Hardware Acceleration for Fully Homomorphic Encryption

Suchir Vemulapalli
Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX, USA
suchirvemulapalli@gmail.com

***Abstract*—Fully Homomorphic Encryption (FHE) presents a transformative encryption paradigm that reconciles data usability with privacy, enabling operations directly on encrypted data. However, conventional implementations on GPUs and CPUs suffer from significant performance limitations. In contrast, specialized hardware solutions like Field Programmable Gate Arrays (FPGAs) emerge as a compelling alternative. Leveraging their parallel and adaptable architecture, FPGAs, when organized into clusters, effectively address key computational bottlenecks, substantially enhancing FHE performance. This exploration delves into the reasons why FPGAs stand out as the optimal solution, elucidates the underlying architecture facilitating FHE acceleration, and contemplates the potential extension of FPGA-based hardware acceleration to other encryption schemes like Post Quantum Cryptography (PQC) and Zero Knowledge Proofs (ZKP).**

***Index Terms*—FPGA, FHE, Fully Homomorphic Encryption, Hardware Acceleration, Parallelism**

I. Introduction

Fully Homomorphic Encryption (FHE) is an encryption scheme that allows you to perform operations directly on encrypted data without requiring decryption. To motivate why FHE is useful, we present an example. Consider a hospital that wants a researcher to compute certain statistics about a data set, but the hospital can't directly reveal the data to the researcher because of privacy concerns related to patient confidentiality. Thus, to solve this problem, the hospital can encrypt the data using FHE and create a ciphertext that that researcher directly operates on. The hospital can later decrypt the ciphertext to view the results of the statistical analysis [1]. FHE is unique in that it supports both homomorphic multiplication and homomorphic addition. Earlier, more naive homomorphic encryption schemes known as Partial Homomorphic Encryption (PHE) support either homomorphic multiplication or homomorphic addition [2].

A Field Programmable Gate Array (FPGA) is a reconfigurable integrated circuit that functions as a blank canvas for digital circuits. It comprises a multitude of configurable logic blocks interconnected by programmable routing resources. FPGAs offer the unique capability of being programmable and reprogrammable, enabling users to design and implement custom digital circuits. This programmability facilitates rapid prototyping and iteration, making FPGAs highly adaptable to diverse applications. Moreover, FPGAs are excellent at exploiting parallelism which makes them suitable for low latency applications.

II. FHE Operations

The first step necessary in FHE is to encode a input message into a plaintext polynomial. The plaintext polynomial is then encrypted into a ciphertext, c, consisting of two polynomials.

$$c = (a, b) \tag{1}$$

In the equation above, a, b are two polynomials with N integer coefficients modulo a large integer Q. Additionally, N, Q are both parameters that must be defined before performing FHE [2].

A. Homomorphic Multiplication

Homomorphic multiplication involves both polynomial multiplication and relinearization. Polynomial multiplication is the process by which two vectors that each contain two polynomials are multiplied together to create a new ciphertext.

$$(a_0, b_0) * (a_1, b_1) = (a_0a_1, a_0b_1 + b_0a_1, b_0b_1) \tag{2}$$

$$(a_0a_1, a_0b_1 + b_0a_1, b_0b_1) = (c_2, c_1, c_0) = c \tag{3}$$

In the equation above, (a_0, b_0) represents the first vector, and (a_1, b_1) represents the second vector. An intermediate ciphertext result that contains three polynomials is created as a result of the multiplication. Since FHE only supports ciphertext with two polynomial entries, the intermediate result is converted into a vector with two polynomial entries in a process known as relinearization.

$$(c_2, c_1, c_0) = (a_{\text{new}}, b_{\text{new}}) = c_{\text{new}} \tag{4}$$

Thus, in order to support homomorphic multiplication, the accelerator must support both polynomial multiplication and relinearization [2].

