Appears in FPL 2021

Performance Modeling and FPGA Acceleration of Homomorphic Encrypted Convolution

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

Sanmukh R. Kuppannagari
Department of Electrical Engineering
University of Southern California
Los Angeles, USA
kuppanna@usc.edu

Rajgopal Kannan
US Army Research Lab
Playa Vista, USA
rajgopal.kannan.civ@mail.mil

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

***Abstract*—Privacy of data is a critical concern when applying Machine Learning (ML) techniques to domains with sensitive data. Homomorphic Encryption (HE), by enabling computations on encrypted data, has emerged as a promising approach to perform inference on ML models such as Convolution Neural Network (CNN) in a privacy preserving manner. A significant portion of the total inference latency is in performing convolution over homomorphic encrypted data (HE-Convolution). For performing convolution over plaintext data, low latency accelerator designs have been proposed using algorithms such as im2col, frequency domain convolution, etc. However, developing accelerators for the HE versions of these algorithms is non-trivial. In this work, we develop a unified FPGA design that enables low latency execution of both im2col and frequency domain HE-Convolution. To enable selection of the efficient algorithm for each convolution layer of a CNN, we develop a performance model that takes the parameters about the encryption and convolution layer as input and outputs the computation and resource requirements of the two algorithms for that layer. We use the performance model to select convolution algorithm for each layer of ResNet-50 and obtain the first low latency batch-1 inference accelerator for CNN inference with HE-Convolution targeting FPGAs using HLS. We compare our design against prior techniques on CPUs and show that our accelerator achieves speedups in the range of 3.4×∼6.7× in latency.**

***Index Terms*—Homomorphic Encryption, Convolutional Neural Network, FPGA**

I. INTRODUCTION

Data security and privacy have gained importance in recent years. This has had great impact on domains which use cloud computing to perform machine learning analytics on large amount of data such healthcare, financial analytics, recommendation systems [1], [2]. Data security becomes a critical concern when uploaded data or results returned from the cloud are sensitive and confidential. Encryption of data provides protection for data in transit. However, as cloud platforms are public, shared resources, the application's data remain vulnerable if decoded on the cloud platform for execution [2].

Fully homomorphic encryption (FHE) [3] offers a promising solution to this challenge by enabling arbitrary computations on encrypted data. In the context of ML analytics on cloud, the encrypted user data are sent to the cloud server which performs computations over the ciphertext to produce the results. The results are sent back to the user for decryption. All the intermediate and final results are encrypted, thereby, guaranteeing end-to-end privacy. FHE has recently become popular in implementing ML applications such as Convolution Neural Networks (CNN) [2], [4], [5].

However, a major challenge in the efficient implementation of CNN inference over homomorphic encrypted data is the large computational complexity and storage requirements of executing convolution layers. This is because FHE converts the inputs into high degree polynomials, typically 1K to 32K [6]. Additionally, the computations can only be performed in a coefficient-wise manner and require expensive rotation operations to align the coefficients [2]. These challenges have prevented the porting of efficient convolution algorithms such as im2col and frequency domain convolution which have been successfully used for developing low latency CNN implementations targeting FPGAs [7]. Thus, while FPGA implementations of basic FHE operations exist [8], [9], there does not exist an FPGA accelerator for HE based CNNs.

In this work, we develop a unified FPGA design that enables low latency execution of both im2col and frequency domain HE-Convolution targeting FPGAs. For each convolutional layer, the layer parameters and the security parameters determine the complexity of the HE-Convolution algorithms. Thus, to determine the algorithm with the best performance for each layer, we develop a performance model that takes the layer parameters and the security parameters as input, and outputs the computation time and the FPGA resources consumed for each HE-Convolution algorithm. We leverage the model to select algorithms for each convolutional layer of ResNet-50 to obtain a low latency batch-1 inference accelerator. The specific contributions of this work are:

- We develop an efficient FPGA design for performing HE-Convolution. This design supports both im2col and frequency domain algorithms for different convolutional layers without reconfiguration.

- We develop a performance model that takes security parameters and convolutional layer dimensions as input, and outputs the computation time and the FPGA resources consumed for each HE-Convolution algorithm.
- We use the performance model to select convolution algorithm for each layer of ResNet-50 and obtain the first low latency batch-1 inference accelerator for a HE-CNN targeting FPGAs.
- We implement our design in HLS, compare it against state-of-the-art CPU based designs and show that our accelerator obtains speedups in the range of 3.4×∼6.7× in latency for the convolution layers of ResNet-50.

II. Related Work

Acceleration of FHE based Neural Network inference has received attention recently due to their significance in enabling privacy preserving ML applications. CryptoNets [4] was the first work to enable neural network inference over encrypted data. The authors designed a framework for neural network inference on a batch of images. As FHE does not support non-polynomial computations, their framework only uses average pooling and replaces all activation functions with the square function $f(z) = z^2$. CryptoDL [10] further explored polynomial approximations for typical activation functions, i.e., ReLU, Sigmoid and Tanh, to improve the prediction accuracy. HCNN [11] implemented the first GPU acceleration based on CryptoNets and achieved low amortized latency for each image in the batch. However, these works perform inference on large batches and are not suitable for performing low latency inference on a single image.

