finite integer field. It takes a degree N polynomial as input and uses the powers of the N_{th} root of unity modulo a prime as twiddle factors to perform modular arithmetic operations. The output of NTT is also a polynomial of degree N. N is a power-of-two and typically ranges from 2^{10} to 2^{17} [2]. For Residue Number System (RNS) variant of HE schemes, one NTT with large modulus is represented as np numbers of independent NTTs, one for each RNS basis [11]. Table 1 shows the NTT parameters used in previous works. These parameters can differ vastly depending on the use cases.

The fine grained parallel computing capabilities of FPGAs make them an attractive platform to accelerate NTT. Nevertheless, programming FPGA using hardware description language or high-level synthesis still requires extensive hardware knowledge and a long development cycle [13, 31]. Several designs are proposed to operate on fixed NTT parameters [24, 38, 42]. Porting these designs to different parameters involves time-consuming RTL programming. Therefore, it is desirable to have a framework to generate optimized designs for various NTT use cases without worrying about the FPGA implementation details.

It is non-trivial to develop a framework that can generate low latency designs for various NTT parameters. NTT algorithm requires complex data access patterns between computation stages. Purposely designed memory layout and fully connected crossbar are often used to perform the permutation [28, 35], which are not only expensive to implement in hardware but also difficult for automation in the generation of optimized designs. The modular

arithmetic operations in NTT are costly in both hardware resources and latency. Using prime moduli that satisfy certain properties can significantly reduce the hardware complexity in modular reduction [42]. In addition, different levels of parallelism can be exploited in NTT. Identifying the optimal parallelization strategy under application and resource constraints is crucial for low latency NTT designs. Existing frameworks [9, 28, 29, 35] not only do not consider all the aforementioned trade-offs but also require users to specify hardware design parameters manually.

Table 1: NTT Parameters of Previous Works

Works	HE scheme	N	np	prime size
[21, 34]	BFV	$2^{11} - 2^{16}$	1	60 bits
[6, 11, 35]	RNS-CKKS	$2^{11} - 2^{14}$	$2 - 15$	$30 - 60$ bits
[18, 22, 24]	Bootstrappable RNS-CKKS	$2^{14} - 2^{17}$	$21 - 42$	$45 - 62$ bits

Motivated by the challenges, we propose NTTGen, a hardware generation framework for low latency NTT designs targeting HE-based applications. The inputs to the framework are the application parameters (polynomial degree and prime moduli), latency and hardware resource constraints (bandwidth, available BRAM/DSP resources). The output of the framework is Verilog code of latency optimized NTT designs. NTTGen produces designs at various scales by exploiting parallelism tailored to NTT algorithm: data and pipeline parallelism – folding in the NTT parallel I/O dimension and computation stage dimension – are used to accelerate NTT for a given prime modulus, batch parallelism is employed for computing RNS-based NTTs. NTTGen utilizes Streaming Permutation Network (SPN) [10] to overcome the challenge due to the complex access patterns within NTT. SPN avoids the expensive crossbars and can be easily generalized to any permutation stride. For certain types of primes known as the generalized Mersenne primes [39], NTTGen uses a novel low latency NTT core to perform modular arithmetic. NTTGen selects between this low latency NTT core and a general purpose one based on the prime moduli. We further develop a performance model which identifies the optimized hardware design parameters. The key contributions of this paper are:

- We propose a hardware generation framework, NTTGen, to accelerate NTT computation on FPGA. The generated design is optimized for HE-based applications that avoid computationally expensive bootstrapping.
- Our framework utilizes SPN to avoid complex memory layout and expensive interconnect. SPN is extended to support data access patterns in various NTT stages at runtime.
- NTTGen supports both a general purpose and a low latency NTT core optimized for generalized Mersenne primes. It selects the appropriate one based on the prime moduli at design time.
- NTTGen achieves low latency by varying data, pipeline and batch parallelism. We develop a performance model which enables design space exploration to identify the design parameters that satisfy the user requirements.
- We demonstrate our framework by generating NTT designs with various NTT settings. The designs achieve up to 2.9× improvement in latency over the state-of-the-art FPGA implementations.

2 BACKGROUND

2.1 NTT in Homomorphic Encryption

NTT reduces the complexity of multiplying two polynomials of degree N from $O(N^2)$ to $O(N \log N)$ [1]. Algorithm 1 shows the NTT algorithm. Each iteration of the outermost loop is called a stage. There are $\log N$ stages in total, and each stage has $N/2$ independent instances of modular arithmetic (Line 9 – 12) that can be computed in parallel. The key computations in NTT include modular multiplication, addition and subtraction. Note that Inverse NTT (INTT) has almost identical computational patterns [26] and can be supported by the same hardware. In this paper, we focus on NTT, but the design can be directly applied to INTT. The NTT parameters are determined by the Homomorphic Encryption (HE) scheme and the security requirement. Typically N is in the range of 2^{10} to 2^{17} [2]. HE requires modulus Q to be up to hundreds of bits depending on the multiplicative depth of the function to be evaluated [2], which is expensive to process. The Residue Number System variant of HE schemes [3] enables representing a polynomial with $\log Q$-bit coefficients as np polynomials with narrower coefficients, where Q is the product of np co-primes $Q = \prod_{i=1}^{np} p_i$. An NTT over Q is equivalent to np independent NTTs over p_i.

Algorithm 1: Number Theoretic Transform

Input: Coefficients $A = (A[0], A[1], ..., A[N-1])$ and twiddle factors in bit-reversed order $\phi = (\phi[0], \phi[1], ..., \phi[N-1])$
Output: $A \leftarrow \text{NTT}(A)$ in bit-reversed order

```
1  t ← N
2  for (m ← 1; m ≤ N; m ← 2m) do
3      t ← t/2
4      for (i ← 0; i < m; i ← i + 1) do
5          j1 ← 2 · i · t
6          j2 ← j1 + t − 1
7          S ← φ[m + i]
8          for (j ← j1; j ≤ j2; j ← j + 1) do
9              U ← A[j]
10             V ← S · A[j + t]
11             A[j] ← U + V mod q
12             A[j + t] ← U − V mod q
13         end
14     end
15 end
```

The encryption noise is accumulated over each homomorphic operation [17]. If the noise exceeds a threshold, decryption becomes impossible [12, 16]. HE schemes of this type are called Leveled HE (LHE). Bootstrapping can reset the noise and enable Fully Homomorphic Encryption (FHE) computation. However, bootstrapping requires very large N and np, therefore it is less commonly used in practice [7, 14, 21, 37]. Although NTTGen can support any batch

size (np), and thus, can support FHE applications that use bootstrapping, we focus on optimizing NTT for HE-based applications that avoid bootstrapping. Our objective is to obtain low latency NTT implementation with "small" batch sizes, i.e. $np \leq 20$.

