Data Flow Oriented Hardware Design of RNS-based Polynomial Multiplication for SHE Acceleration

Joël Cathébras[1], Alexandre Carbon[1], Peter Milder[2], Renaud Sirdey[1] and Nicolas Ventroux[1]

[1] CEA, LIST, F-91191 Gif-sur-Yvette, France. firstname.lastname@cea.fr
[2] Stony Brook University, Stony Brook, NY 11794-2350, USA. peter.milder@stonybrook.edu

Abstract. This paper presents a hardware implementation of a Residue Polynomial Multiplier (RPM), designed to accelerate the full Residue Number System (RNS) variant of the Fan-Vercauteren scheme proposed by Bajard et al. [BEHZ16]. Our design speeds up polynomial multiplication via a Negative Wrapped Convolution (NWC) which locally computes the required RNS channel dependent twiddle factors. Compared to related works, this design is more versatile regarding the addressable parameter sets for the BFV scheme. This is mainly brought by our proposed twiddle factor generator that makes the design BRAM utilization independent of the RNS basis size, with a negligible communication bandwidth usage for non-payload data. Furthermore, the generalization of a DFT hardware generator is explored in order to generate RNS friendly NTT architectures. This approach helps us to validate our RPM design over parameter sets from the work of Halevi et al. [HPS18]. For the depth-20 setting, we achieve an estimated speed up for the residue polynomial multiplications greater than 76 during ciphertexts multiplication, and greater than 16 during relinearization. It thus results in a single-threaded Mult&Relin ciphertext operation in 109.4 ms (×3.19 faster than [HPS18]) with RPM counting for less than 15% of the new computation time. Our RPM design scales up with reasonable use of hardware resources and realistic bandwidth requirements. It can also be exploited for other RNS based implementations of RLWE cryptosystems.

Keywords: Homomorphic Encryption · Polynomial Multiplication · Residue Number System · Negative Wrapped Convolution · Hardware Implementation

1 Introduction

Since the first Fully Homomorphic Encryption (FHE) scheme presented by Gentry [G+09] in 2009, homomorphic cryptography has been an active research area. The interesting property of an homomorphic encryption scheme is its ability to perform computations over encrypted data without the necessity of decrypting them. Among other uses, it is viewed as a promising solution to guarantee data privacy in cloud computing services.

The initial work from Gentry, impractical due to complexity and exponential noise growth, has been followed by numerous advances [BV11, Bra12, FV12, GHS12] to reach real yet modest practical applications (e.g. [CNS+16]). Improvements have been made: in the definition of schemes to make them simpler, in noise expansion control during ciphertext operations, and in implementation approaches for practical performances.

At the time of writing, four generations of Somewhat/Fully Homomorphic Encryption (S/FHE) schemes can be identified. The first starts with Gentry's initial work and is articulated around bootstrapping-based noise management. The second results from noise

IACR Transactions on Cryptographic Hardware and Embedded Systems ISSN 2569-2925, Vol. 2018, No. 3, pp. 69–88
DOI:10.13154/tches.v2018.i3.69-88

management improvements known as key and modulus switching, allowing the definitions of Leveled-FHE schemes, further improved in scale-invariant L-FHE. A third generation began with the GSW scheme from Gentry et al. [GSW13] built upon Brakerski's LWE-based scheme [Bra12] and removing the need for relinearization in scale-invariant Leveled-FHE. It has been quickly followed by Khedr et al. [KGV16] presenting a ring variant of GSW named SHIELD. Finally, the fourth generation returns to bootstrapping procedure, making it faster as it is part of the schemes somehow [DM15, CGGI16]. This paper focuses on the FV scheme [FV12] and its full Residue Number System (RNS) variant brought by Bajard et al. [BEHZ16] and further improved by Halevi et al. [HPS18].

Due to significant performance overheads in the encrypted domain, hardware acceleration appears necessary to address practical applications for S/FHE. In RLWE [LPR10] based cryptosystems like FV, a common approach to perform polynomial multiplication is through the NTT-based negative wrapped convolution [PG12]. The implementations of hardware acceleration for RLWE scheme seem to preferably target FPGA [ÖDSS15, RJV+15, PNPM15, CRS17, MRL+18]. GPU acceleration is also explored [DDS14, KG18] but is mostly considered for NTRU-based schemes.

The NTT-based polynomial ring multiplication reduces the computational asymptotic complexity due to the degree n of the handled polynomial, but the complexity of coefficient arithmetic, due to large modulus q, is still a problem for parameter sets targeting important multiplicative depth evaluation capability. An interesting approach is the use of Residue Number System (RNS) representation to reduce the size of basic arithmetic and bring parallelism [ÖDSS15, RJV+15, CRS17]. One difficulty with the coupled approach of RNS representation and NTT-based polynomial multiplication is brought by the large amount of precomputed values. Indeed, each RNS channel has its own twiddle factors and weight-vectors to perform a Negative Wrapped Convolution (NWC). In related implementations, this issue is handled either by storing all the required values on the FPGA [CRS17], or by storing them on the host side, and sending them along with the polynomials [ÖDSS15]. In [RJV+15], the authors choose an in-between solution by storing in ROM only a subset of the twiddle factors and computing the others when needed.

Our contribution. Following the mainstream approach to improve homomorphic evaluations based on RNS and NWC, this work explores the feasibility of a pipelined Residue Polynomial Multiplier (RPM) in a single flow.

To design this RPM, we present a generalization of the DFT architectures generated by the SPIRAL hardware backend, presented in [MFHP12], in order to generate NTT architectures independent of a predefined finite field. The resulting streaming NTT design is finite-field independent by means of cyclic reprogramming of twiddle factors memories.

Another contribution is the design of a twiddle factor generator that makes our approach scalable over practical homomorphic encryption parameter sets. Indeed, with n being the degree of polynomial handled and k the size of the RNS basis, our local generation requires $O(n)$ memory resources compared to $O(kn)$ or $O(k \log n))$ with local storage approaches, and this with negligible bandwidth utilization compared to an external storage.

To the best of our knowledge, our design is competitive with related hardware acceleration works, but with a much more versatile scalability over practical parameters of the full RNS variant of FV.

Outline. Section 2 presents notions and notations used throughout this paper. Related works on hardware acceleration for FV like schemes are discussed in section 3 to present our motivations. Then, our residue polynomial multiplier design is detailed in section 4. A proof-of-concept implementation is presented section 5, followed by a projection of our approach on more practical homomorphic encryption parameter sets. Finally, the conclusion highlights the main teachings of this work, and draws some perspectives.

2 Notations and Basic Notions

2.1 Notations

In this paper we consider polynomial rings of the form $\mathbb{Z}[X]/(f(X))$, with $f(X)$ a monic irreducible polynomial of $\mathbb{Z}[X]$. In particular the polynomial rings where $f(X)$ is a cyclotomic polynomial of order m a power of two, that is $f(X) = \Phi_m(X) = X^n + 1$ and $n = m/2$. From now on, $R = \mathbb{Z}[X]/(X^n + 1)$ is the ring of polynomials with degree strictly inferior to n and integer coefficients.

For a prime $p_i \in \mathbb{Z}$, $\mathbb{Z}_{p_i}$ denotes the finite-field $(\mathbb{Z}/p_i\mathbb{Z}, +, *)$ of all congruence classes modulo p_i. In further discussions, we will be interested in the product ring $\mathbb{Z}_q \cong \prod_{1\leq i\leq k} \mathbb{Z}_{p_i}$. Considering the polynomial ring $R_q = \mathbb{Z}_q[X]/(X^n + 1)$, in which coefficients are integers in $[-q/2, q/2)$, and the k-sized basis of mutually prime moduli $p_1, ..., p_k$, the RNS representation of a polynomial $A \in R_q$ is the vector of residue polynomial $(A_1, ..., A_k)$, such that $A_i \in R_{p_i} = \mathbb{Z}_{p_i}[X]/(X^n + 1)$.

When computing Negative Wrapped Convolution (NWC) some precomputed values are required. Weight values for weighted convolution and actual twiddle factors for underlying NTT are indifferently called twiddle factors here. The concatenation of all twiddle factors for a specific field $\mathbb{Z}_{p_i}$, is called a twiddle factor set. Four subsets of a twiddle factor set appear in our discussions: the input weight-vector $\Psi_i = (\psi_i^j)_{0\leq j<n}$, the twiddle factors for the forward NTT $\Omega_i = \{\omega_i^j\}_{0\leq j<n/2}$, the twiddle factors for the inverse NTT $\Omega_i^{-1} = \{\omega_i^{-j}\}_{0\leq j<n/2}$, and the output weight-vector $\Psi_i^{-1} = (\psi_i^{-j})_{0\leq j<n}$.

When considering FV parameters, the plaintext modulus is noted t, the size of the ciphertext modulus $S_q = \log_2 q$, the degree of the cyclotomic polynomial n, the evaluation multiplicative depth L, the prime sizes s, and the security coefficient λ.