B. Homomorphic Addition

Homomorphic addition is the process by which two vectors consisting of two polynomial entries each are added together. The addition is simple element-wise polynomial addition [2].

$$c_0 + c_1 = (a_0 + a_1, b_0 + b_1) = c_{\text{new}} \tag{5}$$

C. Rotation

Rotation is another basic FHE operation that involves rotating entries of a certain vector, x, by k entries. For example, let $k = 2$ [3].

$$x = (a, b, c, d, e) \tag{6}$$

$$x_{\text{rotated}} = (d, e, a, b, c) \tag{7}$$

D. Bootstrapping

The noise within a ciphertext continuously grows with each multiplication, addition, and rotation operation. Once the noise crosses a certain threshold, it is impossible to decrypt the ciphertext. This functionally destroys the ciphertext. Thus, it becomes imperative to reduce the noise within a ciphertext to acceptable levels. Bootstrapping is a process in FHE that resets the noise to a lower level to allow more computations on the data to occur. Bootstrapping may need to be performed multiple times on the same data set depending on how many total operations need to be performed on the encrypted ciphertext. The discovery and subsequent implementation of bootstrapping was one of the key breakthroughs necessary to allow for FHE [3].

III. The Advantages of FPGAs

In the landscape of FHE schemes, various approaches and platforms have been explored, yet FPGAs emerge as the optimal solution.

Current software homomorphic encryption libraries that utilize CPUs are much too slow to be viable. For instance, a computation that takes only 1 second with plaintext extends to an alarming 11 days when executed on a CPU-based FHE library, resulting in a staggering 1,000,000x slowdown. In contrast, FPGAs present a significantly more efficient alternative, introducing only a 26x slowdown for the same calculation.

While GPUs can leverage parallelism akin to FPGAs, their excess idle floating point compute units become a drawback for FHE, which primarily operates on integers. This inefficiency necessitates a higher utilization of GPU resources to match the computational power of an FPGA. Moreover, the rising costs of industry-grade GPUs render them economically less attractive for FHE. Additionally, the substantial memory bandwidth requirements due to large public key sizes (ranging from 70MB to 2.3GB) further diminish the appeal of both GPU and CPU solutions [11].

Comparatively, Application-Specific Integrated Circuits (ASICs) are similar to FPGAs in that they are specialized microchips that combine multiple integrated circuits for a specific function. However, ASICs lack the reprogrammability inherent in FPGAs. While ASICs demonstrate superior speed in executing a designated task due to their tailored design, FPGAs offer versatility by being reprogrammable for diverse functions. Despite ASICs outpacing FPGAs in FHE execution speed, the constant evolution and refinement of FHE acceleration techniques render ASICs impractical. The expense and inflexibility associated with producing new ASICs for every modification in FHE implementation underscore the importance of FPGAs in achieving a balance between performance and adaptability [4].

IV. Key Targets for Acceleration

In the pursuit of enhancing the performance of FHE, the focus on key computational targets becomes paramount for the successful implementation of an FPGA-based hardware accelerator. Efficient acceleration hinges on identifying and optimizing the computational bottlenecks.

Relinearization, rotation, and bootstrapping all require a supplementary operation called key switching. Key switching transformations change the encryption of a ciphertext from one secret key to another secret key. Key switching is the main computational bottleneck in FHE. Any accelerator that seeks to improve the efficiency of FHE must prioritize improving the speed of a key switch.

While the primary focus should revolve around reducing the latency of these bottlenecks, it's worth noting that FPGAs can also expedite auxiliary functions. Notably, polynomial multiplication and addition benefit from accelerated processing through the application of the Number Theoretic Transform (NTT) and Inverse Number Theoretic Transform (INTT). NTT, essentially a Fast Fourier Transform (FFT) over a finite set of integers, proves instrumental in boosting the performance of these fundamental operations [2].

