FAB: An FPGA-based Accelerator for Bootstrappable Fully Homomorphic Encryption

Rashmi Agrawal[1] Leo de Castro[2] Guowei Yang[1] Chiraag Juvekar[3] Rabia Yazicigil[1]
Anantha Chandrakasan[2] Vinod Vaikuntanathan[2] Ajay Joshi[1]
[1]Boston University, Boston, MA, USA; [2]MIT, Cambridge, MA, USA; [3]Analog Devices, Boston, MA, USA
{rashmi23, guoweiy, rty, joshi}@bu.edu, {ldec, anantha, vinodv}@mit.edu, chiraag.juvekar@analog.com

***Abstract*—Fully Homomorphic Encryption (FHE) offers protection to private data on third-party cloud servers by allowing computations on the data in encrypted form. To support general-purpose encrypted computations, all existing FHE schemes require an expensive operation known as "bootstrapping". Unfortunately, the computation cost and the memory bandwidth required for bootstrapping add significant overhead to FHE-based computations, limiting the practical use of FHE.**

In this work, we propose FAB, an FPGA-based accelerator for bootstrappable FHE. Prior FPGA-based FHE accelerators have proposed hardware acceleration of basic FHE primitives for impractical parameter sets without support for bootstrapping. FAB, for the first time ever, accelerates bootstrapping (along with basic FHE primitives) on an FPGA for a secure and practical parameter set. The key contribution of this work is the architecture of a balanced FAB design, which is not memory bound. In our design, we leverage recent algorithms for bootstrapping while being cognizant of the compute and memory constraints of our FPGA. In addition, we use a minimal number of functional units for computing, operate at a low frequency, leverage high data rates to and from main memory, utilize the limited on-chip memory effectively, and perform careful operation scheduling.

We evaluate FAB using a single Xilinx Alveo U280 FPGA and by scaling it to a multi-FPGA system consisting of eight such FPGAs. For bootstrapping a fully-packed ciphertext, while operating at 300MHz, FAB outperforms existing state-of-the-art CPU and GPU implementations by 213× and 1.5× respectively. Our target FHE application is training a logistic regression model over encrypted data. For logistic regression model training scaled to 8 FPGAs on the cloud, FAB outperforms a CPU and GPU by 456× and 9.5× respectively, providing practical performance at a fraction of the ASIC design cost.

I. Introduction

The last decade has seen rapid growth in data-centric applications. Given the sheer amount of data that is used by these applications, cloud services are commonly used to accelerate these applications. However, cloud services also introduce privacy concerns as it provides unrestricted data access to third-party cloud servers.

Fully homomorphic encryption (FHE) [46, 21] is currently regarded as the "gold standard" to preserve data privacy while enabling data processing in an untrusted cloud environment. FHE enables computing on *encrypted* data, allowing the cloud to operate on the data without having access to the data itself. Despite this incredible achievement of theoretical cryptography, the large compute and memory requirements of FHE remain a serious barrier to its widespread adoption. For example, to train a logistic regression (LR) model for 30 iterations using plaintext data (11,982 samples with 196 features), it takes ∼1.05sec on a CPU. The same LR model training on encrypted data takes ∼124mins on the same CPU [28], a slowdown of about 7086×. At the same time, the size of encrypted data (198MB) used in this training is 152× larger than the size of the plaintext data (1.3MB).

To address the compute and memory requirements of FHE, several optimizations and acceleration efforts are in progress. For FHE computing on CPUs, SEAL [51], PALISADE [52], HELib [32, 25], NFLLib [2], Lattigo [39], and HEAAN [35] software libraries accelerate various FHE schemes. Unfortunately, CPUs do not have the capability to adequately exploit the inherent parallelism available in FHE. GPU-based FHE implementations [16, 3, 34, 44] have had more success. These GPU-based implementations exploit the inherent parallelism in FHE, but the GPUs have massive floating-point units that are completely underutilized as FHE workloads consist almost entirely of integer-only operations. Moreover, neither CPU nor GPU can provide adequate main memory bandwidth to handle the data-intensive nature of FHE workloads.

Consequently, Samardzic et al. proposed custom FHE ASICs called F1 [49] and CraterLake [50], and Kim et al. proposed BTS [38] and ARK [36]. These ASIC solutions are promising as they outperform CPU/GPU implementations and narrow the gap between the performance of computing on plaintext vs. ciphertext. To achieve this performance improvement, they leverage a large number of custom modular multipliers, large register files, and large on-chip memory (as shown in Table I). However, to enable such large on-chip memory, these ASIC implementations need to use an expensive advanced technology node like 7nm or 12nm. Moreover, the design and fabrication of these ASIC proposals require several months of engineering effort, incurring large non-recurring engineering costs. In addition, the FHE algorithms are not standardized and are under active development. So changes to the FHE algorithms would require significant efforts to redesign an ASIC solution.

In this work, we adopt the middle path of using FPGAs as they enable us to design custom hardware solutions that provide practical performance and outperform CPU/GPU solutions. At the same time, FPGA solutions are comparatively inexpensive than ASIC solutions with a quick turnaround time for design updates, and thus, highly resilient to future FHE algorithm changes. We propose FAB, an FPGA-based accelerator for bootstrappable FHE that supports the Cheon-Kim-Kim-Song (CKKS) [14] FHE scheme. FAB makes use of state-of-the-art analysis of the bootstrapping algorithm [17] to design the FHE

TABLE I
RECENT ASIC DESIGNS FOR FHE.

Work	Parameters (N, $\log q$)	Modular multiplier count	On-chip register file (MB)	On-chip memory (MB)
F1 [49]	2^{14}, 32	18432	8	64
BTS [38]	2^{17}, 50	8192	22	512
CraterLake [50]	2^{16}, 28	14336	256	-
ARK [36]	2^{16}, 54	20480	76	512

operations and select parameters that are optimized for the hardware constraints. This allows FAB to support practical FHE parameter sets (i.e. parameters large enough to support bootstrapping) *without* being bottlenecked by the main-memory bandwidth. We evaluate FAB on a target application of training a logistic regression model over encrypted data. Our evaluation demonstrates that FAB outperforms all CPU/GPU implementations (9.5× to 456×) and provides practical execution latency.

We architect FAB for the Xilinx Alveo U280 FPGA accelerator card containing a High Bandwidth Memory (HBM2). FAB is highly resource efficient, requiring only 256 functional units, where each functional unit supports various modular arithmetic operations. FAB exploits maximal pipelining and parallelism by utilizing these functional units as per the computational demands of the FHE operations. FAB also makes efficient use of the limited 43MB on-chip memory and a 2MB register file to manage the >100MB working dataset at any given point in time. Moreover, FAB leverages a smart operation scheduling to enable high data reuse and prefetching of the required datasets from main memory without stalling the functional units. In addition, this smart scheduling evenly distributes the accesses to main memory to efficiently utilize the limited main memory bandwidth through homogeneous memory traffic. In summary, our contributions are:

- We propose FAB, a novel accelerator that supports all homomorphic operations, including fully-packed bootstrapping, in the CKKS FHE scheme for practical FHE parameters.
- FAB tackles the memory-bounded nature of bootstrappable FHE through judicious datapath modification, smart operation scheduling, and on-chip memory management techniques to maximize the overall FHE-based compute throughput.
- FAB outperforms all prior CPU/GPU works by 9.5× to 456× and provides a practical performance for our target application: secure training of logistic regression models.

The performance, cost, and flexibility of FAB suggest that FPGA provides a "sweet spot" for FHE acceleration. FAB significantly outperforms both CPU and GPU implementations of FHE. In contrast to ASICs, FAB only uses standard, commercially available hardware (FPGA) that is highly accessible to the general public (e.g. via the AWS F1 cloud) and can be deployed immediately for under a dollar per hour [5, 40]. By enabling competitive levels of performance with the same availability as a high-end GPU, FAB demonstrates that FPGAs are the most viable option for near-term hardware acceleration of FHE.

II. BACKGROUND

In this section, we briefly review the CKKS [14] homomorphic encryption scheme and the relevant parameters for FAB. A summary of these parameters is given in Table II.

A. The CKKS FHE Scheme

The CKKS [14] scheme supports operations over vectors of complex numbers. A plaintext in the CKKS scheme is an element of $\mathbb{C}^n$, where $\mathbb{C}$ is the field of complex numbers. The plaintext operations are component-wise addition and component-wise multiplication of elements of $\mathbb{C}^n$. In addition to the plaintext size n, CKKS is parameterized by a ciphertext coefficient modulus $Q \in \mathbb{Z}$ (where $\mathbb{Z}$ is the ring of integers) and a ciphertext polynomial modulus x^N+1, where N is chosen to be a power of 2. CKKS ciphertexts are elements of $\mathscr{R}_Q^2$, where $\mathscr{R}_Q := \mathbb{Z}_Q[x]/(x^N+1)$. We denote the encryption of a vector $\mathbf{m} \in \mathbb{C}^n$ by $[\![\mathbf{m}]\!] = (\mathbf{a_m}, \mathbf{b_m})$ where $\mathbf{a_m}$ and $\mathbf{b_m}$ are the two elements of $\mathscr{R}_Q$ that comprise the ciphertext.

CKKS supports the following operations over encrypted vectors. All arithmetic operations between two plaintext vectors are *component-wise*.

- $\mathsf{Add}([\![\mathbf{m}_1]\!], [\![\mathbf{m}_2]\!]) \rightarrow [\![\mathbf{m}_1 + \mathbf{m}_2]\!]$, where the addition is component-wise over $\mathbb{C}$.
- $\mathsf{Mult}([\![\mathbf{m}_1]\!], [\![\mathbf{m}_2]\!]) \rightarrow [\![\mathbf{m}_1 \odot \mathbf{m}_2]\!]$, where $\odot$ represents the component-wise product of two vectors.
- $\mathsf{Rotate}([\![\mathbf{m}]\!], k) \rightarrow [\![\phi_k(\mathbf{m})]\!]$, where ϕ_k is the function that rotates a vector by k entries. As an example, when $k=1$, the rotation $\phi_1(\mathbf{x})$ is defined as follows:

$$\mathbf{x} = \begin{pmatrix} x_0 & x_1 & \dots & x_{n-2} & x_{n-1} \end{pmatrix}$$
$$\phi_1(\mathbf{x}) = \begin{pmatrix} x_{n-1} & x_0 & \dots & x_{n-3} & x_{n-2} \end{pmatrix}$$

- $\mathsf{Conjugate}([\![\mathbf{m}]\!]) \rightarrow [\![\overline{\mathbf{m}}]\!]$ where $\overline{\cdot}$ represents the complex conjugate operation.