2.2 Comparison between NTT and FFT

The structure of the NTT algorithm is similar to the Fast Fourier Transform (FFT), but they have completely different basic computations. NTT performs modular arithmetic over a finite ring of integers, which is more complex than the floating point operations in FFT. Modular arithmetic can be costly due to the additional multiplication and/or division involved. Barrett and Montgomery reductions [4, 30] are two ways to mitigate this problem. In FFT, twiddle factors can be re-used when batching and executing multiple N-point FFTs together. In contrast, twiddle factors needed for NTT grow proportionally with the number of prime moduli (np). Therefore, existing methods in FFT cannot be applied to NTT directly.

3 RELATED WORK

Many recent works [24, 32, 33, 38, 41, 42] have been proposed on FPGA acceleration of NTT for *fixed* parameters. Kim et al. [24] focused on NTT in RNS-CKKS with bootstrapping [18], optimizations in their design are not directly applicable for other schemes that require a smaller polynomial degree or number of prime moduli. Customized memory layout was often used to enable parallel and conflict-free memory access between NTT stages [24, 28, 29, 33, 35, 38]. However, these designs require costly all-to-all connections between the NTT cores and the intermediate data buffer. A systolic array approach for NTT acceleration was presented in [32]. Data parallelism in the NTT algorithm is not explored and computation inside each NTT stage is serialized. Tian et al. [41] used SPN to accelerate NTT on FPGA. By fully unrolling all the NTT stages in hardware and using dedicated SPN in each stage, the SPN in their design only supports a fixed access pattern. In addition, the NTT core used in [41] is restricted to one special prime number, which limits its use in HE-based applications. In [9, 28, 29, 35], the authors proposed parameterized NTT implementations on FPGA. Parallelization across NTT stages and batch dimension is not explored. Customizing the NTT cores for different use cases is not considered. In addition, these designs require users to specify the hardware design parameters manually, which may lead to suboptimal designs for those who are not familiar with FPGA. Several works have developed fast NTT implementations using GPUs and ASICs. Lee et al. exploited an efficient algorithm strategy for GPU to avoid warp divergence and improve parallelism [25]. The authors in [22] proposed on-the-fly root generation to reduce GPU memory bandwidth for RNS-CKKS with bootstrapping. F1 [36] vectorized NTT using two-dimensional decomposition. But this approach requires extra matrix transposition steps.

Modular Multiplication is the key operation in NTT. The modular reduction algorithm proposed in [27] allows the output to be slightly greater than the modulus. This optimization avoids division operations, but requires additional long latency multiplications. The modular multiplication in HEAX requires pre-computations depending on the twiddle factors, which consumes more on-chip memory [35]. Authors in [42] designed an architecture for a specific prime that avoids any multiplication in the modular reduction. Kim et al. [23] used primes with low hamming weight to reduce the latency in Barrett reduction [4].

4 FRAMEWORK OVERVIEW

4.1 NTTGen Workflow

NTTGen abstracts away the hardware design details and only exposes high level parameters to users. Figure 1 illustrates the workflow of our design automation framework. NTTGen takes the application parameters and resource constraints as inputs. Application parameters are specified by:

- Latency ($\mathcal{L}_{max}$): The desired latency upper bound in μs to perform one batch of NTT operations.
- Polynomial degree (N): Polynomial degree for number theoretic transform. Our framework supports any polynomial degree at design time.
- List of prime moduli (np): The size of each prime modulus determines the data width of the NTT hardware. The length of the list is the batch size.

Hardware resource constraints are specified by:

- DSP ($\mathcal{D}_{max}$) and BRAM ($\mathcal{B}_{max}$) constraints: DSP and BRAM resources that can be used by the NTT hardware.
- I/O bandwidth ($\mathcal{BW}_{max}$): Available bandwidth of the target NTT design. The bandwidth is used to stream in and out the polynomials.
- Metadata: Platform related metadata such as FPGA platform, bit width of DSP and memory size per BRAM.

Using these inputs, the design space exploration in NTTGen (Section 6) determines the design parameters (Section 4.2) that can meet the constraints. NTTGen then generates Verilog code of the optimized NTT designs based on the hardware templates.

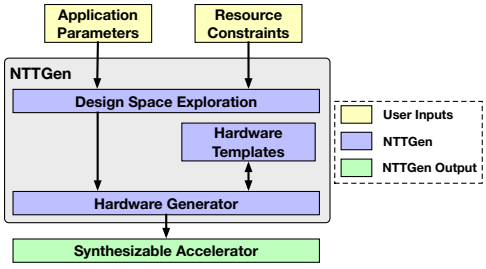

Figure 1: NTTGen workflow.

4.2 Design Parameters

NTT algorithmic and architecture settings are exposed as parameters in the hardware templates, allowing the same architecture to support various NTT parameters. Input and output coefficients are folded and processed in a streaming manner to satisfy I/O bandwidth constraint and to reduce resource consumption. The following parameter values are generated by the design space exploration at design time to customize our architecture:

- Data Parallelism (dp): This parameter determines the number of coefficients being processed per cycle in every NTT stage ($2 \leq dp \leq N$). Larger dp reduces latency but requires more

I/O bandwidth and hardware resources. dp is restricted to be a power-of-two.
- Pipeline Parallelism (pp): This parameter determines the unrolled NTT stages in the hardware ($1 \leq pp \leq \log N$). Higher unrolling increases the pipeline depth and reduces latency (Section 5), but consumes more resources.
- Batch Parallelism (bp): For a given batch size (np), this parameter determines how many NTTs in the batch are executed concurrently. In order to achieve low latency, we fix this parameter to np in the design space exploration.

