Bandwidth Efficient Homomorphic Encrypted Matrix Vector Multiplication Accelerator on FPGA

Yang Yang*, Sanmukh R. Kuppannagari†§, Rajgopal Kannan‡ and Viktor K. Prasanna*
* Department of Electrical and Computer Engineering, University of Southern California
† Department of Computer and Data Sciences, Case Western Reserve University
‡ DEVCOM US Army Research Lab
Email: {yyang172, prasanna}@usc.edu, sanmukh.kuppannagari@case.edu, rajgopal.kannan.civ@army.mil

2022 International Conference on Field-Programmable Technology (ICFPT) | 978-1-6654-5336-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICFPT56656.2022.9974369

***Abstract*—Homomorphic Encryption (HE) is a promising solution to the increasing concerns of privacy in Machine Learning (ML) as it enables computations directly on encrypted data. However, it imposes significant overhead on the compute system and remains impractically slow. Prior works have proposed efficient FPGA implementations of basic HE primitives such as number theoretic transform (NTT), key switching, etc. Composing the primitives together to realize higher level ML computation is still a challenge due to the large data transfer overhead.**

In this work, we propose an efficient FPGA implementation of HE Matrix Vector Multiplication (M×V), a key kernel in HE-based Machine Learning applications. By analyzing the data reuse characteristics and the encryption overhead of HE M×V, we show that simply using the principles of unencrypted M×V to design accelerators for HE M×V can lead to a significant amount of DRAM data transfers. We tackle the computation and data transfer challenges by proposing a bandwidth efficient dataflow that is specially optimized for HE M×V. We identify highly reused data entities in HE M×V and efficiently utilize the on-chip SRAM to reduce the DRAM data transfers. To speed up the computation of HE M×V, we exploit three types of parallelism: partial sum parallelism, residual polynomial parallelism and coefficient parallelism. Leveraging these innovations, we demonstrate the first FPGA accelerator for HE matrix vector multiplication. Evaluation on 7 HE M×V benchmarks shows that our FPGA accelerator is up to $3.8\times$ (GeoMean $2.8\times$) faster compared to the 64-thread CPU implementation.

***Index Terms*—FPGA acceleration, homomorphic encryption, matrix vector multiplication, parallel computing**

I. Introduction

Homomorphic Encryption (HE) provides a promising solution for implementing privacy-preserving Neural Network (NN) inference by allowing *direct computations* on encrypted data [1]. With HE, the client provides encrypted data to a cloud server, on which the computation is performed *without* decrypting the data. While HE offers strong privacy guarantees for NN inference in public cloud [2], [3], it comes at a high cost: inference using HE NN is orders of magnitude slower than inference on unencrypted data [4].

Matrix-Vector multiplication (M×V) is a basic building block in a wide range of neural networks (NN), especially in single batch (batch size $= 1$) NN inference. Fully-connected (FC) layers in a Convolutional Neural Network (CNN) are implemented with M×V [5]. In Recurrent Neural Network (RNN), M×V operations are performed on the input and the cell state at each time step [6], [7]. Such CNN and RNN models are ubiquitous in image classification, image segmentation and natural language processing [8], [9].

The hardware acceleration of unencrypted M×V is straightforward. The input and output vectors are typically stored on-chip to maximize data reuse while the input matrix is streamed. However, homomorphic encryption (HE), including HE M×V, imposes a significant overhead on the memory bandwidth [10] and has completely different computation characteristics. Figure 1 shows the external DRAM data transfer breakdown (assuming no on-chip SRAM) for an HE M×V of $1K \times 1K$. The input vector constitutes only 4% of the total data transfer while the majority of the data transfer is dominated by HE specific data entities such as the NTT twiddle factors and the partial sum ciphertext. Therefore the acceleration techniques for unencrypted M×V are not suited for HE M×V.

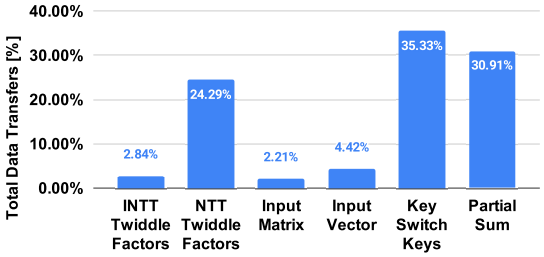

Fig. 1. HE M×V data transfer breakdown.

Several works have been proposed to accelerate HE, but they either only focus on the basic primitives [11], [12], [13] or do not address the memory challenges of HE M×V [14], [15]. HEAX [11] and the design from Roy et al [12] developed efficient HE primitives on FPGA, composing them together to realize higher level applications such as M×V is challenging. Naively chaining the implementations of basic primitives from prior works will lead to sub-optimal designs as global data reuse optimizations is overlooked at primitive-level. Gazelle [14] implements HE M×V on a multi-core CPU. Their implementation simply processes the data on-demand using caches and does not explicitly explore the data reuse opportunities of various HE data entities.

To the best of our knowledge, we develop the first FPGA accelerator for Homomorphic Encrypted Matrix Vector Multiplication (HE M×V). To reduce external DRAM data transfers, we utilize the FPGA on-chip SRAMs to only store highly

§This work was done while the author was with USC.

reused HE data entities. We further identify and exploit various parallelization dimensions to speedup the HE computation.

Specifically, the key contributions of this paper are:

- By analyzing the access pattern and the data reuse of HE M×V primitives, we show that the compute and memory characteristics of HE M×V is completely different from its unencrypted counterpart and requires a different acceleration strategy.
- Motivated by the analysis, we propose a bandwidth efficient dataflow. We tackle the significantly increased DRAM data transfer overhead by only storing the highly reused HE data entities on-chip. We identify three dimensions of parallelism to speedup the computation of HE M×V: partial sum parallelism, residual polynomial parallelism and coefficient parallelism.
- Based on the proposed memory and compute optimizations, we design the first FPGA accelerator for HE M×V. The accelerator is parameterized and can support various HE parameters.
- Experimental results on 7 HE NN benchmarks including CNNs and NLPs show that our design achieves up to $3.8\times$ (GeoMean $2.8\times$) speedup compared to the 64-thread CPU implementation.