2.2 Residue Number System

The Residue Number System is a non-positional representation of numbers according to a basis of mutually prime moduli $p_1, ..., p_k$. This representation is a direct consequence of the Chinese Remainder Theorem (CRT) which expresses the ring isomorphism $\mathbb{Z}_q \cong \prod_{1\leq i\leq k} \mathbb{Z}_{p_i}$. Under this representation, modular arithmetic modulo $q = \prod_{1\leq i\leq k} p_i$ is performed with k smaller and independent modular operations. For additions, subtractions and multiplications, the RNS representation is an efficient way of creating parallelism, but when it comes to divisions, some more complex computations like basis extensions are required. It is possible to exploit the parallelism brought by the RNS representation for large integer arithmetic. This only requires the RNS basis to be large enough to cover the dynamic range of the considered operations over $\mathbb{Z}$.

In lattice-based cryptography, and particularly in its use for homomorphic cryptography, polynomials with large size coefficients are manipulated. The size of these coefficients can reach several hundreds of bits, which implies an important complexity constant when performing polynomial operations using classical multi-precision arithmetic. Moreover, multi-precision arithmetic is less suitable for parallelism due to intermediate results propagation. For those reasons the RNS representation is considered as an interesting candidate for limiting the impact of complexity brought by arithmetic of large integers in lattice-based cryptography.

2.3 Negative Wrapped Convolution

One of the main performance bottlenecks of lattice based cryptography is brought by the underlying multiplications over the ring $R = \mathbb{Z}[X]/(f(X))$. In both hardware and software implementations, a common strategy to improve performances is to exploit the NTT-based negative wrapped convolution theorem to perform those multiplications [PG12]. This

approach restricts the choice of $f(X)$ to cyclotomic polynomial of order m a power of two ($f(X) = \Phi_m(X) = X^n + 1$, with $n = m/2$).

Under RNS representation, the multiplications over R are computed through multiple smaller multiplications over polynomial rings of the form $\mathbb{Z}_{p_i}[X]/(X^n + 1)$. This implies a reduction in the choice of RNS basis elements to ensure the applicability of a negative wrapped convolution over each finite-field $\mathbb{Z}_{p_i}$. To compute such a convolution, one has to find an n-th primitive root of -1, which exists if and only if $p_i = 1 \bmod 2n$. In this paper, RNS basis elements are selected as primes using the prime selection algorithm of NFLlib [AMBG+16].

Multiplications over rings $\mathbb{Z}_{p_i}[X]/(X^n + 1)$ are performed with NTT-based weighted convolutions of size n. That is to say, for each finite-field $\mathbb{Z}_{p_i}$, computation of the twiddle set $\{\psi_i^j\}_{0 \leq j < 2n}$ with ψ_i a n-th primitive root of -1 over $\mathbb{Z}_{p_i}$ is required. In practice, ψ_i is chosen such that $\omega_i = \psi_i^2 \bmod p_i$ is a n-th primitive root of unity over $\mathbb{Z}_{p_i}$. Doing so, twiddle factor sets for the n-point NTT, and inverse NTT, are subsets of respectively $\{\psi_i^j\}_{0 \leq j < n}$ and $\{\psi_i^{-j}\}_{0 \leq j < n}$.

For A_i and B_i in $\mathbb{Z}_{p_i}[X]/(X^n + 1)$, ring product R_i is computed as in equation (1), with $\Psi_i = \{\psi_i^j\}_{0 \leq j < n}$ and $\Psi_i^{-1} = \{\psi_i^{-j}\}_{0 \leq j < n}$.

$$R_i = \Psi_i^{-1} \odot INTT_i\left((NTT_i\left(\Psi_i \odot A_i\right)) \odot (NTT_i\left(\Psi_i \odot B_i\right))\right) \tag{1}$$

3 Related Works & Motivations

The underlying hardware acceleration strategy targets the scheme proposed by Fan and Vercauteren in [FV12] and its full RNS variant brought by Bajard et al. in [BEHZ16] and further improved by Halevi et al. in [HPS18]. Nevertheless, our analysis and contributions could be exploited in others' RLWE based cryptosystems as our work mainly focuses on polynomial ring arithmetic.

3.1 Polynomial Ring Multiplication

The main motivation behind our work is to bring a consistent hardware implementation strategy to improve homomorphic evaluation performance. In a previous work [CCSV17], we profiled an homomorphic evaluation of Trivium [CCF+16], using an FV implementation based on FLINT [CDS15]. More than 99% of the estimated cycles are spent in ciphertext multiplications and relinearizations. At a lower arithmetic level, on the overall evaluation of Trivium, more than 75% of the estimated cycles are spent in FFT convolutions to compute polynomial multiplications.

The complexity of the underlying polynomial multiplications comes both from the size of the coefficients and from the degree of the polynomials. A common strategy to tackle that complexity is with the combination of RNS representation and NTT-based polynomial multiplication. Following this approach, Bajard et al. [BEHZ16] and Halevi et al. [HPS18] have proposed a full RNS version of the FV scheme. Our acceleration strategy fits with this aforementioned BFV scheme.

At the time of writing, the most recent software implementation of the BFV scheme is accessible in the PALISADE library [PRR]. For more accurate projections regarding our hardware acceleration strategy, the profiling of critical functions is directly extracted from [HPS18][1] as reminded in Table 1. According to their paper, the main bottleneck is still due to NTT required to compute multiplications over the polynomial rings $\mathbb{Z}_{p_i}[X]/(X^n+1)$. Hence, this work addresses the acceleration of these polynomial multiplications.

[1] This profiling refers to the first version of their article from January 2018. The second version from June 2018 has slightly different parameters which reduce the part of the NTT computations while increasing CRT extension and scaling parts.

Table 1: Profiling of BFV ciphertext multiplication and relinearization by Halevi et al. reproduced from [HPS18]. Single-threaded mode, Linux CentOS, Intel Core i7-3770 CPU 4 cores at 3.40GHz with 16 GB of RAM; plaintext space $t = 2$, $s \approx 47$, security $\lambda > 128$

L	n	S_q	k	Total	Mul.	Relin.	Mult. details		
							CRT ext. & Scaling	NTT	Others
				ms	ms	ms			
1	2^{12}	94	2	17.7	15.9	1.76	34 %	60 %	6 %
5	2^{13}	141	3	53.7	46.3	7.42	34 %	62 %	4 %
10	2^{14}	235	5	197.8	158	39.8	33 %	62 %	5 %
20	2^{14}	376	8	349.6	258	91.6	37 %	59 %	4 %
30	2^{15}	564	12	1,334	858	476	40 %	56 %	4 %

3.2 Residue Multiplication Over Polynomial Rings

Related works explore different strategies to perform polynomial multiplications. A hardware/software co-design of a Karatsuba polynomial multiplication from Migliore et al. [MRL+18] brings an alternative to the popular NTT-based approach for small parameter sets (for FHE evaluation with small multiplicative depth: 4 in their case). Migliore et al. identified a turning point in their approach for degree $6,144$ and coefficient of size 512 bits, upon this range of parameters, the asymptotic complexity of Karatsuba does not permit to compete with NTT-based approach. It has to be emphasized that neither the Karatsuba approach, nor the NTT-based approach from Pöppelmann et al. [PNPM15] to which they compare, handle polynomial coefficients under RNS representation.

In [ÖDSS15], Öztürk et al. proposed a RNS and NTT based polynomial multiplication. As their architecture is not pipelined, it cannot start a new polynomial multiplication before the previous one finishes. Its latency is then paid numerous time for the computation of a polynomial multiplication over $\mathbb{Z}[X]/(X^n + 1)$ (as much as the size of the extended RNS basis). Furthermore, Öztürk et al. choose to pre-compute the different NTT twiddle factor sets on the host side, and send them along with the polynomial coefficients through the bus on which their accelerator is connected. Doing so, the communication cost between the host and the accelerator is doubled.

Cousins et al. [CRS17] developed an Homomorphic Encryption Processing Unit to accelerate the LTV scheme, which is not scale-invariant like FV, but also has its bottleneck complexity in polynomial ring multiplications. They implemented a pipelined NTT as a primitive of the HEPU, and contrary to [ÖDSS15], they chose to store the NTT twiddle factors in ROM filled up at compile time. As they point out, the storage capacity required for the different twiddle factor sets, one for each element p_i of the modulus chain, is quite important and uses a large part of the available BRAM on the targeted FPGA. This problem arises also for the FV scheme when polynomials are handled under RNS representation of their coefficients.

Sinha Roy et al. [RJV+15] present a co-processor (HE-processor) implementing building block operations for RLWE-based schemes, and in particular NTT and CRT primitives. They implement a memory access iterative NTT with improved routing of coefficients. They store in ROM only a subset of each required twiddle factor set and compute the others when needed. This results in a reduced memory requirement ($O(k \log_2(n))$) compared to [CRS17] ($O(kn)$). Nevertheless, they note that the computation of the other twiddles inserts some bubbles into the NTT computation (up to $\sim 10,000$ bubbles for $n = 2^{16}$).

