Accelerator Design and Performance Modeling for Homomorphic Encrypted CNN Inference

Tian Ye
Department of Computer Science
University of Southern California
Los Angeles, USA
tye69227@usc.edu

Rajgopal Kannan
US Army Research Lab
Los Angeles, USA
rajgopal.kannan.civ@mail.mil

Viktor K. Prasanna
Department of Electrical Engineering
University of Southern California
Los Angeles, USA
prasanna@usc.edu

***Abstract*—The rapid advent of cloud computing has brought with it concerns on data security and privacy. Fully Homomorphic Encryption (FHE) is a technique for enabling data security that allows arbitrary computations to be performed directly on encrypted data. In particular, FHE can be used with convolutional neural networks (CNN) to perform inference as a service on homomorphic encrypted input data. However, the high computational demands of FHE inference require a careful understanding of the tradeoffs between various parameters such as security level, hardware resources and performance. In this paper, we propose a parameterized accelerator for homomorphic encrypted CNN inference. We first develop parallel algorithms to implement CNN operations via FHE primitives. We then develop a parameterized model to evaluate the performance of our CNN design. The model accepts inputs in terms of available hardware resources and security parameters and outputs performance estimates. As an illustration, for a typical image classification task on CIFAR-10 dataset with a seven-layer CNN model, we show that a batch of 4K encrypted images can be classified within 1 second on a device operating at 2 GHz clock rate with 16K MACs, 64 MB on-chip memory and 256 GB/s external memory bandwidth.**

***Index Terms*—Homomorphic Encryption, Neural Network, Cloud Computing, ASIC and FPGA**

I. Introduction

Cloud computing provides users with various kinds of services that are easy to deploy and scale without too much effort. For example, in order to perform neural network inference, users can send their input data to a cloud server that holds a pretrained neural network and provides inference as a service, without building and training the model themselves. However, data security becomes a critical concern when uploaded data or results returned from the cloud are sensitive and confidential. Although the data can be encrypted while being transferred between client and server by underlying network protocols, they are still vulnerable because they must be decrypted on the cloud before computations are performed. Even if users trust the cloud provider not to misuse their data intentionally, it is still possible that the cloud provider may fail to defend the server from being hacked.

Fully Homomorphic Encryption (FHE) [1] offers a good solution to this problem. FHE allows computations to be directly performed on encrypted data without revealing the original plaintext. In a cloud computing scenario, users encode and encrypt their data locally before sending the ciphertext to the cloud server. The cloud provider performs computations on ciphertext where all the intermediate and final results are encrypted. The user then decrypts the results received from the cloud server. As long as the user keeps her secret keys private, it is impossible for others to understand the messages in the entire process even if the data are leaked or the server is hacked. Cloud computing with FHE does not require users to trust the cloud provider. The only assumption is that the cloud provider always performs computations correctly.

FHE is computationally demanding and requires a lot of hardware resources because high degree polynomials are used to achieve desired security level. It is necessary to accelerate such services with ASIC or FPGA. Recently, in [2], implementations on FPGAs for basic homomorphic encrypted operations that exploit the parallelism of those operations have been proposed. However, they do not design an overall architecture for a specific application such as CNN inference.

In this paper, we propose an accelerator for CNN inference based on FHE. We first propose a design for basic primitives of the RNS variant of CKKS [3], a homomorphic encryption scheme. We train the CNN model [4] where the inference is compatible with FHE operations and propose a parameterized design for inference on a seven-layer CNN model for an image classification task on CIFAR-10 dataset. We also show a model of hardware resource requirements that can be used to estimate the performance of our design. For parameters in terms of security level and hardware (e.g., clock rate), our model can be used to evaluate the trade-offs and map our design to a target device.

The contributions of this work are:

- We design hardware modules for basic primitives of the RNS variant of CKKS [3], an approximate homomorphic encryption scheme. Supported primitives are homomorphic addition, scaling and multiplication that includes number-theoretic transform (NTT), relinearization and rescaling operation. Each module contains configurable number of small circuit units to process high-degree polynomials in parallel.
- We propose an accelerator design for a seven-layer CNN model for an image classification task on CIFAR-10 dataset [5] using the basic primitives of CKKS. For map-

ping various CNN layers, the parallelism and hardware optimization are explored to address the challenges of the huge encrypted data size and limited on-chip storage.

