Hardware Acceleration of FHEW

Jonas Bertels, Michiel Van Beirendonck, Furkan Turan, Ingrid Verbauwhede
COSIC, KU Leuven
Kasteelpark Arenberg 10, Bus 2452, B-3001 Leuven-Heverlee, Belgium
{firstname.lastname}@esat.kuleuven.be

***Abstract*—The magic of Fully Homomorphic Encryption (FHE) is that it allows operations on encrypted data without decryption. Unfortunately, the slow computation time limits their adoption. The slow computation time results from the vast memory requirements (64Kbits per ciphertext), a bootstrapping key of 1.3 GB, and sizeable computational overhead (10240 NTTs, each NTT requiring 5120 32-bit multiplications). We accelerate the FHEW bootstrapping in hardware on a high-end U280 FPGA.**

To reduce the computational complexity, we propose a fast hardware NTT architecture modified from [5] with support for negatively wrapped convolution. The IP module includes large I/O ports to the NTT accelerator and an index bit-reversal block. The total architecture requires less than 225000 LUTs and 1280 DSPs.

Assuming that a fast interface to the FHEW bootstrapping key is available, the execution speed of FHEW bootstrapping can increase by at least 7.5 times.

I. Introduction

Homomorphic encryption means encryption, where someone can still do operations on encrypted data. Fast-working homomorphic encryption would significantly impact the financial sector [7], the web services sector, and many other sectors where a company needing computation cannot trust the company doing the computation to access their data.

There are two types of homomorphic encryption: Leveled Homomorphic Encryption (LHE) and Fully Homomorphic Encryption (FHE). LHE allows for a few computations, and FHE supports infinite computations. "Bootstrapping" is used to turn LHE into FHE. The computing party decrypts the ciphertext to allow for another LHE. However, the data owner has encrypted the key used in this decryption and the ciphertext with another encryption key. In this way, the decryption of the ciphertext by the computing party results in another ciphertext, and the computing party does not get access to the plaintext.

The FHEW scheme [4] is part of the third generation of FHE schemes [1], [2]. FHEW tackles the bootstrapping process's long length by executing only one NAND gate (other simple gate functions are also possible), followed immediately by a bootstrap step. Bootstrapping makes it a true FHE, as the bootstrapping completes within a reasonable amount of time, namely 137 milliseconds for NAND + bootstrap for 128-bit security on an Intel(R) Core(TM) i7-9700 CPU for the STD128 security parameters listed in Table I [6]. While one only computes one NAND gate at a time, there is no limit on the depth of our circuit, and we can write all functions as a combination of logic gates; there is no limit on the functions that can be executed.

In this work, we propose an FPGA-based accelerator for the compute-intensive bootstrapping step of FHEW. Our accelerator achieves a factor of 7.5 speed-up starting from an open-source NTT core developed by Mert et al. [5].

II. Background

In this section, we explain FHEW bootstrapping, focusing on the implementation aspects. See [2], [6] for a more in-depth mathematical view.

A. Preliminaries on the FHEW algorithm

1) LWE: The Learning With Errors problem forms the basis for the FHEW scheme. In LWE, we operate on elements defined in $\mathbb{Z}_q$, the ring of integers modulo q (usually $q = 512$).

We define an LWE ciphertext with parameters: the dimension n, the message modulus $t \geq 2$, and the ciphertext modulus q [2]. We also define a randomized rounding function $\chi : \mathbb{R} \rightarrow \mathbb{Z}$. We use m to denote our messages and $m \in \mathbb{Z}_t$. The secret key is a vector consisting of n elements of $\mathbb{Z}_q$, chosen uniformly from this space. The encryption of m under $\mathbf{s}$ is then ([2], equation 2):

$$\text{LWE}_s^{t/q}(m) = (\mathbf{a}, \chi(\mathbf{a} \odot \mathbf{s} + mq/t) \bmod q) \in \mathbb{Z}_q^{n+1} \quad (1)$$

We have $\mathbf{a}$ a vector (similar to s) consisting of n random elements of $\mathbb{Z}_q$. To decrypt this ciphertext, one takes $b - a \odot s$ and rounds, then scales by t/q.

2) RLWE: We also define Ring LWE (RLWE) in an almost identical way to LWE, but with our message $\tilde{m} \in R_q = \mathbb{Z}_q[x]/(x^{2^p} + 1)$. In other words, our message is a polynomial (which can be represented as a vector of coefficients). Similar to LWE, we define a and s as vectors of these polynomials, and the encryption of m is then [6]:

$$\text{RLWE}_s(\tilde{m}) = (a, as + e + \tilde{m}) \quad (2)$$

with and $e \leftarrow \chi_\sigma^{2^p}$ with p some integer.