The inherent complexity of FHE renders the reliance on a single FPGA an infeasible solution. Instead, a more practical approach involves the collaborative efforts of multiple FPGAs working concurrently in the cloud. This simultaneous collaboration not only distributes resource utilization effectively but also mitigates traffic congestion, thus averting traffic bottlenecks. The cohesive operation of groups of FPGAs in such a manner is commonly referred to as FPGA clusters. Significantly, FPGA acceleration for FHE can leverage existing FPGA clusters offered by certain cloud providers, repurposing the same infrastructure by reconfiguring the FPGAs for FHE. This approach significantly reduces the setup time required for the implementation of a functional accelerator [5].

V. Architecture

This section initiates with an exploration of the holistic system architecture of an FPGA within a FHE framework. Subsequently, it delves into a detailed examination of specific modules, elucidating their architectures, including cluster topology, NTT, relinearization, bootstrapping, and key switching.

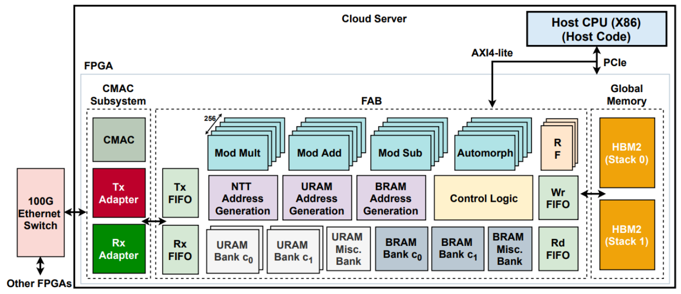

Fig. 1. Diagram of the Overall FPGA System Architecture [3]

A. Overall System Architecture

- A host CPU communicates with the FPGA in order to upload the RTL design which is sent as kernel code. The host CPU also defines the parameters necessary for FHE such as N, Q and sends them to the FPGA. This communication is enabled through PCIe.
- The kernel code instantiates the functional logic units that perform the arithmetic operations necessary for FHE. These functional logic units will house more specific FHE modules such as NTT/INTT. One implementation example using the Alveo 280 board has 256 functional logic units for one FPGA.
- Read and Write FIFOs allow for data transfer from the global memory to the on-chip memory. This transfer is especially important considering the RTL design is initially uploaded into a subset of the global memory and must make it onto the on-chip memory. Additionally, transfer from the on-chip memory to the global memory is critical to ensure that the modified ciphertext can be transferred back to the host after operations are performed. The global memory is larger than the on-chip memory because it needs to permanently store all values necessary for FHE such as public keys while the on-chip memory can pull values from the global memory as it needs them. The Alveo 280 implementation utilizes an 8GB global memory for each FPGA.
- The on-chip memory consists of both Ultra RAM (URAM) and Block RAM (BRAM) cells. The BRAM is dual-ported which allows for multiple read/writes to occur simultaneously. This is another way FPGAs increase the overall parallelism of an FHE scheme.
- Register files (RF) store the parameters (N, Q) for FHE once the host CPU uploads them, and they also store intermediate results during an FHE operation. For example, during homomorphic multiplication, there is an intermediate vector with three polynomials that must be relinearized into a vector with two polynomials. In the Alveus 280 implementation, the RF is 2MB in size.
- Transmit (Tx) and Receive (Rx) FIFOs enable inter-FPGA communication. These FIFOs are critical for the FPGA cluster since they facilitate connections from one FPGA to the rest of the cluster. The Alveus 280 model connects the Tx and Rx FIFOs to a 100Gb/s ethernet switch [3].

B. Cluster Topology

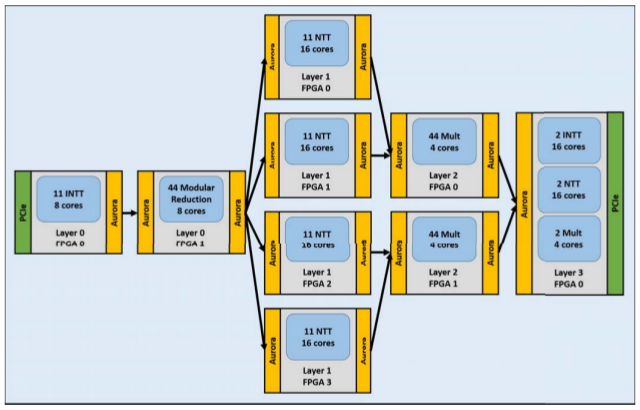

Fig. 2. Diagram of Cluster Topology [5]