5 HARDWARE ARCHITECTURE

5.1 Architecture Overview

The diverse NTT parameters and various constraints in HE-based applications require a scalable architecture. NTTGen is the first to simultaneously exploit three levels of parallelism, bp, dp and pp, in one architecture. Figure 2 depicts the proposed architecture. A design consists of bp instances, each instance is fully pipelined and performs NTT for one prime modulus. There are pp number of *Super Pipelines* per instance connected directly *without* any intermediate buffers. A degree N polynomial is streamed through the Super Pipelines of each instance at the rate of dp coefficients per cycle. Inside each Super Pipeline, $dp/2$ numbers of NTT Cores read the twiddle factors and perform the innermost loop computation (Line 9 – 12 in Algorithm 1). We tackle the challenge of the convoluted access patterns in NTT algorithm using the Streaming Permutation Network (SPN). SPN permutes coefficients based on the stride of a given NTT stage. Muxes are used to route coefficients from the last Super Pipeline back to the first Super Pipeline. This is required when a Super Pipeline is reused by more than one NTT stage ($pp < \log N$). With this architecture, NTTGen can generate a wide spectrum of NTT implementations ranging from resource efficient designs to low latency designs.

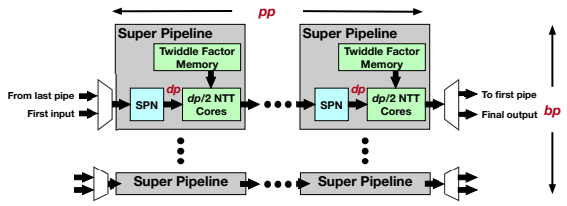

Figure 2: Hardware architecture of NTTGen.

5.2 NTT core

Each NTT core receives two coefficients as inputs and generates two coefficients as outputs. The key operation of NTT core is the modular multiplication. It performs modulo q after multiplying one of the coefficients with the twiddle factor, where q is one of the np prime moduli. Efficient modular multiplication is critical in order to achieve low latency in NTT implementations. Leveraging certain properties of special primes can greatly simplify the reduction logic and lower the latency, but it limits the prime moduli to choose from. Depending on the HE scheme and use case, there are additional restrictions in selecting the prime moduli. For instance, it is common to choose modulus with similar value in the RNS-CKKS scheme [11].

NTTGen provides the flexibility to implement two types of NTT cores. The general purpose NTT core can process any prime number while the low latency NTT core is optimized to accelerate generalized Mersenne primes [39]. If all the np prime moduli are generalized Mersenne primes, the low latency NTT core is used, otherwise NTTGen falls back to using the general purpose NTT core.

5.2.1 General purpose NTT core. The general purpose NTT core first computes the modular multiplication between one of the input coefficients and the twiddle factor. Then it performs modular addition and subtraction between the multiplication result and the other input coefficient to produce the two output coefficients. We use Barrett reduction algorithm [19] to implement modular multiplication. This algorithm includes three integer multiplications. The first multiplication is a full width multiplication and multiplies two input operands. The other two are additional half width multiplications for Barrett reduction. Similar to [23], our design is fully pipelined and can process two coefficients every cycle. The downside of this NTT core is that the additional half width multiplications add extra latency. For example, to perform 28-bit modular multiplication, it needs 12 cycles with 12 27×18 DSPs. 4 cycles are spent on the full width multiplication and 8 cycles are spent on the two half width multiplications. After the modular multiplication, the modular addition and subtraction can be done in parallel.

5.2.2 Low latency NTT core. The low latency NTT core is designed to optimize generalized Mersenne primes in the form of $2^i \pm 2^j \pm 1$, where i and j are positive integers. Modular reduction of such primes, i.e., division in mod q operation, can be implemented using a fixed number of less expensive modular operations (additions and subtractions) [39]. The input operands of these operations are extracted by selecting and shifting a range of bits from the result of the full width multiplication. Each generalized Mersenne prime has an associated modular reduction weight matrix, which specifies the bit-range for all the operands and the corresponding modular operation. Details on calculating the weight matrix can be found in [39]. Figure 3 shows the number of Mersenne primes for various operation counts and prime sizes. There is a limit in terms of how many primes can be used, but this does not make the NTT hardware less applicable for HE. For instance, 6 20-bit primes can be reduced using 4 operands (3 modular operations). More primes are available if allowing more modular operations.

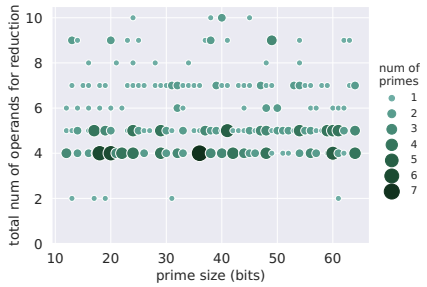

Figure 3: Distribution of generalized Mersenne primes with various reduction operands and bit-width.

Using this property, we design a resource efficient and low latency NTT core, as shown in Figure 4(b). The difference between

this NTT core and the general purpose NTT core lies in the modular multiplication module. After the full width multiplication, a modular reduction tree is used to replace the two half width multipliers. This reduces DSP resource usage and latency. The reduction tree is implemented using a binary tree of two input adders in hardware, where each node indicates a modular add/sub operation and each edge represents an operand. The result of the modular reduction tree is sent to the Mod Add and Mod Sub modules to generate the output coefficients. For these special primes, NTTGen computes the weight matrix offline, identifies each modular operation and uses that to determine the input and depth of the reduction tree. The bit selection information is programmed into the reduction configuration table. To perform the same 28-bit modular multiplication, if the number of modular operations required is 4 (reduction tree depth is 2), the design only takes 6 cycles with 4 27×18 DSPs.

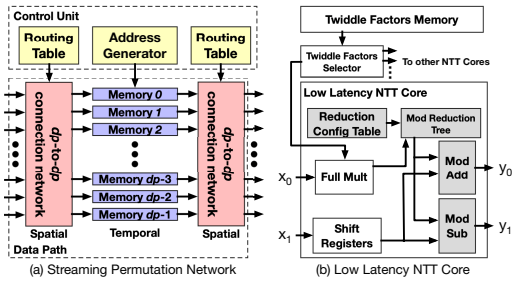

Figure 4: Architecture of (a) streaming permutation network, (b) special-purpose NTT core.

5.3 Twiddle Factor Memory

The Twiddle Factor Memory module stores the twiddle factors which are used by the NTT cores in each Super Pipeline. At stage i, only 2^i unique values of twiddle factors are used in Algorithm 1. Since there are $dp/2$ NTT cores, $dp/2$ unique twiddle factors are accessed per cycle in the worst case. Therefore we pack $dp/2$ twiddle factors together and store them in BRAMs such that they can be accessed in parallel. Figure 5 shows the twiddle factors memory layout for $pp = 1$ designs, i.e., the Twiddle Factor Memory stores all the factors. One row is read out each time and stored in the row buffer. The Selector module then picks the right twiddle factor for each NTT core.

