An FPGA-based Programmable Vector Engine for Fast Fully Homomorphic Encryption over the Torus

Y. Serhan Gener*, Parker Newton*, Daniel Tan*, Silas Richelson*, Guy Lemieux†, and Philip Brisk*

*Department of Computer Science and Engineering, University of California, Riverside
†Department of Electrical and Computer Engineering, University of British Columbia

***Abstract*—This paper describes an FPGA-based vector engine to accelerate the bootstrapping procedure of Fast Fully Homomorphic Encryption over the Torus (TFHE), a popular and high-performance fully homomorphic encryption scheme. Most TFHE bootstraping comprises many matrix-vector operations that are implemented using Torus polynomials, which are not efficiently implemented on today's standard arithmetic hardware. Our implementation achieves linear performance scaling with up to 16 vector lanes. Future work will switch to an FFT-based polynomial multiplication scheme and switch to larger FPGA parts to accommodate more vector lanes.**

I. Introduction

Fully Homomorphic encryption (FHE) is a cryptographic primitive which enables computing arbitrary functions on encrypted data [7], [9], [11], [12], [20]–[22], [32]. Applications of FHE include such as privacy-preserving inference on neural networks [5], [6], [16], [23] and private analysis of genomic data [4], [24], [25]. The client encrypts their private data using FHE and transmits the ciphertext to an untrusted server. The server then homomorphically computes the desired function on the encrypted data, obtaining a ciphertext of the output. The client then decrypts the ciphetext to obtain the result.

Many FHE schemes rely on a ciphertext refreshing procedure called *bootstrapping* [2], [14], [19]. Each homomorphic operation introduces error into the ciphertext; eventually, the error will grow too large to be decrypted correctly. Bootstrapping the ciphertext reduces its error to a fixed value that can be tolerated. Bootstrapping has emerged as the computational bottleneck of the FHE schemes that employ it.

This paper presents a preliminary hardware accelerator for the bootstrapping procedure employed in *Fast Fully Homomorphic Encryption over the Torus (TFHE)* [14]. We observe that the computational bottleneck of TFHE is a polynomial multiplication procedure which is repeatedly called within TFHE's bootstrapping procedure. Our accelerator adds TFHE-specific custom instruction extensions to an FPGA-based programmable vector engine. We observe linear performance scaling as the number of vector lanes increases from 4 to 16; however, our overall performance is limited compared to state-of-the-art software and GPU-accelerated TFHE implementations. The key difference is that our polynomial multiplier uses an $O(n^2)$ algorithm, which could be reduced to $O(nlogn)$ using a *Fast Fourier Transform (FFT)*. We plan to address this shortcoming in future work while switching to larger FPGA parts to enable evaluation using additional vector lanes.

II. Related Work

FPGA accelerators for the CKKS [28] and BFV [29], [33] FHE schemes have been published recently. CKKS performs approximate computation on reals [13], while BFV performs integer computation exclusively [3], [8], [20]. In contrast, TFHE is universal, as it homomorphically evaluates logic gates, which can construct both exact and approximate circuits [14]. We acknowledge earlier FHE hardware accelerators (e.g., Refs. [17], [18]), but eschew discussion to conserve space.

III. Background

Notation: We denote the output x of algorithm $\mathcal{A}$ by $x \leftarrow \mathcal{A}$. We denote sampling an element x from a distribution $\mathcal{D}$ by $x \leftarrow \mathcal{D}$. Let S be a set. We denote sampling an element s from the uniform distribution on S by $s \leftarrow S$. If $n \in \mathbb{Z}_{>0}$, then we denote the set of vectors of dimension n with elements in S by S^n. If $\mathbf{v} \in S^n$, then we write the ith component of $\mathbf{v}$ as $v_i \in S$. If $q \in \mathbb{Z}_{>0}$, then let $\mathbb{Z}_q$ be the ring of integers modulo q, and let $\mathbb{Z}_q[X]_{<N}$ denote the ring of polynomials over $\mathbb{Z}_q$ of degree less than N. If $x \in \mathbb{R}$, then we say that x modulo 1 is $x \pmod 1 = x - \lfloor x \rfloor \in [0, 1)$. Let $\mathbb{T} = \mathbb{R}/\mathbb{Z}$ be the Torus, a ring which is isomorphic to $[0, 1)$[1]. We denote the rings of polynomials over the Torus of degree less than N by $\mathbb{T}_N[X]$.

A. TFHE Overview

TFHE evaluates an arbitrary function f by first decomposing it into a Boolean circuit. TFHE then homomorphically evaluates each logic gate in the circuit, composing them until the entire function is evaluated. The output is the encrypted (ciphertext) result of the function applied to the original encrypted (ciphertext) input message. Plaintext inputs are messages $m_1, m_2 \in \{0, 1/4\}$, which lie on the Torus.

TFHE employs *Learning with Errors (LWE)* Encryption to encode plaintext messages (scalars) into a vector. LWE Encryption's semantic security derives from the conjectured hardness of the LWE lattice problem [10], [26], [27].

Let χ be an error distribution on $\mathbb{T}$ with the property that sample $e \leftarrow \chi$ has low norm[2] with high probability. LWE Encryption generates a symmetric key vector $\mathbf{s} \leftarrow \mathbb{Z}_2^n$, where $n \in \mathbb{Z}_{>0}$ is a security parameter. Message $m \in \mathbb{T}$ is encrypted by randomly choosing $\mathbf{a} \leftarrow \mathbb{T}^n, e \leftarrow \chi$ and generating an LWE

[1]The ring operations in $[0, 1)$ are addition and multiplication modulo 1.
[2]Typically, χ is a discrete Gaussian distribution on $\mathbb{T}^n$.

TABLE I: Homomorphic logic gates under TFHE.

