FPGA Accelerator for Homomorphic Encrypted Sparse Convolutional Neural Network Inference

Yang Yang[1], Sanmukh R. Kuppannagari[1], Rajgopal Kannan[2] and Viktor K. Prasanna[1]
[1]Department of Electrical and Computer Engineering, University of Southern California
[2]US Army Research Lab
Email: {yyang172, kuppanna, prasanna}@usc.edu, rajgopal.kannan.civ@mail.mil

***Abstract*—Homomorphic Encryption (HE) is a promising solution to the increasing concerns of privacy in machine learning. But HE-based CNN inference remains impractically slow. Pruning can significantly reduce the compute and memory footprint of CNNs. However, homomorphic encrypted Sparse Convolutional Neural Networks (SCNN) have vastly different compute and memory characteristics compared with unencrypted SCNN. Simply extending the design principles of existing SCNN accelerators may offset the potential acceleration offered by sparsity. To realize fast execution, we propose an FPGA accelerator to speedup the computation of linear layers, the main computational bottleneck in HE SCNN batch inference. First, we analyze the memory requirements of various linear layers in HE SCNN and discuss the unique challenges. Motivated by the analysis, we present a novel dataflow specially designed to optimize HE SCNN data reuse coupled with an efficient scheduling policy that minimizes on-chip SRAM access conflicts. Leveraging the proposed dataflow and scheduling algorithm, we demonstrate the first end-to-end acceleration of HE SCNN batch inference targeting CPU-FPGA heterogeneous platforms. For a batch of 8K images, our design achieves up to $5.6\times$ speedup in inference latency compared with the CPU-only solution for widely studied 6-layer and 11-layer HE CNNs.**

***Index Terms*—FPGA acceleration, homomorphic encryption, sparse neural networks, parallel computing**

I. Introduction

Homomorphic Encryption (HE) provides a promising solution for privacy-preserving machine learning. In HE, the client provides encrypted data to a cloud server, on which the computation is performed *without* decrypting the data [1]. During encryption, the unencrypted input data is transformed into high degree polynomial representation. All the HE computations are carried out over a polynomial ring with a large modulus (up to several hundreds of bits [2]). While HE offers strong privacy guarantees for CNN inference in public cloud [3], [4], it comes at a high cost: inference using HE CNN is orders of magnitude slower than inference on unencrypted data [5]. The primary bottleneck in HE CNN batch inference acceleration is the high computation complexity and the large memory usage of the linear layers (convolutional and fully-connected layers) [3], [6], [7]. As reported in [6], more than 93% of the inference time is spent on the linear layers in a 11-layer CNN.

Standard (unencrypted) CNN models are typically pruned to produce Sparse CNNs (SCNN) with reduced compute and memory footprint [8], [9]. While accelerating SCNNs for unencrypted data is well studied [10], [11], little work has been done on accelerating SCNN inference for Homomorphic Encrypted data. Using SCNN models on HE data offers a straightforward approach to reduce the number of homomorphic operations. However, due to the overheads imposed by the encryption process that lead to a significantly larger activation memory footprint, the inference latency for HE SCNNs is still significant.

Figure 1 shows the size of one input tile of the convolutional layers in CryptoNets [6] for a batch size of 8192, where an input tile is defined as the amount of data required to compute one output. With HE, the size increases substantially to a few hundreds MiB (Section II). As the encrypted input tile is too large to be completely stored in on-chip SRAM, data reuse is critical to amortize the overhead of DRAM data transfers. Also, the unstructured sparsity of parameters in different output channels that convolve the same encrypted input tile poses a challenge for optimizing data reuse. Using the same dataflows that are used to design sparse accelerators for unencrypted data (e.g., input or output stationary [12]) can lead to suboptimal designs. Faster CryptoNets [13] is the first work that successfully implements HE SCNN on a multi-core CPU. Due to the lack of optimized dataflow, their design cannot fully exploit the data reuse in HE. The inference incurs high latency. The fine grained parallel computing capability of FPGA makes it an attractive platform to accelerate HE. However, the scarcity of FPGA resources (e.g., on-chip SRAM) necessitates non-trivial approaches to create efficient designs.

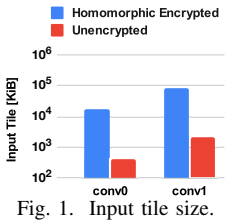


Fig. 1. Input tile size.

To enable efficient HE SCNN batch inference, we propose an FPGA accelerator to speed up the time consuming portion of the computation (i.e., linear layers). We first analyze the large memory footprint of the activations in HE SCNN batch inference, which is significantly different from the unencrypted SCNN. Motivated by the analysis, we propose a novel dataflow specially designed to optimize HE computation of linear layers. By rearranging the processing order of the polynomials in the ciphertext, the proposed dataflow allows more output polynomials along the channel dimension to be stored in the on-chip SRAM, thereby improving data reuse. Coupled with the dataflow, we propose a maximum bipartite matching based scheduling algorithm to minimize pipeline stalls due to sparsity. The scheduling algorithm assumes no sparsity structure in the neural network parameters. An FPGA accelerator is

designed to support the proposed dataflow and scheduling algorithm for various linear layers in HE SCNN. The key contributions of this paper are:

- We provide a comprehensive analysis of the memory requirement of HE SCNN batch inference to identify the bottleneck and data reuse opportunity in the linear layers. The analysis effectively shows that HE SCNN is vastly different from its unencrypted counterpart and requires a different accelerator design.
- We propose a novel dataflow by rearranging the processing order of the polynomials in the ciphertext. This dataflow improves the data reuse using the on-chip SRAM and reduces DRAM data transfers.
- We design a maximum bipartite matching based scheduling algorithm for the irregular on-chip polynomial buffer access due to sparsity. Compared with the commonly used approach where the scheduling is based on the order of the non-zero indices, the proposed algorithm significantly reduces pipeline stalls and inference latency.
- We design an FPGA accelerator using high-level synthesis based on the proposed dataflow and scheduling algorithm. The accelerator can be programmed to process various linear layers in HE SCNN.
- We demonstrate the first end-to-end acceleration of HE-SCNN batch inference targeting CPU-FPGA heterogeneous platforms. For a batch of 8K images, our design achieves up to $5.6\times$ speedup in inference latency compared with the CPU-only solution for widely used 6-layer and 11-layer CNN models.