To perform inference on a single image, authors of Gazelle [5] developed a single image encoding scheme and proposed algorithms for secure linear algebra kernels including homomorphic matrix-vector multiplication and homomorphic convolution. They also designed a secure protocol based on garbled circuits (GC) to perform non-linear computations with the collaboration between the server and the client. Cheetah [2] optimized Gazelle by modeling the noise growth and selecting the minimal FHE parameters to reduce the computation complexity. They further implemented the algorithms on custom accelerators. However, both Gazelle and Cheetah require performing expensive Rotation operations (Section III-A). In contrast, authors of ENSEI [12] developed a frequency domain algorithm for HE-Convolution that simply uses Hadamard products and accumulations.

All the prior works have targeted either a CPU or a GPU platform. A few works have accelerated FHE primitives on FPGAs. These include [9] that accelerates the primitives of the BFV [13] [14] encryption scheme, and HEAX [8] that accelerates the primitives of the CKKS scheme [15] including the Number Theoretic Transform (NTT) that transforms polynomials into the evaluation space to simplify polynomial multiplications. It is non-trivial to develop an efficient HE-CNN implementation using these primitives as naive implementation will result in significant computational complexity due to Rotation operations. To the best of our knowledge, no implementation exists for performing HE-CNNs on FPGAs.

This work proposes the first FPGA-based low latency inference accelerator for convolutional layers of ResNet-50. We achieve it by implementing the im2col [16] and frequency domain convolution [12] over homomorphic encrypted data, which avoid expensive Rotation operations, on FPGA.

III. Background

A. Homomorphic Encryption Scheme

Homomorphic encryption (HE) provides a practical way for privacy-preserving computations. It was first proposed by Gentry [3] using lattice-based cryptography. It allows direct computations, including addition and multiplication, over ciphertext without revealing the original data. Several homomorphic encryption schemes have been developed recently, including BGV [17], BFV [13] [14], CKKS [15], etc. A noise term is included in a ciphertext of HE to hide the underlying message. The noise is multiplicative and grows rapidly when homomorphic multiplications are performed. Every ciphertext has a restriction on the maximum amount of noise (called "noise budget"). When a ciphertext runs out of its noise budget, the original plaintext will be corrupted and lost. Therefore, practical HE applications usually limit the maximum depth of computations to control the noise.

In this work, we choose the BFV scheme. The scheme has three parameters (N, p_E, Q), where N defines the degree of polynomials, p_E is the plaintext modulus, and Q is the ciphertext modulus. The plaintext space is $\mathbb{Z}_{p_E}^N$, which means a vector of N elements from the finite ring of integers $\mathbb{Z}_{p_E}$, i.e., integers modulo p_E. Denote $\mathcal{R} = \mathbb{Z}[X]/(X^N + 1)$ where N is a power of 2. Then $\mathcal{R}$ is a ring for polynomials with degree of $N - 1$. The ciphertext space is $\mathcal{R}_Q^2 = (\mathcal{R}/Q\mathcal{R})^2$, which means a pair of polynomials with coefficients from $\mathbb{Z}_Q$.

For two plaintext vectors $\mathbf{u}, \mathbf{v} \in \mathbb{Z}_{p_E}^N$, we denote their encrypted ciphertexts as $[\mathbf{u}], [\mathbf{v}] \in \mathcal{R}_Q^2$ respectively. The primitive computations supported by the BFV scheme are as follows:

- Addition: $\mathrm{Add}([\mathbf{u}], [\mathbf{v}]) = [\mathbf{u} + \mathbf{v}]$.
- Plaintext-Ciphertext Multiplication (*Pt-Ct Mult*): $\mathrm{Mult}([\mathbf{u}], \mathbf{v}) = [\mathbf{u} \circ \mathbf{v}]$, where $\mathbf{u} \circ \mathbf{v}$ is the Hadamard product (i.e., element-wise product) of two vectors.
- Ciphertext-Ciphertext Multiplication (*Ct-Ct Mult*): $\mathrm{Mult}([\mathbf{u}], [\mathbf{v}]) = [\mathbf{u} \circ \mathbf{v}]$.
- Rotation: Let $\mathbf{u} = (u_0, u_1, ..., u_{N-1})$, then $\mathrm{Rotate}([\mathbf{u}], k) = [u_k, u_{k+1}, ..., u_{N-1}, u_0, ..., u_{k-1}]$ for $k \in \{0, 1, ..., N-1\}$.

Among these, Addition and Pt-Ct Mult have low computation complexity and can be embarrassingly parallel. However, Ct-Ct Mult and Rotation are more expensive because they are composed of a long sequence of procedures [13].

A plaintext vector is generated as follows: (1) N integers from $\mathbb{Z}_{p_E}$ are packed into an initial vector. (2) The initial vector is mapped into a polynomial from $\mathcal{R}_{p_E}$ using the Chinese

Remainder Theorem (CRT) [4]. The polynomial is represented as a coefficient vector from $\mathbb{Z}_{p_E}^N$. (3) The coefficient vector is transformed with NTT into the plaintext vector.