Gate	Homomorphic Evaluation
NOT	$(\mathbf{0}, 1/4) - \mathbf{c_1}$
NAND	Bootstrap($(\mathbf{0}, 5/8) - \mathbf{c_1} - \mathbf{c_2}$)
AND	Bootstrap($(\mathbf{0}, -1/8) + \mathbf{c_1} + \mathbf{c_2}$)
OR	Bootstrap($(\mathbf{0}, 1/8) + \mathbf{c_1} + \mathbf{c_2}$)
XOR	Bootstrap($2 \cdot (\mathbf{c_1} - \mathbf{c_2})$)

ciphertext (a vector) $\mathbf{c} = (\mathbf{a}, b) \in \mathbb{T}^{n+1}$, where $b = \langle \mathbf{a}, \mathbf{s} \rangle + e + m$, to obscure m. To decrypt an LWE ciphertext $\mathbf{c}$, the secret key $\mathbf{s}$ must be known. Then $z = b - \langle \mathbf{a}, \mathbf{s} \rangle = e + m \in \mathbb{T}$. Recall that e was sampled from an error distribution χ such that e has low norm. It follows that we can round off e and decode $e + m$ back to m; see [26], [27] for details.

Let $\mathbf{c}_1$ and $\mathbf{c}_2$ be LWE ciphertexts for m_1 and m_2. The homomorphic addition operation computes $\mathbf{c}_3 = \mathbf{c}_1 + \mathbf{c}_2 = (\mathbf{a}_1 + \mathbf{a}_2, b_3) \in \mathbb{T}^{n+1}$, where $b_3 = \langle \mathbf{a}_1 + \mathbf{a}_2, \mathbf{s} \rangle + (e_1 + e_2) + (m_1 + m_2)$. $\mathbf{c}_3$ then decrypts to $(e_1 + e_2) + (m_1 + m_2) \in \mathbb{T}$, which, by using the low norm of e_1, e_2, we can round and decode to $(m_1 + m_2) \in \mathbb{T}$.

Homomorphic addition can implement basic logic gates (see Table I); however, each operation increases the error $(e_1 + e_2)$. Eventually, this error will grow too large to be decrypted to the correct value. As mentioned in the Introduction bootstrapping can refresh the error to a tolerable level. Bootstrapping applies encrypted ciphertexts of the secret key to homomorphically evaluate the decryption algorithm of an FHE scheme, refreshing the ciphertext of the encrypted gate output, so that the resulting error depends on the depth of the decryption circuit.

The Appendix describes TFHE in greater detail.

IV. Bootstrapping Accelerator Architecture

We implemented an accelerator for the TFHE bootstrap function using an FPGA-based programmable vector engine [31] (MXP) that can be customized [30] with up to 16 application-specific instruction set extensions. TFHE involves a mixture of vector-vector and matrix-vector operations; while some of the matrices involved are sparse, they have a well-defined structure with dense sub-matrices and other sub-matrices where all values are 0. This lends itself to regular access patterns and SIMD parallelism. While we employ an FPGA for prototyping, and tuned the performance of the system as-deployed, we anticipate that the design principles of the vector accelerator will readily transfer to standard-cell CMOS technology.

MXP is implemented through extensions to a scalar processor, which could be either a hard CPU integrated into the FPGA fabric, or a soft CPU synthesized on programmable logic. MXP has a dedicated scratchpad memory, whose size ranges from roughly 4kB to 2MB, supported by specialized DMA read and write options. Scalar, vector, and DMA operations can execute concurrently. MXP executes multi-cycle instructions on vectors of arbitrary length. The number of vector lanes (m) in hardware is configurable and determines

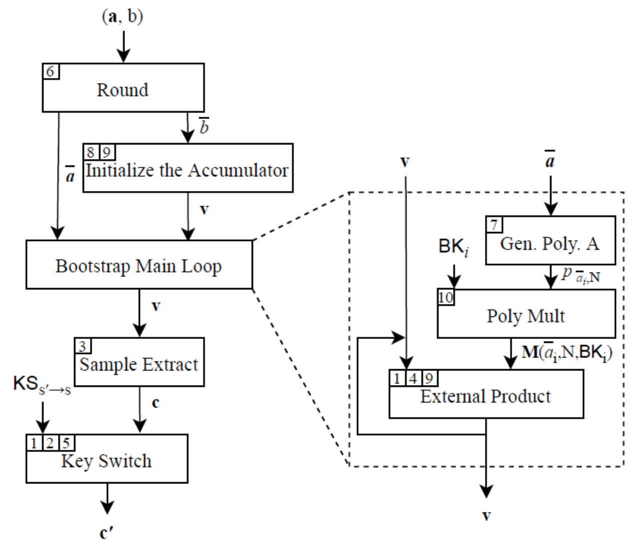

Fig. 1: TFHE bootstrap accelerator; the numbers in the upper left-hand corner of each module correspond to the vector instruction set extensions listed in Table II.

the amount of parallelism; in software, the programmer independently chooses a vector length of convenience and lets the hardware scheduler determine how many m-wide micro-operations, aka wavefronts, to issue to complete each instruction. MXP is programmed in C/C++. Vector operations are specified as non-blocking function calls initiated by the scalar CPU. The hardware scheduler handles pipelining, interlocks, and out-of-order execution of vector and DMA operations.

Figure 1 depicts an architectural specification for a TFHE Bootstrap accelerator. The modules in Figure 1 correspond to the specification in Algorithm 1. The input is a LWE ciphertext $(\mathbf{a}, b)$, switching key $\mathsf{KS}_{\mathbf{s}' \to \mathbf{s}}$, and a set of n bootstrapping keys $\{\mathsf{BK}_i\}_{i=1}^{n}$. The output is a noise-reduced LWE ciphertext $\mathbf{c}'$.

We implemented ten custom vector instructions for TFHE, listed in Table II; some require multiple invocations to complete larger-scale operations. We implemented each block in Figure 1 using a mix of scalar and vector engine instructions; this yields a more compact design than implementing each block in RTL. Each module shown in Figure 1 lists the custom vector instruction set extensions that it uses.

