Turbo-FHE: Accelerating Fully Homomorphic Encryption with FPGA and HBM Integration

Hassan Nassar*, Lars Bauer and Jörg Henkel*
*Chair for Embedded Systems, Karlsruhe Institute of Technology, Germany
*email: {nassar, henkel}@kit.edu

I. Introduction

The landscape of IT infrastructure shifted with the adoption of cloud computing, allowing organizations to use on-demand services, relieving them from owning IT infrastructure. However, this transformation comes with escalating concerns over privacy and security as the data processing is done on plaintext. If the cloud provider is malicious or vulnerable, the data of the users can be breached [1].

Homomorphic Encryption (HE) offers a solution by allowing data processing directly on encrypted data without decrypting it, thus maintaining confidentiality. HE has applications in healthcare, AI, e-voting, finance, and encrypted searches [1]. Despite its benefits, HE incurs substantial computational and memory costs. Fully Homomorphic Encryption over the Torus (TFHE) is a key mathematical optimization that is post-quantum secure [2], but still slow, hence hardware acceleration can help increase speed.

FPGAs are an important component of the computing landscape. They can accelerate and optimize a wide array of applications. Cloud providers integrate FPGAs in their compute infrastructure, allowing users to accelerate arbitrary applications on their cloud FPGAs. Therefore, FPGAs are ideal for accelerating HE on the server/cloud side [3].

HE is both computationally and memory bound [4] and one step of HE typically involves the processing of MiBs of data. High Bandwidth Memory (HBM), as integrated in newer generations of FPGAs, can be used to resolve such memory bottlenecks. Previous works tackled using HBM-enabled FPGAs to accelerate approximate versions of HE [4], [5]. However, such accelerators are mainly suitable for ML computations but not suitable for accuracy-critical applications. In this work, we design **Turbo-FHE**: an accurate accelerator for TFHE on an HBM-enabled FPGA to speed up its operation. Our novel contributions are summarized as follows:

- Turbo-FHE is the first work to address the memory bottlenecks of fully accurate TFHE using HBM.
- Turbo-FHE carefully analyzes the data access pattern and *it* maps independently accessed data across memory channels utilizing the HBM bandwidth fully.
- We propose a fast, parameterizable, recursive multiplier (utilizing the Karatsuba algorithm) that can easily scale to various TFHE accelerator implementations and even offer a trade-off between resource utilization and performance.

Direct questions and comments about this article to Hassan Nassar. This work was partially funded by DFG as part of SPP 2377 project ARTS-NVM, and by the BMBF grant 01IS23066 Software Campus Project HE-Trust. We thank DFG (Project Number: 405422836, NVM-OMA).

II. Homomorphic Encryption

Homomorphism is a function that preserves the structure of the groups. More specifically, given two groups $(G, \cdot)$ and $(H, \times)$, a function $h : G \to H$ is a group homomorphism, if and only if

$$h(u \cdot v) = h(u) \times h(v).$$

Let $(P, \diamond, C, \circ, e, d)$ be our homomorphic encryption scheme, where P denotes the plaintext group with the group operation $\diamond$ and C denotes the ciphertext group with the group operation $\circ$. The functions e and d denote the encryption and decryption algorithms, respectively. Given two plaintexts $a \in P$ and $b \in P$, the encryption scheme satisfies the following

$$e(a) \circ e(b) = e(a \diamond b).$$

By doing this, it allows us to perform our modified operation $\circ$ on encrypted data. The decrypted result will be the same as performing our intended operation $\diamond$ on the plaintext, but without any data knowledge during the calculation step. Only at decryption, the final result can be evaluated as

$$d(e(a) \circ e(b)) = a \diamond b.$$

The exact mathematics behind HE is based on Eigenvalue and Eigenvector algebra. We will not go into details for brevity, but before encrypting the plaintext, noise n is intentionally added to remove the possibility that the plaintext can be obtained via Gaussian elimination [2]. This causes a problem, as with several operations on the data, the noise grows and would eventually corrupt the data [2]. However, if decryption is done before a certain number of operations, the noise is eliminated.