1) Homomorphic Levels & RNS Representation: To efficiently operate over elements of $\mathscr{R}_Q$, we represent Q as a product of primes $q_1, \dots, q_\ell$ where each q_i is roughly the size of a machine word. This follows the standard residue number system (RNS) representation of ciphertext moduli [26, 13]. We call each q_i a *limb* of the modulus Q, and we say a modulus

TABLE II
CKKS FHE PARAMETERS AND THEIR DESCRIPTION.

Parameter	Description
N	Number of coefficients in the ciphertext polynomial.
n	Number of plaintext elements in a ciphertext ($n \leq N/2$ is required).
Q	Full modulus of a ciphertext coefficient.
q	Prime modulus and a limb of Q.
L	Maximum number of limbs in a ciphertext.
ℓ	Current number of limbs in a ciphertext.
dnum	Number of digits in the switching key.
α	$\lceil (L+1)/\mathsf{dnum} \rceil$. Number of limbs that comprise a single digit in the key-switching decomposition. This value is fixed throughout the computation.
P	Product of the extension limbs added for the raised modulus. There are $\alpha+1$ extension limbs.
$\mathsf{fftIter}$	Multiplicative depth of a linear transform in bootstrapping.

$Q := \prod_{i=1}^{\ell} q_i$ has $\ell - 1$ *levels*. We call the set $\mathscr{B} := \{q_1, \ldots, q_\ell\}$ an *RNS basis*. Each multiplication operation reduces the size of the modulus by one limb. A modulus with ℓ levels can support a circuit of multiplicative depth ℓ before bootstrapping is required. Addition, rotation, and conjugation operations do not change the number of levels in a modulus.

This representation allows us to operate over values in $\mathbb{Z}_Q$ without any native support for multi-precision arithmetic. Instead, we can represent $x \in \mathbb{Z}_Q$ as a length-ℓ vector of scalars $[x]_{\mathscr{B}} = (x_1, x_2, \ldots, x_\ell)$, where $x_i \equiv x \pmod{q_i}$. We refer to each x_i as a *limb* of x. To add two values $x, y \in \mathbb{Z}_Q$, we have $x_i + y_i \equiv x + y \pmod{q_i}$. Similarly, we have $x_i \cdot y_i \equiv x \cdot y \pmod{q_i}$. This allows us to compute addition and multiplication over $\mathbb{Z}_Q$ while only operating over standard machine words.

In memory, ciphertext data can be viewed as an $\ell \times n$ matrix, where each row is a limb and each column corresponds to a single coefficient modulo Q. Arranging this matrix in "row-major order" where there is locality for elements in the same row is called *limb-wise* access as it enables efficient data access within a single limb. By contrast, arranging this matrix in "column-major order" where there is locality for elements in the same column is known as *slot-wise* access as it is best for accessing the same slot of data across all ciphertext limbs.

2) Number Theoretic Transform: To efficiently multiply elements of $\mathscr{R}_Q$, we make use of the number theoretic transform (NTT), which is the analog of FFT modulo q. All polynomials in the CKKS scheme are represented by default as a series of N evaluations at fixed roots of unity, allowing fast polynomial multiplications (in $O(N)$ time instead of $O(N^2)$). NTT is the finite field version of the fast Fourier transform (FFT) and takes $O(N \log N)$ time and $O(N)$ space for a degree-$(N-1)$ polynomial. We call the output of the NTT on a polynomial its *evaluation representation*. If any operations are to be done over the polynomial's *coefficient representation*, then we need to perform an inverse NTT (iNTT) to move the polynomial back to its *coefficient representation*.

We would like to note that in addition to NTT polynomial transform, the CKKS scheme also uses the FFT polynomial transform. On the client side, during CKKS encryption and decryption, a complex FFT must be run on vectors of complex numbers to map them to polynomials that can be encrypted. Correspondingly, on the cloud side during bootstrapping, this complex FFT (that was performed on the client side) must be *homomorphically* evaluated on the encrypted data.

3) Bootstrapping: In order to compute indefinitely on a CKKS ciphertext, there exists an operation known as bootstrapping [12] that raises the ciphertext modulus Q while maintaining the correct structure of the ciphertext. This operation is the main bottleneck for CKKS FHE. Efficient implementation of bootstrapping is the main focus of FAB.

We refer the reader to [17] for a detailed description of the bootstrapping algorithm that we use. At a high level, the bootstrapping operation consists of three major steps: a linear transform, a polynomial evaluation, and finally another linear transform (inverse of the first step). All these steps consist of the same homomorphic operations: Add, Mult, Rotate, and Conjugate. The polynomial evaluation is constrained by application parameters, and we use the same polynomial as Bossuat et al. [7] to support non-sparse CKKS secret keys. The multiplicative depth of this polynomial evaluation is 9. The two linear transforms in bootstrapping are FFT and inverse FFT, which must be homomorphically evaluated on the encrypted data. There is a depth-performance trade-off in this algorithm that has been carefully studied in prior works [11, 27, 17]. This trade-off is parametrized by the chosen multiplicative depth of the FFT algorithm, which we denote as fftIter. We provide a detailed discussion of the effects of this parameter in Section II-B.

4) Bootstrapping Performance Metric: After a bootstrapping operation, the resulting ciphertext can support ℓ computation levels before needing to be bootstrapped again. As each level corresponds to a multiplication, the performance metric [34] for bootstrapping is as follows:

$$T_{\mathsf{Mult},a/\mathsf{slot}} := \frac{T_{\mathsf{Boot}} + \sum_{i=1}^{\ell} T_{\mathsf{Mult}}(i)}{\ell \cdot n} \tag{1}$$

This is known as the *amortized multiplication time per slot*. Here, n is the number of slots in the ciphertext, T_{Boot} is the bootstrapping time, and $T_{\mathsf{Mult}}(i)$ is the time to multiply at level i. The number of levels ℓ in the resulting ciphertext are equal to the maximum supported levels in the starting bootstrapping modulus minus the depth of bootstrapping. The depth of our bootstrapping algorithm is $L_{\mathsf{Boot}} := 2 \cdot \mathsf{fftIter} + 9$.

5) Switching Keys & Major Subroutines: The Mult, Rotate, and Conjugate operations all produce intermediate ciphertexts that are decryptable under a different secret key than the input ciphertext. These operations all use a common subroutine known as KeySwitch [10] to switch the secret key back to the original value. KeySwitch is a major bottleneck in low-level homomorphic operations.

All instances of KeySwitch require a *switching key*, which is a special type of public key generated using the secret key. The KeySwitch operation takes in a switching key $\mathsf{ksk}_{\mathbf{s}\to\mathbf{s}'}$ and a ciphertext $[\![\mathbf{m}]\!]_{\mathbf{s}}$ decryptable under a secret key $\mathbf{s}$ and produces a ciphertext $[\![\mathbf{m}]\!]_{\mathbf{s}'}$ that encrypts the same message but can be decrypted under a different key $\mathbf{s}'$. We use the structure of the switching key proposed by Han and Ki [29], where the switching key is parameterized by a length dnum and is a $2 \times \mathsf{dnum}$ matrix of polynomials.

$$\mathsf{ksk} = \begin{pmatrix} \mathbf{a}_1 & \mathbf{a}_2 & \ldots & \mathbf{a}_{\mathsf{dnum}} \\ \mathbf{b}_1 & \mathbf{b}_2 & \ldots & \mathbf{b}_{\mathsf{dnum}} \end{pmatrix} \tag{2}$$

We omit the descriptions of individual CKKS subroutines and refer the reader to [17] for a thorough description of these subroutines. For reference, we make use of the Decomp, ModUp, ModDown, KeySwitchInnerProd (which we call KSKIP), and Automorph subroutines. During the course of key switching, the ciphertext coefficient modulus is raised from Q to $P \cdot Q$, where P is a fixed product of "extension limbs". The ciphertext modulus Q is split into at most dnum digits of equal size, and P must be larger than the largest product of the limbs in a single digit of Q. We refer the reader to [29] for

TABLE III
PARAMETER SET FOR FPGA IMPLEMENTATION.

$\log q$	N	L	dnum	fftIter	λ
54	2^{16}	23	3	4	128

more details and note that the parameter $P \cdot Q$ is the maximum modulus for which security must be maintained.

B. Practical Parameter Set for FAB

To prototype an efficient FAB design on the Xilinx Alveo U280 FPGA, we identify an optimal FHE parameter set that can support CKKS bootstrapping as well as the computational requirement of a realistic machine learning application. The parameter N must be a power of two for the efficiency of the NTT, and the largest power of two that still leaves the limbs small enough to fit in the on-chip memory is $N = 2^{16}$. Given this N, the maximum ciphertext modulus we can support is $\log(PQ) = 1674$, which achieves a 128-bit security level [4, 7]. These parameters also meet our FPGA's constraints. The maximum size of a single ciphertext is only 27.4MB (based on the maximum number of raised limbs, which is 31 limbs). We can fit an entire ciphertext in the limited on-chip memory (43MB) of our FPGA and thus limit the data movement to/from the main memory. Table III lists our choice of the other parameters based on the selected values for $\log(PQ)$ and N.