TFHE transforms the LWE ciphertext into a Torus LWE (TLWE) ciphertext, in which vector elements are Torus polynomials, whose coefficients lie in the range $[0, 1)$. We represent these values in a 32-bit fixed-point format, with a 1-bit integer component and 31 fractional bits. The custom vector instructions perform modulo-1 arithmetic operations, which correspond to standard operations on the Torus ring.

Instructions 1-3 perform addition, subtraction, and 2's complement negation (modulo 1) on vectors of Torus coefficients.

Instructions 4-5 perform decomposition for the external product and key switch functions. Decomposition extracts ℓ (external product) or t (key switch) MSBs from each value in a length-N vector of scalars and, without loss of generality, writes each bit to a sequence of length-N bit-vectors,

TABLE II: Custom vector instruction set extensions for TFHE.

	Instruction Name	Short Description	Mathematical Representation	Invocations to Complete
1	add_mod1	Addition under modulo 1	$(\mathbf{x}+\mathbf{y})(\text{mod } 1)$	1
2	sub_mod1	Subtraction under modulo 1	$(\mathbf{x}-\mathbf{y})(\text{mod } 1)$	1
3	neg_mod1	Two's compliment under modulo 1	$(\sim \mathbf{x}+1)(\text{mod } 1)$	1
4	Decompose_ep	Extract the ℓ MSBs from each vector element	See Fig. 2	l
5	Decompose_ks	Extract the t MSBs from each vector element	See Fig. 2	t
6	Round	Isolate and round the $(\log_2(2N)+1)$ MSBs	$\lfloor x_{[n-1:n-(\log_2(2N)+1)]} \rceil$	1
7	Gen. Poly. A	Generate polynomial as per. Eq. (2)	$p_{\bar{a}_i,N}(X) : \mathbb{Z}^2 \rightarrow \mathbb{Z}[X]/(X^N+1)$	1
8	Gen. Poly. B	Generate polynomial as per Eq. (3)	$p_{\bar{b},N}(X) : \mathbb{Z}^2 \rightarrow \mathbb{T}[X]/(X^N+1)$	1
9	Binary Poly. Mult	Polynomial multiplication; one operand has binary coefficients	$(\mathbf{x} * \mathbf{y}) \in \mathbb{T}[X]/(X^N+1), x_i \in \{0,1\}$	N
10	Poly. Mult	Polynomial multiplication; no operand restrictions	$(\mathbf{x} * \mathbf{y}) \in \mathbb{T}[X]/(X^N+1)$	$N+1$

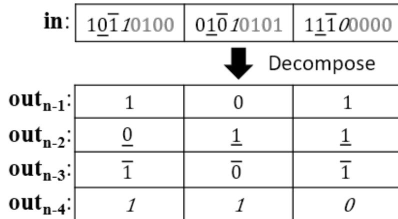

Fig. 2: Illustration of Decompose instruction for $\ell = 4$

$\mathbf{out}_{n-1}, \ldots, \mathbf{out}_{n-\ell}$, as shown in Figure 2. MXP supports one output per vector lane, requiring repeated invocations (ℓ or t times) to obtain all of the bits: the input vector is always the same, while each invocation extracts a different bit position. Both instructions share the same datapath logic but have slightly different control circuitry.

Instruction 6 isolates and rounds the $\log_2(2N)+1$ highest-order bits of an integer. If the $\log_2(2N)+1^{st}$ bit is 1, then the remaining $\log_2(2N)$ higher-order bits are incremented. The rounded LWE ciphertext is denoted $(\bar{\mathbf{a}}, \bar{b})$.

Instruction 7 transforms a rounded integer into an integer polynomial in $\mathbb{Z}[X]/(X^N+1)$; it is called once for each rounded integer $\bar{a}_i \in \bar{\mathbf{a}}$.

Instruction 8 transforms a rounded integer into a Torus polynomial ($\mathbb{T}[X]/(X^N+1)$); it is called once for $\bar{b}$.

Instructions 9-10 compute the product of two Torus polynomials $\mathbf{x}$ and $\mathbf{y}$; the former is a special case where $\mathbf{x}$ satisfies the property that the LSB of each coefficient x_i is binary (0 or 1) and all higher-order bits are zero; in this case, multiplication reduces to a bitwise-AND operation.

A. Operation Schedule

All $n+1$ integers in the LWE ciphertext $(\mathbf{a}, b)$ are rounded upfront. Within the Bootstrap Main Loop, the polynomials derived from $\mathbf{a}$ (denoted $p_{\bar{a}_i,N}(X)$) are processed independently; however, the scratchpad memory is not large enough to store all of the polynomials. To avoid excess DMA traffic, the scheduler generates the polynomials on-the-fly as needed, noting that the size of the polynomial $p_{\bar{a}_i,N}(X)$ is much larger than that of the rounded integer value $\bar{a}_i$.

B. Polynomial Multiplication

Within the Bootstrapping Main Loop (Fig. 1), each external product invocation executes polynomial addition and multiplication $\ell(k+1)^2$ times, where ℓ and k are security parameters; this dominates the execution time of bootstrapping.

In TLWE, each polynomial has degree $N-1$. The product of two $(N-1)$-degree polynomials is a $2(N-1)$-degree polynomial; however, the Torus polynomial ring is defined modulo-(X^N+1), so a modular reduction is necessary.

