coxHE: A software-hardware co-design framework for FPGA acceleration of homomorphic computation

Mingqin Han*†, Yilan Zhu*†, Qian Lou‡, Zimeng Zhou†, Shanqing Guo†, Lei Ju*†
† Key Laboratory of Cryptologic Technology and Information Security, Ministry of Education, Shandong University
School of Cyber Science and Technology, Shandong University, Qingdao, China
‡ Intelligent Systems Engineering department, Indiana University, Bloomington, Indiana

***Abstract*—Data privacy becomes a crucial concern in the AI and big data era. Fully homomorphic encryption (FHE) is a promising data privacy protection technique where the entire computation is performed on encrypted data. However, the dramatic increase of the computation workload restrains the usage of FHE for the real-world applications. In this paper, we propose an FPFA accelerator design framework for CKKS-based HE. While the KeySwitch operations are the primary performance bottleneck of FHE computation, we propose a low latency design of KeySwitch module with reduced intra-operation data dependency. Compared with the state-of-the-art FPGA based key-switch implementation that is based on Verilog, the proposed high-level synthesis (HLS) based design reduces the operation latency by 40%. Furthermore, we propose an automated design space exploration framework which generates optimal encryption parameters and accelerators for a given application kernel and the target FPGA device. Experimental results for a set of real HE application kernels on different FPGA devices show that our HLS-based flexible design framework produces substantially better accelerator design compared with a fixed-parameter HE accelerator in terms of security, approximation error, and overall performance.**

***Index Terms*—Homomorphic encryption, FPGA acceleration, high-level synthesis, design space exploration**

I. Introduction

Data privacy becomes an increasing important concern in the AI and big data era. When user data are manipulated on the cloud servers, how to ensure data privacy is now a critical trust gap between the data owner and the cloud service provider. Fully homomorphic encryption (FHE) is a very promising data privacy framework, where the entire computation on the server is operated on the encrypted data [1]. FHE has drawn great attention from both academia and industry in the past few years. For instance, Microsoft has launched the open-source SEAL library for FHE [2]. Google has recently open-sourced a general-purpose "transpiler" to convert high-level code to be used with Fully Homomorphic Encryption [3].

Many different schemes has been proposed to realize FHE, such as BGV [4], BFV [5], TFHE [6], and CKKS [7]. However, one of the critical obstacles to deploy these schemes for real-world practice is the enormous computation overhead. In general, a single float or integer data would be encrypted into a series of polynomials involving thousands of coefficients, which makes the FHE schemes incur 3-4 orders of magnitude higher computation and storage overhead. For instance, Cryptonets [8] requires 250 seconds to perform CNN inference with FHE, while the latency would be at millisecond level for doing the same task with plaintext.

As a result, design domain specific architecture and accelerator for FHE is of paramount importance for practical FHE kernels. Dai et al. [9] proposed a GPU-based FHE library "cuHE". In April 2021, Intel has released Homomorphic Encryption Acceleration Library (HEXL) [10] which uses the Intel Advanced Vector Extensions 512 (Intel AVX512) instruction set to enable high-performance multi-threaded execution of the SEAL library. FPGA acceleration for basic FHE operations has been studied in [11]–[13]. Recently in [14], Riazi et al. proposed FPGA-based accelerator for CKKS FHE, where the fundamental number-theoretic transform (NTT) blocks and high-level parallel operations can be scaled to different encryption parameters and hardware resources. In [15], multi-level parallelism design has been exploited to achieve better FPGA resource utilization for the Multiply-Accumulate operations.

Existing FPGA accelerators focus on the hardware design of low- and/or high- level FHE operations (e.g., NTT, KeySwitch) with the exploration of the structure and data dependency of FHE operations. However, the kernel-level information is crucial for practical FHE acceleration, because:

- Given an FHE application, many encryption parameter combinations are available (e.g., related to the multiplication depth of the software kernel). The encryption parameter selection leads to trade-offs between the security level, hardware resource usage, and FHE approximation error.
- The software kernel determines the required FHE operations, dependency, and frequency. In order to fully utilize the FPGA resource for an optimal accelerator design, the kernel-level information must be incorporated in the design flow.

In this paper, we propose a flexible FPGA co-design acceleration framework for CKKS-based FHE kernels (coxHE). For a given software FHE kernel and target FPGA device, the proposed coxHE framework automatically determines the encryption parameters as well as FHE operations hardware core design. To the best of our knowledge, this is the first FPGA FHE accelerator that is entirely built with high-level synthesis (HLS) design methodology, which provides extreme flexible design and fast deployment across different FPGA devices. The contributions of this paper are summarized as follows:

This work is supported by the Natural Science Foundation of China (grant numbers 92064008, 62102230), Shandong Provincial Natural Science Foundation (grant numbers ZR2020LZH002, ZR2021QF019, ZR2020MF055, ZR2021LZH007), and Qilu Young Scholar Program of Shandong University.