- We propose a model to predict the performance and estimate the requirements on hardware resources, e.g., required number of MACs, on-chip storage and external memory bandwidth, in terms of the parameters of the design, e.g., dimensions of the convolutional layers, clock rate of target device and number of small units in each module for the basic primitives.
- We provide an analysis of our design for a sample image classification task. We show that a batch of 4096 encrypted images from CIFAR-10 can be classified within 1 second on an accelerator operating at 2 GHz with 16K MACs, 64 MB on-chip memory and 256 GB/s external memory bandwidth.

II. Background

A. Homomorphic Encryption

Homomorphic encryption provides a practical way for privacy-preserving computations, which was first proposed by [1] using lattice-based cryptography. More homomorphic encryption schemes have been proposed recently, including BGV [6], LTV [7], BFV [8] [9], BLLN [10] and CKKS [11]. A scheme is called fully homomorphic encryption (FHE) if arbitrary circuits of addition and multiplication can be constructed with the scheme. However, homomorphic encryption schemes introduce a small noise term to the ciphertext, which grows quickly as computations, especially multiplications, are performed. The original message will be corrupted when the noise is larger than a limit. This leads to a limitation on the maximum depth of circuits that can be constructed depending on configurated parameters of a homomorphic encryption scheme. In order to support arbitrary levels of circuits, whenever the noise is too large, a bootstrapping operation must be called to decrypt and then encrypt the ciphertext in a homomorphic way, which resets the noise into a small one. Due to the high complexity of bootstrapping operations, practical implementations either try to reduce its frequency [12], or simply not implement it when the depth of computation for a specific application is known so that parameters can be configured to ensure the noise not to exceed the limit.

In this work, we select the CKKS scheme, which is an approximate homomorphic encryption scheme that perfectly represents real number arithmetic on integer polynomial rings. Before starting encryption, users need to encode multiple original values into a polynomial as plaintext such that computations are performed in a SIMD style.

We choose the RNS variant of CKKS [3]. The main components of the scheme are defined as follows. Denote $R = \mathbb{Z}[X]/(X^N + 1)$ where N is a power of 2. Then R is a ring for polynomials with maximum degree $N - 1$. Denote $R_q = R/qR$ as a ring where coefficients of all polynomials are integers modulo q. The scheme defines L moduli $q_0, q_1, ..., q_{L-1}$. Each of them has approximately the same size. A ciphertext ct is defined as L pairs of polynomials, i.e., $ct = (ct^{(0)}, ct^{(1)}, ..., ct^{(L-1)})$, where $ct^{(i)} = (c_0^{(i)}, c_1^{(i)}) \in R_{q_i}^2$ for $i = 0, 1, ..., L - 1$.

TABLE I
CNN Model for Classification on CIFAR-10

Layer	Description
Input	3 channel images of size 32×32
Conv1	32 kernels of size 3×3, stride 1
Activation	square function
Pool1	average pooling, size 2×2, stride 2
Conv2	64 kernels of size 3×3, stride 1
Activation	square function
Pool2	average pooling, size 2×2, stride 2
Conv3	128 kernel of size 3×3, stride 1
Activation	square function
Pool3	average pooling, size 2×2, stride 2
FC	2048 inputs, 10 outputs

CKKS scheme supports additions and multiplications. Addition of two ciphertexts is simply coefficient-wise addition of all corresponding pairs of polynomials. Multiplication between ciphertexts is much more expensive because multiplication between a pair of polynomials generates three polynomials. It requires a relinearization operation to reduce the three polynomials into two that contain the same message. A rescaling operation is also necessary to divide the generated coefficients by a constant so that the values will not grow explosively. To reduce the complexity of multiplication of two $(N-1)$-degree polynomials, we first apply number-theoretic transform (NTT) to the polynomials, which reduces the complexity from $O(N^2)$ into $O(N \log N)$. Due to the limited space of this paper, we refer the readers to the original paper [3] for more details about the multiplication algorithm as well as encryption and decryption.

In Section III, we will describe hardware design of a homomorphic addition module and multiplication module. Note that there is a sub-module for NTT as well as inverse NTT (INTT) inside the multiplication module. We will also describe a homomorphic scaling module which performs multiplication between a scalar integer and a ciphertext.