- Layer 0 receives the initial data from the host CPU. The data undergoes INTT and modular reduction within this layer. This layer utilizes two FPGAs in the cluster daisy-chained together. One FPGA handles INTT, and the other FPGA handles modular reduction. This reduces the congestion caused by performing both operations in one FPGA since neither of the FPGAs are overburdened.
- Layer 1 contains multiple NTT modules that work in parallel to simultaneously perform NTT on separate parts of the data received by the first layer to increase the throughput. In this layer, 4 FPGAs are used to both increase the parallelism and distribute resource utilization across more URAM and BRAM cells to account for a high memory bandwidth. This reduces the possible bottlenecks caused by a full memory where read/writes are delayed.
- Layer 2 uses FPGAs that contain multiplier modules for evaluation key multiplication and summation. Evaluation key multiplication and summation is a core component of bootstrapping. Layer 2 uses at a minimum two FPGAs to distribute the utilization of URAM.
- Layer 3 is the final layer. This layer receives data from Layer 2, adds all polynomials of the same modulus together in a process known as Residue Number System (RNS) flooring, and finally sends the data back to the host CPU since the FHE operations have been completed. Layer 3 requires only one FPGA for compatibility.

It's important to note that the cluster above represents the minimum cost solution required to see genuine increases in throughput and speedup for FHE. It's likely that more FPGAs will result in greater acceleration and performance, but the trade off becomes unattractive due to diminishing returns and rising costs of using more FPGAs [5].

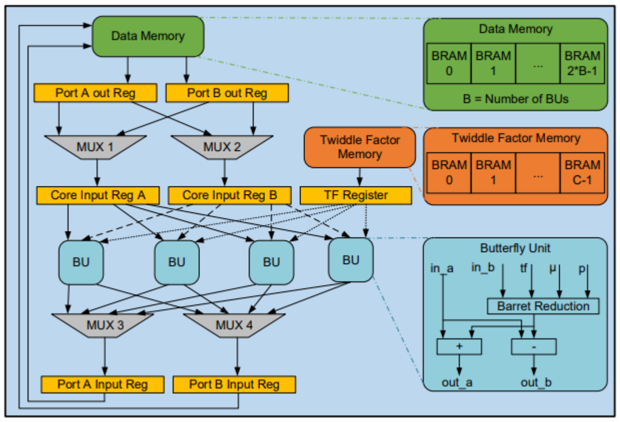

Fig. 3. Diagram of NTT Module [5]

C. NTT Architecture

The NTT module utilizes a multi-cycle design where different stages of NTT are computed sequentially using the same hardware. This avoids excessive hardware costs in an FPGA that would result from using multiple copies of the same hardware for separate NTT stages. Available resources on an FPGA are scarce and in high demand from multiple functions, so eliminating redundant hardware is critical.

- Butterfly Units (BUs) contain the bulk of the computational logic necessary to perform NTT. They contain a modular multiplier, adder, and subtractor. Multiple BU units are placed within the same stage to increase parallelism for throughput.
- Multiplexers (MUXs) facilitate the transfer of data from memory blocks to the BUs.
- Twiddle factors are precomputed to avoid the resource and latency costs of recomputing them on the fly. Twiddle factors are trigonometric constant coefficients that are fundamental to the functionality of NTT [5].

