HEAP: A Fully Homomorphic Encryption Accelerator with Parallelized Bootstrapping

Rashmi Agrawal*, Anantha Chandrakasan†, Ajay Joshi*
*Boston University, †MIT
{rashmi23, joshi}@bu.edu, anantha@mit.edu

***Abstract*—Fully homomorphic encryption (FHE) is a cryptographic technology with the potential to revolutionize data privacy by enabling computation on encrypted data. Lately, the CKKS FHE scheme has become quite popular because it can process real numbers. However, CKKS computing is not pervasive yet because it is resource-intensive both in terms of compute and memory, and is multiple orders of magnitude slower than computing on unencrypted data. The recent algorithmic and hardware optimizations to accelerate CKKS computing are promising, but CKKS computing continues to underperform due to an expensive operation known as bootstrapping. While there have been several efforts to accelerate bootstrapping, it continues to remain the main performance bottleneck. One of the reasons for this performance bottleneck is that unlike the non-bootstrapping parts of CKKS computing the bootstrapping algorithm is inherently sequential and exhibits interdependencies among the data.**

To address this challenge, in this paper, we introduce HEAP an accelerator that uses a hybrid scheme-switching approach. HEAP uses the CKKS scheme for the non-bootstrapping steps, but switches to the TFHE scheme when performing the bootstrapping step of the CKKS scheme. The hybrid approach transitions to the TFHE scheme by extracting coefficients from a single RLWE ciphertext to represent multiple LWE ciphertexts. We incorporate the bootstrapping function into the TFHE $\mathsf{BlindRotate}$ operation and simultaneously apply the $\mathsf{BlindRotate}$ operation to all LWE ciphertexts. A parallelized execution of bootstrapping is then feasible because there are no data dependencies between distinct LWE ciphertexts. With our approach, we require smaller-sized bootstrapping keys leading to about $18\times$ less amount of data to be read from the main memory for the keys. In addition, we introduce a variety of hardware optimizations in HEAP—from modular arithmetic level to NTT and $\mathsf{BlindRotate}$ datapath optimizations. The approach in HEAP is agnostic of the hardware and can be mapped to any system with multiple compute nodes. To evaluate HEAP, we implemented it in RTL and mapped it to a single FPGA system and an eight-FPGA system. Our comprehensive evaluation of HEAP for the bootstrapping operation shows a $15.39\times$ improvement when compared to FAB. Similarly, evaluation of HEAP for the logistic regression model training shows $14.71\times$ and $11.57\times$ improvement when compared to FAB and FAB-2 implementations, respectively.

***Index Terms*—CKKS, TFHE, scheme switching, bootstrapping, FPGA acceleration**

I. Introduction

With the growing dependence on large-scale data to make business decisions, businesses are commonly using cloud services to process their data. As a result, cloud environments have become attractive targets for attacks. Data breaches during processing have become quite common these days. In fact, in 2022 alone there were more than 1800 data breaches in the cloud during data processing [32]. So there is a critical need to provide data security and privacy in the cloud when processing the data.

Over the past decade, fully homomorphic encryption (FHE) [24] has emerged as one of the plausible ways to enforce privacy during processing by enabling computing on encrypted data. Even though FHE provides strong data privacy guarantees, it is not yet widely adopted. This is because computing on encrypted data using FHE is inherently slow due to its compute and memory-intensive nature [21], [27], [54]. This is true irrespective of the FHE scheme [9], [15], [16], [26] used in practice.

Out of the currently available FHE schemes, the CKKS FHE scheme can handle computations on real numbers. Given the extensive use of real numbers in machine learning training and inference and the pervasiveness of machine learning (ML)-based applications today and in the foreseeable future, the CKKS scheme is currently the most popular. While the CKKS scheme can support the "deep" ML models, to be able to do that we need to perform an expensive operation known as bootstrapping. This bootstrapping operation, depending upon the scheme parameters, can take several seconds to several minutes [27].

While all the recent algorithmic [1], [10], [30], software [6], [34], [51], and hardware [2], [36]–[38], [49], [50] optimizations to the CKKS primitive and bootstrapping operations have helped reduce the bootstrapping execution time, bootstrapping still remains a performance bottleneck as it consumes up to 95% of the total application execution time [34], [36]. This underscores the critical need for accelerating bootstrapping. The state-of-the-art CKKS bootstrapping algorithm is inherently serial and does not scale well to multiple CPUs/GPUs/FPGAs with all the data dependencies within a single RLWE ciphertext. One of the recent works FAB [2] observed only 20% improvement in application performance when mapping the solution to multiple FPGAs. The performance improvement was limited by the bootstrapping implementation, which could not be parallelized.

In this work, we propose HEAP which uses a hybrid scheme-switching approach that uses a combination of CKKS and TFHE schemes. The core idea is to perform the non-bootstrapping operations using the CKKS primitives and switch to the TFHE scheme to perform the CKKS bootstrapping operation. In the case of a conventional CKKS bootstrapping algorithm, it is necessary to work with large parameter sets, typically ranging from $N = 2^{15}$ to $N = 2^{17}$. This ensures that there are a sufficient number of limbs (typically 17 to 19 limbs) available to perform bootstrapping itself as well as perform application operations. If one were to use small parameters such as $N = 2^{13}$, there would not be a sufficient number of limbs to complete the bootstrapping process itself. Consequently, supporting any real-world practical applications is infeasible with smaller N and $\log Q$ parameters. Our scheme-switching approach for CKKS bootstrapping utilizes only a single limb during the bootstrapping process. Therefore, with our approach, real-world practical applications are feasible using small parameters such as $N = 2^{13}$.

Our scheme-switching approach to perform the CKKS bootstrapping also enables parallelization of the bootstrapping operation. Parallelism during bootstrapping is introduced by the fact that a single RLWE ciphertext can be represented using multiple LWE ciphertexts in the TFHE scheme. The bootstrapping function is embedded within the $\textbf{BlindRotate}$ operation in TFHE, which operates on distinct LWE ciphertexts in parallel, and can be spread across multiple compute platforms. This parallel execution is feasible because there is no data dependency between the distinct LWE ciphertexts. The resultant ciphertexts from these $\textbf{BlindRotate}$ operations are accumulated into a single RLWE ciphertext by streaming results from several compute platforms to a single compute platform.

To facilitate this hybrid strategy, in addition to all the primitive CKKS operations, we implement $\textbf{Extract}$ (short for sample extraction), $\textbf{ModulusSwitch}$, and $\textbf{BlindRotate}$ TFHE operations. We propose several novel microarchitecture-level optimizations at a fine-grain modular arithmetic level to NTT datapath and $\textbf{BlindRotate}$ datapath, which reduce the on-chip memory requirements and the number of memory transfers. Moreover, through our optimized $\textbf{ExternalProduct}$ operation scheduling, we extract as much parallelism as possible in the $\textbf{BlindRotate}$ operation. In addition to the benefits of parallelizing bootstrapping, our hybrid scheme-switching offers benefits in terms of support for smaller parameter sets while enabling bootstrapping operation. Furthermore, as our bootstrapping utilizes only a single level, the selected parameter set allows for less frequent bootstrapping, resulting in further performance enhancements. Moreover, we require smaller-sized bootstrapping keys leading to about $18\times$ less amount of data to be read from the main memory for the keys.

In particular, we make the following contributions:

- We propose HEAP, a multi-compute platform scalable bootstrapping accelerator that parallelizes CKKS bootstrapping operation by using a hybrid scheme-switching approach. To the best of our knowledge, HEAP is the first accelerator to implement a hybrid scheme-switching approach.
- In HEAP we use low-latency tightly coupled functional units with fine-grained pipelining. We optimize the NTT and $\textbf{BlindRotate}$ datapath to enable parallel scheduling of operations while reducing the data accesses from the main memory. The configuration of our on-chip memory complements the implementation of functional units and the optimization of the datapath.

The approach in HEAP is agnostic of the hardware and can be mapped to a multi-CPU, multi-GPU, multi-FPGA, or multi-ASIC system. However, following the FAB work which suggests that FPGA provides a sweet spot for FHE acceleration, we map HEAP onto a single-FPGA and an eight-FPGA system. On average HEAP performs bootstrapping $3283\times$, $12.7\times$, $4\times$, and $15.39\times$ faster than existing state-of-the-art CPU, GPU, ASIC, and FPGA implementations. We evaluate two applications including logistic regression (LR) model training and ResNet-20 inference utilizing this hybrid methodology for bootstrapping. HEAP performs on average $5293\times$, $59.35\times$, $37.8\times$, and $13.14\times$ faster LR model training than existing state-of-the-art CPU, GPU, ASIC, and FPGA implementations. Similarly, HEAP performs on average $39708\times$, $3.7\times$, and $1.2\times$ faster ResNet-20 inference than existing state-of-the-art CPU, GPU, and ASIC implementations.

TABLE I
NOTATIONS USED IN THIS PAPER.

Symbols	Description
$\overrightarrow{a}$, $\overrightarrow{b}$	Denotes two vectors a and b.
$\langle\overrightarrow{a},\overrightarrow{b}\rangle$	Denotes the inner product of vectors a and b.
N	Denotes the ring dimension and a value of power of two.
n	Denotes the number of plaintext elements in a ciphertext. Maximum possible value is $N/2$.
n_t	Denotes LWE mask and the value ranges from 256-4096.
Q	Denotes the full modulus of ciphertext
q	Denotes a limb of Q, and is typically a machine word-sized prime number.
p	Denotes additional limb in raised basis, and is typically a machine word-sized prime number.
Δ	Denotes scale factor of CKKS plaintext, and takes a value close to the limb of a ciphertext.
$\mathcal{R}$	Denotes a $2N$th cyclotomic ring, $\mathcal{R}=\mathbb{Z}[X]/(X^N+1)$.
$\mathcal{R}_Q$	Denotes a quotient ring, $\mathcal{R}_Q=\mathcal{R}/Q\mathcal{R}$.
L	Denotes maximum number of limbs in a ciphertext.
$\boldsymbol{a}$	Denotes a ring element, $\boldsymbol{a}=\boldsymbol{a}(X)$ $\boldsymbol{a}=a_0+a_1.X+\cdots+a_{N-1}.X^{N-1}\in\mathcal{R}$.

II. BACKGROUND

In this section, we present a brief mathematical background on the two FHE schemes: CKKS and TFHE, along with the approach to switch between the two schemes. The security of both the FHE schemes is based on a hard lattice problem called Learning With Errors (LWE), and its variant, such as Ring LWE (RLWE) [42]. Note that in this work, we do not use sparse secret keys as there are security concerns with using sparse keys [14], [52]. The notations used throughout the paper are listed in Table I.

