Exploring Bitslicing Architectures for Enabling FHE-Assisted Machine Learning

Soumik Sinha, Sayandeep Saha, Manaar Alam, Varun Agarwal, Ayantika Chatterjee, Anoop Mishra, Deepak Khazanchi, and Debdeep Mukhopadhyay, *Senior Member, IEEE*

***Abstract*—Homomorphic encryption (HE) is the ultimate tool for performing secure computations even in untrusted environments. Application of HE for deep learning (DL) inference is an active area of research, given the fact that DL models are often deployed in untrusted environments (e.g., third-party servers) yet inferring on private data. However, existing HE libraries [*somewhat* (SWHE), *leveled* (LHE) or *fully* homomorphic (FHE)] suffer from extensive computational and memory overhead. Few performance optimized high-speed homomorphic libraries are either suffering from certain approximation issues leading to decryption errors or proven to be insecure according to recent published attacks. In this article, we propose architectural tricks to achieve performance speedup for encrypted DL inference developed with exact HE schemes without any approximation or decryption error in homomorphic computations. The main idea is to apply quantization and suitable data packing in the form of bitslicing to reduce the costly noise handling operation, Bootstrapping while achieving a functionally correct and highly parallel DL pipeline with a moderate memory footprint. Experimental evaluation on the MNIST dataset shows a significant (37×) speedup over the nonbitsliced versions of the same architecture. Low memory bandwidths (700 MB) of our design pipelines further highlight their promise toward scaling over larger gamut of Edge-AI analytics use cases.**

***Index Terms*—Accelerators, bitslicing, encrypted analytics, homomorphic encryption (HE).**

Manuscript received 6 August 2022; accepted 6 August 2022. Date of publication 17 October 2022; date of current version 24 October 2022. This article was presented in the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES) 2022 and appears as part of the ESWEEK-TCAD special issue. This work was supported in part by the SERB SUPRA DST Government of India and in part by U.S.–India Cooperation in Higher Education With University of Nebraska at Omaha. This article was recommended by Associate Editor A. K. Coskun. *(Corresponding author: Soumik Sinha.)*

Soumik Sinha, Ayantika Chatterjee, and Debdeep Mukhopadhyay are with the Indian Institute of Technology Kharagpur, Kharagpur 721302, India (e-mail: soumiksinha21@gmail.com; ayantika@atdc.iitkgp.ac.in; debdeep@cse.iitkgp.ac.in).

Sayandeep Saha is with Nanyang Technological University, Singapore (e-mail: sayandeep.saha@ntu.edu.sg).

Manaar Alam is with New York University Abu Dhabi, Abu Dhabi, UAE (e-mail: alam.manaar@gmail.com).

Varun Agarwal is with Delhi Technological University, New Delhi 110042, India (e-mail: agarwalvarun2000@gmail.com).

Anoop Mishra and Deepak Khazanchi are with the University of Nebraska Omaha, Omaha, NE 68182 USA (e-mail: amishra@unomaha.edu; khazanchi@unomaha.edu).

Digital Object Identifier 10.1109/TCAD.2022.3204909

I. Introduction

ADVANCED analytics workloads, involving machine learning (ML) or deep learning (DL), are often deployed in public cloud servers, such as AWS, Azure, GCP so that the vast compute infrastructures could be exploited to attain better performance and scalability. Considering the recent vulnerabilities reported in these public servers as published in reports due to [1], [2], and [3], enforcing the security of mission-critical ML workloads deployed in cloud servers is of utmost importance. *Encrypted analytics* is a viable solution in this direction with the support of a family of homomorphic encryption (HE) scheme.

Most of the existing approaches for encrypted DL are constructed over leveled HE (LHE) or Somewhat HE (SWHE) schemes, both of which support homomorphic operations only for limited circuit depth [4], [5], [6], [7]. Beyond this depth, the ciphertext noise grows to an extent where a successful decryption becomes impossible. While SWHE or LHE schemes are still preferred because of performance benefits, their support for limited circuit depth becomes an issue for complex inference tasks where deeper networks (therefore, deeper circuit depths) are required. Fully HE (FHE) addresses this issue with bootstrapping which can reduce the ciphertext noise during homomorphic computation, and, therefore, enables computation for arbitrary circuit depths [8]. Bootstrapping "refreshes" a ciphertext by running the decryption function on it homomorphically, thus reducing the ciphertext noise. However, such power comes at a cost of Bootstrapping, which is both computationally intensive and memory hungry. Ducas and Micciancio [9] proposed an FHE bootstrapping mechanism which takes less than 1 s. Later, works due to Chilotti *et al.* [10] further improved this timing in their work, which proposes a Torus polynomial-based FHE bootstrapping procedure within 13 ms and also compresses the bootstrapping key size from 1 GB to 16 MB thus hugely optimizing the memory overhead. We shall revisit this scheme in later sections of this article to explain its significance in connection to our proposed encrypted ML architecture.

While the choice between LHE/SWHE and FHE can still be debated over based on the performance issue, recent research has shown that FHE is indeed a safer choice from the perspective of security. It has been found that the secret key of certain LHE schemes (such as CKKS [11]) can be extracted with the knowledge of a few decrypted ciphertexts [12], [13]. Such attack can fully compromise the privacy, which is the sole purpose of choosing encrypted analytics. Further, many efficient FHE-assisted ML approaches use approximations in the underlying homomorphic scheme

to avoid costly bootstrapping. These encrypted ML models only support approximations of nonlinear activations which incorporates approximation errors or support only specific activation functions, like square activations [4], [14]. In this situation, FHE schemes with efficient bootstrapping support seem to be a judicious choice for realizing DL inferences.

Noteworthy here is the fact that contrary to traditional algorithm implementations, FHE domain implementations require circuit-based realization of algorithms on our traditional unencrypted processors, which is a nontrivial task. This is due to the fact that existing FHE libraries support basic computational gates and homomorphy can only be achieved by realizing algorithms on top of them [10]. Again to support arbitrary noise-free FHE computations, usage of costly Bootstrapping operation is mandatory, but that leads to serious performance bottleneck and huge ciphertext size blow-up. Moreover, FHE gates can only support integer domain computations, which is a major drawback to realize any ML model directly in FHE domain. All these limitations motivate us to make an effort in developing a robust FHE encrypted ML inference framework with performance acceleration and without any inference accuracy loss with respect to unencrypted domain. It is to note that in this era of ML as a service (MLaaS), both model parameters and data in ML scenarios can be considered as intellectual property (IP). However, following existing literature and considering the overhead of HE schemes, the threat model of our proposed framework takes data privacy into account and it is developed based on public model private data assumption. Lately, FHE-based ML architectures with reduced timing overheads have been proposed by state-of-the-art libraries like Concrete [15]. However, our investigations in this work highlight how Concrete FHE library may lead to erroneous decryption. That is why we build up the proposed encrypted ML inference on mathematically accurate NuFHE [16], which is the extension of FHE library over Torus (supports fastest bitwise homomorphy in literature within few milliseconds) [17]. Our proposed end-to-end encrypted inference pipeline never tweaks the library or does not require any approximation in activation implementation due to the support of basic gate-level homomorphy. However, straightforward realization of ML models only with FHE gates incurs huge timing as well as memory overhead. As an example, our baseline implementation with only MNIST dataset requires around 80 h for FHE inference and 6.7-GB memory on average.

To reduce this huge timing and memory overhead, we propose a new FHE ML architecture where we primarily design a wrapper to optimize the overall design with two main architectural tricks: 1) quantization and 2) bitslicing. Quantization is an approach widely adopted for reducing the memory footprint of DL model so that they can be easily deployed at resource-constrained embedded devices. Reducing the memory footprint is also essential in the context of FHE. Bitslicing, on the other hand, is a parallelization technique primarily used in cryptography [18], where bitwise logical operations are quite prevalent. The main idea is to pack the operands of several bitwise operations in the same register so that they can be processed in parallel with one processor instruction. In FHE context, bitslicing is mainly adapted as tensor packing technique to reduce the number of bootstrapping operation. This in turn can boost the performance of both Conv2D layers and dense operators. Further, the prerequisite of FHE-based implementation setting is discretized tensors as opposed to real valued ones and we achieve this using quantization supported inference. Our design ensures that bitsliced accelerator with FHE operations work on integer ciphertext operands. With all these modifications, following are the major contributions of this article.