In view of implementation issues previously expressed, our acceleration strategy is to implement a data flow oriented NTT-based polynomial ring multiplier, with on-the-fly computation of the twiddle factor sets.

3.3 Towards Automatic Generation of RTL Level Design

In our approach, the most complex operation to implement in hardware is the Number Theoretical Transform (NTT). This operation is similar to a Discrete Fourier Transform (DFT) in which complex arithmetic is replaced with modular arithmetic. With this in mind, our work explores the generalization of the hardware backend of the SPIRAL tool, from Milder et al. [MFHP12], to generate NTT designs in addition to DFT designs.

The SPIRAL project studies automatic generation of hardware and software for digital signal processing and other areas. Thus the DFT structure has already been explored in great details forming an ideal starting point to generalize towards NTT implementations. For example, in a PhD thesis work, LingChuan Meng [Men15] explores the automatically generating tuned software libraries for modular polynomial multiplication. However, a similar extension to SPIRAL's hardware generation capability has not been explored; for example [MFHP12] focuses on using SPIRAL to generate hardware for linear DSP transforms; while [ZMP16] generates hardware for sorting networks. In this paper, we investigate the DFT hardware created by the SPIRAL tool and propose generalizations required to make it compliant with NTT-based polynomial ring multiplications.

The long-term perspective is to be able to express high level directives to an NTT design generator, allowing a system designer to tune the performance of its NTT according to application and system requirements. Tuned parameters could be related to lattice-based cryptosystem parameters, like NTT size n and manipulated word size s, or part of the implementation parameters like architecture type, radix size or streaming width.

In this work, we modify the DFT hardware produced by SPIRAL to convert it into a practical NTT structure for polynomial ring multiplications by making two sets of changes. First, we replace the DFT's arithmetic blocks with those that perform modular arithmetic. Second, we adapt the design's twiddle factor storage system. This second change is crucial to our task, as in our application context, a notable difference from classical DFT hardware implementation is the necessity to change the twiddle factors of the NTT each time it handles a polynomial of a different RNS channel (ring $\mathbb{Z}_{p_i}[X]/(X^n+1)$).

Thus, a part of the contributions presented in this paper is a method for handling circularly-buffered twiddle factors to make NTT design compliant with regular, if not systematic, changes of twiddle factor sets. Regarding time constraints, only fully-streaming architectures for NTT generated with the SPIRAL hardware backend are considered in this paper. Nevertheless, further work could explore the adaptation of our first hand-made solution to other NTT designs.

4 Residue Polynomial Multiplier Design

This section describes our design of a multiplier over power of two cyclotomic polynomial rings ($\mathbb{Z}_{p_i}[X]/(X^n+1)$). The main difference from related works is the generation of required twiddles values for NWC in parallel of the data path. It results in an hardware accelerator imposing no choice of RNS basis at compile time, beside the size of the primes.

4.1 Global Architecture Overview

Our first analysis of FV evaluation complexity, detailed in [CCSV17], leads us to accelerate the overall Residue Polynomial Multiplication (RPM). Furthermore, regarding the number of RPM to perform during ciphertext operations, it has been chosen to design a streaming architecture, in order to pipeline the RPM of different channels. When considering RPM through NWC, the operations to perform are summarized by equation (1).

The NWC requires the set of precomputed values $\{\Psi_i, \Psi_i^{-1}\} = \{\psi_i^j\}_{0\leq j<2n}$, and the pair (p_i, v_i) (see section 4.4), to perform a multiplication over $\mathbb{Z}_{p_i}[X]/(X^n+1)$. With n and q being large for practical HE, tens of different polynomial rings require $2n+2$ values.

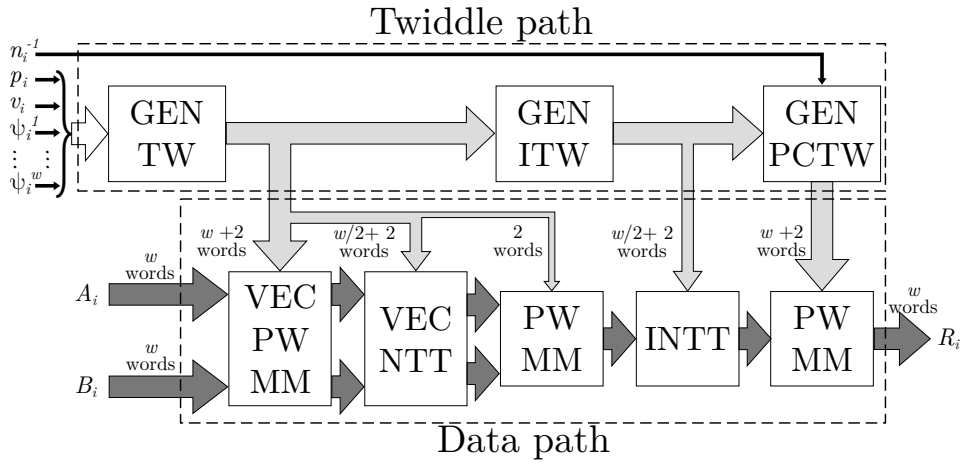


Figure 1: Residue Polynomial Multiplier (RPM) flow.

For scalability over RNS basis size, it has been chosen to locally generate the twiddle factor sets and use them on-the-fly.

The overall architecture flow is presented in Figure 1 without control and artificial latency for representation simplicity. The architecture is generic regarding the size of the NTT n, the width of the data path w (called streaming-width), and the prime size s in bits, with n and w being powers of two and $s \leq 64$.

In the following description, multiplication refers to multiplication over the ring $\mathbb{Z}_{p_i}$. As presented in section 4.4, modular multiplications are performed following the NFLlib algorithm [AMBG+16], and require appropriate prime p_i and reciprocal v_i as inputs.

There are two parallel paths in this architecture: the twiddle path and the data path. On one side, the twiddle path feeds the data path with the appropriate twiddle values, consistent with the actual polynomial ring $(\mathbb{Z}_{p_i}[X]/(X^n+1))$ of the residue polynomials A_i and B_i. On the other side, the data path performs the negative wrapped convolution of the two input polynomials seen as n-sequence of coefficients.

Data path. Five distinct steps are required to perform a NWC on inputted polynomials.

The first step is performed by VEC PW MM and consists of inner-products of the input polynomials with the weight-vector $\Psi_i = (\psi_i^j)_{0 \leq j < n}$ to output the polynomials $\Psi_i \odot A_i$ and $\Psi_i \odot B_i$. Only the n first elements of the twiddle factor set are required.

The second step VEC NTT computes forward NTT on each input and outputs simultaneously the transformed polynomials $NTT_i(\Psi_i \odot A_i)$ and $NTT_i(\Psi_i \odot B_i)$. It needs $\Omega_i = \{\omega_i^j\}_{0 \leq j < n/2} = \{\psi_i^{2j}\}_{0 \leq j < n/2}$ which is a subset of the values involved in Ψ_i.

The third step PW MM corresponds to the inner-product of the two weighted polynomials in the NTT domain $NTT_i(\Psi_i \odot A_i) \odot NTT_i(\Psi_i \odot B_i)$. Only the pair (p_i, v_i) is required to perform this inner-product.

The fourth step INTT reverts the polynomial from the NTT domain, and twiddles $\Omega_i^{-1} = \{\omega_i^{-j}\}_{0 \leq j < n/2} = \{\psi_i^{-2j}\}_{0 \leq j < n/2}$ are required. Ω_i^{-1} is a subset of the weight-vector Ψ_i^{-1} used in the fifth step.

Finally, the fifth step PW MM performs, in a single step, the scaling by n_i^{-1} mod p_i required at the end of INTT, and the inner-product with the weight-vector $\Psi_i^{-1} = (\psi_i^{-j})_{0 \leq j < n}$. Thus, only the n last elements of the twiddle factor set are required.

Twiddle path. As emphasized in the description of the data path, the twiddle values are not all required at the same time. Consequently, the computation of the twiddles is

decomposed in three steps.

The first step consists of the generation of the n first powers of ψ_i, namely $\Psi_i = \{\psi_i^j\}_{0 \leq j < n}$. Along with the corresponding (p_i, v_i) pair, they feed the first three steps of the data path. The twiddle generator GEN TW, described in section 4.3, only requires the first w elements $(\psi_i^1, ..., \psi_i^w)$ of the Ψ_i sequence, and outputs the n sized sequence at a rate of w elements per cycle, after a certain latency.

The second step GEN ITW outputs, after a certain latency, the sequence $\Psi_i^{-1} = \{\psi_i^{-j}\}_{0 \leq j < n}$ at a rate of w elements per cycle. The computation of this sequence is done by first computing the sequence $\{\psi_i^{-(n-j)}\}_{0 \leq j < n}$, and then reordering it to obtain $\{\psi_i^{-j}\}_{0 \leq j < n}$. The sequence to reorder is computed by subtracting each element of Ψ_i from p_i[1]. Half of the Ψ_i^{-1} sequence feeds the inverse NTT, because only $\{\psi_i^{-2j}\}_{0 \leq j < n/2}$ is required.