Let $\mathbf{x}$ and $\mathbf{y}$ be the coefficient vectors of two degree $N-1$ polynomials and let $\mathbf{z} = \mathbf{xy}$ be the coefficient vector of their product. One way to perform the reduction is to first compute a $2(N-1)$ degree polynomial, $\mathbf{z}'$, and then compute $\mathbf{z}$ as follows: $z_i = z'_i - z'_{i+N}$, $0 \leq i \leq N-1$. An alternative is to compute the coefficients of $\mathbf{z}$ directly:

$$z_r = \sum_{i+j=r} x_i * y_j - \sum_{i+j=r+N} x_i * y_j, \quad 0 \leq r \leq N-1 \quad (1)$$

Table III and Figure 3 show an example multiplying two degree-3 ($N=4$) polynomials. Each row of Table III represents the scalar multiplication of coefficient x_i with the full coefficient vector (y_0, y_1, y_2, y_3) of $\mathbf{y}$. In Figure 3, coefficients of $\mathbf{x}$ are stored in flip-flops, coefficients for $\mathbf{y}$ and $\mathbf{z}$ are in vectors, and a sequence of add/subtract control signals is stored in a shift register. The first invocation loads the values of $\mathbf{x}$ into the flip-flops; a copy of $\mathbf{y}$ is appended to itself (in software) and loaded into the scratchpad memory. The rotation pattern shown in Table III is implemented by first calling the custom instruction with starting index N, and decrementing the index prior to each subsequent invocation. The shift register to control add/subtract operations is initialized to **1**: between invocations, it shifts left by 1, shifting a zero into the LSB. Polynomial multiplication completes after N invocations.

For Binary Polynomial Multiplication, each coefficient of $\mathbf{x}$ can be represented with a single bit: the LSB is either 0 or 1,

TABLE III: Polynomial multiplication reordered.

z_0	z_1	z_2	z_3
$x_0 * y_0$	$x_0 * y_1$	$x_0 * y_2$	$x_0 * y_3$
$x_1 * y_3$	$x_1 * y_0$	$x_1 * y_1$	$x_1 * y_2$
$x_2 * y_2$	$x_2 * y_3$	$x_2 * y_0$	$x_2 * y_1$
$x_3 * y_1$	$x_3 * y_2$	$x_3 * y_3$	$x_3 * y_0$

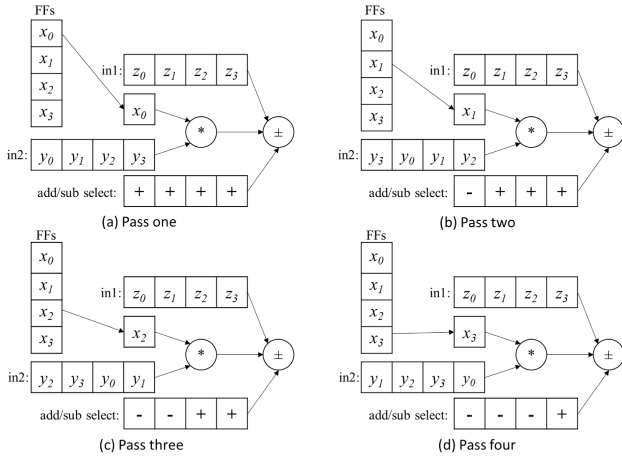

Fig. 3: Polynomial multiplication for $N = 4$

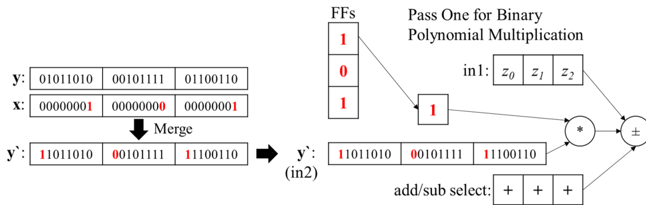

Fig. 4: Merge two inputs and save 1-bit vector in BPM's FFs for $N = 3$

and all higher-order bits are zero. Since each coefficient y_i of $\mathbf{y}$ is in the range $[0, 1)$, the MSB of y_i is always zero. As shown in Figure 4, the two inputs can be merged by packing the binary value into the MSB of y_i; the updated vector of packed coefficients is denoted $\mathbf{y}'$. The packed MSBs are copied to the flip-flops used in the polynomial multiplier unit, prior to performing the operation; a bitwise-AND operation on the 31 least significant bits of y'_i implements the one-bit multiply; the highest-order (packed) bit is forced to 0. The more general polynomial multiplication instruction is used exclusively in the Bootstrap Main Loop to compute $p_{\bar{a}_i,N}(X) \times \mathsf{BK}_i$.

V. Experiments

This section evaluates the performance of the TFHE vector engine [30], [31] (MXP) which we implemented on a Digilent ZedBoard Zynq-7000 ARM/FPGA SoC Development Board. The TFHE software runs on one ARM Cortex-A9 core in the FPGA, leveraging the programmable MXP vector engine for acceleration; MXP and ten TFHE-specific custom instructions (Table II) are synthesized on the programmable logic of the FPGA. We configured the vector engine to use 8 parallel lanes for computation and 64kB of scratchpad memory; a wider vector engine would obtain higher performance, but would require a larger and more expensive FPGA.

Our evaluation compares four different implementations. We executed a GPU-based implementation of TFHE on an Nvidia GeForce RTX 2080 Ti (cuFHE) [1], the TFHE C/C++ reference implementation [15] on one core of an Intel Core i7-8750H @2.20GHz CPU (TFHE-x86) and on one ARM Cortex-A9 core on the FPGA (TFHE-ARM), and lastly using the 8-lane MXP vector accelerator with the ten custom instructions (MXP-VL8). Table IV reports the runtime of the TFHE bootstrap algorithm on all four implementations. These results indicate orders of magnitude performance difference between the four implementations, with the cuFHE achieving the highest performance and MXP-VL8 achieving the lowest. While our current MXP implementation is not yet competitive with software, we believe that future work can narrow the gap by switching to an FFT-based polynomial multiplier within the external product (**Instruction 10**) and transitioning to a higher-capacity FPGA device to accommodate more vector lanes and, if needed, a larger scratchpad memory.

TABLE IV: Runtime of all architectures.