* Mingqin Han and Yilan Zhu are co-first authors of the article, Lei Ju (Email: julei@sdu.edu.cn) is the corresponding author.

- We remove data dependencies within the bottleneck KeySwitch operation to reduce its latency by up-to 48%, compared with the state-of-the-art FPGA CKKS accelerator. Meanwhile, the HLS-based implementation of the FHE operations are parametric to enable fast design space exploration (DSE) between hardware resource usage and performance.
- The coxHE framework considers multiple design objectives including security level, performance, and approximation error. It automatically determines the Pareto optimal design solutions within the enormous system design space composed by possible encryption parameters and hardware core design choices (e.g., parallel levels, data placement).
- While most literature work on FPGA FHE accelerator evaluate the design at FHE operation level (e.g., NTT, KeySwitch), coxHE framework contains the full-fledged design flow and is evaluated on a set of real-world FHE software kernels. Compared to Intel HEXL based on multi-threading and AVX512 on i7-8700@3.70GHz, the generated accelerator achieves up-to 42.7X energy efficiency on Xilinx ZCU102.

II. Background

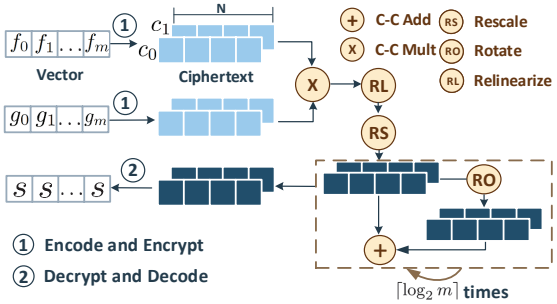


Fig. 1. Encrypted vector multiplication. The circled operations are high-level CKKS operations, and the dashed box is repeated $\lceil \log_2 m \rceil$ times.The output vector contains the results where $s = \sum_{i=1}^{m} f_i \cdot g_i$.

The primary advantage of FHE is to enable computation on encrypted data without decrypting them. It is attributed to the homomorphism of its addition and multiplication which can be represented by the following formula.

$$\begin{aligned} HE.enc(a+b) &= HE.enc(a) + HE.enc(b) \\ HE.enc(a \cdot b) &= HE.enc(a) \cdot HE.enc(b) \end{aligned} \tag{1}$$

CKKS is a popular FHE scheme that is designed for the arithmetic of approximate numbers to support homomorphic evaluations on floating-point numbers. Fig. 1 illustrates the process of the CKKS scheme for encrypted vector multiplication. The data is encoded and encrypted as a pair of ciphertext polynomials. Ciphertexts are then input as operands to homomorphic addition and multiplication operations at the server side, and obtain the result ciphertexts. Finally, the result ciphertexts are decrypted and decoded to obtain the correct results at the user side. Compared with encode, encrypt, decrypt and decode operations, the homomorphic evaluation operations are generally the bottleneck of execution time and energy consumption for CKKS scheme. Therefore, the homomorphic evaluation optimization is crucial for CKKS optimization.

A ciphertext is denoted as two polynomials $(\boldsymbol{c}_0, \boldsymbol{c}_1)$, where $\boldsymbol{c}_k = \sum_{i=0}^{N-1} a_i x^i \pmod{Q}$, $k \in (0, 1)$. N is usually selected as 2^{10}, 2^{11} ... 2^{14}, 2^{15}, and the bit-width of Q is up to hundreds of bits. The computationally intensive modular arithmetic on big integers cause the dramatic performance degradation for the FHE computation. Fortunately, the RNS variant of CKKS [16] could divide Q into small q_i, $Q_l = \prod_{i=1}^{l} q_i$ and make high parallelism possible in FPGA.

To perform homomorphic addition (HADD) and multiplication (HMULT), CKKS provides support for plaintext-ciphertext (P-C) operation, ciphertext-ciphertext (C-C) operation, rescaling, relinearization and rotation. Among these operations, P-C and C-C operations are basic and simple to implement. Relinearization is performed after C-C Mult to prevent the expansion of the number of the ciphertext polynomials. It is worth mentioning that CKKS supports SIMD-style technique that can encode a vector into plaintext and then encrypt it. That is, the same operation can be performed on each element of the vector with only a homomorphic operation. Rotation can change the order of the vector elements in one direction when it is encrypted. Then we can make a sum of all the elements in a vector through consecutive rotation and C-C Add. It is easy calculate the number of consecutive rotations as $\lceil \log_2 m \rceil$ shown in Fig. 1. Therefore, this feature is often used in computing homomorphic vector multiplication and matrix multiplication if the elements are packed in a ciphertext. Besides, Rescaling can be used to reduce the number of small modulus q_i and avoid the underlying plaintext values in the ciphertext from blowing up.