II. Background

A. Threat Model

The threat model this paper assumes is similar to that of prior works [2], [16], [17], [18]. We assume a client-server model where the client sends an encrypted input vector to the server to perform HE M×V. The server owns the unencrypted matrix M and performs M×V on the encrypted input vector, without decrypting it. The encrypted result is returned to the client after computation.

B. Homomorphic Encryption

In this paper, we use the CKKS scheme [19]. Cleartext is first encoded as a plaintext polynomial (*pt*), which is then encrypted as a pair of ciphertext polynomials (*ct*). Using Single Instruction Multiple Data (SIMD) packing, the CKKS scheme encodes and encrypts a *vector* of $N/2$ complex numbers in $N/2$ slots of a plaintext and ciphertext, where N is a power-of-two number that defines the degree of plaintext and ciphertext polynomials. With SIMD packing, one HE operation is simultaneously applied on every slot of the plaintext or ciphertext. The coefficients of the plaintext and ciphertext are represented modulo q. Let $\mathcal{R} = \mathbb{Z}[X]/(X^N+1)$, a plaintext is a polynomial in the ring $\mathcal{R}_q = \mathcal{R}/q\mathcal{R}$ with coefficients from $\mathbb{Z}_q$, i.e. integers modulo q. The ciphertext space is $\mathcal{R}_q^2 = (\mathcal{R}/q\mathcal{R})^2$, which means a pair of polynomials with coefficients from $\mathbb{Z}_q$. CKKS requires ciphertext modulus q to be hundreds of bits depending on the multiplicative depth of the function to be evaluated [20], which is expensive to process. The Residue Number System (RNS) [19] enables representing a ciphertext polynomial with $\log q$-bit coefficients as multiple polynomials with narrower coefficients. Let q be a product of l co-primes $q = \prod_{i=1}^{l} p_i$. A polynomial in $\mathcal{R}_q$ can be represented as l polynomials, where the coefficients of the i-th polynomial is from $\mathbb{Z}_{p_i}$. l is also referred as the level of the ciphertext. The encryption noise is accumulated over each homomorphic operation [21]. When the noise grows beyond a noise budget, decryption becomes impossible. Bootstrapping can reset the noise and enable Fully Homomorphic Encryption computation (i.e., unlimited number of HE operations). As the number of HE operations is a known apriori in HE M×V, bootstrapping can be avoided in accordance with several other works [2], [3], [4], [16], [22].

Notation. Throughout the paper, we use normal case letters to denote integers, e.g., p_i. Polynomials and vectors are written in bold, e.g., $\mathbf{u}$. Vectors of polynomials and matrices are denoted in uppercase bold, e.g. the plaintext of vector $\mathbf{u}$ is $\mathbf{U}$. $[\cdot]$ denotes the homomorphic encryption of a vector. We use superscripts to represent the two components of a ciphertext when needed, e.g., $[\mathbf{u}] = (\mathbf{U}^0, \mathbf{U}^1)$. We use subscripts to represent the indices, e.g. $\mathbf{U}_i^0$ is the i-th polynomial of the first component of ciphertext $[\mathbf{u}]$.

C. Primitive Operations of CKKS.

ct-ct add performs addition between two encrypted vectors $[\mathbf{u}]$, $[\mathbf{v}]$ and outputs the encryption of the element-wise sum of the two vectors, $[\mathbf{u} + \mathbf{v}]$. pt-ct mult multiplies a plaintext $\mathbf{U}$ with a ciphertext $[\mathbf{v}]$ and outputs $[\mathbf{u} \circ \mathbf{v}]$, where $\circ$ denotes element-wise multiplication of the two vectors. ct-ct mult consists of multiple steps, including polynomial multiplications and relinearization. As the matrix is not encrypted in our setting, ct-ct mult is not needed. We therefore omit the details and refer the reader to [19]. To avoid noise overflow, a rescaling operation is performed after each HE multiplication. Rescaling involves Number Theoretic Transform (NTT), Inverse-NTT (INTT) and element-wise operations, as shown in Algorithm 1. This operation takes the RNS and NTT form of a ciphertext as input and reduces its level from l to $l-1$.

Algorithm 1: CKKS Rescaling

Input: $[\mathbf{c}] = (\mathbf{C}^0, \mathbf{C}^1) \in (\prod_{i=1}^{l} \mathbb{Z}_{p_i}^N)^2$.
Output: $[\tilde{\mathbf{c}}] = (\widetilde{\mathbf{C}}^0, \widetilde{\mathbf{C}}^1) \in (\prod_{i=1}^{l-1} \mathbb{Z}_{p_i}^N)^2$.

for $k \leftarrow 0$ **to** 1 **do**
  $\mathbf{a} \leftarrow \mathsf{INTT}(\mathbf{C}_l^k, p_l)$
  for $i \leftarrow 1$ **to** $l-1$ **do**
    $\mathbf{a}' \leftarrow \mathsf{NTT}(\mathbf{a}, p_i)$
    $\widetilde{\mathbf{C}}_i^k \leftarrow \mathsf{Mod}(p_l^{-1} \cdot (\mathbf{C}_i^k - \mathbf{a}'), p_i)$
  end
end

Let $\mathbf{u} = (u_0, u_1, u_2, ..., u_{N-1})$, ct rotation outputs $[\mathbf{u}'] = [(u_k, u_{k+1}, ..., u_{N-1}, u_0, u_1, ..., u_{k-1})]$, namely the elements of $\mathbf{u}$ are circularly shifted by k slots. To perform rotation homomorphically, one first computes an *automorphism* on each residual polynomial. Automorphism moves coefficients of a polynomial via a mapping $i \mapsto \sigma_k(i)$, where i is the index of coefficient c_i and $\sigma_k(i)$ is defined as

Algorithm 2: CKKS Key Switching

Input: $[\mathbf{c}] = (\mathbf{C}^0, \mathbf{C}^1) \in (\prod_{i=1}^{l} \mathbb{Z}_{p_i}^N)^2$,
ksk0 $\in (\prod_{j=1}^{l+1} \mathbb{Z}_{p_j}^N)^l$, ksk1 $\in (\prod_{j=1}^{l+1} \mathbb{Z}_{p_j}^N)^l$.
Output: $[\tilde{\mathbf{c}}] \in (\prod_{i=1}^{l} \mathbb{Z}_{p_i}^N)^2$.