The typical value of N varies from 1K to 32K [6]. The corresponding ciphertext modulus Q can be hundreds of bits, which exceeds the word width of actual processors. The solution is to use CRT again to break the large Q into several smaller moduli [18] that fit the machine word width. All the moduli usually have approximately the same bit width denoted as $\log q$. Each ciphertext is transformed into multiple pairs of polynomials with smaller coefficients. The computations of Addition and Pt-Ct Mult can be performed on all pairs of polynomials independently. Note that before the computations, all polynomials in the ciphertext have been transformed into the evaluation space using NTT to simplify polynomial multiplications to Hadamard products.

B. *Privacy Preserving CNN Inference Protocol*

We define client as the owner of the plaintext data. In this work, the data is an image. Server is defined as the host/owner of the pre-trained CNN model. The client needs to perform an inference on the image using the CNN model hosted by the server. The privacy of the input image and the inference result needs to be guaranteed. Our CNN inference protocol uses BFV scheme of homomorphic encryption for linear computations (convolutional layers and fully connected layers). The key steps in our protocol are as follows:

1) *Setup.* The client locally sets up a public key and private key. The public key is used to encrypt the image into ciphertext. The private key is used to decrypt the ciphertext (result) back to plaintext.
2) *Convolutional layers.* (a) On the client, the input images are transformed to the format required by the chosen convolution algorithm (Section IV-A). The transformed data are packed into plaintext vectors which are then encrypted into ciphertexts and sent to the server. (b) The server receives the ciphertexts from the client, computes the convolution, and sends the result back to the client. The focus of this work is to accelerate the computation in this step using FPGAs. (c) After receiving the results from the server, the client decrypts them into plaintexts and performs reverse transformations as required by the chosen convolution algorithm.
3) *Fully connected layers.* The protocol for the fully connected layers are similar to the convolutional layers with the only difference being that format transformations are not needed.
4) *Non-linear computations.* FHE does not natively support non-linear computations. Thus, non-linear layers such as pooling or activation layers are performed jointly by the client and the server under the GC protocol. More details are described in Gazelle [5].

In this paper, we focus on accelerating and modeling the computations for convolutional layers which are executed on the server.

IV. Algorithms and Accelerator Design for HE-Convolution

Convolutional layers are the main components of most common CNN models. As convolution is essentially composed of Multiply and Accumulate (MAC) operations, it can be performed securely with homomorphic encryption. In this section, we describe how to apply two popular convolution algorithms, im2col and frequency domain convolution, over homomorphic encrypted data. Then we propose an FPGA architecture to accelerate the two algorithms.

A. *HE-Convolution Algorithms*

A typical convolutional layer consists of an $n \times n \times f_{in}$ input image X convolved with a set of filters W of dimension $f \times f \times f_{in} \times f_{out}$ to output a $u \times u \times f_{out}$ image M. Here, f_{in} and f_{out} denote the number of input and output channels, respectively. Let s denote the stride.

Prior works [5] [2] perform straightforward HE-Convolution and thus require expensive Rotation operations. In contrast, we use im2col convolution or frequency domain convolution that only use Pt-Ct Mult and Additions.

Im2col convolution converts convolution into matrix multiplication. Specifically, X is transformed into a matrix X' of dimension $u^2 \times (f^2 f_{in})$, and W is transformed into a matrix W' of dimension $(f^2 f_{in}) \times f_{out}$. Then the convolution $X * W$ is equivalent to the matrix multiplication $X'W'$. Each column of $X'W'$ contains all pixels of one channel of the output image. Assuming the input image is padded, we have $u = 1 + \frac{n-1}{s}$.

In order to allow the server to compute $X'W'$ securely, the client should pack all elements from X' into vectors of length N as plaintexts, and then encrypt them into homomorphic ciphertext. Our packing scheme guarantees that each vector only contains elements from the same column of X'. The concrete packing scheme depends on the vector size N:

1) When a single vector is large enough, i.e., $N \geq u^2$, we can pack $\lfloor N/u^2 \rfloor$ copies of the same column into a vector to compute multiple output channels simultaneously. After receiving the encrypted vectors, the computations on the server are $f^2 f_{in} f_{out} / \lfloor N/u^2 \rfloor$ Pt-Ct Mult and Additions.
2) When the vector size is not large enough for multiple copies of a column, i.e., $N < u^2$, we pack each column of X' into $\lceil u^2/N \rceil$ vectors. After receiving the encrypted vectors, the computations of the server are $f^2 f_{in} f_{out} \lceil u^2/N \rceil$ Pt-Ct Mult and Additions.

For both cases, we can approximate the number of computations as $f^2 f_{in} f_{out} u^2/N$.

Frequency domain convolution converts a convolution into Hadamard products and accumulations by transforming the input image X and filters W into frequency domain X' and W' using two-dimensional NTT. The transformation of W' is pre-processed by the server. On the client, each channel of the input image is padded into size of $u \times u$, transformed into frequency domain, flattened into one dimension, and finally packed into vectors. Here u is the minimum power of 2 and no less than $n + f - 1$.