Arch.	Bootstrap	Cost (USD)
cuFHE	0.16 ms	$1199 (GPU only)
TFHE-x86	34.9 ms	$395 (CPU only)
TFHE-ARM	1.73 s	$449 (Dev. board)
MXP-VL8	17.64 s	$449 (Dev. board)

TABLE V: Single polynomial multiplication runtime.

Implementation-Architecture	Runtime (ms)
TFHE-Naive-x86	0.312
TFHE-Karatsuba-x86	0.143
TFHE-FFTW-x86	0.018
TFHE-FFT-AVX-x86	0.007
TFHE-Naive-ARM	6.734
TFHE-Karatsuba-ARM	2.255
TFHE-FFT-ARM	0.330
MXP-VL4	2.636
MXP-VL8	1.324
MXP-VL16	0.668

Table IV also reports the cost of each of the platforms; for cuFHE and TFHE-x86, we only report the cost of the GPU and CPU parts respectively, omitting the full system (e.g., memory, storage, motherboard, chassis, etc.); for TFHE-ARM and MXP-VL8, we report the cost of the Zedboard, which is fully integrated and requires no additional hardware. Looking at the costs underscores the point that the MXP-VL8 results are reported for a relatively low-cost and low-capacity FPGA part, and that additional performance gains are expected after it becomes possible to port to a higher-capacity FPGA.

Table V reports the runtime of different implementations of TFHE polynomial multiplication based on the reference software [15]. The implementations labeled Naive perform polynomial multiplication as described in Eq. (1); those labeled Karatsuba and FFT employ divide-and-conquer to accelerate polynomial multiplication. The highest performance on the x86 CPU is achieved using a non-portable FFT library written to use AVX instructions. The FFT libraries that TFHE employs were not easily portable to the ARM CPU, so we implemented our own FFT software routine, which ran faster than Karatsuba. Implementing MXP with a single custom instruction for polynomial multiplication enabled us to scale the number of vector lanes as high as 16. Doubling the

TABLE VI: Custom instruction resource utilization.

Custom Instruction	LUT	FF
Generate Polynomial A	5,055	33
Generate Polynomial B	3,941	32
Binary Polynomial Mult.	1,965	2,093
Polynomial Mult.	2,491	3,123
All Other Instructions	944	19

number of vector lanes approximately halves the execution time, indicating that our current implementation is compute-bound for these vector widths.

Table VI reports the resource utilization of the custom instructions added to the MXP configured with 8 parallel vector lanes. None of the instructions require BRAMs beyond those already allocated to the MXP's scratchpad. The polynomial generation and multiplication instructions dominate the resource requirements of the other six instructions. Altogether, the vector engine (including custom instructions) uses 36,373 LUT slices (68%), 23,815 FFs (22%), 82.5 BRAMs (59%), and 40 DSP blocks (18%). Greater performance can be obtained by increasing the number of vector lanes (and the widths of the custom instructions), but doing so would require a larger FPGA device. There is ample room for growth, as the largest Zynq UltraScale+ device (ZU19EG) is about 10 times larger than the one we used. Additionally, an FFT-based implementation of polynomial multiplication may have considerably different implementation requirements.

VI. Conclusion and Future Work

This paper presented a preliminary architecture to accelerate TFHE bootstrapping. The architecture is based on a customizable vector engine (MXP) that can be augmented with application-specific custom instructions. The custom instructions that we provided perform modular arithmetic operations on Torus polynomials, and benefit from fast access to the MXP's scratchpad memory which is filled through DMA transfers from off-chip memory.

There are several immediate directions for future work. The first is to replace our current polynomial multipliers with FFT-based designs; the second is to port MXP to a larger FPGA so that we can increase the number of vector lanes; the third is to build an FPGA-based TFHE server, so that we can measure the performance of the accelerator in the context of network latencies, including all requisite data transfers (LWE ciphertexts, secret key material, etc.). Another direction is to specify each module in Figure 1 in RTL, rather than relying on a programmable solution. Lastly, we plan to implement a compiler to enable TFHE evaluation of multi-gate logic circuits and measure their performance.

Acknowledgements

The authors would like to thank the anonymous referees for their detailed reviews and suggestions. This material is based upon work supported by the National Science Foundation under Grant No. 1528181, Grant No. 1545097, Grant No. 1763795 and an NSERC Discovery Grant.

References

[1] Cuda-accelerated fully homomorphic encryption library. [Online]. Available: https://github.com/vernamlab/cuFHE

[2] J. Alperin-Sheriff and C. Peikert, "Faster bootstrapping with polynomial error," in *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I*, ser. Lecture Notes in Computer Science, J. A. Garay and R. Gennaro, Eds., vol. 8616. Springer, 2014, pp. 297–314. [Online]. Available: https://doi.org/10.1007/978-3-662-44371-2_17

[3] A. A. Badawi, Y. Polyakov, K. M. M. Aung, B. Veeravalli, and K. Rohloff, "Implementation and performance evaluation of RNS variants of the BFV homomorphic encryption scheme," *IACR Cryptol. ePrint Arch.*, vol. 2018, p. 589, 2018. [Online]. Available: https://eprint.iacr.org/2018/589

[4] M. Blatt, A. Gusev, Y. Polyakov, and S. Goldwasser, "Secure large-scale genome-wide association studies using homomorphic encryption," *IACR Cryptol. ePrint Arch.*, vol. 2020, p. 563, 2020. [Online]. Available: https://eprint.iacr.org/2020/563

[5] C. Boura, N. Gama, M. Georgieva, and D. Jetchev, "Simulating homomorphic evaluation of deep learning predictions," in *Cyber Security Cryptography and Machine Learning - Third International Symposium, CSCML 2019, Beer-Sheva, Israel, June 27-28, 2019, Proceedings*, ser. Lecture Notes in Computer Science, S. Dolev, D. Hendler, S. Lodha, and M. Yung, Eds., vol. 11527. Springer, 2019, pp. 212–230. [Online]. Available: https://doi.org/10.1007/978-3-030-20951-3_20