II. Background

A. Homomorphic Encryption

In this paper, we use the BFV scheme [14]. The analysis and acceleration techniques can also be applied to other HE schemes (e.g. CKKS [15]). In BFV, cleartext is first encoded as a plaintext polynomial (pt), which is then encrypted as a pair of ciphertext polynomials $ct = (c_0, c_1)$. After homomorphic computation, the output ciphertext is decrypted to plaintext and decoded to cleartext. There are three parameters in BFV: N, t and Q, where N is a power-of-two number that defines the degree of plaintext and ciphertext polynomials, t is the plaintext modulus and Q is the ciphertext modulus. Let $\mathcal{R} = \mathbb{Z}[X]/(X^N+1)$, a plaintext is a polynomial in the ring $\mathcal{R}_t = \mathcal{R}/t\mathcal{R}$ with coefficients from $\mathbb{Z}_t$, i.e., integers modulo t. The ciphertext space is $\mathcal{R}_Q^2 = (\mathcal{R}/Q\mathcal{R})^2$, which means a pair of polynomials with coefficients from $\mathbb{Z}_Q$. BFV supports the following HE arithmetic:

- pt-ct add: The addition of plaintext m and ciphertext (c_0, c_1) is ciphertext $(c_0 + m, c_1)$.
- ct-ct add: The addition of ciphertext $(c_{0,0}, c_{0,1})$ and $(c_{1,0}, c_{1,1})$ is $(c_{0,0} + c_{1,0}, c_{1,0} + c_{1,1})$.
- ct scaling: The result of multiplying ciphertext (c_0, c_1) with a scalar k is ciphertext $(k \cdot c_0, k \cdot c_1)$.
- pt-ct mult: The multiplication of plaintext m and ciphertext (c_0, c_1) is $(m \cdot c_0, m \cdot c_1)$.
- ct-ct mult: The multiplication of two ciphertext consists of multiple steps, including polynomial multiplications and relinearization. As the acceleration of ct-ct mult is not the focus of this paper, we omit the details and refer the reader to [14].

The BFV parameters are determined by the security requirement. Typically N is in the range of 2^{10} to 2^{16} [2]. t is chosen based on the range of the data during computation. HE requires ciphertext modulus Q to be hundreds of bits depending on the multiplicative depth of the function to be evaluated [2], which is expensive to process. The Residue Number System (RNS) variant of the BFV scheme [14] enables representing a ciphertext polynomial with $\log Q$-bit coefficients as multiple polynomials with narrower coefficients. Let Q be a product of r co-primes $Q = \prod_{i=1}^{r} q_i$. A polynomial in $\mathcal{R}_Q$ can be represented as r polynomials, where the coefficients of the i-th polynomial are from $\mathbb{Z}_{q_i}$. HE operations are carried out on the RNS representation (r polynomials) in parallel. A limitation of the BFV scheme is that the number of HE operations is limited due to noise accumulation [16]. Bootstrapping can reset the noise and enable Fully Homomorphic Encryption computation. As the number of HE operations is known apriori in HE CNNs, bootstrapping can be avoided in accordance with several other works [3], [4], [5], [6], [13], [17].

B. Homomorphic Encrypted CNN Batch Inference

HE only supports addition and multiplication on polynomials. Thus, implementation of HE CNN requires: (i) transforming linear layers into polynomial operations, and (ii) devising alternate methods to compute non-linear layers (e.g. ReLU). For (ii), two approaches are typically adopted: *Hybrid* HE CNN [4], [18], [19] uses other secure computation techniques such as multi-party computation to compute non-linear layers while linear layers are performed by HE. However, Hybrid HE suffers from heavy communication cost between server and client [4], [20], [21]. To mitigate this issue, HE-*only* CNN approximates the non-linear layers using low degree polynomials, in which case the inference can be computed solely on the server [3], [6], [22]. Although most HE-only CNNs are not deep, they can still obtain competitive inference accuracy [6], [23], [24] and are applicable in many cases of interest [25], [26], [27], [28]. In this paper, we demonstrate an end-to-end acceleration of HE-*only* CNN.

For (i), we use the CNN to HE CNN transformation used in works such as [3], [17], [6]. Under this transformation, inference is carried out on *batches* of images to fully utilize the parallelism offered by polynomial operations. We denote h_{in}, w_{in}, c_{in} as the height, width and number of channels of the input tensor, $h_{out}, w_{out}, c_{out}$ as the dimensions of the output tensor. We use f to represent the convolution filter window. Given the polynomial degree N, N images of size $h_{in} \times w_{in} \times c_{in}$ are batched together and packed into $h_{in} \times w_{in} \times c_{in}$ plaintext polynomials: elements at the same coordinates across the N images are packed into a single plaintext. Then, the plaintext is encrypted into ciphertext. In this transformation, the parameters (filters) are not encrypted.

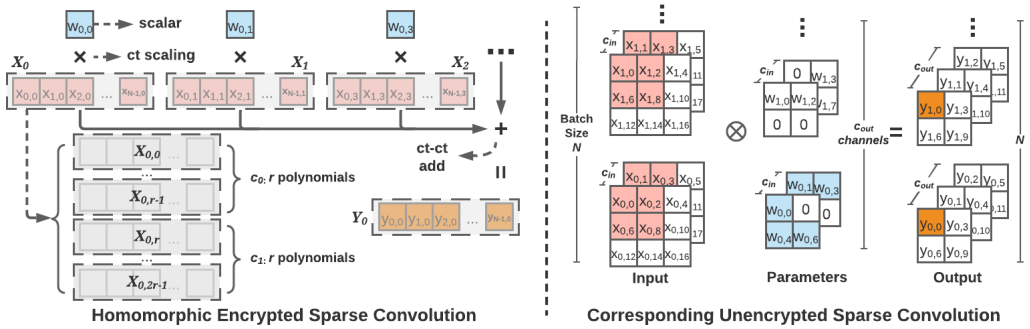


Fig. 2. Sparse convolution implementation in HE with the SIMD packing scheme.

Thus, convolution with a $f^2 \times c_{in} \times c_{out}$ filter w is performed by taking each element of w and scaling the input polynomials using ct scaling and accumulating to the output polynomial using ct-ct add. Computation over the polynomials are applied on all the N elements in a SIMD fashion. Fully-connected (fc) layers can be expressed using $1 \cdot 1$ convolution [29], thus we use the construction of convolution to represent both conv and fc layers. We use the following notations to describe the input and output of a linear layer in HE CNN:

- $x_{m,n}$ is the n-th element of the m-th input sample. $y_{m,n}$ is the corresponding output element. Both $x_{m,n}$ and $y_{m,n}$ represent unencrypted data.
- X_i is the input ciphertext of the i-th element. It encrypts a plaintext vector (polynomial) of $[x_{0,i}, x_{1,i}, ..., x_{N,i}]$. In addition, $X_{i,j}$ defines the j-th polynomial in ciphertext X_i under RNS representation ($0 \leq j < 2 \cdot r$). Y_i and $Y_{i,j}$ are the corresponding output ciphertext and polynomial.
- w is the unencrypted convolution filter.

Linear layers are implemented by HE as the following:

- Conv and fc layer can be expressed as $Y_j = \sum_i w_i \cdot X_i$. w_i is a parameter from the convolution filter.
- Pooling layer. Only linear pooling (e.g. average pooling) is supported by HE. For example, an average pooling of size $2 \cdot 2$ can be expressed as $Y_j = \sum_{i=1}^{4} X_i/4$.