1) First, we try to highlight some limitations of state-of-the-art Concrete library which incorporates decryption errors for real-valued tensors.
2) Further, FHE computations are only possible on integer values and Bootstrapping operations on 32-bit tensors could worsen the overall performance. To mitigate these issues, we adopt quantized inference on *INT*8 or 8-bit Integer tensors. Without accuracy degradation, *INT*8 quantization empowers us in performance boosting.
3) Next, we revisit the building blocks of convolutional neural network (CNN) or deep neural network (DNN) architectures and optimize them in the FHE context. Finally, we design a memory-efficient ML accelerator with architectural tricks like packing and software bitslicing. We evaluate our techniques on both CNNs as well as DNNs and demonstrate the speedup potential of around 37 times.
4) We perform an in-depth memory profiling and try to benchmark our inference accelerator with other state-of-the-art approaches reported in [4], [5], [7], and [19]. These reported works not only incorporate approximations, most of them also incur huge memory requirement of around 2–3 GB for evaluating the inferences. Works like [5] and [7] are low latency pipelines but they are certainly memory intensive (consuming at least few GB's for inference on moderate architectures). Our bitslicing-based design paradigm, however, optimizes significantly on memory footprint and is capable of CNN inference within tight memory budgets of around 700 MB.

With this motivation, our paper is organized as follows. Introduction section (Section I) highlights the security vulnerabilities of public cloud servers and explains the motivations to adopt HE for encrypted ML architecture design. Section II highlights the background and limitations of existing works. Section III explains few preliminary concepts with important terminologies. Section IV describes the design details of proposed ML architecture for encrypted Inference accelerated by architectural optimizations like quantization, bitslicing, etc. This section also includes the optimized design details for FHE counterparts of CNN building blocks. Finally, Section V mentions the experimental details and results comparing with state-of-the-art encrypted ML frameworks followed by conclusion in Section VI.

II. Background and Limitations of Related Works

In this section, we highlight few important encrypted ML architectures reported in recent literature based on the existing

TABLE I
HE SCHEMES TAXONOMY

HE Scheme	Examples	Open-source Libraries
Partial(PHE)	RSA,EL-Gamal [20], [21]	OpenSSL
Somewhat(SHE)/Leveled(LHE)	BGV [11], [22]	HELib, SEAL Palisade
Fully/FHE	Chilotti's TFHE [10]	NuFHE, Concrete

HE libraries as enlisted in Table I. Among these homomorphic ML frameworks, the encrypted networks mentioned in [4] and [23] suffer from liberty of selecting the activation functions. These encrypted networks can only incorporate square activations due to the choice of limited homomorphic schemes. Further, efforts like [24] can only work with approximated layers with low-degree polynomials. The work in [7] reports only estimated results for deeper networks and only provides implementation results for moderate sized networks that can be evaluated with predefined multiplicative depth of an LHE scheme. Again, the reported FHE ML inference in [14] approximates the existing FHE scheme on Torus [17] to improve the performance. However, the scheme suffers from the limitations of the signed activation functions, which sometimes leads to erroneous predictions.

All these aforementioned efforts are based on either some leveled or arithmetically approximated encryption libraries, which incorporate some inherent limitations. Next, we will try to vividly highlight the major bottlenecks of different classes of encrypted ML frameworks which dominate the literature.

1) *Binarized or Discretized Inference-Based Encrypted ML Approaches:* The promise of neural network inference with binarized or discretized tensors has lately caught attention of researchers due to its performance efficiency. Most relevant approaches report the acceleration potential of quantized inference with bitwise logical operations. As described in [25], this set of models works with Binarized tensors (all tensors represented as *Power-of*-2) and hence, all multiplications on those tensors could be achieved with logical bit shift operators. This property makes this technique very effective with inference workloads in FHE domain, as in binarized inference the costly FHE multiplications can be replaced with bit shifts thereby saving huge bootstrapping complexity as reported in [7]. Borrowing from the concept of binarized neural networks, the effect of discretized neural networks has further been explored in Bourse's work [14]. While the former methods largely resort to multiplication less inference networks, this method performs some thresholding of activation tensors *a priori* so that the approximation function computation will come down to a matter of just changing sign bits. Noteworthy here, that all these quantized weights or even predefined thresholds of activation units are applicable to supervised learning paradigms only where they are trained before inference during training phase. As a caveat, this logarithmic quantization trick might not work in such a straightforward manner for other ML use cases which employs linear or ridge regression or some other unsupervised learning algorithm.
2) *Approximation Schemes for CNN Inference:* To avoid costly bootstrapping in terms of memory blowup and performance, many HE-assisted ML approaches avoid FHE and opt for approximated LHE as underlying scheme. Due to the absence of bootstrapping operation, nonlinear activation functions need to be replaced with low-degree polynomial functions to be evaluated by LHE libraries. For example, [6] is such an approach which approximates nonlinear activations with arithmetic polynomials. Inference frameworks proposed in [4] and [24] build up on same ideas and also produce good accuracy. However, scheme proposed in [11] is one of the important approximated library which is getting used to develop the basic building blocks of recent encrypted CNN architectures. Recent attack [12] highlights the vulnerabilities of this underlying HE scheme and thus encrypted ML models designed based on this scheme cannot be considered fully secure anymore.
3) *LHE or SWHE-Based Encrypted ML Inference:* LHE or SWHE only allow limited number of homomorphic computations (limited multiplications or additions). Simpler neural net architecture might run smoothly with underlying LHE libraries as reported in [4] and [24]. However, with the growth of multiplicative depth of neural networks, ciphertext noise amplifies to an extent when successful decryption is no longer possible. For complex real-life use cases, inference on deeper nets may become erroneous (and training is infeasible). That makes the acceptance of LHE and SWHE schemes very limited for DNNs. To resolve this limitation of SWHE or LHE schemes, usage of FHE has been proposed. Recently, Chillotti *et al.* [26], Lee *et al.* [19], and Bourse *et al.* [14] demonstrated efforts in building neural net inference frameworks which runs on FHE encrypted ciphertexts with bootstrapping.
4) *Concrete-Based Torus FHE (TFHE) Encrypted ML Inference:* Zama's Concrete library is an optimized library for FHE ML inference [15]. However, our experimental observations point out that homomorphic operations supported by this library incur decryption errors computed on integer plaintexts. To elucidate on this, we perform some cubic functions on ciphertexts and evaluate few leveled computations. Tables II and III highlight errors that Concrete [15] encounters during plaintext decryptions during Bootstrappable FHE or LHE computations (Level_*i* Error in the table indicates error occurred after *i*th addition). These errors can amplify as the level grows and the approximations performed by Concrete can lead to misleading inference in ML processing. Concern of significant accuracy drop (around 80%) is already reported in [26] for deeper networks and that questions the consistency of Concrete library for encrypted ML architecture design.

In this context, our objective is to keep the underlying FHE encryption library intact and provide architectural optimizations to achieve overall performance improvement.

TABLE II
CONCRETE FHE AND LHE ERRORS

Plaintext	Target Function=x^3	FHE Error	LHE Error
-2.0	-8	3.14	.004809
-4.0	-64	5.3174	.00236
7.0	343	3.154	.00772
5	125	7.94	.004809

TABLE III
CONCRETE LEVELED COMPUTATION ERRORS

Plaintext	Level_1 Error	Level_2 Error	Level_3 Error	Level_4 Error
8.2+5.6+1.2+2.2+5.0	0.032	0.028	0.026	0.0321
4.2+5.6+1.2+2.2+5.0	0.00031	0.01022	0.02004	0.0503
10+1.1+0+2.2+5.0	0.022	0.0292	0.0264	0.0329
0+2.2+3.3+4.4+5.5	0.0133	0.0239	0.0264	0.0329
1.0+2.0+3.0+4.0+5.0	0.0097	0.0154	0.0225	0.036

To summarize, in this work, we try to address few major challenges ahead.

1) Floating point inputs or weights vectors cannot be processed by FHE schemes, which will be handled by suitable quantization without any accuracy loss.
2) Design of scalable building blocks for FHE inference pipeline for example Conv2D, Dense, MaxPool, and Average Pool (AvgPool) will be while ensuring low bootstrapping complexity of FHE operations in the process.
3) DNNs involve computation of activation functions (like tanh, sigmoid, [rectified linear unit (ReLU)]. It is a generic challenge for any limited HE scheme to realize these functions. Our proposed architecture will be capable enough to support all kind of activation without any approximation error.