3) RLWE' and RGSW: We now informally introduce two more definitions based on RLWE, which we need when discussing FHEW. RLWE' is defined as [2] [6] a vector of RLWE ciphertexts decomposed into a base B_g with $(B_g)^k = q$. The reason for this definition is that we want a system that creates as little noise as possible when operations are completed. By decomposing RGSW (to be defined later) into a larger base, less noise is generated (see section 5.1 in [6]). Using this RLWE' scheme, we can define multiplications between constants and an RLWE' ciphertext so that the result is another ciphertext. This multiplication is homomorphic, meaning that

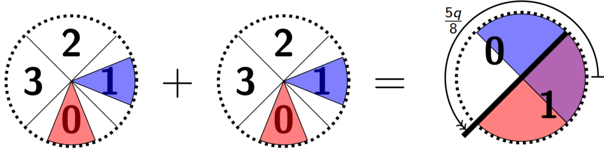

Fig. 1. NAND gate created by taking $\text{LWE}_s^{4/q}(\tilde{m})$ (with $e = q/16$) + $\text{LWE}_s^{4/q}(\tilde{m})$ (with $e = q/16$) = $\text{LWE}_s^{2/q}(\tilde{m})$ (with $e = q/8$) [3]

the result of two multiplied ciphertexts is the encryption of the product of the two ciphertexts along with some error, similar to equation 6.

And finally, using these definitions, our RGSW scheme can be defined, allowing the multiplication of ciphertexts. We need this ability to run our RLWE decryption while everything is encrypted under RGSW.

$$\text{RGSW}_s(m) = (\text{RLWE}'_s(-s * m), \text{RLWE}'_s(m)) \quad (3)$$

B. Introduction to Bootstrapping in FHEW

Bootstrapping means executing decryption operations homomorphically, i.e., doing the decryption operations with an encrypted secret key. This reduces the noise of the $\text{LWE}_s^{2/q}(\tilde{m})$ ciphertext. To bootstrap efficiently, the FHEW algorithm [2] only performs one NAND gate before bootstrapping or a similarly small binary gate operation. We go from $\text{LWE}_s^{4/q}(\tilde{m})$ to $\text{LWE}_s^{2/q}(\tilde{m})$, or in other words, from an LWE ciphertext deciphering to four possible messages with a maximum error of $q/16$ to a ciphertext deciphering to two possible messages with a maximum error of $q/8$ [3] as shown on Figure 1.

Ideally, we would like to continue working with this $\text{LWE}_s^{2/q}(\tilde{m})$ ciphertext. But if we add another LWE ciphertext to this result, our error would overflow, and we would no longer get the correct result back after decryption. Hence, we get rid of this noise via bootstrapping.

RGSW provides the encryption for the ciphertext. Encrypting our secret keys under RGSW allows us to safely send them to the server without breaking security by giving the server the means to decipher the LWE ciphertext into plain text. We call the secret key encrypted under RGSW the bootstrapping key.

Bootstrapping consists of 3 steps: Initialization, Accumulation, and Extraction (which simply consists of taking the first value of the second part of the RGSW result). The Accumulation, which consists of n RGSW$\times$RLWE$'$ $\rightarrow$ RLWE multiplications, is the bottleneck of the FHEW algorithm, and as such, the focus of our acceleration.

When we bootstrap, we will wish to use our secret key s encrypted under RGSW for multiplications. We can multiply an RGSW ciphertext $\text{RGSW}(m_1) = (\mathbf{c}, \mathbf{c'})$ with an RLWE' ciphertext $\text{RLWE}_s(m_0, e_0) = (a, b)$ and get a result that when decrypted gives the product of the 2 messages [6]:

$$(a, b) \diamond (\mathbf{c}, \mathbf{c'}) = \langle (a, b), (\mathbf{c}, \mathbf{c'}) \rangle = a \odot \mathbf{c} + b \odot \mathbf{c}' \quad (4)$$

$$= a \odot \text{RLWE'}_s(-s * m_1) + b \odot \text{RLWE'}_s(m_1) \quad (5)$$

$$= \text{RLWE}_s((b - a * s) * m_1) = \text{RLWE}_s((m_0 + e_0) * m_1) \quad (6)$$

n	q	B_r
512 elements	512 entries	23
LWE size	LWE modulus	RLWE' decomposition base