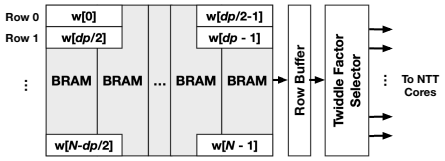

Figure 5: Twiddle factors memory layout with $pp = 1$.

5.4 Streaming Permutation Network (SPN)

Accessing all the coefficients every cycle requires a significant amount of bandwidth. For instance, to supply coefficients to an NTT module with $N = 1024$, $np = 1$ and 64-bit per coefficient, the hardware has to support at least 819 GB/s bandwidth even at 100 MHz frequency. Increasing N and np further increases the bandwidth. To mitigate this problem, coefficients are folded and sent over N/dp cycles. But folding complicates the irregular strided access pattern of NTT even more. Folding requires coefficients to be permuted not only within the same cycle (spatial permutation) but also across different cycles (temporal permutation). Existing designs use complex memory layout and expensive crossbar to ensure conflict free parallel access [28, 33, 35]. Generating the memory access patterns with various dp and pp is both costly and non-trivial. NTTGen uses SPN [10] as the basis to perform data permutation. In order to meet the requirements of the proposed architecture, we improve the functionality of SPN by supporting access patterns in multiple NTT stages at runtime. SPN uses a folded version of Benes multi-stage routing network [5], which reduces the interconnect cost and memory layout complexity. For completeness, we include a brief overview of the SPN in this paper. More details can be found in [10].

The SPN can achieve arbitrary permutation [10]. It has three subnetworks – two spatial permutation networks and one temporal permutation network, as shown in Figure 4(a). Spatial permutation shuffles the coefficients that are received in the same cycle whereas temporal permutation rearranges the coefficients across different cycles. A spatial permutation network, as illustrated in Figure 6(a), uses 2×2 switches to recursively compose a dp-to-dp connection. Temporal permutation is achieved by issuing reads and writes to dp dual-port memory using independently pre-computed addresses. Figure 6(b) shows an example of stride 4 permutation by the temporal network. Each column is written in parallel. After 4 cycles, coefficients with stride 4 are read out from the buffers concurrently. The control signals come from the Routing Tables and Address Generator. Routing Tables are used to program the switches. Address Generator sends out read and write addresses for temporal permutation. The overall execution of SPN is as follows: N coefficients stream through the first spatial permutation network in N/dp cycles, then the data are written into the dp memory blocks. After a fixed delay, dp coefficients are read out every cycle and permuted again by the second spatial permutation network.

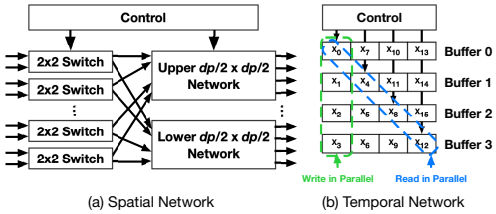

Figure 6: Architecture of SPN. (a) dp-to-dp spatial network. (b) Temporal network that performs stride 4 permutation.

Extension to SPN: The original SPN only provided support for a fixed permutation pattern at runtime. But the SPN in one Super Pipeline could be handling the permutation of multiple NTT stages when $pp < \log N$. We extend the SPN to support this by offline pre-computing the Routing Tables and Address Generators for all the NTT stages that one SPN needs to process. At runtime, a state machine is added in the Control Unit. The state machine tracks the progress of the NTT and selects the corresponding routing tables and address generator to generate the required control signals.

Compared to a naive crossbar interconnect, which requires $O(dp^2)$ connections, each spatial permutation network has $(dp/2) \cdot \log dp$

2×2 switches. The downside of SPN is an increased latency in permutation. By trading off latency with hardware resources, the SPN asymptotically has lower complexity. The design space exploration in NTTGen finds the trade-off point such that the generated SPN can meet the latency and resource constraints.

6 DESIGN SPACE EXPLORATION

6.1 Performance Modeling

Latency ($\mathcal{L}$): The overall latency is the sum of execution time for all the NTT stages, which consists of SPN latency and NTT core latency. Each spatial permutation network in the SPN has $\log dp$ cycles delay and the latency of the temporal network can be calculated based on the permutation stride. General purpose NTT core has a fixed latency. The prime moduli have an impact on the latency of the low latency NTT core as they determine the depth of the reduction tree.

DSP resources ($\mathcal{D}$): DSP consumption depends on the NTT core type. For the low latency NTT core, only one full width multiplier is needed (Section 5.2.2). In contrast, the general purpose NTT core requires two additional half width multipliers (Section 5.2.1). The number of DSPs in a design is

$$\mathcal{D} = bp \cdot pp \cdot dp/2 \cdot d_{core} \tag{1}$$

The factor d_{core} refers to the number of DSPs in one NTT core. There are $dp/2$ NTT cores per Super Pipeline.

BRAM resources ($\mathcal{B}$): Each SPN has dp buffers. Each row of the buffer stores one coefficient and there are N/dp rows. Each SPN buffer requires multiple BRAMs to be chained horizontally and/or vertically. Horizontal chaining is needed when bit-width per coefficient exceeds the bits per row of a BRAM (*bits_bram_row*). Vertical chaining is used when N/dp is greater than the number of rows per BRAM (*bram_rows*). The BRAM consumption per SPN ($\mathcal{B}_{spn}$) is

$$\mathcal{B}_{spn} = \lceil \frac{bits\_coefficient}{bits\_bram\_row} \rceil \cdot \lceil \frac{N}{dp \cdot bram\_rows} \rceil \cdot dp \tag{2}$$

The first term calculates the BRAM instances for each buffer to supply *bits_coefficient* bits per cycle. The second term estimates the BRAM instances given the depth of each buffer. *bram_rows* and *bits_bram_row* are the BRAM aspect ratio. NTTGen uses two BRAM configurations: $1K$ rows by 36 bits per row or $2K$ rows by 18 bits per row. Each Super Pipeline also uses BRAMs to store the twiddle factors in one or more NTT stages. Stage i has 2^i unique twiddle factors. The twiddle factors are laid out in memory in a 2-D fashion where the number of factors per row is determined by dp (Section 5.3). The BRAM resources for the twiddle factors ($\mathcal{B}_{tf}$) per Super Pipeline is modeled similarly to Equation 2. The total number of BRAMs in a design is

$$\mathcal{B} = bp \cdot pp \cdot (\mathcal{B}_{spn} + \mathcal{B}_{tf}) \tag{3}$$

I/O bandwidth ($\mathcal{BW}$): In $\mathcal{L}$ period of time, bp polynomials of degree N are loaded from and stored to the external memory respectively. The required I/O bandwidth is

$$\mathcal{BW} = 2 \cdot \frac{bp \cdot N \cdot bytes\_coefficient}{\mathcal{L}} \tag{4}$$

6.2 Design Space Exploration

The design space exploration (DSE) uses the performance model (Section 6.1) to evaluate design candidates. DSE assumes that the generated NTT designs operate at a fixed frequency. Having a high frequency (f_{max}) assumption in DSE produces more candidate designs for synthesis, place and route. But some designs may exceed the latency upper bound if the f_{max} after implementation is lower than the assumed frequency. On the contrary, having a low frequency assumption may miss valid designs. By default, we empirically use 250 MHz as the target frequency to include more potentially valid design candidates.