B. CNN Model and Encoding Scheme

To illustrate the accelerator design, we use the modules for basic homomorphic encrypted operations in an image classification task on the CIFAR-10 dataset.

As the homomorphic encrypted scheme only supports addition and multiplication, some components of CNN are not directly compatible, e.g., batch normalization, softmax, and non-polynomial activation functions. Some previous works [4] proposed adapted CNN that is compatible with homomorphic operations. Specifically, the activation function is replaced by the square function $f(z) = z^2$ and only average pooling is used in pooling layers. Besides, no bias are included in convolutional layers and fully connected layers. We use their CNN model to classify the CIFAR-10 dataset. The architecture of the CNN model is shown in Table I.

We assume that the adapted CNN model is trained with CIFAR-10. All weights in the model are scalar values and not

encrypted. Before doing inference, a batch of N input images of size 32×32 are encoded into plaintext polynomials using the method proposed by [13]. Pixels with the same coordinate from all the N images are assembled into a vector. There are in total 32×32 vectors of length N. Each vector is mapped into a plaintext polynomial of degree $N-1$ using Chinese Remainder Theorem. Then, all the plaintext polynomials are encrypted into ciphertexts.

C. Parameter Selection

In the homomorphic encryption scheme, the values of N and $Q = \prod_{i=1}^{L} q_i$ are related to the level of security. Typically, $N = 2^{12}$, 2^{13} or 2^{14}. When N is less than 2^{12}, the encryption scheme does not have sufficient security level. When N is greater than 2^{14}, the polynomials are huge and thus the complexity of computation is prohibitively high. In our analysis, we select $N = 2^{12}$. We further let Q be 300 bits to maintain an acceptable security level according to the standard for homomorphic encryption [14]. Let each q_i have 30 bits so that we can use 32-bit MACs to handle basic computations easily on hardware platforms. From the selected values of Q and q_i, we can derive $L = 10$.

III. Designs for Basic Primitives

In this section, we describe designs for basic primitives of homomorphic operations, including addition, scaling and multiplication. In our design, all coefficients of polynomials have a word size of 32 bits. We assume the computing units (MACs) of the target device are 32 bits where a single MAC can perform 32-bit multiplication in one clock cycle.

A. Homomorphic Addition and Scaling

The module for homomorphic addition primitive includes n_A addition units, each of which takes two coefficients as input, and outputs their sum modulo p_i. Here p_i is one of the L moduli. As the sum of two coefficients is less than $2p_i$, the modulo operation requires at most one subtraction. In the addition module, we allocate two registers to contain n_A coefficients from two polynomials respectively. In each cycle, $2n_A$ coefficients are loaded from on-chip memory into the two registers. The coefficients are then fed into the n_A addition units as input. In order to fully utilize all addition units, we enable double buffering to the registers so that data can be loaded and consumed simultaneously. Outputs of the addition units are saved into another pair of registers and then sent to on-chip memory.

The design for homomorphic scaling primitive is similar to the one for addition operations, except that n_{Sc} scaling units replace n_A addition units. Note that a modulo operation is required in the scaling unit, which is quite efficient to implement on hardware using Barrett's algorithm [15] when the modulus is a known constant integer.

B. Homomoprhic Multiplication

Homomorphic multiplication is a long procedure with multiple steps, and we refer the readers to CKKS paper for details about the theoretical procedure. In this section, we focus on the design of a hardware module for homomorphic multiplication. Since the CNN inference contains multiplications only for the square function in the activation layer, we can just focus on the special case where two inputs of a multiplication operation are identical.

As is mentioned in Section II-A, modules for NTT and INTT are critical phases for homomorphic multiplication that reduce its complexity. An NTT module for a $(N-1)$-degree polynomial includes $\log N$ stages where each stage updates N coefficients. Let n_N denote the number of NTT units in the module, each of which updates two coefficients at a time. The module contains a butterfly network to feed inputs into correct NTT units. Three groups of parallel on-chip memory are required in the NTT module for the coefficients of the input polynomial, the pre-computed twiddle factors, and the coefficients of the transformed polynomial as output. The inverse NTT (INTT) module has a similar design as the NTT module, except that INTT units replace NTT units and the butterfly network is reversed. Note that a previous work [2] proposed the architecture for the NTT module on two specific types of FPGAs. However, our version is parameterized for general devices.