III. PRELIMINARIES

In this section, we elaborate some HE primitives important for further discussion of this article. HE is an elegant mathematical tool which allows computations on the ciphertext without any explicit decryption. Most HE schemes support at least one of the two fundamental computations either homomorphic addition (HADD) and homomorphic multiplication (HMUL). For example, the classical RSA cipher is a multiplicative HE, which supports multiplication operation in encrypted domain. According to the power of encrypted computation, HE schemes can be classified broadly into three categories: 1) partial HE (PHE) allows only one homomorphic operation either HADD or HMUL; 2) SWHE allows limited number of one operation, generally HMUL; and 3) LHE allows limited number of homomorphic operations and finally FHE allows arbitrary number of computations. FHE schemes, in general contain elementary homomorphic gate operations like AND, OR, NAND, etc., followed by Bootstrapping which is a crucial operation for denoising in FHE circuits. To understand the concept of noise in homomorphic circuits, consider the following example [27].

Considering an HE scheme with odd number p as shared secret key.

1) To encrypt a bit m, a random large q, and small r are chosen and output ciphertext is computed as $c = pq + 2r + m$. Ciphertext is close to a multiple of p and $m =$ LSB of distance to nearest multiple of p.
2) The decryption is performed as: $m = (c \bmod p) \bmod 2$.

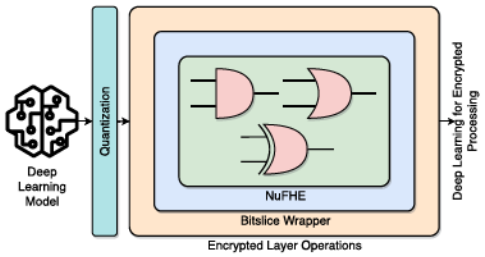

Fig. 1. Overview of the proposed framework.

Now, we explain why the above-mentioned scheme is considered to be homomorphic. Considering two ciphertexts $c_1 = q_1 p + 2r_1 + m_1$ and $c_2 = q_2 p + 2r_2 + m_2$, the addition and multiplication operations are defined with the following equations. Addition of the ciphertexts can be defined as

$$c_1 + c_2 = (q_1 + q_2)p + \underbrace{2(r_1 + r_2) + (m_1 + m_2)}_{\text{Distance to nearest multiple of p}}$$
$$c_1 + c_2 \bmod p = \underbrace{2(r_1 + r_2)}_{\text{error-term}} + (m_1 + m_2).$$

Multiplication of the ciphertexts can be defined as

$$c_1 \cdot c_2 = (c_1 q_2 + q_1 c_2 - q_1 q_2)p + 2(2r_1 r_2 + r_1 m_2 + m_1 r_2) + m_1 m_2$$
$$c_1 \cdot c_2 \bmod p = \underbrace{2(2r_1 r_2 + \ldots)}_{\text{error-term}} + m_1 \cdot m_2.$$

Homomorphic property retains till the error-term is within a certain limit. However, the error-term (or noise) grows faster for multiplication compared to addition. Hence, the scheme is somewhat homomorphic. Gentry's work [27] has shown the first noise handling technique termed as Bootstrapping and explained the first plausible construction of the FHE scheme.

Bootstrapping a ciphertext indicates homomorphically decrypting it, using an HE of its own secret key. In the end, we get an encryption of the same plaintext, but with reduced noise level. In practice, this operation is the main support to retain homomorphy for arbitrary computations but it is again the main reason for serious performance bottleneck and memory hungry nature of FHE. Hence, FHE literature is still evolving to improve the Bootstrapping technique. Among other existing schemes, the Torus-based FHE scheme proposed on the concept of learning with errors (LWE) paradigms [28], [29] offers fastest gate bootstrapping within few milliseconds.

Our proposed HE ML architecture as illustrated in Fig. 1 is implemented as based on python library NuFHE [16], which is an extension of that TFHE scheme [10]. Hence, we restrict our HE related discussion in context of TFHE. More formally, any FHE scheme can be defined as a tuple of four algorithms: (Gen, Enc, Dec, and Eval). Considering security parameter λ and two public parameters N and k, the algorithms related to TFHE can be defined as follows.

1) *Gen* (1^λ, *N, k):* This keygeneration algorithm generates secret key *Sk* and evaluation key *Ek*, which will be used in subsequent algorithms.
2) *Enc*(*m, Sk*)*:* This encryption algorithm encrypts message $m \in \mathcal{M}$, where message space $\mathcal{M} = \mathbb{T}[X]/(X^N + 1)$ (N sized Torus polynomial defined over Torus $\mathbb{T}$). Since, the secret key *Sk* consists of *k* number of *N*-sized polynomials, the ciphertext *Ct* is computed as (A, B), where A is a random public parameter and $B = \sum_{i=1}^{k} A[i] \cdot Sk[i] + m + e$. Parameter e is Ring-LWE (RLWE) noise polynomial, where each e_j is chosen from Gaussian distribution.
3) *Eval (Ct, F, Ek):* This homomorphic evaluation step evaluates arbitrary function *F* on RLWE ciphertext *Ct* by intermediate LWE ciphertext generation. In FHE mode, homomorphic evaluation incorporates circuit bootstrapping procedure, that takes an LWE ciphertext as input, reduces its noise, and converts it back to a GSW ciphertext suitable for subsequent packed operations. Later, on the final LWE output from the evaluation technique is converted to final RLWE ciphertext.
4) *Dec (Ct, Sk):* Final decryption step involves computation of $\phi = B - \sum_{i=1}^{k} A[i] \cdot Sk[i]$ to get $m + e$. Finally, each of the N coefficients of ϕ are rounded up to return the coefficients of m.

Consequently, a couple of Torus-based libraries has been publicly released by Chilotti *et al.* of which TFHE [17], Zama's Concrete [30], and NuFHE [16] are widely being used to this date.

A. Convolutional Neural Networks

CNNs proposed by [31] have evolved into widely popular classification algorithms for advanced analytics tasks, credited to their phenomenal success in the image or video classification domain. A CNN is a multilayered neural network with three following layers.

1) *Convolution:* The convolution operation extract features from an image through the dot product of image pixels and a custom image kernel or image filter. An $H \times W$ image is typically convoluted with $M \times M$ kernel to get $(H-M+1) \times (W-M+1)$ output. In convolution, operations like stride and pooling are integrated to preserve the dimensions of the image. Feature maps are generated in convolution operation, which provides insights into the features extracted from the convolution. For 2-D convolution operation, each feature map is a 2-D matrix that contains features as data vectors that can be visualized on the image space.
2) *Activation Function:* Activation functions are used to transform the weighted sum of the inputs with bias into an output. It is used to provide nonlinearity to the output. ReLU, softmax, and sigmoid are a few most used activation functions in CNN.
3) *Pooling:* Pooling is a downsampling mechanism for the image and is usually applied after the convolution operation. It reduces the computation load. MaxPool and AvgPool are the most commonly used pooling mechanism where maximum and average values are taken from the matrix produced by convolution layer.

TABLE IV
NOTATIONS AND TERMINOLOGIES

Symbol	Description
p_{ij}	ij^{th} pixel of image matrix
w_{ij}	ij^{th} weight value
H	Height of the Image Matrix
W	Width of the Image Matrix
M	Dimension of $(M \times M)$ Convolution Filter
N	Packing width for bitslicing
c	Number of channels

B. Bitslicing in Software

Originally proposed by Biham [18], for accelerating the computation of DES [32] in software, bitslicing is a parallelization technique widely utilized for cryptographic implementations. The main idea is to utilize the (bitwise) logical instructions (AND, XOR, NOT, OR) present in modern processors in an SIMD fashion. The entire computation is first represented as a circuit consisting bitwise logical operations. Next, a number of parallelizable bit operations is packed together in registers and processed with logical instructions. For example, if a processor architecture contains *n*-bit registers, and there are *n* bitwise logical-AND operations in the computation that can happen in parallel, then, the operands are packed in two registers and processed with a single bitwise AND instruction. This trick computes AND for all *n*-bits in parallel, which results in significant speedup. For state-of-the-art block ciphers, which can always be represented as a sequence of bitwise operations, such bitslicing can enable parallel processing of *n* plaintexts, where *n* is the width of the registers in the processor.

IV. Quantized Inference With Bitslicing

The goal of this research is to address the challenges of realizing large and complex neural networks in FHE domain with reasonable timing and memory footprints. This is achieved through two basic steps: 1) quantization and 2) bitslicing. We note that quantization is a well-known model compression technique in DL research. However, we found that it is also an essential step for implementing FHE-encrypted DL. The impact of quantization is explained in the next section. We also present some basic notations in Table IV, which we shall use throughout the following sections.