A. The CKKS Scheme

The CKKS scheme is an approximate homomorphic encryption scheme where plaintext is a vector of length n with each entry chosen from a field of complex numbers $\mathbb{C}$. Before a message is encoded into a plaintext vector, it is scaled by a scale factor Δ to preserve as many precision bits as possible. The encryption of a plaintext $\boldsymbol{m}$ results in a ciphertext $\mathbf{ct} = (\boldsymbol{a},\boldsymbol{b})$, which is an RLWE ciphertext consisting of a pair of polynomials in $\mathcal{R}_Q$. The coefficients of these polynomials are elements of $\mathbb{Z}_Q$. Hereafter, throughout the paper, we denote the CKKS ciphertext by the notation $\mathbf{ct}_{\mathcal{R}}$ indicating an RLWE ciphertext.

The precision of Q can range from hundreds to thousands of bits depending upon the use case. Unfortunately, it is impractical to perform operations on such large coefficients using today's computing systems that typically have word sizes of 64-bit or lower. Therefore, the polynomials are decomposed into an equivalent representation using a residue number system (RNS). The smaller polynomials have machine word-sized coefficients modulo each of the q_i and each of these polynomials is known as a limb of ciphertext with respect to smaller modulus q such that $Q=\prod_{i=0}^{L-1}q_i$. This allows us to perform addition and multiplication over $\mathbb{Z}_Q$ using standard machine words.

All addition and multiplication operations on these polynomials involve scalar modular additions and scalar modular multiplications. Current commercially available computing platforms do not inherently support these modular operations. Furthermore, computing modular reduction requires performing an expensive division operation. Hence, optimizing modular arithmetic operations holds substantial importance in enhancing the overall computational efficiency of CKKS computing.

Polynomial multiplications can be efficiently performed using number theoretic transform (NTT). By default, CKKS assumes ciphertext polynomials to be in the NTT domain (also referred to as the evaluation representation). Whenever any operation requires ciphertext to be in coefficient representation, an inverse NTT (iNTT) operation is performed to transform the ciphertext from evaluation to coefficient domain. The addition of two polynomials and multiplication of a polynomial by a scalar is $O(N)$ in both coefficient and evaluation representation. However, this conversion between NTT and iNTT domain takes $O(N\log N)$ time and $O(N)$ space for our degree $N-1$ polynomials.

The CKKS scheme consists of various primitive operations including **PtAdd**, **Add**, **PtMult**, **Mult**, **Rotate**, and **Conjugate**. For further details on these individual primitive operations, interested readers can refer to [12], [15], [21]. As mentioned in Table I, the scale factor is the size of one of the limbs of the ciphertext. During **PtMult** and **Mult**, when the homomorphic multiplications are performed, the underlying scale factor Δ also gets multiplied and becomes Δ^2. Therefore, the scale factor must be shrunk down to Δ by performing a **Rescale** operation.

This **Rescale** operation approximates the division by Δ and rounds the result to the nearest integer. Even when the **Rescale** operation helps keep the scale factor roughly the same throughout the computation, it shrinks the ciphertext modulus. Therefore, if a ciphertext begins with L limbs, only $L-1$ multiplications are feasible due to the reduction in the ciphertext modulus by a count of limbs equivalent to the circuit's multiplicative depth being homomorphically evaluated. Once the multiplicative depth is fully utilized, an operation known as bootstrapping [24] becomes necessary.

Bootstrapping operation refreshes the noise in a ciphertext by homomorphically re-encrypting the ciphertext after removing the noise. Bootstrapping is the most expensive operation and hence the bottleneck when evaluating FHE applications, especially applications that implement a deep circuit. Therefore, there is a need to accelerate the bootstrapping operation.

B. The TFHE Scheme

The TFHE scheme was initially proposed as an improvement of the FHEW scheme, and then it started developing in a broader direction to cover a wider range of applications. It differs from the CKKS scheme in terms of bootstrapping. TFHE performs a special bootstrapping, which is able to evaluate a function at the same time as it reduces the noise. For positive integers q and n_t, basic LWE encryption of $m \in \mathbb{Z}$ under the secret key $\overrightarrow{s} \in \chi_{key}$ is given as follows:

$$\mathbf{ct} = (\overrightarrow{a}, b) = (\overrightarrow{a}, -\langle \overrightarrow{a}, \overrightarrow{s} \rangle + e + m) \in \mathbb{Z}_q^{(n_t+1)} \quad (1)$$

Here, $\overrightarrow{a} \in \mathbb{Z}_q^{n_t}$ and error $e \in \chi_{err}$. Hereafter, throughout the paper, we denote the TFHE ciphertext by the notation $\mathbf{ct}_{\mathcal{L}}$ indicating an LWE ciphertext. In addition to LWE, TFHE uses several other ciphertext formats that we briefly discuss below. RLWE ciphertext of TFHE follows the same format as the CKKS RLWE ciphertext and Ring GSW (RGSW) ciphertext can be simply viewed as a two-dimensional matrix of RLWE ciphertexts.

GLWE[1] ciphertext is represented as a vector of $(h+1)$ polynomials each with a degree $N-1$, where each polynomial is represented as a vector of N integers. Here, h is the GLWE mask and typically takes a value between 1 to 4. The univariate function that is evaluated as part of the **BlindRotate** operation is stored using this GLWE format. The **BlindRotate** keys (**brk**) is a vector of n_t RGSW ciphertexts, where each GGSW[2] ciphertext is a $(h+1)\cdot d \times (h+1)$ matrix of degree $N-1$ polynomials. Here, d is the decomposition degree similar to the decomposition number in the CKKS scheme. We set this value to 2 both for the CKKS and the TFHE scheme. The key switching key is a vector of $h \cdot N \cdot d$ LWE ciphertexts. Although the TFHE scheme comprises many operations, the **ModulusSwitch**, **BlindRotate**, and **Extract** operations are of interest to us in this work. During **ModulusSwitch**, each element in LWE is switched from the modulus q to the modulus $2N$. This operation is not expensive to perform as N is always a power-of-two number.

The steps in **BlindRotate** operation are described in Algorithm 1 [16]. **BlindRotate** transforms a single LWE ciphertext into an RLWE encryption of $f \cdot X^a$. The result is accumulated into the RLWE ciphertext that is denoted as **ACC**. The public keys required for **BlindRotate** are given by $\mathbf{brk} = \{\text{RGSW}(s_i^+), \text{RGSW}(s_i^-)\}_{i \in [0, N-1]}$. These **brk** public keys can be computed offline and must be generated in advance. For all n LWE ciphertexts, we iteratively compute **ACC** using the equation in step 3. The result of the **BlindRotate** operation is an RLWE ciphertext $\mathbf{ct}(f \cdot X^a)$. The polynomial $\boldsymbol{a}_f$ has a as its constant term. After **BlindRotate**, we get n RLWE ciphertexts that encrypt polynomials $\boldsymbol{a}^{(i)}$ where only the constant coefficient of the encrypted polynomials contains useful information. Therefore, as a final step, we combine all constant coefficients of encryption of $\boldsymbol{a}^{(i)}$ into a single encrypted polynomial without decryption. Chen et al. [11] proposed an efficient repacking technique using an automorph operation that we adopt in HEAP.

Extract works by extracting a specific coefficient (the constant term) from the RLWE ciphertext polynomial to form the new LWE ciphertext. Through **Extract**, we get $\mathbf{ct}_{\mathcal{L}} = (\overrightarrow{a}^{(i)}, b_i)$ encrypted under $\overrightarrow{s} = (s_0, \ldots, s_{N-1})$ for all $i \in [0, N-1]$ from $\boldsymbol{a} \in \mathbf{ct}_{\mathcal{R}}$ where

$$\overrightarrow{a}^{(i)} = (a_i, a_{i-1}, \ldots, a_0, -a_{N-1}, -a_{N-2}, \ldots, -a_{i+1}) \quad (2)$$

This operation typically follows the **BlindRotate** operation to maintain the ciphertext representation in LWE format.

III. CKKS Bootstrapping using Scheme-switching

In this section, we first describe how the scheme-switching approach works and then describe the modified CKKS bootstrapping

Algorithm 1 **BlindRotate**(f, $\mathbf{brk} = \{RGSW(s_i^{\pm}\}$, $(\overrightarrow{a}, b) \in \mathbb{Z}_{2N}$)

```
1: $\mathbf{ACC} \leftarrow (0, f \cdot X^b)$
2: for $(i=0; i<n; i=i+1)$ do
3:     $\mathbf{ACC} \leftarrow \mathbf{ACC} * (RGSW(1) + (X^{a_i} - 1) \cdot RGSW(s_i^+) + (X^{-a_i} - 1) \cdot RGSW(s_i^-))$
4: end for
5: return ACC
```

[1] GLWE is a generalization for both LWE and RLWE ciphertexts.

[2] GGSW is a generalization for RGSW ciphertexts.

algorithm that employs this scheme-switching approach.

A. The Scheme Switching Approach

The goal of scheme switching is to take advantage of the parallelism that is inherent to CKKS and TFHE schemes to accelerate the different operations.

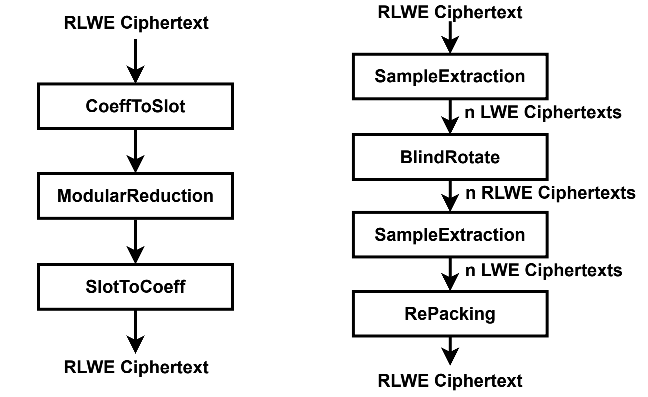
The inherent parallelism in CKKS (through RLWE ciphertext polynomials) can be exploited to perform linear operations like matrix-vector multiplications or convolution operations in machine learning. However, non-linear operations like comparison, sigmoid, exponentiation, and many others need to be evaluated using polynomial approximation as they do not map directly to any homomorphic operations in the CKKS scheme. Thus, the evaluation of non-linear operations using higher-degree polynomials becomes a bottleneck as it requires performing too many homomorphic multiplication operations, which consumes too many limbs in the ciphertext thereby leading to frequent bootstrapping.

Similarly, the TFHE scheme is faster at performing non-linear operations but turns out to be expensive when performing linear operations. So with the scheme-switching approach, we want to integrate the best of both worlds and enable the ability to seamlessly transition between schemes as required by a given application. The scheme switching works by extracting LWE ciphertexts from an RLWE ciphertext. For each extracted LWE ciphertext, we perform the blind rotation with some initial function f. The function f can be set as required by the application under execution. For example, f can be set to evaluate sigmoid, exponentiation, or ReLU function. As an output of the $\mathbf{BlindRotate}$ operation, we obtain RLWE encryptions of $\boldsymbol{a}^{(i)}$ which has a constant term of a_i. We extract these constant terms into multiple LWE ciphertexts. Finally, we repack our RLWE encryptions of $\boldsymbol{a}^{(i)}$ into a single RLWE encryption of $\boldsymbol{a}$.