A. Types of Homomorphic Encryption

To deal with the noise problem, several algorithms of HE exist. There are three types. First is Partially HE (PHE) [6], which just supports one operation type (addition or multiplication) for an arbitrary amount of time without losing the ability to decrypt the data. Second is Somewhat HE (SHE) [7], which supports multiple operation types but only for a limited amount of operations. Third is Fully HE (FHE) [2], which supports multiple operation types with mitigation strategies to limit the noise growth.

B. Fully HE over the Torus (TFHE)

TFHE is based on the Learning With Errors and Ring Learning With Errors (RLWE) problems [2]. The calculations for TFHE are done over the real Taurus $\mathbb{T} = \mathbb{R}/\mathbb{Z}$ in a quantized finite format $\mathbb{T}_q$ with $q = 2^{32}$. The numbers are represented as $\{0, \frac{2^0}{2^{32}}, ..., \frac{2^{32}-1}{2^{32}}\}$. $\mathbb{T}_q$ can be identified with $\mathbb{Z}_q = \mathbb{Z}/q\mathbb{Z}$ which is uniform and easier to compute. Therefore, the calculations are done over $\mathbb{Z}_q$. The polynomial ring used for RLWE is $\mathbb{R}_q = \mathbb{Z}_q[X]/(X^N + 1)$ with $N = 2^9$ and the polynomial coefficients being modulo q.

To mitigate noise, TFHE performs bootstrapping. It extends an SHE scheme with ciphertext $k(p)$ where k is the key and p is the encrypted text. The SHE scheme supports N operations, needing $N - m$ for encryption over the homomorphic ciphertext, allowing m homomorphic operations. We then re-encrypt the ciphertext using a new encrypted key $k(k')$ to get $k(k'(p))$. Using the old key in encrypted form $k'(k)$, we decrypt the ciphertext. The result is $k'(p)$ with low noise as the decryption involves a noise elimination process.

All FHE algorithms involve bootstrapping. TFHE allows to perform a user-defined Look-Up Table based computation during bootstrapping, which is why it is referred to as programmable bootstrapping (PBS) [2]. Bootstrapping is computationally intensive, and homomorphic decryption is even more complex. The main overhead comes from the external product step, where $k(p)$ is re-encrypted to produce $k(k'(p))$. In this work, we implement an accelerator for this step.

C. Related Work

Previous HE accelerators exist, first, accelerators are targeting TFHE [8], [9], [10]. They do not focus on solving the memory bottleneck, but they work under the assumption that the data will be available when needed, which practically does not hold.

Second, there are HE accelerators that use Karatsuba-based multipliers [7], [11]. They either target SHE or implement a generic multiplier that can be used for HE. In both cases, the multiplier is less complex than the one implemented in this work and is not suitable for TFHE acceleration. [11] uses a semi-Karatsuba multiplier, reducing complexity but lacking full effectiveness and optimizes for ASIC area and power not FPGA. [7] reduces DSP usage via modular reductions and performs subset of the multiplications with the rest of multiplications performed in software.

Finally, there are HE accelerators that use HBM [4], [5]. Both accelerators target the Cheon-Kim-Kim-Song (CKKS) scheme that can work in an FHE-like mode. However, CKKS uses approximate arithmetic and therefore cannot be used for processing that requires the highest accuracy like health-related algorithms, electronic voting, and financial data processing.

III. Turbo-FHE's Design & Implementation

We aim to design Turbo-FHE as an accelerator that will speed up the bottleneck of PBS. We design it to use accurate multipliers so it can be used for all applications. Moreover, we use HBM to utilize the parallelism offered by 3D memories and reduce memory contention. We build a full system on an FPGA to accelerate the PBS step of TFHE and evaluate the benefit of using HBM. For accelerating PBS we implement a Karatsuba-based accelerator (see Section III-B). We use a MicroBlaze to initialize the memory with the data. We have an HBM interface to read and write the data, it is accessible both via MicroBlaze (to initialize the data) and via our accelerator to use the data. Similarly, we have interfaces for two off-chip DRAMs, which we use for comparison purposes.