Sparsity in Homomorphic Encrypted CNNs: Pruning is applied to the conv and fc layers of a CNN. After pruning, w turns into a sparse tensor. Figure 2 illustrates the HE sparse convolution that is accelerated in this paper. The left part shows the HE convolution and the right part shows the corresponding convolution in the unencrypted space. They both compute one output from each sample. HE convolution follows the same computation as the unencrypted convolution. The key differences are: 1) The input (X_i) and output (Y_i) of the convolution are ciphertext represented in RNS using $2 \cdot r$ polynomials. 2) Multiplication between X_i and parameters is achieved using ct scaling: coefficient-wise modular multiplication between the $2 \cdot r$ polynomials and the parameter. 3) Partial sum accumulation is performed using ct-ct add.

III. Memory Requirement and Dataflow Analysis

We adopt the magnitude-based unstructured pruning algorithm from [8] to obtain parameter sparsity and train two widely studied HE-CNNs for image classification: a 6-layer CNN for MNIST (CNN-6) and an 11-layer CNN for CIFAR-10 (CNN-11) [25], [6], [3], [13], [17]. The architecture of the two models is shown in Table I and Table II. The non-linear activation functions are approximated using degree-2 polynomials ($y = ax^2 + bx + c$) which can be computed using HE multiplications and additions [3], [6], [13], [20], [22]. Coefficients (a, b, c) are learnt through stochastic gradient descent during training. Because BFV only allows integer modular computations, fractional parameters of each layer are scaled up by a common factor Δ and rounded to integers [3], [17]. We empirically choose 8-bit of precision for the parameters of both models based on the study in [17]. Compared with the baseline — the dense models of HE CNN, CNN-6 and CNN-11 have little to no test accuracy drop when they are pruned to 90% and 50% sparsity respectively. The sparse CNN-6 achieves the same test accuracy as the original network (98.8%). The test accuracy drop of the sparse CNN-11 is only 0.4% (from 77.1% to 76.7%). Our sparse models achieve comparable test accuracy to the prior works [3], [13], [17].

TABLE I
CNN-6 for MNIST

Layer	Input Shape	Configuration
conv0	$28 \times 28 \times 1$	5 filters of size $5 \times 5 \times 1$
act0	$14 \times 14 \times 5$	Square
conv1	$14 \times 14 \times 5$	50 filters of size $5 \times 5 \times 5$
act1	$5 \times 5 \times 50$	Square
fc0	$5 \times 5 \times 50$	100 outputs
fc1	1×100	10 outputs

TABLE II
CNN-11 for CIFAR-10

Layer	Input Shape	Configuration
conv0	$32 \times 32 \times 3$	32 filters of size $3 \times 3 \times 3$
act0	$32 \times 32 \times 32$	Degree-2 polynomial
pool0	$32 \times 32 \times 32$	Average 2×2 pooling
conv1	$16 \times 16 \times 32$	64 filters of size $3 \times 3 \times 32$
act1	$16 \times 16 \times 64$	Degree-2 polynomial
pool1	$16 \times 16 \times 64$	Average 2×2 pooling
conv2	$8 \times 8 \times 64$	128 filters of size $3 \times 3 \times 64$
act2	$8 \times 8 \times 128$	Degree-2 polynomial
pool2	$8 \times 8 \times 128$	Average 2×2 pooling
fc0	$4 \times 4 \times 128$	256 outputs
fc1	1×256	10 outputs

Achieving speedup in inference latency requires overcoming several fundamental challenges. First, the intrinsic overheads imposed by the encryption lead to a significantly larger memory footprint for HE SCNNs. Since the encrypted input tiles are too large to be completely stored on-chip, data reuse becomes essential to amortize the overhead of DRAM data transfers. Concomitantly, the uneven sparsity of parameters in

different output channels that convolve the same encrypted input poses a challenge for optimizing data reuse. Finally, the scarcity of DSP and on-chip SRAM resources necessitates non-trivial approaches to avoid bottlenecks that impact inference latency. We tackle the above challenges through a dataflow specially designed to optimize HE SCNN data reuse coupled with an efficient scheduling policy that minimizes pipeline stalls in order to achieve true low latency inference. To motivate our optimized dataflow, resource allocation and pipeline scheduling algorithm, we first analytically quantify the memory overhead of HE SCNN.

A. Memory Requirement Analysis

We analyze the memory footprint of the linear layers in CNN-6 and CNN-11 using batch size of N. A ciphertext is composed of $2{\cdot}r$ degree-N polynomials, where r is determined by the RNS representation ($Q = \prod_{i=1}^{r} q_i$). Each coefficient in the polynomials is a $\log q_i$ bits integer. The size (bytes) of a ciphertext is $bytes_{ct} = 2 \cdot r \cdot N \cdot bytes_{coefficient}$. The input activation memory footprint is therefore $h_{in} \cdot w_{in} \cdot c_{in} \cdot bytes_{ct}$. The memory footprint of the parameters is $f^2 \cdot c_{out} \cdot c_{in} \cdot bytes\_per\_param \cdot (1 - sparsity)$.

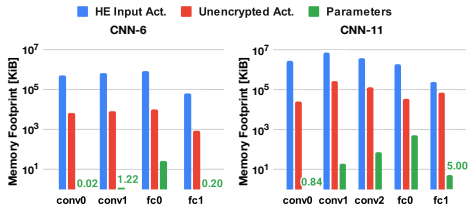


Fig. 3. Memory footprint of various linear layers in CNN-6 and CNN-11.

We choose HE parameters to ensure at least 91-bit security [2], [30]. The polynomial degree (N) of the plaintext and the ciphertext is 8192. The ciphertext modulus (Q) is 218-bit ($r = 5$) and 304-bit ($r = 7$) wide for CNN-6 and CNN-11 respectively. Each q_i is represented using a 64-bit integer. $bytes\_per\_param$ is 8 bytes. Figure 3 shows the memory footprint of various linear layers. Pooling layers in CNN-11 are not shown due to space limitation but they have similar footprint. As a comparison we also show the corresponding activation size before encryption. The memory footprint of HE activations can increase by one to two orders of magnitude after encryption. For unencrypted CNN, one can use a smaller batch size to reduce the memory footprint. Due to the polynomial representation, reducing the batch size in HE CNN does not lead to *any* memory footprint reduction. As a result, an input tile (i.e. $f^2 \cdot c_{in}$ ciphertext) that produces one output ranges from 17 MiB to 180 MiB, far exceeding the on-chip SRAM capacity in the state-of-the-art FPGAs [31], [32]. In contrast, the parameter size is quite small after pruning.