D. Relinearization Architecture

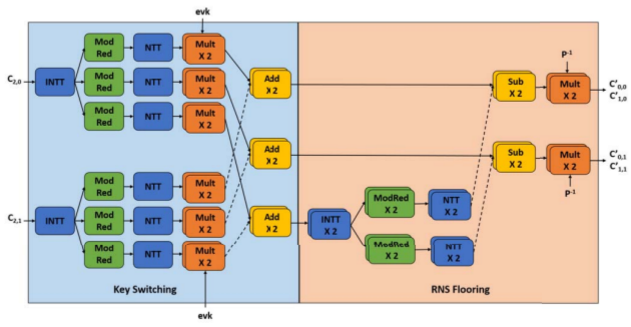

Fig. 4. Diagram of Relinearization Module [5]

Relinearization is divided into two stages. The first stage is a key switch operation. Thus, by accelerating key switch, we also massively improve relinearization. The second stage of relinearization involves RNS flooring which is simply adding all polynomials of the same modulus together.

While relinearization can be fully pipelined, it is much too impractical to implement and wastes too many vital resources. Because of this, relinearization is only partially pipelined in a way that effectively balances performance and resource utilization. Specific optimizations in relinearization include sending all polynomials of the same modulus simultaneously since the RNS flooring operation can process these inputs without waiting. Furthermore, the throughput may need to be balanced in order to maintain compatibility with other parts of the network due to clocking issues. This is done by increasing or reducing the number of multipliers in parallel [5].

E. Bootstrapping Architecture

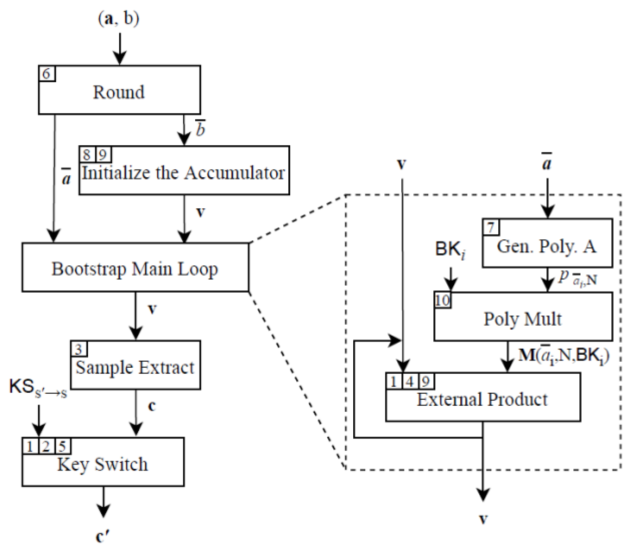

Fig. 5. Diagram of Bootstrapping Module [6]

Similar to relinearization, bootstrapping achieves the most speedup by improving the efficiency of key switching. However, some general strategies to improve bootstrapping involve loop unrolling, parallel arithmetic modules, pipelining, and double buffering.

Loop unrolling is the process by which you can convert a loop that runs for a fixed number of iterations into the actual series of statements that the loop executes. Instructions within a loop must be executed sequentially, so by loop unrolling you allow statements to be executed in parallel which increases performance.

Double buffering allows for data in one buffer to be processed while new data is being read into another buffer. This increases the throughput since more data can be processed in less time due to the overlap [2] [6].

F. Key Switch Architecture

A key switch involves four smaller operations: Decomp, ModUp, KSKIP, and ModDown that are chained together to produce the final datapath.

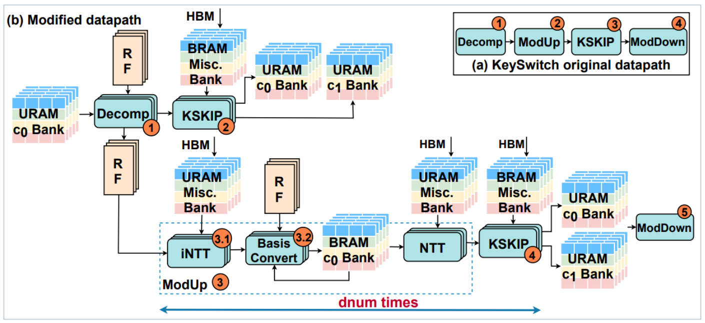

Fig. 6. Diagram of Modified Key Switch Datapath [3]

- Decomp: This operation divides the encrypted data into smaller parts called limbs
- ModUp: It takes each of these limbs, performs some operations, and produces a new set of limbs.
- KSKIP: This step involves combining the limbs produced from ModUp with the limbs of a secret key that the ciphertext is changing encryption to. KSKIP creates the intermediate result between the input polynomial and the secret key.
- ModDown: This operation is similar to ModUp but works in reverse. It takes the results from the previous steps and reduces them back to a specific size.