The third step GEN PCTW scales the sequence outputted by GEN ITW by n_i^{-1} (inverse of n in $\mathbb{Z}_{p_i}$). It then feeds the point-wise multiplier (again with (p_i, v_i)) at the end of the data flow which, thus, can complete the negative wrapped convolution.

Data flow operations. The overall architecture is data flow oriented, meaning that it starts a new polynomial multiplication, over a different RNS channel (polynomial ring $\mathbb{Z}_{p_i}[X]/(X^n + 1)$), every $T = n/w$ cycles. From now on, T will be identified as the throughput of the RPM design.

As the overall design is pipelined, the streaming NTT architecture has to manage multiple twiddle sets at a time, one for each RNS channel simultaneously active on the data path. Moreover, contrary to classical DFT architecture in which twiddle factors do not change with the inputs, the twiddle memories have to be programmable in our case. In the next section, we describe our proposed architecture which desirably achieves no stalling in the NTT data path by means of cyclic reprogramming of the twiddle set memories.

For the RPM to achieve a throughput of $T = n/w$ cycles, the different twiddle sequences, computed by the twiddle path, have to be generated with the same throughput. Section 4.3 details the generation of the initial sequence $\Psi_i = \{\psi_i^j\}_{0 \leq j < n}$ from the first w elements $(\psi_i^1, ..., \psi_i^w)$. Then, the generation of subsequent sequences with the required throughput is quite straightforward.

4.2 Number Theoretical Transform

The forward NTTs and the inverse NTT have the same architecture, the only difference is in the twiddle sets, namely Ω_i for the forward one and Ω_i^{-1} for the inverse one. The core of the NTT architecture is generated by the hardware backend of SPIRAL [MFHP12], and modified to handle multiple RNS channels in the data path. For simplicity, but without loss of generality, all figures and examples in the following description consider $w = 2$.

Modified NTT architecture. In the initial SPIRAL generated fully-streaming architecture, the NTT is composed of several type of stages. When $w = 2$ (and n a power of two) there are three types of stages: permutation (*P Stage*), multiply (*M Stage*) and butterfly (*B Stage*). For each type of stage, we look for the required precomputed values, specific to a RNS channel, to now consider them as inputs for the stage.

No modification is required for a permutation stage as it does not require any twiddle values, nor the (p_i, v_i) pair. Each multiply stage requires a subset of the twiddle factors, depending on the considered multiply stage, plus the (p_i, v_i) pair to perform multiplications over $\mathbb{Z}_{p_i}$ (see section 4.4 for more details on modular arithmetic). Finally, a butterfly

[1] First $\psi_i^n = -1 \bmod p_i$ implies $\psi_i^{2n} = 1 \bmod p_i$. Then, $\psi_i^{-(n-j)} = \psi_i^{-n}\psi_i^j = \psi_i^n\psi_i^j \bmod p_i$. And, lastly, $\psi_i^n\psi_i^j = (p_i - 1)\psi_i^j = p_i - \psi_i^j \bmod p_i$. Hence, $p_i - \psi_i^j = \psi_i^{-(n-j)} \bmod p_i$.

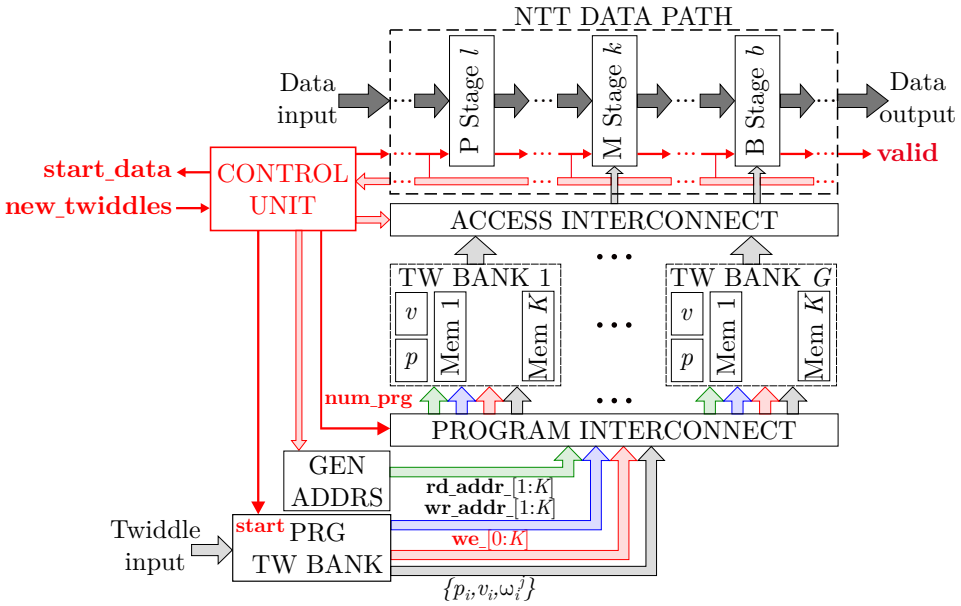


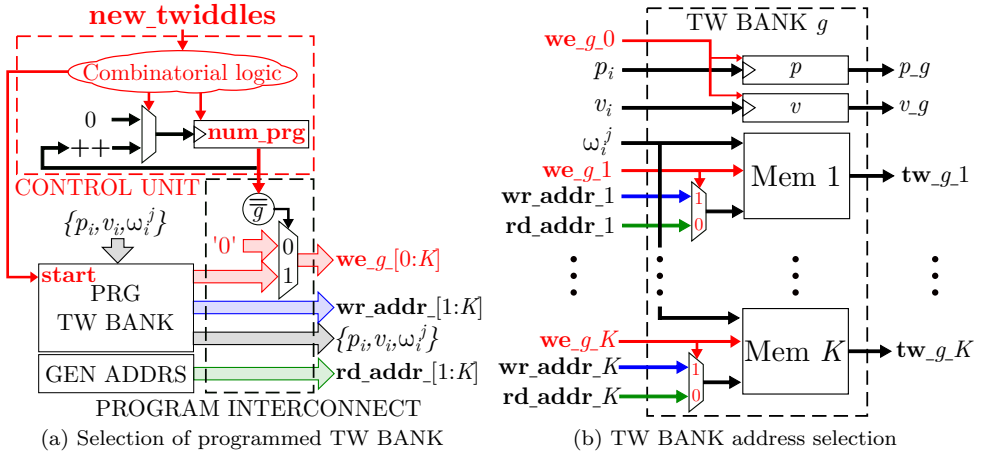
Figure 2: Number Theoretical Transform (NTT) flow. Schematic for $w = 2$.

stage requires only the value p_i to perform its operations. For different values of w, and depending on other architecture parameters (like radix size), some stages can be hybrid of butterfly and multiply. Nevertheless, required twiddles can be identified for each one of them, and same modifications described below can be applied.

Initially, each multiply stage had its own dedicated twiddle memory implemented as ROM and filled up at compile time. The extension of the NTT architecture implemented here requires disassociating the twiddle memories of all concerned stages, implementing them as RAM, and handling them as a bank of memories. From now on, a twiddle bank refers to the concatenation of all twiddle memories for a specific RNS channel. Each twiddle bank stores a twiddle factor set of one RNS channel at a time, and is reprogrammed with a new set when required. The maximum number of simultaneous RNS channels in the data path is $\lceil lat_{NTT}/T \rceil$. To avoid any overlap between programming and accessing twiddle banks, the architecture instantiates $G = \lceil lat_{NTT}/T \rceil + 1$ of them.

Figure 2 shows the resulting NTT architecture. The G different twiddle banks feed the NTT data path through an interconnect controlled by CONTROL UNIT. The same control unit selects also the twiddle bank currently programmed by the PRG TW BANK unit. PRG TW BANK generates the **we_**[0:K] and **wr_addr_**[1:K] signals for each memory of the programmed bank, consistent with the current $w/2$ twiddle factors flowing through. It also updates the (p_i, v_i) pair of the programmed bank at the beginning of the reprogramming procedure. The banks that are not currently programmed are accessed according to the **rd_addr_**[1:K] signals generated by GEN ADDRS. For each memory in the bank, the **wr_addr_**k signal generation is updated by the CONTROL UNIT that receives control feedback from the corresponding stage on the data path flow.

Reprogramming a twiddle bank. A twiddle bank (TW BANK) is the concatenation of all the twiddle memories required in the NTT data path. For $w = 2$ there are $K = \log_2(n) - 1$ multiply stages that require a twiddle memory. Butterfly stages only require the prime p_i, and permutation stages require no RNS channel specific values. In this case, the memory of the k-th multiply stage (with $k \in \{1, ..., K\}$) contains 2^k twiddles. For each RNS channel,



(a) Selection of programmed TW BANK (b) TW BANK address selection

Figure 3: Bank of memory reprogramming. Schematics for $w = 2$.

only $n/2$ different twiddle factors are required to compute the NTT, but half of the factors are duplicated from one multiply stage to another. It results in a total of $n-2$ single words stored in K different address spaces, plus a pair of single words (p_i, v_i) that characterizes the current RNS channel in bank.