B. Dataflow Analysis

As, in convolution, outputs in different output channels convolve the same data, an input tile could be read from DRAM many times. Reusing input ciphertext as many times as possible once they are loaded into the on-chip SRAM is crucial for reducing the inference latency. Different dataflows have distinct data movement patterns, which imply different reuse opportunities. Unencrypted CNN accelerators can be classified into a taxonomy of five dataflows [12]. Mapping these dataflows to HE SCNN leads to inefficient designs. Here we discuss mapping the input stationary [33] and output stationary [34] dataflows to HE. Other dataflows in the taxonomy require either multicasting the input activations or duplicating the partial sums across PEs. They are not practical for HE because the large size of input/output ciphertext would make the on-chip SRAM requirement too high. All the dataflows are used to compute $1 \cdot 1 \cdot c_{out}$ outputs. To compute the entire output tensor, the same dataflow is repeated $h_{out} \cdot w_{out}$ times.

HE Input Stationary (Dataflow #1): We store m polynomials from $\lceil \frac{m}{2 \cdot r} \rceil$ input ciphertext on-chip and stream the c_{out} partial sum ciphertext, where m is a design time parameter and is determined based on on-chip SRAM capacity. Only the input ciphertext that are indexed by non-zero parameters are loaded. A partial sum ciphertext is reused for at most $\lceil \frac{m}{2 \cdot r} \rceil$ times. Then, it is written back to DRAM and a new partial sum is read from DRAM. Input ciphertext are read once and reused for up to c_{out} times.

HE Output Stationary (Dataflow #2): m partial sum polynomials from $\lceil \frac{m}{2 \cdot r} \rceil$ output ciphertext are kept in on-chip SRAM until all the computations to produce m outputs are finished. The input ciphertext, indexed by the non-zero parameters, are streamed from DRAM. This process repeats until the c_{out} output ciphertext are computed. This dataflow avoids storing and re-loading partial sums back and forth by reusing them for up to $f^2 \cdot c_{in}$ times. The input ciphertext, however, is read from DRAM for up to $\lceil \frac{2 \cdot r \cdot c_{out}}{m} \rceil$ times.

While Dataflow #1 and Dataflow #2 are standard dataflows, they offer limited data reuse for HE SCNN. Due to RNS polynomial representation, the number of activation ciphertext that can be stored on-chip is very small in these dataflows, which severely reduces the data reuse. We tackle this challenge by introducing an HE Optimized Dataflow, which aims to improve data reuse by rearranging the processing order of polynomials in HE convolution.

HE Optimized Dataflow (Dataflow #3): With RNS representation, $2 \cdot r$ polynomials in a ciphertext can be processed independently [35], i.e., the first polynomial of each output ciphertext only depends on the first polynomial of the corresponding input ciphertext. Unlike the aforementioned dataflows, this dataflow computes the i-th polynomial of all the output ciphertext first before processing the $(i+1)$-th polynomial of the output ciphertext ($1 \leq i \leq 2 \cdot r$). m partial sum polynomials, *one for each output ciphertext*, are fixed on-chip to produce m output polynomials. After the computation, the next set of m output polynomials are processed and so on. By reordering the computation of polynomials in the ciphertext, this dataflow allows the input ciphertext to be reused for up to m times, greater than Dataflow #2 by a factor of $2 \cdot r$. Each output ciphertext is still written to DRAM once.

Figure 4 uses an example to illustrate the 3 dataflows, where a $2 \cdot 2$ convolution is performed and $c_{in} = 1, c_{out} = 4, m = 4, r = 2$. In Dataflow #1, polynomials from Y_i are read from

and written to DRAM 4 times to produce the output. Dataflow #2 stores Y_i to DRAM once but loads X_i to on-chip SRAM for up to 4 times (depending on the sparsity). Dataflow #3, being the most optimal one among the three, can fully reuse the polynomials in X_i. Hence, this dataflow loads X_i and stores Y_i for at most 1 time.

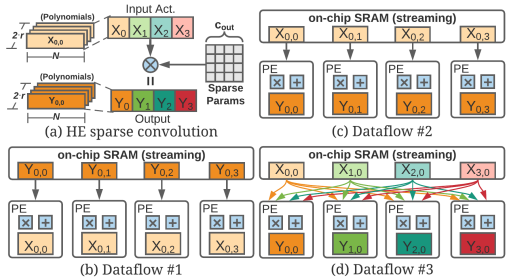


Fig. 4. Dataflow for the computation of linear layers in HE SCNN.

We quantitatively analyze the impact of the 3 dataflows on the DRAM data transfers. We use the HE parameters discussed in Section III-A. We choose $m = 8$ as the resources required by the dataflows are within the scope of possible implementation on an FPGA. As Figure 5 shows, Dataflow #1 almost always incurs the largest amount of data transfers due to the additional write traffic from partial sums. For layers that have large c_{out}, Dataflow #3 avoids a substantial amount of data transfers due to the increase in data reuse. This motivates us to design a hardware accelerator for Dataflow #3.

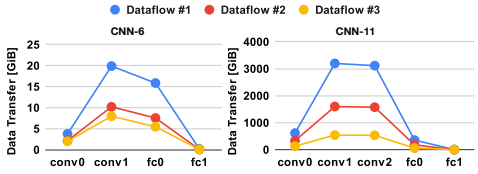


Fig. 5. Off-chip data transfer for different dataflows.

IV. Hardware Design

A. Accelerator Architecture

We design a fully pipelined FPGA accelerator optimized for Dataflow #3, as shown in Figure 6. The accelerator has an array of m Processing Elements (PEs). Each PE is responsible for computing one or more output ciphertext from different output channels. All the PEs run in lock-step and process the n-th polynomial in each ciphertext ($1 \leq n \leq 2 \cdot r$). The PEs are connected to k Polynomial Buffers (Poly Buffers) using a crossbar, where each buffer stores different input polynomials in the input tile. Both m and k are determined at design time. The Sequencer module runs a state machine to coordinate between the Poly Buffers and the PEs. At the start of a linear layer computation, k polynomials are loaded into the Poly Buffers. Non-zero parameters along with information on which Poly Buffer to access (Buffer ID) are transferred to the Sparse Params Buffer. Then each PE fetches the input polynomials based on the Buffer IDs and sends them to the MAC array. To improve performance and reduce design complexity, we adopt a static scheduling microarchitecture, in which the fetching of the k input polynomials and the order of accesses from the PEs are pre-determined offline using a scheduling algorithm (Section IV-B). After all the PEs have consumed the k input polynomials, the Sequencer initiates the Poly Buffers to fetch the next set of k input polynomials. The PEs write m output polynomials back to DRAM once they are generated and repeat this process for $\lceil \frac{2 \cdot r \cdot h_{out} \cdot w_{out} \cdot c_{out}}{m} \rceil$ times.

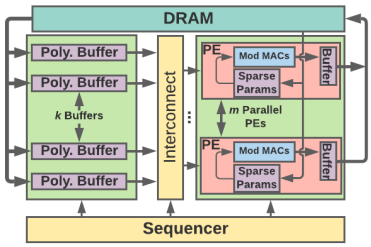
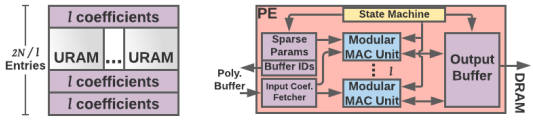


Fig. 6. HE SCNN accelerator architecture.