Limb Bit-Width. We fix the bit-width for each limb ($\log q$) as 54 bits for several reasons. First, a 54-bit limb width enables effective utilization of both the 18-bit multipliers and the 27-bit preadders within the DSP slices of the FPGA through multi-word arithmetic [31]. DSP slices have multipliers that are 18×27-bit wide. Using multi-word arithmetic, we can split 54-bit operands into multiple 18-bit operands and operate over them in parallel. To perform integer additions, we split the 54-bit operands into two 27-bit operands to leverage the 27-bit preadders in the DSP blocks. Second, a 54-bit limb width allows us to make the most of the scarce on-chip memory resources, which includes both UltraRAM (URAM) and Block-RAM (BRAM). On U280 cards, a URAM block can store 72-bit wide data and a BRAM block can store 18-bit wide data. Therefore, with 54-bit (a multiple of 18) coefficients in the vectors, we can effectively utilize the entire data width of the on-chip memory resources by combining multiple B/URAM blocks to store single/multiple coefficients at a given address. We discuss a detailed on-chip memory layout later in Section IV.

Higher-level Parameters. The dnum and fftIter parameters directly impact a number of factors that determine the final amortized multiplication time per slot, including the bootstrapping runtime as well as the number of compute levels available after bootstrapping. Figure 1 shows that as we increase the dnum value, we add more compute levels after bootstrapping, but at the same time, we increase the size of KeySwitch keys, further increasing the compute and on-chip memory requirements. At dnum = 3, we are able to make the best use of on-chip memory for the corresponding KeySwitch key size.

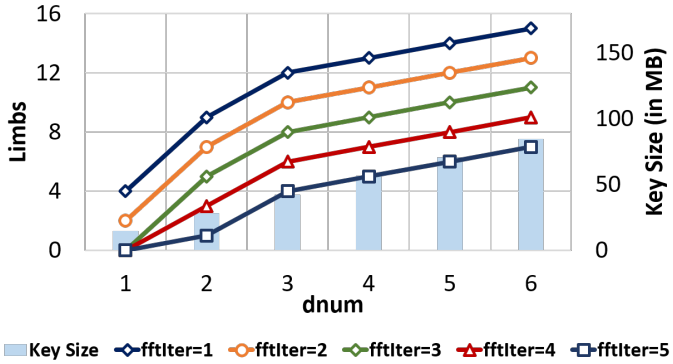

Fig. 1. Impact of changing the dnum parameter on the compute levels after bootstrapping and the switching key size. Note that we use the key-compression technique from [17] to halve the size of the keys.

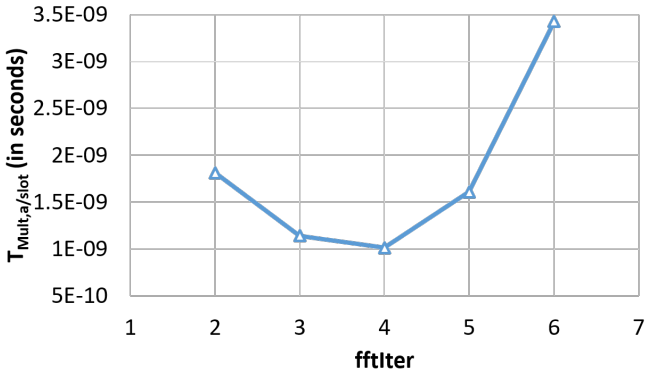

Fig. 2. Execution time of bootstrapping vs fftIter. For all benchmarks we set $N = 2^{16}$, $\log(PQ) = 1674$, $\log(q) = 54$, and dnum $= 3$.

As mentioned earlier in Section II-A, bootstrapping performs an FFT and an inverse FFT as the first and last steps of the algorithm. There is a depth-performance trade-off between the chosen multiplicative depth of the FFT algorithm and the amount of compute required for the bootstrapping operation. This trade-off is parameterized by fftIter, which is the multiplicative depth of the FFT algorithm. As the value of fftIter increases, the multiplicative depth also increases implying fewer compute levels after bootstrapping. However, as we increase fftIter, the radix of the FFT sub-matrices reduces, thus requiring a fewer number of rotations during each multiplication. Figure 2 shows how increasing the fftIter parameter impacts the overall bootstrapping execution time and the number of NTT operations to be computed. The metric used to measure bootstrapping time is the amortized per slot multiplication time in Equation 1. We pick fftIter $= 4$ as it generates an optimal balance between the computations (both rotations and NTTs) and the number of compute levels after bootstrapping. This fixes the total depth of our bootstrapping circuit as $L_{\text{Boot}} = 2 \cdot \text{fftIter} + 9 = 17$.

III. Overall System Architecture

In this section, we present the overall system architecture (system in the cloud) that uses our proposed FAB hardware accelerator. As shown in Figure 3, our overall system consists

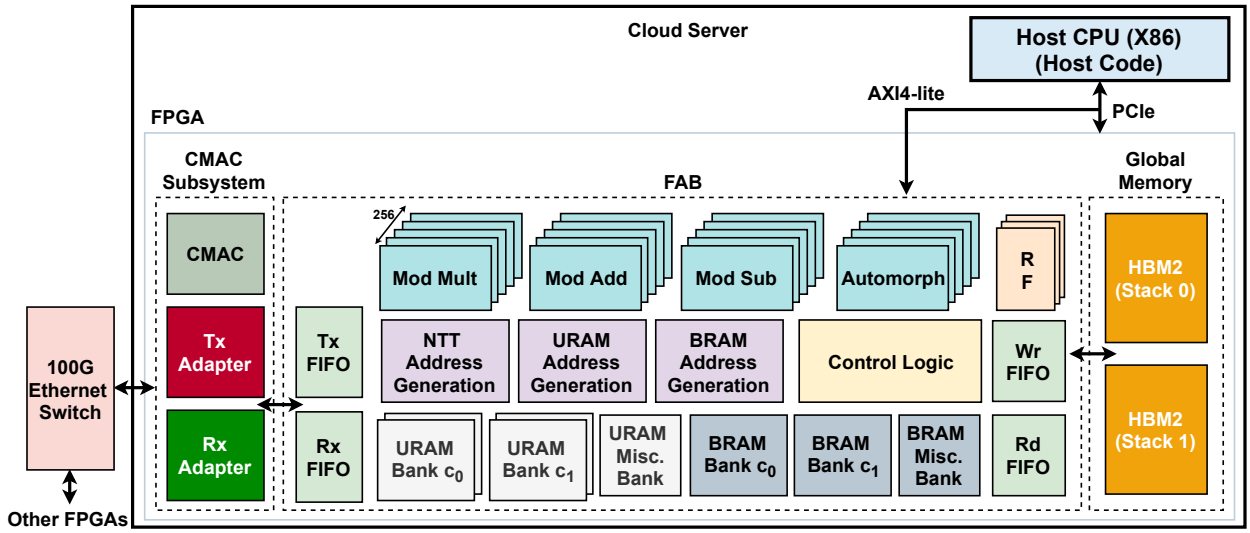

Fig. 3. An FPGA-based system for FHE-based computing. We map our FAB accelerator to the FPGA programmable logic. The host CPU interacts with FPGA via PCIe. CMAC subsystem enables interaction between multiple FPGAs via Ethernet Switch.

of four key components: a host CPU (X86 in our case) that offloads the FAB RTL design and data to the FPGA, FAB RTL design that is packaged as a kernel code, global memory on the FPGA comprising of two HBM2 stacks (4GB each), and a 100G Ethernet (CMAC) subsystem to enable transmit/receive data between FPGAs without involving the host. The host CPU is attached to the FPGA accelerator board (Alveo U280) via PCIe. This PCIe interface enables the data transfer between the host and the global memory on the FPGA board. To enable this data transfer, the host allocates a buffer of the dataset size in the global memory. The host code communicates the base address of the buffer to the kernel code using atomic register reads and writes through an AXI4-Lite interface. The host code also communicates all kernel arguments consisting of the system parameters like prime moduli, the degree of the polynomial modulus N, and the pre-computed scalar values (to be stored in the register file) through this interface. A kernel is started by the host code (written in native C++) using the Xilinx runtime (XRT) API call. It is worth noting that this XRT API call can be seamlessly replaced by an OpenCL [42] API call with trivial modifications to the host code. Once the kernel execution starts, no data transfer occurs between the host and the global memory so as to interface all 32 AXI ports from the HBM to the kernel code. The results are transferred back to the host code once the kernel execution completes.

The kernel code instantiates 256 functional units consisting of modular arithmetic and automorph units. A small register file (RF), 2MB in size, stores all the required system parameters and the precomputed scalar values that are received from the host. The RF also facilitates temporary storage of up to four polynomials that may be generated as intermediate results. The kernel has 32 memory-mapped 256-bit interfaces that are implemented using AXI4 master interfaces to enable bi-directional data transfers to/from the global memory. The read (Rd) FIFO and write (Wr) FIFO stream the data from global memory onto the on-chip memory and vice-versa. The URAM and BRAM resources on the FPGA are organized into various banks to be used as on-chip memory within the kernel code. All of the URAM memory banks are single-port banks as URAMs do not support dual ports, while the BRAM memory banks are dual-port banks. The transmit (Tx) and receive (Rx) FIFO stream the data to and from the CMAC subsystem.

The Alveo U280 FPGA has an integrated IP block for 100G Ethernet (CMAC) core, providing a 100Gb/s Ethernet port to transfer data between FPGAs that are connected to different hosts. The CMAC core has an internal clock operating at 322MHz and the data interface with the kernel code can be either 256/512-bit wide. We implement a 512-bit interface in our kernel code to keep up with 100Gbps transfer rates. This is because, with a 512-bit interface at 300MHz, we can theoretically process data at $\sim$153Gbps, which is faster than the Ethernet IP. In contrast, with a 256-bit interface, we can process data at $\sim$76Gbps, which is comparatively slower than the Ethernet IP and will end up dictating the final time it takes to transmit/receive the data to/from other FPGAs. With our 512-bit interface, it takes $\sim$11,399 clock cycles to transmit a single limb of the ciphertext (polynomial of size 0.4MB) and $\sim$546,980 clock cycles to transmit the entire ciphertext.