A summary of the integrated pipeline for homomorphic multiplication is shown in Table II, which contains stages including NTT, INTT, homomorphic addition and scaling. Besides, there are stages for coefficient-wise multiplication, denoted as "mult", that is similar to homomorphic scaling except that both inputs are coefficients. We assume that all polynomials have been transformed into NTT form before being loaded to the accelerator so that there are fewer occurrence of NTT/INTT transformations.

TABLE II
Pipeline Stages for Multiplication

Stage	# of units ($N = 2^{12}$)	# of polynomials to output
Mult1	$n_{Mul1} = 3n_{Act}$	3
INTT1	$n_{I1} = 6n_{Act}$	1
Scale1	$n_{Sc1} = n_{Act}$	1
Add1	$n_{A1} = n_{Act}$	1
Mod1	$n_{Mod1} = n_{Act}$	1
NTT1	$n_{N1} = 6n_{Act}$	1
Mult2	$n_{Mul2} = 4n_{Act}$	4
INTT2	$n_{I2} = 24n_{Act}$	4
Scale2	$n_{Sc2} = 2n_{Act}$	2
Add2	$n_{A2} = 2n_{Act}$	2
Mod2	$n_{Mod2} = 2n_{Act}$	2
SubMult1	$n_{Sm1} = 2n_{Act}$	2
NTT2	$n_{N2} = 12n_{Act}$	2
Add3	$n_{A3} = 2n_{Act}$	2
Add4	$n_{A4} = 2n_{Act}$	2
Scale3	$n_{Sc3} = 2n_{Act}$	2

Each stage of the pipeline has multiple computation units and the number of units is shown in the second column in Table II. In order to fully utilize all modules as well as balance the workload of every stage, we need to compute the relationship between the numbers of units for each pair of adjacent stages. For example, let n_{Mul1} be the number of units of *Mult1* stage which is coefficient-wise multiplication, and n_{I1}

be the number of units of *INTT1* stage. According to the algorithm, whenever *Mult1* generates three polynomials, *INTT1* needs to consume one polynomial. Then ideally we require $3N/n_{Mul1} = N \log N/2n_{I1}$, i.e., $n_{I1} = \log N \cdot n_{Mul1}/6$. The configurations for all stages are tuned in the similar way to balance the throughput. For $N = 2^{12}$, we can determine the numbers of units for all stages as shown in the second column of Table II. Here n_{Act} is a configurable parameter. The multiplication module can be scaled by increasing the value of n_{Act}.

Double buffered on-chip memory are allocated after each stage in the pipeline as the output of the current stage and the input of the following stages. For example, there is on-chip memory storing two polynomials at the output side of *INTT1* stage. During every $N \log N/2n_{I1}$ cycles, the INTT stage outputs one polynomial and stores it into the on-chip memory, which is then read by the following stage starting from the next cycle.

Now we have described modules for implementing basic primitives with homomorphically encrypted data. Using these modules, one can execute arbitrary applications based on the CKKS scheme. In the next section, we describe an integrated design for CNN inference using these modules.

IV. Mapping of CNN Layers

The huge size of the ciphertext poses a critical challenge in implementing homomorphic encrypted CNNs, for example, $\approx$ 0.29 MB, for a single ciphertext in a typical scenario ($N = 2^{12}$, $L = 10$ and $\log p_i = 30$). Thus, given limited on-chip storage capacity, the majority of ciphertexts, including the initial inputs, final outputs and the results of intermediate layers must be stored in external memory. This makes external communication bandwidth a limiting factor. Only active ciphertexts that are the inputs or outputs of currently ongoing computations can be temporarily saved on chip. In comparison, since weights are scalar values (not polynomials), the size of the CNN model is not large and we can keep the weights of the entire CNN model on-chip. Existing techniques to optimize CNN inference do not work due to limited on-chip ciphertext, thus our approach is to reduce data traffic and exploit parallelism during CNN inference.

It is a frequent workflow in the CNN model to apply a convolutional layer, followed by an activation layer and a pooling layer. Therefore, we design a pipeline to process three layers on the target device. The pipeline can be reused in a CNN inference task. For example, we can reuse the pipeline three times during an inference on the model for CIFAR-10. Besides, we also need a shorter pipeline for fully connected layer to generate the final output.