for $i \leftarrow 1$ **to** l **do**
  $\mathbf{a} \leftarrow \mathsf{INTT}(\mathbf{C}_i^1, p_i)$
  for $j \leftarrow 1$ **to** l **do**
    $\mathbf{a}' \leftarrow (i == j)\ ?\ \mathbf{C}_i^1 : \mathsf{NTT}(\mathbf{a}, p_j)$
    $\widetilde{\mathbf{C}}_j^0 \leftarrow \mathsf{Mod}(\widetilde{\mathbf{C}}_j^0 + \mathbf{a}' \cdot \mathrm{ksk0}[i,j], p_j)$
    $\widetilde{\mathbf{C}}_j^1 \leftarrow \mathsf{Mod}(\widetilde{\mathbf{C}}_j^1 + \mathbf{a}' \cdot \mathrm{ksk1}[i,j], p_j)$
  end
  $\mathbf{b} \leftarrow \mathsf{NTT}(\mathbf{a}, p_{l+1})$
  $\widetilde{\mathbf{C}}_{l+1}^0 \leftarrow \mathsf{Mod}(\widetilde{\mathbf{C}}_{l+1}^0 + \mathbf{b} \cdot \mathrm{ksk0}[i, l+1], p_{l+1})$
  $\widetilde{\mathbf{C}}_{l+1}^1 \leftarrow \mathsf{Mod}(\widetilde{\mathbf{C}}_{l+1}^1 + \mathbf{b} \cdot \mathrm{ksk1}[i, l+1], p_{l+1})$
end
$[\tilde{\mathbf{c}}] \leftarrow \mathsf{Rescale}((\widetilde{\mathbf{C}}^0, \widetilde{\mathbf{C}}^1), p_{l+1})$
$[\tilde{\mathbf{c}}] \leftarrow \mathsf{Add}([\mathbf{c}], [\tilde{\mathbf{c}}])$

$$\sigma_k(i) = i \cdot 5^k \mod N, i \in 0, 1, ..., N-1$$

After the automorphism, the ciphertext is encrypted under a different secret key, requiring a *key switch* operation to produce the output of rotation (see Algorithm 2).

D. Homomorphic Encrypted M×V

HE M×V is performed between an unencrypted matrix $\mathbf{M}$ of shape $m \times n$ and an encrypted vector $\mathbf{v}$ of length n. For simplicity, we assume $n < N/2$ in the following analysis. The client packs all the elements of $\mathbf{v}$ into $\lceil 2n/N \rceil$ plaintext, then encrypts them into ciphertext. To efficiently encode the matrix into plaintext, we use the state-of-the-art method which combines row-based packing and diagonal packing [14], [15]. In this method, $\mathbf{M}$ is represented by m vectors, where each vector has n elements. The n elements in each vector are selected along the extended diagonals such that they have distinct indices in the column dimension of the matrix. These plaintext vectors are multiplied with m rotations of $\mathbf{v}$. The multiplication produces ciphertext that has k chunks of the partial sum, where $k = 1$ if $m \geq n$, and $k = \lfloor N/2m \rfloor$ otherwise. Each chunk has m partial sums corresponding to the m outputs. The partial sums that contribute to the same output within a ciphertext (when $k \geq 1$) are accumulated using the rotate and sum algorithm [14], [15].

Figure 2 uses a matrix of size 4×8 to illustrate the steps to compute HE M×V. Each element in the matrix is denoted by its row and column indices. The matrix is packed into four plaintext, shown as *pt* in the figure. Each plaintext has two extended diagonals ($k = 2$). The first step ❶ of the computation is the pt-ct mult between the plaintext vector and the rotated ciphertext vector. Each pt-ct mult outputs a ciphertext that encrypts a vector that contains two chunks of partial sums, each of which has 4 partial sums corresponding to the 4 outputs. Next ❷, we need to rotate and accumulate the partial sums within the same ciphertext. This step involves $\lceil \log k \rceil = 1$ ct rotation and ct-ct add operations. Finally ❸,

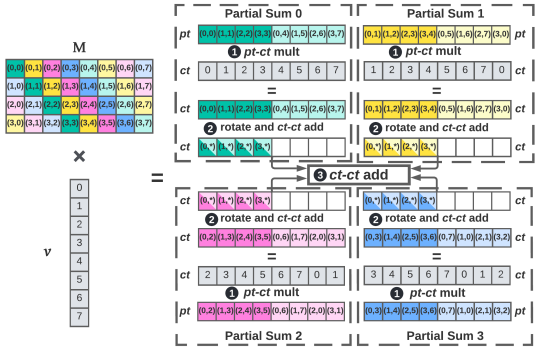

Fig. 2. HE matrix vector multiplication.

the 4 partial sum ciphertext are accumulated to produce the final output.

III. HE M×V Algorithmic Analysis and Dataflow

HE M×V incurs significant overhead in compute and memory requirements due to the polynomial representation and modular arithmetic. We tackle these challenges by identifying the reuse of various data entities in HE M×V and devise an efficient dataflow given the limited DSP and SRAM resources.

A. Memory Requirement

A ciphertext is composed of $2 \cdot l$ degree-$(N-1)$ polynomials. Each coefficient in the polynomials is a $\log p_i$ bits integer. The size (bytes) of a ciphertext is

$$\mathcal{B}_{ct} = 2 \cdot l \cdot N \cdot bytes_{coefficient} \tag{1}$$

Because each ciphertext can encode a vector of size $N/2$, the size of the encrypted input vector with length n is

$$\mathcal{B}_v = \lceil 2n/N \rceil \cdot \mathcal{B}_{ct} \tag{2}$$

Each plaintext is one half of the size of a ciphertext

$$\mathcal{B}_{pt} = l \cdot N \cdot bytes_{coefficient} \tag{3}$$

We use $m \cdot \lceil 2n/N \rceil$ plaintext to represent the unencrypted matrix $\mathbf{M}$. The size of the encoded matrix is

$$\mathcal{B}_{\mathbf{M}} = m \cdot \lceil 2n/N \rceil \cdot \mathcal{B}_{pt} \tag{4}$$

The NTT and INTT operations in the rescaling and key switching operations require additional twiddle factors, which is proportional to l. The size of the twiddle factors is

$$\mathcal{B}_{ntt\_tf} = \mathcal{B}_{intt\_tf} = l \cdot N \cdot bytes_{coefficient} \tag{5}$$

INTT operations require a different set of twiddle factors but the total size is the same as the NTT. Finally, the key switch keys for the rotation are stored as a matrix of polynomials. The size is

$$\mathcal{B}_{ksk0} = \mathcal{B}_{ksk1} = l \cdot (l+1) \cdot N \cdot bytes_{coefficient} \tag{6}$$

Note that automorphism with a different rotation distance requires a different set of keys.