III. Motivation

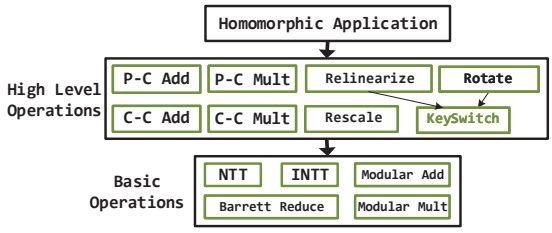


Fig. 2. The hierarchy of homomorphic operations.

KeySwitch Acceleration. Fig. 2 shows the hierarchy of various CKKS homomorphic operations. Among the high-level operations, Relinearize and Rescale both invoke the KeySwitch operation, with some additional pre-/post- operations. Generally, KeySwitch is the most computationally intensive and time-consuming operation in FHE. For example, the running time of one single KeySwitch invocation is almost 10 times that of P-C multiplication, 7 times of C-C multiplication, 2 to 3 times of Rescale. The total time consumed by KeySwitch operations is even as high as 90% for homomorphic matrix multiplication. Therefore, designing an efficient KeySwitch accelerator is crucial for FHE accelerator on FPGA.

HEAX [14] has designed a fine-grained pipeline architecture for KeySwitch, which achieves optimal throughput when the pipeline is perfectly filled. However, in this paper, we argue that attention should also be paid to the latency optimization.

As the encrypted vector multiplcation example shown in Fig. 1, the KeySwitch operation (invoked by the Rotate opera-

tion) in the dashed box will be executed $\lceil \log_2 m \rceil$ times. However, due to the data dependency exists between consecutive loop iterations, the execution of the KeySwitch operation cannot be pipelined. Therefore, the latency (rather than the pipelined throughput) is the key optimization goal in this scenario. And this is a very common case in typical CKKS kernels.

DSE for multi-objective optimization. Existing work on FPGA acceleration of FHE focus on the hardware design of FHE low- and/or high-level operations to improve the performance of FHE. However, a practical CKKS kernel accelerator typically has multiple design objective including security, performance (energy efficiency), and approximation error (computation accuracy). Given a software kernel, hundreds of combinations of encryption parameters (i.e., N and q_i) are possible, each has potentially different impacts on the design objectives. Meanwhile, for a given encryption parameters selection, we have many possible hardware configurations for each CKKS low- and high-level operation (e.g., intra-parallelism, data layout) on a particular FPGA device. Therefore, automatic design space exploration is vital for FHE kernel acceleration.

TABLE I
TWO POSSIBLE DESIGN DESIGN CHOICES.

	N	q_i	security	latency(ms)	error
S1	8192	33	192	34.56	9.53×10^{-6}
S2	8192	37	192	67.34	9.53×10^{-7}

For instance, assume that we accelerate a CKKS kernel for plain-cipher matrix multiplication on Xilinx ZCU104 device. If we set the encryption parameters N and q_i bit-width to be 16384 and 40, it requires at least 47.19Mbit on-chip memory, which is larger than the total available on-chip memory of ZCU104 and thus not a feasible encryption parameter selection. On the other hand, Table I shows two possible design solutions S1 and S2. The encryption parameters selected for S2 lead to a KeySwitch accelerator with lower internal parallelism due to hardware resource constraints. Therefore, considering the same security level and similar approximation error (between 10^{-6} and 10^{-7}), S1 could be a better design choice for most circumstances. In the proposed framework, we perform automatic design space exploration to generate Pareto optimal solutions with trade-offs among different design objectives, out of the large design space due to encryption parameter selection and possible hardware configurations.

IV. ACCELERATOR ARCHITECTURE

In this section, we first introduce our low-latency accelerator design of the KeySwitch operation in Section IV-A. In Section IV-B, we present the parametric design of CKKS operations based on the high-level synthesis (HLS) methodology [17], which enables automatic design space exploration for the system-level software hardware co-design.