Polynomial Buffer and PE: Our PE exploits coefficient-wise parallelism by processing l coefficients per cycle, where l is determined at design time. Figure 7(a) shows the data layout of polynomials in the Poly Buffer, which is implemented using URAM. Each entry of the buffer stores l coefficients. There are $2 \cdot N/l$ entries in each buffer to store two polynomials (double buffered). Figure 7(b) shows the microarchitecture of the PE. A PE has a Sparse Params Buffer, an Input Coefficient Fetcher, an Output Buffer and a MAC array of size l. The Sparse Params Buffer stores the non-zero parameters and the corresponding Buffer IDs. The output buffer is sized to store two polynomials, one that is being processed and the other that is being transferred to DRAM. Each PE operates on the sparse parameters and input polynomials in the order defined by Dataflow #3. An output polynomial is produced after all the non-zero parameters and the corresponding input polynomials are multiplied and accumulated. Using the Buffer IDs, a polynomial from one of the Poly Buffers is streamed into a PE at the rate of l coefficients per cycle. At the same time, l coefficients are distributed to the l modular MAC units every cycle. An offline scheduling algorithm is executed to determine how many polynomials to read and the corresponding Buffer IDs in each iteration. This information is programmed to the State Machine in each PE at the beginning of layer execution. The State Machine keeps track of the progress of the execution and communicates with the Sequencer after producing each output polynomial.



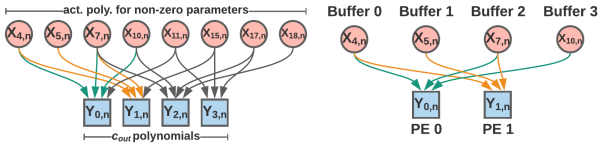
(a) Polynomial Buffer (b) PE Architecture

Fig. 7. Architecture of the Polynomial Buffer and the PE.

B. Polynomial Buffer Access Scheduling

The uneven distribution of sparse parameters poses challenges in terms of Poly Buffer access. Figure 8(a) shows an example of the irregular polynomial accesses to produce the n-th polynomials in the output ciphertext. Node $X_{i,n}$ represents an input polynomial ($0 \leq i < f^2 \cdot c_{in}$). Node $Y_{j,n}$ represents an output polynomial ($0 \leq j < c_{out}$). A connection from $X_{i,n}$ to $Y_{j,n}$ means the output j has a non-zero parameter at index i.

Figure 8(b) shows a *scheduling instance* of Figure 8(a) when mapping onto the proposed accelerator ($m = 2, k = 4$). We keep m output polynomials stationary in m PEs and process k input polynomials at a time. For every m output polynomials, there are up to $\lceil \frac{f^2 \cdot c_{in}}{k} \rceil$ scheduling instances. Given m parallel PEs, at worst, one Poly Buffer needs to send the coefficients to all m PEs in the same cycle, which violates the single access principle of URAM [31]. Therefore only one PE can access the buffer while other PEs have to be stalled and the stall duration can be as long as N/l cycles.



(a) Irregular Access of Polynomials (b) Mapping to Accelerator

Fig. 8. Graph representation of the irregular input polynomial access and its mapping to our accelerator.

A commonly used scheduling policy is to issue the Poly Buffer reads based on the index order of the non-zero parameters. We call this *Index Order* scheduling. However, this approach can lead to significant stalls when node $X_{i,n}$ has large degrees. We propose an algorithm to reduce pipeline stalls due to multiple PEs accessing the same Poly Buffer. The proposed scheduling algorithm is aimed to maximize the number of non-stalled PEs every cycle. The key idea of our scheduling algorithm is that we want to group the accesses from the PEs such that in each cycle they access different Poly Buffers. We formulate this scheduling algorithm as a **Maximum Bipartite Matching (MBM)** problem between node $X_{i,n}$ and node $Y_{j,n}$.

Algorithm 1: Maximum bipartite matching based Polynomial Buffer access scheduling

Input: Sparse parameters P.
Output: Set $S = \{s_1, s_2, ..., s_p, ...\}$ for scheduling.

1 **for** $i \leftarrow 0$ **to** c_{out} **by** m **do**
2 c_out_range of this iteration is from i to $i + m$
3 $indices \leftarrow$ get_unique_indices(P, c_out_range)
4 **for** $j \leftarrow 0$ **to** $len(indices)$ **by** k **do**
5 $input\_range$ of this iteration is from j to $j + k$
6 Construct bipartite graph B using P, c_out_range, $input\_range$
7 **while** *there are remaining edges in B* **do**
8 $s \leftarrow$ get_edges_max_bipartite_matching(B)
9 S.append(s)
10 Delete s from B
11 **end**
12 **end**
13 **end**

The proposed algorithm is shown in Algorithm 1. For every m output polynomials, we first find the input polynomials that have at least one non-zero parameter associated with it (Line 1–3). Only these input polynomials are streamed into the Poly Buffers. Each loop iteration in Line 4 works on a new scheduling instance, which can be viewed as a bipartite graph similar to Figure 8(b). We construct a unit scheduling set $s_p = \{(X_\alpha, Y_\beta), ..., (X_{\alpha'}, Y_{\beta'})\}$ that satisfies the following constraints: (1) Elements in s_p have distinct $X_{i,n}$ and $Y_{j,n}$ nodes. This implies that only one Poly Buffer is read by one PE during this unit of schedule. (2) For each pair of $(X_{i,n}, Y_{j,n})$ in s_p, there must exist an edge from $X_{i,n}$ to $Y_{j,n}$. (3) The length of pairs in s_p cannot exceed m. This reflects the fact that at most m PEs are busy. Each set s_p represents N/l cycles of read access from m PEs. Using MBM, we find a set that can cover most PEs, thereby achieving high parallelism. We find all the units of scheduling by iteratively running the MBM algorithm for a scheduling instance (Line 7–11). After each iteration, the matching edges are removed from the bipartite graph. A scheduling instance is fully scheduled when the bipartite graph B becomes empty. Then the algorithm moves to the next scheduling instance and repeats. The output of Algorithm 1, scheduling set $S = \{s_1, s_2, ..., s_p, ...\}$, contains the unit scheduling sets to produce $1 \cdot 1 \cdot c_{out}$ output ciphertext. Note that the $2 \cdot r$ polynomials within a ciphertext can reuse the same S. S is used by the Sequencer and the State Machines of the PEs to control the fetching of input polynomials and the order of access. For each trained HE CNN, this algorithm is only run once as a preprocessing step.