IV. FAB Microarchitecture

FAB consists of a functional unit, on-chip memory (URAM and BRAM), RF, FIFO, address generation units, and control logic. In this section, we discuss the microarchitecture of each of these units. These units are architected so as to achieve a balanced full-system design.

A. Functional Unit

All operations in FHE break down to integer modular arithmetic i.e., modular addition and modular multiplication operations. Therefore, each of the 256 functional units in FAB consists of modular multiplication, modular addition,

Algorithm 1 Modular Reduction in $\mathbb{F}_q$

1: Modulus q, integer a, $shifts = 6$ ▷ a is $(2\log q - 1)$ bits
2: Precompute: $\mathsf{madd}[i-1] = \sum_{j=0}^{5} i[j] \cdot 2^{\log q + j} \pmod q$ for $i = 1$ to $2^{shifts} - 1$ ▷ madd has $\log q$-bit elements
3: Set (A[1],A[0]) $\leftarrow a$, $count = 0$, $as_1 = 0$
4: **while** $count < \log q$ **do**
5: $(carry, as_1) = A[1] \ll shifts$ ▷ $carry$ is $shifts$ bit
6: $A[1] = as_1 + \mathsf{madd}[carry - 1]$
7: $count = count + shifts$
8: **end while**
9: $c = A[1] + A[0]$
10: **if** $c > q$ **then**
11: $c = c - q$
12: **end if**
13: **return** c

modular subtraction, and an automorph unit. As mentioned in Section II-B, we utilize a multi-word arithmetic approach to reduce 54-bit operations to 27-bit operations for addition and 18-bit operations for multiplication. This facilitates efficient utilization of specialized DSP arithmetic blocks on the FPGA.

For multi-word modular addition and subtraction, we follow Algorithms 2.7 and 2.8 respectively, proposed by Hankerson et al. [31]. Both of these algorithms require a correction step (on line 2 in Algorithms 2.7 and 2.8) for modular reduction, which leads to 54-bit addition/subtraction operations again. Subsequently, we modify the correction step in both the algorithms to perform multiple 27-bit operations instead. With multi-word arithmetic and all the pipeline registers in place for the DSP blocks, both algorithms perform modular addition and subtraction in 7 clock cycles.

The modular multiplication is accomplished by first multiplying the operands as integers, followed by a modular reduction in a pipelined fashion. For the integer multiplication, we follow the operand scanning algorithm (Algorithm 2.9 [31]) that adopts the schoolbook approach to perform multi-word multiplication. As we split our input 54-bit operands into three 18-bit operands, a naïve implementation of this algorithm will require 21 clock cycles to perform a single multiplication. Given the fact that most FHE workloads have 50% of the operations as integer multiplication, a latency of 21 clock cycles for a single integer multiplication is too high. Consequently, we perform loop unrolling on this algorithm and compute various multiplication operations in parallel, reducing the multiplication latency to 12 clock cycles while still adding all the required pipeline registers for DSP multipliers.

For modular reduction, we propose a hardware-friendly fast modular reduction algorithm by modifying Will and Ko's reduction technique [54]. As opposed to standard modular reduction approaches like Barrett reduction [6], which requires performing multiple expensive multiplication operations, Will and Ko's technique requires only shift and addition operations. It works by shifting a single bit of the input number, making it a good choice for inputs with smaller bit widths. For our $(2\log q - 1)$-bit wide number, Will and Ko's technique takes $2\log q$ cycles (for $\log q = 54$ it takes 108 cycles) to compute a modular reduction. To reduce this latency while leveraging the simplicity of their approach, we propose Algorithm 1 that can instead perform multiple bit shifts, requiring only 12 clock cycles for $\log q = 54$ for the modular reduction operation.

We set the number of shifts to 6 (line 1 in Algorithm 1) in our implementation, but it is worth noting that this algorithm is generic and can work with any number of bit shifts depending on the latency requirement and space constraints. This algorithm requires precomputing an array madd (line 2 in Algorithm 1) having 63 elements, where each element is $(\log q)/2$-bit wide. In our case, we need to perform modular reduction w.r.t 31 different primes, implying that we will need to precompute 31 such madd arrays requiring $<$ 7KB of storage space in total. However, this precompute is done offline, so there is no compute overhead associated with it. All other steps in the proposed algorithm can be performed using inexpensive shift and addition operations.

The final operation that forms part of the functional unit is Automorph, which performs permutation for the Rotate operation. The function of the automorph unit is to read a polynomial from the on-chip memory and store it in the register file in the permuted order as per the given rotation index k. Any original slot indexed by i in ciphertext maps to the rotated slot through the given automorphism equation:

$$\mathsf{new\_index}_k(i) = ((5^k - 1)/2) + 5 \cdot i \pmod N \qquad (3)$$

Due to the limited number of rotation indices (about 60 different values) being used in bootstrapping, we precompute and store the various powers of 5 corresponding to each of the rotation index k. The division by two is a simple bit-shift, and the reduction modulo N is significantly simplified because N is always a power of two. Thus, reduction modulo N can be achieved by simply performing the AND operation with $N-1$.

To summarize, the functional units in FAB are optimized for the available hardware to reduce resource overhead. They make effective use of high-performance multipliers and adders in DSP blocks to perform low-latency modular arithmetic. FAB efficiently utilizes these functional units through fine-grained pipelining and by issuing multiple scalar operations in a single cycle.

B. On-chip Memory

The Alveo U280 accelerator board has single-cycle access URAM and BRAM blocks. There are 962 blocks of URAM where each block is 288Kb and can be used as single-port memory. There are 4032 blocks of BRAM where each block is 18Kb and can be used as both single and dual-port memory. FAB uses a combination of single and dual-port memory banks constructed using URAM and BRAM blocks, providing a total capacity of 43MB and a 30TB/s internal SRAM bandwidth.

As shown in Figure 4-(a), each URAM block has a data width of 72 bits and a depth of 4096. We combine three such URAM blocks to achieve a data width of 216 bits. This allows us to store four 54-bit coefficients ($216/4 = 54$) at any given address. Consequently, we need to layout 64 of these 216-bit

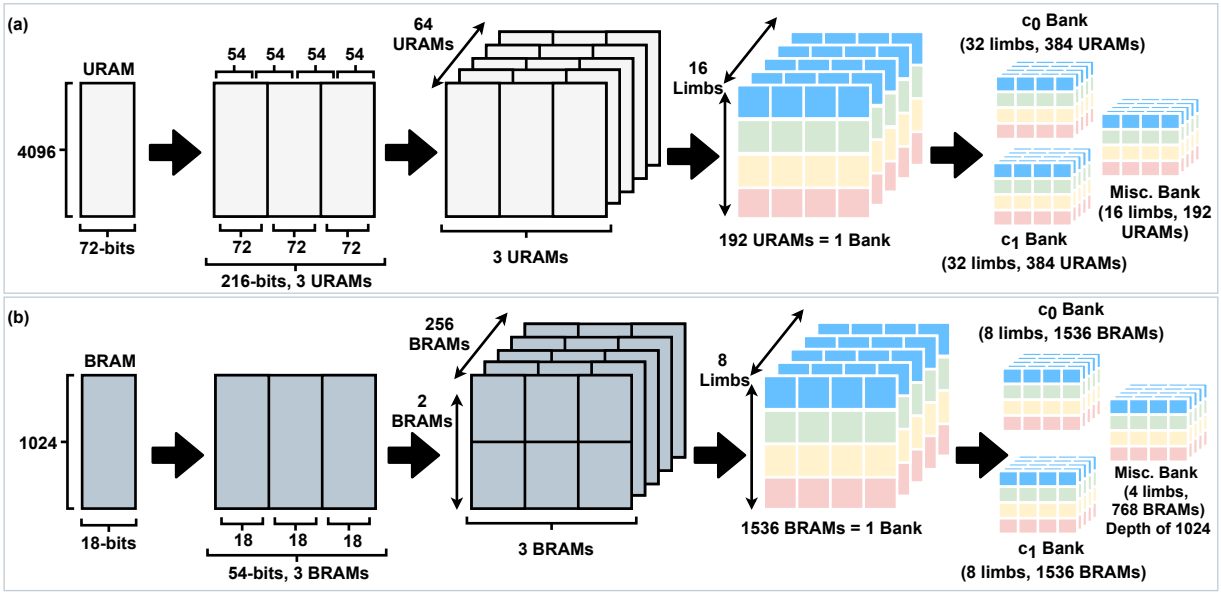

Fig. 4. On-chip memory configuration; (a) URAMs are organized as single-port memory banks and (b) BRAMs are organized as dual-port memory banks.

wide URAMs into a single memory bank to enable storage of 256 coefficients. Thus, with every read and write, we can access 256 coefficients in the same cycle, aligning with the number of functional units in the design. With this layout, a single memory bank consists of $64 \times 3 = 192$ URAMs and can store 16 polynomials ($\sim$7.08MB). We organize the available URAM blocks into five such banks that are divided as follows: The first two banks ($\mathbf{c}_0$ bank-1 and 2) store 31 limbs (23 original and 8 extension) of the $\mathbf{c}_0$ ring element of the ciphertext. The next two banks ($\mathbf{c}_1$ bank-1 and 2) store 31 limbs (23 original and 8 extension) of the $\mathbf{c}_1$ ring element of the ciphertext. The fifth bank can store 16 polynomials. We call the fifth bank the "miscellaneous" bank as it is used to store multiple data items such as twiddle factors, KeySwitch keys, and plaintext vectors that are read in from the main memory.

As shown in Figure 4-(b), BRAM blocks are organized as 54-bit wide memory banks by combining three 18-bit wide BRAMs. As each address can store only a single 54-bit coefficient, we need 256 BRAM blocks to store 256 coefficients. In addition, the depth of each BRAM block is only 1024; therefore, we stack two BRAM blocks to get a depth of 2048, thus enabling storage of 8 polynomials in a single BRAM bank. Similar to the URAM bank organization, we organize BRAM blocks into multiple banks. We have three BRAM banks in total, where two banks consist of 1536 BRAMs each and can store 8 polynomials and thus, are ideal to store the extension limbs. While the third bank consists of 768 BRAMs and can store 4 polynomials. We again call this third bank the "miscellaneous" bank and use it to store temporary data from the main memory during various operations.