A. Custom HBM Interface

We implement our accelerator targeting an FPGA with HBM. The HBM has 32 Pseudo Channels (PCs), each of size 256 MiB. The chip includes a generic interface containing an ASIC interconnect between the PCs and the FPGA. However, we bypass this interface and implement our own custom HBM interface to obtain the highest throughput possible and we compare with the standard HBM interface in Section IV. Figure 1-I shows our custom interface for the TFHE-777 accelerator (more information in Section III-C) that performs the external product step of PBS. For one PBS, a total of 25 MiB is read. The data needed for the external product is packed in 777 3D arrays, each of the size $\{4, 4, 512\}$ of 32 bit words. We divide the 32 PCs over the 3D arrays equally to minimize the memory contention. Each PC is used to only read 256 words, distributing the load symmetrically.

To access each of the PCs independently, we create 32 AXI memory interfaces, and each is capable of managing the read and write requests for one PC. Our Karatsuba-based accelerator (more information in Section III-B) is contained in an AXI wrapper with 32 independent ports. Each PC has a data output width of 64 bits. For each port, we use a bitwidth of 512 bit packing four words from each PC together. The 256 words are read sequentially over 16 read operations. To amortize the latency as much as possible, we use double buffering. Therefore, the data is already available immediately when it is needed. Note that our memory interface does not focus on only having parallel instances of the AXI memory modules but rather on partitioning the memory in channel granularity. This granularity helps in distributing the data reading load equally across all channels and reaching the highest possible bandwidth.

B. Accelerating the External Product of PBS

The external product allows a ciphertext multiplication whose result is an encryption of the product of plaintexts. Note that the mathematics of TFHE is done over polynomials of size n. It is called 'external' because it combines the homomorphic ciphertext with the external new bootstrapping key in a homomorphic ciphertext form. It is defined as follows

$$A_i = (BK_i \cdot ((A_{i-1} * C_i) - A_{i-1})) + A_{i-1}$$

The computation goes from $i = 0$ to $i = n$. A_i is the i-th coefficient of the new ciphertext, BK is the bootstrapping

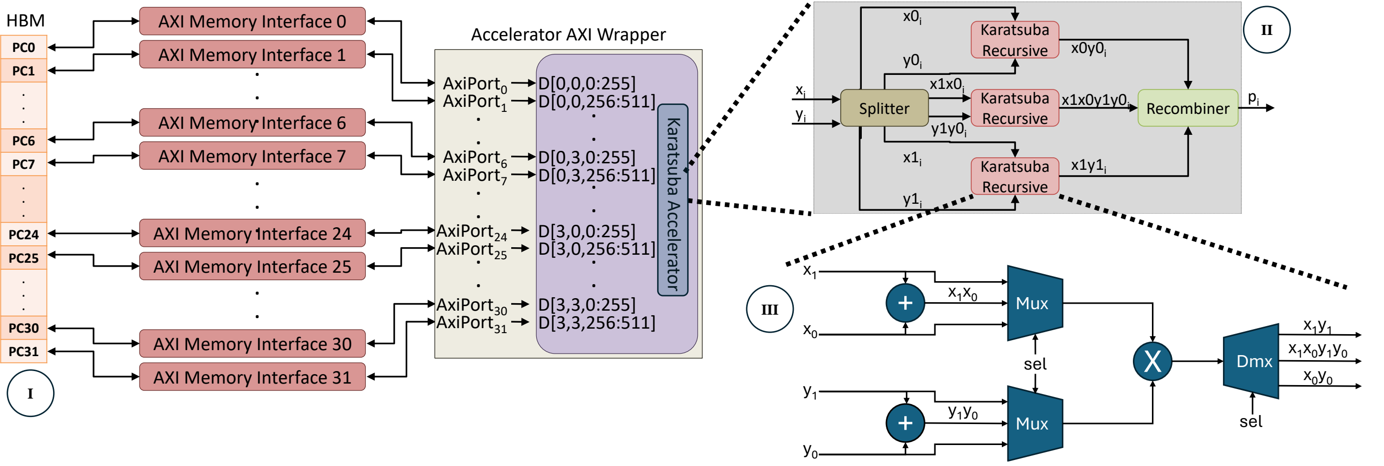

Fig. 1: Design of the accelerator. (I) shows the interface between HBM and the accelerator. (II) shows the structure of the recursive stage of the Karatsuba multiplier. (III) shows the final stage of the Karatsuba multiplier.

key in ciphertext form, C is the homomorphic ciphertext, A_{-1} is initialized to V, which is the optional lookup table that can be performed during the PBS.