The DSE determines the type of the NTT core based on the list of prime moduli. To generate design candidates, DSE enumerates possible values of dp and pp. dp is iterated in power-of-two steps ($2 \leq dp \leq 256$). We limit the maximum dp to 256 as we observed significant frequency reduction when $dp > 256$. pp is iterated from 1 to $\log N$. DSE computes $\mathcal{L}$, $\mathcal{D}$, $\mathcal{B}$ and $\mathcal{BW}$ for each design candidate. If a design candidate satisfies all the constraints against $\mathcal{L}_{max}$, $\mathcal{D}_{max}$, $\mathcal{B}_{max}$ and $\mathcal{BW}_{max}$, it is added to the list of valid designs for synthesis and place-and-route.

7 EXPERIMENTS AND RESULTS

7.1 Experimental Setup

We use NTTGen to generate a wide range of NTT designs based on HE use cases (Table 1). We select polynomial degrees (N) from 2^{10} to 2^{14}, number of primes (np) between 1 and 21 and prime size from 28-bit to 52-bit. We assume that the input and output polynomials are stored in the DRAM. We run post place-and-route simulations to report latency, our main performance metric. The latency is defined as the duration from loading the input polynomials from DRAM to storing the results to DRAM. We compare the measured latency with the DSE projected latency to validate the effectiveness of NTTGen. We further analyze the impact of various application and design parameters on latency and hardware resources.

Our FPGA designs are synthesized and place-and-routed using Xilinx Vivado 2020.2. To illustrate the adaptability of NTTGen, the experiments are conducted on 3 FPGAs: XCU200, Virtex-7 XC7VX690 and XCU280 [40]. We use XCU200 as the primary platform for NTTGen evaluation and the other two FPGAs for comparison with the state-of-the-art implementations. Table 2 summarizes the breakdown of resources of each FPGA.

Table 2: Summary of FPGA Resources

FPGA	LUT	FF	BRAM	DSP	External Bandwidth
XCU200	1182K	2364K	2160	6840	78 GB/s
XC7VX690	433K	866K	1470	3600	78 GB/s
XCU280	1304K	2607K	2016	8490	460 GB/s

7.2 Framework Evaluation

To demonstrate NTTGen's effectiveness to generate designs at various scales, we perform design space exploration by placing constraints on DSP and BRAM resources. We measure the latency of all the design candidates produced by NTTGen given a latency upper bound. We discuss the results for $np = 4$ and 30-bit prime

moduli in this section. Other batch sizes and prime bit widths show similar results. Table 3 shows the 30-bit prime moduli used by the low latency NTT core. We vary the constraint on the amount of DSP and BRAM resources the hardware can use, from 20% to 100% of the total resources on XCU200. Figure 7(a) and Figure 7(b) show the designs using the low latency NTT core with $N = 4096$ and $N = 8192$ respectively. The star (★) represents the latency target and the dots (●) indicate the latency of the design candidates. Figure 7(c) and Figure 7(d) show the same configuration using the general purpose NTT core. The latency of the generated designs matches closely with the projections from design space exploration (DSE). Since very few designs run at higher than 250 MHz after place-and-route, setting a 250 MHz frequency in the DSE does not miss many valid designs. Designs with large dp and/or pp tend to achieve lower frequency, as a result they may exceed the latency upper bound. The general purpose NTT core consumes more DSP resources and has longer latency, therefore there are fewer valid designs in Figure 7(c) and Figure 7(d).

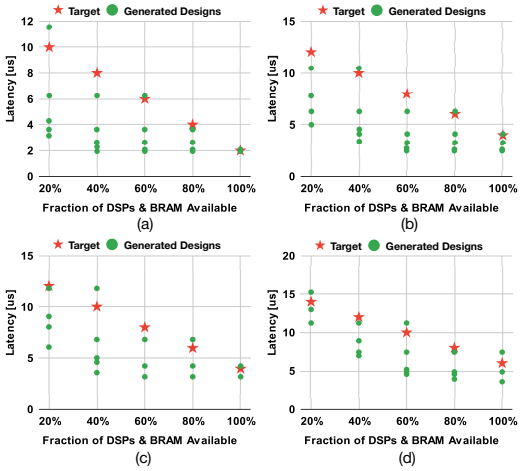

Figure 7: Achieved latency of the generated designs. (a) $N = 4096$, low latency NTT core. (b) $N = 8192$, low latency NTT core. (c) $N = 4096$, general purpose NTT core. (d) $N = 8192$, general purpose NTT core.

Table 3: Moduli Used by the Low Latency NTT Core

30-bit Prime	Reduction Tree Depth	52-bit Prime	Reduction Tree Depth
$2^{30} - 2^{9} - 1$	2	$2^{52} - 2^{12} + 1$	3
$2^{30} - 2^{16} - 1$	3	$2^{52} - 2^{19} - 1$	2
$2^{30} - 2^{18} - 1$	3	$2^{52} - 2^{20} + 1$	3
$2^{30} - 2^{18} + 1$	3	$2^{52} - 2^{47} - 1$	4

7.3 Sensitivity Analysis on Design Parameters

We evaluate the impact of latency by varying N, dp and pp. We use $np = 4$ in this set of experiments. In Figure 8(a), we show various designs with $pp = 2$ and vary dp and N to measure latency. We observe close to linear latency reduction as dp increases for a given N. The effect of pp is shown in Figure 8(b). dp is fixed at 16 in this case. Note that pp can reduce latency only when the permutation and computation latency of all the Super Pipelines is less than N/dp (Section 5.1). After that it has no impact on latency. Therefore its impact on latency is less effective than dp.