TABLE I
LWE PARAMETERS FOR STD128

N	$\log_2(Q)$	B_g
1024 elements	27 bits	128
Polynomial size	Data width (NTT)	RGSW base

TABLE II
RGSW PARAMETERS FOR STD128

This RGSW $\times$ RLWE$'$ $\rightarrow$ RLWE can be turned into a RGSW $\times$ RLWE$'$ $\rightarrow$ RLWE$'$. To summarize the accumulation process, each polynomial coefficient is broken down into B_g through a signed digit decomposition. Then a multiplication with an RLWE' vector occurs. As mentioned at the start of the section, using RLWE' instead of RLWE for the multiplication reduces the generated noise. Once the multiplication completes, the results are summed together. The Number Theoretic Transform is used to optimize the multiplications.

III. Hardware Architecture

The reference FHEW software implementation provided in the PALISADE library can execute one binary gate in 137 ms [6]. For many applications, the execution of 7 binary gates per second is too slow to be practical, especially when an entire CPU must be dedicated to one binary gate.

Since we attempt to accelerate this FHEW software implementation, we use the STD128 parameter set, for which timing results were available and which fit the parameters of the available NTT accelerators best. Generally, these accelerators are designed for a ring depth of $N = 1024$ and a modulus Q bit size of either 14 bits or around 32 bits. The parameters for which our hardware implementation was designed are given in Table I.

A. The Inner Control Loop

The FHEW algorithm requires $n = 512$ iterations which can be broken up into 4 parts: two Inverse Number Theoretic Transforms (INTTs), Signed Digit Decomposition, eight ($= 2 * d_g$) Number Theoretic Transforms (NTTs), and RGSW bootstrapping key multiplication. Of these, the NTTs and INTTs are by far the most time-critical. The hardware accelerator is therefore based on a design for accelerating NTTs (see Mert et al. [5]).

This highly parametric design allows 3 parameters to be set: the Ring Size N, the Modulus Size Q, and the number of Processing Elements PE. The ring and modulus sizes are found in the parameter set STD128 (see Table I). The number of processing elements determines the speed and, to a degree, the area of the design.

For this implementation, we chose a PE number of 32. Figure 2 shows the original datapath of one Processing Element. Based on this design, we created our hardware accelerator (Figure 3). The Gentleman-Sande (GS) INTT block contains

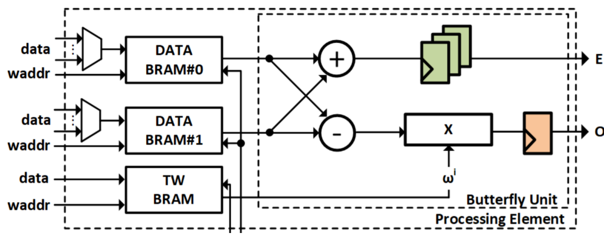

Fig. 2. Mert et al.'s datapath [5]

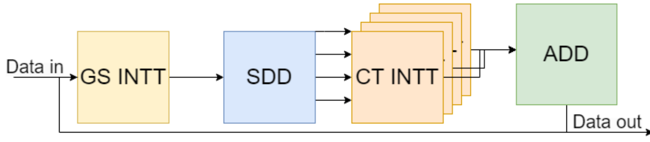

Fig. 3. Diagram of the RGSW accumulator implementation, with a Gentleman-Sande INTT, a Signed Digit Decomposition, a Cooley-Tukey NTT and an addition of the 4 CT NTT results

32 instances of Figure 2, while the CT NTT block contains 32 instances of a modified datapath for the Cooley-Tukey NTT.

Our implementation extends the work of Mert et al. [5]. We add hardware support for bit-reversal of the indices (see Figure 4), signed digit decomposition, and multiplication with the bootstrapping key (see Figure 5). We describe our modifications in the next sections.