B. Memory Optimization: Data Reuse

As shown in the previous analysis, data movement is the largest bottleneck in HE computations. Understanding the temporal data reuse patterns of various data entities in HE M×V is essential to effectively allocate the on-chip SRAM resources and reduce external DRAM transfers. We introduce an analytical cost model to evaluate the total DRAM transfers by storing various data entities data on-chip.

We define the *cost* of an HE data entity as the total amount of DRAM transfers due to reading the data during the entire HE M×V computation. For simplicity, we consider $\lceil 2n/N \rceil = 1$ in the following analysis. The twiddle factors are highly reused data entities in HE M×V due to the frequent transformations between NTT domain and coefficient domain in HE primitives. To compute one partial sum ciphertext, the NTT twiddle factors are reused by $(l+4)$ times: each Rescale operation reuses the NTT twiddle factors twice and there are 2 Rescale ops per partial sum computation; the key switching operation reuses the NTT twiddle factors l times (Line 1 in Algorithm 2). There are m partial sum ciphertext. Thus the total data transfer cost of NTT twiddle factors is

$$\mathcal{C}_{tf\_ntt} = m \cdot \mathcal{B}_{tf\_ntt} \cdot (l+4) \tag{7}$$

The INTT twiddle factors are reused a lot less than the NTT ones. Rescale operation performs INTT for the last residual polynomial. Key switching operation uses the INTT twiddle factors once. The cost of INTT twiddle factor data transfers is

$$\mathcal{C}_{tf\_intt} = m \cdot \mathcal{B}_{tf\_intt} \cdot (1 + 2/l) \tag{8}$$

Similarly, the input vector ciphertext is reused m times.

$$\mathcal{C}_{in\_vector} = m \cdot \mathcal{B}_{ct} \tag{9}$$

The partial sum ciphertext (Line 6-7 in Algorithm 2) is also reused frequently. The cost is

$$\mathcal{C}_{partial\_sum} = m \cdot \mathcal{B}_{ct} \cdot l \tag{10}$$

There is no reuse of the input plaintext matrix. Each partial sum ciphertext is rotated $1 + \lceil \log k \rceil$ times, where the first term is to align the SIMD slots of the partial sum ciphertext with the output ciphertext and k is the number of partial sum chunks as defined in Section II-D. The rotation distances for the accumulation of k chunks are the same across various partial sum ciphertext, therefore the key switch keys for the $\lceil \log k \rceil$ rotations can be reused. The cost of the input matrix and the key switch keys are:

$$\mathcal{C}_{in\_matrix} = m \cdot \mathcal{B}_{pt} \tag{11}$$

$$\mathcal{C}_{ksk} = (\mathcal{B}_{ksk0} + \mathcal{B}_{ksk1}) \cdot (m + \lceil \log k \rceil) \tag{12}$$

We quantitatively analyze the impact of DRAM data transfers and the minimal required on-chip SRAM capacity. We use an HE M×V of size $m = 1024, n = 1024$ in the analysis while other problem sizes follow similar trend. We select $N = 2^{14}$ and $l = 7$ to achieve 128-bit security. Figure 3 shows the analysis result. The bar chart shows the cost and the breakdown (only read traffic is considered for the sake of simplicity). From left to right, we assume that additional data entities can be stored on-chip completely. The line chart shows the minimal required on-chip SRAM capacity. The result indicates that the NTT twiddle factors and partial sum ciphertext are among the most costly data entities. While there is a decent amount of reuse of the input vector and the INTT twiddle factors, the total data transfers that are incurred by them is relatively low. *The data reuse pattern of HE M×V is significantly different from unencrypted M×V in which the input and output vectors are the main sources of data reuse.* In terms of the SRAM capacity, we would need close to 6 MB in order to minimize the DRAM data transfers.

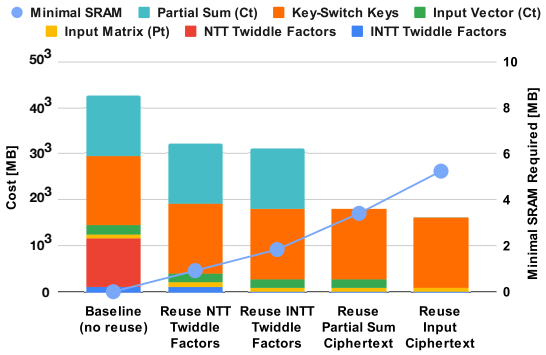

Fig. 3. Reuse of various data entities in HE M×V.

C. Compute Optimization: Parallelization

Due to the polynomial representation and the modular arithmetic in HE, the total number of operations of HE M×V is significantly increased [10]. Parallelizing the computation is necessary to avoid compute bottlenecks that impact latency. We identify three parallelization dimensions from various HE M×V primitives. We select these parallelization dimensions such that each one operates on a different abstraction level, from ciphertext (highest level) to coefficients in a residual polynomial (lowest level), and can be easily mapped to a parameterized hardware accelerator (Section IV).

- Partial sum parallelism: The high-level parallelism is the number of partial sum ciphertext being processed concurrently. The maximum degree of parallelism is m, where m is the length of the output vector. We denote this as the high level parallelism because the computation of partial sum ciphertext itself involves a series of lower level HE primitives.
- Residual polynomial parallelism: This is the mid-level parallelism. It determines the number of residual polynomials being processed simultaneously within each partial sum ciphertext computation. Some primitives in HE M×V have data dependency across residual polynomials which enforces the processing order. For example, the first $l-1$ output residual polynomials in Rescale depends on the last residual polynomial. Therefore the maximum degree of parallelism for Rescale is $l-1$.
- Coefficient-wise parallelism: As the low-level parallelism, this dimension determines how many coefficients in a

residual polynomial are processed concurrently. There are many element-wise operations between coefficients of polynomials in various HE primitives, which are straightforward to parallelize. Other operations such as NTT and INTT require non-trivial techniques to parallelize due to the access pattern of the algorithm (Section IV-B).