1) When a single vector is large enough, i.e., $N \geq u^2$, we can pack $\lfloor N/u^2 \rfloor$ copies of the same channel into a vector to compute output multiple channels simultaneously. The computation on the server are $f_{in} f_{out} / \lfloor N/u^2 \rfloor$ Pt-Ct Mult and Additions.
2) When the vector size is not large enough to contain an entire channel, i.e., $N < u^2$, we can pack a channel into $\lceil u^2/N \rceil$ vectors. The computation on the server are $f_{in} f_{out} \lceil u^2/N \rceil$ Pt-Ct Mult and Additions.

For both cases, the number of computations can be approximated as $f_{in} f_{out} u^2 / N$.

Our packing scheme for im2col and frequency domain convolution guarantees that any pair of elements to be summed or multiplied are in the same position in the vector. Therefore, we only need Pt-Ct Mult and Additions without the expensive Rotations.

B. Accelerator Design

On the server, the computations for both im2col and frequency domain convolution over homomorphic encrypted data are essentially Pt-Ct Mult and Additions. For a ciphertext, the computations are performed on all its polynomials independently. Each polynomial is represented by a coefficient vector. Then both convolution algorithms can be abstracted as follows. Let n_{out} be the number of output vectors, and n_{in} be the number of required input vectors for computing each output. The server computes

$$\mathbf{C}_j = \sum_{i=0}^{n_{in}-1} \mathbf{A}_i \circ \mathbf{B}_{j,i} \tag{1}$$

for $j = 0, 1, ..., n_{out} - 1$. Here $\mathbf{A}_i \in \mathbb{Z}_q^N$ is a vector of size N representing an evaluation space polynomial from the ciphertext input images. $\mathbf{B}_{j,i} \in \mathbb{Z}_{p_E}^N$ is a vector of size N generated from the plaintext filters. $\mathbf{C}_j \in \mathbb{Z}_q^N$ is a vector representing an evaluation space polynomial of the output ciphertext. The operator $\circ$ is the Hadamard product of two vectors. To make the architecture more flexible for different amounts of hardware resources, each vector $\mathbf{A}_i$ is split into r sub-vectors $\mathbf{A}_i^{(0)}, \mathbf{A}_i^{(1)}, ..., \mathbf{A}_i^{(r-1)}$. Each sub-vectors has a size of $N_e = N/r$. The vectors $\mathbf{B}_{j,i}$ and $\mathbf{C}_j$ are split into r sub-vectors similarly. The architecture shown in Figure 1 is to compute one output sub-vector $\mathbf{C}_j^{(l)}$ for a certain $l \in \{0, 1, ..., r-1\}$. The architecture is reused r times sequentially to get the complete $\mathbf{C}_j$.

To achieve a low latency, our design uses the following ideas. First, we use multiple processing elements (PEs) to compute the Hadamard products and accumulations for multiple input sub-vectors simultaneously. An adder tree is then used to accumulate the output of all PEs and get $\mathbf{C}_j^{(l)}$. Second, as all of $\mathbf{C}_0^{(l)}, ..., \mathbf{C}_{n_{out}-1}^{(l)}$ depend on $\mathbf{A}_0^{(l)}, ..., \mathbf{A}_{n_{in}-1}^{(l)}$, we reuse these n_{in} input sub-vectors until all of the n_{out} output sub-vectors

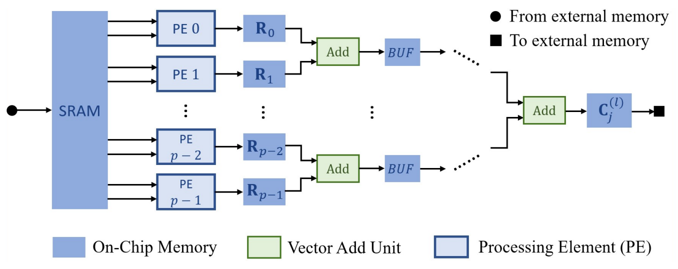

Fig. 1. Top Level Architecture

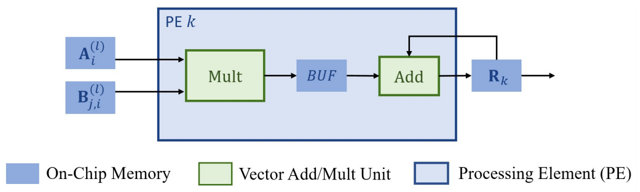

Fig. 2. Processing Element (PE)

are computed. Third, all the input and output data are double-buffered to completely hide the latency of data communication between the FPGA and the external memory.