B. Modified CKKS Bootstrapping

Consider an RLWE ciphertext $\mathbf{ct} = (\boldsymbol{a},\boldsymbol{b}) \in \mathcal{R}_q^2$ in the CKKS scheme with L RNS limbs. After performing repeated $\mathbf{Rescale}$ operations, we exhaust the limbs of the ciphertext and additional homomorphic operations can destroy the message $\boldsymbol{m}$. Therefore, we need to regain the lost limbs and increase the modulus to a bigger modulus denoted by Q'. As part of the CKKS bootstrapping [12], when the modulus is raised to Q', an additional term $\boldsymbol{k}\cdot q$ gets added to the message, modifying the message to $\boldsymbol{m}+\boldsymbol{k}\cdot q$. Here, $\boldsymbol{k}$ is some polynomial with small integer coefficients and q is the modulus for the input ciphertext. The primary goal of the bootstrapping operation is to homomorphically evaluate the modular reduction operation modulo q on this message, returning the message back to m.

As shown in Figure 1 (a), the first step of bootstrapping is a linear transformation step that converts the ciphertext representation from the coefficient to the evaluation domain. A polynomial approximation of the modular reduction function helps remove the $\boldsymbol{k}\cdot q$ part from the message. Then, as a last step of bootstrapping, another linear transformation step is performed to convert ciphertext back to evaluation representation. The challenging part in bootstrapping is to accurately approximate the modular reduction function. Moreover, based on the parameter set chosen for the CKKS scheme implementation, the bootstrapping step may itself consume anywhere from 15-19 levels, thus, leaving only a few compute levels for the application itself and requiring more frequent bootstrapping.



(a) CKKS Bootstrapping Steps (b) Modified CKKS Bootstrapping Steps

Fig. 1. The CKKS Bootstrapping; (a) Steps in state-of-the-art bootstrapping algorithm, and (b) Steps in the modified bootstrapping algorithm.

Algorithm 2 CKKS-Bootstrapping($\mathbf{ct} = (\boldsymbol{a},\boldsymbol{b}) \in \mathcal{R}_q^2$)

```
$\mathbf{ct}' \leftarrow 2N\cdot\mathbf{ct} \pmod{q} \in \mathcal{R}$
$\mathbf{ct_{ms}} \leftarrow (\frac{2N\cdot\mathbf{ct}-\mathbf{ct}'}{q}) \in \mathcal{R}_{2N}$
$\mathbf{ct_{kq}} \leftarrow \mathbf{BlindRotate}(\mathbf{Extract}(\mathbf{ct_{ms}},q,Qp)) \in \mathcal{R}_{Qp}$
$\mathbf{ct}'' \leftarrow \mathbf{ct_{kq}} + \mathbf{ct}' \pmod{Qp} \in \mathcal{R}_{Qp}$
$\mathbf{ct_{boot}} = \mathbf{Rescale}(\frac{p}{2N}\cdot\mathbf{ct}'',p) \in \mathcal{R}_Q$
return $\mathbf{ct_{boot}} = (\boldsymbol{a}_{\mathbf{boot}},\boldsymbol{b}_{\mathbf{boot}}) \in \mathcal{R}_Q^2$
```

To address these challenges, we propose using the scheme-switching approach to perform CKKS bootstrapping as shown in Figure 1 (b). We adopt the idea proposed by Kim et al. [35] that suggests removing the $\boldsymbol{k}\cdot q$ part from the message through subtraction instead of approximate modular reduction. Therefore, in our modified CKKS bootstrapping approach using scheme-switching, we first compute encryption of $-\boldsymbol{k}\cdot q$ by using the blind rotation technique in the TFHE scheme. We denote this newly encrypted ciphertext by $\mathbf{ct_{kq}} = (\boldsymbol{a}_{\mathbf{kq}},\boldsymbol{b}_{\mathbf{kq}}) \in \mathcal{R}_Q^2$ and it contains the encryption of the scaled value $\boldsymbol{k}\cdot q$. Then, we add this newly encrypted ciphertext $\mathbf{ct_{kq}}$ to our original ciphertext $\mathbf{ct}$ to eliminate $\boldsymbol{k}\cdot q$ from the underlying message m.

The detailed steps of the modified bootstrapping algorithm are as shown in Algorithm 2. We begin with an RLWE ciphertext $\mathbf{ct}$. As TFHE operates in $2N$ modulus domain, we first need to perform a $\mathbf{ModulusSwitch}$ operation. As part of $\mathbf{ModulusSwitch}$, we first compute $\mathbf{ct}' = 2N\cdot\mathbf{ct} \pmod{q}$ to obtain $\mathbf{ct}'(\boldsymbol{s})$. Now both $2N\cdot\boldsymbol{a}-[2N\cdot\boldsymbol{a}]_q$ and $2N\cdot\boldsymbol{b}-[2N\cdot\boldsymbol{b}]_q$ are divisible by q, thus, we obtain our modulus switched ciphertext, $\mathbf{ct_{ms}} = (\boldsymbol{a}_{\mathbf{ms}},\boldsymbol{b}_{\mathbf{ms}})$.

Then, we evaluate $\mathbf{BlindRotate}(\mathbf{ct_{ms}},q,Qp)$ and the output is a ciphertext $\mathbf{ct_{kq}}(\boldsymbol{s}) = (\boldsymbol{a}_{\mathbf{kq}},\boldsymbol{b}_{\mathbf{kq}})$ w.r.t. $\pmod{Qp}$. Here, p is an auxiliary prime by which we will rescale later at the end of the bootstrapping procedure. Then, we add $\mathbf{ct_{kq}}$ to $\mathbf{ct}'$ modulo Qp to obtain $\mathbf{ct}'' = 2N\cdot\boldsymbol{m} \pmod{Qp}$. To get rid of the scaling factor $2N$ in the message, we multiply $\mathbf{ct}''$ by $\frac{p}{2N}$ and then $\mathbf{Rescale}$ the result by p.

C. Parameter Set for HEAP

We briefly discuss the parameter set choices for HEAP. To support NTT, the parameter N has to be a power of two. We pick the value of N to be 2^{13} to place an upper bound on the number of RLWE ciphertexts that will be generated as part of the $\mathbf{BlindRotate}$ operation. To enable 128-bit security, the corresponding ciphertext modulus, $\log\, Q$ that we pick is 216. Therefore, the size of our RLWE ciphertext will be $2 \times \log Q \times N = 2 \times 216 \times 8192 = \sim 0.44$MB.

Now, for the RNS decomposition of our ciphertext, we pick a value of $\log q = 36$ for each of our RNS limbs. The size of each RNS limb is $\sim$0.04MB. This gives us $L = 6$, implying we can perform 5 multiplications before requiring to perform a CKKS bootstrapping operation. The choice of 36 bits for an RNS limb enables us to make use of fast DSP multipliers, adders, and on-chip memory (URAM and BRAM) blocks efficiently on the underlying FPGA. Section IV describes this in more detail. The size of each LWE ciphertext is $\sim$2.3 KB with $n_t = 500$ and $\log q = 36$.

The gadget decomposition d value is set to 2 for RGSW elements. The key in $\mathbf{BlindRotate}$ is a vector of n_t GGSW ciphertexts, where each ciphertext is a $(h+1) \cdot d \times (h+1)$ matrix of degree $N-1$ polynomials. A typical value for $h = 1$ and thus, the size of the key is $\sim$3.52 MB. We need $n_t = 500$ such keys, implying that the total key size will be 1.76 GB to perform the entire CKKS bootstrapping using the hybrid scheme-switching approach. Note that the values for d and h are carefully chosen as 2 and 1, respectively. We choose these values so that we do not incur additional memory overhead for the bootstrapping keys as the size of the key linearly scales with these two values.

In the case of state-of-the-art CKKS bootstrapping, the size of each key is about $\sim$126 MB for bootstrappable parameters. To perform a single bootstrapping we need about 25 such keys (24 keys for rotation and 1 key for multiplication), considering the optimized bootstrapping implementation [1]. Thus, in total, we need to read about $\sim$32 GB of keys to perform the entire CKKS bootstrapping. With our approach, we require about 18× less amount of data to be read from the main memory just for the keys.

IV. HEAP Microarchitecture

In this section, we describe the microarchitecture of HEAP which consists of functional units, register files, FIFOs, on-chip memory, and various control units. In addition, we describe our datapath optimizations to the NTT and blind rotation operation. Note that the design described here maps to a single FPGA and the overall system design with multiple FPGAs is described in detail in Section V.

A. *Functional Units*

We categorize the functional units in HEAP into modular arithmetic units, permute units, and external product units. The modular arithmetic units consist of modular adders, subtractors, and multipliers. The permute unit consists of an automorph unit for CKKS $\mathbf{Rotate}$ and a rotation unit for TFHE $\mathbf{BlindRotate}$ operation. The external product units consist of multiply and accumulate (MAC) units that perform element-wise multiplication between two polynomials and accumulate their partial sums. This unit is required both during basis conversion operation in CKKS $\mathbf{KeySwitch}$ operation and in TFHE $\mathbf{BlindRotate}$ operation.

Modular Arithmetic Units: The lower-level operations in both the CKKS and TFHE schemes are modular arithmetic operations. Our RNS limb size is 36 bits and the fast multipliers and adders in the DSP blocks on FPGA are 18-bit and 32-bit wide, respectively. Consequently, to map 36-bit multiplications and additions to the DSP blocks, we follow the modular arithmetic circuit design principles, wherein one can compose the existing multipliers and adders into larger word-size arithmetic units. Following the same principle, a 36-bit integer multiplier can be realized by composing two 18-bit integer multipliers and similar holds for the adder as well.

To perform the modular reduction during addition and subtraction operations, we use the conditional operator, one of the least expensive ways of doing modular reduction. We perform standard Barrett reduction [4] following the integer multiplication operation. However, we combine the integer multiplication and Barrett reduction in a way so as to start the reduction process as soon as the partial result of the multiplication gets generated. This helps in reducing the overall latency of the modular multiplication operation. We do not utilize Montgomery reduction as this approach is shift-based and would consume LUTs on the FPGA. Instead, we employ the Barrett reduction approach, which allows us to utilize DSP multipliers instead of LUTs. This frees up LUTs to be allocated for other operations. Our modular addition, modular subtraction, and modular multiplication take 7 clock cycles to perform a single scalar modular operation. We instantiate a total of 512 modular arithmetic units after a detailed analysis of the compute utilization that can match the memory throughput.