B. Overview of Index Bit-Reversal

Index bit-reversal introduces read and write access conflicts. The difficulty with the bit-reversing is the organization of the memory because each BRAM is instantiated with only one read port and one write port. This means we must carefully consider which values are swapped in memory as an index bit-reverse operation is a series of swaps. We cannot swap two or more values residing in the same BRAM in the same clock

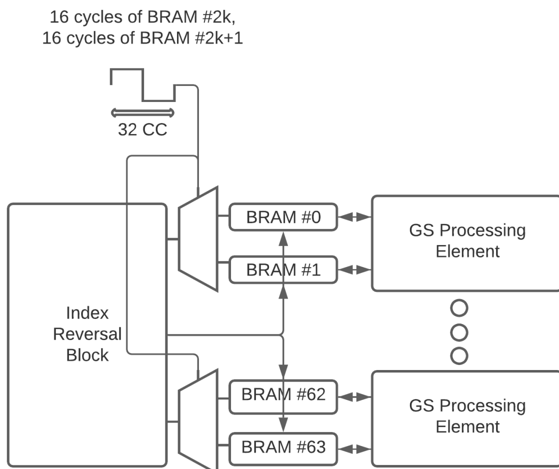

Fig. 4. Bitreversal in the GS INTT

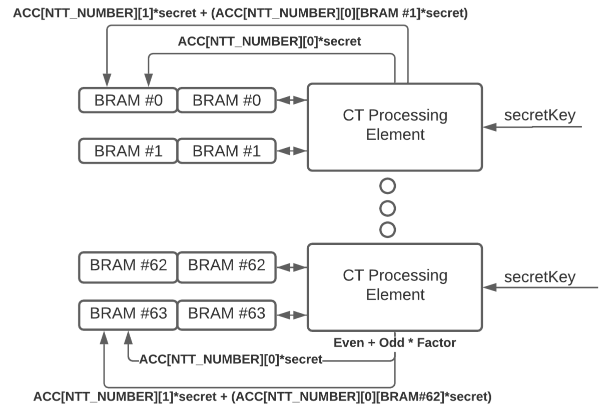

Fig. 5. Bootstrapping key accumulation integrated into CT

cycle, as this would require two reads from the same BRAM in one clock cycle. Note that BRAMs can be instantiated with two write and read ports, which requires doubling their size from 18 Kbit to 36 Kbit. It would not simplify the bit-reversal problem, although it would allow bit-reversal in half the time.

We thus used conflict-free address scheduling combined with a crossbar to ensure maximum throughput to 32 BRAMs. This allows us to process our polynomial in $\frac{N}{32} = \frac{1024}{32} = 32$ clock cycles. The bit-reversal step at the start and the bit-reversal step at the end, plus the integrated multiplication by $\frac{1}{N}$ required for the INTT, adds a total of 80 cycles to the INTT run time of 250 cycles.

C. Non-NTT Components

The Signed Digit Decompose is implemented almost verbatim from the PALISADE library and is placed between the readout from the INTT BRAMs and the NTT BRAMs. The bootstrapping key multiplication is integrated into the CT NTT. Because we run both parts of the RGSW accumulator on the same datapath (multiplexed in time), the BRAMs of the CT NTT are extended to fit in the results of the bootstrapping key multiplication with the first part until the second part is calculated and is summed together with the first part inside the CT NTT datapath (Figure 5). As previously mentioned, Mert et al. under-utilizes the BRAM resources, so this does not increase the area taken up by the memory [9].

The full datapath has been tested in simulation and works for test vectors generated by the PALISADE library.

IV. Memory Interface

The next step is to map the architecture onto the Alveo U280 FPGA platform while considering the specifics of the U280 memory architecture. To perform the multiplication with the bootstrapping key in time with the CT NTT, we must be able to fetch $N * d_r * 2 = 8096$ elements in the ≈ 900 cycles at 100 MHz it takes to complete half of one accumulator step. In other words, we would need a memory BW of

$$\frac{8 * N * f * \log_2(Q)}{\text{CC 1/2 of Accumulation}} = \frac{8096 * 100 * 10^6 * 27}{900} = 24.3\text{Gbps} \tag{7}$$

Task	Input	**ACC**$+ = a_i * s_i$	Output	1/16 Bootstrap
Time	2052 CC	3616 CC	2049 CC	114327 CC

TABLE III
SIMULATION RUN TIME

Frequency	WNS	LUT	FF	BRAM	DSP
100MHz	1.595 ns	133971	56565	146	768

TABLE IV
IMPLEMENTATION RESULTS FOR 1 INTT AND 2 NTT RUN

LUT	FF	BRAM	DSP
223285	94275	240	1280

TABLE V
EXTRAPOLATED IMPLEMENTATION RESULTS FOR FULL IMPLEMENTATION

This data rate is achieved by storing the key in the HBM memory and then building 16 256-bit AXI interfaces to communicate between the FPGA and the HBM memory [8].