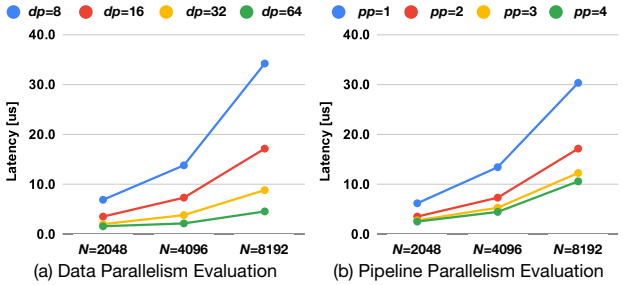

Figure 8: dp and pp performance evaluation using special-purpose NTT core.

In Figure 9, we show the I/O bandwidth needed for each design reported in Figure 8. We assume that the twiddle factors are pre-loaded to the on-chip SRAM and only the input and output polynomials incur DRAM traffic. This is the combined bandwidth of the entire batch ($np = 4$). The DRAM bandwidth demand goes up as dp and pp are increased. This is simply because the latency is reduced while the total data to be transferred does not change with the values of dp and pp. Bandwidth requirements for all the designs is well within the available bandwidth of the target FPGA.

7.4 Impact of SPN and NTT Cores

7.4.1 SPN. SPN is one of the key building blocks in our architecture. We evaluate the resource consumption of SPN using parameters $N = 4096$, $np = 2$, $pp = 1$ and 52 bits prime modulus. These parameters correspond to the design configuration in [35]. Three experiments are conducted by setting $dp = 16, 32, 64$. Table 5 shows the comparison results. The resource consumption in [35] is derived by deducting the resources used by the NTT cores from the total resources of the NTT module. As evident from the table, the logarithmic interconnect in SPN reduces the LUT and FF resource consumption significantly.

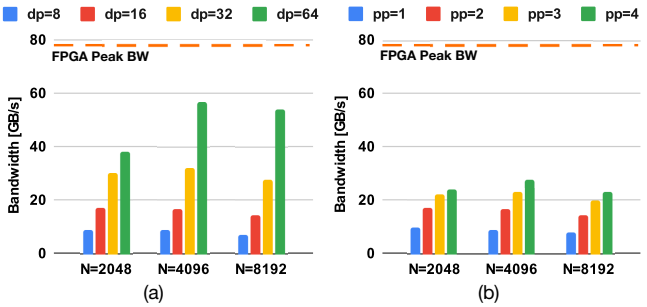

Figure 9: I/O bandwidth of various designs. (a) Required bandwidth when varying dp. (b) Required bandwidth when varying pp.

In Figure 10, we show the latency impact of SPN by varying dp. As discussed in Section 5.4, SPN trades-off interconnect complexity by increasing latency. The latency increases with the permutation stride. This is because earlier received coefficients need to be

Table 4: Comparison with State-of-the-art FPGA Works

Work	[41]	This paper $dp = 32$ $pp = 10$	[38]	This paper $dp = 4$ $pp = 2$	[35]	This paper $dp = 32$ $pp = 5$
Device	Virtex-7	Virtex-7	XCZU9	XCU200	Stratix10[a]	XCU200
N	1024	1024	4096	4096	8192	8192
np	1	1	6	6	4	4
moduli bit width	28	28	30	30	52	52
kLUT	95	206	64	54.1	569	1,107
kFF	104	159	25	56.2	1,549	1,002
DSP	640	640	200	288	1,280	1,920
BRAM	80	80	400	84	2,900	1,600
Freq [MHz]	215	210	200	250	300	220
Latency [μs]	0.9	1.1	73	24.7	6.0	4.5

[a]Stratix10 FPGAs use 20K bits per M20k (BRAM) and 27 × 27 per DSP. Xilinx FPGAs use 32K bits per BRAM and 27 × 18 DSP.

Table 5: Interconnect Resource Comparison

dp	Design	LUT	FF	BRAM
16	SPN	7,013	10,415	16
	[35]	19,808	46,543	16
32	SPN	16,836	23,505	32
	[35]	34,809	95,453	32
64	SPN	36,496	55,212	64
	[35]	76,188	185,853	64

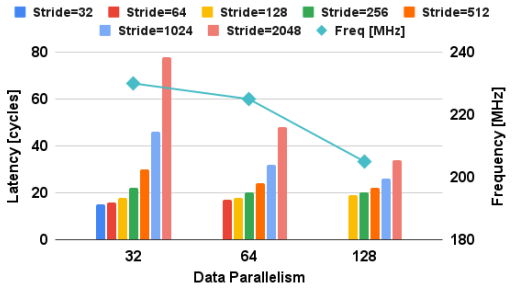

Figure 10: Latency of SPN in different permutation strides. Varying dp from 32 to 128. 52-bit per input and output coefficients are used.

buffered in the temporal network until all the coefficients with the proper stride arrive. Tuning the architecture parameters dp and pp can help bring down the latency. By using DSE, we can strike a balance between SPN latency overhead and resource consumption.

7.4.2 NTT Core. We evaluate the resource consumption and the latency of the two proposed NTT cores. The low latency NTT core uses the prime moduli in Table 3. We perform experiments on standalone NTT cores without twiddle factor memory. The results are shown in Table 6. Since the experiment is conducted on a standalone NTT core, the achieved frequency is higher than the integrated designs with SPN and multiple NTT cores. The low latency NTT core is more resource efficient with regard to DSPs but consumes slightly more LUTs due to the additional modular adders (Section 5.2.2). The latency of the low latency core is also lower due to less pipeline stages.

Table 6: Evaluation of NTT Cores

	30-bit Low Latency	30-bit General Purpose	52-bit Low Latency	52-bit General Purpose
LUT	1123	1081	2074	1837
FF	679	920	1237	1682
DSP	4	12	6	17
Latency	9 cycles	14 cycles	12 cycles	19 cycles
Frequency	400 MHz	400 MHz	400 MHz	400 MHz

7.5 Comparison with state-of-the-art

To the best of our knowledge, there are no frameworks for NTT acceleration on FPGA or GPU. We compare the designs generated by NTTGen with various baseline designs which are optimized for *fixed* parameters. We generate each design using a comparable FPGA device. Designs with similar NTT application parameters and resource consumption as the baselines are reported to make a fair comparison with regard to latency. It is not difficult to find enough generalized Mersenne primes for $np = 1$ and $np = 4$ use cases (Table 3), therefore we use the low latency NTT core in these cases. For the other cases, the general purpose NTT core is used.