Permute Unit: The permute unit consists of an automorph unit for CKKS $\mathbf{Rotate}$ and a rotation unit for TFHE $\mathbf{BlindRotate}$ operation. The automorph unit follows the index mapping equation as $i_r = i \cdot 5^r \pmod{N}$. So the automorph unit moves the i-th coefficient of a polynomial to the position indexed by i_r where r is the number of positions by which rotation is to be performed. The values $5^r \pmod{N}$ are precomputed. Then the multiplication with the current index becomes a very low-cost computation as it can be done using shift operations.

We instantiate 512 automorph units that can operate on 16 elements each to rotate all the coefficients in a polynomial. Therefore, it takes 16 cycles to finish the entire automorph operation on a single ciphertext limb. Note that this does not include the cycle count for the $\mathbf{KeySwitch}$ operation that follows automorph operation in the CKKS $\mathbf{Rotate}$ operation.

The rotation unit for TFHE performs a polynomial negacyclic rotation [5]. The rotation unit rotates the coefficients by k positions where k comes from $f \cdot X^k$ as described in Section III-A. The rotation operation is followed by a subtraction or addition (as required), which is performed using the adder within this rotation unit. Note that rotation performed here is different from automorph operation as it is dependent on any sort of ring mapping.

External Product Unit: The MAC units within the external product unit perform an element-wise multiplication between two polynomials and accumulate their sums. These MAC units perform an integer multiplication followed by a modular addition operation. Each MAC unit is a low-latency fused multiplier and adder where the addition starts immediately after the partial results of multiplication are available. Furthermore, the modular reduction starts immediately after the partial result of addition is generated. We leverage the idea of lazy reduction by performing modular reduction only after both multiplication and addition operations are done. This lowers the latency of the external product operation and incurs less resource utilization on the FPGA.

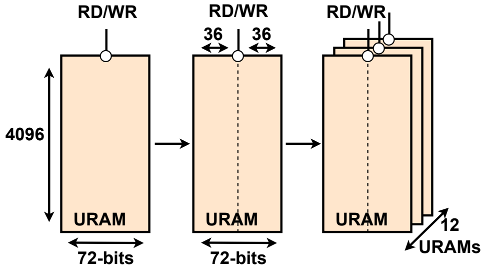


Fig. 2. On-chip memory configuration: URAM Layout.

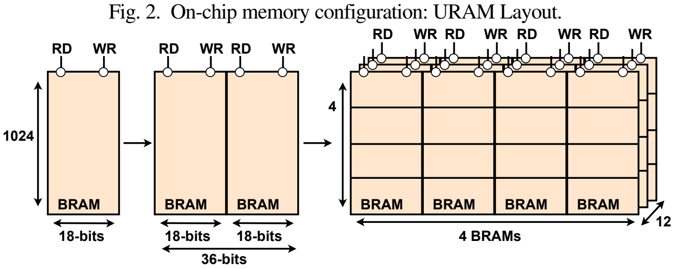


Fig. 3. On-chip memory configuration: BRAM Layout.

To enable basis conversion operation [21] during $\mathbf{ModUp}$ and $\mathbf{ModDown}$ in CKKS $\mathbf{KeySwitch}$ operation, the external product unit is bundled with a dual-port BRAM block. Two full (192×2) BRAM blocks are tightly coupled with the MAC units to enable input and output worth at least two ciphertexts. During the TFHE $\mathbf{BlindRotate}$ operation, these BRAM blocks enable the reading of multiple LWE ciphertexts and accumulate the results in RLWE ciphertexts. As soon as one of the RLWE ciphertexts is accumulated in the BRAM block, it is written out to the main memory.

B. Register Files and FIFOs

HEAP design comprises multiple register files (RFs) with a combined capacity of 1 MB. These RFs are distributed throughout the design and serve functional, address generation, and control units. Each RF features multiple read/write ports, each with a single-cycle access latency. Approximately half of the RF is allocated for storing the ciphertexts while computing on them and one-fourth of the remaining RF is allocated for storing pre-computed values and scheme-related parameters. The pre-computed values are written by the host CPU through atomic writes before initiating the execution of the kernel code.

The HEAP design incorporates 32 synchronous read (RD) and write (WR) FIFOs, accommodating 32 AXI ports on the HBM side for seamless data streaming between the main memory and on-chip memory. These FIFOs are constructed using the distributed RAM on the FPGA board, with each FIFO's data width matching the data width supported by each AXI port (256 bits). The RD FIFO has a depth of 512 to handle up to four outstanding reads, while the WR FIFO has a depth of 128 to facilitate an HBM burst length of 128. The RD FIFO's clock input comes in from the memory-side clock domain so the RD FIFO operates at a clock frequency of 450 MHz. The WR FIFO's clock input comes in from the kernel-side clock domain so the WR FIFO operates at a clock frequency of 300 MHz. In addition, Transmit (TX) and Receive (RX) FIFOs are instantiated to facilitate data streaming between the CMAC subsystem and on-chip memory. These TX and RX FIFOs, like the others, are synchronous and feature a 512-bit data interface.

C. On-chip Memory Organization

The Alveo U280 FPGA has 962 UltraRAM (URAM) blocks and 4032 Block RAM (BRAM) blocks. Each URAM block can store 4096 data elements where each element is 72 bits wide while each BRAM block can store 1024 data elements where each element is 72 bits wide. URAM blocks can only be used as single-port memory while BRAM blocks can be used as both single and dual-port memory.

As shown in Figure 2, each URAM address can store two coefficients as each RNS limb coefficient is 36 bits. From our $\mathbf{ct}_{\mathcal{R}} = (\boldsymbol{a}, \boldsymbol{b})$, we store the coefficient from the limbs (with respect to the same modulus) of $\boldsymbol{a}$ and $\boldsymbol{b}$ adjacent to each other. The goal is to fetch the coefficients from the limbs with respect to the same modulus at once so that all the pre-computed values including the twiddle factors for NTT/iNTT can be read at once instead of reading them twice. With this organization of data in URAMs, we need 12 URAM blocks to store all the limbs within both the ring elements of the ciphertext. We can utilize up to 960 URAM blocks to store 80 RLWE ciphertexts when performing $\mathbf{BlindRotate}$ operation during bootstrapping.

As shown in Figure 3, each BRAM address can store only half a coefficient, i.e., 18 bits from the 36-bit wide RNS limb coefficient. We combine two BRAM blocks to store one entire 36-bit coefficient. We further combine BRAM blocks to enable the storage of two coefficients adjacent to each other and 4096 coefficients depth-wise. By doing so we match the storage organization within the BRAM blocks to that of the URAM blocks. Thus, the address generation logic to fetch the coefficients of the ciphertext limbs remains the same irrespective of URAM or BRAM. We need 192 BRAM blocks in total to store all the limbs of the ciphertext and we can utilize up to 3840 BRAM blocks to store 20 RLWE ciphertexts.

It takes one clock cycle to perform a read as well as a write operation to and from the URAM and BRAM blocks. We avoid using URAM blocks during the MAC operations as they add to the latency with only a single port to perform both read and write operations. We store all the read-only data fetched from the main memory to URAM blocks. This read-only data consists of all the evaluation keys, pre-computed twiddle factors, and the $\mathbf{BlindRotate}$ keys. We use BRAM blocks during the MAC operations as they enable simultaneous reads and writes within the same clock cycle.

D. NTT Datapath Optimization

NTT is one of the most compute-intensive operations and becomes a key bottleneck while performing homomorphic operations. The computational complexity of NTT is $O(N \log N)$ where $N/2$ butterfly operations are performed in each of the $\log N$ stages of NTT. We leverage our modular arithmetic units to perform these butterfly operations (radix-2). When performing the butterfly operations using the 512 modular arithmetic units, we can process 1024 coefficients at once.

As described in Section IV-C, we access one coefficient from two different limbs in parallel at once, therefore, we perform NTT on two limbs in parallel. However, these two limbs are from different ring elements but belong to the same modulus, requiring the same twiddle factors to perform the NTT operation. Consequently, we benefit from reading the same twiddle factors from the main memory to the on-chip memory, avoiding reading them twice. This optimization reduces the amount of data to be fetched from the main memory and thus helps lower the main memory bandwidth requirements. Our first set of 256 modular arithmetic units operates on 512 coefficients from the first limb and the second set of 256 modular arithmetic units operates on 512 coefficients from the second limb.

Our NTT datapath follows the data access pattern as per the Cooley-Tukey algorithm [46]. We simplify the address generation logic and fetching of the twiddle factors from the on-chip memory to the compute units by optimizing the data access pattern as well. We group the coefficients in each stage based on the twiddle factor that they need for the butterfly operation. We denote the current stage of NTT using notation cs. Each group has $n_c = N/2^{cs}$ coefficients and coefficients within each group can be indexed using $i_{nc} = 0$ to $n_c - 1$. Therefore, the number of groups $n_g = N/n_c$ and we denote i_g to index into these groups. Now, we can generate the address for the coefficients to be processed by butterfly units as $address = i_g + i_{nc} * 2^{cs}$ Thus, the address generation logic gets highly simplified. This optimization has the added advantage of enabling on-the-fly twiddle factor generation, which is helpful when the on-chip memory is not sufficient to store all the twiddle factors at once and we have available compute bandwidth. Therefore, by setting an appropriate control signal, we can easily switch between reading the twiddle factors from memory versus generating them on the fly. Note that our iNTT implementation follows the Cooley-Tukey algorithm datapath as well.

E. **BlindRotate** *Datapath Optimization*

The sequence of sub-operations in **BlindRotate** operation includes rotation, decompose, **ExternalProduct**, and **Extract**. These sub-operations are to be performed sequentially on a ciphertext, limiting the performance of the **BlindRotate** operation. Moreover, every **ExternalProduct** requires a new **BlindRotate** key (**brk**), increasing the overall data transfers. We can extract more performance and reduce the memory footprint of this operation by scheduling **BlindRotate** operation on multiple LWE ciphertexts instead.

Using this parallel scheduling approach, we can schedule up to 512 ciphertexts in parallel. This is because we have 512 functional units to enable the processing of 512 ciphertexts in parallel. We perform rotation on all 512 ciphertexts first, then decompose all of them, and then move forward to perform the **ExternalProduct** operation. Note that rotation and decompose sub-operations happen on the ciphertext in the coefficient domain. So, before performing the **ExternalProduct** operation, we need to perform NTT on the ciphertext to enable fast element-wise polynomial multiplication operations. The literature makes use of either FFT or NTT operation depending on the underlying compute platform. We use our optimized NTT implementation as complex FFT will be expensive to perform.