A. Convolutional Layer

We consider a convolutional layer with input dimensions (f_{in}, n, n), filter dimensions (f_{in}, f_{out}, f, f) and output dimensions (f_{out}, n', n'). Here f_{in} (resp. f_{out}) is the number channels for the input (resp. output). The size of the input image (resp. output) is $n \times n$ (resp. $n' \times n'$). The size of the filter kernel is denoted as $f \times f$.

Before being sent to computation modules for the convolutional layer, input images or intermediate feature maps are stored in external DRAMs. Due to the restricted resources of on-chip memory and prohibitively large size of ciphertext, only very limited number of pixels can be held on chip. It is impossible to hold all necessary inputs at a time in order to compute even a single pixel of all the output channels, which requires $(f^2 f_{in} + f_{out})$ pixels to be on chip. It turns out that the same input pixels must be fetched from external memory to on-chip memory multiple times. Therefore, a proper fetching policy is important to the performance as well as the traffic between the device and external memory. Here we propose two fetching policies:

1) Fetch a $f \times f$ block from one input channel at a time and use it to compute k partial output pixels. With this policy, the convolutional layer requires on-chip memory to store $(f^2 + k)$ pixels. The total data traffic between external memory and on-chip memory during the entire convolution is $f^2 f_{in} f_{out} n'^2/k$ pixels.
2) Fetch a $(f+s) \times (f+s)$ block from one input channel at a time and use it to compute $4k$ partial pixels. Although more on-chip memory is required to store $(f+s)^2 + 4k$ pixels, the total data traffic during the entire convolution is reduced to $(f+s)^2 f_{in} f_{out} n'^2/4k$ pixels.

As is shown above, the two fetching policies lead to a trade-off between on-chip memory resources and external traffic. However, an advantage of the second policy is that the outputs of convolution are 2×2 pixels, which is exactly the window size of the pooling layers in our target CNN model. Thus, the outputs of the convolution layer can be directly fed into the pooling layer without being stored into extra on-chip memory. Of course, there could be more policies, e.g., fetching larger blocks at a time, but the required on-chip memory resources would be prohibitively huge.

After data are fetched from external memory to on-chip memory, we use a scaling module and an addition module to perform convolution. To form an efficient pipeline for the convolutional layer, all scaling and addition units are further divided into fine-grained pipelines so that they generate output every clock cycle. Denote n_{Conv} as the number of units in the scaling module. To balance the throughput, the number of units in the addition module is also n_{Conv}. If the second fetching policy is taken, in order to generate k-many 2×2 pixels as output of convolution, the scaling and addition module need to generate $4 f_{in} f^2 k$ pixels. Denote the clock frequency of the device as F. Assume double buffering is used so that data are fetched from external memory continuously and computation modules are fully utilized. Then the duration of this process is $T_{Conv} = 8 f_{in} f^2 k N L / n_{Conv}$ cycles and the lower bound of the bandwidth from external memory is $n_{Conv}(f+s)^2 F \log p/4kf^2$, where $\log p$ denotes the number of bits for every coefficient.

At the output side of the convolutional layer, on-chip memory is used to store $4k$ pixels as the output of convolution

as well as the input of the following activation layer.

B. Activation Layer

In our CNN model, the squaring function $f(z) = z^2$ is used as the activation function. Then we can simply use the design for homomorphic multiplication as the activation layer.

To better utilize the computing resources, the number of units in the convolutional layer and the activation layer should be tuned. Whenever the convolutional layer generates one pixel as output, it processes $f_{in}f^2$ input pixels. All filters in our seven-layer CNN are of size $f = 3$. However, the three convolutional layers have quite different f_{in}, which are 3, 32 and 64, respectively. When allocating computing resources, we should focus on the case for $f_{in} = 64$, where the convolutional layer is the bottleneck, to ensure a good performance. It takes $T_{Act} = 4kLN/n_{Act}$ cycles for the activation layer to finish processing $4k$ pixels. Then we require $T_{Conv} \geq T_{Act}$, i.e., $n_{Act} \geq n_{Conv}/2f_{in}f^2$. The reason why n_{Act} is greater than $n_{Conv}/2f_{in}f^2$ is that the activation layer becomes the bottleneck when f_{in} is 3 or 32. By doing so, we can better utilize available resources.