Prior to the execution, the input vectors, including $\mathbf{A}_0^{(l)}, ..., \mathbf{A}_{n_{in}-1}^{(l)}$ and $\mathbf{B}_{j,0}^{(l)}, ..., \mathbf{B}_{j,n_{in}-1}^{(l)}$, are pre-fetched from the external memory into the on-chip memory. The on-chip input vectors are split into p partitions and assigned to p PEs. PE$_k$ computes the Hadamard-accumulation

$$\mathbf{R}_k = \sum_{i=k \cdot n_{in}/p}^{(k+1) \cdot n_{in}/p - 1} \mathbf{A}_i^{(l)} \circ \mathbf{B}_{j,i}^{(l)} \tag{2}$$

There are n_{in}/p pairs of $\mathbf{A}_i^{(l)}$ and $\mathbf{B}_{j,i}^{(l)}$ sent into PE$_k$ sequentially, multiplied and accumulated. As any coefficient of the ciphertext is in the finite ring of integers $\mathbb{Z}_q$, all integer computations in the PE are followed by a modulo q opreation. We use Barrett Reduction Algorithm [19] for efficient modular multiplication. This algorithm performs three integer multiplications without expensive division operations. In each PE, a Mult unit computes the Hadamard product of two sub-vectors, and an Add unit computes the sum of two sub-vectors. Each unit has a pipeline that computes m elements every cycle. After all the PEs finish the computation and save the results of all $\mathbf{r}_k$ to the buffers, an adder tree starts to sum them up to get the output sub-vector $\mathbf{C}_j^{(l)}$.

V. Performance Modeling for HE-Convolution

A. Parameters

We first summarize all the parameters for the performance model as follows.

- Parameters about the encryption
  - N: polynomial degree. It is also the length of each coefficient vector.
  - $\log p_E$: width of plaintext modulus.
  - $\log Q$: width of ciphertext modulus depending on N.
  - $\log q$: width of each ciphertext modulus split from Q (See Section III-A).

- Parameters about the convolutional layer
  - $n \times n \times f_{in}$: dimension of the imput image.
  - $f \times f \times f_{in} \times f_{out}$: dimension of the filters.
  - s: convolution stride.
- Parameters about the hardware design
  - p: number of PEs in the architecture.
  - m: unrolling factor for the computation in each PE.
  - n_{out}: number of output sub-vectors for the layer.
  - n_{in}: number of input sub-vectors for each output.
  - N_e: length of each input/output sub-vector.

$\log p_E$ and $\log q$ are assumed as fixed values. It is observed that, for a given N, a larger $\log Q$ leads to more noise budget but lower security level [6]. We select the smallest N such that there exists a $\log Q$ providing enough noise budget as well as at least 128-bit security level. As the noise budget consumption depends on the amount of computations, the values of N and $\log Q$ vary for im2col and frequency domain convolution. We denote the values of them as N_{ic} and Q_{ic} for im2col, and N_{fd} and Q_{fd} for frequency domain.

B. Performance Model for the Architecture

Latency: As a Mult unit is a pipeline that computes m elements per cycle, its latency is $T_{mult} = t_{mult} - 1 + \frac{N_e}{m}$ cycles. Similarly, the latency of an Add unit is $T_{add} = t_{add} - 1 + \frac{N_e}{m}$. The term t_{mult} and t_{add} are the latency for the unit to generate the first output element, which are assumed as known values. Then the latency of computing one output vector $\mathbf{C}_j$ is

$$T_{out} = (T_{mult} + T_{add}) \cdot \frac{n_{in}}{p} + T_{add} \cdot \log p \tag{3}$$

cycles. The first term is for the PEs and the second term is for the adder tree.

DSP resources: Our architecture has p PEs, and each PE performs multiplications and additions on m elements in parallel. Thus the number of DSPs is

$$\mathcal{D} = mp \cdot d_{mult} \tag{4}$$

The factor d_{mult} refers to the number of DSPs to perform the modular multiplication, which depends on $\log p_E$ and $\log q$.

On-chip memory: According to Section IV-B, we need to keep the sub-vectors $\mathbf{A}_0^{(l)}, ..., \mathbf{A}_{n_{in}-1}^{(l)}$ and $\mathbf{B}_{j,0}^{(l)}, ..., \mathbf{B}_{j,n_{in}-1}^{(l)}$ on chip using double buffer strategy. Thus the number of bits to save on chip is

$$\mathcal{M} = 2N_e n_{in}(\log p_E + \log q) \tag{5}$$

External bandwidth: Every T_{out} cycles, n_{in} sub-vectors $\mathbf{B}_{j,0}^{(l)}, ..., \mathbf{B}_{j,n_{in}-1}^{(l)}$ are loaded onto the chip, and one sub-vector $\mathbf{C}_j^{(l)}$ is exported to the external memory. Due to the reuse of $\mathbf{A}_0^{(l)}, ..., \mathbf{A}_{n_{in}-1}^{(l)}$ mentioned in Section IV-B, n_{in} sub-vectors are loaded every $n_{out}T$ cycles. In total, the external bandwidth is

$$\mathcal{B} = \frac{N_e}{T_{out}} \cdot (n_{in} \log p_E + \frac{n_{in}}{n_{out}} \log q + \log q) \tag{6}$$

C. Performance Model for Convolutional Layers

We build a performance model for both im2col and frequency domain convolution by predicting the number of times the FPGA design is reused to execute the computations. The proposed architecture computes n_{in} Pt-Ct Mult and Additions over a sub-vector of size N_e.