A. Low-latency KeySwitch acceleration

As we have discussed in Section III, KeySwitch is highly optimized in the proposed coxHE framework to reduce latency. As shown in the Fig. 3, the coxHE eliminates two data dependencies of KeySwitch to reduce the execution time compared with HEAX [14]. In HEAX, the INTT1 module needs to wait for k iterations of the former layer which brings a long halt

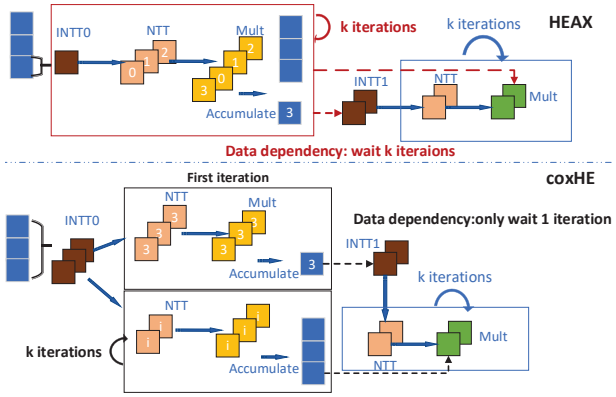


Fig. 3. Elimination of data dependency in KeySwitch.

for the subsequent modules. We find that the INTT1 only needs the output of the input polynomials computing with the last modulus, while each iteration consumes all moduli in HEAX. Then we eliminate this unnecessary dependency so that the KeySwitch operation achieves lower latency. Specifically, we reorganize the computing order of the moduli and their corresponding polynomials in the former layer while keeping the same whole computing amounts. coxHE firstly computes the last modulus within the input polynomials which is essential and enough for INTT1. Thus, INTT1 only need wait one iteration of the former layer to start its computing task, so the overall latency would be reduced a lot. And the latency reduction becomes significant for a larger number of moduli which is correlated to the encryption parameter selection.

B. HLS-based accelerator design

Listing 1. Coding style

```
typedef ap_uint<BITWIDTH> UDTYPE;
template<unsigned mod_count>
void P-C_Mult(...)
{#pragma HLS INLINE off
    ...
    for (unsigned i = 0; i < 2; i++)
        for (unsigned j = 0; j <
            coeff_mod_count; j++)
        #pragma HLS UNROLLL factor=4
            Modular_Mult(...);}
```

HLS provides the inbuilt directives allowing us to reconfigure our design without changing significantly the original code. Our coxHE framework is designed to support polynomials of any degree and any coefficients size. The coxHE also supports to change the coarse- or fine- grained of parallelism among both high level modules and basic operations listed in Fig. 2. Taking the P-C Mult operation as an example, we can adjust its fine-grained parallelism by adding the UNROLL directive and its factor for the corresponding loops to achieve design tradeoffs between resource usage and performance.

To change the bit-width of the q_i, we can use ap_uint data type. As for data arrangement, coxHE can set various memory layout for better optimization, e.g., the on-chip memory partition or resource type pragmas. We also use the template coding style which makes our code flexible to reconfigure. The flexibility of coxHE makes it possible to explore the wide design space towards various application kernels.

V. Design space exploration

It is generally a non-trivial task for a system developer to find the optimal design choice for a given FHE kernel and FPGA device. In this section, we formulate a multi-objective optimization problem by modeling the performance, resource utilization and approximation error for FHE operations, and use them for fast design space exploration. The proposed framework solve the optimization problem to generate a set of Pareto optimal designs with HE parameters and hardware configurations, which can be used by the HLS design flow to automatically generate FPGA accelerators.

A. Problem formulation

The resource-constrained performance optimization problem can be formulated as follows:

$$\min \{L, E\}, \max S$$
$$s.t. \begin{cases} R = f_R(N, Pintra, Q, packnum, portnum) \leq R_{max}, \\ L = f_L(N, Pintra, packnum, portnum), \\ E = f_E(N, Q), \\ S = f_S(N, Q). \end{cases} \tag{2}$$

For a given application and platform, our optimization goal is to minimize the execution latency (L) as well as the approximation error (E) and maximize the security level (S). Our design factors include the polynomial length (N), the set of small modulus Q where $Q = \{q_i | i \in (1, l)\}$, the basic modules' degree of parallelism ($Pintra$), the number of input data packed on a single word ($packnum$) and the number of AXI ports we actually used in a high level module ($portnum$). The optimization goals can be defined as several functions $E = f_E, L = f_L$ and $S = f_S$ which take several design factors as arguments. Also, the optimization goals should satisfy the constraint of resource limit. We define the resource usage function $R = f_R$ which is affected by all design factors. These functions can be calculated by our performance model, resource model, error and security model, respectively, which are discussed in the next subsections.