The issue with this datapath arises in the KSKIP step. In the original datapath, when ModUp creates the new limbs, there is not enough space in the on-chip memory to store them. The limbs must be written into main memory, then they are read back into on-chip memory for KSKIP since KSKIP needs to read all limbs simultaneously. The read and write latencies to main memory bottleneck the pipeline since they take much more time compared to read and writes to the on-chip memory. Compounding the problem is the fact that KSKIP requires NTT operations. These factors result in the slow compute time for key switch.

The solution is to split KSKIP into two separate steps in the modified datapath. Instead of performing KSKIP all at once, as soon as operands are available in the on-chip memory, KSKIP can begin. KSKIP now becomes greedy. This avoids the need to read limbs into the main memory and write them back into the on-chip memory. This reduces the total number of DRAM transfers necessary thereby reducing the latency caused by main memory traffic. Moreover, the total number of NTT operations are also reduced since there is no need to perform an initial NTT on the data received from main memory. Additionally, the modified datapath exploits smart operation scheduling which hides the latency of read and writes to main memory behind other computations. These strategies alleviate the bottleneck caused by key switching and lead to massive improvements in FHE acceleration [3].

VI. Post Quantum Cryptography Acceleration

Post Quantum Cryptography (PQC) refers to a group of encryption schemes on classical computers where no known

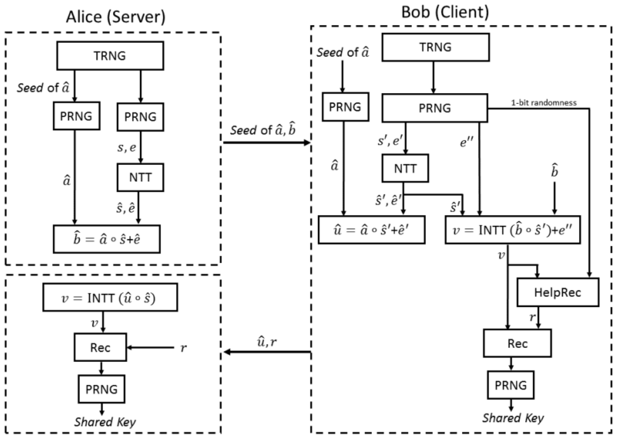
Alice (Server)
Bob (Client)
TRNG
Seed of $\hat{a}$
PRNG
s, e
NTT
$\hat{s}, \hat{e}$
$\hat{b} = \hat{a} \circ \hat{s} + \hat{e}$
Seed of $\hat{a}, \hat{b}$
1-bit randomness
s', e'
e''
$\hat{s}', \hat{e}'$
$\hat{u} = \hat{a} \circ \hat{s}' + \hat{e}'$
$v = \text{INTT}(\hat{b} \circ \hat{s}') + e''$
HelpRec
Rec
$\hat{u}, r$
$v = \text{INTT}(\hat{u} \circ \hat{s})$
Shared Key

Fig. 7. Diagram of PQC Hardware Architecture [7]

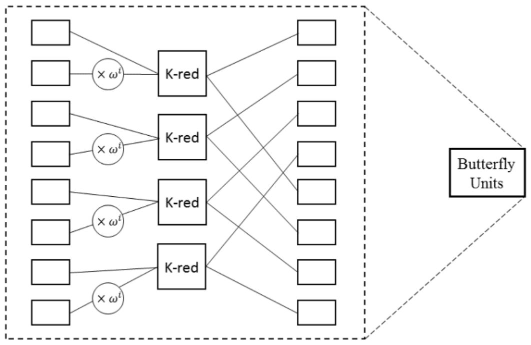

Fig. 8. Diagram of PQC Butterly Unit [7]

classical or quantum algorithm can efficiently recover a user's private key (crack the decryption). PQC and FHE schemes heavily utilize the same underlying principles for acceleration because they both involve similar operations such as polynomial multiplication. For instance, accelerating a Post Quantum Key Exchange using hardware necessitates the need for NTT/INTT units as well as BUs [7]. Additionally, FPGAs are still preferred over any other acceleration platform.