[6] F. Bourse, M. Minelli, M. Minihold, and P. Paillier, "Fast homomorphic evaluation of deep discretized neural networks," in *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part III*, ser. Lecture Notes in Computer Science, H. Shacham and A. Boldyreva, Eds., vol. 10993. Springer, 2018, pp. 483–512. [Online]. Available: https://doi.org/10.1007/978-3-319-96878-0_17

[7] Z. Brakerski, "Fully homomorphic encryption without modulus switching from classical gapsvp," in *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, ser. Lecture Notes in Computer Science, R. Safavi-Naini and R. Canetti, Eds., vol. 7417. Springer, 2012, pp. 868–886. [Online]. Available: https://doi.org/10.1007/978-3-642-32009-5_50

[8] Z. Brakerski, "Fully homomorphic encryption without modulus switching from classical gapsvp," in *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, ser. Lecture Notes in Computer Science, R. Safavi-Naini and R. Canetti, Eds., vol. 7417. Springer, 2012, pp. 868–886. [Online]. Available: https://doi.org/10.1007/978-3-642-32009-5_50

[9] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "Fully homomorphic encryption without bootstrapping," *IACR Cryptol. ePrint Arch.*, vol. 2011, p. 277, 2011. [Online]. Available: http://eprint.iacr.org/2011/277

[10] Z. Brakerski, A. Langlois, C. Peikert, O. Regev, and D. Stehlé, "Classical hardness of learning with errors," in *Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013*, D. Boneh, T. Roughgarden, and J. Feigenbaum, Eds. ACM, 2013, pp. 575–584. [Online]. Available: https://doi.org/10.1145/2488608.2488680

[11] Z. Brakerski and V. Vaikuntanathan, "Efficient fully homomorphic encryption from (standard) LWE," in *IEEE 52nd Annual Symposium on Foundations of Computer Science, FOCS 2011, Palm Springs, CA, USA, October 22-25, 2011*, R. Ostrovsky, Ed. IEEE Computer Society, 2011, pp. 97–106. [Online]. Available: https://doi.org/10.1109/FOCS.2011.12

[12] Z. Brakerski and V. Vaikuntanathan, "Fully homomorphic encryption from ring-lwe and security for key dependent messages," in *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*, ser. Lecture Notes in Computer Science, P. Rogaway, Ed., vol. 6841. Springer, 2011, pp. 505–524. [Online]. Available: https://doi.org/10.1007/978-3-642-22792-9_29

[13] J. H. Cheon, A. Kim, M. Kim, and Y. S. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I*, ser. Lecture Notes in Computer Science, T. Takagi and T. Peyrin,

Eds., vol. 10624. Springer, 2017, pp. 409–437. [Online]. Available: https://doi.org/10.1007/978-3-319-70694-8_15

[14] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds," in *Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I*, ser. Lecture Notes in Computer Science, J. H. Cheon and T. Takagi, Eds., vol. 10031, 2016, pp. 3–33. [Online]. Available: https://doi.org/10.1007/978-3-662-53887-6_1

[15] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "TFHE: Fast fully homomorphic encryption library," August 2016, https://tfhe.github.io/tfhe/.

[16] I. Chillotti, M. Joye, and P. Paillier, "Programmable bootstrapping enables efficient homomorphic inference of deep neural networks," *IACR Cryptol. ePrint Arch.*, vol. 2021, p. 91, 2021. [Online]. Available: https://eprint.iacr.org/2021/091

[17] A. Cilardo and D. Argenziano, "Securing the cloud with reconfigurable computing: An FPGA accelerator for homomorphic encryption," in *2016 Design, Automation & Test in Europe Conference & Exhibition, DATE 2016, Dresden, Germany, March 14-18, 2016*, L. Fanucci and J. Teich, Eds. IEEE, 2016, pp. 1622–1627. [Online]. Available: http://ieeexplore.ieee.org/document/7459572/

[18] D. B. Cousins, K. Rohloff, and D. Sumorok, "Designing an fpga-accelerated homomorphic encryption co-processor," *IEEE Trans. Emerg. Top. Comput.*, vol. 5, no. 2, pp. 193–206, 2017. [Online]. Available: https://doi.org/10.1109/TETC.2016.2619669

[19] L. Ducas and D. Micciancio, "FHEW: bootstrapping homomorphic encryption in less than a second," in *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, ser. Lecture Notes in Computer Science, E. Oswald and M. Fischlin, Eds., vol. 9056. Springer, 2015, pp. 617–640. [Online]. Available: https://doi.org/10.1007/978-3-662-46800-5_24

[20] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," *IACR Cryptol. ePrint Arch.*, vol. 2012, p. 144, 2012. [Online]. Available: http://eprint.iacr.org/2012/144

[21] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009*, M. Mitzenmacher, Ed. ACM, 2009, pp. 169–178. [Online]. Available: https://doi.org/10.1145/1536414.1536440

[22] C. Gentry, A. Sahai, and B. Waters, "Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based," in *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*, ser. Lecture Notes in Computer Science, R. Canetti and J. A. Garay, Eds., vol. 8042. Springer, 2013, pp. 75–92. [Online]. Available: https://doi.org/10.1007/978-3-642-40041-4_5

[23] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. E. Lauter, M. Naehrig, and J. Wernsing, "Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy," in *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, ser. JMLR Workshop and Conference Proceedings, M. Balcan and K. Q. Weinberger, Eds., vol. 48. JMLR.org, 2016, pp. 201–210. [Online]. Available: http://proceedings.mlr.press/v48/gilad-bachrach16.html