Now when performing **ExternalProduct**, we schedule the elements requiring the same **brk** at once in parallel. Consequently, we need to fetch one key at a time, perform the external product using the key, and then discard the key. We do not need to read the same key again, enabling the scope of on-the-fly **brk** generation in cases when the on-chip memory is limited. After multiplication with **brk**, we store the resultant ciphertext in a dual-port BRAM. Following the second multiplication between the ciphertext and **brk**, the first multiplication result gets read from the BRAM and added to this result. Thus, using dual-port BRAMs, we perform partial accumulation during the **ExternalProduct** compute and save reads and writes from/to the main memory.

Note that our basis conversion operation in the CKKS **KeySwitch** follows the same datapath as the **ExternalProduct** operation. Hence, we do not describe the optimized **KeySwitch** datapath in detail here.

V. Overall System Architecture

In this section, we introduce the overall system architecture employing multiple FPGAs for the deployment of our proposed HEAP hardware accelerator.

We first describe the system architecture that maps to a single FPGA. As shown in Figure 4, our full system comprises various components. The first component is an X86 host CPU responsible for offloading both data and the HEAP RTL design to the FPGA. The host CPU is connected to the Alveo U280 FPGA through a PCIe interface. This PCIe connection facilitates data transfer between the host and the global memory on the FPGA board. Facilitating this data transfer involves the host allocating a buffer in the global memory with a size corresponding to that of the dataset. An AXI4-Lite interface is used by the host to communicate the base address of the buffer to the RTL code as well as other required parameters. A C++ host code kicks off the RTL code using the OpenCL API call. Once the RTL execution code completes, the results are transferred back to the host code.

The second component is our HEAP RTL design comprising of functional units, on-chip memory (URAMs and BRAMs), RFs, FIFOs, control logic, and address generation logic. In addition, the RTL design implements 32 memory-mapped 256-bit interfaces. These interfaces are AXI4 master interfaces that enable bidirectional data transfers to/from the global memory. The FPGA's global memory consists of two HBM2 stacks each with a capacity of 4 GB with up to 460 GB/s bandwidth. The RD and WR FIFO stream the data from global memory onto the on-chip memory and back while the TX and RX FIFO stream the data to and from the CMAC subsystem.

The third and final component is a 100G Ethernet (CMAC) subsystem facilitating the transmission and reception of data between FPGAs, without requiring the host's involvement. The integrated IP block on the FPGA provides a 100 Gbps Ethernet port to transfer data between FPGAs that are connected to different hosts. The CMAC core operates at a clock frequency of 322 MHz. Our RTL code implements a 512-bit interface with the CMAC core to enable 100 Gbps data transfer rates. It takes about 458 clock cycles to transmit an entire RLWE ciphertext for our chosen parameter set.

To extend our system architecture to multiple FPGAs we map the same HEAP RTL code to eight FPGAs that are connected to eight different hosts. These FPGAs interact with each other via the CMAC subsystem without involving the host. Note that, if required, these FPGAs can interact via the host CPU as well but that increases the communication latency. Among the eight FPGAs, we designate one FPGA as the primary FPGA and the remaining seven FPGAs as the secondary FPGAs.

The primary FPGA is responsible for distributing the LWE ciphertexts to the secondary FPGAs, and then receiving the LWE ciphertexts and repacking them into a single RLWE ciphertext.

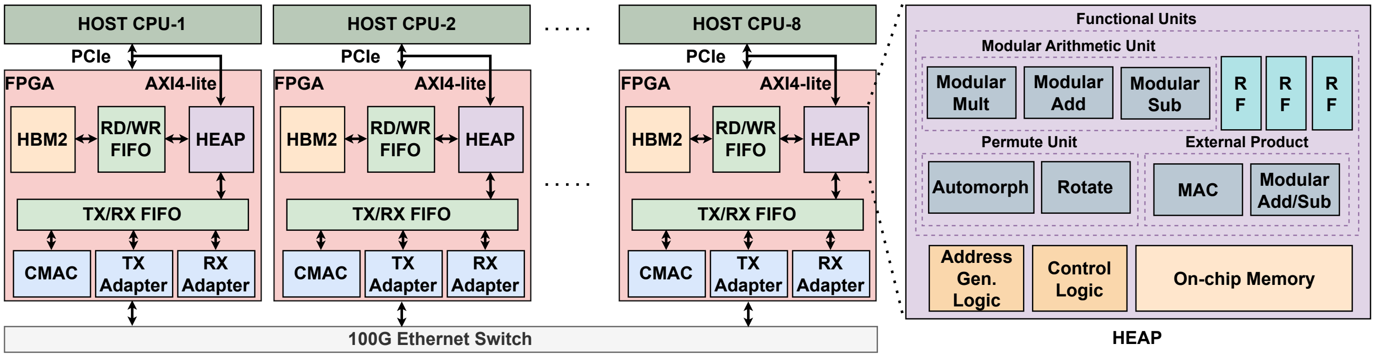


Fig. 4. Multi-FPGA system that uses HEAP for CKKS computing. The detailed microarchitecture of HEAP is shown in the inset on the right.

TABLE II
HEAP HARDWARE RESOURCE UTILIZATION ON A SINGLE FPGA.

Resource	Available	Utilized	% Utilization
LUTs	1304K	1012K	77.61
FFs	2607K	1936K	74.26
DSPs	9024	6144	68.08
BRAM blocks	4032	3840	95.24
URAM blocks	962	960	99.80

The primary FPGA sends all the ciphertexts intended for one of the secondary FPGAs before sending the ciphertexts for the next one. This approach not only helps the HEAP architecture to utilize the optimized $\mathbf{BlindRotate}$ datapath but also prevents the network from becoming the bottleneck. A secondary FPGA starts sending the resultant LWE ciphertext to the primary FPGA for accumulation as soon as the $\mathbf{BlindRotate}$ operation is completed on the ciphertexts. We overlap the communication between FPGAs and the computation in FPGAs through smart scheduling such that no FPGA is sitting idle i.e. communication between the FPGAs is not the bottleneck.

Note that we have a parameter namely n_{br} in the HEAP state machine that controls the number of $\mathbf{BlindRotate}$ operations that need to be scheduled on an FPGA. This is required because not all applications require us to use all of the slots in a ciphertext (fully packed ciphertext). Some may instead need to do a sparse packing. Thus, for a given application that needs to be evaluated, the number of slots that are packed in a ciphertext varies and we add the capability to easily modify the number of $\mathbf{BlindRotate}$ operations that need to be performed on a single FPGA. Consequently, the performance can be tuned for the application under evaluation.

VI. EVALUATION

As mentioned earlier, we map our HEAP accelerator to single and multiple Alveo U280 FPGAs. We design HEAP using Xilinx Vivado 2022.2 and Vitis 2022.2 EDA design tools. The RTL code is written in Verilog 2001. The Vitis compiler takes care of compiling and linking the RTL code to generate the binary that gets mapped to an FPGA. The execution of this binary is initiated by the host code in the cloud environment. The maximum operating frequency that we are able to achieve for HEAP is 300 MHz. We would like to note that throughout this section, if unspecified, HEAP's performance numbers are based on its implementation using eight FPGAs.

A. FPGA Resource Utilization

In Table II, we present the hardware resource utilization of the various components of HEAP. Overall, HEAP utilizes 1012K LUTs which is about 77.61% of the total LUTs present on the FPGA. The functional units utilize the most LUTs, i.e., 42% of the total LUTs utilized. HEAP utilizes a total of 1936K FFs in the functional units, RFs, FIFOs, address generation logic, and control logic. The entire DSP utilization is for the modular adder, subtractor, multipliers, and MAC operations within the functional units. HEAP achieves 95.24% BRAM and 99.8% URAM utilization for the on-chip memory.

B. Area and Power Comparison

Although we cannot directly compare the area and power of FPGA designs to that of ASIC designs, we present a brief comparison of the area in terms of the number of modular multipliers instantiated and the total on-chip memory. HEAP on a single FPGA instantiates 512 modular multipliers and uses 43 MB on-chip memory. Thus, HEAP on eight FPGAs instantiates 4096 modular multipliers and uses 344 MB on-chip memory in total. The ASIC designs instantiate 4096-20480 modular multipliers and the on-chip memory ranges from 72-512 MB. Moreover, the ASIC designs have access to these resources on a single chip in a coherent fashion, making it much easier to make optimal use of these resources. This is not the case with HEAP design using eight FPGAs. To the first order, power consumption is proportional to the area of a chip, which is proportional to the number of compute units and memory. Typically, ASICs consume less power than FPGAs. Given the smaller compute unit count and smaller on-chip memory in HEAP, we expect its power consumption to be comparable, if not better, than ASICs.

C. Parameter Choice for Comparison with State-of-the-art

For parameters $N = 2^{16}$ and $N = 2^{13}$, to achieve a 128-bit security, ciphertext comprises of 24 and 6 limbs, respectively. Interestingly, conventional bootstrapping for $N = 2^{16}$ and scheme-switching-based bootstrapping for $N = 2^{13}$ utilizes 19 limbs and 1 limb, respectively, leaving 5 limbs in the ciphertext in both cases. So we are left with the same number of limbs for performing other operations within the application. Consequently, applications that use $N = 2^{16}$ with conventional bootstrapping and $N = 2^{13}$ with scheme-switching-based bootstrapping, will perform homomorphic encryption operations on the same number of ciphertext limbs.

Now to account for the difference in the number of slots while evaluating bootstrapping, we do not compare the bootstrapping runtime directly but compare the $T_{\mathbf{Mult},a/slot}$. This equation is used by all the state-of-the-art works [34], [37], [38], [50] to compare bootstrapping performance as it accounts for any difference in

TABLE III
EXECUTION TIME (IN MS) FOR PERFORMING BASIC FHE OPERATIONS AND SPEEDUP ACHIEVED USING HEAP (SINGLE FPGA). NOTE THAT $\mathbf{Add}$, $\mathbf{Mult}$, $\mathbf{Rescale}$, AND $\mathbf{Rotate}$ ARE NOT SUPPORTED IN TFHE [17], WHILE $\mathbf{BlindRotate}$ IS NOT SUPPORTED BY FAB [2], GPU [34] AND GME [51].

Operation	**Scheme**	**HEAP**	**FAB** [2]	**GPU** [34]	**GME** [51]	**TFHE** [17]	**Speedup vs FAB**	**Speedup vs GPU**	**Speedup vs GME**	**Speedup vs TFHE**
$\mathbf{Add}$	CKKS	0.001	0.04	0.16	0.028	-	40×	160×	28×	-
$\mathbf{Mult}$	CKKS	0.028	1.71	2.96	0.464	-	61.1×	105.71×	16.57×	-
$\mathbf{Rescale}$	CKKS	0.010	0.19	0.49	0.069	-	19×	49×	6.9×	-
$\mathbf{Rotate}$	CKKS	0.025	1.57	2.55	0.364	-	62.8×	102×	14.56×	-
$\mathbf{BlindRotate}$	TFHE	0.060	-	-	-	9.40	-	-	-	156.7×

TABLE IV
NTT THROUGHPUT (OPERATIONS/SECOND) FOR HEAP (SINGLE FPGA). (PARAMETER SET USED $N=2^{13}$ AND $\log Q=218$)

Operation	**HEAP**	**FAB [2]**	**HEAX [48]**	**Speedup vs FAB**	**Speedup vs HEAX**
NTT	210K	103K	90K	2.04×	2.34×

number of slots. When it comes to comparing the performance of applications, the difference in the slot count because of $N = 2^{13}$ and $N = 2^{16}$ does not matter. This is because we are matching the number of slots used for LR model training and ResNet-20 with what has been used in state-of-the-art prior work [36]–[38], [50].