A naive polynomial multiplication requires n^2 scalar multiplications. We choose to use a Karatsuba multiplier [11] that reduces the needed scalar multiplications to $n^{1.58}$, which makes a significant reduction for large values of n. E.g., for $n = 512$, the number of scalar multiplications reduces from 262,144 to 19,683.

We use SystemVerilog to implement our Karatsuba multiplier as SystemVerilog supports recursive calls of HDL designs. Figure 1-II shows the design of the Karatsuba multiplier. Each unit includes a splitter, recombiner, and three recursive instances of the Karatsuba multiplier. The number of polynomials given to each recursive unit, e.g., $x1_i$, equals half the number of polynomials of the main input x_i. Our accelerator is parameterizable. It takes two main parameters in the module instantiation; the first is the polynomial size, the second is the stopping size. By default, the Karatsuba algorithm requires implementing log_2(polynomial size) levels of the Karatsuba recursive unit. At the final stage, it will implement the actual multipliers. Hence, we get a fully parallel implementation. However, as the polynomial can be of high value, we use the 'stopping size' as second parameter. It switches from a parallel implementation to a sequential implementation at the configured level and adds registers to save the intermediate results. At the end, the number of levels implemented would be log_2(polynomial size/stopping size). This makes our accelerator very easy to fine-tune, based on the resource constraints of any FPGA it would be implemented on.

C. Accelerator Implementation

For our accelerator, we implement two different variants of TFHE: TFHE-777 and TFHE-50 [2]. Both versions are equivalent on the algorithmic level. The main difference is the volume of the processed data, and subsequently, the overhead from processing each of them. The key size of TFHE-50 is smaller than TFHE-777 and therefore, TFHE-50 is not considered fully secure [2]. We included TFHE-50 to compare performance scaling with different key sizes. TFHE-50's security is under 64 bits, much lower than TFHE-777's 128 bits [2]. Enhancing TFHE-50's security requires adjusting parameters, raising computation costs. The alternative is to increase encryption noise which will increase security but also raise ciphertext noise significantly above 2^{-40} [2]. The data for our accelerators are packed in 4D arrays of 32 bit words. The multiplication is done over sub-3D arrays of the data.

As mentioned in Section III-B, our accelerator is parameterized to build log_2(polynomial size/stopping size) levels of the Karatsuba algorithm. Ideally, we want the stopping size to equal 1, i.e., everything is parallel. The FPGA from the VCU128 board has 9024 DSP slices, each of which can be used as a multiplier. For TFHE-50, the polynomial size is 64. This means we build 6 levels of the Karatsuba recursive unit with an overall 729 multipliers. This is fine as they can all be mapped to the DSP slices. However, for the TFHE-777, with a polynomial size of 512, we would need to build 9 levels that would use 19,683 multipliers. This is far more than the available DSP slices. Therefore, we use a stopping size of 2, which reduces the number of needed DSP slices to 6,561. This fits on the FPGA, leaving a couple of hundred DSP slices for any other needed computations. The downside is that now the multiplier is 3× slower than its maximum fastest theoretical throughput if all levels were parallel. Algorithm 1 shows the steps needed to run the implemented accelerator for TFHE-777.

Figure 1-III shows the design of the final stage of the multiplier when accelerating TFHE-777. The last three multiplications use the same multiplier. Two adders that are implemented in LUTs prepare the x_1x_0 and the y_1y_0 terms. Then, via muxes and demuxes, the products are produced. The selection line of the muxes is controlled via an FSM. It has 4 values, '0' the muxes are turned off, they get constant 0 as input and the equivalent output from the demux is disconnected. Then the

Algorithm 1: TFHE-777 Accelerator Operation

```
Input: A: 4D arrays containing cipher text and key
Result: R: new cipher text
init(HBM) ; /* MicroBlaze loads arrays
 to HBM                                   */
start(load(A[0])) ; /* start loading first
 array                                    */
for i in 777 do
    Wait for load(A[i]) ; /* Wait till data
     loads                                */
    if i < 776 then
        start(load(A[i+1])) ; /* load next
         array                            */
    end
    [x, y] ← first_split(A_i, R_{i-1}) ; /* Calculate
     the first two values for the
     multiplier                           */
    init(Karatsuba_r(x,y,polynomial_size)) ;
    /* Load the data to the recursive
     multiplier                           */
    Run Karatsuba Multiplier
    R[i] ← recombine(p_r) ; /* Calculate the
     ciphertext over recursive products
     */
end
return R;
```

values '1', '2', and '3' control the output of the multiplications of the terms x_0y_0, $x_1x_0y_1y_0$, and x_1y_1 respectively.