[24] M. Kim, A. Harmanci, J.-P. Bossuat, S. Carpov, J. H. Cheon, I. Chillotti, W. Cho, D. Froelicher, N. Gama, M. Georgieva, S. Hong, J.-P. Hubaux, D. Kim, K. Lauter, Y. Ma, L. Ohno-Machado, H. Sofia, Y. Son, Y. Song, J. Troncoso-Pastoriza, and X. Jiang, "Ultra-fast homomorphic encryption models enable secure outsourcing of genotype imputation," *bioRxiv*, 2020. [Online]. Available: https://www.biorxiv.org/content/early/2020/07/04/2020.07.02.183459

[25] M. Kim, Y. Song, B. Li, and D. Micciancio, "Semi-parallel logistic regression for GWAS on encrypted data," *IACR Cryptol. ePrint Arch.*, vol. 2019, p. 294, 2019. [Online]. Available: https://eprint.iacr.org/2019/294

[26] C. Peikert, "Public-key cryptosystems from the worst-case shortest vector problem: extended abstract," in *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009*, M. Mitzenmacher, Ed. ACM, 2009, pp. 333–342. [Online]. Available: https://doi.org/10.1145/1536414.1536461

[27] O. Regev, "On lattices, learning with errors, random linear codes, and cryptography," in *Proceedings of the 37th Annual ACM Symposium on Theory of Computing, Baltimore, MD, USA, May 22-24, 2005*, H. N. Gabow and R. Fagin, Eds. ACM, 2005, pp. 84–93. [Online]. Available: https://doi.org/10.1145/1060590.1060603

[28] M. S. Riazi, K. Laine, B. Pelton, and W. Dai, "HEAX: an architecture for computing on encrypted data," in *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, J. R. Larus, L. Ceze, and K. Strauss, Eds. ACM, 2020, pp. 1295–1309. [Online]. Available: https://doi.org/10.1145/3373376.3378523

[29] S. S. Roy, F. Turan, K. Järvinen, F. Vercauteren, and I. Verbauwhede, "Fpga-based high-performance parallel architecture for homomorphic computing on encrypted data," in *25th IEEE International Symposium on High Performance Computer Architecture, HPCA 2019, Washington, DC, USA, February 16-20, 2019*. IEEE, 2019, pp. 387–398. [Online]. Available: https://doi.org/10.1109/HPCA.2019.00052

[30] A. Severance, J. Edwards, H. Omidian, and G. Lemieux, "Soft vector processors with streaming pipelines," in *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 117–126. [Online]. Available: https://doi.org/10.1145/2554688.2554774

[31] A. Severance and G. G. Lemieux, "Embedded supercomputing in fpgas with the vectorblox mxp matrix processor," in *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2013, pp. 1–10.

[32] N. P. Smart and F. Vercauteren, "Fully homomorphic encryption with relatively small key and ciphertext sizes," in *Public Key Cryptography - PKC 2010, 13th International Conference on Practice and Theory in Public Key Cryptography, Paris, France, May 26-28, 2010. Proceedings*, ser. Lecture Notes in Computer Science, P. Q. Nguyen and D. Pointcheval, Eds., vol. 6056. Springer, 2010, pp. 420–443. [Online]. Available: https://doi.org/10.1007/978-3-642-13013-7_25

[33] F. Turan, S. S. Roy, and I. Verbauwhede, "HEAWS: an accelerator for homomorphic encryption on the amazon AWS FPGA," *IEEE Trans. Computers*, vol. 69, no. 8, pp. 1185–1196, 2020. [Online]. Available: https://doi.org/10.1109/TC.2020.2988765

APPENDIX

Notation: Let $k, \ell \in Z_{>0}$. We denote the $(k+1) \times (k+1)$ identity matrix by $\mathbf{I}_{k+1}$ and the gadget matrix $(1/2, 1/4, \ldots, 1/2^\ell) \otimes \mathbf{I}_{k+1} \in \mathbb{T}_N[X]^{(k+1)\ell \times (k+1)}$ by $\mathbf{G}$ [2]. The gadget matrix $\mathbf{G}$ induces a function $\mathbf{G}^{-1} : T_N[X]^{k+1} \to \mathbb{T}_N[X]^{(k+1)\ell}$ which is similar to bit decomposition; see [14] for details. We define the function $g : \mathbb{T} \to \{0, 1/4\}$ by $g(x) = 1/4$ if $x \in [1/4, 3/4]$, and 0 otherwise.

A. TFHE Bootstrapping Procedure

TFHE employs a generalized version of Regev Encryption that uses *Torus LWE (TLWE)* ciphertexts. A TLWE ciphertext $\mathbf{c}' \in \mathbb{T}_N[X]^{k+1}$ differs from an LWE ciphertext $\mathbf{c} \in \mathbb{T}^{n+1}$ in that the plaintext space is $\mathbb{T}_N[X]$ as opposed to $\mathbb{T}$, and the symmetric key space is $(\mathbb{Z}_2[X]_{<N})^k$ as opposed to $\mathbb{Z}_2^n$.

TFHE bootstrapping relies on the following functions, operators, and subroutines (see Ref. [14] for details):

- TFHE defines a polynomial $p_{\bar{a}_i,N}(X) \in \mathbb{Z}[X]/(X^N+1)$ in terms of parameters $\bar{a}_i, N \in \mathbb{Z}_{>0}$:

$$p_{\bar{a}_i,N}(X) = \begin{cases} 0 & \bar{a}_i \in \{0, 2N\} \\ -X^{N-\bar{a}_i} - 1 & 0 < \bar{a}_i < N \\ -2 & \bar{a}_i = N \\ X^{2N-\bar{a}_i} - 1 & N < \bar{a}_i < 2N \end{cases} \quad (2)$$

As an abuse of notation for convenience, $p_{\bar{a}_i,N}(X)$ can represent either the polynomial itself or a function that transforms the pair $(\bar{a}_i, N) \in \mathbb{Z}^2_{>0}$ into the polynomial.