Storing the entire bootstrapping key on the U280 FPGA would not be possible at all, as it takes up a total of [6]:

$$4nNd_rB_rd_g \log_2(Q) = 10.4\text{Gigabits} = 1.3\text{GB} \quad (8)$$

The bootstrapping key exceeds the maximum of 32.1 Megabits that can be generated [8].

We verified the correctness of the computation blocks. In simulation, we used bootstrapping keys of the correct size to verify correctness. In synthesis, we preferred a bootstrapping key hardwired to the design instead of reading from an external HBM memory. Because the multiplier used was the modular multiplier from the CT NTT, any bootstrapping key used in our design must be multiplied with the same factor R as our twiddle factors. Beyond that, the data in the bootstrapping key must be structured so that each BRAM receives the correct bootstrapping key at the correct clock cycle.

V. RESULTS

A. Simulation Results

Because the bootstrapping key is over 10 Gigabit, we decided only to run part of the bootstrapping process in simulation, with only 1/16th completed. The result of this simulation corresponded with the execution of 1/16th of the bootstrapping algorithm in PALISADE. The simulation gave us the results in Table III for our processing speed. When extrapolating this to a full bootstrap, which consists of n $(= 512)$ times adding $a_i * s_i$, we see that the full bootstrapping process executes in 1855493 clock cycles (CC). The full bootstrapping process has a runtime of less than 18.5 ms, assuming a clock frequency of at least 100 MHz.

B. Implementation Results

An HBM interface was created and tested in hardware but could not write the full bootstrapping key to the FPGA. As such, we could not verify the full functionality of the implementation with the HBM interface.

When we scale our implementation to 2 NTTs and 1 INTT, we receive the results described in Table IV. Suppose we extrapolate these results with the synthesis results on a smaller prototyping device. The whole design (minus the bootstrapping key memory) would use the resources described in Table V. The limitation to verifying the whole design is the High Bandwidth Memory requirements, which require significant design time to get right.

VI. CONCLUSION

Bootstrapping is the bottleneck of Fully Homomorphic Encryption for schemes like FHEW. Acceleration via FPGAs or other hardware is a promising way to improve the throughput of homomorphic operations. Our current design promises a speed-up of 7.5 times over the software implementation when run at 100 MHz. As the NTT core, which makes up most of our design, can be run at 200 MHz, we project that a speed-up by a factor of 15 is achievable. FHEW has large memory bandwidth requirements and is only computation limited when full advantage is taken of the HBM bandwidth. If the designer cannot use High Bandwidth Memory, the memory bandwidth of the FPGA limits the performance of the design.

ACKNOWLEDGMENT

This work was supported in part by CyberSecurity Research Flanders with reference number VR20192203, the Horizon 2020 ERC Advanced Grant (101020005 Belfort) and by Darpa DPRIVE (Contract No. HR0011-21-C-0034). Michiel Van Beirendonck is funded by FWO PhD fellow (1SD5621N).

REFERENCES

[1] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "TFHE: fast fully homomorphic encryption over the torus," *J. Cryptol.*, vol. 33, no. 1, pp. 34–91, 2020. [Online]. Available: https://doi.org/10.1007/s00145-019-09319-x

[2] L. Ducas and D. Micciancio, "Fhew: Bootstrapping homomorphic encryption in less than a second," in *Advances in Cryptology – EUROCRYPT 2015*, E. Oswald and M. Fischlin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 617–640.

[3] ——, "Fhew: Homomorphic encryption bootstrapping in less than a second," University Lecture, 2015.

[4] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005, special issue on "Program Generation, Optimization, and Platform Adaptation".

[5] A. C. Mert, E. Karabulut, E. Ozturk, E. Savas, and A. Aysu, "An extensive study of flexible design methods for the number theoretic transform," *IEEE Transactions on Computers*, pp. 1–1, 2020.

[6] D. Micciancio and Y. Polyakov, "Bootstrapping in fhew-like cryptosystems," in *WAHC '21: Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography, Virtual Event, Korea, 15 November 2021*. WAHC@ACM, 2021, pp. 17–28. [Online]. Available: https://doi.org/10.1145/3474366.3486924

[7] H.-T. Peng, W. W. Hsu, J.-M. Ho, and M.-R. Yu, "Homomorphic encryption application on financialcloud framework," in *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, 2016, pp. 1–5.

[8] *Alveo U280 Data Center Accelerator Card*, Xilinx, 11 2019.

[9] *UltraScale Architecture Memory Resources*, Xilinx, 3 2021, block Ram Summary.