D. Bandwidth Efficient Dataflow

We design a dataflow specifically optimized for HE M×V. The dataflow combines various memory and compute optimizations: i) Given the DRAM transfer cost of various HE data entities and the available on-chip SRAM capacity, selectively store highly reused data on-chip to reduce memory bandwidth. ii) Efficiently utilize the compute resources by enabling various parallelization dimensions. iii) Apply HE primitive fusion to avoid additional DRAM transfers for intermediate ciphertext.

SRAM Allocator: The SRAM allocator takes the cost of various data entities in HE M×V as the input. An HE data entity is defined as the entire data structure for an operation. For example, the whole set of NTT twiddle factors or key switch keys is considered as one data entity. While we can further break these down into smaller chunks, we chose this granularity for simplicity. The allocator produces an allocation strategy as the output, where the allocation strategy specifies the location of the corresponding data entity (on-chip or in the DRAM). We use a greedy algorithm to allocate the SRAM space. The SRAM allocator always favors data entities that have higher cost (larger data transfer overhead). *Fusion:* To further reduce DRAM transfers, our dataflow fuses the processing of the HE primitives. The allocator reserves enough space to store the intermediate ciphertext whenever there is a producer and consumer relationship between two HE primitives.

Computation Dataflow: HE M×V computation can be implemented using nested loops, where the outermost loop iterates and accumulates the m partial sum computations while the inner loops are used for the computation of a single partial sum ciphertext. While the outermost loop (partial sum parallelism) is straightforward to exploit, naively increasing the partial sum parallelism can quickly use up the available resources of an FPGA device, especially the on-chip SRAM. Our dataflow applies loop tiling technique to allow g partial sum computations be performed concurrently, where g is a parameter of the dataflow. Within the computation of one partial sum, We apply two levels of loop tiling and unrolling to further utilize the compute resources. The first level exploits the residual polynomial parallelism (p) while the second level exploits the coefficient-wise parallelism (c).

IV. FPGA Accelerator Design

A. Accelerator Architecture Overview

We design a parameterized accelerator that is optimized for the bandwidth efficient dataflow. Table I lists the configuration parameters of the accelerator. The architecture of the accelerator is shown in Figure 4.

TABLE I
Configuration Parameters of the Accelerator

Parameter	Description
g	Number of PE groups.
p	Number of PEs per PE group.
c	Number of lanes in the MAC array and SPN per PE.
b	Number of banks of the scratchpad per PE.

The accelerator is composed of a number of PE groups. The number of PE groups equals to the outermost loop (partial sum parallelism) unrolling factor. A Partial Sum Reduction Engine connects all the PE groups and performs partial sum accumulation (ct-ct add). There are multiple PEs within a PE group. An Inter-PE Bus is used to enable PE-to-PE data communication within a PE group during key switching (Line 2-7 in Algorithm 2).

Within each PE, there are two major functional units: Modular MAC Array and Streaming Permutation Networks (SPN) [23] with NTT Cores. The Modular MAC Array is used by element-wise polynomial operations with up to 3 input polynomial operands and 1 output polynomial operand. We utilize SPN to enable streaming coefficient permutation for NTT, INTT and automorphism. Both functional units have multiple lanes to enable parallel processing of multiple coefficients per cycle (coefficient parallelism). Each PE has a local scratchpad memory that stores the residual polynomials and twiddle factors. A Control Sequencer runs a state machine and schedules tasks to each functional units.

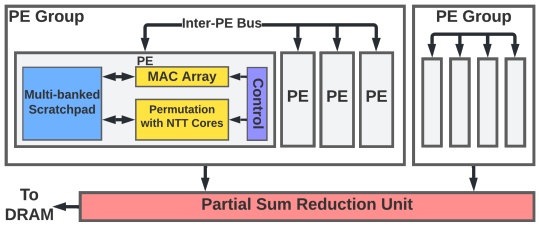

Fig. 4. Top-level architecture of the proposed accelerator.

B. Design Details

1) PE Architecture: Each PE is capable of producing one or more residual polynomials of a partial sum ciphertext. The number of residual polynomials a PE processes is $\lceil l/p \rceil$, where l is the number of residual polynomials in the ciphertext. The PEs in a group work collaboratively to compute a partial sum ciphertext. The computation is further broken down into HE primitives, which are mapped onto the two major functional units. Both functional units are fully pipelined and have a throughput of c coefficients per cycle.

Modular MAC Array: This functional unit performs coefficient-wise modular arithmetic between polynomials such as addition, subtraction, multiplication and accumulation. It is used by all the HE primitives except NTT, INTT and automorphism. It can load up to $3c$ coefficients and store c coefficients per cycle. We use Barrett reduction algorithm [24] to implement modular multiplication. This algorithm includes three integer multiplications. The first multiplication is a full width multiplication and multiplies two input operands. The other two are additional half width multiplications for Barrett

reduction. Similar to [25], our design is fully pipelined and can process two coefficients every cycle.

Permutation Networks with NTT Cores: NTT, INTT and automorphism require permutation of coefficients within a residual polynomial. Due to the complex data access pattern, such operations are difficult to parallelize. Naive solution requires a fully connected crossbar (unscalable) with carefully designed memory accesses to avoid bank conflicts [11]. To overcome this challenge, we utilize the NTTGen [26] to enable parallel processing of NTT and automorphism without costly corssbar. NTTGen uses Streaming Permutation Network (SPN) [23], a folded version of the multi-stage Benes network [27], to enable parallel data permutation. The SPN can achieve arbitrary permutation [23]. It has three subnetworks – two spatial permutation networks and one temporal permutation network, as shown in Figure 5. Spatial permutation shuffles the c coefficients that are received in the same cycle whereas temporal permutation rearranges the coefficients across different cycles. A spatial permutation network uses 2×2 switches to recursively compose a c-to-c connection. Temporal permutation is achieved by issuing reads and writes to c dual-port memory using pre-computed addresses.