- TFHE defines a polynomial $p_{\bar{b},N}(X) \in T_N[X]$ in terms of parameters $\bar{b}, N \in \mathbb{Z}_{>0}$:

$$p_{\bar{b},N}(X) = \begin{cases} X^{\lfloor N/2 \rfloor + \bar{b}} & \bar{b} - \lfloor N/2 \rfloor < 0 \\ X^{\bar{b} - \lfloor N/2 \rfloor} & 0 \leq \bar{b} - \lfloor N/2 \rfloor < N \\ -X^{\bar{b} - \lfloor N/2 \rfloor - N} & N \leq \bar{b} - \lfloor N/2 \rfloor \leq 2N \end{cases} \quad (3)$$

As an abuse of notation for convenience, $p_{\bar{b},N}(X)$ can represent either the polynomial itself or a function that transforms the pair $(\bar{b}, N) \in \mathbb{Z}^2_{>0}$ into the polynomial.

- TFHE defines a function $q_{\bar{b},N}(X) \in T_N[X]$ as the product of $p_{\bar{b},N}(X)$ and another polynomial

$$q_{\bar{b},N}(X) = (-1/8) p_{\bar{b},N}(X) \cdot \sum_{i=0}^{N-1} X^i. \quad (4)$$

- TFHE defines $\mathbf{M}(\bar{a}, N, \mathbf{B}) \in \mathbb{T}_N[X]^{(k+1)\ell \times (k+1)}$, a function applied to integers $\bar{a}, N \in \mathbb{Z}_{>0}$ and a matrix $\mathbf{B} \in \mathbb{T}_N[X]^{(k+1)\ell \times (k+1)}$:

$$\mathbf{M}(\bar{a}, N, \mathbf{B}) = \mathbf{G} + p_{\bar{a},N}(X) \cdot \mathbf{B}. \quad (5)$$

- TFHE defines the *external product* operator $\boxdot$ of a matrix $\mathbf{C} \in T_N[X]^{(k+1)\ell \times (k+1)}$ and vector $\mathbf{z} \in \mathbb{T}_N[X]^{k+1}$ as:

$$\mathbf{C} \boxdot \mathbf{z} = \mathbf{G}^{-1}(\mathbf{z}) \cdot \mathbf{C} \in T_N[X]^{k+1}. \quad (6)$$

- TFHE defines a subroutine, $\mathrm{SampleExtract}$, which transforms a TLWE ciphertext $\mathbf{v} \in \mathbb{T}_N[X]^{k+1}$ of a message $m(X) \in \mathbb{T}_N[X]$ into an LWE ciphertext $\mathbf{c} \in \mathbb{T}^{kN+1}$ of the constant term of $m(X)$. The input ciphertext $\mathbf{v}$ is encrypted under secret key $\mathbf{s}'' \in \mathbb{T}_N[X]^k$, and the output ciphertext $\mathbf{c}$ is encrypted under secret key $\mathbf{s}' \in \mathbb{Z}^{kN}$:
- TFHE defines a subroutine, $\mathrm{KeySwitch}$, which transforms a *switching key* $\mathsf{KS}_{\mathbf{s}' \to \mathbf{s}} \in (\mathbb{T}^{n+1})^{(kN)t}$ ($\mathbf{s}' \in \mathbb{Z}_2^{kN}, \mathbf{s} \in \mathbb{Z}_2^n$, $t \in \mathbb{Z}_{>0}$ is a precision parameter) and an LWE ciphertext $\mathbf{c} \in \mathbb{T}^{kN+1}$ of message $m \in \mathbb{T}$ under $\mathbf{s}'$ into an LWE ciphertext $\mathbf{c}' \in \mathbb{T}^{n+1}$ of $m \in \mathbb{T}$ under secret key $\mathbf{s}$.

Algorithm 1 summarizes the TFHE bootstrapping procedure in terms of the functions, operators, and subroutines described above (see Ref. [14] for correctness proofs).

Algorithm 1 TFHE Bootstrap

Input: Security parameters $N, n \in \mathbb{Z}_{>0}$; LWE ciphertext $(\mathbf{a}, b) \in \mathbb{T}^{n+1}$ of message $m \in \mathbb{T}$ under secret key $\mathbf{s} \in \mathbb{Z}_2^n$; bootstrapping key $\{\mathsf{BK}_i\}_{i=1}^n \in (\mathbb{T}_N[X]^{(k+1)\ell \times (k+1)})^n$; KeySwitch key $\mathsf{KS}_{\mathbf{s}' \to \mathbf{s}} \in (\mathbb{T}^{n+1})^{(kN)t}$, where $\mathbf{s}' \in \mathbb{Z}^{kN}$.

Output: LWE ciphertext $\mathbf{c}' = (\mathbf{a}', b') \in \mathbb{T}^{n+1}$ of message $g(m) \in \mathbb{T}$ under secret key $\mathbf{s} \in \mathbb{Z}_2^n$.

1: **Round the input:**
   $\bar{a}_i \leftarrow \lfloor 2Na_i \rceil \in \mathbb{Z}, \forall i \in [n]$
   $\bar{b} \leftarrow \lfloor 2Nb \rceil \in \mathbb{Z}$
2: **Initialize the accumulator:**
   $\mathbf{v} \leftarrow (\mathbf{0}, q_{\bar{b},N}(X)) \in \mathbb{T}_N[X]^{k+1}$.
3: **Main loop:**
   For $i = 1, \ldots, n$:
      $\mathbf{v} \leftarrow \mathbf{M}(\bar{a}_i, N, \mathsf{BK}_i) \boxdot \mathbf{v}$
4: **SampleExtract:**
   $\mathbf{c} \leftarrow \mathrm{SampleExtract}(\mathbf{v})$
5: **KeySwitch:**
   $\mathbf{c}' \leftarrow \mathrm{KeySwitch}(\mathsf{KS}_{\mathbf{s}' \to \mathbf{s}}, (\mathbf{0}, 1/8) + \mathbf{c})$
6: **Output:** $\mathbf{c}' \in \mathbb{T}^{n+1}$