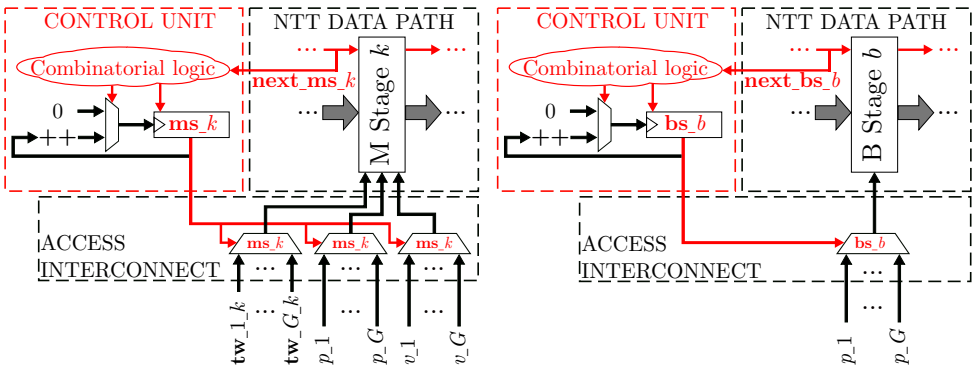
To reprogram a twiddle bank, the pair (p_i, v_i) and the twiddle factors $\{\omega_i^j\}_{0 \leq j < n/2}$ are sent through PRG TW BANK along with appropriate **write address** and **write enable** signals for each memory of the bank. As seen in Figure 3a, the bank currently programmed receives the **we_**[0**:**K] signals from PRG TW BANK: bank number g is reprogrammed when **num_prg** is equal to g. Other banks are only addressed for reads, using the simple mechanism of address selection in Figure 3b. The choice of the bank currently programmed is done by cyclically updating the **num_prg** register in $\{1, ..., G\}$ with the arrival of new twiddle factor sets, signaled with **new_twiddles** going high for one cycle.

The signal generation of PRG TW BANK depends on two factors: the way the twiddle sequence $\{\omega_i^j\}_{0 \leq j < n/2}$ is inputted, and the way they have to be dispatched in the different memories of a bank. To respect the throughput of the overall architecture, the reprogramming has to be done in at most $T = n/w$ cycles. As an example, the case $w = 2$ is presented in the following description.

The sequence of twiddle factors is inputted one per cycle in increasing order of power. For $k \in \{1, ..., K\}$, the k-th memory of the bank contains the subset $\{\omega_i^{(n*j)/2^{k+1}}\}_{0 \leq j < 2^k}$. Therefore, the address **wr_addr_**k and the signal **we_**k are updated every $n/(2^{k+1})$ cycles. The required throughput is thus achieved.

Accessing the twiddle factors. The mechanism instantiated in the ACCESS INTERCONNECT, responsible of feeding appropriate values to each stage of the NTT data path, is similar to the selection of the programmed bank.

A distinction has to be made between the different types of stages. As an example with $w = 2$, Figure 4a shows for a multiply stage, and Figure 4b for a butterfly stage, the selection of the correct values among the outputs of the G different twiddle banks. In both cases, the principle is the same: the arrival of a different RNS channel in the data flow is signaled by a **next** signal. This signal is responsible of the cyclic update of the corresponding register in the CONTROL UNIT (**ms_**k and **bs_**b in Figure 4). The **next** signals of multiply stages are also responsible of the re-synchronization of the **rd_addr_**k generators in the GEN ADDRS unit, but this is not shown in Figure 4a for simplicity.



(a) Twiddle access for a multiply stage (b) Twiddle access for a butterfly stage

Figure 4: Control of the twiddle access to feed NTT data path. Schematics for $w = 2$.

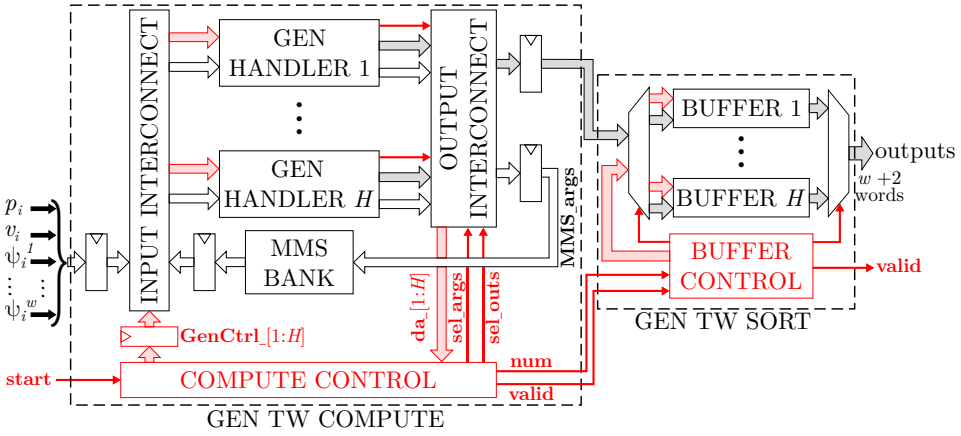


Figure 5: Generation of twiddles (GEN TW) flow.

4.3 Twiddle Factor Generator

Our acceleration strategy is based on the generation of the n-sequence $\Psi_i = \{\psi_i^j\}_{0 \leq j < n}$ with the required throughput $T = n/w$, from the initial knowledge of the first w elements $(\psi_i^1, ..., \psi_i^w)$ only. From a high level point of view, it is required to compute n elements in T cycles, so if the generator outputs w elements per cycle the required throughput is achieved. The difficulties of this generation come both from the dependence between the elements of the sequence to be generated and from the latency of the modular multipliers that compute the elements. This problem can be expressed as the search for an overlap in a dependency graph, in which different solutions can be found regarding different constraints. As the problem of generating the power sequence of a number is outside the scope of this document, the following brief description simply presents the generator architecture and details only how it meets the needs of the RPM architecture.

The principle of our solution is that when there are inevitable bubbles in the generation of a set, due to expectation of intermediate results, the generator fills these bubbles with calculations from another set's generations ready to be performed. It results in a mixed set output sequence from which each n-length sequences has to be sorted out.

Consequently, the generation is done in two steps presented in Figure 5. The generator handles up to H different twiddle set generations at the same time, and schedules them on

the single computing resource MMS BANK, which contains exactly w modular multipliers. When T is large in front of the modular multiplier latency (which is true for lattice-based cryptography applications), $H = 3$ is sufficient to saturate the MMS BANK with twiddle computations and achieve the required throughput to feed the rest of the RPM design.

Each twiddle set is associated to a specific GEN HANDLER which instantiates data handling according to chosen generation heuristic[1]. COMPUTE CONTROL schedules the different twiddle set generations by supervising their sequential access to MMS BANK. It updates at each cycle the **GenCtrl** signal for each GEN HANDLER, and selects appropriately which one feeds MMS BANK with new arguments (**MMS_args**), and which one outputs the w further elements of its twiddle set.

Each output of GEN TW COMPUTE unit is associated to **num** and **valid** signals that specify the validity of the output and its origin. The outputs of GEN TW COMPUTE are sorted in H different buffers according to these signals. When a GEN HANDLER finishes its twiddle set generation, BUFFER CONTROL initiates the output of the n-sequence stored in the corresponding BUFFER, w elements per cycle. The concerned GEN HANDLER and BUFFER can then be used for a new twiddle set generation.

The latency of the twiddle factor generator, namely the number of cycles between the input of the initial elements $\psi_i^1, ..., \psi_i^w$ and the w first outputs of the n-sequence by GEN TW SORT, is a bit larger than T. Consequently, the RPM requires artificial latencies in the data path to synchronize the output of the twiddles with the inputs of the coefficients. On experimental grounds, these latencies are not too large, but still uses some BRAM resources on an FPGA implementation. It is nevertheless a relatively small cost regarding the impact of BRAM utilization for NTT permutations for large n.

4.4 Modular arithmetic

Our RPM design is based on modular arithmetic, which is dependent on the considered modulus (p_i). It is considered here that RNS basis elements are selected using the prime selection algorithm from NFLlib [AMBG+16]. In addition to prime selection, NFLlib proposes a modular reduction algorithm compliant with selected primes. This modified Barrett reduction algorithm requires a $(s + 2)$-bit reciprocal related to the modulus p_i ($v_i = \lfloor 2^{2(s+2)}/p_i \rfloor \mod 2^{(s+2)}$).

For modular additions and modular subtractions, inputs are bounded by the modulus p_i (s-bit), thus they require only one addition, one subtraction and one comparison to be performed. Modular multipliers are instantiated by a classical s-bit multiplication followed by the modular reduction from NFLlib. It requires three s-bit multiplications, one $2s$-bit addition, two subtractions (one $2s$-bit and one s-bit), and one comparison.