Comparison with FPGA implementations: Table 4 summarizes the comparison results. Tian et al. [41] accelerate NTT on FPGA and develop a highly optimized "point design". It achieves better latency than NTTGen for a small set of NTT parameters. For the specific case shown in Table 4, it fully unrolls all the NTT stages in hardware, their approach is not generalizable to produce efficient implementations for various application parameters and hardware constraints. To reduce resource consumption, the NTT core in [41] is restricted to one specific prime number, which severely limits its applicability in HE. HEAX [35] is limited to $pp = 1$ and the specialization of NTT core is not explored. Having multiple Super Pipelines lowers the latency because separate NTT cores can be used for different NTT stages to avoid pipeline stall. Low latency NTT core further reduces the latency (Section 5.2.2). We observe 1.3× improvement in latency compared with HEAX. A low resource design with $np = 6$ is proposed in [38]. Our framework produces a design that achieves 2.9× improvement in latency.

While all the baselines are specifically designed for the given parameters, NTTGen is a general framework and does not optimize the hardware architecture for specific NTT parameters. The NTTGen *auto*-generated designs outperform the baselines by up to 2.9×.

Table 7: Comparison with State-of-the-art GPU Work

Work	[22]	This paper
Device	Titan V	XCU280
(N, np)	(16384, 21)	(16384, 21)
moduli bit width	60	52
Freq [MHz]	1200	180
External Bandwidth [GB/s]	653	460
Latency [μs]	44.1	39.6
Compute Efficiency	35.5%	67.3%
Norm. Number of NTTs Computed per Unit Bandwidth	1	1.58

Comparison with GPU implementation: Table 7 lists the comparison. Note that the implementation in [22] is only optimized for bootstrappable HE use cases while NTTGen is applicable for a variety of other application parameters (Table 1). Our FPGA design uses $dp = 32$ and $pp = 1$. We estimate the GPU compute efficiency (percentage to peak performance) based on Algorithm 4 in [22]. Each butterfly operation needs 20 32-bit integer operations. Thus the entire NTT algorithm needs 0.096 GOPs (in 44.1 μs) while the peak throughput of the GPU (int32) is 6.14 TOPs/s. Memory efficiency is defined as the number of NTTs computed per unit bandwidth. Although the GPU offers higher compute and external bandwidth, our design has lower latency and achieves much better compute and memory efficiency. This is owing to the customization of the on-chip memory for twiddle factors, utilization of a dedicated dataflow and elimination of global synchronizations that are required in the GPU baseline.

8 CONCLUSION

In this paper, we presented a framework for generating low latency NTT implementations on FPGA. NTTGen exploits data, pipeline and batch level parallelism to optimize latency subject to resource constraints. Unlike previous works, NTTGen uses the streaming permutation network to tackle the challenge of complex access pattern in NTT algorithm. We proposed a novel NTT core for a special class of prime moduli to reduce latency. A design space exploration tool was developed to automatically identify the optimal design parameters. The designs generated by NTTGen result in up to 2.9× improvement in latency over state-of-the-art.

ACKNOWLEDGMENTS

This work has been sponsored by the U.S. National Science Foundation under grant number SaTC-2104264. Equipment grant by Xilinx is greatly appreciated.

REFERENCES

[1] Alfred V. Aho and John E. Hopcroft. 1974. *The Design and Analysis of Computer Algorithms* (1st ed.). Addison-Wesley Longman Publishing Co., Inc., USA.
[2] Martin Albrecht, Melissa Chase, Hao Chen, and et al. 2018. *Homomorphic Encryption Security Standard.* Technical Report.
[3] Jean-Claude Bajard, Julien Eynard, M. Anwar Hasan, and Vincent Zucca. 2017. A Full RNS Variant of FV Like Somewhat Homomorphic Encryption Schemes. In *Selected Areas in Cryptography – SAC 2016*, Roberto Avanzi and Howard Heys (Eds.).
[4] Paul Barrett. 1987. Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor. In *Proceedings on Advances in Cryptology—CRYPTO '86* (Santa Barbara, California, USA). Springer-Verlag, Berlin, Heidelberg, 311–323.
[5] V. E. Beneš. 1964. Optimal rearrangeable multistage connecting networks. *The Bell System Technical Journal* 43, 4 (1964), 1641–1656. https://doi.org/10.1002/j.1538-7305.1964.tb04103.x
[6] Fabian Boemer and et al. 2019. NGraph-HE2: A High-Throughput Framework for Neural Network Inference on Encrypted Data. In *Proceedings of the 7th ACM Workshop on Encrypted Computing and Applied Homomorphic Cryptography* (London, United Kingdom) *(WAHC'19)*.
[7] Joppe W. Bos, Wouter Castryck, Ilia Iliashenko, and Frederik Vercauteren. 2017. Privacy-Friendly Forecasting for the Smart Grid Using Homomorphic Encryption and the Group Method of Data Handling. In *Progress in Cryptology - AFRICACRYPT 2017*, Marc Joye and Abderrahmane Nitaj (Eds.). Springer International Publishing, Cham, 184–201.
[8] Florian Bourse, Michele Minelli, Matthias Minihold, and Pascal Paillier. 2018. Fast Homomorphic Evaluation of Deep Discretized Neural Networks. In *Advances in Cryptology – CRYPTO 2018 (Lecture Notes in Computer Science, Vol. 10993)*. Springer, 483–512. https://doi.org/10.1007/978-3-319-96878-0_17
[9] Joël Cathébras, Alexandre Carbon, Peter Milder, Renaud Sirdey, and Nicolas Ventroux. 2018. Data Flow Oriented Hardware Design of RNS-based Polynomial Multiplication for SHE Acceleration. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (Aug. 2018), 69–88.
[10] R. Chen and V. K. Prasanna. 2015. Automatic generation of high throughput energy efficient streaming architectures for arbitrary fixed permutations. In *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*. 1–8.
[11] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. 2019. A Full RNS Variant of Approximate Homomorphic Encryption. In *Selected Areas in Cryptography – SAC 2018*, Carlos Cid and Michael J. Jacobson Jr. (Eds.). Springer International Publishing, Cham, 347–368.
[12] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2016. Homomorphic Encryption for Arithmetic of Approximate Numbers. Cryptology ePrint Archive, Report 2016/421. https://eprint.iacr.org/2016/421.
[13] Jason Cong, Peng Wei, Cody Hao Yu, and Peng Zhang. 2018. Automated Accelerator Generation and Optimization with Composable, Parallel and Pipeline Architecture *(DAC '18)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3195970.3195999
[14] Christoph Dobraunig and et al. 2018. Rasta: A Cipher with Low ANDdepth and Few ANDs per Bit. In *Advances in Cryptology – CRYPTO 2018*, Hovav Shacham and Alexandra Boldyreva (Eds.). Springer International Publishing, Cham.
[15] Nathan Dowlin, Ran Gilad-Bachrach, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. 2016. CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48* (New York, NY, USA) *(ICML'16)*. JMLR.org, 201–210.
[16] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat Practical Fully Homomorphic Encryption. Cryptology ePrint Archive, Report 2012/144. https://eprint.iacr.org/2012/144.
[17] Craig Gentry. 2009. A Fully Homomorphic Encryption Scheme. PhD Dissertation 2009.
[18] Kyoohyung Han and Dohyeong Ki. 2020. Better Bootstrapping for Approximate Homomorphic Encryption. In *Topics in Cryptology – CT-RSA 2020*, Stanislaw Jarecki (Ed.). Springer International Publishing.
[19] Darrel Hankerson, Alfred J. Menezes, and Scott Vanstone. 2003. *Guide to Elliptic Curve Cryptography*. Springer-Verlag, Berlin, Heidelberg.
[20] Wonkyung Jung, Eojin Lee, Sangpyo Kim, Keewoo Lee, Namhoon Kim, Chohong Min, Jung Hee Cheon, and Jung Ho Ahn. 2020. HEAAN Demystified: Accelerating Fully Homomorphic Encryption Through Architecture-centric Analysis and Optimization. arXiv:2003.04510 [cs.DC]
[21] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. 2018. GAZELLE: A Low Latency Framework for Secure Neural Network Inference. In *Proceedings of the 27th USENIX Conference on Security Symposium (SEC'18)*. USENIX Association, USA.
[22] S. Kim, W. Jung, J. Park, and J. Ahn. 2020. Accelerating Number Theoretic Transformations for Bootstrappable Homomorphic Encryption on GPUs. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*.
[23] S. Kim, K. Lee, W. Cho, J. H. Cheon, and R. A. Rutenbar. 2019. FPGA-based Accelerators of Fully Pipelined Modular Multipliers for Homomorphic Encryption. In *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*.
[24] S. Kim, K. Lee, W. Cho, Y. Nam, J. H. Cheon, and R. A. Rutenbar. 2020. Hardware Architecture of a Number Theoretic Transform for a Bootstrappable RNS-based