A. Quantized Inference of CNNs

In simple words, model quantization aims to reduce the precision of the trained weights (and inputs) in a model, so that they can fit within a reduced bit width. Mathematically, quantization maps all elements in the range $[R_1, R_2]$ to the range $[-(2^{(B-1)}), (2^{(B-1)} - 1)]$, where B is the bit-precision after quantization. For example, if 32-bit floating point weights can be compressed to 8-bit weights it leads to a significant reduction in the memory footprint of a model.

In the context of FHE, however, model compression has several other benefits. The memory blowup for FHE is primarily

TABLE V
ARCHITECTURES (MENDELEY AND MNIST)

Layer	Output-shape (Mendeley)	Trainable Parameters (Mendeley)	Output-shape (MNIST)	Trainable Parameters (MNIST)
input_layer	(224,224,3)	3	(28,28)	3
reshape	(224,224,3)	1	(28,28,1)	1
Conv2d	(222,222,3)	27	(26,26,1)	15
Max_pooling2d	(111,111,3)	1	(13,13,1)	1
Flatten	(36963)	1	(169)	1
Dense	(2)	73926	(10)	1705

attributed to the ciphertext size. For Boolean FHE operations, each plaintext bit is mapped to a multibit ciphertext. High precision (i.e., larger bit width) operands can therefore result in excessive memory overhead. Moreover, larger bit-width also results in higher number of ciphertexts which eventually increases the timing overhead. For example, the bootstrapping time increases significantly while processing on 32-bit floating point (*FP*32) representations, which is a common weight representation in many DL libraries. Significant improvement due to quantization is therefore evident.

However, an obvious impact of quantization is the degradation in inference accuracy. Fortunately, such accuracy loss can be bounded to a reasonable limit, thanks to the efficient quantization routines implemented in modern DL pipelines such as TFLite [33]. In general, at most 2% of accuracy loss is observed due to a quantization from *FP*32 to 8-bit integers (*INT*8). Even without FHE, this provides a speedup of 2 times over the *FP*32 counterparts [34].

Throughout this article, we utilize two representative datasets (MNIST [35] and Mendeley [36]) for validating our ideas. The MNIST dataset comprises 60 000 (28×28) images, all representing digits between ($0 \ldots 9$). The Mendeley dataset is dedicated to the task of developing a binary classifier for concrete building images labeled as cracks and no-cracks. Each label class has $20K$ images each having dimensions of $224 \times 224 \times 3$.

We first observe the performance of the models corresponding to these datasets after quantization. It is evident that not every model is supposed to give the same accuracy after quantization. We figured out CNN architectures for both datasets which retain the accuracy even after quantization. In fact, for the Mendeley dataset, we observed slightly higher accuracy for the *INT*8 quantized version than the baseline *FP*32. While such increase in accuracy is rare, we note that it is not impossible. For example, such increase in accuracy has also been observed in [33], which also depends on the range of the quantization. Our reference architecture (see Fig. 2 and Table V) remains the same for both datasets except changes in few hyperparameters. More precisely, for MNIST we utilize ($W = 28$, $H = 28$ $M = 3$, $c = 1$), where $W \times H$ is the dimension of the input image, and $M \times M$ is the dimension of the CNN filters. c denotes the number of channels in the image. The model also uses AvgPool as well as flattening layers. For the Mendeley dataset, the parameters are ($W = 224$, $H = 224$ $M = 3$, $c = 3$).

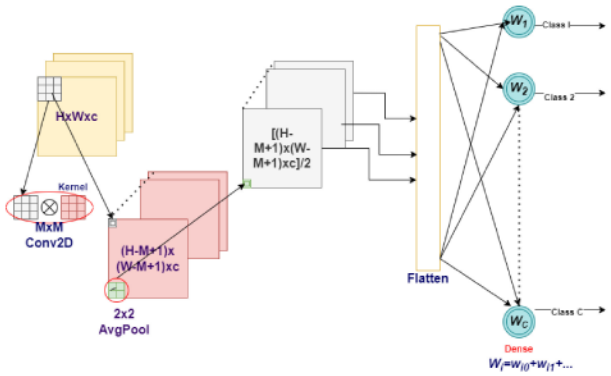

Fig. 2. *CNN architecture block diagram*—this model architecture has been adopted for MNIST ($W = 28, H = 28, M = 3, c = 1$) as well as Mendeley bridge ($W = 224, H = 224, M = 3, c = 3$) datasets which are then mapped to bitsliced simulations of FHE circuits.

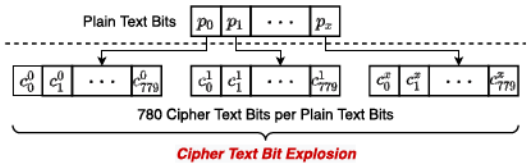

Fig. 3. Blowup in ciphertext size.

TABLE VI
ACCURACY WITH/WITHOUT QUANTIZATION

Dataset	Test/Train Split	Accuracy ($FP32$)	Accuracy ($INT8$)
MNIST	48000/12000	96.43 %	96.33 %
Mendeley	30488/7622	96.82 %	97.24 %

The inference accuracy of the models after quantization is presented in Table VI. It can be observed that the degradation in accuracy for MNIST is quite reasonable. On the contrary, for Mendeley, we observe a slight increase in accuracy. While we believe that such observation is specific to this model instance, this has been observed previously for multiple use cases [37], [38]. Overall, we achieve reasonable accuracy, while compressing the model significantly making it amenable to FHE conversion.

B. Bitslicing

Quantization is indeed beneficial in terms of timing and memory. However, it is not sufficient to accelerate the FHE versions of a DL model. A straightforward FHE implementation of this model (for MNIST) with NuFHE library takes almost 80 h and 6.7 GB (on average) of memory to make a single inference. NuFHE generates a 780-bit ciphertext corresponding to each bit of computation and leverages the FHE gates to process them. The blowup in ciphertext size is illustrated in Fig. 3. The memory and computational overhead is evident from this blowup. A crucial question is can these performance figures be improved?

While one might have several attractive results for LHE and SWHE libraries (or for FHE libraries like Concrete with some output errors), none of these is an ultimate solution as pointed out in the introduction. Therefore, we focus on

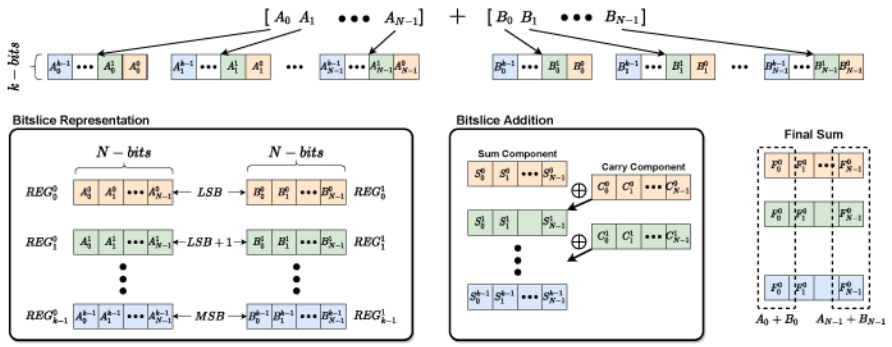

Fig. 4. BFA.

improving the timing and memory overhead for NuFHE library which contains bootstrapped Boolean gates. In this work, Conv2D, MaxPool, AvgPool, and Dense layers are implemented using gates from NuFHE. As we explain throughout this section, there are several points of performance optimizations while implementing these operations which result in significant performance boost. The core component behind these performance boost is a bitsliced adder module, which we describe in the next section. At this point it is important to note that none of our implementations including the bitsliced adder need any augmentation to be made in the NuFHE library. Rather, we build a software wrapper over the library to implement the layer operations, which are later utilized to construct the DL models.

1) Bitsliced Full Adder: The main idea behind our bitsliced full Adder (BFA) module is presented in Fig. 4. It performs addition between two arrays of integers in a bitsliced manner. Referring to Fig. 4, the two lists to be added are denoted as $A_{lst} = [A_0, A_1, \ldots, A_{N-1}]$ and $B_{\text{lst}} = [B_0, B_1, \ldots, B_{N-1}]$. The output is $\text{Sum} = [(A_0 + B_0), (A_1 + B_1), \ldots, (A_n + B_n)]$. Without loss of generality let us assume that each A_i (resp. B_i) is represented as $A_i = \langle A_i^{k-1}, \ldots, A_i^1, A_i^0 \rangle$ with k bits. A straightforward addition of A_{lst} and B_{lst}, therefore, requires N invocations of a k-bit full adder. Each of these k-bit adder invocation requires $4k$-XORs, $2k$ AND operations (assuming a ripple carry adder). Since each gate operation in NuFHE is bootstrapped, total $N \times 6k$ bootstrapping operations are performed. Even assuming some parallelization (especially for the k XORs computing the sum of two k-bit numbers), the number of bootstapping remain high, as the carry chain cannot be parallelized easily. One of the main optimization challenges in this regard is to reduce the number of bootstrapping. In NuFHE context, it implies a reduction in the number of gates which is difficult for a small circuit like an adder. Instead, we exploit the fact that NuFHE gates can process multiple ciphertexts (corresponding to different bits) when they are packed together. We exploit this feature in an SIMD fashion.