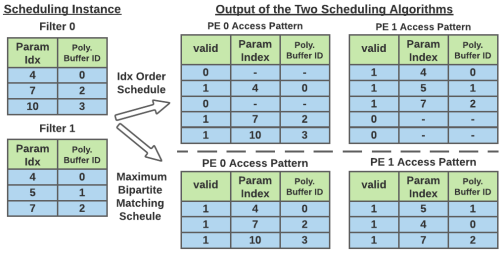


Fig. 9. Comparing the Index Order scheduling and the Maximum Bipartite Matching scheduling based on Figure 8(b).

Figure 9 shows the output of Index Order scheduling and MBM scheduling based on Figure 8(b). Each row in the output tables on the right represents a unit scheduling set. Only the rows with $valid = 1$ should be included by the scheduling set. For instance, the first unit scheduling set of the Index Order scheduling is $s_1 = \{(4, 1)\}$ while the MBM algorithm produces $s_1 = \{(4, 0), (5, 1)\}$. With Index Order scheduling, each PE is stalled 2 times, which can lead to $2 \cdot N/l$ stall cycles. Using the MBM scheduling, the PEs are not stalled during the execution of this convolution.

V. Experiments

A. Experimental Setup

We implement our design on a CPU-FPGA heterogeneous platform. The target platform has an AMD Ryzen 3990X CPU, which has 64 cores (128 threads) running at 2.9 GHz. A Xilinx U200 FPGA [31] is connected to the CPU via the PCI-E Gen3 x16 interface. The FPGA has 1182K LUTs, 2364K FFs, 35 MB on-chip SRAM and 6840 DSPs. 64 GB of DRAM is attached to the FPGA, providing a peak bandwidth of 77 GB/s. We partition the HE SCNN inference into two parts: the linear layers are accelerated on the FPGA and the activation layers are executed on the host CPU. Data is copied between CPU

and FPGA through the PCI-E interface. The FPGA accelerator is implemented using Vitis high-level synthesis v2020.2 [36]. The activation layers are implemented using Microsoft SEAL library v3.6 [30] and OpenMP v4.5.

Applications. We use CNN-6 and CNN-11, two widely studied HE optimized CNNs [3], [6], [17], [13], as the target models for benchmarking. They are pruned to 90% and 50% sparsity as described in Section III. The one-time setup cost for generating the MBM scheduling is 1.47s and 9.71s for CNN-6 and CNN-11 respectively. The inference is performed on batches of images with batch size of 8192.

TABLE III
ACCELERATOR ARCHITECTURE CONFIGURATIONS.

Config	Num PEs (m)	Num Poly. Buffers (k)	PE MAC Array Size (l)
Accel-L	8	8	8
Accel-M	4	4	8
Accel-S	2	2	8

Accelerator configurations. Table III summarizes the architecture configurations. We evaluate three configurations, ranging from Accel-L (Large), Accel-M (Medium) to Accel-S (Small). They are targeted at FPGA devices that have different resource availability. Previous study [37] showed that the effective DRAM bandwidth can be improved by increasing the DRAM burst length. Because MAC array size (l) is directly related to the DRAM burst length in our design, we choose $l = 8$ in all the configurations.

B. Accelerator Performance

1) Linear Layer Execution Time: We compare the execution time of Index Order scheduling and Maximum Bipartite Matching (MBM) scheduling. Figure 10 shows the execution time of various linear layers. The top 3 sub-figures show the performance of CNN-6 while the bottom 3 sub-figures present the performance of CNN-11. CNN-6 is a much easier task than CNN-11, therefore the execution time is much shorter. Comparing different accelerator configurations, Accel-L achieves significantly lower latency especially for layers that have high computational demand (e.g., conv1 and conv2 in CNN-11). The MBM scheduling algorithm gives a consistent performance improvement across all the linear layers, providing a speedup of up to 35%. The improvement from the MBM algorithm increases in accelerators with more PEs and Polynomial Buffers. The reason behind this gap is that it is more difficult to find the same number of matching edges as m and k decrease.

Pooling layer is intrinsically memory bound as there is no data reuse. Thus, the execution time is mostly determined by the data transfer from FPGA DRAM to the accelerator. Our accelerators achieve 1.4s, 0.6s and 0.3s for pool0, pool1 and pool2 layers in CNN-11 respectively.

2) End-to-end Inference: The computation of the activation functions are performed on the CPU. We exploit thread-level parallelism (OpenMP) by partitioning the activation tensor across different CPU threads. Data is copied between FPGA DRAM and host DRAM before and after each activation

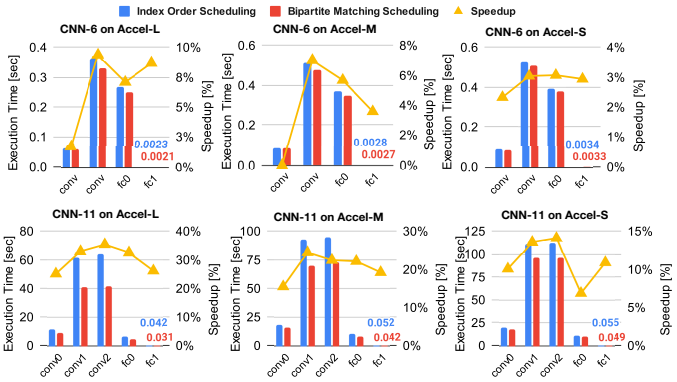


Fig. 10. Comparing the scheduling algorithms for conv and fc layers in CNN-6 and CNN-11.

layer. The end-to-end inference time breakdown on Accel-L is shown in Table IV. The results of Accel-M and Accel-S have similar trend. For CNN-11, the linear layers are still the primary bottleneck after acceleration, taking 80% of the run time. CNN-6 has less computational demand in linear layers. Activation layers and linear layers have similar run time *after acceleration*.

TABLE IV
INFERENCE TIME BREAKDOWN OF CNN-6 AND CNN-11 ON ACCEL-L.

	CNN-6	CNN-11
Linear Layers	0.639s [42.2%]	98.384s [80.1%]
Activation Layers	0.734s [48.5%]	21.657s [17.6%]
CPU/FPGA Data Transfer	0.140s [9.3%]	2.771s [2.3%]
Latency (8K images)	1.513s	122.813s

To better understand the performance difference between FPGA and CPU, we implemented the linear layers of CNN-6 and CNN-11 on the CPU using OpenMP (64 threads). For a fair comparison, we use two CPU baselines: one performs CNN-6 and CNN-11 inference without pruning, another uses the same sparse models as the CPU-FPGA design. Figure 11 summarizes the speedups offered by Accel-L, using the CPU dense implementation as the baseline (1×). Accel-L consistently outperforms the two CPU implementations across all the layers. Compared to the CPU dense (**sparse**) implementations, Accel-L achieves a network level speedup of 9.5× (**2.1**×) and 9.9× (**5.6**×) for CNN-6 and CNN-11 respectively. The speedup offered by the Index Order scheduling based designs demonstrates the effectiveness of the specially designed dataflow for HE SCNN.