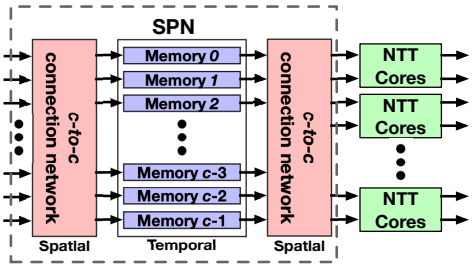

Fig. 5. Architecture overview of the SPN.

NTT Cores are only used by the NTT and INTT computations (bypassed during automorphism). There are $c/2$ NTT Cores per PE. The microarchitecture of the NTT Core is depicted in Figure 6. Each NTT Core receives two coefficients as inputs, multiplies with the twiddle factors, and generates two coefficients as outputs.

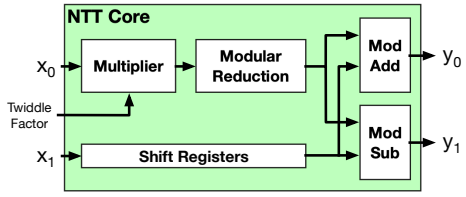

Fig. 6. Design of the NTT Core.

Multi-bank Scratchpad: We design a multi-bank scratchpad to allow reading up to $3c$ coefficients and writing c coefficients per cycle. Figure 7 shows the data layout of polynomials in the scratchpad. Each bank is implemented using one or more dual-ported BRAMs/URAMs and can store at least one polynomial or one set of twiddle factors. An entry in each bank stores c coefficients. The number of banks should match the throughput of the Modular MAC Array and Permutation Module to avoid pipeline stall.

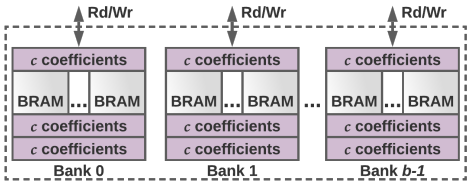

Fig. 7. Design of the multi-bank scratchpad with b banks.

2) Inter-PE Communication and Parial Sum Reduction: The key switching operation takes a ciphertext that is encrypted by a different secret key as the input and produces a new ciphertext that can be decrypted using the original secret key. The computation of each residual polynomial in the new ciphertext requires modular multiply-accumulate from all the residual polynomials of the input ciphertext (Line 4 - 8 in Algorithm 2). This algorithm requires data communication between the PEs. The Inter-PE Bus is designed for this purpose. The outermost loop (Line 1 in Algorithm 2) of the Key Switching operation is processed sequentially. In each iteration, one PE broadcasts its residual polynomial to all the other PEs in the same group at the rate of c coefficients per cycle. The residual polynomial is stored in each PE's local scratchpad. Upon receiving the residual polynomial, each PE can start the computation as specified by the innermost loop body. The Partial Sum Reduction Unit is a modular adder tree which takes c coefficients from the same residual polynomial in g partial sum ciphertext (PE groups) and adds them together to produce the final output ciphertext.

V. Evaluation

A. Experimental Setup

We implement our accelerator on Xilinx U200 FPGA [28] using SystemVerilog. The FPGA has 1,182K LUTs, 2,364K FFs, 35 MB on-chip SRAM and 6,840 DSPs. 64 GB of DRAM is attached to the FPGA, providing a peak bandwidth of 77 GB/s. Our FPGA designs are synthesized and place-and-routed using Xilinx Vivado 2020.2. We assume that the input matrix and vector are stored in the FPGA DRAM. We run RTL simulations to report latency, our main performance metric. The latency is defined as the duration from loading the input from DRAM to storing the output vector to DRAM.

HE Parameters. We implement our designs based on three sets of HE parameters. The parameters are extracted from Microsoft SEAL library [29] and satisfy 128-bit security. Table II lists the HE parameters.

TABLE II
HE Parameters Used in the Evaluation.

Parameters	Security	Poly Degree (N)	Computation Depth (l)
Set-A	128-bit	8,192	3
Set-B	128-bit	16,384	7
Set-C	128-bit	32,768	14

Accelerator Configurations. Table III shows the accelerator configuration for parameters Set-A, Set-B and Set-C respectively. The parameters are selected based on a resource estimator. We choose the number of PEs per PE group empirically.

We set the size of the scratchpad per PE to store the input vector, the NTT twiddle factors and the partial sum ciphertext. Then we maximize the number of PE groups based on the available on-chip SRAM resources.

TABLE III
ACCELERATOR CONFIGURATIONS.

	Set-A	Set-B & Set-C
Num PE Groups (g)	4	2
Num PEs per Group (p)	3	7
Num Lanes per PE (c)	16	16
Scratchpad Size per PE	1.5 MiB	1.5 MiB
Num Banks per Scratchpad (b)	3	3

Benchmarks. We consider single batch (batch size $= 1$) as the target inference deployment scenario [30]. Under this setting, fully-connected (FC) layers are equivalent to M$\times$V operations. We evaluate the performance of HE M$\times$V using various FC layers in state-of-the-art computer vision and speech DNN models. Table IV shows the list of the benchmarks. There are 7 layers in total obtained from AlexNet, VGG, RNN-T and Deep Speech 2.

TABLE IV
M$\times$V FROM STATE-OF-THE-ART DNN MODELS.

Name	Model	m	n
Alex-6	AlexNet	4,096	9,216
Alex-7	AlexNet	4,096	4,096
Alex-8	AlexNet	1,000	4,096
VGG-6	VGG	4,096	25,088
RNNT-LSTM	RNN-T	1,024	2,048
RNNT-FC0	RNN-T	512	1,344
DS2-GRU	Deep Speech 2	1,600	1,600

CPU Baseline. We compare the performance of our accelerator with a multi-core CPU baseline implementation. The CPU of the server is an AMD Ryzen 3990X CPU, which has 64 cores (128 threads) running at 2.9 GHz. The server has 256 GB DDR4 with 200 GB/s peak bandwidth to DRAM. We use TenSeal v0.3.8 [31] and 64 CPU threads to implement the benchmarks listed in Table IV.