For im2col, the total number of Pt-Ct Mult and Additions is $f^2 f_{in} f_{out} u_{ic}^2 / N_{ic}$. Here $u_{ic}^2 = (1 + \frac{n-1}{s})^2$ is the size of an output image channel for im2col. Thus the architecture should be reused

$$\alpha_{ic} = \frac{N_{ic}}{N_e} \cdot \frac{f^2 f_{in} f_{out} u_{ic}^2}{n_{in} N_{ic}} \cdot \frac{\log Q_{ic}}{\log q} \tag{7}$$

times. We then derive the total latency for im2col as $T_{ic} = \alpha_{ic} T_{out}$ cycles.

For frequency domain, the total number of Pt-Ct Mult and Additions is $f_{in} f_{out} u_{fd}^2 / N_{fd}$. Here u_{fd}^2 is the size of an output image channel for frequency domain. u_{fd} is the minimum power of 2 no less than $n + f - 1$. Thus the architecture should be reused

$$\alpha_{fd} = \frac{N_{fd}}{N_e} \cdot \frac{f_{in} f_{out} u_{fd}^2}{n_{in} N_{fd}} \cdot \frac{\log Q_{fd}}{\log q} \tag{8}$$

times. Then the total latency for frequency domain is $T_{fd} = \alpha_{fd} T_{out}$ cycles.

For a given convolutional layer, the performance model predicts the better algorithm with lower latency by the value of

$$\frac{\alpha_{ic}}{\alpha_{fd}} = f^2 \cdot \frac{u_{ic}^2}{u_{fd}^2} \cdot \frac{\log Q_{ic}}{\log Q_{fd}} \tag{9}$$

When $\alpha_{ic}/\alpha_{fd} < 1$, im2col outperforms frequency domain. When $\alpha_{ic}/\alpha_{fd} > 1$, frequency domain has a better performance. Note that Equation 9 is valid when a single FPGA design is used to accelerate both algorithms for all convolutional layers in a CNN without reconfiguration.

VI. Experimental Results

A. Experimental Setup

We target ResNet-50, a deep CNN with state-of-the-art image classification accuracy [20] for implementation. We select this model as it exhibits a wide variety in its convolutional layers configurations — n ($7 \sim 224$), f ($1 \sim 7$), f_{in} ($3 \sim 2048$) and f_{out} ($64 \sim 2048$). We use the notation *convx_y* for the layers consistent with the original notations for ResNet-50.

For baselines, we implement and optimize im2col and frequency domain algorithms for HE-Convolution in C++ using Microsoft SEAL library [21]. We target a state-of-the-art server with an AMD Ryzen Threadripper 3990X CPU @ 2.90 GHz with 64 cores and 128 threads. The server has 256 GB DDR4 with 200 GB/s peak bandwidth to DRAM. We use OpenMP 4.5 library [22] for parallel computations. Our FPGA architecture is synthesized and place-and-routed using Xilinx Vivado HLS 2020.1 [23]. The experiments are conducted on Xilinx Virtex UltraScale+ xcvu7pflva2104 FPGA [24]. It has 4560 DSPs, 1576K FFs, 788K LUTs, 51 Mb of BRAM and