3) Sensitivity to CNN Sparsity: Apart from the sparse parameters obtained from CNN-6 and CNN-11, it is also important that the scheduling algorithm can perform efficiently for other sparsity patterns. Similar to [33], we randomly generate the parameters of the conv and fc layers in CNN-11 at different sparsity levels. Figure 12 shows the results on Accel-L. On the x-axis we sweep the sparsity from 10% to 70%, on the y-axis is the total inference time of CNN-11. The MBM scheduling algorithm consistently outperforms the Index Order scheduling. The improvement increases as the sparsity level drops, reaching a 60% improvement at 10% sparsity. At higher sparsity levels, there are less Poly Buffer access conflicts to begin with, thus the benefit of MBM scheduling

is less pronounced.

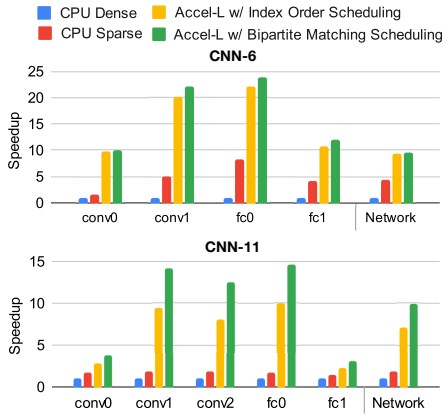


Fig. 11. Conv, fc and network speedup using CPU dense implementation as the baseline.

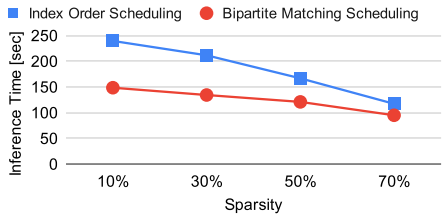


Fig. 12. Inference latency of CNN-11 with various sparsity.

C. Accelerator Resource Consumption

The resource utilization of Accel-L, Accel-M and Accel-S are listed in Table V. URAMs are heavily used in our design to store the polynomials in a double buffered manner. Due to the modular arithmetic, each MAC unit in the MAC array uses 36 DSPs, which is significantly greater than the MAC unit for unencrypted computation. The fully-connected crossbar brings additional complexity to the design, which can incur frequency drop with larger number of PEs.

TABLE V
ACCELERATOR RESOURCE CONSUMPTION ON FPGA.

	LUT	FF	BRAM	URAM	DSP	Freq.
Accel-L	360K	424K	698	264	2320	175 MHz
Accel-M	287K	406K	622	132	1168	200 MHz
Accel-S	265K	391K	585	66	592	250 MHz

D. Comparison with State-of-the-art

Multicore CPU. CryptoNets [3] proposed the first CNN using HE. It performs inference on a batch of 8K images and achieves a latency of 570s for dense CNN-6. CryptoDL [6] reduces the dense CNN-6 latency to 139.6s. A CNN model that has similar architecture to dense CNN-11 is also evaluated by CryptoDL and the inference latency is 11,686s. Faster CryptoNets (FCryptoNets) uses pruning and quantization to reduce HE operations. It does not use SIMD packing and thus the effective batch size is 1. FCryptoNets achieves a latency of 39.2s (22,372s) per image for sparse CNN-6 (CNN-11). NGraph-HE2 [19] improves the dense CNN-6 latency to 2.05s using a batch of 4K images. **GPU.** HCNN [17] uses NVIDIA V100 GPU to accelerate HE CNN inference on a batch of 8K images. HCNN obtained an inference latency of 5.1s and 304s for dense CNN-6 and CNN-11 respectively.

To the best of our knowledge, our paper is the first attempt to study the FPGA acceleration of HE CNN with batch inference. The evaluation results demonstrate that the proposed design can achieve a low latency of 1.5s and 122.8s for sparse CNN-6 and CNN-11 on the target CPU-FPGA platform.

VI. Related Work

CryptoNets [3] was the first work to enable HE CNN inference. CryptoDL [6] further explored using polynomial approximation as activations to enable deeper CNNs (e.g., CNN-11). HCNN [17] implemented the first GPU acceleration of batch inference of HE CNN [3]. Cheetah [5] and [38] focus on the acceleration of linear layers for *single image inference* with a different HE encoding scheme from this paper. Their designs accelerate dense CNNs, applying their designs to sparse CNNs will result in significant overhead as unnecessary computations are not skipped. F1 [39] proposed ASIC acceleration of BGV scheme. Their design was evaluated on a simulator assuming extremely high memory bandwidth. Prior FPGA implementations [40], [41], [42] accelerate HE primitives (multiplication/addition) and do not consider the data layout and reuse in HE CNN. Naively using these designs for HE SCNN inference will result in excessive off-chip data transfers as the data reuse is not exploited. CryptoPIM [43] accelerates NTT and does not provide the design of other HE operations that are required for end-to-end inference. Tian et al. [44] proposed analytical models of FPGA accelerated HE CNN, while many implementation details are ignored. In contrast, we propose a practical implementation. Faster CryptoNets [13] was the first work that combines sparsity with HE CNN. Although using an inefficient encoding scheme and targeting at CPU platform, this work shows the viability of leveraging sparsity to speedup HE CNN inference.

VII. Conclusion

In this paper, we performed a thorough analysis to show the bottlenecks and data reuse opportunities of the linear layers in HE Sparse CNN (SCNN) batch inference. Motivated by the analysis, we proposed a novel dataflow by rearranging the processing order of the polynomials in the ciphertext. This dataflow improves the data reuse of input ciphertext and reduces off-chip data transfers. To mitigate the pipeline stalls caused by sparse input polynomial accesses, we designed a maximum bipartite matching based algorithm to enable efficient access of polynomials from parallel PEs. We adopted the proposed dataflow and scheduling algorithm to design an FPGA accelerator that can be programmed to process various linear layers in HE SCNN. We demonstrated end-to-end HE SCNN batch inference on a CPU-FPGA heterogeneous platform. Compared to CPU-only solutions, our accelerator achieves up to 5.6× speedup in inference latency on widely studied 6-layer and 11-layer HE CNNs.

VIII. Acknowledgement

This work has been sponsored by the U.S. National Science Foundation under grant number SaTC-2104264. Equipment grant by Xilinx is greatly appreciated.

REFERENCES

[1] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*. Association for Computing Machinery, 2009.

[2] M. Albrecht, M. Chase, H. Chen, and et al, "Homomorphic encryption security standard," Tech. Rep., 2018.

[3] 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," ser. ICML'16.

[4] P. Mishra, R. Lehmkuhl, A. Srinivasan, W. Zheng, and R. A. Popa, "Delphi: A cryptographic inference service for neural networks," in *29th USENIX Security Symposium (USENIX Security 20)*.