D. Basic Building Block's Performance

Here we present the execution time for basic operations in the CKKS & TFHE scheme in Table III[3]. For HEAP the parameter set used is $N=2^{13}$, $\log Q=216$. The 128-bit secure FAB [2] and GME [51] numbers reported in the table are using the parameter set $N=2^{16}$ and $\log Q=1728$. For the numbers in the GPU column, we use the most optimized performance numbers reported in the work by Jung et al., [34] for the parameter set $N=2^{16}$, $\log Q=1693$, and 100-bit security. We compare using this parameter set even when HEAP uses a smaller parameter set because if we were not using the scheme switching approach to support CKKS bootstrapping, HEAP will need to work with at least $N=2^{16}$ and $\log Q=1728$ parameter set to enable CKKS bootstrapping. For the basic CKKS operations, HEAP achieves an average 45.72×, 104.18×, and 16.51× speedup (when comparing absolute execution time) compared to FAB, GPU, and GME work, respectively. Note that we can use a smaller N and smaller $\log Q$ for HEAP because of the scheme-switching approach. This leads to better performance in HEAP. Finally, we compare the performance of the $\mathbf{BlindRotate}$ operation in HEAP with its TFHE [17] implementation on the CPU. We observe a 156.7× speedup, with a runtime of 0.06ms for $\mathbf{BlindRotate}$ in HEAP compared to 9.40ms in TFHE.

In Table IV, we compare the throughput of HEAP with HEAX [48], and FAB [2] when running NTT. We use the same parameter set ($N = 2^{13}$ and $\log Q = 218$) for all three implementations for a fair comparison. HEAP achieves an average 2.04× and 2.34× higher throughput than FAB and HEAX, respectively. The higher performance of HEAP can be attributed mainly to the inclusion of low-latency modular arithmetic modules to perform butterfly operations, the fine-grained pipelining for these units, and the optimization of the NTT datapath to a high degree.

E. Bootstrapping Performance

For the parameter set mentioned in Section III-C, we first evaluate the performance of our modified CKKS bootstrapping. We utilize all the slots ($n = 4096$) of the ciphertext, i.e., we perform a fully-packed bootstrapping. Both the number of LWE ciphertexts and $\mathbf{BlindRotate}$ operations to be performed are equal to n. We distribute these ciphertexts equally amongst all eight FPGAs, implying that each FPGA processes 512 LWE ciphertexts. As a result, with our scheme-switching approach, the time it takes to perform a single CKKS bootstrapping is $\sim$1.5 ms (In Algorithm 2, steps 1&2 take 0.0025 ms, step 3 takes 1.3303 ms, and steps 4&5 take 0.1672 ms). We observe this low latency because, with our scheme-switching approach, we do not need to perform hundreds of expensive CKKS $\mathbf{KeySwitch}$ operations (that are part of $\mathbf{Rotate}$ and $\mathbf{Mult}$ operations) as required in the state-of-the-art CKKS bootstrapping algorithm [21].

We next compare the bootstrapping performance of HEAP with the existing state-of-the-art CPU, GPU, FPGA, and ASIC implementations (see Table V). Throughout this section, we use the following terminology for various implementations; CPU implementation is referred to as Lattigo [6], GPU implementation for 97-bit security by Jung et al. [34] is referred to as GPU, BTS ASIC proposal referred to as BTS-2 [38] for the best case numbers reported by the authors, and CraterLake [50] ASIC proposal (for their parameters achieving 128-bit security) referred to as CL in abbreviated form. Other works are referred to in the same way as before. To fairly compare the bootstrapping latency, we follow the literature and use the amortized per slot multiplication time $T_{\mathbf{Mult},a/slot}$ given as follows:

$$T_{\mathbf{Mult},a/\mathsf{slot}} := \frac{T_{\mathsf{BS}} + \sum_{i=1}^{\ell} T_{\mathbf{Mult}}(i)}{\ell \cdot n} \tag{3}$$

Here, n is the number of slots in the ciphertext, T_{BS} is the bootstrapping time, and $T_{\mathbf{Mult}}(i)$ is the time to multiply at level i. The number of levels ℓ in the resulting ciphertext is equal to the maximum supported levels in the starting bootstrapping modulus minus the depth of bootstrapping. The depth of our bootstrapping algorithm is 1. Note that all the previous state-of-the-art work report bootstrapping performance using this metric.

In Table V, for CKKS, we compare the performance of a single conventional bootstrapping operation vs. a single bootstrapping using the proposed scheme-switching approach. From Table V, we observe that HEAP outperforms CPU, GPU, FPGA, and some of the prior ASIC proposals in terms of absolute bootstrapping time. HEAP does not outperform ARK and SHARP ASIC proposals due to its lower operating frequency. However, when we compare the performance in terms of clock cycles, HEAP outperforms all prior ASIC proposals and performs at least 1.29× better than all prior

[3] Note that we do not compare the performance of basic operations with CPU and other ASIC works as these works do not report numbers for these basic operations.

TABLE V

Speedup using HEAP when performing bootstrapping operations. Slots = #packed slots in ciphertext while bootstrapping. Bootstrapping time is computed using $T_{mult,a/slot}$ in Equation 3.

Work	Freq. (GHz)	Slots	Time (μs)	Speedup (Time)	Speedup (Cycles)
Lattigo [6]	3.5	2^{15}	101.78	3283×	38313×
GPU [34]	1.2	2^{15}	0.716	23.10×	92.4×
GME [51]	1.5	2^{16}	0.074	2.39×	11.93×
F1 [49]	1	1	254.46	8208×	27334×
BTS-2 [38]	1.2	2^{16}	0.0455	1.47×	5.87×
CL [50]	1	2^{15}	4.19	13.96×	46.49×
ARK [37]	1	2^{15}	0.014	0.45×	1.50×
SHARP [36]	1	2^{15}	0.012	0.39×	1.29×
FAB [2]	0.3	2^{15}	0.477	15.39×	15.39×
HEAP	0.3	2^{12}	0.031	-	-

TABLE VI

Comparison of Average Training Time per iteration for LR model when using sparsely-packed ciphertexts [38].

Work	Time (sec)	Speedup (Time)	Speedup (Cycles)
Lattigo [6]	37.05	5293×	58221×
GPU [34]	0.775	111×	443×
GME [51]	0.054	7.7×	38.57×
F1 [49]	1.024	146×	486×
BTS-2 [38]	0.028	4×	16×
ARK [37]	0.008	1.14×	3.8×
SHARP [36]	0.002	0.29×	0.96×
FAB [2]	0.103	14.71×	14.71×
FAB-2 [2]	0.081	11.57×	11.57×
HEAP	0.007	-	-

work. HEAP achieves better performance because of less compute overhead during bootstrapping and because of the scheme-switching approach that enables the use of a smaller parameter set. Note that F1 performs only a single-slot bootstrapping while others perform a fully-packed bootstrapping. Furthermore, the parameter set used in F1 is not large enough to support fully-packed CKKS bootstrapping following the usual algorithm [13].

F. Application Performance

In this section, we evaluate HEAP when running two applications: Logistic Regression (LR) model training and ResNet-20 inference.

1) Logistic Regression Model Training: We evaluate the application of HEAP in conducting LR model training for binary classification on a subset of MNIST data [22] specifically labeled 3 and 8. This task aligns with the focus of HELR work [29], and it serves as the consistent benchmark for all the comparisons with other works. The designated subset of the dataset comprises 11,982 training samples, each featuring 196 attributes. Following the sequence of homomorphic operations proposed by Han et al. [29] for LR model training, we train the model for 30 iterations and perform a bootstrapping operation after every iteration. We pack only 256 slots in ciphertext i.e., use sparsely-packed ciphertexts following the state-of-the-art implementation [2], [37], [38], [50] in the literature for a fair comparison.

As shown in Table VI, we observe 5293× speedup (when considering absolute training time per iteration) when compared to the existing CPU implementation. This is attributed to the fact that CPUs are not well-suited to the requirements of FHE computing. We observe 111× and 7.7× speedup when compared to the GPU and GME implementations, respectively. While both works involve GPU implementations, the reduced performance gap in comparison to GME is attributed to their microarchitectural enhancements designed to specifically support FHE computing.

TABLE VII

Performance comparison for ResNet-20 inference.

Work	Time (sec)	Speedup (Time)	Speedup (Cycles)
CPU [40]	10602	39708×	436786×
GME [51]	0.982	3.7×	18.39×
CL [50]	0.321	1.20×	4×
ARK [37]	0.125	0.47×	1.56×
SHARP [36]	0.099	0.37×	1.23×
HEAP	0.267	-	-

When compared to F1 (when it uses a similar parameter set as HEAP), BTS-2, and ARK ASIC proposals, we observe 146×, 4×, and 1.14× speedups for HEAP. The performance improvement in HEAP is purely because of the scheme-switching approach followed for bootstrapping. Furthermore, when comparing HEAP with BTS-2 and ARK, we observe a higher performance improvement for LR model training than what we see for fully-packed bootstrapping. This is because sparser packing means less number of LWE ciphertexts and $\mathbf{BlindRotate}$ operations that need to be performed in HEAP. This provides a better overall performance for the LR model training application. However, when compared to SHARP, we observe almost similar performance in terms of the number of cycles as both SHARP and HEAP are highly optimized 36-bit architectures. Thus, HEAP does not have an advantage over SHARP in terms of the time taken to perform scalar operations on the polynomial coefficients. Despite this fact and a lower operating frequency, HEAP achieves similar performance to SHARP, due to a reduction in the % of bootstrapping time spent in a single LR model training iteration when compared to SHARP.

In comparison to FAB (single FPGA implementation) and FAB-2 (implementation on eight FPGAs), we observe 14.71× and 11.57× speedup in HEAP. This is because the operations within CKKS bootstrapping are sequential and are implemented in that order in FAB, limiting the parallelism that can be extracted. Furthermore, the CKKS bootstrapping does not scale well to multiple FPGAs in FAB, again limiting FAB's performance. Thanks to our scheme-switching approach for bootstrapping, we can scale to multiple FPGAs and perform most operations in bootstrapping in parallel. This also helps bring down the overall bootstrapping time in the LR model training. FAB observed $\sim$70% of LR training time spent in bootstrapping while in HEAP that reduces to $\sim$21% Thus, the compute-to-bootstrapping ratio changes from 0.3 to 0.79 per iteration.