In order to improve the efficiency, we introduce the concept of *bitsliced packing* PACK(j) of ciphertexts. More precisely, we utilize k arrays ($\text{REG}_0, \text{REG}_1, \ldots, \text{REG}_{k-1}$), each of size N. The REG_j stores the jth least significant (LSB) bit of the integers $\{A_i\}_{i=0}^{N-1}$. The PACK(j) function is illustrated in Algorithm 1 and Fig. 4.

Algorithm 1 PACK(j)

Input: $A_1 \cdots A_N$, REG_j
Output: REG_i
1: **for** $t \leftarrow 0$ to $N-1$
$REG_j[t] \leftarrow A_t^j$
2: **end for**
3: **return** REG_j

We further demonstrate the workflow of packing algorithm with a short example, where we pack four integer operands into four copies of four bit arrays. Suppose, the integers are $A_{lst} = [1, 3, 5, 6]$ whose binary representations are given by 0001, 0011, 0111, and 0110. The packing registers are $\text{REG}_0, \text{REG}_1, \text{REG}_2$, and REG_3. In this example, $N = 4$, so as per our notation, the operands are $A_0, A_1, \ldots, A_3$ where $A_0 = [A_0^3, A_0^2, A_0^1, A_0^0] = [0001]$. Similarly, $A_1 = [A_1^3, A_1^2, A_1^1, A_1^0] = [0011]$, $A_2 = [A_2^3, A_2^2, A_2^1, A_2^0] = [0111]$, $A_3 = [A_3^3, A_3^2, A_3^1, A_3^0] = [0110]$. After the execution of PACK(j) for $0 \leq j < 4$, REG_0, REG_1, REG_2, and REG_3 contains [1110], [0111], [0011], and [0000], respectively.

Next, we show how to utilize this bitsliced packing to implement a full-adder module for adding two arrays.

The Adder Circuit: Let us consider two lists A_{lst} and B_{lst} packed in two sets of arrays REG_j^0 and REG_j^1, where $0 \leq j < k$ and $|\text{REG}_j^h| = N$ ($h = \{0, 1\}$). Referring to Fig. 4, an N-bit bitwise XOR can be deployed for each (REG_j^0, REG_j^1) pair to calculate the sum component of addition. Similarly, the carry logic can be implemented between these two registers. It is worth noting that for REG_j^h's after the LSB position, there is a carry propagation to enable the ripple carry (see Fig. 4) which further needs N-bit XORs. After the carry propagation completes, the final sum is also stored within one of the REG_j^hs.

Let us now analyze the timing and memory overhead of this bitsliced adder compared to the naïve approach of invoking N, k-bit adders. The number of gate operations essentially remains the same. However, due to the bitsliced packing, each bootstrapped gate works on N bits in BFA. In contrast, the carry chain of naïve k-bit adders work on single bit at a time. Consequently, number of required bootstrapping increase N-fold for the naïve case, compared to the BFA case. This is the sole reason behind the timing improvement that we observe

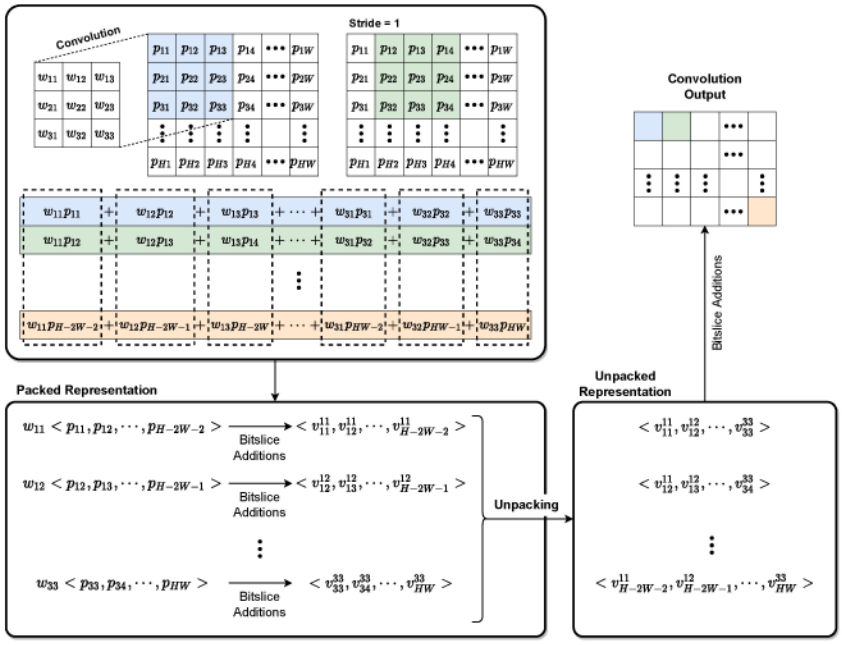

Fig. 5. Data packing with Bitslicing for FHE-Conv2D workflow.

in our implementations. Reduction of bootstrapping also significantly improves the memory overhead (reduces it from 5.2 GB to 280 MB). As already pointed out in the preliminaries, bootstrapping involves several complex operations, such as key switching and blind rotate which are highly memory intensive. While multithreading may improve the timing overhead of bootstrapping to some extent, it is not so effective in the context of memory as each thread consumes around 200 MB, which can quickly blow up with N. Overall, the BFA adder is more efficient for adding two arrays compared to naïve addition. It only requires a software wrapper over the NuFHE library without incurring any significant coding complexity.

2) Bitslicing in DL: We next implement different layer operations of DL using the BFA as a core component. Referring to the generic architecture in Fig. 2, here, we implement three major operations, namely, Conv2D, MaxPool2d, and Dense layer. These operations are also representative to most of the existing DL layer operations.

Encrypted Conv2D/Dense Operators: The core component of Conv2D or Dense operations can be represented as vector-matrix products. Most of the cases it takes the form $y = \sum_{i=1}^{n} w_i x_i$, where w_is are the weights, and x_is is the inputs or intermediate values. We note that for encrypted inference, the weights are in plaintext and the x_is are encrypted. Therefore, in FHE, such multiplications can be implemented as scalar products followed by an addition. The scalar products, on the other hand, can be implemented using repeated additions. A natural way of implementing these repeated additions is to use a loop with the w_i in loop counter.

We begin our discussion in the regard with the Conv2D operation. Let us assume that a convolution filter of dimension $M \times M$ works on an image with dimension $W \times H$. For the sake of illustration, here, we take $M = 3$. In Conv2D, the filter moves over an image in steps called strides (as shown in Fig. 5), calculating the convolution. Referring to Fig. 5, each convolution operation can be represented as $\sum_{i=t_1}^{(t_1+2)} \sum_{j=t_2}^{(t_2+2)} w_{(i \bmod 3)(j \bmod 3)} p_{ij}$, where w_{ij}s represent the weights, p_{ij}s represent the inputs, and (t_1, t_2) represents the row, column index of a stride. Now, considering two consecutive row-wise strides, one may observe that the convolution operations contains several product terms having the same multiplicand w_{ij}. For example, in Fig. 5, one may observe that $P_{w_{11}} = (p_{11}, p_{12}, \ldots, p_{(W-2)H-2})$ gets multiplied with the same weight w_{11}. Since the scalar product is nothing but consecutive addition, therefore, the scalar product $\langle w_{11}, P_{w_{11}} \rangle$ can be represented as repeated addition of the integer array/vector $P_{w_{11}}$. This repeated addition is implemented by invoking the BFA unit $(w_{11} - 1)$ times.

Once the product terms of convolution are computed, the next step is to perform the addition of these product terms. In order to perform this step, one needs to unpack the bitsliced products. This is a straightforward step and an inverse of the PACK(j) operation. Interestingly, the unpacking also results in several arrays of integers to be added together. The BFA unit is invoked once again to perform this operation, which eventually returns the convolution output.