[5] B. Reagen and et al, "Cheetah: Optimizing and accelerating homomorphic encryption for private inference," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021.

[6] E. Hesamifard, H. Takabi, and M. Ghasemi, "Cryptodl: towards deep learning over encrypted data," in *Annual Computer Security Applications Conference (ACSAC 2016), Los Angeles, California, USA*.

[7] K. Garimella, N. K. Jha, Z. Ghodsi, S. Garg, and B. Reagen, "Cryptonite: Revealing the pitfalls of end-to-end private inference at scale," 2021.

[8] M. Zhu and S. Gupta, "To prune, or not to prune: Exploring the efficacy of pruning for model compression," in *6th International Conference on Learning Representations*, 2018.

[9] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," in *4th International Conference on Learning Representations*, Y. Bengio and Y. LeCun, Eds., 2016.

[10] M. Zhu, T. Zhang, Z. Gu, and Y. Xie, "Sparse tensor core: Algorithm and hardware co-design for vector-wise sparse neural networks on modern gpus," ser. MICRO '52, 2019.

[11] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: Efficient inference engine on compressed deep neural network," in *ISCA'2016*.

[12] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[13] E. Chou, J. Beal, D. Levy, S. Yeung, A. Haque, and L. Fei-Fei, "Faster cryptonets: Leveraging sparsity for real-world encrypted inference," 2018.

[14] J.-C. Bajard, J. Eynard, M. A. Hasan, and V. Zucca, "A full rns variant of fv like somewhat homomorphic encryption schemes," in *Selected Areas in Cryptography – SAC 2016*, R. Avanzi and H. Heys, Eds., 2017.

[15] 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.

[16] C. Gentry, "A fully homomorphic encryption scheme," 2009, phD Dissertation 2009.

[17] A. QaisarAhmadAlBadawi, J. Chao, and et al, "Hcnn, the first homomorphic cnn on encrypted data with gpus," *IEEE Transactions on Emerging Topics in Computing*.

[18] 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.

[19] F. Boemer and et al, "Ngraph-he2: A high-throughput framework for neural network inference on encrypted data," in *Proceedings of the 7th ACM Workshop on Encrypted Computing and Applied Homomorphic Cryptography*, ser. WAHC'19, 2019.

[20] T. Gale and et al, "Sisyphus: A cautionary tale of using low-degree polynomial activations in privacy-preserving deep learning," in *2021 ACM Conference on Computer and Communications Security (Workshop)*, 2021.

[21] F. Boemer, R. Cammarota, D. Demmler, T. Schneider, and H. Yalame, "Mp2ml: A mixed-protocol machine learning framework for private inference," ser. PPMLP'20. New York, NY, USA: Association for Computing Machinery, 2020.

[22] F. Boemer, Y. Lao, R. Cammarota, and C. Wierzynski, "Ngraph-he: A graph compiler for deep learning on homomorphically encrypted data," in *Proceedings of the 16th ACM International Conference on Computing Frontiers*, ser. CF '19, 2019.

[23] R. Dathathri, B. Kostova, O. Saarikivi, W. Dai, K. Laine, and M. Musuvathi, "Eva: An encrypted vector arithmetic language and compiler for efficient homomorphic computation," ser. PLDI 2020, 2020.

[24] R. Dathathri, O. Saarikivi, H. Chen, K. Laine, K. Lauter, S. Maleki, M. Musuvathi, and T. Mytkowicz, "Chet: An optimizing compiler for fully-homomorphic neural-network inferencing," ser. PLDI 2019, 2019.

[25] A. Brutzkus, R. Gilad Bachrach, and O. Elisha, "Low latency privacy preserving inference," in *Proceedings of the 36th International Conference on Machine Learning*, 2019.

[26] N. Tajbakhsh, J. Y. Shin, S. R. Gurudu, R. T. Hurst, C. B. Kendall, M. B. Gotway, and J. Liang, "Convolutional neural networks for medical image analysis: Full training or fine tuning?" *IEEE transactions on medical imaging*, May 2016.

[27] D. Syed, S. S. Refaat, and O. Bouhali, "Privacy preservation of data-driven models in smart grids using homomorphic encryption," *Information*, vol. 11, no. 7, p. 357, 2020.

[28] R. Miotto, F. Wang, S. Wang, X. Jiang, and J. T. Dudley, "Deep learning for healthcare: review, opportunities and challenges," *Briefings in bioinformatics*, vol. 19, no. 6, pp. 1236–1246, 2018.

[29] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," 2014.

[30] "Microsoft SEAL (release 3.6)," https://github.com/Microsoft/SEAL, Nov. 2020, microsoft Research, Redmond, WA.

[31] Xilinx, "Xilinx UltraScale+ FPGAs," https://www.xilinx.com/products/boards-and-kits/alveo/u200.html.

[32] Intel, "Stratix 10 MX FPGAs," https://www.intel.com/content/www/us/en/products/programmable/sip/stratix-10-mx.html.

[33] A. Parashar, M. Rhu, and et al, "Scnn: An accelerator for compressed-sparse convolutional neural networks," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017.

[34] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: Shifting vision processing closer to the sensor," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015, pp. 92–104.

[35] D. Pei, A. Salomaa, and C. Ding, *Chinese remainder theorem: applications in computing, coding, cryptography*. World Scientific, 1996.

[36] Xilinx, "Xilinx Vitis Development Platform," https://www.xilinx.com/products/design-tools/vitis.html.

[37] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," ser. FPGA '15.

[38] T. Ye, S. Kuppannagari, R. Kannan, and V. Prasanna, "Performance modeling and fpga acceleration of homorphic encrypted convolution," in *2021 International Conference on Field Programmable Logic and Applications (FPL)*, 2021.

[39] N. Samardzic, A. Feldmann, A. Krastev, S. Devadas, R. Dreslinski, C. Peikert, and D. Sanchez, *F1: A Fast and Programmable Accelerator for Fully Homomorphic Encryption*. New York, NY, USA: Association for Computing Machinery, 2021.

[40] S. Sinha Roy, F. Turan, K. Jarvinen, F. Vercauteren, and I. Verbauwhede, "Fpga-based high-performance parallel architecture for homomorphic computing on encrypted data," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019.

[41] M. S. Riazi, K. Laine, B. Pelton, and W. Dai, "Heax: An architecture for computing on encrypted data," ser. ASPLOS '20.

[42] V. Migliore, C. Seguin, M. M. Real, V. Lapotre, A. Tisserand, C. Fontaine, G. Gogniat, and R. Tessier, "A high-speed accelerator for homomorphic encryption using the karatsuba algorithm," *ACM Trans. Embed. Comput. Syst.*, vol. 16, no. 5s, 2017.

[43] H. Nejatollahi, S. Gupta, M. Imani, T. S. Rosing, R. Cammarota, and N. Dutt, "Cryptopim: In-memory acceleration for lattice-based cryptographic hardware," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020.

[44] 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)*, 2020.