IV. Evaluation

We evaluate our solution on a VCU128 board, based on the system implemented in Section III, using Vivado 2022.2. To evaluate several metrics we implemented both TFHE-50 and TFHE-777 once using the Karatsuba Multiplier and once using a normal multiplier. Moreover, for the memory interface, we build three different variations. The first one uses the off-chip DRAM to have a baseline without any HBM usage. The second one uses the on-chip HBM with the generic interface from Xilinx. The final one is using our proposed custom HBM interface. The same optimization is done for both the custom and generic HBM interfaces. We enable request and coherency in reordering, look ahead pre-charge and activate, and data is set to be accessed in a linear mode. For the generic interface, we let the address space consider the whole memory as one address space and let the data storage be done over all channels. For our custom interface, we treat each channel as a separate memory to be read and written independently.

A. Performance of the Accelerator

We evaluate for TFHE-777 and TFHE-50 how many PBSs per second can be performed using our accelerator. To evaluate the benefit of the Karatsuba multiplier and the custom HBM interface, for each TFHE variant, we evaluate six different combinations between the multiplier and the memory interface with the Karatsuba multiplier and the custom HBM interface being our solution Turbo-FHE.

Figure 2a shows the performance of TFHE-50. Using the off-chip DRAM achieves the lowest number of PBS, as expected. The execution is dominated by the memory accesses. The Karatsuba multiplier is effective in comparison to the normal multiplier, reaching on average $4.9\times$ improvement in PBS per second when using the same memory interface. Even using the Karatsuba multiplier with the generic HBM interface is performing $4.7\times$ better than the normal multiplier using the custom HBM interface. Our custom HBM interface outperforms the generic interface. Using Turbo-FHE, i.e., our custom interface and our Karatsuba multiplier can perform 49715 PBS per second in comparison to 18835 PBS per second using the generic interface. In comparison to the DRAM baseline, our Turbo-FHE has a $211\times$ speedup.

For the TFHE-777 the same trends generally hold with a few differences in Fig. 2b. First, as the data and computations are significantly more, the PBS per second achieved using the Karatsuba multiplier and the custom HBM interface (Turbo-FHE) is 2627. Second, the difference between using custom and generic interfaces is smaller as the Karatsuba multiplier achieves 1763 PBS using the generic interface. Third, using the custom or the generic interface makes little difference for the normal multiplier as it is now dominated by the computation of the multiplication, not the data loading. This makes sense as the number of multiplications grows from roughly 64^2 to roughly 512^2. This huge increase in the multiplications makes Turbo-FHE have a higher speedup in comparison to the DRAM baseline of $438\times$.

We also evaluate the trade-off between the stopping size and the achievable PBS per second. We run different syntheses with different values of the stopping size for TFHE-777 and see the effect on the PBS. Figure 2c shows the results. For a stopping of 1, i.e., fully parallel implementations the needed DSPs exceed the available DSPs as mentioned in Section III-C but theoretically it would achieve around 10,000 PBS per second. Going to a stopping size of 3 significantly reduces the DSP utilization and put the PBS at 1,000 PBS per second. This is useful for implementing more TFHE parts on the same FPGA, however, this is not investigated further as we focus on getting highest performance for the main bottleneck which is the external product. Going to higher stopping sizes is not beneficial as the PBS per second degrades significantly.