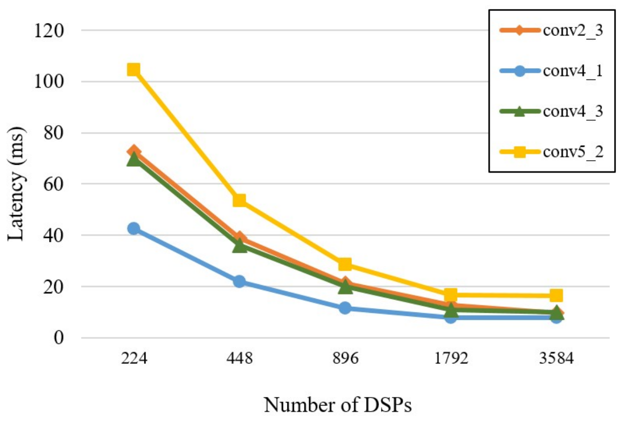

Fig. 3. Scalability with respect to DSP resources

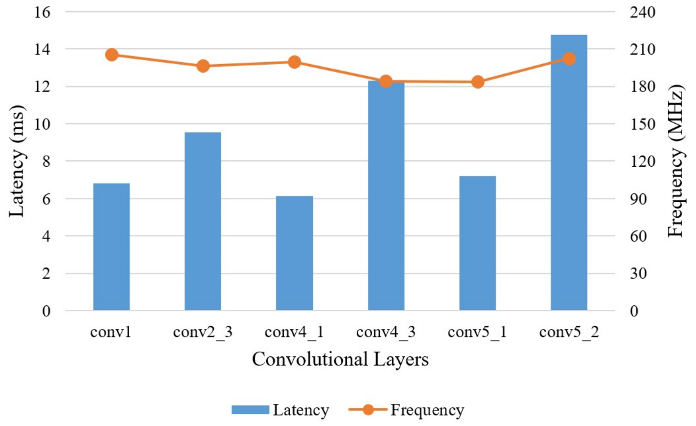

Fig. 4. Latency and frequency for various convolutional layers

TABLE I
PARAMETERS AND PERFORMANCE OF CONVOLUTIONAL LAYERS

Layer	n	f	f_{in}	f_{out}	s	CPU (ms)	Im2col			Frequency Domain			Speedup
							N_{ic}	$\log Q_{ic}$	FPGA (ms)	N_{fd}	$\log Q_{fd}$	FPGA (ms)	
conv1	224	7	3	64	2	70	2048	54	26.32	4096	109	**18.05**	3.9×
conv2_3	56	1	64	256	1	43	2048	54	**12.03**	4096	109	18.05	3.6×
conv4_1	28	1	512	256	2	34	4096	109	**7.33**	2048	54	24.07	4.6×
conv4_3	14	1	256	1024	1	81	4096	109	14.67	2048	54	**12.03**	6.7×
conv5_1	14	1	1024	512	2	27	4096	109	**7.90**	2048	54	24.07	3.4×
conv5_2	7	3	512	512	1	79	4096	109	35.53	4096	109	**18.05**	4.4×

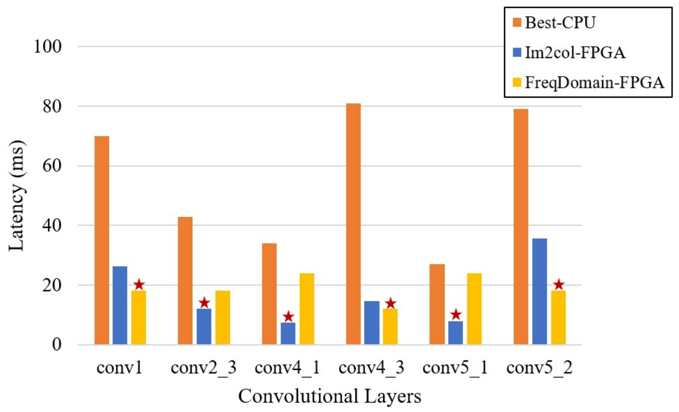

Fig. 5. Comparison of our FPGA implementation with the CPU baseline

180 Mb of UltraRAM. The peak external memory bandwidth is 78 GB/s.

For each layer, we assume that the transformed input data for each algorithm is stored in the DRAM. The latency is defined as the total time to load the data from the DRAM, perform the computations (as per the protocol in Section III-B), and write the results to the DRAM.

B. Experimental Evaluation

To demonstrate the scalability of the architecture, we first analyze the trade-offs between latency and DSP resources. Due to space limitations, we show four convolutional layers in Figure 3. We vary mp as 32, 64, 128, 256 and 512. Corresponding number of DSPs are 224, 448, 896, 1792 and 3584, respectively. For each layer and each number of DSPs we enumerate the possible values of m, p, N_e and n_{in}, and estimate the latency $\min\{T_{ic}, T_{fd}\}$ and resources consumed in terms of bandwidth $\mathcal{B}$ and on-chip memory $\mathcal{M}$ using the performance model defined in Section V. We perform place-and-route for the top-4 designs with the lowest estimated latency, and then report the lowest actual latency in Figure 3. As evident from the figure, for each layer, increasing the number of DSPs leads to a decrease in latency. However, the drop in latency is not linear because: (i) Latency of the adder tree increases with higher parallelism (Equation 3); (ii) The initialization latency of the pipeline (t_{mult}, t_{add}) is not affected by increasing DSPs and contributes to overall latency; (iii) Increased resources increases the interconnection complexity leading to a slight reduction in frequency. Note that the frequency variation of the designs is very low — 250 MHz (224 DSPs) to 220 MHz (1792 DSPs). When 3584 DSPs are used, the frequency drop is larger as significant resources are consumed.

In Figure 4, we show the performance of our architecture for six convolutional layers of ResNet-50 of different sizes. For each layer, we perform a design space exploration to determine the values of m, p, N_e and n_{in} that minimize the estimated latency. Then we perform place-and-route for the six selected designs. As illustrated in Figure 4, all the designs achieve similar frequency with a low variance. Note that in this subsection, we generate optimized designs for each layer separately to evaluate the performance our architecture. In the next subsection, a single FPGA design will be used to implement all the layers to avoid reconfiguration at runtime.

C. Comparison with the Baseline on CPU

For all the convolutional layers of ResNet-50, we measure the performance of the CPU baseline and the FPGA implementation. In contrast to the previous subsection, here, we use a single FPGA architecture to execute all the layers, i.e., our design does not require expensive runtime reconfiguration of FPGA to execute the entire CNN. We perform design space exploration to determine the values of m, p, N_e and n_{in} using our performance model to minimize the sum of $\min\{T_{ic},T_{fd}\}$ for all the convolutional layers. All of the four parameters are assumed to be powers of 2. Our design space exploration identified a design with the following parameters: N_e = 2048, n_{in} = 16, p = 4, m = 128. We then perform place-and-route for this design. The resulting design uses 3584 DSPs, 278.7K LUTs, 157.5K FFs, 76.8 GB/s external bandwidth and 3.69 Mb on-chip memory, and operates at 165.4 MHz. The latency of a single execution is 1.47 μs. The design space exploration takes less than 1 second on a CPU.