According to [16], the value range of N can be $2^{11} \sim 2^{15}$ and q_i can be 14bit $\sim$ 60bit. Based on our experiment observation, we found that it is hard for the synthesis tool to generate the IP core if $Pintra > 32$ which means that $Pintra$ can be set to $2^0 \sim 2^5$. Thus the size of the design space is roughly 894-points. Our approach of solving the optimization problem is based on branch and bound search to reduce some unnecessary points, e.g., if we use ZCU102 to evaluate the benchmark c from Table II, we can set $N = 4096, q_i = 25$ bit and $Pintra = 8$ according to our model. Then we testify the point in the synthesis tool and see if it can meet the plaform's resource constraint as our expectation. If not, we try to reduce the $Pintra$ by dividing it to 2. Thereby, for a specific N and Q, we do not need evaluate all designs considering the value range of $Pintra$. The process of DSE costs roughly 1-2 days to finish. If the platform has more resources resulting in a bigger design space, it is necessary to design more efficient methods such as the gene algorithm to make the process of DSE faster.

B. Performance model

As we mentioned before, the basic modules consist of Add (Modular Add), Mult (Modular Mult), BarrettReduce and NTT/INTT which can be organized to C-C Mult, C-C Add, KeySwitch, Rescale module and so on. The execution time of these modules is analytically modeled as several functions of design variables which are validated by performing the design points and running them on the FPGA accelerator. In our design, the inputs of the NTT/INTT are all existed on on-chip BRAM. According to the NTT/INTT's compute flow [14], we can easily get the execution time of NTT/INTT:

$$L_{NTT} = logN * (N/2/PIntra_{NTT}) \tag{3}$$

For other basic modules, i.e., ModAdd, ModMult and BarrettReduce, they all compute the data sequentially. Thus the latency of these modules can be expressed as several functions related to the degree of parallelism as follows:

$$L_{Module} = N/PIntra_{Module} \tag{4}$$

For those data that is accessed sequentially, such as the input data and Relinearization/Galois keys which need enormous memory space to store, we arrange them on the DRAM to enable burst access and reduce the on-chip memory usage. And we also use a data package strategy to enable the modules process in parallel. For example, as shown in Fig 3, the Mult modules in the former layer need to read Relinearization/Galois keys. To reduce the overall latency of KeySwitch module, we need guarantee the Mult modules have similar latency with other modules otherwise there will be unnecessary halt in the internal pipeline structure of Keyswitch. So our strategy is that we package multiple Relinearization/Galois keys or input data together to make the Mult module process in parallel. According to the performance model of the basic modules, $packnum$ and $portnum$ should satisfy the following relationships:

$$Pintra_{Module} = packnum \cdot portnum/4 \tag{5}$$

For KeySwitch and Rescale, we ensure that a group of the same basic modules run in parallel as shown in Fig 3. Based on the performance models of basic modules, we can get the latency of Keyswitch module and Rescale module which belongs to high-level operations:

$$L_{KeySwitch} = L_{INTT} + \max\{L_{module}\} \cdot (k + 4) \tag{6}$$
$$L_{Rescale} = L_{INTT} + L_{BarrettReduce} + L_{NTT} \tag{7}$$

where k denotes the number of moduli.Therefore, the overall latency can be formulated as follow:

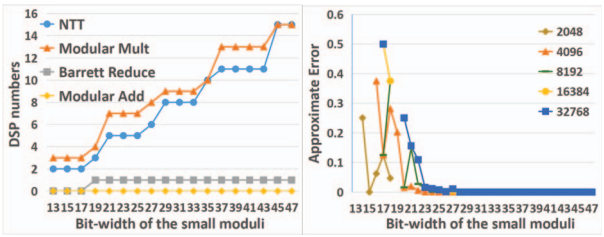
$$L = \alpha \cdot L_{KeySwitch} + \beta \cdot L_{rescale} + \gamma \cdot L_{Mult} + \lambda \cdot L_{Add} \tag{8}$$

where $\alpha, \beta, \gamma, \lambda$ are constants for a given FHE kernel.

C. Resource model

It is not feasible to analytically model the FPGA computing and logical resource utilization of the kernel implemented by HLS because of the optimizations performed in the HLS tools. Therefore, we use the synthesis results to empirically model the FPGA resource utilization.

We present the DSP resource usage of the basic modules in different bit-widths of the small modulus q_i while setting the intra parallel degree to 1 as Fig.4(a). As we can see, for NTT/INTT, ModMult and BarrettReduce, the DSP functional relationship presents a ladder-like overall upward trend, which



(a) Resource model. (b) Error model.

Fig. 4. The DSP and error for varied bit-width in basic modules.

means that different bit-widths may correspond to the same DSP usage. However, for a given bit-width range, the approximation error with different bit-width varies from each other which is shown in Fig.4(b). For all the basic modules, the amount of FF and LUT increases roughly with the bit-width grows.