2) ResNet-20 Inference: The ResNet-20 inference is performed on the CIFAR-10 dataset following the sequence of homomorphic operations proposed by Lee et al. [39]. For this application, we pack 1024 slots in a ciphertext and thus, perform operations on 1024 LWE ciphertexts when performing the bootstrapping operation. Note that we pack 1024 slots in the ciphertext for a fair comparison with state-of-the-art implementations that follow the same approach. In Table VII, we compare HEAP's performance with CPU, GPU, and other ASIC proposals for ResNet-20 inference. When compared to CPU and GPU implementations, HEAP achieves 39708× and 3.7× performance improvement for ResNet-20 inference. When compared to ASIC proposals, HEAP outperforms CL in absolute

TABLE VIII
RUNTIMES AND SPEEDUP DUE TO SCHEME SWITCHING (SS) AND. DUE TO HARDWARE OPTIMIZATIONS. SPEEDUP 1 IS THE RATIO OF RUNTIME OF SS ON CPU TO RUNTIME CKKS ONLY ON CPU. SPEEDUP 2 IS THE RATIO OF RUNTIME OF SS ON HEAP TO RUNTIME OF SS ON CPU.

Workload	CKKS only on CPU	SS on CPU	Speedup 1	SS on HEAP	Speedup 2
Bootstrapping	4168ms	436ms	9.6×	1.5ms	290.7×
LR Model Training	37.05s [6]	2.39s	15.5×	0.007s	341.4×
ResNet-20 Inference	10602s [40]	309.7s	34.2×	0.267s	1160×

time despite its lower operating frequency. Furthermore, HEAP outperforms the ARK and SHARP ASIC proposals by 1.56× and 1.23×, respectively in terms of the clock cycles. The performance improvement in HEAP is because of the speedup achieved in bootstrapping operation as in ResNet-20 ∼80% of the time is spent in bootstrapping. With our proposed approach, overall time spent in bootstrapping reduces to ∼44% of the total ResNet-20 inference time. Therefore, the ratio of compute-to-bootstrapping time changes from 0.2 to 0.56 for inference.

Although we observe speedups when compared against most works (as shown in Table VII), we do not observe the same speedups as those for LR model training. This is because we are now operating on four times more LWE ciphertexts during bootstrapping. During LR model training, we utilized 256 slots in the ciphertext, resulting in 256 LWE ciphertexts during bootstrapping. However, during ResNet-20 inference, we utilize 1024 slots in the ciphertext, resulting in 1024 LWE ciphertexts during bootstrapping. This means we are generating four times more ciphertext to operate on during ResNet-20 inference compared to LR model training.

3) Model Accuracy: Note that the accuracy of both the LR model and ResNet-20 inference model does not diminish when using our scheme-switching approach to bootstrapping. On HEAP, the observed accuracy for the LR model is ∼97% and for the ResNet-20 model is ∼94%, which is better than the accuracy observed by the state-of-the-art works [36], [37]. This improved accuracy is mainly because, unlike the conventional CKKS bootstrapping algorithm, we do not perform any polynomial evaluation for CKKS bootstrapping with our scheme-switching approach. The polynomial evaluation in conventional bootstrapping is an approximation of a non-linear function such as sine or cosine, and this approximation reduces accuracy.

4) Performance Benefits from Scheme Switching vs. Hardware Optimizations: In Table VIII, we present the runtimes for various workloads for three cases 1) CKKS on CPU; 2) Scheme switching (SS) on CPU; and 3) SS on HEAP. This helps us understand the performance benefits derived solely from the scheme-switching approach versus those achieved solely through hardware optimizations. The enhancements achieved solely from the scheme-switching approach range from 9× to 34×, while those attained with HEAP range from 290× to 1160×. The primary reason for lower performance improvement when using SS on CPU is the limited parallelization opportunities available on the CPU compared to HEAP.

VII. DISCUSSION

In this section, we present two other aspects of our proposed scheme-switching approach and HEAP implementation.

A. Standalone TFHE scheme on HEAP

As discussed throughout the paper, HEAP is readily applicable to the CKKS scheme, enabling an arbitrary number of operations with support for bootstrapping. Here, we briefly discuss HEAP's applicability to support standalone TFHE scheme, if required. We already implemented two of the key operations in TFHE scheme, i.e., $\mathbf{ExternalProduct}$ and $\mathbf{BlindRotate}$ operations. $\mathbf{BlindRotate}$ is the core of programmable bootstrapping ($\mathbf{PBS}$) operation in TFHE, implying that $\mathbf{BlindRotate}$ with $\mathbf{PBS}$ keys can perform $\mathbf{PBS}$ in a straightforward way. The $\mathbf{Extract}$ operation is also implemented as a part of $\mathbf{BlindRotate}$ operation.

The $\mathbf{KeySwitch}$ operation consists of $\mathbf{Decomposition}$ and $\mathbf{ExternalProduct}$ with the evaluation keys. $\mathbf{Decomposition}$ is a gadget decomposition operation similar to the one in the CKKS scheme and is performed using the same set of operations, but using a different set of pre-computed values. The $\mathbf{ModulusSwitch}$ operation can be performed using scalar multiplication operations. The $\mathbf{CMux}$ operation can be mapped via simple multiplication, addition, and subtraction operations.

Finally, we have the $\mathbf{InternalProduct}$ operation that needs to be performed as part of the TFHE scheme. This $\mathbf{InternalProduct}$ between GGSW ciphertexts can be defined using the $\mathbf{ExternalProduct}$. Typically, a GGSW ciphertext can be viewed as a list of GLWE ciphertexts. As the $\mathbf{ExternalProduct}$ is a product between GLWE and GGSW ciphertexts, the $\mathbf{InternalProduct}$ can be defined as a list of independent $\mathbf{ExternalProduct}$s. For each of these independent $\mathbf{ExternalProduct}$s, one of the inputs is GGSW ciphertext, and all the GLWE ciphertexts form the second GGSW ciphertext input. The output of these independent $\mathbf{ExternalProduct}$s is the GLWE ciphertexts, which can be put together into one single GGSW ciphertext.

B. Applicability to other Compute Platforms

While we prototype our proof-of-concept implementation on single and multiple FPGA systems, it is essential to note that our proposed scheme-switching approach and optimizations are not exclusive to FPGA. The scheme-switching approach for bootstrapping can be readily adapted to enhance CKKS bootstrapping on various underlying compute platforms, including CPU, GPU, and ASIC.

VIII. RELATED WORK

CKKS Acceleration Efforts: Recent research on FHE accelerators has been predominantly centered around the acceleration of the CKKS FHE scheme. In the early software implementations [15] of bootstrapping, attempts were made to minimize the number of rotations needed in the linear transformation step (which involves converting ciphertexts from coefficient to evaluation representation and back) by incorporating the baby-step giant-step (BSGS) algorithm [28]. Chen et al. [10] introduced a method for level-collapsing in combination with the BSGS algorithm as a means to enhance the efficiency of the linear transformation step and reduce the number of required rotations. Subsequently, Han and Ki [30] presented a hybrid key-switching method designed to effectively control the noise introduced during the key-switching operation. Bossuat et al. [6] went a step further in reducing the operational complexity of linear transformations by streamlining rotations through the strategic application of the hybrid key-switching approach. This optimization effectively minimizes the number of basis conversion operations and, in turn, decreases the accesses to main memory.

Jung et al. [34] introduced the first GPU-based implementation of CKKS bootstrapping. Their optimization strategies, including both inter- and intra-kernel fusion, predominantly concentrate on enhancing memory bandwidth utilization. One of the recent GPU implementations [51] proposed several microarchitectural extensions to the AMD GPU to accelerate the CKKS FHE scheme. To reduce main memory bottlenecks, the authors integrate a lightweight on-chip compute unit side hierarchical interconnect to retain ciphertext in cache across FHE kernels. They also propose a locality-aware block scheduler that exploits the temporal locality available in various FHE primitive blocks.

Simultaneously, there have been initiatives to accelerate CKKS on both FPGA and ASIC platforms. HEAX [48] emerged as an early FPGA-based accelerator for FHE which accelerated only CKKS encrypted multiplication, with all other operations being delegated to a host processor. In a similar vein, there are a few other FPGA-based acceleration efforts specific to CKKS that exclusively implement NTT [55] and key switch [31] operations. One of the more recent FPGA implementations [2] implemented CKKS packed bootstrapping for the first time on FPGA with a support for practical parameter set. The first ASIC design was proposed by Samardzic et al. [49]. The same authors proposed a series of additional optimizations in their subsequent work [50]. In parallel, Kim et al. proposed three other ASIC designs namely BTS [38], ARK [37], and SHARP [36]. These designs incorporate various optimizations and have demonstrated remarkable performance in bootstrapping.

TFHE Acceleration Efforts: Accelerators based on TFHE have been the subject of numerous efforts aimed at enhancing the performance of TFHE operations. These initiatives have sought to accelerate the TFHE scheme on a range of platforms, including CPU [44], GPU [18], [20], FPGA [23], [45], [56], ASIC [33], [47], and compute-enabled RAM [53]. Morshed et al. [44] explored both CPU and GPU parallelization of TFHE. They use parameters that enable 110 bits of security along with a 32-bit torus discretization. They report programmable bootstrapping (PBS) runtimes of $\sim$113ms for CPU and 22ms for GPU. The cuFHE library [20] uses CUDA to implement TFHE on GPUs, achieving a latency of 0.5ms for PBS.

NuFHE's GPU acceleration uses either NTT or FFT, with FFT outperforming NTT. One of the FPGA acceleration efforts [56] employs NTT for polynomial multiplication, achieving a 1.3× throughput improvement compared to NuFHE. ASIC acceleration in Matcha [33] outperforms prior accelerators using bootstrapping key unrolling [8] that reduces blind rotation iterations at the cost of increased key size. Strix [47] advances TFHE acceleration by utilizing two-level ciphertext batching and specialized functional units. Their streaming architecture-based accelerator can support various TFHE parameters and achieves 7.4× higher throughput than Matcha. Takeshita et al. [53] observed some slowdown in PBS relative to state-of-the-art works using GPU and FPGA, indicating that compute-enabled RAM's relative strength of parallelism is not effective for the TFHE scheme.

Scheme Switching Approaches: OpenFHE [3] represents the initial open-source CPU library with the capability to switch between the CKKS and the DM/CGGI scheme. This flexibility enables the execution of non-linear operations, such as comparisons, using the DM/CGGI scheme. Moreover, there are ongoing efforts to extend the support for broader scheme-switching within the library, with plans to incorporate this functionality in future updates. Chimera [7] employs a scheme-switching approach to switch between CKKS/BFV and TFHE schemes to perform faster inner products in the TFHE domain.