Let us now explain the computational complexities due to this bitsliced Conv2D operation. One should note that Conv2D is one of the most costly operations for FHE as it involves several scalar multiplications and additions. The packing width ($|P_{w_{ij}}|$) here determines how much parallelization can be obtained. Considering row-wise movement of the convolution filter, there can be total $(W - M + 1)$ strides which can be parallelized. Therefore, we have $|P_{w_{ij}}| = (W - M + 1)$. On the other hand, for a convolution filter of dimension $M \times M$, there can be total M^2 weight values. As a result, during the addition phase of the product terms, the width of each array/vector will be of M^2, and there will be $(W - M + 1)$ such arrays/vectors. For MNIST images, we have $(W - M + 1) = (28 - 3 + 1) = 26$

and $M^2 = 9$. For Mendeley images, however, we face an implementation obstacle. The NuFHE allows packing of at most 32 ciphertexts, which limits the maximum size of $P_{w_{ij}}$ to 32. For 224×224 images in Mendeley, we therefore need $224/32 = 7$ independent packings to be implemented for a complete row-wise convolution. The overheads for the convolution can be calculated in a similar manner.

The implementation of the encrypted dense layer is similar to the Conv2D layer, in principle. In this case, we figure out (encrypted) inputs having a common weight multiplier. This is a straightforward preprocessing as the weights are in plaintext. Once a set of ciphertexts with a common multiplier is identified, we apply the same bitsliced processing approach as explained for Conv2D to process them in parallel. One critical point here is that unlike the Conv2D case, the width of $P_{w_{ij}}$s for Dense layer will not be equal. Also, the unpack step will require some extra effort to properly arrange the product terms for addition. However, handling the first issue is easy, as our approach does not depend upon the length of $P_{w_{ij}}$s. To rearrange the product terms we maintain a dictionary storing the actual location of product terms in the computation. The rearranging after unpack can be performed in constant time only by referring to this dictionary. Therefore, we can obtain similar parallelization as for Conv2D even in the case of Dense.

AvgPool2D and MaxPool2D: The objective of AvgPool2D is to calculate the average of Conv2D outputs. Averaging requires addition followed by division operations. It is worth mentioning that the addition here is not bitsliced. This is because the addition step in AvgPool2D only adds a few integers for which the overall overhead is not very significant. However, the divisions are costly in FHE. The straightforward way of performing division as repeated subtraction has to be ruled out here because results can be in floating points which is not inherently supported by FHE schemes. Moreover, the loop terminating criterion for this repeated subtraction is in the encrypted domain, which makes it even more complex to process in FHE.

The first step toward avoiding the costly divisions is to limit ourselves to divisions by 2^l, which (for integers) can be performed by bit shifts on the ciphertext vectors. As (2×2) AvgPools are very popular, computing AvgPool2D is just division by 4 (i.e., 2 right shift (RS) operations. Thus, we get our simplified FHE circuit mappings for AvgPool2D calls. As demonstrated, in Fig. 6, (C_1, C_2, C_3, C_4) are four ciphertexts which will be fed into a 2×2 AvgPool2D module. Our circuit first adds these four ciphertexts and then obtains the *AvgPool2D* result by performing two logical right-bit shift (2*RS*) operations. Likewise, MaxPool2D FHE-circuit mappings could be realized by an encrypted multiplexer (FHE-MUX). The circuit selects the number based on the MSB difference of two input numbers. Our circuit, in this regard, is a cascade of FHE MUXs. For example, if we apply MaxPool2D from 3×3 kernel, then, our circuit is a cascade of $(3 \times 3) - 1 = 8$ MUXs. Each MUX selects the maximum of two numbers based on sign of difference of the numbers. An illustration for the MaxPool is shown in Fig. 7. At this point, we note that the AvgPool2D is significantly lightweight

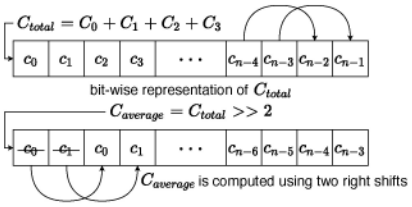

Fig. 6. FHE-AvgPool2d (logical bitwise circuitry).

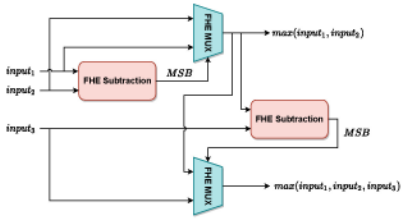

Fig. 7. FHE-MaxPool2D logic.

compared to the MaxPool. Therefore, in both of our models, we utilize the AvgPool2D circuit.

Encrypted Activation Functions: An Activation Function decides whether a neuron should be activated or not, to be specific, whether the neuron's input to the network is important or not in the process of prediction using simpler mathematical operations. Since linear activations are not much effective in DNN or CNN scenario, we have mostly focused on efficient FHE counterpart design for nonlinear activations.

Among nonlinear activations, ReLU is one of the most used functions in DL applications. As discussed in Section II, existing encrypted ML architectures are either based on LHE or some approximations of FHE schemes with some inherent limitations that does not allow to incorporate ReLU in encrypted domain. That is the reason why the existing schemes mostly use either square activations or sign activations. To elaborate the reason of this limitation further, we revisit the basic operations required to implement ReLU activation.

ReLU ($f(x)$) as an activation that selects the positive part of its argument as $f(x) = \max(0, x)$, where x is the input to a neuron. Realization of maximum function in homomorphic domain requires encrypted decision making, that is not feasible with LHE computations [39], [40]. In our FHE-based implementation, we realize the encrypted decision making with encrypted multiplexer considering the FHE subtraction result as selection input to choose between encrypted 0 and encrypted x.

However, other nonlinear activations like *Softmax* or *tanh* include the standard exponential function computation on encrypted input vectors. As explained in [41], exponent computation becomes infeasible with the encrypted power term considering our underlying unencrypted processors. However, this limitation can be outdone in ML processing with prior domain knowledge, where the maximum value of such input vectors is known [41].

V. Experiments and Results

In this section, we present the timing and memory overhead of our proposed encrypted inference architecture. We

TABLE VII
SPEEDUP FOR BITSLICING DNN ARCHITECTURE W.R.T. NONBITSLICING

Experiment Setup	NN-Inference Time (Hours)
PyFHE	120
NuFHE(without Bitslicing)	80
NuFHE(Bitsliced)	2.15

TABLE VIII
BITSLICED CNN TIMING PROFILES (MNIST)

(28×28) MNIST Data	Time in Minutes (=Hours)
Conv2D	$139.78(= 2.32)$
AvgPool2D	$11.53(= 0.2)$
Conv2D+AvgPool2D	$151.31(= 2.52)$
Conv2D+AvgPool2D+Dense	$268.21(= 4.47)$

TABLE IX
BITSLICED CNN TIMING PROFILES (MENDELEY)

$(224 \times 224 \times 3)$ Mendeley Data	Time (Estimated Hours)
Conv2D	108
AvgPool2D	30
Conv2D+AvgPool2D	138
Conv2D+AvgPool2D+Dense	140

evaluate all our experimental studies on an Ubuntu 18.04 LTS $64-bit$ platform with 256-GB RAM and 24-GB GPU memory support. All underlying FHE blocks are supported by existing FHE encryption library NuFHE [16] and PyFHE [42], which is another version of TFHE for performance comparison. NuFHE [16], which is a GPU powered TFHE encryption library implemented in Python, where PyFHE does not have any GPU support.

We start with a simple DNN architecture which includes three layers: two layers of 50 neurons and a final fully connected layer of 10 neurons. Table VII compares the bitsliced implementation of a simple DNN architecture with nonbitsliced versions of the same architecture. In bitsliced implementation, we first break down our target architecture to proposed FHE counterparts of Conv2D, AvgPool2D, and Dense. These basic blocks are accelerated by underlying bitsliced adder functions developed on top of NuFHE homomorphic blocks. As shown in Table VII, compared to nonbitsliced inference with timing overhead of 80 h, our bitslicing-based software wrappers can accelerate the same task to 2.15 h in case of DNN implementation.

Tables VIII and IX show the timing profiles of the bitsliced implementation of the CNN architecture for both MNIST Digit Classification [35] and Mendeley use-case [36]. The reference architectures for both the use-cases have been elaborated in Table V and in Fig. 2. We provide an estimated timing requirement for execution of this same CNN architecture on the Mendeley Bridge dataset [36] as shown in Table IX. The dimensions of this dataset is $224 \times 224 \times 3$. We note that this dataset is more complex than other representative datasets, such as CIFAR10 [43] (dimension $32 \times 32 \times 3$), in terms of input dimension which is a critical factor in FHE as it drastically increases the computation. As per Table VIII, the bitsliced Conv2D takes about 2.3 h while complete evaluation of an MNIST image takes about 4.5 h and it may take around 140 h for Mendeley (Table IX).