To summarize, FAB efficiently utilizes the available U/BRAM blocks on the FPGA as on-chip memory. Mapping the data width of the polynomials to that of U/BRAM blocks enables storage of up to 43MB of data on-chip. FAB overcomes the limited main memory bandwidth by utilizing a combination of single and dual-port memory banks that complement the operational needs of the underlying FHE operations, resulting in a more balanced FPGA design.

C. Register File

Our design consists of multiple register files (RFs). The total capacity of all the RFs is 2MB. The RFs are spread across the design and are used by functional, address generation, and control units. Each RF has multiple read/write ports with single-cycle access latency. About one-fourth of the RF is used to store pre-computed values and system parameters, which are written by the host CPU through atomic writes before launching the kernel code execution. The remaining RFs are used to store up to four intermediate polynomials that are generated as part of Rotate or Mult operations.

D. FIFOs

We instantiate 32 synchronous Wr and Rd FIFOs (supporting 32 AXI ports on the HBM side) to stream the data between the main memory and the on-chip memory. These FIFOs are composed of the distributed RAM available on the FPGA board. The data width of each FIFO is equal to the data width supported by each AXI port i.e., 256 bits. The depth of the Wr FIFO depth is 128 to support an HBM burst length of 128. The depth of the Rd FIFO is 512 to support up to four outstanding reads. Rd FIFO is driven by the memory-side clock domain having a clock frequency of 450MHz, while the Wr FIFO is driven by the kernel-side clock domain having a clock frequency of 300MHz. We also instantiate a Transmit (Tx) and Receive (Rx) FIFO to stream the data between the CMAC subsystem and the on-chip memory. These are also synchronous FIFOs having a 512-bit data interface.

V. Datapath Optimizations

In this section we present out datapath optimizations for the most compute-intensive NTT and the most memory-intensive

KeySwitch operations of the CKKS scheme, while efficiently utilizing FAB microarchitecture.

A. NTT/iNTT datapath

Our NTT datapath uses a unified Cooley-Tukey algorithm [43] for both NTT and inverse-NTT (iNTT). Using a unified NTT algorithm provides the convenience of leveraging the same data mapping logic for both NTT and iNTT. The 256 modular addition, subtraction and multiplication units operate in parallel as radix-2 butterfly units, processing 512 coefficients of a polynomial at once. This allows us to perform $\log N$ stages in approximately $\log N \cdot \frac{N}{512}$ cycles instead of $\log N \cdot \frac{N}{2}$ cycles. The NTT address generation unit (shown in Figure 3) takes care of uniquely mapping the data within each stage of the NTT/iNTT using a sub-unit, i.e. a data mapping unit. Furthermore, a twiddle factor mapping sub-unit within the NTT address generation unit takes care of reading the required twiddle factors for an NTT stage from the URAM miscellaneous bank. Both of these sub-units leverage the data and stage counters to generate the addresses on-the-fly using inexpensive shift, and AND operations. Thus, we efficiently leverage pipelining and parallelism while computing NTT/iNTT by distributing the computations over the functional units, a data mapping unit, and a twiddle factor mapping unit. It is worth noting that we do not take into account the latency of the bit-reversal operation here as bit-reversal is carried out along with automorph/multiplication operation that is performed just before NTT/iNTT.

B. KeySwitch datapath

A KeySwitch operation comprises of four sub-operations, i.e., Decomp, ModUp, KSKIP, and ModDown. With limited on-chip memory, these operations require smart operation scheduling to efficiently utilize the on-chip memory. This is because KeySwitch not only needs to operate on the extension limbs (the factors of P) but it also needs to perform inner product with the KeySwitch keys that are almost $3\times$ the size of our ciphertext. Below we present a description of how we schedule and reorganize the sub-operations in KeySwitch to manage $\sim$112MB of data (84MB keys and 28MB ciphertext) within the available 43MB on-chip memory without writing any resultant limbs back to the main memory.

The Decomp sub-operation splits the limbs in $\mathbf{a_m}$ ring element (ciphertext has two ring elements $\mathbf{a_m}$ and $\mathbf{b_m}$) into dnum digits. Each of these digits is then passed to the ModUp sub-operation. ModUp outputs $L+1+\alpha$ limbs; after all ModUp operations are complete we have $\text{dnum} \cdot (L+1+\alpha)$ limbs in total. In our case, $\text{dnum} = 3$, $L = 23$ and $\alpha = 8$. This results in $3 \cdot 31 = 93$ total limbs as the output of the ModUp sub-operation. The KSKIP step is an inner product between the input polynomial and the switching key in the raised basis. KSKIP takes as input these 93 limbs and performs an inner-product with the corresponding 93 limbs of KeySwitch keys, resulting in a ciphertext with $L+\alpha = 31$ limbs. ModDown follows almost a similar sequence of operations as ModUp and reduces the ciphertext limbs back to $L = 23$.

To manage all 93 limbs in on-chip memory (without having to write any resultant limb back to main memory), we modify the KeySwitch datapath to reorganize the execution of sub-operations in KeySwitch. Naïvely, one would execute the sub-operations one after another following the original datapath shown in Figure 5 (a). However, there are challenges associated with this approach.

In the original KeySwitch data path, we perform the full ModUp step to extend the RNS basis of the input polynomial, then the full KSKIP. There is not enough space in the on-chip memory to hold all the extended limbs, and so the original datapath first reads in the input limbs in evaluation representation to perform the Decomp step, then writes the limbs to main memory in coefficient representation after the ModUp step. These limbs are then read back into the on-chip memory to perform the KSKIP step. However, KSKIP needs to perform inner product in evaluation representation, requiring an expensive NTT operation. Moreover, KSKIP needs to read all 93 limbs of the KeySwitch keys at once, with a single limb read latency of 300 cycles from the main memory. As there is not enough spare on-chip memory to read multiple key limbs in advance and an inner product takes only 275 cycles to finish, we can end up introducing bubbles in the compute pipeline while performing KSKIP sub-operation. FAB addresses the above-mentioned challenges by modifying the datapath (refer Figure 5 (b)) and using smart operation scheduling.

Modified datapath: We modify the original KeySwitch datapath so as to split the KSKIP step into two steps (step 2 and step 4 in Figure 5) (b). Rather than performing the KSKIP step all at once, we "greedily" make progress on the inner product by performing the multiplications and additions as soon as the operands are in memory, which reduces DRAM transfers.

We begin with L limbs in $\mathbf{a_m}$ that are in evaluation representation. The Decomp step (step 1 in Figure 5 (b)) takes these L limbs and splits them into $\beta \leq \text{dnum}$ blocks of α limbs each. These α limbs then take two paths. First, these α limbs are used to begin the KSKIP (step 2 in Figure 5 (b)), and the intermediate sum is written out to URAM. We can perform this KSKIP operation as the α limbs are still in evaluation representation. In the second path, these α limbs are input to the iNTT step (step 3.1 onwards in Figure 5 (b)) so that the extension limbs can be generated. Once these extension limbs are generated, they are used to complete the KSKIP step (step 4 in Figure 5 (b)). Therefore, with this datapath modification, we need not write the limbs to off-chip memory in coefficient representation after the ModUp step and then read them back again into the on-chip memory and convert the limbs into evaluation representation to perform the KSKIP. **Thus, the modified datapath not only reduces the number of NTT computations (the most expensive subroutine in KeySwitch operation) but also helps alleviate the memory bandwidth bottleneck by reducing memory traffic.**

We note that splitting KSKIP into two steps does not change the KeySwitch algorithm, only the order in which the steps are

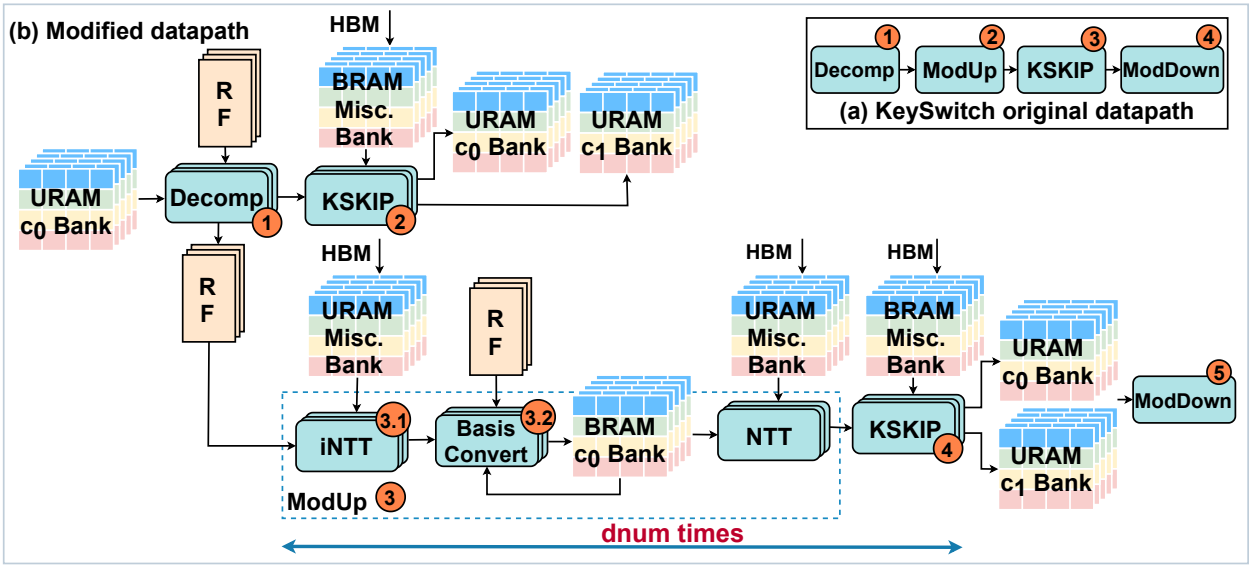

Fig. 5. (a) Original datapath. (b) Modified datapath for efficient on-chip memory utilization for sub-operations (Decomp, ModUp, KSKIP, and ModDown) within the KeySwitch Operation. Our approach avoids reads/writes of the ciphertext limbs to the main memory, which lowers the latency of FHE computing.

performed changes. The resulting noise from the KeySwitch algorithm is identical with or without this reordering.