Pegasus [41] proposed a scheme-switching methodology to switch between the CKKS and the FHEW scheme to accelerate non-linear functions in machine learning applications. They accelerate sigmoid, ReLU, min, max, division, sorting, and max-pooling functions and demonstrate a reduction of conversion keys from 80GB to 12MB required to switch between the schemes. Several other works [19], [25], [43] have shown that conversions between LWE and RLWE combined with blind rotation technique can be an efficient methodology for evaluating non-linear functions. However, none of these aforementioned works use the scheme-switching approach in terms of accelerating the CKKS bootstrapping operation itself. Kim et al. [35] were the first to propose the idea of performing the CKKS bootstrapping using a combination of RLWE and LWE ciphertexts. We leverage their basic idea to propose a novel scheme-switching approach that parallelizes the CKKS bootstrapping step to accelerate FHE-based computing.

IX. Conclusion

Despite the advancements in FHE schemes, bootstrapping remains a bottleneck, consuming a significant portion of the execution time. Our work introduced HEAP, an accelerator that uses a novel hybrid scheme-switching approach, capitalizing on the CKKS SIMD nature for fundamental operations and seamlessly transitioning to the TFHE scheme during bootstrapping. We proposed a variety of hardware optimizations in HEAP—from modular arithmetic level to NTT and $\mathbf{BlindRotate}$ datapath optimizations. A proof-of-concept implementation of HEAP on an Alveo U280 FPGA achieves impressive improvement in bootstrapping performance when scaled to eight FPGAs. On average, HEAP demonstrated bootstrapping speeds that are 3283×, 12.7×, 4×, and 15.39× faster than those of current state-of-the-art CPU, GPU, ASIC, and FPGA implementations. We assessed the efficacy of this hybrid approach for bootstrapping through the evaluation of two applications: logistic regression (LR) model training and ResNet-20 inference. For LR model training, HEAP achieved an average speedup of 5293×, 59.35×, 37.8×, and 13.14× compared to existing state-of-the-art CPU, GPU, ASIC, and FPGA implementations, respectively. Similarly, in the case of ResNet-20 inference, HEAP delivered an average speedup of 39708×, 3.7×, and 1.2× faster than existing state-of-the-art CPU, GPU, and ASIC implementations. This research contributes to the ongoing efforts to enhance the practicality and scalability of FHE schemes, demonstrating a promising stride towards making secure computation on encrypted data more efficient and viable for real-world applications.

Acknowledgment

This research was supported by RedHat Collaboratory.

References

[1] R. Agrawal, L. de Castro, C. Juvekar, A. Chandrakasan, V. Vaikuntanathan, and A. Joshi, "Mad: Memory-aware design techniques for accelerating fully homomorphic encryption," *MICRO'23, October 28-November 1, 2023, Toronto, ON, Canada*, 2023.

[2] R. Agrawal, L. de Castro, G. Yang, C. Juvekar, R. Yazicigil, A. Chandrakasan, V. Vaikuntanathan, and A. Joshi, "Fab: An fpga-based accelerator for bootstrappable fully homomorphic encryption," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 882–895.

[3] A. Al Badawi, J. Bates, F. Bergamaschi, D. B. Cousins, S. Erabelli, N. Genise, S. Halevi, H. Hunt, A. Kim, Y. Lee *et al.*, "Openfhe: Open-source fully homomorphic encryption library," in *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, 2022, pp. 53–63.

[4] 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.

[5] J. W. Bos, M. Naehrig, and J. V. D. Pol, "Sieving for shortest vectors in ideal lattices: a practical perspective," *International Journal of Applied Cryptography*, vol. 3, no. 4, pp. 313–329, 2017.

[6] 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.

[7] C. Boura, N. Gama, M. Georgieva, and D. Jetchev, "Chimera: Combining ring-lwe-based fully homomorphic encryption schemes," *Journal of Mathematical Cryptology*, vol. 14, no. 1, pp. 316–338, 2020. [Online]. Available: https://doi.org/10.1515/jmc-2019-0026

[8] F. Bourse, M. Minelli, M. Minihold, and P. Paillier, "Fast homomorphic evaluation of deep discretized neural networks," in *Advances in Cryptology–CRYPTO 2018: 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19–23, 2018, Proceedings, Part III 38*. Springer, 2018, pp. 483–512.

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

[10] 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.

[11] H. Chen, W. Dai, M. Kim, and Y. Song, "Efficient homomorphic conversion between (ring) lwe ciphertexts," in *International Conference on Applied Cryptography and Network Security*. Springer, 2021, pp. 460–479.

[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] ——, "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, M. Hhan, S. Hong, and Y. Son, "A hybrid of dual and meet-in-the-middle attack on sparse and ternary secret lwe," *IEEE Access*, vol. 7, pp. 89 497–89 506, 2019.

[15] 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.

[16] 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.

[17] ——, "TFHE: Fast fully homomorphic encryption library," August 2016, https://tfhe.github.io/tfhe/.

[18] I. Chillotti, M. Joye, D. Ligier, J.-B. Orfila, and S. Tap, "Concrete: Concrete operates on ciphertexts rapidly by extending tfhe," in *WAHC 2020-8th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, 2020.

[19] I. Chillotti, M. Joye, and P. Paillier, "Programmable bootstrapping enables efficient homomorphic inference of deep neural networks," in *Cyber Security Cryptography and Machine Learning: 5th International Symposium, CSCML 2021, Be'er Sheva, Israel, July 8–9, 2021, Proceedings 5*. Springer, 2021, pp. 1–19.

[20] W. Dai and B. Sunar, "cuhe: A homomorphic encryption accelerator library," in *Cryptography and Information Security in the Balkans: Second International Conference, BalkanCryptSec 2015, Koper, Slovenia, September 3-4, 2015, Revised Selected Papers 2*. Springer, 2016, pp. 169–186.

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

[22] 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.

[23] S. Gener, P. Newton, D. Tan, S. Richelson, G. Lemieux, and P. Brisk, "An fpga-based programmable vector engine for fast fully homomorphic encryption over the torus," in *SPSL: Secure and Private Systems for Machine Learning (ISCA Workshop)*, 2021.

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

[25] A. Guimarães, E. Borin, and D. F. Aranha, "Revisiting the functional bootstrap in tfhe," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 229–253, 2021.

[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] S. Halevi and V. Shoup, "Algorithms in helib," in *Advances in Cryptology – CRYPTO 2014*, J. A. Garay and R. Gennaro, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 554–571.

[28] ——, "Faster homomorphic linear transformations in helib," in *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part I*, ser. Lecture Notes in Computer Science, H. Shacham and A. Boldyreva, Eds., vol. 10991. Springer, 2018, pp. 93–120. [Online]. Available: https://doi.org/10.1007/978-3-319-96884-1_4

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

[30] 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.

[31] 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.

[32] "Ibm data breach report," Online: https://www.ibm.com/reports/data-breach, January 2023.

[33] L. Jiang, Q. Lou, and N. Joshi, "Matcha: A fast and energy-efficient accelerator for fully homomorphic encryption over the torus," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022, pp. 235–240.

[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*, vol. 2021, no. 4, p. 114–148, Aug. 2021. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/9062

[35] A. Kim, M. Deryabin, J. Eom, R. Choi, Y. Lee, W. Ghang, and D. Yoo, "General bootstrapping approach for rlwe-based homomorphic encryption," *IEEE Transactions on Computers*, 2023.

[36] J. Kim, S. Kim, J. Choi, J. Park, D. Kim, and J. H. Ahn, "Sharp: A short-word hierarchical accelerator for robust and practical fully homomorphic encryption," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–15.

[37] 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.

[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] E. Lee, J.-W. Lee, J. Lee, Y.-S. Kim, Y. Kim, J.-S. No, and W. Choi, "Low-complexity deep convolutional neural networks on fully homomorphic encryption using multiplexed parallel convolutions," in *International Conference on Machine Learning*. PMLR, 2022, pp. 12 403–12 422.

[40] J.-W. Lee, H. Kang, Y. Lee, W. Choi, J. Eom, M. Deryabin, E. Lee, J. Lee, D. Yoo, Y.-S. Kim *et al.*, "Privacy-preserving machine learning with fully homomorphic encryption for deep neural network," *IEEE Access*, vol. 10, pp. 30 039–30 054, 2022.

[41] W.-j. Lu, Z. Huang, C. Hong, Y. Ma, and H. Qu, "Pegasus: bridging polynomial and non-polynomial evaluations in homomorphic encryption," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 1057–1073.

[42] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," in *Advances in Cryptology – EUROCRYPT 2010*, H. Gilbert, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 1–23.

[43] D. Micciancio and J. Sorrell, “Ring packing and amortized fhew bootstrapping,” *Cryptology ePrint Archive*, 2018.

[44] T. Morshed, M. M. Al Aziz, and N. Mohammed, “Cpu and gpu accelerated fully homomorphic encryption,” in *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2020, pp. 142–153.

[45] K. Nam, H. Oh, H. Moon, and Y. Paek, “Accelerating n-bit operations over tfhe on commodity cpu-fpga,” in *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, 2022, pp. 1–9.

[46] Norton and Silberger, “Parallelization and performance analysis of the cooley–tukey fft algorithm for shared-memory architectures,” *IEEE Transactions on Computers*, vol. 100, no. 5, pp. 581–591, 1987.

[47] A. Putra, Y. Chen, J. Kim, J.-Y. Kim *et al.*, “Strix: An end-to-end streaming architecture with two-level ciphertext batching for fully homomorphic encryption with programmable bootstrapping,” *arXiv preprint arXiv:2305.11423*, 2023.

[48] 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.

[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*, ser. MICRO ’21. New York, NY, USA: Association for Computing Machinery, 2021, p. 238–252. [Online]. Available: https://doi.org/10.1145/3466752.3480070

[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] K. Shivdikar, Y. Bao, R. Agrawal, M. Shen, G. Jonatan, E. Mora, A. Ingare, N. Livesay, J. L. AbellÁN, J. Kim, A. Joshi, and D. Kaeli, “Gme: Gpu-based microarchitectural extensions to accelerate homomorphic encryption,” in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023, p. 670–684.

[52] Y. Son and J. H. Cheon, “Revisiting the hybrid attack on sparse secret lwe and application to he parameters,” in *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, ser. WAHC’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 11–20. [Online]. Available: https://doi.org/10.1145/3338469.3358941

[53] J. Takeshita, D. Reis, T. Gong, M. Niemier, X. S. Hu, and T. Jung, “Accelerating finite-field and torus fhe via compute-enabled (s)ram,” *IEEE Transactions on Computers*, pp. 1–14, 2023.

[54] 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.

[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] T. Ye, R. Kannan, and V. K. Prasanna, “Fpga acceleration of fully homomorphic encryption over the torus,” in *2022 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2022, pp. 1–7.