TABLE X
SHE/BITSLICING MEMORY PROFILE BASELINE COMPARISONS

Number of Conv2D Operations	Memory Overhead (SHE with TFHE)	Memory Overhead(NuFHE-Bitslicing)
30	6GB	$<$ 200MB
50	9GB	$<$200MB
100	18GB	$<$400MB
676 (MNIST)	$100GB^*$	570MB

Memory Profiling of Bitsliced Inference: Recent concerns about security and privacy guarantees in IoT systems have drawn interests about secure computation in the IoT domain. However, huge memory consumption and computation overhead are two prime bottlenecks to implement HE schemes on resource-constrained embedded platforms and this is an important recent research direction [44]. With this motivation, we have performed layer-by-layer memory profiling of both our DNN and CNN architectures as shown in Fig. 8(a) and (b). As depicted in Fig. 8(a), bitslicing supported CNN framework is quite memory efficient (about 700 MB). The DNN framework, which takes about 1.9 GB also fits into the modest memory bounds of edge devices, i.e., 2–4 GB as highlighted in Fig. 8(b).

Furthermore, we have compared our memory profiles to the state-of-the-art encrypted inference approaches. We benchmark our bitsliced design pipeline with an open source implementation[1] due to [7]. This approach provides an implementation of scalar products with small examples, e.g., eight ciphertexts. We use this codebase to estimate their memory growth and we highlight the same in Table X. To do this, we approximate their implementation, i.e., eight scalar products as equivalent to a single Conv2D operation (the reason being with a 3×3 kernel, Conv2D is just nine scalar products which the concerned framework replaces with logical bit-shifts). We check experimentally that a single Conv2D operator amounts to around 180-MB memory overhead. Then, we run multithreaded implementation of this Conv2D block and observe the memory explodes proportionally as highlighted for several thread values $(16, 20, 24)$ in Fig. 9. Therefore, as Table X depicts, an evaluation of 50 convolutions amounts to about 9 GBs. We also evaluate memory footprints of Cryptonets [4], another pioneering approach by Microsoft for secure CNN inference. This approach exhausts 2-3 GB memory for running a simple MNIST pipeline with a Conv2D stacked with couple of Dense layers. Research report in [5] also points out that most encrypted CNN approaches amount to at least few GB of RAM usage. Table XI shows the merits and demerits of our proposed architecture with some state-of-the-art encrypted inference models [4], [5], [7], [14]. Comparisons are done on the basis of power of homomorphic computations (leveled or fully homomorphic), activations supported, associated memory overheads, and possibility of decryption errors. It is to note that existing inference models may outperform in terms of timing requirement, but our proposed decryption error-free model has its own significance in terms of better accuracy without any approximation and hugely reduced memory overhead. After memory profiling all

[1]https://github.com/qianlou/SHE

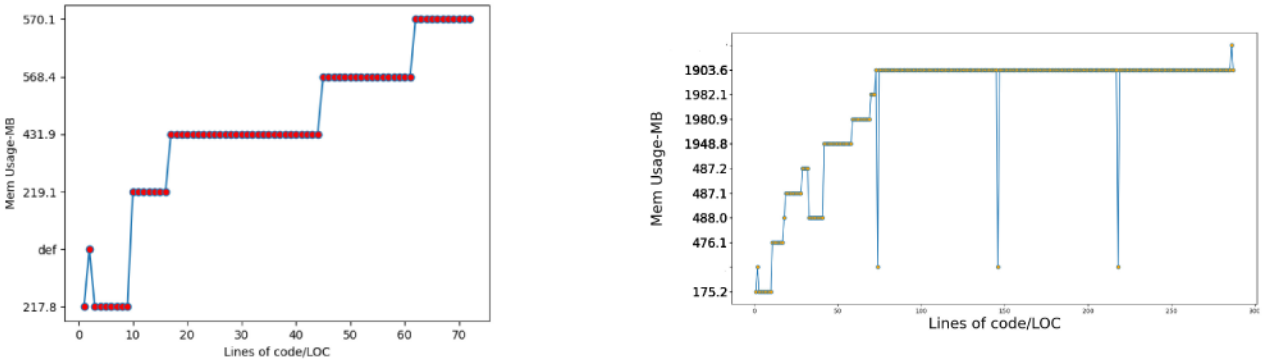

Fig. 8. Memory profiling statistics of bitsliced inference (our method). (a) Conv2D design-memory profiles (with bitslicing). (b) Memory profiles of only the *Dense Layer* of Conv2D+AvgPool+Dense inference pipeline (with bitslicing).

TABLE XI
COMPARISON WITH EXISTING APPROACHES

Framework	Library	Activations (Exact/Approximate)	Activations Supported	Memory Overhead	Erroneous Decryption
Cryptonets	LHE	Exact	Square	$>$ 100GB (CIFAR-10)	No
Lola-Cryptonets	LHE	Exact	Square	$2-3$GB (MNIST)	No
FHE-DiNN	FHE	Approximated	Signed	–	Yes
SHE (estimated for deeper nets)	LHE	Approximated	ReLU	100 GB (MNIST estimated)	No
Our Approach	FHE	Exact	All	$< 700MB$ (MNIST)	No

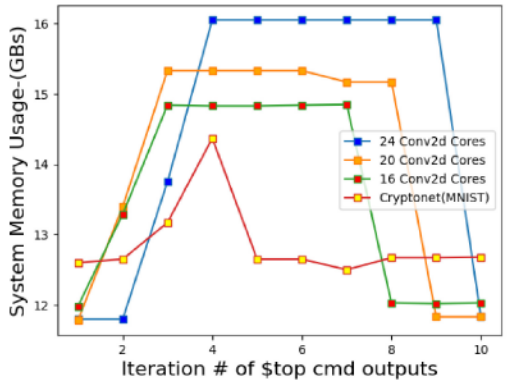

Fig. 9. Memory profiles of SHE and Cryptonet baselines.

these baselines, we would like to conclude that the proposed scheme hugely optimizes the memory footprint and this aspect perhaps cements our model's potential for low resource edge platforms.

VI. CONCLUSION AND FUTURE RESEARCH

In this work, we realize encrypted CNN suitable for real-world DL-based applications. In this article, we provide several key insights. Performance and blowup in ciphertext size are two major bottlenecks, which we address with architectural tricks, such as quantization and bitslicing. A potential future work in this regard is to further optimize the inference timing for a larger spectrum of real-world applications. Another direction of future research is to adopt the proposed optimizations beyond Boolean FHE.

REFERENCES

[1] P. Nicholson. "AWS hit by largest reported DDoS attack of 2.3 TBPS." 2020. [Online]. Available: https://www.a10networks.com/blog/aws-hit-by-largest-reported-ddos-attack-of-2-3-tbps/

[2] D. Ramel. "EPIC DDoS fail: Azure cloud fends off 'largest attack ever reported in history.'" 2022. [Online]. Available: https://virtualizationreview.com/articles/2022/01/31/azure-ddos.aspx

[3] E. Root. "Cryptominers Threaten GCP Virtual Servers." 2022. [Online]. Available: https://www.kaspersky.com/blog/attacks-on-google-cloud-platform/43312/

[4] N. Dowlin, R. Gilad-Bachrach, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing, "CryptoNets: Applying neural networks to encrypted data with high throughput and accuracy," in *Proc. 33rd Int. Conf. Int. Conf. Mach. Learn. (ICML)*, vol. 48, 2016, pp. 201–210.

[5] A. Brutzkus, O. Elisha, and R. Gilad-Bachrach, "Low latency privacy preserving inference," in *Proc. Int. Conf. Mach. Learn.*, 2019, pp. 1–9.

[6] N. Jain, K. Nandakumar, N. K. Ratha, S. Pankanti, and U. Kumar, "Efficient CNN building blocks for encrypted data," 2021, *arxiv.abs/2102.00319*.

[7] Q. Lou and L. Jiang, "She: A fast and accurate deep neural network for encrypted data," in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., Curran Assoc., Inc.

[8] C. Gentry, "Computing on encrypted data," in *Proc. Int. Conf. Cryptol. Netw. Security*, 2009, p. 477.

[9] L. Ducas and D. Micciancio, "FHEW: Bootstrapping homomorphic encryption in less than a second," in *Proc. Adv. Cryptol. (EUROCRYPT)*, 2015, pp. 617–640.