D. Error and security model

This part involves the influence of homomorphic parameter selection on FHE kernels, which belongs to the software level of our DSE. The parameters for us to choose are N and q_i of the formula $Q_l = \sum_{i=1}^{l} q_i$, which mainly impact the security level and the accuracy of results after decrypted. Based on [2], [16], [18], we can learn that after setting the safety parameter λ, the range of values corresponding to N and the bit-width of Q_l can be obtained. For example, when $\lambda = 128$, if $N = 2048$ or 4096, bitwidth of Q_l can not exceed 54 or 109 respectively. We denote this relationship as $\lambda \xrightarrow{G} (N, Q_l)$, then there is

$$f_S(N, Q) = G^{-1}(N, Q_l) \tag{9}$$

Then we model the accuracy for FHE kernels which called approximate error exactly. A major part of error in CKKS comes from the difference between q and q_i, where q is a scale factor multiplied on the data in encryption step. Then the value of scale is accumulated with the multiplication depth grows. For example, there are two plaintexts denote by $q \cdot \mu_1$ and $q \cdot \mu_2$, the homomorphic multiplication result is $q^2 \cdot \mu_1 \cdot \mu_2$ and its scale is q^2. Rescale is performed after each multiplication to stop the growth of scale, which has the effect of reducing the result by a factor of q_i. Then the result of multiplication and rescale denoted as $r_R = \frac{q^2}{q_i} \cdot \mu_1 \cdot \mu_2$, and exact result without error should be $r_C = q \cdot \mu_1 \cdot \mu_2$. According to [19] and [16], there is $q/q_i \in (1-2^{-\epsilon_i}, 1+2^{-\epsilon_i})$. The error after one multiplication is represented in (10):

$$\begin{aligned} E = \frac{|r_R - r_C|}{r_C} &= \frac{|\frac{q^2}{q_i} \cdot \mu 1 \cdot \mu 2 - q \cdot \mu 1 \cdot \mu 2|}{q \cdot \mu 1 \cdot \mu 2} \\ &\leq \frac{2^{-\epsilon_i} \cdot q \cdot \mu 1 \cdot \mu 2}{q \cdot \mu 1 \cdot \mu 2} = 2^{-\epsilon_i}. \end{aligned} \tag{10}$$

The approximation error given in the above equation is an upper bound. The error can be deduced from the error of one multiplication when the multiplication depth is d, that is

$$f_E(Q) = f_E(\epsilon_1, \ldots, \epsilon_l) = \sum_{k=1}^{d} 2^{-\epsilon_{l-k-1}+(d-k)}. \tag{11}$$

Fig. 4(b) shows the relationship between the approximate error ratio and the bit-width of q_i for different N when multiplication depth is 1. Due to the impact of security on bit-width of Q_l, not all the bit-width can be set for each N. For example, when $N = 2048$, it can only select the number of bits between 14 and 18. For the same bit-width of q_i, the smaller N means the smaller error in general. However, if high accuracy and security are required, a larger N could be set to achieve it. It seems that when the bit-width is greater than 27, its error will be close to zero. Equation (11) shows the error will increase greatly when the multiplication depth grows.

VI. Experiments

A. Performance comparison with HEAX

We first evaluate the performance of the bottleneck KeySwitch operation compared with the state-of-the-art FPGA CKKS accelerator HEAX [14]. The pipelined KeySwitch design of HEAX targets to maximize it throughput. Our proposed design archives the same throughput with equal absolute number of pipeline stages. On the other hand, our design reduces the latency of KeySwitch, which benefits the overall performance when the pipeline is not fully filled in common cases.

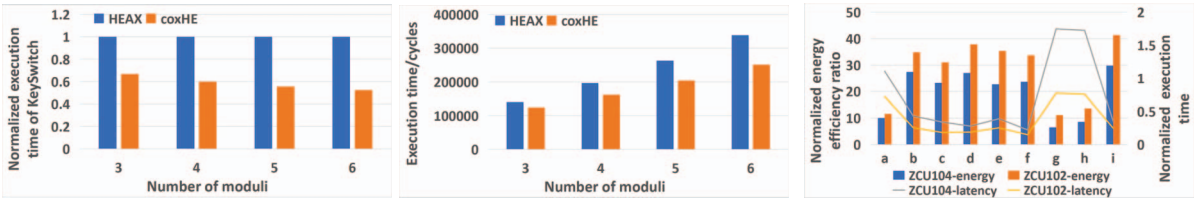
As shown in Fig. 5(a), for different number of moduli, we reduce the KeySwitch operation latency by 33%-48% compared with HEAX. While HEAX only reports the performance of individual operations, we extend the HEAX framework and compare the kernel-level performance under the same settings (i.e., $Pintra_{NTT/INTT} = 16$, $Pintra_{Mult} = 8$). As shown in Fig. 5(b), the proposed framework achieve 11.4%-25.77% overall performance improvement with the same simple deployment (i.e., without design space exploration).