Smart operation scheduling: Smart operation scheduling complements the modified datapath by generating an entire block of extension limbs at once for the given original α limbs. Altogether we have dnum blocks of α limbs for which extension limbs must be generated during ModUp. If we generate all extension limbs at once for all dnum blocks of α limbs, we will not have enough space in on-chip memory to store all the extension limbs. So, instead, as soon as all the extension limbs for one block of α limbs are computed, we perform an NTT followed by KSKIP on these limbs before generating the next block of extension limbs. This process is repeated dnum times to generate all $\text{dnum} \cdot (L+1+\alpha)$ limbs.

Our smart operation scheduling has several advantages. First, from Figure 4, we know that BRAM c_0 bank can store only $\alpha = 8$ limbs. Therefore, by repeating ModUp+KSKIP step dnum times for α limbs at once, we can manage the generation of all extension limbs using limited capacity (α-limb) BRAM c_0 bank. Moreover, as the BRAM c_0 bank is dual-ported and all extension limbs are always stored in this bank, it supplements BasisConvert operation (step 3.2 in Figure 5 (b)) by allowing data reads and writes at the same time as we perform inner products in *limb-wise* fashion. Thus, we eliminate the need to switch between *limb-wise* and *slot-wise* accesses, which is one of the key memory bottlenecks in KeySwitch operation. Another advantage is that we do not need to fetch all the KeySwitch keys at once; we only need to fetch the block of the key corresponding to the α block that has been just extended. This is feasible as we schedule the KSKIP in between extension limb generation (instead of generating the extension limbs all at once). Through these methods, we perform the entire KSKIP using a small BRAM miscellaneous bank (that can store only 4 limbs), while hiding the key read latency (which is about 300 clock cycles) from the main memory behind the computations. The ModDown (step 5 in Figure 5 (b)) operation follows a similar operation scheduling as ModUp operation. **Thus, through smart scheduling we enable high data reuse, exploit inherent *limb-wise* parallelism and maintain a uniform address generation logic by avoiding switching between *limb-wise* and *slot-wise* accesses, and further reduce the main memory traffic by not writing/reading any resultant ciphertext limbs to the main memory.**

VI. Evaluation

We have designed FAB in Verilog 2001 and synthesized it using Xilinx Vivado 2020.2 to operate at 300MHz frequency. The host CPU code is written in C++. FAB RTL code is packaged into kernel code using the Xilinx Vitis 2020.2 development platform. The kernel code is compiled and linked into an FPGA executable (.xclbin binary file) by the Vitis compiler. In the cloud environment, this binary file is mapped to a Xilinx Alveo U280 FPGA and its execution is initiated by the host CPU via host code. This accelerator FPGA card is built on the Xilinx 16nm UltraScale architecture and offers 8GB of HBM2 with up to 460GB/s bandwidth.

A. Resource Utilization

Table IV provides the hardware resource utilization of the various components of FAB. Overall, FAB requires ∼899K LUTs, and the functional units have the largest share (∼37%) of these LUTs. Out of the 2,073K flip flops (FFs) used by FAB, most of them are utilized by the distributed register file, control logic, and the functional unit. The entire 56.7% of the DSP utilization is for the modular arithmetic operations within the functional units. As mentioned in Section IV-B, FAB uses almost the entire URAM and BRAM blocks on FPGA, with a 95.24% BRAM utilization and 99.8% URAM utilization.

B. Basic FHE Operations

Table V presents the execution time (in ms) for basic operations in the CKKS FHE scheme and compares the performance with the existing GPU implementation [34]. Note that state-of-the-art CPU implementation [7] does not give low-level benchmarks for the individual homomorphic operation. The GPU numbers used in the table are the most optimized performance numbers reported in [34] for the parameter set $N = 2^{16}$, $\log Q = 1693$, and 100b security. We compare against these numbers as the parameter set is closest to our parameter set. FAB achieves an average 2.4× speedup when compared to GPU in absolute execution time for these basic primitives. For completeness, in Table VI, we compare the performance of NTT and Mult with an existing state-of-the-art FPGA implementation (HEAX [45]). For a fair comparison, we use the same parameter set ($N = 2^{14}$ and $\log Q = 438$) as used in HEAX. FAB achieves an average 3× higher throughput (measured in operations per second) when compared to HEAX. The performance gain in FAB is largely due to low latency modular arithmetic modules within functional units, fine-grained pipelining of the functional units, highly optimized NTT datapath, and the modified KeySwitch datapath.

C. Bootstrapping Latency

As described earlier, the bootstrapping operation is the key bottleneck for performing unbounded FHE computations. In this section, we compare the bootstrapping latency of FAB with the existing state-of-the-art CPU, GPU, and ASIC implementations. Throughout this section, we refer to these implementations as Lattigo [7] (CPU implementation) and GPU-1 for 97-bit security and GPU-2 for 173-bit security (GPU implementations of [34]). For comparison, we use the amortized per slot multiplication time $T_{\mathsf{Mult},a/slot}$ defined in Equation 1. This is the same bootstrapping performance metric used by these existing works. As seen in Table VII, FAB outperforms CPU, GPU-1, and GPU-2 implementations in absolute time by at least 1.5×. This is despite the lower operating frequency of FAB, and when comparing the clock cycles against prior works FAB compares even more favorably. FAB exhibits at least a 6.14× better performance due to improved arithmetic intensity by overcoming memory bandwidth bottleneck.

In Table VII, for completeness, we also compare FAB against ASIC designs for bootstrapping operation. FAB is about 9× (absolute time) and 4× (clock cycles) slower than the best case numbers reported by BTS-2 [38]. This difference in performance is mainly because of the large on-chip memory and a large number of modular multipliers used for computation in BTS when compared to FAB. FAB is about 3× (absolute time) slower than numbers reported by CraterLake [50] (abbreviated as CL in Table VII) for their parameters achieving 128-bit security. However, CraterLake's performance is comparable to FAB in clock cycles implying that the improvement in absolute time only comes from the higher operating frequency (1GHz) in CraterLake. FAB is about 30× (absolute time) and 10× (clock cycles) slower than ARK [36]. ARK achieves better performance with its algorithmic optimization that reduces the number of switching keys required in bootstrapping. However, this optimization relies on the 512MB on-chip memory and is not easily portable to other designs. We do not compare FAB against F1 [49] as F1 does not support parameters large enough for fully-packed bootstrapping.

TABLE IV
FAB HARDWARE RESOURCE UTILIZATION

Resource	Available	Utilized	% Utilization
LUTs	1,304K	899,232	68.96
FFs	2,607K	2,073K	79.54
DSP	9,024	5,120	56.70
BRAM	4,032	3,840	95.24
URAM	962	960	99.80

TABLE V
EXECUTION TIME (IN MILLISEC) FOR PERFORMING BASIC CKKS FHE OPERATIONS AND SPEEDUP ACHIEVED USING FAB.

Operation	FAB	GPU [34]	Speedup vs GPU
Add	0.04	0.16	3.85×
Mult	1.71	2.96	1.73×
Rescale	0.19	0.49	2.62×
Rotate	1.57	2.55	1.62×

TABLE VI
THROUGHPUT (IN OPERATIONS PER SECOND) COMPARISON FOR BASIC OPERATIONS WITH HEAX. PARAMETER SET USED $N = 2^{14}$ AND $\log Q = 438$.

Operation	FAB	HEAX [45]	Speedup vs HEAX
NTT	167K	42K	3.97×
Mult	5.7K	2.6K	2.12×

D. Logistic Regression (LR) Training

In this section, we evaluate the use of FAB to perform LR model training for binary classification over a subset of MNIST data [18] labeled 3 and 8. This is the task considered in the HELR work [28], and it is the same task used to benchmark all works we compare against. This subset of the dataset contains 11,982 training samples where each sample has 196 features. The LR model is trained for 30 iterations with 1024 encrypted images in a mini-batch. We adopt the sequence of operations proposed by Han et al. [28] for efficient logistic regression training on encrypted data. Using this algorithm, LR training for 30 iterations has an evaluation depth of 150, and thus, it requires us to perform a bootstrapping operation after every iteration. We perform training using sparsely-packed ciphertexts (using 256 slots only). This is largely to perform a fair comparison with existing works that have only considered sparsely-packed ciphertexts (256 slots) for LR training. This is because 256 slots are optimal for the specific benchmark task, but our LR training implementation can easily scale to larger applications (i.e. applications that require fully-packed bootstrapping).

To evaluate LR training, we present two different FPGA designs here; 1) FAB-1: a single-FPGA design and 2) FAB-2: a multi-FPGA design that utilizes eight FPGAs. Note that FAB-2 will incur eight times the resource utilization as FAB-1. We

TABLE VII
SPEEDUP ACHIEVED USING FAB WHEN PERFORMING BOOTSTRAPPING OPERATIONS. SLOTS DEFINE THE NUMBER OF PACKED SLOTS IN CIPHERTEXT WHILE BOOTSTRAPPING. BOOTSTRAPPING TIME IS COMPUTED IN $T_{mult,a/slot}$.