We measure the computation time of each convolutional layer of ResNet-50 using the CPU implementation of both im2col and frequency domain algorithms, and use the one with lower latency as the baseline. All layers can be accelerated by reusing the same FPGA design without runtime reconfiguration. For each layer, we compute α_{ic} and α_{fd}, i.e., the number of times the FPGA design is reused to execute the computations of the im2col and frequency domain algorithm. We obtain the execution time of the two algorithms by multiplying α_{ic} and α_{fd} by the latency of a single FPGA execution (1.47 μs). Table I shows the parameters and performance of the chosen convolutional layers of ResNet-50. The layers were chosen to illustrate the speedup for various parameters of ResNet-50 layers. Figure 5 illustrates the performance comparison for these layers. The FPGA design obtains speedups in the range of 3.4$\times$ $\sim$ 6.7$\times$. The first two layers show lower speedups because their CPU baseline are faster than the others. We also use our performance model to predict the algorithm with lower latency by computing the value of α_{ic}/α_{fd} (Section V). The prediction using our model for each layer is marked with a red star in Figure 5. It shows that our performance model predicts the algorithm with lower latency correctly for all the chosen layers.

VII. Conclusion

In this paper, we proposed a unified FPGA design that accelerates both im2col and frequency domain HE-Convolution. We developed a performance model that enables design space exploration and algorithm selection for each convolutional layers. We targeted ResNet-50 for acceleration and experimentally demonstrated that our designs achieved speedup in the range of 3.4$\times$ $\sim$ 6.7$\times$ in latency.

Acknowledgement

This work has been sponsored by the U.S. National Science Foundation under grant numbers OAC-1911229 and CNS-2009057. Equipment and support by Xilinx are greatly appreciated.

References

[1] T. Ye, R. Kannan, and V. K. Prasanna, "Accelerator design and performance modeling for homomorphic encrypted cnn inference," in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2020, pp. 1–7.

[2] B. Reagen, W. Choi, Y. Ko, V. Lee, G.-Y. Wei, H.-H. S. Lee, and D. Brooks, "Cheetah: Optimizing and accelerating homomorphic encryption for private inference," 2020.

[3] C. Gentry, "Fully homomorphic encryption using ideal lattices," ser. STOC '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 169–178.

[4] R. Gilad-Bachrach, N. Dowlin, 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 Machine Learning*, 2016, pp. 201–210.

[5] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan, "Gazelle: A low latency framework for secure neural network inference," in *Proceedings of the 27th USENIX Conference on Security Symposium*, ser. SEC'18. USA: USENIX Association, 2018, p. 1651–1668.

[6] 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*, 2017.

[7] Y. Meng, S. Kuppannagari, R. Kannan, and V. Prasanna, "Dynamap: Dynamic algorithm mapping framework for low latency cnn inference," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2021, pp. 183–193.

[8] 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.

[9] S. S. Roy, F. Turan, K. Jarvinen, F. Vercauteren, and I. Verbauwhede, "Fpga-based high-performance parallel architecture for homomorphic computing on encrypted data," Cryptology ePrint Archive, Report 2019/160, 2019, https://eprint.iacr.org/2019/160.

[10] E. Hesamifard, H. Takabi, and M. Ghasemi, "Cryptodl: Deep neural networks over encrypted data," 2017.

[11] 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, "Towards the alexnet moment for homomorphic encryption: Hcnn, the first homomorphic cnn on encrypted data with gpus," Cryptology ePrint Archive, Report 2018/1056, 2018, https://eprint.iacr.org/2018/1056.

[12] S. Bian, T. Wang, M. Hiromoto, Y. Shi, and T. Sato, "Ensei: Efficient secure inference via frequency-domain homomorphic convolution for privacy-preserving visual recognition," in *2020 IEEE/CVF CVPR*, 2020, pp. 9400–9409.

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

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

[15] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," Cryptology ePrint Archive, Report 2016/421, 2016, https://eprint.iacr.org/2016/421.

[16] K. Chellapilla, S. Puri, and P. Simard, "High performance convolutional neural networks for document processing," in *Tenth International Workshop on Frontiers in Handwriting Recognition*. Suvisoft, 2006.

[17] 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. Association for Computing Machinery, 2012, p. 309–325.

[18] J.-C. Bajard, J. Eynard, A. Hasan, and V. Zucca, "A full rns variant of fv like somewhat homomorphic encryption schemes," Cryptology ePrint Archive, Report 2016/510, 2016, https://eprint.iacr.org/2016/510.

[19] 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*, 1987, p. 311–323.

[20] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[21] "Microsoft SEAL (release 3.0)," http://sealcrypto.org, Oct. 2018, microsoft Research, Redmond, WA.

[22] OpenMP Architecture Review Board, "OpenMP application program interface version 4.5," 2015.

[23] Xilinx, "Vivado design suite user guide: High-level synthesis ug902 (v2020.1)," 2020.

[24] Xilinx, "Ultrascale architecture and product data sheet: Overview (ds890)," 2021.