[10] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "TFHE: Fast fully homomorphic encryption over the torus," *J. Cryptol.*, vol. 33, no. 1, pp. 34–91, Jan. 2020.

[11] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," IACR Cryptol. ePrint Arch., Lyon, France, Rep. 2016/421, 2016. [Online]. Available: https://ia.cr/2016/421

[12] B. Li and D. Micciancio, "On the security of homomorphic encryption on approximate numbers," in *Proc. Adv. Cryptol. (EUROCRYPT) 40th Annu. Int. Conf. Theory Appl. Cryptograph. Techn.*, vol. 12696, 2021, pp. 648–677.

[13] U. Crypto. "CKKS key recovery." Mar. 2021. [Online]. Available: https://github.com/ucsd-crypto/CKKSKeyRecovery

[14] F. Bourse, M. Minelli, M. Minihold, and P. Paillier, "Fast homomorphic evaluation of deep discretized neural networks," in *Proc. Annu. Int. Cryptol. Conf.*, 2018, pp. 483–512.
[15] ZAMA. "ZAMA.AI/Concrete." 2021. [Online]. Available: https://github.com/zama-ai/concrete
[16] NuFHE. "NuFHE, A GPU-Powered Torus FHE Implementation." [Online]. Available: https://nufhe.readthedocs.io/en/latest/
[17] N. GAMA. "TFHE." [Online]. Available: https://github.com/tfhe/tfhe
[18] E. Biham, "A fast new DES implementation in software," in *Proc. 4th Int. Workshop Fast Softw. Encrypt. (FSE)*, vol. 1267, 1997, pp. 260–272.
[19] J. Lee *et al.*, "Privacy-preserving machine learning with fully homomorphic encryption for deep neural network," *IEEE Access*, vol. 10, pp. 30039–30054, 2022.
[20] R. L. Rivest *et al.*, "On data banks and privacy homomorphisms," *Found. Secure Comput.*, vol. 4, no. 11, pp. 169–180, 1978.
[21] T. E. Gamal, "A public key Cryptosystem and a signature scheme based on discrete logarithms," in *Proc. Adv. Cryptol. CRYPTO*, vol. 196, 1984, pp. 10–18.
[22] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) fully homomorphic encryption without bootstrapping," in *Proc. Innov. Theor. Comput. Sci.*, 2012, pp. 309–325.
[23] A. A. Badawi *et al.*, "Towards the AlexNet moment for homomorphic encryption: HCNN, the first homomorphic CNN on encrypted data with GPUs," *IEEE Trans. Emerg. Topics Comput.*, vol. 9, no. 3, pp. 1330–1343, Jul.–Sep. 2021.
[24] E. Hesamifard, H. Takabi, and M. Ghasemi, "CryptoDL: Deep neural networks over encrypted data," 2017, *arxiv.abs/1711.05189*.
[25] B. Moons, D. Bankman, and M. Verhelst, *Embedded Deep Learning: Algorithms, Architectures and Circuits for Always-On Neural Network Processing*. Cham, Switzterland: Springer, Jan. 2019.
[26] I. Chillotti, M. Joye, and P. Paillier, "Programmable bootstrapping enables efficient homomorphic inference of deep neural networks," Cryptol. ePrint Archive, Rep. 2021/091, 2021. [Online]. Available: https://ia.cr/2021/091
[27] C. Gentry. "A fully homomorphic encryption scheme." Accessed: Jun. 6, 2022. [Online]. Available: https://crypto.stanford.edu/craig/craig-thesis.pdf
[28] O. Regev, "On lattices, learning with errors, random linear codes, and cryptography," *J. ACM*, vol. 56, no. 6, pp. 1–40, 2009.
[29] C. Gentry, A. Sahai, and B. Waters, "Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based," in *Proc. IACR Cryptol. ePrint Arch.*, 2013, p. 340.
[30] ZAMA. "ZAMA.Ai/concrete." GitHub. Accessed: Jun. 10, 2022. [Online]. Available: https://github.com/zama-ai/concrete
[31] I. J. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016. [Online]. Available: http://www.deeplearningbook.org
[32] *Data Encryption Standard (DES)*, NIST, Gaithersburg, MD, USA, 1995.
[33] "Tensorflow model optimization." Tensorflow. 2022. [Online]. Available: https://www.tensorflow.org/model$_o$ptimization/guide/quantization/training
[34] NVIDIA. "NVIDIA/FasterTransformer." 2021. [Online]. Available: https://github.com/NVIDIA/FasterTransformer
[35] Y. LeCun, C. Cortes, and C. Burges, *MNIST Handwritten Digit Database*, ATT Labs, Washington, DC, USA, 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist
[36] C. F. Ozgenel. "Concrete Crack Images for Classification." 2019. [Online]. Available: https://data.mendeley.com/datasets/5y9wdsg2zt/2
[37] "Quantized TFLite model gives better accuracy then TF model." 2020. [Online]. Available: https://stackoverflow.com/questions/64030764/quantized-tflite-model-gives-better-accuracy-than-tf-model
[38] V. Praveen. "Improving INT8 accuracy using quantization aware training and the NVIDIA TAO toolkit." 2020. [Online]. Available: https://developer.nvidia.com/blog/improving-int8-accuracy-using-quantization-aware-training-and-tao-toolkit/#::text\protect$\relax=$Conclusion,compar%20%20FP%20%20FP32
[39] A. Chatterjee and I. Sengupta, "Translating algorithms to handle fully homomorphic encrypted data on the cloud," *IEEE Trans. Cloud Comput.*, vol. 6, no. 1, pp. 287–300, 2018.
[40] A. Chatterjee, M. Kaushal, and I. Sengupta, "Accelerating sorting of fully homomorphic encrypted data," in *Proc. Cryptol. INDOCRYPT 14th Int. Conf. Cryptol. India*, vol. 8250, 2013, pp. 262–273.
[41] A. Ghosh, A. Raj, and A. Chatterjee, "Encrypted operator design with domain aware practical performance improvement," in *Proc. 7th Int. Conf. Math. Comput. (ICMC)*, vol. 1412, 2021, pp. 93–107.
[42] "pyFHE Github Repository." Jan. 2020. [Online]. Available: https://github.com/virtualsecureplatform/pyFHE
[43] A. Krizhevsky. "The Cifar10 Dataset." 2009. [Online]. Available: http://www.cs.toronto.edu/~kriz/cifar.html
[44] D. Natarajan and W. Dai, "SEAL-embedded: A homomorphic encryption library for the Internet of Things," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2021, no. 3, pp. 756–779, 2021.

Soumik Sinha is currently pursuing the Ph.D. degree with the Indian Institute of Technology Kharagpur, Kharagpur, India.

His research interest lies at the confluence of system security and AI.

Sayandeep Saha received the Ph.D. degree from the Indian Institute of Technology Kharagpur, Kharagpur, India.

He is currently a Postdoctoral Researcher with Nanyang Technological University, Singapore. His research interests include formal methods and hardware security.

Manaar Alam received the Ph.D. degree from the Indian Institute of Technology Kharagpur, Kharagpur, India.

He is currently a Postdoctoral Researcher with New York University Abu Dhabi, Abu Dhabi, UAE. His research interests lie at the confluence of deep learning and security.

Varun Agarwal received the bachelor's degree in technology from Delhi Technological University, New Delhi, India, in 2022.

His research interest includes image cryptography, hardware security, and homomorphic encryption.

Ayantika Chatterjee received the Ph.D. degree from the Indian Institute of Technology (IIT) Kharagpur, Kharagpur, India.

She is currently an Assistant Professor with the IIT Kharagpur. Her research interest includes security, embedded systems, and encrypted computation.

Anoop Mishra is currently pursuing the Ph.D. degree with the University of Nebraska Omaha, Omaha, NE, USA, working in different areas of artificial intelligence, Internet of Nano Things, and edge computing.

Deepak Khazanchi received the Ph.D. degree in business administration from Texas Tech University, Lubbock, TX, USA.

He is a Professor of Information Systems and Quantitative Analysis with the University of Nebraska Omaha, Omaha, NE, USA.

Debdeep Mukhopadhyay (Senior Member, IEEE) received the B.Tech. and Ph.D. degrees from the Indian Institute of Technology (IIT) Kharagpur, Kharagpur, India.

He is a Professor with the Department of Computer Science and Engineering, IIT Kharagpur. His research interests include cryptography, VLSI of cryptographic algorithms, hardware security, and side channel analysis.