Work	Freq. (in GHz)	Slots	Time (in μs)	Speedup achieved (Time)	Speedup achieved (Cycles)
Lattigo [7]	3.5	2^{15}	101.78	213×	2485×
GPU-1 [34]	1.2	2^{15}	0.716	1.50×	6.14×
GPU-2 [34]	1.2	2^{16}	0.740	1.55×	6.35×
BTS-2 [38]	1.2	2^{16}	0.0455	0.09×	0.38×
CL [50]	1	2^{15}	0.13	0.27×	0.91×
ARK [36]	1	2^{15}	0.014	0.03×	0.10×
FAB	0.3	2^{15}	0.477	-	-

follow the data partitioning and packing technique proposed in Han et al. [28] to pack the data efficiently into ciphertexts. Our FAB-1 design is a straightforward mapping of FAB onto an Alveo U280 board with all the ciphertexts and KeySwitch keys (∼6.65GB data) offloaded to the HBM2. For the FAB-2 design, we instantiate FAB on all eight FPGA boards in the cloud environment, where each FPGA is connected to a host CPU. We form FPGA pairs, and in each pair, we designate a primary FPGA and a secondary FPGA to enable point-to-point communication between the FPGAs. In addition, one of the eight FPGAs acts as a master FPGA that can broadcast a ciphertext to the entire pool of FPGAs. We limit the impact of network communication on the performance of our FAB-2 design by using direct network communication between the FPGAs instead of involving the host CPUs. All the host CPUs launch the required kernel code on their respective FPGAs in parallel, which allows multiple ciphertexts to be processed in parallel. Since the ciphertexts are sparsely-packed, each FPGA needs to compute on 128 ciphertexts. Communication between the FPGAs is required only twice during an entire LR iteration. This communication overhead is about 12ms per LR iteration.

Table VIII presents the average training time per LR iteration. In terms of absolute execution times, FAB-2 is 456× and 9.5× faster than existing state-of-the-art CPU and GPU implementations. When comparing to ASIC proposals, FAB-2 outperforms F1 by 12× while achieving a competitive performance when compared to BTS-2. However, ARK achieves about 10× better performance than FAB. We do not compare against CraterLake as we do not know their average training time per iteration. They do not explicitly specify the number of iterations that were used to train the logistic regression model.

Compared to FAB-1, FAB-2 (using eight FPGAs) does not observe a corresponding 8× speedup as the amount of parallelism that can be extracted is limited by the bootstrapping runtime (following Amdahl's law). FAB is designed to perform bootstrapping on a single FPGA, so the performance increase is due to parallelizing the other operations within a logistic regression iteration. As part of future work, we would like to explore the possibility of improving the performance of FAB-2 by scaling bootstrapping operation to multiple FPGAs through instruction-level parallelism. This will require

TABLE VIII
PERFORMANCE COMPARISON FOR LR TRAINING WHEN USING SPARSELY-PACKED CIPHERTEXTS [38]. TIME REPORTED HERE IS THE AVERAGE TRAINING TIME PER ITERATION.

Work	Time (in sec)	Speedup achieved using FAB-2 (Time)	Speedup achieved using FAB-2 (Cycles)
Lattigo [7]	37.05	456×	5318×
GPU-2 [34]	0.775	9.5×	39×
F1 [49]	1.024	12×	41×
BTS-2 [38]	0.028	0.3×	1.4×
ARK [36]	0.008	0.01×	0.33×
FAB-1	0.103	1.3×	1.3×
FAB-2	0.081	-	-

distributing the operations (on a single ciphertext) to multiple FPGAs while minimizing data hazards and managing the large communication overhead.

Comparison with Leveled FHE Approach: As mentioned earlier, we use bootstrapping to periodically denoise the ciphertext. We briefly compare our bootstrapping-based FHE approach against the leveled FHE approach. The leveled FHE approach avoids bootstrapping by having the cloud host send a ciphertext with no remaining compute levels back to the client, who holds the decryption key. The client then decrypts the ciphertext and re-encrypts the resulting message into a new ciphertext with some fixed number of compute levels. This new ciphertext is sent back to the cloud host to proceed with the homomorphic computation.

A single iteration of logistic regression consumes 5 compute levels and requires bootstrapping at the end of each iteration. For logistic regression training with the leveled FHE approach, a single logistic regression iteration without bootstrapping takes 25.8ms. The encryption and decryption on the client (using the sub-routines in the SEAL library) take 162ms and 8.8ms with a 2.8GHz CPU respectively. Data transfer between the cloud host and FPGA (back and forth) via PCIe takes 40ms and data transfer between cloud host and the client takes 12.7 seconds. Hence, the total time for a single logistic regression iteration is 12.937 seconds. In our FAB design, a single logistic regression iteration with bootstrapping using eight FPGAs takes just 0.081 seconds, which is 160× faster than the leveled FHE approach.

Applicability to Other Schemes: Although FAB implements the CKKS scheme-specific bootstrapping, our implementations of the basic operations such as Add, Mult, and Rotate that are common across schemes can be used for the BGV [9] and B/FV [8, 20] schemes. FHE schemes like TFHE [15] and FHEW [19] evaluate Boolean gates on encrypted data while incurring several GB memory footprint for encrypted keys [24]. Even for these schemes, optimizations similar to our KeySwitch datapath and smart operation scheduling are very relevant. However, the specific steps within the KeySwitch operation differ across schemes, and so a thorough analysis is required to determine the exact operation scheduling so as to develop a balanced FPGA design.

Porting FAB to Other FPGAs: Broadly, the optimizations that we have proposed in this paper are generic and can be used as is when porting FAB to other FPGAs. For example, we can port the FAB design without any modification to the latest Intel Agilex M series FPGA [1], which has ∼48 MB on-chip memory, 3.9 million LUTs, 12K DSP blocks, and provides access to HBM. These resources are more than enough for our FAB design. There are other smaller FPGAs like Xilinx Alveo U50 with HBM access, which do not have enough on-chip resources. So one cannot map our FAB design as is, but our optimization ideas like KeySwitch datapath optimization and smart operation scheduling can still be leveraged when designing a solution for these smaller FPGAs.

VII. Related Work

CPU/GPU-based Acceleration: Many software libraries such as SEAL [51], HELib [32, 25], PALISADE [52], Lattigo [39], and HEAAN [35] implement the CKKS scheme on CPU. Despite these efforts, a pure-CPU implementation of FHE remains impractical. Several works [44, 37, 56, 23] focus on accelerating just the NTT computation on a GPU. Jung et al. [34] proposed the first GPU implementation of the CKKS scheme that includes basic operations as well as bootstrapping.

FPGA-based Acceleration: HEAX [45] was one of the early FPGA-based HE accelerators, but it accelerates only CKKS encrypted multiplication. Other operations are deferred to a host processor. Similarly, there are a couple of other limited FPGA-based CKKS acceleration efforts that only implement NTT [55] and KeySwitch [30] operations. All three FPGA implementations support only smaller parameter sets, which is not sufficient for bootstrapping or applications such as logistic regression training or ResNet. Given these limitations, it is unclear how to implement a full-scale CKKS FHE workload and an end-to-end application on an FPGA using HEAX. There are several other FPGA-based acceleration efforts [47, 48, 53, 41] that implement basic FHE operations for BFV scheme.

ASIC-based Acceleration: Samardzic et al. presented the F1 [49] and CraterLake [50] hardware accelerator architectures for FHE computations. Although the F1 accelerator supports multiple FHE schemes including BGV [22], CKKS, and GSW [33], it only supports operations on smaller parameter sets. To support multi-slot bootstrapping and larger real-time applications (e.g. machine learning), larger parameters are required, making the F1 chip unsuitable for these applications. CraterLake implements packed bootstrapping for large parameters using a large number of modular multipliers in their compute block. However, it observes a poor utilization ($\sim 40\%$) of the functional units during bootstrapping.

Kim et al. proposed the BTS [38] and ARK [36] accelerators specifically tailored to support CKKS bootstrapping. Both BTS and ARK employ a massive number of processing elements to exploit parallelism in various homomorphic operations and make use of large on-chip memory. BTS, however, does not modify the underlying algorithm to fully utilize the available memory bandwidth. This results in unrealized performance when compared to the resources BTS uses. ARK includes an algorithmic optimization that reduces the memory bandwidth of CKKS bootstrapping by reducing the number of switching keys required in the homomorphic FFT evaluation. However, this optimization crucially relies on a large on-chip memory (at least a few hundred MB).

Broadly, ASIC solutions will always have higher performance than FPGA solutions, but ASIC solutions are a lot more expensive than FPGA solutions. In addition, ASIC solutions are not future-proof and will require a non-trivial amount of redesign as the FHE algorithms evolve in the future.

VIII. Conclusion

We propose FAB, an FPGA-based accelerator for bootstrappable FHE. FAB leverages a combination of algorithmic and architectural optimizations to overcome the memory bandwidth bottleneck and to perform the first-ever fully-packed bootstrapping on FPGA for a practical parameter set. Through datapath optimizations and smart operation scheduling, FAB efficiently utilizes the limited compute and memory resources on the FPGA to deliver 456× and 9.5× better performance than CPU and GPU, respectively, for LR training application. More importantly, for the same LR training application FAB uses only currently-existing hardware while delivering practical performance at a fraction of ASIC design costs for bootstrappable FHE. FAB is also immediately accessible to the general public as all the resources required to support FAB exist in public commercial cloud environments.

Acknowledgement

This research was supported by RedHat Collaboratory, and in part by DARPA under Agreement No. HR00112020023 and by an NSF grant CNS-2154149.

References

[1] "Intel Agilex M Series," https://www.intel.com/content/www/us/en/products/details/fpga/agilex/m-series.html, 08 2022, Intel, Santa Clara, CA.

[2] C. Aguilar-Melchor, "Nfllib: Ntt-based fast lattice library," in *Cryptographers' Track at the RSA Conference*. Springer, 2016, pp. 341–356.

[3] A. Al Badawi, B. Veeravalli, C. F. Mun, and K. M. M. Aung, "High-performance fv somewhat homomorphic encryption on gpus: An implementation using cuda," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 70–95, 2018.

[4] M. R. Albrecht, R. Player, and S. Scott, "On the concrete hardness of learning with errors," Cryptology ePrint Archive, Report 2015/046, 2015, https://ia.cr/2015/046.

[5] M. Asiatici and P. Ienne, "Large-scale graph processing on fpgas with caches for thousands of simultaneous misses," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 609–622.

[6] P. Barrett, "Implementing the Rivest-Shamir-Adleman public key encryption algorithm on a standard digital signal processor," in *Advances in Cryptology — CRYPTO' 86*, A. M. Odlyzko, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1987, pp. 311–323.