As our RPM design is data flow oriented, all the modular operators implemented are pipelined. Modular additions and subtractions have two cycles latency ($Lat_{MADD} = Lat_{MSUB} = 2$), and modular multipliers' latency depends on the underlying s-bit multipliers ($Lat_{MM} = 3 * Lat_M + 3$, with Lat_M the latency of a s-bit multiplier).

5 Results and Approach Validation

This section provides implementation results for a proof-of-concept set of small cryptosystem parameters. Then, it studies the scaling of our approach to sets of larger cryptosystem parameters by changing SPIRAL generated DFT into NTT. This part allows us to explore performances of the RPM architecture on most of the parameter sets from [HPS18]. Finally, it shows the positive impact of the twiddle set generator on the scalability of the overall RPM for BFV-like homomorphic schemes.

[1] A example of heuristic : $\psi_i^{2j} = \psi_i^j \psi_i^j \mod p_i$ and $\psi_i^{2j+1} = \psi_i^j \psi_i^{j+1} \mod p_i$ for all j in $\{1, ..., n/2-1\}$.

Table 2: Resource utilization post implementation on a Virtex xc7vx690t. Synthesis, placement and route using Xilinx Vivado 2016.3. Frequency 200MHz.

Ressources		**RPM**					BCHI
type	available	**total**	NTT	MM	GTW	Others	& WRAP
LUT	432,368	**54,188**	41,964	5,198	5,906	1,120	27,775
LUTRAM	173,992	**14,402**	10,710	2,056	1,550	86	5,425
FF	864,736	**66,444**	50,961	6,755	7,761	967	39,614
BRAM	1,470	**208**	147	0	21	40	153
DSP	3,600	**517**	363	88	66	0	48
IO	600	**0**	0	0	0	0	59
Pcie	3	**0**	0	0	0	0	1

5.1 Implementation Results

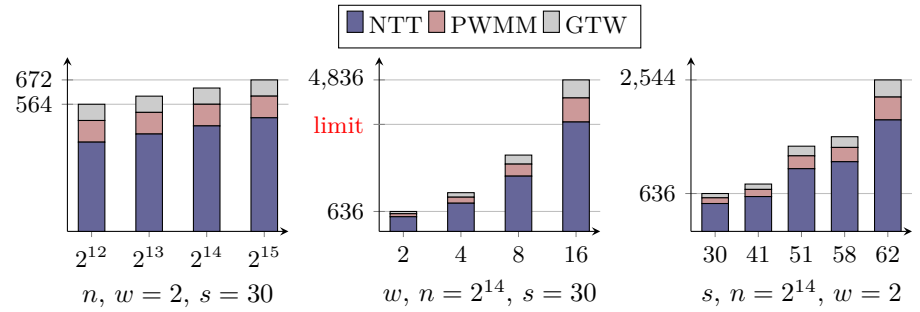
This subsection presents the implementation results, as a proof of concept, of the RPM design with $n = 4096$, $w = 2$ and $s = 30$. The experimentation has taken place on an Alpha-Data board ADM-PCIE-7V3, embedding a Xilinx Virtex 7 xc7vx690t, and connected to host PC through PCIe Gen3 $\times 8$ lanes. The RPM design is synthesized, placed and routed along with the Bridge Host Controler Interface (BHCI) IP, provided by Alpha-Data, controlling the PCIe and DMA that access the RPM design. Synthesis, placement and route have been completed with integrated tools of Xilinx Vivado 2016.3. The achieved running frequency is 200MHz.

In Table 2, the resource utilization post-implementation is shown for the proof-of-concept RPM design. Considering only the RPM design w.r.t. the FPGA resources, the critical resources are DSP and BRAM tiles with respectively 14,4% and 14,2% utilization, 12,5% for LUT, and 8,3% for LUTRAM. The larger part of the resource utilization comes from the three NTT (70,2% of DSP, 70,8% of BRAM, and 77,4% of LUT). The twiddle path, embedding our twiddle factor generator, uses roughly around 10%-13% of DSP, BRAM and LUT. The inner-products in the overall data flow consume 17% of the DSP, and the various latencies synchronizing the data path and the twiddle path together take 20% of the BRAM utilization. As expressed in section 4.3, the hardware cost for the synchronization can be considered constant as it becomes relatively small for larger n.

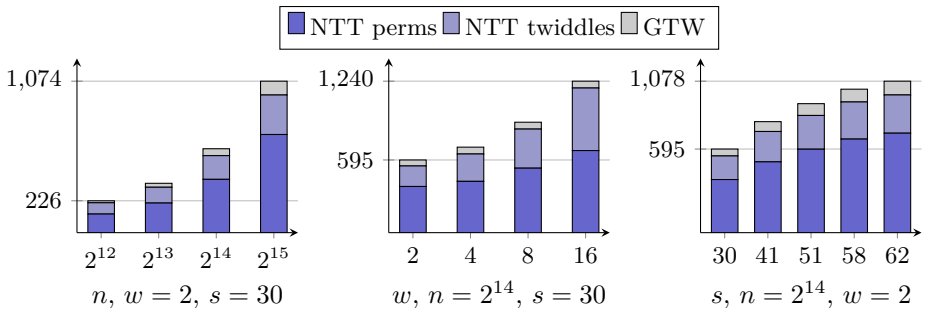
In the next section, we study the scalability of our approach over more practical parameter sets for homomorphic encryption. It has to be emphasized that it is a pessimistic study, as one can see for the case $n = 4096$, $w = 2$ and $s = 30$, when comparing the following estimate to the post-implementation hardware utilization presented in Table 2. Nevertheless, we prefer not to take into account in our discussion the potential optimizations specific to an implementation environment.

5.2 Scalability Over More Practical Parameter Sets

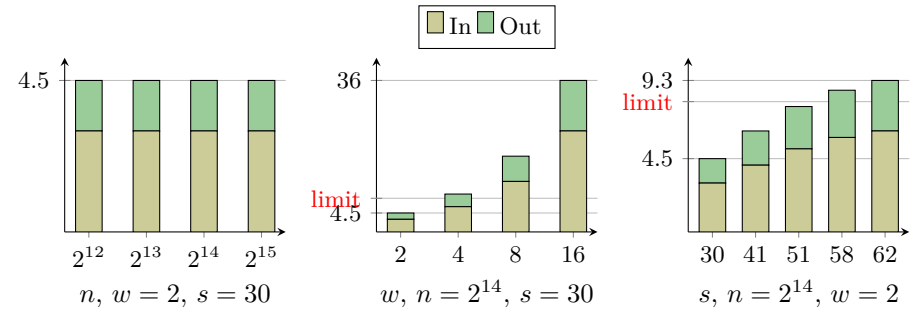
In order to analyze the scalability of our hardware acceleration approach, its behaviour under the concrete parameter sets from [HPS18] is studied in this section. The estimations presented here are built on two basis : the concrete implementation for $n = 4096$, $w = 2$ and $s = 30$, presented in previous subsection, and the estimated changes of SPIRAL generated DFT into NTT. For each estimation, we examined the resource count of the appropriate DFT design, and adjusted the costs of the arithmetic, memories, and required bandwidth to match the requirements of the corresponding modified NTT design. When considering a data flow design, going from DFT to NTT mainly impacts the hardware cost of the design, as the throughput does not change for a specific transform size. Similarly,



(a) Influence of (n, w, s) on DSP utilization.



(b) Influence of (n, w, s) on BRAM utilization.



(c) Influence of (n, w, s) on communication bandwidth requirements (GB/s).

Figure 6: Estimation of resource utilization under the influence of sizing parameters.

the impact on the latency of changing DFT into NTT is not considered here regarding the number of pipelined RPM to perform in practice.

Hardware cost. The development of the RPM design was oriented towards FPGA implementation and in the following discussion the hardware cost is expressed as the number of DSP and BRAM. A DSP refers to 7 series DSP48E1, and a BRAM refers to a 36Kb Block RAM. The utilization estimate is based on the corresponding Xilinx IP core generators. Note that neither the optimization from [CRS17] to reduce BRAM utilization for twiddle storage, nor potential synthesizer optimizations has been taken into account, resulting in a pessimistic estimate. Finally, the number of LUT is neglected because it does not appear as a critical resource in practice, similarly as in [CRS17].

All sizing parameters n, w and s have a significant impact on resource utilization. Figure 6a shows their influence on DSP utilization, Figure 6b on BRAM utilization and Figure 6c on communication bandwidth requirement. The *limit* value represents the

Table 3: Timing estimate derived from the profiling result of [HPS18]. Single-threaded mode, Linux CentOS, Intel Core i7-3770 CPU 4 cores at 3.40GHz and 16 GB of RAM; plaintext space $t = 2$, $s \approx 47$, security $\lambda > 128$.