B. Evaluation Results

1) Comparison with CPU Baseline: We compare the performance of our accelerator against a multi-core CPU implementation (1-thread and 64-thread) on 7 benchmarks selected from AlexNet, VGG, RNN-T and Deep Speech 2. Figure 8 shows the absolute execution time (bar chart) and speedup (line chart) using the parameters in Set-B. We observe a similar trend when using the other two parameter sets. Our implementation achieves up to 9$\times$ speedup compared to the 1-thread CPU implementation. Although the CPU platform has almost 3$\times$ DRAM bandwidth, our design significantly outperforms the general purpose implementation with up to 3.8$\times$ (GeoMean 2.8$\times$) faster than the 64-thread baseline. The speedup offered by our FPGA accelerator demonstrates the effectiveness of the specially designed dataflow for HE M$\times$V.

Comparison with FPGA Baseline: We compare our design with HEAX [11], the state-of-the-art FPGA accelerator for CKKS. HEAX cannot be used to accelerate HE M$\times$V due to the lack of support for automorphism. We compare the performance of key switching, the most time consuming operation in HE M$\times$V. Both designs use the same HE parameters (Set-B). Because HEAX only reports the number of key switching operations per second, we use the same metric to compare. By leveraging various parallelization dimensions, our design achieves up to 1.95$\times$ speedup compared to HEAX.

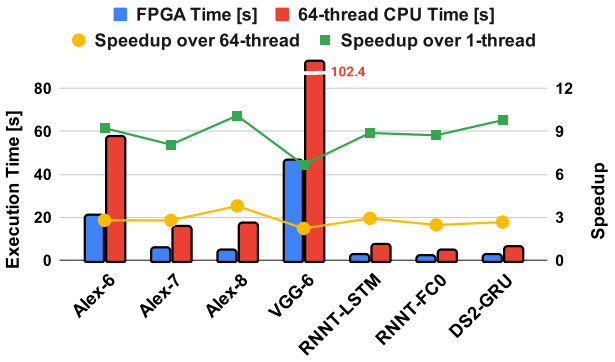

Fig. 8. Execution time and speedup across various benchmarks.

TABLE V
KEY SWITCHING PERFORMANCE COMPARISON.

	Our Design	HEAX [11]	Speedup
Operations per Second	5,122	2,616	1.95$\times$

2) Execution Time Breakdown: Figure 9 shows the execution time breakdown in percentage for various HE operations using the design for parameter Set-B. The execution time is dominated by the key switching and rescale operations, which involve several time-consuming NTT and INTT. In contrast, pt-ct mult, automorphism and partial sum reduction take less time to execute because they involve less number of operations (e.g., element-wise or permutation without computation).

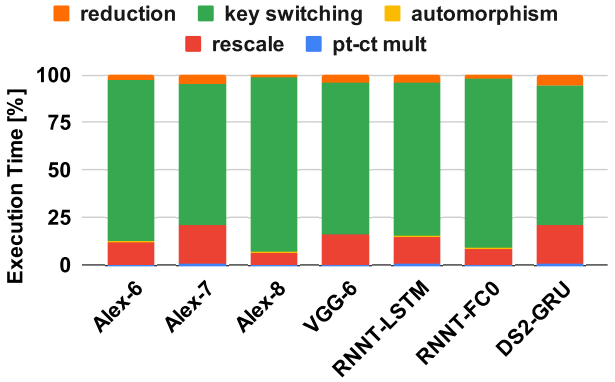

Fig. 9. Execution time breakdown by HE primitives.

Table VI shows the execution time of various HE primitives on one residual polynomial. NTT and INTT are the most time consuming compute kernels. Therefore HE operations that require multiple NTTs and/or INTTs have high latency. Our NTT Cores have a latency of 19 cycles.

3) Scalability Analysis: To evaluate the scalability of the architecture, we instantiate two designs for the three HE parameter sets, as listed in Table III. We reuse the same FPGA design without runtime reconfiguration for Set-B and Set-C. Figure 10 shows the execution time of various benchmarks.

TABLE VI
EXECUTION TIME OF VARIOUS HE PRIMITIVES.

	NTT/INTT	Automorphism	Element-wise
Cycles	14,658	1,024	1,024

The architecture is scalable from 3 PEs per group to 7 PEs per group. With the bandwidth-efficient dataflow, the bottleneck of HE M×V has shifted from memory bound to compute bound. The performance is proportional to the total number of operations of each benchmark.

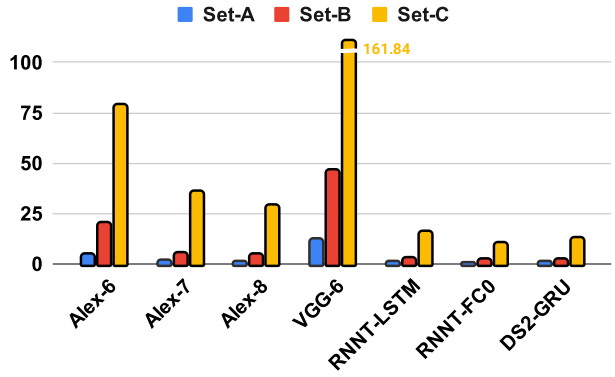

Fig. 10. Execution time of the benchmarks across various HE parameter sets.

4) Resource Utilization: The resource utilization of accelerator designs for Set-A and Set-B/C is listed in Table VII. URAMs are heavily used in our design to store the polynomials. Due to the modular arithmetic, each modular multiplier uses 17 DSPs. BRAMs are used by the SPN (temporal permutation). SPN reads coefficients of a polynomial in sequential order and permutes the coefficients such that coefficients with a stride defined by the NTT algorithm are produced in the same cycle. Therefore, SPN requires internal buffers to temporarily store the coefficients. The size of the buffers is determined by the maximum degree of the polynomial that can be supported. The maximum degree supported by Set-B/C design is much greater than Set-A design, therefore Set-B/C design requires more BRAM resources. The LUTs are used to implement modular adders. Set-B/C design has more PEs and thus requires more LUT resources.

TABLE VII
ACCELERATOR RESOURCE CONSUMPTION ON FPGA.

	***k*LUT**	***k*FF**	**BRAM**	**URAM**	**DSP**	**Freq.**
Set-A	756.3	610.4	224	656	4,896	200 MHz
Set-B/C	835.6	1,093.2	896	742	5,712	180 MHz

VI. RELATED WORK

CPU. HE optimized frameworks [32], [33], [29] are proposed to ease the tasks of programming HE applications on CPU. Primitive level and graph level optimizations were explored to speedup HE computation. Gazelle [14] developed novel packing schemes under the BFV scheme to reduce the number of rotations in HE M×V and HE convolution. Castro et al [34] proposed data layout optimizations to improve DRAM efficiency for various HE operations. Due to the lack of fine-grained dataflow and parallelism, HE computations on CPU still incur high latency [14].