[7] J.-P. Bossuat, C. Mouchet, J. Troncoso-Pastoriza, and J.-P. Hubaux, "Efficient bootstrapping for approximate homomorphic encryption with non-sparse keys," in *Advances in Cryptology – EUROCRYPT 2021*, A. Canteaut and F.-X. Standaert, Eds. Cham: Springer International Publishing, 2021, pp. 587–617.

[8] Z. Brakerski, "Fully homomorphic encryption without modulus switching from classical gapsvp," in *Advances in Cryptology – CRYPTO 2012*, R. Safavi-Naini and R. Canetti, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 868–886.

[9] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) fully homomorphic encryption without bootstrapping," in *ITCS '12*, 2012.

[10] 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

[11] H. Chen, I. Chillotti, and Y. Song, "Improved bootstrapping for approximate homomorphic encryption," in *Advances in Cryptology – EUROCRYPT 2019*, Y. Ishai and V. Rijmen, Eds. Cham: Springer International Publishing, 2019, pp. 34–54.

[12] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "Bootstrapping for approximate homomorphic encryption," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2018, pp. 360–384.

[13] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "A full RNS variant of approximate homomorphic encryption," in *Selected Areas in Cryptography – SAC 2018*, C. Cid and M. J. Jacobson Jr., Eds. Cham: Springer International Publishing, 2019, pp. 347–368.

[14] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *Advances in Cryptology – ASIACRYPT 2017*, T. Takagi and T. Peyrin, Eds. Cham: Springer International Publishing, 2017, pp. 409–437.

[15] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Tfhe: fast fully homomorphic encryption over the torus," *Journal of Cryptology*, vol. 33, no. 1, pp. 34–91, 2020.

[16] W. Dai and B. Sunar, "cuhe: A homomorphic encryption accelerator library," in *International Conference on Cryptography and Information Security in the Balkans*. Springer, 2015, pp. 169–186.

[17] L. de Castro, R. Agrawal, R. T. Yazicigil, A. P. Chandrakasan, V. Vaikuntanathan, C. Juvekar, and A. Joshi, "Does fully homomorphic encryption need compute acceleration?" *CoRR*, vol. abs/2112.06396, 2021. [Online]. Available: https://arxiv.org/abs/2112.06396

[18] L. Deng, "The mnist database of handwritten digit images for machine learning research [best of the web]," *IEEE signal processing magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[19] L. Ducas and D. Micciancio, "Fhew: bootstrapping homomorphic encryption in less than a second," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2015, pp. 617–640.

[20] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," Cryptology ePrint Archive, Report 2012/144, 2012, https://ia.cr/2012/144.

[21] C. Gentry, *A fully homomorphic encryption scheme*. Stanford university, 2009.

[22] C. Gentry, S. Halevi, C. Peikert, and N. P. Smart, "Ring switching in bgv-style homomorphic encryption," in *International Conference on Security and Cryptography for Networks*. Springer, 2012, pp. 19–37.

[23] J.-Z. Goey, W. K. Lee, B.-M. Goi, and W.-S. Yap, "Accelerating number theoretic transform in gpu platform for fully homomorphic encryption," *The Journal of Supercomputing*, vol. 77, 02 2021.

[24] S. Gupta, R. Cammarota, and T. Rosing, "Memfhe: End-to-end computing with fully homomorphic encryption in memory," *arXiv preprint arXiv:2204.12557*, 2022.

[25] S. Halevi *et al.*, "Algorithms in helib," in *Advances in Cryptology – CRYPTO 2014*. Springer, 2014, pp. 554–571.

[26] S. Halevi, Y. Polyakov, and V. Shoup, "An improved rns variant of the bfv homomorphic encryption scheme," in *Topics in Cryptology – CT-RSA 2019 - The Cryptographers' Track at the RSA Conference 2019, Proceedings*, M. Matsui, Ed. Germany: Springer Verlag, 2019, pp. 83–105.

[27] K. Han, M. Hhan, and J. H. Cheon, "Improved homomorphic discrete fourier transforms and fhe bootstrapping," *IEEE Access*, vol. 7, pp. 57 361–57 370, 2019.

[28] K. Han, S. Hong, J. H. Cheon, and D. Park, "Efficient logistic regression on large encrypted data," *Cryptology ePrint Archive*, 2018.

[29] K. Han and D. Ki, "Better bootstrapping for approximate homomorphic encryption," in *Topics in Cryptology – CT-RSA 2020*, S. Jarecki, Ed. Cham: Springer International Publishing, 2020, pp. 364–390.

[30] M. Han, Y. Zhu, Q. Lou, Z. Zhou, S. Guo, and L. Ju, "coxhe: A software-hardware co-design framework for fpga acceleration of homomorphic computation," in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2022, pp. 1353–1358.

[31] D. Hankerson, A. J. Menezes, and S. Vanstone, *Guide to elliptic curve cryptography*. Springer Science & Business Media, 2006.

[32] "Open source (release 2.2.1)," https://github.com/homenc/HElib, Oct. 2021.

[33] R. Hiromasa, M. Abe, and T. Okamoto, "Packing messages and optimizing bootstrapping in gsw-fhe," *IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences*, vol. 99, no. 1, pp. 73–82, 2016.

[34] W. Jung, S. Kim, J. H. Ahn, J. H. Cheon, and Y. Lee, "Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with gpus," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 114–148, 2021.

[35] A. Kim, "HEAAN," Online: https://github.com/snucrypto/HEAAN, 2018.

[36] J. Kim, G. Lee, S. Kim, G. Sohn, J. Kim, M. Rhu, and J. H. Ahn, "Ark: Fully homomorphic encryption accelerator with runtime data generation and inter-operation key reuse," *arXiv preprint arXiv:2205.00922*, 2022.

[37] S. Kim, W. Jung, J. Park, and J. H. Ahn, "Accelerating number theoretic transformations for bootstrappable homomorphic encryption on GPUs," in *2020 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2020, pp. 264–275.

[38] S. Kim, J. Kim, M. J. Kim, W. Jung, M. Rhu, J. Kim, and J. H. Ahn, "Bts: An accelerator for bootstrappable fully homomorphic encryption," *arXiv preprint arXiv:2112.15479*, 2021.

[39] "Lattigo 1.3.0," Online: http://github.com/ldsec/lattigo, Dec. 2019, ePFL-LDS.

[40] S. Li, D. Niu, Y. Wang, W. Han, Z. Zhang, T. Guan, Y. Guan, H. Liu, L. Huang, Z. Du *et al.*, "Hyperscale fpga-as-a-service architecture for large-scale distributed graph neural network," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 946–961.

[41] A. C. Mert, E. Öztürk, and E. Savacs, "Design and implementation of encryption/decryption architectures for bfv homomorphic encryption scheme," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 2, pp. 353–362, 2019.

[42] A. Munshi, "The opencl specification," in *2009 IEEE Hot Chips 21 Symposium (HCS)*. IEEE, 2009, pp. 1–314.

[43] A. Norton and A. J. Silberger, "Parallelization and performance analysis of the cooley? tukey fft algorithm for shared-memory architectures," *IEEE Transactions on Computers*, vol. 36, no. 05, pp. 581–591, 1987.

[44] Ö. Özerk, C. Elgezen, A. C. Mert, E. Öztürk, and E. Savacs, "Efficient number theoretic transform implementation on gpu for homomorphic encryption," *The Journal of Supercomputing*, vol. 78, no. 2, pp. 2840–2872, 2022.

[45] M. S. Riazi, K. Laine, B. Pelton, and W. Dai, "Heax: An architecture for computing on encrypted data," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 1295–1309.

[46] R. L. Rivest, L. Adleman, and M. L. Dertouzos, "On data banks and privacy homomorphisms," *Foundations of Secure Computation, Academia Press*, pp. 169–179, 1978.

[47] S. S. Roy, K. Järvinen, J. Vliegen, F. Vercauteren, and I. Verbauwhede, "Hepcloud: An fpga-based multicore processor for fv somewhat homomorphic function evaluation," *IEEE Transactions on Computers*, vol. 67, no. 11, pp. 1637–1650, 2018.

[48] S. S. Roy, F. Turan, K. Jarvinen, F. Vercauteren, and I. Verbauwhede, "Fpga-based high-performance parallel architecture for homomorphic computing on encrypted data," in *2019 IEEE International symposium on high performance computer architecture (HPCA)*. IEEE, 2019, pp. 387–398.

[49] N. Samardzic, A. Feldmann, A. Krastev, S. Devadas, R. Dreslinski, C. Peikert, and D. Sanchez, "F1: A fast and programmable accelerator for fully homomorphic encryption," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 238–252.

[50] N. Samardzic, A. Feldmann, A. Krastev, N. Manohar, N. Genise, S. Devadas, K. Eldefrawy, C. Peikert, and D. Sanchez, "Craterlake: a hardware accelerator for efficient unbounded computation on encrypted data," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 173–187.

[51] "Microsoft SEAL (release 3.7)," https://github.com/Microsoft/SEAL, Sep. 2021, microsoft Research, Redmond, WA.

[52] D. Technologies, "PALISADE library," Online: https://gitlab.com/palisade/palisade-release, 2019.
[53] F. Turan, S. S. Roy, and I. Verbauwhede, "Heaws: An accelerator for homomorphic encryption on the amazon aws fpga," *IEEE Transactions on Computers*, vol. 69, no. 8, pp. 1185–1196, 2020.
[54] M. A. Will and R. K. Ko, "Computing mod without mod," *Cryptology ePrint Archive*, 2014.
[55] G. Xin, Y. Zhao, and J. Han, "A multi-layer parallel hardware architecture for homomorphic computation in machine learning," in *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2021, pp. 1–5.
[56] Y. Zhai, M. Ibrahim, Y. Qiu, F. Boemer, Z. Chen, A. Titov, and A. Lyashevsky, "Accelerating Encrypted Computing on Intel GPUs," *ArXiv*, 2021.