L	n	S_q	Total	CRT ext. & Scaling		**Mul.RPM**		**Relin.RPM**		Others	
			ms	ms	%	**ms**	%	**ms**	%	ms	%
1	2^{12}	94	17.7	5.4	30.6	**10.3**	58.3	**1.7**	9.5	0.3	1.6
5	2^{13}	141	53.7	15.7	29.3	**30.2**	56.2	**7**	13.1	0.7	1.4
10	2^{14}	235	197.8	52.1	26.4	**104.3**	52.7	**37.8**	19.1	3.6	1.8
20	2^{14}	376	349.6	95.5	27.3	**160.5**	45.9	**87**	24.9	6.6	1.9
30	2^{15}	564	1,334	343.2	25.7	**507.9**	38.1	**452.2**	33.9	30.7	2.3

available hardware/bandwidth resource within Alpha-Data board of section 5.1, taking into account the BCHI usage and a 10% margin for a concrete implementation.

The degree n of the handled polynomials, in addition to reducing the RPM throughput ($T = n/w$), mainly impacts the number of BRAM required, in particular for the permutations in the NTT. The streaming width w improves the throughput of the RPM significantly, but has a heavy drawback on the DSP utilization, and on the required communication bandwidth. The elements size s has a balanced impact on BRAM utilization, DSP utilization and required communication bandwidth, but has no impact on RPM throughput. Nevertheless, some increments of s have a more significant impact on DSP utilization, and increase the latencies of basic arithmetic operators if one wants to keep the same running frequency.

Performance scalability. Some additional estimations have been made to study the performance scalability of the RPM design. The profiling from Halevi et al. [HPS18] considers the complexity at NTT level rather than RPM level. It is assumed that inner-products required for RPM operations are counted as part of *Others* in their profiling (Table 1). For the following projections, it is estimated that 80% of *Others* are in fact inner-products to performs RPM operations. Furthermore, not knowing the ciphertext relinearization primitive detailed profiling, it is estimated that 95% of the relinearization is spent performing the equivalent of RPM operations. Table 3 presents the resulting estimated profiling over which the following study is based.

Considering the complexity at RPM level makes us consider more NTTs than in the PALISADE implementation. During ciphertext multiplications, each polynomial is transformed to the NTT domain only once in their work. Considering RPM operations, polynomials are transformed each time they are required, i.e. twice. Even if comparisons based on timing with different abstraction levels are subject to caution, it can reasonably be considered here as a disadvantage in terms of acceleration results. Nevertheless, it is beyond the scope of this paper to study the choice of RPM acceleration rather than NTT acceleration, and this question is delayed to further works.

In Table 4 performance results over different parameter sets from [HPS18] are presented. The number of RPM performed during ciphertext multiplication and ciphertext relinearization depends on RNS basis sizes k and k'. Namely, tensor product of BFV ciphertext multiplication requires $3(k + k')$[1] residue polynomial multiplications, and each scalar product in ciphertext relinearization requires k^2 of them.

Here it is considered that $k' = k+1$ should be sufficient in practice to conduct operations in R during the tensor product in a ciphertext multiplication, as long as the primes are

[1] Using a Karatsuba-like approach.

Table 4: Estimated performance of our RPM design over the different parameter sets. Throughput $T = n/w$. Timings are estimated with a RPM design clocked at 200MHz. *Total* corresponds to the new timing for ciphertext Mult&Relin. (**su** stands for speedup).

Parameters						RPM	**Mul.RPM**		**Relin.RPM**		Total
L	n	S_q	s	k	w	1/ms	#	ms(**su**)	#	ms(**su**)	ms
1	2^{12}	94		4		97.7	27	0.3(**37.3**)	32	0.3(**5.1**)	6.3
5	2^{13}	141		5		48.8	33	0.7(**44.7**)	50	1.0(**6.9**)	18.2
10	2^{14}	235	30	8	2	24.4	51	2.1(**49.9**)	128	5.2(**7.2**)	63
20	2^{14}	376		13			81	3.3(**48.4**)	338	13.8(**6.3**)	119.3
30	2^{15}	564		19		12.2	117	9.6(**53**)	722	59.1(**7.6**)	442.6
					2	24.4		3.3(**48.4**)		13.8(**6.3**)	119.3
20	2^{14}	376	30	13	4	48.8	81	1.7(**96.7**)	338	6.9(**12.6**)	110.7
					8	97.7		0.8(**193.5**)		3.5(**25.1**)	106.4
					16	195.3		0.4(**387**)		1.7(**50.3**)	104.2
			30	13			81	3.3(**48.4**)	338	13.8(**6.3**)	119.3
			41	10			63	2.6(**62.2**)	200	8.2(**10.6**)	112.9
20	2^{14}	376	51	8	2	24.4	51	2.1(**76.8**)	128	5.2(**16.6**)	109.4
			58	7			45	1.8(**87.1**)	98	4.0(**21.7**)	108
			62	7							

correctly distributed in each basis.

5.3 Positive Impact of the Twiddle Factors Generator

The scalability of our approach is brought by the local generation of twiddle sets which is compared here to two other straightforward strategies. First, local storage in FPGA ROM at compile time, similar to the work of Cousins et al. [CRS17]. Second, external storage and communication along with polynomials, similar to the work of Öztürk et al. [ÖDSS15].

Local storage. For the first strategy, the proposed twiddle factor generator saves a large amount of BRAM, and makes the RPM design scalable regarding the RNS basis size. Indeed, the cost of handling multiple twiddle sets is now independent of k. In Table 5, are compared, in terms of FPGA resource utilization, the twiddle generation implemented in the RPM design, and the scenario where the twiddles are stored in ROM, on the FPGA, at compile time. To be more specific, it is considered that only the $\Psi = \{\psi^i\}_{1 \leq i \leq n}$ are stored for each twiddle set, and that the subsequent required values are computed similarly as in the RPM design without online twiddle factor generation.

Unsurprisingly, the number of BRAM needed to store all the different twiddle sets exponentially increases with larger parameters sets (to gain in multiplicative depth). Indeed, both the size of each set (depending on n) and the number of sets ($k + k'$) get larger (depending on S_q, for fixed prime size s). The number of instantiated BRAM is fixed by H (not considering the data path here) when using our twiddle factor generator. These BRAM are mainly used for the H different BUFFER (storing n elements) that sort the twiddles generated by GEN TW COMPUTE (Figure 5), and in practice $H = 3$ because T is large enough in front of a modular multiplier latency.

External storage. For the second strategy, the different twiddle sets are stored on external memories (from the accelerator's viewpoint). In this case, the required input bandwidth to receive the twiddle factors from external storage space is compared to the one required

Table 5: Resource utilization for local storage and local generation of twiddle factors.

Parameters						Local Storage		**Local Generation**	
L	n	S_q	s	k	w	DSP	BRAM	DSP	**BRAM**
1	2^{12}	94		4		24	44	48	**20**
5	2^{13}	141		5		24	91	48	**35**
10	2^{14}	235	30	8	2	24	266	48	**70**
20	2^{14}	376		13		24	432	48	**70**
30	2^{15}	564		19		24	1,053	48	**135**
					2	24	378	48	**70**
20	2^{14}	376	30	13	4	48	378	96	**70**
					8	96	432	192	**80**
					16	192	432	384	**80**
			30	13		24	406	48	**70**
			41	10		30	437	60	**95**
20	2^{14}	376	51	8	2	54	437	108	**115**
			58	7		60	442	120	**130**
			62	7		96	476	192	**140**

for local generation that is required by our RPM design. The memory footprint of the two approaches is also compared.

In the first approach the memory footprint is $O(kn)$ elements of size s, compared to $O(kw)$ in our approach. In the case of external storage, it is again considered that only half of a twiddle set is stored. The result of the comparison is viewed in the Table 6. The memory footprint of the twiddle factors goes from 0,14 MBytes to 4,79 MBytes for the considered parameter sets, this is not critical in practice, but still, it could be avoided with local generation requiring at most 1620 bytes (not considering word-wise storage).

A stronger disadvantage in the case of a data flow oriented RPM is the input bandwidth requirements for the precomputed values. Considering the needs of the RPM twiddle flow, namely w words of s-bit per cycle, storing the twiddle sets on external memories requires at least 1,5 GB/s, for RPM clocked at 200MHz, of communication bandwidth between the storage space and the RPM unit. With local generation of twiddle sets as instantiated in our RPM design, only w words of s-bit are required every T cycles, thus saving precious bandwidth to feed the accelerator with data leading to effective speedup.

6 Conclusion and Future Work

In this work, we designed a Residue Polynomial Multiplier to scale up evaluation capability for homomorphic encryption based on RLWE. The RPM design has been constructed studying the full RNS variant of the FV scheme, proposed by Bajard et al. [BEHZ16], and further improved by Halevi et al. [HPS18]. The resulting RPM is then fully compatible with the RNS representation w.r.t. polynomial coefficients, and implements an NTT-based negative wrapped convolution to perform polynomial ring multiplications.