Homomorphic Encryption Scheme. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 56–64.

[25] Wai-Kong Lee, Sedat Akleylek, Wun-She Yap, and Bok-Min Goi. 2019. Accelerating Number Theoretic Transform in GPU Platform for qTESLA Scheme. In *Information Security Practice and Experience*.

[26] Patrick Longa and Michael Naehrig. 2016. Speeding up the Number Theoretic Transform for Faster Ideal Lattice-Based Cryptography. In *Cryptology and Network Security*, Sara Foresti and Giuseppe Persiano (Eds.). Springer International Publishing, Cham, 124–139.

[27] Patrick Longa and Michael Naehrig. 2016. Speeding up the Number Theoretic Transform for Faster Ideal Lattice-Based Cryptography. Cryptology ePrint Archive, Report 2016/504. https://eprint.iacr.org/2016/504.

[28] A. C. Mert, E. Karabulut, E. Öztürk, E. Savaş, M. Becchi, and A. Aysu. 2020. A Flexible and Scalable NTT Hardware : Applications from Homomorphically Encrypted Deep Learning to Post-Quantum Cryptography. In *2020 Design, Automation and Test in Europe Conference Exhibition*.

[29] Ahmet Can Mert, Erdinç Öztürk, and Erkay Savaş. 2020. FPGA implementation of a run-time configurable NTT-based polynomial multiplication hardware. *Microprocessors and Microsystems* 78 (2020), 103219.

[30] Peter L. Montgomery. 1985. Modular multiplication without trial division. *Math. Comp.* 44 (1985), 519–521.

[31] Razvan Nane and et al. 2016. A Survey and Evaluation of FPGA High-Level Synthesis Tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2016).

[32] H. Nejatollahi, S. Shahhosseini, R. Cammarota, and N. Dutt. 2020. Exploring Energy Efficient Quantum-resistant Signal Processing Using Array Processors. In *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*.

[33] Rogério Paludo and Leonel Sousa. 2021. Number Theoretic Transform Architecture suitable to Lattice-based Fully-Homomorphic Encryption. In *2021 IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*.

[34] Brandon Reagen, Wooseok Choi, Yeongil Ko, Vincent Lee, Gu-Yeon Wei, Hsien-Hsin S. Lee, and David Brooks. 2020. Cheetah: Optimizing and Accelerating Homomorphic Encryption for Private Inference. arXiv:2006.00505 [cs.CR]

[35] M. Sadegh Riazi, Kim Laine, Blake Pelton, and Wei Dai. 2020. HEAX: An Architecture for Computing on Encrypted Data. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 1295–1309.

[36] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Srinivas Devadas, Ronald Dreslinski, Christopher Peikert, and Daniel Sanchez. 2021. F1: A Fast and Programmable Accelerator for Fully Homomorphic Encryption. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21)*.

[37] SEAL 2020. Microsoft SEAL (release 3.6). https://github.com/Microsoft/SEAL. Microsoft Research, Redmond, WA.

[38] S. Sinha Roy, F. Turan, K. Jarvinen, F. Vercauteren, and I. Verbauwhede. 2019. FPGA-Based High-Performance Parallel Architecture for Homomorphic Computing on Encrypted Data. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[39] Jerome A. Solinas. 1999. *Generalized Mersenne Numbers*. Technical Report.

[40] Xilinx. [n.d.]. Xilinx UltraScale+ FPGAs. https://www.xilinx.com/products/boards-and-kits/alveo.html.

[41] Tian Ye, Yang Yang, Sanmukh R. Kuppannagari, Rajgopal Kannan, and Viktor K. Prasanna. 2021. FPGA Acceleration of Number Theoretic Transform. In *High Performance Computing*, Bradford L. Chamberlain, Ana-Lucia Varbanescu, Hatem Ltaief, and Piotr Luszczek (Eds.).

[42] N. Zhang, B. Yang, C. Chen, S. Yin, S. Wei, and L. Liu. 2020. Highly Efficient Architecture of NewHope-NIST on FPGA using Low-Complexity NTT/INTT. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2020 (Mar. 2020), 49–72.