B. FPGA Resource Utilization

We implement our accelerator to use the most possible resources and make it as parallel as possible. Moreover, we implement our own HBM interface which has an increased cost. Table I shows our resource utilization for each module implemented. All the components reside simultaneously on the FPGA except for TFHE-777 and TFHE-50; only one of them exists on the FPGA at one time. Our resource utilization for TFHE-777 is notably high, however, this is by design as we

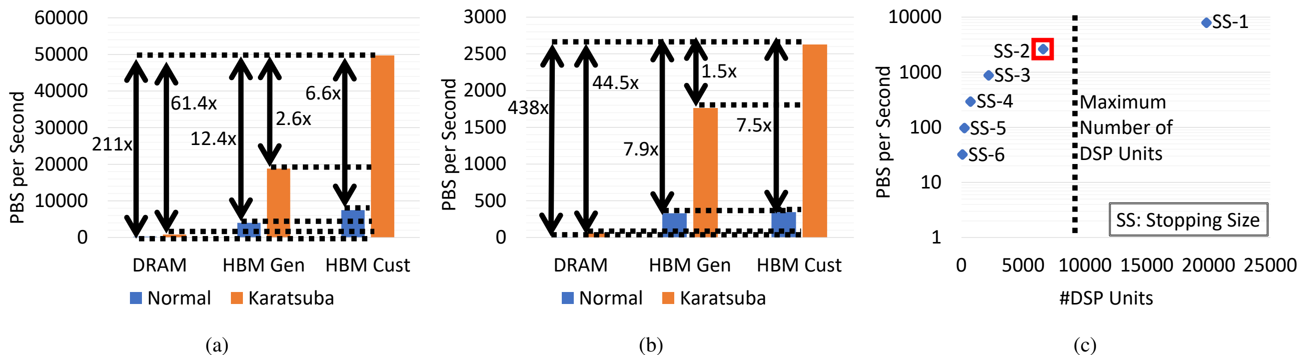

Fig. 2: Maximum programmable bootstrappings per second. (a) for TFHE-50 with different multipliers and memory interfaces (b) the same for TFHE-777 (c) for different variations of the stopping size for the recursive Karatsuba multiplier when accelerating TFHE-777

try to use all possible DSPs alongside the needed LUTs. The design of TFHE-777 is semi-sequential as the number of DSPs on the FPGA is not enough for a fully parallel version and therefore the difference between it and TFHE-50 is not higher than 3×. It should be noted that it still fits alongside the TFHE-777 on the FPGA. Moreover, if our accelerator would be used in an ASIC design, then it would replace the generic HBM interface from Xilinx, reducing the overhead correspondingly.

TABLE I: Resource utilization on the VCU128 board. For TFHE-777 we managed to utilize most of the 1,182,240 LUTs of that FPGA

Module	LUT	Register	BRAM	DSP
TFHE-777	891,684	189,705	1,586	6,613
TFHE-50	332,242	31,843	254	1,348
Custom HBM interface	113,574	76,257	151	0
Generic HBM interface	38,648	39,194	105	0
MicroBlaze	1,192	810	0	0
DRAM controller	30,236	35,712	51	6
AXI bus	19,771	34,226	0	0

C. Comparison to Related Work

Table II compares our accelerator to the state-of-the-art of HE accelerators. Accelerators from [10], [9], [8] work under the assumption that the data will be available, i.e., they do not consider the memory bottleneck. Accelerators from [4], [5] use HBM to resolve the memory bottleneck. However, they have even lower accuracy as they target CKKS, which by design only supports approximate calculations. Moreover, compared to the CKKS accelerators, our Turbo-FHE achieves the highest bandwidth utilization of 406 GiB/s as its memory controller is tailored to get maximum throughput to the multiplier implemented. In contrast, both Ref. [4] and Ref. [5] use generic memory controllers limiting the bandwidth utilization. Finally, compared to Refs. [11], [7], our work targets TFHE while they target generic and SHE algorithms.

V. Conclusion

We introduce Turbo-FHE, a fully homomorphic encryption accelerator on an FPGA with HBM. Turbo-FHE accelerates the state-of-the-art TFHE algorithm using an accurate and fast Karatsuba multiplier. We implement the Karatsuba multiplier recursively and in a parameterized matter to adjust it to the resource requirements of the system. To load the data with high throughput, we create our custom HBM interface. We analyzed the memory access patterns of TFHE to design and implement a custom HBM interface minimizing the data contention. Using this interface and the Karatsuba algorithm we achieved speedups of 211× and 438× for TFHE-777 and TFHE-50, respectively, compared to a baseline implementation using DRAM and a standard multiplier. Compared to the state-of-the-art, Turbo-FHE is the only TFHE accelerator that supports accurate calculations along with fully benefiting from HBM bandwidth.

TABLE II: Comparison to state-of-the-art HE accelerators. Our Turbo-FHE achieves the highest bandwidth utilization.