C. Pooling Layer

As the homomorphic encryption is not order-preserving, it is infeasible to use pooling layers based on comparison approaches, e.g., max pooling. A practical option is average pooling. In our target CNN, the pooling filter is 2×2 with stride 2, i.e., $y = (z_1 + z_2 + z_3 + z_4)/4$, where y is a pixel in the output of the pooling layer and z_1, z_2, z_3 and z_4 are inputs within the same pooling filter. We can simplify the pooling operation as $y = z_1 + z_2 + z_3 + z_4$, since the outputs are scaled by the same factor and the classification results are not affected. As is mentioned in Section IV-A, when the second fetching policy is selected in the convolutional layer, the output of convolution and activation is several 2×2 pixels. Then the pooling layer simply accumulates every four input pixels without any other dependency.

Denote n_{Pool} as the number of units in the addition module for the pooling layer. It takes $T_{Pool} = 8kLN/n_{Pool}$ to finish additions on $4k$ pixels. Let $T_{Act} = T_{Pool}$ to make it fully utilized. Then we have $n_{Pool} = 2n_{Act}$. The output of the pooling layer is exported to external memory and then waits for the next iteration of convolution, activation and pooling. In T_{Pool} cycles, k pixels should be exported. Thus, the output bandwidth is at least $n_{Pool}F \log p/4$.

D. Fully Connected Layer

In our design, fully connected layer is a standalone workflow from the other CNN layers at the end of inference. The layer includes a scaling stage and an addition stage with the same number of scaling units and addition units. Let n_{FC} denote the number of units in each stage. In the CNN for CIFAR-10, the fully connected layer reads 2048 pixels as input and the total amount of scaling and addition is 20480. Therefore its latency is $20480LN/n_{FC}$ cycles, and the input bandwidth is $n_{FC}F \log p$.

V. Model for Resources and Performance Estimation

Previously, we described a modular pipelined design of a typical CNN workflow using basic homomorphic primitives along with a brief analysis of the hardware requirements and performance of each module. In this section, we provide an overall model to estimate the resource requirements for the entire design and predict the performance of CIFAR-10 inference with a deep CNN model in terms of the variables shown in Table III. Note that the proposed parallel computations have been carefully organized and it can be mapped to the accelerator so the memory accesses do not lead to pipeline stalls. This leads to fully pipelined implementations where the memory access latencies do not lead to any performance degradation.

A. Performance Prediction

Assume that the pretrained seven-layer CNN model is stored in the cloud. The CNN model is public where the weights are scalar integer values that are not encrypted. In contrast, the input is private to the user and should be encrypted. The input is a piece of $32 \times 32 \times 3$ ciphertexts encoded from a batch of $N = 2^{12}$ images from CIFAR-10 using the method described in Section II. As is shown in Table I, an inference with the seven-layer network includes three iterations of convolution, activation and pooling, and one iteration of fully connected layer. We predict the performance for each iteration separately.

For the iterations of convolution, activation and pooling, the bottleneck is either the activation layer or the convolutional layer. When the bottleneck is the activation layer, as is mentioned in Section IV, it takes LN/n_{Act} cycles for the activation layer to output one pixel. Then it takes $n'^{(j)2}f_{out}^{(j)}LN/n_{Act}$ cycles for an activation layer to generate $n'^{(j)2}f_{out}^{(j)}$ pixels. Here the superscript j means that the parameter is for the j-th iteration where $j = 1, 2$ or 3. Considering the large number of pixels to be processed, we can ignore the time it takes for the first pixel to go through the three layers. Therefore, it takes approximately $T_1(j) = n'^{(j)2}f_{out}^{(j)}LN/n_{Act}$ cycles to complete one iteration. When the bottleneck is the convolutional layer which performs addition on $n'^{(j)2}f_{in}^{(j)}f_{out}^{(j)}f^{(j)2}$ pixels in this iteration, it takes approximately $T_2(j) = 2n'^{(j)2}f_{in}^{(j)}f_{out}^{(j)}f^{(j)2}LN/n_{Conv}$ cycles to complete the iteration, ignoring the small initial latency to complete processing the first pixel. Therefore, the latency for the j-th iteration is $T(j) = \max\{T_1(j), T_2(j)\}$

As for the fully connected layer, which takes $m = 2048$ pixels as input and generates 10 pixels for 10 classes as output, its total latency is $T_{FC} = 20mLN/n_{FC}$ cycles.