In order to address practical parameter sets (for reasonably large multiplicative depth), our RPM embedded its own twiddle factor generator. This generator makes the RPM design BRAM utilization independent of the RNS basis size while avoiding a non-negligible communication cost between the host and the accelerator.

Compared to the software implementation of [HPS18], it is estimated that our RPM design speeds-up the overall ciphertext multiplication and relinearization by a factor between 2.81 ($n = 2^{12}$,$w = 2$,$s = 30$) to 3.19 ($n = 2^{14}$,$w = 2$,$s = 51$). These performance improvements occur while staying in achievable FPGA hardware utilization and PCIe com-

Table 6: Memory footprint and communication bandwidth requirements for external storage strategy and local generation of twiddle factors.

Parameters						External Storage		**Local Generation**	
L	n	S_q	s	k	w	MEM MB	BW GB/s	MEM **B**	BW **kB/s**
1	2^{12}	94		4		0.14		**67.5**	**0.73**
5	2^{13}	141		5		0.34		**83**	**0.37**
10	2^{14}	235	30	8	2	1.04	1.5	**128**	**0.18**
20	2^{14}	376		13		1.66		**203**	
30	2^{15}	564		19		4.79		**293**	**0.09**
					2		1.5	**203**	**0.18**
20	2^{14}	376	30	13	4	1.66	3	**405**	**0.73**
					8		6	**810**	**2.93**
					16		12	**1,620**	**11.72**
			30	13		1.66	1.5	**203**	**0.18**
			41	10		1.76	2.1	**216**	**0.25**
20	2^{14}	376	51	8	2	1.78	2.6	**217**	**0.31**
			58	7		1.78	2.9	**218**	**0.35**
			62	7		1.9	3.1	**233**	**0.38**

munication bandwidth requirements. After acceleration, the new performance bottleneck is located mainly in RNS extension and RNS scaling procedures (more than 75% of the new timing), which parallelize well according to [HPS18].

Further work will compare the acceleration of only NTT rather than RPM, to take into account algorithmic optimizations that reduce the equivalent number of NTT. This comparison should be followed by a concrete prototype, with real timing and hardware utilization results.

Acknowledgments

We would like to thanks the anonymous reviewers for their constructive comments which lead to improvements of the present paper.

References

[AMBG+16] Carlos Aguilar-Melchor, Joris Barrier, Serge Guelton, Adrien Guinet, Marc-Olivier Killijian, and Tancrède Lepoint. NFLlib: NTT-based fast lattice library. In *Topics in Cryptology - CT-RSA 2016*, pages 341–356. Springer Nature, 2016.

[BEHZ16] Jean-Claude Bajard, Julien Eynard, Anwar Hasan, and Vincent Zucca. A Full RNS Variant of FV like Somewhat Homomorphic Encryption Schemes. In *Selected Areas in Cryptography - SAC*, St. John's, Newfoundland and Labrador, Canada, August 2016.

[Bra12] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In *Advances in Cryptology–CRYPTO 2012*, pages 868–886. Springer, 2012.

[BV11] Zvika Brakerski and Vinod Vaikuntanathan. Efficient Fully Homomorphic Encryption from (Standard) LWE. In *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*. IEEE, oct 2011.

[CCF+16] Anne Canteaut, Sergiu Carpov, Caroline Fontaine, Tancrède Lepoint, María Naya-Plasencia, Pascal Paillier, and Renaud Sirdey. Stream Ciphers: A Practical Solution for Efficient Homomorphic-Ciphertext Compression. In *Fast Software Encryption*, pages 313–333. Springer Nature, 2016.

[CCSV17] Joël Cathébras, Alexandre Carbon, Renaud Sirdey, and Nicolas Ventroux. An Analysis of FV Parameters Impact Towards Its Hardware Acceleration. In *Financial Cryptography and Data Security*, pages 91–106. Springer International Publishing, 2017.

[CDS15] Sergiu Carpov, Paul Dubrulle, and Renaud Sirdey. Armadillo: A compilation Chain for Privacy Preserving Applications. In *Proceedings of the 3rd International Workshop on Security in Cloud Computing*. Association for Computing Machinery (ACM), 2015.

[CGGI16] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster Fully Homomorphic Encryption: Bootstrapping in Less Than 0.1 Seconds. In *Advances in Cryptology – ASIACRYPT 2016*, pages 3–33. Springer Nature, 2016.

[CNS+16] Sergiu Carpov, Thanh Hai Nguyen, Renaud Sirdey, Gianpiero Constantino, and Fabio Martinelli. Practical Privacy-Preserving Medical Diagnosis Using Homomorphic Encryption. In *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*. IEEE, jun 2016.

[CRS17] David Bruce Cousins, Kurt Rohloff, and Daniel Sumorok. Designing an FPGA-Accelerated Homomorphic Encryption Co-Processor. *IEEE Transactions on Emerging Topics in Computing*, 5(2):193–206, apr 2017.

[DDS14] Wei Dai, Yarkin Doroz, and Berk Sunar. Accelerating NTRU based homomorphic encryption using GPUs. In *High Performance Extreme Computing Conference (HPEC), 2014 IEEE*, pages 1–6. IEEE, 2014.

[DM15] Léo Ducas and Daniele Micciancio. FHEW: Bootstrapping Homomorphic Encryption in Less Than a Second. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015*, pages 617–640. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

[FV12] Junfeng Fan and Frederik Vercauteren. Somewhat Practical Fully Homomorphic Encryption. *IACR Cryptology ePrint Archive*, 2012:144, 2012.

[G+09] Craig Gentry et al. Fully homomorphic encryption using ideal lattices. In *STOC*, volume 9, pages 169–178, May 2009.

[GHS12] Craig Gentry, Shai Halevi, and Nigel P Smart. Homomorphic evaluation of the AES circuit. In *Advances in Cryptology–CRYPTO 2012*, pages 850–867. Springer, 2012.

[GSW13] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In *Advances in Cryptology–CRYPTO 2013*, pages 75–92. Springer, 2013.

[HPS18] Shai Halevi, Yuriy Polyakov, and Victor Shoup. An Improved RNS Variant of the BFV Homomorphic Encryption Scheme. Cryptology ePrint Archive, Report 2018/117, jan 2018. https://eprint.iacr.org/2018/117.

[KG18] Alhassan Khedr and Glenn Gulak. SecureMed: Secure Medical Computation Using GPU-Accelerated Homomorphic Encryption Scheme. 22:597–606, mar 2018.

[KGV16] Alhassan Khedr, Glenn Gulak, and Vinod Vaikuntanathan. SHIELD: Scalable Homomorphic Implementation of Encrypted Data-Classifiers. *IEEE Transactions on Computers*, 65(9):2848–2858, sep 2016.

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. *On Ideal Lattices and Learning with Errors over Rings*, pages 1–23. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[Men15] Lingchuan Meng. *Automatic Library Generation and Performance Tuning for Modular Polynomial Multiplication*. PhDthesis, Drexel University, 2015.

[MFHP12] Peter Milder, Franz Franchetti, James C. Hoe, and Markus Püschel. Computer Generation of Hardware for Linear Digital Signal Processing Transforms. *ACM Transactions on Design Automation of Electronic Systems*, 17(2):1–33, apr 2012.

[MRL+18] Vincent Migliore, Maria Mendez Real, Vianney Lapotre, Arnaud Tisserand, Caroline Fontaine, and Guy Gogniat. Hardware/Software Co-Design of an Accelerator for FV Homomorphic Encryption Scheme Using Karatsuba Algorithm. *IEEE Transactions on Computers*, 67(3):335–347, mar 2018.

[ÖDSS15] Erdinç Öztürk, Yarkin Doröz, Berk Sunar, and Erkay Savas. Accelerating Somewhat Homomorphic Evaluation using FPGAs. *IACR Cryptology ePrint Archive*, 2015:294, 2015.

[PG12] Thomas Pöppelmann and Tim Güneysu. *Towards Efficient Arithmetic for Lattice-Based Cryptography on Reconfigurable Hardware*, pages 139–158. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[PNPM15] Thomas Pöppelmann, Michael Naehrig, Andrew Putnam, and Adrian Macias. Accelerating Homomorphic Evaluation on Reconfigurable Hardware. In *Lecture Notes in Computer Science*, pages 143–163. Springer Berlin Heidelberg, 2015.

[PRR] Yuriy Polyakov, Kurt Rohloff, and Gerard Ryan. PALISADE lattice cryptography library.

[RJV+15] Sujoy Sinha Roy, Kimmo Järvinen, Frederik Vercauteren, Vassil Dimitrov, and Ingrid Verbauwhede. Modular Hardware Architecture for Somewhat Homomorphic Function Evaluation. In *Lecture Notes in Computer Science*, pages 164–184. Springer Berlin Heidelberg, 2015.

[ZMP16] Marcela Zuluaga, Peter Milder, and Markus Püschel. Streaming sorting networks. *ACM Trans. Des. Autom. Electron. Syst.*, 21(4):55:1–55:30, May 2016.