However, the notable difference between PQC and FHE is that PQC involves many more families of cryptography compared to the lattice-based FHE. PQC can be lattice-based, multivariate, hash-based, code-based, or isogeny-based. The inclusion of the different families means that specific hardware requirements change depending on the type of PQC scheme. For example, XMSS, a hash-based PQC scheme requires all accelerators to contain a SHA-256 module. Another example is the Niederreiter cryptosystem, a code-based PQC scheme which requires a Gaussian eliminator unit over finite fields [8].

VII. Zero Knowledge Proofs Acceleration

Zero Knowledge Proofs (ZKPs) are a cryptography scheme where one party is able to prove to others that a statement is true without leaking any information.

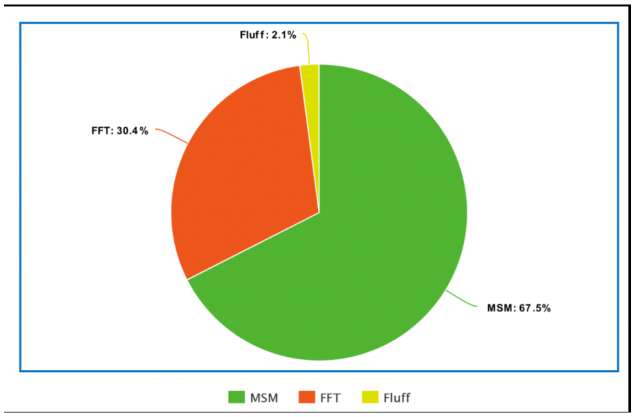

Fig. 9. Runtime Graph for ZKP [9]

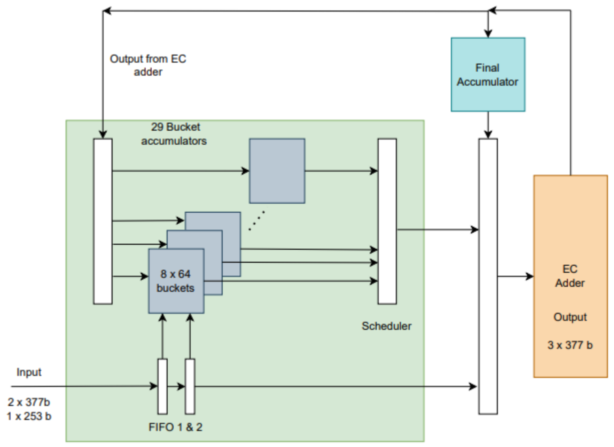

Fig. 10. Diagram of MSM Module [10]

ZKP, FHE, and PQC are similar in the fact that they all require NTT units to speed up polynomial multiplication and addition. Additionally, FPGAs are still currently considered the most viable way to accelerate ZKPs. However, a growing demand for ZKP solutions coupled with better ZKP acceleration algorithms will lead to a burgeoning market for Zero Knowledge Processing Units (ZPUs). ZPUs are special function hardware designed solely for ZKPs and will outperform any other acceleration platform including ASICs. However, they are currently plagued by the same pitfalls that ASICs have including prohibitive cost and inability to reprogram.

The main difference between ZKP and FHE acceleration is that the majority of the computational power and time in ZKPs is spent on multi-scalar multiplication (MSM). MSM is an algorithm that calculates the sum of multiple scalar multiplications. It consists of both modular multiplication and elliptic curve addition and enables both variable-base and fixed-base multiplication. Thus, in order to accelerate ZKP, an MSM module must be implemented into the accelerator [9].