In conclusion, for an inference with the seven-layer CNN, the total latency is $T = T(1) + T(2) + T(3) + T_{FC}$ cycles.

B. Resource Estimation

Based on our design, we further estimate the requirements of hardware resources for the seven-layer CNN inference in terms of external memory bandwidth, computing units (MACs)

TABLE III
SUMMARY OF VARIABLES

Class	Variable	Definition
CKKS	L	# of moduli
	N	# of coefficients in a polynomial
	$\log p$	Size of a coefficient
CNN	$f_{in}^{(j)}, f_{out}^{(j)}$	# of input/output channels for the j-th convolutional layer
	$n^{(j)}, n'^{(j)}$	Size of input/output for the j-th convolutional layer
	$f^{(j)}, s^{(j)}$	Size of filter/stride for the j-th convolutional layer
Design	k	# of channels to compute for a single fetching in conv. layers
	$n_{Conv}, n_{Act}, n_{Pool}, n_{FC}$	Factors of # of units for CNN layers

TABLE IV
SIZE OF DATA FOR VARIOUS PARAMETERS

N	$\log p$	L	Single Ciphertext Size	Input Size
2^{12}	30	10	150 KB	450 MB
2^{13}	30	10	300 KB	900 MB
2^{14}	30	10	600 KB	1800 MB

and on-chip storage. Table IV illustrates the size of a single ciphertext and the total size of the input for various sets of parameters. It shows that a ciphertext is so large that we must save most of the input as well as the output of the intermediate layers into the external memory. Therefore, the on-chip memory and the external memory bandwidth are critical to the design for achieving high performance.

External memory bandwidth: For the convolutional layers, data is fetched from the external memory before convolution. Assume the second data fetching policy described in Section IV-A is selected. During the time when the convolutional layer performs $4f^2k$ additions, i.e., $8f^2kLN/n_{Conv}$ cycles, the device needs to fetch $(f+s)^2$ pixels, i.e., $2(f+s)^2LN\log p$ bits, from the external memory. Then the external memory traffic for fetching data should be at least $n_{Conv}(f+s)^2F\log p/4kf^2$. The outputs are exported whenever the pooling layer performs addition on 4 pixels, so the memory traffic for exporting data is at least $\frac{1}{4}n_{Pool}F\log p$. Then the total externl memory traffic for the iteration is $n_{Conv}(f+s)^2F\log p/4kf^2 + \frac{1}{4}n_{Pool}F\log p$. Similarly, the traffic for the fully connected layer is $n_{FC}F\log p$. Since the layers are computed sequentially, the actual external memory traffic is the larger of these.

On-chip memory: In our design, most of the on-chip memory is allocated to the convolutional layer. As is mentioned in Section IV-A, we need on-chip memory to store $(f+s)^2+4k$ pixels when the second data fetching policy is employed. Here f is the size of the filter, s is the stride, and k is the number of output channels computed at a time. Because the memory should be double buffered, the size of memory for this is $((f+s)^2+4k)\times 2LN\log p$ bits. For the seven-layer CNN model, $f=3$ and $s=1$, where k is a parameter that affects both the on-chip memory and the external memory bandwidth. In the pipeline of the activation and pooling layer, $64+8L$ polynomials need to be stored in the on-chip memory between the pipeline stages. As the fully connected layer requires very little on-chip storage, it can reuse the memory in the convolutional layer. Therefore, the total on-chip memory needed is at least $(4(f+s)^2L+16kL+8L+64)N\log p$ bits.

MACs: We assume that a MAC can perform 32-bit integer arithmetic, e.g., addition, subtraction and multiplication, in one cycle. Table V shows the number of MACs in each kind of unit. According to Table II which shows the number of units in every module, we derive that the activation layer requires $753n_{Act}$ MACs. Similarly, we need $12n_{Conv}$ MACs for the convolutional layer, $4n_{Pool}$ MACs for the pooling layer and $12n_{FC}$ MACs for the fully connected layer. Therefore, our design requires in total $753n_{Act}+12n_{Conv}+4n_{Pool}+12n_{FC}$ MACs.