TABLE II
The settings of benchmarks.

	Benchmark	Operation	Size
a	DotPlainBatchAxis_CKKS	Plain-Cipher	(100x10)x(10x1)
b	DotCipherBatchAxis_CKKS	Cipher-Cipher	(100x10)x(10x1)
c	MatMultVal_CKKS	Plain-Cipher	(1x2048)x(2048x1)
d	MatMultVal_CKKS	Plain-Cipher	(1x10)x(10x1)
e	MatMultVal_CKKS	Cipher-Cipher	(1x2048)x(2048x1)
f	MatMultVal_CKKS	Cipher-Cipher	(1x10)x(10x1)
g	MatMultVal_CKKS	Plain-Cipher	(100x10)x(10x1)
h	MatMultVal_CKKS	Cipher-Cipher	(100x10)x(10x1)
i	Average_pool	Cipher-Cipher	(20x20)x(4x4)

B. Performance comparison with HEXL

The Intel Homomorphic Encryption Acceleration Library (HEXL) supports Microsoft SEAL CKKS library with Intel AVX-512 instructions. HEXL achieves 1.23X-5.46X speedup for various CKKS operations compared with SEAL for single-threaded execution, and the performance gain is scalable with number of CPU cores for multi-threaded execution. In our evaluation, HEXL is executed with 12 threads on Intel Core i7-8700@3.70GHz (6-core, 12-thread, 95W TDP, 50-90W measured runtime power). The proposed coxHE framework generates CKKS accelerator on Xilinx Zynq UltraScale+ MPSoC ZCU102 (600K logic cells, 32.1Mbit BRAM, 2520 DSP slices, 6-8W measured runtime power) and ZCU104 (504K logic cells, 38Mbit BRAM+URAM, 1728 DSP slices, 5-7W measured runtime power). We use the SEAL Sample Kernel benchmark plus an "average pooling" kernel to evaluate the performance and energy efficiency of HEXL and coxHE (Table II). Results shown in Fig. 5(c) indicates that coxHE achieves reasonable



(a) KeySwitch performance. (b) C(1x10)*P(10x1) execution time. (c) Normalized energy efficiency and performance.

Fig. 5. Performance comparison of the proposed framework with HEAX and HEXL.

TABLE III
CONFIGURATION AND PERFORMANCE OF OPTIMAL CHOICES SAMPLE USING OUR DSE METHOD.

	Homomorphic parameters		hardware configuration				%Utilization					Performance			
	N	qi	board	Pintra	portnum	packnum	DSP	FF	LUT	BRAM	URAM	error	security	Latency(ms)	Power(W)
A1	2048	18	ZCU104	8	1	2	79	14	43	100	17	4.68×10^{-2}	128	7.5008	6.035
A2			ZCU102	16	2	2	64	11	31	47				5.63	5.857
B1	4096	25	ZCU104	8	1	2	51	12	32	100	17	4.88×10^{-4}	192	15.1632	5.355
B2			ZCU102	16	2	2	69	17	49	66				9.3292	7.498
C1		33	ZCU104	8	1	2	40	8	21	61	17	9.65×10^{-6}	128	15.127	4.684
C2			ZCU102	16	2	2	95	23	37	83				8.17	6.451
D1	8192	25	ZCU104	8	1	2	50	13	30	88	92	4.88×10^{-4}	256	33.9432	5.751
D2			ZCU102	16	2	2	74	18	54	80				17.5857	7.909
E1		33	ZCU104	8	1	2	78	16	40	88	92	9.65×10^{-6}	256	34.5601	6.444
E2			ZCU102	16	2	2	96	21	56	70				20.7656	7.773
F1		43	ZCU104	4	1	1	41	11	28	69	58	1.86×10^{-8}	192	67.3475	5.368
F2			ZCU102	16	2	2	100	29	58	88				20.8008	7.854

performance speedup and substantial energy efficiency improvement on embedded FPGA devices (5-8W power) compared with HEXL on desktop CPUs (95W TDP).