	HBM	Alg.	Board	DSP	BW
Ours	✓	TFHE	VCU128	6,613	406 GiB/s
Ref. [8]	✗	TFHE	Zed	40	N.A.
Ref. [9]	✗	TFHE	Zed	128	N.A.
Ref. [10]	✗	TFHE	VU13P	12,288	N.A.
Ref. [4]	✓	CKKS	U280	6,144	316 GiB/s
Ref. [5]	✓	CKKS	U280	3,072	272 GiB/s
Ref. [7]	✗	SHE	DE5	100	N.A.
Ref. [11]	✗	Gen.	ASIC	N.A.	N.A.

References

[1] K. B. Johnson, W.-Q. Wei, D. Weeraratne, M. E. Frisse, K. Misulis, K. Rhee, J. Zhao, and J. L. Snowdon, "Precision medicine, AI, and the future of personalized health care," *Clinical Translational Science*, vol. 14, no. 1, pp. 86–93, 2020.

[2] I. Chillotti, M. Joe, and P. Paillier, "Programmable bootstrapping enables efficient homomorphic inference of deep neural networks," Zama AI, 2020. [Online]. Available: https://github.com/zama-ai/tfhe-rs

[3] Amazon Web Services (AWS). (2021) EC2 F1 instances. Amazon.com, Inc. [Online]. Available: https://aws.amazon.com/ec2/instance-types/f1/

[4] Y. Yang, W. Long, R. Kannan, and V. K. Prasanna, "Fpga acceleration of rotation in homomorphic encryption using dynamic data layout," in *IEEE FPL*, 2023, pp. 174–181.

[5] Y. Yang, H. Zhang, S. Fan, H. Lu, M. Zhang, and X. Li, "Poseidon: Practical homomorphic encryption accelerator," in *IEEE HPCA*, 2023, pp. 870–881.

[6] J. Ryu, K. Kim, and D. Won, "A study on partially homomorphic encryption," in *IEEE IMCOM*, 2023, pp. 1–4.

[7] V. Migliore, M. M. Real, V. Lapotre, A. Tisserand, C. Fontaine, and G. Gogniat, "Hardware/software co-design of an accelerator for fv homomorphic encryption scheme using karatsuba algorithm," *IEEE TC*, vol. 67, no. 3, pp. 335–347, 2018.

[8] S. Gener, P. Newton, D. Tan, S. Richelson, G. Lemieux, and P. Brisk, “An fpga-based programmable vector engine for fast fully homomorphic encryption over the torus,” in *ACM Secure and Private Systems for Machine Learning*, 2021, pp. 1–7.
[9] B. Bulut, S. N. Bicakci, and I. San, “Hw/sw co-design of tfhe homomorphic or gate via ntt-based polynomial multiplication on a programmable soc,” in *IEEE Innovations in Intelligent Systems and Applications Conference*, 2022, pp. 1–6.
[10] T. Ye, R. Kannan, and V. K. Prasanna, “Fpga acceleration of fully homomorphic encryption over the torus,” in *IEEE HPEC*. IEEE, 2022, pp. 1–6.
[11] W. Tan, B. M. Case, A. Wang, S. Gao, and Y. Lao, “High-speed modular multiplier for lattice-based cryptosystems,” *IEEE TCAS-II*, vol. 68, no. 8, pp. 2927–2931, 2021.

Hassan Nassar received his M.Sc degree from Ulm University, Germany in 2019 and his B.Sc degree –with highest honours– from the German University in Cairo, Egypt, in 2016. His research interests are hardware security, Reconfigurable Architectures, and Cloud FPGAs.

Lars Bauer received the Diploma and Ph.D. degrees in computer science from the University of Karlsruhe, Germany, in 2004 and 2009. Dr. Bauer received two dissertation awards (EDAA and FZI), two best paper awards (AHS’11 and DATE’08) and several nominations. His research interests include architectures and management for adaptive multi-/manycore systems.

Jörg Henkel received the Diploma and Ph.D. from the Technical University of Braunschweig. His research interest includes co-design for embedded hardware/software systems w.r.t. low power, reliability/security and means of embedded machine learning. He is the Vice President Publications at IEEE CEDA and a Fellow of both the ACM and the IEEE.