TABLE V
REQUIRED MACS FOR A SINGLE CIRCUIT UNIT

Circuit Unit	Required MACs
Addition Unit	4
Scaling Unit	8
Multiplication Unit	8
NTT Unit	12
INTT Unit	12
Modulo Unit	7

As an illustration, we can now estimate the predicted performance and the hardware requirements for an inference on the seven-layer CNN for $N=2^{12}$. We assume the target device operates at $F=2$ GHz. A possible set of parameters are $n_{Act}=2, n_{Conv}=1152, n_{Pool}=4, n_{FC}=10$ and $k=16$. Then the design requires 15466 MACs. The required external bandwidth is 247.5 GB/s. The required on-chip memory is 49 MB. With this set of parameters, the latency of the inference on 4K encrypted images is 0.755 second.

VI. CONCLUSION

In this work, we designed a parameterized accelerator for homomorphic encrypted CNN inference. The accelerator is based on basic modules for homomorphic encrypted primitives. For an image classification task on CIFAR-10 dataset with a seven-layer CNN, we developed a model to predict the performance and estimate the hardware resource requirements.

Homomorphic encryption offers a solution for privacy-preserving computation, but is computationally demanding. As FPGAs and ASICs provide fine-grained parallelism, they are promising devices for accelerated homomorphic encrypted libraries to be deployed in the cloud. We believe our design of the accelerator as well as the model of performance and resources estimation will provide a guidance for implementing accelerated secure applications. In the future, we will further optimize the design and look into the opportunity of mapping existing optimization techniques for normal CNN inference, e.g., [16] [17], into the encrypted space.

ACKNOWLEDGEMENT

This work was supported by the U.S National Science Foundation (NSF) under grants OAC-1911229 and CCF-1919289.

REFERENCES

[1] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*, ser. STOC '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 169–178. [Online]. Available: https://doi.org/10.1145/1536414.1536440

[2] 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*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1295–1309. [Online]. Available: https://doi.org/10.1145/3373376.3378523

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

[4] A. A. Badawi, J. Chao, J. Lin, C. F. Mun, J. J. Sim, B. H. M. Tan, X. Nan, K. M. M. Aung, and V. R. Chandrasekhar, "The alexnet moment for homomorphic encryption: Hcnn, the first homomorphic cnn on encrypted data with gpus," 2018.

[5] A. Krizhevsky, "Learning multiple layers of features from tiny images," *University of Toronto*, 05 2012.

[6] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," in *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ser. ITCS '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 309–325. [Online]. Available: https://doi.org/10.1145/2090236.2090262

[7] A. López-Alt, E. Tromer, and V. Vaikuntanathan, "On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption," in *Proceedings of the Forty-Fourth Annual ACM Symposium on Theory of Computing*, ser. STOC '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 1219–1234. [Online]. Available: https://doi.org/10.1145/2213977.2214086

[8] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," Cryptology ePrint Archive, Report 2012/144, 2012, https://eprint.iacr.org/2012/144.

[9] Z. Brakerski, "Fully homomorphic encryption without modulus switching from classical gapsvp," Cryptology ePrint Archive, Report 2012/078, 2012, https://eprint.iacr.org/2012/078.

[10] J. W. Bos, K. Lauter, J. Loftus, and M. Naehrig, "Improved security for a ring-based fully homomorphic encryption scheme," Cryptology ePrint Archive, Report 2013/075, 2013, https://eprint.iacr.org/2013/075.

[11] 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.

[12] T. Rondeau, "Data protection in virtual environments (dprive)," Mar 2020.

[13] N. Dowlin, R. Gilad-Bachrach, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing, "Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy," in *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ser. ICML'16. JMLR.org, 2016, p. 201–210.

[14] M. Chase, H. Chen, J. Ding, S. Goldwasser, S. Gorbunov, J. Hoffstein, K. Lauter, S. Lokam, D. Moody, T. Morrison *et al.*, "Security of homomorphic encryption," *HomomorphicEncryption. org, Redmond WA, Tech. Rep*, 2017.

[15] P. Barrett, "Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor," in *Proceedings on Advances in Cryptology—CRYPTO '86*. Berlin, Heidelberg: Springer-Verlag, 1987, p. 311–323.

[16] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 161–170. [Online]. Available: https://doi.org/10.1145/2684746.2689060

[17] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16. IEEE Press, 2016, p. 367–379. [Online]. Available: https://doi.org/10.1109/ISCA.2016.40