GPU. Prior GPU implementations [35], [36] accelerate NTT on GPU and do not consider the other primitives. Jung et al. used GPU to accelerate all the CKKS primitives [13]. However, the data transfer overhead introduced by naively chaining the primitives to compose higher level operations such as HE M×V was not considered.

FPGA. Prior FPGA implementations [11], [12], [37] accelerate HE multiplication and addition. However, HE rotation was not supported. In addition, they do not consider the data layout and reuse in HE M×V. Using these designs will result in excessive DRAM data transfer overhead. Tian et al [38] proposed an FPGA accelerator for HE convolution. Their design avoids costly HE rotations by using the frequency domain convolution. This technique cannot be applied to HE M×V. [39] proposed analytical models of FPGA accelerated HE CNN. However, many implementation details were ignored. To the best of our knowledge, no prior work proposed an FPGA-based design that can perform HE M×V.

ASIC. Cheetah [4] optimized BFV HE algorithms based on Gazelle [14] and designed an AISC for them. It only supports small HE parameters that do not require RNS decomposition, which makes the implementation less practical. F1 [40] proposed ASIC acceleration of the BGV scheme. Their design uses the row-wise packing to process HE M×V, not only requiring more rotations but also producing m times more output ciphertext than the scheme used in this paper, where m is the length of the output vector. BTS [41] is a purposely built ASIC for the CKKS scheme. To support bootstrapping, BTS is a very costly design that has a projected area of more than 350mm^2. While Cheetah, F1 and BTS achieve high performance, they require extremely high resource and bandwidth availability such as 2 GHz operating frequency (F1) or 192 MB on-chip SRAM (BTS). Further, the performance of these accelerators were evaluated on a simulator. In contrast, we propose a practical implementation.

VII. CONCLUSION

By comprehensively analyzing the encryption overhead and the data reuse opportunities of HE M×V, we show that the compute and memory characteristics of HE M×V is significantly different from its unencrypted counterpart and requires a different accelerator design. Motivated by the analysis, we proposed a bandwidth efficient dataflow by only storing the highly reused HE data entities on-chip and thereby dramatically reducing the DRAM data transfers. The dataflow exploits three dimensions of parallelism to speedup the computation of HE M×V. Based on the proposed dataflow, we design the first FPGA accelerator for HE M×V. We evaluate the proposed accelerator on 7 benchmarks. Experimental results show that our FPGA accelerator is up to 3.8× (GeoMean 2.8×) faster than the 64-thread CPU implementation.

VIII. ACKNOWLEDGEMENT

This work has been sponsored by the U.S. National Science Foundation under grant SaTC-2104264 and CNS-2009057. Equipment grant from AMD 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] 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.

[3] 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)*.

[4] 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.

[5] T. N. Sainath, O. Vinyals, A. Senior, and H. Sak, "Convolutional, long short-term memory, fully connected deep neural networks," in *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2015.

[6] R. Dey and F. M. Salem, "Gate-variants of gated recurrent unit (gru) neural networks," in *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*, 2017, pp. 1597–1600.

[7] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, 1997.

[8] 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.

[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] L. de Castro, R. Agrawal, R. Yazicigil, A. Chandrakasan, V. Vaikuntanathan, C. Juvekar, and A. Joshi, "Does fully homomorphic encryption need compute acceleration?" *arXiv preprint arXiv:2112.06396*, 2021.

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

[12] 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.

[13] W. Jung, S. Kim, J. H. Ahn, J. H. Cheon, and Y. Lee, "Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with gpus," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 114–148, 2021.

[14] 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.

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

[16] 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*.

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

[18] 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.

[19] 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.

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

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

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

[23] R. Chen and V. K. Prasanna, "Automatic generation of high throughput energy efficient streaming architectures for arbitrary fixed permutations," in *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, 2015, pp. 1–8.

[24] D. Hankerson, A. J. Menezes, and S. Vanstone, *Guide to Elliptic Curve Cryptography*. Berlin, Heidelberg: Springer-Verlag, 2003.

[25] S. Kim, K. Lee, W. Cho, J. H. Cheon, and R. A. Rutenbar, "Fpga-based accelerators of fully pipelined modular multipliers for homomorphic encryption," in *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 2019.

[26] Y. Yang, S. R. Kuppannagari, R. Kannan, and V. K. Prasanna, "Nttgen: A framework for generating low latency ntt implementations on fpga," in *Proceedings of the 19th ACM International Conference on Computing Frontiers*, ser. CF '22, 2022, p. 30–39.

[27] V. E. Beneš, "Optimal rearrangeable multistage connecting networks," *The Bell System Technical Journal*, vol. 43, no. 4, pp. 1641–1656, 1964.

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

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

[30] V. J. Reddi and et al, "Mlperf inference benchmark," in *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ser. ISCA '20. IEEE Press, 2020, p. 446–459.

[31] A. Benaissa, B. Retiat, B. Cebere, and A. E. Belfedhal, "Tenseal: A library for encrypted tensor operations using homomorphic encryption," 2021.

[32] 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.

[33] 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.

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

[35] Y. Zhai, M. Ibrahim, Y. Qiu, F. Boemer, Z. Chen, A. Titov, and A. Lyashevsky, "Accelerating encrypted computing on intel gpus," *arXiv preprint arXiv:2109.14704*, 2021.

[36] S. Kim, W. Jung, J. Park, and J. Ahn, "Accelerating number theoretic transformations for bootstrappable homomorphic encryption on gpus," in *2020 IEEE International Symposium on Workload Characterization (IISWC)*, 2020.

[37] S. S. Roy, A. C. Mert, S. Kwon, Y. Shin, D. Yoo *et al.*, "Accelerator for computing on encrypted data," *Cryptology ePrint Archive*, 2021.

[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] 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.

[40] 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.

[41] S. Kim, J. Kim, M. J. Kim, W. Jung, M. Rhu, J. Kim, and J. H. Ahn, "Bts: An accelerator for bootstrappable fully homomorphic encryption," *arXiv preprint arXiv:2112.15479*, 2021.