One possible implementation of an MSM module in an FPGA calculates partial sums of the MSM in parallel. These partials sums are then fed simultaneously into one adder to compute the final MSM result which creates a low latency, highly parallel MSM module [10].

VIII. Conclusion

In conclusion, this literature review has underscored the pivotal role of FPGAs as a preeminent solution in advancing the field of FHE. The transformative potential of FHE in reconciling data usability with privacy is evident, yet the conventional implementations on GPUs and CPUs present formidable performance challenges. FPGAs, distinguished by their parallel and adaptable architecture, offer a compelling alternative. By strategically organizing FPGAs into clusters, this study has demonstrated their remarkable capacity to address key computational bottlenecks, resulting in a substantial enhancement of FHE performance. The optimal attributes of FPGAs, detailed in this exploration, extend beyond FHE acceleration, prompting contemplation of their application to other encryption paradigms such as PQC and ZKP. As we navigate the evolving landscape of cryptographic solutions, FPGAs stand as a beacon, not only facilitating the acceleration of FHE but also inspiring further exploration into their broader potential across diverse encryption schemes.

Acknowledgment

We would like to thank Dr. Vishwanath and the TAs for a great semester, and for any help they offered us over the course of this report!

References

[1] “What is Fully Homomorphic Encryption? - FHE Explained,” Inpher. https://inpher.io/technology/what-is-fully-homomorphic-encryption/
[2] “Accelerating Fully Homomorphic Encryption with an Open-source FPGA...,” Intel. https://www.intel.com/content/www/us/en/developer/articles/technical/homomorphic-encryption/accelerating-homomorphic-encryption-for-fpga.html.
[3] R. Agrawal et al., ”FAB: An FPGA-based Accelerator for Bootstrappable Fully Homomorphic Encryption,” 2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA), Montreal, QC, Canada, 2023, pp. 882-895, doi: 10.1109/HPCA56546.2023.10070953.
[4] R. Agrawal and L. Sturmann, “Preserving privacy in the cloud: speeding up homomorphic encryption with custom hardware,” Red Hat Research, Nov. 2022. https://research.redhat.com/blog/article/privacy-in-the-cloud-speeding-up-homomorphic-encryption-with-fpgas/
[5] H. Liao et al., ”TurboHE: Accelerating Fully Homomorphic Encryption Using FPGA Clusters,” 2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS), St. Petersburg, FL, USA, 2023, pp. 788-797, doi: 10.1109/IPDPS54959.2023.00084.
[6] Y. Serhan 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.” [Online]. Available: https://people.ece.ubc.ca/lemieux/publications/gener-spsl2021.pdf
[7] P.-C. Kuo et al., “High Performance Post-Quantum Key Exchange on FPGAs,” ePrint IACR, 2017. https://eprint.iacr.org/2017/690.
[8] Wang, Wen, ”Hardware Architectures for Post-Quantum Cryptography” (2021). Yale Graduate School of Arts and Sciences Dissertations. 242. $https://elischolar.library.yale.edu/gsas_dissertations/242$
[9] I. Segol, “Hardware Review: GPUs , FPGAs and Zero Knowledge Proofs,” Ingonyama, May 04, 2023. https://www.ingonyama.com/blog/hardware-review-gpus-fpgas-and-zero-knowledge-proofs
[10] C. F. Xavier, “PipeMSM: Hardware Acceleration for Multi-Scalar Multiplication,” ePrint IACR, 2022. https://eprint.iacr.org/2022/999.

[11] X. Cao, C. Moore, M. O'Neill, E. O'Sullivan and N. Hanley, "Optimised Multiplication Architectures for Accelerating Fully Homomorphic Encryption," in IEEE Transactions on Computers, vol. 65, no. 9, pp. 2794-2806, 1 Sept. 2016, doi: 10.1109/TC.2015.2498606.