C. Design space exploration

The proposed coxHE automatically generates the Pareto optimal solutions for a given software kernel and target FPGA device. We evaluate the effectiveness of coxHE design space exploration with the plain-cipher matrix multiplication kernel, where the Pareto optimal design choices (out of hundreds of possible solutions) are listed in Fig. III. For example, when $N = 8192$, there are three optimal configurations for ZCU104, i.e., D1, E1, and F1. The error of F1 is minimal while its latency is the longest one due to the bit-width of moduli is 43bit which leads to low intra-operation parallelism due to resource limitation. While both D1 and E1 have higher security level compared with F1, D1 is the most energy-efficiency choice among these 3 solutions if the approximation error is tolerable. On the other hand, E1 is a preferable choice for application with very high accuracy demand, and it is only 2% slower than D1 (but requires more hardware resources).

VII. CONCLUDING REMARKS

In this paper, we propose an FPGA acceleration framework for fully-fledged CKKS based FHE kernels with HLS design flow. The proposed framework automatically explores the huge design space including encryption parameters and hardware configurations, and produce Pareto optimal solutions that balance security, performance, and approximation error.

REFERENCES

[1] A. Acar, H. Aksu, A. S. Uluagac, and M. Conti, "A survey on homomorphic encryption schemes: Theory and implementation," *ACM Comput. Surv.*, vol. 51, no. 4, pp. 79:1–79:35, 2018.

[2] H. Chen, K. Laine, and R. Player, "Simple encrypted arithmetic library - SEAL v2.1," *IACR Cryptol. ePrint Arch.*, p. 224, 2017.

[3] S. Gorantala, R. Springer, S. Purser-Haskell, and et al., "A general purpose transpiler for fully homomorphic encryption," *IACR Cryptol. ePrint Arch.*, p. 811, 2021.

[4] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," in *Innovations in Theoretical Computer Science, USA*, S. Goldwasser, Ed. ACM, 2012, pp. 309–325.

[5] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," *IACR Cryptol. ePrint Arch.*, p. 144, 2012.

[6] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds," in *ASIACRYPT*, vol. 10031, 2016, pp. 3–33.

[7] J. H. Cheon, A. Kim, M. Kim, and Y. S. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *ASIACRYPT, Hong Kong, China*, vol. 10624. Springer, 2017, pp. 409–437.

[8] R. Gilad-Bachrach, N. Dowlin, K. Laine, and et al., "Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy," in *ICML, USA*, vol. 48. JMLR.org, 2016, pp. 201–210.

[9] W. Dai and B. Sunar, "cuHE: A homomorphic encryption accelerator library," in *Second International Conference on Cryptography and Information Security in the Balkans*, vol. 9540. Springer, 2015, pp. 169–186.

[10] F. Boemer, S. Kim, G. Seifu, F. D. M. de Souza, and V. Gopal, "Intel HEXL: accelerating homomorphic encryption with intel AVX512-IFMA52," *CoRR*, vol. abs/2103.16400, 2021.

[11] Y. Doröz, E. Öztürk, and B. Sunar, "A million-bit multiplier architecture for fully homomorphic encryption," *Microprocess. Microsystems*, vol. 38, no. 8, pp. 766–775, 2014.

[12] D. B. Cousins, K. Rohloff, and D. Sumorok, "Designing an fpga-accelerated homomorphic encryption co-processor," *IEEE Trans. Emerg. Top. Comput.*, vol. 5, no. 2, pp. 193–206, 2017.

[13] A. Mkhinini, P. Maistri, R. Leveugle, and R. Tourki, "HLS design of a hardware accelerator for homomorphic encryption," in *DDECS*, 2017.

[14] M. S. Riazi, K. Laine, B. Pelton, and W. Dai, "HEAX: an architecture for computing on encrypted data," in *ASPLOS, Switzerland*, J. R. Larus, L. Ceze, and K. Strauss, Eds. ACM, 2020, pp. 1295–1309.

[15] G. Xin, Y. Zhao, and J. Han, "A multi-layer parallel hardware architecture for homomorphic computation in machine learning," in *ISCAS*, 2021.

[16] J. H. Cheon, K. Han, A. Kim, and et al., "A full RNS variant of approximate homomorphic encryption," in *SAC*, 2018.

[17] J. Zhao, L. Feng, S. Sinha, W. Zhang, Y. Liang, and B. He, "COMBA: A comprehensive model-based analysis framework for high level synthesis of real applications," in *ICCAD*, 2017.

[18] M. R. Albrecht, "On dual lattice attacks against small-secret LWE and parameter choices in helib and SEAL," in *EUROCRYPT*, 2017.

[19] A. Kim, A. Papadimitriou, and Y. Polyakov, "Approximate homomorphic encryption with reduced approximation error," *IACR Cryptol. ePrint Arch.*, p. 1118, 2020.